Begleitmaterial zur Vorlesung
Prozessrechnertechnik
(Eingebettete Realzeit-Systeme)
Sommersemester 1999

Peter Marwedel
Informatik XII
(Technische Informatik)
Universität Dortmund

5. April 1999

Dieser Begleittext ist nur zur Benutzung durch die Teilnehmer der Vorlesung gedacht.
Es wird keinerlei Gewähr dafür übernommen, daß der Text frei von Urheberrechten ist.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Gegenstand der Vorlesung

1.1.1 Definition eingebetteter Systeme

Informatik kann als Wissenschaft von der Verarbeitung von Informationen verstanden werden [Man]. Für viele Informatiker erfolgt dabei implizit eine Verengung auf die Verarbeitung von Informationen in PCs, Workstations und Großrechnern. Kennzeichnend für diese Systeme sind u.a. Tastaturen, Monitore und relativ geringe Anforderungen an das Einhalten strenger zeitlicher Restriktionen. Weiter werden in der Regel keine physikalischen Größen (wie z.B. Beschleunigungswerte) direkt aufgenommen oder verarbeitet. Dies ist bei den sog. 1. Vorles.

> **eingebetteten** informationsverarbeitenden **Systemen**[1]

(im Folgenden kurz EIS genannt) grundsätzlich anders.

Eine mögliche Definition von EIS ist die folgende:

Def.: Eingebettete Systeme sind informationsverarbeitende Systeme, die physikalische Größen aufnehmen, verarbeiten und beeinflussen.

In den üblichen Vorlesungen über Programmierung beschäftigt man sich in der Regel mit Programmen, die zu gewissen Eingaben nach etwas Rechenzeit Ausgaben liefern sollen. Letztlich berechnen diese Programme eine *Funktion*. Man denke etwa daran, dass die Diskussion der Berechenbarkeit v.a. von der Aufgabe ausgeht, durch ein Programm eine Funktion berechnen zu lassen. Die Zeit spielt nur insofern eine Rolle, als man die Laufzeit nicht allzu stark wachsen lassen möchte.

Im Unterschied dazu handelt es sich bei eingebetteten Systemen um sog. **reaktive Systeme**.

Def. [BLR95]: *A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment.*

Die Reaktion eines reaktiven Systems hängt dabei von der Eingabe und dem aktuellen Zustand des Systems ab. Reaktive Systeme können damit gut durch Automaten modelliert werden, wobei im Unterschied zu den klassischen Automaten der Automatentheorie auch das Zeitverhalten im Detail betrachtet werden muss. Vielfach wird eine maximale Reaktionszeit verlangt.

Def.: Ein EIS, welches Zeitbedingungen einhalten muss, heißt **Realzeit-System**.

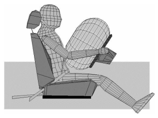
Def.: Zeitbedingungen, deren Nichteinhaltung zu einer Katastrophe führen kann, heißen **harte Zeitbedingungen** (engl. *hard real-time constraints*).

[1] Meist nennt man diese Systeme einfach **eingebettete Systeme**. Da dieser Begriff nicht sehr spezifisch ist, verwenden wir hier den Zusatz **informationsverarbeitend**. Aus demselben Grund wird im englischen z.Tl. die Bezeichnung *embedded computing system (ECS)* benutzt.

1.1.2 Beispiele eingebetteter Systeme

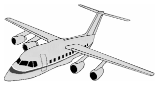
- **EIS in der Fahrzeugelektronik**

  – **Automobilelektronik**

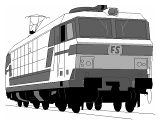
  Moderne Automobile sind überhaupt nur mit umfangreicher elektronischer Ausrüstung auf dem Markt konkurrenzfähig. Einfache Systeme aus Analogschaltkreisen werden dabei verstärkt durch digitale Informationsverabeitung abgelöst. Hochwertige Automobile besitzen aus diesem Grund schon ca. 100 Mikroprozessoren.

  – **Flugzeugelektronik**

  Zur Information der Piloten werden elektronische Systeme schon seit Jahrzehnten genutzt. Der Anteil der Elektronik wird dabei ständig ausgeweitet, wie z.B. durch die Einführung eines eigenen Radarsystems zur Vermeidung der Kollision mit anderen Flugzeugen. Eine große Diskussion wurde durch die Einführung des *fly-by-wire* Systems ausgelöst, welches durch hydraulische Steuerung in manchen Flugzeugmodellen abgelöst hat. Unfälle haben deutlich vor Augen geführt, welche hohen Anforderungen an den Entwurf zu stellen sind.

  – **Schienenfahrzeugelektronik**

  Moderne Schienenfahrzeuge nutzen die Elektronik für vielfältige Aufgaben, beispielsweise zur Kommunikation mit der Zugwegs-Sicherheitstechnik, zur Information der Reisenden, zum effizienten Umgang mit der Energie, zur Information des Zugführers über den Zustand des Zuges usw.

- **EIS in der Telekommunikation**

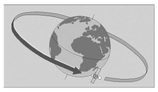
  

  Abb. 1.1 zeigt als Beispiel ein Blockdiagramm eines Ein-Chip Videotelefons [LNV+97].

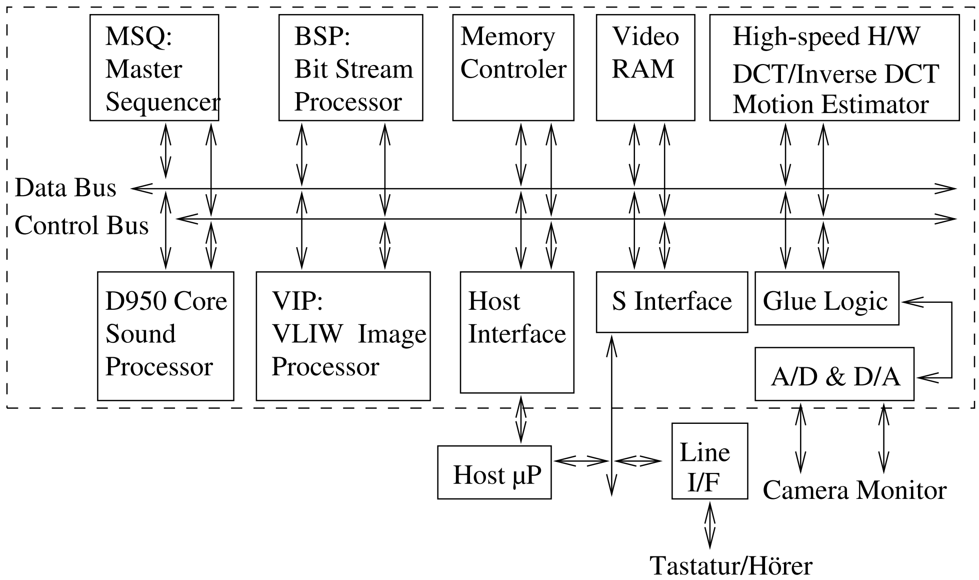
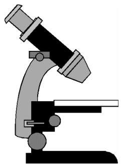


Abbildung 1.1: Ein-Chip Videotelefon (SGS Thomson)

- **EIS in der Medizintechnik** (medizinische Analyse- und Überwachungssysteme)



- **EIS in militärischen Anwendungen**
- **Zahlungssysteme auf der Basis EIS**

  Beispiel:

  Abb. 1.2 zeigt den sog. Smartpen.

  Dies ist ein Schreibgerät, welches der Identifizierung des Schreibers dient. Zur Identifizierung wird hier nicht nur die Unterschrift benutzt, sondern auch die Bewegungen des Schreibgeräts während des Unterschreibens. Hierzu gibt es verschiedene Sensoren im Schreibgerät, welches es einem Rechner erlauben, aus den per Funk übertragenen Daten ein Bild der Bewegung des Geräts im Raum zu rekonstuieren.

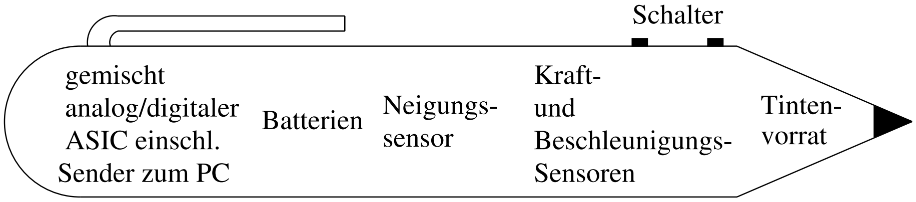
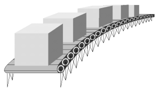


Abbildung 1.2: SMARTpen

- **EIS in der Fertigungstechnik** (Fabriksteuerungen, Industrieroboter, Speicherprogrammierbare Steuerungen)

  Beispiel (nach Kopetz [Kop97]):

  Gegeben sei ein Behälter mit einer Flüssigkeit, deren Durchflussmenge durch ein Ventil geregelt werden soll (siehe Abb. 1.3).

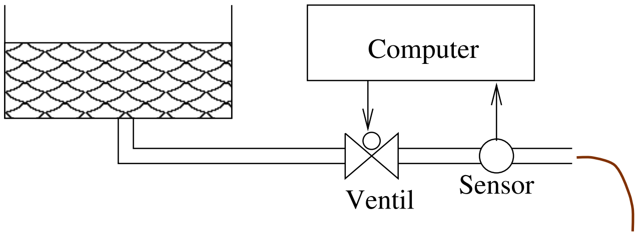


Abbildung 1.3: Regelung eines Ventils

Diese Durchflussmenge soll unter allen Umständen erhalten bleiben, unabhängig von dem Füllstand des Behälters, der Temperatur usw. Zu diesem Zweck enthält das System einen Sensor, der den Fluss misst. Das Ventil ist ein spezielles Beispiel eines **Aktors**, einer Komponente, über die das physikalische System gesteuert werden kann. Wie in diesem Beispiel werden viele Aktoren mit Sensoren kombiniert, um deren Funktion zu überwachen.

Bei der Konstruktion eines Regelungsverfahrens für dieses Beispiel muss beachtet werden, dass es eine gewisse Zeit dauert, bis das Ventil geöffnet oder geschlossen ist. Man kann z.B. annehmen, dass es 10 Sekunden dauert, das Ventil voll zu öffnen oder zu schließen. Weiter ist zu berücksichtigen, dass der Sensor auch nur eine begrenzte Genauigkeit hat, z.B. eine solche von 1%. Dann könnte man alle 100 ms einen Sensormesswert dem regelnden Computer zuführen, da sich in dieser Zeit auch das Ventil um 1% bewegen kann. Weiter ist die Zeitverzögerung zwischen Änderungen am Ventil und den Auswirkungen am Sensor zu berücksichtigen. Alle diese Effekte müssen bereits beim Entwurf eines einfachen Regelsystems beachtet werden. Viel komplexer sind Regelvorgänge mit vielen Sensoren und Aktoren.

Die Abbildungen 1.4 und 1.5 zeigen zwei Beispiele von Prototypen von Industrierobotern. Die Abbildung 1.4 zeigt einen sog. Rohr-Krabbler, ein Gerät, welches sich weitgehend selbständig (auch durch gekrümmte) Rohre bewegen kann.

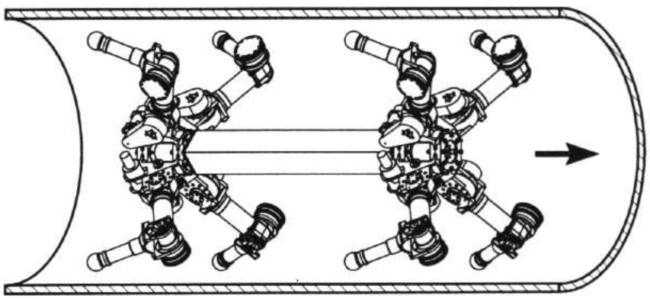


Abbildung 1.4: Rohr-Krabbler

Abbildung 8.12 zeigt das mechanische Modell einer Stabheuschrecke, welche ein Vorbild für den Rohr-Krabbler war.

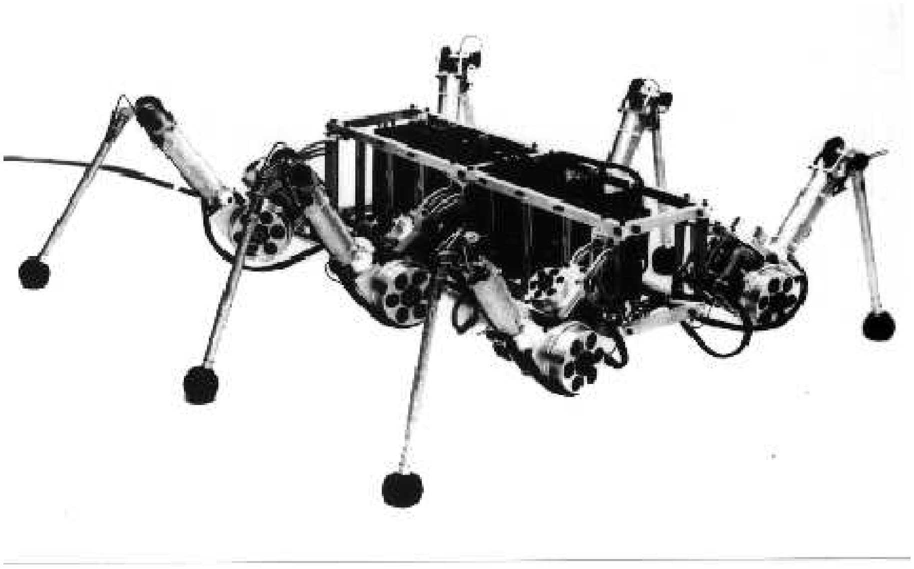


Abbildung 1.5: Stabheuschrecke (Modell)

- **EIS in der Gebäudeautomatisation**

  Beispielhaft sei hier ein System der Telefongesellschaft Helsinki (HSY) genannt [Pag98]:

  Bei diesem System melden Sensoren den Eintritt in einen intelligenten Raum. Licht und Belüftung werden in diesem Moment eingeschaltet. Gekoppelt ist dies mit Helligkeitsreglern und Sensoren. Um Kosten zu sparen, werden unbenutzte Räume nicht unnötig gewärmt oder gekühlt. Ein Kohlendioxid-Sensor beeinflusst die Luftzufuhr benutzter Räume. Das durch Lüfter verursachte Geräusch wird auf das jeweils benötigte Minimum reduziert. Ein zentraler Monitor gibt es jederzeit Auskunft über den Belegungszustand der einzelnen Räume. Der Energieverbrauch kann für jeden Raum erfasst werden. Das zur Kommunikation mit Sensoren und Regler benötigte Netz ist mit dem lokalen Rechnernetz integriert.

Diese Klassen und Beispiele geben einen Eindruck von der Vielfalt eingebetteter Systeme.

1.1.3 Markt für eingebettete Systeme

Der **Markt** für eingebettete Systeme ist sehr stark im Wachsen begriffen und wird voraussichtlich den Markt für Terminal-basierte Anwendungen übertreffen. Bereits heute werden nach Angaben der Zeitschrift *Electronic Design* 79% aller hochwertigen Mikroprozessoren in eingebetteten Systemen eingesetzt. Das bedeutet: pro Pentium-Prozessor, dessen man sich bewusst ist, existieren ca. vier hochwertige Prozessoren in eingebetteten Systemen, von deren Existenz man in der Regel nicht weiß. Gerade für Europa bilden eingebettete Systeme den Hauptmarkt im Bereich informationsverarbeitender Systeme (Rechner sind ja v.a. ein Haupt-Exportgut der USA). Daher zählt der Entwurf eingebetteter Systeme zu den häufigen Aufgaben europäischer Spezialisten für Informationsverarbeitung und soll daher hier behandelt werden.

Die Bedeutung eingebetteter Systeme wird vielfach unterschätzt. Dies belegt auch das folgende Zitat von Mary Ryan [Rya95]:

> *Embedded chips aren't hyped in TV and magazine ads ... but embedded chips form the backbone of the electronics driven world in which we live. ... they are part of almost everything that runs on electricity.*

Das Marktvolumen eingebetteter Systeme wird mit 31 Milliarden US $ angegeben (zum Vergleich: Marktvolumen für *general purpose computing*: 46,5 Milliarden US $) [Gup98]. Die Steigerungsrate des Marktvolumens für EIS bei 18% pro Jahr, beim *general purpose computing* bei 10% pro Jahr [Gup98].

Trends beinhalten eine zunehmende Flexibilität der Systeme durch den Einsatz von Prozessoren und Software statt von Hardware, eine Zunahme der durchschnittlichen Programmgröße und eine Zunahme der Entwicklungskomplexität.

1.1.4 Charakteristika eingebetteter Systeme

EIS besitzen in der Regel die folgenden Eigenschaften:

- EIS benötigen **Sensoren** zur Aufnahme von Daten, **Aktoren** zur Beeinflussung physikalischer Daten sowie ein *man/maschine interface* (MMI).
- Sie stellen häufig harte Anforderungen an das Zeitverhalten. Die Verarbeitungsgeschwindigkeit darf vielfach nicht stark schwanken (Forderung nach wenig *jitter*). Dies macht die Verwendung von *caches* und virtuellem Speicher problematisch.
- Sie sind reaktiv (d.h. Eingaben bewirken eine vom gegenwärtigen inneren Zustand abhängige Ausgabe und einen Übergang in einen neuen Zustand; das Ein/Ausgabeverhalten lässt sich also besser über Automaten, weniger über eine einzelne zu berechnende Funktion beschreiben.
- EIS sind für bestimmte Anwendungen entworfen. Ihr Verhalten ist meist während des Entwurfs vollständig bekannt. Es gibt keine neuen "Anwendungsprogramme" die später hinzukommen.
- Es werden vielfach hohe Anforderungen gestellt an:
  - die **Zuverlässigkeit** (die Wahrscheinlichkeit eines Ausfalls muss klein sein),
  - die **Sicherheit** (falls ein System ausfällt, dann dürfen keine gefährlichen Situationen eintreten),
  - die **Wartbarkeit** (ein ausgefallenes System muss schnell wieder zur Verfügung stehen)
  - die **Verfügbarkeit** (die Wahrscheinlichkeit eines nicht arbeitsfähigen Systems muss klein sein),
  - die *security* (die Vertraulichkeit und Authentizität von Daten muss gewährleistet sein)
- EIS sind in der Lehre und in öffentlichen Diskussionen vielfach unterrepräsentiert, da sie meist in Geräten "versteckt" sind und man sich ihrer Bedeutung nicht so leicht bewußt wird.

  > Embedded chips aren't hyped in TV and magazine ads ... but embedded chips form the backbone of the electronics driven world in which we live. ... they are part of almost everything that runs on electricity [Rya95].

  Auch sind EIS häufig so komplex, dass ein Entwurf im Rahmen von Lehrveranstaltungen schwierig ist.
- Es sind **effiziente Lösungen** erforderlich. Mobile Geräte müssen beispielsweise mit wenig Energie auskommen. Siliziumfläche sollte bei Massenproktukten effizient genutzt werden.
- EIS kommen vielfach ohne Tastatur, Maus und Bildschirm aus. Sie bedienen sich anderer Interfaces mit Benutzern; z.B. üblicher Anzeige- und Kontrollinstrumente in Fahrzeugen.

Wegen der letztgenannten Eigenschaft kann man salopp auch sagen: *alle informationsverarbeitenden Systeme, die ohne Bildschirm und Tastatur auskommen, sind EIS.*

Bei der Benutzung des Begriffs "EIS" werden vielfach die Unterschiede zwischen EIS in verschiedenen Anwendungsbereichen übersehen (und dies wird durch die zuletzt genannte Definition noch verstärkt). Portable Geräte stellen ganz andere Anforderungen als fest installierte Geräte, z.B. hinsichtlich der verfügbaren elektrischen Leistung.

Nicht jedes EIS muss alle o.a. Eigenschaften besitzen. Wird eine größere Zahl dieser Eigenschaften gefordert, dann wird man jedoch von einem EIS sprechen. Wegen der gemeinsamen Eigenschaften von EIS in verschiedenen Anwendungsbereichen lohnt es sich, zusammenfassend über den Entwurf von EIS zu sprechen und nicht jeden Anwedungsbereich separat zu betrachten.

In Anlehnung an Kopetz [Kop97] wollen wir hier auszugsweise zusammenstellen, was man aus einem einleitenden Kapitel zum Thema Realzeit-Systeme behalten sollte.

- *A real-time computer system must react to stimuli from the controlled object (or the operator) within the time interval* **dictated** *by its environment. If a catastrophe could result in case a firm deadline is missed, the deadline is called* **hard**.
- Die Wahrscheinlichkeit, dass ein perfekt entworfenes System mit garantierter Systemantwort versagt, ist durch die Wahrscheinlichkeit gegeben, dass die Annahmen über die Systemlast und die möglichen Fehler sich als falsch herausstellen.
- Die **Zuverlässigkeit** $R(t)$ ist die Wahrscheinlichkeit, dass ein System zum Zeitpunkt t noch korrekt funktioniert, unter der Annahme, dass es zu einem Zeitpunkt $t = 0$ korrekt funktionierte.
- **Wartbarkeit** (engl. *maintainability*) ist die Wahrscheinlichkeit $M(d)$, dass ein System in einer Zeit d nach einem Fehler wieder korrekt funktionieren kann.
- **Verfügbarkeit** (engl. *availability*) ist die Wahrscheinlichkeit, dass ein System zum Zeitpunkt t eine korrekte Funktion anbieten kann.
- Eine **garantierte Systemantwort** (*engl. guaranteed system response*) muss auch ohne Argumentation mit Wahrscheinlichkeiten begründet werden können.
- *An embedded real-time system is part of a well-specified larger system, an* **intelligent product**. ...

- Die Vorab-Kenntnis der Produktfunktion kann ausgenutzt werden, um den Ressourcenbedarf und die Robustheit des Produkts zu verbessern.
- Die Kosten einer Produktionsanlage sind in der Regel weniger wichtig als die Kosten der Produktion.
- *The embedded system market is expected to grow significantly ...*

1.2 Motivation und Struktur dieser Vorlesung

Aus den folgenden Gründen ist eine Beschäftigung mit dem Thema der Vorlesung wichtig:

- Die Bedeutung von EIS, gerade für die Wirtschaft Europa, ist größer als allgemein angenommen wird.
- EIS werden in Vorlesungen üblicherweise nur am Rande oder implizit behandelt.
- Gerade der Lehre an einer technisch ausgerichteten Hochschule und in einem Studiengang "Ingenieurinformatik" kommen EIS eine besondere Bedeutung zu.
- Wenngleich eine Überlappung mit den Inhalten von manchen Vorlesungen der Fakultät Elektrotechnik vorliegt, ist eine separate Vorlesung für die Informatik-Studiengänge wichtig. Die einschlägigen Veranstaltungen der Fakultät sind nicht breit angelegt, wie es für Informatiker aufgrund des möglichen Stundenvolumens notwendig ist und können natürlich nicht alle Bezüge zu Informatikveranstaltungen herstellen.

Die Vorlesung behandelt insbesondere folgende Themen:

- Hardware eingebetteter Systeme (Sensoren, Verarbeitungseinheiten, Prozessbusse, Aktoren)
- Digitale Signalverarbeitung
- Realzeit-Betriebssysteme
- Realzeit-Sprachen
- Entwicklungsumgebungen
- Prozess-Steuerungen und -Regelungen
- Roboter und maschinelles Sehen

Die Vorlesung muss also relativ umfassend gestaltet werden.

Bei der Planung der Vorlesung stellt sich immer wieder die Frage: Präsentation des Stoffes *top down* oder *bottom up*? Für diese Vorlesung haben wir uns weitgehend (d.h.: mit Ausnahme des ersten Kapitels) für ein *bottom up*-Vorgehen entschieden. Wir werden also mit der Hardware eingebetteter Systeme beginnen. Dies sollte nicht dem Entwurfsprozess entsprechen, bietet aber zwei Vorteile:

1. Man beginnt "auf sicherem Boden" und mit Themen, die für praktisch alle eingebetteten Systeme relevant sind. Bei einem *top-down*-Vorgehen hängt die Themenwahl stark von dem Anwendungsbereich der EIS ab.

2. Bei einem *top-down*-Vorgehen müsste recht früh über Spezifikationssprachen gesprochen werden, die aber bislang Gegenstand der Vorlesung "Rechnergestützter Entwurf und Produktion" sind.

1.3 Einbindung der Vorlesung in das Lehrangebot

Während CAD-Methoden für den Entwurf der Mikroelektronik von EIS Gegenstand der Vorlesung "Rechnergestützter Entwurf und Produktion (Mikroelektronik)"ist, soll in der Vorlesung "Prozessrechnertechnik" ein grundsätzlicher Einblick in den möglichen Aufbau und die Arbeitsweisen von EIS gegeben werden (siehe Abb. 1.6).

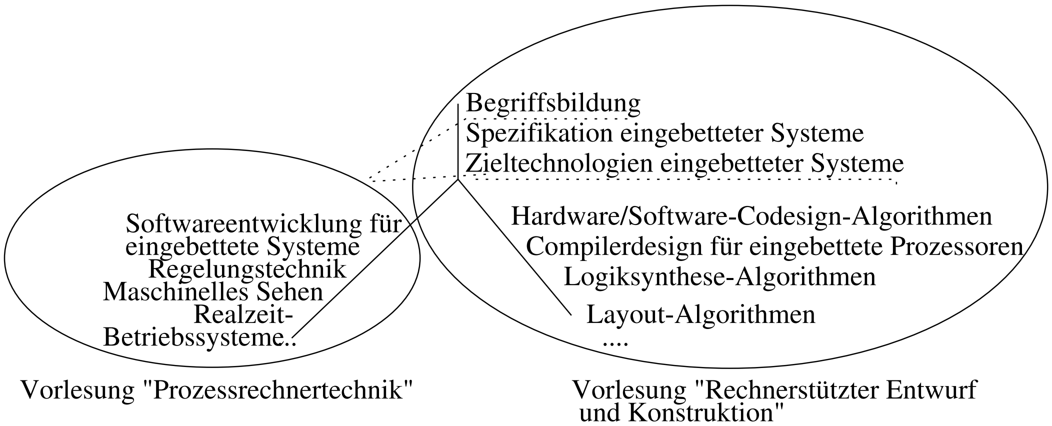


Abbildung 1.6: Relation zur CAD-Vorlesung

Es geplant, die Aufteilung des Stoffes auf die Vorlesungen künftig zu reorganisieren: eine einführende Vorlesung "Eingebettete Systeme" würde danach Spezifikationsmethoden, Zielarchitekturen und grundlegende Designtechniken enthalten. Eine darauf aufbauende CAD-Vorlesung würde die CAD-Algorithmen einschliessen. Dieser Aufbau setzt aber eine neue Prüfungsordnung voraus.

Vorlesungen über eingebettete Systeme sind auch an vielen Informatikfachbereichen üblich, so z.B. in Aachen, Karlsruhe, Darmstadt, München und Bielefeld. Aufgrund der lokalen Gegebenheiten kann dabei an der Universität Dortmund der Stoff nur in einer einzigen Vorlesung geboten werden, obwohl an anderen Standorten mehrere Veranstaltungen (etwa mit Titeln wie "Realzeitsysteme", "Robotics", "eingebettete Systeme") in den beschriebenen Bereich fallen. In Kaiserslautern gibt es sogar eine in einer Prüfungsordnung niedergelegte Studienrichtung "Eingebettete Systeme".

Im Sinne der Prüfungsordnungen wird die Vorlesung wie eine Vorlesung "Prozessrechnertechnik" ohne irgendwelche Namenszusätze behandelt. Sie ist damit eine Wahlpflicht-Vorlesung des Studiengangs "Angewandte Informatik" und gleichzeitig eine Spezialvorlesung für die Studiengänge "Informatik"und "Lehramt Informatik, Sek. II". Über den Inhalt der Vorlesung können mündliche Prüfungen abgelegt werden.

Die Vorlesung besitzt thematische Anknüpfungspunkte an andere Stamm- und Wahlpflichtvorlesungen des Fachbereichs. Es ergeben sich insbesondere thematische Beziehungen zu den Vorlesungen "Rechnergestützter Entwurf und Produktion (Mikroelektronik)", "Software-Technologie" "Künstliche Intelligenz" und "Betriebssysteme". Manche (aber längst nicht alle) der Themen werden ausführlicher in Vorlesungen der Fakultät Elektrotechnik behandelt. Für Informatiker dürfte der Besuch aller einschlägigen Vorlesungen dieser Fakultät allerdings aus Aufwandsgründen und wegen der zu hohen Spezialisierung ausscheiden.

Gelegentlich kann ein besseres Verständnis des Stoffes zu erzielen sein, wenn die Vorlesung "Rechnerarchitektur" vor dem Besuch der Vorlesung "Prozessrechnertechnik" gehört wurde. Allerdings soll nicht ausgeschlossen werden, dass die Vorlesung "Prozessrechnertechnik" ohne Vorliegen dieser Voraussetzung gehört wird, da benötigte Grundlagen zumindest innerhalb der Vorlesung (wenngleich nicht immer in diesem Skript) kurz angesprochen wwerden.

1.4 Literatur

Leider ist es nicht möglich, sich bei der Stoffauswahl auf eine kleine Zahl von umfassenden Büchern zu beziehen. Es gibt eine Reihe älterer Bücher über Prozessdaten-Verarbeitung, die sich jedoch vielfach nur mit der Hardware beschäftigen oder Themen in den Vordergrund stellen, die Informatik-Studenten ohnehin bekannt sind. Neuere Bücher behandeln vielfach nur Teilaspekte des oben umrissenen Themenkreises. Aufgrund des breit angelegten Themenkreises reichen diese Bücher allein nicht aus. Die Vorlesung basiert daher auf einer großen Zahl von Büchern, einzelnen Zeitschriftenartikeln und zum kleinen Teil Informationen aus den Skripten R. Gupta [Gup98].

Zu den einzelnen Gebieten sollen hier die folgenden Bücher besonders genannt werden:

- Hardware eingebetteter Systeme
  Die Verarbeitung von physikalischen Daten beginnt mit deren Erfassung mittels Sensoren. Der große Bereich solcher Sensoren kann nur anhand einiger, weniger Auszüge aus Zeitschriften- und Prospektmaterial gestreift werden.

  Erfasste Daten müssen den Verarbeitungseinheiten zugeführt werden. Das Buch von Schürmann [Sch97b] enthält eine sehr gute Darstellung der hierfür geeigneten Prozessbusse und von Bussen in der Automatisierungstechnik, auf die zurückgegriffen werden soll. Eine ältere Darstellung von Hardwaretechnologie findet sich im Buch von Färber [Fae94].

  Zur eigentlichen Verarbeitung geeignete Prozessoren und Spezialhardware wird bereits im Skript "Rechnergestützter Entwurf und Konstruktion" [Mar98] beschrieben und hier z. Tl. übernommen.

  Für den Bereich der steuernden Hardwareelemente (Aktoren) und der Anzeigeelemente muss wieder auf einzelne Artikel zurückgegriffen werden.

- Digitale Signalverarbeitung

  Für dieses Gebiet wird v.a. auf das Buch von Oppenheim und Schäfer [OS95] zurückgegriffen. Weitere einschlägige Bücher stammen von Embree und Kimble [EK91] sowie von Kammeyer und Kroschel [KK89].

- Realzeit-Betriebssysteme

  Einige Kernanforderungen an Realzeit-Betriebssysteme (engl. *real-time operating systems, RTOS*) werden bei Kopetz [Kop97] beschrieben und hier übernommen. Weiter werden wir auf wenige Beispiele erhältlicher Realzeitbetriessysteme eingehen.

- Realzeit-Sprachen

  Zu diesem Thema gibt es eine Fülle von Büchern. Grundlage der laufenden Vorlesung ist v.a. das Buch von Burns und Welling [BW90]. Ergänzend können die Bücher von Zöbel [Zoe87], Young [You82] und Fiedler[Fie94] sowie Material über Java [Job96] benutzt werden. .

- Entwicklungsumgebungen

  Der Entwicklungsprozess wird u.a. in den Büchern Krishna et al. [KS97] sowie von Burns und Welling [BW90] diskutiert.

- Prozess-Steuerungen und -Regelungen

  Prozess-Steuerungen und -Regelungen stellen ein relativ klassisches Gebiet dar, für das eine Vielzahl von Büchern existiert. In der Vorlesung wird ein Schwergewicht im Bereich der Fuzzy-Systeme liegen. Hierfür werden wir v.a. das Buch von Kiendl [Kie97] benutzen Weitere Bücher dieses Gebiets stammen z.B. von Pressler [Pre67] und Kruse [KGK95].

- Roboter

  Roboter werden in dem vorliegenden Text nur kurz anhand des Buches von Kreuzer et al. [KMLT94] dargestellt. Ergänzend kann das Buch von Fu, Gonzales und Lee [FGL87] benutzt werden.

- Mustererkennung

  Für die Mustererkennung benutzen wir das Buch von Fritsch [Fri91]. Ergänzend kann wieder das Buch von Fu, Gonzales und Lee [FGL87] eingesetzt werden.

Die jeweils aktuelle Fassung des vorliegenden Skripts ist über die Web-Seite des Lehrstuhls 12 (http://ls12-www.cs.uni-dortmund.de) zu beziehen (Format: postscript). Für Hilfe bei der Erstellung des Skripts sei folgenden Personen gedankt: Frau Bauer für das Schreiben der vieler Abschnitte, den Kollegen Krumm und Müller für Hinweise bei der Vorbereitung, Herrn Markhof für die Rechnerbetreuung und den Autoren von Linux, Latex und des Editors `ce` für die Arbeitsumgebung, in der die postscript-files erzeugt wurden.

Kapitel 2

Hardware eingebetteter Systeme

2.1 Sensoren

Die Verarbeitung von physikalischen Daten beginnt mit deren Erfassung mittels Sensoren. Informationsverarbeitung physikalischer Größen setzt voraus, daß diese Größen zunächst einmal erfaßt und der (z. Zt. praktisch ausschließlich elektronischen) Informationsverarbeitung zugänglich gemacht werden. Dies leisten die **Sensoren**. Im Bereich der Sensoren hat es dabei in den letzten Jahren eine enorme Entwicklung gegeben, welche nicht eine der Bedeutung entsprechende Beachtung gefunden hat. Ohne die rapide Entwicklung in der Sensorik wären kaum die Fortschritte beispielsweise der Automobilelektronik möglich gewesen. 2. Vorles.

Mit Sensoren können praktisch alle physikalischen Größen erfaßt werden, also z.B.

- Abmessungen
- Gewichte
- Beschleunigungen
- Feldstärken
- Temperatur
- Helligkeiten

u.s.w.

Auch für andere Größen gibt es Sensoren, also beispielsweise für chemische Stoffe.

Eine große Zahl von physikalischen Gesetzen und Effekten kann zur Realisierung von Sensoren genutzt werden. Als Beispiele seien hier genannt:

- das Induktionsgesetz (Erzeugung einer Spannung durch ein Magnetfeld)
- der Hall-Effekt (-"-)
- lichtelektrische Effekte (Veränderung der Leitfähigkeit durch Licht)

Insgesamt ist das Gebiet der Sensorik so groß, daß hier kein vollständiger Überblick gegeben werden kann.

<u>Beispiele jüngerer Entwicklungen:</u>

- Beschleunigungsensoren auf der Basis der Mikrosystemtechnik
- Bewegungssensoren auf der Basis des Halleffekts

  Auf der Basis des Halleffekts kann eine Änderung eines Magnetfeldes in eine Spannung umgesetzt werden (Abb. 2.1).

  Anwendungen finden sich z.B. bei ABS-Systemen oder bei Gleichstrommotoren, die ohne Kohlenschleifer auskommen.

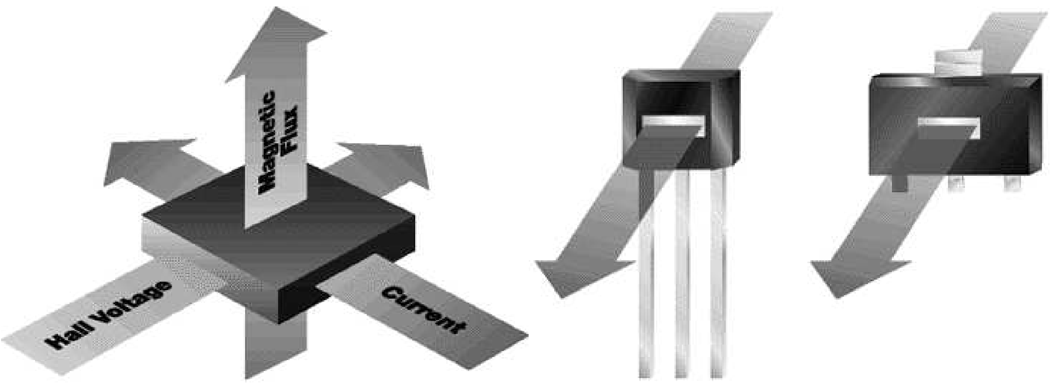


Abbildung 2.1: Halleffekt und passende Sensoren(©IIT Intermetall)

- Sensoren für Fingerabdrücke auf der Basis der CMOS-Technik (siehe Abb. 2.2)

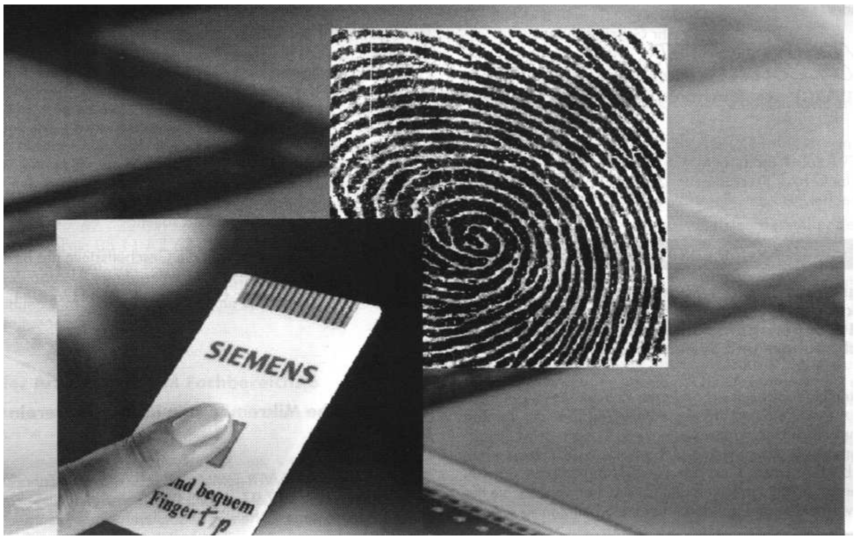


Abbildung 2.2: Sensor zur Erkennung von Fingerabdrücken (©VDE, Siemens)

Der Sensor besteht aus 256 × 256 Sensorelementen. Er hat eine Auflösing von 500 dpi.

> Wird der Finger auf die Siliziumfläche des Chips gelegt, nehmen Sensorzellen die Änderungen des elektrischen Feldes auf, das die erhabenen Linien und die Vertiefungen auf der Fingerfläche hervorrufen, und erzeugen daraus ein elektrisches Abbild [Sie98].

Da sich ein dreidimensionales Bild des Fingers ergibt, kann der Sensor auch nicht mittels Fotos überlistet werden. Da auch die Leitfähigkeit gemessen wird, reicht auch kein Wachsabdruck zur Identifizierung.

- Kameras auf der Basis von *charge coupled devices* (CCDs; deutsch: Eimerkettenschaltungen).

> *Obwohl die Geometriegenauigkeit und das Auflösungsvermögen der Röhrenkameras auf ein beträchliches Niveau gebracht werden konnte, werden sich für das Maschinelle Sehen Kameras auf der Basis von Festkörperbildsensoren durchsetzen. Dies sind hochintegrierte optomikroelektronische Schaltkreise mit einem Rechteckmosaik von von strahlungempfindlichen Elementen, bei denen wie im Target einer Bildaufnahmeröhre ein dem auftreffendem Strahlungsstrom proportionales Ladungsbild erzeugt wird. Die Ladungen werden über Register und entsprechende Leitungen getaktet ausgelesen.... Sensoren nach dem Interline- (Spaltenauslese-) Prinzp bestehen im wesentlichen aus nebeneinanderangeordneten CCD-Zeilen. Sensoren nach dem Frametransfer- (Bildauslese-) Prinzip enthalten eine lichtempfindliche Matrix von Einzelsensoren, die innerhalb eines Bruchteils der Integrationszeit*

> *in eine zweite lichtgeschützte Bildpuffermatrix ausgelesen und zwischengespeichert wird.* [Zitat Fritsch [Fri91]]

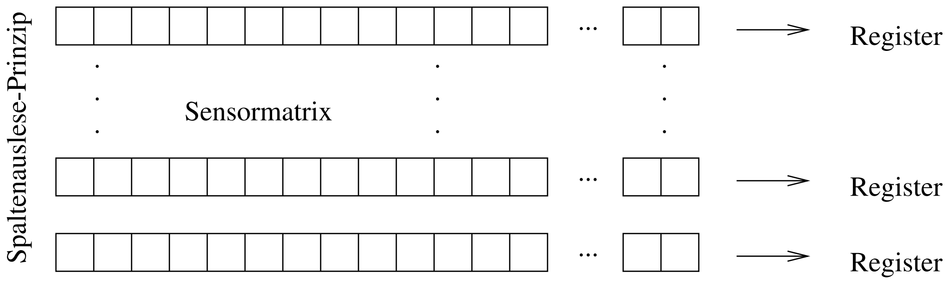


Abbildung 2.3: Spaltenauslese-Prinzip

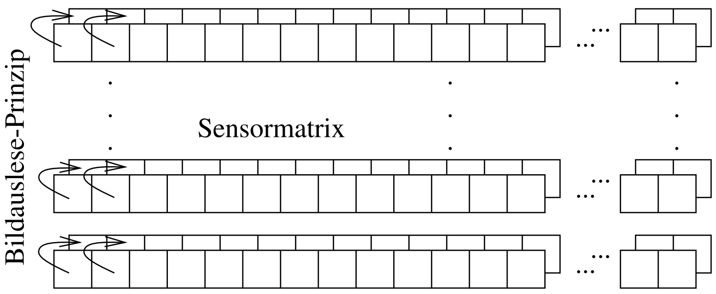


Abbildung 2.4: Bildauslese-Prinzip

- Lichtempfindliche Sensoren, welche bei Blinden in das Auge implantiert werden können, (das sog. künstliche Auge)
- Abgassensoren zur Motorregelung
- Strahlungssensoren für Kraftwerke und die Weltraumfahrt
- Regensensoren zum automatischen Start von Scheibenwischern.

> *Sensoren vermehren sich wie Kaninchen. Heinzelmännchen im modernen Auto. Die fünf Sinne reichen nicht mehr zum Autofahren.* [Zitat ITT Automotive, web-Seiten der Fa. ITT]

2.2 Diskretisierung der Zeit: Sample-and-hold-Schaltungen

In dieser Vorlesung werden wir nur die Informationsverarbeitung mit digitalen Komponenten betrachten[1]. Da mit bekannten Digitalrechnern nur zeitdiskrete Folgen von Werten verarbeitet werden können, müssen müssen wir auf zeitdiskrete Folgen übergehen. Auf die Folgen hiervon werden wir in Kapitel 3 eingehen.

Die Zeitdiskretisierung kann mit Hilfe von *sample and hold*-Schaltungen vorgenommen werden (siehe Abb. 2.5).

Idealerweise würde das Taktsignal den Schalttransistor nur extrem (infinitesimal) kurz öffnen lassen und dabei die Amplitude zu 100% an den Kondensator weiterleiten. Spannungswerte am Kondensator zu den Takzeitpunkten würden dann die Folgenwerte bilden. Reale Sample-and-hold-Schaltungen benötigen allerdings eine gewisse Zeit zum Aufladen des Kondensators.

[1] Allerdings beklagt die Industrie eine unzureichende Ausbildung von Analogdesignern. Hier wäre eine Ausbildungsaufgabe (die aber nicht mit dieser Vorlesung gelöst werden kann und soll).

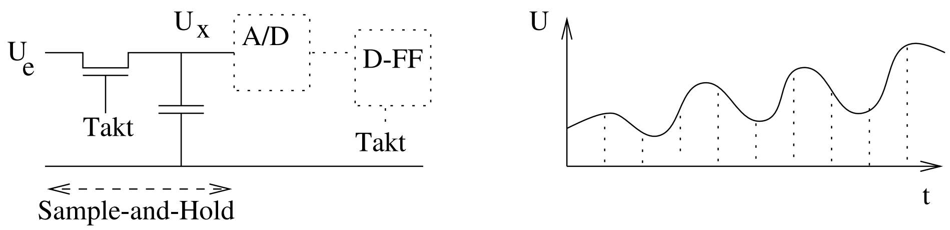


Abbildung 2.5: Sample-and-hold-Schaltung

2.3 Diskretisierung der Amplitude: Analog/Digitalwandler

Viele der Eingabewerte eines eingebetteten Systems sind analoge Werte. Wir schließen für den folgenden Teil der Vorlesung eine analoge Informationsverarbeitung aus.

Die einzelnen Folgeelemente sind also zu digitalisieren. Hierfür gibt es viele Verfahren. (siehe z. B. Bähring [Bae94], Zimmermann [ZH74]). Wir wollen hier zwei Verfahren vorstellen, die sich in Geschwindigkeit und Genauigkeit unterscheiden:

1. Direkter Vergleich

   Das Prinzip dieser Verfahrens ist sehr einfach (siehe Abb. 2.6).

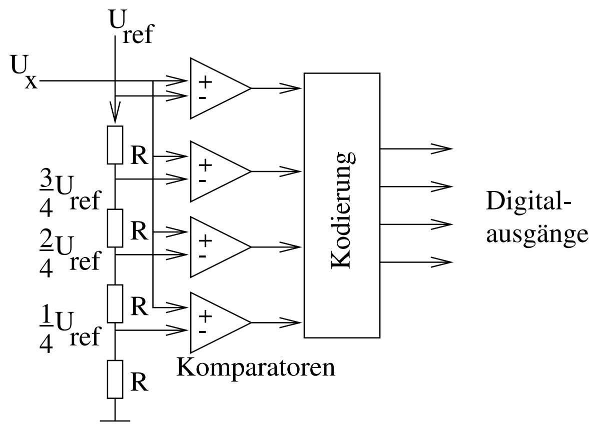
   

   Abbildung 2.6: A/D-Wandler mit direktem Vergleich

   Die Analogspannung wird parallel, d.h. gleichzeitig, mit verschiedenen genau abgestuften Referenzspannungen verglichen. Die Kodierstufe erkennt den höchsten Referenzwert, der von U_x noch überschritten wird und leitet ein entsprechendes Signal an die Digitalausgänge [ZH74]. Da die Vergleichsspannungen gleichzeitig zur Verfügung stehen, sind AD-Wandler auf der Basis des direkten Vergleichs sehr schnell (man kann sagen, der Zeitaufwand zur Wandlung ist unabhängig von der Auflösung, oder $O(1)$).

   Wandler auf dieser Basis können z. Tl. zur direkten Digitalisierung von HF-Signalen eingesetzt werden. Diese Wandler sind aber sehr aufwendig. Zur Erzeugung von Digitalwerten einer Genauigkeit von n Werten wird ein Aufwand von $n - 1$ Vergleichern benötigt.

   Diese Wandler werden daher v.a. zur Digitalisierung von schnellen Vorgängen, für die eine begrenzte Genauigkeit ausreicht, eingesetzt. Dazu gehört z.B. die Digitalisierung von Bewegtbildern.

2. Wandler nach dem Wägeprinzip

   Wandler nach diesem Prinzip folgen dem in Abb. 2.7 angegebenen Schema.

   Die Grundidee der Wandler basiert auf der binären Suche. Nacheinander werden die einzelnen Bits des Digitalausgangs auf '1' gesetzt und analysiert, ob der entsprechende Analogwert größer oder kleiner als der zu wandelnde Wert ist. Ist er größer, wird das betreffende Bit auf '0' gesetzt, sonst bleibt es erhalten. Der Vorgang beginnt bei dem höchstwertigen Bit und endet bei dem niedrigwertigsten (engl. *successive approximation*).

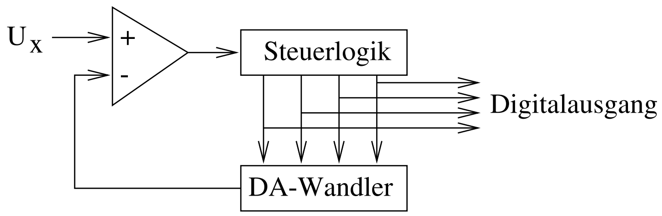


Abbildung 2.7: A/D-Wandler nach dem Wägeprinzip

Realisierungen von D/A-Wandlern werden im Abschnitt 2.6 beschrieben.

Der Aufwand zur Erzeugung von digitaler Codes mit n möglichen Werten ist durch den Aufwand für die D/A-Wandler gegeben (in der Regel $ld(n)$), die Zeitkomplexität ebenfalls $ld(n)$.

Diese Wandler sind weniger aufwendig als die im vorigen Abschnitt beschriebenen und langsamer als diese, lassen aber bei geringem Aufwand eine hohe Genauigkeit zu (diese ist im wesentlichen durch die Genauigkeit des D/A-Wandlers bestimmt).

2.4 Prozesbusse, Busse in der Automatisierungstechnik

2.4.1 Anforderungen

Die zu bearbeitenden Größen müssen in einem komplexen System zwischen einer Vielzahl von Geräten und Komponenten ausgetauscht werden. Im allgemeinen ist daher ein hierarchisches Modell des Gesamtsystems erforderlich (siehe Abb. 2.8)[2].

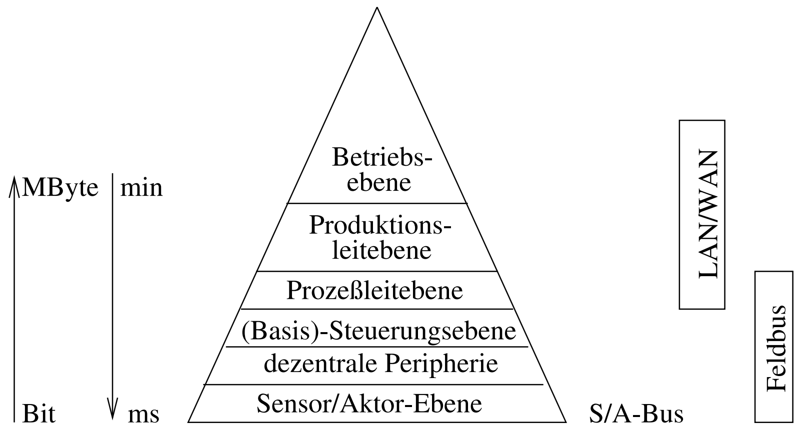


Abbildung 2.8: Netzwerkhierarchie

Da beispielsweise die Kommunikation mit einem Roboter in der Fabrikhalle ganz andere Anforderungen besitzt als die Kopplung zweier Workstations in einem Planungsbüro, lassen sich die unterschiedlichen Anforderungen nicht durch ein einziges Kommunikationsprotokoll realisieren. Die prinzipielle Tendenz bei diesen Anforderungen ist, daß auf den oberen Ebenen große Datenpakete bei geringen Anforderungen an das Einhalten von Zeitbedingungen ausgetauscht werden, wohingegen auf den unteren Ebenen wenige Bits innerhalb von Sekundenbruchteilen ausgetauscht werden müssen. Diesen Anforderungen entsprechend wurden ebenenspezifische Bussysteme und Protokolle entwickelt (siehe Abb. 2.8, rechts). Diese müssen in einer industriellen Umgebung meist die folgenden Eigenschaften aufweisen [Sch97b]:

- Unempfindlichkeit gegenüber Störungen und Beschädigungen
- Fehlertoleranz
- leichte und schnelle Wartbarkeit, Fehlerdiagnosemöglichkeit

[2] Diese Abbildung orientiert sich ebenso wie der übrige Inhalt dieses Abschnitts an dem Buch von Schürmann [Sch97b].

- Echtzeitverhalten des Buszugriffverfahrens
- Möglichkeit der ereignisorientierten Kommunikation
- auf den Einsatzbereich zugeschnittene Übertragungsgeschwindigkeit
- Wirtschaftlichkeit

Um die Kosten gering zu halten, sieht man häufig vor, dass Geräte mit geringer Stromaufnahme ihren Betriebsstrom über den Bus statt über eigene Netzteile beziehen können.

2.4.2 Basistechniken für verlässliche Kommunikation

Zur Verbesserung der Unempfindlichkeit gegenüber Störungen werden verschiedene Techniken eingesetzt, wie etwa die Abschirmung der Kabel (z.B. STP-Kabel), der Einsatz optischer Übertragungsstrecken oder die Benutzung differentieller Bussignale.

Bei der Übertragung von Informationen über Kabel ist es vielfach üblich, Informationen durch Spannungen darzustellen, die auf eine Masseleitung oder auf "Null" bezogen gemessen werden. So kann eine '1' durch einen Spannungsbereich (z.B. von 3,5 bis 5 Volt) zwischen einem Signalkabel *und Masse* kodiert werden. Man bezeichnet diese Form der Signalübertragung als *single ended* oder *asymmetrisch*.

Wenn man an einer hohen Übertragungssicherheit interessiert ist, verwendet man statt "single ended"-Signalen sog. *differentielle Signale* oder *symmetrische Signale*. Für jedes differentielle Signal werden zwei Leitungen benötigt, über die jeweils gegenpolige Spannungen geführt werden, bei einer Spannung von U_+ auf der einer Leitung also -U_+ auf der anderen (siehe Abb. 2.9). An der Empfangsseite befindet sich ein Differenzverstärker. Der Differenzverstärker verstärkt die Spannungsdifferenz zwischen seinen beiden Eingängen um einige Zehnerpotenzen, wobei der resultierende Wert durch die Betriebsspannung oder einen anderen festen Wert (z.B. 0 Volt) begrenzt wird. Ist die Spannung zwischen beiden Eingängen dieses Verstärkers positiv, so generiert er so eine '1' und sonst eine '0'.

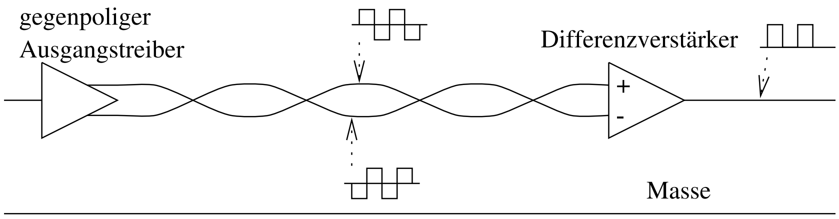


Abbildung 2.9: Differentielle Signale

Meist kombiniert man die Verwendung differentieller Signale mit der Verwendung von verdrillten Kabeln (sog. *twisted pairs*), da auf diese Störungen relativ gleichmäßig übertragen werden.

Die Vorteile differentieller Signale sind die folgenden:

- Störsignale addieren sich in der Regel zu den Signalen auf beiden Leitungen, bleiben also durch die Differenzbildung wirkungslos.
- Der resultierende Logikwert hängt nur von der gegenseitigen Polarität der Signale am Differenzverstärker ab. Änderungen der absoluten Größe der Spannungen durch Dämpfung des Signals entlang der Leitung oder Reflexion spielen keine Rolle.
- Die Anforderungen an die Qualität der Masseverbindungen sind geringer, da über diese keine Signalströme fließen.
- Mit differentiellen Signalen sind wegen der bereits genannten Eigenschaftten meist höhere Übertragungsraten als mit *single ended* Signalen möglich.

Die Nachteile differentieller Signale sind:

- Es werden negative Spannungen benötigt (es sei denn, man arbeitet mit komplementären (logisch negierten) statt mit gegenpoligen Signalen).
- Es werden doppelt so viele Signalkabel und Steckeranschlüsse benötigt.

Differentielle Signale finden u.a. in den folgenden Bereichen Anwendung: *differential SCSI*, *token ring*-Netze (s.u.), Datenübertragung nach dem RS 422-Standard, ISDN, *shielded/unshielded twisted pair* Kabel (sog. STP- bzw. UTP-Kabel), hochwertige analoge Audioübertragung, u.a.m.

Zur Verbesserung der Fehlertoleranz werden fehlererkennende und fehlerkorrigierende Busprotokolle benutzt.

Um das Echtzeitverhalten zu garantieren, verwendet man spezielle Busprotokolle. Das vom Ethernet her bekannte CSMA/CD-Verfahren (*carrier sense, multiple access/collision detect*) beispielsweise ist nicht geeignet, weil es Konflikte beim Zugriff auf den Bus auf eine Weise auflöst, die keinerlei obere Zeitschranke für die Zuteilung des Busses garantiert.

Eine Alternative ist das CSMA/CA-Verfahren (CA steht für collision avoidance). *Jeder Teilnehmer erhält eine Kennung (ID), die seiner Priorität entspricht. Nach Beendigung einer Busübertragung beginnen alle sendewilligen Teilnehmer gleichzeitig (synchron) ihre Kennung zu senden, wobei die Busleitung eine wired-OR ... Verknüpfung realisiert. ...* Bei einer wired-OR-Verknüpfung ist eine '1' auf dem Bus dominant gegenüber einer '0' [Bae94]. *Die Übertragung beginnt mit dem höchstwertigen Bit. Sobald die auf dem Bus anliegende, durch Überlagerung verknüpfte Kennung größer als seine eigene Kennung ist, zieht sich der entsprechende Teilnehmer von der Arbitrierung zurück und versucht später wieder zu senden. ... Voraussetzung zur Erkennung des jeweils dominanten Bits ist, dass die Signallaufzeit* t_s *vernachlässigbar klein gegenüber der Schrittweite (Dauer eines Bits) ist ....*

Ist die Paketlänge begrenzt, hat der Teilnehmer mit höchster Priorität Echtzeitverhalten. Der Bus kann allerdings für niederpriore Teilnehmer blockiert werden, wenn der Teilnehmer mit höchster Priorität ständig senden würde. Er sollte daher nach einem Übertragungszyklus eine bestimmte Zeit warten, bevor er sich wieder um den Bus bemüht.... [Zitat Schürmann].

Eine andere Technik, Realzeitverhalten zu garantieren, besteht in der Kommunikation mittels *token rings* (siehe [Mar97]) oder den verwandten *token busses.*

Weitere Basistechniken betreffen beispielsweise den Schutz vertraulischer Daten mittels Kryptographie, den Aufbau virtuell privater Netze usw.

2.4.3 Sensor/Aktor-Busse

2.4.3.1 Prinzipien

Sensor/Aktor-Busse bilden das Übertragungsmedium auf der den physikalischen Größen nächsten Ebene. Sie verbinden Sensoren und Aktoren vielfach mit Robotern, numerischen Steuerungen (CNC-Steuerungen] und speicherprogrammierbaren Steuerungen (SPS). Echtzeitverhalten und Kosten spielen bei diesen Bussen eine große Rolle.

Nach Schürmann [Sch97b] gibt es drei mögliche Realisierungen der Anschlussorganisation :

1. Sensoren sind direkt mit der Steuerung verbunden (siehe Abb. 2.10 links). Hierfür werden nur einfache Sensoren, aber viele Leitungen benötigt.
2. Sensoren sind über spezielle Anschlußeinheiten über einen Bus mit der Steuerung verbunden (siehe Abb. 2.10 mitte). Hierfür reichen einfache Sensoren aus, die Leitungskosten können aber reduziert werden.
3. Sensoren haben selbst einen Busanschluß, über den sie verbunden sind. Dies ist die flexibelste, zugleich aber bezüglich der Sensoren eine evtl. teure Lösung.

Da Sensoren sehr einfach sein können (sie können z.B. aus einem einzigen Schalter bestehen), spielen die Anschlusskosten eine erhebliche Rolle.

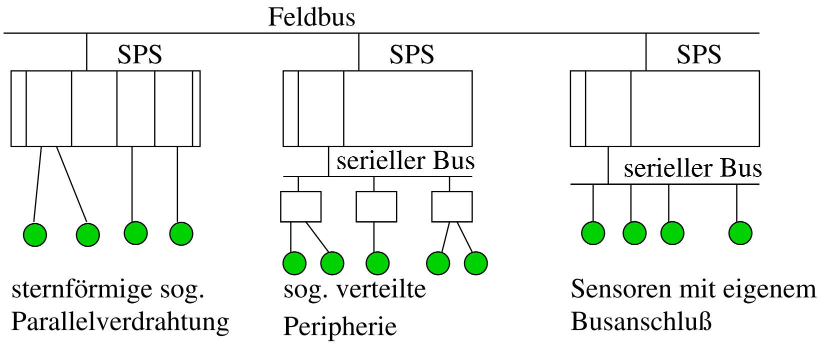


Abbildung 2.10: Anschlussmöglichkeiten von Sensoren

2.4.3.2 Interbus-S

Als Beispiel eines Sensor/Aktorbusses betrachten den Interbus-S.

Dieser Bus besitzt eine Ringstruktur (siehe Abb. 2.11).

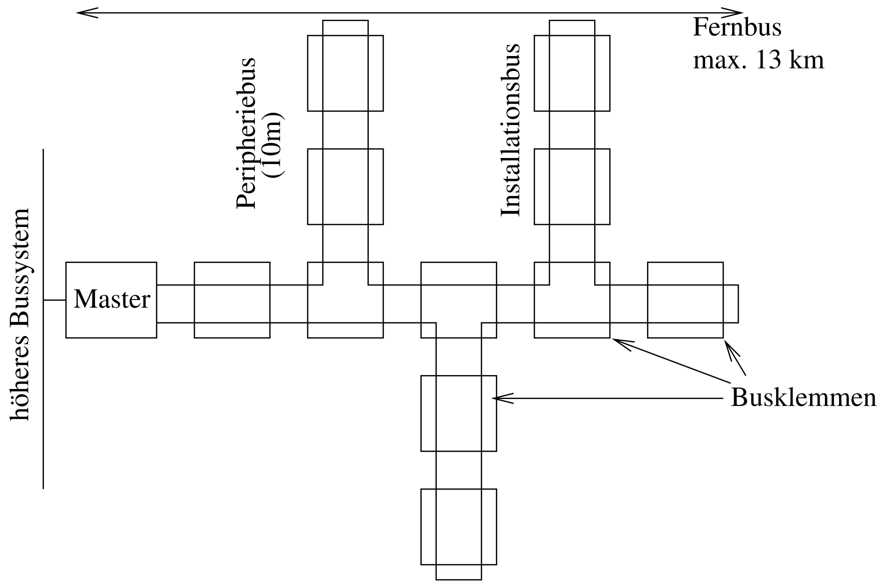


Abbildung 2.11: Topologie des Interbus-S

Als Übertragungsmedium wird ein Lichtwellenleiter oder eine 5-adrige Twisted-Pair-Leitung benutzt. Das Netzwerk darf eine Gesamtlänge von 13 km besitzen.

Auf die Anwendung angepaßt ist das Übertragungsprotokoll: es wird ein Zeitscheibenverfahren benutzt, bei dem jeder Busteilnehmer einen festen Zeitschlitz erhält. Dieser Zeitschlitz wird den einzelnen Stationen verschoben (siehe Abb. 2.12).

Das Zeitschlitzverfahren sorgt für eine vorhersagbare Kommunikationszeit.

2.4.3.3 Das Sensor/Aktor-Interface ASI

ASI wurde von Herstellern von Sensoren und Aktoren entwickelt, um die aufwendigen, direkten Verbindungen zu den Steuerungen zu vermeiden. ASI-Komponenten besitzen einen direkten Busanschluss und folgen daher dem rechten Modell der Abb. 2.10. Die Topologie des Busses ist beliebig. Er kann mit handelsüblichen 2-Draht Stegleitungen zur Elektroinstallation oder mit einer spezifischen, ebenfalls ungeschirmten Flachbandleitung realisert werden. Ein Querschnitt von 1,5 mm^2 erlaubt die Stromversorgung von maximal 31 Komponenten mit jeweils max. 100 mA. Der Anschluss an den Bus erfolgt mittels Durchkontaktierung. Es gibt spezielle ICs zur Ankopplung von Komponenten an diesen Bus. Die Buszuteilung erfolgt durch zyklische Abfrage (Polling) durch einen Bus-Master. Die Fehlererkennung basiert

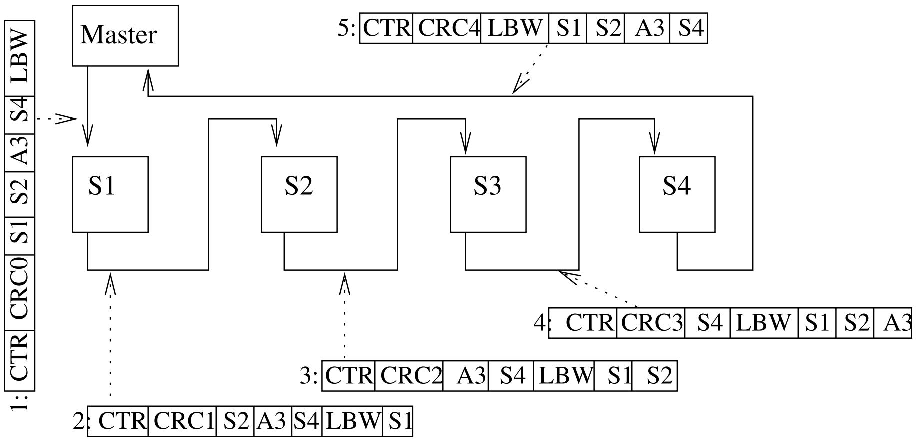


Abbildung 2.12: Schieberegisterstruktur des Interbus-S

im wesentlichen auf einer Erkennung der korrekten Impulsform.

2.4.4 Feldbusse

Feldbusse sind oberhalb der Sensor/Aktor-Ebene angesiedelt, werden aber gelegentlich diesen zugerechnet. Für Feldbusse gelten die folgenden Anforderungen [Sch97b]:

- Die Ausdehnung sollte zwischen einigen Metern und einigen Kilometern liegen können.
- Zusätzliche Busteilnehmer sollten problemlos eingebracht werden können.
- Harte Zeitanforderungen sollten eingehalten werden können.
- Aus wirtschaftlichen Gründen sind serielle Busse parallelen vorzuziehen.
- Aufgrund der Störanfälligkeit sind besondere Maßnahmen vorzusehen.

Zwei der genormten Feldbusse wollen wir hier vorstellen: Profibus und CAN.

2.4.4.1 Process Field Bus (Profibus)

Der Profibus wurde herstellerübergreifend entwickelt und ist in Deutschland eine nationale Norm (DIN E 19245). Eigenschaften des Profibus sind u.a. die folgenden:

- Die physikalische Struktur des Profibus ist ein linearer Bus, realisert mit einem Lichtwellenleiter oder einem abgeschirmten, verdrillten Kabel.
- Die Busverteilung basiert auf einem Token-Passing-Verfahren.
- Die Übertragungsgeschwindigkeit liegt je nach Bus-Länge zwischen 500 kBit/s (bei 200 m) und 9,6 kBit/s (bei 1.200 m).
- Max. eine Nachricht hoher Priorität pro möglichem Bus-Master darf immer versandt werden, Nachrichten niedrigerer Priorität nur, wenn die Token-Umlaufzeit unterhalb eines festgelegten Parameters bleibt. Auf diese Weise kann ein Echtzeitverhalten realisiert werden.
- *Zeitkritische Anwendungen sind mit dem Profibus aufgrund seines geringen Datendurchsatzes nicht möglich* [Sch97b].

2.4.4.2 Controller Area Network (CAN)

CAN wurde 1981 von Bosch und Intel mit dem Ziel der Vernetzung komplexer Controller und Steuergeräte entwickelt. Internationale Verbreitung fand CAN vor allem im Automobilbereich (Mercedes, BMW) zur Ersetzung der immer komplexer werdenden Kabelbäume (bis zu 2 km, 100 kg), aber auch im Haushaltsgeräteseктor (Bosch), in Textilmaschinen, in Apparaten der Medizintechnik und einigen anderen Anwendungen. Da der Bus im Prinzip auch als Sensor/Aktor-Bus unter Einhaltung von Echtzeitanforderungen einsetzbar ist, erschließen sich in jüngerer Zeit immer mer Anwendungsfelder wie etwa die Gebäudeautomation. Ein Vorteil von CAN liegt in den preisgünstigen Buskoppelkomponenten aufgrund von hohen Stückzahlen, nicht zuletzt im Automobilbereich. [Zitat Schürmann]

CAN besitzt die folgenden Eigenschaften:

- Übertragung über eine abgeschirmte, verdrillte Zweidrahtleitung.
- Nachrichtenkodierung im symmetrischen RS-485-Standard (symmetrische Variante von RS-232)
- Hohe Übertragungssicherheit
- Übertragungsrate von 10 kBit/s bis zu 1 MBit/s
- Nachrichten hoher und niedrigerer Priorität; max. Reaktionszeit von 134 μs für Nachrichten hoher Priorität
- Übertragung kurzer Nachrichten von max. 8 Byte Länge
- CSMA/CA als Buszugriffsverfahren
- Identifizierung von Nachrichtentypen, nicht von Empfängern

2.4.4.3 European Installation Bus (EIB)

Der European Installation Bus ist v.a. für die Gebäudeautomatisation entwickelt worden. Dieser Bus hat die folgenden Eigenschaften:

- Hierarchischer Bus mit galvanischer Trennung von Teilnetzen
- Übertragung über eine geschirmte, verdrillte, nicht abgeschlossene Vierdrahtleitung, von der nur zwei für die Übertragung von Informationen und Betriebsstrom benutzt werden
- Übertragungsrate max. 9600 Bit/s
- Busarbitrierung nach CSMA/CA
- max. 11.500 Teilnehmer, max. 700 m Abstand
- Stromversorgung mit 28 Volt, 320 mA

Leider reichen die Übertragungsraten kaum aus, um Multimediadaten zu übertragen oder um PCs untereinander zu vernetzen.

2.4.4.4 IEEE 488

Die Norm IEEE 488 beschreibt einen Standard zur Verbindung von Laborgeräten über einen Byteparallelen Bus. Diese wird heute von vielen Messgeräten unterstützt.

2.4.4.5 MAP

MAP ist ein Bus für Fertigungsstätten, der bei General Motors entwickelt wurde. Er basiert auf einem *token bus*.

2.5 Verarbeitungseinheiten

3. Vorles.

Der Menge der möglichen Komponenten zur Informationsverarbeitung ist sehr groß, entsprechend der Vielfalt eingebetteter Systeme.

Beispielhaft sollen in den folgenden Unterabschnitten einige Komponenten genannt werden. Diese Auswahl betrifft zunächst einmal nur die Hardware. Zusätzlich gibt es eine Vielfalt an Verarbeitungsalgorithmen, die teilweise in den folgenden Kapiteln vorgestellt werden. Eine Realisierung der Algorithmen benötigt meist noch weitere Basistechniken, wie z.B. Betriebsysteme, Laufzeitumgebungen für Programmiersprachen usw.

Unter den einfachen Systemen findet man noch CNC-Maschinen und speicherprogrammierbare Steuerungen (SPS) , die aber zunehmend durch PCs und Prozessoren abgelöst werden.

2.5.1 Prozessoren

Prozessoren besitzen ähnlich wie speicherprogrammierbare Steuerungen den Vorteil der Flexibilität. Ändern sich Anwendungen, so braucht lediglich ein Programm geändert und neu geladen werden, um diese Änderung zu berücksichtigen. Dabei sind Prozessoren leistungsfähiger als SPS, insbesondere wenn Operationen auf Daten auszuführen sind.

Gleichzeitig sind Prozessoren Standardkomponenten, die preisgünstig rasch verfügbar sind und für die auch Entwicklungswerkzeuge existieren.

Diese Kombination von Flexibilität und Benutzung einer Standardkomponente ist bei vielen anderen Hardware-Komponenten nicht vorhanden. Dies führt zu einer zunehmenden Beliebtheit von Prozessoren, wobei viele EIS schon immer mit Hilfe von Prozessoren realisiert wurden.

Dies wird auch durch die folgenden Zitate belegt:

> *At the chip level, embedded chips include microcontrollers and microprocessors. Microcontrollers are the true workhorses of the embedded family. They are the original 'embedded chips' and include those first employed as controllers in elevators and thermostates.* [Zitat Ryan [Rya95]]

> *... the New York Times has estimated that the average American comes into contact with about 60 microprocessors every day ...* [Zitat Camposano und Wolf [CW96]]

Nach Informationen der Zeitung *EEDesign* wird der Markt für eingebettete Mikrocontroller im Zeitraum von 1995 bis zum Jahr 2000 von 9,9 auf 19,6 Milliarden Dollar wachsen. Dazu kommt der Markt für übrige eingebettete Mikroprozessoren, der 1995 bereits ein Volumen von 3,9 Milliarden Dollar hatte.

Nach eigenen Informationen enthalten neueste hochwertige BMWs über 100 Prozessoren.

2.5.1.1 Mikrocontroller

Viele dieser über 100 Prozessoren dürften Mikrocontroller sein, die Steuerungsaufgaben übernehmen und nur verhältnismäßig geringe Anforderungen an den Datendurchsatz erfüllen können (also insbesondere für Multimediaaufgaben nicht geeignet sind).

Typische Eigenschaften solcher Controller lassen sich am Beispiel des Intel 8051[3] demonstrieren. Dieser Controller kann schlichtweg als **der** Controller bezeichnet werden. Er besitzt die folgenden Eigenschaften:

1. 8-Bit CPU, für Steuerungsaufgaben optimiert
2. Viele Befehle zur Bearbeitung von Booleschen Datentypen
3. Programmspeicher-Adressraum von 64 kBytes
4. Separater Datenspeicher-Adressraum von 64 kBytes

[3] In einer Vorlesung mit dem Namen "Prozessrechner" sollte es angemessen sein, einige konkrete für Prozesssteuerungen geeignete Rechner (Prozessoren) und deren typische Eigenschaften kennenzulernen.

5. 4 kByte Programmspeicher auf dem Chip
6. 128 Bytes Datenspeicher auf dem Chip
7. 32 bidirektionale, einzeln adressierbare E/A-Leitungen
8. Zwei 16-Bit-Zähler auf dem Chip
9. Asynchroner Receiver/Transmitter (UART) auf dem Chip
10. 5 Interrupt-Vektoren mit zwei Prioritätsebenen
11. Taktgenerator auf dem Chip
12. Existiert in vielen populären Variationen

Insbesondere die Punkte 1,2,5,6,7,8,9 und 11 zeigen, welche Eigenschaften typisch für Controller sind.

2.5.1.2 Allgemeine Prozessoren

Ursprünglich wich die Architektur von Prozessrechnern von der üblicher (Groß-) Rechner ab. Groß-Rechner verarbeiteten in der Regel Batch-Jobs und die Prozesswechselzeit war nicht sehr wichtig. Prozessrechner waren dagegen von Anfang an auf kurze Prozesswecheselzeit und außerdem auch auf effiziente, durchschaubaure Interrupt-Strukturen und den kostengünstigen Anschluss von peripheren Einheiten angewiesen. Dies hat sich verändert, viele der in PCs und Workstations eingesetzten Prozessoren haben Konzepte der frühen Prozessrechner übernommen. Insofern haben sich die Architekturen angeglichen.

Außerhalb des Bereichs der Industriesteuerungen allerdings stellen EIS schon spezielle Anforderungen an die Prozessoren. Wichtig ist u.a. die *Effizienz*. Portable Geräte müssen effizienten Gebrauch von der verfügbaren Energie machen. Sofern die gesamte Elektronik auf einem Chip integriert ist (sog. *system-on-a-chip*) , ist die Silizium-Fläche effizient einzusetzen. Bei Massenprodukten wie Controllern im Auto spielen die Kosten eine erhebliche Rolle. Daher haben sich für EIS Prozessoren entwickelt, die zwar für verschiedene Anwendungsbereiche einsetzbar (also "allgemein") sind, aber dennoch effizienter sind als übliche Prozessoren in PCs.

Ein Aspekt der Effizienz ist die **Kompaktheit des benötigten Programmcodes**. Folgende Techniken werden u.a. eingesetzt, um zu einem möglichst kompakten Programmcode zu kommen:

- Benutzung von CISC-Prozessoren

  CISC-Prozessoren besitzen kompakteren Code besitzen als RISC-Prozessoren und besitzen einen Vorteil, wenn die Kompakheit des Codes wichtig ist.

- Kompression

  Variationen von normalen Datei-Komprimierungstechniken können benutzt werden, um Code zu komprimieren. Allerdings müssen beliebige Sprünge unterstützt werden.

- Dekomprimierung vor dem Laden in den Cache

- Prozessorspezifische Komprimierung/Dekomprimierung

Eine andere Möglichkeit der Effizienzsteigerung ist *predicated execution*. *predicated execution* bedeutet, dass die Ausführung eines Maschinenbefehls von einem Prädikat abhängig ist. Diese Technik ist unter anderem Namen bereits lange bekannt ist, siehe z.B. [Mar80]. Früher nannte man solche Befehle auch *guarded commands*.

Die Prädikate beziehen sich in der Regel auf Belegungen der Condition-Code Register. Als Spezialfall ist auch die unbedingte Ausführung möglich.

Mit Hilfe von *predicated execution* können IF-Statements in einer Maschine realisiert werden, ohne bedingte Sprünge zu benutzen. Dies bietet eine Reihe von Vorteilen:

- Bedingte Sprünge passen nicht sehr gut zur Fließbandverarbeitung. Die volle Leistung eines Fließbandes (d.h. eine Auslastung von 100%) kann bei Anwesenheit von bedingten Sprüngen nur realisiert werden, wenn besondere Hardware eingebaut wird (wie z.B. *branch target buffer*, siehe [Mar97]). Dieser Zusatzaufwand bedeutet einen Effizienzverlust.
- Bedingte Sprünge schränken die Möglichkeiten ein, Befehle innerhalb von Befehlssequenzen zu plazieren. Sie beschränken damit prinzipiell mögliche Optimierungen, wie z.B. das globale Scheduling.
- VLIW-Maschinen (siehe weiter unten) sind nur dann effizient, wenn die Basisblöcke (d.h. verzweigungsfreie Codesequenzen) des betrachteten Programms groß sind. Mit *predicated execution* kann die Größe der verzweigungsfreien Sequenzen erhöht werden.

predicated execution soll auch zur Realisierung der nächsten Generation von Pentium-Prozessoren eingesetzt werden [].

Beispiele moderner Prozessoren:

1. ARM

   Prozessoren der Firma Advanced Risc Machines (ARM) zählen zu den wenigen europäischen Beiträgen zum Prozessormarkt, die weltweit Bedeutung erlangt haben. ARM-Prozessoren gelten als Prozessoren mit besonders geringer Leistungsaufnahme. Sie werden deshalb vielfach in Lizenz gefertigt und z.B. in Apple's Newton eingesetzt.

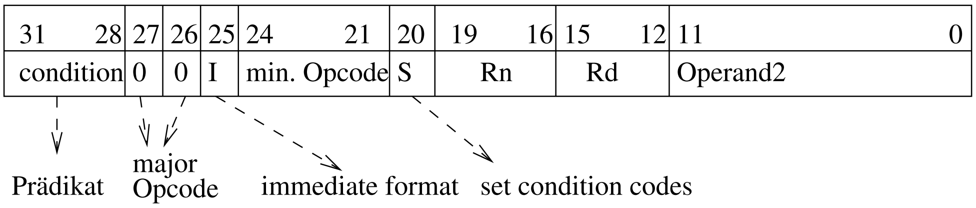
   

   Abbildung 2.13: Befehlsformat von ARM-Befehlen mit dem 'major opcode' 00

   ARM-Prozessoren besitzen im Prinzip ein 32-Bit Befehlsformat (siehe Abb. 2.13). Sie zählen zu den ersten Prozessoren, die *predicated execution* kommerziell realisieren. Hierzu kann in den Bits 31 bis 28 eine Bedingung kodiert werden.

   Neuere 32-Bit-ARM Prozessoren enthalten einen vereinfachten 16-Bit THUMB-Befehlssatz, um den Programmcode zu reduzieren. Dabei wird von der *predicated execution* abgegangen, es kann nur ein Teil der Register adressiert werden und der Wertevorrat für Konstanten ist kleiner (siehe Abb. 2.14).

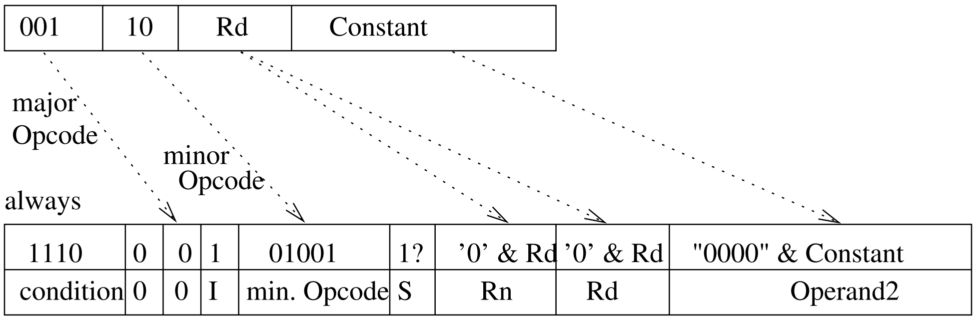
   

   Abbildung 2.14: Dynamische Dekompression vom THUMB- in den ARM-Befehlssatz

   Ein Nachteil dieses Vorgehens ist, dass für den neuen Befehlssatz modizierte Compiler, Debugger usw. benötigt werden.

2. Motorola FlexCores

   Motorola bietet Varianten der bekannten Motorola 68000-er Familie zur Integration innerhalb eines *systems-on-a-chip* an. Da es sich bei der 68000-er Familie um CISC-Prozessoren handelt und da CISC-Prozessoren einen kompakteren Code besitzen als RISC-Prozessoren, zeichnet sich diese Familie durch Effizienz hinsichtlich des Codes aus. Verschiedene Varianten einschließlich passender Zusatzkomponenten, wie SCSI- und Parallel-Interfaces, sind von Motorola erhältlich.

2.5.1.3 DSP-Prozessoren

Neben einer hohen Codedichte sind eine geringe Leistungsaufnahme und eine an die Verarbeitungsaufgabe angepasste Architektur sehr wichtig.

Eine wichtige Teilaufgabe von eingebetteten Systemen besteht in der digitalen Signalverarbeitung (DSP). Für diese Aufgaben sind spezielle, DSP-Aufgaben effizient und mit hoher Leistung verarbeitende Prozessoren entwickelt worden. DSP-Prozessoren besitzen die folgenden spezifischen Eigenschaften:

- *saturating arithmetic:*
  Bei DSP-Aufgaben treten gelegentlich Überschreitungen des darstellbaren Zahlenbereichs auf. Bei Standard-Arithmetik werden in diesem Fall die zurückgegebenen Bitvektoren nicht auf spezielle Werte gesetzt (außer bei der Gleitkomma-Arithmetik). Bei DSP-Algorithmen ist es dagegen häufig am besten, wenn ein der größten bzw. der kleinsten Zahl entsprechender Bitvektor abgeliefert wird, der dann z.B. die größte Helligkeit auf dem Bildschirm oder die größte Lautstärke bezeichnet. Auf eine solche Arithmetik kann bei den meisten DSP-Prozessoren umgeschaltet werden.
- *spezielle Adressierungsarten:*
  Aufgrund der Anwendung in üblichen DSP-Algorithmen sehen DSP-Prozessoren häufig spezielle Adressierungsarten vor. Die Modulo-Adressierung beispielsweise erlaubt die effiziente Realisierung von verzögerten Signalen mit Hilfe von Ringpuffern.
- *Eingeschränkte Parallelität:*
  Die Rechenwerke der meisten DSP-Prozessoren erlauben es, in einem Takt Zuweisungen zu mehreren Registern gleichzeitig auszuführen. Diese Prozessoren stellen diese Form der begrenzten Parallelität dann meist auch an der Befehlsschnittstelle zur Verfügung. Gängig sind z.B. *parallel moves* genannte Befehle, die gleichzeitig eine Arithmetik-Operation und einen Datentransport veranlassen.
- *Heterogene Registersätze*
  Nicht alle Register besitzten dieselbe Funktionalität.
- *multiply/accumulate-Befehle*
  Diese Befehle realisieren Folgen von Additionen und Multiplikationen und sind besonders für die Realisierung von digitalen Filtern geeignet.
- *Realtime-Fähigkeit:*
  Es ist wichtig, die maximale Laufzeit eines Programms möglichst exakt und nicht nur im Mittel angeben zu können. Deshalb wird z.B. vielfach auf Caches, die eine datenabhängige Laufzeit bewirken würden, verzichtet.

DSP-Prozessoren bieten v.a. für DSP-Applikationen mehr Rechenleistung pro Watt Leistungsverbrauch, weshalb sie v.a. in portablen Geräten vielfältig Einsatz finden.

Beispiele: Zwei der komplexesten DSP-Prozessoren sind der Philips TriMedia und der der Texas Instruments 'C60-Prozessor:

1. TMS 320C60x

   Der TMS 320C60x-Prozessor ist ein *very large instruction word* (VLIW)-Prozessor. Derartige Prozessoren besitzen ein sehr breites Befehlswort, über das pro Takt mehrere Rechenoperationen gesteuert werden können. Zu diesem Zweck enthalten VLIW-Prozessoren mehrere funktionale Einheiten (Addierer, Multiplizierer usw.). Bereits zur Übersetzungszeit muss geprüft werden, welche Rechenoperationen unter den gegebenen Hardware-Beschränkungen in einem Takt ausgeführt werden können. Auf diese Weise soll Hardware eingespart werden, die sonst zur Überprüfung auf gleichzeitige Ausführbarkeit während der Programmlaufzeit gebraucht würde.

   Ein Problem bei üblichen VLIW-Prozessoren ist, dass nicht immer so viele Operationen in einem Schritt ausgeführt werden können, wie dies aufgrund der in der Regel festen Befehlswortbreite von VLIW-Maschinen möglich wäre.

Der TMS 320C60x bietet hierfür (wohl erstmalig) eine Lösung an: Befehlsworte enthalten bis zu 8 Befehle von je 32 Bit. Die Menge der gleichzeitig auszuführenden, in einem Befehlswort kodierten Befehle heißt auch **ein Befehlspaket**. Ein Bit in jedem Befehl gibt Auskunft darüber, ob der nächste Befehl noch zu demselben Befehlspaket gehören soll oder nicht (siehe Abb. 2.15). Auf diese einfache Weise wird eine variable Befehlswortbreite realisiert.

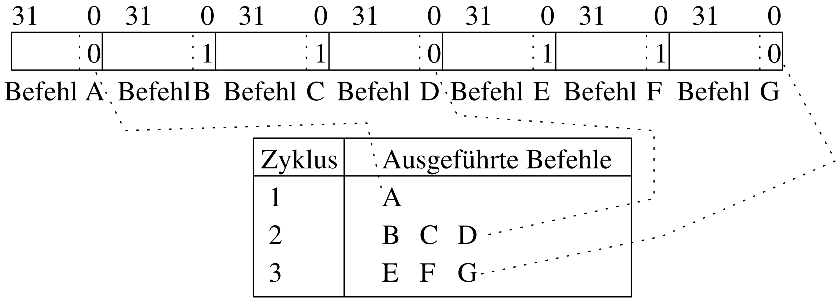


Abbildung 2.15: Parallelverarbeitung beim VLIW-Prozessor TMS 320C60x

<u>Eigenschaften des TMS320C60x:</u>

- Maximal können 8 Befehle (256 Bit) pro Takt geholt und ausgeführt werden.
- Die maximale Leistung wird mit 1,6 Milliarden Befehlen pro Sekunde angegeben.
- Es gibt 2 parallele Rechenwerke, von denen jedes folgendes enthält:
  - 1 Multiplizierer
  - 1 Addierer
  - 1 Adder/Shifter
  - 1 Adder/Shifter/Normalizer
  - 1 Registerfile
  - 1 Lese- und 1 Schreibpfad zum Speicher
  - 1 Verbindung zum anderen Rechenwerk
- Der Prozessor besitzt Standard DSP-Funktionalität: Modulo-Adressierung, *saturating arithmetic*, integrierte Ein/Ausgabe, keine Memory-Management-Einheit.
- Effizienter Code kann nur erzeugt werden, wenn Befehle über Basisblockgrenzen hinaus verschoben werden können (sog. globales Scheduling).

2. Philips Trimedia

   Der Philips Trimedia-Prozessor ist speziell zur Verarbeitung von Multimediadaten entworfen worden. Für verschiedene Daten besitzt er eigene Funktionseinheiten (siehe Abb. 2.16) [Phi].

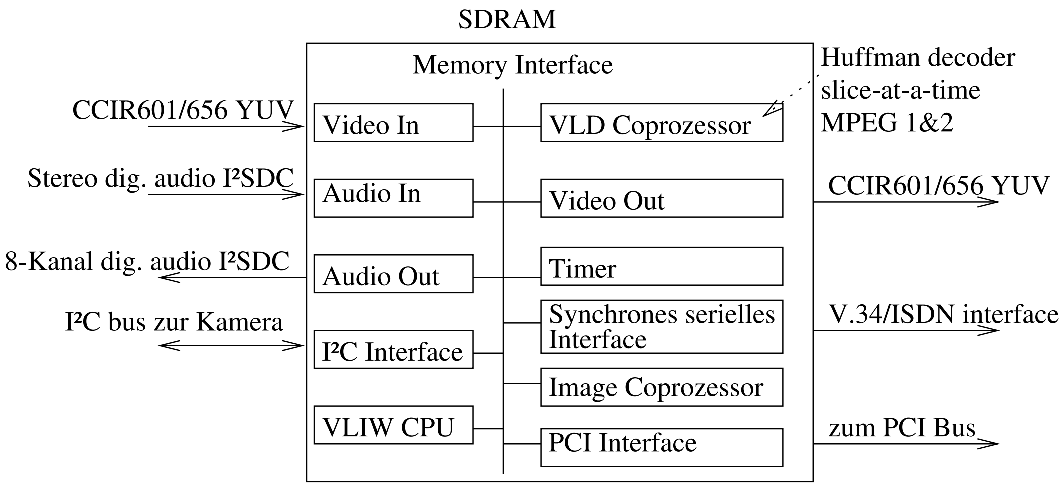


Abbildung 2.16: Blockdiagram des Philips TriMedia TM-1

<u>Eigenschaften des TriMedia TM-1:</u>

- Er bearbeitet simultan Audio-, Video-, Graphik- und Kommunikationsdaten.
- Es kann bis zu 4 Milliarden Operationen pro Sekunde ausführen.
- Er besitzt eine spezielle Einheit zur Lauflängen-Kodierung (siehe Kap. 3).
- Er enthält einen optimierten VLIW-Prozessor, der bei einem Takt von 100 MHz bis zu 5 Operationen pro Takt ausführen kann. Mittels *predicated execution* können bedingte Sprünge vermieden und damit die Anzahl parallel ausgeführter Befehle erhöht werden.
- Er erlaubt das gleichzeitige Kopieren zwischen seinem SDRAM-Speicher[4] und dem Video-Puffer des Host-Rechners (z.B. eines PCs).
- Die Unterstützung erfolgt durch die Philips TriMedia-Gruppe in Sunnyvale.

2.5.1.4 Eigenschaften eingebetteter Prozessoren

Allgemein können wir folgende Eigenschaft eingebetteter Prozessoren festhalten:

- Die Effizienz besitzt eine hohe Bedeutung; die Einfachheit der Programmierung ist nicht allein ausschlaggebend.
- Aufgrund der Vielfalt der Anwendungsbereiche und der notwendigen Effizienz für jeden Bereich gibt es eine entsprechende Vielfalt von Prozessoren.
- Viele der Prozessoren haben eine Harvard-Architektur (d.h. separate Daten- und Befehls-Speicher, -Busse und Adressübersetzungstabellen).
- Die Register haben häufig eine unterschiedliche Funktionalität. So können zur effizienten Realisierung digitaler Filter häufig nur wenige Register eingesetzt werden.
- Prozessoren besitzen häufig die Fähigkeit zur parallelen Ausführung auf Befehlssatz-Ebene. Es kann sich hierbei um einige wenige parallel ausführbare Operationen oder um echte VLIW-Maschinen handeln.
- Sie besitzen vielfach verschiedene Betriebsmodi, wie z.B. Modi für *saturating arithmetic* bzw. *wrap-around arithmetic*.
- Sie haben häufig einen spezialisierten Befehlssatz, der beispielsweise Bitfeld-Befehle, Modulo-Adressierung, *multiply/accumulate*-Befehle und Sonderbefehle zur Realisierung der schnellen Fourier-Transformation (siehe Kapitel 3) enthalten.
- Gelegentlich enthalten sie mehrere **Speicherbänke**. Die Anzahl der parallel ausführbaren Operationen kann dann erhöht werden, wenn sie sich auf unterschiedliche Speicherbänke beziehen.
- Viel "Fett" (Komponenten, welche für die eigentlichen Berechnungen nicht benötigt werden), das klassische Prozessoren inzwischen angesetzt haben, wird bei eingebetteten Prozessoren nicht vorgesehen. So fehlen vielfach Speicherverwaltungss-Bausteine (*memory management units*, MMUs), *branch target buffer*, **Speicherschutz** und zum Teil auch Caches. Dies ist natürlich auch darauf zurückzuführen, dass die Menge der Anwendungen bekannt ist und Effizienz eine große Rolle spielt.

2.5.2 Industrie-PCs (IPCs)

4. Vorles.

Für Industriesteuerungen werden PCs aufgrund der stark sinkenden Preise von PCs und dem für die meisten Entwickler gewohnten Umgang zunehmend beliebter. Allerdings sind die Besonderheiten eingebetteter Systeme zu beachten. Zunächst sind die üblichen Desktop-PCs für den Einsatz in rauher Umgebung nicht robust genug und weisen eine unzureichende Zahl von Ein/Ausgabe-Schnittstellen aus. Es werden daher spezielle robuste PCs (Industrie-PCs) gefertigt. Weiter sind die üblichen Betriebssysteme wie Windows 95 und zum Teil auch Windows NT nicht für den Realzeitbetrieb brauchbar. Sie müssen durch spezielle Systemteile ergänzt werden, was den Aufwand für die Pflege von Software erheblich erhöht (siehe auch Kapitel 4.1).

[4] Siehe [Mar97].

2.5.3 Anwendungsspezifische Schaltkreise (ASICs)

Bei sehr hohen Leistungsanforderungen müssen anwendungsspezifische Schaltkreise entwickelt werden, was allerdings auch mit sehr hohen Kosten verbunden ist.

2.5.4 Field Programmable Gate Arrays (FPGAs)

Großer Beliebtheit erfreuen sich in jüngster Zeit Variationen von Gate Arrays, welche noch im Gehäuse programmierbar sind, die sog. *field programmable gate arrays* (FPGAs).

FPGAs bestehen aus konfigurierbaren logischen Blöcken (*configurable logic blocks, CLBs*), die über programmierbare Leitungen miteinander verbunden werden können. Hierbei gibt es verschiedene Alternativen:

- Multiplexer-basierte CLBs (siehe Abb. 2.17).

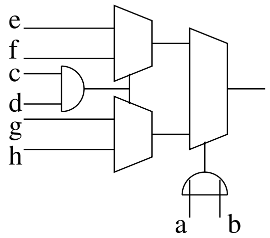


Abbildung 2.17: Multiplexer-basierter CLB der Fa. ACTEL

  Die verschiedenen Eingänge können mit festen Signalwerten konfiguriert werden.

- SRAM-basiert

  Ein Beispiel dafür sind die CLBs der Fa. Xilinx, in Abb. 2.18 gezeigt anhand der CLB-Struktur der FPGA-Serie XC4000.

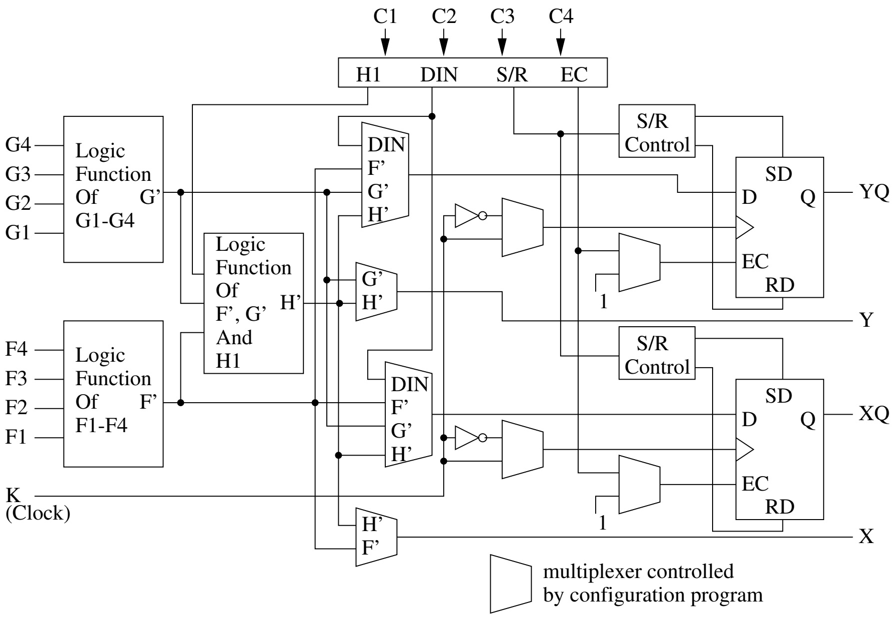


Abbildung 2.18: Vereinfachtes Blockdiagramm des XC4000 CLB

Diese CLBs enthalten jeweils zwei Speicher von 16 Bit, welche alle Booleschen Funktionen von 4 Variablen realisieren können. Weiterhin gibt es noch einen dritten Speicherblock, mit Hilfe dessen auch einige Funktionen von 5 Variablen direkt realisiert werden können. Diese Speicher werden beim Start des FPGAs oder auch später geladen.

- EPROM-basierte CLBs.

Die Verbindungen der Bausteine untereinander können über zwei Methoden realisiert werden:

- Mittels *anti fuses*:

  In diesem Fall wird die Isolierung zwischen zwei eng benachbarten Leitungen durch einen kurzen Überspannungsimpuls geschmolzen.

  Diese Vorgang ist nicht reversibel; er führt aber dafür zu einem schnellen Pfad.

- Mittels RAM-gesteuerter MOS-Schalter:

  Dieser Vorgang ist reversibel (und daher an Universitäten aus Kostengründen beliebt). Er führt allerdings zu flüchtigen und nicht sehr schnellen Verbindungen.

In der Serie XC4000 sind FPGAs mit 64 bis 1.024 CLBs erhältlich. Eine typische Verschaltung zur Konfiguration mit Daten aus einem EPROM zeigt die Abb. 2.19.

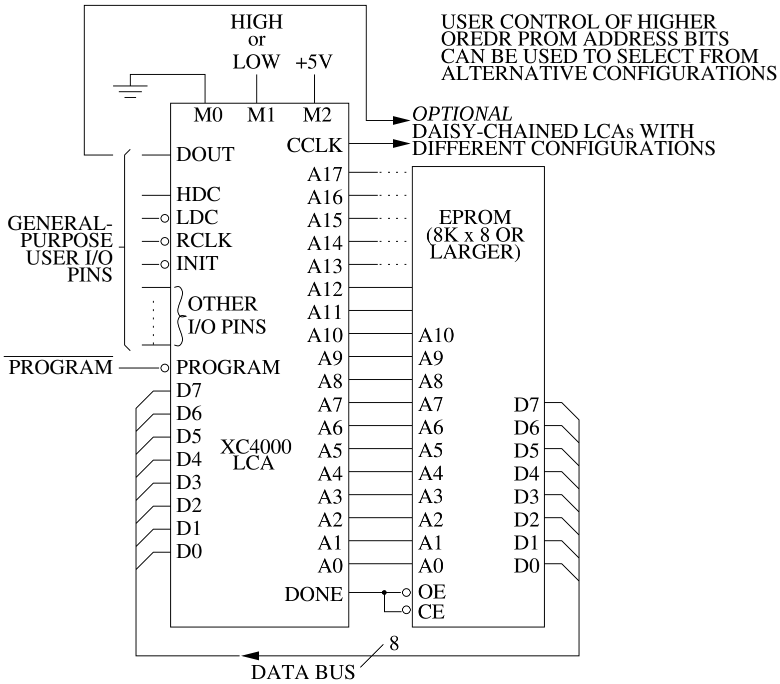


Abbildung 2.19: Laden von Konfigurationsdaten aus einem EPROM

Neben der Realisierung endgültiger Schaltungen mit Hilfe von FPGAs werden diese vielfach auch während des *rapid prototypings* zur Erprobung der Funktion von EIS in realer Umgebung eingesetzt (siehe Kapitel 6).

2.5.5 PALs und PLDs

PALs sind programmierbare Bausteine, bei denen im Unterschied zu PROMs und PLAs nur die UND-Ebene programmierbar ist (siehe Abb. 2.20 und 2.21).

Abb. 2.22 zeigt, wie eine Schaltfunktion mittels eines PLAs realisiert werden kann.

2.6 Ausgabeeinheiten

Ebenso vielfältig wie die Menge der möglichen Verarbeitungseinheiten ist auch die Menge der möglichen Ausgabeeinheiten.

Beispielhaft seien hier genannt:

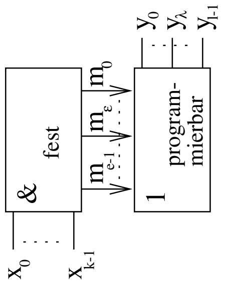


Abbildung 2.20: PROM (feste UND-Struktur)

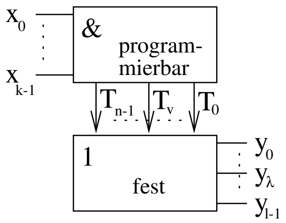


Abbildung 2.21: PAL (programmierbare UND-Struktur, feste ODER-Struktur)

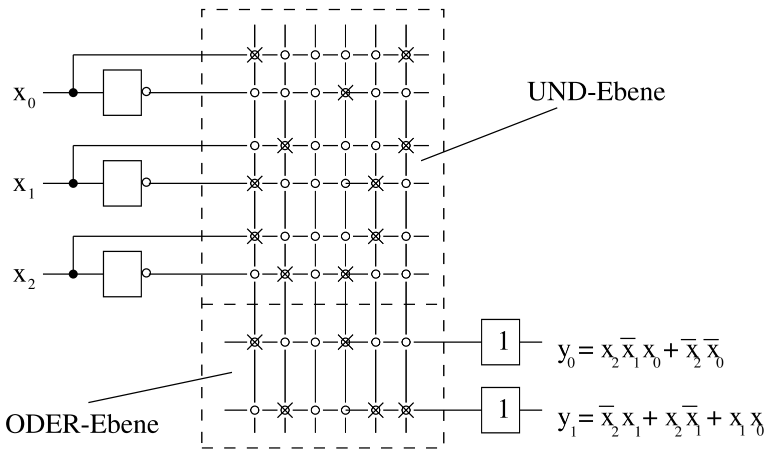


Abbildung 2.22: Beispiel einer Realisierung einer Schaltfunktion mittels eines PLAs

- Aktoren, z.B. mikromechanischer Motor (siehe Abb. 2.23.
- Displays
- Digital/Analog-Wandler

  In der Schaltung der Abb. 2.24 sorgt ein (hier als ideal vorausgesetzter) Operationsverstärker dafür, dass die Spannung zwischen den beiden Eingängen 0 wird.

  Für die Ausgangsspannung gilt:

$$U = R_1 * I$$

wobei I der Strom durch den Widerstand der Größe R_1 ist. Da der (ideale) Operationsverstärker keinen Eingangsstrom benötigt, ist I gleich dem von links über das Widerstandsnetzwerk fließenden Strom. Für diesen ergibt sich

$$|I| = U_{ref}/R * \sum_{i=0}^{3} x_i * 2^{i-3}$$

.

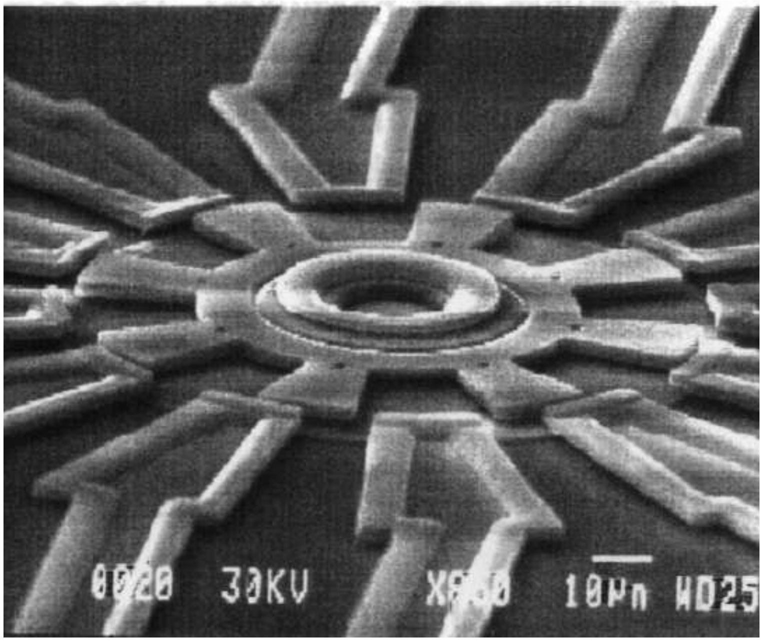


Abbildung 2.23: Mikromechanischer Motor (©Microelectronic Center of North Carolina)

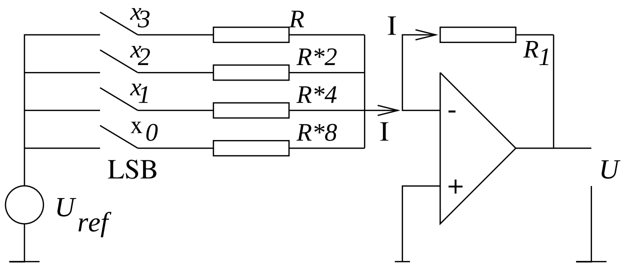


Abbildung 2.24: Digital/Analog-Wandler

Damit ist

$$|U| = U_{ref} * R_1/R * \sum_{i=0}^{3} x_i * 2^{i-3} = U_{ref} * R_1/R * nat(x)/8$$

wobei *nat* die Wandlung eines Bitvektors x in eine natürliche Zahl bedeutet. Die Ausgangsspannung ist also proportional zu dem Wert, der in x kodiert wird.

Ein solcher D/A-Wandler ist sehr schnell. Seine Genauigkeit richtet sich im wesentlichen nach der Genauigkeit, mit der man die Widerstände fertigen kann. Eine gute Genauigkeit erreicht man z.B. mit Justierungen mittels Laser.

Kapitel 3

Digitale Signalverarbeitung

Die digitale Signalverarbeitung (engl. *digital signal processing*, DSP) ist eine der Grundtechniken zur Realisierung von EIS[1]. Als **Signal** können wir dabei eine zeitvariable Ein- oder Ausgangsgröße eines EIS verstehen. Mathematisch kann ein Signal durch eine Funktion beschrieben werden, welche die Zeit als ein Argument besitzt. Wir setzen in dieser Vorlesung voraus, daß nur zeitdiskrete Signale zu verarbeiten sind.

3.1 Zeitdiskrete Signale

Ein zeitdiskretes Signal kann durch eine Folge

$$x = (..., x[-2], x[-1], x[0], x[1], ...)$$

beschrieben werden, wobei x für die Folge insgesamt stehen soll. Ein konkretes Element der Folge bezeichen wir als

$$x[n]$$

wobei wir n als Zeitparameter interpretieren können. Die Folge x repräsentiert eine Funktion der (diskreten) Zeit.

Für Folgen ist auf naheliegende Weise die Summe zweier Folgen durch die komponentenweise Summation definiert:

Def.: Seien x und y Folgen. Die Folge $z = x + y$ ist definiert durch: $\forall n : z[n] = x[n] + y[n]$

Die Multiplikation einer Folge mit einem konstanten Faktor α ist durch die komponentenweise Multiplikation definiert:

Def.: Sei x eine Folge und $\alpha \in \mathbb{R}$ Die Folge $z = \alpha \cdot x$ ist definiert durch: $\forall n : z[n] = \alpha \cdot x[n]$

Def.: Eine Folge y heißt gegenüber einer Folge x um $n_0 \in \mathbb{Z}$ **verzögert**, falls gilt: $\forall n \in \mathbb{Z} : y[n] = x[n - n_0] =$.

Def.: Die **Einheitsabtastfolge** bzw. der **Einheitsimpuls** δ ist definiert durch deren Elemente

$$\delta[n] = \begin{cases} 1, & \text{für } n = 0 \\ 0, & \text{sonst} \end{cases} \qquad (3.1)$$

[1]In diesem Kapitel des Skripts orientieren wir uns vorwiegend an dem Buch von Oppenheim und Schafer [OS95].

Mit Hilfe des Einheitsimpulses kann eine beliebige Folge als eine gewichtete Summe von verzögerten Einheitsimpulsen dargestellt werden. Die Folge p, deren Elemente durch

$$p[n] = a_{-2}\delta[n+2] + a_0\delta[n] + a_3\delta[n-3]$$

gebildet werden, zeigt beispielsweise die Abb. 3.1.

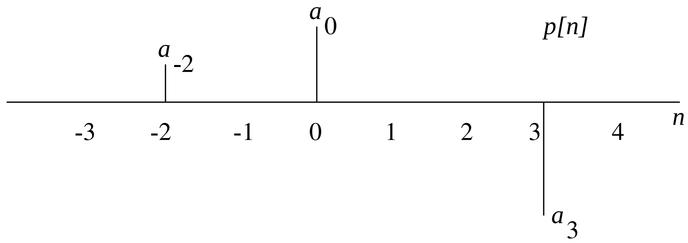


Abbildung 3.1: Gewichtete Summe von Einheitsimpulsen

Generell können Folgenelemente mittels der nachstehenden Gleichung dargestellt werden:

(3.2) $$x[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot \delta[n-k]$$

Def.: Der **Einheitssprung** ist eine Folge mit den Elementen

(3.3) $$u[n] = \begin{cases} 1, & \text{für } n \geq 0 \\ 0, & \text{sonst} \end{cases}$$

Def.: Eine Folge mit den Folgenelementen $x[n] = A\cos(\omega_0 n + \phi)$ heißt **sinusförmige Folge**[2] wobei A reell ist.

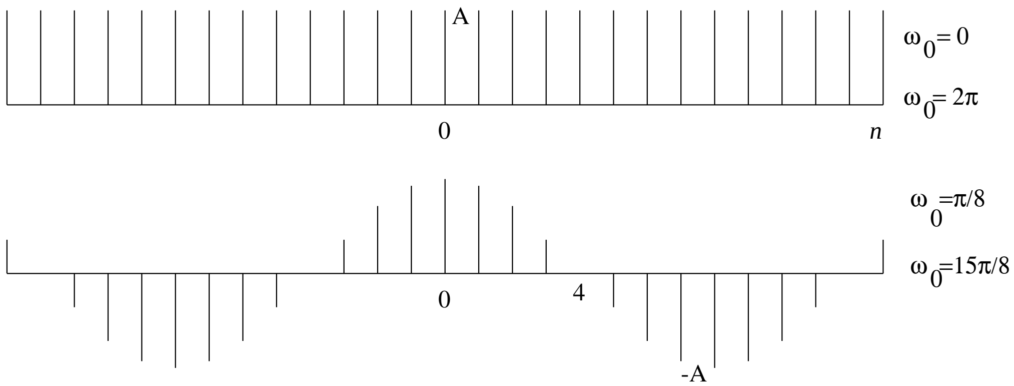


Abbildung 3.2: Sinusförmige Folge, niedrige Frequenzen

Zu beachten ist, daß sich die sinusförmigen Folgen (und auch die Exponentialfolgen, s.u.) für $\omega = (\omega_0 + 2\pi r)$ mit $r \in \mathbb{Z}$ nicht von denen für die Frequenz ω_0 unterscheiden (siehe auch Abb. 3.2 und 3.3). Wir können uns daher darauf beschränken, ω_0 in einem Intervall der Länge 2π zu betrachten, also etwa die Intervalle $(-\pi, \pi]$ oder $[0, 2\pi)$.

Def.: Eine Folge mit den Folgenelementen $x[n] = A\alpha^n$ heißt **Exponentialfolge**.

[2] Da wir stets eine Phase ϕ zulassen, ist es nicht notwendig, von einer cosinusförmigen Folge zu sprechen.

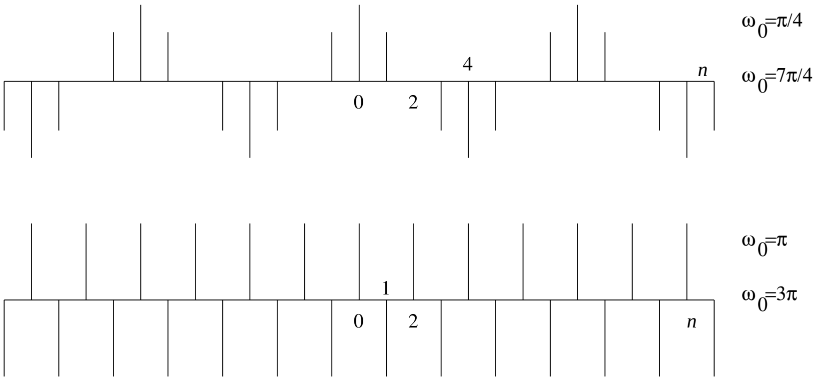


Abbildung 3.3: Sinusförmige Folge, hohe Frequenzen

Sowohl für A als auch für α erlaubt man in der Regel neben reellen auch komplexe Werte. Für $A = |A|e^{j\phi}$ und $\alpha = |\alpha|e^{j\omega_0}$ gilt:

$$\begin{aligned} x[n] = A\alpha^n &= |A|e^{j\phi}|\alpha|^n e^{j\omega_0 n} \\ &= |A||\alpha|^n e^{(j\omega_0 n+\phi)} \\ &= |A||\alpha|^n \cos(\omega_0 n + \phi) + j|A||\alpha|^n \sin(\omega_0 n + \phi) \end{aligned} \tag{3.4}$$

Für $|\alpha| = 1$ folgt daraus:

$$x[n] \quad = \quad |A|e^{(j\omega_0 n+\phi)} = |A|(\cos(\omega_0 n + \phi) + j\sin(\omega_0 n + \phi)) \tag{3.5}$$

3.2 Zeitdiskrete Systeme

5. Vorles.

Wir sind v.a. daran interessiert, unsere Signale zu verarbeiten, d.h. die Folgen einer Transformation T zu unterwerfen (siehe Abb. 3.4).

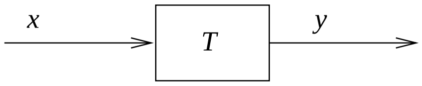


Abbildung 3.4: Transformation eines Signals x durch ein System T

Wir können dies beschreiben durch die Gleichung

$$y \quad = \quad T(x) \tag{3.6}$$

T ist eine Transformation, welche die Funktion x auf die Funktion y abbildet. Wollen wir konkret den Wert zum Zeitpunkt n erhalten, so schreiben wir T_n oder auch $T(x[n])$. Wir müssen uns über die Bedeutung der dieser Notation klar sein: $T(x[n])$ ist das Folgenelement $y[n]$, welches durch die Transformation T aus der Folge x entsteht.

Wir nennen T auch die Transformation des von uns zu realisierenden informationsverarbeitenden Systems. Gelegentlich werden wir im folgenden nicht zwischen dem System und der durch das System bewirkten Transformation unterscheiden. Nachfolgend wollen wir für T einige spezielle Fälle betrachten.

Beispiel 3.1: T heißt **ideale Zeitverzögerung** mit Verzögerung $n_d \in \mathbb{Z}$, falls für $y = T(x)$ gilt:

$$\forall n \in \mathbb{Z} : y[n] \quad = \quad x[n - n_d] \tag{3.7}$$

Beispiel 3.2: Für $y = T(x)$ heißt T heißt Bildung des **Kurzzeitmittelwerts** (engl. *moving average*), falls für alle $n \in \mathbb{Z}$ gilt:

$$y[n] \quad = \quad \frac{1}{M_1 + M_2 + 1} \sum_{k=-M_1}^{M_2} x[n-k] \tag{3.8}$$

In diesem Fall berechnet T einen Mittelwert des Signals x im Intervall $[-M_1, M_2]$.

Beispiel:

Abb. 3.5 zeigt ein Zeitfenster von für den Fall $M_1 = 0, M_2 = 5$. Der Folgenwert $y[n]$ ergibt sich als Summe der letzten 6 Werte von x, geteilt durch 6.

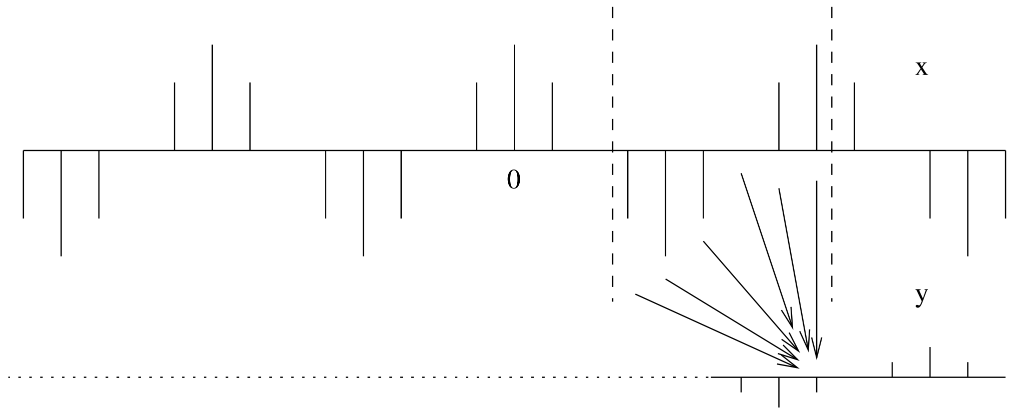


Abbildung 3.5: Bildung des Kurzzeitmittelwertes

Def.: Für $y = T(x)$ heißt eine Transformation T heißt **gedächtnislos**, falls jedes Element $y[n]$ nur eine Funktion von $x[n]$ ist.

Beispiel 3.3: $y = \log(x[n])$ ist gedächtnislos.

Def.: Sei $y_1 = T(x_1)$, $y_2 = T(x_2)$ und $\alpha \in \mathbb{R}$. T heißt linear, falls die folgenden beiden Gleichungen gelten:

$$T(x_1 + x_2) \quad = \quad T(x_1) + T(x_2) \tag{3.9}$$
$$T(\alpha \cdot x_1) \quad = \quad \alpha \cdot T(x_1) \tag{3.10}$$

Beispiel 3.4: T heißt **Akkumulator**, falls für $y = T(x)$ gilt:

$$\forall n \in \mathbb{Z} : y[n] \quad = \quad \sum_{k=-\infty}^{n} x[k]. \tag{3.11}$$

Def.: Sei $y = T(x)$. T heißt **zeitinvariant**, falls für alle $n, n_0 \in \mathbb{Z}$ gilt:

$$y[n - n_0] = T(x[n - n_0]) \tag{3.12}$$

Die Beispiele 3.1 bis 3.3 beschreiben zeitinvariante Transformationen.

Def.: Sei $y = T(x)$. T heißt **kausal**, falls $y[n]$ nur von den Folgenwerten $x[k]$, mit $k \leq n$ abhängig ist, also gilt:

$$\forall n : y[n] = T((..x[n-1], x[n])) \tag{3.13}$$

Die Bildung des Kurzzeitmittelwertes ist für $M_1 \leq 0$ kausal, für $M_1 > 0$ nicht kausal.

Beispiel 3.5: T heißt **Rückwärts-Differenzen-System**, falls für $y = T(x)$ gilt:

$$\forall n \in \mathbb{Z} : y[n] = x[n] - x[n-1] \tag{3.14}$$

Das Rückwärts-Differenzen-System ist kausal.

Def.: Eine Folge x heißt **beschränkt**, falls es einen festen, positiven, endlichen Wert B_x gibt, für den gilt:

$$\forall n \in \mathbb{Z} : |x[n]| \leq B_x < \infty \tag{3.15}$$

Def.: Ein System T heißt **stabil** (hat die BIBO=*bounded input, bounded output*-Eigenschaft), wenn für jede beschränkte Eingangsfolge x ein fester, positiver, endlicher Wert B_y existiert, für den gilt:

$$\forall n : |y[n]| \leq B_y \tag{3.16}$$

Der Akkumulator aus Beispiel 3.4 ist nicht stabil. Um dies zu zeigen, können wir den Einheitssprung benutzen. Für diesen ergibt sich:

$$\begin{aligned} y[n] &= \sum_{k=-\infty}^{n} u[k] \\ &= (n+1) \text{ für } n \geq 0 \end{aligned}$$

Diese Folge ist unbeschränkt, obwohl die Eingangsfolge beschränkt ist.

3.3 Lineare zeitinvariante Systeme

Wir wollen uns im folgenden mit einer wichtigen Klasse von Systemen beschäftigen, den linearen, zeitinvarianten Systemen (engl. *linear, time-invariant (LTI) system*).

Zunächst folgt aus $y = T(x)$ ganz generell wegen

$$x[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot \delta[n-k] \tag{3.17}$$

daß gilt:

$$y[n] = T\Big(\sum_{k=-\infty}^{\infty} x[k] \cdot \delta[n-k]\Big) \tag{3.18}$$

Wegen der nun vorausgesetzten Linearität von T folgt:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot T(\delta[n-k]) \tag{3.19}$$

$T(\delta[n-k])$ ist die Antwort des Systems zur Zeit n auf einen Impuls zum Zeitpunkt k. Wir bezeichnen diese Antwort als $h_k[n]$. Damit folgt:

(3.20)
$$y[n] \quad = \quad \sum_{k=-\infty}^{\infty} x[k] \cdot h_k[n]$$

Wegen der Zeitinvarianz gilt: Wenn $h[n]$ die Antwort zur Zeit n auf einen Impuls zur Zeit 0 ist, dann ist $h[n-k]$ die Antwort zur Zeit n auf einen Impuls zur Zeit k, also $h_k[n]$. Damit folgt:

(3.21)
$$y[n] \quad = \quad \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k]$$

Diese Formel liefert eine Vorschrift zur Berechnung der Folge y aus den Folgen x und h. Wir nennen diese Vorschrift **Faltungsoperation** und schreiben dafür auch:

(3.22)
$$y \quad = \quad x \star h$$

Ersetzt man in Gleichung 3.21 $n-k$ durch m, so folgt:

(3.23)
$$y[n] \quad = \quad \sum_{m=-\infty}^{\infty} x[n-m] \cdot h[m]$$

(3.24)
$$= \quad \sum_{m=-\infty}^{\infty} h[m] \cdot x[n-m]$$

(3.25)
$$y \quad = \quad h \star x$$

Die Faltungsoperation $\star$ ist also kommutativ.

Aus Gleichung 3.21 können wir schließen:

Die Impulsantwort h beschreibt das Verhalten eines LTI-Systems vollständig. Bei Kenntnis der Impulsantwort kann für jedes Eingabesignal x das Ausgabesignal y ausgerechnet werden.

Zur anschaulichen Interpretation der Faltung gehen wir von Gleichung 3.23 aus. Danach berechnet sich ein "aktueller" Folgenwert $y[n]$ des Signals y durch die Summation:

- des aktuellen Werts von x, d.h. $x[n]$, gewichtet mit $h[0]$,
- des vorhergehenden Werts von x, d.h. $x[n-1]$, gewichtet mit $h[1]$,
- des davor liegenden Werts von x, d.h. $x[n-2]$, gewichtet mit $h[2]$,
- der noch weiter zurückliegenden Vergangenheit von x. Hierbei ist zwischen zwei Fällen zu unterscheiden:
    1. Es existiert ein m', so daß für alle $m > m'$ gilt: $h(m) = 0$. In diesem Fall spricht man von einem *finite impulse response (FIR)*-System.
    2. Das unter 1 beschriebene m' existiert nicht. Dann spricht man von einem *infinite impulse response (IIR)*-System.
- von zukünftigen Werten $x[n+1], x[n+2]$ usw. dann, wenn T nicht kausal ist, d.h. es existiert ein $m' < 0$ mit $h[m'] \neq 0$.

Beispielhaft geben wir hier die Impulsantworten von drei der bislang betrachteten Systeme an:

- Das Kurzzeitmittelwert-System für $M_1 = 0$:

  Für dieses ergibt sich nach einfacher Rechnung:

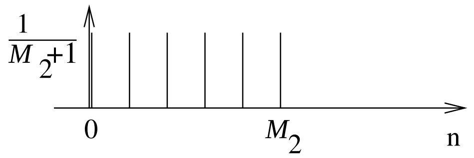


Abbildung 3.6: Impulsantwort des Kurzzeitmittelwert-System für $M_1 = 0$

$$h[n] = \begin{cases} \frac{1}{M_2+1}, & \text{für } 0 \geq n \geq M_2 \\ 0, & \text{sonst} \end{cases}$$

- Das Verzögerungssystem:

  Für dieses ergibt sich nach einfacher Rechnung:

$$h[n] = \begin{cases} 1 & \text{für } n = n_d \\ 0, & \text{sonst} \end{cases}$$

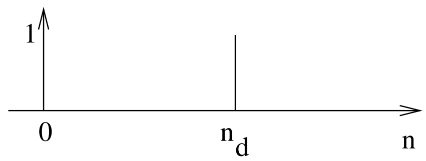


Abbildung 3.7: Impulsantwort des Verzögerungssystems

- Der Akkumulator:

$$h[n] = \begin{cases} 1 & \text{für } n \geq 0 \\ 0, & \text{sonst} \end{cases}$$

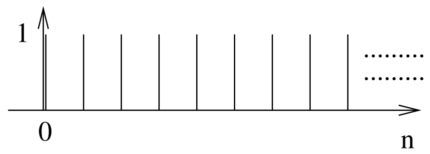


Abbildung 3.8: Impulsantwort des Akkumulators

3.4 Eigenschaften linearer zeitinvarianter Systeme

Aus Gleichung 3.21 folgt, daß auch die Faltung $\star$ auch distributiv ist:

$$x \star (h_1 + h_2) \quad = \quad x \star h_1 + x \star h_2 \tag{3.26}$$

Bei einer Parallelschaltung von LTI-Systemen haben beide Systeme dasselbe Eingangsignal und die einzelnen Ausgangssignale werden addiert. Daraus ergibt sich die Impulsantwort für die Parallelschaltung:

$$h \quad = \quad h_1 + h_2 \tag{3.27}$$

Viele LTI-Systeme erfüllen lineare Differenzengleichungen der Form

$$\sum_{k=0}^{N} a_k \cdot y[n-k] = \sum_{k=0}^{M} b_k \cdot x[n-k] \tag{3.28}$$

Beispielsweise gilt für den Akkumulator:

$$y[n] = y[n-1] + x[n] \tag{3.29}$$

$$y[n] - y[n-1] = x[n] \tag{3.30}$$

Gemäß Gleichung 3.30 ist eine rekursive Realisierung des Systems möglich (siehe Abb. 3.9).

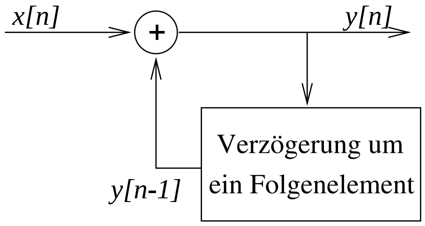


Abbildung 3.9: Rekursive Darstellung des Akkumulators

Diagramme wie das in Abb. 3.9 nennt man **Signalflußgraphen** (engl. *signal flow graphs (SFG)*. Signalflußgraphen und die sie definierenden Gleichunge können gut als Ausgangsbasis zur Realisierung eines Systems T dienen. Man kann beispielsweise die Gleichung 3.30 direkt mittels Anweisungen in C realisieren:

```
for (n=0;;n++) {
y[n] = y[n-1] + x[n] } % mit y[-1]=0
```

Dabei werden, wenn `y[n]` sofort nach Berechnung ausgegeben wird, stets nur zwei Elemente von `y` gleichzeitig benötigt. Hierfür kann gut die Modulo-Adressierung eingesetzt werden:

```
for (n=0;;n++) {
y[n%2] = y[(n-1)%2] + x_in() } % mit y[0] = 0
```

wobei `x_in` stets den letzten Eingabewert darstellen soll.

Auch der Kurzzeitmittelwert kann (wie viele andere Systeme) rekursiv berechnet werden:

$$y[n] - y[n-1] = \frac{1}{M_2+1}(x[n] - x[n-M_2-1]) \tag{3.31}$$

3.5 Darstellung zeitdiskreter Signale und Systeme im Frequenzbereich

Es ist besonders bemerkenswert, daß sinusförmige Signale am Eingang von LTI-System wiederum sinusförmige Signale an deren Ausgang erzeugen.

Die Definition von LTI-Systemen erlaubt es, neben Sinusfolgen mit reellen Werten auch komplexe Exponentialfuktionen als deren Verallgemeinerung komplexen Werten zu betrachten. Dies ist für viele Anwendungen vorteilhaft. Insbesondere läßt es sich mit komplexen Exponentialfolgen aufgrund der Rechenregeln für die Exponentialfunktion häufig leichter rechnen als mit Sinusfunktionen. Benötigte Lösungen im Reellen kann man einfach durch Bildung des Realteils bestimmen. Vielfach bilden auch Real- und Imaginärteil für sich eine Lösung, sodaß man mit einer Rechnung gleich die gesamte Lösung erhält.

Wir betrachten daher als Eingangsfolge eines Systems die Folge

$$x[n] = e^{j\omega n} \tag{3.32}$$

Nach Gleichung 3.24 bestimmt sich daraus das Ausgangssignal eines LTI-Systems zu

$$y[n] = \sum_{k=-\infty}^{\infty} h[k] \cdot e^{j\omega(n-k)} \tag{3.33}$$

$$= e^{j\omega n} \cdot (\sum_{k=-\infty}^{\infty} h[k] \cdot e^{-j\omega k}) \tag{3.34}$$

Wir nennen

$$H(j\omega) = \sum_{k=-\infty}^{\infty} h[k] \cdot e^{-j\omega k} \tag{3.35}$$

den Frequenzgang des Systems T. Mit $H(j\omega)$ gilt:

$$y[n] = H(j\omega) \cdot e^{j\omega n} \tag{3.36}$$

Allgemein nennt man im Falle einer linearen Transformation G, für welche die Gleichung

$$G(f(t)) = E \cdot f(t) \tag{3.37}$$

gilt, den Faktor E einen **Eigenwert** und $f(t)$ eine **Eigenfunktion**. Also ist $e^{j\omega n}$ eine Eigenfunktion von LTI-Systemen mit dem dazugehörigen Eigenwert $H(j\omega)$. Hierin liegt die Bedeutung der komplexen Exponentialfolgen für LTI-Systeme.

Als Beispiel betrachten wir den Frequenzgang des Verzögerungssystems. Es gilt:

$$y[n] = x[n - n_d] \tag{3.38}$$

$$y[n] = e^{j\omega(n-n_d)} = e^{-j\omega n_d} \cdot e^{j\omega n} \tag{3.39}$$

Durch Vergleich mit Gleichung 3.36 folgt:

$$H(j\omega) = e^{-j\omega n_d} \tag{3.40}$$

Viele Funktionen kann man mittels Summen sinusförmiger Signale approximieren. Ein Beispiel einer solchen Approximation einer Funktion durch ihre Frequenzanteile zeigt die Abbildung 3.10[3]. In diesem Fall wird ein Rechtecksignal durch Überlagerung von Sinussignalen immer höherer Frequenz immer genauer approximiert.

Eine solche Approximation erfordert im allgemeinen Fall sowohl die Verwendung von Sinus- und Cosinussignalen, deren jeweilige Anteile separat berechnen werden können. Eine Vereinfachung der Berechnung gelingt, wenn man Sinus- und Cosinusfunktion mittels des Eulerschen Satzes in Form einer komplexen Exponentialfunktion darstellt. Eine solche Darstellung zeigt die Gleichung 3.41.

$$x[n] = \sum_{k} \alpha_k \cdot e^{j\omega_k n} \tag{3.41}$$

Wegen der Linearität von LTI-Systemen kann man das Signal am Ausgang von LTI-Systemen beschreiben kann als Summe der Reaktion des Systems auf einzelne Exponentialfunktionen:

$$y[n] = \sum_{k} \alpha_k H(j\omega_k) \cdot e^{j\omega_k n} \tag{3.42}$$

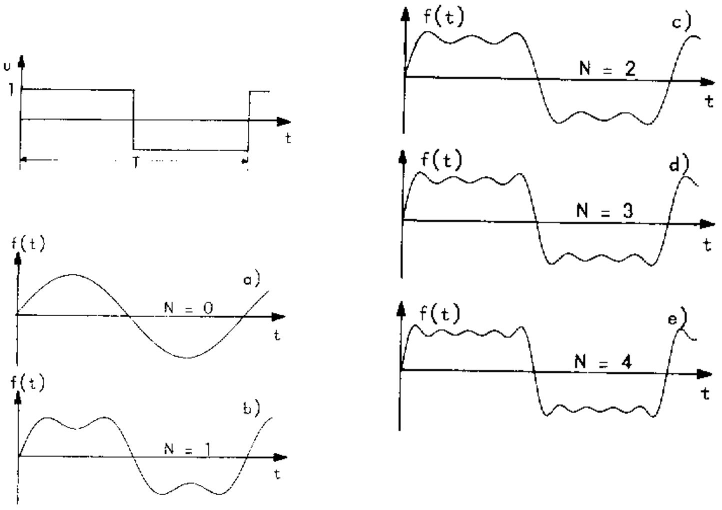


Abbildung 3.10: Approximation eines Rechtecksignals durch sinusförmige Signale

3.6 Fourier-Transformation von Folgen

6. Vorles.

Als nächstes wenden wir uns der Frage zu, wie wir die Koeffizienten der Gleichung 3.41 für ein Signal erhalten können.

Gleichung 3.41 stellt eine endliche Summe von periodischen Funktionen dar und muß daher als Ergebnis wiederum eine periodische Funktion liefern. Für nicht-periodische Funktionen muß diese Summe verallgemeinert werden. Dies gelingt mit dem Übergang auf immer mehr Frequenzen und infinitesimal kleine Abstände zwischen den Frequenzen. Wir kommen so von der Summenformel nach Gleichung 3.41 zu der Integralformel in Gleichung 3.43.

$$x[n] \quad = \quad \mathcal{F}^{-\infty}\{X\} = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(j\omega)\cdot e^{j\omega n}d\omega \tag{3.43}$$

Wegen der Periodizität im Frequenzbereich wird hier wieder nur Intervall der Länge 2π betrachtet. Die Größe $X(j\omega)$, d.h. die einzelnen Frequenzanteile, kann mann nach der Gleichung 3.44 bestimmen.

$$X(j\omega) \quad = \quad \mathcal{F}\{x\} = \sum_{n=-\infty}^{\infty} x[n]\cdot e^{-j\omega n} \tag{3.44}$$

Gleichung 3.44 beschreibt die (diskrete) **Fourier-Transformation**, welche eine Funktion x der Zeit auf eine Funktion X der Frequenz abbildet. Gleichung 3.43 ist die **Fourier-Rücktransformation** oder inverse Transformation, welche aus X wiederum x erzeugt (Beweis siehe unten).

Mit Gleichung 3.43 wird $x[n]$ wird als Überlagerung vieler Funktionen der Form

$$\frac{1}{2\pi}X(j\omega)\cdot e^{j\omega n}$$

beschrieben.

Durch Vergleich der Gleichungen 3.35 und 3.44 kann man erkennen, daß die Übertragungsfunktion eines LTI-Systems die Fourier-Transformierte der Impulsantwort ist. Umgekehrt kann man die Impulsantwort mittels Fourier-Rücktransformation aus der Übertragungsfunktion erhalten:

$$h[n] \quad = \quad \frac{1}{2\pi}\int_{-\pi}^{\pi} H(j\omega)\cdot e^{j\omega n}d\omega \tag{3.45}$$

[3] Das Original dieser Abbildung hat schon fast nostalgischen Wert: es wurde vom Autor dieses Begleittexts während der Tätigkeit als studentische Hilfskraft mit Hilfe eines Lochkartenprogramms und eines Lochsteifen-gesteuerten Zuse-Plotters erstellt.

Die Gleichungen 3.43 und 3.44 bilden ein Paar inverser Transformationen. Um dies zu zeigen, gehen wird von der Folge x aus, transformieren sie mittels 3.44 in den Frequenzbereich und mittels Gleichung 3.43 wieder zurück. Wir erhalten auf diese Weise:

$$\frac{1}{2\pi}\int_{-\pi}^{\pi}(\sum_{m=-\infty}^{\infty} x[m]\cdot e^{-j\omega m})e^{j\omega n)}d\omega \tag{3.46}$$

Wenn in 3.46 die Summe gleichmäßig konvergiert, können wir Summation und Integration vertauschen und erhalten:

$$\sum_{m=-\infty}^{\infty} x[m]\cdot(\frac{1}{2\pi}\int_{-\pi}^{\pi}\cdot e^{j\omega(n-m)}d\omega) \tag{3.47}$$

$$= \sum_{m=-\infty}^{\infty} x[m]\cdot\frac{sin\ \pi(n-m)}{\pi(n-m)} \tag{3.48}$$

Wegen

$$\lim_{x\to 0}\frac{\sin x}{x} = \lim_{x\to 0}\frac{x+O(x^3)+..}{x} = 1$$

ist die obige Summe gleich:

$$\sum_{m=-\infty}^{\infty} x[m]\cdot\delta[n-m] \tag{3.49}$$

$$= x[n] \tag{3.50}$$

Wir erhalten also wieder die Folge x, was zu zeigen war.

3.7 Das Faltungstheorem

Zu den wichtigsten Aussagen über die Fouriertransformation und die Faltung gehört das folgende Theorem: sofern die Beziehungen

$$x[n] = \mathcal{F}^{-1}\{X(j\omega)\} \tag{3.51}$$

$$h[n] = \mathcal{F}^{-1}\{H(j\omega)\} \tag{3.52}$$

$$y[n] = \mathcal{F}^{-1}\{Y(j\omega)\} \tag{3.53}$$

$$y[n] = h\star x \tag{3.54}$$

gelten, so folgt:

$$Y(j\omega) = X(j\omega)\cdot H(j\omega) \tag{3.55}$$

(Faltungstheorem:) Zu einer Faltung von Folgen im Zeitbereich ist die Multiplikation der zugehörigen Fouriertransformierten äquivalent.

Beweis:

Wir können x über das Fourierintegral darstellen:

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(j\omega)\cdot e^{j\omega n}\,d\omega \tag{3.56}$$

Dieses stellt das Ergebnis eines Grenzübergangs der Betrachtung der Approximation mit Basisfunktionen mit sehr kleinem Frequenzabstand $\Delta\omega$ dar:

$$x[n] = \lim_{\Delta\omega\to 0}\frac{1}{2\pi}\sum_k X(j\,k\,\Delta\omega)\cdot e^{j\,k\,\Delta\omega\,n}\,\Delta\omega \tag{3.57}$$

Für jede der Frequenzen $k\,\Delta\omega$ gilt die Gleichung 3.36 und damit erhalten wir:

$$y[n] = \lim_{\Delta\omega\to 0} \frac{1}{2\pi} \sum_k H(j\,k\,\Delta\omega)\cdot X(j\,k\,\Delta\omega)\cdot e^{j\,k\,\Delta\omega\,n}\,\Delta\omega \tag{3.58}$$

$$= \frac{1}{2\pi}\int_{-\pi}^{\pi} H(j\,\omega)\cdot X(j\,\omega)\cdot e^{j\,\omega\,n}\,d\omega \tag{3.59}$$

Wenn sich das Signal y im Zeitbereich durch Fourier-Rücktransformation aus dem Produkt von H und X ergibt, dann ist y im Frequenzbereich durch das Produkt selbst gegeben:

$$Y(j\omega) = H(j\omega)\cdot X(j\omega) \tag{3.60}$$

3.8 Das Abtastheorem

Unter den vielen Möglichkeiten, zeitkontinuierliche Signale abzutasten, ist die periodische Abtastung die häufigste. Für diese gilt:

$$x[n] = x_c(nT) \text{ mit } -\infty < n < \infty \tag{3.61}$$

T heißt **Abtastperiode** und $f_s = \frac{1}{T}$ heißt **Abtastfrequenz**. Ein System, welches die Operation 3.61 ausführt, heißt **Abtaster** oder *continous to discrete (CD-) converter* (siehe Abb. 3.11). Es ist eine Idealisierung der Schaltung nach Abb. 2.5.

Abbildung 3.11: Idealer Abtaster

Es ist nicht in jedem Fall möglich, aus dem zeitdiskreten Signal wieder das ursprüngliche Signal x_c zu rekonstruieren. Allerdings kann man zeigen, daß dies bei einer Einschränkung der Klasse der Eingangssignale dennoch gelingt.

Zur mathematischen Behandlung ist es zweckmäßig, den Abtastprozeß in in zwei Teilprozesse zu zerlegen, nämlich in

1. die Multiplikation von x_c mit einem Signal s, welches an den Abtastzeitpunkten 1 und sonst 0 ist, sowie
2. dem Übergang von dem resultierenden zeitkontinuierlichen Signal zu einem zeitdiskreten Signal.

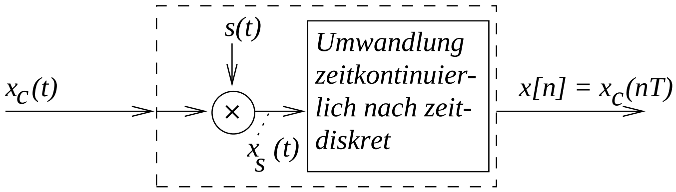


Abbildung 3.12: Modellvorstellung des Abtastvorgangs

$s(t)$ kann dargestellt werden als Summe von Impulsen, die jeweils zu einer Zeit $t = nT$ gleich Eins sind und sonst Null:

$$s(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{3.62}$$

Damit ergibt sich das Signal $x_s(t)$ zu

(3.63) $$x_s(t) = x_c(t) \cdot s(t) = x_c(t) \cdot \sum_{n=-\infty}^{\infty} \delta(t - nT)$$

Da $x_c(t)$ außerhalb der Abtastzeitpunkte ohnehin mit Null multiplizeirt wird, gilt auch

(3.64) $$x_s(t) = \sum_{n=-\infty}^{\infty} x_c(nT) \cdot \delta(t - nT)$$

Da sich $x_s(t)$ aus dem Produkt von $s(t)$ und $x_c(t)$ zusammensetzt, ist die Fouriertransformierte $X_s(j\Omega)$ von $x_s(t)$ gleich der Faltung der Fouriertransformierten $S(j\Omega)$ und $X_c(j\Omega)$ von $s(t)$ bzw. $x_c(t)$:

(3.65) $$X_s(j\Omega) = S(j\Omega) \star X_c(j\Omega)$$

Die Fouriertransformierte der Impulsfolge $s(t)$ ist [OS95]:

(3.66) $$S(j\Omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\Omega - k\Omega_s) \text{ mit}$$

(3.67) $$\Omega_s = \frac{2\pi}{T}$$

Daraus folgt:

(3.68) $$X_s(j\Omega) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c(j\Omega - k\Omega_s)$$

Die Gleichung 3.68 besagt, daß die Fouriertransformierte von $x_s(t)$ sich aus periodisch wiederholenden Kopien der Fouriertransformierten von $x_c(t)$ zusammensetzen. Diese Kopien sind jeweils um ein ganzzahliges Vielfaches der Abtastfrequenz Ω_s verschoben. Diese Situation ist in Abb. 3.13 zu sehen.

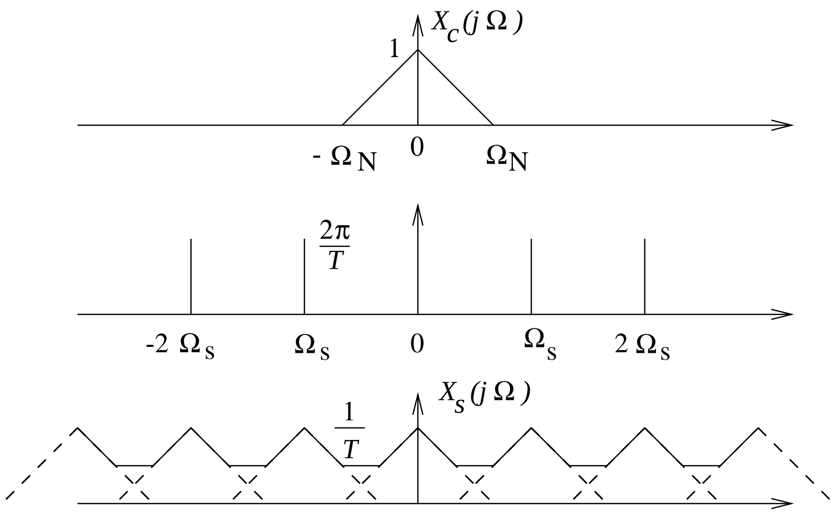


Abbildung 3.13: Auswirkungen im Frequenzbereich bei einer Abtastung im Zeitbereich

Dem Bild ist zu entnehmen, daß sich die Kopien unter der Bedingung

(3.69) $$\Omega_s \quad > \quad 2\Omega_N$$

nicht überlappen und nach der Bildung der Summe noch als identische Kopien zur Verfügung stehen. Mit einem idealen Filter, welches eine der Kopien herausfiltert und der inversen Fouriertransformation läßt sich unter dieser Bedingung das originale Signal $x_c(t)$ rekonstruien.

Daraus ergibt sich das folgende Theorem [Nyquist, 1928; Shannon, 1949]:

Nyquist-Theorem: Sei $x_c(t)$ ein bandbegrenztes Signal mit $X_c(j\Omega) = 0$ für $|\Omega| > \Omega_N$. Dann ist $x_c(t)$ eindeutig bestimmt durch seine Abtastwerte $x[n] = x_c(nT)$ (mit $n \in \mathbb{Z}$) bestimmt, wenn

(3.70) $$\Omega_s = \frac{2\pi}{T} > 2\Omega_N$$

Die Frequenz Ω_N wird als **Nyquist-Frequenz** bezeichnet.

Das Ausgangssignal läßt sich also eindeutig rekonstuieren, falls die Abtastfrequenz mindestens gleich dem Doppelten der höchsten in $x_c(t)$ vorkommenden Frequenz ist.

3.9 Kompressionsverfahren

7. Vorles.

3.9.1 Übersicht

Das Problem der Realisierung von Realzeitverhalten tritt auch für viele Systeme auf, welche aus Gründen beschränkter Speicher- und Übertragungskapazität mit Kompression von Daten arbeiten müssen. Mittels Kompression läßt sich die benötigte Datenrate (und entsprechend auch die benötigte Speicherkapazität) erheblich senken, wie etwa die Tabelle 3.1 zeigt.

Anwendung	Datenraten	
	unkomprimiert	komprimiert
Sprache 8 kHz, 8 Bit	64 kBit/s	2-4 kBit/s
Video-Konferenz 15 fps, 352x240, 8bit	30,41 MBit/s	64-768 kBit/s
Digital-Audio (Stereo) 44,1 kHz; 16 bit	1,5 MBit/s	0,128-1,5 MBits/s
HDTV 59,94 fps, 1280x1024, 8 bit	1,33 GBit/s	20 MBit/s

Tabelle 3.1: Datenraten

Wir wollen uns hier mit den Grundlagen einiger dieser Kompressionsverfahren beschäftigen, da diese zu den Basistechniken zur Realisierung eingebetteter und Realzeitsysteme gehören [4].

Es existieren viele Kompressionsverfahren. Abb. 3.14 zeigt eine Übersicht.

Bei den modellbasierten Verfahren gehen Annahmen über das Anwendungsgebiet und die zu modellierenden Daten ein. Beispiele sind die Kodierung im THUMB-Befehlssatz und eine spezielle Dateinkodierung auf der Basis der Postscript-Syntax. Allgemein verwendbare, modellunabhängige Komprimierungsverfahren können derartige Informationen nicht ausnutzen.

Eine Grundtechnik der Kompression ist die **differentielle Kodierung** auf der Basis des Rückwärts-Differenzen-Systems. Wegen der starken Korrelation zwischen aufeinanderfolgenden Signalelementen beeinflußt diese Transformation die Verteilung der Signalwerte erheblich (siehe Abb. 3.15).

Die nachfolgend beschriebenen Kompressionsverfahren erzielen nach Vorverarbeitung durch das Rückwärts-Differenzen-System eine erheblich bessere Kompressionsrate.

[4] Diese Grundlagen werden hier anhand des Buches von Bhaskaran dargestellt [BK95] dargestellt.

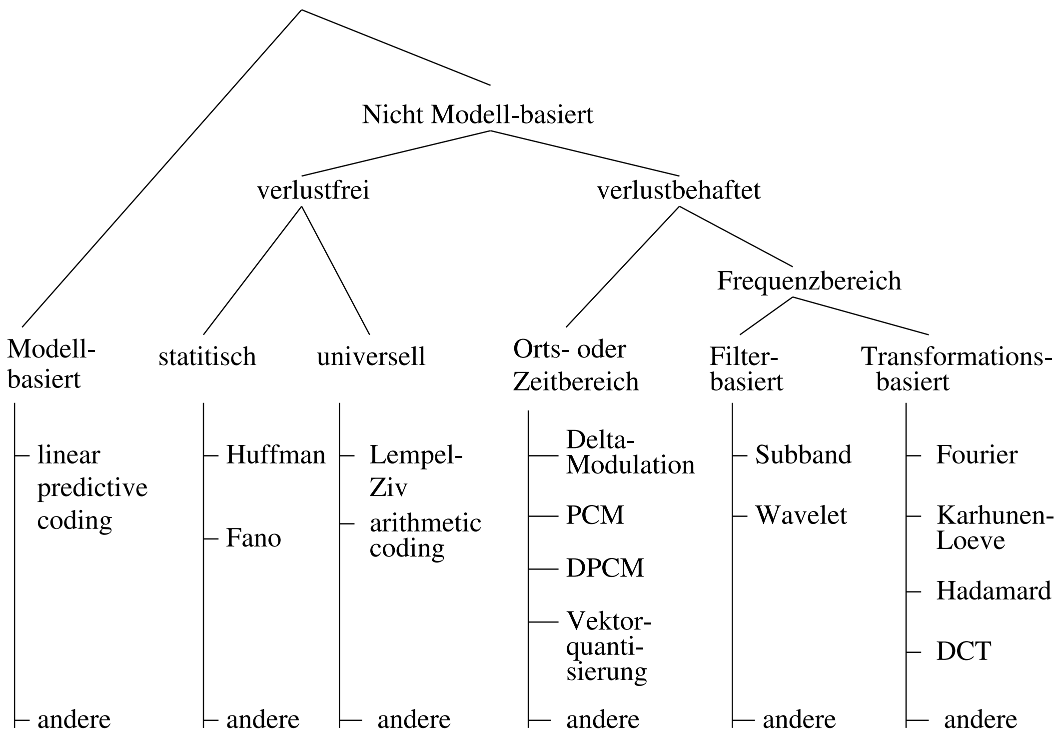


Abbildung 3.14: Kompressionsverfahren

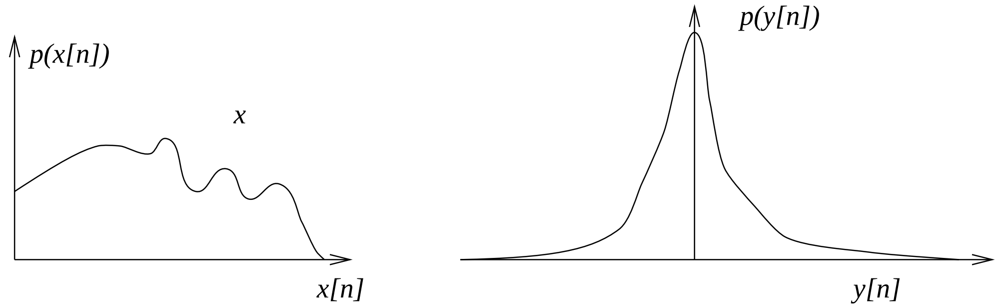


Abbildung 3.15: Häufigkeit von Werten vor und nach differentieller Kodierung

3.9.2 Huffmann-Encoder

Bei der Huffman-Kodierung (Huffman, 1952) geht man von einem Signal in Form eines Stromes von Werten oder Symbolen aus. Die Erzeugung eines **Huffman-Codes** basiert dann auf folgenden Schritten:

1. Ordne die Symbole nach ihren Häufigkeiten.
2. Wähle die beiden Symbole mit der geringsten Häufigkeit und kodiere sie in einem neuen Symbol. Die Häufigkeit dieses Symbols ist gleich der Summe der Häufigkeiten der beiden einzelnen Symbole.
3. Wiederhole Schritt 2 bis nur ein einziges Symbol verbelibt.

Dieser Konstruktionsprozeß kann als Erezugung eines binären Baumes angesehen werden. Tabelle 3.2 zeigt links zunächst die Symbole und deren Häufigkeiten. Die folgenden Spalten entsprechen jeweils einer Iteration des Algorithmus.

Verfolgt man das Zusammenfassen der Symbole rückwärts, so kommt man zu dem Baum der Abb. 3.16. Der Code eines Symbols besteht aus der Folge der fettgedruckten Werte entlang des Pfades von der Wurzel zu dem Symbol. Man beachte, daß die einzelnen Symbole in unterschiedlicher Länge kodiert sind: die häufigen Symbole haben kürzere Codes.

Zur Dekodierung eines Stroms von Nullen und Einsen kann man verschiedene Verfahren benutzen.

		Iter.:	1	Iter.:	2	Iter.:	3	Iter.:	4	Iter.:	5	Iter.:	6
e	0,3	e	0,3	e	0,3	e	0,3	{l,r}	0,4	{k,w,?,u,e}	0,6	{k,w,?,u,	1,0
l	0,2	l	0,2	l	0,2	{k,w,?,u}	0,3	e	0,3	{l,r}	0,4	e,l,r}	
r	0,2	r	0,2	r	0,2	l	0,2	{k,w,?,u}	0,3				
u	0,1	u	0,1	{k,w,?}	0,2	r	0,2						
?	0,1	?	0,1	u	0,1								
k	0,05	{k,w}	0,1										
w	0,05												

Tabelle 3.2: Beispiel der Anwendung des Huffman-Algorithmus

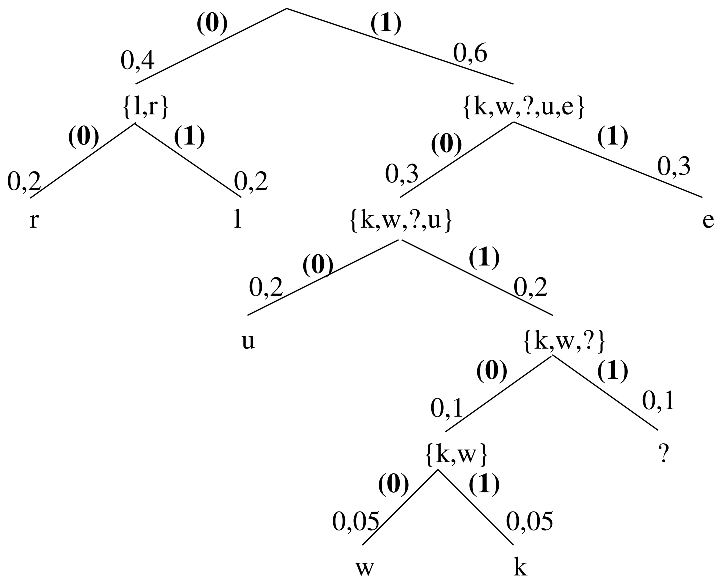


Abbildung 3.16: Baum zur Konstruktion von Huffman-Codes

1. Man kann den Strom bitweise lesen und jeweils den Baum durchlaufen, bis ein Blatt erreicht ist.
2. Unter der Annahme einer maximalen Länge m eines Codes kann man eine Tabelle aufbauen, welche 2^m Einträge besitzt und jeweils das erste zu erzeugende Symbol und die Anzahl der davon erfaßten Codebits erhält.

Ein Problem der Huffman-Codes ist ihre möglicherweise sehr große Länge. Es gibt Verfahren, welche Huffman-Codes einer begrenzten Länge erzeugen. Diese Verfahren erreichen natürlich nicht ganz die Kompressionsraten von Huffman-Codes unbegrenzter Länge. Sie sind aber leichter zu realisieren, insbesondere, wenn zur Dekodierung die zweite der oben angegebenen Möglichkeiten genutzt wird.

3.9.3 Diskrete Cosinus-Transformation (DCT)

Mit verlustfreien Kompressionsverfahren wie dem Huffman-Algorithmus lassen sich vielfach nicht die Kompressionsfaktoren erzielen, die man benötigt. Man kann diese Faktoren erhöhen, sofern man darauf verzichtet, das Original exakt wiederherzustellen.

Aus der Menge der möglichen Verfahren werden wir hier die blockbasierten Transformationsverfahren vorstellen. Wir nehmen dazu an, daß wir uns mit der Kompression von 2D-Daten beschäftigen. Wir gehen davon aus, daß ein Bild oder ein Ausschnitt daraus als $N \times N$-Matrix $\mathbf{x}$ gegeben ist. Eine lineare Transformation kann dann durch $\mathbf{y} = T_c \mathbf{x} T_r^t$ ausgedrückt werden, wobei T_r^t die Transponierte einer Matrix T_r ist. T_c und T_r heißen **Transformationskerne**. Für sog. **symmetrische Kerne** gilt: $T_c = T_r = T$ und $\mathbf{y} = T\mathbf{x}T^t$. In allen praktischen Anwendungen verlangt man, daß eine Umkehrmatrix U existiert, sodaß $\mathbf{x} = U\mathbf{y}U^t$ gilt.

Mit der Transformation möchte man eine kompakte Repräsentation erreichen. Zu diesem Zweck eliminiert man den weniger signifikanten Teil von $\mathbf{y}$.

Häufig benutzte Transformationen sind:

- die diskrete Fourier-Transformation DFT (im 2-dimensionalen),
- die Karhunen-Loeve-Transformation KLT,
- die diskrete Cosinus-Transformation DCT,
- die diskrete Sinus-Transformation DST,
- die diskrete Hadamard-Transformation DHT.

Für das Beispielbild in Abb. 3.17

Abbildung 3.17: Bildoriginal (©Kluwer, 1995)

zeigen die Abb. 3.19 und 3.18 summarisch die Ergebnisse. In diesen beiden Abbildungen bedeuten dunkle Regionen, daß die Koeffizienten der (2-dimensionalen) Basisfunktionen sehr klein sind.

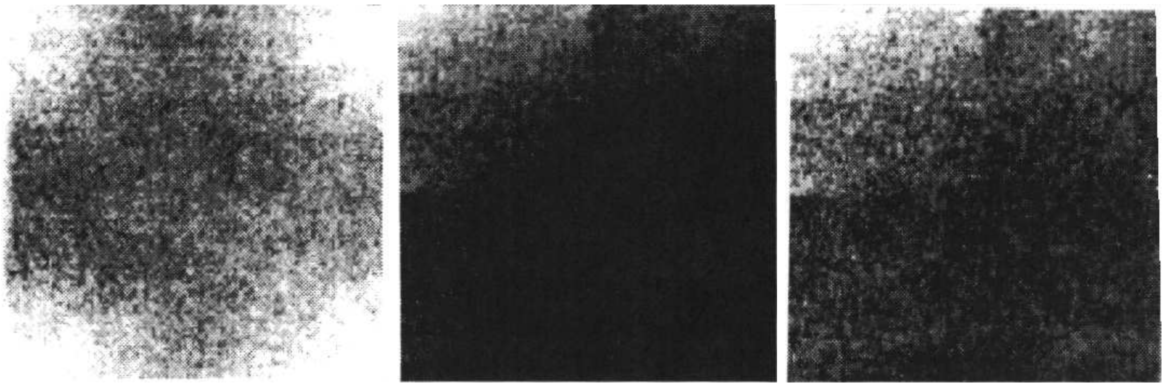


Abbildung 3.18: Koeffizienten: links: DFT, mitte: KLT, rechts: DCT; (©Kluwer, 1995)

Es ist zu sehen, daß KLT die größte Kompaktheit erreicht. DHT und DFT sind relativ schlecht geeignet.

Bei der KLT sind die Transformationsmatrizen abhängig von Bildinhalt. Sie ist daher relativ aufwendig. Alle übrigen Transformationen sind unabhängig vom Bildinhalt. Unter diesen liefert die DCT die beste Kompression.

Die DCT liefert eine Approximation eines Bildausschnitts mit Hilfe gewichteter Anteile von Basisfunktionen. Sinnvoll ist dies immer nur für Bildausschnitte, innerhalb derer eine Korrelation vorhanden ist. In der Praxis werden Ausschnitte von 8×8 Bildpunkten benutzt. Die DCT-Basisfunktionen hierfür zeigt die Abb. 3.20.

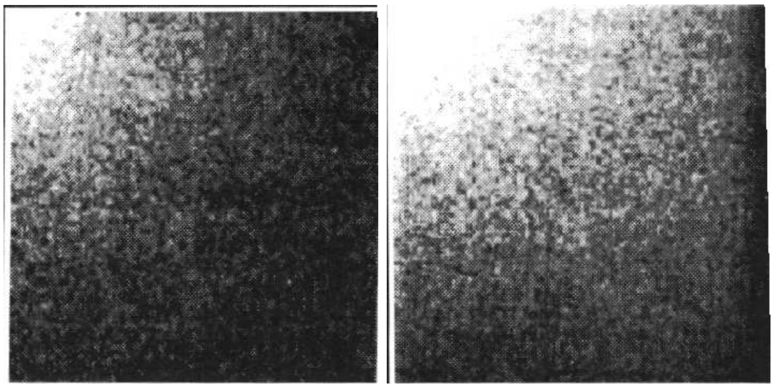


Abbildung 3.19: Koeffizienten: links: DST, rechts: DHT; (©Kluwer, 1995)

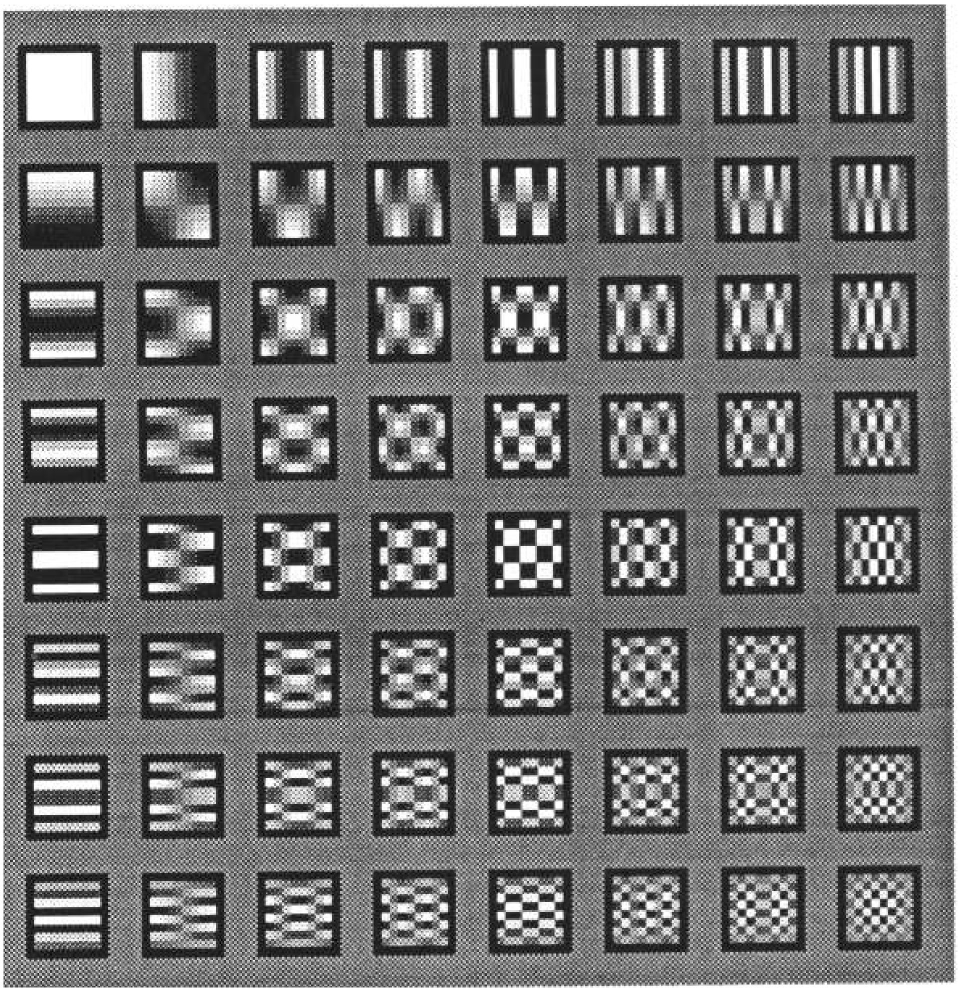


Abbildung 3.20: Orthogonales Funktionensystem für die Cosinustransformation (©Kluwer, 1995)

Formal berechnet sich die DCT nach der folgenden Formel:

$$y_{kl} = \frac{c_k c_l}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} x_{ij} \cos\left(\frac{(2i+1)k\pi}{16}\right) \cos\left(\frac{(2j+1)l\pi}{16}\right)$$

mit $k, l \in [0..7]$ und $c_k = \frac{1}{\sqrt{(2)}}$ für $k = 0$ und $c_k = 1$ sonst.

Die inverse DCT (IDCT) berechnet sich ähnlich wie die DCT [BK95]:

$$x_{ij} = \sum_{k=0}^{7} \sum_{l=0}^{7} \frac{c_k c_l}{4} y_{kl} \cos\left(\frac{(2i+1)k\pi}{16}\right) \cos\left(\frac{(2j+1)l\pi}{16}\right)$$

Für die IDCT kann daher praktisch derselbe Hardwarebeschleuniger wie für die DCT benutzt werden.

Den Kontext des üblichen Einsatzes der DCT zeigt die Abbildung 3.21.

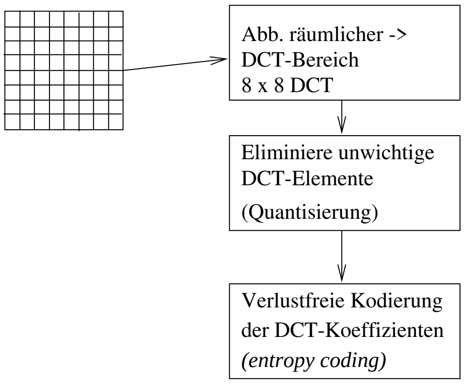


Abbildung 3.21: Kontext des Einsatzes der DCT

In der VLC-Stufe (engl. VLC=*variable length coding*) werden dabei häufig aufeinanderfolgende identische Symbole durch das Symbol und einen Wiederholungszähler kodiert und zusätzlich eine Huffman-Kodierung durchgeführt. .

3.9.4 Bewegungskompensation

Bei Bewegtbildern (Videos) muß zur Erzielung einer hohen Kompressionsrate auch ausgenutzt werden, daß es eine große Korrelation der einzelnen Bilder untereinander gibt. Nachfolgende Bilder zeigen meist Szenen, in denen sich Objekte nur geringfügig gegenüber den vorhergehenden bewegt haben. Häufig bewegen sich viele der Objekte auch überhaupt nicht. Dies kann man nutzen, um nur die Information über die Änderungen zu übertragen.

Zu diesem Zweck muß zunächst vor allem bestimmt werden, wie sich die Objekte bewegt haben. Üblicherweise wird für einen bestimmten Block eines Folgebildes bestimmt, mit welchem Block eines Vorgängerbildes sich die beste Übereinstimmung ergibt und wie groß der entsprechende Bewegungsvektor ist (siehe Abb. 3.22).

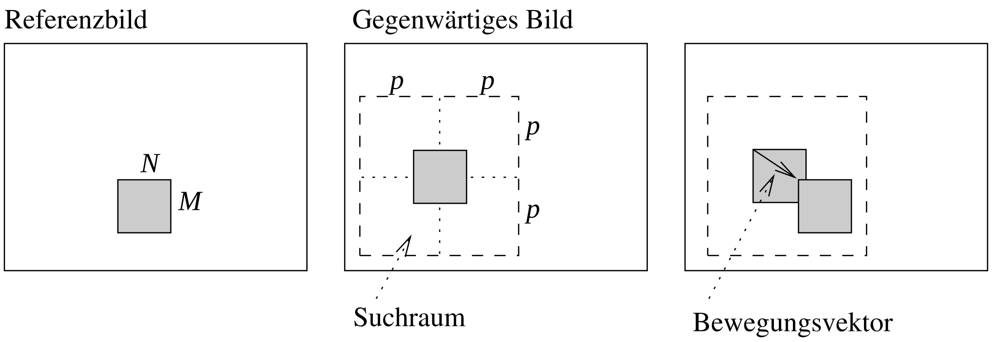


Abbildung 3.22: Bestimmung des Bewegungsvektors

Für diesen Bewegungsvektor muß ein Fehlerkriterium minimal werden.

Für die Wahl der Werte von N, M und p muß ein Kompromiß von Aufwand und Leistung gefunden werden. Für übliche Videos wählt man z.B. $N = M = 16$. Für Sportfilme erreicht man mit $p = 63$, für andere mit $p = 15$ gute Resultate.

Es gibt eine Vielfalt von Algorithmen zur Bestimmung von Bewegunsgvektoren (siehe [BK95]). Wir wollen uns hier v.a. mit deren Einsatz im MPEG-Standard beschäftigen.

3.9.5 MPEG

1988 schuf die *International Standards Organization* (ISO) eine Gruppe von Experten, die sog. *Motion Picture Experts Group* (MPEG). 1991 stellte diese Gruppe Kompressionsverfahren für Video- und Audio-Datenströme vor, die heute als MPEG-1 bekannt sind. Nachfolgende Arbeiten führten zu MPEG-2 und MPEG-4.

MPEG-Standards spezifizieren im Prinzip nur die Bild- und Audioformate und lassen damit Freiheiten für die Implementierung der Kompression. Zusätzlich stehen aber auch Beschreibungen in Form von C-Programmen zur Verfügung.

Bei der Kodierung von Bewegtbildern ist zu berücksichtigen, daß Videoszenen auch geschnitten werden können. Daher muß zumindest ein Teil der Bilder die volle, ursprüngliche Information enthalten.

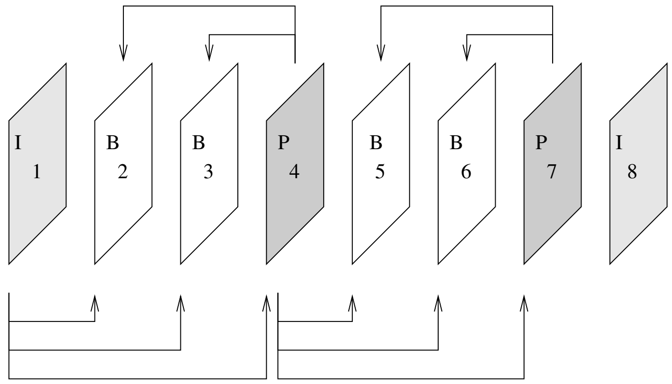


Abbildung 3.23: Bildtypen bei MPEG

Daher definiert MPEG-1 drei unterschiedliche Bildtypen (siehe auch Abb. 3.23):

1. I-Bilder: dies sind vollständige Bilder, die ohne Bewegungskompensation berechnet werden,
2. P-Bilder werden mittels Bewegungsschätzung aus vorhergehenden P- oder I-Bildern berechnet.
3. B-Bilder werden mittels Bewegungsschätzung aus vorhergehenden und/oder nachfolgenden Bildern berechnet. Sie selbst sind nicht mehr Ausgangspunkt für weitere Interpolationen. Daher dürfen sie stärkere Verfälschungen enthalten und können deshalb stärker komprimiert werden.

Für die einzelnen Bildtypen ergibt sich ein unterschiedlicher Speicherbedarf (siehe Abb. 3.24).

Ein typisches Blockschaltbild eines MPEG-Dekoders ist in Abb. 3.25 zu sehen.

Es werden Puffer-Stufen benötigt, um die schwankende Datenrate auszugleichen. Die inverse Quantisierung ist mit Q^{-1} gekennzeichnet. Der VLC-Dekoder nimmt eine Huffman-Dekodierung vor.

MPEG-2 ist zu MPEG-1 aufwärts kompatibel. Es zeichnet sich durch zusätzliche Formatoptionen aus und unterstützt auch den *interlaced*-Modus, bei dem zwei Halbbilder jeweils nur die geraden bzw. die ungeraden Zeilen enthalten.

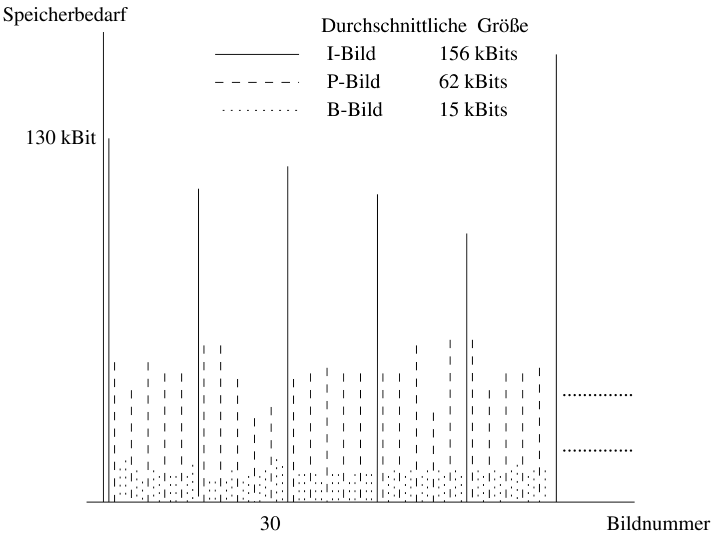


Abbildung 3.24: Speicherbedarf einer Videosequenz (nach Bhaskaran [BK95])

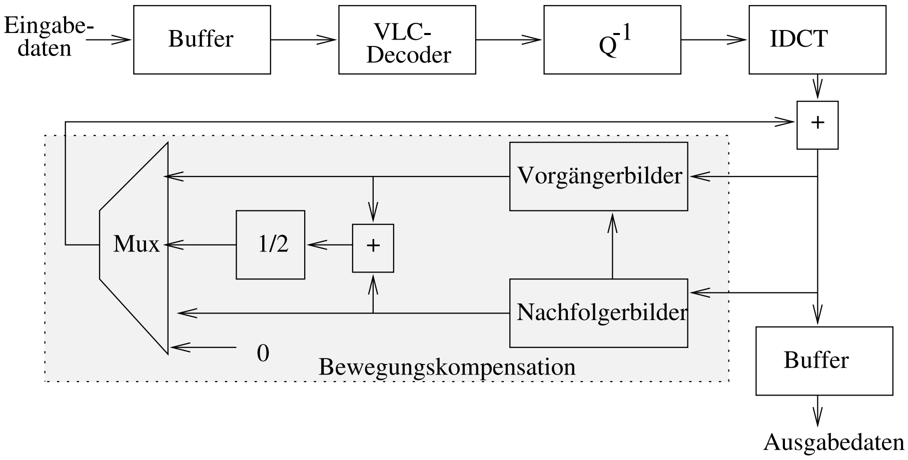


Abbildung 3.25: Blockdiagramm eines MPEG-Dekoders

Kapitel 4

Realzeit-Betriebsysteme

8./9. Vorles.

4.1 Funktionalität von Realzeit-Betriebssystemen

Bei der Realisierung eingebetteter Systeme werden immer wieder Standardfunktionen benötigt, wie z.B. die Realisierung eines Mehrprozeßsystems. die Kommunikation zwischen mehreren Prozessen und die Zeitüberwachung und -steuerung. Diese Funktionen sollten sicherlich nicht für jede Applikation neu konzipiert werden. Sie sollten als vorentworfene Komponenten eines Realzeit-Betriebssystems (RTOS) während der Anwendungsentwicklung bereits bekannt sein. Im Unterschied zu Standardbetriebssystemen sollte es jedoch möglich sein, nur die für eine Anwendung wirklich wichtigen Komponenten einzusetzen.

Realzeit-Betriebssysteme sollten die folgenden Eigenschaften haben (siehe auch [Zoe87]):

- **Flexible Konfigurationsmöglichkeiten**

  So vielfältig wie eingebettete Systeme sind auch deren Anforderungen an Betriebssysteme. Wegen der geforderten Effizienz eingebetteter Systeme muß für jede Anwendung gerade die Betriebssystem-Funktionalität bereitgestellt werden, die auch benötigt wird. Dies ist auch möglich, da alle Anwendungsprogramme während des Systementwurfs bekannt sind. Auf der Basis dieser Kenntnis kann ein spezialisiertes Betriebssystem generiert werden (dies schließt vielfach eine Neucompilation ein).

- **Direkter Zugang zu E/A-Geräten**

  Ein direkter Zugang zu den E/A-Geräten ist wünschenswert, da die Abwicklung von Ein/Ausgaben durch ein Betriebssystem zuviel Overhead erzeugt. Dies gilt insbesondere, da Ein/Ausgaben bei Realzeit-Systemen häufig nur aus wenigen Bytes bestehen und die Geräte vielfach auch keine Synchronisation erfordern (d.h. im Sinne der Vorlesung Rechnerarchitektur *immediate devices* sind). Auch ist kein Schutz der Benutzer gegeneinander erforderlich.

- **Interrupts auf Anwenderebene**

  Zur Organisation des Ablaufs verschiedener Prozesse ist die Benutzung von Interrupts erforderlich. Interrupts dienen also nicht ausschließlich dem Aufruf von Betriebssystemfunktionen.

- **Zeitverwaltung**

  Ein Realzeit-Betriebssystem "muß in der Lage sein, zu beliebigen, aber festen Zeitpunkten Aufträge zu starten oder fertiggestellt zu haben. Dazu gehört die Verwaltung einer Auftragsliste, der sog. Weckliste, in der Echtzeiten und die zugehörigen Aktionen, bzw. Prozeßaufrufe vermerkt sind" [Zitat Zöbel].

- **Scheduling und Synchronisation**

  "Unter dem Scheduling versteht man die zeitliche Zuteilung des Prozessors an die jeweiligen Anwenderprozesse. Bei Echtzeitsystemen sind die vorhandenen Prozesse sowie erlaubte zeitliche Anordnungen weitgehend bekannt. Sie sind durch die Aufgabenstellung vorgegeben. Es gibt grundsätzlich zwei Möglichkeiten, wie Echtzeitsysteme zeitliche Anordnungen ermöglichen:

  1. **statisches Scheduling**
     (Statische Ausgabe von Ablaufreihenfolgen in Form von Präzedenzgraphen oder Gantt-Tabellen.

Dieser Fall wird auch im Buch von Kopetz beschrieben:

"TT Systems

In an entirely time-triggered system, the temporal control structure of all tasks is established *a priori* by off-line support-tools. This temporal control structure is encoded in a *Task-Descriptor List (TDL)* that contains the cyclic schedule for all activities of the mode (Figure 4.1). This schedule considers the required precedence and mutual exclusion relationships among the task such that an explicit coordination of the tasks by the operating system at run time is not necessary.

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T3	20
47	send M3	

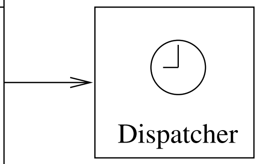


Abbildung 4.1: Task descriptor list in a TT operating system

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant. If a task is started, the operating system informs the task of its activation time, which is synchronized within the cluster. After task termination, the operating system copies the results of the task to the CNI[1].

A task of a TT system with non-preemptive S-Tasks[2] is one of the two states; inactive or active (Figure 4.2)

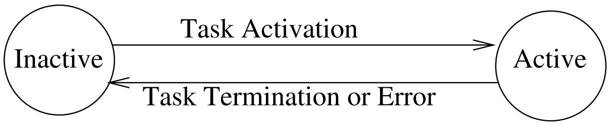


Abbildung 4.2: State diagram of non-preemptive S-tasks

In a preemptive S-task, two sub-states of the active state can be distinguished, ready or running, depending on whether the task is in possession of the CPU or not (Figure 4.3)

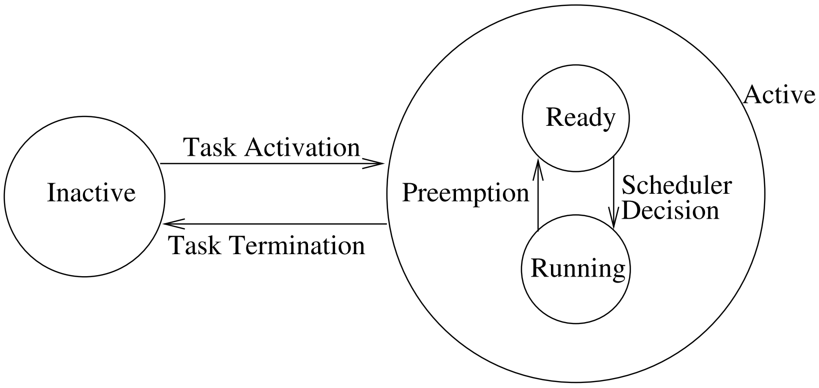


Abbildung 4.3: State diagram of preemptive S-tasks

The Application Program Interface (API):

The application program interface (API) of an S-task in a TT system consists of three data structures and two operating system calls. The data structures are the input data structure, the output data structure, and the h-state data structure .... of the task. A stateless S-task

[1] Ausgabe-Bereich

[2] Dies sind einfache Tasks ohne Interprozeßkommunikation.

does not have an h-state data structure at its API. The system calls are TERMINATE TASK and ERROR. The TERMINATE TASK system call is executed whenever the task has reached its termination point. In case of an error that cannot be handled within the application task, the task terminates its operation with the ERROR system call."

2. "Bereitstellung von Synchronisationsoperationen, mit deren Hilfe zeitliche Reihenfolgen, Verzögerungen und gegenseitiger Anschluß ... formuliert werden können" [Zitat Zöbel]

   Kopetz schreibt dazu:

   "**ET Systems with S-tasks**

   In an entirely event-triggered system, the sequence of task executions is determined dynamically by the evolving application scenario. Whenever a significant event happens, a task is released to the active (ready)state, and the dynamic scheduler is invoked. It is up to the scheduler to decide at run-time which one of the ready tasks is selected for the next service by the CPU. .... The WCET (Worst-Case Execution Time) of the scheduler contributes to the WCAO (Worst-Case Administrative Overhead) of the operating system.

   The significant events that cause the activation of a task are:

   (a) an event from the node's environment, i.e., the arrival of a message or an interrupt from the controlled object, or
   (b) a significant event inside the host, i.e., the termination of a task or some other condition within a currently task, or
   (c) the progression of the clock to an specified point in time. This point can be specified either statically or dynamically.

   Non-premptive S-tasks

   An ET operating system that supports non-premptive S-tasks will take a new scheduling decision after the currently running task has terminated. This simplifies the task management in the operating system but severely restricts its responsiveness. If a significant event arrives immediately after the longest task has been scheduled, this event will not be considered until this longest task has completed.

   Preemptive S-tasks

   In a RT operating system that supports task preemption, each occurence of a significant event can potentially activate a new task and cause an immediate interruption of the currently executing task to invoke a new decision by the scheduler. Depending on the outcome of the dynamic scheduling algorithm, the new task will be selected for execution or the interrupted task will be continued (Figure 4.3). Data conflicts between concurrently executing S-tasks can be avoided if the operating system copies all input data requred by this task from the global data area and the communication-network interface (CNI) into a private data area of the task at the time of task activation.

   The Application Program Interface (API):

   The API of an operating system that supports event-triggered S-tasks requires more system calls than an operating system that only supports time-triggered tasks. Along with the data structures and the already introduced systems calls of a TT system, the operating system must provide system calls to ACTIVATE a new task, either immediately or at some future point in time. Another system call is needed to DEACTIVATE an already activated task.

   ET Systems with C-Tasks

   The state transistion diagram of an ET system with C-tasks[3] has three sub-states of the active state, as shown in Figure 4.4.

   In addition to the ready and running state, a C-task can be in the blocked state waiting for an event outside the c-task to occur. Such an event can be a time-event, meaníng that the real-time clock has advanced to a specified point, or any occurence that has been specified in the wait statement. An example of a blocked state is the suspension of the task execution to wait for an input event message. The WCET of a C-task cannot be determined independently of the other tasks in the node. It can depend on the occurence of an event in the node environment, as seen from the example of waiting for an input message. The timing analysis is not a local issue of a single task anymore; it becomes a global system issue. In the general case it is impossible to give an upper bound for the WCET.

[3] Komplexe Tasks mit Interprozeßkommunikation.

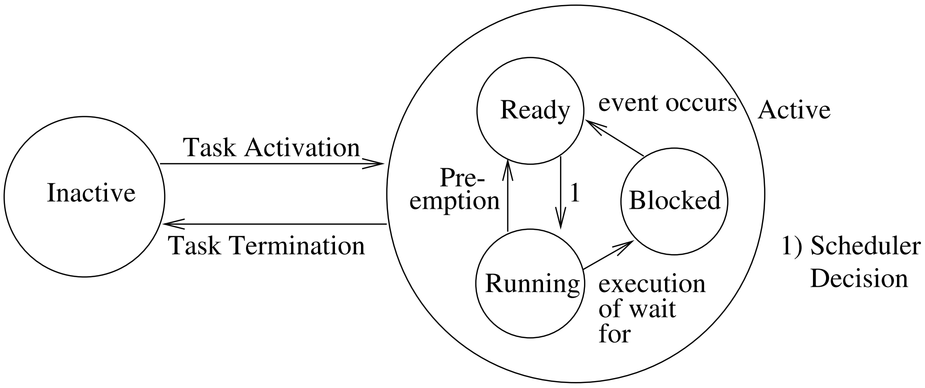


Abbildung 4.4: State diagram of a preemptive C-tasks with blocking

4.2 Globale Zeit

4.2.1 Begriffe

Wir wollen uns im nächsten Abschnitt näher mit der Zeitverwaltung beschäftigen. Für diesen Teil des Begleittextes diente das im ersten Kapitel angegebene Buch von Kopetz [Kop97] als Grundlage.

Der Begriff der Zeit ist offensichtlich zentral für verteilte Realzeitsysteme. Innerhalb der Realzeitsysteme wird eine präzise Kenntnis der Zeit beispielsweise benötigt, um Aufschluß über den zeitlichen Ablauf von Ereignissen und damit über deren ggf. feststellbare Reihenfolge zu gewinnen.

Das Kontinuum der realen Zeit kann durch eine gerichtete Zeitachse modelliert werden. Diese besteht aus einer unendlichen Menge T von Zeitpunkten (engl. *instants*) mit folgenden Eigenschaften:

1. T ist eine geordnete Menge. Die Ordnung zwischen je zwei Zeitpunkten heißt **zeitliche Ordnung**.

2. T ist dicht. Zwischen je zwei verschiedenen Zeitpunkten existiert ein weiterer.

Ein Intervall aus T heißt **Zeitraum** oder **Dauer**.

Zeitstandards

Zwei (allerdings diskrete) Zeitstandards sind für unsere Zwecke wichtig:

1. Die Internationale Atomzeit (franz. *temps atomique internationale*, TAI):

   Die Internationale Atomzeit definiert eine Sekunde als ein Vielfaches der Frequenz einer bestimmten radioaktiven Strahlung. TAI ist frei von 'Sprüngen'.

2. Die *Universal Time Coordinated* (UTC):

   UTC wird von astronomischen Daten abgeleitet. Aufgrund der Unregelmäßigkeiten der Erddrehung schwankt die Länge der abgeleiteten Zeiteinheit. TAI und UTC stimmten für den 1. Januar 1958 überein. In der Zwischenzeit war es notwendig, insgesamt 30 Korrektursekunden einzufügen. Dieses Schema ist nicht unproblematisch, da beim Einfügen in der Neujahrsnacht das neue Jahr zweimal beginnt.

Ereignisse

Ein **Ereignis** (engl. *event*) findet an einem Zeitpunkt statt. Wenn zwei Ereignisse zu demselben Zeitpunkt stattfinden, heißen sie **gleichzeitig** (engl. *simultaneous*). Zeitpunkte sind total geordnet. Ereignisse sind es nicht, da gleichzeitige Ereignisse nicht ohne weiteres Elemente einer Ordnungrelation sind.

In vielen Realzeitsystemen gibt es kausale Abhängigkeiten zwischen Ereignissen. Ein Ereignis, welches Anfangselement einer Kette von kausalen Abhängigkeiten ist, heißt **Ursprungsereignis**. Kausale Abhängigkeiten definieren eine **kausale Ordnung**. Zwei Ereignisse können nur dann kausal geordnet sein, wenn ihre zeitliche Ordnung dies nicht ausschließt. Sichere Ausssagen über die zeitliche Ordnung sind ein wichtiges Mittel, um ggf. kausale Abhängigkeiten verneinen zu können.

Eine schwächere Ordnung ist die des Eintreffens von Informationen am Kommunikationscontroller (engl. *delivery order*). Diese Ordnung erlaubt keine Aussagen über kausale Abhängigkeiten.

Uhren

Uhren sind 'Geräte', welche periodisch aufgrund von Ereignissen einen Zeitzähler erhöhen. Diese Ereignisse heißen **Mikrozeitschläge** (engl. *microticks*). Der Zeitraum zwischen zwei aufeinanderfolgenden *microticks* heißt **Granularität**.

Im folgenden bezeichnen wir mit $microtick_i^k$ den Mikrozeitschlag i einer Uhr k.

Wir nehmen weiter an, daß es eine Mutteruhr z mit einer Taktfrequenz f^z gibt, welche perfekt ist. $g^z = 1/f^z$ ist die Granularität der Mutteruhr. Wir nehmen an, daß f^z so groß ist, daß die Diskretisierung der Zeit der Mutteruhr im folgenden außer Acht gelassen werden kann.

Um festzustellen, wann Ereignisse stattgefunden haben, sehen wir vor, daß man mit Hilfe von Uhren den Events Zeitstempel geben kann. Mit $clock(e)$ bezeichnen wir den Zeitstempel eines Ereignisses e durch eine Uhr $clock$. $z(e)$ heißt **absoluter Zeitstempel**.

g^k ist die Anzahl der *microticks* von z zwischen zwei Zählereignissen von k. Wir nennen g^k die Granularität von k.

Die zeitliche Ordnung von zwei Ereignissen, die zwischen zwei *Microticks* von z stattfinden, ist grundsätzlich nicht zu bestimmen.

Die **Drift** einer Uhr k zum Zeitpunkt i ist bestimmt durch

$$drift_i^k = \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k}$$

wobei n_k die nominelle (d.h. die eigentlich gewünschte) Anzahl von *microticks* pro Takt der Uhr k ist. Die Driftrate ist definiert als:

$$\rho_i^k = |\frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} - 1|$$

ρ ist für praktische Uhren niemals 0.

Wir können zwischen zwei Fehlern realer Uhren unterscheiden (siehe auch Abb. 4.5):

1. Die Drift übersteigt die ursprünglich spezifizierte Drift.
2. Die Uhr liefert einen völlig falschen Zählerstand.

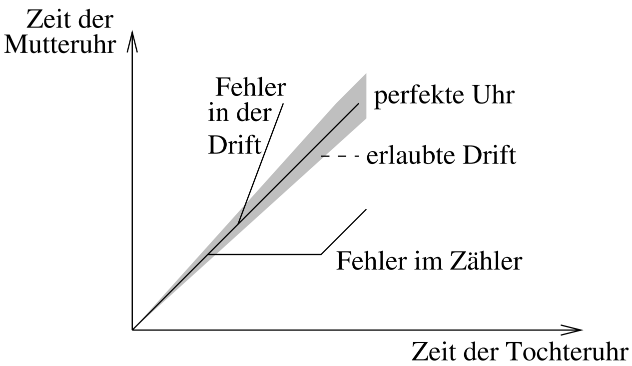


Abbildung 4.5: Fehler realer Uhren

Der **Offset** zweier Uhren j und k mit gleicher Granularität zum Zeitpunkt i ist definiert als

$$offset_i^{jk} = |z(microtick_i^k) - z(microtick_i^j)|$$

Sei eine Menge $\{1, ..., n\}$ an Uhren gegeben. Der Wert

$$\Pi_i = \max_{j,k}(offset_i^{jk})$$

heißt **Präzision** der Menge **zum Zeitpunkt** i. Das Maximum von Π_i über alle Zeiten in dem uns interessierenden Bereich heißt **Präzision**.

Der Offset einer Uhr k zur Mutteruhr beim *microtick* i heißt **Genauigkeit** von k **zum Zeitpunkt** i und wird mit $accuracy_i^k$ bezeichnet. Das Maximum dieses Wertes über die für uns relevanten i heißt **Genauigkeit**, in Zeichen: $accuracy^k$.

4.2.2 Tochteruhren

Wenn alle lokalen Uhren (Tochteruhren) perfekt synchronisiert wären, dann hätten wir (bis auf die Ungenauigkeit durch die Diskretisierung der Zeit) kein Problem, auf der Basis von Zeitstempeln kausale Abhängigkeiten auszuschließen. Leider läßt sich dies nicht erreichen.

Mit Kopetz wählen wir ersatzweise den folgenden Weg: Wir setzen voraus, daß wir lokale Tochteruhren haben, für welche die Abweichung von der Mutteruhr nicht größer ist als Π:

$$|z(microtick_i^j) - z(microtick_i^k)| < \Pi$$

Wir wählen dann eine Teilmenge der lokalen *microticks*, z.B. jeden zehnten *microtick*, als *macrotick* (siehe Abb. 4.6).

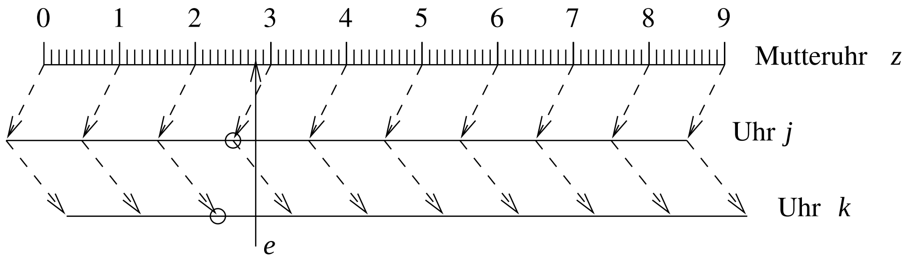


Abbildung 4.6: Zeitstempel eines einzelnen Ereignisses

Wir nennen ein System **vernüftig** (engl. *reasonable*), wenn die Granularität aller lokalen Uhren größer ist als Π. Wenn diese Bedingung erfüllt ist, dann unterscheiden sich die Zeitstempel eines Ereignisse höchstens um 1:

$$|t^j(e) - t^k(e)| \leq 1$$

Konsequenzen kann man anhand von Abb. 4.7 studieren.

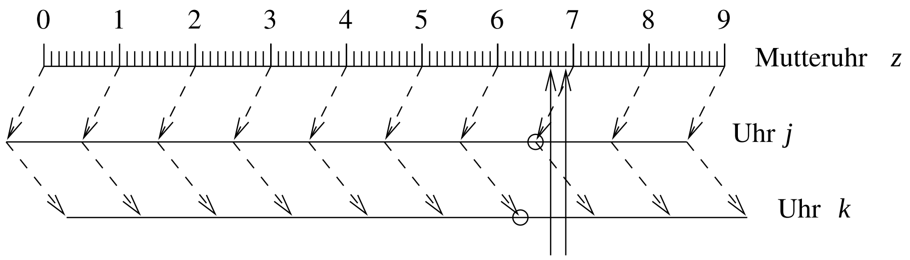


Abbildung 4.7: Zeitliche Ordnung zweier Ereignisse mit einer Zeitstempeldifferenz von 1

Das Ereignis zur absoluten Zeit 67 erhält von Uhr j den Zeitstempel 7, das Ereignis zur absoluten Zeit 69 von der Uhr k den Zeitstempel 6, obwohl es nach dem zuerst genannten Ereignis eingetreten ist. Aus der Zeitstempel-Differenz von 1 kann nicht auf die Reihenfolge geschlossen werden. Wenn die

Differenz allerdings mindestens 2 beträgt, kann auf die Reihenfolge geschlossen werden, weil die Summe von Synchronisierungs- und Diskretisierungsfehler stets kleiner als zwei ist.

Messung von Zeitintervallen

Ein Zeitintervall ist durch zwei Ereignisse definiert, dem Startereignis und dem Endereignis. Die Ungenauigkeit der Bildung der Differenz ist durch die Summe aus Synchronisations- und Diskretisierungsfehler begrenzt.

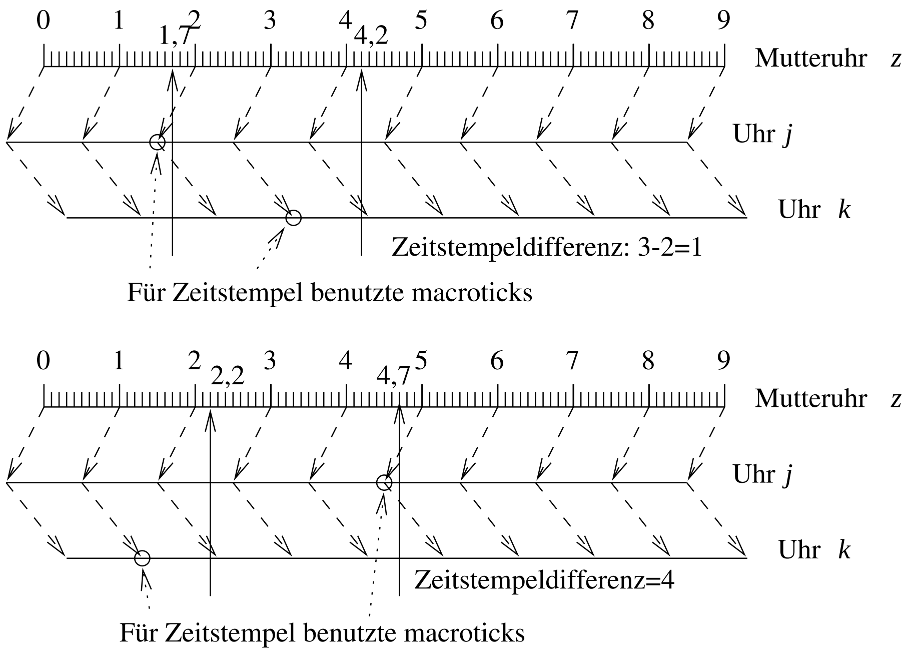


Abbildung 4.8: Fehler bei der Messung von Intervallen

Da 'vernünftige' Systeme vorausgesetzt werden, ist die Ungenauigkeit auf $< 2g$ begrenzt. Es folgt, daß für die wahre Länge eines Intervalls die folgende Ungleichung gilt:

$$(d_{obs} - 2g) < d_{true} < (d_{obs} + 2g)$$

wobei d_{obs} die Differenz der Zeitstempel ist.

π/Δ-Präzedenz

Man betrachte die Situation der Abbildung 4.9, in der Ereignisse im Abstand von Δ den Zeitstempeln verschiedener Tochteruhren versehen werden. Ein externer Beobachter sollte die Ereignisse innerhalb des kleinen Intervalls π nicht ordnen, wohl aber die Ereignisse im Abstand Δ.

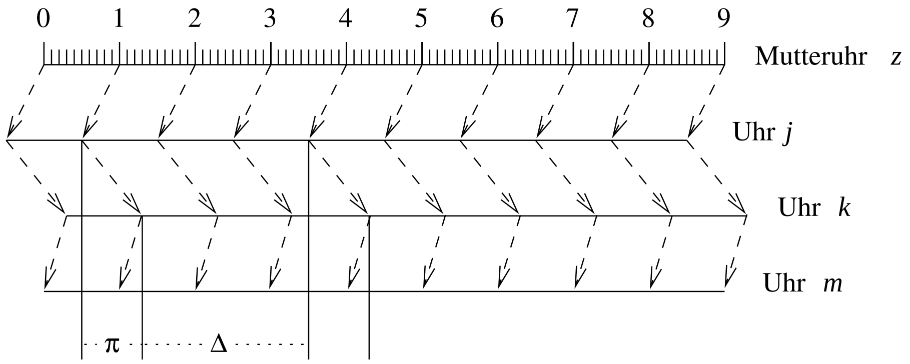


Abbildung 4.9: π/Δ-Präzedenz

Wie groß muß Δ sein, damit sich ein externer Beobachter tatsächlich so verhalten kann? Wir beschreiben diesen Mindestabstand mit Hilfe der sog. π/Δ-Präzedenz.

Def.: Gegeben seien eine Menge E von Ereignissen und zwei Zeitintervalle π und Δ, mit $\pi << \Delta$. Die Menge E heißt π/Δ-präzedent, wenn für alle Paare von Ereignissen gilt:

$$(|z(e_i) - z(e_j)| \leq \pi) \vee (|z(e_i) - z(e_j)| > \Delta)$$

Ein Beobachter kann sich korrekt entscheiden, welche Ereignisse zeitlich zu ordnen sind, wenn die Ereignismenge $0/3g$-präzedent ist, also für alle Uhren t gilt: $|t^j(e1) - t^k(e2) \geq 2$.

Dichte Zeit

Die Erkennung der zeitlichen Reihenfolge ist möglich, wenn Ereignisse nicht zu rasch nacheinander generiert werden. Eine derartige Kontrolle des Abstands ist innerhalb eines verteilten Systems möglich. Für Ereignisse, welche von außen kommen, ist dies nicht möglich. Abb. 4.10 zeigt ein Problem, welches entstehen kann, wenn Ereignisse von außen einen Abstand $< 3g$ haben.

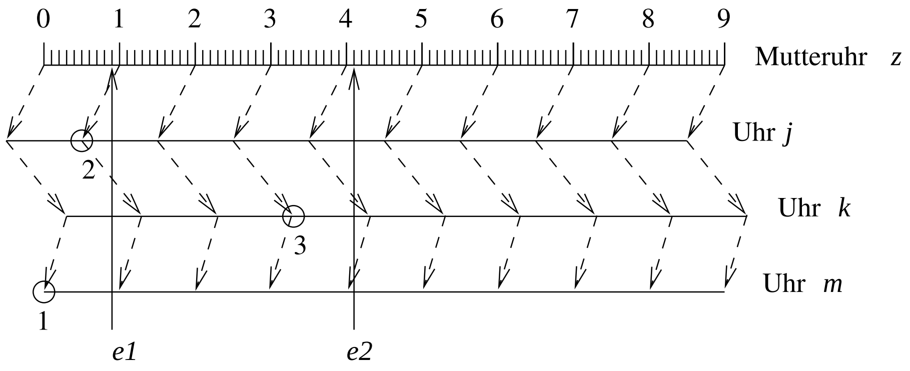


Abbildung 4.10: Beobachtung verschiedener Reihenfolgen von $e1$ und $e2$

Die Ereignisse haben einen Abstand von 3,2g. Das Ereignis $e1$ wird von j mit einem Zeitstempel von 2 gesehen, während m die Zeit 1 zuordnet. k beobachtet $e2$ zur Zeit 3 und teilt dies j und m mit. j berechnet eine Zeitdifferenz von 1 und schließt, daß eine zeitliche Ordnung nicht möglich ist. m berechnet eine Differenz von 2 und behauptet, $e2$ habe sich definitiv später als $e1$ ereignet. j und m kommen also zu inkonsistenten Schlüssen.

Um solche inkonsistenten Schlüsse zu vermeiden, kann ein *agreement*-Protokoll eingesetzt werden. In der ersten Phase eines solchen Protokolls teilen sich alle Knoten die jeweiligen Beobachtungen mit. In einer zweiten Phase werden konsistente Schlüsse berechnet. Die Ergenisse können in einer dritten Phase noch wieder verteilt werden, insbesondere, wenn nur ein Teil der Knoten selbst gerechnet hat.

Der Nachteil solcher *agreement*-Protokolle ist, daß sie sehr aufwendig sind und auch den Ablauf verzögern.

4.2.3 Interne Synchronisation von Uhren

Der Zweck der internen Synchronisation von Uhren ist, stets eine Präzision von Π für alle Uhren einzuhalten. Zu diesem Zweck kann man im Abstand von R_{int} eine Resynchronisation durchführen. Auch eine Resynchronisation ist eventuell nicht perfekt und führt noch zu einem Offset von Φ. Sei Γ der maximale Abstand zweier intakter Uhren voneinander unmittelbar vor der Resynchronisation. Es gilt:

$$\Gamma = 2\rho R_{int}$$

Damit die Präzision Π stets gewahrt bleibt, muß gelten (siehe Abb. 4.11):

$$\Phi + \Gamma \leq \Pi$$

"Byzantine Error"

Eine lokale Synchronisation erklärt beispielsweise den Durchschnitt der Zeiten der lokalen Uhren als neuen Richtwert der Zeit. Dies führt ggf. dann nicht zu einer hinreichenden Approximation der Zeit der

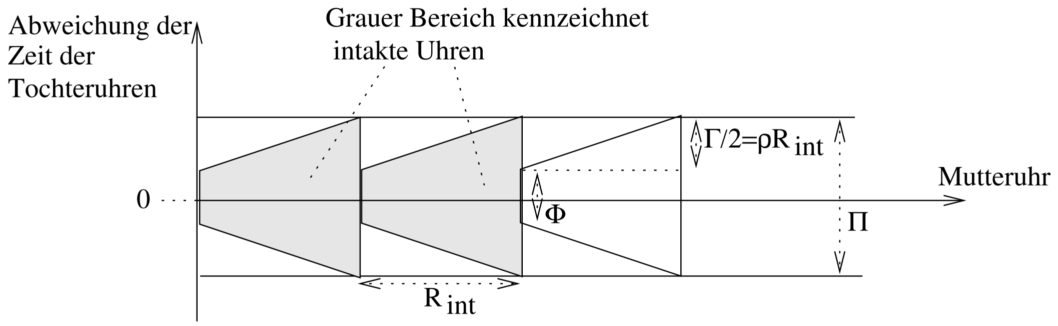


Abbildung 4.11: Synchronisationsbedingung

Mutteruhr, wenn lokale Uhren ein Fehlverhalten zeigen. Ein solches Fehlverhalten heißt *malicous error oder* Byzantino error. Es gibt spezielle Algorithmen, welche in Gegenwart derartiger Fehler fehlerhafte Uhren ausschließen. Diese Algorithmen sind aufgrund des Austauschs von Informationen aufwendig. Man kann zeigen [Kop97], daß Synchronisation in Gegenwart von *Byzantine error* nur garantiert werden kann, wenn mehr als $3k + 1$ lokaler Uhren vorhanden sind, wobei k die Anzahl derartig fehlerhafter Uhren ist.

Synchronisation mit einer zentralen Uhr

Zur Synchronisation kann ein zentraler Knoten periodisch Zeit-Meldungen an die übrigen senden. Die Differenz zwischen der lokalen Zeit und der Zeit in der Nachricht abzüglich der Laufzeit der Nachricht ergibt den Korrekturwert. Für diesen Algorithmus ist es wichtig, die Zeit möglichst genau zu kennen. Die mögliche Variation der Laufzeit heißt *latency jitter* ϵ. Für die Präzision des Algorithmus ergibt sich

$$\Pi_{zentral} = \epsilon + \Gamma$$

Der zentrale Algorithmus wird vielfach in der Startphase eines verteilten Systems benutzt.

Verteilte Synchronisation

Verteilte Synchronisationsverfahren benutzen typischerweise drei verschiedene Phasen:

1. In der ersten Phase sammeln alle Knoten Informationen von den übrigen Knoten
2. In der zweiten Phase werden aus diesen Informationen Korrekturwerte für die jeweilige lokale Uhr berechnet.
3. Schließlich werden die Korrekturen durchgeführt.

Vorhandene Algorithmen unterscheiden sich in der Art, in der diese drei Schritte realisiert werden. Zu 1: Die Genauigkeit der ersten Phase, d.h.E., hängt sehr stark davon ab, auf welcher Systemebene diese Informationen ausgetauscht werden. Auf normaler Prozeßebene ist der Jitter durch den Einfluß des Schedules besonders groß (siehe Tabelle 4.1).

Ebene	Jitter (ca.)
Anwendungsebene	500 μsec. bis 5 msec
Kern des Betriebssystems	10 μsec bis 100 μsec
Hardware des Kommunikationscontrollers	$< 10\mu$sec

Tabelle 4.1: Größenordnung des Jitters

Zu 2: In Gegenwart von *Byzantine errors* kann man die Zeiten aller Uhren sortieren. Die k größten und die k kleinsten Werte werden aus dieser Liste entfernt. Der Mittelwert über die verbleibenden Werte ergibt die neue Zeit.

Zu 3: Die Zeitkorrektur kann entweder durch direktes Setzen des neuen Wertes oder durch Anpassen der Geschwindigkeit des lokalen Oszillators geschehen. Bei der ersten Methode können poblematische Sprünge der lokalen Zeit entstehen. Bei der zweiten Methode sollte die durchschnittliche Korrektur null sein, sonst würden die Uhren immer schneller.

4.2.4 Externe Synchronisation

Interne Synchronisation kann nicht garantieren, daß die Zeit in verteilten System der physikalischen Zeit entspricht. Um dies zu erreichen, kann man Verbindungen zu externen Zeitstandards nutzen.

In der jüngsten Zeit erfolgte vielfach ein Übergang auf GPS *(Global Positioning System)* als Zeitreferenz (siehe Abb. 4.12).

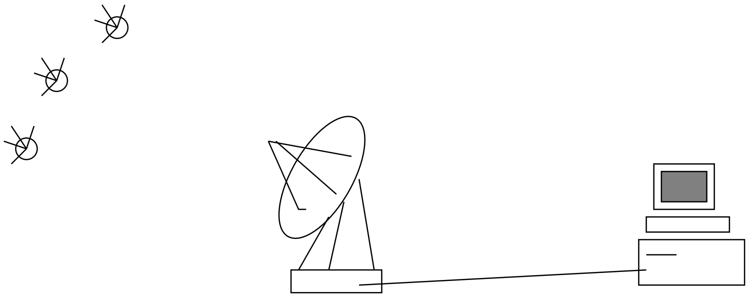


Abbildung 4.12: Informationsfluß bei externer Synchronisation

GPS sendet Informationen sowohl über die TAI wie auch die UTC-Zeitskala. GPS bietet eine zeitliche Genauigkeit von ca. 100 nsec. Die Verfügbarkeit von GPS hat umfangreiche Arbeiten stimuliert, welche auf eine Ausnutzung der technischen Möglichkeiten zielen [Sch97a, SSH97]. So beschäftigen sich allein drei Ausgaben der Zeitschrift *real-time-systems* des Jahres 1997 exklusiv mit diesem Thema. Externe Synchronisation ist ein unidirektionaler Vorgang. Lokale Systeme übernehmen ganz einfach die Zeit, die ihnen vom externen Zeitstandard übermittelt wird. Aus Sicht der Fehlertoleranz ist dieser unidirektionale Vorgang ein potentielles Problem: sofern der externe Zeitstandard mit Fehlern behaftet ist, werden diese zu allen Knoten kopiert. Als Sicherheitsmaßnahme werden von allen Knoten nur kleine Änderungen ihrer Zeit akzeptiert. Externe Zeitformate leiden leider auch unter der Tendenz, für Zeitformate nicht ausreichend Kapazität bereit zu stellen. Das standardisierte NTP-Protokoll sieht leider nur einen Jahresbereich bis zum Jahr 2036 vor (siehe Abb. 4.13)

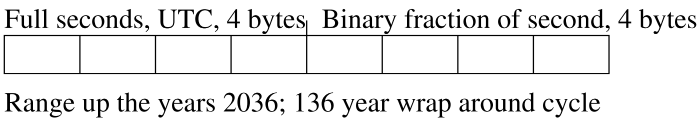


Abbildung 4.13: Time format in the Network Time Protocol (NTP)

4.3 Beispiele von Realzeit-Betriebssystemen

- **OS/9**

  OS/9 ist ein BS mit Unix-ähnlichem Filesystem, welches auf verschiedenen Prozessoren (z.B. auf dem Motorola 6800) ablauffähig ist. OS/9000 ist eine neuere Variante davon.

- **Realzeitkerne in Kombination mit Windows oder DOS**

  PCs werden aufgrund ihres Preises und der Vertrautheit der Benutzer auch für für Prozeßsteuerungen zunehmend beliebter. Allerdings erfüllen Betriebssysteme wie Windows 95 die o.a. Anforderungen nicht. Daher müssen zusätzliche Realzeitkerne in das an sich schon vorhandene Betriebssystem integriert werden. Dies verursacht hohe Kosten und Wartungsprobleme.

  Wrobel beschreibt in [Wro96] das von der Fa. Siemens entwickelte RMOS3. RMOS3 besitzt volle Kontrolle über die Hardware und führt Windows 3.11 als einen Prozeß niedriger Priorität aus. Abstürze von Windows 3.11 führen damit nicht zu Problemen der übrigen Prozesse. Diese können weiterhin ausgeführt werden, während Windows 3.11 nei gebootet wird.

- **VxWorks**

  VxWorks ist ein Betriebssystem der auf Realzeit-Systeme spezialisierten Firma *Wind River Systems* [Sys].

- **Windows CE**

  Windows CE soll für den Markt der eingebetteten Syteme geeignet sein. Die Realzeitfähigkeit ist aber für die bisherigen Versionen nicht gegeben. Aber Bill Gates möchte auch in diesen wachsenden Markt dominieren ....

- **Java** [Sun]

  Mit der Möglichkeit der Verwendung mehrerer Threads erfüllt Java eine der Kernanforderungen an Progammiersprachen für eingebettete Realzeitsysteme. Auch enthält der Java-Interpreter bereits einen *run-time scheduler* zur Umschaltung zwischen Threads. Üblicherweise setzt Java auf einem vorhandenen Betriebssysstem auf. Allerdings versucht man auch, Java als *stand-alone*-System zu betreiben. Zu diesem Zweck werden spezielle Versionen von Java entwickelt, die nicht zur Benutzung mit Terminal und Tastatur gedacht sind. Auf die entsprechenden Pakete wird verzichtet.

  Das folgende Zitat [Zei98] beschreibt die aktuell diskutierten Varianten:

  “Genau diese Palette soll Java noch in diesem Jahr abdecken. Dazu werden aus der bisherigen Entwicklungsplattform, dem Java Developement Kit (JDK), einerseits spezielle Bestandteile heruntergebrochen, andererseits Elemente hinzugefügt. In drei grobe Bereiche läßt sich Java künftig einteilen: eine Sprache für Geräte ohne grafische Ausgabe, in eine für Clients und in eine für Server. In den ersten Bereich fällt Cardjava und Ejava. Cardjava ist mit 20 Kilobyte Code ein extrem kleines Subset des JDK, das für den Einsatz in Smartcards konzipiert wurde. Ein besonderes Augenmerk bei Cardjava wurde auf Verschlüsselungsalgorithmen und Funktionen für die Zugangskontrolle gelegt.

  Mercedes setzt in der E-Klasse auf Ejava

  Für Embedded-Devices hat Sun Ejava entworfen. Hierzu zählen neben Controllern in Telefonen und Set-top-boxes auch besonders Steuerungsrechner in Fahrzeugen. Die Idee von Ejava ist, die bisher meist in Hardware gegossenen Funktionen der Embedded Systeme zu flexibilisieren und damit die Time-to-market zu beschleunigen. Die entsprechend erweiterten Funktionen entstanden in enger Zusammenarbeit mit Unternehmen wie etwa Daimler-Benz, Ericsson und Siemens. Genau wie bei Cardjava sind bei Ejava europäische Firmen tonangebend.”

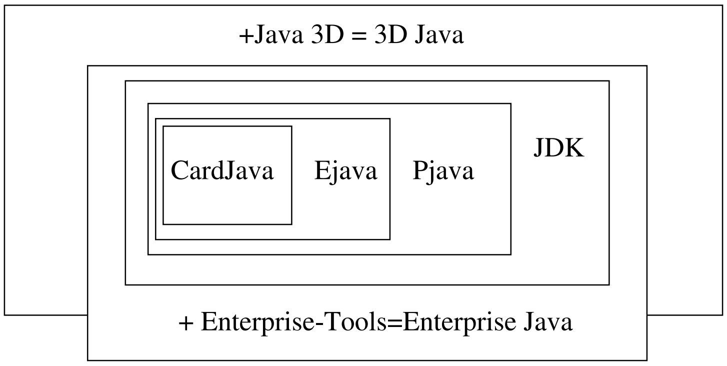


Abbildung 4.14: Java-Subsets

Allerdings muß Java auch ergänzt werden. Folgendes muß bei Realzeit-Java beachtet werden (siehe auch [PMP+98])

1. Die automatische dynamische Speicherverwaltung von Java führt zu unvorhersehbaren Verzögerungen. Für Realzeit-java sind diese Verzögerungen zu vermeiden.
2. Der eingebaute *run-time scheduler* führt zu nicht genau vorhersehbaren Prozeßumschalttungen. Er muß durch einen speziellen Realzeit-Scheduler ersetzt werden.

Kapitel 5

Realzeit-Sprachen

10. Vorl.

5.1 Anforderungen an Realzeit-Sprachen

Folgende Anforderungen werden an Sprachen zur Realisierung von Realzeit-Systemen gestellt (siehe auch Burns [BW90], Marwedel [Mar98]):

1. **Beschreibung nebenläufiger Prozesse**

   Bei der Spezifikation vieler eingebetteter Realzeit-Systeme ist eine Beschreibung in Form von nebenläufigen Prozessen die geeigneste. So läßt sich die Reaktion auf bestimmte externe Alarmmeldungen und die Erzeugung von Ausgaben an ein bestimmtes Gerät meist auf 'natürliche Weise' jeweils in Form eines Prozesses spezifizieren.

2. **Verläßlichkeit**

   Gerade eingebettete Systeme müssen ein hohes Maß an Sicherheit gegen Fehlverhalten und Manipulation durch Dritte bieten. Aus diesem Grund sollten als Minimalforderung nur verhältnismäßig sichere Sprachen (und insbesondere keine Assemblersprache), Hardware-Zugriffsschutz, ggf. Verschlüsselung usw. vorhanden sein.

   Aus Gründen der Sicherheit gegen Hardwarefehler wird man insbesondere bei sicherheitskritischen Anwendungen Fehlertoleranzmaßnahmen vorsehen. Der Bereich der Maßnahmen reicht von einfachen Paritätsbits über Fehlererkennende und -korrigierende Codes bis hin zur Dreifachauslegung des Systems und Mehrheitsentscheid (engl. *triple modular redundancy*).

3. **Ausnahmebehandlung** und *error recovery*

   Im Falle eines Fehlers in einem eingebetteten System ist es wenig hilfreich und meist sogar unmöglich, als einzige Reaktion eine Ausgabe auf einem Bildschirm zu erzeugen. Das System sollte selbstständig eine angemessene Reaktion auf einen Fehler einleiten können. Beispielsweise könnte ein Programmsegment beim Auftreten eines internen flüchtigen Hardwarefehlers einfach wiederholt werden.

4. **Synchronisation nebenläufiger Prozesse**

   Es gibt im wesentlichen zwei Gründe für die Synchronisation von Prozessen:

   - Die Prozesse benötigen Betriebsmittel, auf die ein exklusiver Zugriff gewährleistet sein muß. Es werden Hilfsmittel benötigt, die einen solchen exklusiven Zugriff garantieren.
   - Die Prozesse müssen darauf warten, bis andere Prozesse einen bestimmten Betriebszustand erreicht haben.

5. **Nachrichtenaustausch nebenläufiger Prozesse**

   Der Nachrichtenaustausch zwischen Prozessen für einbettete System extrem wichtig, da Meßgrößen, Operateurseingriffe usw. vielfach verschiedenen Prozessen mitgeteilt werden müssen.

6. **Ereignisbehandlung**

Wir haben schon im Kapitel 4.1 gesehen, daß die Steuerung des Gesamtsystems über Ereignisse sinnvoll sein kann. Hierfür müssen auch die erforderlichen programmiersprachlichen Konstrukte vorhanden sein. Beispielsweise sollte es Möglichkeiten geben, durch Interrupts direkt Prozesse starten oder beenden zu können.

7. **Definition und Prüfung von Zeitschranken**

   Schon der Name 'Realzeit-Systeme' läßt erkennen, daß die Zeit in den von uns betrachteten Systemen eine große Rolle spielt. Es muß dementsprechend möglich sein, zeitliche Anforderungen und ggf. auch Zusicherungen spezifizieren zu können.

8. **Zustandorientiertes Verhalten**

   Eingebettete Realzeit-Systeme zeigen üblicherweise ein sog. **reaktives Verhalten**. Dies bedeutet, daß sie Ausgaben aufgrund der aktuellen Eingaben und des aktuellen Zustandes generieren und aufgrund dieser beiden Informationen in einen neuen Zustand übergehen. Folglich ist das Automatenmodell ein geeignetes Modell des Systems.

9. **Fähigkeit zur Behandlung von Non-Standard E/A-Geräten**

   Eingebettete Systeme benutzen vielfach spezielle Sensor/Aktor-Schnittstellen, die nicht in das Schema üblicher Block- und Byte-Geräte mit möglicher Wiederholung von Lese- und Schreibvorgängen, Pufferung usw. passen. Daher muß die E/A-Programmierung entsprechend angepaßt sein.

10. **Effizienz**

    Mehrfach haben wir bereits davon gesprochen, daß eingebettete Systme effizient sein müssen. Dementsprechend werden beispielsweise hochgradig optimierende Übersetzer von der Speizifikation in die Maschinensprache benötigt.

11. **Unterstützung des Entwurfs großer Systeme**

    Um eingebettete Realzeitsysteme mit umfangreicher Software entwerfen zu können, müssen alle Anforderungen an den Entwurf großer Softwaresysteme auch in unserem Kontext eingehalten werden. Dies bedeutet heutzutage, daß solche Systeme mittels objektorientierter Methoden entwickelt werden sollten. Diese Anforderung macht auch die Unterstützung hierarchischer Spezifikationen erforderlich.

12. **Unterstützung spezifischer Eigenheiten von Anwendungsbereichen**

    Spezielle Anwendungsbereiche besitzen Besonderheiten, die zumindest dann berücksichtigt werden müssen, wenn effiziente Systeme zu generieren sind. So erfordern DSP-Applikationen beispielsweise eine Unterstützung von verzögerten Signalen. Eine einfache Realisierung von Signalen mit Hilfe von Arrays würde extrem viel Speicher kosten bzw. bei unendlichen Signalen überhaupt nicht möglich sein.

13. **Spezifikation nicht-funktioneller Systemeigenschaften**

    Neben den rein funktionellen Eigenschaften müssen eingebettete Systeme auch eine Vielzahl anderer Eigenschaften aufweisen. Genannt seien beispielsweise: die elektromagnetische Verträglichkeit, die Zuverlässigkeit, die Fehlertoleranz, das Gewicht, die Stromaufnahme, die Schock-Widerstandsfähigkeit, die Entsorgbarkeit am Ende der Produktlebensdauer, der zulässige Temperaturbereich u.v.a.m.

14. **Lesbarkeit**

    Diese Forderung dürfte selbstverständlich sein.

15. **Portierbarkeit**

    Portierbarkeit ist erforderlich, um rasch auf andere Plattformen umstellen zu können. Wegen der großen Zahl möglicher Plattformen ist dies wichtiger als bei PC-Software.

16. **Flexibilität**

    Es sollte relativ leicht möglich sein, Spezifikationen zu ändern und daraus wieder Realisierungen abzuleiten. Dies führt zu deutlichen Problemen für die Realisierung mit Spezialhardware.

5.2 Realisierung der Anforderungen

5.2.1 Nebenläufige Prozesse

5.2.1.1 Konzepte für nebenläufige Prozesse

In den vorhandenen Sprachen werden unterschiedliche Prozeßkonzepte realisiert. Zwei Unterscheidungsmerkmale sind die folgenden:

1. **Anzahl der Prozesse**
   - **Statisch**
     In vielen Sprachen ist die Anzahl der Prozesse zur Compilezeit bekannt und fest.
   - **Dynamisch**
     In anderen Sprachen können Prozesse zur Laufzeit dynamisch erzeugt werden.
2. **Verschachtelung**
   - **verschachtelt**: innerhalb jedes Prozesses können weitere erzeugt werden.
   - **flach**: Prozesse sind auf der äußersten Programmebene zu definieren.

Tabelle 5.1 zeigt einen Vergleich verschiedener Sprachen:

Sprache	Anz. der Prozesse	Verschachtelung
concurrent PASCAL	statisch	flach
VHDL	statisch	flach
SDL	dynamisch	flach
Modula-2	dynamisch	flach
occam	statisch	verschachtelt
ADA	dynamisch	verschachtelt
C	dynamisch	verschachtelt
Java	dynamisch	verschachtelt

Tabelle 5.1: Prozeßkonzepte in verschiedenen Sprachen

Weiter kann man noch unterscheiden, ob Prozessen bei deren Initialisierung Parameter mitgegeben werden können oder nicht.

Zusätzlich kann man noch betrachten, unter welchen Bedingungen Prozesse terminieren können: am Ende der Ausführung des Rumpfes, durch 'Selbstmord' [BW90], durch Abbruch aus einem anderen Prozeß heraus, aufgrund einer nicht abgefangenen Fehlerbedingung, niemals oder wenn nicht mehr benötigt.

Schließlich gibt es noch verschiedene Möglichkeiten, Nebenläufigkeit zu beschreiben:

1. **Koroutinen**

   Koroutinen sind ähnlich wie Prozeduren aufgebaut. Sie erlauben es jedoch, die Kontrolle von einer Koroutine explizit an eine andere zu übergeben. Diesem Zweck dient die `resume`-Anweisung. Wenn eine Koroutine eine solche Anweisung ausführt, so wird sie angehalten, behält aber alle Zustandsinformation. Wenn eine andere Prozedur für sie ein `resume` erzeugt, so wird sie wieder ausgeführt. Jede Koroutine kann als ein Prozeß gesehen werden. Allerdings kann aufgrund der beschriebenen Semantik immer nur eine Koroutine ausgeführt werden.

2. **Fork and join**

   Die `fork`-Anweisung erzeugt einen neuen Prozeß. Mittels der `join`-Anweisung kann ein Prozeß auf die Beendigung eines anderen Prozesses warten. Beispiel [1]:

[1] Nachfolgend werden alle reservierten Worte groß geschrieben, auch wenn dies in der konkreten Sprache nicht erlaubt sein sollte.

```
FUNCTION f RETURN ...;
PROCEDURE p;
...
c:= fork f;
....
j:=join(c);
....
END p
```

`fork` und tt join erlauben die dynamische Erzeugung von Prozessen. Mittels Parametern kann man den Prozessen Informationen mitgeben. `fork` und `join` sind flexibel, erlauben aber keinen strukturierten Entwurf von Mehrprozeßsystemen.

3. `COBEGIN`

   `COBEGIN` ist eine strukturierte Methode der Erzeugung von Prozessen. Beispiel:

```
COBEGIN
 s1;
 s2;
 s3;
 s4;
COEND
```

   Die Statements `s1,s2,s3,s4` werden nebenläufig ausgeführt. Die `COBEGIN`-Anweisung terminiert, wenn alle Teilprozesse terminieren. Die Teilprozesse können selbst wieder andere Prozesse aufrufen.

4. **Explizite Erzeugung von Prozessen**

   Programme können klarer gestaltet werden, wenn man Prozesse explizit erzeugen kann. Das folgende Beispiel zeigt dies. Es sind zwei Meßsensoren zu überwachen, von denen angenommen wird, daß sie jeweils bei Änderung um 1% gegenüber dem bisherigen Wert einen neuen Ausgabewert generieren und Prozeduren `new_temperature` bzw. `new_humidity` blockieren, bis ein solcher Wert vorliegt. Nach einer Vorverarbeitung seien diese Werte zusammen mit einer Identifikation des Sensors in einen Ausgabepuffer zu schreiben (siehe Abb. 5.1).

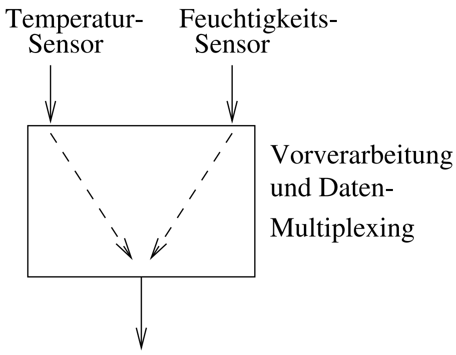


Abbildung 5.1: Einfache Meßwerterfassung mit Weiterleitung über eine gemeinsame Leitung

Eine einfache Realisierung könnte in einer Endlosschleife bestehen (in Modula/PASCAL-artiger Notation):

```
MODULE main;
 TYPE some_channel = (temperature, humidity);
      some_sample : RECORD
                     value : integer;
                     line  : some_channel
                    END;
```

```
 ..
 LOOP
   sample.value := new_temperature; (* blockiert, bis neuer Wert eingelesen *)
   IF sample.value > 30 THEN ....
   sample.line := temperature;
   to_fifo(sample);
   sample.value := new_humidity;    (* blockiert, bis neuer Wert eingelesen *)
   sample.line := humidity;
   to_fifo(sample);
 END;
```

Bei dieser Lösung wird muß die Anzahl der Temperaturwerte gleich der Anzahl der Feuchtigkeitswerte sein. Dies wird in der Regel nicht der Aufgabenstellung entsprechen.

In einem zweiten Versuch könnten wir eine Statusanzeige benutzen, die uns eine Abfrage bei aktiven Warten (engl. *busy waiting*) erlaubt:

```
..
LOOP
 IF changed_temperature THEN
  BEGIN
   sample.value := new_temperature; (* Rückgabe eines neuen Wertes*)
   IF sample.value > 30 THEN ....
   sample.line := temperature;
   to_fifo(sample);
  END;
 IF changed_humidity THEN
  BEGIN
   sample.value := new_humidity;    (* Rückgabe eines neuen Wertes*)
   sample.line := humidity;
   to_fifo(sample);
  END;
END;
```

Auch diese 'Lösung' kommt kaum ernsthaft in Betracht, da sie ständig Rechenzeit verbraucht.

Alternativ zu einem `wait`-Befehl mit einer Liste von Weck-Ereignissen kann diese Aufgabe mit Hilfe mehrerer Prozesse beschrieben werden:

```
MODULE main;
 TYPE some_channel = (temperature, humidity);
      some_sample : RECORD
                     value : integer;
                     line  : some_channel
                    END;
 PROCESS get_temperature;
 VAR sample : some_sample;
 BEGIN
  LOOP
   sample.value := new_temperature; (* blockiert, bis neuer Wert eingelesen *)
   IF sample.value > 30 THEN ....
   sample.line := line;
   to_fifo(sample);
  END
 END get_temperature;

 PROCESS get_humidity;
 VAR sample : some_sample;
 BEGIN
  LOOP
```

```
      sample.value := new_humidity;    (* blockiert, bis neuer Wert eingelesen *)
      sample.line := line;
      to_fifo(sample);
     END
    END  get_humidity;

   BEGIN
     get_temperature; get_humidity;
   END;
```

Ein verbleibendes Problem ist der Zugriff von Prozessen auf gemeinsame Ressourcen. Nicht immer kann man Meßwerte wie oben angegeben in einem gemeinsamen FIFO abspeichern. Im nächsten Abschnitt (siehe Seite 74) werden wir Lösungen dafür vorstellen.

5.2.1.2 Nebenläufige Prozesse in ADA

Aufgrund der Unterstützung der Nebenläufigkeit ist ADA für die Programmierung von Echtzeitsystemen relativ popolär. In ADA heißen Prozesse **Tasks**. Tasks können auf jeder Programm-Ebene deklariert werden. Sie werden implizit gestarted, wenn die Kontrolle in den Scope eintritt. Beispiel [BW90]:

```
PROCEDURE example1 IS
 TASK a;
 TASK b;
 TASK BODY a IS
  -- lokale Deklarationen von a
 BEGIN
  -- Anweisungen von a
 END a;
 TASK BODY b IS
  -- lokale Deklarationen von b
 BEGIN
  -- Anweisungen von b
 END b;

 BEGIN
  -- Tasks a und b starten vor der Ausführung der ersten Anweisung
  -- des Rumpfes von example1
 END;
```

5.2.1.3 Nebenläufige Prozesse in Java

Aufgrund der Unterstützung der Nebenläufigkeit wird Java für die Programmierung von Echtzeitsystemen derzeit intensiv diskutiert. Java unterstützt leichtgewichtige Prozesse, *threads* (deutsch: **Fäden**) genannt. Fäden sind Teile eines normalen Prozesses. Prozesse sind über eigene Adreßräume voneinander abgeschottet. Alle Fäden eines Prozesses werden im Unterschied dazu in demselben Adreßraum ausgeführt.

Zur Implementierung von Threads gibt es in Java die Klasse `java.lang.Thread`, nachfolgend auch kurz `Thread` genannt. Eigene Anwendungen von *threads* können auf zwei Arten realisiert werden [Job96]:

1. **als Unterklasse von `Thread`**

   Wenn in einem Programm eine Klasse v.a. die Eigenschaft hat, *thread* zu sein, dann sollte sie als Unterklasse von `Thread` realisiert werden.

2. **durch Implementierung der sog. `Runnable`-Schnittstelle**

   Sofern ein Objekt zu modellieren ist, welches ganz wesentlich Unterklasse einer anderen Klasse sein sollte (also zum Beispiel einer GUI-Klasse), so kann dieses Objekt nicht zugleich Unterklasse von `Thread` sein, denn in Java gibt es keine Mehrfachvererbung. Aus diesem Grund ist ein *applet* kein *thread*, aber es kann sich wie ein solches verhalten.

Das nachfolgende Beispiel zeigt diese beiden Arten der Realisierung eines *thread*-artigen Verhaltens:

```
CLASS ErsterThread EXTENDS Thread {
 PUBLIC VOID run () {
  FOR (int i = 0; i < 10; i++)
   TRY {
    sleep (Math.round (1000.0 * Math.random ()));
    System.out.println (toString () + " " + i);
    }
   CATCH (InterruptedException e) {
    System.out.println (e);
   }
  }
}
CLASS ZweiterThread IMPLEMENTS Runnable {
 Thread thread;
 ZweiterThread () {
  thread = NEW Thread (this);
 }
 PUBLIC VOID start () {
  thread.start ();
 }
 PUBLIC VOID join() THROWS InterruptedException {
  thread.join ();
 }
PUBLIC VOID run () {
 FOR (INT i = 0; i<10; i++)
  TRY {
   thread.sleep (Math.round (1000.0*Math.random ()));
   System.out.println (thread.toString () + " " + i);
  }
  CATCH (InterruptedException e) {
   System.out.println (e);
  }
 }
}

PUBLIC CLASS ThreadDemo {
 STATIC PUBLIC VOID main (String args[]) {
  ErsterThread thread1 = NEW ErsterThread () ;
  thread1.start ();
  ZweiterThread thread1 = NEW ZweiterThread () ;
  thread2.start ();
  TRY {
   thread1.join () ; // Warte auf den 1. Thread
   thread2.join () ; // Warte auf den 2. Thread
  }
  CATCH (InterruptedException e) {
  }
 }
}
```

Bei `ErsterThread` wird die `run`-Methode von `Thread` überschrieben. `ZweiterThread` implementiert eine `Runnable`-Schnittstelle. Da `ZweiterThread` kein *thread* ist, wird eine Instanz von `Thread` benötigt. Da `ZweiterThread` kein *thread* ist, fehlen ihm die Methoden wie `start` und `join`

Das Hauptprogramm startet die beiden *threads*. Aufgrund des Aufruf der Zufallsfunktion werden die beiden *threads* jeweils eine zufällige Anzahl von Malen ausgeführt, bevor auf den jeweils anderen *thread* umgeschaltet wird. `main` wartet auf das Ende der beiden *threads*.

5.2.2 Verläßlichkeit

11. Vorles.

Unter dem Begriff der **Verläßlichkeit** (engl. *dependability*) faßt man alle Eigenschaften eines Systems zusammen, welche sicherstellen, daß sich ein System wie erwartet verhält:

Def. (Laprie, 1985): The dependability of a system is that property which allows reliance to be justifiably placed on the service it delivers.

Verläßlichkeit (*dependability*) umfaßt die älteren Begriffe der *reliability* (deutsch **Zuverlässigkeit**), der *security*) und der *safety* (siehe Abb. 5.2).

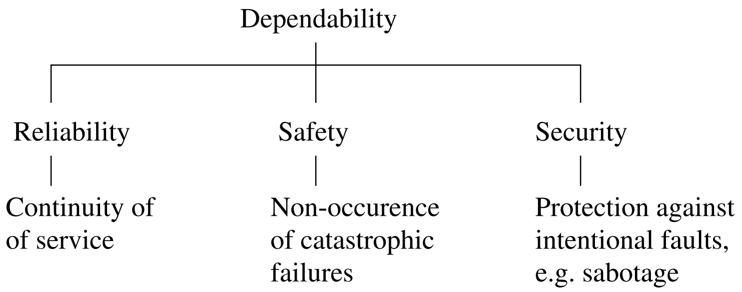


Abbildung 5.2: Aspekte der Verläßlichkeit [BW90]

Die Zuverlässigkeit berücksichtigt insbesondere die reinen Hardwarefehler, aber auch Fehler in der Software.

Ein System ist *safe*, falls es bei Abwesenheit von Sabotage nicht zu katastrophalen Fehlern kommen kann. Beispielsweise muß ein solches System sicherstellen, daß ein Air-Bag auch dann nicht versehentlich ausgelöst wird, wenn ein Prozessor einen Hardwarefehler hat (in der Technik heißen solche Systeme häufig auch **eigensicher**).

Security beinhaltet zusätzlich den Schutz gegen böswillige Angreifer, Hacker etc.

Eine Grundanforderung an Sprachen zur Programmierung verläßlicher Systeme ist die Möglichkeit, explizit auf **Ausnahmen** (engl. *exceptions*) reagieren zu können.

In ADA können für Blöcke, Unterprogramme, Pakete und Prozesses Ausnahmen und die Reaktion darauf angegeben werden. Beispiel:

```
BEGIN
 sequence of statements
EXCEPTION
 exception handler
END;
```

Der Entwurf verläßlicher Systeme besitzt viele weitere Facetten, die wir hier nicht alle aufzeigen können.

5.2.3 Synchronisation und Kommunikation

5.2.3.1 Unteilbarkeit

Vielfach ist es erforderlich, eine Menge von Operationen entweder ganz oder gar nicht auszuführen. Dieses Phänomen ist insbesondere auch von Datenbanken her bekannt. Aus diesem Grund muß es möglich sein, Mengen von unteilbaren Operatione zu identifizieren.

Unteilbarkeit (Atomizität) kann beispielsweise auf einem Ein-Prozessorsystem durch Blockierung der Interrupts realisiert werden, wobei eine solche Realisierung ggf. zu Lasten der Reaktionszeit gehen kann.

Explizite Operationen zur Sicherstellung der Atomizität kann man sich dann sparen, wenn ein Dispatcher während der Bearbeitung unteilbaren Codes ohnehin nicht auf einen anderen Prozeß umschalten wird.

5.2.3.2 Gemeinsamer Speicher

Der Austausch von Informationen zwischen Prozessen kann im wesentlichen über zwei Mechanismen geschehen, nämlich über das Konzept des **gemeinsamen Speichers** (engl. *shared memory*) sowie über *message passing*.

Durch den ungeregelten Zugriff auf gemeinsame Daten kann es zu Anomalien kommen, wenn beispielsweise Prozessen nach einem unvollständigen Update einer globalen Datenstruktur der Prozessor entzogen wird oder ein anderer Prozeß zeitlich verzahnt ebenfalls auf dieselbe globale Struktur zugreift. Aus diesem Grund muß der Zugriff auf gemeinsame Daten geregelt werden.

Hierfür sind im Zusammenhang mit Betriebssystemen schon vor vielen Jahren Techniken entwickelt worden. Bekannt sind u.a.

- **Semaphore**
- **Conditional critical regions**
- **Monitore**

  Monitore bestehen aus:

  - Daten, die nur von Zugriffsroutinen aus bearbeitet werden können
  - Zugriffsroutinen
  - genau definierten Zugängen zum Monitor
  - sog. *condition*-Variablen

  Zu jedem Zeitpunkt kann sich höchstens ein Prozeß im Monitor 'befinden' (d.h. Befehle für sich ausführen lassen). Damit kann der gegenseitige Ausschluß sichergestellt werden. Die *condition*-Variablen dienen zum Warten auf das Eintreten von Ereignissen und zum Benachrichtigen von Prozessen.

  Ein Problem der Verwendung von Monitoren liegt in der nicht erlaubten Schachtelung: Wenn ein Monitor eine Prozedur ruft, die wiederum zum Monitoraufruf führen kann, so hat man ein Dilemma.

Monitore sind auch in Java realisiert. Klassen können mit dem Modifizierer `SYNCHRONIZED` versehen werden. Für eine solche Klasse erzeugt das Laufzeitsystem einen Monitor. Beispiel [Job96]:

Falsch:

```
CLASS Punkt {
 FLOAT x, y;
 Punkt (FLOAT x, FLOAT Y) {
  THIS.x = x;
  THIS.y = y;
 }
 VOID setzePunkt (FLOAT x, FLOAT y) {
  THIS.x = x;     // Der Vorgang könnte hier in
  THIS.y = y;     // einem Thread unterbrochen werden
 }
 Punkt liesPunkt () {
  RETURN NEW Punkt (x,y);
 }
}
```

Richtig:

```
CLASS Punkt {
 FLOAT x, y;
 Punkt (FLOAT x, FLOAT Y) {
  THIS.x = x;
  THIS.y = y;
 }
```

```
  SYNCHRONIZED VOID setzePunkt (FLOAT x, FLOAT y) {
   THIS.x = x;     // Der Vorgang könnte hier in
   THIS.y = y;     // einem Thread unterbrochen werden
  }
  SYNCHRONIZED Punkt liesPunkt () {
   RETURN NEW Punkt (x,y);
  }
 }
```

5.2.3.3 Nachrichtenaustausch

Als Alternative zum Modell des gemeinsamen Speichers kann das Modell des Nachrichtenaustauschs (engl. *message passing*) benutzt werden. Man beachte, daß es sich bei dem Begriff des Nachrichtenaustauschs um die Sicht des Programmierers handelt. Nicht immer muß dem Nachrichtenaustausch ein Versenden von Nachrichten zugrunde liegen, und schon gar nicht das Senden an einen anderen Rechner. Das Modell des Nachrichtenaustauschs kann auch auf einer Hardware realisiert werden, die üner gemeinsamen Speicher verfügt und umgekehrt kann das Programmier-Modell des gemeinsamen Speichers auf einer Hardware realiert werden, welche hierfür Nachrichten austauschen muß.

Konkrete Realisierungen dieses Modells unterscheiden sich in drei Aspekten: dem Modell der Synchronisation, der Benutzung von Prozeßnamen und der Struktur der Nachrichten.

5.2.3.3.1 Modell der Synchronisation: Man kann zwischen drei Methoden der Synchronisation unterscheiden:

1. **Asynchrone Kommunikation** (engl. auch *non-blocking communication* genannt): In diesem Fall wird der Absender nicht angehalten, nachdem er seine Nachricht abgesandt hat (im täglichen Leben entspricht dies dem Modell des Brief-Sendens).

2. **Synchrone Kommunikation, Rendezvous-Technik** (engl. auch *blocking communication* genannt): Der Absender wird angehalten (blockiert), bis der Empfänger die Nachricht erhalten hat. Im täglichen Leben entspricht dies dem Modell der Telefonkommunikation, und zwar in etwa der Kommunikation mit einem Anrufbeantworter. Diese Technik wird beispielsweise in CSP und dem daraus abgeleiteten occam eingesetzt.

3. **Erweiterte Rendezvous-Technik** (engl. *remote invocation*): Der Absender kann erst fortfahren, wenn eine Bestätigung vom Empfänger eingetroffen ist. Der Empfänger kann dabei noch Aktionen ausführen (z.B. nachdenken), bevor eine Antwort gesandt wird. Diese Technik entspricht der Telefonkommunikation mit einem persönlichen Gesprächspartner.

Diese Formen der Kommunikation können durcheinander simuliert werden. Aus zwei asynchronen Kommunikationen kann man eine synchrone realisieren. Durch zweifache synchrone Kommunikation kann man eine erweiterte Rendezvous-Technik simulieren.

Dies führt zu der Frage, ob es nicht ausreicht, asynchrone Kommunikation als Sprachprimitiv anzubieten und andere Formen der Kommunikation durch den Benutzer zusammensetzen zu lassen. Diese Frage kann man aus den folgenden Gründen verneinen:

- Es werden potentiell unendlich große Puffer benötigt, um Nachrichten zu speichern, für welche die Antwort noch aussteht.
- In den meisten Fällen muß eine Bestätigung erfolgen.
- Programme mit asynchroner Kommunikation sind komplexer.

Asynchrone Kommunikation kann auch mittels synchroner Kommunikation simuliert werden, nämlich in dem ein Pufferprozeß zwischen die kommunizierenden Prozesse geschaltet wird.

5.2.3.3.2 Prozeßnamen: Hier kann man zwischen zwei Klassen von Mechanismen unterscheiden, nämlich

1. **der direkten Bennenung des Kommunikationspartners**

   Beispiel:

   `SEND (Nachricht) TO (Prozeßname)`

   Der Vorteil dieser Form der Kommunikation ist die Einfachheit.

2. **der indirekten Bennenung des Kommunikationspartners**

   Beispiel:

   `SEND (Nachricht) TO (mailbox | pipe)`

   Der Vorteil dieser Form der Kommunikation ist die Dekomposition der Gesamtbeschreibung in unabhängige Module.

Die Benennung kann symmetrisch oder asymmetrisch sein: im ersten Fall benutzen beide Partner dieselbe Form der Kommunikation; im zweiten Fall ist diese unterschiedlich.

Im Falle der indirekten Kommunikation können weitere Unterscheidungen getroffen werden:

- eine n-zu-1-Beziehung (viele Klienten, ein Server),
- eine n-zu-m-Beziehung (viele Klienten, viele Server),
- eine Eins-zu-Eins-Beziehung (in diesem Fall müssen keine Puffer benutzt werden),
- eine 1-zu-n-Beziehung (diese wird selten benutzt).

5.2.3.3.3 Nachrichtenstruktur: Ursprünglich konnten nur elementare Maschinendatentypen als Nachrichten benutzt werden. Diese Beschränkung wurde wurde inzwischen meist aufgehoben.

5.2.3.3.4 Das occam2-Modell: In occam2 sind Prozesse nicht benannt, daher ist nur eine indirekte Kommunikation mittels Kanälen möglich. Jeder Kanal kann nur einen Schreib- und einen Leseprozeß haben. Beispiel:

```
ch ! x  -- schreibe x in den Kanal ch
...
ch ? y  -- lese y vom Kanal ch
```

Die Kommunikation ist synchron. Der jeweils zuerst bei der Kanaloperation eintreffende Prozeß muß auf das Rendezvous warten.

5.2.3.3.5 Das ADA-Modell: Das Akzeptieren von Nachrichten folgt in ADA dem Modell des Aufrufs einer externen Prozedur. Aufrufbare 'Prozeduren' eines Prozesses müssen in ADA als `ENTRY` deklariert werden. Hierzu gehört wie bei Prozeduren eine formale Parameterliste. Beispiel:

```
TASK screen_out IS
 ENTRY call(val:character; x, y: integer);
END screen_out;
```

Das Senden einer Nachricht sieht wie ein Prozeduraufruf aus:

```
screen_out.call('Z',10,20);
```

Für den Fall, daß der Prozeß, der die Nachricht erhalten soll, nicht mehr existiert, kann eine explizite Ausnahmebehandlung beschrieben werden:

```
BEGIN
 screen_out.call('Z',10,20);
 EXCEPTION
  WHEN tasking_error => (Ausnahmebehandlung)
END;
```

Das Empfangen einer Nachricht wird durch `ACCEPT`-Anweisungen ermöglicht. Beispiel:

```
 TASK BODY screen_out IS
 ...
 BEGIN
  ACCEPT call (val : character; x, y : integer) DO
  ...
  END call;
  ...
END screen_out;
```

`ACCEPT`-Anweisungen können überall dort stehen, wo auch andere Anweisungen zulässig sind. Sie führen zu einem Warten des aufrufenden Prozesses auf das Rendezvous. Fehlerbedingungen können durch `EXCEPTION`-Anweisungen abgefangen werden. Beispiel:

```
ACCEPT open(f: file_type) DO
 ....

EXCEPTION
 WHEN file_does_not_exist =>
 file_handler.create(f);
...
```

Da Prozesse auch auf das Eintreffen von Nachrichten verschiedener `ENTRY`-Einträge warten können, ist es noch erforderlich, ein solches Warten auf den Aufruf eines von mehreren *entries* angeben zu können. Hierfür dient die `SELECT`-Anweisung. Beispiel:

```
TASK screen_output IS
 ENTRY call_ch(val:character; x, y: integer);
 ENTRY call_int(z, x, y: integer);
END screen_out;
TASK BODY  screen_output IS
...
 SELECT
  ACCEPT call_ch ...  DO ..
  END call_ch;
 OR
  ACCEPT call_int ... DO ..
  END call_int;
 END SELECT;
...
```

Die Benennung von Prozessen ist in ADA asymmetrisch; nur einer der beteiligten Prozesse benennt den Kommunikationspartner.

5.2.4 Definition und Prüfung von Zeitbedingungen

Zeitbedingungen spielen für Realzeit-Systeme eine zentrale Rolle. Nach Burns [BW90] kann die Einführung der Zeit in eine Progammiersprache am besten durch die folgenden vier Anforderungen beschrieben werden:

1. Zugriff auf eine Uhr, mit deren Hilfe die Zeit gemessen werden kann,
2. Verzögerung von Prozessen um eine definierte Zeit,
3. Programmierung von *timeouts*, damit das Ausbleiben eines Ereignisses erkannt und behandelt werden kann,
4. Spezifikation von Zeitschranken (engl. *deadlines*) und Ablaufplanung (engl. *scheduling*), damit die notwendigen Zeitbedingungen spezifiziert und eingehalten werden können.

5.2.4.1 Zugriff auf eine Uhr

Der Zugriff auf eine Uhr kann direkt in der Sprache unterstützt sein. Dabei kann zwischen unterschiedlichen Komfort unterschieden werden. Eine sehr einfache Möglichkeit hierfür bietet occcam2. In occam2 kann der Wert einer Uhr über einen Kanal abgefragt werden. In occam2 ist nur eine Kommunikation zwischen einem Prozeß zu einem anderem Prozeß möglich, daher benötigt jeder Prozeß seine eigene Uhr (seine eigenen *timer*).

Beispiel

```
TIMER clock;
INT Time:
 SEQ
  clock? Time -- read time
```

Beim Lesen von Uhren kann es für den lesenden Prozeß nicht zu einer Blockierung kommen, da Uhren immer bereit sind, Werte zu liefern. Diese Werte repräsentieren in occam2 nur eine relative Zeit und sie werden lediglich mit dem Datentyp `int` dargestellt, haben also ggf. nur eine unzureichende Anzahl von Werten. Als anderes Extrem bietet ADA ein vordefiniertes Bibliothekspaket mit Namen `CALENDAR`. `CALENDAR` enthält eine Funktion `clock` sowie Konvertierungfunktionen, welche Jahre, Monate, Tage und Sekunden berechnen. Sekunden werden als Gleitkommazahlen mit einer implementationsabhängigen Anzahl von Mantissenzahlen dargestellt.

In anderen Programmiersprachen ist der Zugriff auf Uhren vielfach nur durch Programmierung eines Gerätetreibers für einen Uhrenbaustein möglich. Entsprechende Treiber können in Paketen abgelegt werden.

5.2.4.2 Verzögerung von Prozessen

Realzeit-Sprachen erlauben es in der Regel, die Verzögerung oder das Aufrufen von Prozessen zu spezifizieren. Beispiel in ADA:

```
DELAY 10.0
```

Man muß beachten, daß die angegebenen Zeiten stets Minimalzeiten sind. Wenn Prozessoren nach Ablauf der angegebenen Zeit durch Prozesse hoher Priorität blockiert sind, dann kann die Wartezeit der zu startenden Prozesse im Prinzip beliebig groß werden.

5.2.4.3 Programmierung von *timeouts*

Timeouts werden v.a. bei der Kommunikation zwischen Prozessen benötigt. Sie sind daher in vielen Realzeit- Sprachen vorgesehen. Beispiele:

1. occam2

   ```
   WHILE TRUE
     SEQ
      ALT
       call ? new_temp
       -- andere Aktionen
       clock? AFTER (10*G)
       -- Aktionen im Falle eine Timeouts
   ```

2. ADA

   ```
   LOOP
    -- lese neue Temperatur t
    SELECT
     controller.call (t)
    OR
     DELAY 0.5
   ```

```
      NULL
     END SELECT
    END LOOP
```

Nach max. 0,5 Sekunden fährt der aufrufende Prozeß mit der Bearbeitung fort.

5.2.4.4 Spezifikation von Zeitschranken und Ablaufplanung

Realzeit-Programme müssen nicht nur logisch korrekt sein. Sie müssen auch ein korrektes zeitliches Verhalten zeigen. In der Praxis herrschen vielfach ad-hoc Ansätze zur Überprüfung des zeitlichen Verhaltens vor.

Formale Methoden der Überprüfung des Zeitverhaltens fallen in zwei Klassen:

1. Beweisorientierte Methoden zeitbehafteter Logik (z.B. Temporale Logik)
2. Scheduling

wir werden uns in diesem Text ausschließlich mit der zweiten Klasse von Verfahren beschaäftigen.

Zur Beschreibung des Zeitverhaltens führen wir den Begriff des *temporal scopes (TS)* ein. Hierunter verstehen wir eine Menge von Anweisungen, für die Zeitbedingungen formuliert werden können. Für diese können wir zwischen fünf Zeitbedingungen unterscheiden (siehe Abb. 5.3)

1. die *deadline* bis zur Beendigung von TS,
2. die minimale Verzögerung bis zum Start von TS,
3. die maximale Zeit bis zum Start von TS,
4. die maximale Ausführungszeit von TS,
5. die maximal während der Ausführung von TS verstrichene Zeit.

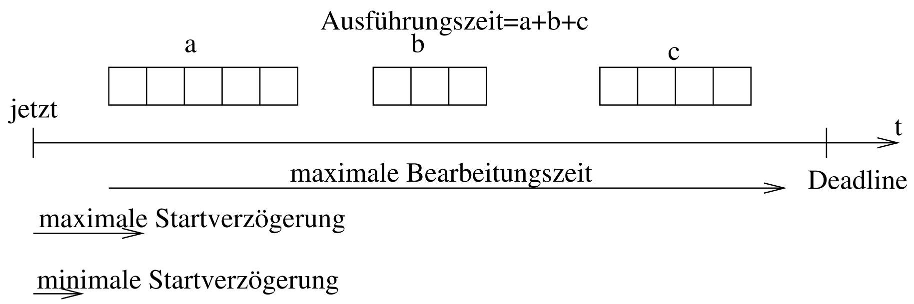


Abbildung 5.3: Zeitbedingungen

Wir unterscheiden zwischen **harten** und **weichen** *deadlines*. *Deadlines* sind hart, sofern das Nichteinhalten zu einer Katastrophe oder zu riesigen Kosten führen könnte. Andernfalls heißen *deadlines* weich.

Wir unterscheiden weiter zwischen **periodischen** und **aperiodischen Prozessen**. Periodische Prozesse werden typischerweise in einer Schleife ausgeführt. Aperiodische Prozesse werden üblicherweise durch Vorgänge in der physikalischen Umgebung angstoßen. Aperiodische Prozesse können ein Problem darstellen, sofern es keine untere Schranke für den Abstand zwischen zwei anstoßenden äußeren Ereignissen gibt. Existiert eine solche Schranke, heißt der Prozeß **sporadisch** (engl. *sporadic*).

Prioritäten

Prozesse werden üblicherweise mit **Prioritäten** versehen, um dem **Dispatcher** die unterschiedliche Wichtigkeit der Prozesse zu signalisieren.

Beispiel:

In ADA kann die Priorität mit Hilfe einer Compilerdirektive, eines sog. **Pragmas** angegeben werden.

```
TASK TYPE device_driver IS
  PRAGMA priority (p) --p: statischer Ausdruck
   ENTRY call (.....);
   ENTRY result  (.....);
END device_driver;
```

Die Semantik von ADA verlangt, daß ein **verdrängender (sog. preemptiver) Dispatcher** eingesetzt wird, d.h. Prozesse hoher Prioriät verdrängen stets solche niedrigerer Priorität.

Es gibt nur einen einzigen Zusammenhang, in dem die Priorität eines ADA-Prozesses geändert wird:

Während eines Rendezvous wird die Priorität auf das Maximum der beteiligten Prozesse angehoben. Aufgrund der statischen Prioritäten sowie des nicht definierten Verhaltens eines ADA-Programms im Falle mehrerer erfüllter `SELECT`-*guards* gilt der Prioritätsmechanismus in ADA als unzureichend. Dasselbe gilt auch für occam2. Auch für Java ergibt sich beim Einsatz als Realzeit-Sprache das Problem, daß das Verhalten des Dispatchers nicht vollständig spezifiziert ist [PMP+98].

Deadlines

Das folgende Programm zeigt das typische Verhalten eines periodischen Prozesses

```
PROCESS periodic;
...
BEGIN
 LOOP
 <idle>
  Beginn des temporal scope (TS)
  ...
  Ende des temporal scope
 END LOOP
END PROCESS;
```

Typischerweise beginnt TS mit dem Einlesen eines Wertes. Die anschließende Verarbeitung muß bis zum Ende der *deadline* abgeschlossen sein. Danach ist der Prozeß unbeschäftigt (idle), bis es Zeit ist, den nächsten Wert einzulesen. *Deadlines* haben in diesem Zusammenhang v.a. den Sinn, sicherzustellen, daß die Verarbeitung bis zum Eintreffen des nächsten Datenwertes sichergestellt ist. Die verfügbaren Realzeit-Sprachen erlauben unglücklicherweise die Angabe von *deadlines* nicht. Ersatzweise muß mit Verzögerungen gearbeitet werden.

Beispiel:

```
TASK BODY periodic IS
BEGIN
  LOOP
   -- lesen der Uhr und berechnen
   -- der Verzögerung
   DELAY del
   -- lesen des nächsten Werts
   -- Verarbeiten des nächsten Werts
  END LOOP;
END periodic;
```

Mit dieser Methode kann nicht garantiert werden, daß die *deadline* nicht bereits vor Ausführung der `DELAY`-Anweisung verpaßt wurde.

Zu den wenigen Sprachen, in denen explizit die zu erreichende Periode spezifiziert werden kann, gehört Pearl [DIN90]

Beispiel:

```
AFTER 10  MIN ALL 60 SEC ACTIVATE periodic;
```

5.2.5 Zustandsorientiertes Verhalten

12. Vorl.

In den bisherigen Beispielen haben wir Anwendungen betrachtet, die durch Variationen verschiedener Programmiersprachen beschrieben werden können. Die Motivation dafür liegt in umfangreichen Berechnungen, die für viele Anwendungen erforderlich sind. Sind die Berechnungen einfach und ist die Bedeutung der **Zustände**, in denen sich Systeme befinden groß, so sind Automatenmodelle von eingebetteten Systemen angemessener.

Klassische Automatenmodell werden dabei sehr schnell unübersichtlich. Dies wird durch die Sprache StateCharts vermieden, die auf Harel zurückgeht [Har87, DH89]. Harel zufolge wurde der Name StateCharts gewählt, weil StateChart *the only unused combination of 'flow' or 'state' with 'diagram' or 'chart'* war.

Die erste wesentliche Erweiterung gegenüber klassischen Zustandsdiagrammen besteht in der Einführung von Hierarchie. Zustände können danach zu Superzuständen (engl. *superstates*) zusammengefaßt werden. Dies zeigt die Abb. 5.4.

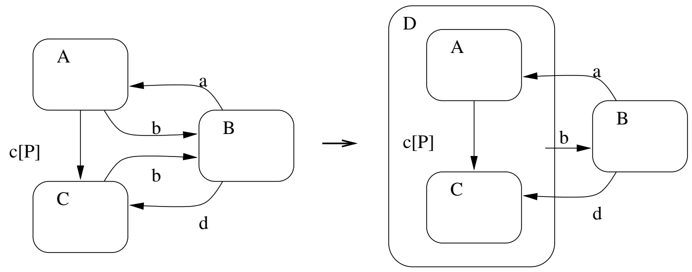


Abbildung 5.4: Zusammenfassen zu hierarchischen Zuständen

Da aus den Zuständen `A` und `C` nach `B` verzweigt wird, ergibt sich eine Reduktion der Anzahl der Kanten, wenn man `A` und `C` zu einem Superzustand `D` zusammenfaßt. Weiter kann es sinnvoll sein, die Reaktion auf `C(P)` in `D` zu verstecken. Die Kante aus `D` heraus bedeutet: `D` wird als Reaktion auf `b` verlassen, unabhängig davon, in welchem der Teilzustände von `D` sich das System aufhält. Bei der hier beschriebenen Form von Superzuständen schließen sich `A` und `C` dabei gegenseitig aus. Das System kann nur in einem der beiden Zustände sein und die beiden Zustände heißen **exklusiv**.

Neben den sich gegenseitig ausschließenden Teilzuständen gibt es auch die Möglichkeit sog. UND-verknüpfter oder **orthogonaler** Zuständen. Bei UND-verknüpften Teilzuständen befindet sich das System gleichzeitig in den Teilzuständen. UND-verknüpfte Teilzustände werden graphisch mit einer durchbrochenen Linie kenntlich gemacht. In Abb. 5.5 ist ein System zu sehen, welches sich gleichzeitig in den Zuständen `B` und `C` befindet, sofern `A` betreten wurde.

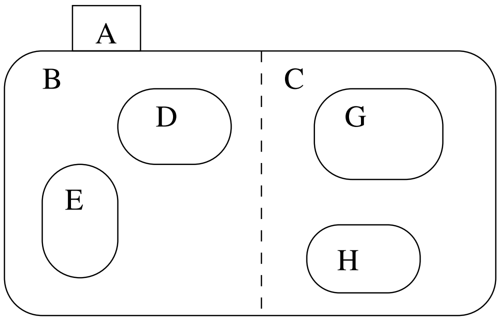


Abbildung 5.5: UND-verknüpfte Zustände

Mit UND-verknüpften Zuständen wird die Forderung nach Modellierung von Nebenläufigkeit erfüllt. Detaillierte Information zu StateCharts enthält das Skript zur Vorlesung "Rechnergestützter Entwurf und Produktion" [Mar98].

5.2.6 Behandlung von Non-Standard E/A-Geräten

Realzeit-Sprachen bieten meist die Möglichkeit, direkt die E/A-Hardware anzusprechen. Eine minimale Unterstützung besteht in der Möglichkeit, E/A-Register als Variable mit manuell zugewiesenen Speicheradressen zu benutzen. Dies ist in vielen Sprachen erlaubt. Beispiel (in ADA):

```
rdbr : character;
FOR rdbr USE AT 8#177560#
```

Mit einer sog. **Darstellungsklausel** (engl. *representation clause*) wird mitgeteilt, mit welcher Speicheradresse die Variable `rdbr` zu realisieren ist (in diesem Fall mit der Adresse eines E/A-Registers).

Weiter ist es erforderlich, auf Interrupts reagieren zu können. In ADA können für diesen Zweck Interrupts mit `ENTRY`-Vereinbarungen verbunden werden:

```
TASK handler IS
 ENTRY interrupt;
 FOR interrupt USE AT (interrupt adresse)
 PRAGMA priority(high)
END;
```

Leider mangelt es ADA an Möglichkeiten, die Priorität hinreichend flexibel beeinflussen zu können.

5.3 Beispiele von Realzeit-Sprachen

Im Rahmen von Beispielen haben wir verschiedene Echtzeitsprachen schon angesprochen. Ergänzende Information zu einzelnen Sprachen soll in diesem Abschnitt gegeben werden.

- **Pearl**

  Die Sprache Pearl wurde in Deutschland mit Unterstützung durch das Forschungsministerium entwickelt. Sie war über viele Jahre in der Industrie sehr verbreitet. Pearl enthält ein reichhaltiges Repertoir an Anweisungen zur Kontrolle von Prozessen. Implementierungen von Pearl beinhalten praktisch ein komplettes eigenes Betriebssystem. Aufgrund einer Reihe von Einschränkungen (Semaphore als Synchronisationsprimitive, keine Objektorientierung, weitgehend nationale Verbreitung) ist diese Sprache heute weitgehend abgelöst worden.

- **ADA**

  ADA (benannt nach Gräfin Ada Lovelace) wurde auf Initiative des US-amerikanischen Verteidigungsministeriums (*Department of Defense, DoD*) entwickelt, um eine einheitliche Sprache for alle eingebetteten Systeme des DoD benutzen zu können. So wurde beschlossen, eine neue Sprache zu entwickeln. Ausgangsbasis für drei verschiedene Sprachentwürfe waren PASCAL, PL/I und Algol86. Ein Entwurf auf der Basis von PASCAL hat sich schließlich durchgesetzt. Bei der Entwicklung von Produkten für das DoD ist die Verwendung von ADA vorgeschrieben.

  ADA ist Ausgangsbasis für die Hardwarebeschreibungssprache VHDL gewesen, welche ebenfalls im Auftrag des DoD entwickelt wurde.

- **CHILL**

  CHILL wurde speziell zur Programmierung von Telekommunikationseinrichtungen durch das CCITT in Auftrag gegeben und später auch von den namhaften europäischen Herstellern solcher Einrichtungen benutzt. Auch heute findet man CHILL beispielsweise als Sprache zur Realisierung von Fernmeldezentralen.

- **Modula/Modula-2**

  Modula und Modula-2 besitzen im Unterschied zu PASCAL auch ein Modul-Konzept. Darüber hinaus unterstützen diese Sprachen auch die explizite Beschreibung von Prozessen. Modula-2 ist sicherlich übersichtlicher als ADA, hat sich aber gegenüber ADA, C usw. nicht durchsetzen können.

  Tabelle 5.2 zeigt einen Vergleich der o.a. Sprachen [Zoe87]:

	Pearl	Modula-2	CHILL	ADA	occam
Modularisierung	mäßig	sehr gut	gut	sehr gut	—
Synchronisierung	Semaphore	Semaphore	u.a. Monitore	Kommunikation	Kommunikation
E/A-Fähigkeiten	umfassendes Sprachkonzept	Funktions-pakete	Funktions-pakete	Funktions-pakete	über Kanäle
Echtzeit-Sprachelemente	sehr gut	—	gut	gut	gut
Ereignis-behandlung	eigene Sprachelemente	über Interrupts	eigene Sprachelemente	eigene Sprachelemente	guarded commands
Ausnahme-behandlung	gut	—	gut	gut	—

Tabelle 5.2: Vergleich verschiedener Sprachen für die Echtzeitprogrammierung

- **Java**

  Java ist ursprünglich für die Entwicklung von eingebetteten System entwickelt worden, aber zunächst durch den Einsatz im Internet bekannt geworden. Nunmehr wird verstärkt versucht, Java auch für eingebettete Systeme zu verwenden. Als Vorteile von Java für diesen Einsatzbereich werden genannt [Nil96, PMP+98, KR96]:

  - Java ist **objektorientiert**.

    Dies bietet Möglichkeiten zur Beherrschung komplexer Systeme.
  - Java ist **einfach**.

    Vom Konzept her ist Java einfacher als C++ und vermeidet dadurch manche der in C++ möglichen Fehlerquellen.
  - Java **verkürzt Entwicklungszyklen**.

    Aufgrund der Unterstützung von interpretiertem und compiliertem Code kann die Programmerstellung beschleunigt werden.
  - Java ist **kompakt**.

    Laut Sun belegt der Basis-Interpreter 40 kBytes, Thread-Support und Basis-Bibliotheken belegen weitere 175 kBytes [Nil96]. Andere Quellen sprechen von 10 kByte minimaler Codegröße für CardJava.
  - Java ist **robust**.

    Durch die automatische *garbage collection* werden 'Speicherlöcher' und ungültige Zeiger vermieden. Ausnahmen können abgefangen werden.
  - Java ist **Plattform-unabhängig** und **portabel**.

    Im Bereich der eingebetteten Systeme gibt es viele Prozessoren, auf die jeweils der Code zu portieren ist. Aufgrund des standardisierten Bytecodes kann Code schnell auf eine neue Maschine übertragen werden.
  - Java ist **dynamisch**.

    Im Bedarfsfall können Klassen nachgeladen werden.
  - Java ist **sicher**.

    Geladener Code wird auf Viren und andere Sicherheitsprobleme untersucht, bevor er ausgeführt wird.

  Allerdings sind auch gravierende Probleme zu lösen, bevor Java in eingebetteten Systemen eingesetzt werden kann:

  - **Entzug des Prozessors zum Zwecke der Speicherverwaltung**: Java gestattet das explizite Aufrufen der *garbage collection*. Außerdem wird diese immer dann aufgerufen, wenn nicht mehr ausreichend Speicher zur Verfügung steht. *Garbage collection* führt bei den meisten Implementierungen zum Entzug des Prozessors für eine nicht tolerierbare Zeit.

– **Unspezifizierter Dispatcher:** Die meisten Java-Implementierungen benutzen nicht-unterbrechende Dispatcher. Dies kann bei Realzeit-Anwendungen zu nicht tolerierbaren Laufzeiten führen. Da insgesamt das Verhalten des Dispatchers ist nicht vollständig spezifiziert ist, können bei Standard-Java auch kaum sinnvolle Aussagen zu *worst case*-Laufzeiten gemacht werden.

– **Lizenzprobleme**: Angeblich verbieten die Sun-Lizenzen den Einsatz von Java in sicherheitskritischen Anwendungen[2].

Erste Vorschläge zur Lösung der angesprochenen Probleme stammen von Nilsen [Nil96]. Zu den Vorschlägen gehören u.a. hardwareunterstützte *garbage collection*, spezielle Kennzeichnung von Codesegmenten zur Unterstützung der Laufzeit-Analyse, der Austausch des Dispatchers usw. Auch die Fa. Sun arbeitet an entsprechenden Varianten von Java.

- **Weitere**

  Es soll erwähnt werden, daß es viele weitere Sprachen zur Programmierung insbesondere von verteilten Systemen gibt, wie z.B. SDL, Estelle, Esterelle, Petri-Netze usw. (siehe u.a. [Mar98]).

- **DIN 40719, IEC 848**

  Für die Programmierung speicherprogrammierbarer Steuerungen wird die standardisierte graphische Darstellung nach DIN 40719 bzw. IEC 848 benutzt. Dieser Standard beschreibt letztlich nur die Darstellung von Automaten durch Ablaufpläne und bietet eine um Größenordnungen geringere Funktionalität als die bislang besprochenen Sprachen. Ein Beispiel einer Darstellung ist die Abb. 5.6 [Fei97].

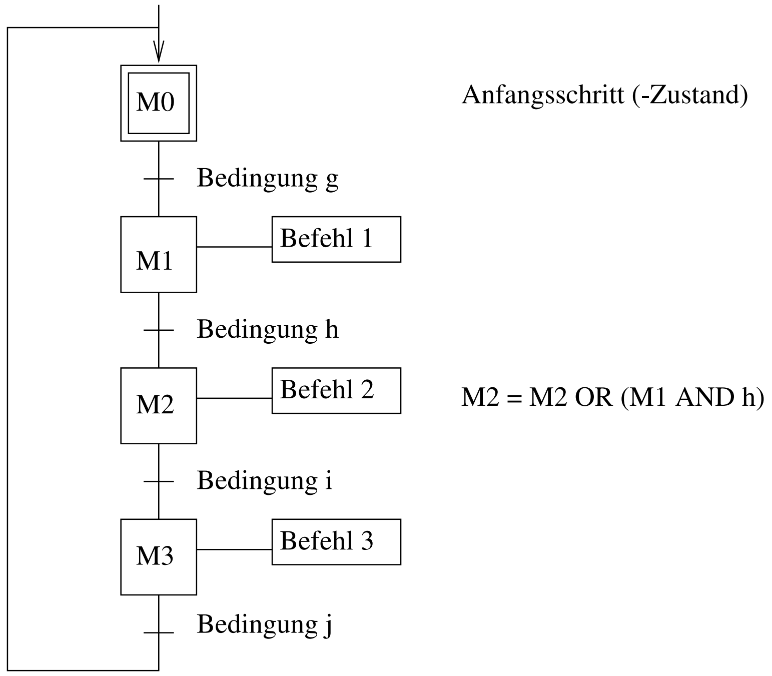


Abbildung 5.6: Funktionsplan einer Ablaufsteuerung nach DIN 40719

Die einzelnen Zustände heißen in dieser Norm Schritte. Es gibt die Vorstellung, daß jedem Schritt ein Bit eines Zustandswort zugeordnet ist und mittels Boolescher Gleichungen die Bedingungen für das Setzen bzw. Rücksetzen der Kodebits formuliert werden können. Diese Booleschen Gleichungen können dann in verschiedenen zugelassenen Programmiersprachen (C, BASIC, PASCAL) aufgeschrieben werden.

Die Norm beschreibt auch Aufspaltungen und Zusammenführungen des Ablaufs sowie die Darstellung der jeweiligen Ausgabe. Abb. 5.7 enthält hierfür zwei Beispiele.

Techniken, welche aus derartigen Darstellungen optimierte Schaltwerke realisieren, sind schon seit Jahrzehnten bekannt (Stichwort: **Controller-Synthese**).

[2]Diese werden dann wohl weiter z.B. in STEP 7 programmiert, s.u.

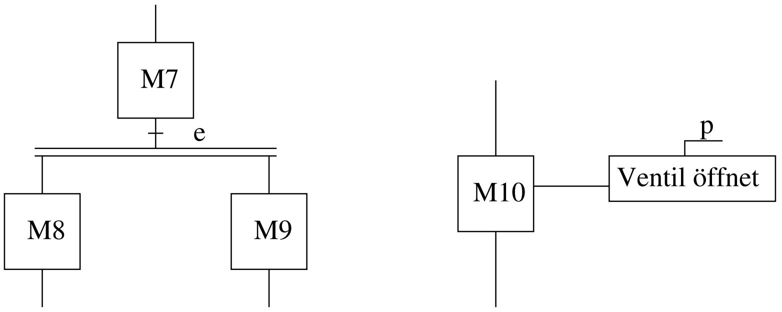


Abbildung 5.7: Symbole nach DIN 40719: Aufspaltung (links) und Ausgabe (rechts)

- **STEP 7**

  In noch tiefere Niederungen der Programmierung kommt man bei der Betrachtung der Sprachen, in denen noch heute speicherprogrammierbare Steuerungen (SPS) programmiert werden. Steuerungen der Fa. Siemens werden in STEP 7 programmiert. Hierin sind zu verwenden:

  - E0.0 bis E0.7, E1.0 bis E1.7 usw. als Bezeichner von Eingangsleitungen
  - A0.0 bis A0.7, A1.0 bis A1.7 usw. als Bezeichner von Ausgangsleitungen
  - M0.0 bis M0.7, M1.0 bis M1.7 usw. als Bezeichner von 'Merkerspeichern'

  Mit den assemblerartigen Anweisungen
  `U xxx`,
  `O xxx`,
  `UN xxx`
  können UND-, ODER- und nicht-UND-Verknüpfungen zwischen den o.a. Bezeichnern und einem impliziten Akkumlator ausgeführt werden.

  Dies liegt unter dem Niveau eines Assemblers, der zumindest symbolische Speichernamen statt (oktaler) Adressen erlauben würde!

5.4 Programmierung einer Schachtentwässerung

Als Beispiel einer Programmentwicklung betrachten wir die Programmierung einer Schachtentwässerung in ADA [BW90]. Wir nehmen an, daß eine Schachtanlage mit einer Pumpe zu entwässern ist (vgl. Abb. 5.8).

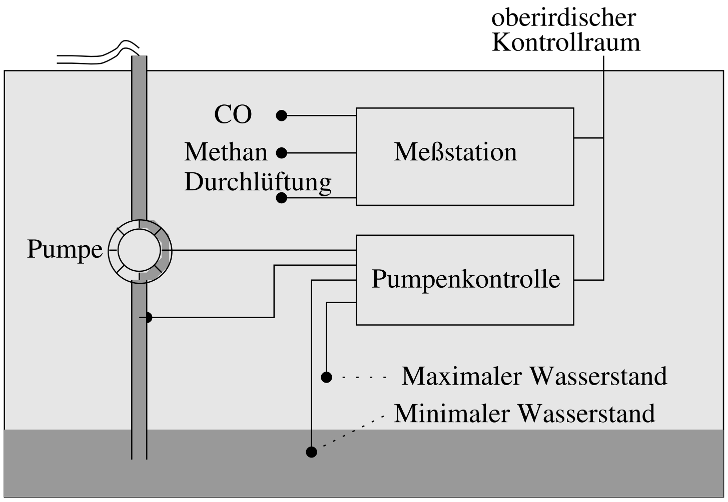


Abbildung 5.8: Schachtentwässerung

In dieser Schachtanlage gibt es Sensoren für den Wasserstand, welche die Pumpe jeweils ein- und ausschalten sollen. Der Fluß des Wassers wird über einen Sensor überwacht. Weiter gibt es Sensoren für die

Kohlenmonoxyd-Konzentration, die Methan-Konzentration und die Durchlüftung. Bei zu hohen Methanwerten darf die Pumpe wegen Explosionsgefahr nicht eingeschaltet werden.

Bei der Entwicklung des Kontrollprogrammes verwenden wir die Symbole, die im Buch von Burns benutzt werden, obwohl es inzwischen neuere Standards hierfür gibt. Die Darstellung einer Aufspaltung eines Systems in zwei Blöcke und die Bedeutung verschiedener Symbole zeigt die Abb. 5.9.

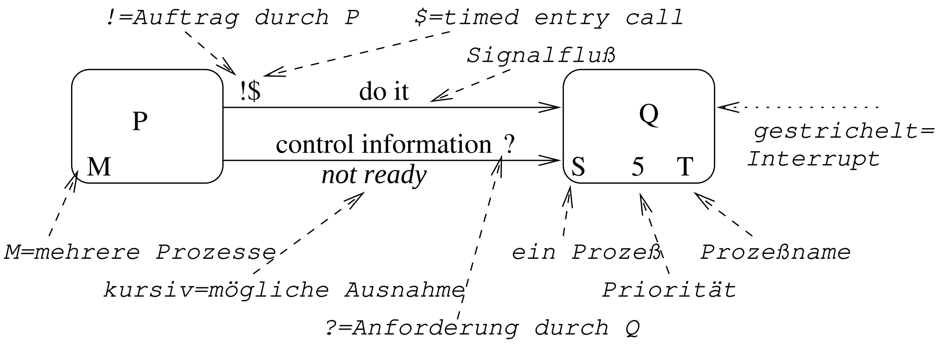


Abbildung 5.9: Symbole der benutzten graphischen Darstellung

Eine graphische Darstellung des Gesamtsystems zeigt die Abb. 5.10. Es wird angenommen, daß die Wassersensoren per Interrupt kommunizieren. Die übrigen Sensoren sollen periodisch abgefragt werden.

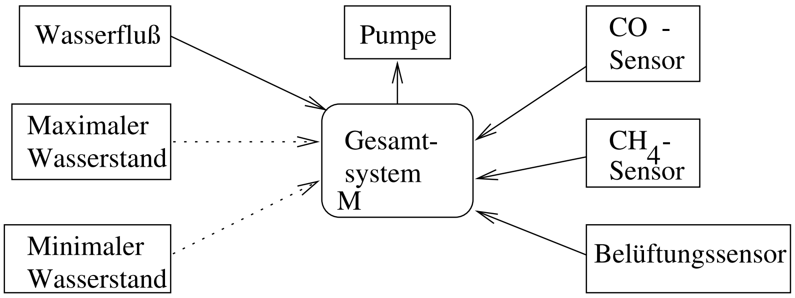


Abbildung 5.10: Darstellung des Gesamtsystems

Eine erste Zergliederung des Systems zeigt die Abb. 5.11.

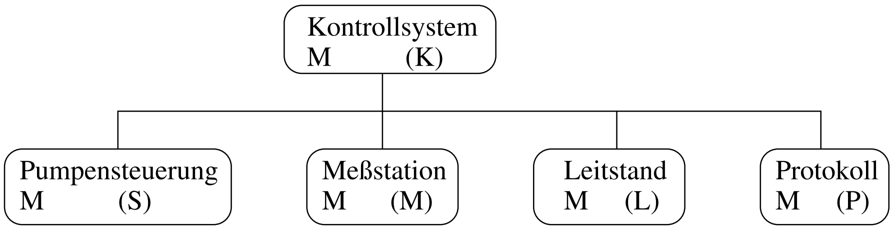


Abbildung 5.11: Zerlegung des Systems

Die Datenflüsse zeigt die Abb. 5.12.

Die Aufrufe an den Protokoll-Block P sind mit einer Zeitüberwachung ($) versehen, damit Verzögerungen beim Protokollieren keinen negativen Einfluß auf den kritischen Teil des Systems haben können. Die Meßstation erzeugt einen Alarm, sobald einer der Meßwerte zu groß ist. Sie informiert außerdem die Pumpensteuerung, sobald der Methangehalt zu groß wird bzw. wenn dieser Gehalt wieder hinreichend weit abgefallen ist. Die Pumpensteuerung S startet die Pumpe nur, wenn der Methangehalt hinreichend niedrig ist. Wenn die Pumpe nicht gestartet werden kann und wenn sie kein Pumpen des Wassers bewirkt, dann wird ein Alarm am Leitstand erzeugt. Vom Leitstand aus kann die Pumpe direkt angesteuert werden, es sei denn, der Methangehalt ist zu hoch.

Nach dieser ersten Strukturierung können wir beginnen, ADA-Beschreibungen anzugeben.

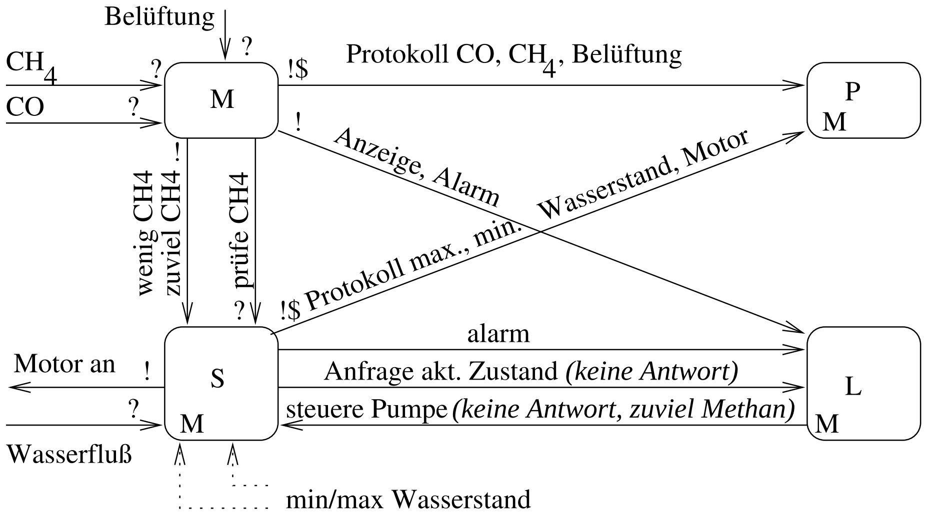


Abbildung 5.12: Datenflüsse im Gesamtsystem

Zunächst sind einige globale Typvereinbarungen erforderlich, deren Bedeutung aus der Bezeichnung klar sein sollte [BW90]:

```
PACKAGE system_types IS
  TYPE pump_status IS (on, off, disabled);
  TYPE methane_status IS (motor_safe, motor_unsafe);
  TYPE water_mark IS (high, low);
  TYPE ch4_reading IS NEW integer RANGE 0.. 1023;
  TYPE co_reading IS NEW integer RANGE 0 .. 1023;
  TYPE motor_state_changes IS (motor_started,
       motor_stopped, motor_safe, motor_unsafe);
  TYPE alarm_reason IS (high_methane, high_co, pump_dead,
                        data_logger_dead, no_air_flow,
                        ch4_device_error, co_device_error,
                        unknown_error);
       water_flow_sensor_period : CONSTANT duration:=30.0;
       air_flow-sensor_period   : CONSTANT duration:=60.0;
       ch4_sensor_period        : CONSTANT duration:=10,0;
       co_sensor_period         : CONSTANT duration:=25,0;
       co_high                  : CONSTANT co_reading:=600;
       ch4_high                 : CONSTANT ch4_reading:=400;
END system_types;
```

Die Definition der Perioden entspricht dabei der Tabelle der Abtastperioden der verschiedenen Sensoren (siehe Tabelle 5.3).

	Periode [Sek]
Methan	10
Kohlenmonoxyd	25
Wasserfluß	30
Belüftung	60

Tabelle 5.3: Periode zyklisch abgefragter Sensoren

Für jedes der vier Teilsysteme geben wir jetzt die Paketspezifikationen an.

Für die Pumpensteuerung S lautet sie wie folgt:

```
WITH system_types; USE system_types;
PACKAGE S IS
 not_safe:EXCEPTION;                  --zuviel Methan
             -- verursacht durch L_interface.set_pump
 not_responding:EXCEPTION;            --keine Antwort
       -- verursacht durch L.set_pump und M.stop_pump
```

```
 PACKAGE M_interface IS
   PROCEDURE not_safe;
   PROCEDURE is_safe;
 END M_interface;
 PACKAGE L_interface IS
  FUNCTION request_status RETURN pump_status;
  PROCEDURE set_pump(to:pump_status);
 END L_interface;
 -- ruft function check_safe in M.S_interface
 -- ruft entries waterflow_log und high_low_water_log und motor_log in
 -- P_retrieval.S_interface
 -- ruft procedure pump_alarm in L.S_interface
END S;
```

Die nächsten Zeilen beschreiben die Software-Schnittstellen der Meßstation M:

```
WITH system_types; USE system_types;
PACKAGE M IS
 PACKAGE S_interface IS
  FUNCTION check_safe RETURN methane_status;
 END pump_control_interface;
-- ruft procedures is_safe und not_safe in S.M_interface
-- ruft procedure alarm in L_interface.M_interface
-- ruft entries airflow_log, co_log, ch4_log in P_retrieval.M_interface
END M;
```

Das dritte Paket beschreibt die Schnittstelle der Protokollierung P:

```
WITH system_types; USE system_types;
PACKAGE P_retrieval IS
  TASK M_interface IS
     ENTRY co_log(reading:co_reading);
     ENTRY ch4_log(reading: ch4_reading);
     ENTRY airflow_log(reading: boolean);
     PRAGMA priority(0);
  END M_interface;
  TASK S_interface IS
    ENTRY high_low_water_log(mark:water_mark);
    ENTRY water_flow_log(reading:boolean);
    ENTRY motor_log(state: motor_state_changes);
    PRAGMA priority(0);
  END S_interface;
  -- auch für den Leitstand wird noch ein Interface gebraucht
END P_retrieval;
```

Das vierte und letzte Paket beschreibt die Schnittstelle der Leitstands L:

```
WITH system_types; USE system_types;
PACKAGE L IS
 PACKAGE M_interface IS
   PROCEDURE alarm(reason:alarm_reason);
 END M_interface;
 PACKAGE S_interface IS
  PROCEDURE alarm(reason:alarm_reason)
    RENAMES M_interface.alarm;
 END S_interface;
 -- ruft request_status in S.L_interface
 -- ruft set_pump in S.L_interface
END L;
```

Nach der Beschreibung der Schnittstellen kommen wir nun zu der Beschreibung der Rümpfe.

Als erstes stellen wir den Rumpf der Pumpensteuerung dar:

```
WITH system; USE system;
PACKAGE BODY S IS
  TASK high_low_water_sensor IS
    ENTRY high_sensor;
```

```
    ENTRY low_sensor;
    FOR high_sensor USE AT 16#40#; -- Interrupt Adresse
    FOR low_sensor USE AT 16#44#; -- Interrupt Adresse
    PRAGMA priority(10);
  END high_low_water_sensor;
  TASK flow_sensor IS
   PRAGMA priority(6);
  END flow_sensor;
  TASK motor IS
   ENTRY start; -- kann not_safe_to_start verursachen
   ENTRY stop;
   ENTRY is_safe;
   ENTRY not_safe;
   ENTRY enquiry_status(current_pump_status; OUT pump_status);
   PRAGMA priority(9);
  END motor;
  TASK BODY high_low_water_sensor IS SEPARATE;
  TASK BODY flow_sensor IS SEPARATE;
  TASK BODY motor IS SEPARATE;
  PACKAGE BODY L_interface IS SEPARATE;
  PACKAGE BODY M_interface IS SEPARATE:
END S;
```

Als zweiter Rumpf folgt der des Leitstandes:

```
SEPARATE (S)
PACKAGE BODY L_interface IS
 FUNCTION request_status RETURN pump_status IS
         current_pump_status:pump_status;
  BEGIN
    motor.enquire_status(current_pump_status);
    RETURN current_pump_status;
  END;
 PROCEDURE set_pump(to:pump_status) IS
  BEGIN
   IF to=off THEN
     motor.stop;
   ELSIF to=on THEN
     motor.start; -- alle Ausnahmen gehen zum Aufrufer
   ELSE
     motor.not_safe;
  END IF;
  EXCEPTION
   WHEN tasking_error => RAISE not_responding;
  END set_pump;
END L_interface;
```

Der dritte Rumpf ist ein Teil der Beschreibung der Meßstation:

```
WITH L;
SEPARATE(S)
PACKAGE BODY M_interface IS
 PROCEDURE not_safe IS
   BEGIN
      motor.not_safe;
      L.S_interface.alarm(high_methane);
   EXCEPTION
    WHEN tasking_error =>  L.S_interface.alarm(pump_dead);
  RAISE not_responding;
END not_safe;
 PROCEDURE is_safe IS
   BEGIN
  motor.is_safe;
   EXCEPTION
  WHEN tasking_error =>  L.S_interface.alarm(pump_dead);
   RAISE not_responding;
```

```
END is_safe;
END M_interface;
```

Als nächstes geben wir den Beschreibung von Kontrollregistern, welche bei der Darstellung der drei Prozesse der Pumpensteuerung benötigt werden:

```
PACKAGE device_register IS
 word     : CONSTANT:=2; -- 2 Bytes / Wort
 one_word : CONSTANT:=16; -- # Bits / Wort
-- Felder des Kontrollregisters
 TYPE device_error     IS (clear,set);
 TYPE device_operation IS (clear,set);
 TYPE interrupt_status IS (i_disabled, i_enabled);
 TYPE device_staturs   IS (d_disabled, d_enabled);
 -- das Register selbst
 TYPE csr IS
 RECORD
   error_bit    : device_error;
   operation    :  device_operation;
   done         : boolean;
   interrupt    : interrupt_status;
   device       : device_status;
END RECORD;
-- Kodierung der Bits des Registerfeldes
FOR device_error     USE (clear=>0, set=>1);
FOR device_operation USE (clear=>0, set=>1);
FOR interrupt_status USE (i_disabled=>0,i_enabled=>1);
FOR device_status    USE (d_disabled=>0,d_enabled=>1);
FOR csr USE
  RECORD AT MODE word;
    error_bit   AT 0 RANGE 15..15;
    operation   AT 0 RANGE 10..10;
    done        AT 0 RANGE 7..7;
    interrupt   AT 0 RANGE 6..6;
    device      AT 0 RANGE 0..0;
  END RECORD;
 FOR csr'size USE one_word;
END device_register;
```

Der erste Prozeß ist derjenige, der mit den Wasserstands-Sensoren verbunden ist: ausgelöst wird:

```
WITH P_retrieval;
WITH L;
WITH device_register; USE device_register;
SEPARATE (S)
TASK BODY high_low_water_sensor IS
  -- sporadischer Prozeß; Definition der Kontroll- und
  -- Statusregister für die beiden Sensoren
  hwcsr : device_register.csr;
  FOR hwcsr USE AT 16#aa10#;
  lwcsr : device_register.csr;
  FOR lwcsr USE AT 16#aa12#;
  water : water_mark;
BEGIN
  --enable der Geräte
  hwcsr.device: =d_enabled;
  lwcsr.device: =d_enabled;
 LOOP
   BEGIN
     -- enable der Interrupts
     hwcsr.interrupt:=i_enabled;
     lwcsr.interrupt:=i_enabled;
     BEGIN
       SELECT
        ACCEPT high_sensor;
        motor.start;
```

```
        water:=high;
       OR
        ACCEPT low_sensor;
        motor.stop;
        water: =low;
       END SELECT;
    EXCEPTION
     WHEN tasking_error=> L.S_interface.alarm(pump_dead);
    END:
    SELECT
     P_retrieval.S_interface.high_low_water_log(water);
    OR
      DELAY 10,0;
    END SELECT;
    EXCEPTION
    WHEN not_safe      => L.S_interface.alarm(high_methane);
    WHEN tasking_error => L.S_interface_alarm(data:logger_dead);
    WHEN OTHERS        => L.S_interface.alarm(unknown_error);
     END;
    END LOOP;
END high_low_water_sensor;
```

Als zweites beschreiben wir den Prozeß, der mit der Kontrolle des Wasserflusses verbunden ist:

```
WITH P_retrieval;
WITH L;
WITH calendar; USE calendar;
WITH device_register; USE device_register;
SEPARATE(S)
TASK BODY flow_sensor IS                --periodischer Prozeß
  start_time         : calendar.time;
  water_flow         : boolean:=false;
  current_pump_status: pump_status;
  -- definiere Kontroll- und Statusregister des Flußschalters
  wfcsr              : device_register.csr;
  FOR wfcsr USE AT 16#aa14#;
 BEGIN
  -- enable des Gerätes
  wfcsr.device:=d_enabled;
  LOOP
   BEGIN
     start_time:=calendar.clock;
     motor.enquire_status(current:pump_status);  -- lese Geräteregister
     water_flow:=(wfcsr.operation=set);
     IF current_pump_status=on AND NOT water_flow THEN
     -- gibt der Pumpe Zeit bis Wasser fliessen kann
     DELAY 10,0;
     -- lese das Kontrollregister noch einmal
     water_flow:=(wfcasr.operation=set);
     IF current_pump_status =on AND NOT water_flow THEN
      L.S_interface.alarm(pump_dead);
   END IF;
 END IF;
   SELECT
     P_retrieval.S_interface.water_flow_log(water_flow);
   OR -- delay until next period
    DELAY(calendar.clock - (start_time + water_flow_sensor_period));
  END SELECT;
  -- kann 0.0 sein
  DELAY(calendar.clock - (start_time + water_flow_sensor_period));
 EXCEPTION
    WHEN tasking_error => L.S_interface.alarm(data_logger_dead);
    WHEN OTHERS        => L.S_interface.alarm(unknown_error);
   END;
  END LOOP;
```

```
END FLOW_sensor;
```

Die Rümpfe der Meßstation und der Pumpensteuerung lassen wir aus Platzgründen aus.

Bei einem echten Entwurf müßte jetzt geprüft werden, ob die Zeitbedingungen, die durch die Umgebung gestellt werden, erfüllt werden können. Auch müßten Fehlertoleranzmaßnahmen diskutiert werden.

Kapitel 6

Softwareentwicklungsprozesse für Realzeitsysteme

13. Vorl.

Softwareentwicklungsprozesse für Realzeitsysteme sind zunächst einmal eines: Softwareentwicklungsprozesse. Das bedeutet: alle Aspekte 'normaler' Softwareentwicklung treffen auch auch die Softwareentwicklung für Realzeitsysteme zu. Spezifikationstechniken, Entwicklungsumgebungen, Entwicklungszyklen, Versionskontrolle usw. werden im allgemeinen auch für Realzeitsysteme benötigt.

Es kommen allerdings einige Aspekte hinzu:

1. Bedingt durch die Benutzung mehrerer Prozesse und die Existenz harter *Deadlines* muß man sich mit dem *scheduling* beschäftigen.
2. Nicht eingebettete Systeme realisieren bekanntlich im wesentlichen eine Funktion und können daher im Prinzip durch Probeläufe mit verschiedenen Mengen von Funktionsargumenten getestet werden. Dies kann *off-line* an einem isolierten Rechner oder in einem isolierten Prozeß geschehen. Dieser Vorgang ist schwierig genug, aber das Überprüfen der Arbeitsweise von eingebetteter Software ist im allgemeinen noch schwieriger. Dies liegt daran, daß dies Abläufe in der echten Umgebung und möglichst in Realzeit überprüft werden müssen.
3. Aufgrund des Dauerbetriebs kommen noch einige Anforderungen an die Software hinzu: z.B. dürfen keine **Speicherlöcher** durch nicht freigegebenen Speicher entstehen und die korrekte Reaktion auf Ausnahmen muß überprüft werden. Letzteres kann sehr schwierig sein und selbst wieder zu Fehlern führen (man denke an Tests von Atomreaktoren!).
4. Aufgrund der hohen Sicherheitsanforderungen muß Software für sicherheitskritische Bereiche hochwertiger sein als Software beispielsweise für die Textverarbeitung.

Wir beginnen in diesem Kapitel zunächst mit einer Diskussion[1] einiger grundlegender Aspekte des Punktes 1.

6.1 Realzeit-Scheduling

6.1.1 Begriffe

Die Aufgabe des Schedulings besteht in der Entscheidung des zeitlichen Ablaufs der Prozesse, aus denen eine Realzeit-Anwendung üblicherweise besteht. Dabei sind alle Randbedingungen hinsichtlich der Ressourcen, der *deadlines* und der Abhängigkeiten der Prozesse untereinander einzuhalten.

Scheduling-Verfahren kann man nach verschiedenen Verfahren klassifizieren. Abb. 6.1 zeigt eine solche Klassifikation.

[1] Für diese Diskussion wird Material des Buches von Kopetz [Kop97] verwendet.

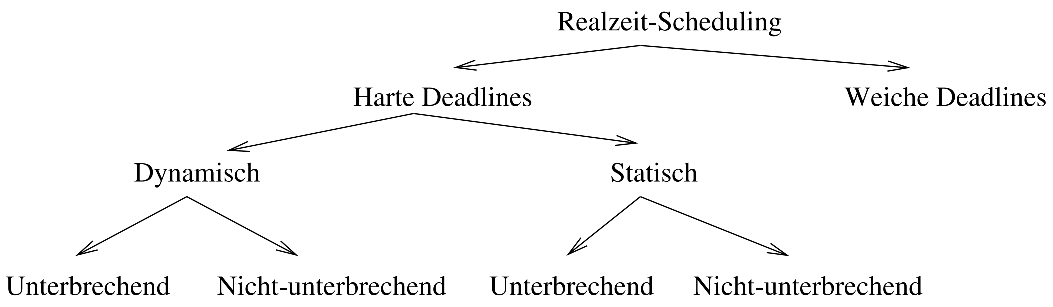


Abbildung 6.1: Klassifikation von Scheduling-Verfahren

Dynamische Scheduler treffen ihre Entscheidungen zur Laufzeit. Sie bilden eine flexible Lösung, verursachen aber ggf. einen hohen Aufwand während der Laufzeit. Außerdem besitzen sie keine Kenntnis globaler Zusammenhänge für eine gegebene Anwendung, es sei denn, zusätzliche Tabellen werden als Ergebnis der Software-Generierung bereitgestellt. Kenntnisse über solche globalen Zusammenhänge sind natürlich gerade bei eingebetteten Systemen wichtig, da hier die Applikation weitgehend in der Entwurfsphase bekannt ist (siehe Kap. 1).

Statische Scheduler basieren auf einer vollständigen Planung des Ablaufs der Prozesse während der Entwurfsphase. Diese erzeugen in der Regel eine Tabelle, welche zur Laufzeit einem einfachen Dispatcher zur Verfügung steht. In dieser Tabelle ist fest eingetragen, zu welcher Zeit innerhalb einer Periode welcher Prozeß zu starten ist.

Nicht-unterbrechende Scheduler gehen davon aus, daß Prozesse ausgeführt werden, bis sie (regulär oder aufgrund einer Ausnahme) terminieren. Der Nachteil dieser Scheduler liegt ganz offensichtlich in einer langen Reaktionszeit, sofern einige Prozesse eine große Ausführungszeit haben. Andererseits sind sie wegen ihrer geringen Belastung der Prozessoren vorteilhaft, wenn alle Prozesse ohnehin nur eine kurze Ausführungszeit haben.

Unterbrechende Scheduler sind zu verwenden, wenn einige Prozesse eine große Ausführungszeit haben oder wenn die spezifizierten Reaktionszeiten klein sind.

Zentralisiertes Scheduling liegt vor, wenn der Scheduler auf einem einzigen Prozessor ausgeführt wird. Zentralisiertes Scheduling bietet keinerlei Redundanz und kann darüber hinaus auch zu einer Belastung der Kommunikationstrukturen führen.

Verteiltes Scheduling vermeidet die Probleme des zentralisierten Schedulings.

Eine zentrale Frage für das Scheduling ist die Frage, ob für eine gegebene Menge von Prozessen ein Schedule existiert, welches alle Randbedingungen einhält (d.h. die Frage der *schedulability*). Wir unterscheiden zwischen exakten, hinreichenden und notwendigen Tests auf *schedulability*.

Exakte Tests liefern nie falsche Aussagen über die *schedulability*. Derartige Tests sind in praktisch allen interessierenden Konstellationen NP-vollständig [GJ79].

Hinreichende Tests können (mit hoffentlich kleiner Wahrscheinlichkeit) behaupten, es gäbe kein gültiges Schedule, obwohl ein solches existiert. Hinreichende Tests können in nicht-exponentieller Laufzeit realisiert werden.

Notwendige Tests können (mit hoffentlich kleiner Wahrscheinlichkeit) behaupten, es gäbe ein gültiges Schedule, obwohl ein solches nicht existiert. Notwendige Tests können in nicht-exponentieller Laufzeit realisiert werden.

Für das folgende ist es wichtig, zwischen periodischen und sporadischen Prozessen zu unterscheiden.

Periodische Prozesse T_i müssen jeweils einmal innerhalb einer Periode p_i ausgeführt werden. Sie mögen eine Rechenzeit von c_i haben. Das *deadline intervall* d_i ist die Zeit zwischen dem Zeitpunkt, zu dem T_i ausführbar wird und dem Zeitpunkt, an dem T_i bearbeitet sein muß. Falls $d_i = c_i$ ist, so muß der Prozeß stets unmittelbar bearbeitet werden. Die Zeit $d_i - c_i$ nennen wir *laxity* des Prozesses T_i.

Bei der Untersuchung der Existenz eines Schedules für eine Menge periodischer Prozesse können wir uns auf das kleinste gemeinsame Vielfache der Perioden der Prozesse beschränken. Dieses kleinste gemeinsame Vielfache heißt *Periode des Schedules*.

Eine notwendige Bedingung für eine Existenz eines Schedules bei m Prozessoren ist die folgende:

$$\mu \quad = \quad \sum \frac{c_i}{p_i} \leq m \tag{6.1}$$

Dies bedeutet, daß die Summe der Auslastungen unter der Anzahl der Prozessoren liegen muß. Gleichung 6.1 ist allerdings nur eine verhältnismäßig einfache Bedingung.

Sporadische Prozesse sind Prozesse, für welche die Zeitpunkte, an denen sie Prozessor-Ressourcen anfordern, nicht vorab bekannt sind. Wir verlangen, daß für den Abstand zwischen je zwei solchen Zeitpunkten eine untere Schranke existiert. Prozesse, für welche eine solche Schranke nicht existiert, heißen **aperiodisch**.

Wenn man alle Zeitpunkte kennt, zu denen Prozesse den Prozessor anfordern, dann kann man im Prinzip immer ein gültiges Schedule erzeugen, sofern es ein solches gibt. Notfalls könnte man ja alle möglichen Schedules ausprobieren. Ein Scheduling-Verfahren, welches immer ein gültiges Schedule findet, sofern eines existiert, heißt **optimales Scheduling-Verfahren**.

Kennt man diese o.a. Zeitpunkte nicht, so funktioniert das Ausprobieren auch nicht. Scheduler, welche diese Zeitpunkte nicht kennen, könnten also ggf. nicht optimal sein. Ein Beispiel zeigt, daß es bei Anwesenheit von Ressourcenkonflikten keinen solchen Scheduler geben kann, der optimal ist.

Man betrachte die Prozesse in Abb. 6.2.

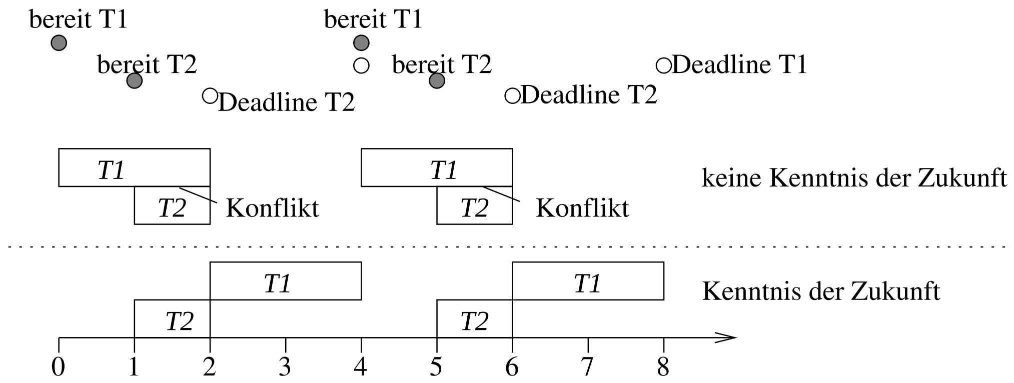


Abbildung 6.2: Fall, in dem die Kenntnis der 'Zukunft' erforderlich ist

Die gleichzeitige Ausführung von $T1$ und $T2$ möge aufgrund von Ressourcenkonflikten nicht möglich sein. $T1$ sei ein periodischer Prozeß mit $c_1 = 2, d_1 = 4, p_1 = 4$. T_2 sei ein sporadischer Prozeß mit $c_2 = 1, d_2 = 1, p_2 = 4$. Sofern T_1 ausführbereit wird (während der Prozessor frei ist), so muß ein Dispatcher, welcher nur die Vergangenheit kennt, diesen Prozeß auch starten. Sonst würde evtl. die verfügbare Prozessorzeit nicht genutzt. Wenn aber, wie im Beispiel der Abb. 6.2 der Prozeß T_2 kurz nach dem Start seine Ausführung verlangt, so wird wegen des bereits gestarteten T_1 die Deadline von T_2 verpaßt.

Kenntnisse über die 'Zukunft', wie sie bei Realzeit-Systemen für periodische Prozesse existieren, sind also nützlich für das Lösen der Scheduling-Aufgabe.

Das Beispiel zeigt, daß kein allgemeiner, optimaler Scheduler existiert, welcher ohne Kenntnis der Zukunft auskommt. Scheduling-Verfahren, welche auf der Basis der Kenntnis der Vergangenheit arbeiten, können nicht in jedem Fall ein gültiges Schedule erzeugen, sofern es existiert.

6.1.2 Dynamisches Scheduling

6.1.2.1 Unabhängige Prozesse

Ein klassischer Algorithmus für unabhängige Prozesse und einen einzelnen Prozessor ist bekannt als *rate monotonic scheduling*. *Rate monotonic scheduling* ist ein unterbrechender dynamischer Algorithmus, welcher auf statischen Prozeßprioritäten basiert. Er stammt von Liu (1973). *Rate monotonic scheduling* basiert auf den folgenden Vorraussetzungen:

1. Alle Prozesse $\{T_i\}$, für welche harte Deadlines existieren, sind periodisch.
2. Alle Prozesse sind unabhängig voneinander.
3. Für alle Prozesse ist $d_i = p_i$.
4. Für alle Prozesse ist die maximale Ausführungszeit c_i bekannt und konstant.
5. Die Zeit für die Prozeßumschaltung kann vernachlässigt werden.
6. Für die Prozessorauslastung durch n Prozesse gilt

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

   Für große n geht $n(2^{1/n} - 1)$ gegen $ln(2)$ (annähernd 0,7).

Der Algorithmus weist den Prozessen aufgrund der Ausführungszeit Prioritäten zu. Der Prozeß mit der kürzesten Ausführungszeit erhält die höchste Priorität usw. Die Priorität ist also eine monoton fallende Funktion der Ausführungszeit. Zur Laufzeit wählt der Dispatcher stets den Prozeß mit der höchsten Priorität.

Rate monotonic scheduling garantiert, daß alle Prozesse ihre Deadline einhalten. Für Einprozessor-Systeme ist der Algorithmus optimal. Der Beweis findet sich bei Liu (siehe [Kop97]). Er basiert auf der Betrachtung sogenannter kritscher Zeitpunkte. Kritische Zeitpunkte sind Zeitpunkte, an denen alle Prozessoranforderungen gleichzeitig eintreffen. Es kann dann zunächst gezeigt werden, daß der Prozeß mit der höchsten Priorität seine Deadline einhält. Anschließend wird der Beweis für den Prozeß mit der zweithöchsten Priorität geführt usw. In einer zweiten Phase wird gezeigt, daß alle Deadlines eingehalten werden, sofern sie bei kritischen Zeitpunkten eingehalten werden.

Falls alle Prozesse eine Periode haben, welche ein Vielfaches der Periode des Prozesses mit der höchsten Priorität ist, so kann auch noch bei hundertprozentiger Auslastung des Prozessors noch ein Schedule gefunden werden, d.h. obige die Voraussetzung 6 kann abgeschwächt werden zu:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$$

Rate monotonic scheduling ist in den letzten Jahren erweitert worden.

Earliest deadline first scheduling (EDF) ist ein unterbrechendes Verfahren, welches den Prozessor stets dem Prozeß zuweist, dessen Deadline als nächstes bevorsteht. EDF ist für Einzelprozessoren optimal und funktioniert auch bei hundertprozentiger Auslastung des Prozessors (theoretisch). Die Prioritäten sind in diesem Fall dynamisch.

Least laxity scheduling (LL) gibt Prozessen mit der kleinsten Differenz $l_i = d_i - c_i$ die höchste Priorität. Für Einprozessor-Systeme ist auch LL optimal.

Für Mehrprozessor-Systeme sind weder EDF noch LL optimal.

6.1.2.2 Abhängige Prozesse

In der Praxis werden Scheduling-Verfahren für voneinander abhängige Prozesse benötigt. Für eine Menge von Prozessen, welche mit Hilfe von Semaphoren den Zugriff auf Ressourcen schützen, zu entscheiden, ob ein Schedule existiert, ist bereits NP-vollständig. Um den Aufwand für das Scheduling so klein wie möglich zu halten, gibt es die folgenden Ansätze:

- Bereitstellung weiterer Hardware-Ressourcen, um das Scheduling einfacher zu gestalten.
- Aufteilung des Scheduling-Problems in zwei Teile. Ein (einfacher) Teil davon wird zur Laufzeit gelöst, ein zweiter Teil bereits vorab. Dieser Weg führt zum statischen Scheduling.
- Einführung vereinfachender Annahmen über die Prozesse.

Beim *kernelized monitor* gehen wir davon aus, daß eine Menge von Prozessen mit kurzen kritischen Abschnitten gegeben ist. Die Abschnitte seien kürzer als eine Konstante q. Das *kernelized monitor*-Verfahren weist Prozessorzeit in ununterbrechbaren Einheiten von q zu, in der Annahme, daß alle kritschen Abschnitte in diesen Zeitabschnitten gestartet und beendet werden können. Der einzige Unterschied zum *rate monotonic scheduling* besteht darin, daß ein Prozeß erst nach einer gewissen Anzahl von Zeitquanten q unterbrochen werden kann. Dieser Unterschied verursacht bereits Probleme.

Beispiel: man betrachte den Fall der Abb. 6.3

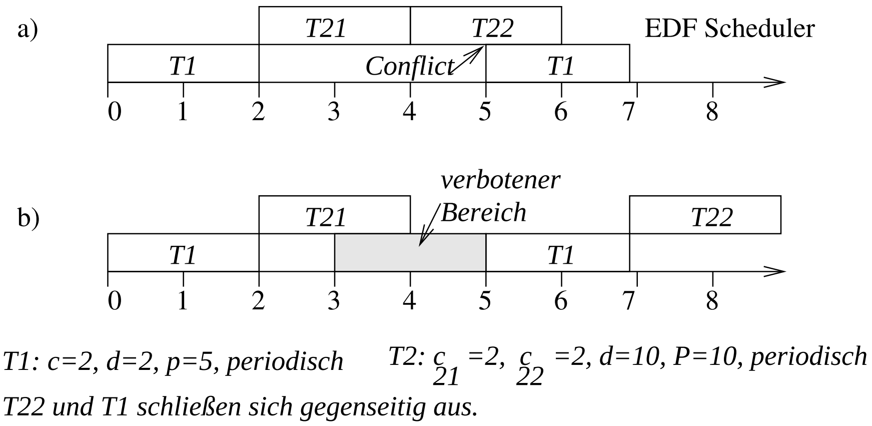


Abbildung 6.3: Verbotene Bereiche

Wir nehmen an, daß $q = 2$ ist. Zur Zeit 4 wird ein EDF Scheduler den Prozeß $T22$ zur Ausführung bringen. Dieser kann danach für zwei Zeiteinheiten nicht unterbrochen werden. Zur Zeit 5 gibt es einen Konflikt, da $T22$ nicht unterbrochen werden kann und da somit $T22$ die Deadline verpassen wird. Einen Ausweg bietet die Reservierung eines verbotenen Bereiches für $T1$. Zur Übersetzungszeit müssen diese verbotenen Bereiche erkannt und dem Dispatcher mitgeteilt werden.

Aufgrund der gegenseitigen Abhängigkeiten zwischen Prozessen kann es zu einer **Prioritätsumkehr** kommen, welche die ursprünglich beabsichtigten Prioritäten vertauscht.

Beispiel: Gegeben seien drei Prozesse $T1, T2$ und $T3$, wobei $T1$ die höchste und $T3$ die niedrigste Priorität haben möge. Es werde *rate monotonic scheduling* benutzt. $T1$ und $T3$ benötigen exklusiven Zugriff zu einer Ressource, welche über eine Semaphore S verwaltet wird. Gegeben sei jetzt eine Situation, in der $T3$ sich im kritischen Abschnitt befindet und rechnet. Wenn jetzt $T2$ die Ausführung verlangt, so wird $T3$ verdrängt, wodurch die Ressource nicht freigegeben werden und somit $T1$ auch nicht zur Ausführung kommen kann. Als Konsequenz verhindert der Prozeß $T2$ mittlerer Priorität die Ausführung des Prozesses $T1$ mit hoher Priorität. Dieses Phänomen heißt **Prioritätsumkehr** (engl. *priority inversion*).

Zur Vermeidung der Prioritätsumkehr wurde das *priority inheritance*-Protokoll definiert. Bei diesem Protokoll wird die Priorität von Prozessen während der Bearbeitung kritischer Abschnitte auf das Maximum der Prioritäten ggf. abhängiger Prozesse gesetzt (siehe dazu auch die Ausführungen zum ADA-Rendezvous im Kapitel 5). Allerdings führt dieses Protokoll zu Deadlocks und Ketten von Blockierungen.

Als Lösung dieser Probleme wurde das *priority ceiling*-Protokoll vorgeschlagen. Bei diesem Protokoll wird jeder Semaphore eine obere Prioritätsschranke zugeordnet. Prozesse, welche eine höhere Priorität haben, können diese Semaphore nicht verwenden. Ein Prozeß kann in einen kritischen Abschnitt nur eintreten, wenn seine Priorität größer ist als die oberen Prioritätsschranken aller durch andere Prozesse gesetzten Semaphore. Alle Prozesse werden mit der zugewiesenen Priorität ausgeführt, es sei denn, sie befinden sich in einem kritischen Abschnitt und blockieren andere Prozesse. In diesem Fall erben sie die höchste Priorität der durch sie blockierten Prozesse.

Ein Beispiel des Ablaufs zeigt Abb. 6.4 [SRL90].

Der Ablauf ist wie folgt:

1. T3 beginnt mit der Ausführung
2. T3 setzt S3
3. T2 wird gestartet und verdrängt T3

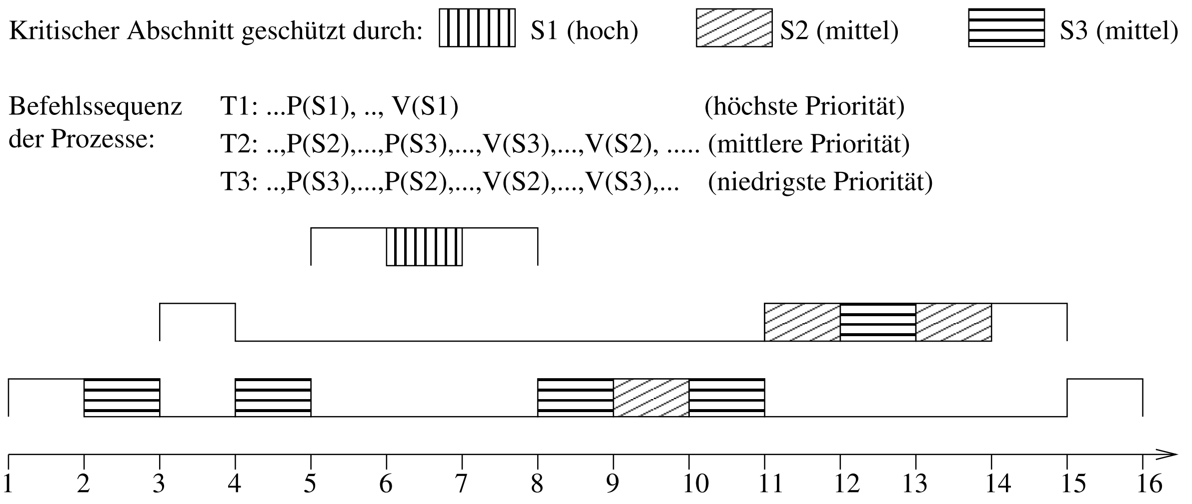


Abbildung 6.4: Beispiel zum *priority ceiling*-Protokoll

4. T2 versucht S2 zu blockieren, wird aber verdrängt, weil die Priorität von T2 nicht größer ist als die Priorität des blockierten S3. T3 führt seinen kritischen Abschnitt mit der ererbten Priorität von T2 weiter aus.
5. T1 wird gestartet und verdrängt T3.
6. T1 setzt die Semaphore S1. Die Priorität von T1 ist höher als die Prioritäten aller gesetzten Semaphoren.
7. T1 setzt die Semaphore S1 zurück.
8. T1 wird beendet. T3 wird mit der Priorität von T2 weiter ausgeführt.
9. T3 setzt Semaphore S2.
10. T3 setzt die Semaphore S2 zurück.
11. T3 setzt die Semaphore S3 zurück und kehrt zur niedrigen Priorität zurück. T2 kann jetzt S2 setzen.
12. T2 setzt auch noch S3.
13. T2 nimmt S3 zurück.
14. T2 nimmt S2 zurück.
15. T2 terminiert, T3 nimmt die Ausführung wieder auf.
16. T3 terminiert.

Sofern zwischen Prozessen gegenseitige Abhängigkeiten bestehen, so kann bereits für Einzelprozessor-Systeme das Einhalten der Zeitbedingungen nur schwer überprüft werden. Noch schwieriger ist es, dies für Mehrprozessor-Systeme zu tun. Viele aktuelle Forschungsarbeiten zielen auf eine solche Überprüfung [Kop97].

6.1.3 Statisches Scheduling

Beim statischen Scheduling wird die Reihenfolge der Abarbeitung von Prozessen bereits während der Programmentwicklung festgelegt. Dabei müssen Reihenfolgebedingungen (*precedence constraints*) und gegenseitiger Ausschluß berücksichtigt werden. Reihenfolgebedingungen werden dabei durch **Präzedenzgraphen** ausgedrückt. Abb. 6.5 zeigt ein Beispiel.

Im Falle eines statischen Schedules wird der vollständige Ablauf der Prozesse während der Softwareentwicklung geplant und dem Dispatcher in Form einer Tabelle mitgeteilt. Diese Tabelle enthält Zeitpunkte

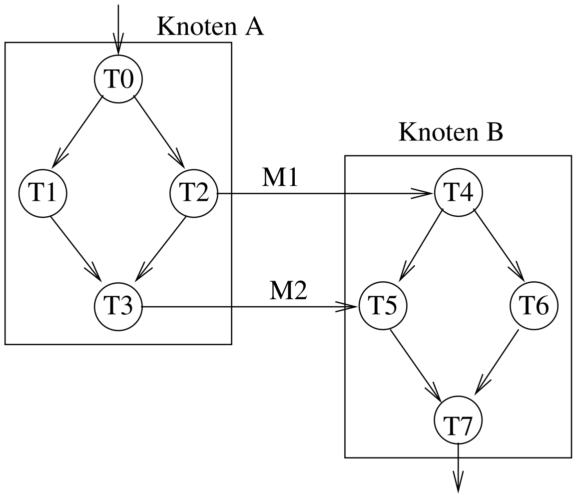


Abbildung 6.5: Präzedenzgraph für ein verteiltes System mit zugeordneten Prozessoren

und die zugehörigen Aktionen. Zu jedem dieser Zeitpunkte erfolgt ein Interrupt durch einen Timer. Dieser Timer bildet die einzige Interruptursache des Systems. Zeiten werden dabei in Form von Vielfachen einer kleinsten Zeit angegeben.

Nach einer gewissen Zeit wiederholt sich der geplante Ablauf. Diese Zeit nennt **Schedule-Periode**.

Statisches Scheduling kann für alle möglichen Formen von Zielarchitekturen angewandt werden. Optimales Scheduling ist aber für praktisch alle interessanten Fälle schon für Einprozessor-Systeme NP-vollständig.

Eine Möglichkeit der systematischen Suche von Schedules ist der Suchbaum (siehe Abb. 6.6).

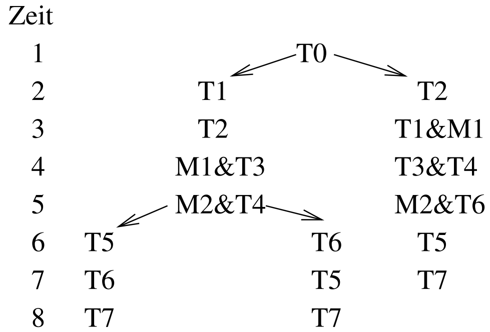


Abbildung 6.6: Suchbaum für den o.a. Präzedenzgraphen

Dieser Baum basiert auf der vereinfachenden Annahme gleicher Prozeßausführungszeiten und Kommunikationszeiten. Es kann ausgenutzt werden, daß ein minimales Schedule gesucht wird, also möglichst viele Prozesse gleichzeitig ausgeführt werden sollen.

Behandlung sporadischer Prozesse

Es gibt mehrere Methoden der Behandlung sporadischer Prozesse im Zusammenhang mit statischem Scheduling:

1. Transformation eines sporadischen Prozesses in einen periodischen Prozeß
2. Einführung eines Server-Prozesses für sporadische Prozesse
3. Umschaltung zwischen verschiedenen Betriebsarten

Wir schließen diesen Abschnitt mit einem Zitat von Xu und Parnas (nach Kopetz):

For satisfying timing constraints in hard real-time systems, predictability of the systems behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system.

6.2 Validierung

14. Vorl.

Validierung beschäftigt sich mit der Frage “Ist dieses System für seinen Zweck geeignet?”. Sicherheitskritische Systeme sollten nur dann in Betrieb genommen werden, werden ihre Sicherheit überzeugend nachgewiesen worden ist. Bei gegenwärtigen Stand der Technik muß ein solcher Nachweis durch eine Kombination fast aller möglichen Methoden der Sicherheitsüberprüfung geführt werden. Ein einziger Nachweis allein reicht nicht aus. In diesem Abschnitt werden wir uns mit verschiedenen Methoden beschäftigen[2]. In Unterabschnitten werden wir uns u.a. mit dem Sicherheitsnachweis im allgemeinen, mit formalen Methoden, mit testbasierten Methoden, mit der Fehlerinjektion und mit der Risikoanalyse beschäftigen.

6.2.1 Simulation

Ein Grundproblem moderner komplexer Systeme besteht darin, daß es bereits sehr schwierig ist, ein System so zu spezifizieren, daß eine Realisierung des Systems zu einem sicheren System, welches den Ansprüchen der Benutzer genügt, führen würde. Im Falle ausführbarer Spezifikationen kann man diese im Prinzip bereits zu einem frühen Zeitpunkt ausführen, die Eigenschaften des späteren EIS also **simulieren**.

Allerdings bereitet die Simulation von EIS spezifische Probleme:

- Simulationen laufen in der Regel wesentlich langsamer ab als bei der späteren Implementierung. Dies bedeutet, ein vorgegebenes Realzeitverhalten kann nicht eingehalten werden. **Zeitspezifikationen werden also verletzt**.
- Die Simulation müßte in der physikalischen Umgebung erfolgen. **Eine Simulation in der physikalischen Umgebung ist aber problematisch**.
- Simulationen werden meist um Zehnerpotenzen langsamer ausgeführt werden als in Echtzeit. **Damit können auch nicht ausreichend viele Eingabedaten verarbeitet werden.**

6.2.2 Rapid Prototyping, Emulation

Aus diesem Grund ist es inzwischen vielfach üblich, aus Spezifikationen sehr schnell Implementierungen zu generieren, anhand derer das Verhalten der endgültigen Implementierung beurteilt werden kann. Ein derartiges *Rapid Prototyping* wird meist Implementierungen verwenden, welche überhaupt nicht effizient sind und bei denen eventuell auch in kleinem Umfang Einschränkungen hinsichtlich des Einhaltens der Zeitbedingungen akzeptiert werden müssen. Das spätere System wird mit Rapid Prototyping-Systemen **emuliert**. Hierfür setzen sich FPGA-basierte Lösungen mehr und mehr durch. Es gibt inzwischen verschiedene Hersteller, welche große Prototypensysteme mit Tausenden von FPGAs vertreiben. Ihr Preis kann im Millionen-DM-Bereich liegen. Die Ausbildung von Elektroingenieuren an ‘kleinen’ Versionen solcher Systeme ist inzwischen durchaus üblich (sofern die Universität hierfür die Mittel aufbringen kann).

Auch in der Automobilindustrie ist es üblich, neue Steuerungsalgorithmen zunächst einmal auf FPGAs-Basis in echten Fahrzeugen zu erproben. Dazu werden im Kofferraum Einschubsysteme mit Hunderten von FPGAs eingebaut und das Auto wird auf Testfahrten geschickt, z.B. in große Höhen und eisige Regionen, um so die Motorsteuerung auszutesten.

Mit v.a. ideeler Unterstützung durch die Automobilindustrie hat die DFG ein Schwerpunktprogramm *Rapid Prototyping* aufgelegt, um in diesem Bereich die Forschung zu unterstützen. Eines der Probleme im *Rapid Prototyping* liegt darin, daß Prozessor-basierte Entwürfe auch heute noch meist eine Programmierung in Assembler verlangen, da die vorhandenen Compiler zu schlecht auf Architekturen und Anwendungen eingebetteter Systeme angepaßt sind und als Folge davon zu ineffizienten Code liefern. Ein Kernbereich der Arbeiten des Lehrstuhls 12 liegt in der Entwicklung von Compilern, welche die Anwendungen und Architekturen eingebetteter Systeme berücksichtigen und damit konkurrenzfähigen Code erzeugen. Aus diesem Grund ist der Lehrstuhl am Schwerpunktprogramm beteiligt.

[2] Für diesen Abschnitt wird v.a. Stoff des Kapitels 12 des Buches von Kopetz [Kop97] verwendet.

6.2.3 Sicherheitsnachweise

Beim Sicherheitsnachweis muß gegenüber einer externen Instanz überzeugend dargelegt werden, warum ein entworfenes eingebettetes System verläßlich ist. Dabei werden sowohl mögliche interne Fehler wie auch Fehler, die aus der Umgebung resultieren, beurteilt. Interne Fehler kann es aus zwei Gründen geben: Hardwareausfälle und unentdeckte Entwurfsfehler. Aufgrund der Vielzahl möglicher Fehler müssen beim Sicherheitsnachweis kombinierte Methoden eingesetzt werden.

6.2.3.1 Geforderte Systemeigenschaften

Folgende Systemeigenschaften werden generell gefordert:

- Der Ausfall einer einzigen Komponente darf nicht zu einem katastrophalen Fehler führen.

  Komponenten von EIS, welche in dieser Hinsicht kritisch sind, sind u.a. die folgenden:

  1. Die zentrale Uhr
  2. Die Stromversorgung und die Masseverbindung
  3. Entwurfsfehler, die in allen Knoten repliziert wurden.
  4. Fehler in der zentralen Kommunikation, z.B. über einen zentralen Bus oder über einen Einzelring.

- Jeder Einzelfehler darf nur eine eng begrenzte Region des System (sog. *error containment region*) beeinflussen.
- Im Falle eines Fehler muß entweder das System einen sicheren Ruhezustand erreichen (sog. *fail-safe application*) oder das System muß weiterhin seine normale Aufgabe erfüllen (sog. *fail-operational application*).
- *composability*: Komponenten des Systems müssen kombiniert werden können, ohne daß Fehlersituationen in einer Komponente zu Fehlern in anderen Komponenten führen[3].

6.2.4 Formale Methoden

Formale Methoden, welche die Verläßlichkeit eines EIS mit Hilfe der gesicherten Methoden der Mathematik nachweisen, werden schon seit vielen Jahren erforscht. Der Pentium-Gleitkommafehler hat diesen Methoden eine vorher nie gekannte Aufmerksamkeit verschafft und die Mehrzahl der amerikanischen Spezialisten auf diesem Gebiet wurde umgehend von der Fa. Intel eingestellt.

Formale Methoden basieren zunächst auf den folgenden Schritten:

1. **Erzeugung eines konzeptuellen Modells:** In dieser Phase muß ein Modell der realen Welt erstellt werden. Üblicherweise geschieht dies in natürlich-sprachlicher Form. Alle Unterlassungen in dieser Phase werden von den späteren Phasen nicht mehr korrigiert.
2. **Formalisierung des Modells:** In dieser Phase wird das konzeptuelle Modell in die Sprache der Mathematik übertragen.
3. **Analyse des Modells:** Das formale Modell kann in dieser Phase untersucht werden. Dafür gibt es unterschiedliche Methoden: von Hand, rechnerunterstützt interaktiv und vollautomatisch.
4. **Interpretation der Ergebnisse**.

Nur der Schritt 3 kann wirklich automatisiert werden.

[3] Dies mag ein Grund für die große Zahl von Prozessoren in Autos sein: würde man Prozessoren für mehrere Aufgaben nutzen, was im Prinzip möglich wäre, so wäre die hier geforderte Eigenschaft nicht mehr erfüllt.

6.2.5 Testen

6.2.5.1 Rolle des Testens in der Validierung

Das Testen eines Hardwaresystems dient normalerweise zwei möglichen Zielen

- **dem Erkennen von Herstellungsfehlern**

  Dies ist erforderlich, da Hardware nicht mit großer Sicherheit reproduziert werden kann.

- **dem Erkennen von Hardwareausfällen**

  Hardwareausfälle können durch Testprogramme sowohl im normalen Betrieb (sog. *on-line testing*) wie auch in einem separaten Betriebsmodus (sog. *off-line testing*) erkannt werden.

In der Systemvalidierung versucht man, sich die reichen Erfahrungen im Testbereich für das Überprüfen an sich fehlerfrei gefertigter, aber möglicherweise falsch entworfener Systeme zu nutze zu machen. Hierdurch wachsen der Entwurfsbereich und der Testbereich enger zusammen. Eine der Hoffnungen ist, daß Testmuster, welche Herstellungsfehler finden, mit großer Wahrscheinlichkeit auch Entwurfsfehler aufdecken.

6.2.5.2 Testmustererzeugung

Während des Testens kann nur ein kleiner Teil möglicher Testmuster verwandt werden. Es sind viele Methoden der Auswahl von Testmuster vorgeschlagen worden. Die Auswahl sollte mindestens die folgenden drei Situationen herbeiführen:

1. **Spitzenlast:** Für viele Systeme ist die maximale Systemlast eine problematische Situation, die sicher behandelt und daher auch getestet werden sollte.
2. *worst case execution time:* Durch Analyse des Quell- oder Objektcodes kann herausgefunden werden, bei welchen Eingaben die größten Laufzeiten entstehen. Diese Eingaben sollten in der Testmustermenge enthalten sein.
3. **Test der Fehlertoleranz:** Auch das Überprüfen der Fehlertoleranzmaßnahmen sollte im Testmustersatz enthalten sein.
4. **Test der arithmetischen Genauigkeit**.

6.2.6 Testfreundlicher Entwurf

Bereits in der Entwurfsphase muß berücksichtigt werden, daß das System zu testen ist. Hardwaremäßig gibt es hierzu viele Maßnahmen:

- *Scan design:* schon vor vielen Jahren ist vorgeschlagen worden, alle Register eines Hardwarebausteins als Schieberegister zu realisieren. In einem Testmodus kann man so alle Register über eine serielle Eingabe- und eine serielle Ausgabeleitung setzen und auslesen. Auf diese Weise kann man mit Techniken für das Testen kombinatorischer Schaltungen arbeiten und benötigt keine besonderen Methoden für das Erzeugen von Prüfmustern für sequentielle Logik.

  Diese Methode wird für IBM-Großrechner seit vielen Jahren eingesetzt.

  Ein Nachteil des *scan designs* ist der größere Hardwareaufwand. Auch kann der Zeitaufwand für das Lesen und Schreiben einer einzigen *scan chain* hoch sein.

- *JTAG: scan design*-Techniken regeln zunächst das Testen innerhalb eines Chips. Es stellt sich die Frage, wie man bei Integration von vielen Chips auf einem Board die *scan chains* miteinander kooperieren läßt. Beantwortet wird diese Frage durch die Einführung des JTAG-Standards. JTAG definiert Register an den Rändern aller Chips (daher auch der Name *boundary scan*), welche über einen kleinen Test-Controller auf JTAG-fähigen Chip kontrolliert werden können. Alle JTAG-Chips können standardisiert untereinander verbunden werden.

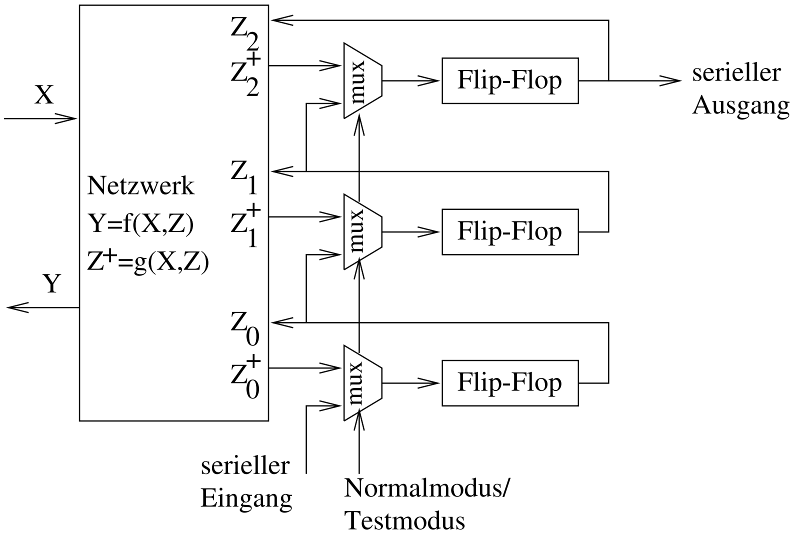


Abbildung 6.7: Mealy-Automat mit *scan path*

- *Integrierte Prüfmuster-Erzeugung und -Kompaktierung:* Um das aufwendige externe Erzeugen und Zuführen von Prüfmustern zu sparen, versucht man Prüfmuster möglichst gleich intern im System zu erzeugen. Hierzu setzt man gern linear zurückgekoppelte Schieberegister ein, welche bei geeigneter Verschaltung Pseudo-Zufallszahlen generieren, was für viele Schaltungen sinnvolle Tests ergibt.

  Um auch das aufwendige Ausgaben der Antwort des Systems auf die Prüfmuster zu sparen, werden die Antworten vielfach wiederum in linear zurückgekoppelten Schieberegistern (engl. *linear feedback shift registers*, LFSR) kompaktiert (es wird eine Art CRC berechnet). Die Wahrscheinlichkeit dafür, daß die kompaktierte Antwort mit der erwarteten übereinstimmt, obwohl die Antworten von den erwarteten abweichen, ist relativ gering ($2^{-\text{Länge des Schieberegisters}}$). Die berechnete kompaktierte Antwort kann seriell ausgelesen und mit dem erwarteten Wert verglichen werden.

  Abb. 6.8 zeigt drei Register, welche für die Testdaten-Erzeugung, Testdaten-Kompression und das serielle Auslesen geeignet sind. Tabelle 6.1 enthält die dazugehörigen Betriebsmodi.

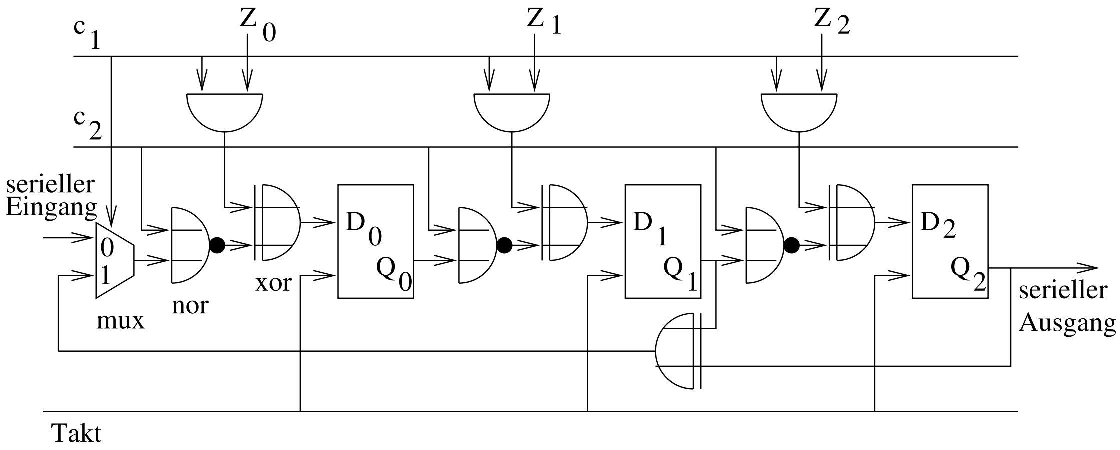


Abbildung 6.8: BILBO-Register

- *Weitere:* Es gibt viele weitere Methoden wie beispielsweise das konsequent synchrone Takten, sowie Techniken in der Logikminimierung.

6.2.7 Fehlersimulation

Eine vollständige analytische Behandlung des Verhaltens von Systemen bei Auftreten von Fehlern ist heute nicht erreichbar. Aus diesem Grund kann man das Verhalten von EIS bei Anwesenheit von Fehlern

c_1	c_2		
'0'	'0'	$'0' \oplus \overline{Q_{i-1}} = \overline{Q_{i-1}}$	Scan Path Mode
'0'	'1'	$'0' \oplus \overline{'1'} =' 0'$	Reset
'1'	'0'	$Z_i \oplus \overline{Q_{i-1}}$	Multi Input LFSR
'1'	'1'	$Z_i \oplus \overline{'1'} = Z_i$	Normaler Modus

Tabelle 6.1: Betriebsmodi der BILBO-Register

simulieren. Man modifiziert dafür das Modell des EIS entsprechend dem gerade zu betrachtenden Fehler und simuliert mit den Eingabewerten. Ein Ziel ist dabei, zu erkennen, ob ein konkreter Fehler sich tatsächlich auch auf der Ebene des Gesamtsystems als Fehler äußert oder ob er eventuell maskiert wird.

Wegen der Vielzahl möglicher Fehler ist die Fehlersimulation natürlich noch zeitaufwendiger als die normale Simulation. Zur Beschleinigung macht man sich vielfach zunutze, daß die Fehlersimulation auf Gatterebene erfolgt und damit pro Maschinenwort des Gastrechners nur 1 Bit belegt. Da sich die meisten Fehler nur in unterschiedlichen Bitmustern äußern, der Kontroll- und Datenfluß im Simulationsprogramm aber gleich bleibt, nutzt man dies für die **parallele Fehlersimulation**. Jedem Bit eines Maschinenworts entspricht dabei ein anderer Fehler un d man erhält je nach Wortbreite des Gastrechners eine erhebliche Beschleunigung.

6.2.8 Fehlerinjektion

Für EIS bleibt die Fehlersimulation allerdings aus Geschwindigkeitsgründen meist inakzeptabel. Als Alternative wird daher die **Fehlerinjektion** benutzt. Bei dieser Technik werden künstlich Fehler in (echten oder emulierten) Systemkomponenten generiert und das Verhalten des Systems im Falle solcher Fehler wird analysiert.

Es gibt zwei Anwendungen einer derartigen Fehlerinjektion:

1. **Prüfen des Auftretens interner Fehler:**

   In diesem Fall wird angenommen, daß bestimmte interne Fehler auftreten und es wird beobachtet, wie sich das System verhält, wie sich also beispielsweise Fehlertoleranzmaßnahmen auswirken.

2. **Beurteilung der Verläßlichkeit bei Auftreten von Fehlern in der Umgebung:**

   In diesem Fall wird angenommen, daß sich die Umgebung anders als ursprünglich spezifiziert verhält. Beispielsweise kann man erproben, was bei zu geringen zeitlichen Abständen von Startanforderungen für sporadische Prozesse passiert.

6.2.8.1 Hardwaremäßige Fehlerinjektion

Hardwaremäßige Fehlerinjektion ist aufwendig, gibt aber recht präzise Aussagen über das Verhalten des echten Systems. Kopetz [Kop97] berichtet über drei Fehlerinjektions-Techniken, welche in dem ESPRIT-Projekt MARS untersucht wurden: Bestrahlung mit Schwermetall-Ionen, Manipulation auf Pin-Ebene und Elektromagnetische Strahlung. Charakteristische Eigenschaften dieser drei Methoden enthält die Tabelle 6.2.

	Schwermetall-Ionen	Manipulation auf Pin-Ebene	Elektromagnetische Strahlung
Räumliche Kontrollierbarkeit	gering	hoch	gering
Zeitliche Kontrollierbarkeit	keine	hoch/mittel	gering
Flexibilität	gering	mittel	hoch
Reproduzierbarkeit	mittel	hoch	gering
Physikalische Erreichbarkeit	hoch	mittel	mittel
Zeitmessung	mittel	hoch	gering

Tabelle 6.2: Eigenschaften verschiedener Fehlerinjektionstechniken

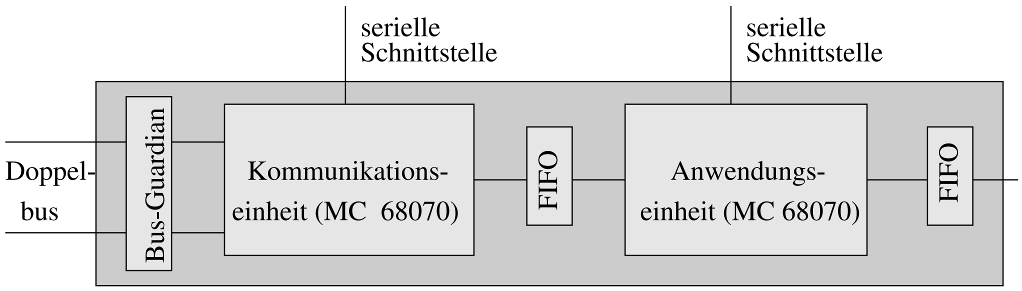


Abbildung 6.9: Knoten der experimentellen Hardware

Mit diesen drei Techniken wurde die Hardware der Abb. 6.9 Experimenten unterzogen.

Die Kommunikationseinheit realisiert ein zeitgesteuertes (TT-) Protokoll. Die *Bus-Guardian* genannte Einheit dient als Schutz gegen eine Kommunikationseinheit, die illegal (mit falschem Timing oder unberechtigt in Form von *'babbling nodes'*) den Bus benutzt.

Drei verschiedene Fehlererkennungsmechanismen sind in dem System enthalten:

- **Hardware-Mechanismen:**

  Die üblichen Hardware-Mechanismen wie *traps* für illegale Befehle, ungültige Adressen, FIFO-Überlauf, *Bus-Guardian*.

- **System-Software:**

  Vom Compiler erzeugtes *assertion testing*, Test der WCET von Realzeit-Prozessen.

- **Anwendungssoftware:**

  Verdoppelte Ausführung, begrenzte verdreifachte Ausführung, CRC über alle Ebenen (d.h., bis zu den Anwendungsprogrammen und nicht nur in der Hardware).

Der in Abb. 6.9 gezeigte Prozessorknoten ist Teil des in Abb. 6.10 gezeigten Testaufbaus.

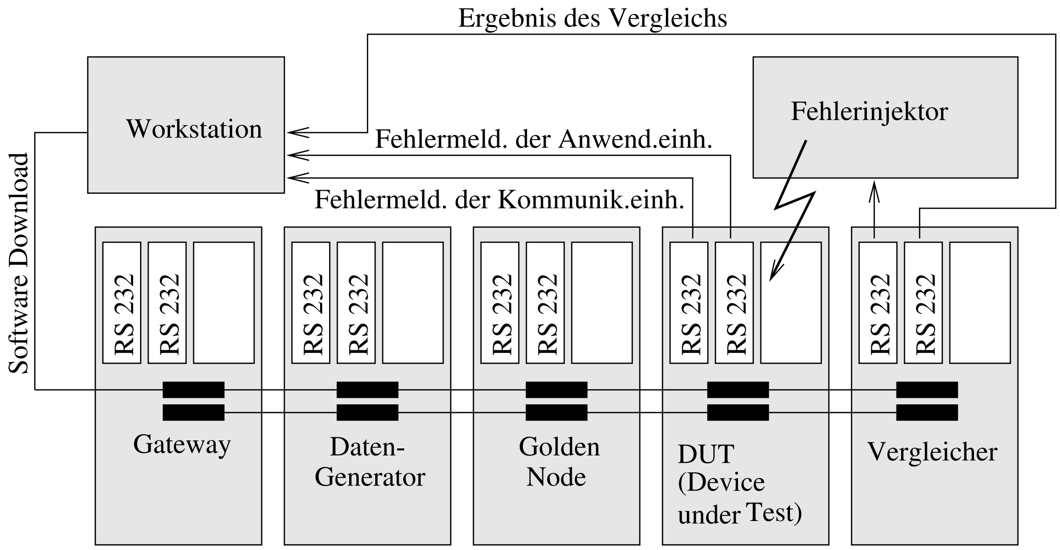


Abbildung 6.10: Testaufbau

Die Eingabewerte werden sowohl von dem zu testenden Prozessorknoten wie auch von dem 'goldenen' stets als korrekt angenommenen Knoten empfangen. Der Vergleicher kann die Anworten der beiden Knoten miteinander vergleichen und das Ergebnis zur Workstation senden. Gleichzeitig können auch die Fehlermeldungen des zu testenden Knotens (DUT) zur Workstation übertragen werden. Wünschenswert sind natürlich Prozessorknoten, die alle vom Vergleicherknoten bemerkten Fehler auch selbst bemerken (keine sog. *fail-silence violations*). Die Frage war, ob der DUT dies leistet.

Das Experiment ergab die folgenden Ergebnisse:

- Wenn alle Fehlererkennungmaßnahmen eingeschaltet waren, dann gab eine keine *fail-silence violations.*
- Die zweifache Ausführung von Prozessen und die CRC-Absicherung über alle Ebenen war für alle Fehlerinjektionsmethoden erforderlich, um eine Fehlererkennung von über 99% zu erreichen.
- Im Falle der Bestrahlung mit Schwermetall-Ionen war die Dreifach-Ausführung erforderlich, um eine 100%-ige Erkennung zu erreichen. In den beiden anderen Fällen reichte hierfür die doppelte Ausführung.
- Die *Bus Guardian*-Einheit war in allen Fällen notwendig, um eine Erkennung von 99% zu erreichen. Sie schloß den kritischsten Fehler aus, die *babbling idiots* [Kop97].

6.2.8.2 Softwaremäßige Fehlerinjektion

Bei der softwaremäßigen Fehlerinjektion werden Speicherzellen modifiziert. Dabei kann es sich um Programm- oder Datenspeicher handeln. Es können auf diese Weise sowohl Fehler im Ablauf wie auch in den Daten erzeugt werden. Die softwaremäßige Fehlerinjektion hat gegenüber der hardwaremäßigen Fehlerinjektion eine Reihe von Vorteilen:

- **Vorhersagbarkeit/Reproduzierbarkeit:**

  Im Gegensatz zu Versuchen mit radioaktiven oder elektromagnetischen Strahlen sind die Experimente reproduzierbar.
- **Erreichbarkeit:**

  Bei softwaremäßiger Fehlerinjektion sind auch interne Register gezielt zu erreichen.
- **Aufwand:**

  Der Aufwand ist geringer als bei hardwaremäßiger Fehlerinjektion.

Fuchs (zitiert nach Kopetz) hat in Experimenten mittel Negieren einzelner Speicherzellen festgestellt,

- daß die Fehlerabdeckung bei beiden Injektionsmethoden vergleichbar ist,
- daß bei softwaremäßiger Fehlerinjektion ein größerer Anteil an Fehlern in der Anwendungssoftware erkannt wird, wohingegen bei hardwaremäßiger Fehlerinjektion die Fehler fast vollständig in der Hardware und von der System-Software erkannt werden.
- daß bei ausgeschalteter Fehlererkennung in der Anwendungssoftware und einfacher Ausführung der Prozesse die softwaremäßige Fehlerinjektion eine größere Anzahl von *fail-silent failures* verursacht als die Injektion elektromagnetischer Strahlung oder Manipulation auf Pin-Ebene.

Die Schlußfolgerung aus diesen Experimenten ist, daß die softwaremäßige Fehlerinjektion mit der hardwaremäßigen Fehlerinjektion vergleichbar ist. Ausgenommen ist allerdings die radioaktive Bestrahlung, da diese schwerwiegendere Fehler erzeugt.

6.2.9 Risiko- und Zuverlässigkeitsanalyse

In sicherheitskritischen Anwendungen wird verlangt, daß die Wahrscheinlichkeit eines katastrophalen Fehlers höchstens bei 10^{-9}/Stunde liegt (entsprechend ca. einem Fehler bei 100.000 Systemen mit einer Betriebszeit von 10.000 Stunden). In vielen Fällen (insbesondere im Bereich der Kernenergie) muß die Wahrscheinlichkeit noch deutlich geringer liegen.

Schäden resultieren aus unsicheren Zuständen (engl. *hazards*). *Hazards* haben eine Schwere und eine Wahrscheinlichkeit. Das Produkt aus Schwere und Wahrscheinlichkeit heißt **Risiko** (dies ist die Größe, die bespielsweise von Versicherungen betrachtet wird). Risiko-Minimierung ist eine der Hauptaufgaben einer ingenieursmäßigen Systemkonstruktion.

Fehlerbaum

Eine der Techniken für die Risiko-Beurteilung ist der **Fehlerbaum**. Bei der Aufstellung eines Fehlerbaums beginnt man mit einem möglichen Fehler. Dieser bildet die Wurzel des Baumes. Anschließend wird analysiert, aus welchen Gründen es zu diesem Fehler kommen könnte. Diese Analyse betrachtet im allgemeinen Konjuktionen und Disjunktionen Vorraussetzungen, unter denen es zu Fehlern kommen kann. Diese Konjunktionen und Disjunktionen bilden die inneren Knoten von Fehlerbäumen. Die Blätter werden durch die eigentlichen Vorraussetzungen gebildet (siehe Abb. 6.11).

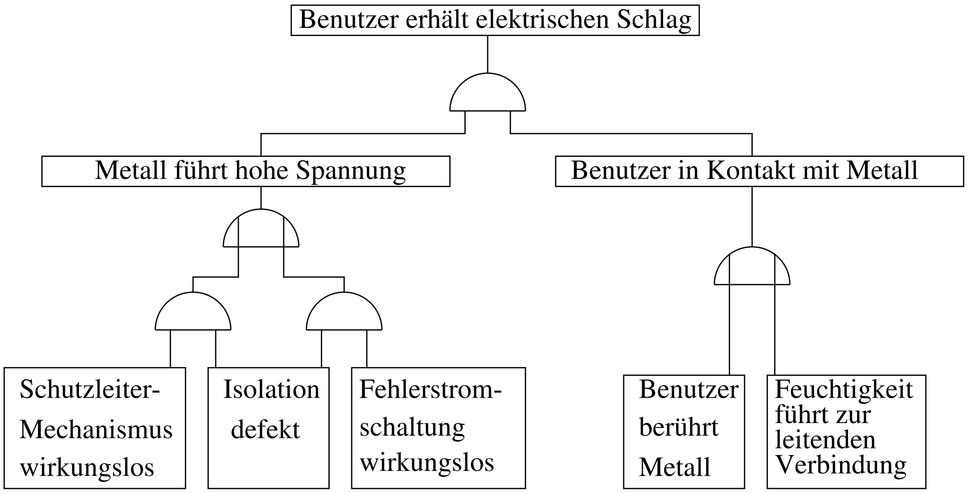


Abbildung 6.11: Fehlerbaum

Fehlerbäume können mit formalen Techniken analysiert werden, z.B. mit Methoden der Wahrscheinlichkeitsrechnung.

Nicht alle Systeme können hinreichend genau mit Fehlerbäumen beschrieben werden. Gemeinsam benutzte Ressourcen, abnehmende Größen nutzbarer Speicherbereiche etc. erfordern komplexere Techniken. Zum Teil ist in diesen Fällen das Modell der Markov-Ketten angebracht.

failure mode and effect analysis

Bei der Fehlerbaum-Methode geht man von dem möglichen Fehlverhalten des Gesamtsystems aus und versucht, mögliche Ursachen in den Teilsystemen zu finden. Auch der umgekehrte Weg ist sinnvoll und bildet eine Ergänzung. Bei der Methode *failure mode and effect analysis* (FMEA) geht man von den einzelnen Elementen aus. Man stellt dabei zunächst Tabellen im Format der Tabelle 6.3 auf.

Komponente	Fehlerursache	Wirkung	Wahrscheinlichkeit	kritisch ?
Speicher	Alpha-Strahlung	betroffene Bits ='1'	10^{-12} Chip/Stunde	nein (ECC)

Tabelle 6.3: FMEA-Tabelle

In einem zweiten Schritt analysiert man die Auswirkung der Komponentenfehler auf das Gesamtsystem. Hierfür gibt es lt. Kopetz erste Werkzeuge [Kop97].

Softwarefehler

Eine vollständige formale Verifikation von komplexer Software ist bislang möglich ist. Formal abgesicherte Konstruktion wird bisher nur in wenigen Fällen benutzt. Als Alternative hat man versucht, Daten aus dem Software-Entwurfsprozeß zu gewinnen. Beispielsweise kann man jeden entdeckten Fehler dokumentieren und die Häufigkeit der Entdeckung über der Zeit protokollieren. Aus diesen Daten kann man versuchen, Aussagen über die Anzahl unentdeckter Fehler abzuleiten. Dies wird u.a. im Software-Sicherheitsstandard IEC-1508, Teil 5, so vorgesehen.

Allerdings ist dieser Ansatz für hohe Sicherheitsanforderungen weniger geeignet, denn wir sind auf so geringe Fehlerraten angewiesen, daß statistische Aussagen keinerlei Wert haben.

Kapitel 7

Prozeß-Steuerungen und -Regelungen

7.1 Einführung

Informationsverarbeitung wird in EIS vielfach eingesetzt, um technische Prozesse zu steuern und zu regeln. Von einer **Steuerung** sprechen wir, wenn die Parameter eines technischen Systems von außen beeinflußt werden, ohne daß die Reaktion des Systems Rückwirkung auf diese Beeinflussung hat.

Ein Beispiel ist ein Heizkörperventil, welches keinen Thermostaten enthält. Die Öffnung des Ventils bleibt in diesem Fall unabhängig von der möglicherweise viel zu kalten oder heißen Zimmertemperatur.

Eine Vorrichtung, mit der man ein technisches System beeinflussen kann, heißt **Stellglied**. Das Ventil eines Heizkörpers ist ein solches Stellglied.

Einrichtungen, welche ihrerseits die Stellglieder einer Steuerung beeinflussen, heißen **Steuerglieder**. In unserem Beispiel ist der Mensch ein solches Steuerglied (sofern er sich nicht von der Raumtemperatur lenken läßt).

Ein allgemeines Blockschaltbild einer Steuerung zeigt die Abb. 7.1.

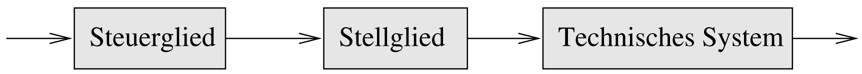


Abbildung 7.1: Steuerung

Das Beispiel des Heizkörperventils zeigt bereits, daß reine Steuerungen für viele Anwendungen unzureichend sind. Man sieht daher vor, daß man die technische Größe, welche durch das Stellglied beeinflußt wird, mittels eines Sensors auch wieder erfaßt, in einem **Regler** Soll- und Istwert dieser Größe geeignet verarbeitet und das Stellglied mit dem gewonnenen Wert beeinflußt. Abb. 7.2 zeigt ein Strukturbild eines technischen Regelkreises [Kie97].

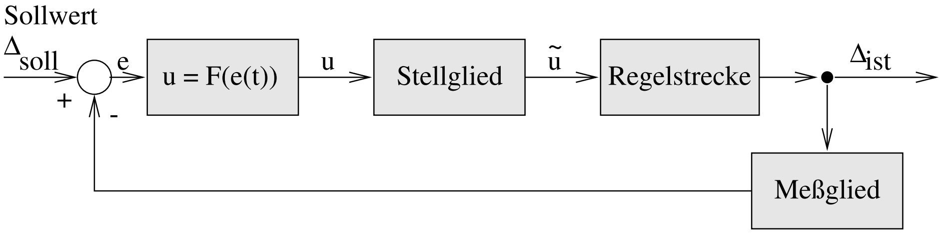


Abbildung 7.2: Regelkreis

Bei den Zielen einer solchen Regelung können wir zwischen dem sog. **Konstanthaltungsproblemen** und dem sog. **Folgeregelungsproblem** unterscheiden.

Bezüglich der Verläßlichkeit von Reglern gelten praktisch dieselben Anforderungen wie für EIS sonst auch.

Eine Beschränkung, die für praktisch alle Regler eingehalten werden muß, ist die Stellgrößenbeschränkung

$$u_{min} \leq u \leq u_{max}$$

7.2 Klassische Regler

Regler können anhand der von ihnen berechneten Funktion F unterschieden werden.

Wohl der einfachste Regler ist der **Zweipunktregler**. Sein Verhalten kann durch die Funktion

$$u(e) = \begin{cases} u_{max}, & \text{für } e > 0 \\ u_{min}, & \text{sonst} \end{cases} \tag{7.1}$$

Diese Funktion wird in der Abb. 7.3 dargestellt.

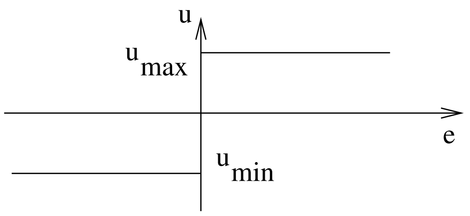


Abbildung 7.3: Kennlinie des Zweipunktreglers

Ein Nachteil dieses Reglers ist sein häufiges Hin- und Herschalten. Dieser Nachteil läßt sich mit dem **Dreipunktregler** beheben. Dieser berechnet die Funktion

$$u(e) = \begin{cases} u_{max}, & \text{für } e > +c \\ u_{min}, & \text{für } e < -c \\ 0, & \text{sonst} \end{cases} \tag{7.2}$$

Diese Funktion wird in Abb. 7.4 dargestellt.

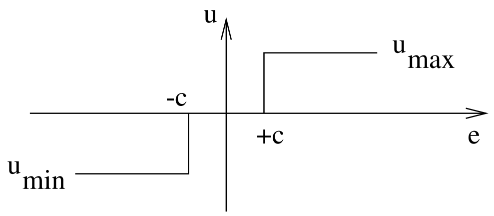


Abbildung 7.4: Kennlinie des Dreipunktreglers

Weitere wichtige Regelstrategien sind (in der Formulierung von Kiendl [Kie97]):

1. 'Je größer die Regelabweichung, desto größer die Gegenreaktion'.
   In diesem Fall gilt

   $$u = c_1 e.$$

   mit der wählbaren Konstanten c_1. Wir nennen diesen Regler einen **Proportionalregler** (P-Regler).

2. 'Je länger eine Regelabweichung bereits angedauert hat, desto größer die Gegenreaktion'.
   In diesem Fall gilt
$$u = c_2 \int_0^t e(t')\ dt'$$
   mit der wählbaren Konstanten c_2. Wir nennen diesen Regler **Integralregler** (I-Regler).
3. 'Je größer die Änderungstendenz der Regelabweichung, desto größer die Gegenreaktion'.
   In diesem Fall gilt
$$u = c_3 \frac{d}{dt} e$$
   mit der wählbaren Konstanten c_3. Wir nennen diesen Regler **Differentialregler** (D-Regler).

Ein PID-Regler ist ein Regler, der Anteile aller drei Regler enthält:

tab
$$u = c_1 e + c_2 \int_0^t e(t')\ dt' + c_3 \frac{d}{dt} e$$

Ein Strukturbild eines PID-Reglers zeigt die Abb. 7.5.

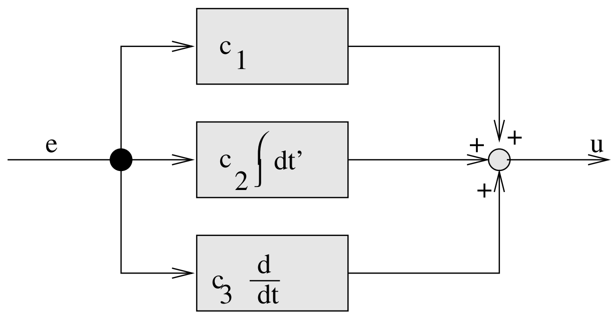


Abbildung 7.5: Struktur eines PID-Reglers

PID-Regler und deren Bestandteile heißen **lineare Regler**, weil für sie das Superpositionsprinzip gilt.

Die Wahl der Parameter c_1, c_2 und c_3 ist entscheidend für das Funktionieren eines PID-Reglers. Wählt man die Verstärkung c_1 zu groß, dann gerät der Regler in Schwingungen. Die Parameter c_2 und c_3 bestimmen, wie empfindlich der Regler auf längerfristige bzw. auf kurzfristige Regelabweichungen reagiert. Wählt man diese Parameter zu groß, so kommt es ebenfalls zu Schwingungen. Wählt man die Parameter zu klein, so erfolgt eine zu geringe Ausregelung. Techniken für die Wahl der Parameter basieren meist auf dem Umrechnung der c_i in einen Parametersatz

$$c_1 = K_R \tag{7.3}$$
$$c_2 = K_R * \frac{1}{T_n} \tag{7.4}$$
$$c_3 = K_R * T_v \tag{7.5}$$

Eine entsprechende Struktur des Reglers zeigt die Abb. 7.6.

Für die Wahl der Parameter können u.a. die Faustregeln von Ziegler und Nichols benutzt werden. Hierfür wird in einem Experiment zunächst das System mit einem P-Regler betrieben. Für dieses System wird die Verstärkung $K_{R,k}$ bestimmt, für welche das System gerade eben zu schwingen bestimmt. Die entsprechende Schwingungsdauer wird mit T_k bezeichnet. Dann sind die Parameter von Reglern nach Ziegler und Nichols gemäß der Tabelle 7.1 zu wählen.

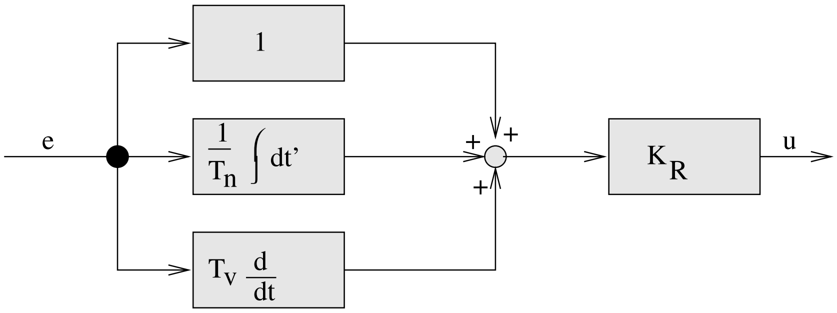


Abbildung 7.6: Struktur eines PID-Reglers

Regler	K_R	T_n	T_v
P	$0,5K_{R,k}$	-	-
PI	$0,45K_{R,k}$	$0,85T_k$	-
PID	$0,6K_{R,k}$	$0,5T_k$	$0,12T_k$

Tabelle 7.1: Faustregeln von Ziegler und Nichols

Diese Zusammenhänge sind in einem gewissen Umfang plausibel. Wenn ein technisches System schnell reagiert, dann entsprechen auch die Parameter T_n und T_v kurzen Zeiten. Wenn die Regelstrecke selbst eine hohe Verstärkung hat, dann wird $K_{R,k}$ klein sein und damit auch die Verstärkung K_R des Reglers.

Für kritische Regelstrecken sind die o.a. Faustregeln nicht geeignet. Für diese müssen optimal auf das jeweilige technische Systmen angepaßte Dimensionierungsverfahren eingesetzt werden. Vielfach basieren letztere auf der Analyse der Sprungantwort der Regelstrecke (siehe z.B. Kiendl, [Kie97]).

Im Falle einer Dimensionierung von drei Parametern kann man die Wahl der Parameter noch anschaulich nachvollziehen. Dies gilt nicht mehr für tatsächlich verwendete Regler, die über zahlreiche zusätzliche Blöcke verfügen (siehe Abb. 7.7).

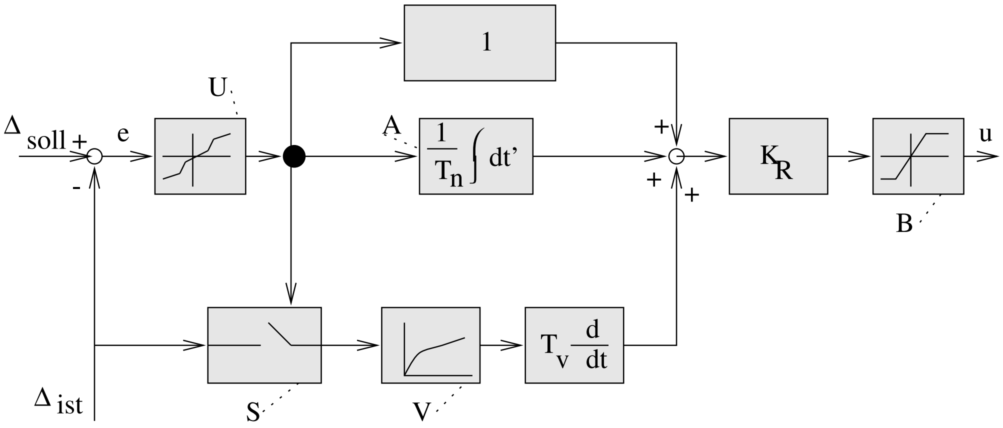


Abbildung 7.7: Praktisch eingesetzte Form eines PID-Reglers

Dieser Regler enthält ein zusätzliches Begrenzungsglied B und eine Unempflindlichkeitszone U für kleine Regelabweichungen. Ein ein Vorfilter V verhindert, daß das Differenzierglied allzu empfindlich auf Schwankungen reagiert. Die Schaltung A soll verhindern, daß das Integrierglied zu große Werte liefert, wenn die Begrenzungsschaltung anspricht.

Derartige praktische Regler können bis zu 20 Parameter beinhalten. Die Dimensionierung dieser Parameter kann nur noch mittels Rechenprogrammen erfolgen. Es ist weitgehend unklar, welche Änderungen am Regler auf einer Änderung an der Regelstrecke notwendig werden.

Weitere Probleme klassischer Regler entstehen, falls es mehrere Einflußgrößen gibt (falls also etwas neben der Temperatur auch die Feuchtigkeit, die Tageszeit, die Sonneneinstrahlung, die durch Beleuchtug eingebrachte Wärme und die Anzahl der Personen in einem Raum berücksichtigt werden sollen.

In solchen Fällen können regelbasierte Regler eine Abhilfe bieten.

7.3 Regelbasierte Regler

Regelbasierte Regler bilden Vorläufer von Fuzzy-Reglern. Solche Regler basieren zunächst einmal auf der Zuordnung von sog. **linguistischen Werten** den üblicherweise reellwertigen Sensorwerten. Beispielsweise kann man Temperaturwerten linguistische Werte wie in Tabelle 7.2 zuordnen.

negativ groß:	NG
negativ klein:	NK
verschwindend:	V
positiv klein:	PK
positiv groß:	PG

Tabelle 7.2: Linguistische Werte

Die Zuordnung zwischen linguistischen und reellen Werten kann man auch der Abb. 7.8 entnehmen. 16. Vorl.

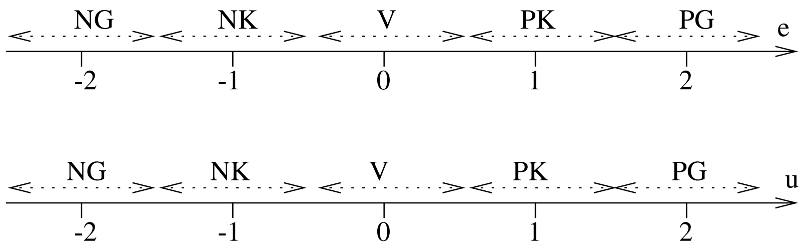


Abbildung 7.8: Zuordnung zwischen linguistischen und reellen Werten

Die Zuordnung kann man auch mit charakteristischen Funktionen beschreiben (siehe Abb. 7.9).

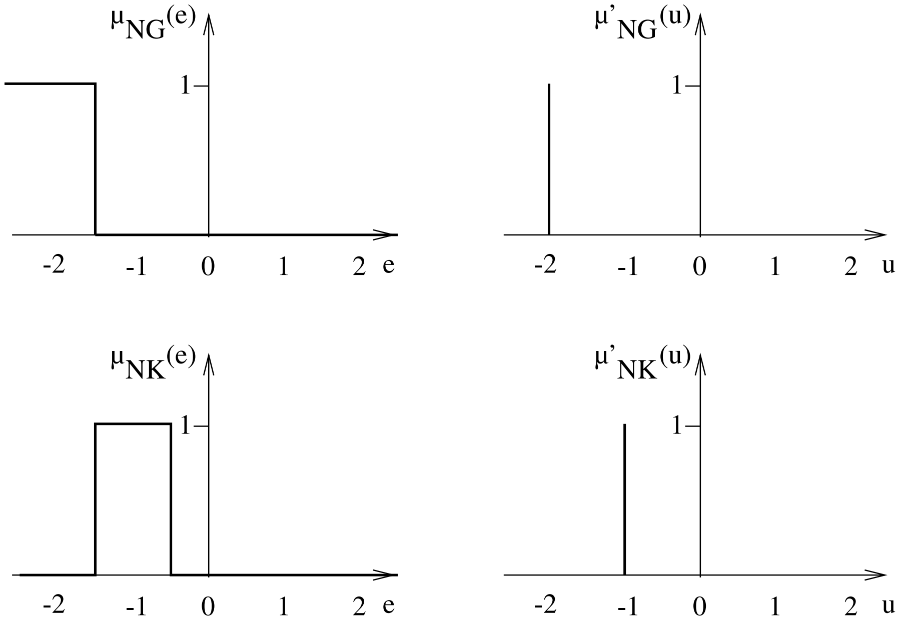


Abbildung 7.9: Charakteristische Funktionen

Mit diesen Werten kann man beispielsweise die Regeln der Tabelle 7.3 aufstellen.

R1:	WENN e=NG DANN u=NG
R2:	WENN e=NK DANN u=NK
R3:	WENN e=V DANN u=V
R4:	WENN e=PK DANN u=PK
R5:	WENN e=PG DANN u=PG

Tabelle 7.3: Regeln

Die zu erfüllende Bedingung heißt **Prämisse**, die angegebene Folge **Konklusion**.

Diese Regelsätze sind bei **einfachen** regelbasierten Reglern so konstruiert, daß maximal eine Prämisse erfüllt ist. Kommen Eingabewerte vor, für welche keine Prämisse erfült ist, so heißt die Regelmenge **unvollständig**.

Wenn Regelsätze es erlauben, daß die Prämissen mehrerer Regeln erfüllt sind, so muß aus den angegebenen Konklusionen eine Stellgröße berechnet werden. Hierfür gibt es mehrere Ansätze (Stichwort: konstruktive bzw. destruktive Inferenz). Allerdings sind diese Ansätze zur Beseitigung der Mehrdeutigkeit nicht sehr ausgefeilt.

Auf dieser Basis kann man das Grundprinzip regelbasierter Regler skizzieren (siehe Abb. 7.10).

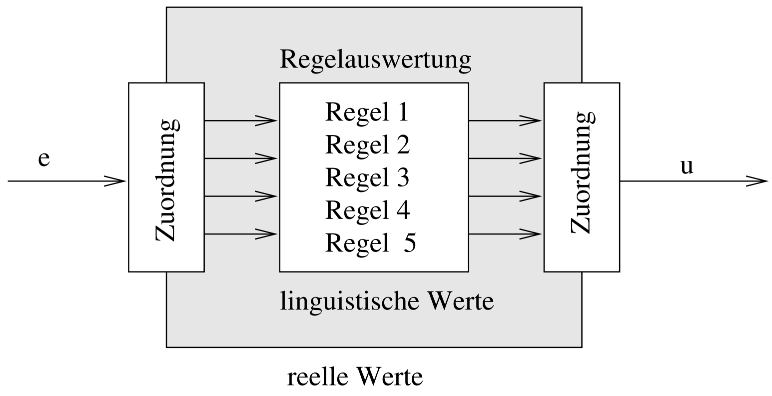


Abbildung 7.10: Grundprinzip zur Nutzung linguistischer Regler

7.4 Fuzzy-Regler

7.4.1 Grundbegriffe

In den vorangegangenen Abschnitten haben wir gesehen, daß die klassische Regelungstechnik recht schnell an ihre Grenzen stößt. Regelbasierte Regler stellen einen ersten Schritt zur Überwindung dieser Grenzen dar. Allerdings sind sie noch mit einer Reihe von Unzulänglichkeiten behaftet:

- Regelbasierte Regler können unmotivierte Unstetigkeiten aufweisen.
- Es gibt nur ad-hoc Verfahren zur Beseitigung von Unvollständigkeit.
- Es gibt nur ad-hoc Verfahren zur Beseitigung von Mehrdeutigkeit.
- Die scharfe Zuordnung zwischen den reellen und den linguistischen Werten paßt nicht zu der mehr oder weniger großen Sicherheit, mit der Regeln angewandt werden sollen.

Diese Mängel werden aufgehoben durch eine **weiche Zuordnung** zwischen reellen und linguistischen Werten. Dies bedeutet, daß einem reellen Wert nicht nur mit 100%-iger Sicherheit ein linguistischer Wert zuordnet wird. Zu diesem Zweck ersetzen wir die charakteristischen Funktionen μ der regelbasierten Regler durch Zugehörigkeitsfunktionen, welche Werte im gesamten Intervall [0..1] annehmen können. Die Abb. 7.11 zeigt eine solche Zugehörigkeitsfunktion.

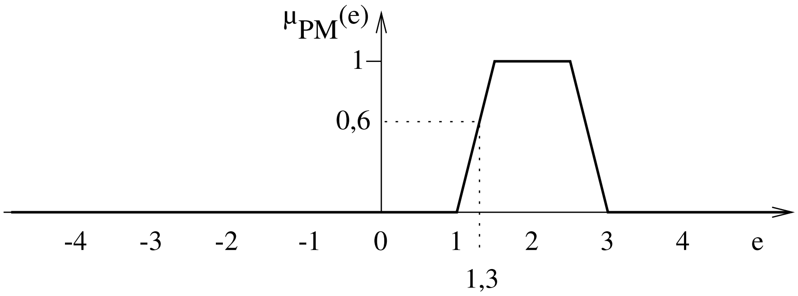


Abbildung 7.11: Trapezförmige Zugehörigkeitsfunktion

Die Zugehörigkeitsfunktion gibt den Grad oder die Wahrscheinlichkeit an, mit der eine Zugehörigkeit zu einer Menge gegeben ist. Von der üblichen Booleschen Logik mit den Wahrheitswerten 0 und 1 kommen wir so zur **Fuzzy-Logik**.

Neben der trapezförmigen Zugehörigkeitsfunktion sind auch noch weitere Funktionen im Gebrauch. Abb. 7.12 zeigt Dreieck-, Polygon- und Glockenkurven-förmige Zugehörigkeitsfunktionen.

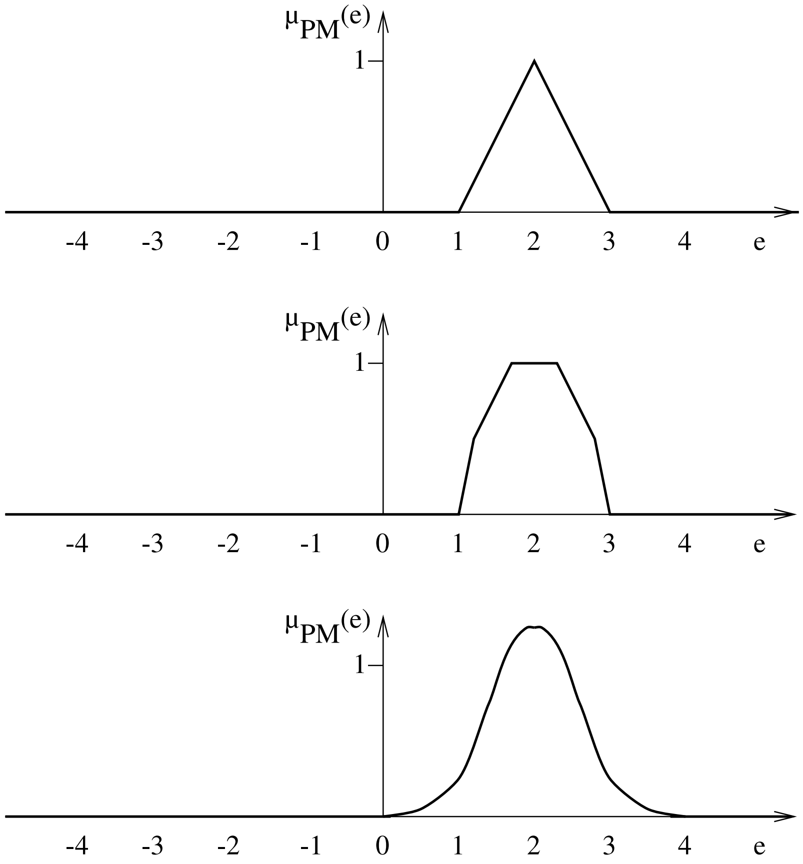


Abbildung 7.12: Weitere Zugehörigkeitsfunktionen

Im Prinzip können als Zugehörigkeitsfunktionen auch andere als die oben gezeigten benutzt werden. Beispielsweise könnte es sinnvoll sein, Funktionen zu benutzen, welche nur in einem Punkt oder einem Intervall 1 sind und sonst 0.

Für das im vorangegangenen Abschnitt betrachtete Beispiel können wir die Zugehörigkeitsfunktionen der Abb. 7.13 verwenden.

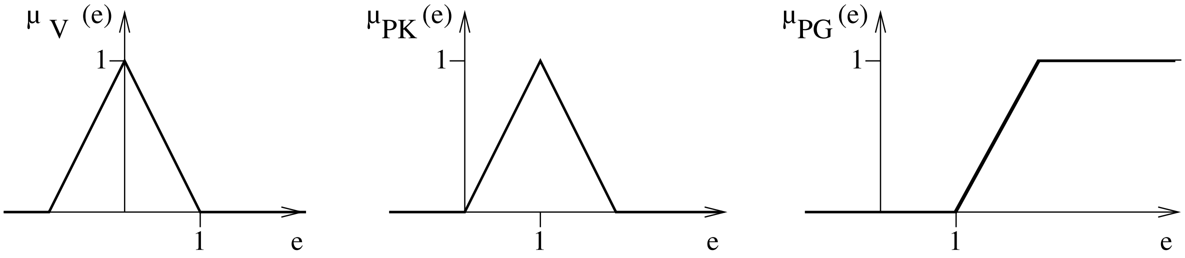


Abbildung 7.13: Zugehörigkeitsfunktionen der Eingangswerte V, PK, PG

Wegen der Stellgrößenbeschränkung wird die Zugehörigkeitsfunktion für die Ausgangswerte meist so definiert, daß eben diese Stellgrößenbeschränkung eingehalten wird. Dabei wird berücksichtigt, daß sich die tatsächliche Stellgröße erst nach der Defuzzifizierung (s.u.) ergibt. Für eine Defuzzifizierung nach der Schwerpunktmethode ist dabei eine Zugehörigkeitsfunktion wie nach Abb. 7.14 sinnvoll.

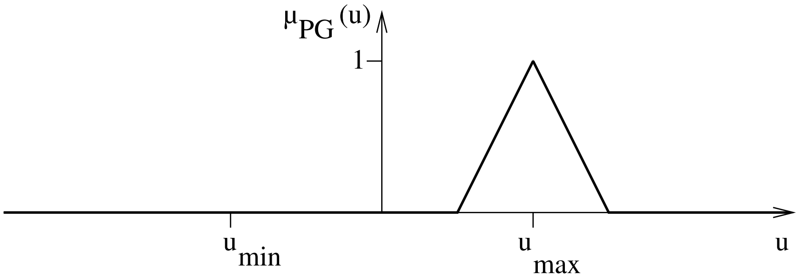


Abbildung 7.14: Zugehörigkeitsfunktion für den Ausgangswert PG

Für die Modellierung von linguistischen Werten werden häufig einander überlappende Zugehörigkeitsfunktionen, deren Summe an jeder Stelle des Argumentsbereichs den Wert 1 ergeben. Ein solches System von Zugehörigkeitsfunktionen heißt **Fuzzy-Informationssystem** (siehe Abb. 7.15).

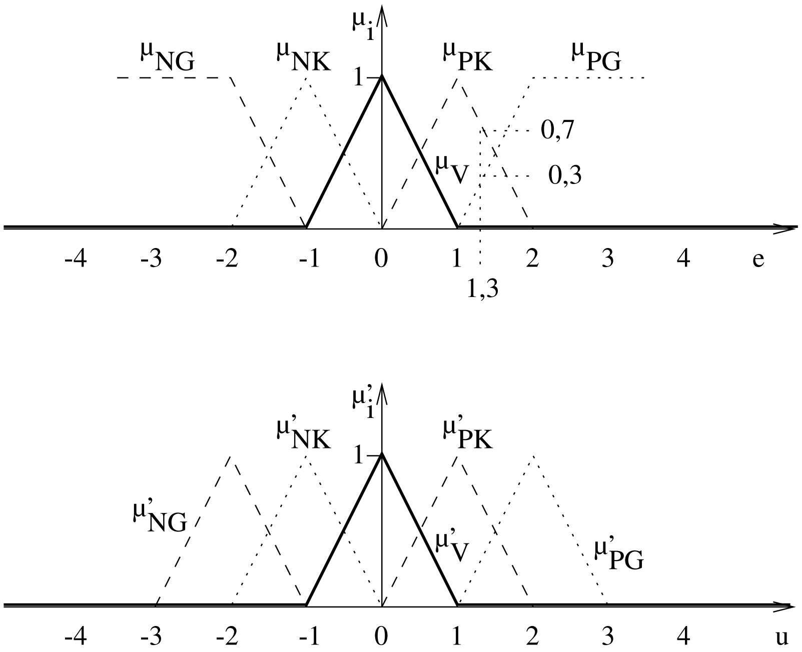


Abbildung 7.15: Fuzzy-Informationssysteme für eingangs- und ausgangsseitige Werte

Bei Verwendung von Zugehörigkeitsfunktionen müssen wir nicht nur die Boolesche Logik zur Fuzzy-Logik erweitern, sondern wir müssen auch die üblichen Mengen zu **Fuzzy-Mengen** erweitern.

Def.: Sei G eine Grundmenge. Dann heißt

$$A = \{(x, \mu_A(x)) | x \in G\}$$

Fuzzy-Menge oder **unscharfe Menge** über der Grundmenge G.

Man beachte, daß A alle Elemente von G enthält. Lediglich die Zugehörigkeitsfunktion μ_A gibt Aufschluß darüber, wie stark x zu A gehört.

Sei eine Fuzzy-Menge A über einer Grundmenge G gegeben.

Def.: Die Menge $Supp_A = \{x \in G | \mu_A(x) > 0\}$ heißt **Träger** oder **Support** von A.

Def.: Die Menge $T_A = \{x \in G | \mu_A(x) = 1\}$ heißt **Kern** oder **Toleranz** von A.

Def.: Für $0 \leq \alpha \leq 1$ heißt $A_\alpha = \{x \in G | \mu_A(x) > \alpha\}$ α-**Schnitt** von A.

Def.: $max(\mu_A)$ heißt Höhe der Fuzzy-Menge A.

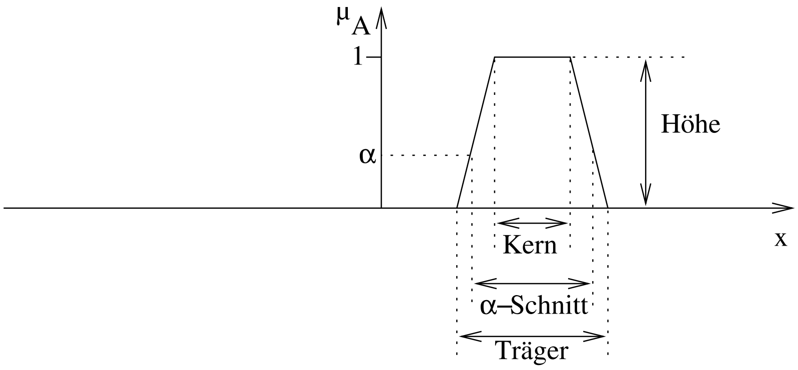


Abbildung 7.16: Träger, Kern, α-Schnitt und Höhe einer Fuzzy-Menge

Die definierten Größen sind in Abb. 7.16 eingetragen.

Nach der Erweiterung des Mengenbegriffs müssen wir auch die logischen Operatoren auf logische Fuzzy-Operatoren erweitern.

Seien $\mu, \mu_1, \mu_2 \in [0..1]$ unscharfe Wahrheitswerte.

Def.: Die **Negation** $\neg\mu$ ist definiert als $1 - \mu$.

Def.: Im Fall des **algebraischen Produkts** ist die UND-Operation zweier unscharfer Wahrheitswerte definiert als

$$\mu_1 \wedge \mu_2 = \mu_1 \cdot \mu_2$$

Def.: Im Fall des **Minimums** ist die UND-Operation zweier unscharfer Wahrheitswerte definiert als

$$\mu_1 \wedge \mu_2 = \min(\mu_1, \mu_2)$$

Def.: Im Fall der **algebraischen Summe** ist die ODER-Operation zweier unscharfer Wahrheitswerte definiert als

$$\mu_1 \vee \mu_2 = \mu_1 + \mu_2 - \mu_1 \cdot \mu_2$$

Def.: Im Fall des **Maximums** ist die ODER-Operation zweier unscharfer Wahrheitswerte definiert als

$$\mu_1 \vee \mu_2 = max(\mu_1, \mu_2)$$

Entsprechend müssen auch die Operatoren auf Mengen erweitert werden. Gegeben seien zwei Fuzzy-Mengen A und B. Dann gilt:

Def.: Der **Durchschnitt zweier Fuzzy-Mengen** A und B ist definiert als $A \cap B = \{(x, \mu_{A\cap B}(x)) | x \in G\}$ mit $\mu_{A\cap B}(x) = \mu_A(x) \wedge \mu_B(x)$.

Def.: Die **Vereinigung zweier Fuzzy-Mengen** A und B ist definiert als $A \cup B = \{(x, \mu_{A\cup B}(x)) | x \in G\}$ mit $\mu_{A\cup B}(x) = \mu_A(x) \vee \mu_B(x)$.

7.4.2 Fuzzy-Regler nach Mamdami

Fuzzy-Regler nach Mamdami gehen aus den regelbasierten Regelern durch den Übergang auf unscharfe Mengen, Wahrheitswerte usw. hervor. Abb. 7.17 zeigt einen einfachen Fuzzy-Regler nach Mamdami.

Im Unterschied zu regelbasierten Reglern können jetzt mehrere Regeln in einem unterschiedlichen Maße aktiv sein. Dementsprechend gibt es auch mehrere Beiträge zu den Steuergrößen, sinnvollerweise gewichtet oder begrenzt auf das Maß, in dem die jeweiligen Regeln aktiv sind. Die Erzeugung einer definierten Steuergröße ist Aufgabe der **Defuzzifizierung**.

Im folgenden betrachten wir die einzelnen Vorgänge in Fuzzy-Reglern nach Mamdami genauer.

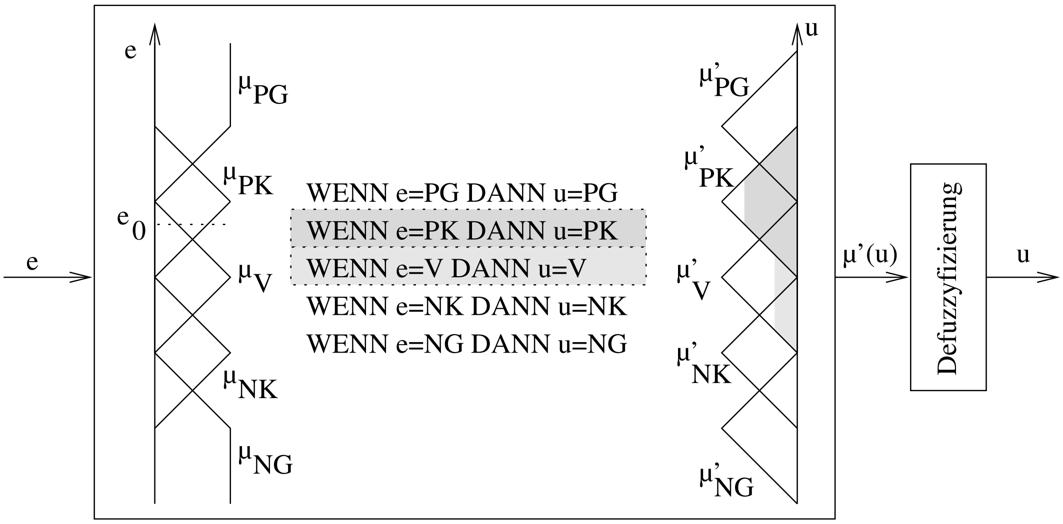


Abbildung 7.17: Fuzzy-Regler nach Mamdami

7.4.2.1 Fuzzifizierung

Die Übersetzung des reellen Eingangswertes in die Zugehörigkeitsgrade der linguistischen Werte wird **Fuzzifizierung** genannt.

Abb. 7.18 zeigt die Fuzzifizierung anhand eines konkreten Eingangswertes e_0.

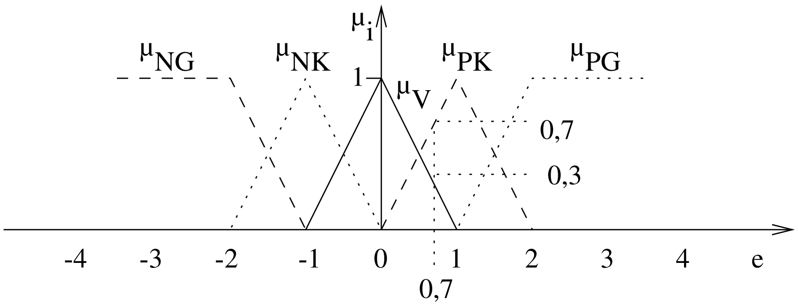


Abbildung 7.18: Fuzzifizierung von Eingangswerten

Es ergibt sich:

$$
\begin{aligned}
\mu_{PK}(e_0) &= 0,7 \\
\mu_V(e_0) &= 0,3 \\
\mu_i(e_0) &= 0 \quad \text{für alle übrigen Funktionen } \mu_i
\end{aligned}
$$

7.4.2.2 Aggregation

Wenn die Prämissen der Regeln nicht aus elementaren Aussagen bestehen, dann ist eine Kombination der Aussagen mittels der Operatoren der Fuzzy-Logik erforderlich.

Beispiel: Wir greifen vor und betrachten einen Regler mit zwei Eingangsgrößen x_1 und x_2. Dann ist für Prämissen wie etwa $(x_1 = PK) \wedge (x_2 = NK)$ die Berechnung einer unscharfen UND-Funktion erforderlich.

7.4.2.3 Aktivierung

Für jeden linguistischen Ausgabewert gelten unterschiedliche Ausgangsgrößen eines Fuzzy-Systems als unterschiedlich gut geeignet. Dementsprechend können wir mit Hilfe einer Zugehörigkeitsfunktion für jeden linguistischen Wert angeben, in welchem Grad ein reeller Wert als Ausgabe geeignet ist. Dies zeigt die Abb. 7.19 (links) für ein Beispiel.

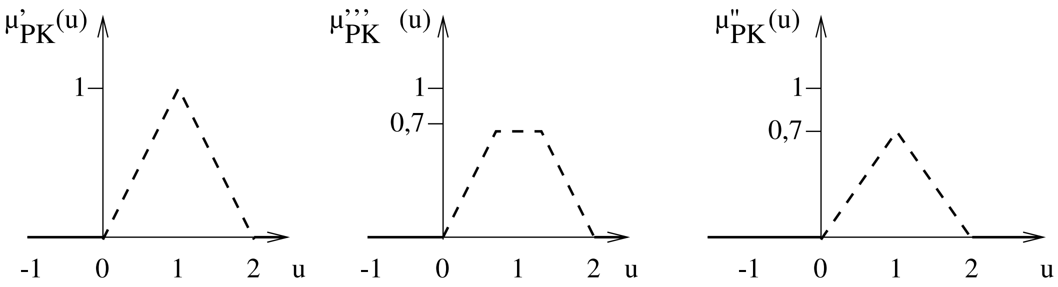


Abbildung 7.19: Zugehörigkeitsfunktion für Ausgabewerte

Eine Regel, welche nicht voll aktiviert ist, sollte auch die Ausgangswerte nicht im vollen Maß beeinflussen. Es gibt zwei Methoden, dies bei der Berechnung der Zugehörigkeitsfunktionen zu berücksichtigen: man kann die Zugehörigkeitsfunktionen begrenzen oder sie entsprechend skalieren (siehe Abb. 7.19 (rechts)).

7.4.2.4 Akkumulation

Als nächstes muß betrachtet werden, wie wir die Ergebnisse der verschiedenen aktivierten Regeln kombinieren. Bei dem sog. Prinzip der **konstruktiven Interferenz** werden die einzelnen Beiträge mittels der Vereinigung der Ausgangsmengen zusammengefaßt.

Wir betrachten dazu eine Menge von r Regeln. Zu jeder Regel $k \in [1..r]$ gehört eine Prämisse $p_k(e)$ und eine Konklusion $c_k(u)$. Hieraus können wir für jede Kombination (e, u) von Eingangs- und Ausgangswerten einen unscharfen Wahrheitswert $\mu(e, u)$ ausrechnen nach:

$$\mu(e,u) = \mu(\bigvee_{k=1}^{r} (p_k(e) \wedge c_k(u))) \tag{7.6}$$

Wir definieren noch für festes $e = e_0$:

$$\mu'(u) = \mu(e,u) = \mu(\bigvee_{k=1}^{r} (p_k(e) \wedge c_k(u))) \tag{7.7}$$

Die Operation der Gleichung 7.6 wird als **Inferenz** bezeichnet. Werden für die Bildung der ODER- und der UND-Operation das Maximum bzw. das Minimum verwendet, so spricht man von der Max-Min-Inferenz.

7.4.2.5 Defuzzifizierung

Die Inferenz liefert meistens nicht einen einzelnen Wert, sondern über die Funktion $\mu'(u)$ für eine Menge von Werten zugehörige unscharfe Wahrheitswerte. Daraus muß noch die Steuergröße u_D berechnet werden.

Eine gängige Methode dazu besteht in der sog. **Schwerpunktmethode**. Für sie berechnet sich die Steuergröße eines kontinuierlichen μ' zu

$$u_D = \frac{\int_{u_{min}}^{u_{max}} \mu'(u) \cdot u \, du}{\int_{u_{min}}^{u_{max}} \mu'(u) \cdot du} \tag{7.8}$$

Darin $[u_{min}..u_{max}]$ der Bereich, in dem die bislang generierten Wahrheitswerte nicht Null sind. Die Schwerpunktmethode kann dann problematisch sein, wenn $\mu'(u_D) = 0$ ist, was nicht ausgeschlossen ist.

7.4.2.6 Gesamtstruktur

Die Gesamtstruktur eines Fuzzy-Reglers nach Mimdami zeigt die Abb. 7.20.

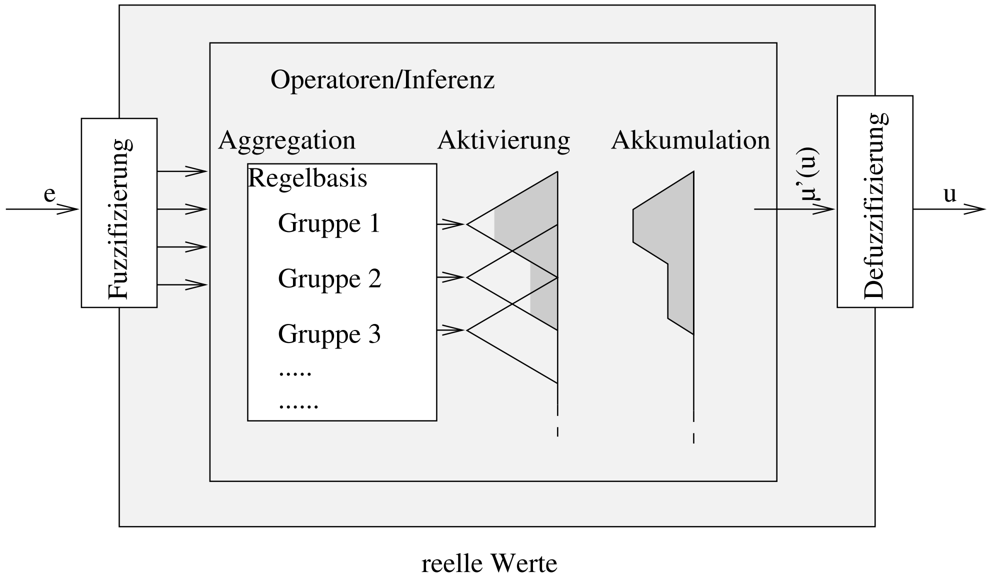


Abbildung 7.20: Fuzzy-Regler nach Midami

Für diese Struktur sind viele Erweiterungen und Varianten möglich. Besonders interessant ist die einfache Art, in der auf mehrere Steuergrößen Rücksicht genommen werden kann.

Ein wesentlicher Vorteil von Fuzzy-Reglern liegt weiterhin in der Anwendbarkeit diskreter Mathematik anstelle der für traditionelle Regler üblichen Analysis-Methoden.

Auf alle diese fortgeschrittenen Themen können wir aber im Rahmen dieser Vorlesung aus Zeitgründen nicht eingehen.

Kapitel 8

Roboter

8.1 Handhabungsroboter

17. Vorl.

8.1.1 Begriffe

Ideen zur Benutzung von Maschinen zur Durchführung mechanischer Tätigkeiten sind sehr als und wurden teilweise bereits von den Griechen entwickelt. Der Name **Roboter** geht zurück auf das Stück 'Rossum's Universal Robot' von Karel Capeks (1921). Das Wort 'robot' ist dabei eine Anlehnung an das tschechische Wort 'robota', was 'schwer arbeiten' bedeutet. In dem Stück sind Roboter menschenähnliche Kreaturen, welche der Menschheit Dienstleistungen erbringen (nach Kreuzer et al. [KMLT94]).

Heutige Roboter sind allerdings recht mechanischer Art. Für sie wird daher auch der Begriff **Handhabungsgerät** verwendet.

Laut ISA-Regelung TR 8373 definiert man **Industrieroboter** als

> universell einsetzbare Handhabungsautomaten mit mindestens drei Achsen, deren Bewegungen ohne mechanischen Eingriff frei programmierbar sind und die mit Endeffektoren, z.B. Greifern und Werkzeugen ausgerüstet werden.

Heutige Roboter gehen auf zwei Entwicklungen zurück, nämlich

1. **die Fernbedienungstechnik für radioaktives Material**

   Diese Technik setzte 1945 mit dem Einsatz von Teleoperatoren ein. Der Antrag erfolgte zunächst von Hand und später servoelektrisch (nach Dillmann [Dil86]).

2. **Numerisch gesteuerte Fertigungstechnik (NC-Maschinen)**

   Diese Entwicklung begann 1949 mit der ersten NC-Maschine, die am MIT hergestellt wurde.

Die weitere Entwicklung läßt sich wie in Tabelle 8.1 charakterisieren [Dil86].

ab 1945	Entwicklung von Teleoperatoren und Kraftsensoren
ab 1950	Entwicklung von NC-Maschinen; einfache fest programmierte *pick-and-placement*-Geräte
ab 1960	Mechanische Informationsspeicher (Nockenwalzen); erste frei programmierbare Handhabungsgeräte
ab 1970	Elektronische Informationsspeicher; rechnergesteuerte Industrieroboter; Bahnsteuerung; textuelle Programmierung
ab 1980	Sensorintegration; Integration in größere Entwicklungs- und Fertigungssysteme
ab 1990	Verbesserungen der Konstruktion: bürstenlose Motore, 32-Bit-Prozessoren; beginnender Einsatz der künstlichen Intelligenz.

Tabelle 8.1: Entwicklung der Roboter

Eingesetzt werden Roboter in vielen Bereichen, so z.B. in der Raumfahrt, der Marinetechnologie, in der Mikromechanik, in radioaktiven Bereichen, in gesundheitsgefährdender Umgebung und neuerdings auch in der Medizintechnik. Im Einsatzbereich ist zwischen Massenanwendungen und Spezialanwendungen (wie z.B. der Raumfahrt) zu unterscheiden. Kreuzer et al. [KMLT94] stellen einen Trend von sehr allgemeinen Robotern hin zu modular aufgebauten Geräten mit austauschbaren Komponenten und hoher Programmierbarkeit fest. Sie geben weiter an, daß sich die ursprünglich prognostizierten Einsatzzahlen nicht erfüllt haben und daß die vielfach diskutierten sozialen Auswirkungen nur eine geringe Bedeutung fr die Akzeptanz der Roboter gehabt haben.

8.1.2 Anforderungen

Anforderungen an Roboter sind u.a. die folgenden [KMLT94]:

- Es muß ein guter Kompromiß zwischen Flexibilität und Spezialisiertheit des Roboters geschlossen werden.
- Wunschvorstellungen: großer Arbeitsraum, hohe Nutzlast, hohe Geschwindigkeit, große Genauigkeit, geringes Eigengewicht, geringer Energieverbrauch, einfache Bedienung, niedrige Anschaffungskosten.
- Möglichst genaue Einhaltung von vorprogrammierbaren Bewegungsabläufen, auch unter veränderlichen, ggf. meßtechnisch erfaßbaren Umweltbedingungen.

Nicht alle diese Eigenschaften können gleichzeitig realisiert werden, z.B. wegen der folgenden Einschränkungen:

- endliche Steifigkeit aller Komponenten,
- vorhandene Massenträgheitsmomente,
- begrenzte Meßfrequenz der Sensoren,
- endliche Meßauflösung der Sensoren,
- Nichtlinearitäten, Reibung, Spiel,
- Grenzen von Rechnerprogrammen und Rechnerhardware

Kreuzer et al. [KMLT94] fassen die Anforderungen an Roboter wie folgt zusammen:

1. Gute mechanische Eigenschaften, vor allem
   - kleine mechanische Zeitkonstanten in den einzelnen Achsen ...,
   - möglichst gleiches Bewegungsverhalten in den einzelnen Achsen ohne Überschwingen ...,
   - hohe maximale Achsen- und kartesische Geschwindigkeiten,
   - gutes Führungs- und Störverhalten,
   - hohe statische Genauigkeit (Wiederholgenauigkeit),
   - groer Positions- und Geschwindigkeitsstellbereich.
2. Geringe Zahl von hochtourig bewegten Bauteilen.
3. Angepaßte, möglichst spielfreie Getriebe.
4. Spezielle .... abgestimmte Antriebsmotoren.
5. Günstige Motorbefestigung.
6. Einfache und leichte Ausführung der Bewegungselemente ...
7. Günstiges Nutzlast-/Eigengewichtsverhältnis.
8. An die Kinematik ... angepaßte Regeleinrichtungen
9. Geringer Aufwand bei der Optimierung der Achsregelkreise.
10. Genaue und störsichere Meßeinrichtungen ....

8.1.3 Kinematische Grundtypen

Um Roboterelemente zu bewegen, werden sowohl Dreh- wie auch Verschiebebewegungen (Translationen) eingesetzt. Meist werden die Bewegungen in zwei Gruppen aufgeteilt, nämlich die groben sog. Makrobewegungen mit drei Freiheitsgraden (geometrischen Parametern) und die Feinbewegungen mit einer Hand mit ebenfalls drei Freiheitsgraden. Sechs Freiheitsgrade reichen aus, um einen Körper in jede beliebige Position innerhalb eines Raumes zu bringen. Die verschiedenen Roboter unterscheiden sich jetzt in der Realisierung dieser insgesamt sechs Freiheitsgrade. Weiter besitzen sie jeweils einen unterschiedlichen Arbeitsraum, d.h. einen unterschiedlichen Raum, der durch die Hand erreichbar ist. Die folgende Liste erhält die unterschiedlichen Grundtypen realisierter Roboter, ohne Betrachtung der jeweiligen Hände.

3 Translationen: , kubischer Arbeitsraum, kartesisches Koordinatensystem, Portal- und Ausleger-Bauform, einfaches Steuerungskonzept, einfache Antriebsart, preiswert.

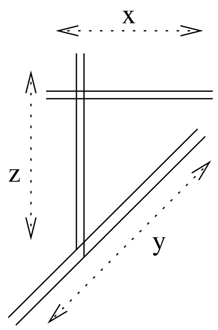


Abbildung 8.1: Bauart ‘Ausleger’

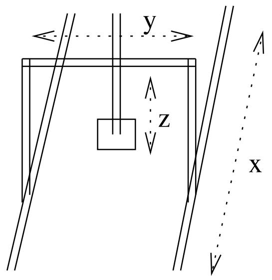


Abbildung 8.2: Bauart ‘Portal’

2 Translationen, 1 Rotation: (Hohl-) Zylinder als Arbeitsraum, Zylinder als Koordinatensystem, Bauform: Dreh-Schubarm, steife Bauart, günstige Motoranordnung möglich.

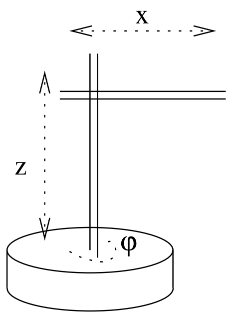


Abbildung 8.3: Bauart ‘Dreh-Schub-Arm’

1 Translation, 2 Rotationen:

- Bauart SCARA (*selective compliance assembly robot arm*) an Hubsäule, steif, genau, größe Höhenverstellbarkeit.

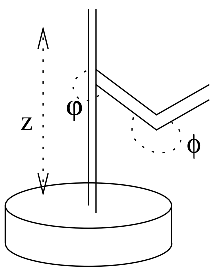


Abbildung 8.4: Bauart 'Scara am Hubarm'

- Bauart SCARA, steif, genau, geringe Höhenverstellbarkeit, keine Handschwenkung

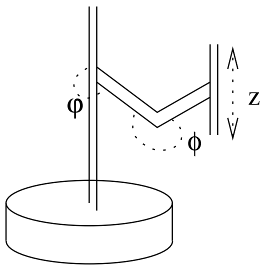


Abbildung 8.5: Bauart 'SCARA'

- Bauart Horizontalparallelogramm, sehr steif
- Bauart Schwenk-Schubarm, steif, sehr günstige Schwerpunktlage, günstige Krafteinleitung

3 Rotationen:

- Bauart Knickarm, großer Verstellbereich, Addition von mehreren Fehlern bis zur Hand, verbreitet

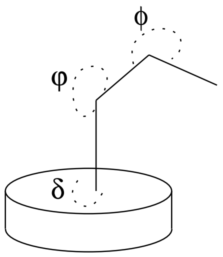


Abbildung 8.6: Bauart 'Knickarm'

- Bauart Vierergelenk, sehr steif

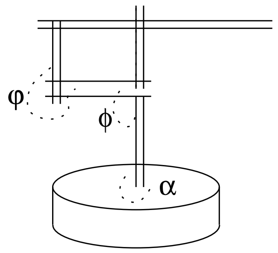


Abbildung 8.7: Bauart 'Vierergelenk'

- Bauart: flexibler Arm, keine Totlagen, nicht sehr genau, nur ein Hersteller bekannt [KMLT94].

Als detaillierter betrachtetes Beispiel wird hier der Roboter HDS 06 der Fa. Nokia betrachtet (siehe Abb. 8.8).

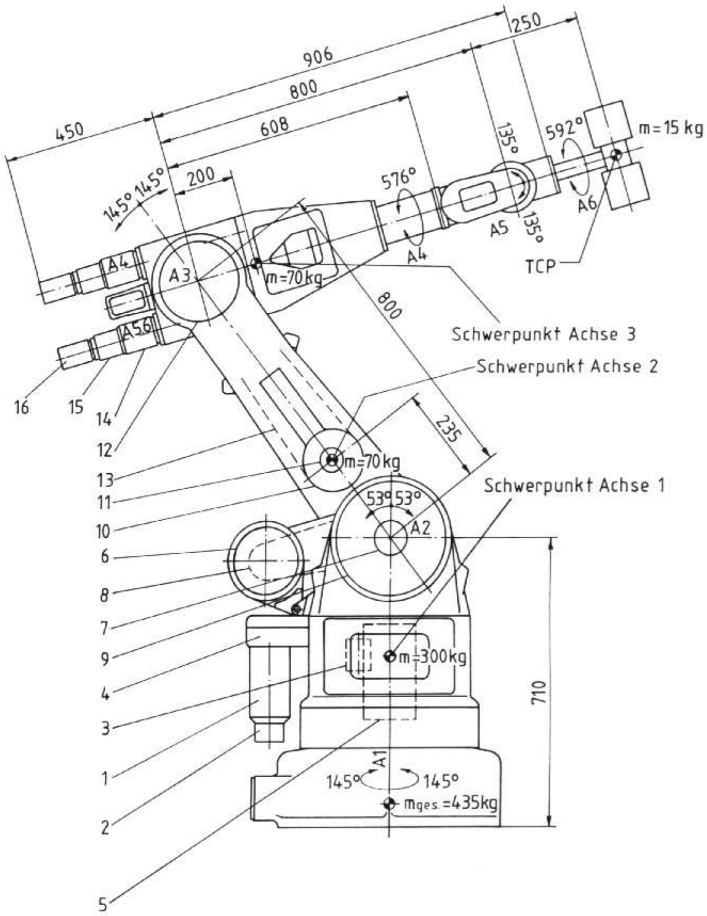


Abbildung 8.8: Gesamtansicht eines Roboters (©Springer-Verlag)

Dieser Roboter verfügt über insgesamt 6 Drehachsen.

8.1.4 Kinematik

8.1.4.1 Transformation von Manipulator- in Weltkoordinaten

Zur Programmierung von Robotern ist es natürlich erforderlich, aufgrund der Stellung der Achsen die jeweilige Lage der Roboterhand (des sog. **Effektors**) auszurechen. Die Winkel (und ggf. die Längen) der einzelnen Roboterabschnitte bilden das sog. **gelenknatürliche Koordinatensystem**. Für unseren Beispielroboter zeigt die Abb. 8.9 das gelenknatürliche Koordinatensystem.

Wir fassen die sechs Parameter zu einem Lagevektor **y** mit den Komponenten

$$\mathbf{y} = [Z, GAO, Y, BE2, AL3, BE3]^T$$

zusammen. Sofern ein Manipulator aus p Teilen besteht, kann die Lage und die Orientierung jedes Teils in bezug auf ein festes Koordinatensystem die Lage des Schwerpunkts C_i und die Orientierung des Teils angegeben werden. Die Lage des Vektors zum Massenschwerpunkt kann durch einen Vektor

$$\mathbf{r}_i = \mathbf{r}_i(\mathbf{y}, t)$$

und eine Drehmatix **S**

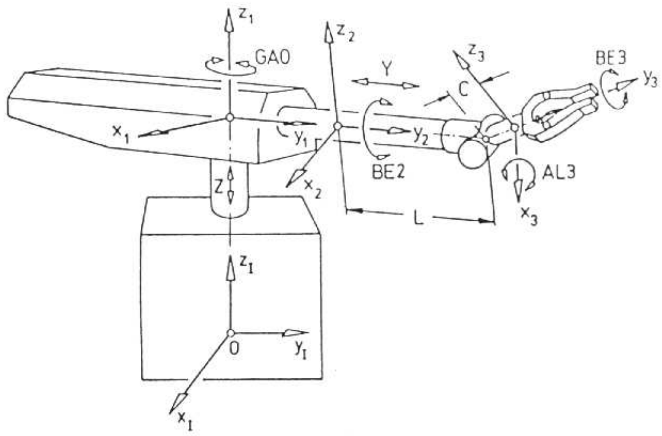


Abbildung 8.9: Koordinatensystem eines Roboters (©Springer-Verlag, Nokia)

$$\mathbf{S}_i = \mathbf{S}_i(\mathbf{y}, t)$$

eindeutig festgelegt werden (siehe auch Abb. 8.10).

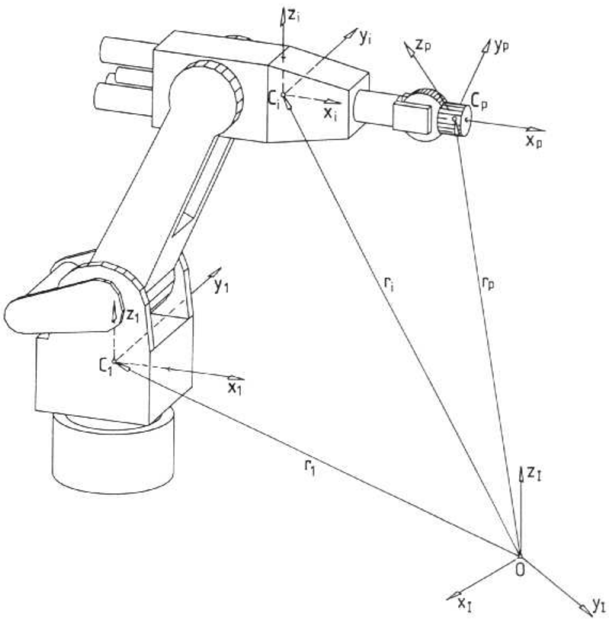


Abbildung 8.10: Koordinatensystem eines Roboters (©Springer-Verlag, Nokia)

Für die Drehung des Koordinatensystems (O_3, x_3, y_3, z_3) in dieser Abbildung gilt beispielsweise

$$(8.1) \qquad {}_2\mathbf{S}_3 \;=\; \begin{bmatrix} cos\phi_3 & 0 & sin\phi_3 \\ 0 & 1 & 0 \\ -sin\phi_3 & 0 & cons\phi_3 \end{bmatrix}$$

Aus Gleichungen wie dieser kann im Prinzip die Lage des Effektors anhand der gelenknatürlichen Koordinaten bestimmt werden. Konkrete Methoden dazu, wie zum Beispiel die **Denavit-Hartenberg-Methode**, sind u.a. bei Keuzer [KMLT94] beschrieben.

8.1.4.2 Transformation von Welt- in Manipulatorkoordinaten

In den Anwendungen muß man natrlich sehr häufig die gewünschte Lage der Roboterhand in Weltkoordinaten-Darstellung in die gelenknatürlichen Koordinaten transformieren. Dieses Problem ist auch als **inverses Problem** bekannt. Im Unterschied zu der umgekehrten Transformation ist die Transformation in gelenknatürliche Koordinaten nicht eindeutig bestimmt. Im Gegenteil: es gibt für eine konkrete Lage der Hand meist viele mögliche Lösungen. In der Praxis schränkt man den Lösungsraum ein und sucht dann mit üblichen mathematischen Verfahren eine der verbleibenden Lösungen.

Für die Rücktransformation der Weltkoordinaten des Effektors müssen man von den Gleichungen, welche Weltkoordinaten und natürliche Koordinaten miteinander in Beziehung setzen, ausgehen. Für Lage und Orientierung des Effektors gilt:

$$\begin{aligned} x_s &= f_1(\theta_i, l_i) \qquad & x_e &= g_1(\theta_i, l_i) \\ y_s &= f_2(\theta_i, l_i) \qquad & y_e &= g_2(\theta_i, l_i) \\ z_s &= f_3(\theta_i, l_i) \qquad & z_e &= g_3(\theta_i, l_i) \end{aligned} \tag{8.2}$$

Diese Funktionen enthalten Summen von Sinus- und Kosinusfunktionen. Sie müssen nach den θ_i bzw. l_i für jeden der Freiheitsgrade aufgelöst werden. Kreuzer [KMLT94] beschreibt hierfür ein Verfahren von Paul.

8.2 Mobile und intelligente autonome Systeme

Eine neuere Tendenz der Robotertechnik ist die Tendenz zu autonomen Systemen. Diese Systeme sind nicht wie klassische Handhabungsroboter für den Einsatz in der Fabrikation oder -allgemeiner- als Handhabungssystem vorgesehen. Vielmehr verrichten sie unabhängig von Fabrikationsanlagen vielfältige Dienste. Die Programmierung derartiger autonomer Systeme stellt auch interessante Herausforderungen an die Informatik. Beispielsweise kommen Techniken der künstlichen Intelligenz zum Einsatz.

Als Beispiel eines solchen Systems betrachten wir die mechanische Nachbildung einer Stabheuschrecke, welche in der Abteilung für Kybernetik der Fachbereichs Biologie der Universität Bielefeld in Kooperation mit einem Maschinenbaulehrstuhl der Technischen Universität München entstanden ist (siehe Abbildungen 8.11 und 8.12).

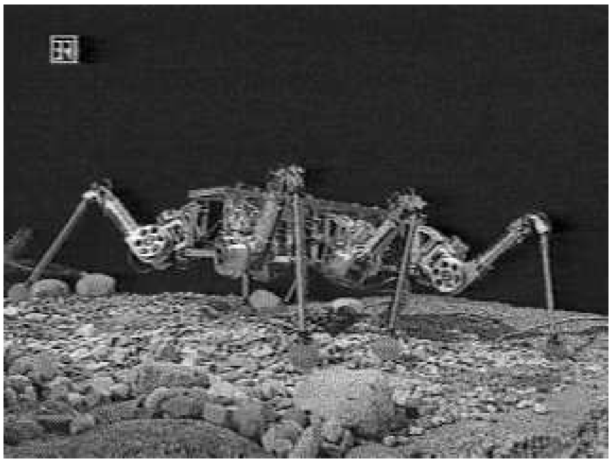


Abbildung 8.11: Nachbildung einer Stabheuschrecke

Mit diesem Modell wurden Untersuchungen zu möglichen Vorgängen im Gehirn der Stabheuschrecke durchgeführt.

In Kombination mit der Konstruktion dieser technischen Stabheuschrecke wurde auch ein technisch einsetzbares Gerät entwickelt, der sog. **Rohrkrabbler** (siehe Abb. 8.13).

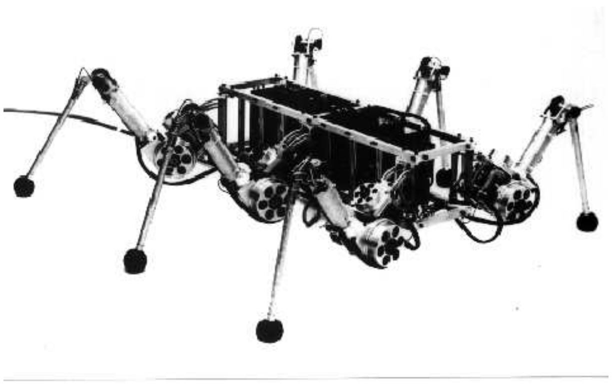


Abbildung 8.12: Nachbildung einer Stabheuschrecke

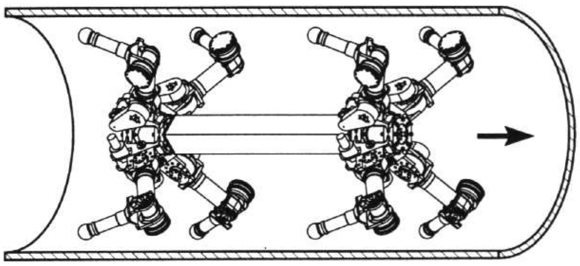


Abbildung 8.13: Rohr-Krabbler (©Springer-Verlag)

Der Rohrkrabbler ist in der Lage, sich selbständig in Rohren zu bewegen und Bilder an die Außenwelt zu übertragen. Es ist daran gedacht, derartige Systeme zur Wartung von Rohrleitungsnetzen einzusetzen.

"Der Rohrkrabbler hat insgesamt acht Beine, von denen jeweils vier vorne und vier hinten sternförmig angeordnet sind. Jedes Bein hat zwei aktive, von Gleichstrommotoren angetriebene Gelenke, welche dem Bein innerhalb der jeweiligen Ebene die volle Beweglichkeit geben. Ein zusätzliches, mit viskoelastischen Materialien gefedertes passives Gelenk ermöglicht kleine Ausgleichsbewegungen senkrecht dazu. ... Jedes aktive Gelenk hat jeweils ein Potentiometer zur Erfassung des Gelenkwinkels und ein Tachometer für die Motordrehzahl. Des weiteren können die Beinkräfte in allen drei Raumrichtungen gemessen werden." [Zitat Roßmann und Pfeiffer [RP95]] 18. Vorl.

Für die Steuerung des Rohrkrabblers gab es folgende Vorgaben[1]:

- Realisierung der Bewegung und Kontrolle der Position,
- Sicheres Beherrschen der Beinkräfte, um ein Durchrutschen von Beinen oder Abgleiten des Krabblers zu verhindern,
- Aufgrund der Steuerungselektronik und der klaren Strukutr weitestmöglich dezentrale Realisierung trotz starker Koppelung tragender Beine,
- Erweiterbarkeit auf schwierigere Rohrgeometrien.

Zur Realisierung dieser Anforderungen wurde eine Struktur aus zwei hierarchisch weiter unterteilten Ebenen entwickelt. Diese Unterteilung basiert u.a. auf den Erfahrungen mit der o.a. Stabheuschrecke. Beide Ebenen können noch in einen zentralen und einen dezentralen (Bein-) Anteil zergliedert werden. Die obere Ebene deckt Koordinierungsaufgaben ab, während die untere die Kontrolle der Lagen und Kräfte durchführt (sog. operative Ebene, siehe Abb. 8.14 und 8.15

Die zentrale Koordinationsebene koordiniert die beiden Beinebenen zueinander und beseitigt globale Störungen. Vor allem wird in dieser Ebene der Wechsel der tragenden Beine bestimmt.

Die dezentrale Koordinationsebene kontrolliert den Schrittablauf eines Beines. Die einzelnen Phasen der Bewegung wie auch lokale Störungen werden hier behandelt.

[1] Darstellung weiterhin nach Roßmann und Pfeiffer [RP95].

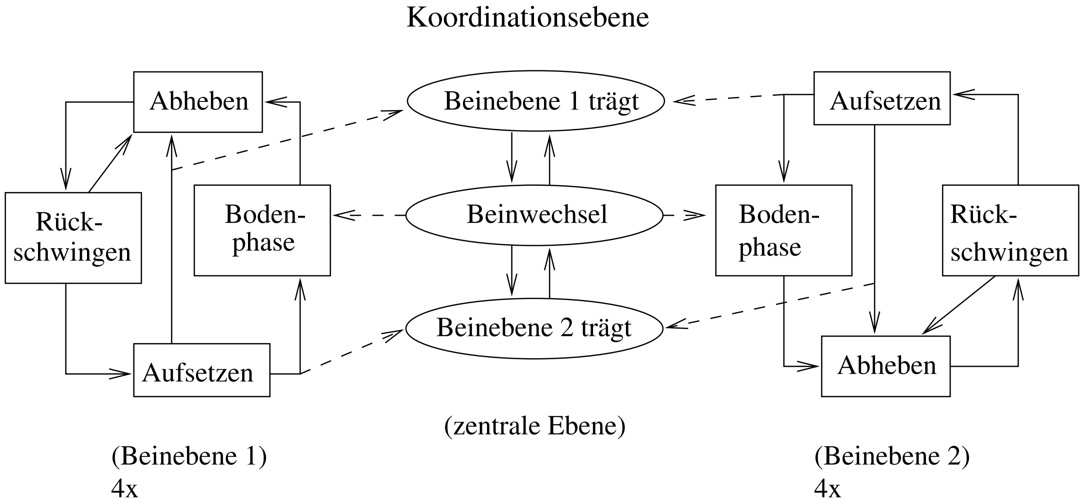


Abbildung 8.14: Koordinationsebene

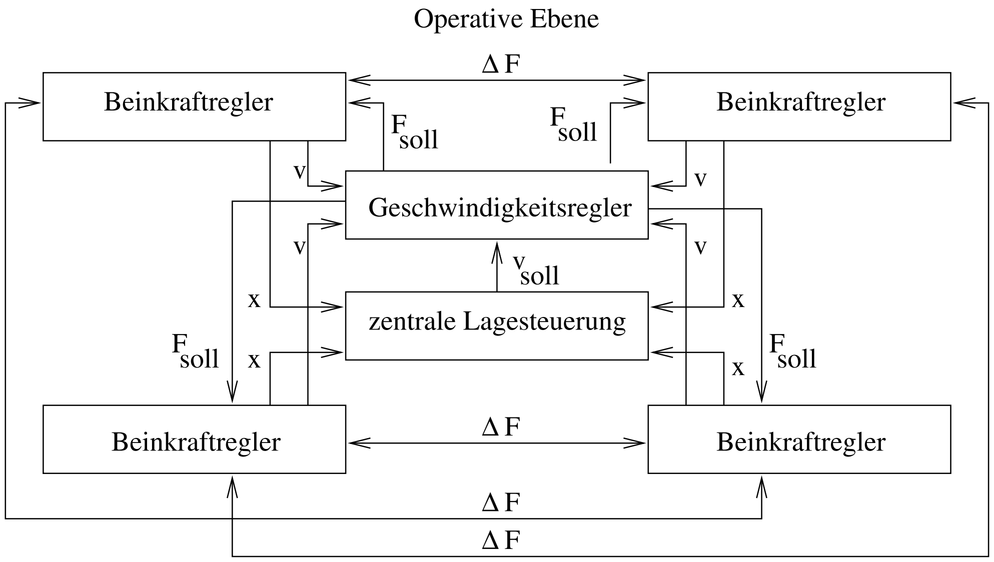


Abbildung 8.15: Operative Ebene

Auf der zentralen operativen Ebene wird die Lage des Zentralkörpers geregelt, indem die Beine als Stellglieder verwendet werden. Eine besondere Schwierigkeit liegt in dem Einhalten der Haftbedingungen.

In der dezentralen operativen Ebene werden die vorgegebenen Kräfte und die Bewegung eines Beines kontrolliert.

Zur Realisierung der Steuerung wird für je 2 Beine ein Mikrocontroller eingesetzt, für die zentrale Koordination kommt noch ein weiterer hinzu.

Die Simulation der mechanischen Eigenschaften des Rohr-Krabblers in Kombination mit der Auslegung der elektrischen Regelkreise ist bei Roßmann et al. beschrieben [RP95].

Das Beispiel des Rohrkrabblers wurde hier besprochen, um exemplarisch zu zeigen, was in der zweiten Hälfte der neunziger Jahre Stand der Technik im Bereich der autonomen Systeme ist. Es sollte hiermit die Überlappung der verschiedenen Fachdisziplinen (Informatik, Elektrotechnik, Maschinenbau, evtl. auch Biologie) deutlich werden. Das Beispiel soll auch die enorme Dynamik in diesem Anwendungsbereich der Informatik belegen. Zu den autonomen Systemen gehört auch das erste Segelboot, das ohne Skipper die Welt umsegeln soll.

Autonome Systeme finden auch in vielen anderen Bereichen Anwendung. So wurden beispielsweise im ESPRIT-Prokjekt Prometheus autonome Fahrzeuglenkungssysteme erforscht. Wir betrachten diese Systeme im folgenden Kapitel im Zusammenhang mit dem **Maschinellen Sehen**.

Kapitel 9

Maschinelles Sehen

9.1 Fahrzeug-Beeinflussung und Fahrzeug-Lenkung

Entwickelte autonome Systeme sind in der Lage, Fahrer bei dem Führen von Autos zu unterstützen und dabei selbständig Gefahrensituationen wie etwa zu kurze Abstände zwischen Fahrzeugen zu erkennen (siehe auch Abb. 9.1). Die Schlüsseltechnologie hierfür liegt in intelligenten Bildverarbeitungsmethoden.

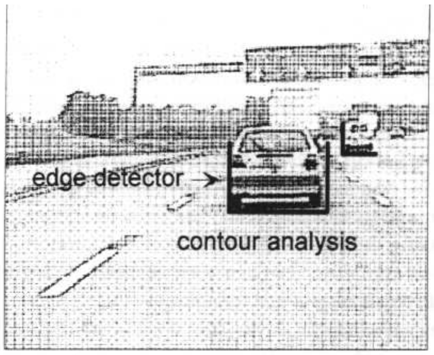


Abbildung 9.1: Intelligente Szenenerkennung (©Springer-Verlag)

Als Beispiel betrachten wir Ergebnisse des ESPRIT-Projekts Prometheus [Dic95]. Mit den Methoden dieses Projekts konnten 1995 die folgenden Fähigkeiten erreicht werden:

- Automatische Fahrzeuglenkung innerhalb einer Fahrbahn mit Anpassung der Geschwindigkeit an die Fahrbahnkrümmung bis zu einer maximalen Geschwindigkeit von 130 km/h,
- Fahren innerhalb einer Fahrzeugschlange mit automatischem Einhalten des Sicherheitsabstandes,
- Fahren im *stop-and-go*-Verkehr,
- Fahrbahnwechsel,
- Anhalten vor einem Hindernis,
- Abbiegen an einer Kreuzung.

Was sind die Grundzüge der Technik eines solchen intelligenten Systems? Abb. 9.2 beschreibt die wesentlichen Komponenten des Systems.

Jeweils zwei Kameras sind im Fahrzeug vorn und hinten montiert und mit einer Kompensation rascher Erschütterungsvorgänge ausgestattet. Ihre Daten gehen an dedizierte Videoverarbeitungsprozessoren, welche jeweils für die Erkennung bestimmter, auf Straßen häufiger Objekte zuständig sind. Ein Prozessor hat besipielsweise anhand der Fahrbahnmarkierungen die Lage und ggf. die Krümmung der Fahrbahn

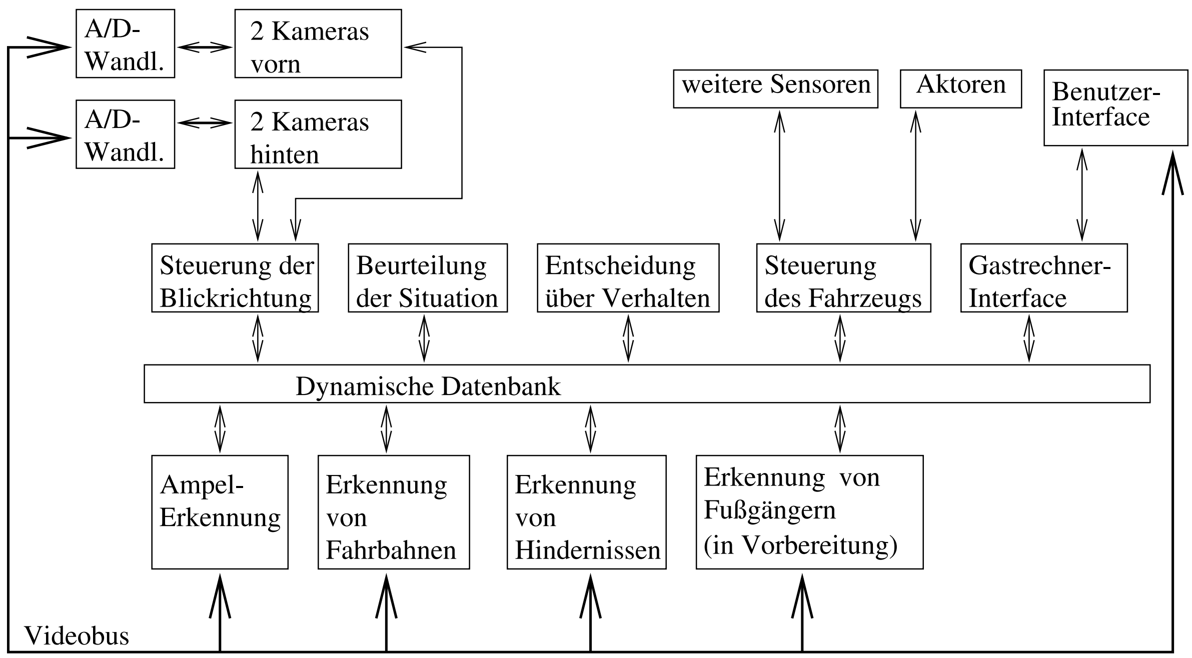


Abbildung 9.2: Komponenten einer automatischen Fahrzeugsteuerung

zu erkennen. Weitere Prozessoren sind für die Erkennung von Hindernissen zuständig. So werden auch vorausfahrende Fahrzeuge erkannt (siehe Abb. 9.1).

Allerdings basiert diese Erkennung auf dem Schatten unterhalb eines Fahrzeugs und ist daher für Landstraßen, neben denen auch Bäume Schatten werfen, weniger geeignet.

In beschränkten Umfang können diese Systeme auch Bremsvorgänge einleiten. Allerdings beschränkt man sich sicherheitshalber vorwiegend auf die Motorbremse (wobei ein automatisches Herunterschalten des Ganges möglich ist). Eine Einwirkung auf die Fußbremse ist in Deutschland nur bis zu einem gewissen Anteil der Bremswirkung der Fußbremse erlaubt.

9.2 Teilschritte maschinellen Sehens

Zur Fahrzeuglenkung werden bereits hierarchische Methoden der Erkennung von Objekten eingesetzt. Dies ist generell der Ansatz des maschinellen Sehens. Abb. 9.3 enthält ein entsprechendes Rahmenmodell [Fri91] [1].

Die Szenenbestrahlung dient in erster Linie zur Erzeugung eine guten Kontrastes, und zur Hervorhebung relevanter Oberflächenstrukturen. Durch räumlich strukturierte Strahlung können geometrische Größen aus der Szene in charakteristische Helligkeitsmuster umgesetzt werden.

Für die Bildaufnahme werden üblicherweise CCD-Kameras verwendet. Aus den einzelnen Abtastwerten $B(i,j)$ kann in Übertragung der Ergebnisse für eindimensionale Abtastung aus den diskreten Amplitudenwerten wieder der kontinuierliche Amplitudenverlauf regeneriert werden:

$$b(x,y) = \sum_{i=0}^{i_0}\sum_{j=0}^{j_0} B(i,,j)\frac{\sin\pi\left(\frac{x}{\Delta x}-i\right)\sin\pi\left(\frac{y}{\Delta y}-j\right)}{\pi\left(\frac{x}{\Delta x}-i\right)\pi\left(\frac{y}{\Delta y}-j\right)}$$

Für den Bereich der Standardalgorithmen der Rasterbildverarbeitung sei hier auf die Vorlesung 'Graphische Systeme' verwiesen, in der Themen wie z.B. die Kantenversteilung, Konturerkennung usw. behandelt werden. Alle diese Techniken gehören zu einem vollständigen Bild autonomer Systeme dazu. Ohne Techniken zur Erkennnung von Objekten könnte die o.a. Fahrzeugsteuerung nicht funktionieren.

[1] Dieser Teil der Vorlesung konnte aufgrund des besonders kurzen Sommersemesters nicht in der eigentlich gewünschten Ausführlichkeit behandelt werden.

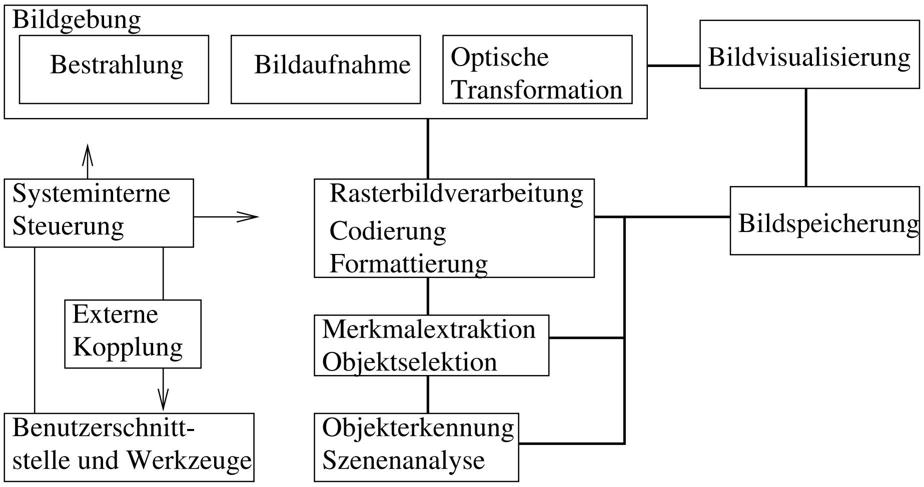


Abbildung 9.3: Rahmenmodell des maschinellen Sehens

In diesem Skript nicht enthalten ist der Stoff der abschließenden Vorlesung über Fuzzy-Logik in der Bildverarbeitung und Qualitätssicherung.

Schlußbemerkung

Das vorliegende Manuskript gibt die wesentlichen Inhalte der im Sommersemester 1998 erstmalig angebotenen Vorlesung 'Prozeßrechner' wieder. Nicht immer gab es während der Vorbereitung der Vorlesung hinreichend Zeit, eine optimale Auswahl und Darstellung des Stoffs zu erreichen. Als Folge davon mußte dieses Skript z.Tl. enger an vorhandene Quellen angelehnt werden, als dies bei einer Wiederholung der Vorlesung der Fall gewesen wäre. Bei einer Wiederholung könnten sowohl anregende didaktische Elemente wie auch die Darstellung des theoretischen Hintergrundes erweitert werden. Vor diesem Hintergrund möchte ich hier dringend zum selbstständigen Studium zumindest der wichtigten der angegebenen Quellen raten. Dies schließt auch das lesenswerte Buch von Krishna und Shin ein [KS97], welches gegen Ende der Vorlesung verfügbar wurde.

Die Vorlesung sollte einen Überblick über eine große Zahl von Themen geben, die mit Realzeitsystemen in Verbindung stehen. Als Vorteil sollte die Verbindung der verschiedenen Themenbereiche untereinander klar werden. Eine solche Übersichtsvorlesung sollte aber durch vertiefende Lehrveranstaltungen ergänzt werden. Im Zusammenhang mit dieser Vorlesung bieten sich viele solcher Veranstaltungen an, so z.B. zu den Themen 'parallele Systeme', 'Softwaretest-Methoden', 'Spezifikationstechniken', 'formale Verifikation', 'Software-Technologie', 'Regelungstechnik', 'Digitale Signalverarbeitung', 'Logische Systeme der Informatik' (im Zusammenhang mit der Fuzzy-Logik) und 'Bildverarbeitung' an. Derartige Ergänzungen werden empfohlen.

Eine frohe, vorlesungsfreie Zeit!

Literaturverzeichnis

[Bae94] H. Baehring. *Mikrorechnersysteme.* Springer, 1994.

[BK95] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards.* Kluwer Acad. Publishers, 1995.

[BLR95] J.-M. Bergé, O. Levia, and J. Rouillard. *High-Level System Modeling.* Kluwer Academic Publishers, 1995.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1990.

[Cal93] J. P. Calvez. *Embedded Real-Time Systems.* John Wiley & Sons, 1993.

[CW96] R. Camposano and W. Wolf. Message from the editors-in-chief. *Design Automation for Embedded Systems*, 1996.

[DH89] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Trans. on Computer Design*, 1989.

[Dic95] E.D. Dickmanns. Performance improvements for autonomous road vehicles. *U. Rembold and R. Dillmann and L.O. Hertzberger and T. Kanade (Hrg.): Intelligent autonomous systems (IAS-4), IOS Press*, pages 2–14, 1995.

[Dil86] R. Dillmann. *Einführung in die Robotik.* Skript des Lehrstuhls Informatik III der Universität Karlsruhe, 1986.

[DIN90] Normenausschuß Informationssysteme DIN. *Programmiersprache PEARL, DIN 66253, Teil 1 bis 3.* Beuth-Verlag, 1990.

[EK91] P. M. Embree and B. Kimble. *C Language Algorithms for Digital Signal Processing.* Prentice Hall, 1991.

[Fae94] G. Faerber. *Prozeßrechentechnik –Grundlagen, Hardware, Echtzeitverhalten.* Springer, 1994.

[Fei97] E.-G. Feindt. *Entwurf und Simulation industrieller Steuerungen für den PC und die SPS.* Oldenbourg-Verlag, 1997.

[FGL87] K.S. Fu, R.C. Gonzalez, and C.S.G. Lee. *Robotics.* McGraw-Hill, 1987.

[Fie94] J. Fiedler. *Unterstützung der objektorientierten Softwareentwicklung unter Berücksichtigung von Echtzeitbetriebssystemen.* VDI Verlag, 1994.

[Fri91] K. Fritsch. *Maschinelles Sehen.* Akademie Verlag, 1991.

[GJ79] M. R. Garey and D. S. Johnson. Computers and intractability. *Bell Labaratories, Murray Hill, New Jersey*, 1979.

[Gup98] R. Gupta. Introduction to embedded systems. *http://www.ics.uci.edu/ rgupta/ics212.html*, 1998.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

[Job96] F. Jobst. *Programmieren in JAVA.* Hanser, 1996.

[KGK95] R. Kruse, J. Gebhardt, and K. Klawonn. *Fuzzy-Systeme.* Teubner, 1995.

[Kie97] H. Kiendl. *Fuzzy Control methodenorientiert.* Oldenbourg, 1997.

[KK89] K.D. Kammeyer and K. Kroschel. *Digitale Signalverarbeitung.* Teubner, 1989.

[KMLT94] E.J. Kreuzer, H.-G. Meißner, J.-B. Lutgenburg, and A. Truckenbrodt. *Industrieroboter - Technik, Berechnung und anwendungsorientierte Auslegung.* Springer-Verlag, 1994.

[Kop97] H. Kopetz97. *Real-Time Systems –Design Principles for Distributed Embedded Applications–.* Kluwer, 1997.

[KR96] J. Kleinöder and U. Rastofer. Java & Echtzeitsysteme? *in: P. Holleczek (Hrg.): PEARL 96, Workshiop über Realzeitsysteme, Springer-Verlag*, 1996.

[KS97] C.M. Krishnan and K. G. Shin. *Real-Time Systems.* McGraw-Hill, Computer Science Series, 1997.

[LNV+97] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. Co-simulation and software compilation methodologies for the system-on-a-chip in multimedia. 1997.

[Man] BI Mannheim. *Informatik-Duden.*

[Mar80] P. Marwedel. The design of a subprocessor with dynamic microprogramming. *in G. Zimmermann (ed.), Informatik Fachberichte Vol. 27*, pages 164–177, 1980.

[Mar97] P. Marwedel. *Skript zur Vorlesung "Rechnerarchitektur".* http://ls12-www.informatik.uni-dortmund.de, 1997.

[Mar98] P. Marwedel. *Skript zur Vorlesung "Rechnergestützter Entwurf und Produktion (Mikroelektronik)".* http://ls12-www.informatik.uni-dortmund.de, 1998.

[Nil96] K. Nilsen. Real-time java. *http://www.newmonics.com/WebRoot/technologies/java*, 1996.

[OS95] A. V. Oppenheim and R.W. Schafer. *Zeitdiskrete Signalverarbeitung.* Oldenbourg, 1995.

[Pag98] J. Pagni. Wie man gebäude wachsam und sparsam macht -integrierte beleuchtung, heizung und klimatisierung-. *in: Zentrum für Technologische Entwicklung Tekes, Im Blickpunkt: Finnische Technologie*, 1998.

[Phi] Philips. Home page for trimedia products. *www.trimedia.philips.com.*

[PMP+98] C. Passarone, J. Martin, R. Passarone, L. Lavagno, and A. Sangiovanni-Vincentelli. Modelling reactive systems in java. *Topics on Design Automation of Embedded Systems (TODAES), to appear*, 1998.

[Pre67] G. Pressler. *Regelungstechnik.* Bibliographishes Institut Mannheim, 1967.

[RP95] Th. Roßmann and F. Pfeiffer. Simulation und regelung eines rohrkrabblers. *in: R. Dillmann, U. Rembold, T. Lüth (Hrg.): Autonome Mobile Systeme 1995, Springer*, 1995.

[Rya95] M. Ryan. Market focus – insight into markets that are making the news in EE Times. *http://techweb.cmp.com/techweb/eet/embedded/embedded.html (Sept. 11)*, 1995.

[Sch97a] U. Schmid. Editorial - global time in distributed systems. *Real-Time Systems (www.kluwer.nl)*, 12:239–242, 1997.

[Sch97b] B. Schürmann. *Rechnerverbindungsstrukturen –Bussysteme und Netzwerke–.* Vieweg, 1997.

[Sie98] R. Sietmann. Augen auf, Finger gezeigt! *C't, 8/98*, pages 100–111, 1998.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Trans. on Computers*, pages 1175–1185, 1990.

[SSH97] K. Schossmaier, U. Schmid, and M. Horauer. Specification and implementation of the universal time coordinated synchronisation unit (utcsu). *Real-Time Systems (www.kluwer.nl)*, 12:295–327, 1997.

[Sun] Sun. Web pages. *http://java.sun.com.*

[Sys] Wind River Systems. Web pages. *http://www.wrs.com.*

[Wro96] M. Wrobel. Betriebssystem-Voraussetzungen für die Integration von Proeßführungs-, SPS-, und B&B- Aufgaben in Einprozessor-Applikationen. *in: P. Holleczek (Hrg.): PEARL 96, Workshiop über Realzeitsysteme, Springer-Verlag*, 1996.

[You82] S.Y. Young. *Real Time Languages –design and development–.* Ellis Horwood, 1982.

[Zei98] Computer Zeitung. Java deckt von smartcard bis enterprise-dv alles ab. *Computer Zeitung, Nr. 15, 19. April 1998*, 1998.

[ZH74] G. Zimmermann and J. Höffner. *Elektrotechnische Grundlagen der Informatik II.* Bibliographishes Institut Mannheim, 1974.

[Zoe87] D. Zoebel. *Programmierung von Echtzeitsystemen.* Oldenbourg, 1987.

Index