Handbuch

Embedded Systems Engineering

V 0.61a

Prof. Dr. Christian Siemers

TU Clausthal
FH Nordhausen

Inhaltsverzeichnis

Abschnitt I: Design von Eingebetteten Systemen

Zusammenfassung und Überblick zu Abschnitt I

Eingebettete Systeme (*embedded systems*) sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind [Sch05]. Diese Umgebungen sind meist maschinelle Systeme, in denen das eingebettete System mit Interaktion durch einen Benutzer arbeitet oder auch vollautomatisch (autonom) agiert. Die eingebetteten Systeme übernehmen komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben für bzw. in diesen technischen Systemen.

Diese Definition steht über dem gesamten Abschnitt I – sowie über dem gesamten Skript. Die Eigenschaft der eingebetteten Systeme, in eine übergeordnete Maschine integriert zu sein, sorgt dafür, dass die Kausalitätskette quasi umgedreht wird: Der Rechner agiert nicht mehr er *re*agiert, und dieser kleine Unterschied ergibt drastische Unterschiede zu stand-alone (Enterprise-)Systemen.

Eingebettete Systeme sind häufig reaktiv aufgebaut, zumindest in zentralen Teilen, und die Reaktivität geht fast immer mit einem *Echtzeitverhalten* einher. Somit befasst sich dieser Abschnitt I mit den Eigenschaften (oder auch Eigenarten) von Echtzeitverhalten und vor allem dem Software Engineering zur Herstellung (und Nachweis) von Echtzeitverhalten.

Kapitel 1 führt in die Begriffswelt von eingebetteten Systemen ein, Kapitel 2 diskutiert allgemeine Definitionen und Probleme um Echtzeit. In Kapitel 3 wird das Problem, eine Echtzeit-fähige Applikation zu erstellen, auf zwei Teilprobleine zurückgeführt:

1. Aufnahme der Kopplungssignale aus dem Umgebungsprozess
2. Echtzeit-fähige Bearbeitung der Signale zwecks Reaktion

Ein in diesem Kapitel eingeführter erster konstruktiver Vorschlag zum Design einer echtzeitfähigen, auf Multithreading-basierten Applikation basiert im Kern darauf, alle echtzeitfähigen Teile auf Interrrupt-Service-Routinen (ISR), die das jeweilige Ereignis vollständig behandeln, abzubilden.

Dieser Ansatz kann nicht beliebig aufrechterhalten werden, da Wechselwirkungen zwischen den eizelnen ISRs problematisch sind und vor allem Threads mit langer Laufzeit (die Eigenschaft «lang» ist hier durch den Umstand definiert, dass die Laufzeit andere Reaktionszeiten übersteigen) ein Echtzeitverhalten unmöglich machen.

Die Abkehr von diesem Design Pattern führt dann zu einem System, das auf Threads basiert ist und eine Software-Event-Steuerung besitzt, die insbesondere die Einfügung eines Schedulings ermöglicht. Dies ist Gegenstand von Kapitel 4.

Den Abschluss in diesem Abschnitt bildet Kapitel 5 mit einem Blick in Richtung Energie- (bzw. Leistungs-) Effizienz.

1 Einführung in informationstechnische Systeme und Informationsverarbeitung

Eingebettete Systeme (*embedded systems*) sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind [Sch05]. Diese Umgebungen sind meist maschinelle Systeme, in denen das eingebettete System mit Interaktion durch einen Benutzer arbeitet oder auch vollautomatisch (autonom) agiert. Die eingebetteten Systeme übernehmen komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben für bzw. in diesen technischen Systemen.

Der Begriff *Informationstechnische Systeme* umfasst wesentlich mehr als nur die eingebetteten Systeme, nämlich grundsätzlich alle Rechnersysteme. Hierunter wird im Allgemeinen ein System verstanden, das in der Lage ist, ein (berechenbares) Problem zu berechnen bzw. zu lösen. Wie dieses Problem dem informationstechnischen System zugänglich gemacht wird, bleibt zunächst offen, doch existieren hierfür zwei Wege: Herstellung eines spezifischen IT-Systems oder Verwendung eines allgemeinen IT-Systems („universelle Maschine“) und Verfassen eines Programms (= außerhalb der Herstellungsprozesses hergestellte und dem System zugänglich gemachte algorithmische Beschreibung) für das spezifische Problem.

Diese Vorlesung beschränkt sich auf die *programmierbaren* informationstechnischen Systeme und darin speziell denjenigen, die als eingebettete Systeme eingesetzt werden. Dies ist keine wirkliche Beschränkung, sondern eher eine Fokussierung, da gerade das Design dieser Systeme eher schwieriger zu bewerten ist als das allgemeiner Rechner und deren Anwendungen.

Die Vorlesung wurde weiterhin so konzipiert, dass die Software im Vordergrund steht. Es geht um (binärwertige) digitale Systeme, die einerseits programmierbar sind, andererseits bei der Ausführung einen Zeitverlauf aufweisen, und deren Entwurf insbesondere in eingebetteten Systemen. Hierzu sollte gleich zu Beginn beachtet werden, dass mit System sowohl das Rechnersystem als auch die relevante Umgebung bezeichnet sein kann. Um hier Verwirrungen zu vermeiden, sei für diesen Kurs mit *System* das digitale System gemeint, also dasjenige, das konzipiert und konstruiert werden soll, während die Umgebung mit *Prozess* oder – präziser – mit *Umgebungsprozess* bezeichnet wird.

Im Vordergrund steht also das System. Die eingebetteten Systeme zeigen dabei eine große Spannweite, denn es ist ein großer Unterschied, eine Kaffeemaschine oder ein Flugzeug zu steuern. Zunächst muss also einmal klassifiziert werden, um die Vielfalt zu beherrschen, und dann werden bestimmte Teile näher behandelt.

Im Anschluss daran soll verdeutlicht werden, worin die eigentlichen Schwierigkeiten bei der Entwurfsmethodik bestehen werden: Der Umgebungsprozess setzt Randbedingungen, und diese Randbedingungen (constraints) müssen neben der

algorithmischen Richtigkeit zusätzlich eingehalten werden. Dies wird anhand der Zeitbedingungen deutlich werden (Abschnitt 1.3).

1.1 Klassifizierung

Definition 1.1:

Ein *eingebettetes System (embedded system)* ist ein binärwertiges digitales System (Computersystem), das in ein umgebendes technisches System eingebettet ist und mit diesem in Wechselwirkung steht.

Das Gegenstück zu *Embedded System* wird *Self-Contained System* genannt. Als Beispiele können Mikrocontroller-basierte Systeme im Auto, die Computertastatur usw. genannt werden.

Hinweis: Die Definition der eingebetteten Systeme ist eine ”weiche“ Definition, aber sie ist trotzdem sehr wichtig! Der Grund bzw. der Unterschied zu den Self-Contained Rechnern besteht darin, dass – wie erwähnt – die Korrektheit bzw. Erfüllung auch in den Randbedingungen (und nicht nur im Algorithmus) einzuhalten ist.

1.1.1 Allgemeine Klassifizierung von Computersystemen

Die heute verfügbaren Computersysteme können in drei unterschiedliche Klassen eingeteilt werden [Sch05]: (rein) transformationelle, interaktive und reaktive Systeme. Die Unterscheidung erfolgt in erster Linie durch die Art und Weise, wie Eingaben in Ausgaben transformiert werden.

Transformationelle Systeme transformieren nur solche Eingaben in Ausgaben, die zum Beginn der Systemverarbeitung vollständig vorliegen [Sch05]. Die Ausgaben sind nicht verfügbar, bevor die Verarbeitung terminiert. Dies bedeutet auch, dass der Benutzer bzw. die Prozessumgebung nicht in der Lage ist, während der Verarbeitung mit dem System zu interagieren und so Einfluss zu nehmen.

Interaktive Systeme erzeugen Ausgaben nicht nur erst dann, wenn sie terminieren, sondern sie interagieren und synchronisieren stetig mit ihrer Umgebung [Sch05]. Wichtig hierbei ist, dass diese Interaktion durch das Rechnersystem bestimmt wird, nicht etwa durch die Prozessumgebung: Wann immer das System neue Eingaben zur Fortführung benötigt, wird die Umgebung, also ggf. auch der Benutzer hierzu aufgefordert. Das System synchronisiert sich auf diese *proaktive* Weise mit der Umgebung.

Bei *reaktiven Systemen* schreibt die Umgebung vor, was zu tun ist [Sch05]. Das Computersystem reagiert nur noch auf die externen Stimuli, die Prozessumgebung synchronisiert den Rechner (und nicht umgekehrt).

Worin liegen die Auswirkungen dieses kleinen Unterschieds, wer wen synchronisiert? Die wesentlichen Aufgaben eines interaktiven Systems sind die Vermeidung von Verklemmungen (deadlocks), die Herstellung von ”Fairness“ und die Erzeugung einer Konsistenz, insbesondere bei verteilten Systemen. Reaktive Systeme hingegen verlangen vom Computer, dass dieser reagiert, und zwar meistens rechtzeitig. Rechtzeitigkeit und Sicherheit sind die größten Belange dieser Systeme.

Zudem muss von interaktiven Systemen kein deterministisches Verhalten verlangt werden: Diese können intern die Entscheidung darüber treffen, wer wann bedient wird. Selbst die Reaktion auf eine Sequenz von Anfragen muss nicht immer gleich sein. Bei reaktiven Systemen ist hingegen der Verhaltensdeterminismus integraler Bestandteil. Daher hier die Definition von Determinismus bzw. eines deterministischen Systems:

Definition 1.2:

Ein System weist *determiniertes* oder *deterministisches Verhalten* (*Deterministic Behaviour*) auf, wenn zu jedem Satz von inneren Zuständen und jedem Satz von Eingangsgrößen genau ein Satz von Ausgangsgrößen gehört.

Als Gegenbegriffe können stochastisch oder nicht-deterministisch genannt werden. Diese Definition bezieht sich ausschließlich auf die logische (algorithmische) Arbeitsweise, und das klassische Beispiel sind die endlichen Automaten (DFA, Deterministic Finite Automaton). Nicht-deterministische Maschinen werden auf dieser Ebene in der Praxis nicht gebaut, beim NFA (Non-Deterministic Finite Automaton) handelt es sich um eine theoretische Maschine aus dem Gebiet der Theoretischen Informatik.

1.1.2 Klassifizierung eingebetteter Systeme

Eingebettete Systeme, die mit einer Umgebung in Wechselwirkung stehen, müssen in der Regel immer auf den umgebenden Prozess reagieren. Dennoch ist ihre Implementierung sowohl als interaktives als auch als reaktives System (und natürlich als Mischform) denkbar.

Ein reaktives Systemdesign ist prinzipiell in Richtung Echtzeitverhalten angelegt, gleichwohl aber natürlich nicht automatisch echtzeitfähig. Wird die Einhaltung von Zeitschranken zu einer Hauptsache, d.h. wird die Verletzung bestimmter Zeitschranken sehr kritisch im Sinn einer Gefährdung für Mensch und Maschine, dann spricht man von Echtzeitsystemen.

Für das interaktive Systemdesign muss – zusätzlich zur Reaktionsberechnung in Echtzeit (→ 3.2) – auch gewährleistet sein, dass die Interaktion, die ja vom Rechner ausgeht, ebenfalls rechtzeitig genug erfolgt (→ 3.2). Ist das gewährleistet, spricht nichts gegen ein interaktives Systemdesign als Grundlage des echtzeitfähigen eingebetteten Systems (→ Bild 1.1).

Weitere wichtige Eigenschaften im Sinn der Einbettung sind: Nebenläufigkeit (zumindest oftmals), hohe Zuverlässigkeit und Einhaltung von Zeitschranken. Noch

eine Anmerkung zum Determinismus: Während man davon ausgehen kann, dass alle technisch eingesetzten, eingebetteten Systeme deterministisch sind, muss dies für die Spezifikation nicht gelten: Hier sind nicht-deterministische Beschreibungen erlaubt, z.B., um Teile noch offen zu lassen.

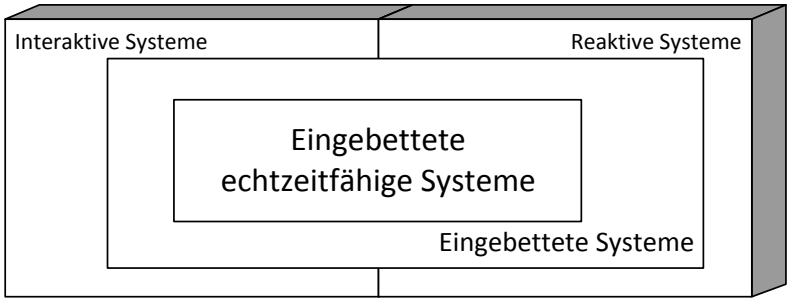


Bild 1.1 Einordnung eingebetteter Systeme

Eingebettete Systeme lassen sich weiterhin nach einer Reihe von unterschiedlichen Kriterien klassifizieren. Hierzu zählen:

- *Kontinuierlich* versus *diskret*: Diese Ausprägung der Stetigkeit bezieht sich sowohl auf Datenwerte als auch auf die Zeit (→ 1.2). Enthält ein System beide Verhaltensweisen, wird es als „hybrides System" bezeichnet.
- *Monolithisch* versus *verteilt*: Während anfänglich alle Applikationen für eingebettete Systeme als monolithische Systeme aufgebaut wurden, verlagert sich dies zunehmend in Richtung verteilte Systeme. Hier sind besondere Anforderungen zu erfüllen, wenn es um Echtzeitfähigkeit geht.
- *Sicherheitskritisch* versus *nicht-sicherheitskritisch*: Sicherheitskritische Systeme sind solche, deren Versagen zu einer Gefährdung von Menschen und/oder Einrichtungen führen kann. Viele Konsumprodukte sind sicherheits-unkritisch, während Medizintechnik, Flugzeugbau sowie Automobile zunehmend auf sicherheitskritischen eingebetteten Systemen beruhen.

1.1.3 Definitionen

In diesem Abschnitt werden einige Definitionen gegeben, die u.a. [Sch05] entnommen sind. Diese Definitionen beziehen sich im ersten Teil auf die informationstechnische Seite, weniger auf die physikalisch-technische.

Definition 1.3:

Unter einem *System* versteht man ein mathematisches Modell **S**, das einem Eingangssignal der Größe x ein Ausgangssignal y der Größe $y = \mathbf{S}(x)$ zuordnet.

Wenn das Ausgangssignal hierbei nur vom aktuellen Wert des Eingangssignals abhängt, spricht man von einem *gedächtnislosen* System (Beispiel: Schaltnetze in der

digitalen Elektronik). Hängt dagegen dieser von vorhergehenden Eingangssignalen ab, spricht man von einem *dynamischen* System (Beispiel: Schaltwerke).

Definition 1.4:

Ein *reaktives System* (*reactive system*) kann aus Software und/oder Hardware bestehen und setzt Eingabeereignisse, deren zeitliches Verhalten meist nicht vorhergesagt werden kann, in Ausgabeereignisse um. Die Umsetzung erfolgt oftmals, aber nicht notwendigerweise unter Einhaltung von Zeitvorgaben.

Definition 1.5:

Ein *hybrides System* (*hybrid system*) ist ein System, das sowohl kontinuierliche (analoge) als auch diskrete Datenanteile (wertkontinuierlich) verarbeiten und/oder sowohl über kontinuierliche Zeiträume (zeitkontinuierlich) als auch zu diskreten Zeitpunkten mit ihrer Umgebung interagieren kann.

Definition 1.6:

Ein *verteiltes System* (*distributed system*) besteht aus Komponenten, die räumlich oder logisch verteilt sind und mittels einer Kopplung bzw. Vernetzung zum Erreichen der Funktionalität des Gesamtsystems beitragen. Die Kopplung bzw. Vernetzung spielt bei echtzeitfähigen Systemen eine besondere Herausforderung dar.

Definition 1.7:

Ein *Steuergerät* (*electronic control unit*, ECU) ist die physikalische Umsetzung eines eingebetteten Systems. Es stellt damit die Kontrolleinheit eines mechatronischen Systems dar. In mechatronischen Systemen bilden Steuergerät und Sensorik/Aktorik oftmals eine Einheit.

Defnition 1.8:

Wird Elektronik zur Steuerung und Regelung mechanischer Vorgänge räumlich eng mit den mechanischen Systembestandteilen verbunden, so spricht man von einem *mechatronischen System*. Der Forschungszweig, der sich mit den Grundlagen und der Entwicklung mechatronische Systeme befasst, heißt *Mechatronik* (*mechatronics*).

Mechatronik ist ein Kunstwort, gebildet aus *Mecha*nik und Elek*tronik*. In der Praxis gehört allerdings eine erhebliche Informatik-Komponente hinzu, da nahezu alle mechatronischen Systeme auf Mikrocontrollern/Software basieren.

1.2 Aufbau und Komponenten eingebetteter Systeme

Während der logische Aufbau eingebetteter Systeme oftmals sehr ähnlich ist – siehe unten – hängt die tatsächliche Realisierung insbesondere der Hardware stark von den Gegebenheiten am Einsatzort ab. Hier können viele Störfaktoren herr-

schen, zudem muss das eingebettete System Sorge dafür tragen, nicht selbst zum Störfaktor zu werden.

Einige *Störfaktoren* sind: Wärme/Kälte, Staub, Feuchtigkeit, Spritzwasser, mechanische Belastung (Schwingungen, Stöße), Fremdkörper, elektromagnetische Störungen und Elementarteilchen (z.B. Höhenstrahlung). Allgemeine und Herstellerspezifische Vorschriften enthalten teilweise genaue Angaben zur Vermeidung des passiven und aktiven Einflusses, insbesondere im EMV-Umfeld (Elektromagnetische Verträglichkeit). Dieses Gebiet ist nicht Bestandteil dieser Vorlesung, aber es soll an dieser Stelle darauf hingewiesen werden.

Der *logische Aufbau* der eingebetteten Systeme ist jedoch recht einheitlich, in der Regel können 5 strukturelle Bestandteile identifiziert werden [Sch05]:

- Die Kontrolleinheit bzw. das Steuergerät (→ Definition 1.7), d.h. das eingebettete Hardware/Software System,
- die Regelstrecke mit Aktoren (bzw. Aktuatoren) (actuator) und Sensoren (sensor), d.h. das gesteuerte/geregelte physikalische System,

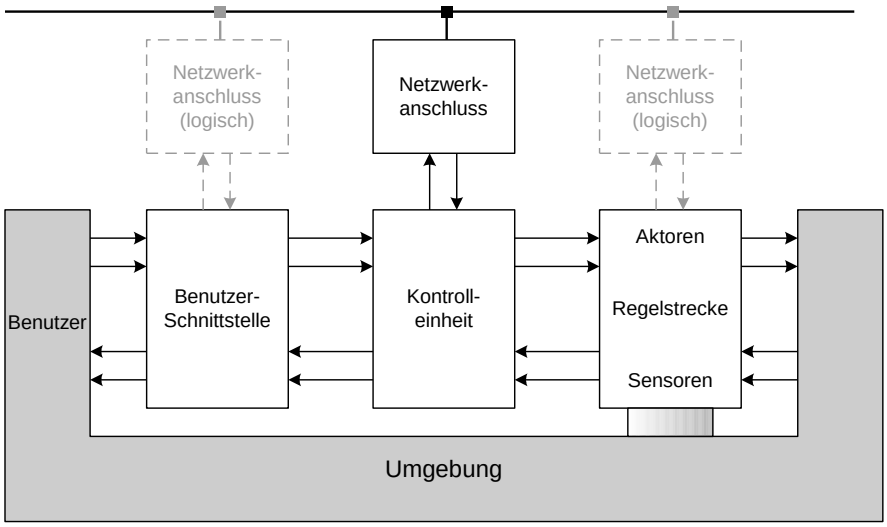


Bild 1.2 Erweiterte Referenzarchitektur eines eingebetteten Systems [Sch05]

- die Benutzerschnittstelle,
- die Umgebung sowie
- den Benutzer.

Mit stark zunehmender Tendenz werden diese Systeme noch vernetzt, so dass sich neben der lokalen Ebene noch eine globale Vernetzungsebene mit physikalischem

Zugang zur Kontrolleinheit und logischem Zugang zu allen Komponenten des Systems ergibt.

Bild 1.2 stellt diese Referenzarchitektur eines eingebetteten Systems als Datenflussarchitektur dar, in der die Pfeile die gerichteten Kommunikationskanäle zeigen. Solche Kommunikationskanäle können (zeit- und wert-)kontinuierliche Signale oder Ströme diskreter Nachrichten übermitteln. Regelstrecke und Umgebung sind hierbei auf meist komplexe Weise miteinander gekoppelt, die schwer formalisierbar sein kann.

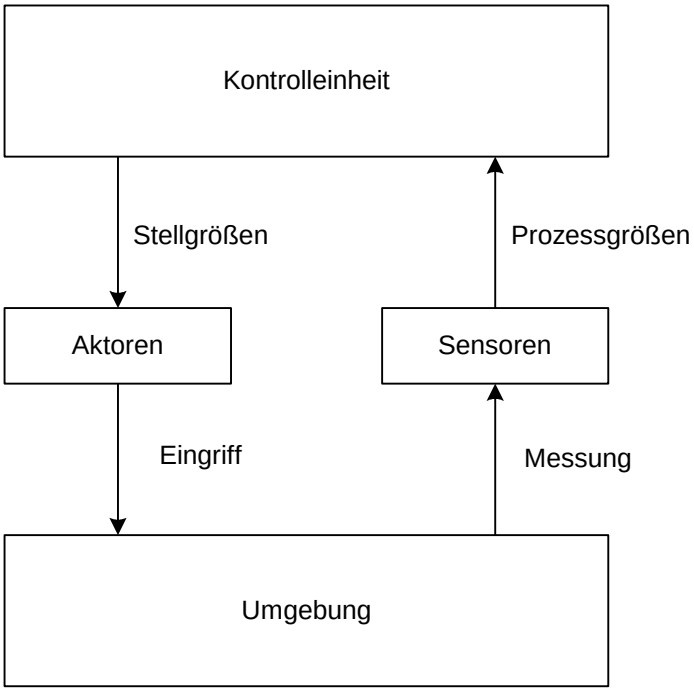


Bild 1.3 Wirkungskette System/Umgebung

Bild 1.3 zeigt die geschlossene Wirkungskette, die ein eingebettetes System einschließlich der Umgebung bildet. Der zu regelnde oder steuernde Prozess ist über Sensoren und Aktoren an das Steuergerät gekoppelt und kommuniziert mit diesem darüber. Sensoren und Aktoren fasst man unter dem (aus dem Von-Neumann-Modell wohlbekannten) Begriff Peripherie (peripheral devices) oder I/O-System (input/output) zusammen.

Zu den einzelnen Einheiten seien einige Anmerkungen hier eingeführt:

Kontrolleinheit

Die Kontrolleinheit bildet den Kern des eingebetteten Systems, wobei sie selbst wieder aus verschiedenen Einheiten zusammengesetzt sein kann. Sie muss das Interface zum Benutzer (falls vorhanden) und zur Umgebung bilden, d.h., sie empfängt Nachrichten bzw. Signale von diesen und muss sie in eine Reaktion umsetzen.

Wie bereits in Abschnitt 1.1.2 dargestellt wurde ist diese Kontrolleinheit fast ausschließlich als reaktives System ausgeführt. Die Implementierung liegt in modernen Systemen ebenso fast ausnahmslos in Form programmierbarer Systeme, also als Kombination Hardware und Software vor. Hierbei allerdings gibt es eine Vielzahl von Möglichkeiten: ASIC (Application-Specific Integrated Circuit), PLD/ FPGA (Programmable Logic Devices/Field-Programmable Gate Arrays) , General-Purpose Mikrocontroller, DSP (Digital Signal Processor), ASIP (Application-Specific Instruction Set Processor), um nur die wichtigsten Implementierungsklassen zu nennen. Man spricht hierbei von einem Design Space bzw. von Design Space Exploration (→ 10.1.3).

Peripherie: Analog/Digital-Wandler

Ein Analog/Digital-Wandler (Analog/Digital-Converter, ADC), kurz A/D-Wandler, erzeugt aus einem (wert- und zeit-)analogen Signal digitale Signale. Die Umsetzung ist ein vergleichsweise komplexer Prozess, der in Bild 1.4 dargestellt ist. Hierbei handelt es sich nicht um eine Codierung, und der Prozess ist nicht exakt reversibel.

Der technisch eingeschlagene Weg besteht aus der Abtastung zuerst (Bauteil: Sample&Hold- bzw. Track&Hold-Schaltung), gefolgt von einer Quantisierung und der Codierung. Die Abtastung ergibt die Zeitdiskretisierung, die Quantisierung die Wertediskretisierung. Man beachte, dass mit technischen Mitteln sowohl die Abtastfrequenz als auch die Auflösung zwar ”beliebig“ verbessert werden kann, aber niemals kontinuierliche Werte erreicht werden. In eingebetteten Systemen werden diese Werte den Erfordernissen der Applikation angepasst.

Für die Umsetzung von analogen Werten in digiatle Werte sind verschiedene Verfahren bekannt: Flash, Half-Flash, Semi-Flash, Sukzessive Approximation, Sigma-Delta-Wandler usw.

Peripherie: Digital/Analog-Wandler

Der Digital/Analog-Wandler, kurz D/A-Wandler (Digital/Analog-Converter, DAC) erzeugt aus digitalen Signalen ein analoges Signal (meist eine Spannung). Dies stellt die Umkehrung der A/D-Wandlung dar. Die Umsetzung erfolgt exakt, abgesehen von Schaltungsfehlern, d.h. ohne prinzipiellen Fehler wie bei der A/D-Wandlung.

Gängige Verfahren sind: Pulsweiten-Modulation (pulse width modulation, PWM) und R-2R-Netzwerke.

Peripherie: Sensoren

Zunächst sei die Definition eines Sensors gegeben [Sch05]:

Definition 1.9:

Ein *Sensor* ist eine Einrichtung zum Feststellen von physikalischen oder chemischen Eingangsgrößen, die optional eine Messwertzuordnung (Skalierung) der Größen treffen kann, sowie ggf. ein digitales bzw. digitalisierbares Ausgangssignal liefern kann.

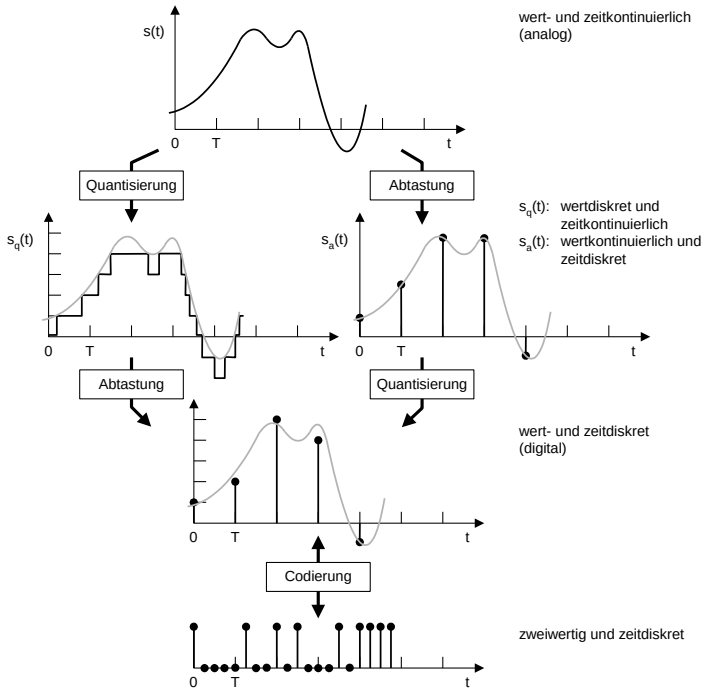


Bild 1.4 Vorgänge bei der AD-Wandlung

Sensoren stellen also das primäre Element in einer Messkette dar und setzen variable, im Allgemeinen nichtelektrische Eingangsgrößen in ein geeignetes, insbesondere elektrisches Messsignal um. Hierbei können ferner *rezeptive Sensoren*, die nur passiv Signale umsetzen (Beispiel: Mikrofon), sowie *signalbearbeitende Sensoren*, die die Umwelt stimulieren und die Antwort aufnehmen (Beispiel: Ultraschall-Sensoren zur Entfernungsmessung), unterschieden werden.

Als *Smart Sensors* bezeichnete Sensoren beinhalten bereits eine Vorverarbeitung der Daten. Hierdurch sind Netzwerke von Sensoren möglich, die auch ganz neue

Strategien wie gegenseitige Überwachungen bzw. Plausibilitätskontrollen ermöglichen.

Peripherie: Aktuatoren

Aktuatoren bzw. Aktoren verbinden den informationsverarbeitenden Teil eines eingebetteten Systems und den Prozess. Sie wandeln Energie z.B. in mechanische Arbeit um.

Die Ansteuerung der Aktuatoren kann analog (Beispiel: Elektromotor) oder auch digital (Beispiel: Schrittmotor) erfolgen.

1.3 Die Rolle der Zeit und weitere Randbedingungen

1.3.1 Verschiedene Ausprägungen der Zeit

In den vorangegangenen Abschnitten wurde bereits verdeutlicht: Die Zeit spielt bei den binärwertigen digitalen *und* den analogen Systemen (Umgebungsprozess) eine Rolle, die genauer betrachtet werden muss. Wir unterscheiden folgende Zeitsysteme:

Definition 1.10:

In *Zeit-analogen Systemen* ist die Zeit komplett kontinuierlich, d.h., jeder Zwischenwert zwischen zwei Zeitpunkten kann angenommen werden und ist Werte-relevant.

Als Folge hiervon muss jede Funktion f(t) für alle Werte $t \in [-\infty, \infty]$ bzw. für endliche Intervalle mit $t \in [t_0, t_1]$ definiert werden. Zeit-analoge Systeme sind fast immer mit Werte-Analogie gekoppelt. Zusammengefasst wird dies als analoge Welt bezeichnet.

Definition 1.11:

In *Zeit-diskreten Systemen* gilt, dass das System, beschrieben z.B. durch eine Funktion, von abzählbar vielen Zeitpunkte abhängt. Hierbei können abzählbar unendlich viele oder endlich viele Zeitpunkte relevant sein.

Folglich wird jede Funktion g(t) für alle Werte $t \in N$ (oder ähnlich mächtige Mengen) oder für $t \in \{t_0, t_1, \ldots t_k\}$ definiert. Zeit-diskrete Systeme sind fast immer mit Werte-Diskretheit gekoppelt, man spricht dann auch von der digitalen Welt.

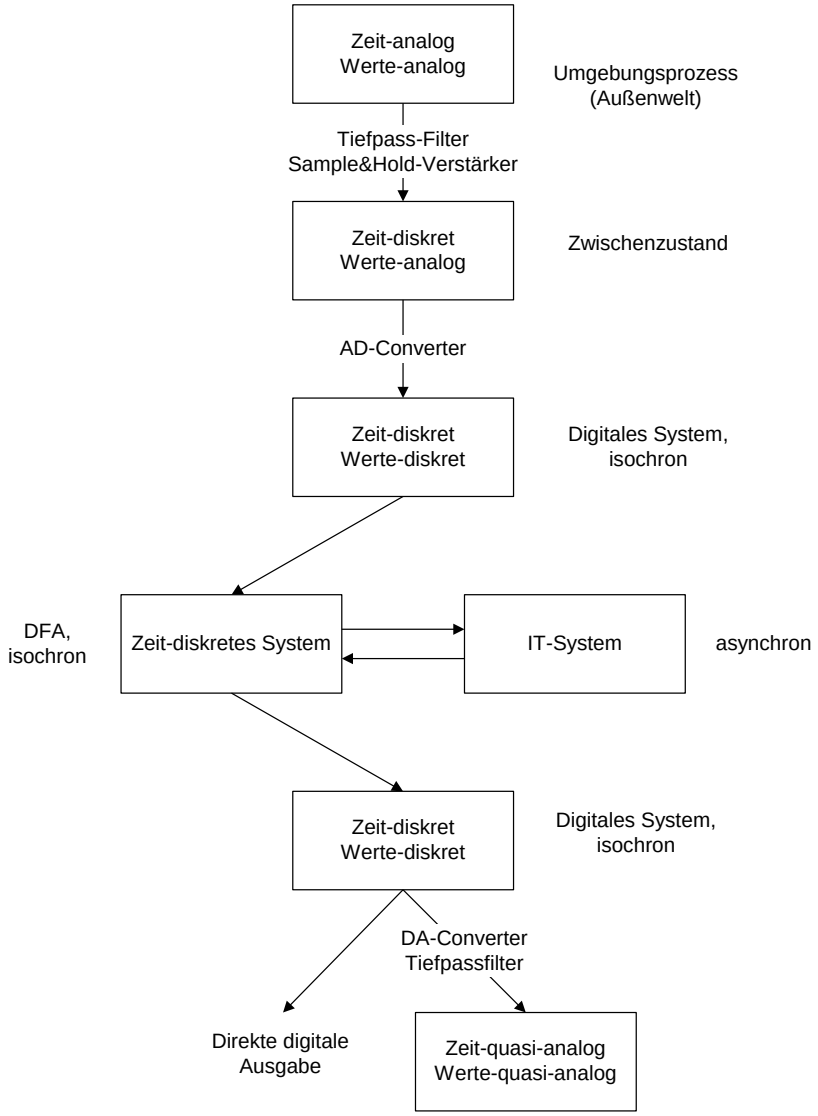


Bild 1.5 Übergänge zwischen den Zeitbindungen

Definition 1.12:

Zeit-unabhängige Systeme sind Systeme, die keine Zeitbindung besitzen. Dies bedeutet nicht, dass sie über die Zeit konstant sind, sie sind nur nicht explizit daran gebunden.

Hiermit wird deutlich, dass zwischen einer realen Zeit (Außenzeit) und der Programmlaufzeit (Innenzeit) unterschieden werden muss. Die Aufgabe einer Echtzeitprogrammierung besteht also darin, zwischen realer Zeit und Programmlaufzeit eine feste Beziehung herzustellen. Die Zeit-unabhängigen Systeme werden häufig auch als informationstechnische Systeme (IT-Systeme) bezeichnet (siehe hierzu auch die Einleitung zu diesem Kapitel).

In der Praxis sieht die Kopplung zwischen diesen drei Zeitbindungen so aus, dass Übergänge durch bestimmte Bausteine oder Vorgänge geschaffen werden. Bild 1.5 stellt dies zusammenfassend dar.

Hieraus lassen sich zwei Probleme identifizieren:

- Es gibt einen Informationsverlust beim Übergang zwischen der analogen und der Zeit- und Werte-diskreten Welt vor. Dieser Informationsverlust ist seit langem bekannt (Shannon, Abtasttheorem) und ausreichend behandelt.
- Im System liegt eine Kopplung zwischen isochronen und asynchronen Teilen vor. Die isochronen Teile behandeln den Umgebungsprozess mit gleicher Zeitbindung, während die asynchronen Systemteile ohne Bindung mit eigener Programmlaufzeit laufen, dennoch jedoch algorithmischen Bezug dazu haben. Diese Schnittstelle ist sorgfältig zu planen.

Die im letzten Aufzählungspunkt geforderte sorgfältige Planung der Schnittstelle führt dann zu den Echtzeitsystemen ($\rightarrow$ 2), bei denen die Anforderungen an das IT-System so gestellt werden, dass das System auf einem gewissen Level wieder isochron arbeitet.

1.3.2 Weitere Randbedingungen für eingebettete Systeme

Die Zeit spielt in Embedded Systems aus dem Grund eine übergeordnete Rolle, weil der Rechner in eine Maschine eingebettet ist, deren Zeitbedingungen vorherbestimmt sind. Insofern hat die Zeit eine übergeordnete Bedeutung.

Aber: Es existieren noch weitere Randbedingungen, insbesondere für den Entwurfsprozess:

- Power Dissipation/Verlustleistung: Welche Durchschnitts- und/oder Spitzenleistung ist vertretbar, gefordert, nicht zu unterschreiten usw.?
- Ressourcenminimierung: Nicht nur die Verlustleistung, auch die Siliziumfläche, die sich in Kosten niederschlägt, soll minimiert werden.

Als vorläufiges Fazit kann nun gelten, dass die Entwicklung für eingebettete Systeme bedeutet, eine Entwicklung mit scharfen und unscharfen Randbedingungen durchzuführen.

2 Echtzeitsysteme

Dieses Kapitel dient dazu, die im vorangegangenen Kapitel bereits skizzierten Probleme der Integration der Zeit noch näher zu spezifizieren und vor allem die Lösungen aufzuzeigen. Dies führt zu den Echtzeitsystemen, und im ersten Teil dieses Kapitels werden Definitionen und Entwicklungsmethoden hierzu formuliert.

Die wirkliche Problematik beginnt genau dann, wenn mehrere Algorithmen nebenläufig zueinander zum Ablauf kommen. Dies ist Inhalt des zweiten Teils, in dem nebenläufige Systeme betrachtet werden.

2.1 Echtzeit

2.1.1 Definitionen um die Echtzeit

Die DIN 44300 des Deutschen Instituts für Normung beschreibt den Begriff Echtzeit wie folgt [Sch05]:

Definition 2.1:

> Unter *Echtzeit* (*real time*) versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.

Demgegenüber wird im *Oxford Dictionary of Computing* das Echtzeitsystem wie folgt beschrieben:

Definition 2.2:

> Ein *Echtzeitsystem* (*real-time system*) ist ein System, bei dem der Zeitpunkt, zu dem Ausgaben vorliegen, bedeutend ist. Das liegt für gewöhnlich daran, dass die Eingabe mit einigen Änderungen der physikalischen Welt korrespondiert und die Ausgabe sich auf diese Änderungen beziehen muss. Die Verzögerung zwischen der Zeit der Eingabe und der Zeit der Ausgabe muss ausreichend klein für eine akzeptable ”*Rechtzeitigkeit*“ (*timeliness*) sein.

Echtzeitsysteme sind also Systeme, die korrekte Reaktionen innerhalb einer definierten Zeitspanne produzieren müssen. Falls die Reaktionen das Zeitlimit überschreiten, führt dies zu Leistungseinbußen, Fehlfunktionen und/oder sogar Gefährdungen für Menschen und Material.

Die Unterscheidung in harte und weiche Echtzeitsysteme wird ausschließlich über die Art der Folgen einer Verletzung der Zeitschranken getroffen:

Definition 2.3:

Ein Echtzeitsystem wird als *hartes Echtzeitsystem* (*hard real-time system*) bezeichnet, wenn das Überschreiten der Zeitlimits bei der Reaktion erhebliche Folgen haben kann. Zu diesen Folgen zählen die Gefährdung von Menschen, die Beschädigung von Maschinen, also Auswirkungen auf Gesundheit und Unversehrtheit der Umgebung.

Typische Beispiele hierfür sind einige Steuerungssysteme im Flugzeug oder im Auto, z.B. bei der Verbrennungsmaschine.

Definition 2.4:

Eine Verletzung der Ausführungszeiten in einem *weichen Echtzeitsystem* (*soft real-time system*) führt ausschließlich zu einer Verminderung der Qualität, nicht jedoch zu einer Beschädigung oder Gefährdung.

Beispiele hierfür sind Multimediasysteme, bei denen das gelegentlich Abweichen von einer Abspielrate von 25 Bildern/sek. zu einem Ruckeln o.ä. führt.

Als Anmerkung sei hier beigefügt, dass fast immer nur die oberen Zeitschranken aufgeführt werden. Dies hat seine Ursache darin, dass die Einhaltung einer oberen Zeitschranke im Zweifelsfall einen erheblichen Konstruktionsaufwand erfordert, während eine untere Schranke, d.h. eine Mindestzeit, vor der nicht reagiert werden darf, konstruktiv unbedenklich ist. Ein Beispiel für ein System, bei dem beide Werte wichtig sind, ist die Steuerung des Zündzeitpunkts bei der Verbrennungsmaschine: Dieser darf nur in einem eng begrenzten Zündintervall kommen.

2.1.2 Ereignissteuerung oder Zeitsteuerung?

Es stellt sich nun unmittelbar die Frage, wie die harten Echtzeitsysteme denn konzipiert sein können. Auf diese Frage wird im Kapitel 3.2 noch näher eingegangen, denn die Grundsatzentscheidung, welches Design zum Tragen kommen soll, hat natürlich erhebliche Konsequenzen für die gesamte Entwicklung.

Zwei verschiedene Konzeptionen, die in der Praxis natürlich auch gemischt vorkommen können, können unterschieden werden: *Ereignisgesteuerte* (*event triggered*) und *zeitgesteuerte* (*time triggered*) *Systeme*.

Ereignis-gesteuerte Systeme werden durch Unterbrechungen gesteuert. Liegt an einem Sensor ein Ereignis (was das ist, muss natürlich definiert sein) vor, dann kann er eine *Unterbrechungsanforderung* (*interrupt request*) an den Prozessor senden und damit auf seinen Bedienungswunsch aufmerksam machen.

Definition 2.5:

Eine *asynchrone Unterbrechung* (*Asynchronous Interrupt Request, IRQ*) ist ein durch das Prozessor-externe Umfeld generiertes Signal, das einen Zustand anzeigt und/oder eine Behandlung durch den Prozessor anfordert. Dieses Signal ist nicht mit dem Programmlauf synchronisiert. Die Behandlung der Unterbrechung erfolgt

im Rahmen der *Interrupt Service Routine* (ISR), die für jede Unterbrechung im Softwaresystem definiert sein muss.

Bei zeitgesteuerten Systemen erfolgt keine Reaktion auf Eingabeereignisse, die Unterbrechungen werden lediglich durch einen, ggf. mehrere periodische Zeitgeber (Timer) ausgelöst. Sensoren werden dann vom Steuergerät aktiv abgefragt.

Dieses Verfahren hat den großen Vorteil, dass das Verhalten sämtlicher Systemaktivitäten zur Compilezeit vollständig planbar. Dies ist gerade für den Einsatz in Echtzeitsystemen ein erheblicher Vorteil, da á priori überprüft werden kann, ob Echtzeitanforderungen eingehalten werden. Dies wird in Abschnitt 3.2 genauer untersucht.

Der Vorgänger des zeitgesteuerten Designs wurde *Polling* genannt. Hierunter wird das ständige (quasi-zyklische), im Programm verankerte Abfragen von Prozesszuständen oder Werten verstanden, während das zeitgesteuerte Verfahren nicht ständig (also durch den Programmlauf bestimmte), sondern zu festgelegten Zeiten abfragt. Man beachte hierbei die Unterscheidung zwischen realer Zeit und Programmlaufzeit ($\rightarrow$ 1.3.1).

Das Design dieser Zeitsteuerung muss allerdings sehr präzise durchgeführt werden, um die Ereignisse zeitlich korrekt aufzunehmen und zu verarbeiten. Ggf. müssen auch Zwischenpufferungen (z.B. bei einer schnellen Datenfolge) eingefügt werden. Um den zeitlichen Ablauf und seine Bedingungen quantifizieren zu können, seien folgende Zeiten definiert:

Definition 2.6:

Die *Latenzzeit* (*Latency Time*) ist diejenige vom Auftreten eines Ereignisses bis zum Start der Behandlungsroutine. Diese Zeit kann auf den Einzelfall bezogen werden, sie kann auch als allgemeine Angabe (Minimum, Maximum, Durchschnittswert mit Streuung) gewählt werden.

Definition 2.7:

Die *Ausführungszeit* (*Service Time*) ist die Zeit zur reinen Berechnung einer Reaktion auf ein externes Ereignis. In einem deterministischen System kann diese Zeit bei gegebener Rechengeschwindigkeit prinzipiell vorherbestimmt werden.

Definition 2.8:

Die *Reaktionszeit* (*Reaction Time*) ist diejenige Zeit, die vom Anlegen eines Satzes von Eingangsgrößen an ein System bis zum Erscheinen eines entsprechenden Satzes von Ausgangsgrößen benötigt wird.

Die Reaktionszeit setzt sich aus der Summe der Latenzzeit und der Ausführungszeit zusammen, falls die Service Routine nicht selbst noch unterbrochen wird.

Definition 2.9:

Die *Frist* (*Dead Line*) kennzeichnet den Zeitpunkt, zu dem die entsprechende Reaktion am Prozess spätestens zur Wirkung kommen muss. Diese Fristen stellen eine der wesentlichen Randbedingungen des Umgebungsprozesses dar.

Dies bedeutet also, dass zu jedem zu den Echtzeitkriterien zählenden Ereignis eine Frist definiert sein muss, innerhalb derer die Reaktion vorliegen muss. Folglich ist nicht die Schnelligkeit entscheidend, es ist Determinismus im Zeitsinn gefragt.

2.1.3 Bemerkungen zu weichen und harten Echtzeitsystemen

Die Konzeption eines harten Echtzeitsystems und vor allem der Nachweis dieser Fähigkeit ist außerordentlich schwierig, insbesondere, wenn man bedenkt, dass die Unterschiede im Laufzeitbedarf für einzelne Aufgaben sehr hoch sein können (für Fußball-spielende Roboter wird von 1:1000 berichtet). Es muss also auf den Maximalfall ausrichtet werden, wenn das System wirklich in jedem Fall in festgelegten Zeiten reagieren soll.

Man muss allerdings auch sagen, dass dieses Echtzeitkriterium aufweichbar ist (was auch z.B. von Anbietern der Echtzeit-Betriebssysteme gemacht wird):

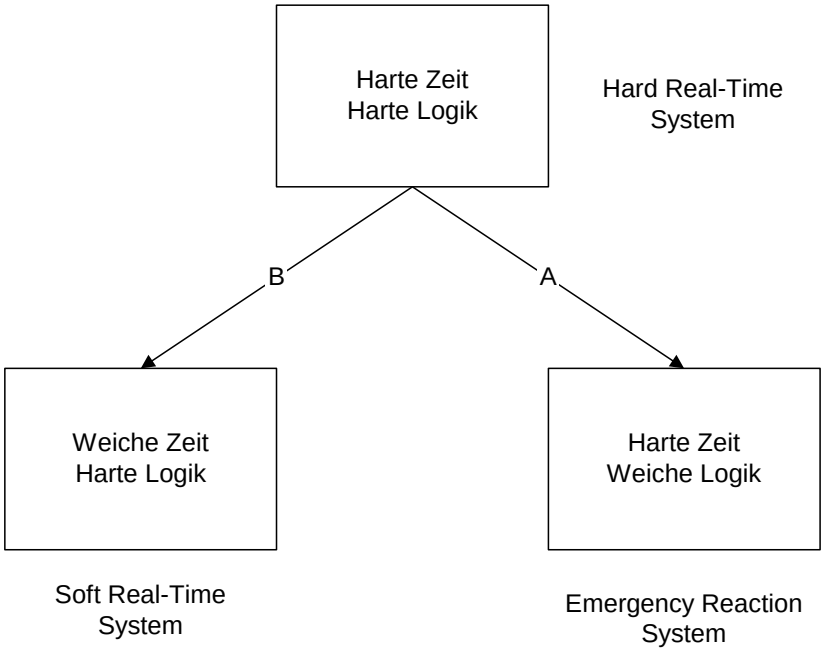


Bild 2.1 Darstellung verschiedener Applikationsklassen

Kann die vollständige, harte Reaktion nicht eingehalten werden, so bietet sich die Wege A und B in Bild 2.1 an. Weg A gilt dabei für Systeme bzw. Ereignisse, bei denen aus einer verspäteten Reaktion Schädigungen bis zur Zerstörung resultieren können. Hier wird nicht mit dem vollständig berechneten Ergebnis gehandelt, sondern mit einem ungefähren Wert, also eine Art rechtzeitige Notreaktion.

Weg B ist der gewöhnliche Ausweg. Hier werden Systeme vorausgesetzt, bei denen eine zeitliche Überschreitung zu einer Güteverminderung (Soft Degradation), nicht jedoch zu einer Schädigung führt. Wie bereits erwähnt bezeichnet man dies dann als *Soft Real-Time*, und dies wird gerne für Betriebssysteme genutzt.

2.2 Nebenläufigkeit

Nebenläufigkeit bildet das Grundmodell für Multiprocessing und Multithreading [Sch05]. Zwei Prozesse bzw. Threads sind dann nebenläufig, wenn sie unabhängig voneinander arbeiten können und es keine Rolle spielt, welcher der beiden Prozesse/ Threads zuerst ausgeführt oder beendet wird. Indirekt können diese Prozesse dennoch voneinander abhängig sein, da sie möglicherweise gemeinsame Ressourcen beanspruchen und untereinander Nachrichten austauschen.

Hieraus kann eine Synchronisation an bestimmten Knotenpunkten im Programm resultieren. Hier liegt eine Fehlerquelle, denn es kann hier zu schwerwiegenden Fehlern, *Verklemmungen* (*deadlocks*) und damit zu einem Programmabsturz kommen.

Die Hauptargumente, warum es trotz der Probleme (sprich: neue Fehlermöglichkeiten für Softwareentwickler) sinnvoll ist, Programme nebenläufig zu entwickeln, sind:

- Die Modellierung vieler Probleme wird dadurch vereinfacht, indem sie als mehr oder weniger unabhängige Aktivitäten verstanden werden und entsprechend durch Sprachkonstrukte umgesetzt werden können. Jede Aktivität kann dann isoliert betrachtet werden, nur die Kommunikation und Synchronisation ist zu beachten. Nebenläufigkeit führt hier zu einer abstrakteren Modellierung, und ob die entstandene Nebenläufigkeit dann wirklich zu einer gleichzeitigen Bearbeitung führt, ist nebensächlich.

Ein Beispiel hierzu wird in Kapitel 4 behandelt, wo Messwertaufnahme und Auswertung in zwei miteinander gekoppelte, aber ansonsten getrennte Aktivitäten modelliert und auch implementiert werden.

- Die Anzahl der ausführenden Einheiten in einem Rechner kann durchaus > 1 sein. Im Zeitalter von Hardware/Software Co-Design, Multi- und Manyprozessorcores, konfigurierbaren Prozessoren, Prozessoren mit eigenem Peripherieprozessor usw. können Aufgaben auf verschiedene Teile abgebildet werden, und dazu müssen sie auch dergestalt modelliert sein. Hier wird die Performance

des Systems entscheidend verbessert, wenn die parallelen Möglichkeiten auch wirklich ausgenutzt werden.

2.2.1 Multiprocessing und Multithreading

Mit *Multitasking* wird allgemein die Fähigkeit von Software (beispielsweise Betriebssystemen) bezeichnet, mehrere Aufgaben scheinbar gleichzeitig zu erfüllen. Dabei werden die verschiedenen Tasks in so kurzen Abständen immer abwechselnd aktiviert, dass für den Beobachter der Eindruck der Gleichzeitigkeit entsteht. Man spricht hier auch oft von Quasi-Parallelität, aber mikroskopisch wird natürlich nichts wirklich parallel zueinander bearbeitet.

Doch was ist eine *Task*? Dies wird üblicherweise als allgemeiner Überbegriff für Prozesse und Threads (= Leichtgewichtsprozesse) genannt. Nun sind auch diese beiden schwer zu unterscheiden (zumindest präzise zu unterscheiden), aber meist reicht auch schon eine etwas unscharfe Definition.

Ein *Prozess* (*process*) ist ein komplettes, gerade ablaufendes Programm. Zu diesem Prozess gehören der gesamte Code und die statischen und dynamisch angelegten Datenbereiche einschließlich Stack, Heap und Register. Der Code wiederum kann mehrere Teile enthalten, die unabhängig voneinander arbeiten können. Solche Teile werden *Threads* (Aktivitätsfäden) genannt.

Definition 2.10:

Ein *Thread* ist ein Aktivitätsträger eines Prozesses mit minimalem eigenem Kontext. Mit *Aktivitätsträger* wird ein in sich geschlossener Bearbeitungsstrang bezeichnet. Der minimale Kontext betrifft diejenigen Daten bzw. Speichereinheiten (Register), die ausschließlich dem Thread zur Verfügung stehen.

Welche Formen des Multiprocessing oder Multithreading gibt es denn? Das am häufigsten angewandte Konzept ist das *präemptive Multiprocessing*. Hier wird von einem Betriebssystem(kern) der aktive Prozess nach einer Weile verdrängt, zu Gunsten der anderen. Diese Umschaltung wird *Scheduling* genannt.

Die andere Form ist das *kooperative Multiprocessing*, das von jedem Prozess erwartet, dass dieser die Kontrolle an den Kern von sich aus zurückgibt. Letztere Version birgt die Gefahr in sich, dass bei nicht-kooperativen Prozessen bzw. Fehlern das gesamte System blockiert wird. Andererseits ist das kooperative Multiprocessing sehr einfach zu implementieren, auch innerhalb einer Applikation, daher wird dies als Beispiel in Kapitel 4 realisiert.

Beim Multithreading ist es ähnlich, wobei allerdings die Instanz, die über das Scheduling der Threads entscheidet, auch im Programm liegen kann (Beispiel: Java-Umgebung). Das Umschalten zwischen Threads eines Prozesses ist dabei wesentlich weniger aufwändig, verglichen mit Prozessumschaltung, weil im gleichen Adressraum verweilt wird. Allerdings sind auch die Daten des gesamten Prozesses durch alle Threads manipulierbar.

2.2.2 Prozesssynchronisation und –kommunikation

Die *Prozesssynchronisation* dient dem Ablauf der nebenläufigen Programmteile und ermöglicht eine Form der Wechselwirkung zwischen diesen. Das Warten eines Prozesses auf ein Ereignis, das ein anderer auslöst, ist die einfachste Form dieser Prozesssynchronisation (gleiches gilt auch für Threads).

Die Prozesskommunikation erweitert die Prozesssynchronisation und stellt somit dessen Verallgemeinerung dar. Hier muss es neben den Ereignissen auch Möglichkeiten geben, die Daten zu übertragen. Die praktische Implementierung ist dann z.B. durch ein Semaphoren/Mailbox-System gegeben: Über Semaphoren wird kommuniziert, ob eine Nachricht vorliegt, in der Mailbox selbst liegt dann die Nachricht. Für ein Multithreadingsystem kann dies direkt ohne Nutzung eines Betriebssystems implementiert werden, da alle Threads auf den gesamten Adressraum zugreifen können. Dies gilt nicht für Multiprocessingsysteme, hier muss ein Betriebssystem zur Implementierung der Mailbox und der Semaphoren verwendet werden.

Bei dieser Kommunikation wie auch der einfachen Synchronisation kann es zu Verklemmungen kommen. Eine Menge von Threads (Prozessen) heißt *verklemmt*, wenn jeder Thread (Prozess) dieser Menge auf ein Ereignis im Zustand „blockiert“ wartet, das nur durch einen anderen Thread (Prozess) dieser Menge ausgelöst werden kann. Dies ist im einfachsten Fall mit zwei Threads (Prozessen) möglich: Jeder Thread wartet blockierend auf ein Ereignis des anderen.

Im Fall der Prozess- oder Threadkommunikation kann dies gelöst werden, indem nicht-blockierend kommuniziert wird: Die Threads (Prozesse) senden einander Meldungen und Daten zu, warten aber nicht darauf, dass der andere sie auch abholt. Am Beispiel in Kapitel 4 wird gezeigt, dass dies auch notwendig für die Echtzeitfähigkeit ist, allerdings sollte nicht übersehen werden, dass hierdurch Daten auch verloren gehen können.

2.2.3 Zeitliche Modelle für die Kommunikation

Bezüglich der Zeit für das Aufbauen der Kommunikation zwischen zwei Prozessen (Threads) gibt es drei Grundannahmen: *Asynchron*, *perfekt synchron* (mit Null-Zeit) und *synchron* (mit konstanter Zeit). Asynchrone Kommunikation bedeutet in diesem Fall, dass die Kommunikationspartner sozusagen zufällig in Kontakt treten (wie Moleküle in einem Gas) und dann wechselwirken. Dieses Modell, als *chemisches Modell* bezeichnet, ist daher nichtdeterministisch und für eingebettete Systeme unbrauchbar.

> Anmerkung: Spricht man im Zusammenhang von Network-on-Chip (NoC) von asynchroner Kommunikation, so ist damit selbst-synchronisierende Kommunikation gemeint. Für RS232, auch eine „asynchrone“ Schnittstelle, bedeutet asynchron, dass der Beginn einer Aussendung für den Empfänger spontan erfolgt. Auf höherer Ebene ist diese Kommunikation natürlich nicht zufällig, sondern geplant.

Das *perfekt synchrone Modell* geht davon aus, dass Kommunikation keine Zeit kostet, sondern ständig erfolgt. Dies lehnt sich an die Planetenbewegung an, wo die Gravitation untereinander und mit der Sonne zu den Bahnen führt, und wird deshalb auch *Newtonsches Modell* genannt. Die so genannten *synchronen Sprachen* basieren auf diesem Modell.

Das dritte Modell, das *synchron*, aber mit konstanter Zeitverzögerung kommuniziert, wird auch *Vibrationsmodell* genannt. Dieser Name entstammt der Analogie zur Kristallgitterschwingung, bei der eine Anregung sich über den Austausch von Phononen fortpflanzt.

Wozu dienen diese Kommunikationsmodelle? Der Hintergrund hierzu besteht darin, Kommunikation und Betrieb in nebenläufigen, ggf. auch verteilten Systemen modellieren zu können. Die Annahme einer perfekt synchronen Kommunikation beinhaltet eigentlich nicht, dass „Null-Zeit“ benötigt wird, vielmehr ist die Übertragungszeit einer Nachricht kleiner als die Zeitspanne zur Bestimmung eines neuen Zustands im Empfänger. Dies bedeutet, dass sich das gesamte System auf diese Meldungen synchronisieren kann und die Kommunikation keinen Beitrag zu Wartezeiten leistet.

3 Design von eingebetteten Systemen

Dieses Kapitel dient dem Zweck, den Zusammenhang zwischen den Systemen, die programmiert werden können, den Entwurfssprachen und der in Kapitel 1 bereits diskutierten Randbedingung *Echtzeitfähigkeit* darzustellen.

Diese Diskussion soll konstruktiv gestaltet werden, d.h. weniger theoretische Konzepte stehen im Vordergrund, vielmehr sollen Lösungsmöglichkeiten und Design Pattern (Architekturmuster) aufgezeigt werden. Zu diesem Zweck werden Ansätze zur Lösung des Echtzeitproblems diskutiert, und zwar in zwei Abschnitten: Abschnitt 3.1 diskutiert, wie die reale Zeit mit dem Ablauf im Mikroprozessor gekoppelt werden kann, Abschnitt 3.2 ist dann dem Systemdesign gewidmet. Im Anschluss daran folgt eine Einführung in Softwareentwicklungssprachen.

3.1 Ansätze zur Erfüllung der zeitlichen Randbedingungen

Gerade in eingebetteten Systemen ist der entscheidende Zeitbegriff derjenige der 'Reaktionszeit', der mit dem deterministischen Echtzeitverhalten des Systems korreliert. Hier geht es nicht um Einsparungspotenzial, sondern um die Erfüllung der zeitlichen Randbedingungen. Um dies zu erreichen, bieten sich 'Design Pattern' an, die in den folgenden Abschnitten dargestellt werden sollen.

3.1.1 Technische Voraussetzungen

Zunächst müssen einige Voraussetzungen für die hier dargestellten Ansätze erläutert werden. Als technische Basis sei ein *Mikroprozessor-basiertes Rechnersystem* angenommen, das kein Betriebssystem und somit keinen Scheduler (= Einheit zur Rechenzeitvergabe an unterschiedliche Tasks) zur Verfügung hat. Diese Einschränkung kann prinzipiell jederzeit aufgehoben werden, allerdings lassen sich an dem Betriebssystem-losen System die Einzelheiten zum Systemdesign präziser darstellen.

In der Praxis werden solche Systeme gerne als „kleine" Systeme eingesetzt. Es ist das Ziel dieses Unterkapitels und des Beispiels aus Kapitel 4, ein Softwaredesign vorzustellen, dass dennoch aus mehreren Teilen besteht und ein Applikations-internes Scheduling enthält.

Weiterhin soll der Begriff *Thread* hier in erweitertem Sinn genutzt werden. Wie in Abschnitt 2.2.1 dargestellt stellt ein Thread einen so genannten Programmfaden dar, der innerhalb eines (Software-)Prozesses abläuft bzw. definiert ist. Diese Definition ist vergleichsweise schwammig.

Für die Zwecke dieses Skripts sei der Begriff *Thread* erweitert. Ohne Betriebssystem existiert nur ein Prozess, der das einzige Programm, das abläuft, darstellt. Innerhalb dieses Prozesses ist ein Thread wie folgt definiert:

Definition 3.1:

Ein *Thread* ist ein in sich geschlossener Programmteil, der mit anderen, nicht zu diesem Programmteil gehörenden Teilen des Prozesses nur indirekt, d.h. ohne feste zeitliche Kopplung (zeitlich asynchron) kommuniziert.

Die zeitlich asynchrone Kopplung lässt sich natürlich leicht (algorithmisch) synchronisieren, indem eine bidirektionale Kopplung vereinbart wird (Handshake mit Warten auf Acknowledge). Es ist aber wichtig, das Interface zwischen verschiedenen Threads derart zu gestalten, damit ein entsprechendes Systemdesign (siehe auch 10.3, Hardware/Software Co-Design) möglich wird. Die Inter-Thread-Kommunikation kann dann mithilfe folgender Mechanismen erfolgen:

- Signale (Hardware) oder gemeinsame (shared) Variablen (Semaphoren, Mailboxen)
- Software-Ereignisse, die mithilfe eines Dispatcher/Scheduler den Threads zur Bearbeitung zugeteilt werden.

3.1.2 Zeit-gesteuerte Systeme (Time-triggered Systems)

Eine Möglichkeit, den realen Bezug zwischen Realzeit und Programmlaufzeit zu schaffen, besteht darin, eine *feste Zeitplanung* einzuführen. Hierzu müssen natürlich alle Aufgaben bekannt sein.

Weiterhin müssen folgende Voraussetzungen gelten:

- Die Verhaltensweisen des Embedded Systems und des Umgebungsprozesses müssen zur *Übersetzungszeit* (*compile time*) vollständig definierbar sein.
- Es muss möglich sein, eine gemeinsame Zeit über alle Teile des Systems zu besitzen. Dies stellt für ein konzentriertes System kein Problem dar, bei verteilten, miteinander vernetzten Systemen muss aber diesem Detail erhöhtes Augenmerk gewährt werden.
- Für die einzelnen Teile des Systems, also für jeden Thread, müssen exakte Werte für das Verhalten bekannt sein. Exakt heißt in diesem Zusammenhang, dass die Zeiten im Betrieb nicht überschritten werden dürfen. Es handelt sich also um eine Worst-Case-Analyse, die mit Hilfe von Profiling, Simulation oder einer exakten Laufzeitanalyse erhalten werden.

Hieraus ergibt sich dann ein planbares Verhalten. Man baut dazu ein *statisches* Scheduling (= Verteilung der Rechenzeit zur Compilezeit) auf, indem die Zykluszeit (= Gesamtzeit, in der aller Systemteile einmal angesprochen werden) aus dem Prozess abgeleitet wird.

Die praktische Ausführung eines Zeit-gesteuerten Systems kann dabei auf zwei Arten erfolgen: Auslösung durch Timer-Interrupt und ein kooperativer Systemaufbau:

- Beim Aufbau mit Hilfe von Timer-Interrupts wird ein zyklischer Interrupt (→ Definition 2.5) aufgerufen. Dies ist zwar auch eine Art Ereignis-Steuerung dar, sie ist aber geplant und streng zyklisch auftretend. In der Interrupt-Service-Routine (→ Definition 2.5) werden dann aller Prozesszustände abgefragt und entsprechende Reaktionsroutinen aufgerufen.
- Beim kooperativen Systemaufbau ist jeder Thread verpflichtet, eine Selbst-Unterbrechung nach einer definierten Anzahl von Befehlen einzufügen. Diese Unterbrechung ist als Aufruf eines Schedulers implementiert, dieser ruft dann einen weiteren Thread auf. Dieses Verfahren ist unschärfer und aufwendiger (die Zeiten müssen festgelegt werden), sodass meist die erste Variante bevorzugt wird.

Innerhalb der entstandenen Zykluszeit kann dann das Gefüge der Aufgaben verteilt werden. Im einfachsten Fall eines Zyklus, d.h. einer kritischen Aufgabe, müssen folgende Ungleichungen gelten:

$$t_{cycle} \leq t_{critical} \tag{3.1}$$

$$t_{thread} \leq t_{cycle} \tag{3.2}$$

Mit $t_{critical}$ ist hierbei die systemkritische Zeit angenommen, die für ein ordnungsgemäßes Arbeiten nicht überschritten werden darf. Diese Zeit wird durch den Prozess definiert und entspricht etwa der maximal möglichen Reaktionszeit. Zu einer genaueren Herleitung siehe 3.2.1.

Ungleichung (3.2) kann auch mehrere Threads enthalten, die ggf. sogar mehrfach berücksichtigt werden, weil sie beispielsweise mehrfach in einem Zyklus vorkommen müssen. Für diese Threads kann eine andere systemkritische Zeit gelten, und diesem Umstand kann man durch den (zeitlich verteilten) Mehrfachaufruf Rechnung tragen.

Die Zeitdefinition im Mikrocontroller kann durch einen Timer erfolgen, dieser Timer stellt eine Hardwareeinheit dar, die Takte zählen kann. Durch die Kopplung des Takts mit der Realzeit aufgrund der fest definierten Schwingungsdauer ergibt sich hierdurch ein Zeitgeber bzw. –messer.

Die exakte Bestimmung der Zykluszeit t_{cycle} wird in Abschnitt 3.2.1 beschrieben. Hier sind ggf. mehrere Bedingungen zu berücksichtigen. Die Berechnungszeit t_{thread} aus Ungleichung (3.2) kann durch die Bestimmung der Worst-Case-Execution-Time (WCET) in Kombination mit der Worst-Case-Interrupt-Disable-Time (WCIDT) berechnet bzw. geschätzt werden (→ 3.2.3).

Diese Variante hat folgende Vor- und Nachteile:

- Garantierte Einhaltung kritischer Zeiten

- Bei verteilten Systemen Erkennung von ausgefallenen Teilen (durch Planung von Kommunikation und Vergleich in den anderen Systemteilen)
- Das System muss hoch dimensioniert werden, weil für alle Teile die Worst-case-Laufzeiten angenommen werden müssen.
- Die Einbindung zeitunkritischer Teile erfolgt entweder unnötig im Scheduling, oder das System wird durch die Zweiteilung komplexer.
- Die Kombination mehrerer, Zeit-gesteuerter Tasks kann sich als sehr aufwendig erweisen, falls die einzelnen Zeitabschnitte in ungünstigem Verhältnis zueinander liegen (siehe nächsten Abschnitt).

3.1.3 Kombination mehrerer Timer-Interrupts

Als nächstes muss die Kombination mehrerer Aufgaben mit Zeitbindung diskutiert werden. Grundsätzlich ist es natürlich möglich, mehrere (unterschiedlich laufende) Zeitsteuerungen durch mehrere Timer-Interrupts durchzuführen. Beispiele hierfür sind die Kombination mehrerer Schnittstellen, etwa RS232 und I²C-Bus, die mit unterschiedlichen Frequenzen arbeiten, sowie die Kombination aus Messwertaufnahme und serieller Schnittstelle.

In diesem Fall wird für jeden Timer die entsprechende Zeitkonstante gewählt, also etwa die Zeit, die zwischen zwei Messungen oder zwei Transmissionen liegt (→ 3.2.1). Das Problem, das sich hierbei stellt, ist die zufällige zeitliche Koinzidenz mehrerer Interrupts, die behandelt werden muss. Das gleichzeitige oder doch sehr kurz aufeinander folgende Eintreffen der Requests bedeutet, dass die Behandlung eines Vorgangs gegenüber dem zweiten zurückgestellt wird. Dies muss zwangsläufig in jeder Kombination möglich sein, da nichts vorbestimmbar ist.

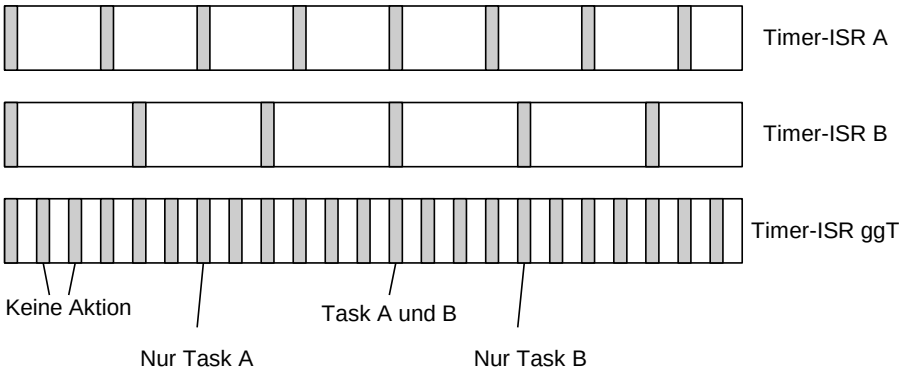


Bild 3.1 Zusammenfügen mehrerer Zeit-gesteuerter ISR zu einer Routine

Ein anderer Weg ist ggf. einfacher zu implementieren: Alle Teilaufgaben, die zyklisch auftreten, werden in einer einzigen ISR, die von einem zyklisch arbeitenden

Timer aufgerufen wird, zusammengefasst. Die Probleme, die dabei auftreten, liegen weniger im grundsätzlichen Design als vielmehr darin, mit welcher Frequenz bzw. mit welchem Zeitwert die ISR aufgerufen wird.

Während bei einer einzigen Aufgabe mit streng zyklischem Verhalten die Wahl einfach ist – die Zeitkonstante, die zwischen zwei Messungen oder zwei Transmissionen liegt, wird als der Timerwert gewählt –, muss nunmehr der größte gemeinsamen Teiler (ggT) der Periodenzeiten als Zeitwert gewählt werden.

Die ggT-Methode (Bild 3.1) ist so vorteilhaft, weil zu Beginn einer Timer-ISR bestimmt werden kann, was alles (und auch in welcher Reihenfolge) behandelt werden soll. Hierdurch lassen sich auch Zeitverschiebungen planen bzw. bestimmen. Andererseits kann der ggT-Ansatz sehr schnell in ein nicht-lauffähiges System münden. Die Anzahl der ISR pro Zeiteinheit kann stark zunehmen (→ Bild 3.1), und jeder Aufruf einer ISR erfordert einen zeitlichen Overhead, auch wenn keine weitere Routine darin abläuft. Als Faustregel sollte man mit mindestens 10 – 20 Befehlsausführungszeiten rechnen, die für Interrupt-Latenzzeit, Retten und Restaurieren von Registern und den Rücksprung in das Programm benötigt werden. In einem System, das 1 µs Befehlsausführungszeit hat und alle 200 µs unterbrochen wird, sind das aber bereits 5 – 10 % der gesamten Rechenzeit, die unproduktiv vergehen. Daher sollte, soweit dies möglich ist, die Periode so gewählt sein, dass der ISR-Overhead klein bleibt (< 5%).

Im Idealfall besteht darin, die Zykluszeiten gegenseitig anzupassen, so dass der ggT gleich dem kürzesten Timerwert ist. Dies führt zumindest zu einem System, das keine ISR-Aufrufe ohne Netto-Aktion (wie in Bild 3.1 dargestellt) hat.

3.1.4 Flexible Lösung durch integrierte Logik

Die Tatsache, dass durch die Wahl des ggT aller Zykluszeiten als die einzige Zykluszeit im Allgemeinen ”leere“ Unterbrechungen erzeugt werden, lässt sich dadurch umgehen, dass man von der periodischen Erzeugung abgeht und nun eine bedarfsgerechte Generierung einführt.

Dies ist durch die Belassung bei mehreren Timern und anschließende OR-Verknüpfung der Unterbrechungssignale zu erreichen, wie Bild 3.2 darstellt. Damit ist dann ein effizientes Timingschema für die Unterbrechungen erzeugt, und die Unterbrechungsroutine würde unterscheiden, welche Aktionen durchzuführen wären.

Dies kann beliebig ausgestaltet werden, und sehr komplexe Interrupt-Schemata können erzeugt werden. Allerdings bleibt festzustellen, dass übliche Mikrocontroller die hierzu notwendige Hardware nicht enthalten, nur Derivate mit umfangreicher Peripherie bieten meist Timer-Arrays mit (begrenzter) Kombinationsfähigkeit an. Diese Form der Lösung bleibt damit meist den rekonfigurierbaren Prozessoren (Mikroprozessor + programmierbare Logik) bzw. der Zusammenstellung solcher Komponenten auf Boardlevel vorbehalten.

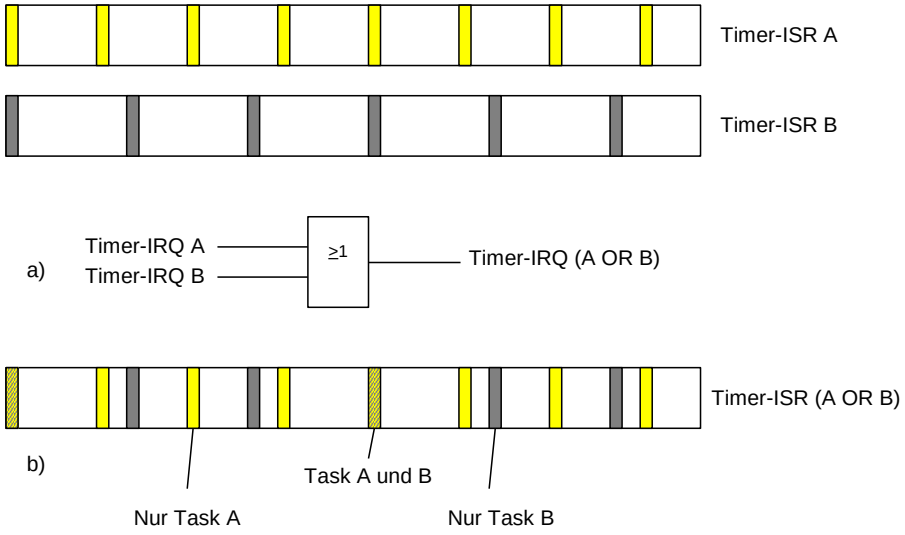


Bild 3.2 Zusammenfassung zweier Unterbrechungsquellen mittels Hardware
a) Verknüpfung der IRQ-Signale b) resultierendes Timingschema

3.1.5 Ereignis-gesteuerte Systeme (Event-triggered Systems)

Timersignale stellen zwar auch eine Unterbrechung des üblichen Programmlaufes dar, allerdings ist dies grundsätzlich planbar, während Unterbrechungen aus dem Prozessumfeld nicht planbar sind.

In einem Ereignis-gesteuerten System reagiert das Gesamtprogramm auf die Ereignisse des Prozesses. Insbesondere werden die Prozesszustände nicht zyklisch abgefragt, sondern es werden Zustandsänderungen an den Prozessor per IRQ (→ 8.3) gemeldet.

Diese Form der Systemauslegung, die selten in reiner Form auftritt, bedingt natürlich einen vollkommen anderen Systemansatz:

1. Der Prozess muss mit exklusiver Hardware ausgestattet sein, die ein Interface zum Prozessor bildet. Diese Hardware muss Zustandsänderungen erkennen und per IRQ zum Prozessor signalisieren.
2. Im Prozessor und (höchstwahrscheinlich) dem Interrupt-Request-Controller muss ein Priorisierungssystem festgelegt werden, das die IRQs in ein Prioritätssystem zwingt und entsprechend behandelt. Zu dieser Priorisierungsstrategie gehören auch Fragen wie ”Unterbrechungen von Unterbrechungs-Serviceroutinen“.

3. Es ist wahrscheinlich, dass neben den IRQ-Serviceroutinen (ISR) auch weitere, normale Programme existieren. Dies erfordert eine Kopplung zwischen ISR und Hauptprogramm.

Hieran ist zu erkennen, dass die Planung dieses Systems alles andere als einfach ist. Insbesondere stecken Annahmen in dem IRQ-Verhalten des Prozesses, die Aussagen zur Machbarkeit erst ermöglichen, so z.B. eine maximale Unterbrechungsrate.

Unter bestimmten Umständen kann die Erfüllung der Realtime-Bedingungen äquivalent zum Zeit-gesteuerten Design garantiert werden. Die Bedingungen hierzu sind:

- keine ISR kann unterbrochen werden (dies ist im Zeit-gesteuerten Design implizit eingeschlossen)
- für jeden IRQ ist eine maximale Frequenz des Auftretens und eine maximale Reaktionszeit gegeben

dann gilt verhält sich das Ereignis-gesteuerte System im Maximalfall entsprechend wie das Zeit-gesteuerte System mit gleichen Zykluszeiten und kann entsprechend ausgelegt werden. Scheinbar spart man Systemkapazität, weil die maximale Auftrittsfrequenz nicht unbedingt eintreten muss, für das Design und die Systemauslegung muss jedoch mit dem Worst-Case gerechnet werden. Insgesamt gilt hier also:

- Bei ’weicher‘ Echtzeit ist eine gute Anpassung an die real benötigten Ressourcen möglich.
- Die Einbindung zeitunkritischer Teile ist sehr gut möglich, indem diese im Hauptprogramm untergebracht werden und so automatisch die übrig bleibende Zeit zugeteilt bekommen.
- Die Bestimmung und der Nachweis der Echtzeitfähigkeit sind außerordentlich schwierig.
- Bei harten Echtzeitbedingungen droht eine erhebliche Überdimensionierung des Systems.
- Die Annahme der maximalen IRQ-Frequenz ist meist eine reine Annahme, die weder überprüfbar und automatisch einhaltbar ist. So können z.B. prellende Schalterfunktionen IRQs mehrfach aufrufen, ohne dass dies in diesem System vermieden werden kann.

Gerade der letzte Punkt ist kritisch, denn die Annahme kann durch die Praxis falsifiziert werden. Hierfür hilft eine Variante im nächsten Abschnitt, aber es bleibt immer noch die Aussage, dass die Systemauslegung zur Erreichung der Echtzeitfähigkeit nicht oder nur marginal wenig gemindert werden kann.

3.1.6 Modified Event-driven Systems

Einer der wesentlichen Nachteile der Ereignis-gesteuerten Systeme liegt in der Annahme, dass die asynchronen Ereignisse mit einer maximalen Wiederholungsfrequenz auftreten. Diese Annahme ist notwendig, um die Machbarkeit bzw. die reale Echtzeitfähigkeit nachweisen zu können.

Andererseits zeigen gerade die Ereignissteuerungen eine bessere Ausnutzung der Rechenleistung, weil sie den Overhead der Zeitsteuerung nicht berücksichtigen müssen. Es stellt sich die Frage, ob ein Ereignis-gesteuertes System nicht so modifiziert werden kann, dass die Vorteile bleiben, während die Nachteile aufgehoben oder gemildert werden.

Der Schlüssel hierzu liegt in einer Variation der Hardware zur Übermittlung und Verwaltung der Interrupt Requests. Mit Hilfe eines spezifisch konfigurierten Timers pro Interrupt-Request-Kanal im IRQ-Controller kann jeder Interrupt nach Auftreten für eine bestimmte Zeit unterdrückt werden. Bild 3.3 zeigt das Blockschaltbild des hypothetischen IRQ-Controllers.

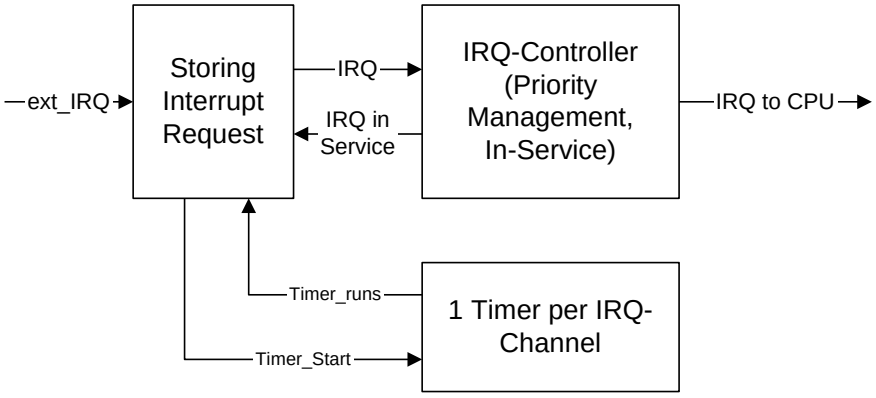


Bild 3.3 Modifizierter IRQ-Controller

Die vorgesehene Wirkungsweise des Timers ist diejenige, dass weitere IRQ-Signale, die vor dem Start der ISR auftreten, weder berücksichtigt noch gespeichert werden, während Signale, die nach dem Start der ISR, aber vor dem Ablauf des Timers eintreffen, gespeichert werden, jedoch vorerst keine Aktion hervorrufen. Diese etwas aufwendige Definition dient dazu, ein Maximum an Systemintegration zu erreichen.

Die Unterdrückung aller weiteren IRQ-Signale bis zum Eintritt in die ISR entspricht dabei der gängigen Praxis, mehrfache IRQs nur einmalig zu zählen. Die aktionslose Speicherung nach dem Eintritt lässt dabei keinen IRQ verloren gehen, und nach dem Timerablauf wird der gespeicherte IRQ aktiv (und startet den Timer sofort neu).

Diese Funktionsweise zwingt die asynchronen Interrupt Requests in ein Zeitschema, für das das Rechnersystem ausgelegt wird. Sind alle IRQs mit diesem Verfahren der Beschränkung der Wiederholungsfrequenz ausgestattet, können für alle Teile des Systems die maximalen Bearbeitungszeiten berechnet werden. Das modifizierte Ereignis-gesteuerte System wird hierdurch genauso deterministisch wie das Zeit-gesteuerte System mit dem Zusatz, dass keinerlei Pollingaktivitäten ablaufen müssen und ungenutzte Ereignisrechenzeiten den zeitunkritischen Programmteilen zugute kommen.

Für das Modified Event-Triggered System sind folgende Vor- und Nachteile anzugeben:

- Deterministische Berechenbarkeit des Zeitverhaltens, wie beim Time.-triggered System.
- Ungenutzte Zeit, die für Ereignisse vorgesehen war, wird an zeitunkritische Teile des Systems weitergegeben, es entsteht kein Overhead.
- Verfahren ist mit Einschränkung auch auf Netzwerke übertragbar, indem die einzelnen Knoten maximale Senderaten bekommen und eine unabhängige Hardware dies überwacht. Die Einschränkungen betreffen den Netzzugang, hier sind nur Collision-Avoidance-Verfahren (z.B. CAN) zulässig.
- Die Systemauslegung orientiert sich weiterhin an Worst-Case-Schätzungen.
- Alle IRQs zählen zu der Reihe der Ereignisse, die auf diese Weise zeitlich determinstisch behandelt werden müssen; Ereignisse mit beliebigen Reaktionszeiten oder ’weichen‘ Behandlungsgrenzen existieren nicht.
- Die variierte Hardware ist derzeit nicht erhältlich, muss also selbst definiert werden (z.B. in programmierbarer Hardware).

3.1.7 Modified Event-triggered Systems with Exception Handling

Während die Einschränkung der tatsächlichen IRQ-Raten den Determinismus in Event-triggered Systemen erzeugen kann, ist das Problem der maximalen Systemauslegung hierdurch noch nicht gelöst oder wesentlich gemildert. Die Einschränkung aus 3.1.6 schafft nur den Determinismus, der zuvor lediglich angenommen werden konnte.

Die Überdimensionierung eines Systems rührt von der erfahrungsgemäß großen Diskrepanz zwischen Worst-Case-Schätzung und realistischen Normalwerten. Natürlich lässt sich ein System nicht auf Erfahrungswerten so aufbauen, dass es zugleich auch beweisbar deterministisch ist.

Folgender Weg bietet unter bestimmten Umständen eine Möglichkeit, einen guten Kompromiss zwischen beweisbarer Echtzeitfähigkeit und Dimensionierung des Systems zu finden. Dieser Ansatz wird als ’*Modified Event-triggered System with Exception Handling*‘ bezeichnet.

Folgende Voraussetzungen sind notwendig, um einen Interrupt Request, der zu der deterministischen Ereignisreihe gehört, in eine zweite Kategorie, die mit *Ereignisreihe mit variierter Reaktionsmöglichkeiten* bezeichnet wird, zu transferieren:

- Grundsätzlich wird das System als Ereignis-gesteuert so ausgelegt wie in den vorangegangenen zwei Abschnitten beschrieben.
- Für das ausgewählte Ereignis muss eine Notreaktionsmöglichkeit existieren, beispielsweise ein allgemein gültiger, ungefährer Reaktionswert, der in einer gesonderten Reaktionsroutine eingesetzt werden kann oder
- Die Berechnungszeit für das ausgewählte Ereignis kann erweitert werden.

Mit Hilfe einer nochmalig erweiterten Hardwareunterstützung im Prozessor und im Interrupt Request Controller kann dann ein erweitertes IRQ-Handling eingeführt werden. Die ergänzende Hardware ist in Bild 3.4 dargestellt.

Die Ergänzung besteht darin, einen weiteren Timer pro Interrupt Request im IRQ-Controller vorzusehen. Dieser Timer wird mit jeder IRQ-Speicherung gestartet und enthält einen Ablaufwert, der der maximalen Reaktionszeit entspricht. Ist die Interrupt-Service-Routine beendet, so muss der Timer natürlich gestoppt werden, z.B. explizit durch zusätzliche Befehle oder implizit durch Hardwareerweiterung in der CPU (erweiterter RETI-Befehl, Return from Interrupt mit IRQ-Nummer).

Der Ablauf eines solchen Timers soll dann eine Time Exception (= Interrupt Request mit hoher Priorität) auslösen und damit eine Ausnahmebehandlung initiieren. Es ist hierbei möglich, alle derart ergänzten IRQs mit einer Time Exception zu versehen und damit in einer Routine zu behandeln.

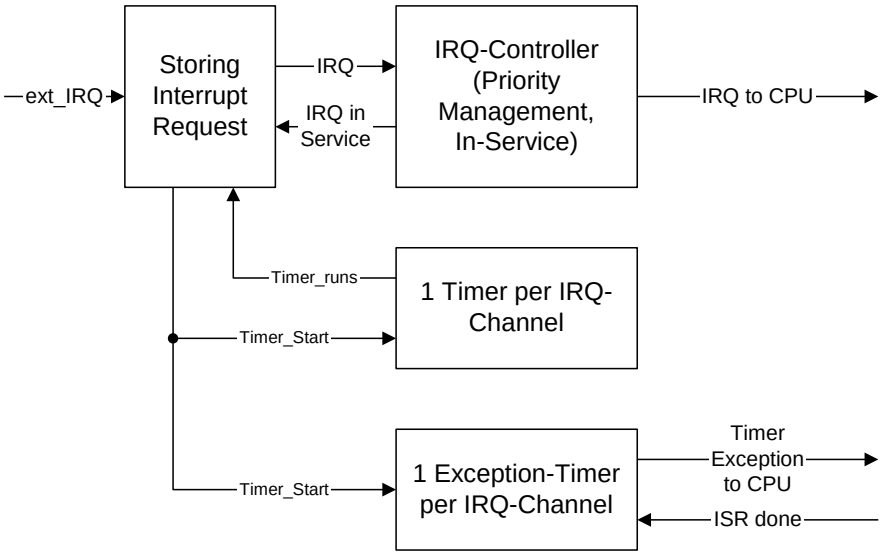


Bild 3.4 Erweiterter IRQ-Controller mit Time Exception

Die Ausnahmeroutine kann dann von fallweise entscheiden, wie vorzugehen ist. Existiert ein Notwert, der z.B. eine bereits berechnete, ungefähre Näherung (aber nicht den exakten Wert) darstellt, kann dieser eingesetzt und die Service-Routine damit für diesen Fall beendet werden. Es kann auch entschieden werden, einen weiteren Zeitabschnitt zu durchlaufen, falls dies für dieses Ereignis möglich ist.

Gerade die Möglichkeit, Näherungswerte einzusetzen, stellt ein mächtiges Instrumentarium dar, um harte Echtzeit bei 'weicher' Logik zu erhalten. Dies ist bei bisherigen Verfahren nur mit sehr großem Rechenaufwand möglich, Aufwand, der gerade aus Zeitknappheit entfallen muss.

Für diesen Ansatz zur Erreichung eines echtzeitfähigen Systems können folgende Vor- und Nachteile angegeben werden:

- Deterministische Berechenbarkeit des Zeitverhaltens, wie beim Time.-triggered und Modified Event-triggered System.
- Ungenutzte Zeit, die für Ereignisse vorgesehen war, wird an zeitunkritische Teile des Systems weitergegeben, es entsteht kein Overhead.
- Die Systemauslegung orientiert sich nicht mehr an Worst-Case-Schätzungen mit vollständigem Rechenweg, sondern für eine deterministische Auslegung nur noch bis zu den Näherungswerten.
- Komplexe Klassifizierung der Ereignisse notwendig: Welche Events sind immer vollständig durchzurechnen, welche können Näherungen haben, für welche sind Zeiterweiterungen (in Grenzen) zulässig?
- Softwareunterstützung ist derzeit nicht erhältlich, folglich ist alles Handdesign.
- Die erweiterte Hardware ist derzeit nicht erhältlich.

3.2 Bestimmung der charakteristischen Zeiten im System

Nachdem nunmehr die Grundzüge eines zeitbasierten Software-Engineerings aufgezeigt sind müssen die charakteristischen Zeiten im (späteren) System definiert bzw. bestimmt werden. Zu diesen Zeiten zählen die Zykluszeit, die aus dem Außenprozess heraus definiert sein muss, dann die Worst-Case-Execution-Time (WCET) und die Worst-Case-Interrupt-Disable-Time (WCIDT), die berechnet bzw. geschätzt werden müssen, nachdem die Routinen geschrieben sind, sowie der Nachweis der Echtzeitfähigkeit für das ganze System.

3.2.1 Zykluszeiten

In diesem Abschnitt wird von der Annahme ausgegangen, dass pro betrachtetem Thread exakt eine Zykluszeit auftritt. Dies bedeutet im Einzelnen:

- Threads ohne Zeitbindung, die dementsprechend zeitlich unkritisch sind, werden nicht betrachtet

- Ein Thread kann zwar mehrere Aufgaben beinhalten, die dann aber der gleichen Zykluszeit unterworfen sind.

Für die Bestimmung der Zykluszeit eines Threads sind ferner folgende Zeiten des Außenprozesses wichtig:

Definition 3.2:

Die *maximal erlaubte Reaktionszeit* T_{mx}**(soll)** ist die maximal mögliche Wartezeit vom Eintreten eines Ereignisses bis zur vollständigen Reaktion darauf (siehe auch Definition 2.8). Das Eintreten des Ereignisses ist hierbei der Zeitpunkt der Aktivierung des Eingangssignals am Mikroprozessor, das zur Unterbrechungsanforderung führt (in Ereignis-gesteuerten Systemen) bzw. das bei Auslesen zu einer Entscheidung im Programm führt, dass hier ein Ereignis behandelt werden muss (in Zeit-gesteuerten Systemen). Mit T_{mx}**(ist)** wird die tatsächlich erreichte maximale Reaktionszeit bezeichnet, die dementsprechend alle Latenzzeiten beinhaltet. T_{mx}**(netto)** stellt die tatsächlich benötigte, maximale Rechenzeit (WCET) für die Reaktion dar.

Definition 3.3:

Die *Wiederholungs- oder Folgezeit* T_F **(soll)** ist die Zeit einer Periode in einem zyklischen oder quasi-zyklischen Design. Dies ist sowohl im Fall des Zeit-gesteuerten Designs als auch im Fall des Ereignis-gesteuerten Designs die Minimalzeit, die zwischen zwei Ereignissen liegt.

Für das Ereignis-gesteuerte System wurde diese Minimalzeit als gegeben angenommen bzw. im erweiterten System (→3.1.6) erzwungen. Im Zeit-gesteuerten System wird sie durch das Systemdesign (Einstellung am Timer) indirekt definiert und muss natürlich mit den Erfordernissen am Prozess im Einklang stehen.

Definition 3.4:

Der *maximal akzeptable Jitter* T_{jt}***(soll)*** ist die maximal tolerable Varianz für die Abweichung vom Soll- bzw. Mittelwert innerhalb eines zyklischen oder quasi-zyklischen Designs. Zur Angabe des Jitters muss die Bezugsgröße mit angegeben werden. T_{jt}***(ist)*** bezeichnet den tatsächlich auftretenden maximalen, T_{jt}***(act)*** den aktuell auftretenden Jitter.

Der Jitter ist eine Zeit bzw. Zeitspanne, die nur im Ausnahmefall zur Beurteilung der Güte benötigt wird. So sind im Normalfall alle Reaktionen zeitlich korrekt, wenn sie im Zeitintervall $[0, T_{mx}]$ liegen. Bei Messwertaufnahmen hingegen möchte man häufig eine möglichst konstante Differenz der Aufnahmezeitpunkte, um die Messungen nicht künstlich zu verfälschen.

Definition 3.5:

Die *Testzeit* T_{Tst}***(period)*** ist die Zeit einer Periode zum Testen des Außenprozesses, ob ein bestimmtes Signal anliegt und der Prozess hierdurch signalisiert, dass eine

Bedienung notwendig ist. Demgegenüber ist $T_{Tst}(exec)$ die WCET der Ausführung dieses Tests.

Zwischen den definierten Zeiten existieren folgende Relationen:

$$T_{mx}(\text{soll}) \leq T_F \tag{3.3a}$$

bzw.

$$T_{mx}(\text{soll}) \leq \overline{T_F} \tag{3.3b}$$

Gl. (3.3a) muss nicht exakt erfüllt sein, sondern lediglich im *zeitlichen Mittelwert über eine fest definierte maximale Anzahl* von aufeinander folgenden Durchläufen (3.3b). In diesem Fall kann die Einzelaktion schneller wiederholt werden als die eigentliche Reaktion darauf erfolgen kann, so dass im System Puffer zur Zwischenspeicherung eingebaut sein müssen.

Ein Beispiel hierfür ist der Empfang von Paketen aus einem Netzwerk oder über eine Schnittstelle. Wenn in schneller Folge eine maximale Anzahl von Paketen über das Netz gesendet werden (so genannter Burst-Betrieb), so bleibt der Empfänger operabel, wenn er die Pakete zwischenspeichern und bis zum Empfang des nächsten Bursts bearbeiten kann.

Andererseits muss zwischen den Zeiten T_{mx}(netto), T_{Tst}(exec) und T_{Tst}(period) einerseits und T_F ebenfalls eine Korrelation bestehen, da die Summe der drei erstgenannten T_F nicht überschreiten darf (zumindest nicht im Mittel).

$$T_{Tst}(\text{period}) + T_{Tst}(\text{exec}) + T_{mx}(\text{netto}) \leq \overline{T_F} \tag{3.4}$$

Gl. (3.4) zeigt – bei Vernachlässigung der Zeit T_{Tst}(exec) –, dass sich T_{mx}(netto) und T_{Tst}(period) gegenseitig Konkurrenz um die zur Verfügung stehenden Zeit T_F machen:

$$T_{Tst}(period) \leq \begin{cases} T_F - T_{mx}(\text{netto}) \text{ für Zeit\_Steuerung} \\ 0 \text{ für Ereignis\_Steuerung} \end{cases} \tag{3.5}$$

Die Testperiode ist im Design „frei“ wählbar, Gl. (3.5) liefert eine obere Grenze des Wählbarkeitsbereiches. Als Periode zur Erkennung eines Ereignisses muss also ein Wert kleiner als T_F gewählt werden, zumeist sogar deutlich kleiner, eine gute Näherung könnte $T_F/2$ darstellen. Im Fall einer Ereignissteuerung ist die Periode automatisch 0, weil spontan reagiert wird, wobei allerdings noch die maximale Latenzzeit begrenzend wirkt (de facto ist die maximale Latenzzeit der schlechteste Wert für die Periode zur Erkennung eines Ereignisses). Im Einzelfall gilt dies auch für die Zeitsteuerung, wie z.B. im Fall einer Messwertaufnahme (hier liegt ausschließlich eine zeitliche Steuerung vor, die im Verhalten der Ereignissteuerung gleicht).

Die Diskrepanz zwischen Ereignis- und Zeitsteuerung kommt nun ausschließlich durch die zusätzliche Abfrage zustande, ob tatsächlich ein Ereignis vorliegt, und

dies in einer zeitlichen Abfolge, dass das Ereignis rechtzeitig erkannt wird. Dies bedeutet, dass ein Overhead in Zeit-gesteuerten Systemen entsteht, der die Abfrage enthält und dessen Auftrittsfrequenz im Allgemeinen mindestens $1/T_{Tst}(period)$ ist:

$$Overhead = \frac{1}{T_{Tst}(period)} \cdot WCET(\text{Timer\_ISR}) \qquad (3.6)$$

Gl. (3.6) liefert damit die untere Grenze der Testperiode: Eine schnellere Abfrage ist nicht möglich. Die eigentliche Belastung des Systems durch die Reaktionsberechnung ist in allen Systemen im Übrigen gleich und erfolgt mit der maximalen Frequenz $1/T_F$.

Anmerkung: Im Allgemeinen geht man davon aus, dass sich Zeit- wie Ereignisgesteuerte Systeme im Rechenzeitbedarf nicht unterscheiden, soweit es um den Nachweis der Echtzeitfähigkeit geht. Konkret haben die Zeit-gesteuerte Systeme haben einen zusätzlichen Overhead, der durch die Signalabfrage entsteht und möglichst minimal gehalten werden soll. Im Extremfall sehr häufiger und auch komplexer Abfragen (→ 4.4.1) kann der zusätzliche Bedarf sehr groß werden, so dass ggf. Sonderlösungen eingeführt werden müssen.

3.2.2 Umsetzung der charakteristischen Zeiten in ein Software-Design

In der Praxis geht man sicher von der Wiederholungszeit T_F eines Ereignisses aus. Dies ist die aus dem Prozess stammende charakteristische Zeit, z.B. die Zeit, die zwischen zwei Messungen liegen darf, um bestimmte Frequenzen noch aufnehmen zu können.

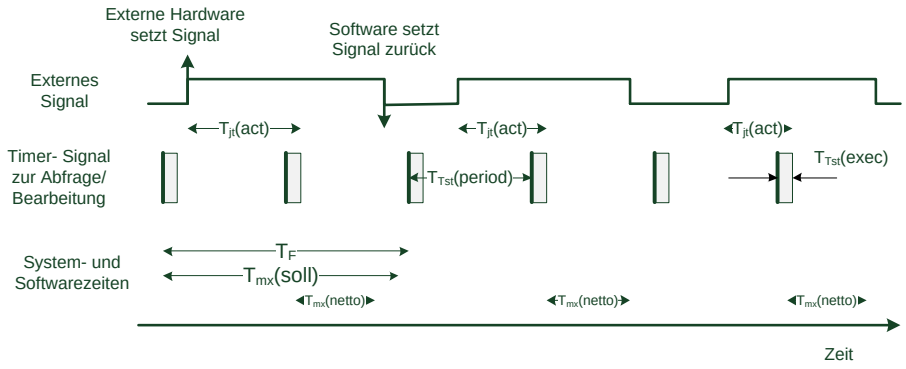


Bild 3.5a Zusammenhang zwischen charakteristischen Zeiten (bei einem Signal, Time-triggered Systemarchitektur)

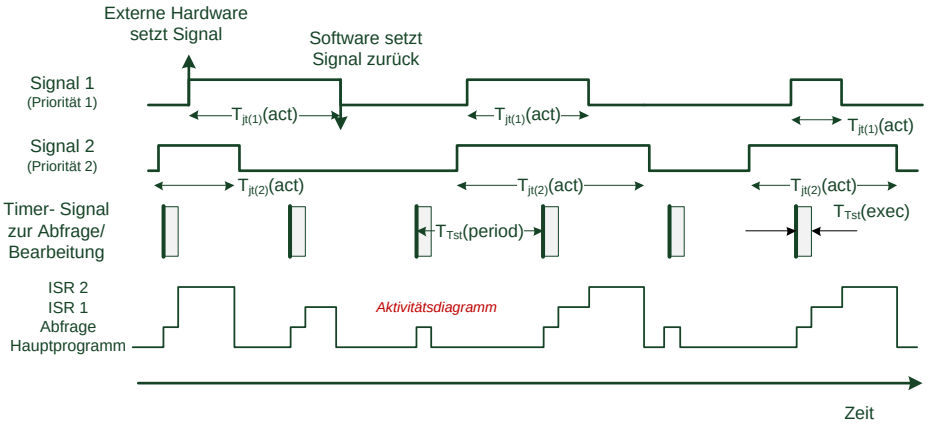


Bild 3.5b Zusammenhang zwischen charakteristischen Zeiten in einem zeitgesteuerten Systemdesign, bei zwei Signalen

Bild 3.5a und 3.5b zeigen beispielhaft die Zeiten und Aktivitäten in einem zeitgesteuerten System, jeweils für ein oder zwei abzufragende Signale. Hierbei lässt sich auch erkennen, wie und warum die tatsächlichen Reaktionszeiten variieren. Die eingezeichneten, aktuellen Jitter beziehen sich auf den Eintritt (Beginn) der jeweiligen Reaktionsroutine.

Aus der vom System geforderten maximalen Reaktionszeit $T_{mx}(soll)$ (mit $T_{mx}(soll) \leq T_F$) lässt sich für die im eingebetteten System zur Verfügung stehende effektive Reaktionszeit die Beziehung (siehe auch Bild 3.5a)

$$T_{mx,}(netto) \leq T_{mx}(soll) - T_{Tst}(period) \leq T_F - T_{Tst}(period) \quad (3.7a)$$

$$T_{Tst}(period) \leq T_F - T_{mx,}(netto) \quad (3.7b)$$

ableiten. Hat man also die maximal mögliche Latenz- und Rechenzeit für eine Reaktion im System bestimmt und ist die tatsächlich benötigte Rechenzeit bekannt, lässt sich hieraus die maximal mögliche Testzeit ableiten. Gl. (3.7) berücksichtigt jedoch nicht negative Einflüsse von anderen Service-Routinen wie in Bild 3.5b. Gl. (3.7b) zeigt die obere Schranke für die Wahl der Testzeit an: $T_{Tst}(period) \leq T_F - T_{mx,}(netto)$.

Zwischen dem Jitter $T_{jt}(ist)$ und der Testzeit $T_{Tst}(period)$ besteht des Weiteren eine Konkurrenzbeziehung. Ist der Jitter „beliebig“, d.h. es liegt keine spezifische Einschränkung der Varianz vor, wird die Testzeit auf den maximal möglichen Wert $T_F - T_{mx,}(netto)$ gesetzt, um den Overhead klein zu halten.

Zwei Ursachen können diesen Maximalwert beschränken:

- Um den Jitter klein zu halten muss die Testzeit verkleinert werden, da automatisch $T_{jt}(ist) \geq T_{Tst}$ gilt.

- Wie in (3.7a) bereits ausgedrückt beschränkt die Testzeit auch die maximal zur Verfügung stehende Reaktionszeit bzw. Rechenzeit im eingebetteten System, da die maximal mögliche Reaktionszeit ja erhalten bleibt.

Ausgehend von der Wiederholungszeit T_F der Ereignisse, den zugehörigen größtmöglichen Varianzen $T_{jt}(soll)$ und der benötigten Rechenzeit $T_{mx,}(netto)$ muss also die Software-dominierte Zeit T_{Tst} angepasst bzw. ausgewählt werden, ggf. auch durch Performanceerhöhung (für $T_{mx,}(netto)$) im Mikroprozessor. Bezüglich der Verkürzung der Testzeit sei allerdings auch auf den Abschnitt 3.1.3 verwiesen: Die dort eingeführte Zeit für die Aktionsperiode, die als ggT der Einzelzeiten definiert wurde, führte ggf. zu einem großen Overhead, ein Effekt, der auch durch Verkürzung der Testzeit T_{Tst} erzeugt werden kann.

3.2.3 Worst-Case-Execution-Time und Worst-Case-Interrupt-Disable-Time

Der nächste Schritt besteht in der Bestimmung der maximalen Reaktionszeiten. Hierzu müssen im ersten Schritt die Worst-Case-Execution-Times (WCET) und im zweiten Schritt dann die Kombination aller Routinen und Reaktionen im System bestimmt werden.

Worst-Case-Execution-Times (WCET)

Die WCET ist diejenige Zeit, die ein Programm oder Programmteil bei ununterbrochener Ausführung maximal benötigt. Diese Zeit kann in Sekunden, aber auch in Prozessortakten angegeben werden. Letztere Angabe ist meist sinnvoll, da der reale Prozessortakt zwischen der Anzahl der Takte und der damit vergangenen Zeit koppelt.

Die Bestimmung der WCET bedeutet, dass die Summe über die Befehle, multipliziert mit der maximalen Ausführungszeit, gebildet werden muss. Zudem muss auch der längste Weg hierfür ausgewählt werden, und dies ist manuell nur für kleine Programme möglich.

Insbesondere muss die Auswertung auf Assembler- bzw. Maschinenbefehlsebene erfolgen, denn in der Hochsprachencodierung kann man die wirkliche Anzahl der Befehle nur sehr schwer erkennen. Allerdings wäre eine Auswertung auf dieser Ebene wesentlich bequemer. Zu dieser Problematik wird noch im Abschnitt 3.3 Stellung genommen.

Weiterhin existieren mehrere Gründe, warum die WCET nicht oder nur sehr unpräzise bestimmt bzw. geschätzt werden kann. Generell gilt es als akzeptabel, wenn die WCET-Werte nicht bestimmt, sondern geschätzt werden, solange die Schätzung sicher ist, d.h. es gilt garantiert WCET(geschätzt) ≥ WCET(real). Die Genauigkeit der geschätzten WCET-Werte gilt bereits als gut, wenn der Fehler < 100% ist, die Schätzung also weniger als den Faktor 2 größer ist als der reale Wert.

```
#include <stdlib.h>
int k, iEnde, iCounter;
int array[1024];

srand();   /* Initialisierung der Pseudo-Zufallszahlen */
iCounter = 0;
iEnde = rand(); /* iEnde besitzt jetzt einen positiven Integerwert */
for( k = 0; k < iEnde; k++ )
{
   iCounter = iCounter + 1;
   if( array[k] == 0 )
      k = 0;
}
```

Bild 3.6 Beispielcode für Schleife, deren Laufzeit nicht zur Übersetzungszeit bestimmt werden kann

```
int k, iCounter;
int array[1024];

iCounter = 0;
for( k = 0; k < 1024; k++ )
{
   iCounter = iCounter + 1;
   if( array[k] == 0 )
      k = 0;
}
```

Bild 3.7 Beispielcode für unendliche Schleife

Die Schätzung kann aus folgenden Gründen unmöglich bzw. unrealistisch sein:

- Wird im System ein Mikroprozessor/Mikrocontroller mit einem Cache-Speicher eingesetzt, dann ist die Verteilung der Instruktionen und Daten zwischen Cache (schnell) und Hauptspeicher (langsam) unbestimmt. Umfangreiche Studien haben gezeigt, dass es unmöglich ist, einen Mindestbeschleunigungsfaktor zu bestimmen, also bei den aktuellen Cache-Architekturen einen garantierten

Beschleunigungswert zu erhalten. Ursache hierfür sind die Ersetzungsstrategien, die einen starken Zufallscharakter haben.

Abhilfe kann hier nur geschaffen werden, wenn Cache mit anderen, Nicht-Standard-Strategien verwendet wird, oder auf Cache zugunsten eines Scratch-Pad-Memory (schneller Zwischenspeicher, in dem ganze Teilprogramme vom eigentlichen Programm her gesteuert gespeichert und ausgeführt werden) verzichtet wird.

Ohne diese Abhilfe kann die WCET nur bestimmt werden, indem der Cache als nicht existent angenommen wird. Dies führt dann zu unrealistisch hohen Werten.

```
int factorial( int iNum )
{
   switch( iNum )
   {
      case 0:
         return( 1 );
         break;

      case 1:
         return( 1 );
         break;

      default:
         return( iNum * factorial( iNum - 1 ));
         break;
   }
}
```

Bild 3.8 Beispielcode für rekursive Programmierung (hier: Fakultätsberechnung)

- Bestimmte Programmkonstrukte sind nicht zur Bestimmung einer WCET geeignet. Hierzu zählen rekursive Programmierung sowie Schleifen, deren Anzahl der Durchläufe nicht zur Übersetzungszeit bestimmbar ist.

Worst-Case-Interrupt-Disable-Times (WCIDT)

Der Jitter T_{jt} in einem System bestimmt sozusagen die Genauigkeit der Aktion. Als eine der möglichen Quellen war bereits die Testzeit TTst bestimmt worden, die in einem Zeit-gesteuerten System die Erkennung eines Signals bestimmt.

Die andere, immer vorhandene Quelle für einen Jitter sind unterbrechungslose Zeiten im Programm. Unabhängig davon, ob ein Ereignis-Interrupt oder ein Timer-Interrupt am Prozessor zu behandeln ist, handelt es sich immer um eine Unterbre-

chung des bisherigen Programmlaufs. Eine solche Unterbrechung ist nicht immer möglich, denn fast immer existieren Programmteile, deren Unterbrechung eine schwere Störung darstellen würde und die ggf. sogar zum Absturz des gesamten Systems führen könnte. Solche Programmabschnitte werden als *atomar* bezeichnet, ein Beispiel hierfür ist in Abschnitt 3.3 gegeben.

Im Softwaredesign wirkt man dieser Gefahr dadurch entgegen, dass gefährdete Programmabschnitte durch die Unterbindung der Interrupts geschützt werden. Die Identifizierung solcher Programmabschnitte ist dabei keineswegs trivial, und hier können nur allgemeine Leitlinien gegeben werden:

- Kandidaten sind zunächst alle Interrupt Service Routinen und alle Funktionen, die im Rahmen einer ISR aufgerufen werden. Häufig wird hier die Strategie verfolgt, dass generell ein (weiterer) Interrupt verboten ist und nur im Ausnahmefall zugelassen wird.
- Weitere Kandidaten sind Abschnitte in Programmen, bei denen auf mehrere globale Variablen, die semantisch zusammenhängend sind, (lesend oder schreibend) zugegriffen wird. Falls diese Variablen durch Interrupt Service Routinen ebenfalls genutzt werden, müssen sie immer gemeinsam geändert werden, weil ansonsten eine nur teilweise geänderte Menge als gültigen Variablensatz gewertet wird.

  Beispiel hierfür ist die Speicherung des aktuellen Zeitpunkts mit Datum und Uhrzeit bis hin zu Millisekunden. Dies wird häufig in mehreren Variablen gespeichert und z.B. durch einen Timer-Interrupt mit zugehöriger ISR ständig wietergesetzt. Wenn nun eine externe Routine diese Zeit benötigt, dann könnte sie beim Stand von 5,999 Sekunden vielleicht 6,999 Sekunden lesen, weil zuerst der Millisekundenstand (999) und dann der Sekundenstand (erst 5, beim Lesen durch Umspringen aber 6) gelesen würde.

Die Worst-Case-Interrupt-Disable-Time (WCIDT) ist nun diejenige Zeit, die die maximale Zeit (auch gemessen in Prozessortakten) darstellt, in der das Programm nicht unterbrechbar ist. Hierzu müssen alle Abschnitte im normalen Programm, die nicht unterbrochen werden dürfen, sowie alle Interrupt Service Routinen, falls diese auch nicht unterbrechbar sind, betrachtet werden.

3.2.4 Nachweis der Echtzeitfähigkeit

Nunmehr besteht zumindest grundsätzlich die Möglichkeit zur Bestimmung der Echtzeitfähigkeit. Das System möge dabei k verschiedene Signale in echtzeitfähiger Weise bearbeiten. Folgende Voraussetzungen seien hierfür angenommen:

- keine ISR kann unterbrochen werden,
- jede ISR behandelt den IRQ vollständig, d.h. die Reaktion ist vollständig hierin beschrieben,
- für jede ISR ist eine eigene Priorität (0 … k–1) gegeben (0 bedeutet dabei die höchste Priorität),

- für jeden IRQ ist eine maximale Frequenz des Auftretens und eine maximale Reaktionszeit gegeben und
- das Hauptprogramm ist jederzeit unterbrechbar.

Folgende Abkürzungen seien ferner für die Bearbeitungs- und Wartezeiten gewählt

Für IRQ(i) sei TF(i) die minimale IRQ-Folge-oder Wiederholungszeit und TMX(i) die maximal zulässige Antwortzeit, TA(i) die Bearbeitungszeit für die i-te Service Routine, alle Zeiten ausgedrückt in Prozessortakten. SP sei diejenige Zeit, die sich als KGV (kleinstes gemeinsames Vielfaches) aller minimalen Folgezeiten TF(i) ergibt, die so genannte *Superperiode*. Ferner sei num(i) = SP/TF(i) die maximale Anzahl der Auftritte pro Zeitintervall SP. Jetzt müssen die Ungleichungen

$$\sum_{i=0}^{k-1} num(i) \cdot TA(i) \leq SP \tag{3.8}$$

$$\forall n \in \{0,\ldots,k-1\} : \sum_{i=0}^{n-1} \left(\left\lceil \frac{num(i)}{num(n)} \right\rceil \cdot TA(i) \right) + \max_{\substack{j=n+1 \\ \ldots k-1}} (TA(j)) + TA(n) \leq Tmx(n) \tag{3.9}$$

gelten. (3.8) bedeutet dabei, dass die Summe aller im Zeitintervall der Superperiode SP auftretenden IRQ-Bearbeitungszeiten dieses Intervall nicht überschreiten darf – eine vergleichsweise einfach zu realisierende bzw. nachzuweisende Forderung, die aber nur notwendig (und nicht hinreichend) ist.

(3.9) bedeutet hingegen, dass für alle IRQ-Ebenen (und Prioritäten) die Einhaltung der maximal möglichen Antwortzeit gewährleistet sein muss. Hierzu muss angenommen werden, dass ein niedriger priorisierter IRQ kurz zuvor auftrat und bearbeitet wird, und dass alle höheren IRQs ebenfalls auftreten und bearbeitet werden.

Der mittlere Term bedeutet, dass alle im Intervall zwischen zwei IRQs der Priorität n auftretenden IRQs höherer Priorität mit berücksichtigt werden müssen. Hierzu ist das Verhältnis zwischen Anzahl num(i) und num(n) entscheidend, und zwar der nächst höhere, ganzzahlige Wert, denn dieser gibt die Anzahl der möglichen IRQs im Zeitabschnitt an.

Gl. (3.9) stellt ein Ungleichungssystem dar, das auch so aufgefasst werden kann, dass man eine Lösung für gegebene Threads oder Tasks sucht, um dieses System zu erfüllen. Im Gegensatz zu üblichen Gleichungs- oder Ungleichungssystemen, in denen Variable enthalten, für die dann Werte zu ermitteln sind, muss hier jedoch bei gegebenen Werten ein Prioritätssystem gewonnen werden, so dass alle Ungleichungen erfüllt sind.

Ein Beispiel möge dies erläutern. Bild 3.9 zeigt ein System mit 4 streng periodisch auftretenden Threads. Hierbei kommt es für Thread 2 zum Konflikt, weil die Reaktion auf den Event nicht rechtzeitig fertig wird.

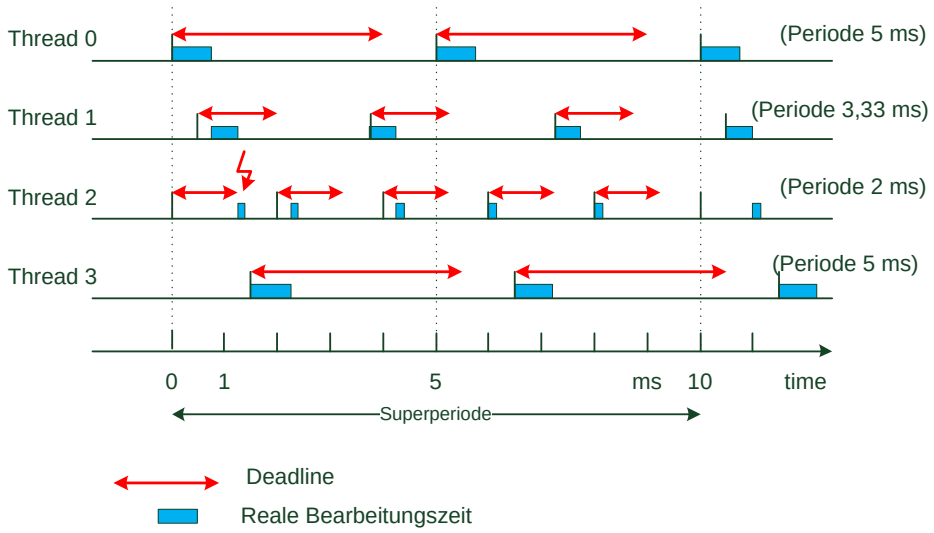


Bild 3.9 Bearbeitung mehrerer Events mit Konflikt

Dieser Konflikt wird auch im Ungleichungssystem (3.9) deutlich. Für die einzelnen Threads gelten zunächst folgende Annahmen:

Thread 0: Periode 5 ms, Bearbeitungszeit 0,75 ms, Deadline 4 ms

Thread 1: Periode 3,33 ms, Bearbeitungszeit 0,5 ms, Deadline 2 ms

Thread 2: Periode 2 ms, Bearbeitungszeit 0,1 ms, Deadline 1,25 ms

Thread 3: Periode 5 ms, Bearbeitungszeit 0,75 ms, Deadline 4 ms

Die Superperiode ist 10 ms, Gl. (3.8) ist über die Summe

$2 * 0{,}75 + 3 * 0{,}5 + 5 * 0{,}1 + 2 * 0{,}75 = 5 \leq 10$

erfüllt.

Bezüglich des Ungleichungssystems (3,9) ergeben sich folgende Ungleichungen:

Thread 0: $0{,}75 + 0{,}75 \leq 4$ (ok)

Thread 1: $1*0{,}75 + 0{,}75 + 0{,}5 \leq 2$ (ok)

Thread 2: (1*0,75 + 1*0,5) + 0,75 + 0,1 ≥ 1,25 (nok)

Thread 3: $(1*0{,}75 + 2*0{,}5 + 3*0{,}1) + 0{,}75 \leq 4$ (ok)

Allerdings lässt sich das in diesem Fall durch eine einfache Umstellung von Thread 2 und 0 beheben. Dann gelten die Ungleichungen:

Thread 0 (neu): $0{,}75 + 0{,}1 \leq 1{,}25$ (ok)

Thread 1: $2*0,1 + 0,75 + 0,5 \leq 2$ (ok)

Thread 2 (neu): $(3*0,1 + 2*0,5) + 0,75 + 0,75 \leq 4$ (ok)

Thread 3: $(3*0,1 + 2*0,5 + 1*0,75) + 0,75 \leq 4$ (ok)

Gl. (3.9) ist außerordentlich schwierig (aufwendig) im Nachweis, oder sie bedeutet, dass das System planmäßig überdimensioniert werden muss. Das Erfüllen dieses Systems von Ungleichungen bedeutet, dass die Echtzeitfähigkeit erreicht ist, das Scheitern jedoch nicht, dass Echtzeitfähigkeit unmöglich ist (nicht einmal, dass es noch nicht erreicht ist), denn (3.9) ist eine Schätzformel! Insgesamt sind folgende Vor- und Nachteile für diese Form der Systemauslegung aufzuzählen:

- Alle zeitkritischen Routinen sind entsprechen priorisiert angelegt, das System kann optimal angepasst werden.
- Der Nachweis harter Echtzeitfähigkeit ist möglich
- Der Nachweis ist sehr aufwendig
- Ein echtes Scheduling im Sinn einer dynamischen Verteilung kann so nicht erreicht werden (Anmerkung: Die IRQs sind dynamisch in ihrer Reihenfolge, alle Aktionen im Hauptprogramm oder gar eine Auswahl anhand von Prioritäten ist nicht vorgesehen).

Der letzte Punkt wird nochmals im Kapitel 4 aufgegriffen, wo ein Design Pattern (Entwurfsmuster) für einen variierten Ansatz diskutiert wird. In diesem Design Pattern kann dann ein Applikations-spezifisches Scheduling durchgeführt werden.

3.3 Kommunikation zwischen Systemteilen

Im bisherigen Systementwurf trat noch keine Kommunikation zwischen (heterogenen) Systemteilen auf, obwohl dies vermutlich schon notwendig ist. Kommunikation wird zwar zumeist mit Netzwerk verbunden, aber innerhalb eines Systems kommunizieren Systemteile (= Threads, Tasks, Prozesse) ebenfalls miteinander.

Datenkommunikation heißt allgemein Austausch von Daten zwischen zwei elektronischen Systemen, wobei zwei unabhängige (Teil-)Programme ebenfalls als je ein elektronisches System gezählt werden sollen. Bezüglich einer Klassifizierung der Kommunikation unterscheidet man in der Art, wie dem Kommunikationspartner die Daten zugänglich gemacht werden:

- Kommunikation per *Shared Memory*
- Kommunikation per Nachrichten (*Message Passing*)

Ferner wird bezüglich des zeitlichen Verhaltens der Kommunikation klassifiziert. Hierbei steht im Vordergrund, wie (und ob) sich die Kommunikationspartner synchronisieren oder nicht (siehe hierzu auch 2.2.3 bez. der grundlegenden Modelle für die Nebenläufigkeit):

- *Nicht-blockierende* Kommunikation

- Synchronisierende, *blockierende* Kommunikation

Hier sei auf die Unterschiede zu 2.2.3 verwiesen. Dort war von (perfekt) synchroner Kommunikation gesprochen worden, dieser Begriff bedeutet, dass die Kommunikationspartner in Null- bzw. konstanter Zeit miteinander kommuniziert haben. Die blockierende Kommunikation hier wartet auf eine Bestätigung oder ein Ergebnis, und von einer Zeitschranke hierfür ist keine Rede.

3.3.1 Kommunikation per Shared Memory versus Message Passing

Kommunikation mittels *Shared Memory* bedeutet, dass die Kommunikationspartner einen gemeinsamen Speicherbereich und mithin einen gemeinsamen Adressraum besitzen. Dies ist im Allgemeinen nur in lokalen Systemen möglich, wo also der Rechner aus einem oder mehreren Prozessoren, aber eben mit mindestens einem gemeinsamen Speicher besteht.

Die Kommunikation besteht darin, dass ein Teilnehmer in den Datenbereich des anderen hineinschreibt und auf eine vereinbarte Weise das Vorhandensein neuer Daten signalisiert. Vor- und Nachteile dieses Verfahrens:

- Einfachheit der Kommunikation
- Die Performance dieser Kommunikation ist optimal, insbesondere große Datenmengen können so in kurzer Zeit übertragen werden
- Das Schreiben in den Adressraum einer anderen Applikation ist gefährlich, weil auch Bereiche unerlaubt überschrieben werden können
- Der Einsatz ist beschränkt auf lokale Systeme, und dabei innerhalb eines Prozesses (falls das Betriebssystem Multiprocessing unterstützt), weil 2 verschiedene Prozesse per definitionem unterschiedlichen Adressräume haben

Kommunikation per *Message Passing* bedeutet, dass man einen ausgezeichneten Kommunikationskanal hat. Dieser Kommunikationskanal kann vom Betriebssystem geschaffen, es kann sich um ein Netzwerk handeln usw. Vor- und Nachteile sind:

- Kommunikation auf „beliebige“ Entfernung
- Gesichertes Modell (gegenüber Fehlhandlungen)
- Schwierigkeiten bei hohen Performance-Anforderungen und bei geforderter sicherer und/oder Echtzeitübertragung
- Komplexere Entwurfsmethodik

3.3.2 Blockierende und nicht-blockierende Kommunikation

Abgesehen von Entfernungs- und Performancefragen, die im Abschnitt 3.3.1 mitdiskutiert wurden, existieren weitere Anforderungen zur Kommunikation, die teilweise in Konkurrenz stehen:

- Sicherstellung der Kommunikation (mit Benachrichtigung bei Fehlschlag oder Fehlern)
- Beeinflussung des eigenen Laufzeitverhaltens, etwa durch Warten
- Garantie einer maximalen Laufzeit (mit Benachrichtigung bei Überschreiten)

Die Garantie einer maximalen Laufzeit wird in erster Linie auf das Netzwerk selbst abgebildet, da im umgekehrten Fall ein nicht-echtzeitfähiges Netzwerk nicht ausgleichbar ist. Die Eigenschaft einer echtzeitfähigen Kommunikation ist in Netzwerken aber nur mit vergleichsweise hohem Aufwand realisierbar:

- Als wichtigste Voraussetzung gilt, dass die Ressource Netzwerk (pro Zeiteinheit ist nur eine Übertragung möglich) deterministisch zugeteilt wird. Als besonders geeignetes Verfahren erweist sich hier die zeitlich gesteuerte Zuteilung (time-triggered), wo alle Netzteilnehmer eine einheitliche Zeit führen und Zeittabellen besitzen, wann sie selbst und wann die anderen senden (dürfen/müssen) (siehe auch 4.4.2 und 4.4.3)
- Die Steuerung per Zeittabellen bei gleichzeitigem ständigen Zeitabgleich ermöglicht sogar eine Ausfallerkennung, allerdings ist der Durchsatz hier suboptimal.

Aus Sicht der Applikation ist die Einführung eines echtzeitfähigen Netzwerks notwendige, aber nicht hinreichende Voraussetzung für die Echtzeitfähigkeit der Gesamtapplikation. Außer der eigenen Echtzeitfähigkeit aller Teilkomponenten muss auch die Kommunikation in der Applikation nicht-blockierend ausgelegt sein, wie an folgendem (lokalen) Beispiel ersichtlich ist:

Bild 3.10 zeigt zu diesem Zweck eine nicht-blockierende Kommunikation zwischen einem *Producer* (hier: Interrupt-Service-Routine, die Messwerte aus einem AD-Umsetzer ausliest und speichert) und einem *Consumer* (Teil der Hauptapplikation). Der Producer wartet nicht darauf, ob ein Wert schon gelesen und verarbeitet wurde (Kennzeichen: semaMess == 0), sondern schreibt den neuen Wert in die vorgesehene Hauptspeicherstelle, auch auf die Gefahr eines Werteverlusts.

Auch der Consumer wartet nicht auf einen neuen Wert, sondern beginnt seine Auswertung nur dann, wenn ein neuer Wert vorliegt, ansonsten könnte etwas anderes durchgeführt werden. Diese Form der nicht-blockierenden Kommunikation ist wichtig für Echtzeitsysteme, denn die Blockierung könnte unabsehbare Auswirkungen auf das Timing haben.

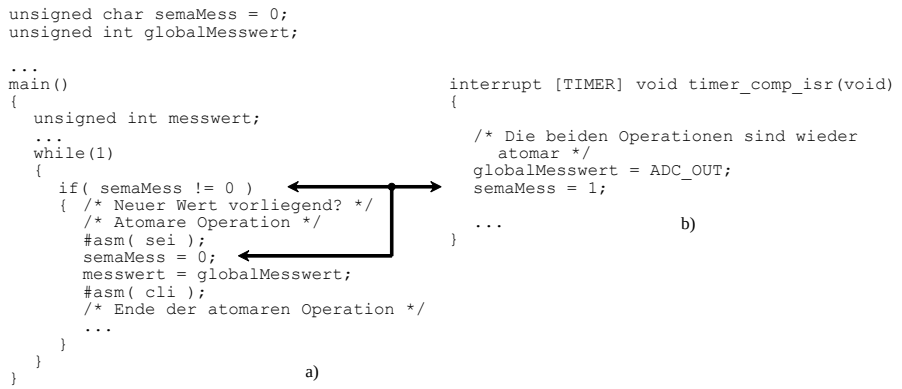


Bild 3.10 Nicht-blockierende Kommunikation zwischen Main- (a) und Interrupt-Routine (b)

Anmerkung: Das Schreiben einer nicht-blockierenden Kommunikation ist oftmals nicht trivial, weil die Aktion, die auf der Kommunikation beruht, ja grundsätzlich ausgeführt werden soll. Hierzu sei ein Beispiel gegeben:

Angenommen, ein Satz von 16 Daten soll in das EEPROM eines Mikrocontrollers geschrieben werden, weil die Werte ein Abschalten der Spannung überdauern sollen. Das Schreiben in das EEPROM wird von modernen C-Compilern mithilfe von Laufzeitbibliotheken unterstützt, aus Sicht der Anwendungsprogrammierung ist dies eine einfache Sache:

```
eeprom int eepIBasicConfig[16];
int k, iConfig[16];
…
for( k = 0; k < 16; k++ )
eepIBasicConfig[k] = iConfig[k];
```

Bild 3.11 Blockierendes Schreiben in Variablen im EEPROM

Das Gefährliche an dieser Codesequenz ist, dass hinter der scheinbar einfachen Wertzuweisung an eepIBasicConfig[] eine ganze Laufzeitroutine mit blockierender Kommunikation steckt, was so nicht ersichtlich ist. Meistens wird der erste Wert sofort geschrieben (weil in der Hardware ein Pufferplatz vorhanden ist, ab dem zweiten Wert wartet man dann auf die Fertigstellung des Schreibens des Vorgängerwerts – was durchaus einige Millisekunden dauern kann.

Die Kunst der nicht-blockierenden Kommunikation will es nun, dass man nur hineinschreibt, wenn dies sofort möglich ist (weil der Puffer frei ist), ansonsten sich merkt, dass noch etwas zu schreiben ist, und zum weiteren Programm zurückkehrt. Hierbei muss dann gewährleistet sein, dass das Schreiben etwa durch

zyklisches Design irgendwann fertig gestellt wird, und dass es keine Seiteneffekte gibt.

Eine Lösung unter Nutzung des Design Pattern aus Abschnitt 4.2 ist in 4.2.3 dargestellt.

Die Routine in Bild 3.10 beinhaltet noch einen atomaren Teil (→ 3.2.3): Durch die Unterbindung aller Interrupt Requests in der Hauptroutine (zwischen #asm(sei) und #asm(cli), Befehle, die das Interrupt Disable Flag löschen bzw. setzen) werden die darin enthalten C-Anweisungen ununterbrechbar und somit immer zusammenhängend durchgeführt.

4 Design-Pattern für Echtzeitsysteme, basierend auf Mikrocontroller

4.1 Quasi-statischer Ansatz zum Multitasking

Die in Kapitel 3 diskutierten Verfahren beruhen auf der Idee, die zeitkritischen Teile in eine Unterbrechungsroutine einzufügen und den Rest der Zeit die relativ zeitunkritischen Teilaufgaben zu rechnen. Es fehlt jedoch noch die Zusammenfassung dieser Teile in einem Programm bzw. ein Design-Pattern für das komplette Systemdesign.

Das hier vorgestellte Designverfahren beruht auf drei Schritten:

- Klassifizierung der Teilaufgaben
- Implementierung der Einzelteile
- Zusammenfassung zum Gesamtprogramm

Ein Hauptaugenmerk muss dabei auf die Kommunikation zwischen den Threads ($\rightarrow$ 3.3) gelegt werden.

4.1.1 Klassifizierung der Teilaufgaben

Das hier dargestellte Designverfahren beruht darauf, die einzelnen Teilaufgaben (diese werden hier immer als Thread bezeichnet) zu klassifizieren, ihren gewünschten Eigenschaften nach zu implementieren und das System dann zu integrieren. Die folgende Klassifizierung ist notwendig, da insbesondere im Zeitbereich verschiedene Randbedingungen für die einzelnen Klassen angenommen werden müssen.

- *Streng zyklisch ablaufende Threads*: Fester Bestandteil dieser Teilaufgaben sind exakte Zeitabstände, in denen diese Threads zumindest gestartet werden und generell auch komplett ablaufen müssen, um der Spezifikation zu genügen. Beispiele hierfür sind Messwertaufnahmen oder die Bedienung von asynchronen Schnittstellen zur Datenkommunikation.
- *Ereignis-gesteuerte Threads*: Das Starten bzw. Wecken einer Threadmit dieser Charakterisierung ist an ein externes Ereignis gebunden, meist in Form eines Interrupt-Requests. Der Startzeitpunkt ist somit nicht zur Compilezeit bestimmbar, so dass diese Threads störend auf den zeitlichen Gesamtablauf wirken können. Typische Vertreter dieser Klasse sind der Empfang von Nachrichten via Netzwerk bzw. die Reaktion darauf oder Schalter in der Applikation, die besondere Zustände signalisieren (etwa ”Not-Aus“).
- *Generelle Tasks mit Zeitbindung*: Die dritte Klasse beschreibt alle Threads in dem System, die zwar keine scharfen Zeitbedingungen enthalten, im Ganzen

jedoch Zeitschranken einhalten müssen. Hiermit sind Threads beschrieben, die beispielsweise Auswertungen von Messwerten vornehmen. Während die einzelne Auswertung ausnahmsweise über einen Messwertzyklus hinaus dauern darf, muss insgesamt die mittlere Auswertezeit eingehalten werden.

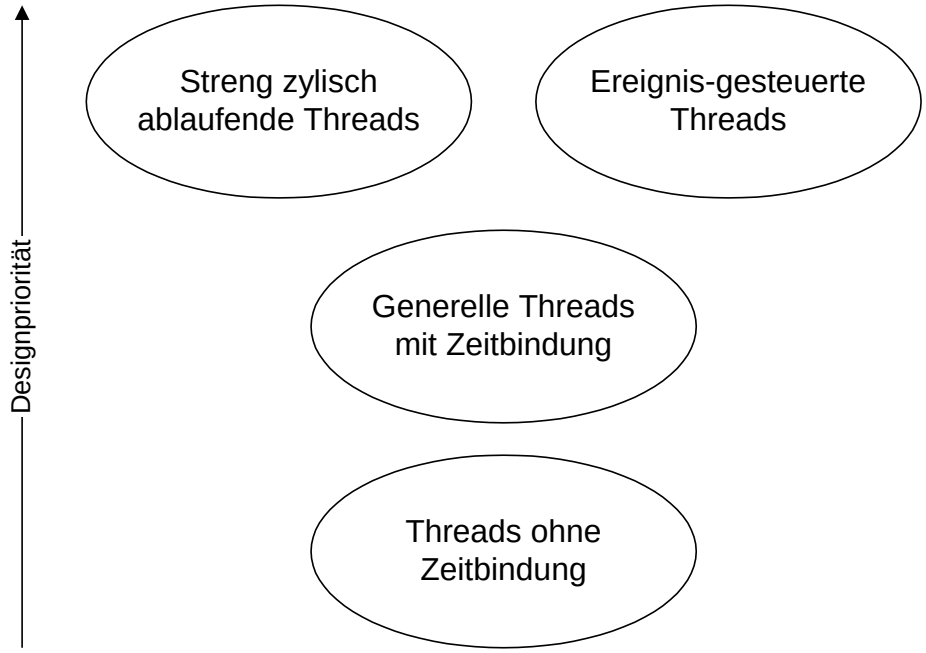


Bild 4.1 Threadklassen und Designprioritäten

Diese drei Grundklassen zeitabhängiger Teilaufgaben stellen das Grundgerüst zum Systemdesign dar. Die erste Aufgabe des Systemdesigners besteht darin, alle in der Beschreibung vorkommenden Aufgaben in dieses Grundgerüst einzuteilen, mit allen dabei auftretenden Schwierigkeiten.

Generell gilt, dass eine Teilaufgabe in eine ”höhere Klasse” integriert werden kann. So kann ein Thread, die überhaupt keine Zeitbindung besitzt – dies dürfte in der Praxis nicht häufig vorkommen, ein Kontrollthread wäre aber hier möglich – natürlich in die Klasse der generellen Threads mit Zeitbindung sortiert werden. Diese Threadklasse ist in Bild 4.1 dargestellt, wurde jedoch nicht in die Klassifizierung aufgenommen, da sie irrelevant für das hier dargestellte Designprinzip ist.

Streng zyklisch ablaufende Threads und Ereignis-gesteuerte Threads sind in ihrer Designpriorität in etwa gleichzusetzen (→ Bild 4.1). In der Praxis kann die Implementierung auch sehr ähnlich sein, indem die zyklischen Threads in Interrupt-Service-Routinen (ISR) mit Timersteuerung und die Ereignis-gesteuerten Threads

in anderen ISRs behandelt werden. Die Unterscheidung soll dennoch aufrecht erhalten bleiben, da zwischen beiden Implementierungen ein fundamentaler Unterschied existiert.

4.1.2 Lösungsansätze für die verschiedenen Aufgabenklassen

Im nächsten Schritt des Designverfahren werden die Mitglieder der einzelnen Klassen zunächst getrennt voneinander implementiert und die maximale Ausführungszeit jeweils berechnet. In erster Näherung werden dafür die WCET der einzelnen Teilaufgaben als voneinander unabhängig angenommen. Um dies wirklich zu erreichen, muss auf ein blockierendes Warten bei Kommunikation zwischen den Threads unbedingt verzichtet werden, denn dies kann zu großen Problemen bei der Bestimmung der WCET bis hin zur Unmöglichkeit führen (→ 3.3.2). Diese Forderung führt zu einem sicheren Design, da sich Abhängigkeiten etwa in der Form, dass, falls Thread 1 den maximalen Pfad durchläuft, Thread 2 garantiert einen kleineren Pfad als seinen maximalen wählt, nur positiv auf die WCET des Gesamtsystems auswirken können.

Bild 4.2 zeigt den gesamten Designprozess (ohne Entscheidungen bzw. Rückwirkungen). Tatsächlich sind in seinem Verlauf einige Abstimmungen und Entscheidungen notwendig, insbesondere in dem grau schattierten Teil der Implementierung zweier ISRs mit gegenseitiger Beeinflussung.

Das Zusammenfügen der einzelnen Applikationsteile, bestehend aus generellen Threads, Timer-ISRs und ggf. Event-ISRs, beinhaltet die Organisation der Kommunikation zwischen den einzelnen Teilen sowie die Abstimmung des Zeitverhaltens. Als Kommunikation zwischen diesen Threads ist ein nicht-blockierendes Semaphoren/Mailbox-System ideal: Semaphore, die seitens eines Threads beschrieben und seitens der anderen gelesen und damit wieder gelöscht werden können, zeigen den Kommunikationsbedarf an, während die eigentliche Meldung in einer Mailbox hinterlegt wird.

Blockieren kann durch eine asynchrone Kommunikation wirksam vermieden werden: Threads warten nicht auf den Empfang bzw. Antwort, sie senden einfach (via Semaphor/Mailbox). Auch die Abfrage von empfangenen Sendungen erfolgt dann nicht-blockierend. Dies lässt sich durch einfache Methoden implementieren, wie am folgenden Beispiel ersichtlich ist.

Entscheidend ist die Einführung einer globalen Variablen zur Steuerung der Kommunikation (semaMess). Trägt diese den Wert 0, so liegt kein Messwert vor, und die Hauptroutine, die in eine Endlosschleife eingepackt ist, läuft weiter. Ansonsten wird der Messwert lokal kopiert und die Semaphore semaMess wieder zurückgesetzt, um für den nächsten Schleifendurchlauf einen korrekten Wert zu haben.

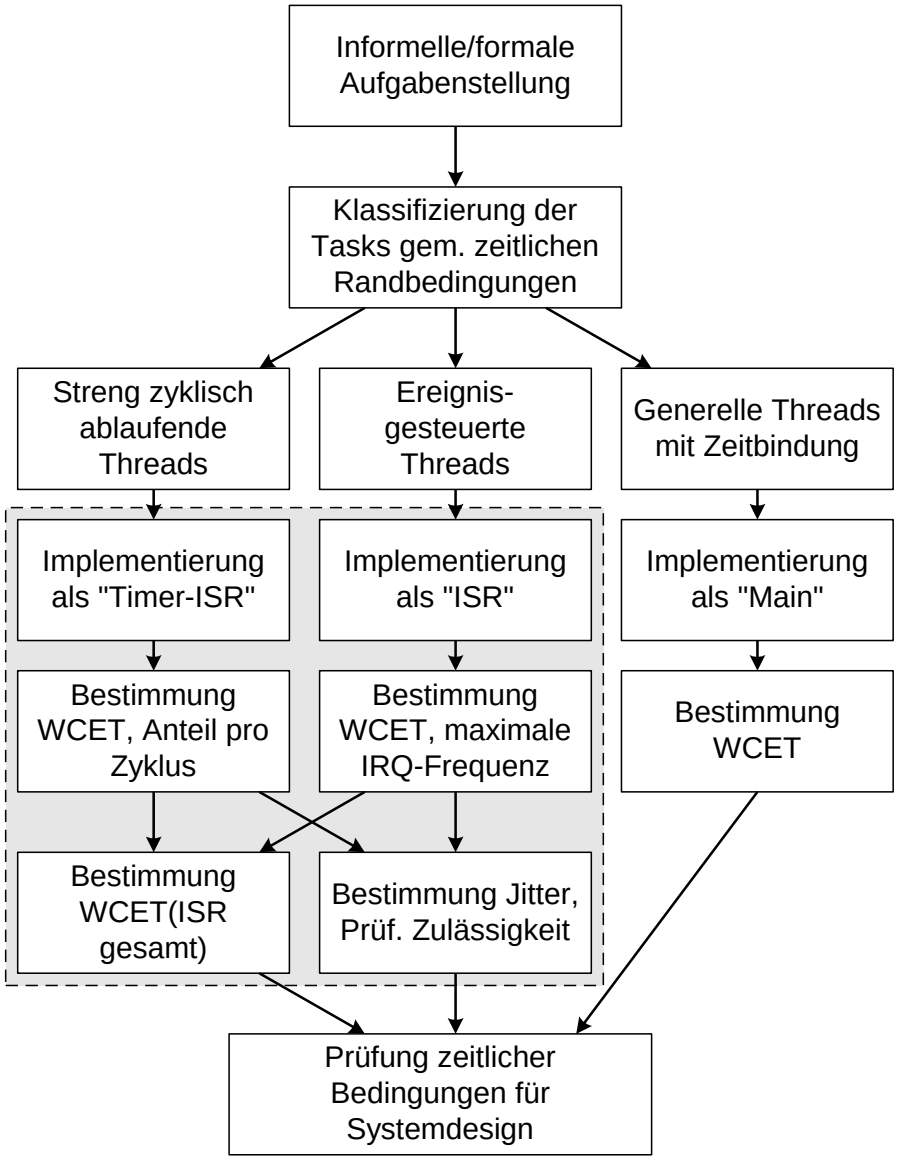


Bild 4.2 Gesamtablauf Systemdesign

Die Interrupt-Service-Routine, hier für einen Timer beschrieben, setzt zugleich den Messwert und die Kommunikationsvariable. Es wird im Codebeispiel davon ausgegangen, dass die ISR nicht unterbrechbar ist, so dass also die beiden Operationen immer hintereinander ausgeführt werden. Um dies im Hauptprogramm zu errei-

chen, muss kurzzeitig der Interrupt gesperrt werden (durch die beiden Assemblerbefehle cli (Clear Interrupt Enable) und sei (Set Interrupt Enable)). Hierdurch erreicht man im Code die geforderte Atomarität, die für einen ungestörten Ablauf zwingend erforderlich ist (→ 3.2.3, 3.3.2).

Die zeitliche Abstimmung der einzelnen Threads ist wesentlich aufwendiger und muss folgende Überlegungen einschließen:

- Wie beeinflussen ununterbrechbare Teile in dem generellen Thread bzw. einer ISR die Latenzzeiten der Interrupts? Die Beantwortung dieser Frage ist insbesondere für die zyklischen Threads mit strenger Zeitbindung wichtig, da man hier davon ausgehen muss, dass Jitter nur in sehr geringem Umfang erlaubt ist.

Die praktische Ausführung sieht so aus, dass tatsächlich die entsprechenden Befehle im Maschinencode (”sei“, ”cli“) gesucht und die WCETs der ununterbrechbaren Zwischenräume bestimmt werden. Diese Zeiten können mit WCIDT (Worst-Case Interrupt Disable Time) bezeichnet werden und sollten auf das absolute Minimum beschränkt sein, z.B. auf ”atomare“ Aktionen zur Kommunikation. Die Bestimmung hierzu muss am endgültigen Maschinencode erfolgen, um eingebundene Laufzeitroutinen zu erfassen.

- Timer-ISR und Event-ISR stehen in Konkurrenzbeziehung, was die Zuteilung der Rechenzeit betrifft. Grundsätzlich sollte der strengen Zeitbindung der Vorrang gegeben werden, und die Routinen hierfür sind auch Kandidaten für eine Ununterbrechbarkeit. Dies allerdings bedeutet die Erhöhung der Latenzzeit für die Event-ISR, was für den Einzelfall zu prüfen ist.

Eine Ausnahme bildet der Fall, dass die Event-ISR sehr hoch priorisiert werden muss, weil bei Auftreten ein sicherer Zustand zu erreichen ist. Dieses Ereignis muss sofort behandelt werden, so dass die Timer-ISR in diesem Fall unterbrechbar sein sollte.

Nach dem Zusammenfügen der einzelnen Teile und der Abstimmung der zeitlichen Randbedingungen kann dann die korrekte Funktionsweise des gesamten Systems nachgewiesen werden. Hierzu wird ein Zeitraum betrachtet, in dem ein gesamter Zyklus ablaufen kann. Insbesondere muss die generelle Task die Berechnung beenden können. In diesem Zeitabschnitt darf die Summe der WCETs, multipliziert mit den entsprechenden Auftrittshäufigkeiten, die Gesamtrechenzeit nicht überschreiten.

Für die Latenzzeiten gelten die gesonderten, oben beschriebenen Bedingungen.

4.2 Design-Pattern: Software Events

Der im vorigen Abschnitt beschriebene Mechanismus scheint darauf hinaus zu laufen, alle zeitkritischen Aktivitäten komplett in ISR zu halten und den Rest im Rahmen einer großen `while(1)`-Schleife im Hauptprogramm zu halten. Das

wäre insgesamt nicht besondere wartungsfreundlich, abgesehen davon, dass die ISR stark überlastet bzw. überfrachtet wären.

Dem Betriebssystem-losen Designansatz aus Abschnitt 4.1 wird nämlich gerne nachgesagt, dass er nur zu einer großen "`main()`-Schleife" ohne jegliche Struktur führt. Das Ergebnis wäre in der Tat schlecht, denn die Pflegbarkeit eines Programms steht und fällt mit der im Programm eingebauten Struktur.

Dieses scheinbare Manko kann aber mit einfachen Designmitteln vermieden werden, indem ein kleiner Teil des Betriebssystems mit sehr einfachen Mitteln nachgebildet wird: Der Scheduler.

Dem *Scheduling* obliegt die Aufgabe, das aktive Rechnen zwischen verschiedenen Teilen des Softwaresystems, hier als Tasks bezeichnet, hin- und herzuschalten. Folgende Struktur sei dem Gesamtsystem zugrunde gelegt:

- Es existieren verschiedene Quellen für Reaktionsanforderungen: Externe Hardware wie z.B. AD-Umsetzer, Netzwerkanschluss oder Sensoren werden mit internen Quellen wie Timer im System gemischt.
- Auf jede Reaktionsanforderung, im Folgenden *Event* genannt, wird auf gesonderte Weise reagiert, d.h. zu jedem Event gehört eine relativ komplexe Reaktionsroutine.

Für ein derartiges System kann die Kopplung zwischen Events verschiedener Quellen und den zugehörigen Reaktionsquellen durch folgendes Designpattern (Entwurfsmuster), das mit *Software Events* bezeichnet wird, erreicht werden:

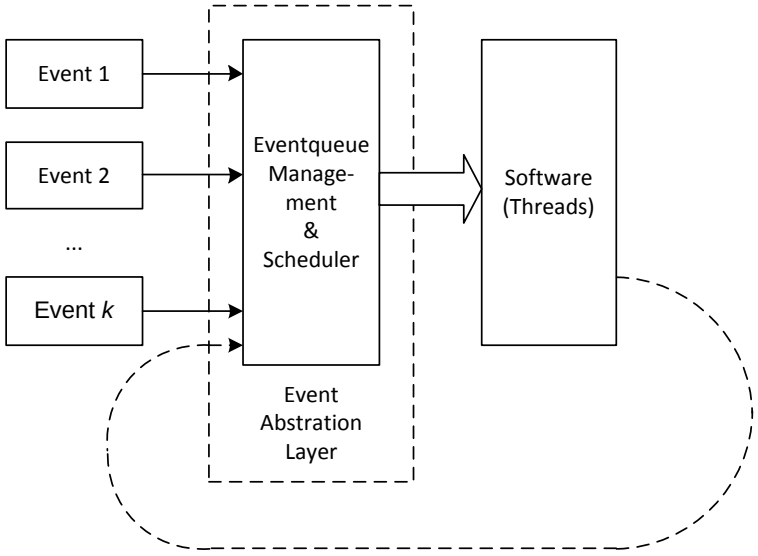


Bild 4.3 Gesamtarchitektur mit Software Events

4.2.1 1. Stufe: Vom Hardware- zum Softwareereignis

Es sei angenommen, dass jedes Ereignis des Außenprozesses in einer Interrupt Service Routine (ISR) behandelt wird. Diese Annahme stützt sich darauf, dass entweder das Ereignis zu einem Interrupt Request selbst führt und dieser dann die ISR aufruft (Ereignis-gesteuertes Design), oder dass im Rahmen der Behandlung eines Timer-Interrupts die Abfrage externer Quellen erfolgt (Zeit-gesteuertes Design).

In beiden Fällen wird in der betreffenden Interrupt Service Routine aus dem Hardware-Ereignis heraus ein Software-Ereignis erzeugt und in einer Ereignisqueue gespeichert. Der entsprechende Code könnte also wie in Bild 4.4 dargestellt aussehen:

Diese Routine schafft die Vereinheitlichung aller Ereignisse zur zentralen Bearbeitung in Software und führt zugleich eine (allerdings nicht perfekte) zeitliche Ordnung ein. Die hierbei genutzte Routine `vStoreNewEvent()` verwaltet dabei die Ereignisqueue, die im einfachsten Fall als FIFO-Buffer geführt und als Ringpuffer verwaltet werden wird. Die Verwaltung gelingt besonders einfach, wenn man die Ereigniseinträge als Struktur definiert:

```
void interrupt vISR()
{
    int iISRRegister;
    iISRRegister = iGetISRRegister();
    if( ISR_FLAG1 == (iISRRegister & ISR_FLAG1) )
    {
         iStoreNewEvent( EVENT_1, 0 );
    }    /* Pseudodatum 0  */
    else if (…)
         …
}
```

Bild 4.4a Rahmen für ISR-Routine zur Umsetzung von Hardware- in Software-Events

```
#define NUM_OF_EVENTS   16
struct stcEvent {
                  int iEventType;
                  int iEventData;
                  };
struct stcEvent stcEventQueue[NUM_OF_EVENTS];
int iEventWrite = 0, iEventRead = 0;
```

Bild 4.4b Definitionen für die Ereignisqueue

Die Funktionen zur Verwaltung der Warteschlange haben dann folgende in Bild 4.5a dargestellte Rahmengestalt:

Die Funktionen zur Verwaltung sind recht einfach gehalten. `iStoreNewEvent()` liefert einen Rückgabewert, der das erfolgreiche Speichern bzw. den Misserfolg signalisiert, in der ISR hier allerdings nicht ausgewertet wird.

Diese Form der Ereignisübermittlung kann beliebig erweitert werden; so sind Subtypen, weitere Daten, Uhrzeit etc. möglich und können direkt eingetragen werden.

```
int iStoreNewEvent( int iEventType, int iEventData )
{
    if( stcEventQueue[iEventWrite].iEventType != EVENT_NO_TYPE )
    {
        return( 0 );
    }   /* Rückgabewert 0 bedeutet, dass die Queue voll ist  */

    else
    {
        stcEventQueue[iEventWrite].iEventType = iEventType;
        stcEventQueue[iEventWrite].iEventData = iEventData;

        iEventWrite++;
        if( iEventWrite >= NUM_OF_EVENTS )
            iEventWrite = 0;

        return( 1 );
    }   /* Rückgabewert 1 bedeutet, dass das Ereignis gespeichert wurde  */
}

/*****************************************************************************/
void vGetNextEvent( struct stcEvent *stclTemp )
{
    if( stcEventQueue[iEventRead].ui8EventType != EVENT_NO_TYPE )
    {
        *stclTemp = stcEventQueue[iEventRead];
        stcEventQueue[iEventRead].iEventType = EVENT_NO_TYPE;
            /* Freigeben des Speichers  */

        iEventRead++;
        if( iEventRead >= NUM_OF_EVENTS )
            iEventRead = 0;
    }

    else
    {
        stclTemp->iEventType = EVENT_NO_TYPE;
    }   /* Bedeutet, dass kein Ereignis vorhanden ist  */
}
```

Bild 4.5a Funktionen zur Verwaltung der Ereigniswarteschlange

Die in Bild 4.5a skizzierten Funktionen, insbesondere iStoreNewEvent(), müssen näher betrachtet werden. Wenn diese Funktionen gemäß Bild 4.3 von zwei verschiedenen Seiten aufgerufen werden können – Threads können ebenso wie Interrupt Service Routinen Ereignisse in die Queue einfügen – dann kann die genutzte Funktion durch die ISR aufgerufen werden, wenn ein Thread sie gerade nutzt. Die ist möglich, da die ISRs ja per (asynchronem) IRQ gestartet werden.

Damit tritt für diese Funktionen das Problem der Reentrantfähigkeit auf (→ 9.3.2.4), d.h., sie müssten den beliebigen Wiedereintritt gestatten. Dies ist aber definitiv nicht der Fall, da jeder Aufruf globale Daten liest und schreibt, und zwar in Abhängigkeit voneinander.

Die Lösung des Problems besteht in der Nutzung des Interrupt-Disable-Flags, das jede Mikroprozessorarchitektur bietet (ggf. als Interrupt-*Enable*-Flag). Vor dem Aufruf aus einem Thread heraus muss dann dieses Flag gesetzt, nach Beendigung gelöscht werden, um eine Unterbrechung und den damit ggf. verbundenen neuen Aufruf zu verhindern. Dies gilt ggf. jedoch nicht, wenn der Aufruf aus einer ISR heraus folgt, da dann weitere Unterbrechungen verhindert sein sollten.

Der Aufruf kann mithilfe einer Wrapperfunktion erfolgen, wie in Bild 4.5b dargestellt ist. Die darin genutzten Funktionen vSetIRQDisableFlag() und vClearIRQDisableFlag() implementieren jeweils das Setzen und Löschen des Interrupt-Disable-Flags.

```
int iStoreNewEventSave( int iEventType, int iEventData )
{
    int iRetValue;
    vSetIRQDisableFlag();
    iRetValue = iStoreNewEvent( iEventType, iEventData );
    vClearIRQDisableFlag();
    return( iretValue );
}
```

Bild 4.5b Wrapperfunktion zur Verhinderung von Wiedereintritten

Während also die ISR weiterhin die Funktion iStoreNewEvent() nutzt (sofern sie nicht unterbrechbar ist), wird iStoreNewEventSave() von Seiten der Threads aufgerufen, und somit ist das Reentrant-Problem gelöst.

4.2.2 2. Stufe: Bearbeitung der Software-Ereignisqueue

Im Hauptprogramm kann die Software-Ereignisqueue innerhalb einer Endlosschleife abgearbeitet werden, da davon ausgegangen wird, dass *alle* Ereignisse und damit *alle* zu bearbeitenden Aufgaben in der Queue stehen. Die Endlosschleife hat damit folgende in Bild 4.6 dargestellte Gestalt:

Die darin verwendete Funktion `vExecuteEventQueue()` stellt den Scheduler dieses Systems dar: In der hier angedeuteten einfachen Form werden alle Ereignisse und deren damit zusammenhängenden Bearbeitungsroutinen der zeitlichen Reihenfolge nach bearbeitet. Hier lassen sich aber auch komplett andere Strategien mit Prioritäten usw. implementieren, so dass die zentrale Entscheidung, was wann bearbeitet wird, hier gefällt werden kann.

```
main()
{
...
    while( 1 )
    {
        vExecuteEventQueue();
    }
}


void vExecuteEventQueue( void )
{
    struct stcEvent stclTemp;

    vGetNextEvent( &stclTemp );

    switch( stclTemp.iEventType )
    {
        case 0:
            Rechne_fuer_Case0();
            break;

        case 1:
            Rechne_fuer_Case1();
            break;

        default:
            break;
    }
}
```

Bild 4.6 Bearbeitung der Ereignisqueue

4.2.3 Beispiele für die Nutzung dieses Design Pattern

Die in 3.3.2 beschriebene, blockierende Kommunikation beim Beschreiben eines EEPROMs kann im Rahmen dieses Design Pattern so gelöst werden, dass der Aufruf des Schreibens nicht mehr blockierend ist. Hierzu muss die dort enthaltene (→ Bild 3.11) For-Schleife muss nunmehr aufgelöst werden.

In Bild 4.7 ist dies so geschehen, dass nur noch der Schleifenkörper einschließlich des Inkrements des Index vorhanden ist. Wenn also eine Instanz, z.B. eine Zeitsteuerung, das Sichern der Variablen aus iConfig[] startet, muss dies durch den Event EVENT_WRITE_EEP mit dem Datenwert 0 (als 1. Index) erfolgen. Die Bearbeitung des Threads erfolgt dann so, dass zunächst getestet wird, ob in den Puffer (und nicht nur das EEPROM, was Warten bedeuten würde). Hierzu dient die (hypothetische) Funktion iTestForEEPWrite(), die mittels Rückgabewert 1 signalisiert, dass der Puffer bereit ist – wobei 0 für den besetzten Puffer steht.

Der Thread ruft sich immer wieder selbst auf, entweder mit altem oder neuem Index. Dieser Aufruf ist allerdings nicht direkt, sondern er erfolgt immer über die Eventqueue und den Scheduler.

```
eeprom int eepIBasicConfig[16];
int iConfig[16];


void vThreadWriteEEPROM( int iWriteIndex )
{
    if( iTestForEEPWrite() == 1 )
    {   /* Schreiben ist erlaubt */
        eepIBasicConfig[iWriteIndex] = iConfig[iWriteIndex];
        iWriteIndex++;
        /* Falls Ende noch nicht erreicht: Weiteres Event setzen */
        if( iWriteIndex < 16 )
            ui8StoreNewEvent( EVENT_WRITE_EEP, iWriteIndex );
    }

    else   /* Schreiben nicht erlaubt, Event nochmals setzen */
        ui8StoreNewEvent( EVENT_WRITE_EEP, iWriteIndex );
}
```

Bild 4.7 Nicht-blockierender Thread zum Schreiben in EEPROM-Variablen

4.2.4 Bestimmung der Puffergröße

Eine wichtige Voraussetzung für das reibungslose Funktionieren dieses Softwaredesigns ist eine genügende Puffergröße für die eintreffenden Softwareevents. Die konkrete Umsetzung dieser Tatsache hingegen erweist sich als schwieriger als zunächst vermutet.

Die Probleme, die bei der Bestimmung auftreten können, lassen sich am Beispiel des Netzwerks demonstrieren. Hier ist es schwierig, eine maximale Empfangsrate an Paketen zu bestimmen, da letztendlich niemand weiß, wie das Netzwerk konkret agieren wird.

> Bei Einsatz eines Ethernet-II-Netzwerks mit 100 Mbit/s beispielsweise beträgt die minimale Paketgröße 64 byte oder 512 bit. Theoretisch kann also alle 5,12 µs ein Paket durch den Ethernetcontroller empfangen werden, das wirklich für diesen Netzwerkknoten bestimmt ist.
>
> Wenn das Netzwerk das zulässt, angenommen innerhalb eines Zeitraums von 10 ms, bedeutet dies, dass der Puffer knapp 2000 empfangene Pakete speichern müsste – und bei einer angenommenen Bearbeitungsrate von 1 Paket/ms würde es 2 Sekunden dauern, bis alles bearbeitet wäre.

Die Dinge sind aber meist noch komplexer. Der Network-IC (NIC) bietet meist einen Zwischenpuffer an, in dem die Pakete im NIC selbst bis zur Bearbeitung gespeichert werden können. Typische Puffergrößen sind im bereich 48 .. 64 Kbyte, so dass lediglich 768 bis 1024 Pakete zwischengespeichert werden können. Dann wird über das Ethernet-II-Protokoll der weitere Paketfluss gebremst (indem der sendende Switch vom drohenden Überlauf informiert wird), und der Mikrocontroller hat kein so großes Puffer-Problem (wohl aber das Netzwerk, denn nun können Pakete auch verloren gehen).

Folgende Strategien können zu einer pragmatischen Lösung des Puffergrößenproblems führen:

- Man bestimme während des Betriebs, z.B. in einer realistischen Testphase, die Größe des genutzten Puffers. Für das Softwaredesign wird dann der größte Wert, multipliziert mit einem Sicherheitsfaktor (z.B. 5), als Puffergröße gewählt.
- Besondere Quellen für Events, die mit Bursts längerer Dauer oder erhöhten Aufkommens aufwarten können, werden ggf. auf eigene Eventqueues, z.B. nur einen einzigen Zähler, abgebildet. So könnte ein Eventzähler für das Netzwerk einfach ankommende Pakete zählen, ohne dass – bei einem 32-bit-Zähler – mit einem Überlauf gerechnet werden muss.
- Für alle Event-Quellen muss man sich eine Strategie überlegen, was mit dem Event passieren soll, wenn die Queue voll ist und die Speicherung abgelehnt wird. Dies muss sehr sorgfältig geschehen, da es versehentlich zu deadlocks etwa durch Sperrung von Puffern kommen kann. Ein Überschreiben älterer Events ist allein deswegen schwierig, weil auch hiermit Locks auf Ressourcen verbunden sein können, deren Aufhebung (aus Gründen der Nicht-Kenntnis) wesentlich schwieriger sein wird.

4.2.5 Bestimmung der Echtzeitfähigkeit eines Designs

Die Bestimmung der Echtzeitfähigkeit eines auf kooperativem Multithreading basierenden Designs erweist sich als ähnlich schwierig, solange keine Prioritäten im Scheduling eingeführt werden. In einem solchen Fall muss im Worst-Case angenommen werden, dass die gesamte Eventqueue (bis auf eine Stelle, in die das aktuelle, zu betrachtende Event geschrieben wird) gefüllt ist, und zwar mit Events, deren Bearbeitung die größte WCET (→ 3.2.3) besitzt – es sei denn, eine solche Konstellation kann explizit ausgeschlossen werden. Dies wird zu unrealistisch hohen Reaktionszeiten führen.

Mit Einführung von Prioritäten ändert sich die Lage entscheidend. Prioritäten können z.B. durch veschiedene Queues implementiert werden, pro Priorität jeweils eine. Die Bearbeitung kann streng nach Prioritäten erfolgen – Prio 0 so lange, bis keine mehr vorhanden ist, dann Prio 1 usw. –, man kann die strenge Reihenfolge aber mildern, indem alle x-mal Priorität k einmal Priorität k+1 ausgeführt wird.

Mit Einführung der Prioritäten muss dann nur noch für die einzelne Priorität die Echtzeitfähigkeit berechnet werden, analog zu den Gl. (3.8) und (3.9).

4.2.6 Kritische Würdigung dieses Design Pattern

Die Einfachheit dieses Multitasking-Ansatzes darf nicht über die Schwächen hinwegtäuschen. Zu den Schwächen zählen:

- Der Ansatz arbeitet nur kooperativ: Die im Scheduler aufgerufenen Funktionen müssen wieder zurückkehren, terminieren, um weitere Ereignisse bearbeitet zu lassen. Präemptives Scheduling, wie es aus Betriebssystemen bekannt ist, lässt sich so nicht realisieren.
- Grundsätzlich besteht sogar die Möglichkeit, die Ereignisqueue selbst zu blockieren, wenn sie mit gerade nicht bearbeitbaren Ereignissen komplett gefüllt wird. Es besteht die Gefahr eines *Deadlocks*, wenn alle Ereignisse gegenseitig aufeinander warten und kein Platz mehr in der Eventqueue vorhanden ist.
- Der Fall, dass Ereignisse vorübergehend nicht mehr gespeichert werden können, ist hier nicht vorgesehen bzw. wird nicht behandelt. Dies gehört aber zu einem sicherheitskritischen System, insbesondere, wenn keine Ereignisse verloren gehen dürfen. Hier ist die Größe des Ereignispuffers in Abhängigkeit von der Bearbeitungs- bzw. Leerungsgeschwindigkeit und der maximalen Erzeugungsrate entscheidend.

Dennoch bleibt positiv anzumerken, dass sich dieser Ansatz des Multithreading auch auf verteilte Systeme erweitern lässt (→ 4.6).

4.3 Komplett statischer Ansatz durch Mischung der Tasks

Ein in [Dea04] dargestellter Ansatz verzichtet sowohl auf ein Scheduling durch ein Betriebssystem als auch auf die Einbindung von Interrupt Service Routinen. Kurz gesagt besteht die Methode darin, den zeitkritischen Teil derart mit dem unkritischeren Teil zu mischen, dass sich – zur Übersetzungszeit berechnet – ein richtiges Zeitgefüge in der Applikation einstellt.

Die Idee wird als ”Software Thread Integration (STI)“ bezeichnet und ist natürlich bestechend einfach. Prinzipiell kann jeder Softwareentwickler dies durchführen, indem – nach sorgfältiger Analyse – die Sourcecodes der einzelnen Threads gemischt werden.

Das Problem ist, dass zugleich ein zyklusgenaues Ausführen des Programms gefordert wird, wenn harte Echtzeitbedingungen einzuhalten sind. Zyklusgenauigkeit ist aber derzeit nur unter mehreren Bedingungen erreichbar:

- Die Anzahl der Ausführungstakte im Mikrocontroller muss zur Übersetzungszeit bestimmbar sein. Hiermit scheiden bisherige Cache-Konzepte aus, denn sie ermöglichen nur statistische, nicht deterministische Aussagen.
- Alternativpfade (if – else) müssen die gleiche Anzahl an Taktschritten aufweisen.
- Die Bestimmung der Anzahl der Ausführungstakte (WCET) muss in der Programmiersprache möglich sein.

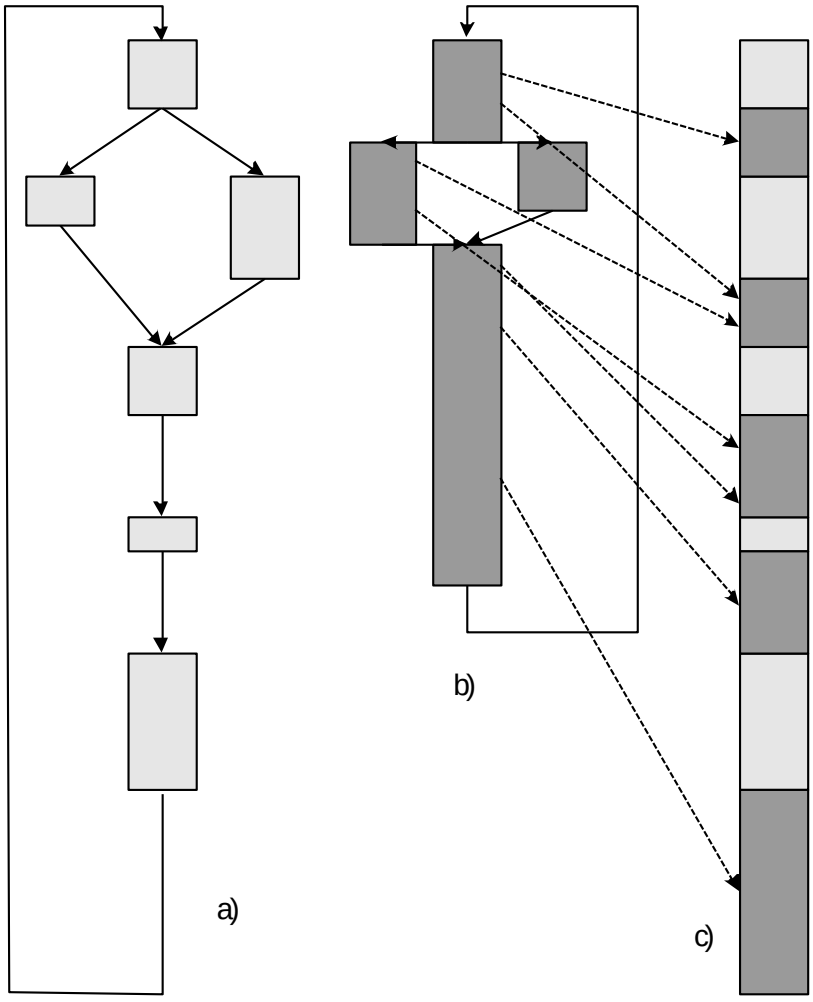


Bild 4.8 Mischung zweier Threads zwecks Software Thread Integration
a) Primärthread, zeitkritisch b) Sekundärthread, zeitunkritisch c) Thread Integration

Der erste Punkt ist fast automatisch dadurch erfüllt, dass sich diese Methode auf kleine Mikrocontroller (”low-MIPS world“) bezieht. Diese Mikrocontroller besitzen keinen Cache, weil sie zumeist auch nur mit geringen Taktraten versehen sind (etwa 20 MHz) und weil der Cache-Speicher sehr teuer wäre.

Punkt 3, die Bestimmung der Anzahl der Ausführungstakte im Rahmen des Codes, ist auf Ebene einer Hochsprache zurzeit nicht möglich. Hier muss man auf Assembler ausweichen, was mit erheblichen Problemen verbunden ist. Hierunter fällt auch zugleich Punkt 2, denn die eventuelle Auffüllung von schnelleren Pfaden mit ’NOP’-Befehlen (no operation) zwecks Angleichung kann wiederum nur auf Assemblerebene erfolgen.

Folgerichtig bemüht sich der Autor in [Dea04] um eine neue Compilertechnologie, die nach Übersetzung in Assemblercode diesen analysiert, die unterschiedlichen Wege in ihrer Ausführungszeit angleicht und schließlich den Code mischt.

Nach Bestimmung der Ausführungszeiten wird (→ Bild 4.8) der zeitkritische Code zyklusgenau in den auszuführenden Softwarethread eingefügt. Hier ist auch offensichtlich, dass alle Zweige einer Verzweigung die gleiche Laufzeit aufweisen müssen, weil ansonsten von dem Zeitschema abgewichen wird. Die Lücken werden dann durch zeitlich unkritische Teile aufgefüllt.

Dieses Verfahren wirft eine Reihe von Fragen auf, die Compilertechnologie betreffend. Möglich ist es grundsätzlich, wenn die Worst-Case Execution Time (WCET) gleich der Best-Case Execution Time (BCET) ist. Die in [Dea04] dargestellten Methoden, um den Code zu mischen, sind dann von der Güte der WCET-Bestimmung und den Möglichkeiten des Compilers, möglichst einfache Threadwechsel einzubauen, abhängig. Der Gewinn an Performance, verglichen mit einem normalen Scheduling, ist allerdings beträchtlich, er wird mit bis zum Faktor 2 an Performance quantifiziert.

4.4 Co-Design Ansatz: Partitionierung in PLD- und Prozessoranteile

Implizit wurde bei allen bisherigen Modellen zur Echtzeitfähigkeit vorausgesetzt, dass die charakteristischen Zeiten wie Reaktionszeit, Antwortzeit usw. wesentlich größer sind als die Zeit, die ein Prozessor zur Bearbeitung eines Befehls benötigt. Dies muss vorausgesetzt werden, weil der Prozessor in der zeitsequenziellen Ausführungsdimension arbeitet: Er benötigt einfach viele Befehle, um ein Programm zu bearbeiten, und jeder Befehl benötigt etwas Zeit (ca. 1 Takt).

Bild 4.9 a) zeigt nun ein Beispiel für eine relativ einfache Ansteuerung eines AD-Wandlungsvorgangs. Diese Routine ist als Interrupt-Serviceroutine ausgelegt. Angestoßen beispielsweise durch einen zyklischen Timer-IRQ wird der AD-Wandler auf einen neuen Wert abgefragt, und dieser neue Wert wird mit gegebenen Grenzen

verglichen. Bleibt der Wert in den Grenzen, passiert nichts, ansonsten wird die out_of_range()-Routine aufgerufen.

Bild 4.9 b) zeigt nun die Assemblerübersetzung dieser Routine für einen hypothetischen Prozessor. In dem Fall, dass kein Grenzwert verletzt wird, benötigt die Routine 14 Instruktionen, bei 1 Instruktion/Takt (RISC-Verhältnis) also 14 Takte oder 140 ns bei 100 MHz.

Dies erscheint als nicht besonders viel, aber bei einer AD-Wandlungsrate von 1 MSPS (Mega-Samples-per-Second) sind dies 14% der gesamten Rechenkapazität des Prozessors. Hieraus lässt sich schon ein ungefähres Maß dafür ableiten, wann die Behandlung von Ereignissen in nicht-exklusiver Hardware schwierig bis unmöglich wird. Folgende Kriterien können angegeben werden:

- Wiederholungsfrequenz > 1/100 .. 1/1000 * Prozessorfrequenz
- Geforderter Jitter (Abweichung des Starts der Reaktionsroutine) < 10 .. 1000 Instruktionszeiten
- Bearbeitungszeit einer ISR > 10% Gesamtbearbeitungszeit

Die angegebenen Grenzen sind unscharf, sie sollen lediglich zeigen, dass man bei keinem noch so gut ausgelegten Prozessor-basierten System beliebig kleine und scharfe Reaktionszeiten erwarten kann. Für diesen Fall bietet sich eine Partitionierung des Systems an, die besonders kritischen Teile können in exklusiver Hardware untergebracht werden.

```
int *p_adc, adc_value, upper_limit, lower_limit;
...
void interrupt read_and_compare_ADC()
{
    adc_value = *p_adc;             // Access to AD converter
    if( adc_value > upper_limit  ||  adc_value < lower_limit )
    {
        out_of_range();             // call to exception routine
    }
}
```

Bild 4.9 a) C-Sourcecode für ISR zur AD-Konvertierung mit Grenzwertvergleich

Aktuell sind hierfür Kombinationen aus Prozessor und PLD am Markt erhältlich. Beide sind programmierbar, wenn auch auf vollkommen verschiedene Weisen, so dass der Entwickler in das Gebiet des Co-Designs (→ 10) gerät. Wie Bild 4.10 zeigt, wird in dem Beispiel die Abfrage des AD-Wandlers sowie der Vergleich mit den Grenzen in dem PLD-Teil implementiert, der damit das komplette Interface zum ADC enthält. Der Mikrocontroller wird lediglich dann unterbrochen, wenn die Grenzwertverletzung auftritt und somit eine ’echte‘ Behandlung notwendig ist.

```
TIMER:  push    r0              ;
        push    r1              ;
        push    r2              ;
        mov     r0, ADC         ; Lesen des AD-Werts, zugleich Neustart der Wandlung
        mov     r1, UP_LIMIT    ; Speicherstelle für oberes Limit
        mov     r2, DN_LIMIT    ; dito, untere Grenze
        cmp     r0, r1          ; Grenzen werden verglichen
        bgt     T1              ; Überschreitung, spezielle Routine!
        cmp     r0, r2          ;
        bge     T2              ; Keine Unterschreitung, dann Sprung
T1:     call    OUT_OF_RANGE;
T2:     pop     r2              ;
        pop     r1              ;
        pop     r0              ;
        reti                    ; Beenden der Serviceroutine
```

Bild 4.9 b) Assemblerübersetzung

Zur Unterbringung von Ereignisbehandlungen ist natürlich auch hergestellte Hardware (ASIC) geeignet, dies stellt lediglich eine Frage der Herstellungszahlen und –kosten dar. Für den Jitter und die Bearbeitungszeiten der Hardware-Routinen kann man allgemein sagen, dass diese in der Größenordnung eines oder weniger Takte liegt.

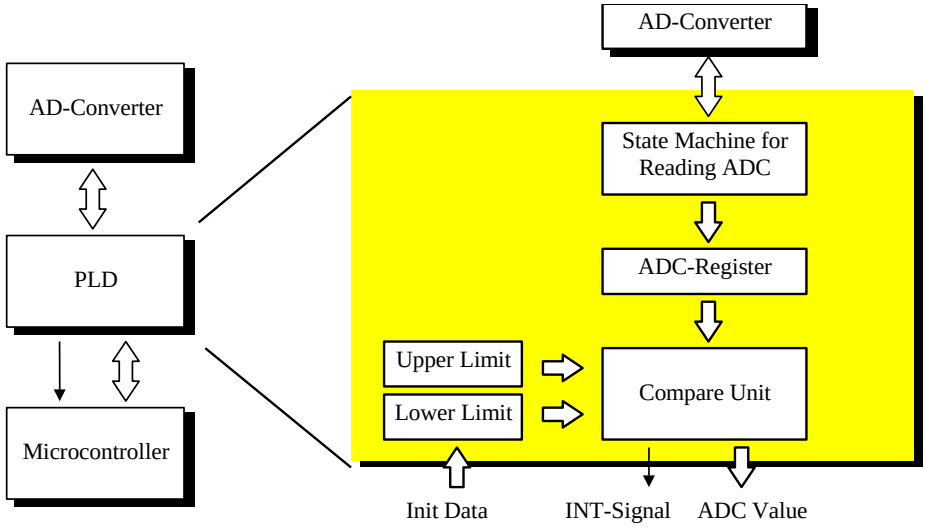


Bild 4.10 Implementierung der AD-ISR in PLD

4.5 Zusammenfassung der Zeitkriterien für lokale Systeme

4.5.1 Kriterien für Co-Design

Aus den bisherigen Betrachtungen lässt sich resümieren, dass einige Zeitkriterien existieren, die die Behandlung und die Implementierungsart entscheidend beeinflussen. Im Wesentlichen sind dies drei Kriterien, die aus der Prozessumgebung stammen:

- Der zeitliche Jitter *Tjt* (auch als maximale Latenzzeit zu bezeichnen, siehe Definition 2.6 und 3.4) gibt diejenige Zeit an, mit der der Start der Reaktionsroutine schwanken darf. Gründe hierfür können zeitsynchrone Aktivitäten sein, für die nur geringe Abweichungen akzeptierbar sind. Liegt dieser Jitter unterhalb ca. 100 Befehlsausführungszeiten, so kann mit Sicherheit davon ausgegangen werden, in einem für Prozessoren kritischen Bereich zu liegen.

Die unkritische Grenze, ab der also mit einem garantierten Verhalten des Prozessors zu rechnen ist, ist natürlich individuell von dem System abhängig. In jedem Fall ist das System sicher konzipiert, wenn der erlaubte Jitter größer ist als die Summe aller höherpriorisierten Ereignisse (unter Einbezug der Auftrittshäufigkeit) bei Ereignis-gesteuerten Systemen bzw. die Zykluszeit bei Zeit-gesteuerten Systemen.

- Die Servicezeit T_{Service} spielt eine scheinbar unwichtige Rolle, da sie ja sowieso eingeplant werden muss. Bei Servicezeiten, die mehr als 30% der gesamten Rechenzeit (im Normalfall oder Worst Case) einnehmen, muss man jedoch davon ausgehen, dass diese Zeit sehr dominant ist und die übrigen Teile des Systems stark beeinflusst. Diese 30%-Grenze ist allerdings unscharf, während Servicezeiten < 1% sicher keinen Einfluss nehmen.

	Kritischer Wert	Unkritischer Wert
T_{Jitter}	< 100 Befehlszeiten	> ∑ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)
T_{Service}	> 30 % der gesamten Rechenzeit	< 1 % der gesamten Rechenzeit
T_{Reaction}	< 100 Befehlszeiten	> ∑ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)

Tabelle 4.1 Zusammenfassung der charakteristischen Zeiten von Ereignissen

- Die maximal geforderte Reaktionszeit T_{Reaction} setzt sich aus der Latenzzeit und der Servicezeit zusammen, allerdings müssen noch mögliche Unterbrechungen mitbetrachtet werden. Kritisch wird es für die Reaktionszeit, wenn diese etwa < 100 Befehlsausführungszeiten ist (die Grenze ist auch hier wieder individuell). Die unkritische Grenze wird wieder bei der Summe über alle Reaktionszeiten bzw. der Zykluszeit erreicht.

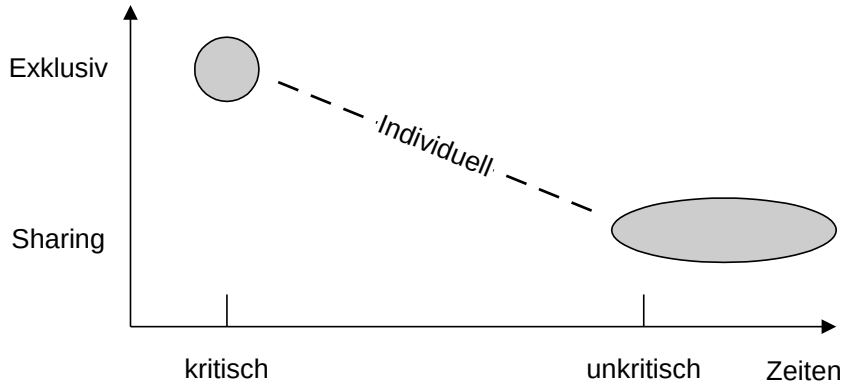


Bild 4.11 Designraum exklusiv/sharing für Systeme mit zeitlichen Randbedingungen

Tabelle 4.1 fasst die charakteristischen Zeiten zusammen. Ist für eine von diesen Zeiten für ein konkretes System die kritische Grenze erreicht, so ist der Systemdesigner aufgefordert, exklusive Hardware hierfür bereitzustellen. Sind hingegen alle Zeiten unkritisch, kann man zu einem Sharing-Betrieb übergehen. Im Zwischenbereich hingegen muss individuell konzipiert werden, was die geeignete Wahl darstellt (siehe Bild 4.11).

Die Notwendigkeit des Wechsels der programmierbaren Plattform kann auch auf andere Weise hervorgehoben werden. Bild 4.12 zeigt hierzu vergleichend die Designbereiche von Mikroprozessoren und programmierbaren Hardwarebausteinen.

Für den Mikroprozessor sei hierfür angenommen, dass er mit einer Taktrate von 100 MHz arbeitet und pro Takt einen Befehl ausführt (RISC-Architektur). Dies bedeutet, dass alle 10 ns (0,00001 ms) ein Befehl beendet wird, woraus sich die Performancegrenze für diesen Mikroprozessor in Bild 4.12 ergibt. Als Komplexitätsmaß wird hier die Anzahl der maximal zu durchlaufenden Instruktionen gewählt.

Die angegebenen Zeiten werden als Rechenzeiten interpretiert, weil in einem Single-Task-System nur eine einzige Aufgabe ansteht. In einem Multi-Task-System – oder Multithread-System – mit Echtzeitfähigkeit hingegen werden die Zeiten als Reaktionszeiten betrachtet, die aus Latenz- und Ausführungszeit bestehen.

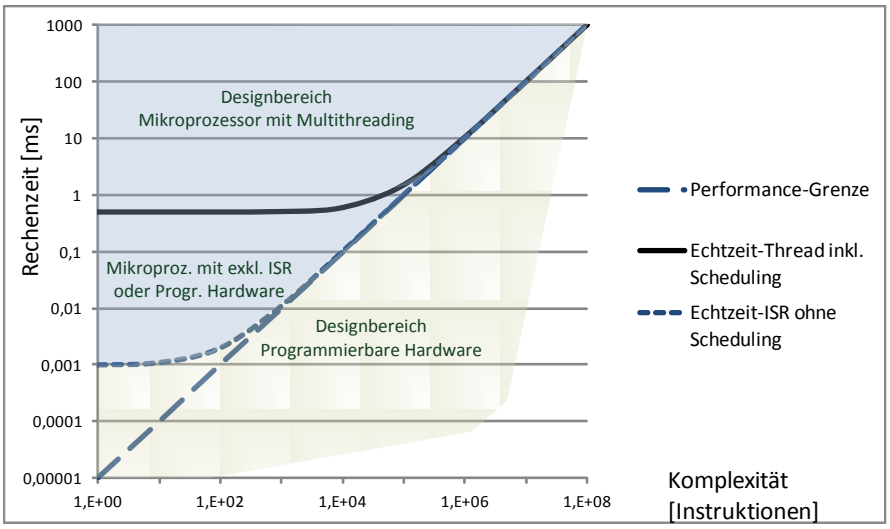


Bild 4.12 Darstellung der Arbeitsbereich von Mikroprozessoren und programmierbarer Hardware

Wählt man ein einfaches Modell, so kann man für die Latenzzeiten konstante (oder eben maximale) Werte annehmen, im Modell 0,5 ms. Damit ist klar, dass der Mikroprozessor auch bei noch so kleinen Algorithmen nicht unterhalb 0,5 ms reagieren kann, zumindest nicht garantiert (wohl aber im Einzelfall).

Eine Chance, den Mikroprozessor auch bei kleineren Reaktionszeiten zu nutzen, besteht noch darin, die gesamte Reaktion in einer Interrupt-Service-Routine hoher Priorität zu implementieren. Dies entspricht dem Ansatz in 3.2, hierdurch kann – bei kleinen Reaktionsroutinen und damit kleinen Ausführungszeiten – entsprechend schnelle Reaktionen implementiert werden. Ab einer (unscharfen) Grenze (in Bild 4.12: 1 µs für die Echtzeit-ISR ohne Scheduling) muss aber auch hier auf eine andere Plattform gewechselt werden.

Mit anderen Worten: Irgendwann muss die Ausführungsplattform gewechselt werden, oder jedem Ereignis wird eine exklusive Ausführungsplattform zur Verfügung gestellt.

Die programmierbare Hardware (Programmable Logic Devices, PLD, oder Field-Programmable Gate Arrays, FPGA) zeigt auch Grenzen, wie Bild 4.12 zu entnehmen ist. Etwa bei Ausführungszeiten von 100 ns (10 MHz Takt) bis herunter zu 10 ns (100 MHz) werden – von Einzelfällen in beide Richtungen abgesehen – Grenzen erreicht, außerdem ist die Kapazität eines derartigen Bausteins ebenfalls begrenzt. Diese Grenzen werden bei dem Äquivalent zu 10^6 bis 10^7 Instruktionen erreicht (wieder mit Ausnahmen). Im Übrigen haben auch Mikroprozessoren Kom-

plexitätsgrenzen, bedingt durch den endlichen Speicherplatz, die aber wesentlich weiter gesteckt sind.

4.5.2 Kriterien für Designentscheidungen im Co-Design

Aus den Überlegungen zu den charakteristischen Zeiten kann man einen Entscheidungsbaum entwerfen, anhand dessen die Abbildung auf die Implementierungsplattform entschieden werden kann. Bild 4.13 zeigt einen solchen Entscheidungsbaum.

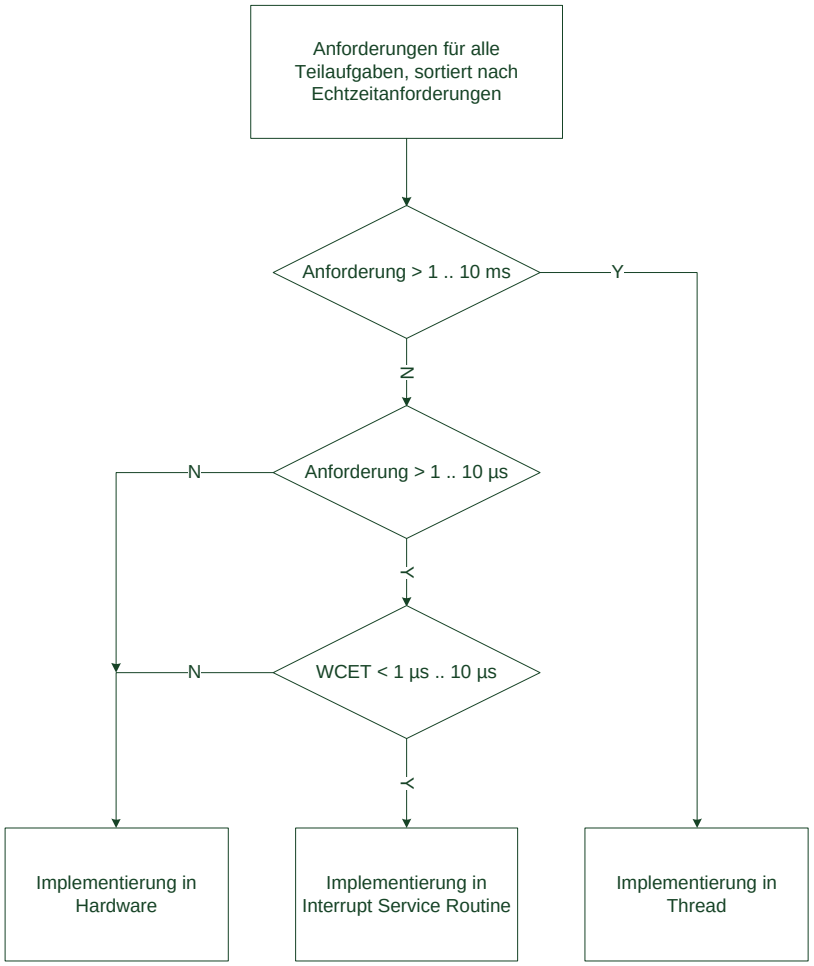


Bild 4.13 Entscheidungsbaum für Implementierungsplattform im Co-Design

In diesem Entscheidungsbaum wurden drei Implementierungsplattformen gewählt: Programmierbare Hardware (PLD, → 4.4), Interrupt Service Routinen ohne Sche-

duling (aber Prioritäten) (→ 3.2.3) und Threads mit (ggf. prioritätsbasiertem) Scheduling (→ 4.2), aber es sind noch weitere Plattformen denkbar, etwa Mehrprozessorsysteme, fixierte Hardware etc.

Eine weitere Vereinfachung besteht darin, dass geforderte Minimalwerte für Jitter nicht beachtet werden, lediglich Laufzeiten sind als relevant für die Entscheidungen zur Abbildung auf eine bestimmte Hardwareplattform benutzt worden.

Sofern ein Thread auf eine exklusive Hardware abgebildet wurde und keine (blockierende oder belastende) Kommunikation auftritt, wird dieser Thread parallel zu allen anderen ausgeführt. Die Threads auf einem Prozessor hingegen belasten sich gegenseitig mit ihrer Rechenzeit, so dass hier im Allgemeinen auch priorisiert werden soll.

Als Faustformel für die Verteilung von Prioritäten innerhalb eine Plattform gilt, dass diese analog zum Verhältnis (maximale Reaktionszeit)/Rechenzeit vergeben werden sollten: Je höher das Verhältnis ist, desto geringer kann dieser Thread priorisiert werden, wobei dann der resultierende Jitter akzeptabel sein muss.

5 Eingebettete Systeme und Verlustleistung

Dieses Kapitel dient dem Zweck, den Zusammenhang zwischen den Systemen, die programmiert werden können, den Entwurfssprachen und den in Kapitel 1 bereits diskutierten Randbedingungen darzustellen.

Hierzu wird der quantitative Zusammenhang zwischen Fläche *A*, Zeit *T* und Verlustleistung *P* untersucht. Dieser Zusammenhang dürfte existieren, die Quantifizierung ist interessant. Hat man nun mehrere Möglichkeiten, kann man das Design optimieren. Man spricht dann auch von dem *Designraum*.

5.1 Der quantitative Zusammenhang zwischen Rechenzeit, Siliziumfläche und Verlustleistung

Rechenzeit und Siliziumfläche

Folgende Gedankenkette zeigt einen zumindest qualitativen Zusammenhang zwischen Zeit und Fläche. Für einen 8-Bit-Addierer existieren viele Implementierungsmöglichkeiten:

- Sequenziell: 1-Bit-Addierer mit Shift-Register als Speicher, getaktete Version. Dieser Addierer berechnet in einem Takt nur ein Summenbit sowie das Carry-Bit, beide werden gespeichert und weiter verwendet.
- Seriell: Ripple-Carry-Adder, 8*1-Bit-Addierer mit seriellem Übertrag. Dieser Addierer ist die bekannte Form und wird gelegentlich auch als sequenziell bezeichnet.
- Total parallel: Addierschaltung, bei der alle Überträge eingerechnet sind. Hier ist die Berechnungszeit unabhängig von der Breite der Eingangswörter.
- Carry Look-Ahead Adder: Zwei Schaltnetze, eines für Carry, ein folgendes für die Addition. Hier wird zwar die im Vergleich zum total parellelen Addierer doppelte Zeit benötigt, aber immer noch unabhängig von der Datenbreite.
- Zwischenformen wie 4*2-Bit-Paralleladdierer usw.

Bild 5.1 zeigt reale Werte für einen 12-Bit-Addierer. Als Standardverzögerungszeit sind 10 ns pro Gatter angenommen, zur Flächenbestimmung wurde die Zahl der Terme (Disjunktive Normalform DNF) herangezogen.

Hieraus und aus anderen Schaltungen kann man zunächst empirisch schließen, dass es für begrenzte Schaltungen ein Gesetz wie

$$A \cdot T^k = \text{const(technology)} \quad \text{mit } k = 1..2 \tag{5.1}$$

gibt. Dieses Gesetz ist zwischenzeitlich auch theoretisch bestätigt worden. Die Exponenten *k* tendieren für arithmetische Operationen gegen 2.

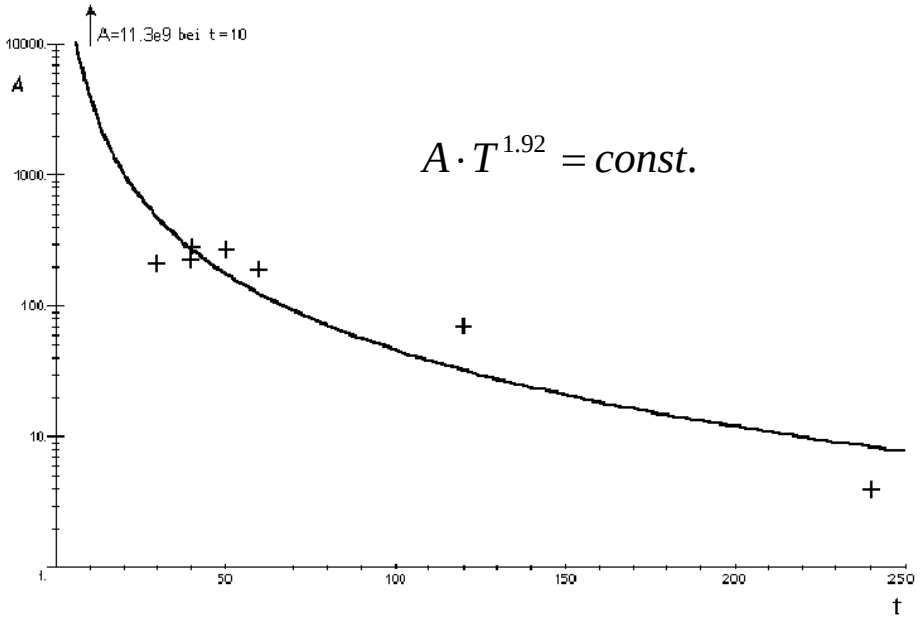


Bild 5.1 AT-Gesetz für 12-Bit-Addierer, verschiedene Implementierungsvarianten

Interpretation: Es liegt hier eine Trade-Off-Funktion vor, die verdeutlichen soll, dass man – je nach Randbedingungen – ein applikationsspezifisches Optimum finden kann.

Weiterhin können einzelne Implementierungen von diesem Zusammenhang signifikant abweichen. Man kann daher die durch diese Funktion gezogene Grenze als Optimalitätskriterium heranziehen, so dass Punkte unterhalb der Kurve (siehe auch Bild 5.1) optimal sind.

Definition 5.1:

Die Flächen-Zeit-Effizienz (space-time-efficiency) $E_{S/T}$ ist definiert als

$$E_{S/T} = \sqrt{\frac{1}{A \cdot T^2}} = \frac{1}{\sqrt{A} \cdot T}$$

Während das $A*T^k$-Gesetz als Zusammenhang für eng begrenzte Operationen, also etwa einen Addierer gefunden wurde, wird es aktuell auch zur Beurteilung ganzer ICs benutzt, beispielsweise für Mikroprozessoren.

Rechenzeit und Verlustleistung

Der Zusammenhang zwischen Verlustleistung und Rechengeschwindigkeit kann etwas genauer betrachtet (und auch hergeleitet) werden. Bei einem CMOS-Design, wie es für Mikroprozessoren State-of-the-Art ist, zählen 3 Komponenten zur Verlustleistung hinzu:

$$P_{total} = P_{SC} + P_{leakage} + P_{switching\_losses} \tag{5.2}$$

P_{SC} (Short Current, Kurzschlussstrom) resultiert aus demjenigen Strom, der kurzzeitig beim gleichzeitigen Umschalten beider Transistoren eines CMOS-Paares fließt. Dies ist prinzipbedingt im CMOS-Design verankert, und die Anzahl der Umschaltungen pro Zeiteinheit ist natürlich proportional zum Takt.

$$P_{SC} = V \cdot I_{SC} \tag{5.3}$$

$P_{leakage}$ (Leakage Current, Leckstrom) entstammt aus dem dauerhaft fließenden Leckstrom einer elektronischen Schaltung. Dieser Strom ist bei CMOS-Schaltungen natürlich sehr klein, weil in jedem Stromkreis mindestens ein Transistor sperrt, er ist aber nicht 0. Aufgrund der enormen Anzahl an Transistoren in aktuellen Schaltungen sowie der ständigen Verkleinerung der Strukturen summieren sich die Ströme zu mittlerweile signifikanten statischen Verlustleistungen:

$$P_{leakage} = V \cdot I_{leakage} \tag{5.4}$$

$P_{switching\_losses}$ (Switching Losses, Schaltverluste) ist derjenige Anteil, der aktuell als dominant betrachtet wird. Dieser Anteil entstammt dem Umladestrom, der durch das Laden und Entladen der Transistorkapazitäten entsteht. Die daraus resultierende *mittlere* Verlustleistung ist bei gegebener Umladefrequenz *f*

$$P_{switching\_losses} = \frac{C}{2} \cdot V^2 \cdot f \tag{5.5}$$

Vernachlässigt man insbesondere den statischen Verlustleistungsanteil – ein Vorgang, den man bei einigen höchstintegrierten Schaltungen bereits nicht mehr machen kann –, dann gilt der bekannte Zusammenhang, dass bei konstanter Spannung die Verlustleistung *P* linear mit der Frequenz *f* steigt.

Also ein linearer Zusammenhang zwischen Verlustleistung und Rechengeschwindigkeit? Nein, denn Gl. (5.5) gilt bei konstanter Spannung, und genau diese Betriebsspannung lässt sich bei sinkender Betriebsfrequenz in modernen CMOS-Schaltungen ebenfalls absenken. Um diesen Effekt zu quantifizieren, sei folgende Ableitung gegeben:

Die Kapazität C im Transistor bleibt konstant und muss beim Umschalten geladen werden. Die dafür notwendige Ladungsmenge ist

$$Q = C \cdot V = I \cdot t_{\min} = \frac{I}{f_{\max}} \tag{5.6}$$

Der Ladestrom I ist von der Betriebsspannung und der Schwellenspannung V_{th} (Threshold-Voltage) abhängig. Diese Abhängigkeit ist etwas komplexer, aktuell wird folgende Näherung angenommen:

$$I = const. \cdot (V - V_{th})^{1,25} \tag{5.7}$$

Die maximal mögliche Frequenz ergibt sich durch Einsetzen von (5.7) in (5.6) und Auflösung nach f_{max}. Hierbei kann eine weitere Näherung für den Fall angenommen werden, dass V von V_{th} weit genug entfernt ist:

$$f_{\max} = const\_1 \cdot \frac{(V - V_{th})^{1,25}}{V} \approx const\_2 \cdot V \quad (\text{für } (V - V_{th}) \geq V_{th}) \tag{5.8}$$

Diese Formel sagt also aus, dass mit der Skalierung der Betriebsspannung V auch die maximale Betriebsfrequenz f_{max} skaliert. Insgesamt gilt mit allen Näherungen der quantitative Zusammenhang

$$P \cdot T^3 = const. \tag{5.9}$$

Interpretation: Dieser Zusammenhang zeigt auf, wie Verlustleistung und Rechengeschwindigkeit sich gegenseitig beeinflussen, wenn Betriebsspannung und Frequenz verändert werden dürfen. Der gewaltige Zuwachs der Verlustleistung (bei verdoppelter Frequenz 8fache Verlustleistung) ist sehr signifikant.

Bild 5.2 zeigt den Zusammenhang zwischen P, A und T (in qualitativer Form). Es wird für die Zukunft angenommen, dass Server-Architekturen optimiert auf Rechengeschwindigkeit, Architekturen für eingebettete Systeme jedoch mehr auf Verlustleistungsminimierung (und damit Flächenminimierung) ausgelegt sein werden.

Anmerkung: Die Reduzierung der Strukturbreiten in den ICs haben aktuell Auswirkungen auf die Betriebsspannung und die Verlustleistung. Durch die kleiner werdenden Strukturen muss die Betriebsspannung gesenkt werden. Dies führt auch zu sinkenden Thresholdspannungen, was wiederum zu drastisch steigenden statischen Verlustleistungen führt. Die Herleitung, insbesondere der Teil nachdem (5.5) den einzigen nennenswerten Beitrag zur Verlustleistung liefert, gilt dann zukünftig nicht mehr. Es kann sogar so sein, dass die statische Verlustleistung überwiegt.

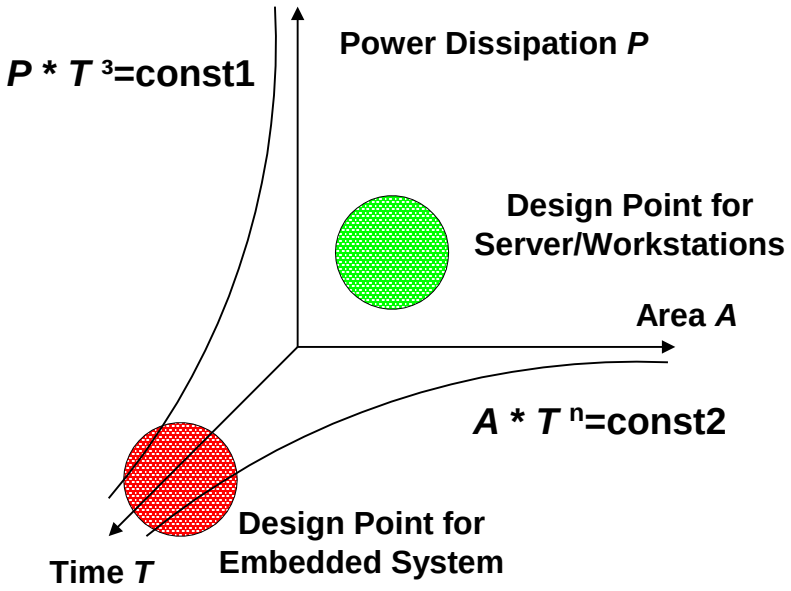


Bild 5.2 Zusammenhang zwischen *P*, *T* und *A*

5.2 Ansätze zur Minderung der Verlustleistung

Wie bereits in Abschnitt 5.1 gezeigt wurde, existiert ein quantitativer Zusammenhang zwischen Verlustleistung und Rechenzeit. Das dort abgeleitete Gesetz, dass $P*T^3 = \text{const.}$ gelten soll, gilt allerdings nur unter der Voraussetzung, dass man sich in einem Design (sprich: eine Architektur) bewegt und Versorgungsspannung sowie Taktfrequenz ändert.

Das ist natürlich auch eine Methode, aber eben nur eine, die zur Verlustleistungsminderung in Frage kommt. In der Realität sind es 4 Methoden, die zur Anwendung kommen:

- Auswahl einer Architektur mit besonders guten energetischen Daten
- Codierung von Programmen in besonders energiesparender Form
- Einrichtung von Warte- und Stoppzuständen
- Optimierung der Betriebsfrequenz und Betriebsspannung nach Energiegesichtspunkten

Und um es vorweg zu nehmen: Dies ist ein hochaktuelles Forschungsgebiet, es gibt Ansätze [BBM00], aber noch keinerlei analytische Lösungen. Im Folgenden sollen diese Ansätze kurz diskutiert werden.

5.2.1 Auswahl einer Architektur mit besonders guten energetischen Daten

Es mag auf den ersten Blick natürlich unwahrscheinlich erscheinen, warum einige Architekturen mehr, andere weniger Verlustleistung (bei gleicher Geschwindigkeit) benötigen, dennoch stellt sich in der Praxis immer wieder heraus, dass es drastische Unterschiede bei Mikroprozessoren und Mikrocontrollern gibt [Bro+00].

Bild 5.3 zeigt einige Mikroprozessoren im Vergleich [Bro+00]. Hierzu wurden die erhältlichen SpecInt2000-Werte pro eingesetzter elektrischer Leistung – bezogen auf den ältesten (und schlechtesten) Sparc-III-Prozessor– dargestellt, und zwar als $(\text{SPEC})^x/\text{W}$ mit $x = 1 \ldots 3$. Die unterschiedliche Metrik war bereits in den Darstellungen aus Abschnitt 5.1 sichtbar: Ist nun P*T konstant oder $P*T^3$?

Diese Unterschiede sind in der unterschiedlichen Mikroarchitektur begründet, manchmal auch darin, dass viel Kompatibilität mitgeschleppt wird. Bild 5.7 zeigt allerdings nur die Hälfte der Wahrheit, indem kommerzielle Mikroprozessorprodukte miteinander verglichen werden.

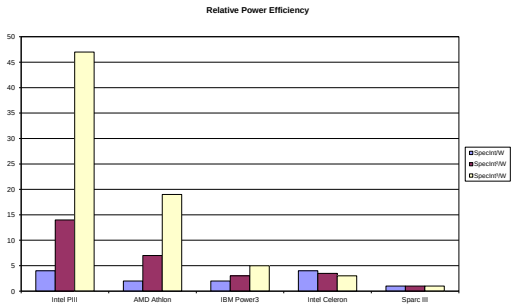


Bild 5.3 Relative Leistungseffizienz im Vergleich

In [MP01] werden zwei Produkte etwa gleichen Erscheinungsdatums miteinander verglichen: Ein AMD Mobile K6 und ein Intel Xscale-Mikrocontroller, der von der ARM (Advanced RISC Machine) StrongARM-Architektur abgeleitet wurde. Der AMD Mobile K6 benötigt bei 400 MHz eine elektrische Leistung von 12 W, der Xscale bei 600 MHz nur 450 mW! Nimmt man grob an, dass beide etwa gleich schnell arbeiten (aufgrund der Superskalarität im AMD-Prozessor ist dieser bei

gleicher Arbeitsfrequenz schneller), ergibt dies ein Verhältnis der elektrischen Leistung von ca. 1:27!

Welches Fazit kann man hieraus ziehen? Die aktuelle Entwicklung der integrierten Schaltkreise geht mehr in die Richtung Leistungseffizienz, nicht mehr Performance. Dies wurde bereits in Bild 5.2 angedeutet, und derzeit sind große Bemühungen zu verzeichnen, diese Effizienz noch zu steigern.

Dies betrifft das Hardwaredesign, und der Systemdesigner kann als Anwender nur die geeignete Architektur auswählen. Ist die Leistungsbilanz bei einem Design im Vordergrund stehend oder auch nur eine wesentliche Randbedingung, sollte man mit der Auswahl des Mikroprozessors/Mikrocontrollers anhand der Daten beginnen und alle anderen Werte wie Betriebsfrequenz usw. als nachrangig betrachten.

5.2.2 Codierung von Programmen in besonders energiesparender Form

Vor einigen Jahren war ein Thema wie energiesparende Software undenkbar, mittlerweile hat es sich jedoch schon etabliert [SWM01]: Man kann die spezifische Leistungsaufnahme pro Befehl bestimmen und dann auswählen, welcher tatsächlich ausgeführt werden soll – falls es Variationsmöglichkeiten gibt. Kandidaten hierfür sind z.B. Multiplikationsbefehle und deren Übersetzung in eine Reihe von Additionsbefehlen.

Insbesondere die Multiplikation einer Variablen mit einer Konstanten kann in diesem Beispiel als möglicher Kandidat gelten. Die Multiplikation mit 5 z.B. wird dann auf einen zweifachen Shift nach links (= Multiplikation mit 4) und anschließender Addition mit dem ursprünglichen Wert ausgeführt, wenn dies energetisch günstiger sein sollte (siehe Bild 5.4).

Bild 5.4 Umsetzung einer Multiplikation mit Konstanten in energetisch günstigere Form

Um dies wirklich auszunutzen, muss die Hilfe eines Compilers in Anspruch genommen werden. Derartige Ansätze sind in der Forschung vertreten, z.B. dargestellt in [SWM01]. Es dürfen jedoch keine Größenordnungen an Energieeinsparung dadurch vermutet werden, die Effekte bleiben im Rahmen einiger 10%.

5.2.3 Einrichtung von Warte- und Stoppzuständen oder Optimierung der Betriebsfrequenz?

Eine andere Möglichkeit zur Energieeinsparung entsteht durch die Einführung von verschiedenen Betriebsmodi insbesondere von Mikrocontrollern. Diese Modi, im

Folgenden mit RUN, IDLE und SLEEP bezeichnet, bieten neben variiertem Funktions- und Reaktionsumfang auch differierende Energiebilanzen. Bild 5.5 zeigt ein Beispiel aus [BBM00] für den Intel StrongARM SA-1100 Mikroprozessor.

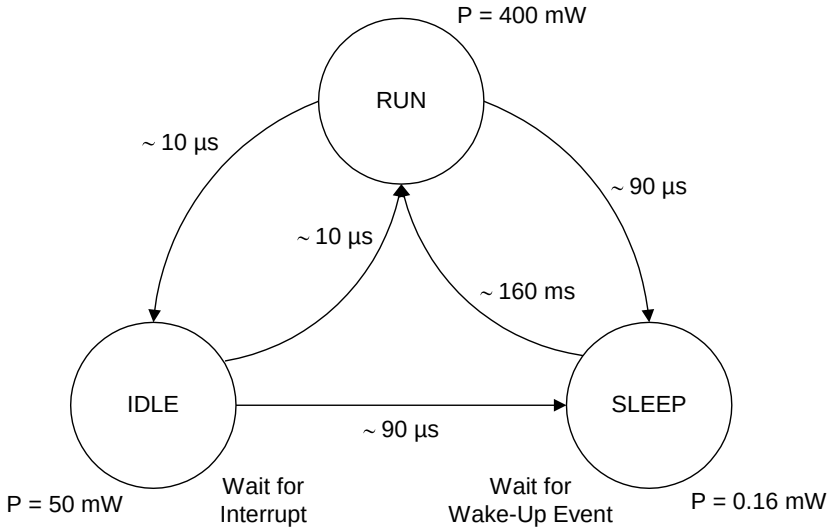


Bild 5.5 Power State Machine für SA-1100

Der Übergang von RUN in IDLE sowie RUN in SLEEP erfolgt üblicherweise durch Software. Hier können spezielle Instruktionen oder das Setzen von Flags zum Einsatz kommen. Im IDLE-Modus ist die Taktversorgung prinzipiell eingeschaltet, insbesondere eine vorhandene PLL, und die Peripherie eines Mikrocontrollers bleibt meist ebenfalls versorgt. Aus diesem Grund können Ereignisse im IRQ-Controller wahrgenommen werden und führen zum Aufwecken des Prozessorkerns.

Im SLEEP-Modus wird die Taktversorgung komplett ausgeschaltet, die PLL ist ausgeschaltet. Dadurch sinkt die Leistungsaufnahme nochmals, auch die peripheren Elemente werden ausgeschaltet. Der Nachteil ist derjenige, dass das Starten des Prozessors/Controllers jetzt recht lange dauert, weil die PLL sich erst wieder einphasen muss. Außerdem können nur noch asynchrone Ereignisse wahrgenommen werden, meist ist dies ein singuläres Ereignis, z.B. der Non-Maskable Interrupt (NMI) oder der Reset.

Die eigentliche Schwierigkeit mit der Power-State-Machine besteht darin, Kriterien zu finden, wann in welchen Zustand übergegangen werden kann. Man denke dabei nur an die verschiedenen Energiesparmodi bekannter Rechner. In Bild 5.5 ist es so, dass der Übergang nur Zeit, keine Leistung kostet. Dies kann im allgemeinen Fall

jedoch anders sein, und ein verkehrtes Abschalten könnte sogar zu erhöhter Verlustleistung führen.

Zurzeit sucht man nach neuen Methoden, die die Übergänge definieren. Für den Systementwickler stellt dies natürlich eine gute Methode dar, unter der allerdings die Echtzeitfähigkeit leiden dürfte. Meist sind jedoch Echtzeitsysteme nicht unbedingt batteriebetrieben, energiesparend sollten sie jedoch trotzdem sein.

Die andere Methode wäre diejenige, auf die Power-State-Machine zu verzichten und die Betriebsfrequenz an das untere Limit zu fahren. Den Netteffekt erfährt man für einen Vergleich nur durch intensive Simulationen, und auch hier dürfte die Echtzeitfähigkeit ggf. leiden.

5.2.4 Neue Ansätze zur Mikroprozessor-Architektur: Clock-Domains und GALS-Architektur

Eine optimale Lösung in Richtung minimaler Energieumsatz bei der Programmausführung wäre es, wenn Betriebsspannung und –frequenz den aktuellen Anforderungen angepasst werden können. In [TM05] wird ein derartiger Ansatz diskutiert, und zwar in einer vergleichsweise feinkörnigen Form.

Die Idee zielt eigentlich auf das Design superskalarer Prozessoren [Sie04]. Diese Prozessoren, die in der Regel sehr groß und damit auch auf der Siliziumfläche ausgedehnt sind, haben besondere Probleme mit einer gleichmäßigen Taktverteilung (ohne Skew), die entweder sehr viel Verlustleistung oder eine Verlangsamung mit sich bringt. Der in [TM05] vorgestellte Ansatz zeigt nun, dass synchrone Inseln, asynchron untereinander verbunden, die bzw. eine Lösung hierfür darstellen.

Diese Architektur wird GALS, Globally Asynchronous Locally Synchronous, genannt. Die lokalen Inseln werden jeweils mit einem Takt (Clock Domain) versorgt, der nun sehr genau an den aktuellen Rechenbedarf angepasst werden kann (Hardware: VCO, Voltage Controlled Oscillator mit DVS, Dynamic Voltage Scaling). Wie aber kann man sich die asynchrone Kommunikation vorstellen?

Asynchron ist eigentlich das falsche Wort hierfür, selbst-synchronisierend ist richtig. Hiermit ist gemeint, dass über die Kommunikationsleitungen nicht nur Daten (und ggf. ein Takt) geführt werden, sondern dass mit den Daten ein Handshake verbunden ist. In etwa verläuft dies nach dem Handshake:

1. (S:) Daten sind gültig
2. (E:) Daten sind übernommen
3. (S:) Daten sind nicht mehr gültig
4. (E:) Wieder frei für neue Daten

Hiermit ist grundsätzlich ein Verfahren möglich, wie die Ausführung von Programmen (Energie- bzw. Verlustleistungs-) optimal angepasst werden kann.

Abschnitt II: Software Engineering für Eingebettete Systeme

6 Modellbasierte Entwicklung eingebetteter Systeme

Dieser Abschnitt befindet sich zurzeit in Planung, voraussichtliche Fertigstellung Ende IV/2012.

6.1 Einleitung: Warum modellbasierte Entwicklung?

Unabhängigkeit von der Hardware
Weniger Fehler

6.2 Einführung in die Petrinetze

Basisdefinitionen (Platz, Transition, Kante, Marke)
Erweiterte Definitionen 1 (Kontakt, Test)
Erweiterte Definitionen 2 (Konflikte, Reset, Timer)

6.3 Codegenerierung

Einfaches Beispiel: Ampelschaltung, Blocksteuerung Eisenbahn
Erweiterung der Ampelsteuerung um Timer

6.4 Unified Modelling Language

7 Einführung in die Sprache C

Die nachfolgende Einführung in die Sprache C ist in wesentlichen Teilen [CKURS] entnommen. C stellt eine sehr populäre, imperative Sprache dar, die sich durch folgende Eigenschaften auszeichnet:

- relativ kleiner Sprachkern, kompakte Notation
- reichhaltiger Satz von Standarddatentypen
- reichhaltiger Satz von Operatoren
- Zeiger, Felder, Verbünde für komplexe Datenstrukturen
- gute Abbildung dieser auf Maschinenebene: hohe Effizienz
- alles andere wie E/A, Speicherverwaltung etc. ist in Standard-Bibliothek untergebracht
- wegen Einfachheit und Verbreitung extrem hohe Portabilität

Die immense Flexibilität und Ausdrucksstärke von C birgt aber auch größte Gefahren in der Hand eines unerfahrenen oder leichtfertigen Programmierers: C ist definitiv keine sichere Sprache, daher ist größte Disziplin geboten, um Fehler zu vermeiden. Der Leitsatz von Ritchie beim Entwurf der Sprache lautete: ”Trust the programmer!“.

Allerdings muss man auch sagen, dass bei entsprechender Programmierdisziplin C auch für sichere Software geeignet ist. Im Anschluss an die Einführung ist daher ein Abschnitt zu Codierungsregeln beigefügt, nähere Literatur siehe [Hat95].

Zunächst jedoch zum Begriff der imperativen Sprache. **Imperative Programmierung** ist ein Programmierparadigma. Ein imperatives Programm beschreibt eine Berechnung durch eine Folge von Anweisungen, die den Status des Programms verändern. Im Gegensatz dazu wird in einer deklarativen Sprache bzw.Programm eine Berechnung beschrieben, in der codiert wird, *was* berechnet werden soll, aber nicht *wie*.

Mit anderen Worten: In imperativen Sprachen wie *C* werden die Algorithmen bis ins letzte Detail so beschrieben, wie sie auch auszuführen sind.

Im weiteren Verlauf dieses Kapitels werden die lexikalischen Elemente, die syntaktischen Elemente, der Präprozessor und die Standardbibliothek behandelt. Den Abschluss bilden besondere Kapitel zur Arbeitsweise eines C-Compilers sowie zu Codierungsregeln.

7.1 Lexikalische Elemente

Der Grundzeichensatz für C-Quelltexte umfasst folgende sichtbare Zeichen:

- Großbuchstaben: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Kleinbuchstaben: a b c d e f g h i j k l m n o p q r s t u v w x y z
- Dezimalziffern: 0 1 2 3 4 5 6 7 8 9
- Unterstrich: _
- Interpunktion: ! „ # % & , ( ) * + , - . / : ; < = > ? [\ ] ^ { | } ~ Zusätzlich können folgende Zeichen vorkommen:

Zeichen	Bedeutung	Ersatzdarstellung
Space	Leerzeichen	
BEL	Alarmglocke (bell)	\a
BS	Rückschritt (backspace)	\b
FF	Seitenvorschub (form feed)	\f
NL	Zeilenvorschub (newline)	\n
CR	Wagenrücklauf (carriage return)	\r
HT	Horizontaltabulator (horizontal tab)	\t
VT	Vertikaltabulator (vertical tab)	\v

Es gibt auch Ersatzdarstellungen für die Anführungszeichen und zwei weitere Sonderzeichen zur Verwendung in Zeichen- und Zeichenkettenkonstanten. Hier dient der Rückschrägstrich dazu, die Sonderbedeutung des betr. Zeichens zu unterdrücken: \", \', \?, \\. Um alle Zeichen des Zeichensatzes der Maschine darstellen zu können, gibt es ferner so genannte numerische Escape-Sequenzen (Ersatzdarstellungen):

- \d, oder \dd oder \ddd d (1...3) ist Oktalziffer (oft gebraucht: ‚\0', die Null)
- \xh oder \xhh oder . . . h (beliebige Anzahl) ist Hexadezimalziffer (0 bis 9, A bis F oder a bis f)

In Zeichen- und Zeichenkettenkonstanten (auch Literale genannt) können alle Zeichen des verwendeten Systems vorkommen.

7.1.1 White Space (Leerraum)

Als Leerraum (white space) gelten Leerzeichen, Zeilenvorschub, Wagenrücklauf, vertikaler und horizontaler Tabulator, sowie Seitenvorschub. Kommentare gelten auch als Leerraum. Leerraum wird syntaktisch ignoriert, außer in Zeichenketten- oder Zeichenkonstanten; er dient dazu, sonst aneinandergrenzende Wörter, Zeichen etc. zu trennen und den Quelltext für Menschen durch übersichtliche Gestaltung, z.B. Einrückungen nach Kontrollstruktur etc., gut lesbar zu machen.

7.1.2 Kommentare

Kommentare werden durch die Zeichenpaare **/*** und ***/** erzeugt. Alles, was dazwischen steht – auf einer Zeile oder mit beliebig vielen Zeilen dazwischen, gilt als Kommentar. Kommentare dürfen nicht geschachtelt werden.

```
/* Das ist zum Beispiel ein Kommentar
. . . und hier geht er immer noch weiter */.
```

7.1.3 Schlüsselwörter

C hat die folgenden 32 Schlüsselwörter (reserved words, keywords):

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void volatile
while
```

7.1.4 Identifier (Bezeichner)

Bezeichner in C (identifier), sonst auch schlicht Namen genannt, werden folgendermaßen gebildet (als regulärer Ausdruck in Unix-Notation):

[A-Za-z_][A-Za-z_0-9]*

d.h. Buchstabe oder Unterstrich optional gefolgt von beliebiger (auch Null) Folge eben dieser, inklusive der Ziffern.

Bezeichner dürfen nicht mit einer Ziffer beginnen, Groß- und Kleinbuchstaben sind als verschieden zu werten. Bezeichner dürfen nicht aus der Menge der o.g. Schlüsselwörter sein (oder aus der Menge von Namen, die für die Standardbibliothek reserviert sind, sie müssen sich mindestens in den ersten 31 Zeichen unterscheiden. Mit Unterstrich beginnende Namen sind für das System reserviert und sollten nicht verwendet werden. Bezeichner mit externer Bindung (d.h. Weiterverarbeitung durch Linker etc.) können weiteren Beschränkungen unterliegen.

7.1.5 Konstanten

C kennt vier Hauptgruppen von Konstanten:

- Ganzzahlkonstanten Dezimal-, Oktal- oder Hex-Darstellung
- Gleitpunktzahlkonstanten mit Dezimalpunkt und/oder Exponentkennung
- Zeichenkonstanten eingeschlossen in ‚. . . ‚
- Zeichenkettenkonstanten eingeschlossen in „. . . „

Numerische Konstanten sind immer positiv, ein etwa vorhandenes Vorzeichen gilt als unärer Operator auf der Konstanten und gehört nicht dazu. Ganzzahlkonstanten sind vom Typ int, wenn das nicht ausreicht, vom Typ long, wenn auch das nicht ausreicht, vom Typ unsigned long. Man kann die größeren Typen auch durch Anfügen von Suffixen erzwingen, wie aus der folgenden Tabelle ersichtlich. Beginnt die Ganzzahlkonstante mit 0x oder 0X, so liegt Hexnotation vor und es folgen eine oder mehrere Hexziffern. Dabei stehen A-F bzw. a-f für die Werte 10...15. Beginnt

andernfalls die Ganzzahlkonstante mit einer 0, so liegt Oktalnotation vor und es folgen eine oder mehrere Oktalziffern, andernfalls liegt Dezimalnotation vor. Gleitpunktzahlkonstanten sind immer vom Typ double, falls nicht durch Suffix als float oder long double gekennzeichnet. Zur Erkennung müssen mindestens der Dezimalpunkt oder die Exponentkennung vorhanden sein.

Dezimalziffern	0 1 2 3 4 5 6 7 8 9
Oktalziffern	0 1 2 3 4 5 6 7
Hexziffern	0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
0	Die Konstante 0 (Null)
l L	Ganzzahlsuffix für long (Verwechselungsgefahr l mit 1!)
u U	Ganzzahlsuffix für unsigned
f F l L	Gleitpunktzahlsuffix für float bzw. long double (s.o.)
e E	Gleitpunktzahlkennung für Exponent

Tabelle 7.1 Darstellungen für Konstanten in C

Eine *Zeichenkonstante* (character constant) ist ein in einfache Hochkommata eingeschlossenes Zeichen aus dem Zeichensatz oder seine (auch mehrere Zeichen umfassende) Ersatzdarstellung. Die Betrachtung sog. wide character constants, sowie sog. multi byte character constants unterbleibt hier. Zeichenkonstanten sind vom Typ int, dürfen aber nicht wertmäßig größer als der entspr. Typ char sein.

Eine *Zeichenkettenkonstante* (string constant) ist eine in sog. doppelte Anführungszeichen eingeschlossene Zeichenkette auf einer Zeile. Sie darf alle Zeichen des Zeichensatzes, incl. etwaiger Ersatzdarstellungen, und (dann signifikanten) Leerraum enthalten. Nur durch Leerraum getrennte Zeichenketten werden vom Präprozessor zusammengefügt und gelten als eine Zeichenkette. Man kann eine Zeile auch umbrechen, indem man sie mit einem Rückschrägstrich terminiert. Die auf diese Weise fortgeführte Zeile gilt dann als eine logische Zeile.

Zeichenketten werden standardgemäß als array of char von niederen zu höheren Adressen mit terminierendem Nullwert im Speicher abgelegt. Ihre Speichergröße ist daher immer um 1 größer als die Größe, die der Anzahl der enthaltenen Zeichen entsprechen würde. Das sind also die allseits verbreiteten so genannten C-Strings. Der Nullwert dient als Terminierungsmarke für alle Routinen der Standardbibliothek und kann folglich im String selbst nicht vorkommen. Der terminierende Nullwert gehört somit nicht zu den Zeichen des Strings und wird folglich bei Ermittlung seiner Länge auch nicht mitgezählt.

Eine Zeichenkette als Typ array of char zu sehen, nimmt man aber nur bei der Initialisierung von Arrays oder der Anwendung des `sizeof`-Operators wahr. Bei den meisten Verwendungen treten jedoch sofort die üblichen syntaktischen Umwand-

lungen von C in Kraft, und man sieht nur noch einen Zeiger auf das erste Zeichen, also den Typ char *, über den man dann alle weitere Verarbeitung steuern kann.

7.2 Syntaktische Elemente

7.2.1 Datentypen

Der Begriff des Datentyps beinhaltet folgendes:

- die Größe und Ausrichtung des belegten Speicherplatzes (size, alignment)
- die interne Darstellung (Bitbelegung)
- die Interpretation und damit den Wertebereich
- die darauf anwendbaren bzw. erlaubten Operationen

C-Typbezeichnung		Gruppe	Klasse	Kategorie	MinBit
char		integer	arithmetic	scalar	8
signed char		integer	arithmetic	scalar	8
unsigned char		integer	arithmetic	scalar	8
short, signed short		integer	arithmetic	scalar	16
unsigned short		integer	arithmetic	scalar	16
int, signed int		integer	arithmetic	scalar	16
unsigned int		integer	arithmetic	scalar	16
long, signed long		integer	arithmetic	scalar	32
unsigned long		integer	arithmetic	scalar	32
enum		integer	arithmetic	scalar	s.d.
float		float	arithmetic	scalar	(32)
double		float	arithmetic	scalar	s.w.u.
long double		float	arithmetic	scalar	s.w.u.
T *	(pointer to T)		pointer	scalar	
T [...]	(array of T)		array	aggregate	
struct {...}			struct	aggregate	
union {...}			union	aggregate	
T (...)	(function returning T)			function	
void				void	

Tabelle 7.2 Übersicht zu den intrinsischen Datentypen in C

ISO-C verfügt über einen reichhaltigen Satz von Datentypen, die sich wie in vorangegangener Übersicht gezeigt organisieren lassen. ISO-C verlangt binäre Codierung der integralen Typen. Für die Wertebereiche aller arithmetischen Typen sind Mindestwerte und Größenverhältnisse festgelegt. Die implementierten Größen dieser Datentypen sind in `limits.h` und `float.h` definiert.

In obiger Tabelle bezeichnet T* einen Zeiger auf den Typ T, T[...] ein Array vom Typ T, T(...) eine Funktion mit Rückgabetyp T. void ist der leere Typ. Als Rückgabetyp einer Funktion deklariert zeigt er an, dass die Funktion nichts zurückgibt, in der Parameterliste, dass sie nichts nimmt. Ein Zeiger auf void ist ein Zeiger auf irgendetwas unbestimmtes, ein generischer Zeiger, den man nie dereferenzieren kann. Variablen oder Arrays vom Typ void können daher nicht deklariert werden. Der Array-Typ T[] und der Funktionstyp T() können nicht Typ einer Funktion sein.

Die Gruppen, Klassen und Kategorien dienen zur Kenntlichmachung der auf diesen Typen und in Verbindung mit diesen Typen erlaubten Operationen. Datentypen können durch die sog. type qualifiers `const` und `volatile` weiter qualifiziert werden. Dabei bedeutet `const`, dass ein so bezeichneter Datentyp nur gelesen werden darf (read only), d.h. er könnte z.B. in einem solchen Speicherbereich oder im ROM abgelegt sein. `volatile` bedeutet, dass die so qualifizierte Größe durch außerhalb des Wirkungsbereichs des Compilers liegende Einflüsse verändert werden könnte, z.B. kann es sich hier um in den Speicherbereich eingeblendete Hardwareregister (sog. Ports) handeln. Dies soll den Compiler davon abhalten, gewisse sonst mögliche Optimierungen des Zugriffs auf die entsprechende Variable vorzunehmen. Beide Qualifizierer können auch zusammen auftreten. Hier einige Beispiele:

```
int i; /* i ist als Variable vom Typ int definiert */
const int ic = 4711; /* ic ist als Konstante vom Typ int
                        definiert */
const int pc; / pc ist Zeiger auf konstanten int */
int const cpi = &i; / cpi ist konstanter Pointer auf int */
const int const cpc = &ic; / konstanter Pointer auf
                              konstanten int */
volatile int vi; /* vi kann durch äußeren Einfluss verändert
                    werden */
const volatile int vci; /* vci ist z.B. ein Timerport */
```

Als `const` vereinbarte Variablen dürfen vom Programm nicht verändert werden. Falls man es versucht, gibt es Fehlermeldungen vom Compiler. Falls man dies jedoch durch in C legale Mittel wie Typumwandlung zu umgehen versucht, kann es je nach System auch zu Laufzeitfehlern führen.

7.2.2 Deklarationen und Definitionen

C ist eine eingeschränkt blockstrukturierte Sprache, d.h. Blöcke sind das strukturelle Gliederungsmittel. Blöcke werden durch die Blockanweisung { ... } erzeugt.

Die Einschränkung ist, dass Funktionsdefinitionen (siehe dort) nur außerhalb von Blöcken möglich sind. Blöcke können beliebig geschachtelt werden. Alles, was außerhalb von Blöcken deklariert oder definiert wird, ist global. Alles, was in einem Block deklariert oder definiert wird, ist lokal zu diesem Block und gilt bis zum Verlassen dieses Blocks. Ein in einem Block deklarierter Name kann einen in einer höheren Ebene deklarierten Namen maskieren, d.h. der äußere Name wird verdeckt und das damit bezeichnete Objekt ist dort nicht mehr zugreifbar.

Der Compiler bearbeitet (man sagt auch liest) den Quelltext (genauer die vom Präprozessor vorverarbeitete Übersetzungseinheit) Zeile für Zeile, von links nach rechts und von oben nach unten. Bezeichner (Identifier, → 7.1.4) müssen grundsätzlich erst eingeführt sein, d.h. deklariert und/oder definiert sein, bevor sie benutzt werden können.

Deklarationen machen dem Compiler Bezeichner (Namen) und ihren Typ bekannt. Sie können auch unvollständig sein, d.h. nur den Namen und seine Zugehörigkeit zu einer bestimmten Klasse bekannt machen, ohne wissen zu müssen, wie der Typ nun genau aussieht. Das reicht dann nicht aus, um dafür Speicherplatz zu reservieren, aber man kann z.B. einen Zeiger auf diesen jetzt noch unvollständigen Typ erzeugen, um ihn dann später, wenn der Typ vollständig bekannt ist, auch zu benutzen. Deklarationen können, abhängig von ihrer Typklasse, auch Definitionen sein. Wenn sie global, d.h. außerhalb von Blöcken erfolgen, sind sie standardmäßig auf den Wert Null initialisiert. Innerhalb eines Blocks ist ihr Wert bei ausbleibender Initialisierung undefiniert. Definitionen haben die Form:

```
Typ Name; oder Typ Name1 ,Name2, . . . ;
```

Definitionen weisen den Compiler an, Speicherplatz bereitzustellen und, wenn das angegeben wird, mit einem bestimmten Wert zu initialisieren. Eine Definition ist gleichzeitig auch eine Deklaration. Eine Definition macht den Typ vollständig bekannt und benutzbar, d.h. es wird Speicherplatz dafür reserviert (im Falle von Datentypen) oder auch Code erzeugt (im Falle von Funktionsdefinitionen, siehe dort).

Definitionen von Datenobjekten mit Initialisierung haben die Form:

```
Typ Name = Wert; oder Typ Name1 = Wert1 ,Name2 = Wert2
, . . . ;
```

7.2.3 Speicherklassen, Sichtbarkeit und Bindung

Außerhalb von Blöcken vereinbarte Objekte gehören zur Speicherklasse `static`. Sie sind vom Start des Programms an vorhanden, und sind global, d.h. im ganzen Programm gültig und sichtbar – sie haben *global scope* und externe Bindung (*external linkage*). Wenn sie nicht im Programm auf bestimmte Werte gesetzt sind, werden sie auf den Wert 0 initialisiert (im Gegensatz zur Speicherklasse auto).

Durch Angabe des Schlüsselworts static kann der Sichtbarkeitsbereich (*scope*) für so vereinbarte Objekte auf die Übersetzungseinheit (Datei) eingeengt werden, das Objekt hat dann interne Bindung (*internal linkage*) und *file scope*.

Deklarationen und Definitionen in Blöcken können nur vor allen Anweisungen (siehe dort) stehen, also zu Beginn eines Blocks. Sie sind lokal zu dem Block, in dem sie erscheinen (*block scope*). Die so vereinbarten Objekte haben die Speicherklasse `auto`, d.h. sie existieren nur, solange der Block aktiv ist und werden bei Eintritt in den Block jedes Mal wieder neu erzeugt, jedoch ohne definierten Anfangswert. Durch Angabe des Schlüsselworts `static` kann die Speicherklasse auf `static` geändert werden, ohne dass Sichtbarkeit und Bindung davon berührt würden. Sie sind von Beginn an vorhanden und behalten ihren Wert auch nach Verlassen des Gültigkeitsbereichs.

Vereinbarungen von Objekten mittels `register` sind nur in Blöcken oder in Parameterlisten von Funktionen erlaubt und dienen lediglich als Hinweis an den Compiler, er möge sie in (schnellen) Prozessorregistern ablegen. Ob das denn auch geschieht, bleibt dem Compiler überlassen. Auf so vereinbarte Objekte darf der Adressoperator & nicht angewandt werden.

Der Sichtbarkeitsbereich einer Marke (label) ist die Funktion, in der sie deklariert ist (*function scope*). Innerhalb einer Funktion weist man bei Vereinbarung eines Namens mit extern darauf hin, dass das Objekt anderweitig definiert ist. Außerhalb von Funktionen gelten alle vereinbarten Objekte defaultmäßig als `extern`.

7.2.4 Operatoren

C verfügt über einen reichhaltigen Satz von Operatoren. Diese lassen sich nach verschiedenen Kategorien gliedern:

- nach der Art: unäre, binäre und ternäre Operatoren
- nach Vorrang – Präzedenz (precedence)
- nach Gruppierung – Assoziativität: links, rechts (associativity)
- nach Stellung: Präfix, Infix, Postfix
- nach Darstellung: einfach, zusammengesetzt

Die Vielfalt und oft mehrfache Ausnutzung der Operatorzeichen auch in anderem syntaktischen Zusammenhang bietet anfangs ein verwirrendes Bild. Der Compiler kann aber immer nach dem Kontext entscheiden, welche der Operatorfunktionen gerade gemeint ist. Zur Erleichterung des Verständnisses der Benutzung und Funktionsweise der Operatoren daher folgend einige Anmerkungen zur Operatortabelle.

`[]` → 7.2.9 Vektoren und Zeiger.

• und `->` → 7.2.9 Vektoren und Zeiger

++, −− (z.B. a++, b--) als Postin- bzw. -dekrement liefern sie den ursprünglichen Wert ihres Operanden und erhöhen bzw. erniedrigen den Wert des Operanden

danach um 1. Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der Seiteneffekt dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines Sequenzpunktes sicher.

++, −− (z.B. ++a, --b) als Präin- bzw. -dekrement erhöhen bzw. erniedrigen sie erst den Wert ihres Operanden um 1 und liefern dann den neuen, so erhöhten Wert.

C-Bezeichnung	Erläuterung d. Funktion	Klasse	Vorrang	Gruppierung
[]	Indexoperator	postfix	16	links
()	Funktionsaufruf	postfix	16	links
.	direkte Komponentenwahl	postfix	16	links
->	indir. Komponentenwahl	postfix	16	links
++ --	Postinkrement, -dekrement	postfix	16	links
++ --	Präinkrement, -dekrement	präfix	15	rechts
sizeof	Größe ermitteln	unär	15	rechts
~	bitweise Negation	unär	15	rechts
!	logische Negation	unär	15	rechts
- +	arithm. Negation, plus	unär	15	rechts
&	Adresse von	unär	15	rechts
*	Indirektion	unär	15	rechts
(type name)	Typumwandlung (cast)	unär	14	rechts
* / %	mult., div., mod.	binär	13	links
+ -	Addition, Subtraktion	binär	12	links
<< >>	bitweise schieben	binär	11	links
< > <= >=	Vergleiche	binär	10	links
== !=	gleich, ungleich	binär	9	links
&	bitweises AND	binär	8	links
^	bitweises XOR	binär	7	links
|	bitweises OR	binär	6	links
&&	logisches AND	binär	5	links
||	logisches OR	binär	4	links
? :	Bedingungsoperator	ternär	3	rechts
=	Zuweisung	binär	2	rechts
+= -= *= /= %=	Verbundzuweisung	binär	2	rechts
<<= >>= &= ^= |=	Verbundzuweisung	binär	2	rechts
,	Sequenzoperator	binär	1	links

Tabelle 7.3 Operatoren in C

Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der *Seiteneffekt* dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines Sequenzpunktes sicher.

`sizeof` dieser Operator arbeitet zur Compilierungszeit (sog. *compile time operator*) und liefert die Größe seines Operanden in Einheiten des Typs `char`: `sizeof(char)` == 1. Der Operand kann ein Typ sein, dann muss er in () stehen, oder ein Objekt im Speicher, dann sind keine Klammern erforderlich. Ist der Operand ein Arrayname, liefert er die Größe des Arrays in `char`-Einheiten.

~ (z.B. ~a): die Tilde liefert den Wert der bitweisen Negation (das Komplement) der Bitbelegung ihres Operanden, der vom integralen Typ sein muss.

! (z.B. !a): liefert die logische Negation des Wertes seines Operanden, der vom skalaren Typ sein muss. War der Wert 0, ist das Ergebnis 1, war der Wert ungleich 0 , ist das Ergebnis 0.

−, + (z.B. -a): die unäre Negation liefert den negierten Wert ihres Operanden, der vom arithmetischen Typ sein muss. Das unäre Plus wurde nur aus Symmetriegründen eingeführt, und dient evtl. lediglich Dokumentationszwecken.

& (z.B. &a) _ liefert die Adresse eines Objektes im Speicher (und erzeugt somit einen Zeigerausdruck, → 5.2.9).

• (z.B. *a): in einer Deklaration erzeugt dieser Operator einen Zeiger auf den deklarierten Typ, in der Anwendung auf einen Zeigerwert, liefert er den Wert des so bezeigten Objekts (→ 5.2.9).

(typename): *typename* ist ein Typbezeichner. Der sog. *type cast operator* liefert den in diesen Typ konvertierten Wert seines Operanden. Dabei wird versucht, den Wert zu erhalten. Eine (unvermeidbare, beabsichtigte) Wertänderung tritt ein, wenn der Wert des Operanden im Zieltyp nicht darstellbar ist, ähnlich einer Zuweisung an ein Objekt dieses Typs. Im Folgenden einige Hinweise zu erlaubten Konversionen:

- Jeder arithmetische Typ in jeden arithmetischen Typ.
- Jeder Zeiger auf void in jeden Objektzeigertyp.
- Jeder Objektzeigertyp in Zeiger auf void.
- Jeder Zeiger auf ein Objekt oder void in einen Integertyp.
- Jeder Integertyp in einen Zeiger auf ein Objekt oder void.
- Jeder Funktionszeiger in einen anderen Funktionszeiger.
- Jeder Funktionszeiger in einen Integertyp.
- Jeder Integertyp in einen Funktionszeiger.

Die Zuweisung von `void`-Zeiger an Objektzeiger und umgekehrt geht übrigens auch ohne den Typkonversionsoperator. In allen anderen Fällen ist seine Anwendung geboten oder erforderlich, und sei es nur, um den Warnungen des Compilers zu entgehen.

`%` (z.B. a%b): modulo liefert den ganzzahligen Divisionsrest des Wertes seines linken Operanden geteilt durch den Wert seines rechten Operanden und lässt sich nur auf integrale Typen anwenden. Dabei sollte man Überlauf und die Division durch Null vermeiden. Bei positiven Operanden wird der Quotient nach 0 abgeschnitten. Falls negative Operanden beteiligt sind, ist das Ergebnis implementationsabhängig. Es gilt jedoch immer: X = (X/Y) * Y + (X%Y).

Die übrigen arithmetischen Binäroperatoren können nur auf Operandenpaare vom arithmetischen Typ angewandt werden, dabei geht, wie üblich und auch aus der Tabelle zu ersehen, Punktrechnung vor Strichrechnung. Bei der Ganzzahldivision wird ein positiver Quotient nach 0 abgeschnitten. Man vermeide auch hier die Null als Divisor. Wenn unterschiedliche Typen an der Operation beteiligt sind, wird selbstständig in den größeren der beteiligten Typen umgewandelt (balanciert).

<<, >> (z.B. a<<b): die Bitschiebeoperatoren schieben den Wert des linken Operanden um Bitpositionen des Wertes des rechten Operanden jeweils nach links bzw. rechts und können nur auf integrale Operandenpaare angewandt werden. Für eine n-Bit-Darstellung des promovierten linken Operanden muss der Wert des rechten Operanden im Intervall 0..n-1 liegen. Bei positivem linken Operanden werden Nullen in die freigewordenen Positionen nachgeschoben. Ob bei negativem linken Operanden beim Rechtsschieben das Vorzeichen nachgeschoben wird (meist so gehandhabt), oder Nullen, ist implementationsabhängig.

Die Vergleichsoperatoren (z.B. `a == b`) können nur auf arithmetische und auf Paare von Zeigern gleichen Typs angewandt werden. Sie liefern den Wert 1, wenn der Vergleich erfolgreich war, sonst 0.

Die bitlogischen Operatoren (z.B. `a&b`) können nur auf integrale Typen angewandt werden und liefern den Wert der bitlogischen Verknüpfung des Wertes des linken mit dem Wert des rechten Operanden (beide als Bitmuster interpretiert).

`&&` (z.B. `a && b`): testet, ob beide Operanden ungleich Null (wahr) sind. Ist der linke Operand wahr, wird auch der rechte getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen UND-Verknüpfung ja schon feststeht (sog. Kurzschlussbewertung, *short circuit evaluation*, mit Sequenzpunkt nach dem linken Operanden). Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

`||` (z.B. `a || b`): testet, ob mindestens einer der beiden Operanden ungleich Null (wahr) ist. Ist der linke Operand gleich Null (falsch), wird auch der rechte

getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen ODER-Verknüpfung ja schon feststeht (sog. Kurzschlussbewertung, short circuit evaluation, wie oben) Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

`X?Y:Z` X muss vom skalaren Typ sein und wird bewertet. Ist X ungleich Null (wahr), wird Y bewertet, andernfalls wird Z bewertet. Y und Z können fast beliebige Ausdrücke sein, auch `void` ist möglich, sollten aber kompatibel sein. Zwischen der Bewertung von X und der Bewertung von entweder Y oder Z befindet sich ein Sequenzpunkt (sequence point). Der Wert des Ausdrucks ist dann der Wert des (evtl. im Typ balancierten) Wertes des zuletzt bewerteten Ausdrucks.

= Der Zuweisungsoperator bewertet seine beiden Operanden von rechts nach links, so sind auch Zuweisungsketten in der Art von a = b = c = d = 4711 möglich. Der Wert des Zuweisungsausdrucks ist der Wert des Zugewiesenen, der in den Typ des linken Operanden transformierte Wert des rechten Operanden. Der linke Operand muss ein Objekt im Speicher darstellen, auf das schreibend zugegriffen werden kann. Aufgrund der speziellen Eigenheit von C, dass die Zuweisung ein Ausdruck und keine Anweisung ist, sowie seiner einfachen Wahr-Falsch-Logik, taucht die Zuweisung oft als Testausdruck zur Schleifenkontrolle auf. Ein markantes Beispiel:

```
while (*s++ = t++); / C-Idiom für Zeichenketten
                      kopie */
```

Die Verbund- oder Kombinationszuweiser bestehen aus zwei Zeichen, deren rechtes der Zuweiser ist. Sie führen, kombiniert mit der Zuweisung verschiedene arithmetische, bitschiebende und bitlogische Operationen aus. Dabei bedeutet a op= b soviel wie a = a op b, mit dem Unterschied, dass a, also der linke Operand, nur einmal bewertet wird.

Der Komma- oder Sequenzoperator (z.B. a,b) gruppiert wieder von links nach rechts und bewertet erst seinen linken, dann seinen rechten Operanden. Dazwischen liegt ein Sequenzpunkt, das heißt, alle Seiteneffekte sind garantiert eingetreten. Der Wert des Ausdrucks ist das Resultat der Bewertung des rechten Operanden. Der Nutzen des Operators besteht darin, dass er einen Ausdruck erzeugt und folglich überall stehen kann, wo ein Ausdruck gebraucht wird. Seine Hauptanwendungen sind die Initialisierungs- und Reinitialisierungsausdrücke in der Kontrollstruktur der `for`-Schleife, wo ja jeweils nur ein Ausdruck erlaubt ist, und manchmal mehrere gebraucht werden.

Einige Operationen erzeugen implementationsabhängige Typen, die in `stddef.h` definiert sind. `size_t` ist der vom `sizeof`-Operator erzeugte vorzeichenlose integrale Typ. `ptrdiff_t` ist der vorzeichenbehaftete integrale Typ, der vom Sub-

traktionsoperator erzeugt wird, wenn dieser auf Zeiger (gleichen Typs!) angewandt wird.

7.2.5 Ausdrücke

C ist eine Ausdrucks-orientierte Sprache. Der Compiler betrachtet die Ausdrücke und bewertet sie. Ein Ausdruck (*expression*) in C ist:

- eine Konstante (constant)
- eine Variable (variable)
- ein Funktionsaufruf (function call)
- eine beliebige Kombination der obigen 3 Elemente mittels Operatoren

Jeder Ausdruck hat einen Typ und einen Wert. Bei der Bewertung von Ausdrücken gelten folgende Regeln: Daten vom Typ `char` oder `short` werden sofort in den Typ `int` umgewandelt (*integral promotion*). Bei der Kombination von Ausdrücken wird balanciert, d.h. der dem Wertebereich oder Speicherplatz nach kleinere Typ wird in den beteiligten, dem Wertebereich oder Speicherplatz nach größeren Typ umgewandelt. Dabei wird versucht, den Wert zu erhalten (*value preservation*).

Die Bewertung der einzelnen Elemente von Ausdrücken folgt Vorrang und Assoziativität der Operatoren. Bei Gleichheit in diesen Eigenschaften ist die Reihenfolge der Bewertung (*order of evaluation*) gleichwohl bis auf wenige Ausnahmen undefiniert, denn der Compiler darf sie auf für ihn günstige Weise ändern, wenn das Ergebnis aufgrund der üblichen mathematischen Regeln gleichwertig wäre. In der Theorie gilt (a * b)/c = (a/c) * b, also darf der Compiler das nach seinem Gusto umordnen, und auch Gruppierungsklammern können ihn nicht daran hindern. Das kann aber bei den Darstellungs-begrenzten Datentypen im Computer schon zu unerwünschtem Überlauf etc. führen.

Soll dies wirklich verhindert werden, d.h., soll der Compiler gezwungen werden, eine bestimmte Reihenfolge einzuhalten, muss die entsprechende Rechnung aufgebrochen und in mehreren Teilen implementiert werden. Die Codesequenzen

```
x = (a * b) / c;
```

und

```
x = a * b;
x = x / c;
```

bewirken tatsächlich nicht automatisch das gleiche, denn im ersten Fall darf der Compiler umsortieren, im zweiten nicht, da das Semikolon einen so genannten Sequenzpunkt (*sequence point*) darstellt, den der Compiler nicht entfernen darf.

Manche Operatoren bewirken sog. Seiteneffekte (side effects), d.h. sie können den Zustand des Rechners verändern, z.B. den Wert von Speichervariablen oder Registern oder sonstiger Peripherie. Dazu gehören neben den Zuweisern auch die Post- und Präinkrement und -dekrement-Operatoren und Funktionsaufrufe. Das Eintreten

der Wirkung dieser Seiteneffekte sollte niemals abhängig von der Reihenfolge der Bewertung sein! Während durch Komma separierte Deklarations- und Definitionslisten strikt von links nach rechts abgearbeitet und bewertet werden, gilt das z.B. für die Reihenfolge der Bewertung in Parameterlisten beim Funktionsaufruf nicht.

7.2.6 Anweisungen

In C gibt es folgende Anweisungen (*statements*):

- Leeranweisung `;` (*empty statement*)
- Audrucksanweisung *expression*`;` (*expression statement*)
- Blockanweisung `{ ... }` (*block statement*)
- markierte Anweisung *label*`:` *statement* (*labeled statement*)
- Auswahlanweisung `if else switch ... case` (*selection statement*)
- Wiederholungsanweisung `for while do ... while` (*iteration statement*)
- Sprunganweisung `goto break continue return` (*jump statement*)

7.2.7 Kontrollstrukturen

Kontrollstrukturen definieren den Ablauf eines Programms. Die einfachste Kontrollstruktur ist die *Sequenz*, d.h. Folge. Der Compiler liest den Quelltext von links nach rechts, von oben nach unten, und setzt ihn in Code um, der eine sequentielle Abarbeitung bewirkt. Um dies zu erzeugen, schreibt man also eine Anweisung nach der andern, von links nach rechts, bzw. besser von oben nach unten, hin.

Die nächste Kontrollstruktur ist die *Auswahl*. C kennt die zwei Auswahl- oder Verzweigungskontrollstrukturen `if else` und `switch case`. Das `if`-Konstrukt hat folgende allgemeine Form:

```
if (expression) /* expression muss vom arithme
                   tischen oder Zeigertyp sein */
   statement1   /* wenn expression ungleich 0,
                   statement1 ausführen */
else
statement2    /* sonst statement2 ausführen */
```

Der `else`-Teil ist optional. Man beachte, dass es in C kein *then* und keine Endmarke (*endif* o.ä.) für diese Konstruktion gibt. Ebenso ist jeweils nur ein *statement* erlaubt; braucht man mehrere, so muss man zum `{`*block statement*`}` greifen. Falls man mehrere geschachtelte `if`-Strukturen verwendet, ordnet der Compiler das `else` immer dem jeweilig direkt vorausgehenden `if` zu, so dass man durch Verwendung von Blockklammern `{}` für die korrekte Gliederung sor-

gen muss, die visuelle Gestaltung des Quelltexts ist nur eine Lesehilfe und hat für die Syntax der Sprache C keine Bedeutung.

Das zweite Auswahlkonstrukt, `switch case`, hat viele Formen, am häufigsten gebraucht wird die folgende allgemeine Form:

```
switch (integral expression) {
  case constintexpr1 : /* Der : ist die Syntaxken
                          nung für eine Marke. */
statement1
statement2
    break; /* hier wird der switch in diesem Fall
              verlassen. */
case constintexpr2 :
statement3
    statement4  /* break fehlt: Es geht weiter zum
                   nächsten Fall! */
default:
statement5
}
```

Die Ausdrücke in den `case`-Marken müssen konstante integrale Ausdrücke sein. Mehrere Marken sind erlaubt. Für den kontrollierenden Ausdruck findet Integer-Erweiterung statt und die `case`-Konstanten werden in den so erweiterten Typ umgewandelt. Danach dürfen keine zwei Konstanten den gleichen Wert haben. Die `default`-Marke darf pro `switch` nur einmal vorhanden sein; sie deckt alles ab, was von den anderen Marken nicht erfasst wird und darf an beliebiger Stelle erscheinen.

Das Problem des `switch`-Konstrukts ist die `break`-Anweisung: fehlt sie, geht die Abarbeitung über die nächste Marke hinweg einfach weiter (sog. *fall through*). Dies kann man natürlich geschickt ausnutzen, ein fehlendes – vergessenes – `break` hat jedoch oft schon zu den seltsamsten Überraschungen geführt. Es ist daher zu empfehlen, einen beabsichtigten Fall von *fall through* durch entspr. Kommentar besonders kenntlich zu machen.

Die nächste wichtige Kontrollstruktur ist die Wiederholung, auch Schleife genannt. Hier hält C drei verschiedene Konstrukte bereit:

```
while (expression) /*solange expression ungleich 0 */
statement       /* statement ausführen */
```

expression muss vom arithmetischen oder Zeigertyp sein und wird bewertet. Falls nicht 0, wird *statement* ausgeführt; dies wird solange wiederholt, bis *expression* zu 0 bewertet wird. Dies ist eine sog. kopfgesteuerte Schleife. Soll das `while`-Kon-

strukt mehrere Anweisungen kontrollieren, greift man üblicherweise zur Blockanweisung.

```
do
   statement          /* statement ausführen */
while (expression); /* solange bis expression zu 0
                       bewertet wird */
```

statement wird ausgeführt, dann wird *expression* bewertet. *expression* muss wie oben vom arithmetischen oder Zeigertyp sein. Falls nicht 0, wird dies solange wiederholt, bis *expression* zu 0 bewertet wird. Man beachte das syntaktisch notwendige Semikolon am Schluss des Konstrukts. Dies ist eine sog. fussgesteuerte Schleife. Für mehrere zu kontrollierende Anweisungen gilt das gleiche wie oben.

```
for (expression1; expression2; expression3)
statement
```

Jeder der drei Ausdrücke in der Klammer des `for`-Konstrukts darf auch fehlen, die beiden Semikola sind jedoch syntaktisch notwendig. Zu Beginn wird einmalig *expression1* bewertet, ihr Typ unterliegt keiner Einschränkung. Sind mehrere Ausdrücke erforderlich, ist dies der Platz für den Einsatz des Sequenzoperators (,). Hier erfolgt daher meist die Initialisierung der Schleife. Als nächstes wird *expression2* bewertet, sie muss vom arithmetischen oder Zeigertyp sein. Ist der Wert ungleich 0, so wird *statement* ausgeführt. Alsdann wird *expression3* bewertet, ihr Typ unterliegt keiner Einschränkung. Hier erfolgt meist die Reinitialisierung der Schleife. Dann wird wieder *expression2* bewertet. Der Zyklus wird solange wiederholt, bis die Bewertung von *expression2* 0 ergibt. Fehlt *expression2*, wird dieser Fall als ungleich 0 bewertet.

Die `for`-Schleife ist gut lesbar und übersichtlich, da Initialisierung, Test und Reinitialisierung dicht beieinander und sofort im Blickfeld sind, bevor man überhaupt mit der Betrachtung des Schleifenkörpers beginnt. Sie ist daher sehr beliebt.

So genannte Endlosschleifen formuliert man in C folgendermaßen:

```
for (;;) statement
```

oder

```
while (1) statement
```

Den letzten Teil der Kontrollstrukturen bilden die sog. Sprunganweisungen:

`goto label;` springt zu einer Marke in der umgebenden Funktion. Diese Anweisung findet in der strukturierten Programmierung keine Verwendung und wird auch im Systembereich nur selten gebraucht. Sie ist jedoch nützlich in der (nicht für menschliche Leser bestimmten) maschinellen Codegenerierung.

`break;` darf nur in `switch` oder in Wiederholungsanweisungen stehen und bricht aus der es umgebenden Anweisung aus.

`continue;` darf nur in Wiederholungsanweisungen stehen und setzt die es umgebende Anweisung am Punkte der Wiederholung fort.

`return expressionopt ;` kehrt aus einer umgebenden Funktion mit der optionalen *expression* als Rückgabewert zurück.

7.2.8 Funktionen

Funktionen sind das Hauptgliederungsmittel eines Programms. Jedes gültige C-Programm muss eine bestimmte Funktion enthalten, nämlich die Funktion `main()`.

Funktionen in C erfüllen die Aufgaben, die in anderen Programmiersprachen *function*, *procedure* oder *subroutine* genannt werden. Sie dienen dazu, die Aufgaben des Programms in kleinere, übersichtliche Einheiten mit klaren und wohldefinierten Schnittstellen zu unterteilen. Funktionsdeklarationen haben die allgemeine Form:

```
Typ Funktionsname(Parameterliste);
```

Wird Typ nicht angegeben, so wird `int` angenommen, man sollte dies aber unbedingt vermeiden. Ist die Parameterliste leer, kann die Funktion eine unspezifizierte Anzahl (auch Null) Parameter unspezifizierten Typs nehmen. Besteht die Parameterliste nur aus dem Schlüsselwort `void`, nimmt die Funktion keine Parameter. Andernfalls enthält die Parameterliste einen oder mehrere Typnamen, optional gefolgt von Variablennamen, als durch Komma separierte Liste.

Als letzter (von mindestens zwei) Parametern ist als Besonderheit auch die Ellipse (…) erlaubt und bedeutet dann eine variable Anzahl sonst unspezifizierter Parameter.

Die Variablennamen haben für den Compiler keine Bedeutung, können aber dem Programmierer als Hinweis auf die beabsichtigte Verwendung dienen, im Sinne einer besseren Dokumentation. Zum Beispiel sind diese beiden Deklarationen,

```
int myfunc(int length, int count, double factor);
```

oder

```
int myfunc(int, int, double);
```

auch *Funktionsprototypen* genannt, für den Compiler identisch. Die Typangaben der Parameterliste, ihre Anzahl und Reihenfolge – auch Signatur (*signature*) genannt – dienen dem Compiler zur Fehlerdiagnose beim Aufruf, d.h. der Benutzung der Funktion. Deshalb sollten Funktionsdeklarationen – die Prototypen – der Benutzung der Funktionen – dem Funktionsaufruf (*function call*) – immer vorausgehen.

> Anmerkung: Da die Variablennamen in der Wahl frei sind (wobei sie natürlich den syntaktischen Bedingungen genügen müssen), können sie auch im Namen den Variablentyp in codierter Form mitführen. So erweist sich die Deklaration
>
> ```
> int iMyfunc(int iLength, int iCount, double dfFactor);
> ```

als besonders gut lesbar in dem Sinn, dass mitten im Text aus dem Variablennamen auch auf den Typ geschlossen werden kann.

Funktionsdefinitionen haben die allgemeine Form:

```
Typ Funktionsname(Parameterliste)
{
Deklarationen und Definitionen
Anweisungen
}
```

Funktionen können nur außerhalb von Blöcken definiert werden. Eine Funktionsdefinition ist immer auch gleichzeitig eine Deklaration. Der Hauptblock einer Funktion, auch Funktionskörper genannt, ist der einzige Ort, wo Code (im Sinne von ausführbaren Prozessorbefehlen) erzeugt werden kann.

Typ und Signatur einer Funktion müssen mit etwaigen vorausgegangenen Prototypen übereinstimmen, sonst gibt es Fehlermeldungen vom Compiler. Beim Funktionsaufruf (*function call*) schreibt man lediglich den Namen der Funktion, gefolgt von den in Klammern gesetzten Argumenten, oft auch aktuelle Parameter genannt, als durch Komma separierte Liste. Die Argumente nehmen den Platz der formalen Parameter ein und werden, da dies ein Zuweisungskontext ist, im Typ angeglichen. Die Reihenfolge der Bewertung dieser Argumentzuweisung ist dabei nicht festgelegt – es ist nur sichergestellt, dass alle Argumente bewertet sind, bevor der eigentliche Aufruf, d.h. der Sprung zum Code des Funktionskörpers erfolgt.

Falls die Funktion ein Ergebnis liefert, den sog. Funktionswert, kann man dieses zuweisen oder weiter verarbeiten, muss es aber nicht (wenn man z.B. nur an dem Seiteneffekt interessiert ist). Ein Beispiel dazu:

```
len = strlen(„hello, world\n"); /* Funktionswert
                                   zuweisen */
printf(„hello, world\n"); /* kein Interesse am
                             Funktionswert */
```

Ein Funktionsaufruf stellt einen Ausdruck dar und darf überall stehen, wo ein Ausdruck des Funktionstyps stehen kann. Eine `void`-Funktion hat definitionsgemäß keinen Funktionswert und ihr Aufruf darf daher nur in einem für diesen Fall zulässigen Zusammenhang erscheinen (z.B. nicht in Zuweisungen, Tests etc.).

Die Ausführung des Funktionskörpers endet mit einer return-Anweisung mit einem entspr. Ausdruck, dies ist wieder als Zuweisungskontext zu betrachten, und es wird in den Typ der Funktion konvertiert. Eine `void`-Funktion endet mit einer ausdruckslosen `return`-Anweisung oder implizit an der endenden Klammer des Funktionsblocks.

Die Funktion main()

Die Funktion `main()` spielt eine besondere Rolle in der Sprache C. Ihre Form ist vom System vordefiniert, sie wird im Programm nicht aufgerufen, denn sie stellt das Programm selbst dar. Die Funktion `main()` wird vom Start-Up-Code, der vom Linker dazu gebunden wird, aufgerufen, d.h. das Programm beginnt mit der Ausführung der ersten Anweisung im Funktionskörper von `main()`. Die Funktion main() hat zwei mögliche Formen, mit oder ohne Parameter:

```
int main(void)
{ Körper von main() }
```

oder

```
int main(int argc, char *argv[])
{ Körper von main() }
```

Die erste Form verwendet man, wenn das Programm keine Parameter nimmt, die zweite, wenn Parameter auf der Kommandozeile übergeben werden, die dann im Programm ausgewertet werden sollen.

Im zweiten Fall – `argc` (argument count) und `argv` (argument vector) sind hierbei lediglich traditionelle Namen – bedeutet der erste Parameter die Anzahl der Argumente, incl. des Programmnamens selbst, und ist daher immer mindestens 1. Der zweite Parameter, also `argv`, ist ein Zeiger auf ein Array von Zeigern auf nullterminierte C-Strings, die die Kommandozeilenparameter darstellen. Dieses Array ist selbst auch nullterminiert, also ist `argv[argc]==0`, der sog. Nullzeiger. Der erste Parameter, `argv[0]`, zeigt traditionell auf den Namen, unter dem das Programm aufgerufen wurde. Falls dieser nicht zur Verfügung steht, zeigt `argv[0]` auf den Leerstring, d.h. `argv[0][0]` ist ‚\0'. Die in `argc` und `argv` gespeicherten Werte, sowie der Inhalt der damit designierten Zeichenketten können vom Programm gelesen und dürfen, wenn gewünscht, auch verändert werden.

Vor Beginn der Ausführung von `main()` sorgt das System dafür, das alle statischen Objekte ihre im Programm vorgesehenen Werte enthalten. Ferner werden zur Interaktion mit der Umgebung drei Dateien geöffnet:

- `stdin` standard input Standardeingabestrom (meist Tastatur)
- `stdout` standard output Standardausgabestrom (meist Bildschirm)
- `stderr` standard error Standardfehlerstrom (meist Bildschirm)

Diese Standardkanäle haben den Datentyp `FILE*` und sind definiert im Header `stdio.h`. In beiden möglichen Formen ist `main()` als `int`-Funktion spezifiziert, der Wert ist die Rückgabe an das aufrufende System und bedeutet den Exit-Status des Programms, der dann z.B. Erfolg oder Misserfolg ausdrücken oder anderweitig vom aufrufenden System ausgewertet werden kann.

Das Programm, bzw. `main()`, endet mit der Ausführung einer return-Anweisung mit einem entspr. Ausdruck, dies ist ein Zuweisungskontext, und es wird in den

Typ von `main()`, d.h. nach `int` konvertiert. `main()` endet auch, wenn irgendwo im Programm, d.h. auch innerhalb einer ganz anderen Funktion, die Funktion `exit()`, definiert in `stdlib.h`, mit einem entspr. Wert aufgerufen wird. Dieser Wert gilt dann als Rückgabewert von `main()`.

Wenn `main()` mit einer ausdruckslosen `return`-Anweisung oder an der schließenden Klammer seines Funktionsblocks endet, ist der Rückgabewert unbestimmt. Bei der Beendigung von `main()` werden erst alle mit `atexit()`, ebenfalls definiert in `stdlib.h`, registrierten Funktionen in umgekehrter Reihenfolge ihrer Registrierung aufgerufen. Sodann werden alle geöffneten Dateien geschlossen, alle mit `tmpfile()` (definiert in `stdio.h`) erzeugten temporären Dateien entfernt und schließlich die Kontrolle an den Aufrufer zurückgegeben.

Das Programm kann auch durch ein – von ihm selbst mit der Funktion `raise()` (definiert in `signal.h`), durch einen Laufzeitfehler (z.B. unbeabsichtigte Division durch Null, illegale Speicherreferenz durch fehlerhaften Zeiger, etc.) oder sonst fremderzeugtes – Signal oder durch den Aufruf der Funktion `abort()` (definiert in `stdlib.h`) terminieren. Was dann im Einzelnen geschieht, wie und ob geöffnete Dateien geschlossen werden, ob temporäre Dateien entfernt werden und was dann der Rückgabestatus des Programms ist, ist implementationsabhängig.

7.2.9 Vektoren und Zeiger

Vektoren (meist *Arrays*, deutsch zuweilen auch Felder genannt) sind als Aggregate komplexe Datentypen, die aus einer Anreihung gleicher Elemente bestehen. Diese Elemente werden aufeinander folgend in Richtung aufsteigender Adressen im Speicher abgelegt, sie können einfache oder selbst auch wieder komplexe Datentypen darstellen. Die Adresse des Arrays ist identisch mit der Adresse des Elements mit der Nummer 0, denn in C werden die Elemente beginnend mit 0 durchnumeriert.

Ein Array wird deklariert mit dem Operator `[]`, in dem die Dimensionsangabe, d.h. die Anzahl der Elemente steht. Die angegebene Anzahl muss eine vorzeichenlose integrale Konstante sein, eine Anzahl 0 ist nicht erlaubt. Der Bezeichner für ein Array ist fest mit seinem Typ verbunden, stellt aber kein Objekt im Speicher dar.

In Zusammenhang mit dem `sizeof`-Operator wird die Größe des Arrays als Anzahl von Einheiten des Typs char geliefert. Der Compiler sorgt für eine korrekte Ausrichtung im Speicher. Ein Beispiel:

```
int iv[10]; /* Ein Array iv von zehn Elementen vom Typ int */
```

Mehr Dimensionen sind möglich, im Gegensatz zu einigen anderen Programmiersprachen gibt es jedoch keine echten mehrdimensionalen Arrays sondern nur Arrays von Arrays (mit beliebiger – vielleicht von der Implementation oder verfügbarem Speicherplatz begrenzter – Anzahl der Dimensionen). Ein Beispiel für die Deklaration eines zweidimensionalen Arrays:

```
double dvv[5][20]; /* Array von 5 Arrays von je 20 doubles */
```

Die Ablage im Speicher erfolgt hierbei zeilenweise (bei zwei Dimensionen), d.h. der der rechte Index (der Spaltenindex bei zwei Dimensionen) variiert am schnellsten, wenn die Elemente gemäß ihrer Reihenfolge im Speicher angesprochen werden. Auf die Elemente zugegriffen wird mit dem Operator `[]`, in dem nun der Index steht, für ein Array mit n Elementen reicht der erlaubte Indexbereich von 0 bis *n*-1.

Beispiel:

```
abc = iv[3]; /* Zuweisung des 4. Elements von iv an abc */
xyz = dvv[i][j]; /* Zuweisung aus dvv, i und j sind
                    Laufvariablen */
```

Die Schrittweite des Index ist so bemessen, dass immer das jeweils nächste – oder vorherige – Element erfasst wird. Es ist erlaubt, negative Indizes oder solche größer als *n*-1 zu verwenden, was jedoch beim Zugriff außerhalb des erlaubten Bereichs des so indizierten Arrays passiert, ist implementationsabhängig (und nicht zu empfehlen!).

Zeiger (*pointer*) sind komplexe Datentypen. Sie beinhalten sowohl die Adresse des Typs, auf den sie zeigen, als auch die Eigenschaften eben dieses Typs, insbesondere, wichtig für die mit ihnen verwendete Adressarithmetik, seine Speichergröße. Zeiger werden deklariert mittels des Operators `*`.

```
int ip; / ip ist ein Zeiger auf Typ int */
```

Zeiger müssen initialisiert werden, bevor sie zum Zugriff auf die mit ihnen bezeigten Objekte benutzt werden können:

```
ip = &abc; /* ip wird auf die Adresse von abc gesetzt */
```

Zeiger können nur mit Zeigern gleichen Typs initialisiert werden, oder mit Zeigern auf `void`, (also auf nichts bestimmtes). Zum Initialisieren von Zeigern wird meist der Adressoperator `&` verwendet, der einen Zeiger auf seinen Operanden erzeugt. In einem Zuweisungszusammenhang gilt der Name eines Arrays als Zeiger auf das erste Element (das mit dem Index 0) des Arrays, d.h. wenn wie im Beispiel weiter oben iv ein Array vom Typ `int` ist:

```
ip = iv; /* gleichwertig mit ip = &iv[0] */
```

Der Zugriff auf das vom Zeiger referenzierte Objekt, (die sog. Dereferenzierung), geschieht mittels des Operators *:

```
if(*ip) /* Test des Inhalts der Var., auf die ip zeigt */
```

Wenn ip auf iv zeigt, dann ist *ip identisch mit iv[0], man hätte auch schreiben können ip[0] oder *iv. Hier zeigt sich nun der grundlegende Zusammenhang zwischen Array- und Zeigernotation in der Sprache C, es gilt:

`a[n]` ist identisch mit `*(a+n)`

Zu beachten ist hierbei lediglich, dass Arraynamen in einem Zugriffskontext feste Zeiger sind (Adressen), sie stellen kein Objekt im Speicher dar und können somit auch nicht verändert werden, wohingegen Zeigervariablen Objekte sind. Ein Zeiger

kann inkrementiert und dekrementiert werden, d.h. integrale Größen können addiert oder subtrahiert werden, der Zeiger zeigt dann auf ein dem Vielfachen seiner Schrittweite entsprechend entferntes Objekt.

Zeiger gleichen Typs dürfen miteinander verglichen oder voneinander subtrahiert werden. Wenn sie in den gleichen Bereich (z.B. ein entspr. deklariertes Array) zeigen, ergibt sich eine integrale Größe, die den Indexabstand der so bezeigten Elemente bedeutet. Wenn das nicht der Fall ist, ist diese Operation nicht sinnvoll. Erlaubt (und häufig angewandt) ist auch das Testen des Wertes eines Zeigers.

Die Zuweisung integraler Werte an einen Zeiger hat die Bedeutung einer Adresse des vom Zeiger bezeigten Typs. Wenn die Bedingungen der Ausrichtung dieses Typs (z.B. ganzzahlige Vielfache einer best. Größe) nicht erfüllt sind oder der Zugriff auf diese Adresse in der entspr. Typgröße nicht erlaubt sein sollte, kann dies zu Laufzeitfehlern führen. Der Wert 0 eines Zeigers hat die Bedeutung, dass dieser Zeiger ungültig ist, ein sog. Nullzeiger (*null pointer*) – der Zugriff auf die Adresse 0 ist in einem C-System, gleich ob lesend oder schreibend, allgemein nicht gestattet.

Zeiger auf den Typ `void` (also auf nichts bestimmtes) dienen als generische Zeiger lediglich zur Zwischenspeicherung von Zeigern auf Objekte bestimmten Typs. Man kann sonst nichts sinnvolles mit ihnen anfangen, auch keine Adressberechnungen. Sie dürfen ohne weiteres allen Typen von Zeigern zugewiesen werden und umgekehrt.

Initialisierung von Arrays

Wenn erwünscht, können Arrays durch Angabe einer Initialisierungsliste mit konstanten Werten initialisiert werden, hierbei darf dann die Dimensionsangabe fehlen, man spricht dann von einem unvollständigen Arraytyp (*incomplete array type*), und der Compiler errechnet sie selbsttätig aus der Anzahl der angegebenen Elemente der Liste (und komplettiert damit den Typ!):

```
int magic[] = {4711, 815, 7, 42, 3}; /* magic hat 5 Elem. */
```

Ist die Dimension angegeben, werden die Elemente beginnend mit dem Index 0 mit den Werten aus der Liste initialisiert und der Rest, so vorhanden, wird auf 0 gesetzt:

```
long prim[100] = {2, 3, 5, 7, 11}; /* ab Index 5 alles 0 */
```

Die Dimensionsangabe darf nicht geringer als die Anzahl der Elemente der Initialisierungsliste sein:

```
float spec[2] = {1.414, 1.618, 2.718}; /* Fehler! */
```

Die Initialisierung geht auch bei mehr als einer Dimension, hier darf nur die höchste (linke) Dimension fehlen, der Compiler errechnet sie dann:

```
int num[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; /* 3 * 3 */
```

Sind alle Dimensionen angegeben und sind weniger Initialisierer da, werden die restlichen Elemente wie gehabt mit 0 initialiert:

```
int num[3][3] = {{1, 2, 3}, {4, 5, 6}}; /* 3 * 3 */
```

Hier – oder im obigen Beispiel – hätte man die inneren geschweiften Klammern auch weglassen können, denn der Compiler füllt bei jeder Dimension beginnend mit dem Index 0 auf, wobei der rechte Index am schnellsten variiert. Oft besteht jedoch die Gefahr der Mehrdeutigkeit und hilfreiche Compiler warnen hier!

Bei der Initialisierung von char-Arrays mit konstanten Zeichenketten darf man die geschweiften Klammern weglassen:

```
char mword[] = „Abrakadabra"; /* mword hat 12 Elemente */
```

anstatt:

```
char mword[] =
     {'A','b','r','a','k','a','d','a','b','r','a','\0'};
```

oder:

```
char mword[] = {"Abrakadabra"};
```

Auch hier zählt der Compiler wieder die Anzahl der Elemente ab (incl. der terminierenden Null) und dimensioniert das Array selbsttätig. Eine evtl. vorhandene Dimensionsangabe muss mindestens der erforderlichen Anzahl entsprechen, überzählige Elemente werden auch hier mit 0 aufgefüllt:

```
char name[64] = „Heiner Mueller"; /* Ab Index 14 alles 0 */
```

Man beachte folgenden wichtigen Unterschied:

```
char xword[] = „Hokuspokus"; /* xword hat 11 Elemente */
char xptr = „Hokuspokus"; / xptr zeigt auf
                            Zeichenkettenkonstante */
```

Im ersten Fall handelt es sich um ein Array namens xword von 11 Elementen in Form eines C-Strings (mit terminierender Null), im zweiten Fall haben wir mit xptr einen Zeiger, der auf einen an anderer Stelle (möglicherweise im Nur-Lesebereich) gespeicherten C-String (jetzt als namenloses Array vom Typ `char`) zeigt.

7.2.10 Strukturen

Eine Struktur (in anderen Sprachen oft als *record*, Verbund, Datensatz bezeichnet) ist als Aggregat ein komplexer Datentyp, der aus einer Anreihung von einer oder mehreren Komponenten (*members*) oft auch verschiedenen Typs besteht, um diese so zusammengefassten Daten dann als Einheit behandeln zu können.

Eine Struktur wird definiert mit dem Schlüsselwort struct gefolgt von einem Block mit den Deklarationen der Komponenten. Beispiel:

```
struct person {
       int num;
       char name[64];
       char email[64];
       char telefon[32];
       char level;
```

```
};
```

Hier werden mit dem Schlüsselwort `struct` und dem Bezeichner person, dem sog. Etikett (structure tag), zusammengehörige Daten in einer Struktur zusammengefasst: Es wird ein neuer, benutzerdefinierter Datentyp namens `struct person` geschaffen.

Die Namen der in dem Strukturblock deklarierten Komponenten befinden sich in einem eigenen Namensraum und können nicht mit anderen (äußeren) Namen oder Namen von Komponenten in anderen Strukturen kollidieren. Es wird hierbei auch noch kein Speicherplatz reserviert, sondern lediglich der Typ bekannt gemacht, seine Form beschrieben, also ein Bauplan zur Beschaffenheit dieses Typs und seiner Struktur vorgelegt.

Speicherplatz kann reserviert und somit Variablen dieses Typs erzeugt werden, indem man zwischen der beendenden geschweiften Klammer des Strukturblocks und dem abschließenden Semikolon eine Liste von Variablennamen einfügt. Übersichtlicher ist wohl aber meist, die Beschreibung der Form von der Speicherplatzreservierung zu trennen. Variablen dieses Typs werden dann z.B. so vereinbart:

```
struct person hugo, pp; / 1 Variable und ein Zeiger */
```

Man kann natürlich auch gleich ganze Arrays von diesem neuen Typ erzeugen:

```
struct person ap[100]; /* Array von 100 struct person */
```

Der Compiler sorgt dafür, dass die Komponenten der Strukturen in der Reihenfolge ihrer Deklaration mit der korrekten Ausrichtung angelegt werden und dass die Gesamtheit der Struktur so gestaltet ist, dass sich mehrere davon als Elemente eines Arrays anreihen lassen. Je nach Gestalt der Struktur, abhängig von Maschinenarchitektur und Compiler können dabei zwischen den Komponenten und am Ende der Struktur auch Lücken entstehen, so dass die Gesamtgröße einer Struktur (zu ermitteln mithilfe des `sizeof`-Operators) u.U. größer ist als die Summe der Größen ihrer Komponenten. Der Speicherinhalt der so entstandenen Lücken bleibt dabei undefiniert.

Auf die Komponenten zugegriffen wird direkt mit dem `.`-Operator:

```
hugo.num = 4711; /* Schreibzugriff auf Komp. num von hugo */
```

Der indirekte Zugriff (über Zeiger) geschieht mithilfe des `->`-Operators:

```
pp = &hugo;
pp->level = 12; /* Zugriff auf Komponente level von hugo */
```

Oder entsprechend bei Zugriff auf ein Element eines Arrays:

```
ap[5].num = 4712; printf( „%d", (ap+5)->num );
```

Strukturen können selbst auch wieder (andere) Strukturen als Komponenten enthalten. Erlaubt ist auch die Definition von Strukturen innerhalb des Strukturdefinitionsblocks – dieser Typ ist dann allerdings auch im Sichtbarkeitsbereich der einbettenden Struktur bekannt, daher sollte dies besser vermieden werden. Wenn die Definition des Strukturblocks nicht erfolgt oder noch nicht abgeschlossen ist,

spricht man von einem unvollständigen (*incomplete*) Datentyp. Davon lassen sich dann zwar keine Variablen erzeugen – Speicherplatzverbrauch und Gestalt sind ja noch unbekannt, es lassen sich aber schon Zeiger auf diesen Typ erstellen. Auf diese Weise können Strukturen Zeiger auf ihren eigenen Typ enthalten, eine Konstruktion, die oft zur Erzeugung von verketteten Listen verwandt wird. Beispiel:

```
struct mlist {
      struct mlist *prev;
      struct mlist *next;
      char descr[64];
};
```

Strukturen können (an Variablen gleichen Typs) zugewiesen werden, als Argumente an Funktionen übergeben und als Rückgabetyp von Funktionen deklariert werden. Die Zuweisung ist dabei als komponentenweise Kopie definiert. Bei größeren Strukturen empfiehlt sich bei den beiden letzteren Aktionen allerdings, lieber mit Zeigern zu arbeiten, da sonst intern immer über temporäre Kopien gearbeitet wird, was sowohl zeit- wie speicherplatzaufwendig wäre. Strukturvariablen lassen sich ähnlich wie Arrays mit Initialisierungslisten initialisieren.

Syntaktisch ähnlich einer Struktur ist die Variante oder Union (`union`), mit dem Unterschied, dass die verschiedenen Komponenten nicht nacheinander angeordnet sind, sondern alle an der gleichen Adresse liegend abgebildet werden. Vereinbart werden sie mit dem Schlüsselwort `union`, gefolgt von einem optionalen Etikett, gefolgt von einem Definitionsblock mit den Definitionen der Komponenten, gefolgt von einem Semikolon. Sie werden benutzt, um Daten unterschiedlichen Typs am gleichen Speicherplatz unterbringen zu können (natürlich immer nur einen Typ zur gleichen Zeit!), oder um den Speicherplatz anders zu interpretieren

Der Compiler sorgt dafür, dass die Größe der Union, ihre Ausrichtung inklusive etwaiger Auffüllung den Anforderungen der Maschine entsprechen, daher ist die Größe einer Unionsvariablen immer mindestens so groß wie die Größe ihrer größten Komponente.

Bitfelder

Als mögliche Komponenten von `struct` oder `union` können Bitfelder vereinbart werden. Ein Bitfeld dient zur Zusammenfassung von Information auf kleinstem Raum (nur erlaubt innerhalb `struct` oder `union`). Es gibt drei Formen von Bitfeldern:

- normale Bitfelder (plain bitfields) – deklariert als `int`
- vorzeichenbehaftete (signed bitfields) – deklariert als `signed int`
- nicht vorzeichenbehaftete (unsigned bitfields) – deklariert als `unsigned int`

Ein Bitfeld belegt eine gewisse, aufeinander folgende Anzahl von Bit in einem Integer. Es ist nicht möglich, eine größere Anzahl von Bit zu vereinbaren, als in der Speichergröße des Typs int Platz haben. Es darf auch unbenannte Bitfelder geben, auf die man dann natürlich nicht zugreifen kann, dies dient meist der Abbildung der Belegung bestimmter Register oder Ports. Hier die Syntax:

```
struct sreg {
      unsigned int
      cf:1, of:1, zf:1, nf:1, ef:1, :3,
      im:3, :2, sb:1, :1, tb:1;
};
```

Nach dem Doppelpunkt steht die Anzahl der Bit, die das Feld belegt. Wie der Compiler die Bitfelder anlegt, wie er sie ausrichtet und wie groß er die sie enthaltenden Integraltypen macht, ist völlig implementationsabhängig. Wenn man sie überhaupt je verwenden will, wird empfohlen, sie jedenfalls als `unsigned int` zu deklarieren.

7.2.11 Aufzählungstypen

Aufzählungstypen – Schlüsselwort `enum` – sind benannte Ganzzahlkonstanten (enumeration constants), deren Vereinbarungssyntax der von Strukturen ähnelt. Im Gegensatz zu mit `#define` vereinbarten Konstanten, die der C-Präprozessor (→ 7.3) verarbeitet, werden die enum-Konstanten vom C-Compiler selbst bearbeitet.

Sie sind kompatibel zum Typ, den der Compiler dafür wählt – einen Typ, aufwärtskompatibel zum Typ `int`: Es könnte also auch `char` oder `short` sein, aber nicht `long`, das ist implementationsabhängig – und lassen sich ohne weiteres in diesen überführen und umgekehrt, ohne dass der Compiler prüft, ob der Wert auch im passenden Bereich liegt. Hier einige Beispiele zur Deklaration, bzw. Definition:

```
enum color {red, green, blue} mycolor, hercolor;
enum month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
            OCT, NOV, DEC};
enum month mymonth;
enum range {VLO=-10, LLO=-5, LO=-2, ZERO=0, HI=2, LHI=5,
            VHI=10, OVL};
enum range myrange, hisrange;
enum level {AF=-3, BF, CF, DF, EF, FF, GF, HF} xx, yy, zz;
```

Bei aller semantischen Nähe zum Typ `int` sind `enum`-Konstanten oft der beste Weg, um mittels benannter Konstanten das Programm übersichtlicher zu machen und „magische“ Zahlen (magic numbers) zu vermeiden, besser oft als die übliche Methode der `#define`-Makros und daher für diesen Zweck sehr zu empfehlen. Diese Art der Verwendung funktioniert natürlich nur für Ganzzahlkonstanten, die den Wertebereich eines `int` nicht überschreiten.

7.2.12 Typdefinitionen

Das Schlüsselwort ist `typedef`. Der Name lässt es zwar vermuten, aber `typedef` dient nicht zur Definition neuer Datentypen, er erzeugt syntaktisch nur andere Namen (Synonyme, Aliasse) für schon bekannte Typen. Das kann, richtig angewandt, zur erhöhten Lesbarkeit des Quelltextes genutzt werden. Einerseits wird `typedef` dazu benutzt, komplizierte oder umständliche Deklarationen zu vereinfachen, andererseits kann durch geschickten Einsatz die Portabilität von Programmcode auf unterschiedliche Umgebungen erhöht werden. Der so erzeugte „neue“ Typ ist mit seinem Ursprungstyp voll kompatibel und syntaktisch quasi-identisch. Die Syntax ist: typedef bekannter-Typ neuer-Typname ;. Ein Beispiel:

```
typedef int          int32
typedef short        int16
typedef signed char  int8
```

7.3 Der C-Präprozessor

Dem C-Präprozessor obliegt die Vorverarbeitung des C-Quelltexts zur sog. Übersetzungseinheit (*translation unit*), die dann dem eigentlichen Compiler zur Weiterverarbeitung übergeben wird. Er arbeitet als zeilenorientierte Textersetzungsmaschine und versteht die C-Syntax *nicht*.

Seine Aufgabe ist es, jede Zeile mit einem *newline character* abzuschließen, unabhängig von der äußeren Form einer Textzeile, durch ihre Entsprechungen zu ersetzen, Zeilen, die mit einem Rückschrägstrich enden, zusammenzufügen, Zeichengruppen zu ersetzen (z.B. Escape-Sequenzen, Makros), Leerraum zu kondensieren, Kommentare (/* ... */) zu entfernen und durch ein Leerzeichen zu ersetzen, Direktiven auszuführen (auch wiederholt und rekursiv), und Dateien einzufügen (mit denen er dann rekursiv das gleiche anstellt).

Präprozessordirektiven werden mit # eingeleitet. Sie beginnen traditionell am linken Rand und stehen auf einer logischen Zeile. Es gibt folgende Direktiven:

#include <datei.h>	Standard-Header hier einfügen
#include „datei.h“	eigenen Header hier einfügen
#define DIES jenes 17	überall >DIES< durch >jenes 17< ersetzen, sog. Makro
#undef XXX	Makrodefinition XXX entfernen
#line 47	nächste Zeilennummer in der Datei
#error „some failure!“	zur Compilierzeit Fehlermeldung erzeugen
#pragma builtin(xyz)	implementationsdefinierte Option
#ifdef FEATURE	bedingte Compilierung
#ifndef FEATURE	bedingte Compilierung

#if	bedingte Compilierung
#elif	bedingte Compilierung
#else	bedingte Compilierung
#endif	bedingte Compilierung
defined	optional zur Verwendung mit #if und #elif

Präprozessor-Makronamen, wie oben z.B. DIES, XXX und FEATURE, werden traditionsgemäß meist komplett in Großbuchstaben geschrieben, führende Unterstriche sind für das System reserviert und sollten nicht verwendet werden.

Der Präprozessor ist eine reine Textersetzungsmaschine, ohne jegliche Kenntnis von C! Semantische Klarheit von Quellcode hat heutzutage jedoch höchste Priorität. Daher ist der moderne Trend in der Anwendungsentwicklung (ca. seit Beginn der 90er Jahre), den Präprozessor nur noch für einfache Makros sowie Inklusion und – falls notwendig – bedingte Compilierung einzusetzen (siehe hierzu auch Codierungsregel 8, → 7.6). In der Systemsoftware sieht es allerdings etwas anders aus, wie man leicht beim Studium der Headerdateien feststellen kann. Man traut den Systemprogrammierern offenbar mehr Durchblick und Disziplin zu!

7.4 Die Standardbibliothek

Während es bei früheren Programmiersprachen allgemein üblich war, die Bedienung der Peripherie, Ein- und Ausgabebefehle, Formatieranweisungen für den Druck, spezielle, für den prospektiven Anwendungsbereich erforderliche mathematische oder textverarbeitende Funktionen und ähnliches alles in der Sprache selbst unterzubringen, wurde C von Anfang an ausgelegt, einen möglichst kleinen Sprachkern in Verbindung mit einer Standardbibliothek zu verwenden. Die Sprache sollte es dem Benutzer auf einfache Weise ermöglichen, diese Bibliothek seinem Bedarf anzupassen und auf Wunsch beliebig zu erweitern. Diese Entwurfsphilosophie ist eines der Hauptkennzeichen von C geblieben.

Zur Sprache C gehört eine Standardbibliothek (*standard C library*), deren Programmierschnittstelle (*application programmer interface*, API) über die weiter unten aufgelisteten, insgesamt 18 sog. Header-Dateien definiert wird.

Sie enthalten die Definitionen für Makros und Datentypen, sowie die Deklarationen von Namen und Funktionen in den entspr. Abschnitten der Bibliothek.

```
<assert.h> <ctype.h> <errno.h> <float.h> <limits.h>
<locale.h> <math.h> <setjmp.h> <signal.h> <stdarg.h>
<stddef.h> <stdio.h> <stdlib.h> <string.h> <time.h>
<iso646.h> <wchar.h> <wctype.h>
```

Die Anbindung entspr. Abschnitte der Standardbibliothek sollte im Quelltext immer über die Einbindung der jeweils zutreffenden Header mittels der #include <...>

Präprozessoranweisungen (→7.3) geschehen, um die notwendigen Definitionen alle korrekt zu übernehmen.

Die Einbindung der entspr. Teile der Objektbibliothek durch den Linker geschieht meist automatisch, zuweilen ist es jedoch notwendig, bestimmte Teile – oft die zu `math.h` gehörenden Funktionen – mittels spezieller Linkeroptionen bei der Compilierung explizit anzufordern.

7.5 Wie arbeitet ein C-Compiler?

Im letzten Teil dieses Kapitels zur Einführung in die imperative Sprache C wird noch ein kurzer Blick auf die Übersetzung von C in Maschinensprache geworfen. Dies erfolgt aus einem besonderen Grund, denn anhand des so genannten Zwischencodes kann man schon vergleichsweise gut auf die in Abschnitt 3.2.3 eingeführten Worst-Case-Execution-Times (WCET) schließen. Doch zunächst folgt einmal ein Blick auf die Compilerphasen.

7.5.1 Compilerphasen

Die Übersetzung eines in C geschriebenen Programms erfolgt in insgesamt 4 Phasen, von denen der Compiler an zweien unmittelbar beteiligt ist. Die 4 Phasen sind:

- Präprozessorphase
- Frontendphase des Compilers
- Backendphase des Compilers
- Linkerphase

Die Präprozessorphase wurde bereits in Abschnitt 5.3 erwähnt, hierbei handelt es sich um eine Vorbereitung des zu übersetzenden Sourcecodes. Textmakros werden ersetzt, die so genannten include-Dateien eingesetzt, Kommentare gelöscht, die Zeilen immer durch ein Newline-Zeichen getrennt usw. Der Output dieser Phase ist ein reiner Sourcecode, der bislang noch keine Überprüfung oder Übersetzung erfahren hat.

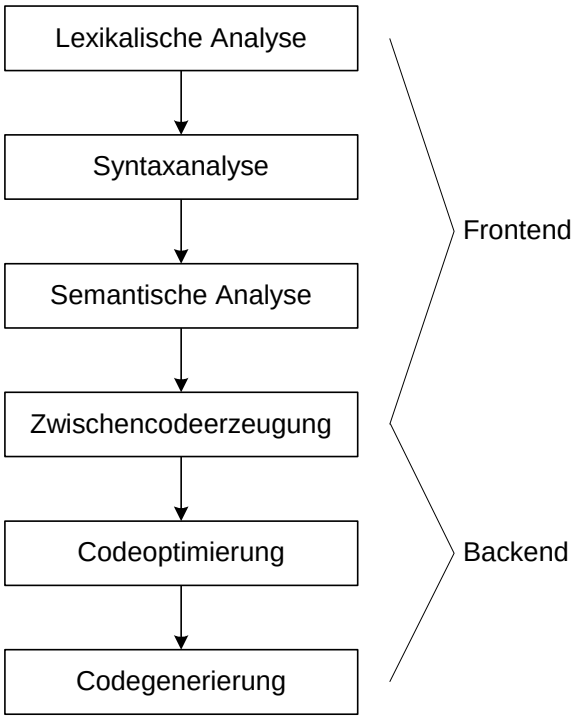


Bild 7.1 Die Phasen eines Compilers [ASU99]

Im Frontend des Compilers (→ Bild 7.1) wird dieser Sourcecode eingelesen (Scanner) und überprüft (Parser). Ziel ist es dabei, die korrekte Syntax zu überprüfen, eine erste Syntaxumwandlung und erste Optimierungen durchzuführen. Das Ziel dieser Phase ist ein Zwischencode, der noch von dem Zielsystem (dem Mikroprozessor) unabhängig ist, aber dennoch die Umsetzung in Assembler- oder Maschinensprache vorbereitet. Der Output dieses Frontendteils wird im nächsten Abschnitt genauer betrachtet.

Im Backend des Compilers erfolgt das Einlesen des Zwischencodes (*intermediate representation, IR*), die Umsetzung in Assemblersprache einschließlich der maschinenspezifischen Optimierung und der Assemblerlauf. Ziel dieses Abschnitts ist der so genannte Objektcode, der neben dem Maschinencode – noch unvollständig – auch Informationen zu den Daten und Programmabschnitten mitführt.

Der Linker liest dann abschließend den Objektcode ein, dazu die angegebenen Standard- und spezifischen Bibliotheken, und fügt das zusammen. Nunmehr sind alle Adressen, auch die der aus der Bibliothek genutzten Funktionen (wie etwa printf) bekannt, und der Maschinencode kann mit allen Adressen vervollständigt

werden. Output des Linkers ist ein ausführbarer Maschinencode (in einem File-format).

7.5.2 Die Erzeugung des Zwischencodes [Sie07a]

Für den Zwischencode existiert kein genormtes Format, jeder Compiler nutzt dort seine hauseigene Syntax. Besonders interessant ist jedoch das Lance2-Compiler-system [Lance2], das aus C ein low-level-C erzeugt und dieses als Zwischencode nutzt. Diese Untermenge von C, die dieses Compilersystem als Zwischencode (Intermediate Representation, IR) nutzt, ist natürlich beschränkt. Wesentliche Merkmale sind:

Anweisungen (Statements):

- Zuweisungen (Assignments): a = b + c, y = function(a, b), ...
- Sprünge (Jumps): goto label_1; (diese Sprünge sind Compiler-berechnet und somit ”zugelassen“)
- Bedingte Sprünge (Conditional Jumps): if(cond) goto label_2;
- Marken (Label): label_1:
- Rücksprung ohne Rückgabewert (return void): return;
- Rücksprung mit Rückgabewert (return value): return x;

Ausdrücke (Expressions):

- Symbole: main, a, count …
- Binäre Ausdrücke (binary expressions): a * b, x / y …
- Unäre Ausdrücke (unary expressions): ~a, *p …
- Type Casts: (int), (char)
- Konstanten (in verschiedenen Formaten): -5, 3.141592653589

Ein kurzer Blick in die obige Liste verrät, dass bei den Anweisungen Schleifen wie for, while und do .. while komplett fehlen. Diese Schleifen werden durch die aufgezählten Konstrukte abgebildet bzw. in diese übersetzt, und es gilt noch zu zeigen, wie dies erfolgt.

Der wichtigste Zusatz, das Zwischencodeformat betreffend, besteht noch in der Beschränkung der Ausdrücke und der Zuweisungen: Sie werden auf ein 3-Adress-format eingeschränkt, d.h., eine Wertzuweisung an ein links stehendes Symbol (a = ...) wird rechtsseitig durch einen unären oder einen binären Ausdruck bestimmt. Längere „Kettenrechnungen“ müssen dementsprechend in Teilrechnungen mit Einfügung temporärer Variablen geteilt werden, eine Aufgabe, die dem Compiler zufällt (zu den Problemen mit Seiteneffekten und Sequenzpunkte siehe hier Anmerkungen in Abschnitt 7.2.4). Der Grund für diese Einschränkung ist sehr offensichtlich: Dem 3-Adressformat entsprechen häufig direkt Assemblerbefehle (etwa: ADD R3, R1, R2, was R3 = R1 + R2 bedeutet).

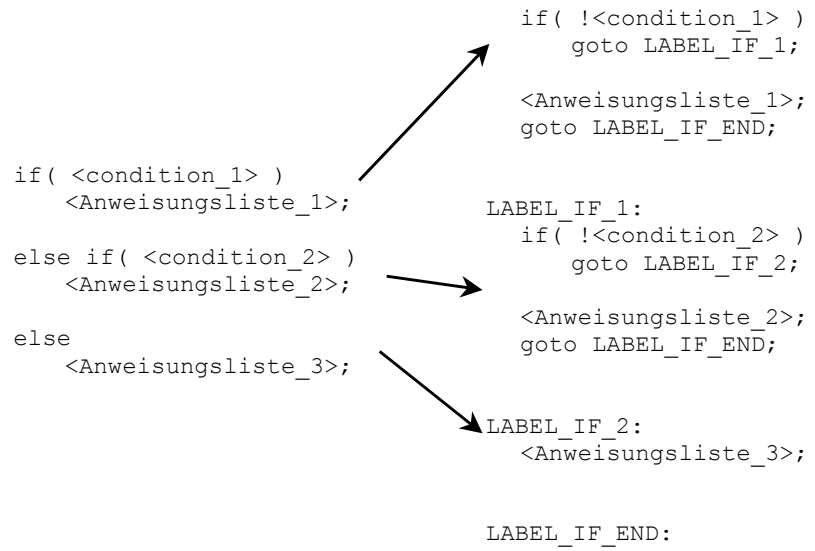


Bild 7.2 Übersetzung der if/else if/else-Verzweigung

Bild 7.2 zeigt die Übersetzung einer if/else if/else-Verzweigung. Dabei wird deutlich, dass nur if-Konstrukte mit anschließendem Sprung (also zusammengefasst der "bedingte Sprung") genutzt werden. Die Bedingungen selbst müssen dabei invertiert ausgewertet werden, da ja die Liste der Anweisungen, die bei Erfüllung der ursprünglichen Bedingung auszuführen sind, nun übersprungen werden.

Dies mag etwas holprig wirken, denn bei Zulassung einer üblichen if-Verzweigung wäre dies wesentlich einfacher zu übersetzen. Diese Form der Übersetzung hat jedoch den entscheidenden Vorteil, dass nur bedingte *Sprünge* verwendet werden, und die lassen sich 1:1 in eine Sequenz von Assemblerbefehlen wie etwa

```
cmp R1, R2;            Auswertung der Bedingung
beq LABEL_IF_1;        Bedingter Sprung
```

übersetzen.

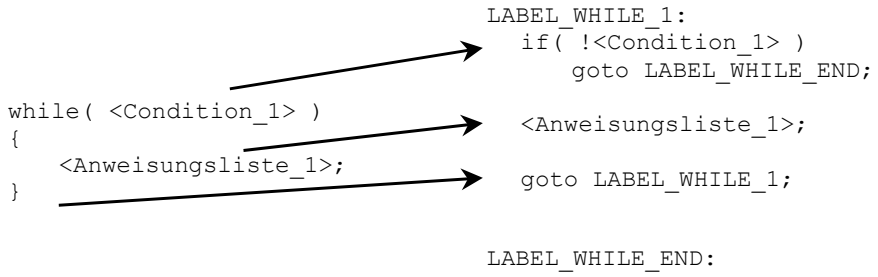


Bild 7.3 Übersetzung einer `while`-Schleife

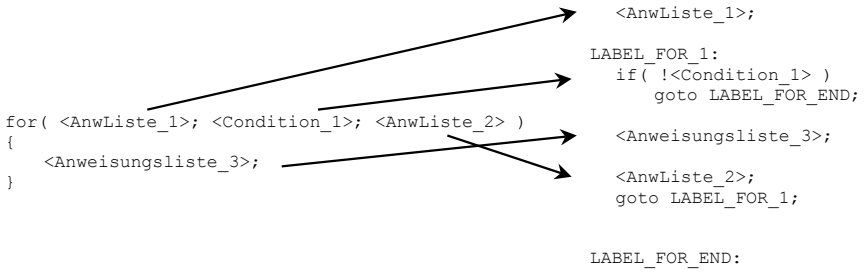


Bild 7.4 Übersetzung einer `for`-Schleife

Bild 7.3 zeigt die Übersetzung der `while`-Schleife, Bild 7.4 die der etwas komplexeren `for`-Schleife. In beiden Fällen werden bedingte und unbedingte Sprünge verwendet, um die Schleifenstruktur entsprechend abzubilden, wobei die Bedingung auch wieder invertiert verwendet werden müssen, um den Sprung aus der Schleife zu beschreiben. Entsprechend den hier gezeigten Codeabschnitten können nun auch die switch/case-Verzweigung (Bild 7.5) und die do..while-Schleife (Bild 7.6) übersetzt werden, wobei die Mehrfach-Fallunterscheidung (switch/case) etwas komplexer ist.

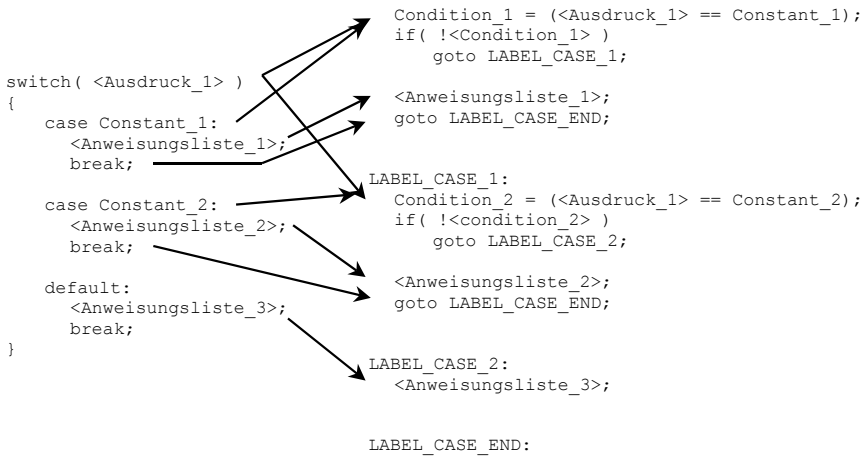


Bild 7.5 Übersetzung der switch/case-Verzweigung

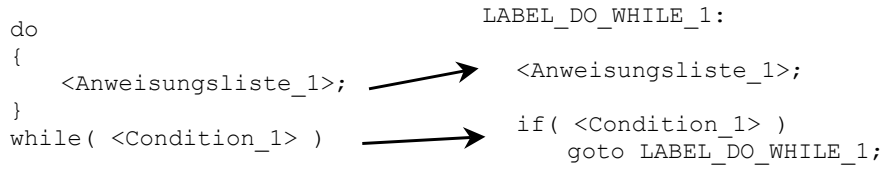


Bild 7.6 Übersetzung einer do .. `while`-Schleife

7.5.3 Laplace-Filter als Beispiel [Sie07b]

Ein zweifellos sehr einfaches Filterprogramm zur Bildverarbeitung besteht in dem Laplace-Filter [Laplace], das näherungsweise die zweite Ableitung eines zweidimensionalen Feldes bildet und dadurch die Kantendetektierung ermöglicht.

Der Algorithmus basiert darauf, dass für jeden neu zu berechnenden Punkt in einem Kantenbild der Wert aus dem ursprünglichen Bild für diesen Punkt genommen wird, mit dem Gewichtsfaktor 4 (im Fall eines 2-dimensionalen Filters) multipliziert wird, und von diesem Wert dann die unmittelbaren Nachbarn (keine Diagonalen) subtrahiert werden. Um die Darstellung des Übersetzungsvorgangs möglichst einfach zu halten, wird hier die eindimensionale Variante gewählt (Bild 7.7).

Dieser Code wird nun entsprechend dem Lance2-Compilers [Sie07a] in einen Zwischencode wie in 7.5.2 beschrieben übersetzt.

```
01:  #define X_DIM 100
02:  short bild[X_DIM];
03:  short kanten[X_DIM];
04:
05:  void laplace_filter_1d(void)
06:  {
07:      short int x;
08:
09:      for( x = 1; x < X_DIM - 1; x++ )
10:      {
11:          kanten[x] = 2 * bild[x] – bild[x-1] - bild[x+1];
12:      }
13:
14:  }
```

Bild 7.7 Sourcecode für eindimensionales Laplace-Filter

Konkret werden im Zwischencode nur wenige Operationen benötigt, so z.B.:

- Wertzuweisungen an Variable, hierbei rechtsseitig einfache Rechnungen mit maximale zwei Operanden und einer Operation; dies wird auch für Adressrechnung bei indizierten Variablen benötigt.
- Vergleiche mit Zuweisung des Booleschen Werts an eine Variable
- `if`-Konstrukt mit einer Variablen, auf deren Wahrheitswert verglichen wird, mit anschließendem (Compiler-berechnetem) `goto`.

```
101: short bild[100];
102: short kanten[100];
```

```
103: void laplace_filter_1d()
104: {
105:  short x_3;
106:  short t1;
107:  short t2;
108:  char *t4;
109:  int t5;
110:  char *t6;
111:  short *t7;
112:  short t8;
113:  short t9;
114:  char *t10;
115:  int t11;
116:  int t12;
117:  char *t13;
```

```
118:  short *t14;
119:  short t15;
120:  short t16;
121:  char *t17;
122:  int t18;
123:  int t19;
124:  char *t20;
125:  short *t21;
126:  short t22;
127:  short t23;
128:  char *t24;
129:  int t25;
130:  char *t26;
131:  short *t27;
```

Bild 7.8 Durch den Frondendteil erzeugter Zwischencode für die Zeilen 1-7 (Bild 7.7)

```
132: x_3 = 1;                    /* Initialisierung for-Schleife */
133: LL1:
134: t1 = x_3 < 99;              /* Berechnung ggf. Schleifenende */
135: t24 = !t1;
136: if( t24 ) goto LL2;

137: t4 = (char *) bild;         /* Berechnung &(bild[x]) */
138: t5 = (int)(x_3 * 2);
139: t6 = t4 + t5;
140: t7 = (short *)t6;
141: t8 = *t7;                   /* Zugriff auf bild[x] */
142: t9 = t8 * 2;

143: t10 = (char *) bild;
144: t11 = (int)(x_3 - 1);       /* Berechnung &(bild[x-1]) */
145: t12 = t11 * 2;
146: t13 = t9 + t11;
147: t14 = (short *)t13;
148: t15 = *t14;                 /* Zugriff auf bild[x-1] */

149: t16 = t9 - t15;             /* Berechnung 2 * bild[x] - bild[x-1] */

150: t17 = (char *) bild;        /* Berechnung &(bild[x+1]) */
151: t18 = (int)(x_3 + 1);
152: t19 = t18 * 2;
153: t20 = t17 + t19;
154: t21 = (short *)t20
155: t22 = *t21;                 /* Zugriff auf bild[x+1] */

156: t23 = t16 - t22;            /* 2 * bild[x] - bild[x-1] - bild[x+1] */

157: t24 = (char *) kanten;      /* Berechnung &(kanten[x]) */
158: t25 = (int)(x_3 * 2);
159: t26 = t24 + t25;
160: t27 = (short *)t26;
161: *t27 = t23;                 /* kanten[x] = ... */

162: t2 = x_3 + 1;                     /* Inkrement der Variablen x */
163: x_3 = t2;
164: goto LL1;
```

Bild 7.9 Zwischencode für den eindimensionalen Laplace-Filter

Generierung des Zwischencodes

Diese Form des Zwischencodes bedeutet aber auch, dass voraussichtlich eine Vielzahl von temporären Variablen benötigt wird, da viele Zwischenrechnungen zu machen sind. Im Zwischencode – hier werden die Zeilen ab 101 durchnummeriert

– in Listing in Bild 7.8 fällt sofort die Vielzahl der Variablen auf, die hier zusätzlich zum Originalcode deklariert werden. Die Zeilen 101-102 entsprechen der Deklaration der Arrays in C, sie werden die Größe 100 haben. Die in der Funktion `laplace_filter_1d` (Bild 7.7) deklarierte Variable taucht in Zeile 105 wieder auf, sie werden lediglich zusätzlich mit einem '_' versehen und durchnummeriert, ansonsten entspricht diese Deklaration der von C.

Die nun folgenden Variablen sind alles vom Compiler erzeugte temporäre Variablen. Man erkennt sie gut an dem fehlenden Unterstrich im Namen. Diese Variablen werden für Berechnungen gebraucht, die im C-Code noch ohne Zwischenschritte auskamen, und sind eine Folge der Übersetzung in der 3-Adresscode. Da der Compiler temporäre Variabeln nicht wieder verwendet, legt er erst einmal für jede dieser Berechnungen eine neue temporäre Variable an. Etwaige Optimierungen sind Aufgabe des nachfolgenden Codegenerators.

Bild 7.9 zeigt den entstehenden Zwischencode bei den gegebenen Voraussetzung. Die einzelnen Berechnungen erscheinen recht komplex, etwa für den Zugriff auf bild[x+1] (Zeile 150-155), sie sind aber notwendig, und in jeder Zeile wird das 3-Adressformat eingehalten. Der Zugriff auf `bild[x+1]` bedeutet nicht anderes, als dass die Adresse `&bild[0] + (x+1)*sizeof(bild[0])` gebildet und dann auf den Inhalt lesend oder schreibend zugegriffen wird. Diese Berechnung der Zugriffsadresse ist in den Zeilen 150 (Basisadresse als char *) sowie 151-154 (Index- und Adressberechnung) codiert und steht dann in der Variablen t22 zur Verfügung. Die Operation `sizeof(bild[0])` liefert dabei den 2 für short.

Das Setzen der Basisadresse &bild[0] findet offenbar mehrfach in den Zeilen 137, 143 und 150 statt. Dies kann eventuell optimiert werden, wobei der Compiler exakt prüfen muss, ob nicht durch externe Programmteile – eine Interrupt Service Routine etwa – die Adresse verändert werden könnte, da bild[] global definiert wurde. In diesem Fall kann aber die Adresse selbst nicht verändert werden – sie ist konstant, nur die Inhalte sind variabel – so dass die Zeilen 143 und 150 entfallen können (→ nächsten Abschnitt).

Die Zeilen 134 bis 136 stellen die Auswertung der Schleifenendebedingung dar. Zunächst wird der aktuelle Wert von x_3 mit 99 (X_DIM-1) verglichen, und das Vergleichsergebnis wird der Compiler-generierten Variablen t1 zugewiesen. Diese Variable fungiert als Boole'sche Variable, d.h., sie soll nur Werte für true und false speichern. Die Zuweisung des negierten Wahrheitswerts an t24 in Zeile 135 und die Auswertung durch einen bedingten Sprung (Zeile 136) komplettieren diesen Abschnitt.

Vom Zwischencode zum Assemblercode

Die Übersetzung des Zwischencodes in einen Assemblercode soll anhand einer Modell-CPU erfolgen. Diese wird als MPM3, Mikroprozessormodell #3, bezeichnet und stellt einen RISC-Prozessor mit einer intrinsischen Verarbeitungsbreite von 16 bit dar [Sie04]. Dieses Modell wurde gewählt, weil die typischen Vorgänge

daran sehr gut gezeigt werden können, ohne auf die Spezialitäten einer marktgängigen Architektur eingehen zu müssen.

Die spontane Übersetzung des Zwischencodes wird durch einen weiteren Vorgang, die Abbildung der Variablen auf Register oder Speicher betreffend, gebremst. Die Abbildung auf den Maschinencode ist tatsächlich nicht besonders schwierig, hingegen sind die Anforderungen an die Daten schon schwieriger erfüllbar. Bei diesem Vorgang müssen insbesondere Randbedingungen wie Laufzeitminimierung erfüllt werden.

```
301:  ORG $0200
302:  BILD      DW 0
303:  ORG $264
304:  KANTEN DW 0
305:  ORG $0300
306:  _laplace_filter_1d:
```

Bild 7.10 Assemblercodegenerierung für die Zwischencodezeilen 101-103

Die Übersetzung des ersten Teils des Zwischencodes – Bild 7.8 – fällt überraschend klein aus. Die gesamte Abbildung der doch eher großen Menge an Variablen erfolgt so, dass lediglich die Variablen bild[] und kanten[] im Speicher angelegt und mit Werten initialisiert wird. Die Initialisierung erfolgt dem Datentyp short gemäß mit 16 bit (DW, define word). Die übrigen Variablen werden auf 7 der vorhandenen 8 Datenregister R0 bis R7 abgebildet (Tabelle 7.4).

Tabelle 7.4 Zuordnung der Variablen zu Registern

Variable	*Register*
x	R0

Variable	*Register*
t4, t10, t17, t24	R1
t5, t25	R2
t6, t26	R3
t8, t9, t16, t23	R4
t11, t12, t13, t18, t19, t20	R5
t15, t22	R6

Es entfallen:
t1, t2, t7, t14, t21, t27

Bild 7.11 zeigt die Übersetzung des Zwischencodes in den Assemblercode und hierbei auch einige Optimierungsmöglichkeiten (die keineswegs immer im Backendgenerator vorhanden sein werden). Wie bereits dargestellt können die Zeilen 143 und 150 durch Optimierung entfallen (im Übrigen bereits im Zwischencode).

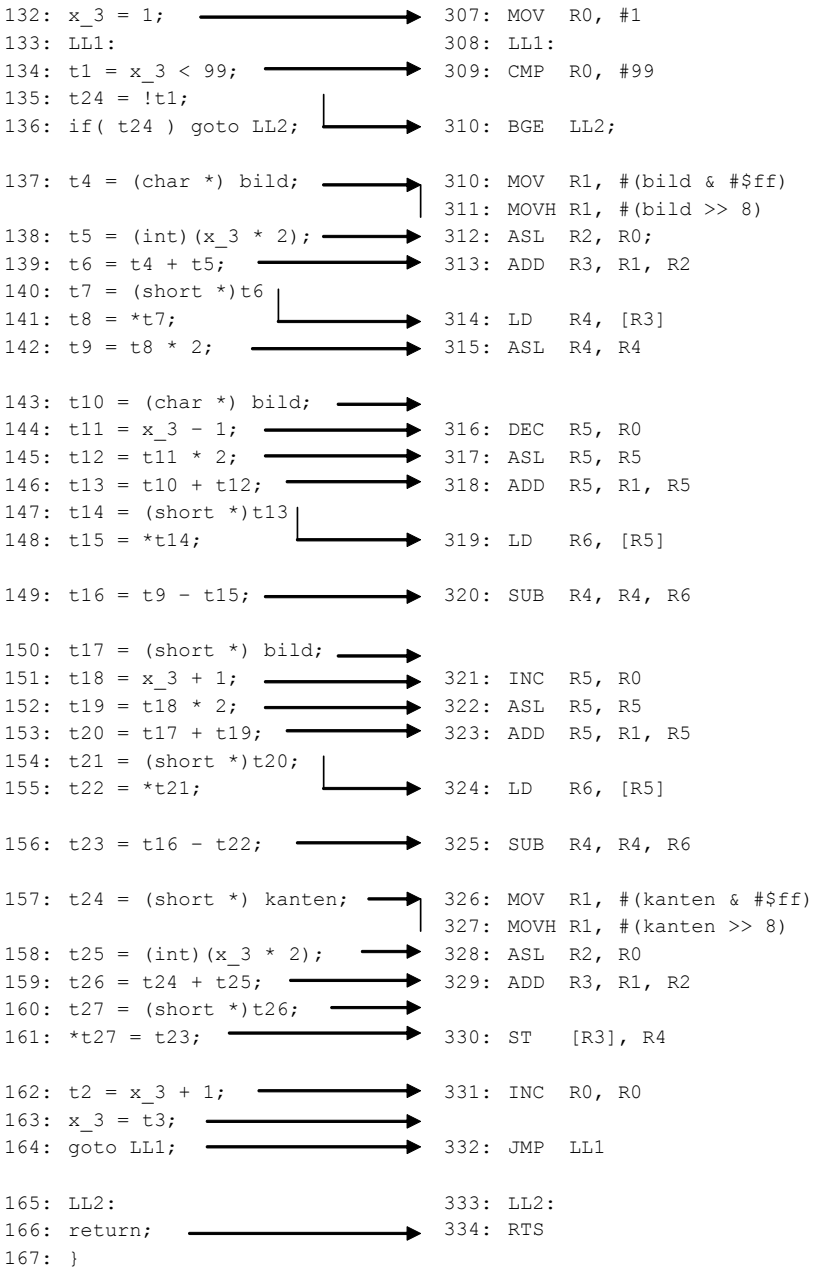


Bild 7.11 Abbildung der Zwischencodezeile 132-167 in Assemblercode.

Die Optimierung kann sogar noch weitergehen: Das Programm läuft ein Weile in der Schleife von `LL1` (Zeile 133) bis goto LL1 (Zeile 164), und in der gesamten Schleife werden der Wert für short *bild (in R1) und short *kanten (ebenfalls in R1) drei- bzw. einmal im Register geschrieben und dann lesend benutzt. Wenn man also den Wert für short *kanten in einem anderen Register speichern kann (z.B. in R7), dann könnten die Zeilen 310/311 und 326/327 im Assemblercode nach oben, also zwischen Zeile 307 und 308 geschoben werden. Dieses Verfahren, *Instruction Scheduling* genannt, bewirkt in diesem Fall, dass die Zeilen nur einmal für alle Schleifen durchlaufen werden und spart somit Rechenzeit.

Noch kurz ein Wort zu den vielleicht ungewöhnlich ausschauenden Zeilen 310/311 bzw. 326/327. In diesen Assemblercodezeilen wird ein Register mit einer 16-bit-Konstanten geladen. Die zugrunde liegende Architektur ist (angenommen) ebenfalls mit 16-bit-Datenstrukturen (Register, ALU) und 16-bit Adressen versehen, und hierbei kommt es zum Problem. Bei RISC-Prozessoren ist es sozusagen Pflicht, dass ein Befehl in einem Takt bearbeitet wird, und wenn nun ein Befehl eine Breite von 16 bit im Speicher hat, dann passen keine Konstanten mit 16 bit Breite dort hinein (weil ja die Operation auch noch beschrieben werden muss).

Die Lösung besteht in dem Befehlspaar MOV/MOVH (Move und Move High Byte). Der erste Teil kopiert den Operanden – der untere Teil der Adresse für bild bzw. kanten – in die entsprechenden Bits des Registers, meist mit Belegung auch der oberen Bits (auf 0 oder mit Vorzeichenerweiterung), und MOVH kopiert dann den Operanden, dem oberen Teil der Adresse entsprechend, in die oberen Bits des Registers.

Letztes Augenmerk soll noch auf die Übersetzung der bedingten Sprünge gelegt werden (Zeile 134-136). Der Zwischencode bestand hier aus drei Anweisungen: die Bedingung wird auf ihren Wahrheitswert berechnet, der Wert wird invertiert, und unter Auswertung dieses entstehenden booleschen Wertes wird dann gesprungen (oder nicht). Im Assemblercode treten hiervon nur noch zwei Anweisungen auf: Die Auswertung der Bedingung wird auf das Setzen von Flags abgebildet (Zeile 309), und diese Flags werden – mit logischer Invertierung, denn BGE bedeutet branch if greater or equal, was die umgekehrte Bedingung zu kleiner (less than) ist – dann in Zeile 310 ausgewertet.

7.5.4 Optimierungsmöglichkeiten

Im vorangegangenen Beispiel wurden die Zeilen 143 und 150 wegoptimiert, weil es sich offensichtlich immer um die gleiche Zuweisung handelt. Im Allgemeinen gilt, dass derartige Optimierung der mehrfachen Wertzuweisung durchgeführt werden können, wenn sichergestellt ist, dass Wertänderungen – auch durch externe Routinen wie ISR – ausgeschlossen sind.

Diese Optimierung, die bereits im Zwischencode erfolgen kann, ist immer dann möglich, wenn die entsprechende Variable

- lokal angelegt ist oder
- bei globaler Speicherklasse konstant ist.

Im Fall der Beispiels des eindimensionalen Laplace-Filters (Bild 7.7) galt die zweite Voraussetzung, da die Adresse des Arrays `bild[]`, also `&(bild[0])`, konstant angelegt wird.

7.5.5 Zusammenhang zwischen Zwischencode und WCET

Abschließend soll der Zusammenhang zwischen dem Zwischencode und den Worst-Cae-Execution-Times beleuchtet werden. Das Beispiel aus dem vorigen Abschnitt ist dabei richtungsweisend, denn der (fast) lineare Zusammenhang zwischen Zwischencode und Assemblercode wird immer beobachtet.

Um einen Schätzwert für die WCET zu erhalten, muss man also den Codezeilen des Zwischencodes per Tabelle maximale Ausführungszeiten zuordnen und zusammenzählen. Da der Code nicht optimiert ist, liegt die Vermutung nahe, dass die berechneten Zeiten wirklich maximale Zeiten darstellen.

In der Praxis jedoch gibt es dabei Detailprobleme. Abgesehen davon, dass die oben geäußerte Vermutung kein Beweis ist (in der Praxis aber „nie“ widerlegt wird), sollten die berechneten WCETs aber auch realistisch sein, und hier wird es schwieriger.

Als Beispiel sei die Zeile 139 aus Bild 7.11 betrachtet:

```
139:   t6 = t4 + t5;
```

Diese wird in die Assembleranweisung

```
313:   ADD  R3, R1, R2
```

übersetzt, also ein einfacher Additionsbefehl mit einer Laufzeit von einem Takt, was dann der geschätzten WCET für diese Zeile entspricht. Dieser eine Takt gilt aber nur, wenn die drei temporären Variablen t4, t5 und t6 in jeweils einem Register gehalten werden. Sind diese Variablen (aus Registermangel) im Speicher, sieht die Übersetzung für diese Architektur ganz anders aus:

```
313a:  MOV  R7, #(t4 & #&ff)
313b:  MOVH R7, #(t4 >> 8)
313c:  LD   R1, [R7]
313d:  MOV  R7, #(t5 & #&ff)
313e:  MOVH R7, #(t5 >> 8)
313f:  LD   R2, [R7]
313g:  ADD  R3, R1, R2
313h:  MOV  R7, #(t6 & #&ff)
313i:  MOVH R7, #(t6 >> 8)
```

```
313j:  ST   [R7], R3
```

Nunmehr sind es 10 Assemblerzeilen geworden, also eine WCET von 10 Takten, weil für jedes Laden erst die Adresse in ein Register, dann der Speicherinhalt geladen werden muss, dann addiert wird, und dann das Ergebnis wieder zurückgeschrieben wird.

Rechnet man also mit einer WCET von 10 für die Zeile 139, kann man diese garantieren, aber die Gefahr, dass nun drastisch überschätzt wird, ist groß. Dieser Widerspruch kann aktuell nur gelöst werden, wenn die WCET auf Maschinencodeebene bestimmt oder geschätzt wird.

7.6 Coding Rules

Abschließend in diesem Kapitel sollen – beispielhaft – Codierungsregeln (Coding Rules) zitiert werden, die gerade für Softwareentwicklung in sicherheitskritischen Bereichen gelten und anerkannt sind. Über Codierungsregeln kann man sich natürlich sehr ausführlich auslassen, jede Firma, jede Entwicklungsgruppe, die etwas auf sich hält, hat mindestens ein Regelwerk, das auch sehr umfänglich sein kann. Die hier zitierten Regeln [Hol06] stellen mit einer Anzahl von 10 ein übersichtliches Regelwerk dar.

Regel 1:

Im gesamten Code sollen nur einfache Kontrollflusskonstrukte verwendet werden. Insbesondere sollen *goto*, direkte oder indirekte Rekursion vermieden werden.

Dies resultiert insbesondere in einer erhöhten Klarheit im Code, der leichter zu analysieren und zu beurteilen ist. Die Vermeidung von Rekursion resultiert in azyklische Codegraphen, die wesentlich einfacher bezüglich Stackgröße und Ausführungszeit analysiert werden können.

Die Regel kann noch dadurch verschärft werden, dass pro Funktion nur ein einziger Rücksprung erlaubt ist.

Regel 2:

Alle Schleifen müssen eine Konstante als obere Grenze haben. Es muss für Codecheck-Tools einfach möglich sein, die Anzahl der durchlaufenen Schleifen anhand einer Obergrenze statisch bestimmen zu können.

Diese Regel dient dazu, unbegrenzte Schleifen zu verhindern. Hierbei müssen auch implizit unbegrenzte Schleifen wie das folgende Beispiel verhindert werden, die wichtige Regel ist also diejenige, dass der Codechecker die Obergrenze erkennen können muss.
Es gibt allerdings eine Ausnahme von dieser Regel: Es gibt immer wieder explizit unendlich oft durchlaufene Schleifen (etwa: `while(1)`), die für bestimmte Aufgaben notwendig sind (Process Scheduler, Rahmen für endlos laufendes Programm etc.). Diese sind selbstverständlich erlaubt.

```
int k, m, array[1024];

for( k = 0, m = 0; k < 10; k++, m++ )
{
          if( 0 == array[m] )
              k = 0;
}
```

Bild 7.12 Implizit unbegrenzte `for`-Schleife (als Negativbeispiel)

Eine Möglichkeit, diese Regel zu erfüllen und bei Überschreiten dieser oberen Grenze einen Fehler bzw. eine Fehlerbehebung einzuführen, sind so genannte `assert()`-Funktionen (siehe auch Hardwarebeschreibungssprachen wie VHDL). Bei Überschreiten wird eine solche Funktion aufgerufen, diese kann dann entsprechende Aktionen einleiten. Es ist zwar möglich, die Fehlerbehebung auch in den eigentlichen Sourcecode einzubauen, die explizite Herausführung dient aber der Übersicht.

Regel 3:

Nach einer Initialisierungsphase soll keine dynamische Speicherallokation mehr erfolgen.

Die Allokationsfunktionen wie `malloc()` und die Freigabe (`free()`) sowie die Garbage Collection zeigen oftmals unvorhersagbare Verhaltensweisen, daher sollte hiervon im eigentlichen Betrieb Abstand genommen werden. Zudem stellt die dynamische Speicherverwaltung im Programm eine hervorragende Fehlerquelle dar bezüglich Speichernutzung nach Rückgabe, Speicherbereichsüberschreitung etc.

Regel 4:

Keine Funktion soll mehr als 60 Zeilen haben, d.h. bei einer Zeile pro Statement und pro Deklaration soll die Funktion auf einer Seite ausgedruckt werden können. (siehe auch 13.1, Lines-Of-Code)

Diese Regel dient einfach der Lesbarkeit und der Übersichtlichkeit des Codes.

Regel 5:

Die Dichte an Assertions (siehe auch Regel 2) soll im Durchschnitt mindestens 2 pro Funktion betragen. Hierdurch sollen alle besonderen Situationen, die im Betrieb nicht auftauchen dürfen, abgefangen werden. Die Assertions müssen seiteneffektfrei sein und sollen als Boolesche Tests definiert werden.

Die `assert()`-Funktionen selbst, die bei fehlgeschlagenen Tests aufgerufen werden, müssen die Situation explizit bereinigen und z.B. einen Fehlercode produzieren bzw. zurückgeben.

Untersuchungen zeigen, dass Code mit derartigen Assertions, die z.B. Vor- und Nachbedingungen von Funktionen, Werten, Rückgabewerten usw. testen, sehr defensiv arbeitet und einer raschen Fehlerfindung im Test dient. Die Freiheit von Seiteneffekten lässt es dabei zu, dass der Code bei Performance-kritischen Abschnitten später auskommentiert werden kann.

Regel 6:

Alle Datenobjekte müssen im kleinstmöglichen Gültigkeitsbereich deklariert werden.

Dies ist das Prinzip des Versteckens der Daten, um keine Änderung aus anderen Bereichen zu ermöglichen. Es dient sowohl zur Laufzeit als auch zur Testzeit dazu, den Code möglichst einfach und verständlich zu halten.

Regel 7:

Jede aufrufende Funktion muss den Rückgabewert einer aufgerufenen Funktion checken (falls dieser vorhanden ist), und jede aufgerufene Funktion muss alle Aufrufparameter auf ihren Gültigkeitsbereich testen.

Diese Regel gehört wahrscheinlich zu den am meisten verletzten Regeln, aber der Test z.B. darauf, ob die aufgerufene Funktion erfolgreich war oder nicht, ist mit Sicherheit sinnvoll. Sollte es dennoch sinnvoll erscheinen, den Rückgabewert als irrelevant zu betrachten, dann muss dies kommentiert werden.

Regel 8:

Die Nutzung des Präprozessors muss auf die Inkludierung der Headerfiles sowie einfache Makrodefinitionen beschränkt werden. Komplexe Definitionen wie variable Argumentlisten, rekursive Makrodefinitionen usw. sind verboten. Bedingte Compilierung soll auf ein Minimum beschränkt sein.

Der Präprozessor kann (leider) so genutzt werden, dass er sehr zur Verwirrung von Softwareentwicklung und Codechecker beitragen kann, daher die Begrenzung. Die Anzahl der Versionen, die man mittels bedingter Compilierung und entsprechend vielen Compilerswitches erzeugen kann, wächst exponentiell: Bei 10 Compilerswitches erhält man bereits $2^{10} = 1024$ verschiedene Versionen, die alle getestet werden müssen.

Regel 9:

Die Nutzung von Pointer muss auf ein Minimum begrenzt sein. Grundsätzlich ist nur ein Level von Dereferenzierung zulässig. Pointer dürfen nicht durch Makros oder `typedef` verschleiert werden. Pointer zu Funktionen sind verboten.

Die Einschränkung bei Zeigern dürfte allgemein verständlich sein, insbesondere aber soll die Arbeit von Codecheckern nicht behindert werden.

Regel 10:

Der gesamte Code muss vom ersten Tag an so compiliert werden, dass die höchste Warnstufe mit allen Warnungen zugelassen eingeschaltet ist. Der Code muss ohne Warnungen compilieren. Der Code muss täglich gecheckt werden, möglichst mit mehr als einem Codeanalysator, und dies mit 0 Warnungen.

Diese Regel sollte peinlichst beachtet werden, denn Warnungen bedeuten immer etwas. Sollte die Warnung als verkehrt identifiziert werden, muss der Code umgeschrieben werden, denn dies kann auch bedeuten, dass der Codechecker den Teil nicht versteht.

Als Tipp für einen Codechecker: Lint bzw. splint (Secure Programming Lint) [lint].

8 Sichere Software und C

Die Abschnitt befindet sich zurzeit in Planung.

9 Hardwarenahe Programmierung

9.1 Einführung

Ein Thema wie *Hardwarenahe Programmierung* (in einer Hochsprache) sollte es eigentlich gar nicht geben, denn Hochsprache implizit Hardwareunabhängigkeit – und nicht ein spezifisches Eingehen auf die Eigenheiten selbiger. Dennoch ist dieses Thema aus der Praxis nicht wegzudenken.

Daher ist die Empfehlung, sich zumindest gedanklich mit den Tücken der Hardwarenähe auseinander zu setzen. In diesem Kapitel kann das nicht vollständig geschehen, sondern eher exemplarisch, aber dennoch kann man die Hardwareabhängigkeiten einigermaßen kategorisieren.

Hardwarenahe Programmierung kann in der Entwicklung und der Praxis bedeuten:

- Ressourcenbeschränkungen (gerade in eingebetteten Systemen)
- Umständliche Konfiguration von Peripherieelementen (diese ist eher auf Bit- und Byte-Ebene zu sehen)
- Extreme Anpassung der Software auf Hardwaregegebenheiten, z.B. bei spezialisierter Hardware oder auch fehlenden Hardwarekomponenten
- Echtzeitprogrammierung bei sehr knappen Rechenzeiten

Bei hardwarenaher Programmierung handelt es sich demnach fast immer um eine Systementwicklung mit Elementen aus dem Bereich Hardware/Software Co-Design.

In der Praxis ist es natürlich vorgesehen, mit den Hochsprachen eine Unabhängigkeit von der ausführenden Hardware zu erreichen, um genau dies zu vermeiden. Insofern ist im Einzelfall eine Balance zu finden, wie weitgehend die Hardwareabhängigkeit gehen darf.

9.2 Ressourcenbeschränkungen am Beispiel der Diskreten Fouriertransformation

9.2.1 Einführung Beispiel 1: Diskrete Fourier-Transformation

Fouriertransformation (FT): Die Fouriertransformation ist die Analyse eines zeitkontinuierlichen Signals auf die beinhalteten Frequenzen. Hierbei geht man zunächst von periodischen Signalen aus, die sich also nach einer bestimmten Zeit

wiederholen. Diese Wiederholungsfrequenz ergibt dann die so genannte Grundfrequenz, das Frequenzspektrum ist diskret.

Der Übergang von periodischen auf aperiodische Signalformen ergibt dann den Übergang vom diskreten auf das kontinuierlichen Spektrum. Umgekehrt kann man die ursprüngliche Signalform g(x) durch phasen- und amplitudenkorrektes Überlagern der im Spektrum vorhandenen Frequenzen wieder erzeugen (Fourier-Synthese). Hier die Formeln zur Berechnung in der reellen Schreibweise, wobei angenommen wird, dass g(x) eine reellwertige Funktion ist:

$$g(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos kx + b_k \sin kx \right]$$

$$a_0 = \frac{1}{\pi} \int_0^{2\pi} g(x) dx$$

$$a_k = \frac{1}{\pi} \int_0^{2\pi} g(x) \cos kx \, dx$$

$$b_k = \frac{1}{\pi} \int_0^{2\pi} g(x) \sin kx \, dx$$

(9.1 bis 9.4)

Der Fourier-Transformation liegt zugrunde, dass man (fast) jedes periodische Signal (, das dem so genannten Dirichlet-Kriterium (endliche Anzahl von Unstetigkeitsstellen und endliche Anzahl von Extrema) entspricht) durch die Überlagerung von Sinus- und Cosinusfunktionen der Grundwelle (= Periode) und ihrer Oberschwingungen zusammensetzen kann.

Diskrete Fouriertransformation (DFT): In Rechnern ist die „Zeitachse“ niemals kontinuierlich, sondern immer diskret. Dies liegt u.a. daran, dass selbst die AD-Wandler niemals zeitkontinuierlich, sondern immer nur zeitdiskret von der analogen in die digitale Welt übertragen können. AD-Wandler diskretisieren in zweifacher Weise: Werte und Zeit werden diskretisiert.

Damit werden die Berechnungsintegrale der allgemeinen Fouriertransformation zu Berechnungssummen, die vergleichsweise einfach in Form von Algorithmen implementiert werden können. Gleichung (9.5) zeigt die Formel für die DFT.

T bezeichnet hierin die Periode des Signals, Δt den zeitlichen Abstand zweier aufeinanderfolgender Messpunkte.

$$T = N\Delta t, \quad t = i\Delta t$$

$$a_0 = \frac{1}{N}\sum_{i=0}^{N-1} g(i\Delta t)$$

$$a_k = \frac{1}{N}\sum_{i=0}^{N-1} g(i\Delta t)\cos(k\varpi \cdot i\Delta t)$$

$$b_k = \frac{1}{N}\sum_{i=0}^{N-1} g(i\Delta t)\sin(k\varpi \cdot i\Delta t)$$

$$\varpi = \frac{2\pi}{N\Delta t}$$

(9.5 bis 9.9)

Die DFT wird genutzt, wenn die Zahl der Messpunkte pro Periode unbekannt oder ungleich einer Zweierpotenz ist. Bei Zweierpotenzen wie 512 oder 1024 jedoch bietet sich die Implementierung als Fast Fourier Transformation (FFT) an. Die Komplexität der DFT liegt bei $O(N^2)$, die der FFT bei $O(N*\log(N))$.

9.2.2 Version 0 des DFT-Algorithmus

9.2.2.1 Die Diskrete Fourier-Transformation

Diese Formel kann man nun vergleichsweise leicht in einen Algorithmus programmieren, z.B. in C:

```
void vComputeDFT( unsigned int uiNumOfPoints, int *iValue )
{
   unsigned int k, m;
   double dCoeffAtemp, dCoeffBtemp;

   for( k = 0; k < NUM_OF_COEFFICIENTS; k++ )
   {
      dCoeffAtemp = 0.L;
      dCoeffBtemp = 0.L;

      for( m = 0; m < uiNumOfPoints; m++ )
      {
         dCoeffAtemp += (double)*(iValue+m) *
                        cos(2.L * PI * m * k / uiNumOfPoints);
         dCoeffBtemp += (double)*(iValue+m) *
                        sin(2.L * PI * m * k / uiNumOfPoints);
      }

      dCoeffA[k] = dCoeffAtemp / (double)uiNumOfPoints;
      dCoeffB[k] = dCoeffBtemp / (double)uiNumOfPoints;
   }
}
```

Listing 9.1: DFT-Algorithmus, Version 0

Dieses Programm kann dann getestet werden, aber dann beginnt die Arbeit wirklich! Zwei wirkliche Fragen müssen beantwortet werden, und zwar durchaus zusammenhängend:

- Ist der Algorithmus richtig implementiert? Liefert er die Rechengenauigkeit, die von ihm zu erwarten ist?
- Ist der Algorithmus effizient implementiert? Genügt er den Anforderungen an Speicherplatzminimierung und Rechenzeitminimierung?

Die Beantwortung der Fragen ist essenziell wichtig, insbesondere bei Implementierung für eingebettete Systeme.

9.2.2.2 Test des DFT-Algorithmus

Zum Test muss man sich zunächst fragen, wie man diesen anlegen soll. Schließlich gehört zum Test die Kontrolle des Ergebnisses, und die ist im Allgemeinen Kopfarbeit. Für einen einfachen Test ist es ideal, wenn man mehrere, nicht-triviale aber bekannte Beispiele nimmt, diese berechnet und mit der bekannten Lösung vergleicht.

Dies ist kein echter Test, der wesentlich mehr abdecken müsste, von einer formalen Verifikation einmal ganz abgesehen. Aber er zeigt zumindest, dass die Implementierung nicht vollkommen falsch ist.

Der Test hier ist z.B. durch synthetische bzw. bekannte (z.B. Rechteckschwingung) Testfunktionen möglich. Dies wird im Beispiel durch 2 Funktionen realisiert und das Ergebnis anschließend kontrolliert. Bei der Rechteckschwingung müssen die Koeffizienten (genauer: Die Wurzel aus der Summe der Quadrate der Koeffizienten) mit $1/(2k+1)$, $k = 0, 1, 2, 3$ ... abnehmen, alle übrigen sind 0. Von der Grundschwingung ausgehend ist also die erste signifikante Oberschwingung diejenige mit dreifacher Frequenz und Amplitude 33,3%. Dies lässt sich leicht auswerten. Bei der synthetisierten Schwingung sind die Koeffizienten natürlich bekannt.

In dem Beispiel wurde hierzu ein Rahmen geschrieben, mit dem Testdaten geschaffen werden, mit denen im Hauptprogramm die DFT-Routine aufgerufen wird. Die Ergebnisse – die ersten 100 Koeffizienten – werden in je eine Datei geschrieben und können anschließend ausgewertet werden. Die Auswertung ist in den folgenden Bildern grafisch erfolgt, einmal linear, einmal logarithmisch.

Die Bilder 9.1 bis 9.3 zeigen die Auswertung der Fouriertransformation für das Rechteck. Während die lineare Darstellung noch recht gut aussieht (hier fallen irgendwelche Abweichungen fast nie auf), muss man zur Begutachtung der Rechengenauigkeit auf die logarithmische Darstellung wechseln. Besonders interessant ist dabei die in Bild 9.3 gewählte Darstellung des Logarithmus der Abweichung vom Sollwert (als Absolutwert), denn dieser sagt wirklich etwas über die Rechengenauigkeit aus.

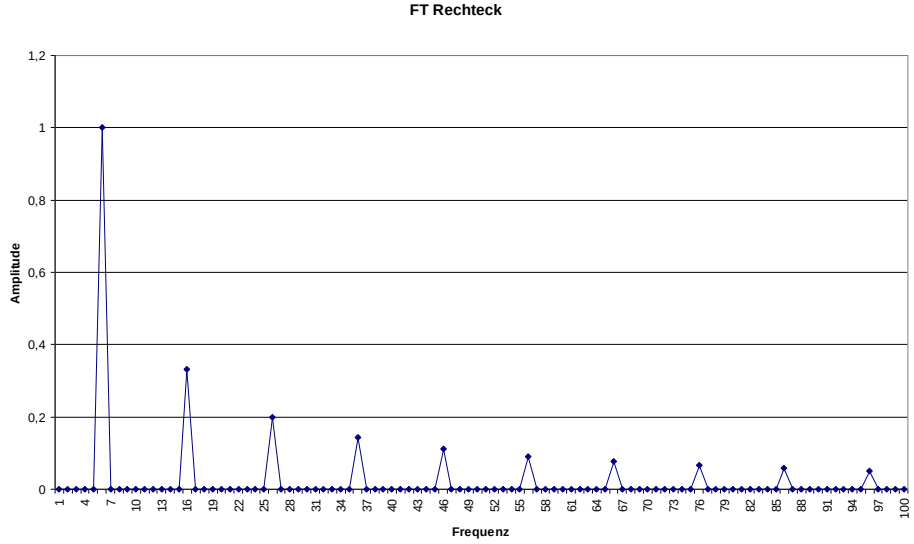


Bild 9.1 Fouriertransformation über 1000 Punkte einer Rechteckschwingung (5 Perioden), lineare Darstellung

Welche Grenze muss dabei eingehalten werden? Die Daten liegen als 12-bit-Werte vor. Zwischen einem n-bit-Wert und einem (n+1)-bit-Wert liegt der Faktor ½, d.h., zwischen zwei annehmbaren Werte im n-bit-Wert passt exakt einer des (n+1)-bit-Wertes. Mit anderen Worten: Die Auflösung erhöht sich pro Bit um den Faktor 2.

Dies wird meist logarithmisch (zur Zahl 10) dargestellt, und $\log_{10}(2) = 0{,}3010$. In der Technik hat es sich eingebürgert, dies mit dem Faktor 20 (für Leistungsmessungen) zu multiplizieren und in dB anzugeben, so dass pro Bit 6,02 dB an Auflösung gewonnen werden.

12 Bit heißt also ca. 72 dB Auflösung, also muss die Rechengenauigkeit unter dieser 72-dB-Grenze liegen. Ein kurzer Blick verrät, dass dies nicht der Fall ist, und man muss sich nun ernsthaft fragen, woran das liegt. Die Beantwortung wird auf eine Diskussion nach dem Blick auf die zweite Analyse verschoben.

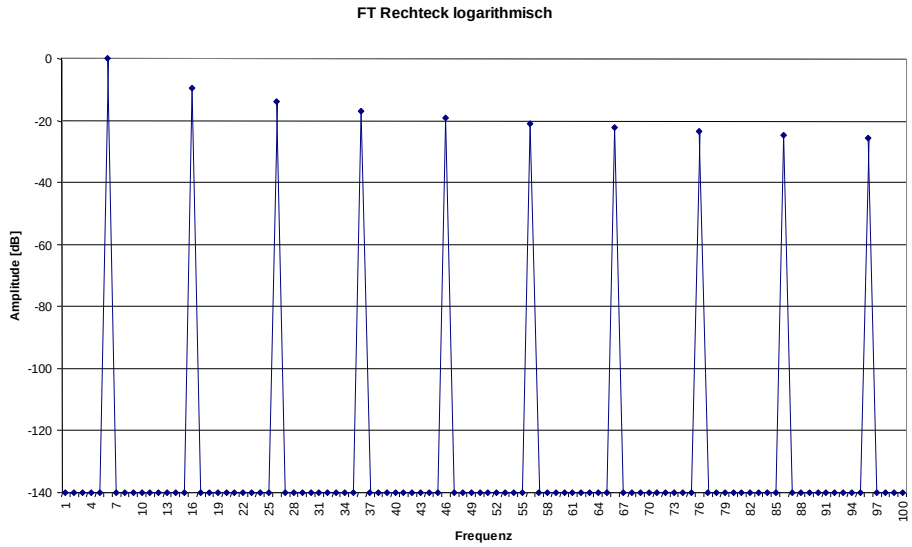


Bild 9.2 Fouriertransformation über 1000 Punkte einer Rechteckschwingung (5 Perioden), logarithmische Darstellung

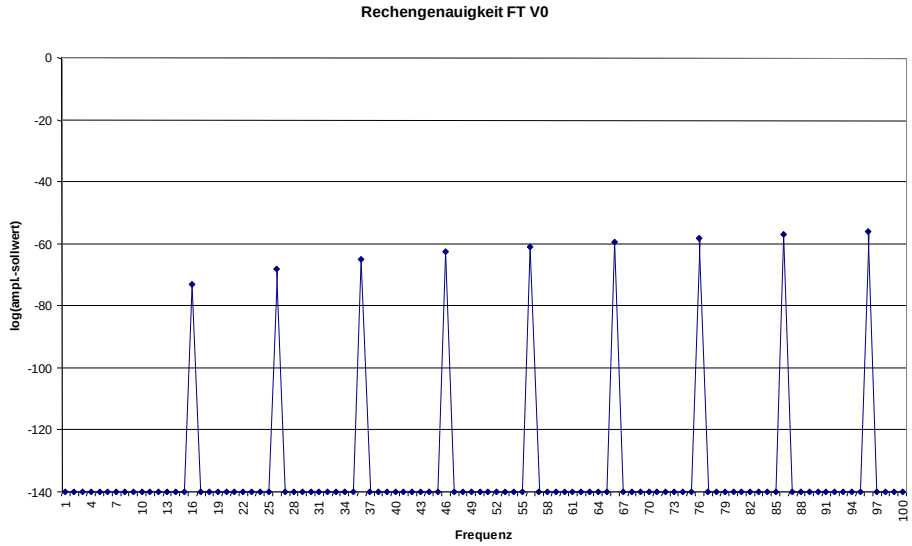


Bild 9.3 Fouriertransformation über 1000 Punkte einer Rechteckschwingung (5 Perioden), Logarithmus der Differenz zum Sollwert

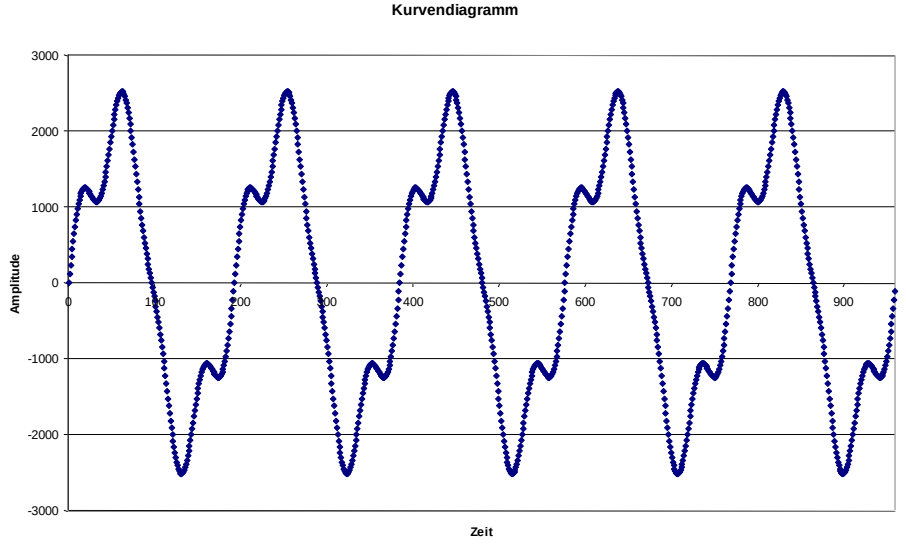


Bild 9.4 Synthetisierte Kurve mit den Koeffizienten $a_x = 0$, $b_0 = 0$, $b_1 = 1$, $b_2 = -0{,}2$, $b_3 = 0{,}1$, $b_4 = 0{,}2$, $b_k = 0$ ($k > 4$) mit 5 Perioden über 960 Punkte

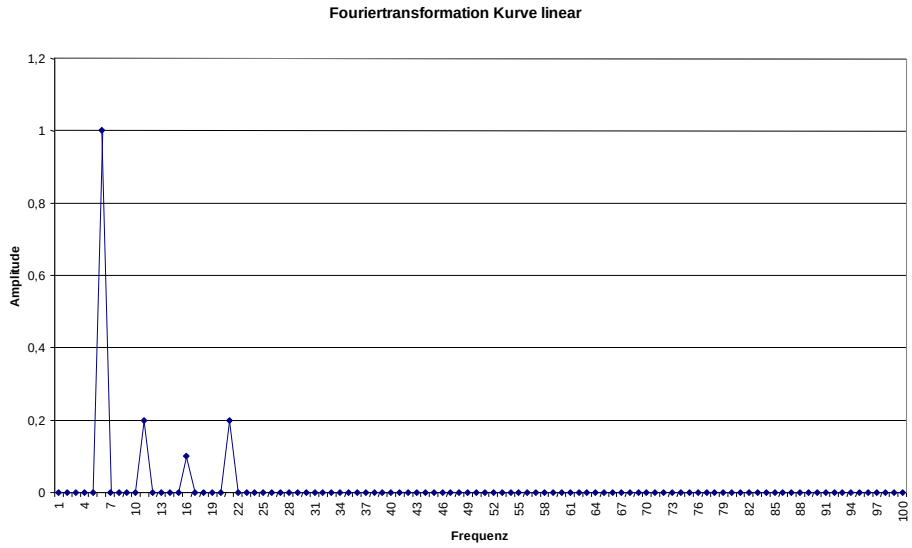


Bild 9.5 Fouriertransformation der synthetisierten Kurve, lineare Darstellung

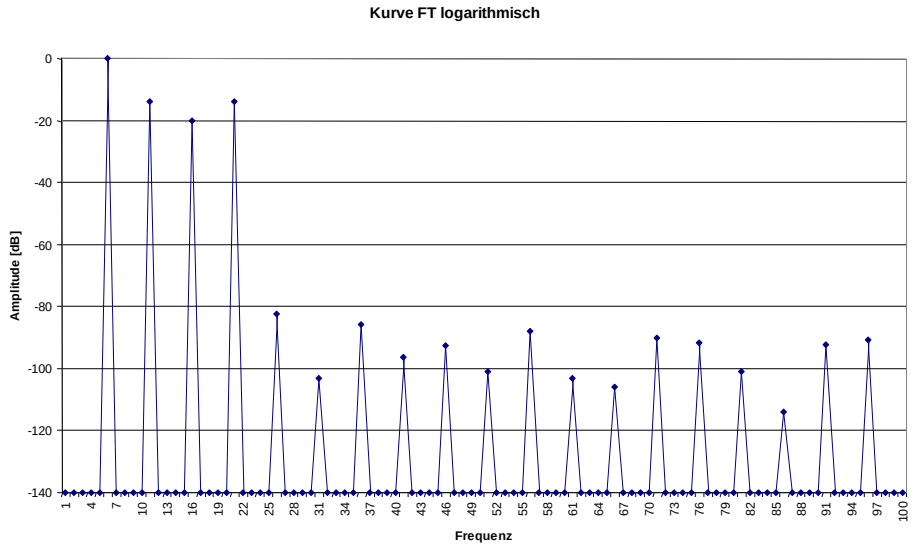


Bild 9.6 Fouriertransformation der synthetisierten Kurve, logarithmische Darstellung

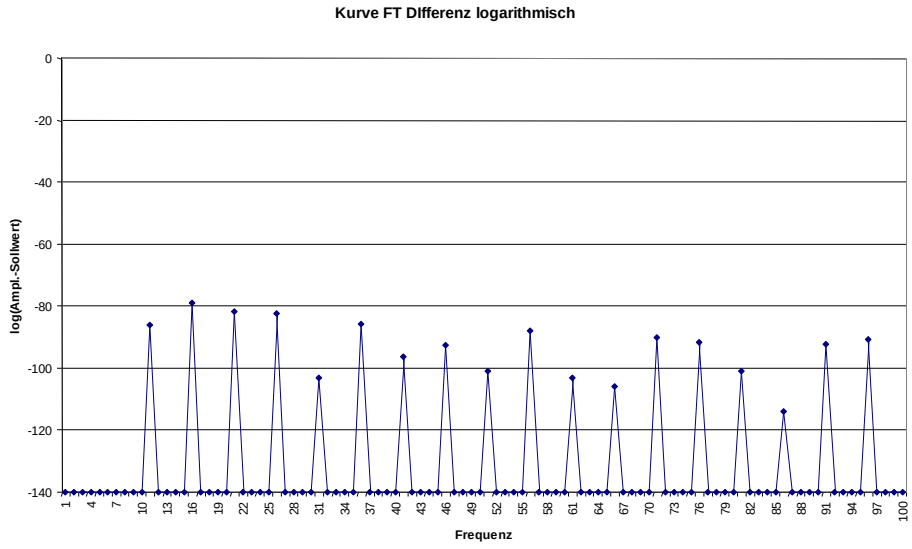


Bild 9.7 Fouriertransformation der synthetisierten Kurve, logarithmische Darstellung der Differenz zu den Sollwerten

9.2.2.3 Bewertung des Tests

Für die synthetisierte Kurve sind die Ergebnisse vollständig im Rahmen der Rechengenauigkeit, denn alles unter der 72-dB-Grenze gilt als ”Rauschen“. Lediglich die Abweichungen in den berechneten Koeffizienten für das Rechteck sind ernst zu nehmen, da diese oberhalb der Rauschgrenze liegen.

Eine der möglichen Ursachen kann darin liegen, dass die Rechteckfunktion gar nicht unter das so genannte Dirichlet-Kriterium fällt, d.h., die Funktion ist überhaupt nicht analysierbar. Hier hilft ein Test weiter, der eine angenäherte Rechteckfunktion erzeugt und analysiert.

9.2.3 Effizienz der Implementierung (Version 0) des Algorithmus

Schaut man sich einmal den Algorithmus sowie die Implementierung in Listing 9.1 an, dann kann man die Anzahl der arithmetischen Operationen gut abschätzen. Bei einer 1000-Punkte-DFT, die 100 Koeffizienten bestimmt, sind dies

$$\begin{aligned} Ops &= 100 \cdot 2 \cdot (1000 \cdot (4[Mult] + 1[Div] + 1[\sin \mid \cos] + 1[Add])) \\ &\approx 1.000.000 \end{aligned} \tag{9.10}$$

Hierbei wurde angenommen, dass die Bestimmung der Sinus- bzw. Cosinusfunktion mit 4 Operationen erfolgt, ein Wert, der bei algorithmischer Bestimmung sicherlich zu klein ist. Wenn nun ein Mikroprozessor in der Lage ist, eine Operation pro Takt auszuführen, kann die Bestimmung der 100 Koeffizienten nicht weniger als 1000000 Takte dauern. In Wirklichkeit müssen auch noch die Daten aus dem Speicher geladen und zurück geschrieben werden, so das man mindestens mit dem Faktor 10 an Overhead rechnen muss.

Bei einer Simulation dieses Programms für einen digitalen Signalprozessor (konkret: Freescale DSP 56800), der nicht unter dem Verdacht steht, besonders langsam zu sein, ergab sich eine Ausführungsdauer von

430.000.000 Takten

was umgerechnet eine Laufzeit von ca. 7 s (bei 60 MHz Taktrate) bedeutet. Dies liegt weit über den beispielhaften Anforderungen, die eine Laufzeit von ca. 100 ms betragen, und gegenüber der Anzahl der Operationen aus (9.10) ergibt sich ein Faktor von 430. Die Gründe für diese schlechte Performance sind:

- Berechnung der Sinus- bzw. Cosinuswerte in jedem Durchlauf der Schleife
- Nutzung von Floating-Point Operationen auf einem DSP oder Mikrocontroller, der hierfür keine spezielle Hardware zur Ausführung hat (z.B. Floating-Point Co-Prozessor)

Letztendlich bleibt also festzuhalten, dass die Version 0 dieses Algorithmus als Änderungsanhalt dient (und natürlich als Referenz, was die Berechnung der Koeffizienten angeht). Mehr aber auch nicht.

Merke:

Ein Kennzeichen hardwarenaher Softwareentwicklung ist es, die Ressourcen der Hardware zu kennen und so zu nutzen, dass der Mikroprozessor optimal rechnen kann. Hierzu gehört die Vermeidung von Datenformaten, die nicht von der Hardware unterstützt werden und daher umständlich emuliert werden müssen.

9.2.4 Ansätze zur Version 1 des Algorithmus

Floating-Point Operationen sollen nun vermieden werden, da der Zielprozessor keine Hardwareunterstützung dafür bietet. Hierzu werden die Zahldarstellungen in diesem Format durch Festkommazahlen (Fixpoint-Darstellung) gewählt. Es stellt sich die Frage, unter welchen Umständen das überhaupt möglich ist (automatisch ist das jedenfalls nicht)!

Hierzu kurz eine Charakterisierung der Integerzahlen: Sie zeichnen sich dadurch aus, dass zwischen zwei darstellbaren Zahlen eine immer konstante Differenz ist, nämlich 1. Dies gilt für Floating-Point Zahlen nur dann, solange man in der normierten Darstellung bei einem Exponenten bleibt (Differenz zwischen zwei "benachbarten" Zahlen $2^{(-24 + \text{Exponent})}$ im einfachen Floating-Point Format). Floating Point Zahlen besitzen damit zwei Eigenschaften: Sie sind sehr fein verteilt bei gleichem Exponenten, und durch den Exponenten überstreichen sie einen großen Zahlenbereich (wobei die Feinheit der Verteilung relativ zum Exponenten konstant bleibt).

Der springende Punkt dabei: Was benötigt man für die Sinus- und die Cosinusfunktion? Diese Funktionen liegen im Intervall [+1, -1], sie überstreichen keine Größenordnungen. Man braucht also nur dafür zu sorgen, dass ein Festkommaformat mit genügend vielen Kommastellen (zur möglichst exakten Darstellung der Werte) zur Verfügung steht.

Die Lösung heißt "Qn.m". Dieses Format drückt aus, ein Festkommaformat zu benutzen (Q) mit *n* binären Stellen vor dem Komma und *m* binären Stellen dahinter. Das gewöhnliche Integerformat mit 32 bit ist also Q32.0 für unsigned und Q31.0 für signed. Eine besonders präzise Darstellung für Sinus/Cosinus wäre also signed Q1.30, wobei die einzige Stelle vor dem Komma zur Darstellung der +1 dient. Dieses Format überführt dann die Berechnungen auf Integer-Berechnungen und hat nur einen Nachteil: Es wird nicht durch den Compiler unterstützt.

Es ist also Handarbeit angesagt. Man definiert ein bestimmtes Format, also z.B. signed Q1.30, speichert die Werte aber (scheinbar) als Integer. Als Beispiel sei dies für Sinuswerte gezeigt:

```
#define PI              3.14159265358979L
#define LIMIT           2048

#define FRACTION_BITS   30
/* This results in a Q1.30 format with sign  */
#define LIMIT_TEST      960
```

```
void main()
{
   int k;
   long int lResult;
   double dfResult;

   for( k = 0; k < LIMIT; k++ )
   {
      dfResult = sin( 2.L * PI * (double)k / (double)LIMIT );
      lResult = (long int) (dfResult * (1 << FRACTION_BITS));
      printf( „%lf  %08x\n", dfResult, lResult );
   }
}
```

Listing 9.2 Erzeugung von Sinuswerten im signed-Q1.30-Format

Die entstandenen Integerwerte (32 bit) können dann als signed-Q1.30-Format interpretiert und gespeichert werden.

9.2.4.1 Übersicht Version 0.8

In der Version 0.8 werden erste Änderungen an Version 0 durchgeführt, die dann eine effizientere Berechnung ermöglichen. Diese Änderungen beinhalten folgendes:

- Alle Rechnungen werden als Integerberechnungen durchgeführt
- Eine (Muster-)Sinuskurve wird in Form von 513 (= 512 + 1) Werten im signed-Q1.30-Format gespeichert, und zwar nur das erste Viertel (weil sich daraus alle anderen ergeben). Die vollständige Kurve hat dann 2048 Werte.
- Der Zugriff auf diese Wertetabelle ist durch Zugriffsfunktionen wie `i32GetCosineValue( uint16 ui16Index )` gekapselt. Dies ist eine Anleihe aus der objektorientierten Programmentwicklung.
- Für die DFT sind eine Sinus- und eine Cosinustabelle notwendig, deren Periode exakt gleich der Anzahl der Messpunkte ist. Diese wird zu Beginn berechnet (interpoliert), und anschließend werden die jeweiligen Werte von dort mithilfe der Funktionen `(i32GetSineTableValue( uint16 ui16Index_k, uint16 ui16Index_m, uint16 ui16NumOfPoints ), i32GetCosineTableValue( uint16 ui16Index_k, uint16 ui16Index_m, uint16 ui16NumOfPoints )` gelesen.

```
/*************************************************************/
/*                                                           */

int32 i32GetCosineValue( uint16 ui16Index )
{
   if( ui16Index > 4 * NUM_OF_POINTS )
      ui16Index = (uint16)(ui16Index % (4 * NUM_OF_POINTS));
```

```
   if( ui16Index < NUM_OF_POINTS )
      return i32SinTab[NUM_OF_POINTS - ui16Index];
   else if( ui16Index < 2 * NUM_OF_POINTS )
      return (0 - i32SinTab[ui16Index - NUM_OF_POINTS]);
   else if( ui16Index < 3 * NUM_OF_POINTS )
      return (0 - i32SinTab[3 * NUM_OF_POINTS - ui16Index]);
   else
      return (i32SinTab[ui16Index - 3 * NUM_OF_POINTS]);
}
```

Listing 9.3 i32GetCosineValue()

Die DFT-Routine lautet nun (Ausschnitt):

```
int32 i32GetInterpolationSineValue( uint16 ui16Index,
       uint16 ui16NumOfPoints )
{
   uint16 ui16Bottom, ui16Fraction;
   int32 i32Sine0, i32Sine1;

   ui16Bottom = (uint16) ((((uint32)ui16Index) <<
               (NUM_OF_INDEX_BITS+2))
               / ui16NumOfPoints);
   ui16Fraction = (uint16) ((((uint32)ui16Index) <<
               (NUM_OF_INDEX_BITS+2)) % ui16NumOfPoints);

   i32Sine0 = i32GetSineValue( ui16Bottom );
   i32Sine1 = i32GetSineValue( (uint16)(ui16Bottom + 1) );
   return ((i32Sine0 + ((i32Sine1 - i32Sine0) *
               (int32)ui16Fraction
            / (int32) ui16NumOfPoints)) >> SCALE_FACTOR);
}
```

Listing 9.4 Interpolations-Routine Version 0.8

```
void vComputeDFT( uint16 ui16NumOfPoints, int16 *i16Value )
{
   uint16 k, m, ui16Index;
   int32 i32CoeffATemp, i32CoeffBTemp;

   for( k = 0; k < NUM_OF_COEFFICIENTS; k++ )
   {
      i32CoeffATemp = 0;
      i32CoeffBTemp = 0;

      for( m = 0, ui16Index = 0; m < ui16NumOfPoints; )
      {
         i32CoeffATemp += *(i16Value+m) *
                          i32CosineTable[ui16Index];
         i32CoeffBTemp += *(i16Value+m) * i32SineTable[ui16Index];
         m++;
         ui16Index = (ui16Index + k) % ui16NumOfPoints;
      }
```

```
            i32CoeffA[k] = i32CoeffATemp;
            i32CoeffB[k] = i32CoeffBTemp;
        }
    }
```

Listing 9.5 DFT-Routine Version 0.8

9.2.4.2 Auswertung der Ergebnisse DFT-Routine Version 0.8

Die folgenden Bilder 9.8 und 9.9 zeigen die Auswertung der DFT-Routinen für die beiden bekannten Testkurven. Das Ergebnis ist außerordentlich ernüchternd, denn die Kurven zeigen ein komplett anderes Verhalten gegenüber Version 0, die als richtig identifiziert wurde.

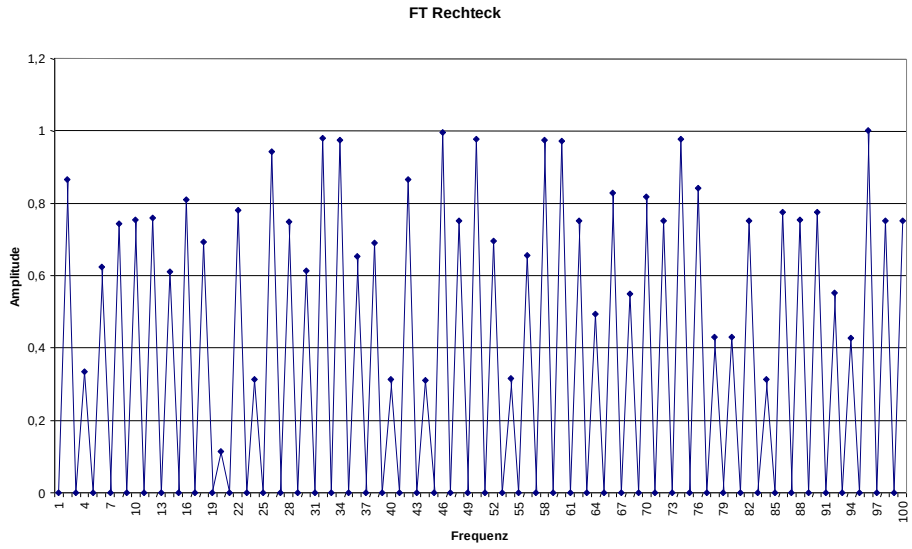


Bild 9.8 Diskrete Fouriertransformation Rechteck, Algorithmus Version 0.8

Hier muss also der Grund identifiziert werden. Es liegt nun nahe, die Schuld komplett dem Wechsel ins Q1.30-Format zuzuschreiben und zurück zu wechseln, was aber mit den erwähnten Problemen in der Rechenzeit behaftet sein wird.

Ein vergleichsweise einfacher Test erlaubt eine erste Diagnose: Im Programm wird die Anzahl der Bits, die für einen Sinuswert genutzt werden, von 31 auf 13 (also signed-Q1.12-Format) zurückgesetzt. Die Auflösung ist damit identisch zu derjenigen der Messwerte. Bild 9.10 zeigt den Effekt.

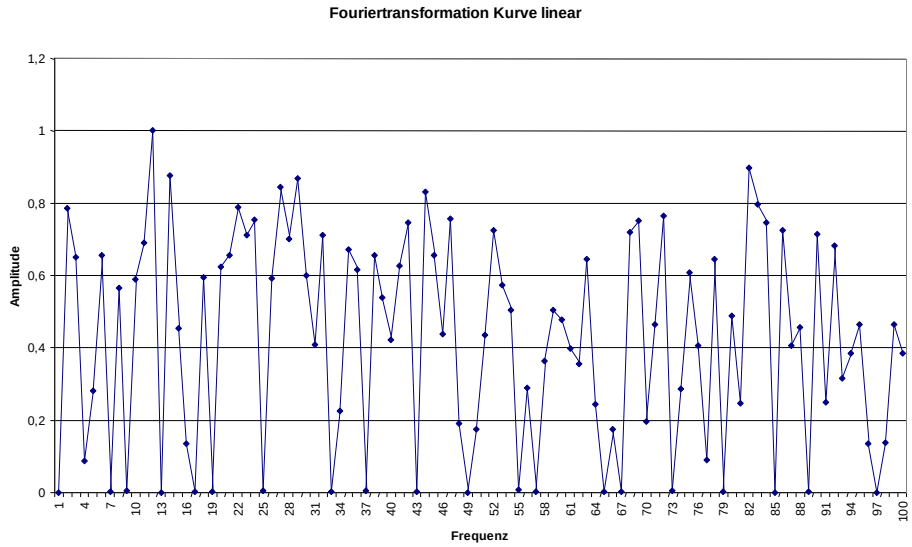


Bild 9.9 Diskrete Fouriertransformation der synthetisierten Kurve, Algorithmus Version 0.8

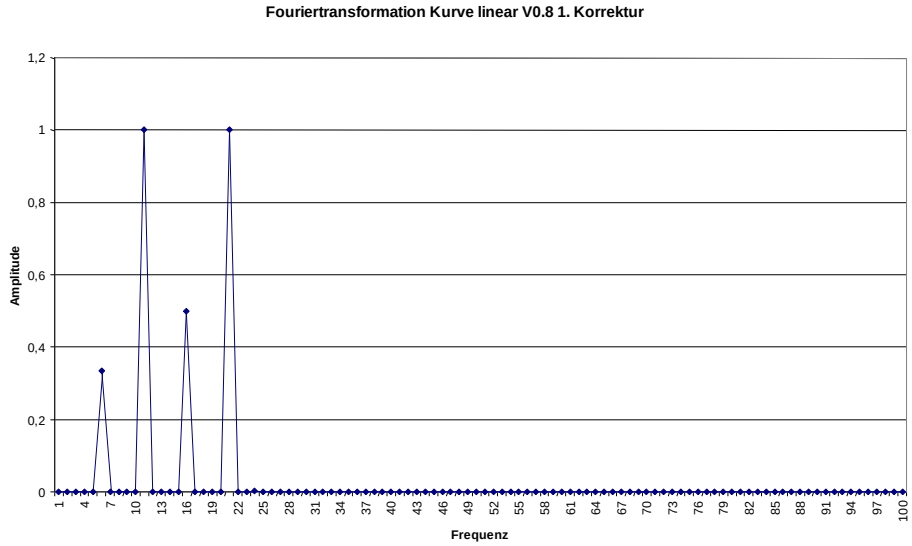


Bild 9.10 Diskrete Fouriertransformation der synthetisierten Kurve, Algorithmus Version 0.8, 1. Korrektur

Die durch diese erste Korrektur erhaltene DFT entspricht schon besser dem Vergleichsspektrum, ist jedoch nicht perfekt. Woran liegt es denn nun, dass man bei schlechterer Auflösung der Sinussignale ein besseres Ergebnis erhät? Die zweite Korrektur, in Bild 9.11 logarithmisch für das Rechteck dargestellt, ist dann schon nahezu perfekt, hier wird im signed-Q1.8-Format für Sinus und Cosinus gearbeitet.

Der Nachteil: Die Rechengenauigkeit leidet. Der tiefere Grund für die Berechnungsfehler liegt in einem Überlauf der Werte, bedingt durch die verwendete 32-bit-Arithmetik bei Integer. Ausweg: 48- oder 64-bit-Arithmetik, vor allem aber:

Der Softwaredesigner ist verpflichtet, die benötigte Rechengenauigkeit einzuhalten und insbesondere den Überlauf bei Integerrechnungen, der zu schwerwiegenden Rechenfehlern führen kann, zu verhindern.

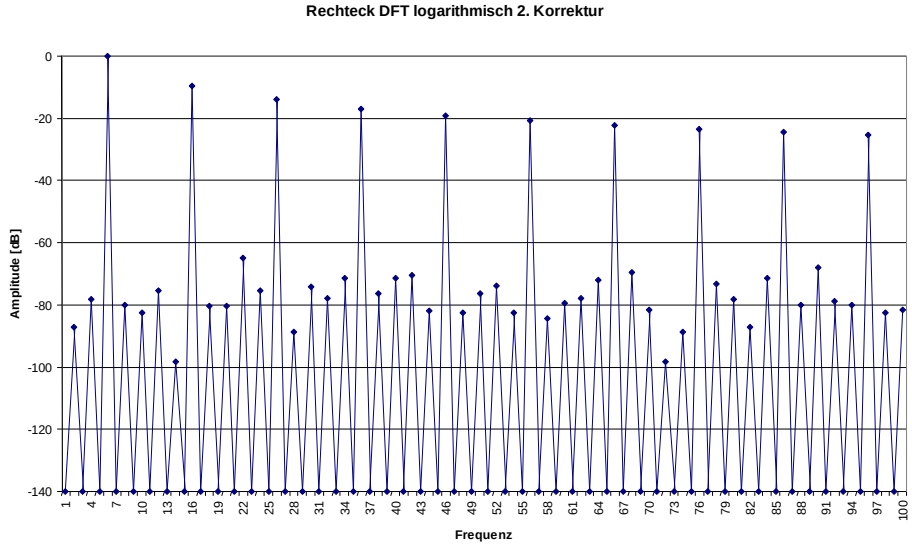


Bild 9.11 Diskrete Fouriertransformation der Rechteckfunktion, Algorithmus Version 0.8, 2. Korrektur

9.2.4.3 Der Weg zur Version 1: 64-bit-Arithmetik

Es gibt in diesem Beispiel keinen wirklichen Kompromiss zwischen Rechengenauigkeit und Korrektheit der Ergebnisse, solange im 32-bit-Format verweilt wird. Konsequenterweise muss also ein größeres Datenformat gewählt werden, um das Problem wirklich zu lösen.

Ein Ansatz besteht darin, den Datentyp für 64 bit zu wählen. Dieser wird meist als `long long` bezeichnet, ist aber keinesfalls immer vorhanden. Hier wird ein anderer Weg beschritten, der auf folgenden Regeln beruht:

- Die Multiplikation einer n- mit einer k-bit-Zahl ergibt eine (n+k)-bit-Zahl
- Die Addition einer n-bit- und einer k-bit-Zahl gibt eine (max(n, k)+1)-bit-Zahl; ist n == k, so ist das Ergebnis eine (n+1)-bit-Zahl.

Angenommen, die Messwerte sind mit 12 bit codiert, und für die Sinus- bzw. Cosinuswerte werden 16 bit gewählt. Eine Multiplikation hat dann 28 bit, und bis zum Maximum von 31 bit (das 32. Bit ist das Vorzeichenbit) kann man 8mal summieren. Nach 8 Aufsummierungen muss es dann einen Übertrag geben. Dies ist im folgenden Codeabschnitt eingeführt.

```
#define MEASUREMENT_PRECISION 12
                              /* Number of bits for measurement */
#define COSINE_PRECISION      16 /* Sine/Cosine precision       */
#define NUM_OF_SUM_BITS       31 -
                        (MEASUREMENT_PRECISION + COSINE_PRECISION)
/* Number of free bits for summing up */
#define SUM_MODULO   (1 << NUM_OF_SUM_BITS)
               /* This is used for managing data overflow   */
#define SCALING_HIGH       MEASUREMENT_PRECISION +
                              COSINE_PRECISION - 20
               /* Shift right for high part of the sum   */
#define MASKING_LOW
  (0xFFFFFFFFU >> (52 - MEASUREMENT_PRECISION - COSINE_PRECISION))
               /* Mask for the low bit part  */

/*****************************************************************/
/*                                                               */
/* Function:      vComputeDFT                                    */
void vComputeDFT( uint16 ui16NumOfPoints, int16 *i16Value )
{
uint16 k, m, ui16Index;
int32 i32CoeffAHigh, i32CoeffBHigh;
int32 i32CoeffATemp, i32CoeffBTemp;
uint16 ui16CoeffALow, ui16CoeffBLow;

for( k = 0; k < NUM_OF_COEFFICIENTS; k++ )
   {
i32CoeffAHigh = 0;
ui16CoeffALow = 0;

i32CoeffBHigh = 0;
ui16CoeffBLow = 0;

i32CoeffATemp = 0;
```

```
i32CoeffBTemp = 0;

for( m = 0, ui16Index = 0; m < ui16NumOfPoints; )
      {
         i32CoeffATemp += *(i16Value+m) *
                          i32CosineTable[ui16Index];
i32CoeffBTemp += *(i16Value+m) * i32SineTable[ui16Index];
m++;
ui16Index = (ui16Index + k) % ui16NumOfPoints;
if( 0 == (m % SUM_MODULO) )
         {
i32CoeffAHigh += (i32CoeffATemp >> SCALING_HIGH);
            ui16CoeffALow += (uint16)(i32CoeffATemp &
                                          MASKING_LOW);

i32CoeffBHigh += (i32CoeffBTemp >> SCALING_HIGH);
            ui16CoeffBLow += (uint16)(i32CoeffBTemp &
                                          MASKING_LOW);

i32CoeffATemp = 0;
i32CoeffBTemp = 0;
         }
      }
/* The next four codelines add the 'rest' of the computation to
   the coefficients, if ui16NumOfPoints is not zero modulo
   SUM_MODULO */
      i32CoeffAHigh += (i32CoeffATemp >> SCALING_HIGH);
      ui16CoeffALow += (uint16)(i32CoeffATemp & MASKING_LOW);

      i32CoeffBHigh += (i32CoeffBTemp >> SCALING_HIGH);
      ui16CoeffBLow += (uint16)(i32CoeffBTemp & MASKING_LOW);

      i32CoeffAHigh += (int32)(ui16CoeffALow >> SCALING_HIGH);
      i32CoeffBHigh += (int32)(ui16CoeffBLow >> SCALING_HIGH);

      i32CoeffA[k] = i32CoeffAHigh;
      i32CoeffB[k] = i32CoeffBHigh;

   }
}
```

Listing 9.6 DFT-Routine Version 1

Dieses Verfahren hat den Vorteil, dass nur so oft wie benötigt der Übertrag summiert wird (im Beispiel alle 8 Rechnungen bei 16 bit Cosinus-Präzision). Zum Schluss wird wieder alles auf 32 bit skaliert, was durchaus nicht immer sein muss, falls die volle Präzision der Ergebnisse erhalten bleiben soll.

Zum Schluss die Ergebnisse der Version 1, also die erste gültige DFT-Routine mit Nutzung eines Integer-Formats.

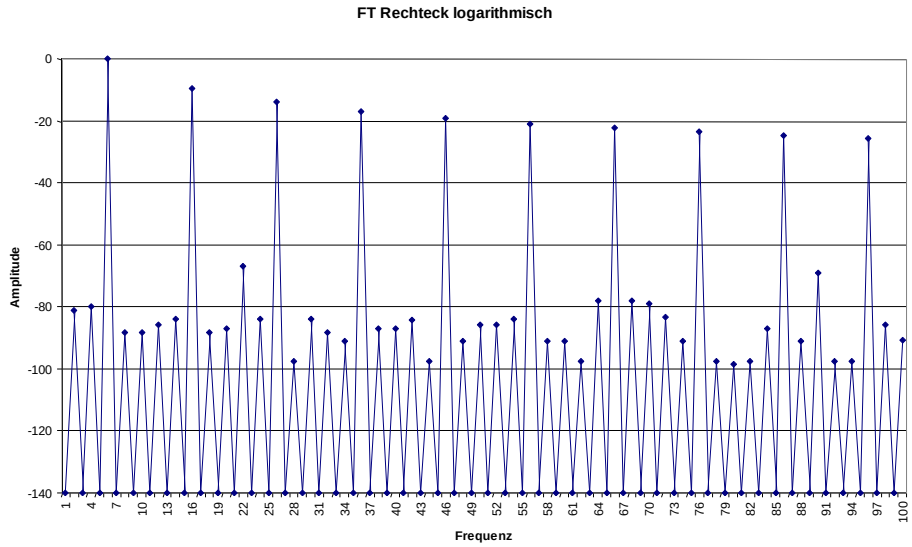


Bild 9.12 Diskrete Fouriertransformation der Rechteckfunktion, Algorithmus Version 1, logarithmische Darstellung

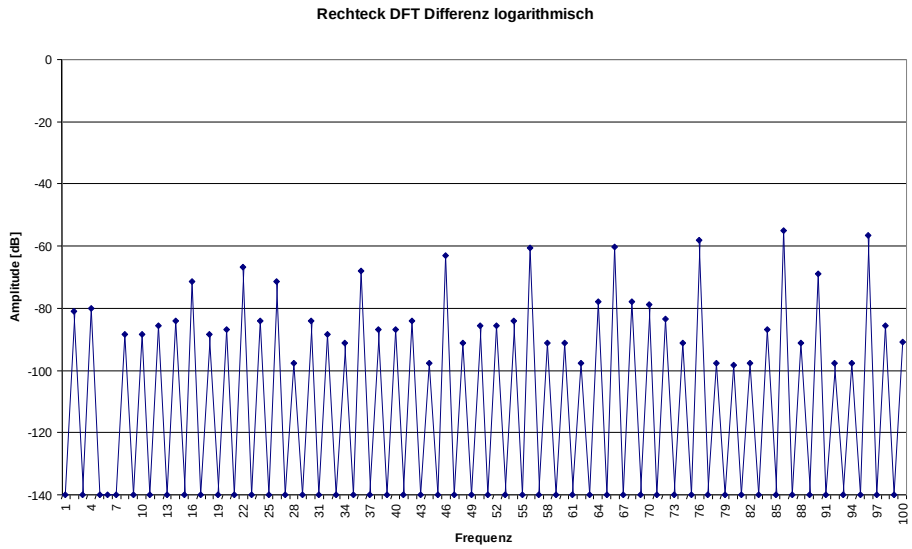


Bild 9.13 Diskrete Fouriertransformation der Rechteckfunktion, Algorithmus Version 1, logarithmische Darstellung

Nun etwas präziser die synthetisierte Kurve:

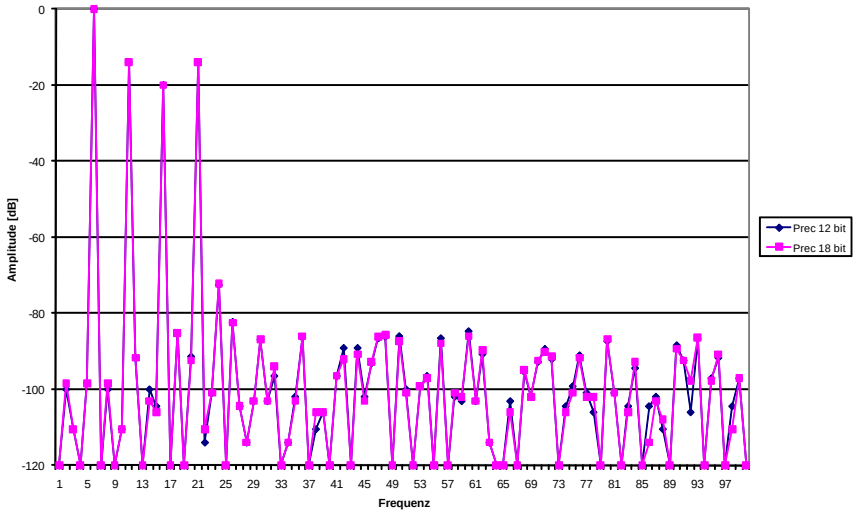


Bild 9.14 Diskrete Fouriertransformation der synthetisierten Kurve, Algorithmus Version 1, Rechengenauigkeit für Sinus/Cosinus 12 und 18 bit, logarithmische Darstellung

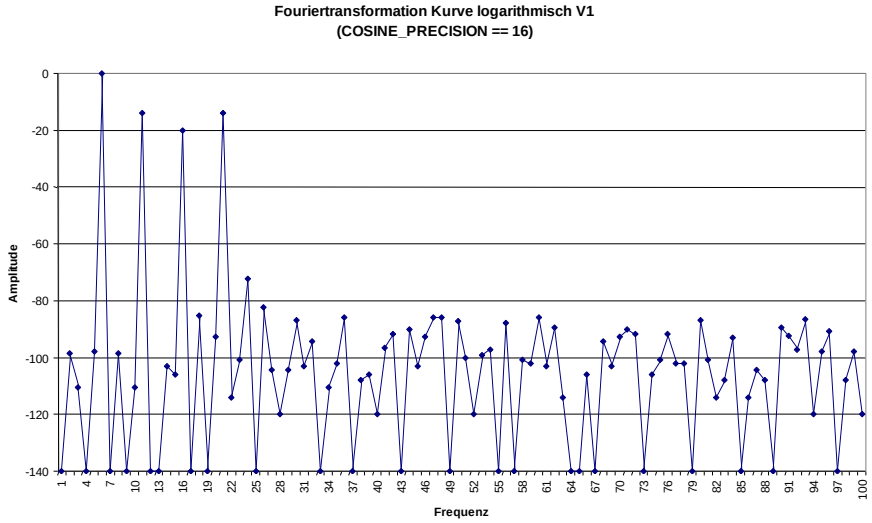


Bild 9.15 Diskrete Fouriertransformation der synthetisierten Kurve, Algorithmus Version 1, Rechengenauigkeit für Sinus/Cosinus 16 bit, logarithmische Darstellung

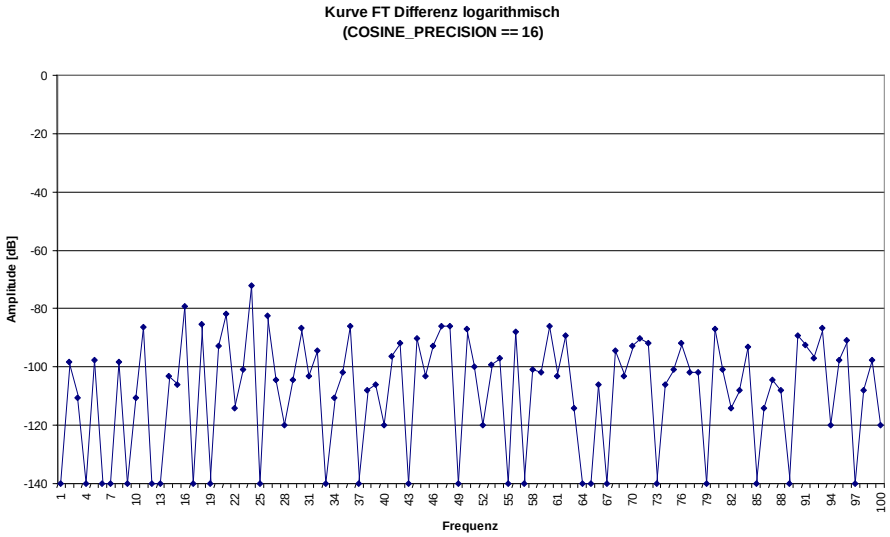


Bild 9.16 Diskrete Fouriertransformation der synthetisierten Kurve, Algorithmus Version 1, Rechengenauigkeit für Sinus/Cosinus 16 bit, logarithmische Darstellung der Differenz zu Sollwerten

Wie sich dabei zeigt, ist eine Genauigkeit der Sinus/Cosinusdarstellung mit 12 bit durchaus ausreichend (Bild 9.14). Darunter würde die Auflösung künstlich verschlechtert werden. 16 bit erscheinen als richtige Wahl (empirisch).

9.2.4.4 Performance der Version 1

Abschließend muss die Performance dieser Version 1 beurteilt werden. Pro berechnetem Koeffizienten benötigt der Algorithmus pro Punkt 8 Additionen, 2 Multiplikationen und 1 Division, ferner 3 Tabellenzugriffe, zusammen also 14 Operationen, abgesehen von der Übertragsrechnung, die bei einem Übertrag pro 8 Punkte immerhin 4 Additionen, 4 Logikoperationen und 1 Division betragen.

Eine Simulation für den DSP56F800, getaktet mit 60 MHz, ergibt dies rund 12 ms pro Koeffizient, bei 1000 Punkten, bei einer Anzahl von 15125 Operationen pro Koeffizient (also 48 Takte pro Operation). Dies ist ein unvermutet schlechtes Ergebnis, denn das Ziel ist eigentlich ca. 100 ms für 100 Koeffizienten, und in dieser Version sind es 1,2 s.

Somit ist der Grundstein für Version 2 gelegt: Optimierung in Richtung Performance.

9.2.5 Version 2: Optimierung der Operationen

9.2.5.1 Umsetzung in Performanceverbesserung

Eine Analyse per Profiler bringt es schnell an den Tag: Es sind in diesem Fall die Divisionen, die einen erheblichen Anteil an der Laufzeit ausmachen (hier: ca. 90 % der gesamten Laufzeit). Die Division tritt zwar nur wenige Male im Code auf, ist dort jedoch deshalb so rechenintensiv, weil keine Hardwareeinheit im DSP vorhanden ist, um die Operation durchzuführen. Vielmehr muss die Division auf eine Emulation durch eine Softwareroutine zurückgeführt werden.

Die entscheidenden Zeilen stehen in den Funktionen `vComputeDFT()`, `i32GetCosineTableValue()` und `i32GetSineTableValue()`. In diesen Funktionen wird jeweils eine Modulo-Division mit Nutzung des Rests genutzt, und – wesentlich wichtiger – diese Division wird pro Schleife einmal durchlaufen. Mit anderen Worten: Hier steckt der wesentliche Anteil der Rechenzeit. Die Zeilen im Einzelnen:

```
ui16Index = (ui16Index + k) % ui16NumOfPoints;
```

und

```
if( ui16Index > 4 * NUM_OF_POINTS )
ui16Index = ui16Index % (4 * NUM_OF_POINTS);
```

Letztere Zeile tritt in den beiden Funktionen `i32GetCosineTableValue()` und `i32GetSineTableValue()` auf. Für diese drei Modulo-Divisionen muss nun ein Ersatz gefunden werden, wenn die Applikation beschleunigt werden soll.

Da hier jeweils nur der Rest der Division genutzt werden soll, bietet sich eine sukzessive Subtraktion an, bis das Ergebnis gerade noch positiv ist. Dies kann für die zweite Zeile so erfolgen:

```
if( ui16Index > (NUM_OF_POINTS << 2 ) )
while( ui16Index >= (NUM_OF_POINTS << 2) )
ui16Index -= (NUM_OF_POINTS << 2);
```

Die Multiplikation `4 * NUM_OF_POINTS` wird hierbei durch einen Shift-Links-Befehl (`NUM_OF_POINTS << 2`) ersetzt, ein Vorgang ohne Bedeutung, da beide entstehenden Konstanten zur Compilezeit berechnet werden. Wichtig für die Laufzeit ist nur die `while`-Schleife: Diese wird solange durchlaufen, wie der Index ui16Index größer als `4 * NUM_OF_POINTS` ist, und bei jedem Durchlauf wird exakt diese Konstante subtrahiert. Ist die Variable noch positiv, aber kleiner als `4 * NUM_OF_POINTS`, wird die Schleife abgebrochen, da beim nächsten Durchlauf das Resultat negativ wäre.

Für die erste Zeile gibt es eine noch einfachere Lösung, deren Bestimmung aber nicht so trivial ist, da Überlegungen zu den möglichen Datenwerten notwendig sind. Die korrekte Übersetzung wäre

```
ui16Index += k;
while( ui16Index >= ui16NumOfPoints )
ui16Index -= ui16NumOfPoints;
```

Nun gelten aber einige Größenbeziehungen zwischen den Variablen:

$$k < \text{NUM\_OF\_COEFFICIENTS} \tag{9.11}$$

$$\text{NUM\_OF\_COEFFICIENTS} <= \text{ui16NumOfPoints} \tag{9.12}$$

woraus sich

$$k < \text{ui16NumOfPoints} \tag{9.13}$$

schließen lässt. Da nun aber `ui16Index` gerade um den aktuellen Wert von `k` erhöht wurde, gilt weiterhin

$$\text{ui16Index} < 2^* \text{ ui16NumOfPoints} \tag{9.14}$$

Durch die Beziehung (9.14) kann die `while`-Schleife durch eine einfache `if`-Bedingung ersetzt werden:

```
ui16Index += k;
if( ui16Index >= ui16NumOfPoints )
ui16Index -= ui16NumOfPoints;
```

Es sei hier nochmals darauf verwiesen, dass solche speziellen Umsetzungen nur unter bestimmten Randbedingungen, die die Daten einhalten müssen, möglich sind.

9.2.5.2 Performance der Version 2

Ein Profiling der Versionen 1 und 2 auf dem PC ergibt eine Verringerung der Laufzeit auf ca. 60 % durch Einführung der Subtraktion anstelle der Division. Die Verbesserung der Performance beträgt also ca. Faktor 1,66. Beim bereits erwähnten DSP sind die Unterschiede wesentlich drastischer, die Laufzeit für eine 100-Punkte-DFT bei 1000 Punkte als Messbasis verringert sich auf 120 ms, also nur noch 1/10 der bisherigen, so dass das Performanceziel nahezu erreicht ist.

9.3 Interrupt-Service-Routinen

9.3.1 Einführung: Interrupt Requests

Interrupt Requests, meist mit IRQ abgekürzt, stellen Unterbrechungsanforderungen dar, die im engeren Sinn von einer externen Hardware generiert werden. So kann z.B. eine Kommunikation fertig sein (Zeichen wurde gesendet oder empfangen und soll nun nachgeladen bzw. abgeholt werden), ein Timer kann ablaufen (”Zeit zum Holen eines Messwerts“), oder bestimmte Aktionen sollen zwingend beginnen (”Notaus“).

Interrupt Requests stellen somit ein probates Mittel zur Kopplung von Rechner und Außenwelt dar, die in einem Multitasking-Ansatz für die Software genutzt werden kann. Hieraus ggf. resultierende Probleme wurden in Kapitel 3.1 behandelt.

Im weiteren Sinn kann ein IRQ auch aus dem Programm heraus erzeugt werden, dies wird dann Software-Interrupt, Exception, Supervisor Call oder ähnlich genannt. Der wesentliche Unterschied zwischen dem extern erzeugten Interrupt und dem internen besteht in der Asynchronität des Ereignisses: Der Softwareaufruf steht im Speicher an einer bestimmten Stelle, der Hardwareinterrupt kann zu jeder Zeit kommen.

Tabelle 9.1 Überblick zu den verschiedenen Interruptkategorien

Kategorie	Auslöser	Eigenschaften
Hardware-IRQ	Spezielle Hardware	Asynchron zum Programmlauf, gewollte Unterbrechung
Software-IRQ	Aufruf im Programm per Assemblerbefehl	Synchron zum Programmlauf, ähnlich einem Funktionsaufruf
Exception	Fehler im Programmablauf, z.B. Division durch 0, unerlaubter Speicherzugriff	Je nach Art kann dies gewollt sein (z.B. „Seitenfehler“ einer Memory Management Unit, d.h. Nachladen gewünscht), oder es ist ein wirklicher Programmfehler, der zu einem Abbruch führt.

Weiterhin können Exceptions durch Programmfehler entstehen, etwa durch eine Division durch 0 oder ähnliche nicht erlaubte Aktionen. Diese Aktionen sind dann nicht gewollt und damit weder asynchron noch synchron zum Programmlauf. Tabelle 9.1 gibt einen Überblick.

Warum sind diese Unterscheidungen wichtig? Nun, jeder Interrupt Request erfordert eine Softwareroutine, Interrupt Service Routine (ISR), die diesen IRQ entsprechend behandelt. Die Entwicklung solcher Routinen zählt mehr zur Systementwicklung, da es sich hierbei um die Kopplung verschiedener Elemente (z.B. Soft- und Hardware oder übergeordnete Fehlerbehandlung) handelt. Die unterschiedlichen Kategorien erfordern aber auch teilweise unterschiedliche Strategien zum Schreiben der ISR.

9.3.2 Tipps für die ISR-Programmierung

9.3.2.1 Allgemeines

Der Mikroprozessor reagiert im Allgemeinen einheitlich auf alle Unterbrechungen – unabhängig von der Kategorie. Die eingehende Unterbrechung wird beim Hardware-IRQ zunächst gespeichert (Flipflop für das Unterbrechungssignal), bei den Software-IRQs und den Exceptions ist dies größtenteils nicht notwendig.

Im zweiten Schritt wird geprüft, ob die Unterbrechungsbehandlung zulässig ist oder nicht. Dies ist notwendig, um ununterbrechbare Programmteile schaffen zu können. Diese werden manchmal auch als ”atomar“ bezeichnet. Die Sperrung einer Unterbrechung erfolgt im einfachsten Fall durch ein ”Interrupt Disable Flag“ (oder ein ”Interrupt Enable Flag“ mit umgekehrter Bedeutung), das – falls auf ’1’ gesetzt – die weitere Behandlung sperrt. Durch die Speicherung des Signals kommt es zu keinem Verlust, es sei denn, ein weiterer IRQ tritt auf und müsste in dem gleichen Flipflop gespeichert werden.

In komplexeren Systemen gibt es meist eine Hierarchie von Sperrungen bzw. Zulassungen: Bestimmte Unterbrechungen können z.B. immer zulässig sein (NMI, ”Non-Maskable Interrupt“) bis hin zu einem Prioritätssystem. Die Notwendigkeiten sind hierbei immer abhängig von dem geplanten System.

Ist die Behandlung zugelassen, dann wird der ggf. noch in Bearbeitung befindliche Befehl beendet, und im Anschluss daran nimmt der Prozessor die Abarbeitung der ISR auf. Hierzu muss zumindest die Rücksprungadresse (wie beim Funktionsaufruf), also die nächste Programmstelle, ab der das bisherige Programm fortgesetzt werden muss, gespeichert werden, meist wird auch noch das Statusregister des Prozessors gesichert. Sicherungsort ist meist der Stack, ggf. auch spezielle Register.

Alles Weitere ist dann Sache der ISR. Aus dem geschilderten Verhalten ergeben sich zwei wichtige Konsequenzen:

- Alle Einheiten, die in einer ISR benutzt werden, müssen grundsätzlich vor Benutzung gesichert und vor dem Rücksprung wieder restauriert werden, damit das eigentliche Programm korrekt weiter arbeiten kann. Dies ist Aufgabe des Softwareentwicklers, die Prozessorhardware bietet hier im Allgemeinen keinen Support.
- Es handelt sich bei der ISR im Allgemeinen um keinen Funktionsaufruf, dementsprechend können auch keine Aufrufparameter mitgegeben werden. Ein Rückgabewert fehlt ebenfalls. Ausnahme hiervon sind die Software-ISR, die planmäßig aufgerufen werden und damit gezielt Werte übergeben können.

In allen anderen Fällen muss die Kommunikation mit anderen Routinen daher über Kommunikationspuffer erfolgen (→ 3.3).

Interrupt-Service-Routinen zur Behandlung der Hardware-IRQs (und auch der Exceptions) gelten allgemein als schwierige Kandidaten bei der Softwareentwicklung. Dies liegt insbesondere daran, dass sie nicht im gewöhnlichen (sequenziellen) Programmablauf eingebunden sind, vielmehr können sie durch Hardwaresignale (Interrupt Requests) zur Ausführung gebracht werden – somit komplett asynchron zum gesamten Programmablauf.

Dies hat einigen Einfluss auf die Aktionen, die zur korrekten Entwicklung von Interrupt-Service-Routinen durchgeführt werden müssen. Im Folgenden sind drei besondere Probleme aufgegriffen: Die Sicherung (und Restaurierung) aller in der ISR verwendeten Register, die Kommunikation der ISR mit dem eigentlichen Programm (soweit benötigt), sowie die Nutzung externer Funktionen in der ISR.

9.3.2.2 Sicherung aller benötigten Register

Die Sicherung der in der ISR benötigten Register erfolgt entweder durch einen so genannten Bankswitch (Umschalten zwischen verschiedenen Registerbänken, d.h. es existieren so genannten Schattenregister) oder durch reales Sichern auf dem Stack. Dies muss zu Beginn der ISR geschehen (Stack: PUSH-Befehle), und am Schluss, also unmittelbar vor Rücksprung, müssen die Inhalte in umgekehrter Reihenfolge vom Stack in die Register kopiert werden (PULL- oder POP-Befehle).

Im Idealfall baut der Compiler diese Sicherung ein, im schlechteren Fall muss sich der Softwareentwickler selbst darum kümmern. Hier ein Beispiel für die Atmel ATmega-Mikrocontroller:

```
interrupt [USART_RXC] void usart_rx_isr( void )
{
            …
}
```

Listing 9.7a C-Code für Interrupt-Service-Routine

```
.CSEG
```

```
_usart_rx_isr:
            ST   -Y,R17 ; save the registers R16 and R17
            ST   -Y,R16

… ; here it is assumed that only R16/R17 are used within ISR
            LD   R16,Y+ ; restore the registers
            LD   R17,Y+
RETI
```

Listing 9.7b Assemblercode für Interrupt-Service-Routine

Das eigentliche Problem besteht darin, einen guten Kompromiss zwischen Sicherheit (= Speichern aller Register) und Performance (= Sichern von möglichst wenig Registern) zu finden. Einige Compiler beherrschen dies, beispielsweise der für den Atmel ATmega-Mikrocontroller gewählte CodeVision AVR C-Compiler. Kann der Compiler dies nicht komplett analysieren, muss er sehr konservative Annahmen machen und letztendlich alles sichern.

9.3.2.3 Kommunikation der ISR mit anderen Programmen

Die Kommunikation der Interrupt-Service-Routine mit anderen Programmen ist ein Problem, weil es sich hier nicht um eine einfache Funktion handelt. Die gezielte Übergabe von Parametern wie bei Unterprogrammen und die gezielte Rückgabe müssen zwangsläufig entfallen, weil der Aufruf der ISR an beliebiger Stelle erfolgen kann. Aufrufwerte müssen aber aus dem Programmlauf heraus bestimmt sein, Rückgabe werte müssen zur weiteren Verwendung gespeichert werden.

Trotzdem gibt es grundsätzlich einen Kommunikationsbedarf zwischen ISR und dem eigentlichen Programm; so kann beispielsweise die Bedienung einer Kommunikationsschnittstelle (RS232 bis LAN) per ISR erfolgen, nur müssen dann empfangene Zeichen übermittelt werden, ebenso müssen zu sendende Zechen gespeichert sein. Hier bietet sich ein Pufferspeicher zwischen beiden Instanzen an, in dem die Zeichen gespeichert sind und über besondere Mechanismen zwischen den Programmteilen kommuniziert.

Es sind also zwei Schritte, die man ggf. ausführen muss. Listing 9.8 zeigt zunächst denjenigen der Kommunikation zwischen ISR und Hauptprogramm. In diesem einfachen Beispiel wird dem Hauptprogramm innerhalb der `while(1)`-Schleife (Endlosschleife!) über die Variable `semaMess` angezeigt, ob ein neuer Messwert vorliegt (`semaMess = 1`) oder nicht (`semaMess = 0`). Damit lässt sich das grundsätzliche Konsumenten/Produzenten-Problem lösen, darin bestehend, dass ein Konsument nicht mehr verbrauchen kann als vorher produziert wurde. Derartige Variable werden häufig als *Semaphoren* bezeichnet.

Weiterhin gilt die Verabredung, dass der Produzent (die ISR) diese Variable nur auf 1 setzen darf, der Konsument nur auf 0. Diese Regel muss nun von der

Softwareentwicklung (und nicht dem Compiler) eingehalten werden. Ist nun der Wert von semaMess != 0, dann wird der zuletzt gespeicherte Messwert ausgelesen und weiter verarbeitet.

```
unsigned char semaMess = 0;
unsigned int globalMesswert;

...
main()
{
  unsigned int messwert;
  ...
  while(1)
  {
    if( semaMess != 0 )
    { /* Neuer Wert vorliegend? */
      /* Atomare Operation */
      #asm( cli );
      semaMess = 0;
      messwert = globalMesswert;
      #asm( sei );
      /* Ende der atomaren Operation */
      ...
    }
  }
}
```

a)

```
interrupt [TIMER] void timer_comp_isr(void)
{

  /* Die beiden Operationen sind wieder
     atomar */
  if( 0 == semaMess )
  {
    globalMesswert = ADC_OUT;
    semaMess = 1;
  }
```

b)

Listing 9.8 Nicht-blockierende Kommunikation zwischen Main- (a) und Interrupt-Routine (b)

Dabei tritt ein weiteres Problem auf, dass der atomaren Operation. Es muss z.B. verhindert werden, dass in dem Hauptprogramm die Semaphore `semaMess` auf 0 gesetzt wird und dann das Programm z.B. durch die ISR unterbrochen wird, noch bevor der Messwert in der Variable `globalMesswert` ausgelesen wurde. Tritt dies nämlich auf, gibt es Komplikationen, hier wird ein Wert überschrieben. Daher werden die beiden C-Anweisungen (semaMess = 0; messwert = globalMesswert) ununterbrechbar hintereinander ausgeführt, indem vorher das Interrupt-Enable-Flag gelöscht und hinterher wieder gesetzt wird. Im Klartext: Unterbrechungen sind während dieser Dauer ausgeschlossen (soweit dies möglich ist).

Bei der ISR ist dies nicht vorgesehen, weil bekannt ist, dass für diesen Mikrocontroller das Interrupt-Enable-Flag bei Eintritt in die ISR gelöscht wird und damit weitere Unterbrechungen unterdrückt werden. Dies muss aber in jedem Einzelfall geprüft werden, vor allem muss sich der Softwareentwickler genau überlegen, ob er/sie innerhalb einer ISR überhaupt weitere Unterbrechungen zulassen will oder darf.

Was passiert nun, wenn die Messdaten neu produziert werden, bevor der alte Wert überhaupt gelesen wurde? Ist dies systematisch der Fall, gilt das System als nicht echtzeitfähig und ist ggf. wertlos. Häufig ist aber auch der Fall möglich, dass eine solche Zeitüberschreitung nur gelegentlich vorkommt, so dass man also die Gewähr hat, dass es im Mittel funktioniert und nur für *k* Messzyklen hintereinander eine vorübergehende Verletzung haben kann. In diesem Fall hilft ein Mess- bzw. Kommunikationspuffer, der mehr als einen Wert zwischenspeichert und damit für deutliche Entspannung im Zeitgefüge sorgt (siehe auch 3.2.1).

Ein solcher Kommunikationspuffer wird vorteilhaft als Ringpuffer ausgeführt. Ein Ringpuffer wiederum kann als Array mit einer Verwaltung aufgefasst werden, wobei sich die Verwaltung auf einen Schreib- und einen Lesezeiger und deren Verhalten bezieht. Listing 9.9 zeigt eine Möglichkeit hierzu (siehe auch 4.2.1).

```
#define TX_BUFFER_SIZE 16

unsigned char ui8gTXBuffer[TX_BUFFER_SIZE];
unsigned char ui8gTXCounter;
unsigned char ui8gTXReadIndex;
unsigned char ui8gTXWriteIndex;


void vSetNextElement( unsigned char ui8lElement )
{
   ui8gTXBuffer[ui8gTXWriteIndex] = ui8lElement;

   ui8gTXWriteIndex++;
   if( ui8gTXWriteIndex > TX_BUFFER_SIZE - (uint8)1 )
      ui8gTXWriteIndex = 0;

   if( ui8gTXCounter < TX_BUFFER_SIZE )
   {
      ui8gTXCounter++;
   }
}


unsigned int ui16GetNextElement( void )
{
   unsigned int ui16lRetValue = 0xFFFF; /* means no element available */

   if( ui8gTXCounter > (uint8)0 )
   {
      ui16lRetValue = (unsigned int)ui8gTXBuffer[ui8gTXReadIndex];

      ui8gTXReadIndex++;
      if( ui8gTXReadIndex > TX_BUFFER_SIZE - (uint8)1 )
         ui8gTXReadIndex = 0;

      ui8gTXCounter--;
   }

   return( ui16lRetValue );
}
```

Listing 9.9 Kommunikationspuffer mit Verwaltung

Im Wesentlichen wird in diesem Codebeispiel die Verwaltung dahingehend geregelt, dass die Ringpufferstruktur entsteht (zyklisches Verhalten von `ui8gTXReadIndex` und `ui8gTXWriteIndex`) und dass beim Lesen die Rückgabe eines nicht-vorhandenen Werts durch ein spezielles Zeilen (0xFFFF) dargestellt ist. Ein Pufferüberlauf wird weder verhindert noch angezeigt, wobei dies relativ leicht einzubauen wäre.

9.3.2.4 Reentrant-fähige Funktionen

Es können selbstverständlich Funktionen aus einer ISR heraus aufgerufen werden, wobei hier Vorsicht geboten ist. Sollte sich der Mikroprozessor zufällig an einer Programmstelle befinden, die zu einer in der ISR aufgerufenen Funktion gehört, und tritt dann ein Interrupt Request auf, dann wird die besagte Funktion zweimal aufgerufen. Um jetzt richtige Ergebnisse zu liefern, muss eine Reentrantfähigkeit vorliegen. Dies soll kurz erläutert werden.

Listing 9.10 zeigt ein Beispiel für eine Funktion, die eine solche Bedingung ggf. nicht erfüllt. Hier wird eine *statische* Variable benutzt, die in diesem Fall den Aufrufparameter speichert. Die gespeicherte Variable wird abschließend zurückgegeben.

Das Problem, warum diese Funktion nicht ohne Probleme zweimal aufgerufen werden kann, kann schnell identifiziert werden. Angenommen, die Funktion wird aus dem Hauptprogramm heraus aufgerufen, die statische Variable speichert also den Aufrufparameter. Der Interrupt Request möge nun während des Durchlaufs durch diese Funktion erfolgen. Ruft die ISR diese Funktion dann ebenfalls auf, allerdings mit anderem Parameter, dann wird dieser gespeichert.

```
int no_reentrant_test( int number )
{
    static int savedNumber = -1;

    savedNumber = number;

…

    return( savedNumber )
}
```

Listing 9.10 Beispiel für eine nicht reentrant-fähig Funktion

Wenn nun die ISR beendet wird und der Prozessor mit der Bearbeitung des Hauptprogramms fortfährt, so springt er wieder in die genannte Routine `no_reentrant_test()`, dort wurde allerdings die Variable `savedNumber` verändert, so dass nun ein falscher Wert zurückgegeben wird.

Dieses Beispiel mag konstruiert sein, in der Praxis treten allerdings solche Fälle auf. Beispielsweise kann ein Netzwerkzugang so programmiert werden, dass der aktuelle Zustand (z.B. ”Versuche Zugriff“, ”Zugriff verweigert“, ”Zugriff erfolgt“ usw.) in einer statischen Variablen gespeichert wird. Damit ist bei jedem Aufruf die Vorgeschichte bekannt, und letztendlich die Aktion davon abhängig.

Soll nun die Ressource Netz von zwei verschiedenen Tasks verwendet werden, kann die einfache Implementierung so nicht verwendet werden, denn ein Neuaufruf von Task 2 muss verhindert werden, wenn Task 1 bereits aktiv ist.

Das eigentliche Problem der nicht reentrant-fähigen Funktionen liegt darin, dass von Bibliotheksfunktionen häufig nicht bekannt ist, ob sie die Fähigkeit besitzen oder nicht.

9.4 Interferenzen zwischen Hard- und Software

Ein weiterer Bereich der hardwarenahen Programmierung besteht darin, aufgrund von Kenntnissen der zugrunde liegenden Hardware die Software zu optimieren bzw. Fehler zu vermeiden. Hierzu sind im Folgenden zwei Teilbereiche dargestellt: Die Speicherung von intrinsischen Daten, die größer sind als 1 Byte (das so genannte Endian-Modell) und die Ausrichtung von Daten an den zugehörigen Adressen (das Alignment von Daten).

9.4.1 Die Endian-Modelle für Daten

Aus historischen Gründen, aber sicherlich auch, weil diese kleinste Speichereinheit für den Betrieb von Mikroprozessoren einen vernünftigen Kompromiss darstellt, sind die allermeisten Speichersysteme für Mikroprozessoren Byte-weise organisiert. Das bedeutet, dass der Zugriff auf den Speicher, ob lesend oder schreibend, mit einer Breite von 8 Bit (= 1 Byte) erfolgt.

Nun hat die eingebaute Verarbeitungsbreite von General-Purpose-Mikroprozessoren, also diejenige, mit der Daten in einer Instruktion geladen, gespeichert und verarbeitet werden können, mittlerweile bei Werten von 64 Bit erreicht. Der Speicher jedoch bleibt in der Regel Byte-orientiert, weil im Übrigen auch die Bearbeitung von einzelnen Bytes erhalten bleibt, so dass sich jetzt die Frage nach der Reihenfolge der Speicherung von Daten ergibt, falls diese mehr als 8 Bit breit sind.

Hier existieren zwei generelle Modelle, als *Big Endian* („Groß-Ender“) und *Little Endian* („Klein-Ender“) bezeichnet. Der Name bezieht sich dabei darauf, womit die Speicherung eines Wortes mit mehr als einem Byte beginnt: Wird an der Stelle mit der kleinsten Adresse das höchstwertige Byte gespeichert, wird dies als Big Endian bezeichnet, weil man mit dem höchstwertigen Teil beginnt (→ Bild 9.17).

Dementsprechend wird die Speicherung beginnend mit dem niederwertigen Byte mit Little Endian bezeichnet. Beide Speicherungen sind im Übrigen vollkommen gleichberechtigt, es gibt keine „natürliche Lösung“. Beispiele für Mikroprozessorarchitekturen, die das Big Endian Modell nutzen, sind die Architekturen von MIPS [mips] und ARM [arm], für die Nutzung des Little Endian Modells diejenigen von Intel [intel].

Ein Problem stellt sich ein, wenn der Datenzugriff von der Reihenfolge der Bytes abhängig wird. Dies kann beispielsweise unter folgenden Umständen der Fall sein:

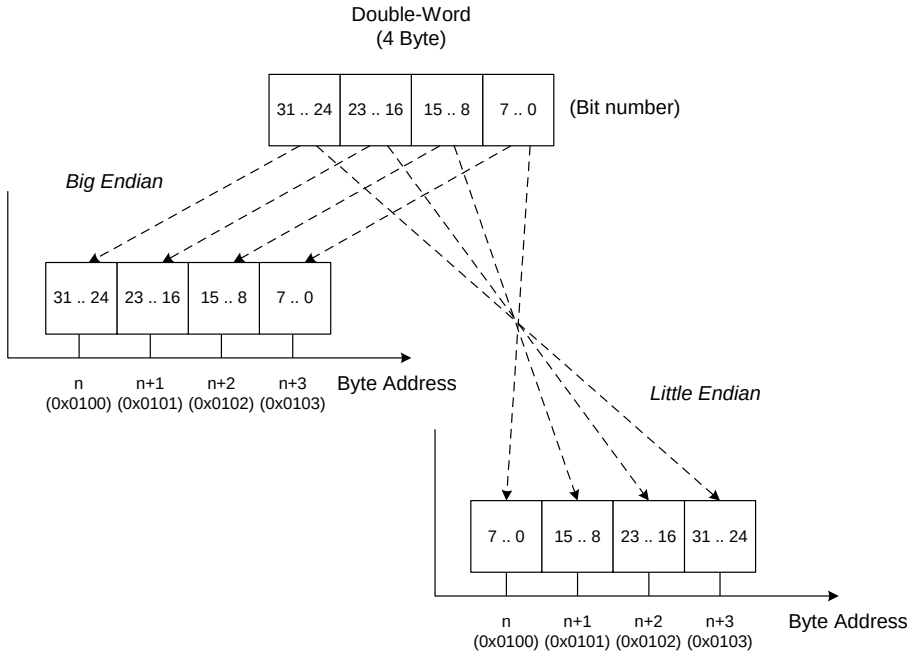


Bild 9.17 Vergleich Little Endian/Big Endian

- Daten werden wort- oder doppelwortweise geschrieben (weil sie etwa von einem Massenspeicher so gelesen werden), im Anschluss daran aber byteweise bearbeitet.
- Daten werden zwischen Mikroprozessoren mit verschiedenen Endian-Modellen ausgetauscht, etwa mithilfe eines gemeinsamen Speichers.
- Daten werden durch externe Hardware wie AD-Converter etc. erzeugt, wobei auch hier eine Inkonsistenz im Endian-Modell existieren kann

In all diesen Fällen muss durch den/die EntwicklerIn auf sehr hardwarenahe Weise dafür gesorgt werden, dass die Daten korrekt sortiert sind. Im Allgemeinen kann das dadurch erfolgen, dass ggf. auf die Daten byteweise zugegriffen wird und diese dann per Shift-Befehlen entsprechend zusammengefügt werden. Listing 9.11 zeigt die Sortierung beispielweise.

Im Übrigen sei darauf hingewiesen, dass bei übergreifenden Standards wie Internet Protokoll oder Ethernet genau darauf geachtet werden muss, welchem Endian-Modell die Daten im Paket unterliegen, falls das Paket direkt bearbeitet werden muss. Im Ethernet-Standard werden beispielsweise die Modelle miteinander vermischt.

```
unsigned int data_big_end;
unsigned int data_little_end;

data_little_end = ((data_big_end >> 24) & 0x000000FF)
                | ((data_big_end >> 8) & 0x0000FF00)
                | ((data_big_end << 8) & 0x00FF0000)
                | ((data_big_end << 24) & 0xFF000000);
```

Listing 9.11 Sortierung von 32-Bit-Werten zwischen Little und Big Endian Modell

9.4.2 Das Alignment von Daten

Eng mit den eben beschriebenen Endian-Modellen verwandt ist das Problem der Ausrichtung der Daten im Speicher. Daten mit Byteweite sind selbstverständlich an jeder beliebigen Byte-Adressen möglich, aber an welchen Stellen beginnen Daten mit größeren Breiten, also Word oder Doubleword?

Diese Ausrichtung auf bestimmte Adressen als Startwert wird als *Alignment* bezeichnet. Am flexibelsten ist natürlich, wenn die Startadresse beliebig sein darf, zugleich bedeutet die für die Hardware im Mikroprozessor jedoch, dass diese entweder sehr komplex wird oder so langsam, dass z.B. ein Takt mehr für den Zugriff auf den Speicher benötigt wird (oder auch beides).

Es ist daher sehr verbreitet, dass Mikroprozessoren einen Zugriff auf Datenobjekte, deren Breite > 8 Bit beträgt, nur „aligned“, also ausgerichtet durchführen können. Dies bedeutet in der Praxis, dass

- ein Zugriff auf ein Word (mit 16 Bit Datenbreite) nur an einer durch 2 teilbaren (Byte-)Adresse beginnen kann,
- ein Zugriff auf ein Doubleword (mit 32 Bit Datenbreite) nur an einer durch 4 teilbaren (Byte-)Adresse beginnen kann, und dass
- ein Zugriff mit einer davon abweichenden Adresse als „unaligned“ gilt und z.B. in einer Ausnahmebehandlung (Exception, → 9.3.1) resultieren kann.

Für die hardwarenahe Softwareentwicklung wiederum bedeutet es, dass ein besonderes Augenmerk auf diese Situationen gelegt werden muss. Wird beispielsweise ein Kommunikationspaket (erhalten z.B. via Ethernet) in einem Puffer abgelegt, so erfolgt das Einlesen vom Netzwerk und Speichern im Memory meist ohne Kenntnis der zugrunde liegenden Struktur der Daten, also z.B. Byte-weise. Wird jetzt an beliebiger Stelle ein Doubleword erwartet, kann dies sehr leicht unaligned sein – mit der Konsequenz, dass sich hier Fehler einschleichen können.

Listing 9.12 zeigt ein solches Beispiel. Gedacht ist hier, dass in dem Array buffer[] Daten eingelesen werden, die dann im Programm strukturiert und weiter verarbeitet werden können. Zu diesem Zweck ist angenommen, dass ab dem Index 3 ein Doubleword steht, und jetzt wird dem Zeiger auf unsigned long (angenommen,

long hat die Breite 32 Bit in diesem System) die Startadresse von diesem Element zugewiesen.

```
unsigned char buffer[256];
unsigned long *ulong_ptr, temp;

ulong_ptr = (unsigned long *)&(buffer [3]);
temp = *ulong_ptr;
```

Listing 9.12 Beispiel für Probleme mit Daten Alignment

Was genau soll der Compiler jetzt dem Zeiger auf unsigned long `ulong_ptr` zuweisen? Unter konservativer Annahme, dass nur ein ausgerichteter Zugriff möglich ist, wird die Adresse, die in ulong_ptr tatsächlich gespeichert wird, auf eine durch 4 teilbare Zahl (und zwar die nächstniedrigere, durch Maskierung) gesetzt, und das wird im Allgemeinen nicht dem entsprechen, was der/die EntwicklerIn hier wollte.

```
unsigned char buffer[256];
unsigned char *uchar_ptr;
unsigned long temp;

uchar_ptr = &(buffer [3]);
temp = ((*uchar_ptr) << 24) | ((*(uchar_ptr+1)) << 16)
     | ((*(uchar_ptr+2)) << 8) (*(uchar_ptr+3));
```

Listing 9.13 Doubleword-Zugriff bei möglicherweise unaligned Daten

Listing 9.13 zeigt eine Lösung für dieses Problem: Hier wird byteweise zugegriffen und dann (im Big-Endian-Format) sortiert. Diese Zeilen sind natürlich sehr abhängig von der Hardware und müssen für andere Systeme ggf. ausgetauscht werden.

10 Hardware/Software Co-Design

In Abschnitt 4.4 war bereits ein erster Hinweis auf ein besonderes Kapitel des Designs digitaler Systeme gegeben worden, dem Hardware/Software Co-Design (kurz: Co-Design). Designmethodiken aus diesem Bereich fristen ein gewisses Schattendasein: Für Mainstream sind sie zu komplex, und die Anzahl von zwingenden Fällen, bei denen es ohne diese Methodik nicht geht, ist auch nicht besonders hoch – aber eben steigend. Aus diesem Grund ist dem Bereich des Co-Design hier ein Kapitel gewidmet.

Hardware/Software Co-Design (die Nennung erfolgte erstmalig 1991) ist ein neuer Aspekt des *Designs* einer bestimmten Klasse von elektronischen Systemen, die in binär-digitaler Weise funktionieren. Diese Systeme basieren – wie bereits dargestellt – zum überwiegenden Teil auf einem Mikroprozessor bzw. -controller, der mit Hilfe einer Software die erwünschte Funktionalität zur Verfügung stellt.

Da die Entwicklung eines solchen Systems bislang fast ausschließlich auf dem klassischen Ansatz *'Hardware First'* basiert, sind Einschränkungen in der Funktionalität – etwa im Zeitverhalten des Systems – kaum verwunderlich. Diese Einschränkungen sind zwar prinzipiell in jedem System vorhanden, da sie nicht vermieden werden können; der 'Hardware First'-Ansatz liefert aber eine quasi unveränderliche Basis, mit der der Systemdesigner (der eigentlich durch den Software-Designer dargestellt wird) leben musste. Die ggf. negativen Auswirkungen der Einschränkungen müssen dann durch im eigentlichen Designprozess nicht vorgesehene Maßnahmen wie 'Tuning' ausgeglichen werden.

Im Gegensatz hierzu versucht der Ansatz *Hardware/Software Co-Design* eine ganzheitliche, parallel zueinander verlaufende Designmethode einzuführen. Dies ist zunächst ein allgemeiner Ansatz, eine Methode ohne Tools, ein Denkansatz: "*Meeting all Constraints on System Level*".

Nach vielerlei Entwicklungen hat es sich als sehr praktisch erwiesen, spezielle Systeme als Zielsystem auszuwählen und hierauf ein Hardware/Software Co-Design auszuführen. Diese Systeme enthalten Software-programmierbare wie Hardware-strukturierbare Elemente, beide also programmierbar, und sind damit prädestiniert für Co-Design.

In diesem Kapitel soll versucht werden, einen Bogen über alle Themen zu spannen. Zunächst soll Co-Design motiviert werden: Welche Möglichkeiten sind darin zu sehen, von der herkömmlichen Methode der alleinigen Abbildung auf einen Mikroprozessor abzuweichen?

Im zweiten Teil geht es um die möglichen programmierbaren Hardwarearchitekturen: Mikroprozessor & Co, Programmable Logic Devices, konfigurierbare ALU-Arrays etc. werden betrachtet.

Der dritte Teil behandelt die Möglichkeiten zur Nutzung von konfigurierbaren Bauteilen für ein Hardware/Software Co-Design. Hier werden Software und die Algorithmen zum Co-Design betrachtet, und zwar zweifach: Zum Einen müssen Methoden gefunden werden, um Design zumindest mit automatischer Unterstützung auf mehrere Plattformen zu verteilen, zum Anderen ist es interessant, einmal zu betrachten, ob es nicht gemeinsame Hochsprachen gibt oder geben kann, die Hard- und Software gleichermaßen definieren.

10.1 Motivation zum Co-Design

Benötigt man eigentlich eine Motivation zur Beschäftigung mit einem neuen Rechner- oder Designparadigma? Die Antwort ist zweiteilig: In der Forschung ist das Neue Motivation genug, in der Anwendung jedoch muss es schon gute Gründe hierfür geben. Glücklicherweise gibt es diese guten Gründe auch.

Hierzu lohnt sich ein Blick auf das Jahr 2010ff und die Vorhersagen für Mainstream-Applikationen [Man03], [Mül03].

10.1.1 Organic Computing

Die Bezeichnung *Organic Computing* wird leider für zwei verschiedene Architekturen bzw. Technologien verwendet: Hardware-Technologen verwenden dies, um die Nutzung von organischen Molekülen für Rechnungen, Speicherung von Daten etc. zu beschreiben, in der Informatik hat sich dies als Beschreibung für eine zukünftige Architektur mit folgenden Eigenschaften etabliert:

- Selbst-konfigurierend
- Selbst-heilend
- Selbst-optimierend
- Selbst-schützend

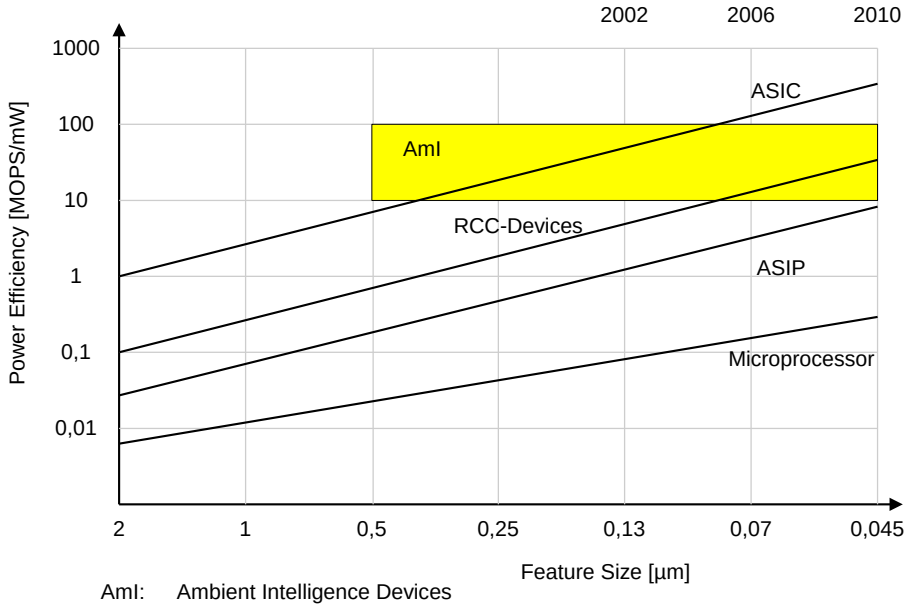
Von diesen Eigenschaften ist die der Selbst-Konfiguration die zentrale, alle anderen basieren darauf. Das *Organic Computing* zielt natürlich insbesondere auf vernetzte Systeme, wobei die einzelnen Netzknoten allerdings einen Großteil der Eigenschaften tragen. Mit anderen Worten: Organic Computing basiert auf rekonfigurierender Hardware.

10.1.2 Ambient Intelligence Devices

Unabhängig von dem eben skizzierten Ansatz zu zukünftigen verteilten Systemen und ihren Architekturen existieren Voraussagen zu dem zukünftigen Mainstream. Hier werden die Nachfolger der jetzigen Personal Digital Assistents (PDAs) prognostiziert, meist mit Ambient Intelligent Devices (AmI) bezeichnet.

Für die AmI-Devices, die sich im Übrigen nahtlos in die Organic-Computing-Welt einfügen, existieren einige Detailschätzungen:

- Ein Batteriebetrieb auf Akkumulatorbasis wird allein aus Kosten- und Mobilitätsgründen angenommen, wobei allerdings die Entwicklung der Kapazitäten sehr konservativ angenommen werden (< 10%/Jahr).
- Typische Applikationen sind: Kommunikation jeglicher Art, hierbei Verschlüsselung, Video-on-demand bzw. live, Spiele, etc.
- Als maximale Rechenleistung wird ca. 100 Giga-Operationen pro Sekunde (100 GOPS) prognostiziert. Spitzenreiter sind dabei Spiele und Video.
- Aus der Entwicklung der Batteriekapazitäten und den Anforderungen für die Rechenleistung ergibt sich die wesentliche Forderung: 2 W Leistungsaufnahme, d.h. eine **Energieeffizienz von 10 – 100 MOPS/mW** [Man03].



AmI: Ambient Intelligence Devices
ASIC: Application-Specific Devices
ASIP: Application-Specific Instruction Set Processor
RCC: Reconfigurable Computing

Bild 10.1 Powereffizienzbereiche [Man03]

Diese Zahlen sind recht erschreckend, denn programmierbare Systeme liefern andere Werte. Bild 10.1 zeigt die bisherige und prognostizierte Entwicklung für (General-Purpose-) Mikroprozessoren, applikationsspezifische Prozessoren (ASIP), Reconfigurable Computing Devices und ASICs – letztere nur zum Vergleich, denn diese sind nicht programmierbar, sondern fixiert. Tabelle 10.1 gibt einen weiteren Aufschluss zu dem Thema Energieeffizienz [MNW03], [www-hh02].

Die darin aufgeführten Mikroprozessoren und –controller sind willkürlich ausgewählt, um die Spannweite der momentanen Effizienzwerte zu demonstrieren. Pentium Pro und Celeron sind bekannt. SH7709 stellt einen 32-Bit-Mikrocontroller von Hitachi (jetzt: Renesas) dar, DEC Alpha 21364 dürfte auch noch bekannt sein.

Der Lutonium Mikroprozessor [MNW03] ist eine Entwicklung vom Caltech, USA. Es handelt sich hierbei um den 8051-Mikroprozessor in einer asynchronen Variante, also ohne Takt. Die beiden Werte sind durch verschiedene Betriebsspannungen entstanden, einmal für 1,8 V. der besonders effiziente Wert für 0,5 V, eine Subthreshold-Spannung, bei der der Mikroprozessor eigentlich nicht mehr arbeiten dürfte.

Mikro-Prozessor	Prozess-technologie	Frequenz	Verlust-leistung	MIPS/mW	Bemerkung
Pentium Pro	0,6 µm	150 MHz	23 W	0,0065	1995
Celeron	0,25 µm	400 MHz	23,4 W	0,017	1999
Athlon (K7)	0,25 µm	700 MHz	45 W	0,019	1999
DEC21364	0,18 µm	1 GHz	100 W	0,047	2000
SH7708	0,5 µm	60 MHz	0,6 W	0,1	1994
Lutonium	0,18 µm	200 MHz	0,1 W	1,8	Asynchron, 1,8 V, 2004
Lutonium	0,18 µm	4 MHz	0,17 mW	23	Asynchron, 0,5 V, 2004
XPP (RCC)	0,18 µm	200 MHz	10 W	20	Peak-Schätzwerte, 2003

Tabelle 10.1 Leistungsdaten einiger Mikroprozessoren [www-hh03] [MNW03] [pact]

Die Tabelle für Mikroprozessoren ist ergänzt durch ein FPFA-Device (→ 10.2) der Firma PACT, XPP (eXtreme Processing Platform). Die hier angegebenen Werte sind Schätzungen für die Peak-Performance (200 GOPS), so dass nur bedingt verglichen werden kann. Man erkennt trotzdem aus der Tabelle, dass wir mit Mikroprozessoren weit von dem 10 MIPS/mW-Punkt entfernt sind. Genau hier setzen die Hoffnungen zum Reconfigurable Computing an: Es soll damit eine bislang unerreichbare Effizienz erreicht werden.

10.1.3 Design Space Exploration

Diese Fakten sind zunächst isoliert stehend, können aber als Anhaltspunkte für größere Zusammenhänge von Verlustleistung und Rechnermodell gesehen werden. In einer Studie wurde an der RWTH Aachen untersucht, wie sich die Aufwände für unterschiedliche Implementierungen einer Aufgabenstellung unterscheiden [BFS+03], [BHF+02]. Als Grundlage wurden dabei diverse Algorithmen aus der Kommunikation (Filterung, FFT etc.) gewählt, die im Wesentlichen Datenstromei-

genschaften haben: Ein vergleichsweise kleines Programm bearbeitet einen Strom von Daten.

Abgesehen von dieser Einschränkung können die Ergebnisse zumindest qualitativ auf alle Bereiche übertragen werden. Bild 10.2 stellt dabei dar, wie sich das Produkt aus Siliziumfläche, Rechenzeit und Energieaufwand für den gleichen Algorithmus über die verschiedenen Architekturen ändert. Der Kehrwert dieser so genannten Kosten wird im Übrigen Effizienz genannt.

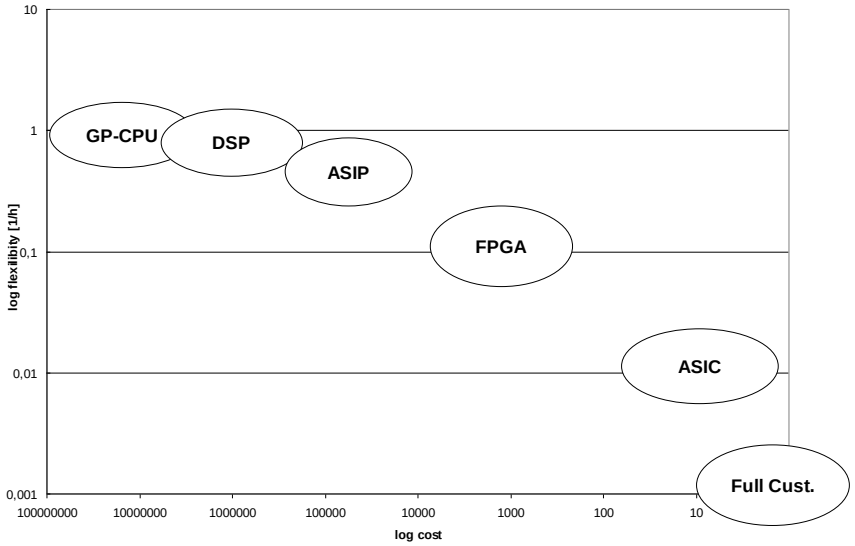


Bild 10.2 Flexibilität versus Kosten für verschiedene programmierbare Architekturen

Während die Werte zu den Kosten aus zumindest genauen Schätzungen stammen, wurde für die Flexibilität eher ungenau geschätzt. Die Anzahl der Stunden, die durch entsprechend geschultes und trainiertes Fachpersonal für eine Änderung benötigt wird, ist hier als Kehrwert eingetragen. Die Unterschiede in den programmierbaren Architekturen resultieren dabei aus den Tatsachen, welche Sprachen benutzt werden können (Assembler, C, VHDL) und ob die Programmierung sehr spezifisch (ASIP, DSP) oder eher generell (portierbar) gestaltet werden kann.

Das Ergebnis spricht für sich: Offenbar sind 8 Zehnerpotenzen Kosten und 3 Zehnerpotenzen Flexibilität in dieser Darstellung vorhanden. Für die Kosten gilt, dass die relativ gleichmäßig auf Si-Fläche A, Laufzeit T und Verlustleistung P entfallen, so dass für jede dieser Größen knapp 3 Größenordnungen Variabilität über den Wechsel der Hardwareplattform bleiben.

Die Aufgabe des Co-Design ist es nun, zwischen der einfachen Programmierbarkeit (Flexibilität) und den Kosten einen jeweiligen Kompromiss zu finden.

Anmerkung: Ein quantitativer Zusammenhang zwischen A, T und P wurde bereits in 5.1 hergestellt. Dort war die Randbedingung, dass man sich auf einer Technologiestufe bewegt und die Form der Programmierbarkeit sowie die Ausführungsdimension nicht ändert, sondern in dem einen Design die Implementierung variiert. A, T und P stehen unter diesen Umständen in Konkurrenz zueinander, die Optimierung betreffend.

Beim Wechsel der Ausführungsdimension (→ 10.2) hingegen kommt es zu einer gleichförmigen Optimierung der drei Designparameter A, T und P. Hier ist – wie dargestellt – die Konkurrenzbeziehung zwischen Flexibilität und dem Produkt A * T * P.

10.2 Operationsprinzipien und Klassifizierungen verschiedener Hardwarearchitekturen

10.2.1 Strukturmodelle

Man kann sich allgemein die Frage stellen, wie digitale, insbesondere programmierbare Systeme eigentlich arbeiten. Diese Systeme sind entweder komplett zur Designzeit fertiggestellt oder im Kern halbfertig. Im letzteren, programmierbaren Fall benötigen sie zur tatsächlichen Funktion zusätzliche Informationen, die beim Ablauf interpretiert werden. Programmierbare Systeme interpretieren also zur Laufzeit das Programm (unabhängig davon, welcher Art diese Informationen sind).

Um insbesondere die programmierbaren Architekturen, auf denen hier das Hauptaugenmerk liegt, diesbezüglich darzustellen, verwendet man Modelle, und zwar meist zwei Modelle: Die Struktur der Architektur einschließlich des Speichers für die Programmierinformationen und die Daten sowie das Ablaufmodell.

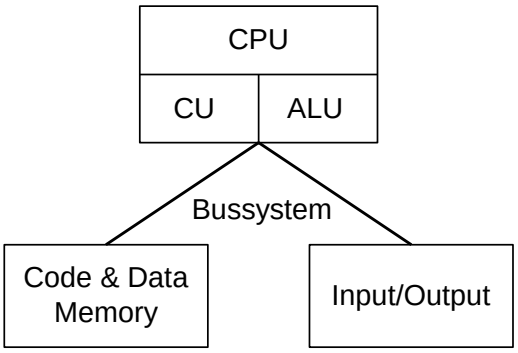


Bild 10.3 Von-Neumann-Strukturmodell

Das Von-Neumann-Modell (Bild 10.3) enthält die Dreiteilung aus Speicher (für Code und Daten), der Central Processing Unit (CPU) mit den Teilen Control Unit CU und Arithmetical-Logical Unit ALU und dem Input/Output-Bereich, oft auch als Peripherie bezeichnet. Zusammen mit dem verbindenden Bussystem ergibt dies ein minimales Rechnersystem.

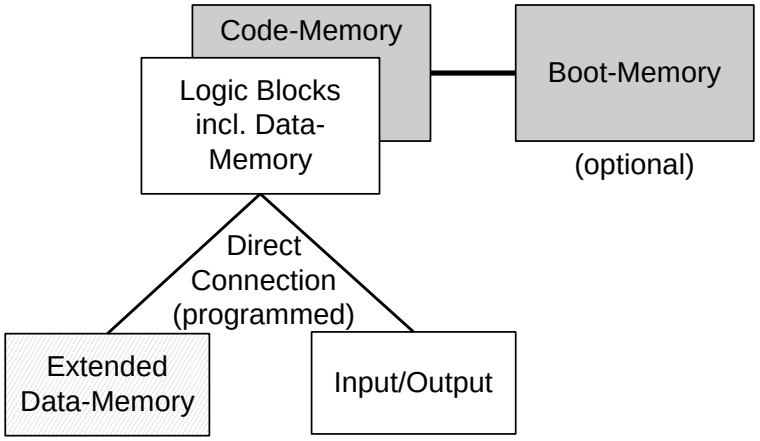


Bild 10.4 Strukturmodell für Programmierbare Logikbausteine

Bild 10.4 zeigt die Struktur für Programmierbare Logikbausteine (PLD, Programmable Logic Devices), die andere am Markt vertretene programmierbare Architektur. Die Bestandteile sind ähnlich, allerdings anders verteilt: Die ALU ist nicht mehr konzentriert, die Berechnungen finden in den (verteilten) Logikblöcken statt, die zugleich die Daten speichern.

Der Codespeicher, der die Arbeitsinformationen für die Logikblöcke aufnimmt, ist dementsprechend auch verteilt und wird durch einen optionalen Bootspeicher unterstützt. Der erweiterte Datenspeicher hingegen ist meist konzentriert und entspricht auch in seinem Zugriff dem (Daten-)Speicher im Von-Neumann-Modell.

10.2.2 Ablaufmodelle

Die Strukturmodelle unterscheiden sich zwar, aber hier lassen sich noch keine Unterschiede im Betrieb ausmachen. Worin aber bestehen denn eigentlich die Unterschiede? Ein Mikroprozessor nimmt eine Programminformation (genannt Instruktion oder Maschinenbefehl) auf, interpretiert sie und arbeitet nach den dort codierten Angaben. Ein programmierbarer Logikbaustein (PLD) nimmt ebenfalls eine Programminformation (genannt Konfiguration) auf, interpretiert sie und arbeitet gemäß der Codierung.

Der Unterschied liegt in dem zeitlichen Ablauf, besser in dem «danach». Tabelle 10.2, ursprünglich in [DW99] publiziert und weiterentwickelt, gibt Auskunft darüber, wie sich die Zeiten bei den beiden programmierbaren Architekturen verhalten.

Diese Tabelle vergleicht zunächst drei Formen aus dem Bereich des Configurable Computing, ein PLD, das einmal programmierbar ist bzw. auf einer Flash-EEPROM-Technologie basiert, ein SRAM-basiertes PLD und ein (virtuelles) Device aus dem Bereich des Reconfigurable Computing mit einem Mikroprozessor auf die charakteristischen Zeiten.

	PLD, one-time progr./Flash	**PLD, SRAM**	**Reconf. Computing**	**µP**
Bindungszeit	Fuseprogram-mierung	Laden Konfig.	Laden Konfig.	Takt
Bindungsdauer	unendlich/ Reprogrammierung	Reset	user-definiert	Takt
Programmierzeit	Sekunden	µs-ms (s)	Takt	Takt
Typische Anzahl Instruktionen	1	1	> 1	>> 1

Tabelle 10.2 Charakteristische Zeiten für digitale Systeme

Für den Mikroprozessor verlaufen alle wesentlichen Vorgänge im Rahmen von (wenigen) Takten: Die aktuelle Arbeitsweise wird in der Fetchphase festgelegt, die Einstellung auf die Arbeitsweise (Decode/Load) dauert ebenfalls ca. einen Takt, und abschließend wird ausgeführt (Execute). Auch die letzte Phase nimmt wenige Takte (im Idealfall einen) in Anspruch. Mit anderen Worten: Der Mikroprozessor arbeitet etwa in einem Taktschema, die Idealbild einer RISC-Architektur). Was aber wesentlich wichtiger ist: Der Mikroprozessor arbeitet nach einer Instruktion die nächste ab: Der Instruktionswechsel ist Maschinen-definiert!

Beim den PLDs in Tabelle 10.2 fällt zunächst einmal auf, dass die Speichertechnologie viel Einfluss auf die Anwendungsweise haben kann: One-Time Programmable und Flash-EEPROM-basierte Bausteine werden programmiert und arbeiten dann, ohne einen weiteren Support bekommen zu müssen. Diese Fuseprogrammierung ist ein Vorgang, der außerhalb des eigentlichen Betriebs durchlaufen wird. Aus diesem Grund spielt auch die Programmierzeit im Bereich von Sekunden nur eine untergeordnete Rolle.

Bei SRAM-basierten PLDs muss aufgrund der Flüchtigkeit der Information bei Ausfall der Betriebsspannung die Konfiguration bei jedem Anschalten geladen werden – Vor- und Nachteil zugleich. Der Vorteil ist derjenige, dass man

zumindest bei jedem Anschalten auswählen kann, welche Funktion ausgeführt werden soll, der Nachteil besteht darin, dass man es auch muss (Boot-Vorgang).

Die Programmierzeit liegt bei Forschungsarchitekturen im Rahmen von Mikrosekunden, bei produzierten Architekturen eher im Rahmen von Sekunden. Dies ist im Wesentlichen eine Frage des Interface und dessen Parallelität. In rekonfigurierbaren Architekturen wird man Wert darauf legen, dass die Rekonfiguration, ggf. partiell, nur wenige Betriebstakte in Anspruch nehmen wird.

Ein wichtiger Aspekt ist auch, dass die Anzahl der verschiedenen Instruktionen (verschieden in Typ, Operanden oder der Stelle im Programm), die ein Algorithmus benötigt und die in einer sequenziellen Form bearbeitet werden müssen, gleich 1 ist für die PLDs und >>1 für Mikroprozessoren. Mit anderen Worten: In PLDs wird *eine* Instruktion spezifisch für den Algorithmus konfiguriert, und zwar in der Fläche, bei Mikroprozessoren werden unterschiedliche Instruktionen in einer zeitlichen Sequenz benutzt. Die wird auch als *Ausführungsdimension* bezeichnet: *Computing in Time* (Mikroprozessor) und *Computing in Space* (PLD).

Bei rekonfigurierbaren Architekturen wird eine Anzahl von >1 (typischerweise 8–16) benutzt, um den Algorithmus darzustellen. Für diese Architekturen wird sowohl die Programmierung in der Fläche als auch die in der Zeit genutzt. Bei den Zeiten drückt sich dies so aus, dass die Bindungsdauer hier nicht Maschinen- sondern User- oder Applikations-definiert ist.

10.2.3 Entwicklung der Configurable Computing Devices (CC-Devices)

Wie kam es eigentlich zu der Entwicklung dieser zweiten Klasse von programmierbaren Bausteinen, den PLDs? In den 30er Jahren wurden erste Ideen zu Rechenmaschinen – oder wie man etwas später auch sagte: *Elektronengehirnen* – entwickelt. Außerhalb der zunächst wirklich eng umgrenzten Fachkreise war daran wohl eines unbegreiflich: Wie kann es eine Maschine geben, die nicht für den endgültigen Zweck produziert wird, sondern nur «halbfertig» ist? Seitdem hat man sich natürlich an Software gewöhnt, sie gibt der Rechenmaschine die für den Moment gültige Funktion und kann jederzeit gewechselt werden. In der Frühzeit der Rechner war diese Software für den Laien unbegreiflich.

Hard- und Software haben in der Welt ihren wohldefinierten Platz gefunden. Die fixierte Hardware, die zunächst (zum großen Teil in der Fabrik) entworfen wird (Hardware First), bleibt unverändert und bietet ein Interface für die Software. Software wird vom Entwickler entworfen und ist flexibel.

Ende der 60er Jahre kamen die ersten Ideen auf, auch die Hardware flexibler zu gestalten. So überraschend es klingen mag, auch hier gibt es wieder einen fabrikatorischen, fixierten Teil (der «echten» Hardware) und ein Stück Beschreibung der endgültigen Funktionalität (die Software, hier meist Konfiguration genannt). Man muss sich schon fragen, wo der Unterschied denn eigentlich noch zu suchen ist.

1977, also nur 6 Jahre nach der Markteinführung des ersten Mikroprozessors Intel 4004, erschienen die ersten kommerziellen Bausteine auf dem Markt, Hersteller war Monolithic Memories Inc. (MMI). Die Entwickler dieser Bausteine, PAL (Programmable Array Logic) genannt, haben mit ihrer Wahl den inneren Aufbau einer Vielzahl von heute verfügbaren ICs beeinflusst. 1985 wurden dann die ersten feldprogrammierbaren ICs, die in der Lage waren, ein «System» aufzunehmen, von Xilinx angeboten.

Die PLDs erleben zurzeit einen wahren Boom, Geschwindigkeit und nutzbare Größe betreffend. Für die Entwicklung von digitalen Systemen bieten sie gegenüber ASICs den Vorteil, programmierbar zu sein, während sie sich von den ebenfalls programmierbaren Mikroprozessoren dadurch unterscheiden, dass ihr Ausführungsprinzip nicht mehr sequenziell ist, sondern dass das Programm in der Struktur liegt (→ 10.2.2). Programmierte PLDs funktionieren wie hergestellte Hardware.

Andererseits stehen wir derzeit an der Schwelle zur Nutzung der dynamischen Reprogrammierbarkeit. Bestimmte Speichertechnologien ermöglichen einen Wechsel des Programms, auch teilweise, zur Laufzeit. Dies bedeutet, dass zukünftig mehr Programm gespeichert wird als aktuell ausführbar ist – ein Zustand, der bei prozessorbasierten Rechner gängige Praxis ist, bei PLDs jedoch neu sein wird.

Als Konsequenz hat sich eine neue Bausteinklasse entwickelt, mit FPFA (Field-Programmable Functional Array) bezeichnet. Diese Bausteinklasse erhält derzeit in der Forschung höchstes Interesse, da hier das zukünftig größte Potenzial vermutet wird. Erste Bausteine sind seit Ende 2000 auf dem Markt, aber eine Marktdurchdringung ist bislang ausgeblieben.

10.2.4 Klassifizierung der CC-Devices

10.2.4.1 Struktur von CC-Devices

Die verschiedenen Bausteinarten, die zum Bereich der Configurable Computing Devices gehören (also die Programmierung in der Struktur aufnehmen), weisen alle einen grundsätzlichen Aufbau gemäß Bild 10.5 auf. Ein äußerer Anschlussring ist zuständig für die Konnektivität der Bausteine. Die meisten dieser Anschlüsse sind einfacher digitaler Art ohne zusätzliche Funktionen (wie etwa Parallel-Seriell-Wandler o.ä., da diese Funktionen unmittelbar in der inneren Hardware abgebildet werden können.

Der innere Teil besteht aus einer (homogenen oder inhomogenen) Matrix von Rechenblöcken mit konfigurierbaren Verbindungen. Die Rechenblöcke – auch Speicherblöcke sind hier möglich – können z.B. eine ALU mit 2*32 bit Eingangsdaten und konfigurierbarer Operation (Addition, Multiplikation, Logik) darstellen, oder es sind Logikblöcke mit 4-64 binärwertigen Eingängen und frei konfigurierbarer Verknüpfung.

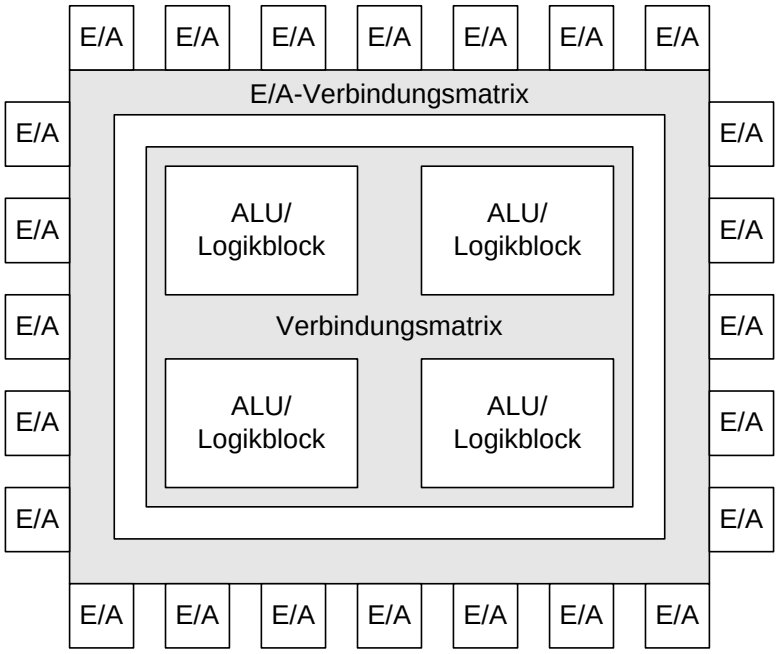


Bild 10.5 Struktur von CC-Devices

Die Verbindungen zwischen dem E/A-Bereich und den Rechenblöcken oder zwischen verschiedenen Rechenblöcken sind ebenfalls konfigurierbar, d.h., es kann aus einer Anzahl von Möglichkeiten ausgewählt werden. An dieser Stelle sei bemerkt, dass in der Regel mehr in den Verbindungen als in den eigentlichen Verknüpfungen konfiguriert werden muss.

Die CC-Devices können nach verschiedenen Gesichtspunkten klassifiziert werden: Granularität (kleinste unteilbare Recheneinheit), Verbindungsmatrix und Speichertechnologie/Programmierbarkeit sind hierbei die wichtigsten Klassifizierungsmerkmale und sollen daher an dieser Stelle behandelt werden.

10.2.4.2 Speichertechnologien bei CC-Devices

Neben der bislang behandelten Architektur der CC-Devices ist es auch wichtig, die Wirkungsweise der Programmierung zu verstehen und die einsetzbaren Speichertechnologien zu betrachten, da sich hieraus einige Konsequenzen für den Betrieb ergeben. In diesem Zusammenhang sei nochmals erwähnt, dass sich die Speichertechnologien ausschließlich auf Konfigurationsspeicher beziehen.

Zentrales Element bei der Programmierung ist der Pass-Transistor (Bild 10.6). Dies ist meist ein selbstsperrender N-Kanal MOSFET-Transistor, der mit Hilfe einer geeigneten Ansteuerung die Zustände sperrend und leitend annehmen kann. In Bild

10.6 a) wird ein potenzielles Eingangssignal *x* an ein AND-Gatter über einen Pass-Transistor geführt. Wird dieser derart angesteuert, dass er sperrt, ist das Eingangssignal nicht sichtbar, und das AND-Gatter erhält via Pull-Up-Widerstand ein High-Signal am Eingang. Für ein OR-Gatter wäre hier entsprechend ein Pull-Down-Widerstand notwendig.

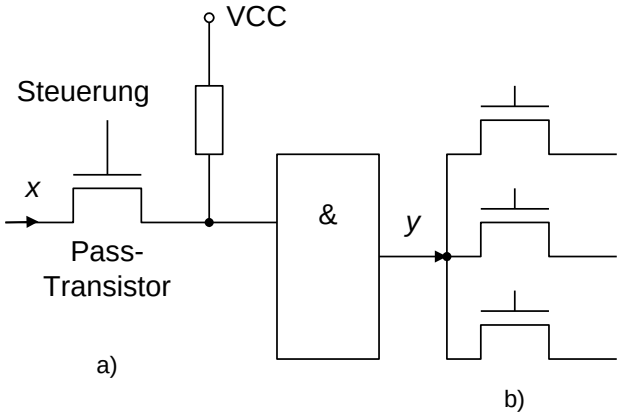


Bild 10.6 Nutzung von Pass-Transistoren in PLDs
a) Beschaltung von Gattereingängen b) Nutzung im Routing der Bausteine

Der Ausgang des AND-Gatters *y* wird mit Hilfe von 3 Pass-Transistoren, die prinzipiell alle leitend sein können, im Baustein geroutet, d.h. elektrisch weitergeführt. Die Programmierung des Bausteins entspricht damit exakt der Steuerung dieser Pass-Transistoren, und diese Steuerinformationen müssen im Baustein gespeichert werden. Prinzipiell dürften bezüglich der Speichertechnologie damit keine großen Differenzen zu Mikrocontrollern und deren Speicher existieren.

Die 3 aktuellen Speichertechnologien, die für Konfigurationsspeicher zum Einsatz kommen, sind Anti-Fuse, (E)EPROM und SRAM. Anti-Fuse, aus Sicht der Mikroprozessoren und -controller der Exot, entstammt der alten Fuse-Technologie. Innerhalb der bipolaren Bausteine waren an den Kreuzungspunkten (den potenziellen Verbindungen) kleine Metallverbindung, die so genannten Fusable Links, angebracht. Programmierung bedeutete Zerstörung dieser «Schmelzsicherungen», wodurch der Kontakt dann unterbrochen wurde.

Der wesentliche Nachteil dieser Fuses bestand in hohen Übergangswiderständen oder großen Strömen bei der Programmierung. Für den Einsatz in PLDs wurde der Programmierprozess umgedreht, Verbindungen können nunmehr einmalig hergestellt (und nicht mehr zerstört) werden. Man nennt dies *Antifuse*, und der Grund zum Wechsel der Programmierrichtung ist leicht in den gegenüber den bipolaren

Fuses verbesserten elektrischen Eigenschaften (Widerstand) zu finden. Bild 10.7 zeigt das Prinzip dieser Zelle.

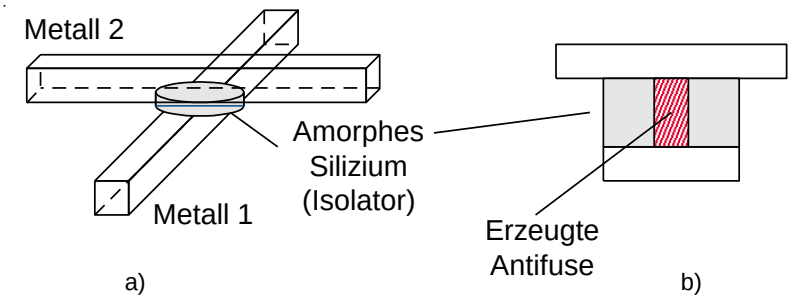


Bild 10.7 Antifuse-Technologie a) Kreuzung zweier Metallleiter im IC
b) Kreuzungspunkt nach Programmierung

Der metallische Leiter zwischen den eigentlichen Metallleitern (in Bild 10.7 b) schraffiert) wird durch kontrollierte Überspannung und einen Überschlag im Baustein erzeugt. Hierdurch kann man festhalten, dass diese Programmierung irreversibel ist, der Kreuzungspunkt jedoch sehr wenig Fläche bei geringem Widerstand und Kapazität beansprucht – Eigenschaften, die diese Technologie für den Betrieb ideal, für die Programmierung und Entwicklung eher unhandlich erscheinen lassen.

Der Hauptvorteil der Antifuse-Technologie besteht in der hohen Geschwindigkeit der programmierten Bausteine, als wesentlicher Nachteil wird meist der nichtreversible Prozess angesehen. Dieser Nachteil wird übrigens manchmal auch als Vorteil gesehen, denn was nicht mehr änderbar ist, kann auch nicht verfälscht werden.

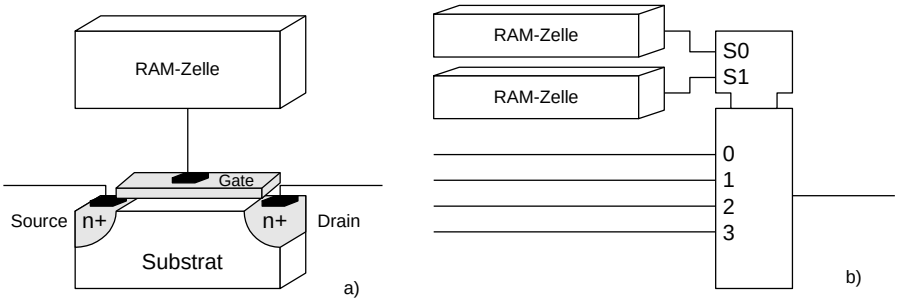


Bild 10.8 Nutzung von SRAM-Zellen für PLDs
a) mit Pass-Transistor b) 2 RAM-Zellen für 1-aus-4-Multiplexer (im Routing)

Das andere Extrem der Programmiertechnologie besteht in statischen RAM-Zellen, die die Strukturinformation tragen. Statisches RAM (SRAM) entspricht der Verschaltung von 5–6 Transistoren zu einem (D-)Flipflop. Hierin kann die Programmierinformation gespeichert werden, die dann mittels eines Pass-Transistors als Verbindung/Trennung (Bild 10.8 a)) oder eines Multiplexers (Bild 10.8 b)) auswertbar ist. Die SRAM-Technologie ist daher beliebig wiederverwendbar, selbst der Löschvorgang wird sehr einfach: Ein Abschalten der Betriebsspannung reicht, der Baustein ist danach unprogrammiert.

Der Programmiervorgang besteht aus einem Ladevorgang, häufig auch als Boot-Vorgang bezeichnet, und ist vergleichbar mit dem eines Programms zur Ausführung in einem PC. Dies ist natürlich jederzeit reversibel, was zugleich Vor- und Nachteil ist: Mit jedem Spannungsverlust ist die Programmierung des CC-Devices schließlich verloren. Um ein Stand-Alone-System zu erhalten, muss ein zusätzlicher Speicherbaustein (meist serielles EEPROM) hinzugefügt werden, der bei Reset oder Einschalten das Programm in den Baustein lädt.

Als Hauptvorteil von SRAM gilt die Flexibilität, die Hauptnachteile bestehen in der relativ niedrigen Geschwindigkeit (hohe Kapazitäten der Pass-Transistoren), in der Anzahl der Transistoren pro Speicherung des Werts sowie im Aufwand beim Einschalten der Spannung.

Die dritte Technologie liegt in gewissem Sinn in der Mitte: EPROM-Transistoren werden programmiert und sind hierdurch entweder durchschaltbar (1) oder nicht (0). Wie bei SRAM ist auch diese Technologie aus der Rechnertechnik bekannt, sie wurde mit Erfolg in die PLD-Welt transferiert.

Die EPROM/Flash-EEPROM/EEPROM-Technologie basiert auf speziellen MOS-Transistortypen mit Floating Gate (→ Bild 10.9). Bei ungeladenem Floating Gate kann ein positives Selektierungssignal den sonst selbstsperrenden MOS-Transistor leitend schalten; dies wird verhindert, wenn das Floating Gate Elektronen trägt.

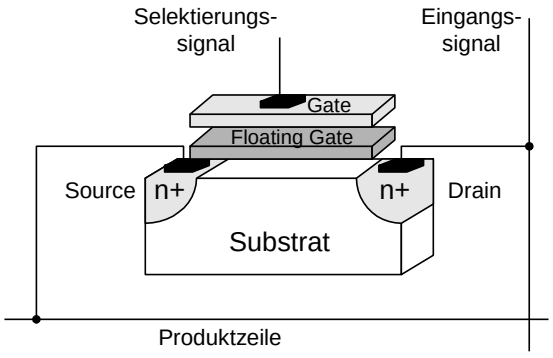


Bild 10.9 EPROM-Transistorzelle

Meist werden jetzt EEPROM- oder Flash-EPROM-Zellen genutzt, die elektrisch löschbar sind. Als weitere Entwicklung sind viele Bausteine In-System-Programmable (ISP), d.h., ein spezielles Programmiergerät entfällt, da die Programmierung über einfache digitale Leitungen (4–5#) durchführbar ist.

Der Hauptvorteil der (E)EPROM-Technologie besteht in einem guten Kompromiss zwischen Flexibilität und Programmspeicherung. Eigentlich ist kein wesentlicher Nachteil zu erwähnen – außer vielleicht der Tatsache, dass keine herausragende Eigenschaft vorhanden ist. Ein wesentlicher Nachteil wird aber zukünftig bemerkbar sein: (E)EPROM-Zellen sind nicht in gleichem Maße skalierbar wie andere MOS-Transistoren, ohne dass es zu drastischen Problemen mit der Zuverlässigkeit kommen würde.

	Anti-Fuse	**(E)EPROM**	**SRAM**
Geschwindigkeit	hoch	niedrig	niedrig
Reprogrammierbarkeit	keine	begrenzte Anzahl	beliebig oft
Mögliche Besonderheiten		In-System-Programmable	(partiell) dynamisch rekonfigurierbar

Tabelle 10.3 Zusammenfassung Speichertechnologien im Configurable Computing

Tabelle 10.3 fasst die Speichertechnologie zusammen. Grundsätzlich sind andere Technologien wie DRAM (dynamisches RAM), FRAM (ferroelektrisches RAM), MRAM (magnetoresistives RAM) und OUM (Ovonic Unified Memory) möglich, derzeit jedoch nicht in Gebrauch.

10.2.4.3 Klassifizierung nach Granularität

Das oberste Merkmal für die in Bild 10.10 gegebene Darstellung der konfigurierbaren Bausteine ist die Rechengranularität, d.h. die intrinsische Datenbreite. Hier wird nach Bitbreite oder Wortbreite (> 1 bit) unterschieden. Dies liefert die Programmable Logic Devices (PLD, z.B. Simple PLD, Complex PLD oder Field-Programmable Gate Array, FPGA) als feingranulare Strukturen und die FPFA (Field-Programmable Function Arrays, auch Field-Programmable Functional Arrays) als grobgranulare Strukturen.

Eng verbunden mit der intrinsischen Bitbreite sind auch die intrinsischen Operationen: Bei einer Datenbreite von 1 bit sind logische Verknüpfungen vollkommen ausreichend, bei > 1 bit müssen auch arithmetischen operationen vorhanden sein.

Definition 10.1:

Eine Operation wird als logisch bezeichnet, wenn für jedes Ergebnisbit dieser Operation gilt, dass es nur von einem einzigen Bit eines jeden Eingangsoperanden abhängt. Im anderen Fall wird die Operation als arithmetisch bezeichnet.

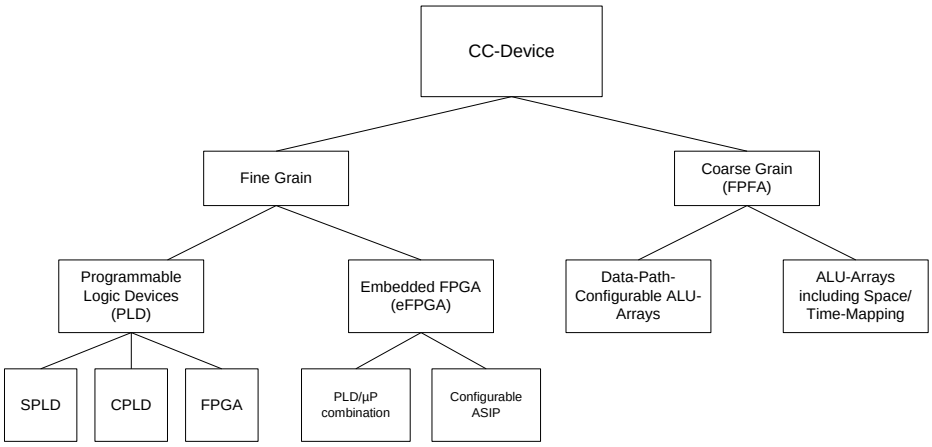


Bild 10.10 Klassifizierung der Architekturen im Bereich Configurable Computing

Beispiele für logische Operationen sind die Verknüpfungen AND, OR, XOR und NOT, auch für Bitvektoren (z.B. 8 Bit), sowie die Shift- und Rotate-Operationen (wobei hier Shift oder Rotate um eine variable Anzahl der Bits als weiterer Eingangswert schon als arithmetisch zählt). Typische Beispiele für arithmetische Operationen sind Addition und Multiplikation. Die logischen Operationen steigen im Hardwareaufwand (Fläche) im Allgemeinen linear mit der Anzahl der Eingangsbits, während die arithmetischen Operationen exponentiell (paralleler Addierer: 4^k, Carry Look-Ahead Adder: 3^k bei k bit pro Eingang) in der Fläche ansteigen.

Unterhalb dieser Ebene der intrinsischen Datenbreite variiert die Klassifizierung im linken und rechten Zweig nach Bild 10.10. Im Zweig der feingranularen Bausteine werden die Einzel- und die eingebetteten Bausteine unterschieden, die dann weiterhin in SPLD (Simple Programmable Logic Devices), CPLD (Complex Programmable Logic Devices) und FPGA (Field-Programmable Gate Arrays) aufgespittet sind – je nach Blockanzahl und –größe. Hierdurch ergeben sich auch Unterschiede in der Verbindungsmatrix, die hier jedoch nicht weiter betrachtet werden sollen.

Im rechten Zweig der grobgranularen Bausteine hingegen wird zwischen den ALU-Arrays mit bzw. ohne besondere Funktionalität – Space/Time Mapping – unterschieden.

10.2.4.4 Klassifizierung nach Programmierbarkeit

Alle Devices aus dem Bereich Configurable Computing sind programmierbar in dem Sinn, dass die endgültige Funktionalität erst durch das ”Einspielen“ des Programms erreicht wird. Tabelle 10.4 fasst die unterschiedlichen Versionen der Programmierbarkeit mit den besonderen Eigenschaften zusammen.

Programmierbarkeit	Speichertechnologie	Spezifische Eigenschaft
One-Time-Programmable	Anti-Fuse	besondere Sicherheit, ASIC-Ersatz
Erasable	EPROM, Flash-EEPROM	In kleinem Maß reprogrammierbar, z.B. für Service
In-System-Programmable	Flash-EEPROM	programmierbar nach IC-Montage, Testinterface
Rekonfigurierbar	SRAM	Beliebig oft reprogrammierbar, mit Aufwand auch zur Laufzeit
Partiell rekonfigurierbar	SRAM	wie rekonfigurierbar, zusätzlich Betrieb möglich bei partieller Reprogrammierung

Tabelle 10.4 Zusammenfassung zur Programmierbarkeit

10.3 CC-Devices und Co-Design

Wenn man die wesentlichen Eigenschaften von CC-Devices, die in Bild 10.2 und Tabelle 10.2 skizziert wurden, einmal resümiert, dann fokussiert es sich auf das zeitliche Verhalten: Mikroprozessoren arbeiten inherent sequenziell, CC-Devices inherent parallel, die Ausführungszeiten eines Algorithmus' sind entsprechend.

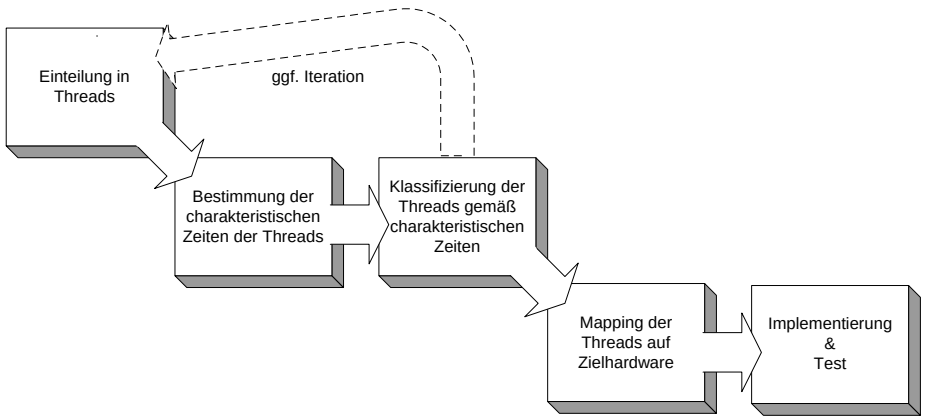


Bild 10.11 Ablauf im Co-Design bei Verwendung von Mikroprozessoren und CC-Devices

Hieraus resultiert, dass man im Co-Design das zeitliche Verhalten von Teilaufgaben betrachten muss, bevor man eine (vernünftige) Aufteilung vornehmen kann –

will man CC-Devices als Partner der Mikroprozessoren (oder auch Ersatz) nutzen. Damit ergibt sich der in Bild 10.11 dargestellte Ablauf.

10.3.1 Temporale Partitionierung im Design

Als ersten Schritt zu einer Partitionierung müssen Threads in der Applikation identifiziert werden (→ 3.1.1, Definition 3.1). Neben einem semantischen und strukturellen Zusammenhang soll bei dieser Identifizierung auch der zeitliche Rahmen des Ablaufs dieses Threads beachtet werden.

Als Beispiel sei hier ein Mikrocontrollersystem betrachtet, das folgende Aufgaben durchführen soll:

a) Messwertaufnahme durch einen AD-Converter, mit 1 MSample/s

b) Vergleich der aufgenommenen Werte mit oberer/unterer Grenze und Alarmierung, falls eine Grenzwertüberschreitung vorliegt.

c) Weiterverarbeitung jedes 1000. Wertes zur Ermittlung einer Periodendauer.

d) Kommunikation via Serial Peripheral Interface (SPI) als Slave mit Kommunikationspuffer für 16 Elemente.

Dieses Grundgerüst der Aufgaben stellt bereits eine (erste) Einteilung in Threads dar und wird im Folgenden als Basis genommen.

Die Threads werden in zweiten Schritt anhand der zeitlichen Randbedingungen festgelegt, indem die charakteristischen Zeiten (maximale Reaktionszeit, Periodendauer etc.) bestimmt werden.

Für die im Beispiel gegebenen Aufgaben seien folgende charakteristische Zeiten festgelegt bzw. bestimmt:

a) Messwertaufnahme: Periode 1 µs, maximale Reaktionszeit (ohne Puffer) 1 µs

b) Vergleich mit Grenzwerten und Start Ausnahmebehandlung: Periode 1 µs, Reaktionszeit 1 µs

c) Weiterverarbeitung der verminderten Samplingrate: Periode 1 ms, maximale Reaktionszeit (ohne Puffer) 1 ms

d) Kommunikation via SPI: Angenommene Übertragungsrate sei 4 MBit/s; dies bedeutet

auf Bitebene eine Periode von 250 ns, eine Reaktionszeit von < 125 ns (das Bit muss rechtzeitig am Bus anliegen)

auf Byteebene ohne Puffer eine Periode von 2 µs, im schlechtesten Fall eine Reaktionszeit von < 125 ns

auf Byteebene mit Puffer von 16 Bytes eine Periode von 32 µs, im schlechtesten Fall eine Reaktionszeit von < 125 ns, im besten Fall < 32 µs

Der hier jeweils angegebene schlechteste Fall bezieht sich darauf, dass innerhalb des Systems zeitlich präzise reagiert werden muss, weil z.B. der Kommunika-

tionspuffer für SPI leer ist, aber erst nach Senden des letzten Bit wieder aufgefüllt werden kann.

Im dritten Schritt werden nun die zeitlichen Randbedingungen klassifiziert, ggf. bereits unter Berücksichtigung von Modifikationen. Diese Modifikationen können z.B. die Auswahl des Mikrocontrollers mit bestimmten Schnittstellen, die Einbeziehung von Datenpufferbereichen z.B. für Kommunikation oder Auswertung usw. betreffen. Es handelt sich also um eine Konkretisierung.

Die Klassifizierung kann z.B. anhand von Kriterien durchgeführt werden, wie sie in Abschnitt 4.5, Tabelle 4.1 gegeben sind.

Im gegebenen Beispiel werden einige Konkretisierungen vorgenommen und Modifikationen eingeführt. Daraus ergibt sich dann folgende Klassifizierung:

Klasse 1: Zeiten $\leq 10\ \mu s$

Hierzu zählen die Messwertaufnahme mit einer Periode von 1 µs sowie der Vergleich mit den Grenzwerten, der ebenfalls eine Periode von 1 µs aufweist.

Die Kommunikation via SPI wird weiter aufgeteilt in die Bedienung des SPI mit charakteristischen Zeiten von 125 ns und das Füllen des Kommunikationspuffers mit 32 µs Reaktionszeit. Da die Produktion der Werte, die übertragen werden sollen mit 1 kSample/s und damit 1 ms Reaktionszeit ausfällt, wird für das Füllen des Kommunikationspuffers ebenfalls 1 ms Reaktionszeit angenommen.

Der Thread «Bedienung des SPI» gehört zur Klasse I.

Klasse 2: Zeiten ≥ 1 ms

Hierzu zählen die Verarbeitung der verminderten Messwertrate (1 kSample/s), die Bedienung des SPI (Füllung des Puffers) sowie (hier nicht weiter aufgeführt) die Ausnahmebehandlung, wenn die Messwerte den zulässigen Bereich verlassen.

Im vierten Schritt werden die Interfaces zwischen den identifizierten und klassifizierten Threads definiert und die Threads einer Implementierungsart zugewiesen (*Mapping*).

Interfaces:

a) Zwischen Messwertaufnahme und Vergleich mit Grenzwerten: Statemachine in Hardware, Signalleitungen.

b) Zwischen Vergleich mit Grenzwerten und Mikroprozessor

zwecks Initialisierung: Signalleitungen (Konfiguration der Datenwerte),

zwecks Alarmierung: Signalleitung auf Interrupt Request Eingang des Mikrocontrollers.

c) Zwischen Messwertaufnahme (verminderte Datenrate) und Auswertung: Signalleitung zum IRQ-Eingang des Mikrocontrollers mit Datenpuffer.

d) Zwischen IRQ-Service-Routine im Mikrocontroller und Auswertungsthread: Software-Event ($\rightarrow$ 4.2).

e) Zwischen Auswertungsthread und SPI: Software-Event

Implementierung:

Die Elemente der Klasse 2 werden als zeitlich unkritisch eingestuft und in einem Mikrocontroller, d.h. in Software implementiert. Die Elemente der Klasse 1 hingegen gelten als sehr kritisch und in Software nur unter besonderen Bedingungen behandelbar. Hier wird eine Implementierung in Hardware gewählt, wobei die Bedienung des SPI selbst durch eine am Mikrocontroller vorhandene Schnittstelle erfolgen kann.

Anmerkung: Zwischen 10 µs und 1 ms ist ein Bereich, der eine gewisse Grauzone darstellt (siehe auch 4.5). Sollten charakteristische Zeiten in diesem Bereich auftreten, muss im Einzelfall über die Klassifizierung entschieden werden.

Es folgen dann natürlich die Implementierung und der Test als letzte Schritte. Das wesentliche Problem dabei ist dann, dass die Implementierungssprache (und auch das Paradigma) gewechselt werden muss, beispielsweise von C/C++ auf VHDL oder Verilog.

10.3.2 Configurable ASIPs als Zielhardware

Das in 10.3.1 vorgestellte Verfahren basiert auf manueller Konstruktion. Ziel ist es dabei, die Reaktionszeiten und damit das Echtzeitverhalten des Systems beherrschbar zu gestalten.

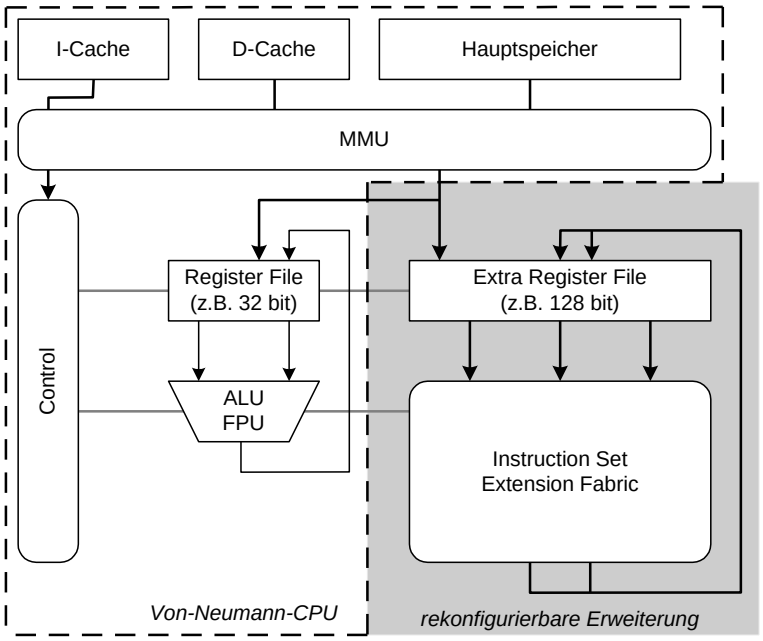


Bild 10.12 cASIP-Architektur

Es existieren auch automatische Verfahren zum Co-Design, allerdings momentan nur mit anderen Zielfunktionen. Ein Verfahren betrifft dabei die konfigurierbaren Mikroprozessoren, auch als cASIP (configurable Application-Specific Instruction Set Processors) bezeichnet.

Um die Mikroprozessorarchitektur konfigurierbar zu gestalten, wird parallel zur ALU (Arithmetisch-logische Einheit) bzw. zur Floating-Point Unit FPU eine konfigurierbare Struktur eingefügt (→ Bild 10.12) [Sie05a]. Diese konfigurierbare Einheit (Instruction Set Extension Fabric) arbeitet mit einem eigenen Registersatz unabhängig von dem üblichen Teil des Mikroprozessors mit ALU, Floating-Point Unit und Register File. Die Verbindung dieser zwei Teile erfolgt über die Control Unit, die die Steuerung jeweils vornimmt, und natürlich den gemeinsamen Speicher.

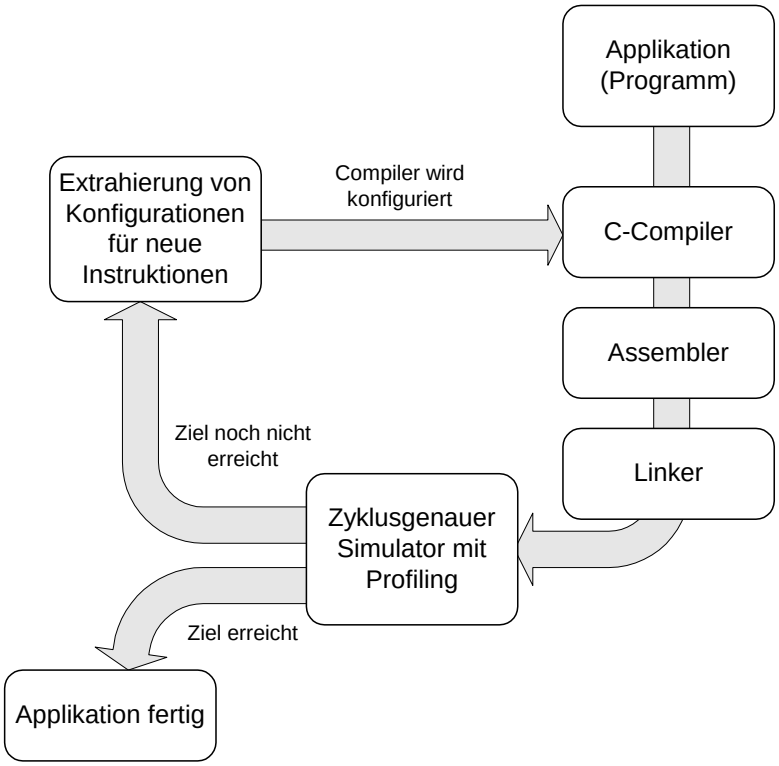


Bild 10.13 Designzyklus bei konfigurierbaren ASIPs (cASIP) [Sie05a]

Die cASIP-Architektur muss in engem Zusammenhang mit einer zugehörigen speziellen Compilertechnologie gesehen werden (→ Bild 10.13). In einer Kombination von Übersetzung einschließlich Transformation von Teilen in die rekonfigu-

rierbare Erweiterung und zyklusgenauer Simulation/Profiling werden Konfigurationen solange erzeugt, bis das Ziel der Laufzeitverbesserung erreicht ist.

10.3.3 FPFAs als Zielhardware

FPFA (Field-Programmable Function|Functional Arrays) stellen eine weniger bekannte, aber am Markt erhältliche programmierbare Architektur dar. Im Gegensatz zu den auf Bitebene konfigurier- oder programmierbaren PLDs ist hier die Verarbeitungsbreite auf Wortbreite eingestellt, ähnlich wie in Mikroprozessoren. Diese FPFAs sind also etwas Mikroprozessor-ähnlicher, und für sie sollen die Möglichkeiten zum automatischen Co-Design erörtert werden.

10.3.3.1 Struktur von FPFAs

Als Grundlage für die folgende Diskussion zur Compilertechnologie dient die von PACT [pact] vorgestellte Architektur, mit XPP bezeichnet (*eXtreme Processing Platform*). Diese Plattform ist als Coprozessor konzipiert, mit Hilfe eines RISC-Prozessorkerns, der auch auf dem Chip integriert ist, wird hieraus ein vollständiger Prozessor mit enormen Rechenkapazitäten.

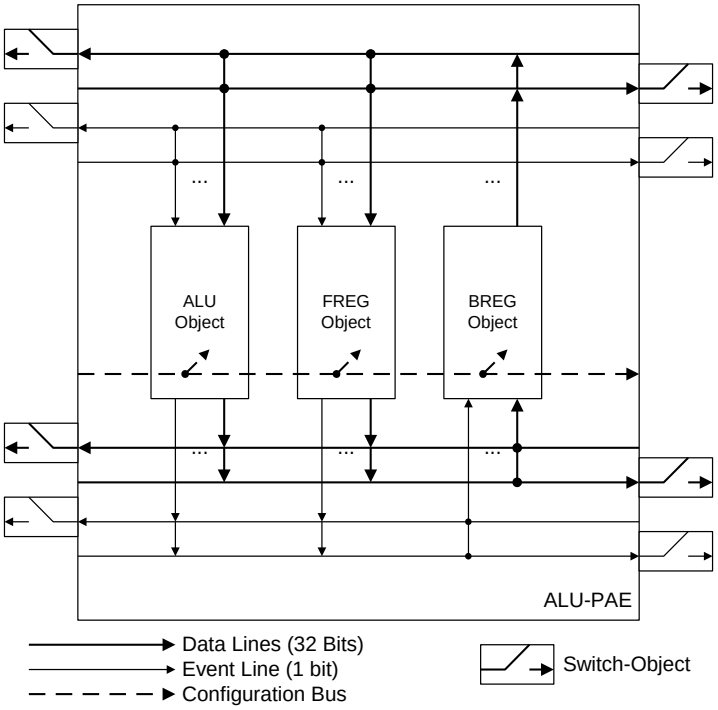


Bild 10.14: Processing Array Element (ALU-Typ) mit Switch Objects

Die XPP-Architektur weist 4 Ebenen der Hierarchie auf, die der Hardware-Objekte, der Processing Array Elements (PAE), der Processing Array Cluster (PAC) und die der Devices selbst. Die Hardware-Objekte beinhalten elementare Operationen wie Verknüpfungen, Speicher oder Routing. Aus Sicht der Konfiguration sind dies die Ziele, mit denen die Laufzeit-Struktur des Bausteins festgelegt wird.

Der/die Softwareentwickler/in sieht hingegen die Processing Array Elements, die aus Objekten zusammengesetzt sind. Er/sie kann ein PAE für den Algorithmus nutzen, teilweise sogar die Objekte darin. Der Algorithmus wird aber nicht nur auf ein PAE, sondern auf den Cluster (PAC) oder sogar auf das gesamte Device abgebildet. Diese nächsthöheren Ebenen dienen damit u.a. der Chipverwaltung und der Skalierbarkeit, weniger der Programmierung.
Etwas vereinfacht dargestellt besteht ein XPP also aus einem oder mehreren Clustern (PAC) von einigen (< 100) ALU-oder ähnlichen Elementen (PAE), stellt also einen FPFA (Field-Programmable Function Array) dar.

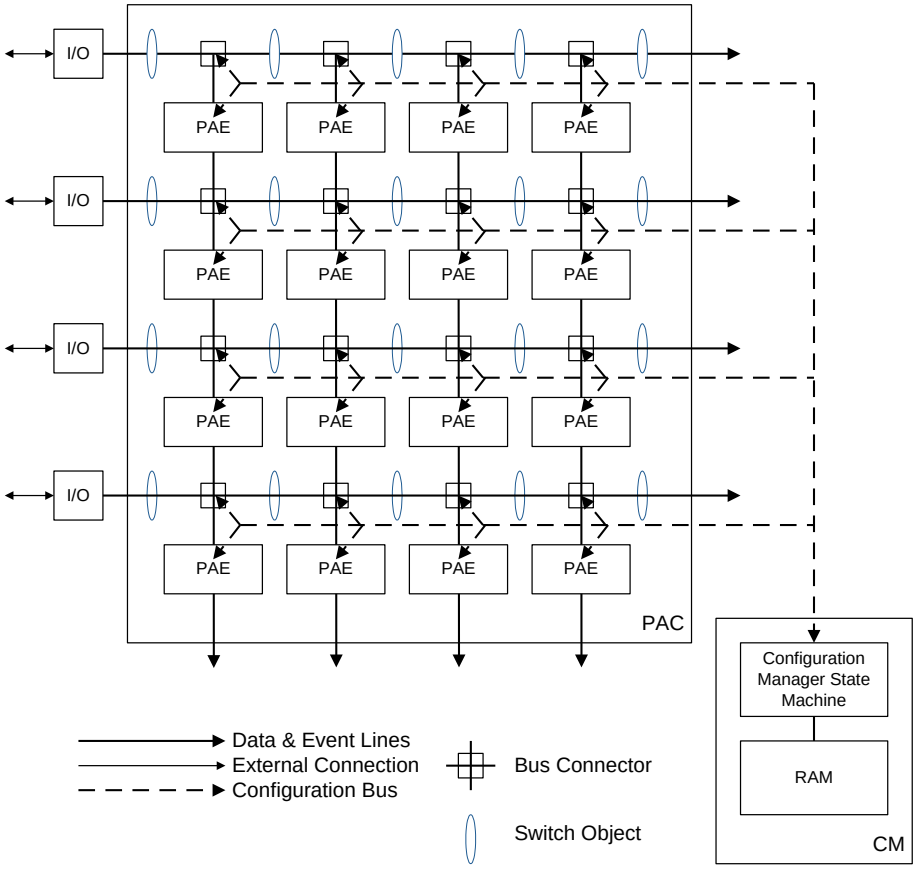


Bild 10.15: Aufbau Processing Array Cluster (PAC) der XPP-Architektur

10.3.3.2 Programmiermodell für FPFAs

FPFA (Field-Programmable Function|Functional Arrays) besitzen ein Programmiermodell, das dem der PLDs entspricht: Die komplette Nebenläufigkeit aller Aktionen, die Kommunikationsfähigkeit über das gesamte Device (zumindest im Prinzip) und die Ausführung einer Aktion in einem Takt, verbunden mit einer gegebenen Synchronisation, ermöglichen ein solches Modell, das wesentlich Basis-orientierter ist als das für µP/PLD-Kombinationen.

Der wesentliche Unterschied zu PLD-Modellen ist natürlich leicht zu erkennen: Die intrinsische Datenbreite ist bei PLDs 1, bei FPFAs jedoch > 1 (z.B. 32). Die ist die wesentliche Nähe zu Mikroprozessoren.

Das Programmiermodell für FPFAs wird zurzeit so behandelt, dass die Anzahl der Array-Elemente und der Register zur Modellierungs- oder Compilierungszeit bekannt ist.

10.3.3.3 Compilertechnologie

Am Beispiel dieser Array-Architektur soll erläutert werden, was derzeit im Rahmen eines C-Compilers an Möglichkeiten zur Programmerzeugung bestehen. Zunächst muss hierzu der Übersetzungsprozess diskutiert werden, schließlich ist die XPP-Architektur als Co-Prozessor ausgelegt. Hieraus können sich Sourcecodeeinschränkungen ergeben.

Die eigentliche Übersetzungsarbeit hat mehrere Eckpunkte: Natürlich lassen sich die einzelnen Statements sehr schön auf die Array-Elemente abbilden, die (Zeit-) sequenzielle Folge ist dann an der konfigurierten Verdrahtung zu sehen. Allerdings ist es das erklärte Ziel dieser Architektur, möglichst viele Operationen parallel zueinander auszuführen

Der typische Übersetzungsprozess für FPFAs ist in [CW02] beschrieben und in Bild 10.16 dargestellt. In der *Preprocessing-Phase* werden Architektur-unabhängige Schirtte durchgeführt, insbesondere aber `FOR`-Loops, die vom Softwareentwickler als solche gekennzeichnet sind, aufgelöst (loop unrolling, [Sie04, 6.2.2]). Wieterhin wird per Datenabhängigkeitsanalyse versucht, die inneren `FOR`-Loops für eine überlappende Ausführung zu identifizieren.

Damit steht bereits ein Kandidat von Programmkonstrukten fest, der für eine Übersetzung in das XPP-Array wichtig ist: `FOR`-Loops. Hiermit sind speziell solche Schleifen gemeint, deren Ausführungsanzahl bereits zur Compilezeit bekannt ist. Schleifen mit Laufzeit-bestimmter Schleifenanzahl werden meist als `WHILE`- bzw. `DO-WHILE`-Schleife bezeichnet.

Der Vorgang der temporalen Partitionierung (TempPart) ist der wohl wichtigste und neueste im Bereich des rekonfigurierbaren Rechnens. Ausgehend davon, dass das Programm nicht komplett in ein PAC passt, wird in dieser Phase versucht, Teilkonfigurationen zu bilden. Hierdurch entsteht echtes *Reconfigurable Computing*.

Im Anschluss an diese Vorgänge sind die Generierung der Module in der Sprache *Native Mapping Language* (die Assemblersprache der XPP-Architekturen, hier werden den mit Koordinaten benannten PAEs Aktionen zugewiesen) und der Rekonfigurationsanweisungen.

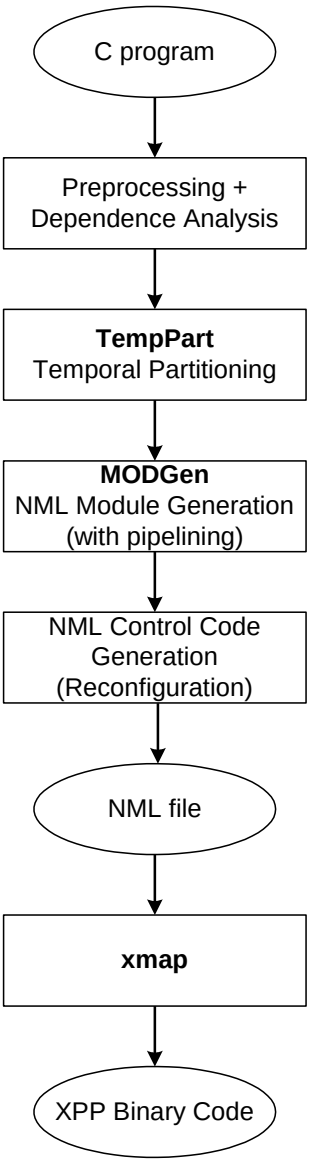


Bild 10.16 XPP-VC Übersetzungsfluss

Anmerung: Der Begriff der temporalen Partitionierung wure bereits im Abschnitt 10.3.1 verwendet. Dort galt es, Threads in einer Applikation nach (Ausführungs-)zeitlichen Gesichtspunkten zu identifizieren, wobei durch den/die SoftwareentwicklerIn eine Semantik berücksichtigt wird.

An dieser Stelle kann keine Semantik genutzt werden, da Compiler nicht in der Lage sind, diese zu identifizieren. Hier wird also ein rein zeitliches Verhalten als Grundlage gewählt, wobei die Identifizierung dieses durchaus problematisch ist.

10.3.3.4 Pipeline Vectorization

Durch ein mit *Pipeline Vectorization* bezeichneten Verfahren [WL01] wird die parallele Ausführung mehrerer Loop-Durchläufe ineinander verschachtelt, so dass möglichst wenig Speicherzugriffe mit möglichst hoher paralleler Ausführung der Rechenoperationen gekoppelt werden.

Das Verfahren umfasst die folgenden Schritte:

- Kandidatenselektion
- Schleifennormalisierung
- Abhängigkeitsanalyse
- Auflösung von Array-Abhängigkeiten
- Generierung des Datenflussgraphen
- Einführung von Feedbacks
- Pipelining und Timing

Kandidatenselektion

Als notwendiges Kriterium für einen geeigneten Kandidaten zur Pipeline-Vektorisierung gilt: Innerste Schleife, `FOR`-Loop (oder auch `WHILE`-Loop mit Indexvariablen) mit konstanten Grenzen.

Schleifennormalisierung

Eine Schleifennormalisierung soll alle Abhängigkeiten auf eine einzige Variable zurückführen. Dies bedeutet, dass weitere Variablen auf diese abgebildet werden müssen. Diese Indexvariable soll mit dem Index 0 beginnen und eine Schrittweite von 1 haben.

Weiterhin ist gefordert, dass alle Abhängigkeiten von dieser Indexvariablen linear sein sollen, also bei der Indexvariablen I in der Form S*I + C. S gilt dabei als die Zugriffsweite (*access' stride*).

Lässt sich eine Forderung nicht erfüllen, gilt die Schleife nicht mehr als Vektorisierungskandidat.

Abhängigkeitsanalyse

Es existieren drei grundsätzliche Typen von Datenabhängigkeiten in einem Programm:

- *True* oder *Flow Dependences*: Der Wert einer Variablen ist in einer Zuweisung definiert und wird lesend in einer nachfolgenden Anweisung benötigt (Read-After-Write, RAW).
- *Anti-Dependences*: Eine Variable wird in einer Anweisung gelesen, und in einer nachfolgenden Anweisung wird der Variablen ein neuer Wert zugewiesen (Write-After-Read, WAR).
- *Output Dependences*: Zwei Wertzuweisung in unterschiedlichen Anweisungen erfolgen an eine Variable (Write-After-Write, WAW).

Diese drei Abhängigkeiten entsprechen denjenigen aus der superskalaren Welt [Sie04, 5.2]. Während in der Softwareanalyse jedoch häufig die internen Prozessorregister analysiert werden, existieren hier diese ausgezeichneten Register nicht: Für FPFAs (Field-Programmable Function Arrays) werden Variablen in Speicherarrays gelagert (skalare Variable können entweder aufgelöst werden, einen temporären Charakter haben und dann in einem internen Register gespeichert werden, oder müssen ebenfalls im Hauptspeicher stehen).

Für das hier beschriebene Verfahren ist es weiterhin interessant, ob die Abhängigkeit Schleifen-unabhängig oder Schleifen-getragen ist. Im ersten Fall stehen die beiden Anweisungen innerhalb einer Schleife, im zweiten Fall über mindestens eine Schleifeniteration hinweg. Als Abhängigkeitsdistanz (Dependence Distance) bezeichnet man die Anzahl der Iterationen zwischen diesen Statements.

Falls nun das Rückschreiben auf das Speicherarray immer in-order verläuft (so lautet die Grundannahme beim XPP), sind ausschließlich die True Dependencies von Interesse. Für diese wird geprüft, ob die analysierte Distanz (0: Loop-independent) immer auftritt, diese werden als regulär bezeichnet.

Die Synthese wird dann unter Beachtung der Abhängigkeiten durchgeführt. Abhängigkeiten führen dann zu weniger Parallelität, weil spätere Schleifendurchläufe nicht mit berücksichtigt werden können.

Auflösung von Array-Abhängigkeiten

Grundsätzlich ist es nicht notwendig, die Array-Abhängigkeiten weiter aufzulösen, allerdings ist das Handling für skalare Variable einfacher (und in dem in [WL01] dargestellten Algorithmus essentiell). Um die Schleifen-getragenen Abhängigkeiten auflösen zu können, werden für jede gelesene Arrayvariable eine skalare substituiert.

Generierung des Datenflussgraphen

Aus dem nunmehr transformierten Sourcecode wird im nächsten Schritt ein kombinatorischer, azyklischer Datenflussgraph erstellt. Hierzu werden die internen Da-

tenabhängigkeiten analysiert, für jede Operation ein eigener Operator ausgewählt und mit den Quellen und Senken verbunden.

Das gewählte Verfahren ist also eine einfache, direkte Übersetzung des Schleifenkörpers. Im Schleifenkörper werden keinerlei Ressourcen gemeinsam genutzt, es wird also mit maximaler Parallelität gearbeitet. Zudem – und das ist das Ziel dieses Verfahrens – ermöglicht dies im Anschluss ein Pipelining über mehrere Schleifendurchläufe.

Da Zyklen nicht erlaubt sind (es wird auch immer nur die innerste Schleife übersetzt!), kann nur linearer Code vorkommen. Verzweigungen werden parallel zueinander übersetzt, und ein Multiplexer wählt das jeweilige Ergebnis aus. Diese Parallelitäten können ggf. sogar wieder auf die gleiche Hardware abgebildet werden, wenn beispielsweise zwei Pfade sich gegenseitig ausschließen (if ..., else ...), aber auf ähnliche Verknüpfungen abgebildet werden.

Für ALU-Arrays gilt, dass verschiedene Zweige, sich gegenseitig ausschließend, grundsätzlich auf die gleiche ALU abgebildet werden können, wenn die ALU-Funktion durch Laufzeit-Bedingungen umschaltbar ist (Event-Funktion bei XPP).

Obwohl dies sehr einfach und direkt aussieht, bergen die bedingten Verzweigungen ein Problem in sich. Angenommen, in verschiedenen Zweigen werden nur einem Teil der Variablen neue Werte zugewiesen, was passiert dann mit den nicht-zugewiesenen Variablen? Diese behalten nach allgemeinem Syntaxverständnis (in Software) ihren Wert, und die muss im Datenfluss berücksichtigt werden. Leider führt dies zu Schleifen im Schleifenkörper, was gesondert behandelt werden muss.

Einführung von Feedbacks

Um die Schleifen-abhängigen Datenabhängigkeiten korrekt umsetzen zu können, müssen Daten rückgekoppelt werden. Hierbei sind der Initialisierungsfall und der Betriebsfall für Schleifen noch zu unterscheiden: Solange die Abhängigkeitsdistanz noch nicht durchlaufen ist, gibt es auch keine korrekten Werte zur Rückkopplung.

Die Strategie ist sehr einfach: Eingangsseitig wird bei den rückgekoppelten Variablen, die also in einem Schleifendurchlauf gelesen und geschrieben werden, zwischen der Initialisierung und der Rückkopplung unterschieden, etwa durch eine Hilfsvariable. Die Rückkopplung erfolgt auf ein Speicherregister zu Beginn der Schleifenhardware, so dass der Durchlauf mit den alten Werten beginnt und die neuen erzeugt.

Pipelining und Timing

Die jetzt vorhandenen Datenflussgraphen zielen auf die Berechnung eines Ergebnisses (oder eines Vektors von Ergebnissen) innerhalb eines Takts, sieht man einmal von der Ladephase ab, falls mehrere Arrayelemente hierfür notwendig sind. Diese Form der Berechnung kann sehr ineffizient sein, wenn sich die Laufzeiten der einzelnen Operatoren addieren (bei erzwungener Sequenzialität).

Die Methode zur Reduzierung der Laufzeit heißt *Pipelining*: Eine Aktion, hier die komplette Berechnung des Schleifenkörpers, wird in mehrere Teilaktionen mit Zwischenspeicherung der Ergebnisse zerlegt. Die Teilaktionen werden sukzessive durchlaufen und die Ergebnisse taktgesteuert gespeichert, wobei eine durchlaufene Pipelinestufe im Anschluss (nächster Takt) wiederum den nächsten Teilwert berechnen kann.

Pipelining erhöht zugleich die Latenzzeit, d.h. die Zeit, die der erste und der letzte Wert zur Fertigstellung benötigen (Filling/Flushing der Pipeline). Dies wird meist nicht als Problem angesehen, weil diese Zeit meist vernachlässigbar klein gegenüber der restlichen Laufzeit ist, stellt aber ein Problem bei Schleifen mit nur wenigen Durchläufen dar.

Pipelining muss für jeden Zweig die gleiche Anzahl der Stufen erzeugen. Die Berechnung kann mit sehr vielen Pipelinestufen versehen werden, mit dem Effekt, dass die Taktzeit sehr klein wird. In der Praxis wird es jedoch Begrenzungen geben: Zum einen ist der oftmals vorhandene Speicherzugriff meist begrenzend, zum anderen können Rückkopplungen, die nicht weiter unterteilt werden können (dies würde das Verhalten ändern!), nur in einer Pipelinestufe realisiert werden. Wenn die Rückkopplungszeit $t_{feedback}$ (Zeit zur Berechnung eines Werts und zur Rückkopplung) und die Anzahl der Taktzyklen, die für einen Speicherzugriff notwendig sind, bekannt sind, gilt für die Anzahl N_{PC} an Taktzeiten T_C eines gegebenen Takts folgende Gleichung:

$$N_{PC} = \max\left(\left\lceil \frac{T_{feedback}}{T_C} \right\rceil, N_{mem}\right) \tag{10.1}$$

Die Zeit, die ein Wert zur Berechnung benötigt, ist dann $N_{PC} * T_C$, bei Vernachlässigung der Füll- und Leerzeiten wird dann eine Schleife mit n Iterationen in der Zeit $n * N_{PC} * T_C$ bearbeitet.

Für ALU-Array-basierte Berechnung (FPFA, XPP) wird für das Pipelining meist ein vereinfachter Ansatz gewählt: Das Ergebnis jeder Berechnung wird einfach gespeichert. Wichtig ist hierbei, dass Zweige mit kürzerem Weg verzögert werden müssen, um die Korrektheit der Ergebnisse sicherzustellen.

10.3.3.5 Loop Transformationen

Loop Unrolling ist ein Begriff aus der Compilertechnologie insbesondere für superskalare Prozessoren [Sie04]. Dies wird auch – stärker spezifizierend – als Software Unrolling bezeichnet. Die wesentlichen Schritte bestehen darin, dass zwei (oder mehrere) Schleifendurchläufe aneinandergelegt, ein (Compiletime) Register Renaming durchgeführt und die Instruktionen dann neu sortiert (instruction scheduling) werden, um z.B. überflüssige Speicherzugriffe zu verhindern. Der positive Laufzeiteffekt tritt durch die Minderung der Speicherzugriffe und die potenzielle

Erhöhung der Instruktionsparallelität ein. Gleiche Effekte können auch durch *Loop Merging* erzielt werden.

Während die Zielrichtung in Prozessor-basierten Systemen fast ausschließlich die Vergrößerung der Schleifen ist, liegt bei FPFA- bzw. FPGA-basierten Systemen eine Notwendigkeit zur Anpassung der Schleifen vor. Der beschriebene Algorithmus vektorisiert immer die innerste Schleife, so dass genau zwei Probleme auftreten: Der Schleifenkörper ist zu groß, um in die Hardware zu passen, oder zu klein, um einen Performancegewinn (aufgrund des Hardware-Overheads) zu bringen.

Loop Tiling adressiert den umgekehrten Fall, der entsteht, wenn eine komplett entrollte Schleife zu groß für die aufnehmende Hardware ist. Der Algorithmus versucht in diesem Fall, nur einen Teil der Schleifendurchläufe zu entrollen und komplett in Hardware zu mappen, während ein Teil der Schleifenstruktur erhalten bleibt.

```
for( i = 0; i < m; i++ )
{
  PRE( i );
  for( j = 0; j < N; j++ )
    F( i, j );
  POST( i );
}
```

Bild 10.17 Schleifenstruktur

Für eine ”gute“ Schleifenzerlegung kann eine Schätzung gegeben werden. Angenommen, die Schleife hat den allgemeinen Charakter wie in Bild 10.17 dargestellt, und es existieren Schätzungen für den Hardwareaufwand für PRE(i) ($area_{PRE}$), POST(i) ($area_{POST}$) und F(i, j) ($area_F$). Bei bekannten Hardwareressourcen ($area_{HW}$) kann dann t_{SIZE} angegeben werden:

$$t_{SIZE} = \left(area_{HW} - area_{PRE} - area_{POST} \right) / area_F \qquad (10.2)$$

```
for( i = 0; i < m; i++ )
{
  PRE( i );
  for( jt = 0; jt < (n-1)/tsize+1; jt++ )
    for( j = 0; j < min(tsize, n-jt*tsize); j++ )
      F( i, j + jt*tsize );
  POST( i );
}
```

Bild 10.18 Loop Tiling in Anwendung auf Code aus Bild 10.17

Hieraus lässt sich eine neue Version der Schleifenkonstruktion aus Bild 10.17 angeben (Bild 10.18). Leider ist dies nicht die Lösung, denn die innere Schleife besitzt keinen konstanten oberen Wert. M.a.W.: Jeder Schleifendurchlauf (in jt-Schleife) hätte einen anderen Schleifenkörper (denn die innere Schleife soll ja gerade vektorisiert werden.

Die Lösung besteht darin, das Minimum min(*tsize*, *n-jt*tsize*) durch *tsize* zu ersetzen (*tsize* ist eine Konstante, so dass das Minimum niemals größer sein kann) und die Korrektheit der Rechnungen (*n-jt*tsize* könnte ja kleiner als *tsize* sein) durch zusätzliche Bedingung

$$j < n - jt \times tsize \tag{10.3}$$

$$jt \neq \left(n-1\right) / tsize \vee j < \left(n-1\right) \% tsize - 1 \tag{10.4}$$

zu garantieren.

10.3.3.6 Temporale Partitionierung

Der letzte Punkt der hier behandelten, aktuellen Compilertechnologie zur Übersetzung in ALU-Arrays betrifft die temporale Partitionierung (siehe auch Bild 10.16). Dies ist deswegen so interessant, weil sich zum ersten Mal der automatische Weg in ein **Re**configurable Computing aufzeigt.

Temporale Partitionierung wird zurzeit dann angewendet, wenn die Zielhardware (bei XPP: ein PAC) zu klein ist, um den kompletten Teil aufzunehmen. Bild 10.19 zeigt den Ablauf beim vektorisierenden XPP-Compiler. Zunächst wird für das C-Programm, bei dem nun schon die oben erwähnten Sourcecode-Transformationen durchgeführt sind, durch den SUIF-Compiler [suif] ein Abstract Syntax Tree (AST) aufgestellt. Hierin befinden sich Knoten, vergleichbar mit Basis-Blöcken für Mikroprozessoren.

Für jeden dieser AST-Knoten wird geschätzt, wie viele Ressourcen er im PAC benötigt, und benachbarte Knoten werden solange hinzugemischt, bis der Cluster voll ist. Oberstes Prinzip ist dabei, dass Partitionen mit einem Eintrittspunkt, ggf. aber mehreren Austrittspunkten zusammengestellt werden.

Die Schätzungen werden im Nachgang auf Validität verifiziert. Bei Überschreitung der PAC-Kapazität wird eine Neuübersetzung bzw. –zusammensetzung initiiert. Die Folge der entstehenden Konfigurationen wird dann durch den Configuration Manager (CM) der XPP-Architektur behandelt.

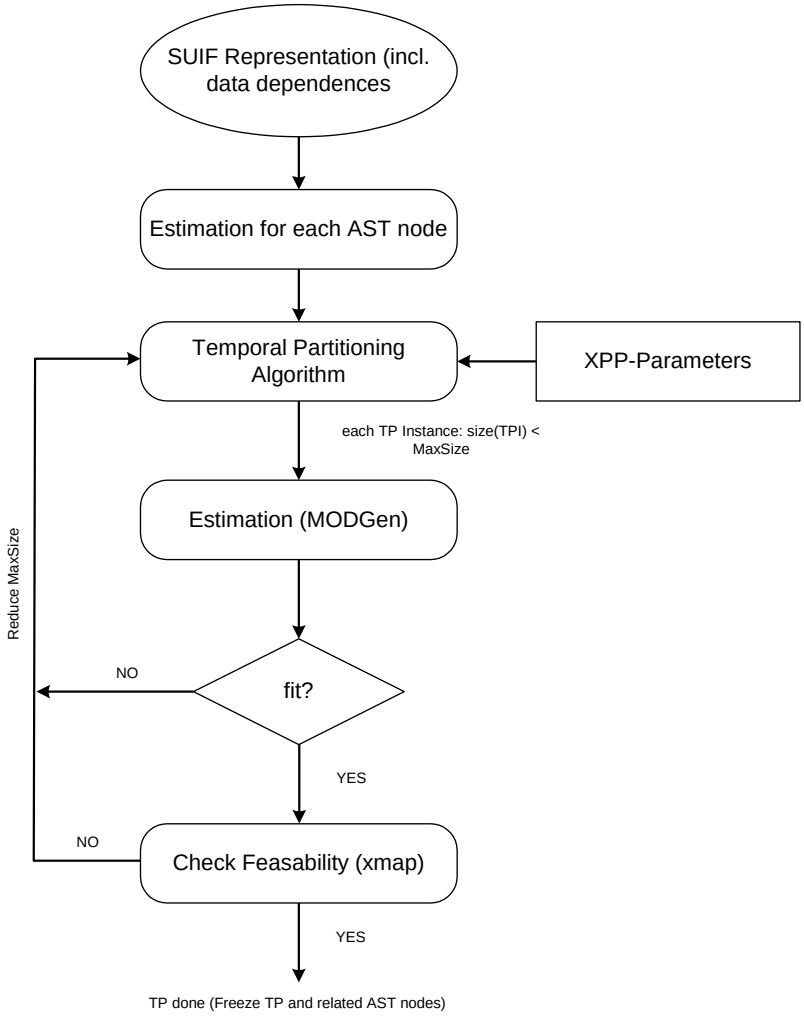


Bild 10.19 Ablauf temporale Partitionierung im XPP-C-Compiler

10.3.4 FPFAs mit Space/Time-Mapping

Space/Time-Mapping bedeutet, dass eine Programmrepräsentation, also z.B. ein Binärcode, zur *Laufzeit* auf einer Hardware ausgeführt werden kann, die in der Zeitsequenz ausführt (Computing in Time), oder auf einer Hardware, die als Struktur in der Fläche ausführt (Computing in Space). Das ist natürlich eine hohe Anforderung, und aktuell verfügbare Bausteine hierfür existieren nicht, dennoch ist dies nicht ausgeschlossen.

Die Möglichkeiten, die sich hierdurch bieten, sind u.a.:

- Das Sourcecodeprogramm muss nur noch einmal übersetzt werden und ergibt eine Binärdatei, die alle notwendigen Informationen zumAblauf beinhaltet. Der unterschiedliche Ablauf der Programme, einmal in der Struktur, einmal in der Zeitsequenz, wäre zwar mit maximaler Effizienz im Übersetzungsvorgang zu berücksichtigen, in der Softwareentwicklung aber eben nicht – zumindest in erster Näherung nicht.
- Die Ausführungsgeschwindigkeit und Echtzeitfähigkeit – und damit das komplette Zeitverhalten – kann zur Laufzeit verändert werden. Die Wahl heißt *Ausführungszeit versus Ressourcen (Fläche)*, und es gilt genähert das Gesetz $A * T^2 = \text{const}$. Diese Möglichkeiten müssen vom Betriebssystem ausgenutzt werden.

Die praktische Ausführung einer Space/Time-Mapping-fähigen Architektur ist allerdings sehr schwierig. Ein Ansatz hierzu ist in [SFS+05] dargestellt und wird im Folgenden erläutert.

UCB - Universal Configurable Blocks

Dieser Ansatz besteht aus zwei Komponenten, einer Architektur und einem zur Laufzeit ausführbaren Algorithmus zur Umwandlung von binären Informationen. Die Architektur – als UCB bezeichnet – besteht aus sehr großen Blöcken, die ihrerseits Elemente der Arithmetisch-Logischen Einheit(en) und Registersätze beinhalten (→ Bild 10.20).

Ein derartiger Block ist in der Lage, einen gewissen Anteil eines Algorithmus (bzw. auch den ganzen) als Konfiguration aufzunehmen, indem die im Algorithmus vorliegende Reihenfolge von (Mikroprozessor-ähnlichen) Instruktionen in eine räumliche Sequenz übertragen wird. Voraussetzung ist lediglich, dass alle Write-After-Write-Hazards (WAW, [Sie04, 5.2]) aufgelöst sind und dass der übersetzte Block ein Basis- oder Hyperblock ist [Sie04, 6.2.3], also keine Programmsprünge enthält.

Hierauf basiert das Ausführungs- und Übersetzungskonzept: Eine ausführende Maschine – mit UCM, Universal Configurable Machine bezeichnet [SFS+05] – enthält einen einfacher Ausführungsteil, der wie ein Mikroprozessor arbeitet, einen Übersetzungsteil, der den Instruktionsfluss mithilfe eines PDSP-Algorithmus (Procedural-Driven Structural Programming) in eine Struktur übersetzt, sowie einen oder mehrere UCBs, die Programmteile komplett speichern und mit erhöhter

Ausführungsgeschwindigkeit und sofortiger Verfügbarkeit ausführen (→ Bild 10.21).

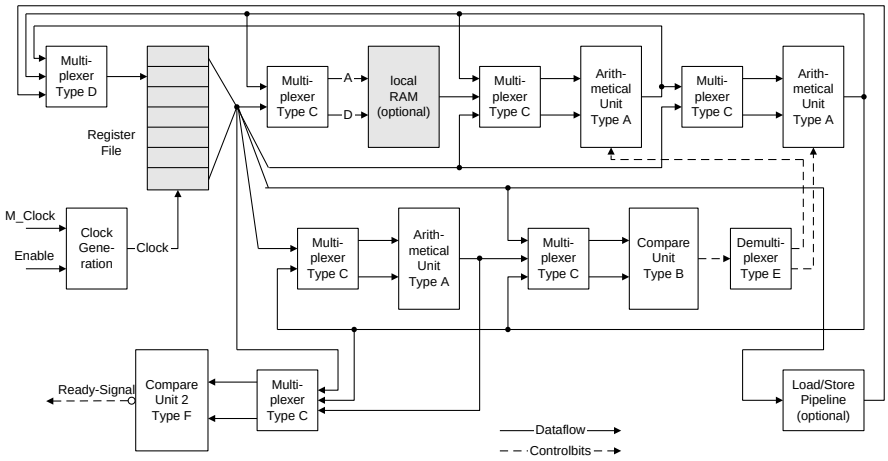


Bild 10.20 Blockstruktur Universal Configurable Block

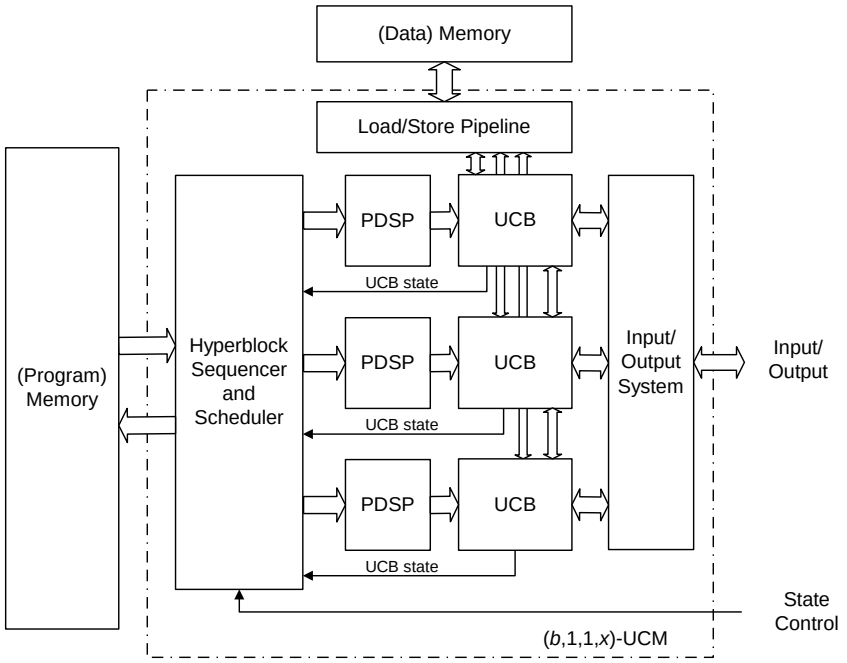


Bild 10.21 Blockstruktur Universal Configurable Machine

Diese Architektur ist tatsächlich universell, und sie kann die Vorteile der verschiedenen Architekturmodelle jeweils ausnutzen, und zwar zur Laufzeit. Demgegenüber steht ein erhöhter Aufwand in der Hardware und auch der Systemsoftware.

Abschnitt III: Verteilte Eingebettete Systeme

11 Netzwerke und Standards

Das Kapitel Netzwerke und Standards – im Rahmen eingebetteter Systeme – ist speziell auf Embedded Systems Engineering ausgerichtet und stellt somit kein allgemeingültiges Kapitel für Netzwerke dar. Die Gründe hierfür sind relativ einfach dargestellt: Zum einen nutzen eingebettete Systeme meist eine Kommunikation auf Layer 2 des ISO/OSI-Layermodells (→ 11.1.1), so dass das Interesse an den Layer 3 bis 6 meist gering ausgeprägt ist. Hier wird gerne – wenn überhaupt – der IP-Stack als Mittel der Wahl genommen.

Zum anderen werden andere Eigenschaften von Netzwerken, z.B. eine Echtzeitfähigkeit, Sicherheitsaspekte etc., viel mehr im Vordergrund stehen als bei Netzen z.B. der Bürokommunikation. Aus beiden Gründen wird in diesem Handbuch darauf verzichtet, eine allgemeingültige Darstellung der Themas Netzwerke zu integrieren, vielmehr werden spezielle Anforderungen in eingebetteten Systemen hier behandelt.

Das Kapitel gliedert sich somit in eine allgemeine Einführung in das ISO/OSI-Referenzmodell, gefolgt von Klassifizierungen von Netzwerken nach verschiedenen Gesichtspunkten. Beide Unterkapitel sind eher „klassischer“ Natur.

Im dritten Unterkapitel werden einige physikalische Grundlagen – Basis für den „Physical Layer“ (1) – behandelt, im vierten dann die Leitungscodierung. Layer 2, als Data Link Layer bezeichnet, wird insbesondere im Hinblick auf Echtzeitfähigkeit im fünften Unterkapitel dargestellt, gefolgt von Betrachtungen zur Sicherheit (Security) in Netzwerken.

11.1 ISO/OSI-Referenzmodell

11.1.1 Schichtenmodell

Ein offenes Kommunikationssystem kann als komplexes System angesehen werden, das keineswegs mehr von nur einer Person alleine realisiert werden kann. Dies allein erfordert grundsätzliche Gedanken zur Strukturierung dieses Systems. Eine strikte Modularisierung der Software, wie sie generell empfohlen wird, bringt weitere Forderungen in Richtung Strukturierung: Die Definition der Architektur von Kommunikationssystemen ist zwingende Voraussetzung für eine ‚offene‘ Implementierung.

Eine Standardarchitektur wurde 1978 von der ISO als Open System Interconnection (OSI) entworfen. Diese Standardarchitektur wurde derart entworfen, dass sie möglichst für alle Anwendungsfelder nutzbar war, mit zwei (negativen) Konsequenzen: Einzelne Schichten können durchaus leer sein, und die Implementierung eines Gesamtprotokolls kann sehr laufzeitintensiv sein.

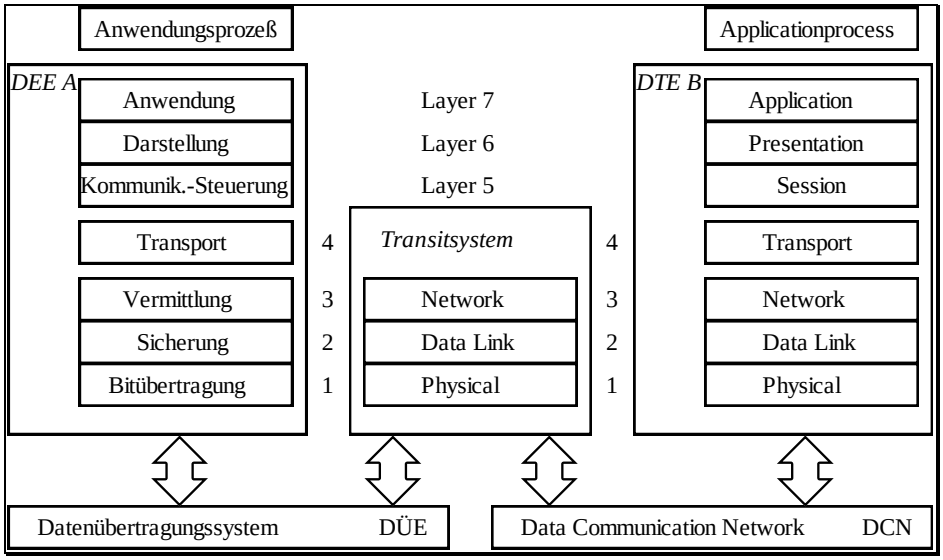


Bild 11.1 OSI-Basisreferenzmodell

Ein Kommunikationssystem wird in *Layer* oder *Schichten* eingeteilt. Jeder Schicht wird dabei eine Menge von Funktionen zugeordnet, wobei den höheren Schichten alle Funktionen der niedrigeren zur Verfügung stehen, selbst jedoch weitere zur Verfügung stellen können (Bild 11.1).

Andererseits soll die eigentliche Applikation keinesfalls den vollen Kanon aller Funktionsaufrufe und Dienste sehen, da dies zu einer unerwünschten Komplexität führen würde. Schicht 7 fasst also die gesamten Funktionen zusammen und bietet dem Anwendungsprozess nur wenige Dienstprimitive (service primitives) an. Im Übrigen sollte diese strikte Trennung der Schichten voneinander auch in den Layern 1 – 6 durchgeführt sein: Ein Schicht n kennt im Idealfall nur die Schichten n-1 und n+1. Die Praxis zeigt dabei aber einen tieferen Durchgriff, als dies in der Theorie gewünscht ist.

Die 7 Schichten werden grob zu zwei Teilen zusammengefasst, die

- Anwendungsprotokolle (Application Protocols) und
- die Übertragungsprotokolle (Transport Protocols)

Zu den Anwendungsprotokollen gehören die Anwendungsschicht, die Darstellungsschicht und die Kommunikationssteuerungsschicht.

Die Anwendungsschicht besteht aus Protokollen, die die Kommunikation zwischen Anwendungsprozessen in Verbindung mit den Verwaltungsfunktionen, die die Anwendungsprozesse unterstützen, durchzuführen. Zu diesem Zweck werden anwendungsorientierte Grunddienste z.B. für File Transfer, Electronic Mail, Remote Job

Entry, Virtual Terminal und damit zusammenhängende Darstellungskonventionen z.B. für die Festlegung der Struktur von auszutauschenden Dokumenten sowie Verwaltungsfunktionen definiert. Diese Protokolle gliedern sich in drei Grundtypen:

- Struktur-Verwaltungsprotokolle für die Kommunikation der Verwaltungsinstanzen globaler Art (gesamte Systemsicht)
- System-Protokolle für die Kommunikation der Verwaltungsinstanzen lokaler Art (Knotensicht: Welche Mittel stehen hier zur Verfügung)
- In-Line-Verwaltungsprotokolle für die Verwaltung der Betriebsmittel im Umfeld der Kommunikationsverbindung: Zugangskontrolle, Deadlock-Vermeidung.

Im Klartext: Viele Dienste der Anwendungsschicht stellen eine Erweiterung des Betriebssystems in Richtung der offenen Kommunikation dar. Dies bewirkt u.a. auch, daß viele Aufgaben des Netzwerkmanagements nicht von denen der Schicht 7 zu trennen sind.

Die *Darstellungsschicht* hat die Aufgabe, die Daten, die ausgetauscht werden, entsprechend den Vereinbarungen zu codieren bzw. zu interpretieren. Dies bedeutet natürlich auch die Vereinbarung einer gemeinsamen Semantik. Die dazu einzig zur Verfügung stehende offizielle Norm lautet *ASN.1* (Abstract Syntax Notation Number One) für den Wertebereich und die Bedeutung der Daten, die konkrete Codierung von Daten in Bytes wird dabei durch *BER* (Basic Encoding Rules) festgelegt. Häufig werden jedoch auch herstellerspezifische Definitionen in Protokollen eingesetzt.

Die *Kommunikationssteuerungsschicht* hat die Aufgabe, die geregelte Kommunikation zwischen zwei Teilnehmern sicherzustellen. Dies umfasst z.B. die online-Synchronisation oder das Einfügen von Synchronisationspunkten, die die jederzeitige Wiederaufnahme einer Kommunikation gestatten.

Die *Übertragungsprotokolle* umfassen die *Transportschicht*, die *Vermittlungsschicht*, die *Datensicherungsschicht* und die *Bitübertragungsschicht*.

Die *Transportschicht* hat die Aufgabe, den höheren Schichten und damit der Applikation eine einheitliche Zugangsschnittstelle zum Netz zur Verfügung zu stellen. Sie baut End-To-End-Verbindungen auf bzw. ab, kann dabei zwischen verschiedenen Übertragungswegen auswählen, kann auch mehrere virtuelle Verbindungen gleichzeitig unterhalten, eine physikalische Verbindung also zwischen mehreren Anwendungen multiplexen.

Die *Vermittlungsschicht* bestimmt einen Weg durch ein grundsätzlich ausgewähltes Netz von Vermittlungsstationen zu dem gewünschten Ziel. Die Zerlegung von zu großen Datenpaketen zur Übertragung fällt ggf. ebenfalls in diesen Aufgabenbereich. Hierbei wird eine Numerierung der erhaltenen Pakete eingefügt, um die richtige Reihenfolge wiederherstellen zu können.

Die *Datensicherungsschicht* garantiert den zuverlässigen Transport von Information zwischen zwei benachbarten (d.h. durch die gleiche Leitung miteinander verbundenen) Rechnern oder Vermittlungsstationen. Dies geschieht per Kontrollinformationen in einem zusätzlichen Rahmen um den bisherigen Datenblock. Zur Datensicherungsschicht gehört auch die Zugangskontrolle zum Medium (Media Access Control), was insbesondere bei LANs eine erhebliche Rolle spielt und aus diesem Grund auch häufig als Layer 2a bezeichnet wird.

Die Bitübertragungsschicht schließlich ist für die Übertragung der einzelnen Bits zwischen zwei Rechnern/Vermittlungsstationen am gleichen Link verantwortlich. Hierzu bedarf es weitreichender Festlegungen z.B. des Mediums, der Darstellung, der Stecker, der elektrischen Größen usw.

11.1.2 Netzwerkmanagement

Netzwerke sind als komplexe, verteilte Systeme aufzufassen, die aufgrund von fehlerhafter Soft- oder Hardware sowie durch äußere Einflüße (Beschädigung, EMV) ständig zu überwachen und zu steuern sind. Diese Überwachung und Steuerung sowie andere Aspekte wie Rekonfiguration, Wartung und Neuinstallation werden unter dem Stichwort *Netzwerkmanagement* zusammengefasst.

Das Netzwerkmanagement wird zumeist zweistufig aufgebaut: Ein *Knotenmanagement* ist eng mit der Anwendungsschicht 7 verbunden; hier werden Informationen aus den darunterliegenden Schichten 1 bis 6 zusammengetragen und zu einer lokalen Ressourcenverwaltung hinzugefügt.

Das eigentliche Netzwerkmanagement nutzt diese lokal vorhandenen Daten, um in Zusammenarbeit mit Datenbankinformationen (MIB, Management Information Base) das gesamte Netz zu steuern. Weiterhin können Laufzeitmessungen durch spezielle Datenpakete ausgelöst werden. Als Resultat werden dann Informationen an die einzelnen Knoten z.B. zum Routing gesendet. Das zugehörige Kommunikationsprotokoll wird dem normalen Netzprotokoll hinzugefügt.

Die Nomenklatur von Rechnernetzen erweitert die übliche Begriffswelt der Informatik um eine Reihe von Begriffen und Definitionen, deren Verständnis und Wissen für das weitere Vorgehen zwingend notwendig ist. Aus diesem Grund werden in diesem Kapitel alle wichtigen Grundbegriffe des Gebiets um Netze und Rechnerkommunikation zusammengefaßt.

11.2 Klassifizierung von Netzwerken

11.2.1 Entfernung

Rechnernetze alleine stehen für eine Kommunikationsmöglichkeit im Rahmen der technischen Gegebenheiten. Weit mehr als diese pure Übertragung von Daten ohne Bewegung von materiellen Datenträgern ermöglichen sie die Zusammenfassung

der ‘Rechenkraft’ der Teilnehmer am Netz, aus globaler Sicht zu einem einzigen rechnenden Gebilde.

Im Unterschied zu einem einzigen Rechner, der innerhalb eines Gehäuses alle zentralen und peripheren Einheiten vereint, müssen die Teilnehmer an einem Netz nicht an einem Ort vereint sein; ihre räumliche Trennung ist nicht nur möglich, sondern der Regelfall bei einem Netz. Rechtliche und physikalische Gründe ergeben nun für ein verteiltes Netz von Rechnern sinnvolle Einteilungen, die mit der Entfernung in ursächlichem Zusammenhang stehen.

Die Menge aller Abstände d_{ij} der Rechner i und j wird zu einer unscharf definierten Metrik für Netze zusammengefasst, da die Handhabung aller Abstände eine unnötige Komplexität bedeuten würde. Diese Menge ließe die Bestimmung eines maximalen Abstands zu, der bei manchen Netzdefinitionen auch vorgeschrieben ist. In der Regel werden sogenannte Ausreißer zur Bestimmung fortgelassen, falls nicht technische Gründe dagegen sprechen: Man spricht dann von einer typischen Entfernung.

Diese typischen oder maximalen Entfernung werden in drei Klassen eingeteilt, die auch rechtlichen Aspekten Rechnung tragen (beispielweise Trägerschaft der Kommunikationsübertragung: Privat oder öffentlich-rechtlich?). Diese Klassen lauten in der üblichen Nomenklatur:

- Entfernungen ≤ 1000 m: Local Area Networks, LAN
- Entfernungen ≤ 10000 m: Metropolitan Area Networks, MAN
- Entfernungen > 10000 m: Wide Area Networks, WAN Neben diesen Bezeichnungen haben sich in der Zwischenzeit weitere eingebürgert:
- Entfernungen ≤ 1 m: Personal Area Network (PAN)
- Entfernungen ≤ 10 m: Controller Area Network (CAN)

Letztere Bezeichnung CAN sollte nicht mit dem CAN-Standard verwechselt werden.

11.2.2 Anwendungen

Neben der Unterscheidung nach Entfernungen, das ein recht formales Kriterium darstellt, kann ebenfalls eine Differenzierung nach Anwendungen durchgeführt werden. Dies führt in sehr viel stärkerem Maß zum Einsatzzweck von Rechnerkommunikationen, wobei andererseits zu bemerken ist, dass eine eindeutige Zuordnung eines Netzes zu einem Anwendungsgebiet ebenso unmöglich wie sinnlos ist.

Die Angabe von Anwendungen für Netze ist vielmehr als Merkmalsangabe zu verstehen; diese Merkmale können bei Planungen den Netzen zugrundegelegt werden, ebenso können die Hauptmerkmale eines Netzbetriebs zeitlich variieren.

11.2.2.1 Lastverbund

Ziel: Die gleichmäßige Auslastung verschiedener Ressourcen.

Methode: Aufteilung stoßweise anfallender Lasten auf verschiedene Rechner.

Jeder einzelne Rechner eines Netzes sollte möglichst lange arbeiten, um in einer großen Anzahl von Rechnern die Ressourcen optimal zu nutzen. Diese Forderung resultiert daraus, dass ein Rechner im Zustand 'IDLE' (nicht ausgefallen, aber zurzeit ohne Aufgabe) eine Investition ohne Nutzung darstellt, und gilt für alle technischen Anlagen. Resultierend daraus müssen Aufgaben auf andere Rechner auslagerbar sein, falls im Netz Berechnungskapazitäten zur Verfügung stehen und auf einzelnen Rechnern stoßweise hohe Kapazitätsansprüche auftreten. Die Forderung lautet also, möglichst homogene Netze für einen Lastverbund zur Verfügung zu stellen, damit mehr als ein Rechner ein Programm ausführen kann.

Ist diese Forderung zumindestens teilweise erfüllt (teil-homogene Netze), kann ein *Betriebsmittel-Pool* eingerichtet werden, über den ein *Lastmanager* verfügt und so die einzelnen Tasks auf die Rechner im Netz aufteilt. Dadurch werden Leerlaufzeiten bei Rechnern (und Nutzern) verringert sowie ggf. andere Ressourcen wie Speicher und Drucker besser ausgelastet.

11.2.2.2 Leistungsverbund

Ziel: Verringerte Antwortzeiten.

Methode: Aufteilung einer Aufgabe in Teilaufgaben.

Falls es gelingt, eine (komplexe) Aufgabe in unabhängige Teilaufgaben zu zerlegen, deren Ergebniszusammenfügung quasi am Schluss ohne größeren Aufwand geschehen kann, können diese Teilaufgaben auf verschiedene Rechner im Netz verteilt werden und führen dann zu einer stark verringerten Antwortzeit des gesamten Systems auf diese Aufgabe.

Dies ist das Ziel des Leistungsverbunds von Rechnern via Netzwerk. Ist die Teilbarkeitsvorraussetzung gegeben, dann kann eine automatische Prozesssteuerung einsetzen, die die einzelnen Tasks verteilt und via Protokoll anschließend wieder zusammenführt. Wichtiger ist in diesem Fall die automatische Trennung von Teilaufgaben, was ein wichtiges Forschungsgebiet darstellt: Welche Granularität der Teiltasks ist erhaltbar (möglichst unter Beibehaltung der 'klassischen' Programmiersprachen, um lediglich Neucompilierung durchführen zu müssen), welche ist optimal zur Durchführung (unter Berücksichtigung des 'Kommunikationsoverheads')?

Rechner, die unter diesen Bedingungen gemeinsam an einem Ziel arbeiten, werden als *Rechnerverbund* oder *verteiltes System* bezeichnet. Die Konzipierung und insbesondere die Kommunikation/Synchronisation verteilter Systeme im eingebetteten Bereich unterliegen meist besonderen Bedingungen.

11.2.2.3 Kommunikationsverbund

Ziel: Übertragung von Daten, insbesondere Nachrichten, an verschiedene, räumlich getrennte Stellen.

Methode: Erstellung eines Briefdienstes (Electronic Mail).

Die Benutzung eines Netzes als Nachrichtendienst stellt eine der ersten und wichtigsten Anwendungen dar, obwohl Nachrichtendienste kaum als Begründung für die Einführung eines LAN ausreichen dürften. Für solche Briefdienste werden entsprechende Services eingerichtet, die den Anwendern das Versenden einer Nachricht ermöglichen, ohne daß dieser den Aufenthaltsort des Adressaten wissen muss. Netzübergänge z.B. zwischen LAN und WAN werden integriert. Voraussetzung ist lediglich, daß die Teilnehmer einen Rechner als 'Personal Computer' z.B. mit Zugangskennung betreiben.

Die technischen Bedingungen, die an ein solches System gestellt werden müssen, lauten: Editor, Benutzeragent(-Programm), der ein Adressregister zur Verfügung stellt, Nachrichten (asynchron) empfängt, speichert und anzeigt, eventuelle Verschlüsselungsverfahren bei vertraulichen Nachrichten sowie ein Netzanschluss. Für Nachrichtendienste mit Netzübergängen müssen ggf. Protokollwandler vorhanden sein.

11.2.2.4 Datenverbund

Ziel: Bessere Auslastung von Harddisks, erhöhte Sicherheit und Verfügbarkeit.

Methode: Speicherung von Daten an verschiedenen Stellen.

Datenverbunde treten häufig in Zusammenhang mit verteilten Datenbanken auf, bei denen zusammenhängende Datenbestände durchaus auf verschiedenen, räumlich dislozierten Rechnern gehalten werden, auf. Die Haltung eines Teilbestands auf einem Rechner ist genau dann sinnvoll, wenn die Hauptnutzung auf diesem stattfindet.

Unter diesen Umständen besteht die wesentliche Aufgabe des Datenbanksystems in der Konsistenzhaltung der Daten sowie dem Management des lokalen und nichtlokalen Zugriffs.

Eine weitere Aufgabe des Datenverbunds außerhalb der Welt der verteilten Datenbanken ist in dem RAID-Standard zu sehen (RAID: Redundant Arrays of Inexpensive Disks). Bei diesem Standard ist vorgesehen, die Datenspeicherung im Allgemeinen in redundanter Weise auf mehreren, auch verteilten Speichermedien, die ihrerseits nicht besonders hohen Ausfallsicherheiten (und Kosten!) unterliegen, zu managen, um durch die Redundanz eines hohe Sicherheit vor Datenverlust zu bekommen.

11.2.2.5 Wartungsverbund

Ziel: Schnellere und billigere Wartung verschiedener Rechner.

Methode: Zentrale Störungserkennung und -behebung.

Im Wartungsverbund, für den ggf. spezielle Formen von Netzen ausschließlich eingesetzt werden (Wählnetz mit Telefon und Modem), werden Ferndiagnosen, Updates per Netzübertragung ebenso durchgeführt wie die Einsatzsteuerung und -unterstützung von Wartungspersonal.

Im ersteren Fall wird ein Serviceprozessor zusätzlich zum Einsatzfall eines Rechners (oder Geräts) eingesetzt, der über ein Netzwerk mit einer Servicezentrale verbunden ist und ggf. aktiviert wird. Der Einsatz dieser Systeme spart insbesondere Personalkosten, da beispielsweise bei Updates kein persönliches Erscheinen notwendig ist.

In anderweitig genutzten Netzen kann über Netzwerkmanagementprozesse der Zustand der Rechner ständig aufgezeichnet und in einer genormten Struktur (MIB, Management Information Base) gespeichert werden. Diese Informationen können dann im Servicefall wesentliche Hilfe darstellen, insbesondere kann bei Netzübertragung das Servicepersonal entsprechend an den Einsatzort gelenkt werden.

11.2.2.6 Funktionsverbund

Ziel: Bereitstellung spezieller Funktionen an verschiedenen Stellen.

Methode: Verteilung spezieller Aufgaben auf spezielle Rechner (Array-Prozessoren, Vektorrechner, Supercomputer, Applikations-spezifische Rechner etc.).

Für bestimmte Rechenaufgaben werden auch heutzutage Spezialrechner benutzt, da sie diese Aufgaben in vergleichsweise geringer Zeit lösen können. Zu diesem Zweck müssen die Aufgaben so codiert werden, dass der Spezialrechner sie lösen kann (Crosscompiler).

Andere Aufgaben können beispielsweise in der Nutzung spezieller Hardware (Farbdrucker, große Plotter etc.) liegen, die dann einen Resourcenmanager am Netz besitzen.

11.2.2.7 Kapazitätsverbund

Ziel: Ausnutzung sämtlicher zur Verfügung stehender Rechenkapazität.

Methode: Versendung von Aufgaben an möglichst viele verschiedene Rechner.

Der Kapazitätsverbund steht in engem Zusammenhang mit dem Lastverbund, den verteilten Systemen und dem Datenverbund. Er stellt letztendlich einen Oberbegriff aus Betreiber- oder Benutzersicht dar, denn es steht weniger eine Aufgabe im Vordergrund, mehr die komplette Nutzung aller Resourcen am Netz. Dies kann

sich sowohl auf Software- und Hardwareresourcen als auch Rechenkapazitäten beziehen.

11.2.3 Übertragungstechnik

Die Kommunikation zwischen Komponenten benötigt Verbindungen, welche in geeigneter Weise Signale übertragen können. Geeignet heißt in diesem Fall, dass ein Interface existiert, um die übertragenen Signale den Komponenten in interpretierbarer Form zur Verfügung steht.

Zu diesem Zweck wurden verschiedene Technologien zur Übertragung entwickelt, die sich sowohl auf physikalische Technik als auch Protokolle beziehen. Die ersten Techniken bestanden in vorhandenen elektrischen Leitungen (Telefonkabel, twisted pair, Koaxialkabel etc.) zwischen jeweils zwei Rechnern (Point-To-Point-Verbindungen). Diese Verbindungen benötigten kaum Protokolle, waren daher sehr einfach handzuhaben und resultierten in den Modem-Übertragungen (Modulator-Demodulator, RS-232 als Verbindung zwischen Modem und Rechner).

Der Wunsch nach Multipoint-Verbindungen ließ neue Verfahren aufkommen: Ethernet für LANs verbreitete sich ab den 70er Jahren, Satelliten sorgten für die interkontinentale Kommunikation mit großen Geschwindigkeiten, in den 80er Jahren starteten dann die Glasfasernetze. Die heutige physikalische Grenze für die Bruttokommunikation liegt bei etwa 100 Giga-Bit pro Sekunde; dies bedeutet 10 ps pro Bit!

11.2.3.1 Vermittlungsnetze, Rundsendesysteme

Ein *Vermittlungsnetz* baut für jede Kommunikationssitzung eine eigene Verbindung zwischen zwei Teilnehmern auf, wobei nun die Verbindung als Point-To-Point-Verbindung angesehen wird. Für diesen Zweck können andere Netze, z.B. für Telefon genutzt werden; die Störung durch andere Teilnehmer ist, da die beiden exklusiv sind, praktisch sehr gering.

Bei Rundsendesystemen steht meist ein leistungsfähiger Kanal für viele Teilnehmer zur Verfügung. Der Zugang erfolgt durch ständiges Mithören am Kanal und Benutzung gleichzeitig (frequency multiplex, FDMA, Frequency Divison Multiple Access) oder in Zeitscheiben (time multiplex, TDMA, Time Division Multiple Access). Dieses Verfahren wird Multi-Point-Kommunikation, auch Multi-Point-Access genannt. Das genaue Zugriffsverfahren entscheidet dabei über Echtzeitfähigkeit und Kanalausnutzung des Übertragungsverfahrens.

Anmerkung: In eingebetteten Systemen, die naturgemäß eine eher geringe räumliche Ausdehnung bez. Der Kommunikation besitzen, werden gerne Rundsendesysteme genutzt. Auf diese Weise kann die Vermittlung entfallen, allerdings muss dann – je nach Anforderung – die Zuteilung entsprechend geregelt werden.

11.2.3.2 Leitungsvermittlung, Speichervermittlung

Bei der *Leitungsvermittlung* (circuit switching) wird zwischen den Endteilnehmern eine exklusive, reale oder virtuelle Leitung geschaltet, die ausschließlich sie benutzen dürfen. Auf diesem Prinzip beruht beispielsweise das Telefonnetz, die Teilnehmer sind in der Regel gegenüber Störungen geschützt. Das Netz ist dabei nicht in der Lage, Zwischenspeicherungen oder Taktumsetzungen vorzunehmen. Aus diesem Grund müssen die Teilnehmer vorher eine Synchronisation ihrer Übertragungsraten (neben den weiteren Werten zur Übertragung wie das Zeichenformat) vornehmen. Wichtigste Vertreter dieser Vermittlungsform sind das Telefonnetz, ISDN und Datex-L. Für Rechnerübertragung wird es jedoch praktisch nicht mehr verwendet, außer zum Zugriff zwischen letzter Vermittlungsstelle und dem Rechner („letzter Kilometer“).

Die *Speichervermittlung* schaltet nur zwischen zwei Relais-Stationen eine echte physikalische Leitung, in der Relais-Station erfolgt eine gewisse Zwischenspeicherung. Die zu übertragenden Daten werden in Pakete fester Länge aufgeteilt und als solche verschickt, und zwar von Relaisstation zu Relaisstation. Der Weg für verschiedene Pakete kann dabei unterschiedlich lang und schnell sein, zudem können durch Überlast Pakete verloren gehen, so dass eine Fluss- und Fehlerkontrolle sowie Behebung unerläßlich ist. Vertreter dieser Gattung: WAN-Netze.

11.2.3.3 Verbindungslose und verbindungsorientierte Kommunikation

Muss sich ein Teilnehmer vor der Übertragung der Informationen beim Empfänger anmelden, so spricht man von einer *verbindungsorientierten*, anderenfalls von einer *verbindungslosen Kommunikation*. Diese Konzepte sind auf logische Basis angesiedelt, verbindungslose wie verbindungsorientierte Kommunikationen lassen sich auf Leitungs- oder Speichervermittlungen anwenden.

Die zu senden Datenpakete einer verbindungslosen Kommunikation werden *Datagramme* genannt.

Anmerkung: In eingebetteten Systemen wird gerne verbindungslose Kommunikation genutzt, da diese keinen Overhead zum Verbindungsaufbau benötigt. Allerdings muss Vorsorge dafür getroffen werden, dass der Paketverlust entweder akzeptabel ist oder erkannt werden kann.

11.2.3.4 Übertragungsmedien

Übertragungsmedien und -technik hängen stark voneinander ab: Die benutzten Signale sind meist elektrischer oder optischer Natur, die frequenz-, amplituden- oder phasenmoduliert sind (Optik: nur Amplituden). In seltenen Ausnahmefällen wird beispielsweise Ultraschall genutzt. Die Änderungsgeschwindigkeit dieser Modulation ist entscheidend für die Übertragungsrate, weniger die Ausbreitungsgeschwindigkeit, die für elektrische/optische Verfahren im Wesentlichen gleich der Lichtgeschwindigkeit ist (bis auf einen Medieneffekt).

Die Medien im Einzelnen:

- ***Verdrillte Kupferkabel*** (twisted pair) gibt es in geschirmter (STP, shielded TP) wie ungeschirmter Form (UTP). Diese Leiter können trotz ihrer Einfachheit bis zu 100 Mbit/s übertragen (bei 100 m Entfernung). Die Übertragungsreichweite und -geschwindigkeit ist im Wesentlichen durch Störungseinflüsse (elektromagnetische Induktion) und die Abflachung der elektrischen Impulse durch endliche Kapazitäten und Induktivitäten begrenzt.
- ***Koaxialkabel*** bestehen aus einem zentralen Leiter, der von einem peripheren Leiter abgeschirmt wird. Eine elektromagnetische Induktion wirkt auf beide Leiter gemeinsam und hebt sich gegenseitig auf, während die elektromagnetische Abstrahlung durch die Bauform sehr gering bleibt. Aus letzterem folgt eine geringe Dämpfung, so dass diese Kabel für größere Entfernungen bei hohen bis höchsten Frequenzen geeignet sind. Schwierig bleibt einzig die Einkopplung.
- ***Lichtwellenleiter*** dienen der Übertragung optischer Signale und sind für höchste Frequenzen wie Reichweiten geeignet. Die Dämpfung beträgt ca 1,5 dB/km, so dass kaum optische Verstärker eingesetzt werden müssen. Die Einkopplung von Signalen, die häufig aus modulierten Laserdioden stammen, erfolgt so, dass der Lichtstrahl entweder durch Totalreflexion im LWL gehalten wird (Stufenindexfaser) oder durch ständige Brechung zum Leitungsinneren (Gradientenfaser). Die Begrenzung der Übertragungsrate liegt in dem Verwaschen der Lichtimpulse, wobei das Licht amplitudenmoduliert wird: Strahlen, die vielfach reflektiert werden, haben einen längeren Weg! (→ 11.3.1)
- ***Richtfunktverbindungen*** benötigen quasi-optische Sicht und werden zur Überbrückung unwegsamen Geländes genutzt.
- ***Satelliten*** werden zur Überbrückung von Kontinenten oder als Rundsendesystem eingesetzt. Die Laufzeit zu geostationären Satelliten und zurück beträgt 200 ms, was zu erheblichen Rückmeldeverzögerungen führen kann.

11.2.3.5 Basisband- und Breitbandnetze

Eine *Basisbandübertragung* nutzt das gesamte Übertragungsspektrum (als Frequenzspektrum gesehen) nur zur Übertragung einer einzigen Kommunikation; die Kommunikation für andere Verbindungen erfolgt zu einem anderen Zeitpunkt, so dass für Basisbandnetze, die mehr als eine Verbindung aufnehmen, der Zeitmultiplex zur wesentlichen Modulationsart wird.

Bei den Zeitmultiplexverfahren werden noch synchrone Verfahren (gleiche Zeitscheibengröße für alle) sowie asynchrone unterschieden: Letztere erlauben – in gewissen Grenzen – die Variation der Zeitscheiben für optimalere Übertragung.

Breitbandnetze teilen das Frequenzspektrum eines Leiters in einzelne Kanäle auf, die sich durch verschiedene Frequenzlagen unterscheiden. Diese Technik ist relativ teuer, da höchste Anforderungen an die Interfaces gestellt werden.

11.2.4 Übertragungswege

Die Wege zur Übertragung, auch bei nichtleitungsgebundener Form, sind nahezu immer gesetzlich sanktioniert: Es gibt private oder öffentliche Anbieter, teilweise Monopole, die als Netzbetreiber den Auftrag zur Verfügbarmachung von Netzleitungen haben. International werden diese Netze durch bilaterale oder multilaterale Abkommen sowie die internationalen Standards geregelt.

11.2.4.1 Öffentliche Netze

Ein Netz wird als öffentlich bezeichnet, wenn im Prinzip jeder zu einer angemessenen Gebühr Nutzer dieses werden kann. Öffentliche Netze werden in der Regel von der Regierung kontrolliert und haben die Verpflichtung zum Angebot an jedermann.

Die Konsequenz daraus ist, dass ein öffentlicher Netzanbieter nicht nur die Pflicht zum Angebot hat, sondern auch das Recht zum alleinigen Angebot; ein LAN beispielsweise ist auch durch die Beschränkung auf privates Gelände gekennzeichnet. Wird nun für ein solches Netz öffentlicher Grund überquert, muss dies (mit Ausnahmen) durch den öffentlichen Anbieter geschehen.

11.2.4.2 Private Netze

Bei Nutzung ausschließlich privater Leitungen, die auch von öffentlichen Anbietern zur alleinigen Nutzung gemietet sein können, spricht man von privaten Netzen. Beispiele: Telefonnebenstellenanlagen, LAN, Buchungsnetze.

11.2.4.3 Elektrische Netze

Die elektrischen Netze sind quasi die natürlichste Form der Vernetzung: Die Darstellung der Informationen in Rechnern erfolgt auf elektrischem Wege, so dass das Interface zur Übertragung entsprechend wenig Aufwand erfordert. Zudem stellt die elektrische Übertragung einen sehr guten Kompromiss zwischen nahezu höchster Bitrate, Verlust, Anschlussfähigkeit und Kosten dar.

11.2.4.4 Lichtwellenleiter

Für Lichtwellenleiter gibt es prinzipiell kaum noch Probleme mit Bandbreiten, da die Übertragung in LWLs eine sehr große Bandbreite zur Verfügung stellen kann. Die Problematik liegt jedoch in den Interfaceeinrichtungen, die die Übertragungen auch gestalten müssen.

11.2.4.5 Satellitennetze

Satelliten werden im WAN-Bereich zur kontinentalen und interkontinentalen Kommunikation benutzt. Die langen Signallaufzeiten (Abstand Erde-Satellit mindestens 36.000 km) sind diese Verbindungen weniger zur interaktiven Kommunikation geeignet; im Bereich von WAN-Übertragungen müssen ggf. spezielle Satelliten-

protokolle zum Einsatz kommen, um bei zu kurzen Rückmeldezeiten (-fristen laut Protokoll) nicht einen Abbruch der Verbindung zu signalisieren, obwohl nur eine lange Signallaufzeit vorliegt.

11.2.4.6 Funknetze

Werden elektromagnetische Wellen als Träger für Daten benutzt, spricht man von ***Funknetzen***. Diese Form wurde zunächst jahrelang zurückgedrängt (Unzuverlässigkeit), erlebte dann allerdings in Form der ***Wireless-LANs*** (WLAN) eine Rückkehr. Die Einsatzgebiete solcher Funknetze sind begrenzt, so z.B. bei temporären Netzen (zwecks Schulung) oder in unzugänglichen Gebieten.

11.2.5 Topologie

Eine weitere Unterscheidungsmöglichkeit für Netze besteht in der *Topologie*. Diese Topologie ist zwar für den späteren Nutzer quasi transparent, da er sich in seinen Netzaktivitäten nicht danach richten muss, jedoch sind Performance, Zuverlässigkeit und weitere Werte stark von der Netztopologie abhängig.

Bild 11.2 gibt einen kurzen Überblick über die wichtigsten Topologien; diese treten in der Realität natürlich auch in gemischter Form auf.

11.2.5.1 Busnetze

Sind alle Rechner am Netz an einem Strang angeschlossen, dann wird dies als Busnetz bezeichnet. Der Strang ist dabei ein einziger elektrischer Leiter (oder ein paralleles Leiterwerk mit logisch voneinander getrennten Funktionen, z.B. zur parallelen Datenübertragung), so daß aus physikalischer Sicht der Bus einem Rundsendesystem entspricht: Jeder Teilnehmer hört ständig am Bus, was passiert.

Aus logischer Sicht kann diesem Bus jedoch eine andere Struktur aufgeprägt werden, die dann z.B. für geordnete Kommunikation sorgt. Ethernet läßt alle mithören, eine sendewillige Station ist für ihren geordneten Sendezugriff selbst verantwortlich. Im Tokenbus kreist eine Sendeberechtigung (Token), während in älteren Systemen (Aloha) ein Master eine Zeitscheibeneinteilung vornimmt: Auf diese Weise wird aus dem System gleichberechtigter Teilnehmer ständig einer ausgezeichnet.

Eine Erweiterung des einfachen Buskonzepts wurde mit FDDI eingeführt: Der *Doppelbus DBDQ (Double Bus Double Queue)*. In diesem Verfahren kreisen zwei Token in zwei Bussystemen, so dass eine bessere Auslastung und vor allem – durch selbstständige Fehlerbehebung – eine wesentlich verbesserte Ausfallsicherheit die Folgen sind. FDDI zählt allerdings zu den Ringen, die durch die Schließung der offenen Busenden erhalten werden:

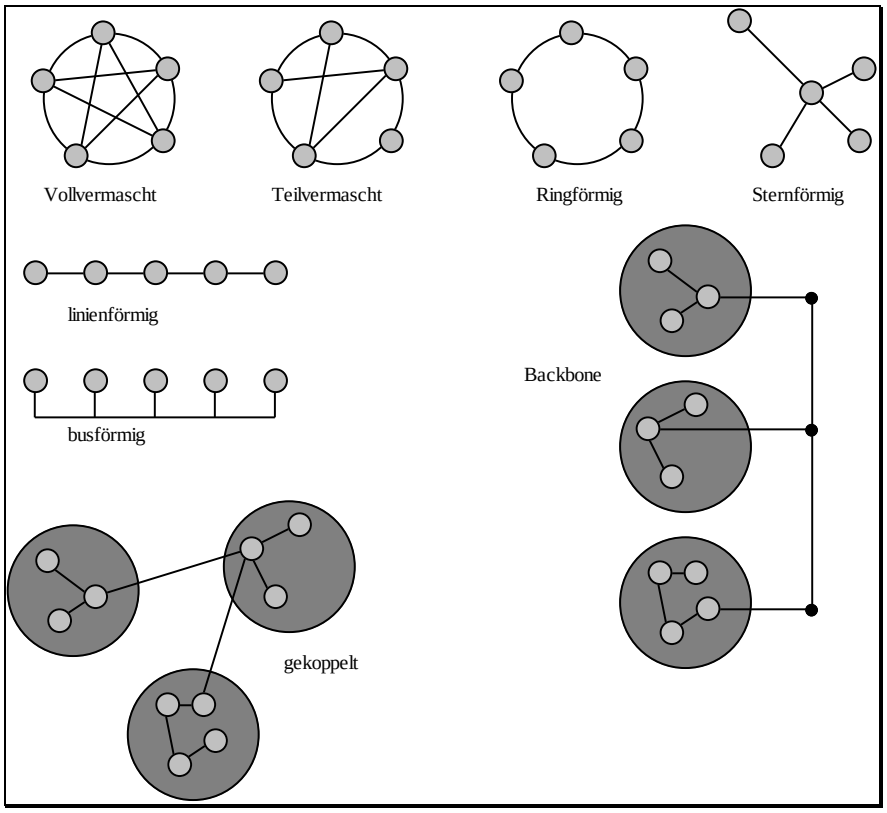


Bild 11.2 Gebräuchliche Netztopologien

11.2.5.2 Ringnetze

Der bereits erwähnte Zusammenschluss der Busenden führt zu einem Ring, bei dem nunmehr kein Rundesendesystem mehr vorliegt, sondern nur in eine Richtung gesendet, aus der anderen empfangen wird. Technisch gesehen entspricht der Ring einer Reihe von Punkt-zu-Punkt-Verbindungen. Die Sendeberechtigung wird zumeist auch in einem Token- oder Doppel-Tokenverfahren weitergegeben.

11.2.5.3 Sternnetze

In Sternnetzen wird die Äquivalenz aller Busteilnehmer aufgegeben: Ein Rechner wird als zentraler Rechner im Stern ausgezeichnet. Dies geschieht im Übrigen häufig auch bei den anderen Netztopologien, nur zumeist auf höheren Ebenen: Die Einrichtung eines zentralen Servers betrifft nicht die eigentliche Netzübertragung.

Der Nachteil eines Sternnetzes ist leicht detektiert: Der zentrale Rechner muss hochzuverlässig sein, ansonsten bricht das gesamte Netz sofort zusammen. Andererseits sind Ausfälle an den peripheren Rechnern sofort zu detektieren, da der zentrale Rechner ein zentrales Logbuch führen kann. Beispiele für sternförmige Netze sind das Telefonnetz und das Ethernet Typ II.

Die Verkabelung von an sich bus- oder ringförmigen Netzen durch sternförmige Kabel mit speziellen Netzzugängen hat sich aus Kostengründen ebenfalls bewährt. Solche zentralen Netzverteiler, die nicht mit einem zentralen Rechner verwechselt werden sollte, werden ***Hubs*** genannt.

11.2.5.4 Vollvermaschte Netze

Der Sinn eines *vollvermaschten Netzes*, bei dem jeder Rechner mit jedem anderen durch eine direkte Verbindung verbunden ist, liegt in der möglichst schnellen Kommunikation. Dies macht bei Rechnern im Leistungs- Last- oder Kapazitätsverbund besonders Sinn, falls sehr große Kommunikationsmengen zu erwarten sind.

Die einzelnen Verbindungen, deren Anzahl mit

$$0{,}5 * n * (n-1)$$

eingeht, werden als Punkt-zu-Punkt-Verbindungen geschaltet. Mit anderen Worten: Der Aufwand im Netz und in den Rechnern steigt quadratisch an.

11.2.5.5 Gekoppelte Netze

Die Kopplung (beliebiger) Rechnernetze miteinander wird *gekoppeltes Netz* (internet) genannt. Die Kopplung selbst kann dabei sehr aufwendig sein, so dass im nächsten Abschnitt kurz darauf eingegangen wird.

11.2.5.6 Backbone-Netze

Eine spezielle Form der Kopplung liegt vor, wenn durch das Koppel-Netz lediglich die Zusammenfügung der bisherigen Teilnetze zu einem größeren, das aber logisch einem einzigen entspricht, erreicht werden soll. Dies geschieht meist durch schnelle Übertragungen und wird *Backbone-Netz* genannt.

11.2.6 Kopplung von Rechnernetzen

Bei der Kopplung von Rechnernetzen miteinander werden hierfür zumeist wiederum Rechner eingesetzt, um die umfangreichen Transformationen entsprechend zu bearbeiten. Ein solches Gerät wird gemäß ISO allgemein *Relais* (relay) genannt. Für diese Relais wird je nach Aufgabenbereich eine weitere Klassifizierung betrieben, die sich an den nicht behandelten Protokollschichten orientiert.

Allgemein gehören zu den Aufgaben eines Relais die Bearbeitung von:

- Adressierung und Namensvereinbarungen

- Wegewahl (routing) und Flusskontrolle
- Informationsgrößen, wie Paketzähler oder Fenstergrößen
- Umsetzung von Paketgrößen
- Umsetzung zwischen verbindungsorientierten und verbindungslosen Diensten
- Fehlerkontrolle
- Zeitüberwachung
- Weiterleitung von Unterbrechungen
- Senden von Statusberichten und Abrechnungen
- Kontrolle des Nutzerzugriffs
- Verbindungsauf- und -abbau

11.2.6.1 Repeater

Repeater können als Verstärker angesehen werden: Die bearbeitete Schicht ist die Schicht 1, die Bitsignale eines Netzes werden in Bitsignale des anderen umgesetzt. Dies kann sich auch auf die Kopplung gleichartiger Netze beziehen, die aus technischen Gründen voneinander getrennt sein müssen (etwa Reichweite).

Alle weiteren Schichten müssen zwischen den Netzen vollkommen identisch sein, eine Umsetzung erfolgt nicht.

11.2.6.2 Bridge

Bridges müssen die Blöcke (frames, Rahmen) des einen Netzes in die des zweiten Netzes und umgekehrt umsetzen können. Die Umsetzung bezieht sich also auf Schicht 2, die Bearbeitung der Adressierung sowie der Blockfehlersumme ist notwendig.

11.2.6.3 Router

Router müssen die Pakete eines Netzes in solche eine anderen Netzes umsetzen. Neben der Anpassung der Adressen betrifft dies auch das Paketformat, das in unterschiedlichen Netzen auch unterschiedlich sein kann. Fehlermeldungen seitens der Router können die Adressierung, sollten jedoch nicht die Paketformate betreffen; letzteres ist Aufgabe der Endteilnehmer.

11.2.6.4 Gateway

Gateways verbinden Netze, deren Protokolle sich bereits in den Anwendungsschichten unterscheiden. Hierzu können diverse Postdienste (Electronic Mail) oder Dateiübertragungen gezählt werden, ebenso kann die Darstellung der Zeichen variieren (beispielsweise ASCII kontra EBCDIC). Der Aufwand in Gateways ist also entsprechend hoch.

11.2.7 Bemerkungen zur Informationstheorie

Die Informationstheorie, durch Shannon im Jahre 1950 als *Kommunikationstheorie* begründet, betrachtet den Informationszustand der miteinander verbundenen Rechner als unterschiedlich: Der sendende Rechner weiß mehr als der empfangene, wobei durch die Übermittlung der Nachricht dies zumindestens gemindert wird.

Dieser unterschiedliche Kenntnisstand wird *Entropie* genannt. Dieser Begriff stammt ursprünglich aus der Physik (Thermodynamik) und beschreibt dort den Ordnungszustand eines statistischen Systems, wobei eine niedrigere Entropie einem geordneteren System entspricht. (Anmerkung: Ein in sich abgeschlossenes physikalisches System vermag seine Entropie nur zu steigern, es wird 'ungeordneter'.)

Der (informatorischen) Entropie H wird ein Maß für die Informationsmenge, die ein Empfänger benötigt, um sein Unwissen zu beseitigen, zugeordnet (differentiell oder absolut). Der Empfänger fragt hierzu, der Sender antwortet mit Ja/Nein. Da die einzelnen Zeichen, aus denen eine Nachricht besteht, unterschiedlich häufig sind, muss dies in Form einer Gewichtung (p_i) beachtet werden. Damit kann die Entropie H mittels

$$H = -\sum_i p_i \cdot \log_2 p_i$$

definiert werden, wobei diese Definition der *ideellen Entropie* entspricht. Die *wirkliche Entropie* kann nur mit Angabe des tatsächlichen Verfahrens errechnet werden und ergibt numerische Werte $\geq H_{ideell}$. Die Bemühung, einen Code zu finden, der der ideellen Entropie entspricht, führt (theoretisch) zu optimalen Codes, da alle auftretenden Fälle ihrer Gewichtung nach codiert werden.

Die theoretischen Codes bleiben für eine praktische Realisierung nur in Form der Angabe einer Obergrenze interessant, da sie sich in idealer Form nur fallbezogen definieren lassen. Rechner und Netze sind aber für den allgemeinen Fall gebaut, so dass ein großer Aufwand für wenig Nutzen entstehen würde. Oberstes Prinzip einer Rechnerkommunikation bleibt die Offenheit, d.h. die Verwendung offener Codes bei eindeutiger Zuordnung einer Codierung zu Bitgruppen (→ 11.4), sowie die Systematik, die weltweit kommuniziert werden kann.

Intensive Bemühungen seitens der Codierungstheorie sind für Übertragungsratenreduktion beispielsweise bei Bildkommunikation (verlustbehaftete Kompression bei hohen Datenraten) und bei allgemeiner Datenübertragung (verlustlose Kompression) sowie in der Verschlüsselung ein aktuelles Forschungsthema.

11.2.8 Zerlegung analoger Funktionen in Oberwellen

Um eine beliebige analoge Funktion für einen digitalen Rechner heutiger Bauart (auf elektrischer Basis) zugänglich zu machen, werden zwei grundsätzliche Funktionen benötigt:

- Die analoge Funktion muss als elektrisches Signal zur Verfügung stehen oder mit Hilfe einer Wechselwirkung (Sensor) und einer eindeutigen Abbildung in dieses umsetzbar sein
- Die analoge Funktion muss digitalisierbar sein, d.h., die Wertemenge des Funktionsbereichs, die i.Allg. ein Intervall der reellen Zahlen darstellt, wird auf eine Menge von diskreten Werten abgebildet (Diskretisierung), die ihrerseits in Form von rechnerlesbaren Zeichen dargestellt werden (Codierung) (→ 1.2).

Umgekehrt werden bei Kommunikationen zwischen Rechnern digitale Signale mit Hilfe von analogen Funktionen übertragen. Die Sendezuordnung ist dabei eindeutig definierbar, bei der Übertragung jedoch kommt es zu Verlusten, so dass empfangsseitig die Zuordnung nicht mehr einfach möglich ist. Um diese Effekte transparent zu machen, wird die Sendefunktion als eindeutige Abbildung (Funktion im mathematischen Sinn)

$$f : T \rightarrow R$$

mit T als Intervall der reellen Zahlen (Zeitfunktion) aufgefasst. Dieser Darstellung im Zeitbereich entspricht eineindeutig eine Darstellung im Frequenzbereich. Dies kann mit Hilfe der *Fouriertransformation* mathematisch formuliert werden.

Die Fouriertransformation basiert auf der Darstellung einer (stetigen, z.T. auch unstetigen mit endlichen Sprungstellen) Funktion durch die Überlagerung von orthogonalen Funktionen trigonometrischer Natur (Sinus- und Cosinus):

$$f(t) = \int_{\omega \in R+} (A_\omega \cdot \sin(\omega t) + B_\omega \cdot \cos(\omega t)) d\omega$$

Für periodische Funktionen wird das Integral zur (unendlichen) Summe über diskrete Frequenzen ω. Die Interpretation des erhaltenen Spektrums wird in diesem Fall besonders anschaulich: Es existiert (neben einem eventuellen Gleichspannungsanteil in B_0) eine unterste Frequenz mit einem Koeffizienten A_ω oder B_ω ungleich 0. Diese Frequenz wird *Grundschwingung* genannt, alle anderen sind ganzzahlige Vielfache dieser Grundfrequenz, sogenannte *Oberwellen*.

Die Bestimmung der Koeffizienten A_ω und B_ω wird Fourieranalyse genannt; dies geschieht im mathematischen Sinne durch Berechnung von bestimmten Integralen über die Taktperiode (bei unperiodischen Funktionen bis ins Unendliche), im algorithmischen Sinn z.B. durch die Diskrete Fouriertransformation (→ 9.2) oder die Fast-Fourier-Transformation für digitale Rechenanlagen. Das Ergebnis sind dabei zwei Amplitudenfunktionen A und B in Abhängigkeit von ω, oder – unter Definition einer Phasenbeziehung – eine Amplituden- und Phasenfunktion von ω. Diese Darstellung wird *Frequenzspektrum* genannt. Wird anstelle der Amplitude die Energie (proportional dem Amplitudenquadrat) über der Frequenz aufgetragen, spricht man von einem *Frequenzenergiespektrum*. Für die praktische Ausführung der Fouriertransformation in Form einer diskreten Transformation siehe auch 9.2.

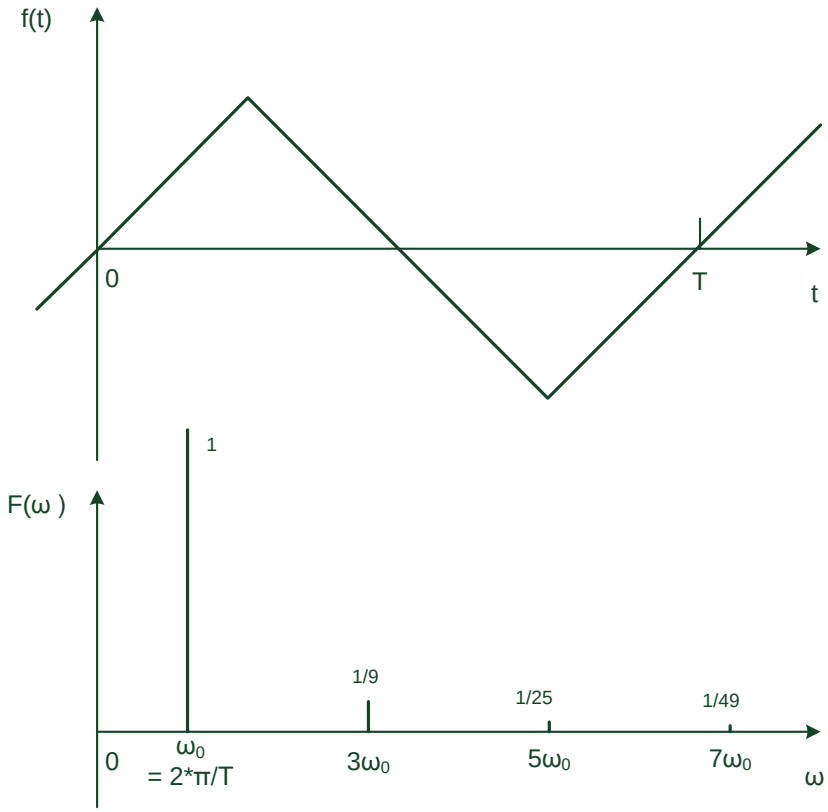


Bild 11.3 Frequenzspektrum einer (unbegrenzten) Dreiecksschwingung

Die übertragende Leitung bildet nun ein Signalverarbeitungssystem mit frequenzabhängigen Komponenten, d.h., die Übertragungsfähigkeit und damit die Dämpfung hängen von der Frequenz des Signals ab. Mit anderen Worten: Hochfrequente Teile des Signals werden ggf. gedämpfter übertragen als die niederfrequenten, das Signal wird verfälscht. Die Oberwellen sind dabei umso höher, je steiler die Flanken in dem Ursprungssignal sind, und sie veschleißen durch höhere Dämpfung stärker. Die Übertragungsstrecke hat modellhaft folgendes, in Bild 11.5 dargestelltes Ersatzschaltbild:

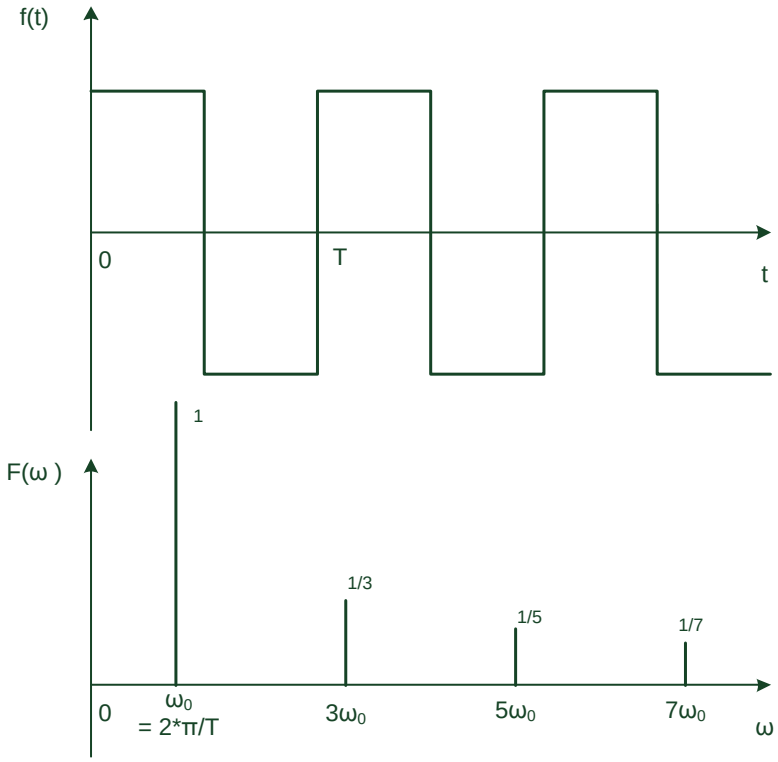


Bild 11.4 Frequenzspektrum eines Rechtecks (→ 9.2)

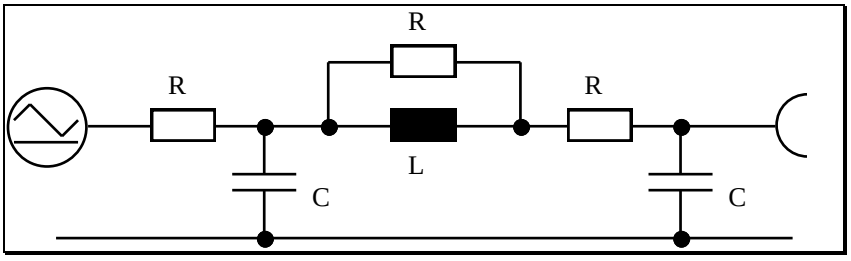


Bild 11.5 Ersatzschaltbild einer Übertragungsstrecke

Zur Ermittlung einer optimalen Signalform müssen jedoch nicht nur Faktoren wie Signifikanz, Übertragungsfunktion und möglichst geringe Wechselwirkung durch gegenseitige Beeinflußung, sondern auch Störunanfälligkeit einbezogen werden.

11.3 Physical Layer, Teil 1: Physikalische Grundlagen

11.3.1 Elektrische Systeme

Wie bereits dargestellt, besteht ein elektrisches Übertragungssystem aus aktiven Elementen (Spannungs- oder Stromquellen), die gesteuert werden und passiven Elementen wie Widerstände, Kondensatoren und Spulen; die modellhafte Vorstellung der reinen Übertragungsleitung geht dabei nur von R, C und L aus.

Die Übertragung eines einzelnen Signals, beispielsweise eines Bits, besteht aus elektrischer Sicht in der Umladung von Kapazitäten mit Hilfe von Strömen, die durch Widerstände und Spulen fließen. Bestimmend für die Umladezeit ist – bei Abwesenheit von Induktivitäten – die Zeitkonstante

$\tau = R \times C.$

Diese Werte können innerhalb der ICs mit z.B. R = 50 Ω .. 100 kΩ bei C ≤ 100 pF angenommen werden, so dass mit τ = 5 ns … 10 µs gerechnet werden kann; mit einer Umladezeit ≈ 2 • τ ergeben sich so maximale Übertragungsfrequenzen von 10 kHz und 100 Mhz, die durch Leitungseffekte weiter eingeschränkt werden.

Die einzelnen Kabelarten für eine elektrische Übertragung sind:

- Einfache Leitung (nebeneinander, unverdrillt)
- Verdrilltes Leitungspaar (twisted pair) mit oder ohne gemeinsame Abschirmung
- Flachbandkabel
- Mehraderkabel, z.B. mit gemeinsamer Abschirmung
- Koaxialkabel

Von diesen Kabelarten sind für hochfrequente Übertragungen oder weite Strecken insbesondere die Twisted Pair Verdrahtungen (→ 11.2.3.4) und die Koaxialkabel geeignet. Die zeichnen sich durch geringe Strahlung (und damit Übersprechen, insbesondere bei Koaxialkabeln) bei moderaten Kapazitäten und Widerständen aus; Twisted Pair ist insbesondere häufig vorhanden und damit benutzbar.

11.3.2 Lichtwellenleiter

Die (gebundene) Nachrichtenübertragung mittels (quasi-)optischer Wechselwirkung beruht auf dem Einsatz von Lichtwellenleitern, in denen sich die elektromagnetische Welle (Licht im sichtbaren oder nahen infraroten Bereich) nahezu ungedämpft ausbreiten kann. Da das Funktionsprinzip aller kommerziell eingesetzten Computer auf elektrischer Basis beruht, muss ein aufwendigeres Interfacing zwischen Rechner und Netz integriert werden. Dies besteht im Kern aus LEDs

(Light Emitting Diodes) oder Halbleiterlasern sendeseitig sowie Photo–transistoren oder -dioden empfangsseitig.

Das Prinzip der Lichtwellenleiter (LWL, fibre optics) beruht auf einer permanenten Brechung oder auf der Totalreflexion, je nach Ausführung des LWL. Der Brechungsindex, aufgetragen über den Querschnitt des LWL, ist für die jeweilige Form verantwortlich:

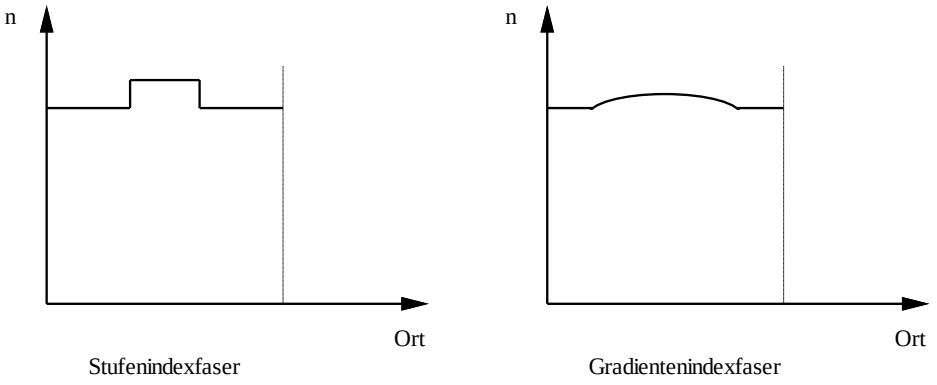


Bild 11.6 Lichtwellenleiterarten

In der Stufenindexfaser werden Lichtstrahlen, die in einem bestimmten Winkel (Akzeptanzwinkel, der Sinus dieses Winkels heißt *Apertur*), auf den LWL auftreffen, durch permanente Totalreflexion im Kernbereich des Leiters gehalten. Dieser Unterschied im Brechungsindex zwischen Kern- und Mantelbereich wird z.B. durch mechanisches Ziehen erhalten, ein relativ preiswertes Verfahren. Der Nachteil dieser Faser liegt in den unterschiedlichen optischen Wegen, die die einzelnen Strahlen durchlaufen müssen. Dadurch wird die maximale Bitfolgerate begrenzt, da die Impulse verlaufen.

Die Gradientenfaser bricht alle Lichtstrahlen innerhalb des Kernbereichs ständig zur Mitte hin; Hierdurch erfolgt die Bündelung, die Lichtstrahlenwege sind jedoch nicht so unterschiedlich, da in Bereichen niedrigeren Brechungsindex' die Lichtgeschwindigkeit größer ist: Der optische Weg bleibt überwiegend für alle Strahlen gleich. Gradientenfasern werden durch gezielte Dotierung erhalten.

LWLs zeigen dennoch Verluste durch Streuung unterschiedlicher Herkunft; diese Streuungen werden an Inhomogenitäten wie Kristalliten, Blasen, und durch grundsätzliche physikalische Prinzipien hervorgerufen und lassen sich auf etwa 0,3 dB/km begrenzen.

11.3.3 Übertragung von Informationen

Die Darstellung von Informationen in elektrischer (und optischer) Form kann auf mehrere Arten geschehen, die im Folgenden zusammengefasst sind.

Die Informationen können auf elektrischer Basis durch

- Spannungsbereiche,
- eingeprägte Ströme oder
- Modulierungen einer Grundschwingung

dargestellt werden. Im optischen Bereich wird zurzeit lediglich eine Informationsdarstellung durch Impulsfolgen, die mit der Soannungs- oder Stromsteuerung vergleichbar sind, genutzt.

Die ***spannungsgesteuerten Schnittstellen***, beispielsweise RS-232 oder RS-422 (→ 11.5), definieren für '1' und '0' Spannungsbereiche, die signifikant voneinander unterschieden werden können. Aus dem empfangenen Signalwert kann dann der Zeichenwert abgeleitet werden.

Ein Beispiel für eine ***stromgesteuerte Schnittstelle*** stellt die 20mA-Schleife dar. Hier wird der Zeichenwert '1' durch einen aufgeprägten Strom von 4 … 20 mA übertragen, '0' entsprechend durch keinen Strom. Die Übertragung ist bereits frühzeitig definiert worden, kann nur mit geringen Baudraten angewendet werden, besitzt dann aber eine hohe Störfestigkeit insbesondere in industriellen Anwendungen!

Die frequenzabhängigen Schnittstellen werden im allgemeinen Modulationsverfahren genannt. Hier wird eine Grundschwingung, die durch Amplitude, Frequenz und Phasenlage bestimmt ist, zeichenabhängig so verändert, dass der Signalwert eindeutig in einen Zeichenwert zu interpretieren ist. Entsprechend den veränderlichen Werten werden dabei ***Amplitudenmodulation*** (AM), ***Frequenzmodulation*** (FM) und ***Phasenmodulation*** (PM) unterschieden.

Bevor analoge Eingangssignale in Form von digitalen Zeichenwerten übertragen werden können, muss eine Quantisierung und Codierung erfolgen – dies war bereits in Abschnitt 1.2 erwähnt worden. ***Diese AD-Wandlung*** unterliegt technischen Randbedingungen:

- Die Anzahl der Quantisierungsstufen muss festgelegt werden; dies wird üblicherweise in Form von 2er-Potenzen geschehen, so dass 256 Stufen eine Codierung in 8 Bits benötigen, diese aber vollständig ausnutzen.
- Die Abtastrate muss so festgelegt werden, dass die höchste darzustellende (und zu übertragende) Frequenz (siehe Frequenzanalyse) digitalisiert werden kann. Nach dem ***Nyquist***-Kriterium muss dazu die Abtastfrequenz mindestens um den Faktor 2 größer sein (besser wäre Faktor 5 – 10, man nutzt beim Telefonieren ca 2,5). Abtastrate * Anzahl der Codierbits ergibt die Nettobitrate für die Übertragung.
- Bei der AD-Wandlung sind zusätzliche Bedingungen zu beachten: Das analoge Signal darf sich während der Digitalisierung nicht ändern (Sample&Hold-Schaltung), und die höchste digitalisierbare Frequenz darf im Eingang nicht überschritten werden (Tiefpassfilter), soll es nicht zu Fehlern kommen.

11.3.4 Informationsübertragungsmodell

Eine der zentralen Aufgaben für den Empfänger ist die Detektierung von Zustandsänderungen am Eingang, um ein Zeichen oder eine Zeichenkette zu erkennen. Diese Verfahren werden in den nächsten Abschnitten näher betrachtet.

Weiterhin muss der Empfänger erkennen können, ob das empfangene Zeichen für ihn bestimmt ist, oder ob es sich um eine Störung oder Falschsendung handelt:

In der ***asynchronen Übertragung*** wird ein Zeichen als Ganzes übertragen, eingeschlossen in Start- und Stoppbits und versehen mit einem entsprechenden Protokoll. Dies funktioniert nur mit Punkt-zu-Punkt-Verbindungen, da es zeichenbasiert ist. Dies sollte nicht mit weiteren, mit asynchron oder synchron in Beziehung stehenden Begriffen verwechselt werden!

Die ***synchrone Übertragung*** nutzt spezielle Start- und Stoppzeichen und umgibt damit die Nutzinformation innerhalb dieser Blockung. Während also die einzelnen Zeichen in einer asynchronen oder synchronen Prozedur übertragbar bleiben, synchronisieren die Start- und Stoppzeichen die Nutzinformation für den Empfänger.

Bei der ***Paketübertragung*** werden zusätzliche Informationen wie Paketnummer, Empfänger usw. mit übertragen. Diese Übertragungsform wird beispielsweise genutzt, um größere Datenmengen zu strukturieren und durch die zusätzlichen Informationen anschließend wieder zu synchronisieren. Innerhalb der Pakete erfolgt wieder die Synchronisation durch die o.g. Verfahren.

11.3.5 Zur Ausnutzung von Kanälen

Eine weitere Frage, die in dem Betrieb einer Kommunikation durchaus sehr wichtig werden kann, besteht in der Ausnutzung der Kanalkapazität. Die theoretisch berechenbare oder technisch nutzbare Kapazität wird in der Regel nicht durch eine einzige Kommunikation ausgenutzt; liegen hier erhebliche Investitionen oder Betriebskosten vor, ist der Betreiber aus wirtschaftlichen Gründen geneigt, diese Kapazität besser zu nutzen, indem er die Ressourcen mit anderen teilt.

Die Mehrfachnutzung der Bandbreite eines Kommunikationskanals kann entweder durch ***Konzentrierung*** oder durch ***Multiplexen*** geschehen. Die ***Konzentrierung*** findet sich häufig in Paketübertragungsnetzen wie X.25 (Datex-P). Ein Kommunikationskanal bleibt ungenutzt, sofern keine zu übertragenden Zeichen vorliegen. Im Fall einer Datenübertragung wird das Paket mit entsprechender Adressierung versehen und erreicht so den Empfänger. Dieses Verfahren, auch in LANs (Ethernet, Token Ring) implementiert, eignet sich besonders für ungleichmäßige und unregelmäßige Kommunikationen.

Beim ***Multiplexen*** werden Teile der gesamten Kanalkapazität den einzelnen Kommunikationen zugeordnet. Dies kann durch Frequenzmultiplexen oder Zeitmultiplexen geschehen; ersteres ordnet verschiedene Frequenzbereiche den einzelnen Kanälen zu und wird mit modulierten Übertragungen angewendet.

Das *Zeitmultiplexen* kann durch die synchrone Zuteilung von Zeitschlitzen an die einzelnen Kanäle geschehen. Diese Technik wird z.B. beim ISDN genutzt, wo die Zeitsynchronisation durch geeignete Steuerzeichen erfolgt. Asynchrone Zeitmultiplexverfahren teilen die Übertragungskapazität zwar in Zeitschlitze auf, ordnen diese jedoch nicht fest zu; leere Schlitze werden gekennzeichnet, und jeder Sender darf einen solchen Zeitschlitz füllen. Diese Technik ist sehr effizient und wird beim ATM (Asynchronous Transfer Mode) im Breitband-ISDN genutzt.

11.4 Physical Layer, Teil 2: Leitungscodierung

Literatur: [KB94] Kowalk, W.P.; Burke, M.: Rechnernetze, Kapitel 4

Die Übertragung von Informationen erfolgt generell über Leitungen oder elektromagnetische Ausbreitung im Vakuum. Das physikalische Prinzip entstammt dabei meist der Elektrizität, in zunehmendem Maße auch der Optik; während elektrische Wechselwirkungen leitungsgebunden wie leitungslos genutzt werden können, beschränkt sich die optische Übertragung auf Lichtwellenleiter und nur in Ausnahmefällen (über kurze Distanzen) auf die Ausbreitung in Vakum/Luft.

Zur Nachrichtenübertragung muss der Sender die Zeichendarstellung der Nachricht in ein (oder eine Reihe von) physikalisches(n) Signal(en) umwandeln, welches der Empfänger als zugehöriges Zeichen interpretiert. Diese physikalische Darstellung wird als *Leitungscodierung* bezeichnet.

11.4.1 Grundsätzliches zur Codierungstechnik

Die Informationen in der Informatik können in der Regel als binäre digitale Werte (d.h. zweiwertig mit diskreten Wertebereichsdarstellungen) aufgefasst werden; die Verfahren, die von einer Nachricht hierhin führen, sind hinlänglich bekannt, allerdings nicht Gegenstand dieses Handbuchs.

Die Darstellung erfolgt dann in sogenannten Bits (als Abkürzung für Binary Digit), wobei mehrere Bits zu einem *Byte*, in der Nachrichtentechnik häufig als *Oktett* bezeichnet (8 Bits), oder auch *Wörtern* mit z.B. 16 oder 32 Bit Länge zusammengefasst werden können. Die Bits können die (logischen) Werte 0 und 1 annehmen und werden entsprechend bezeichnet (0-Bit, 1-Bit).

Bei einer Anzahl von n Bits kann diese Bitanhäufung 2^n-Werte annehmen und somit codieren. Dementsprechend sind in Bytes 256 Werte, in 16-Bit-Wörtern 65536 und in 32-Bit-Wörtern 4294967296 darstellbar; die übliche Codierung eines Alphabets kommt mit 7-Bit bzw. bei Sonderzeichen mit 8-Bit aus. Der z.T. nicht mehr existierende Telexdienst nutzte beispielsweise nur 5 Bits, um Zeit zu sparen, und hatte Umschaltzeichen für Buchstaben und Ziffern.

Reichen die codierbaren Zeichen nicht aus, um alle Sonderfälle wie beispielsweise eine Kommandoübergabe, die nicht als Nachricht zu interpretieren ist, übertragen

zu können, werden gewöhnlich *ESCAPE*-Zeichen (z.B. 0x1B als *Data Link Escape*) definiert, die das eine oder mehrere Zeichen als besonders zu interpretieren darstellen. Dies Verfahren lässt sich besonders an Übertragungen zwischen Rechnern und Druckern beobachten, wo die ESCAPE-Sequenzen Befehle an den Drucker zur Formatierung codieren, während die reine Nachricht eben auszudrucken ist.

Kann eine Nachrichtenübermittlung jede beliebige Zeichenfolge übertragen (also ohne Beschränkung z.B. auf die darstellbaren Zeichen), spricht man von *transparenter* Datenübertragung.

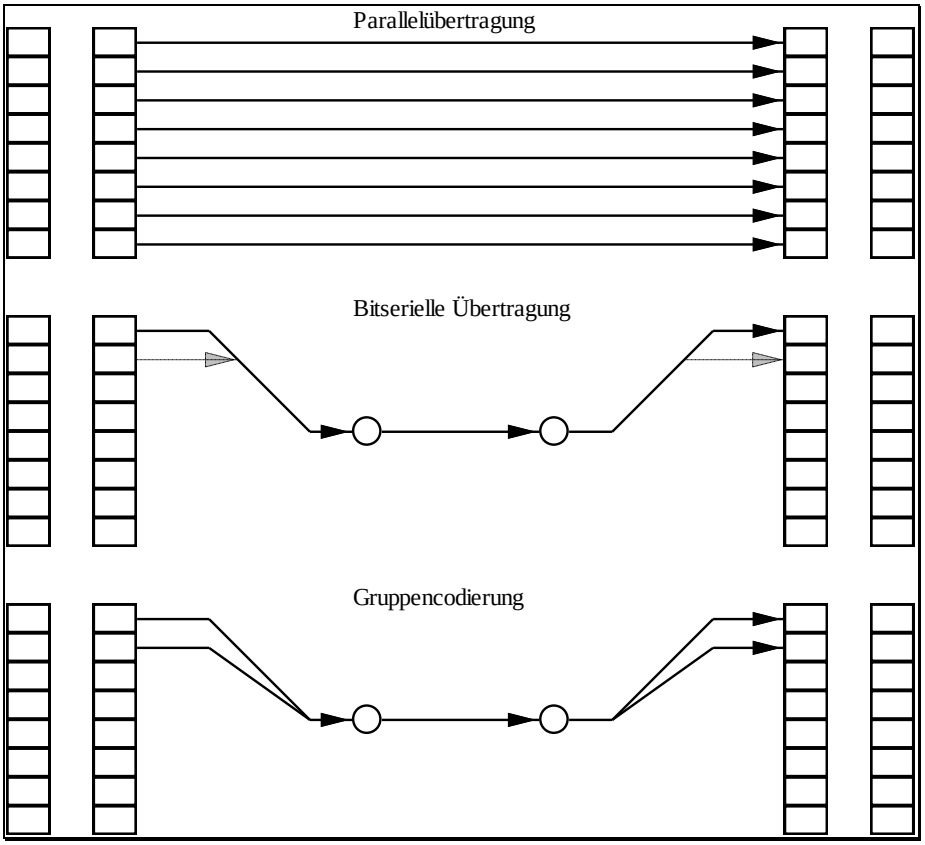


Bild 11.7 Übertragungsverfahren

Zur Übertragung selbst müssen die Zeichen in geeignete Einheiten zerlegt werden (im Zweifelsfall in Bits, dann spricht man von *bit-serieller* Übertragung); diese Einheiten werden dann in physikalische Signale umgewandelt, am Empfänger rückgewandelt und wiederum zusammengesetzt. Bei Übertragung mehrerer Signal-

werte gleichzeitig spricht man von *paralleler* Übertragung (Beispiel: IEEE-488, byte-seriell aber bit-parallel). Hierfür sind naturgemäß mehrere Leitungen notwendig, so dass durch diesen Aufwand bedingt die Übertragungsentfernung nur gering bleiben wird (ca 10 m). Wird jedoch mehr als ein Bit in einem Signalwert codiert, so nennt man dies *Gruppencodierung* (Beispiel: V.42-Norm für DFÜ) .

11.4.2 Betriebsarten

Die reine Übertragung von Informationen von einem Ort zu einem anderen wird (unabhängig von Quittierungen) *Simplexbetrieb* (Richtungsbetrieb, Abkürzung sx) genannt. Dies kommt einem Verteildienst wie Rundfunksendungen gleich. Das Aussenden von einfachen Quittierungen kann dabei durch Rücksendesignale auf Bitbasis geschehen; komplexe Quittierungen sind bei Simplexbetrieb hiermit jedoch kaum hantierbar. Bei Nachrichten mit sehr langer Laufzeit (Satelliten in geostationärer Erdumlaufbahn) wird auf derartige Nachrichten verzichtet.

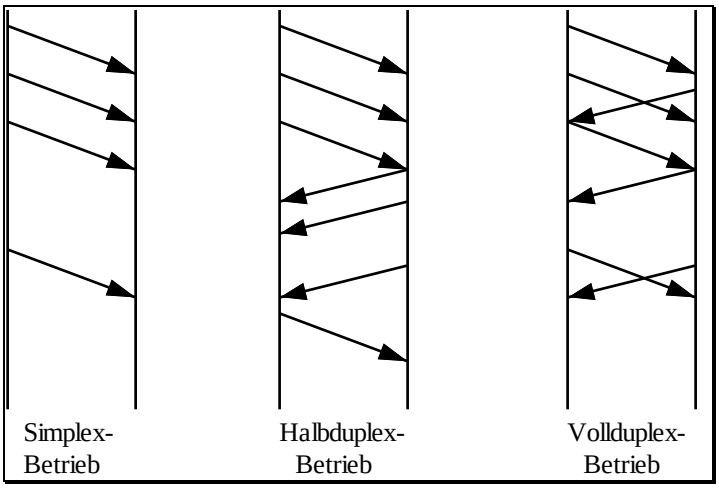


Bild 11.8 Betriebsarten von Übertragungseinrichtungen

Läßt sich die Übertragungsrichtung in dem Sinne ändern, dass zu einem Zeitpunkt nur eine Richtung zulässig ist, im Prinzip aber beide Richtungen möglich sind, wird von *Halbduplex* (Wechselbetrieb, Abkürzung hx) gesprochen. Eine Wechselsprechanlage beispielsweise entspricht diesem Prinzip, während z.B. Ethernet dieser Betriebsart entsprechen kann (hier sind die Betriebsarten meist einrichtbar und hängen von der Verbindung zum Switch ab).

Der *Vollduplexbetrieb* (Gegenbetrieb, dx) wird durch die Sendemöglichkeit beider Stationen zur gleichen Zeit charakterisiert. Dies wird z.B. durch verschiedene physikalische Signalträger (Leitungen) oder Übertragungstrennung mittels verschie-

dener Modulationen erreicht und ermöglicht den kontinuierlichen Datenaustausch einschließlich echtzeitfähiger Quittierungen und Fehlermeldungen.

11.4.3 Übertragungsprozeduren

Während nunmehr die Aussendung von Daten bereits teilweise geklärt ist (es gibt noch eine Vielzahl von Fragen, die die nächsten Abschnitte füllen werden), muss noch größere Sorge für die Synchronisierung von Empfänger und Sender getragen werden. Dies liegt insbesondere daran, dass dem Empfänger die Sendezeitpunkte nicht bekannt sind (von wenigen, sehr extremen Ausnahmen einmal abgesehen). Der Empfänger muss daher ständig (im wahrsten Sinne des Wortes) auf seiner Empfangsleitung mithören, sprich den Pegel abtasten, um Änderungen detektieren und darauf reagieren zu können. Dies liefert ihm zwei Informationen: Eine Sendung beginnt (Startzeichen), und er bekommt die übermittelte Information.

Grundsätzlich kann jedoch kein Empfänger zu jedem Zeitpunkt an der Leitung mithören; die Gründe zum Ausfall sind z.T. trivial, da das Gerät beispielsweise ausgeschaltet ist, die Energieversorgung unterbrochen wurde, ein Kabelbruch vorliegt oder aber der Empfänger von der Informationsflut überrollt wurde. Aus diesem Grund werden *Synchronisationszeichen* ausgetauscht, um den notwendigen Gleichlauf herzustellen. Diese Synchronisationszeichen enthalten keine Fehlerkennungen o.ä., sondern stellen lediglich bei beiden Teilnehmern den gleichen Zustand her, um anschließend auf gesicherter Basis weiter miteinander kommunizieren zu können.

Die Art der Synchronisation unterscheidet die verschiedenen Verfahren zur Informationsübertragung. Die Normung findet im ISO/OSI-Basisreferenzmodell im Rahmen der Bitübertragungs- und Sicherungsschicht statt, wo von *Übertragungsprozeduren* gesprochen wird. Es werden synchrone und asynchrone Übertragungsprozeduren unterschieden, wobei erstere nochmals in bit- und zeichensynchrone Verfahren unterteilt ist.

11.4.3.1 Synchrone Übertragungsprozeduren

Die *synchrone Übertragungstechnik* garantiert einen Gleichlauf zwischen Sender und Empfänger über praktisch beliebig lange Zeiten. Dies kann in Ausnahmefällen über hochgenaue Uhren geschehen oder durch eine ständige *Synchronisierung*. Diese Synchronisierung besteht dabei aus der Definition eines Synchronisationsmasters (fast immer der Sender, ggf. eine zentrale Taktfunktion), der dem anderen Partner mitteilt, wann das nächste Zeichen oder Bit gesendet wird. Die Aussendung kann dabei über eine eigenständige Leitung oder Taktrückgewinnung aus dem im physikalischen Signal codierten Takt geschehen.

Die Konsequenzen eines solchen Verfahrens sind:

- Lange Laufzeiten ohne aufwendige Resynchronisationsverfahren auf Blockbasis sind möglich

- Die Geschwindigkeit lässt sich gegenüber asynchronen Verfahren wesentlich erhöhen (häufig sind Interfacebausteine für Taktverhältnisse 16 : 1 (synchron : asynchron) ausgelegt).

Eine weitere Synchronisierung besteht in der Aufteilung der zu übertragenden Nachricht in *Übertragungsblöcke* (transmission blocks). Ein solcher Block kann in der Regel jederzeit (also asynchron) gesendet werden und wird zur Kenntlichmachung durch *Steuerzeichen* (control character) eingeleitet bzw. begrenzt.

Zur Übertragung werden zunächst einige (2 bis 8) SYN-Zeichen auf die Leitung gelegt; diese dienen dem Empfänger zur eigentlichen (Block-)Synchronisierung, während die Bit-Synchronisierung weiterhin auf der Übertragung des Takts basiert. Der eigentliche Block beginnt mit dem *STX*-Zeichen (Start-of-Text, ASCII-Wert 2), es folgen die Daten (in dem jeweiligen Format) und am Ende das *ETX*-Zeichen (End-of-Text, ASCII-Wert 3). Anschließend werden in der Regel Prüfsummenbytes geschickt, die aus den übertragenen Zeichen berechenbar sind und der Fehlerkontrolle dienen (*BCC*, Block Check Character, i.d.R. 2-4 Bytes).

Bild 11.9 Aufbau eines Übertragungsblocks

Die Blöcke sind meist in der Länge begrenzt, häufig wird die Länge exakt festgelegt oder mit den ersten Zeichen übertragen. Die Begrenzung hat pragmatische Gründe (Blöcke müssen beim Empfänger zwischengespeichert werden), technische (der Gleichlauf zwischen Sender und Empfänger kann mit Sicherheit nur eine gewisse Zeit aufrechterhalten werden) sowie theoretische Gründe (optimale Länge für Fehlersicherung durch eine bestimmte Anzahl von Fehlerzeichen). Um mit dieser Begrenzung Informationen beliebiger Länge übertragen zu können, muß die Nachricht ggf. in mehrere Blöcke unterteilt werden. Man spricht dann von *Fragmentierung* sowie *Füllung* bei zu kleinen Blöcken. Das ETX-Zeichen wird bei Teilblöcken, die nicht der letzte Block sind, in diesem Fall gewöhnlich durch das ETB-Zeichen ersetzt.

In zeichenorientierten Übertragungen kann die Übertragung von Zeichen (beispielsweise aus Grafiken), die Steuerzeichen entsprechen, wiederum durch Escape-Zeichen, die vor das zu übertragende Zeichen gestellt werden, erreicht werden. Ein solches Escape-Zeichen (DLE, ASCII-Wert 27) verhindert die Interpretation des nachfolgenden als Steuerzeichen und wird beim Empfänger einfach wieder entfernt. Diese Technik wird *Codeerweiterungstechnik* (character stuffing) genannt.

Bit-orientierte Übertragungen, die also keine Blockung einer festgelegten Bitanzahl zu Zeichen kennen und somit mit einer beliebigen Anzahl (und Folge) von Bits ablaufen, wird mit einer anderen Technik gearbeitet, dem *Bitstopfen* (bit stuffing). Auf eine Folge von 5 Bits (beispielsweise so im HDLC-Protokoll (high level data

link Protocol) implementiert) von Nutzdaten, die alle den Wert ‘1’ besitzen, folgt eine zusätzlich ‘eingemischte’ ‘0’, die beim Empfänger wiederum herausgefiltert wird. Wird diese Folge nun verletzt, indem eine Folge von 6 ‘1’-Bits gesendet wurde, handelt es sich um ein Steuerzeichen (z.B. flag ‘01111110’), das entsprechend zu interpretieren ist.

11.4.3.2 Asynchrone Übertragungsprozeduren

Asynchrone Übertragungsprozeduren sind grundsätzlich zeichenorientiert, d.h. eine fest definierte Anzahl von Bits (5 - 8) werden als Basiseinheit übertragen. Zwischen Sender und Empfänger muss eine Synchronisation auf den Start gegeben werden, da dieser zu einem beliebigen Zeitpunkt stattfinden kann, die anschließenden Bits werden nicht synchronisiert, d.h. der Zeitpunkt der Bitabtastung muss generell vereinbart werden (Bitfolgerate).

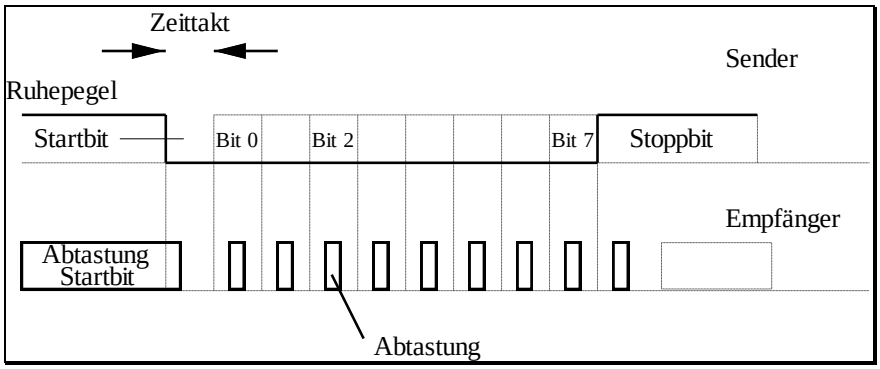


Bild 11.10 Asynchrone Übertragung

Die Synchronisierung geschieht durch Definition eines Ruhepegels und Ausgabe eines Startbits, das von diesem Ruhepegel abweicht. Der Empfänger reagiert auf die Flanke, die durch diesen Wechsel entsteht. Im Einzelnen gehen Sender/Empfänger wie folgt vor:

1. Der Sender hält auf der Leitung den sogenannten Ruhepegel (Signalwert 1), der vorteilhafterweise von einem Wert ohne Kabelanschluß abweicht, um ggf. Kabelbrüche detektieren zu können.
2. Soll ein Zeichengesendet werden, dann geschieht dies durch das sogenannte Startbit, das von dem Ruhewert abweicht (Signalwert 0). Dieses Startbit kann in seiner Länge von den Nutzdatenbits abweichen, muß aber nicht. Der Empfänger muss die Datenleitung ständig abhören, um das Startbit zu detektieren, oder er reagiert auf die (negative) Flanke.
3. Der Sender gibt die vorher vereinbarte Zahl von Nutzdatenbits in seinem lokalen Takt auf der Datenleitung aus; der Empfänger leitet aus der Detektierung

des Startbits und der vereinbarten Bitfolgerate die Zeitpunkte der Datenbitabfragen her und übernimmt den Bitwert der Datenleitung zu diesen Zeitpunkten.

4. Den Datenbits kann ein zusätzliches Paritätsbit folgen (nicht in Bild 11.10).
5. Der Abschluss wird durch ein Stoppbit signalisiert, das den gleichen Wert wie der Ruhepegel hat und somit vom Startbit zu unterscheiden ist. Dieses Stoppbit kann länger als die Datenbits sein (üblich sind 1, 1.5 oder 2 Bits), unmittelbar folgend kann das nächste Startbit gesendet werden.

Für eine korrekte Übertragung müssen demnach die verschiedenen Pegel, die Länge des Startbits, die Bitfolgerate, die Anzahl der Datenbits, eventuelle Paritätbits und die Mindestlänge des Stoppbits vereinbart werden. Die Asynchronität bezieht sich hiermit auf den Abstand zweier beliebiger Zeichen, während die Bits sehr synchron (aber nicht synchronisiert) übertragen werden.

11.4.4 Übertragungscodes

Die Darstellung der Daten wurde bereits als *Zeichenwerte* bezeichnet. Für ein Bit werden üblicherweise die Zeichenwerte ‘0’ und ‘1’ als annehmbare Werte definiert, die ihrerseits zur Speicherung oder Datenübertragung in physikalischen Signalen, den *Signalwerten* dargestellt werden. Die Wechselwirkung und die Form der Darstellung sind dabei für die Logik sehr nebensächlich, solange die Interpretation eindeutig ist. Für einfache, asynchrone Übertragungsverfahren wie RS232 (jetzige Bezeichnung: EIA/TIA 562, auch mit V.24/V.28 bezeichnet) werden Spannungsbereiche zur Darstellung definiert: -3 bis -15 V bedeutet MARK, eine ‘1’, +3 bis +15V SPACE, eine ‘0’. Für die wesentlich sicherere Übertragung nach RS422/485 werden Differenzpegel zwischen zwei Leitungen für 1 und 0 angegeben. Die Abbildung zwischen Zeichen- und Signalwert wird als *Code* bezeichnet.

Die Abbildung zwischen Signal- und Zeichenwert wird im Allgemeinen (aber nicht immer!) als bijektiv (eineindeutig) gefordert, um eine sofortige Interpretation ohne Kenntnis etwaiger Randbedingungen (wie Vorgeschichte) durchführen zu können. Im Folgenden werden einige Codierungen betrachtet.

11.4.5 Eigenschaften von Übertragungscodes

Die Eigenschaften der einzelnen Übertragungscodes können nach ihren verschiedenen Eigenschaften klassifiziert werden, zu denen sowohl elektrische wie informationstechnische gehören:

- Gleichstromanteil, Störabstand
- Ausnutzung der Kanalkapazität
- Anzahl der verschiedenen physikalischen Signalwerte
- Fähigkeit zur Taktrückgewinnung
- Anzahl gemeinsam codierter Zeichen

- Fähigkeit zur Resynchronisation
- Redundanz

Die elektrischen Werte, die bei anderen Signaldarstellungen wie Optik beispielsweise entfallen können, beinhalten (exemplarisch) den *Gleichspannungsanteil* und den *Störabstand*. Der Gleichspannungsanteil eines Codes soll im Weitverkehr möglichst Null sein, um Übertrager einsetzen zu können. Diese einfache Forderung führt jedoch sofort zur Unterscheidung zwischen Daten- und Telekommunikationswelten und entsprechenden Interfaces zwischen beiden. Der Störabstand zwischen den einzelnen physikalischen Werten soll möglichst groß sein, um eine eindeutige Unterscheidung auch bei überlagerten Störungen zu ermöglichen; andererseits führen große Spannungsabstände zwischen den zulässigen Signalwerten zu langen 'Umladezeiten' und damit zu verringerten Übertragungsraten.

Die *Kanalkapazität* wird sehr häufig ebenfalls durch elektrische Größen beeinflusst; wichtig sind die maximale und minimale zu übertragende Frequenz bei gegebener Dämpfung (oder umgekehrt), wobei mit *Dämpfung* die Schwächung des Signalpegels gegenüber dem Nullwert oder dem Rauschen gemeint ist: Unterhalb einer Grenze kann nicht mehr detektiert werden, ob es sich hier um ein gewolltes Signal handelt oder nicht. Die Angabe der Kanalkapazität bezieht sich dabei auf die komplette Strecke, Übertragungsweg wie Sender/ Empfänger.

Die Anzahl der physikalischen Signalwerte kann in Zusammenhang mit dem Gleichspannungsanteil (möglichst 0) und weiteren Effekten wie Echounterdrückung gesehen werden. Das hier behandelte Grundprinzip der digitalen (digitus lat. = Finger) Zeichen/Signalwertzuordnung bleibt hiervon unberührt, lediglich die Zweiwertigkeit kann auf physikalischer Seite verloren gehen. Ein Beispiel hierfür ist der quasi-ternäre Code, der für 0 den Spannungswert 0, für 1 alternierend + und - Spannung bietet. Eine echte Mehrwertigkeit von Signalen ist ebenfalls möglich, führt aber zu geringen Signalstörabständen und kann sehr aufwendig sein.

Die *Taktrückgewinnung* erlaubt durch Codierung des Sendetakts in die Signalwerte die Synchronisation auf Bitbasis. Ist es auch notwendig, eine weitere Synchronisation auf Blockbasis einzuführen, geschieht dies durch spezielle Zeichen (*Rahmenbildung*), beispielsweise Synchronisationszeichen oder auch die bewusste Verletzung einer Codierung.

Wird mehr als ein Zeichen in einem Signalwert codiert, so wurde dies als *Gruppencodierung* bezeichnet. Anwendungen hierfür sind im ISDN und FDDI zu finden, wobei beachtet werden sollte, dass bei einer Gruppencodierung nur ganzzahlige Vielfache einer Gruppe codierbar sind.

Die Darstellung einer Gruppencodierung geschieht häufig durch Angabe der Eingangs- und Ausgangsanzahl der Signale sowie ihrer Wertigkeit. Die abkürzenden Buchstaben dafür sind B (binär), T (ternär) und Q (quartär), die Anzahl der Signal wird durch Ziffern angegeben: 4B2Q wäre die Darstellung von 4 binären Zeichenwerten durch 2 qartäre Signalwerte.

Die *Resynchronisation* geschieht zumeist auf Rahmenbasis und dient unter anderem der Fehlerbegrenzung beispielweise durch einen ‘Seiteneinstieg’ des Empfängers.

Redundante Codes dienen der Fehlererkennung. Je nach Aufwand für die Redundanz (die zusätzlichen Informationen müssen ja mit übertragen werden, führen also zu einer Minderung der Nettoübertragungsrate) kann man hierbei zwischen erkennden und korrigierenden Codes unterscheiden; bei erkennenden Codes, die zumeist angewendet werden, wird bei Fehlern erneut übertragen (→ 14.3.2).

Im Folgenden werden einige Codierung besprochen, wobei die elektrischen Eigenschaften im Wesentlichen nicht angesprochen werden, sondern mehr die informationstechnischen.

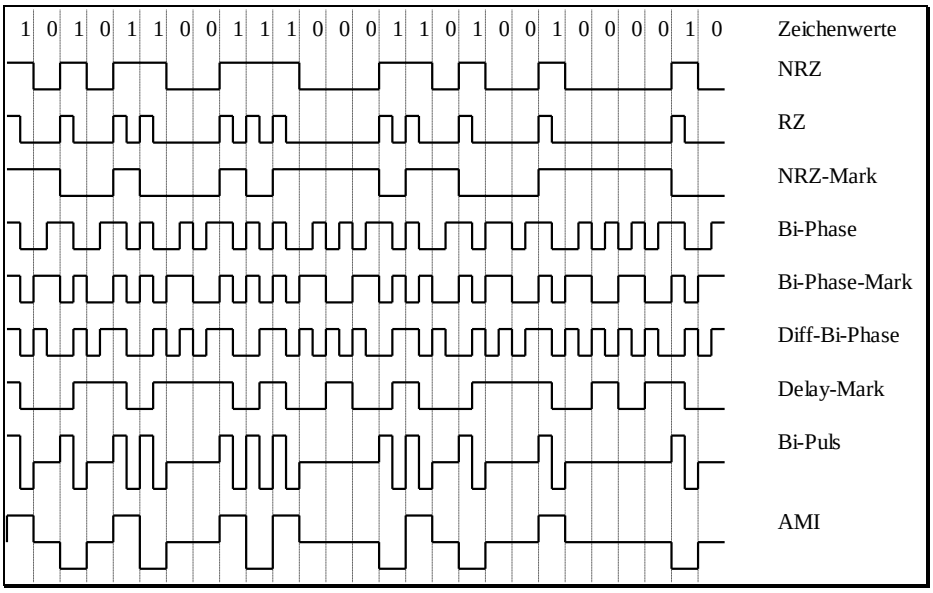


Bild 11.11 Signalwerte verschiedener Codes

Für die Darstellung im folgenden wird für die positive Spannung das ‘+’-Zeichen, für die negative das ‘-’-Zeichen und für den Nullwert des Signal ein ‘=’ vereinbart.

11.4.5.1 Non-Return-to-Zero (NRZ)

Diese einfachste Form der Codierung weist jedem Bit eindeutig einen Signalpegel zu; dies kann z.B. durch Zuweisung der ‘=’-Spannung an ‘0’ sowie der +-Spannung an ‘1’ geschehen, wobei meistens nicht bestimmte Werte, sondern Spannungsbereiche mit einer Trennung voneinander zugewiesen werden.

NRZ-Codes gestatten keine Taktrückgewinnung, die eine minimale Folge von Taktflanken nicht gewährt werden kann. Der Gleichstromanteil kann u.U. sehr stark sein. Fehler wirken sich nur auf einzelne Bits aus.

11.4.5.2 Return-to-Zero (RZ)

In dieser Form der Codierung wird eine 1 als Folge der Signalzustände '+' und '=' für jeweils eine halbe Taktperiode, eine 0 lediglich als '=' dargestellt. Der Gleichstromanteil ist dadurch nur halb so groß wie bei NRZ (und gleicher Bitanzahl), dafür jedoch die maximal zu übertragende Frequenz doppelt so hoch. Eine Taktrückgewinnung ist hier ebenfalls nicht möglich, da bei längeren 0-Folgen keinerlei Wechselsignal zur Verfügung steht.

11.4.5.3 Non-Return-to-Zero-Mark (NRZ-Mark)

Die einzelnen Bits werden nicht mehr durch Signale, sondern durch Signalwechsel dargestellt, wobei eine '1' einen Wechsel zum jeweils anderen Signalzustand ('+' oder '='), eine 0 jedoch keinen Wechsel vollzieht. Durch diese Codierung entsteht der Eindruck eines sehr ruhigen Codes, er entspricht im Frequenzverhalten dem NRZ-Code, auch beim Gleichstromanteil kann dieser beliebig groß werden. Eine Taktrückgewinnung ist ebenfalls nicht möglich.

11.4.5.4 Bi-Phase

Bi-Phase stellt jedes Bit in einem Wechselsignal dar: Eine +/=-Signalfolge stellt eine '1', eine =/+-Signalfolge entsprechend eine '0' dar. Dies bedeutet, daß die Frequenz sehr hoch ist, jedoch jederzeit ein Takt zurückgewonnen werden kann (PLL-Schaltung). Die Unterscheidung zwischen '1' und '0' kann allerdings nur dann mit Sicherheit getroffen werden, wenn eine 1-0-Kombination auftritt (größerer Schaltungsaufwand). Der Gleichspannungsanteil beträgt die Hälfte der '+'-Spannung.

11.4.5.5 Bi-Phase-Mark (Manchester)

Am Anfang einer Taktperiode wird in jedem Fall ein Pegelwechsel vollzogen, in der Taktmitte jedoch nur bei '1'-Bits. Dieses Verfahren sichert ebenfalls die Taktrückgewinnung, liefert sichere Signalstartzeitpunkte für die 0 und darüber für die 1 und ist besonders für die magnetische Aufzeichnung (Harddisk) geeignet.

11.4.5.6 Differential Bi-Phase

Bei der Differential-Bi-Phase wird in jedem Fall ein Signalwechsel in der Mitte des Takts vollzogen, am Anfang jedoch nur bei einer '0'. Die Eigenschaften unterscheiden sich damit nicht von denen der Bi-Phase-Mark-Codierung.

11.4.5.7 Delay-Mark

In der Mitte einer ‘1’ wird ein Signalwechsel vollzogen, am Anfang aller Bitzustände jedoch fortgelassen; dies würde zu einem ähnlichen Code wie NRZ führen; um zusätzlich eine Taktrückgewinnung zu ermöglichen, wird bei Folgen von ‘0’ mit mehr als 2 Bits bei jedem eingeschlossenen ‘0’-Bit am Taktanfang ein Wechsel eingesetzt.

Die Folge dieser Codierung ist ein recht ‘ruhiger’ Code, d.h. es sind keine besonders hochfrequenten Anteile enthalten, wobei trotzdem eine Taktrückgewinnung möglich ist. Der Nachteil liegt in der zusätzlichen Auswertung der Phasenlage eines Wechsels, wobei eine eindeutige Phasenlage nur an ‘101’ oder ‘1001’-Bitfolgen auszumachen ist. Dadurch erhöht sich der Aufwand im Empfänger, und das Verfahren ist fehleranfällig.

11.4.5.8 Bi-Puls (Dipuls)

Die Einführung eines quasi-ternären Signals, das durch seine Definition keinen Gleichspannungsanteil mehr enthält, führt zu weiteren Codierungen, wobei bei der Bi-Puls-Codierung für eine ‘1’ ein +/- -Impuls, für eine ‘0’ ein ‘=’-Signal gesendet wird. Hierdurch ist einerseits die maximale Frequenz recht hoch, der Takt aber nicht rückgewinnbar. Die Bedeutung dieser Codierung ist entsprechend sehr gering.

11.4.5.9 Alternating Mark Insertion (AMI)

Die einfache AMI-Codierung entspricht der NRZ-Codierung, wobei jedoch alternierend eine ‘1’ durch das ‘+’ oder ‘-’-Signal dargestellt wird, die ‘0’ wie bisher durch das ‘=’-Signal. Die Folge dieses quasi-ternären Codes ist kein Gleichspannungsanteil, allerdings auch keine Taktrückgewinnung.

In einer speziellen Ausführung mit Einfügung einer Gruppencodierung und teilweise Verletzung der einfachen Inversionsregeln wurde AMI jedoch als G.703 durch die CCITT zur Standardcodierung für die 2-Mit/s-S0-Kanäle im ISDN. Diese Codierung wird in den Abschnitten 11.4.6.3 und 11.4.6.4 behandelt.

11.4.5.10 Wertung der Codierungsformen

Viele der Codes, die in diesem Subkapitel behandelt wurden, sind mehr von theoretischem Interesse oder gelten als Grundlage zur Weiterentwicklung. Im täglichen Gebrauch sind besonders die NRZ-Codierung (RS232/EIA/TIA562), die Manchester-Codierung (Bi-Phase-Mark für magnetische Aufzeichnung) und AMI als G.703 zu finden.

11.4.6 Gruppencodierung

Unter Gruppencodierung wird die Abbildung einer Menge von Zeichen auf eine Menge von Signalen verstanden, wobei die Elemente der Signalmenge ihrerseits

aus mehr als einem Signalzustand bestehen. Die Abbildung ist hierbei im Sinne von Relationen zu sehen, d.h. es gibt Dupel von Zeichen/Signalgruppen, die erlaubt sind, wobei einem Zeichen durchaus mehrere Signalgruppen, umgekehrt jedoch jedem Signal exakt ein Zeichen zugeordnet wird. Die Codierung kann in deterministischen endlichen Automaten dargestellt werden; da die Sendecodierung offenbar zu mehreren Signalgruppen führen kann, die Empfangsdecodierung jedoch zu eindeutigen Zeichen, ist der Empfangsautomat in der Regel einfacher ausgeführt.

Die Darstellung der Codierung geschieht durch Benennung der Anzahl der Eingangswerte (und deren Wertigkeit, z.B. B für binär) sowie der Ausgangswerte. Diese Benennung ist natürlich nur teilqualifizierend, so dass eine genaue Angabe der Codierung in Form von Abbildungsvorschriften unerläßlich ist.

11.4.6.1 4B/5B-Codierung

Die mithin einfachste Gruppencodierung, die z.B. bei Fast Ethernet (100 MBit/s, → 11.5.3.2) Anwendung findet, besteht darin, dass jeweils 4 Bit des zu übertragenden Datenstrom in 5 Bits übertragen werden [Wiki_4B5B]. Man verfolgt damit die Ziele,

Tabelle 11.1 4B/5B-Codierung

Bezeichnung	4B	5B	Funktion	Bemerkung
0	0000	11110	Hex data 0	
1	0001	01001	Hex data 1	
2	0010	10100	Hex data 2	
3	0011	10101	Hex data 3	
4	0100	01010	Hex data 4	
5	0101	01011	Hex data 5	
6	0110	01110	Hex data 6	
7	0111	01111	Hex data 7	
8	1000	10010	Hex data 8	
9	1001	10011	Hex data 9	
A	1010	10110	Hex data A	
B	1011	10111	Hex data B	
C	1100	11010	Hex data C	
D	1101	11011	Hex data D	
E	1110	11100	Hex data E	
F	1111	11101	Hex data F	
Q	-NONE-	00000	Quiet	Signalverlust
I	-NONE-	11111	Idle	
J	-NONE-	11000	Start #1	
K	-NONE-	10001	Start #2	
T	-NONE-	01101	End	
R	-NONE-	00111	Reset	
S	-NONE-	11001	Set	
H	-NONE-	00100	Halt	

- die Taktrückgewinnung zu unterstützen, indem die normalen Datenhalbbytes (aus dem eigentlichen Datenstrom stammend) niemals durch die Bitfolge „00000“ bzw. „11111“ übertragen werden, sowie
- Steuerungssignale zur Signalflusskontrolle zur Verfügung zu haben

Somit sind weder Fehlererkennung oder –korrektur noch eine Gleichspannungsfreiheit im Fokus dieser Codierung. Tabelle 11.1 zeigt die Zuordnung der 5B-Codes zu den 4B-Codes.

Wie die Steuercodes verwendet werden – meist in Form von 2 Steuerzeichen, hängt vom jeweiligen Netzwerkstandard ab [Wiki_4B5B]. In jedem Fall wird die Brutto-Übertragungsrate nur zu 80% ausgenutzt; aus diesem Grund wird bei Fast Ethernet – 100 MBit/s – die reale Übertragungsrate auf 125 MBit/s hochgesetzt.

11.4.6.2 Paired Selected Ternary (PST, 2B2T)

Bei PST wird jeweils einer Gruppe von 2 Binärwerten eine Gruppe von 2 Ternärwerten zugeordnet: Es liegt also ein 2B2T-Code vor. Dies bedeutet die Zuordnung von 4 Eingangswerten auf maximal 9 Signalgruppen, von denen nur 6 ausgewählt werden. Die Zuordnungstabelle hat folgende Gestalt:

Tabelle 11.2 2B2T-Codierung PST

Eingangssignalgruppe	Gleichspannungsanteil < 0	Gleichspannungsanteil > 0
00	-+	-+
01	=+	=-
10	+=	-=
11	+-	+-

Die Form der Codierung (am Sender!) unterscheidet also zwischen einem Gleichspannungsanteil < 0 und >0, so daß entsprechende Codes ausgewählt werden können, um den Gleichspannungsanteil nahezu bei 0 zu halten. Die Detektierung des jeweiligen Gleichspannungsanteils kann durch einfaches Mitzählen der ‘01’- und ‘10’-Eingangsgruppen bei den jeweiligen Zuständen.

11.4.6.3 4 Binary, 3 Ternary (4B3T)

Die 4B3T-Codierung, der häufig bei digitalen Fernmeldeanlagen eingesetzt wird, werden insgesamt 16 Eingangsgruppen (4B) auf 27 mögliche Ausgangswerte (3T) umgesetzt. Die dabei entstehenden redundaten Signalgruppen werden wiederum für den Ausgleich des Gleichspannungsanteils zu 0 genutzt, können aber auch bestimmte Fehlerinformationen enthalten. Der 4B3T-Code wird nach gemäß Tabelle 11.3 codiert.

Die Vorteile einer solchen Codierung liegen im Übrigen nicht nur in der Reduzierung des Gleichspannungsanteils auf 0 und der Möglichkeit zur Taktrückgewin-

nung, sondern auch in der Reduzierung der maximalen Übertragungsrate um 25% (3 Zustände anstatt 4). Weitere Vorteile liegen in speziellen adaptiven Filteralgorithmen wie Echounterdrückung, die ebenfalls auf dem 3T-Code rechnen können. Für die Datenwelt mit ihrer Zweiwertigkeit ergeben sich aber Gewöhnungsprobleme, da beispielweise ICs, die die Codierung/Decodierung übernehmen sollen, dreiwertig ausgelegt sein müssen!

Tabelle 11.3 4b3T-Codierung

Eingangscode	Gleichsp. < 0	Gleichsp. > 0	Eingangscode	Gleichsp. < 0	Gleichsp. > 0
0000	=-+	=-+	1000	=+-	=+-
0001	-+=	-+=	1001	+-=	+-=
0010	-=+	-=+	1010	+=-	+=-
0011	+-+	-+-	1011	+==	-==
0100	=++	=--	1100	+=+	-=-
0101	=+=	=-=	1101	++=	--=
0110	==+	==-	1110	++-	--+
0111	-++	+--	1111	+++	---

Auch beim 4B3T-Code kann der momentane Gleichspannungsanteil durch Mitzählen der ‘ungleichgewichtigen’ Signalzustände mit entsprechender Wertigkeit (0 bis 3) bestimmt werden.

11.4.6.4 AMI-Codierung

Die im Abschnitt 4.4.10 vorgestellte AMI-Codierung erfüllt bereits die Bedingung, im Durchschnitt keinen Gleichspannungsanteil zu liefern und somit für Übertragersysteme geeignet zu sein. Die zweite wichtige Forderung der Taktrückgewinnung kann aber mit diesem einfachen Verfahren nicht erfüllt werden. Hierzu wird ein *Compatible High Density Binary* genannter Code (CHDBn) eingeführt, der nach einer Folge von n (>1) ‘0’-Bits die n+1-’0’ als ‘1’ mit verkehrtem, d.h. nicht alterniertem Signalpegel aussendet. Der Code wird dabei recht komplex, da z.B. für n = 3 eine Wertetabelle mit Gruppen zu 5 Bits und zusätzlichem Zustand (der vorigen ‘1’-Codierung) auszufüllen ist. Aus diesem Grund sei es bei der verbalen Beschreibung belassen.

Erschwerend kommt beim CHDBn-Code hinzu, dass eine Folge von sehr vielen ‘0’-Bits der Gleichspannungsanteil wieder (zeitlich befristet, aber in der Übertragung sehr wohl bemerkbar) sehr groß werden kann, da die alternierende Regel ständig verletzt wird. Daher wird, falls eine mehrfache einfache Verletzung vorliegen würde, anstelle der einfachen Codierung des vierten Bits (bei CHDB3) die gesamte Gruppe der 4 ‘0’-Bits als ‘+==+’ oder ‘-==-’ codiert, ansonsten alternierend zu den ‘1’-Codierungen. Die Eindeutigkeit der Zuordnung ist dabei durch die Verletzung der Alternierungsregel nicht nach n, sondern nach n-1-Bits gegeben.

Folgendes Beispiel möge eine gewisse Klarheit zur Codierung geben:

Tabelle 11.4 Vergleich AMI und CHDB3-Codierung

Zeichen	0101	1100	0010	1101	0000	0000	1000	0000	1011	0101
AMI	=+=-	+-==	==+=	-+=-	====	====	+===	====	-=+-	=+=-
CHDB3	=+=-	+-==	=-+=	-+=-	===-	===-	+===	+===	-=+-	=+=-
	=+=-	+-==	=-+=	-+=-	+==+	-==-	+-==	-===	+=-+	=-=+

11.4.6.5 Der Code HDB3 als Beispiel für eine Statemachine

Eine andere Variante, mit HDB3 (High Density Bipolarcode of Order 3) bezeichnet, bildet die Grundlage für die ISDN-Übertragungssysteme bei 2,048 Mbit/s, 8,448 Mbit/s und 34,368 Mbit/s (G.703). In dieser Variante, wiederum auf AMI basierend, werden die 0-Bits wie folgt codiert:

- Vier aufeinanderfolgende ‘0’-Bits der Binärfolge werden bei
- gerader Anzahl von ‘1’ seit der letzten Verletzung durch ‘100V’ ersetzt,
- ungerader Anzahl von ‘1’ seit der letzten Verletzung durch ‘000V’ ersetzt.

‘V’ bedeutet dabei die Verletzung der AMI-Regel, jeden von Null abweichenden Pegel zu alternieren; auf diese Weise kann die Vierergruppe von Nullen entdeckt werden. Treten nun mehrere Vierergruppen hintereinander auf, dann wird jede nach obiger Regel codiert, unabhängig, wieviele Vorgängergruppen existieren, wobei die ‘1’ und ‘V’-Pegel von Gruppe zu Gruppe alternieren; dies ergibt sich aus der Anzahl Null von ‘1’ zwischen diesen beiden Gruppen, die Null gilt als gerade Zahl. Beispiel: ‘0000 0000’ wird bei gerader Anzahl von ‘1’ vorher zu ‘100V 100V’ codiert.

Zur Codierung des HDB3-Codes müssen jeweils 4 Bits (B0 .. B3) in ein Schieberegister eingelesen und dann drei Fälle unterschieden werden:

1. Bit B0 ist eine ‘1’: Es folgt eine normale AMI-Codierung, in das Schieberegister wird das nächste Bit eingelesen, B0 wird herausgeschoben. Dieser Fall wird mit ‘H’ codiert
2. Bit B0 ist eine ‘0’, jedoch sind nicht alle Bits B1 bis B3 ‘0’. Jetzt wird eine normale ‘0’ ausgegeben, das nächste Bit eingelesen. Dieser Fall wird mit ‘L’ beschrieben.
3. Die Bits B0 .. B3 sind alle ‘0’, so daß der Ersetzungsfall auftritt. Jetzt wird ‘100V’ oder ‘000V’ auf die Leitung gegeben, je nach Anzahl der ‘1’ vorher. Dieser Fall sei mit ‘E’ codiert.

H, L und E können mit Hilfe eines Schieberegisters mit anschließender Logik – also einem Vor-Automaten – sehr einfach dargestellt werden. In PLD-Assemblernotation (einer Assemblersprache für Boolesche Algebra) lautet dies:

H = B0;

L = /B0 * B1 + /B0 * B2 + /B0 * B3;

E = /B0 * /B1 * /B2 * /B3;

{H, L, E} bilden nun das Eingangsalphabet eines (getakteten) Moore-Automaten, das bezüglich der technischen Realisierung dann auf Binärwerte codiert wird: H := (0, 1), L := (0, 0), E := (1, -) (- bezeichnet „don't care"). Das Ausgangsalphabet sei mit {-, 0, +} definiert, so dass sich das in Bild 11.12 dargestellte Zustandsdiagramm ergibt.

Tabelle 11.5 Schaltwerktabelle für HDB3-Sendeautomat

Z	**b a**	**00**	**01**	**10**	**11**	
S		Z3/11	Z1/11	Z5/11	Z5/11	
Z1		Z2/01	Z4/01	Z6/01	Z6/01	
Z2		Z2/00	Z4/00	Z2/00	Z4/00	
Z3		Z3/00	Z1/00	Z3/00	Z1/00	
Z4		Z3/11	Z1/11	Z5/11	Z5/11	
Z5		Z7/01	Z7/01	Z7/01	Z7/01	
Z6		Z7/00	Z7/00	Z7/00	Z7/00	
Z7		Z8/00	Z8/00	Z8/00	Z8/00	
Z8		Z9/00	Z9/00	Z9/00	Z9/00	Z+/s v
Z9		Z12/01	Z10/01	Z14/01	Z14/01	
Z10		Z11/11	Z13/11	Z15/11	Z15/11	
Z11		Z11/00	Z13/00	Z11/00	Z13/00	
Z12		Z12/00	Z10/00	Z12/00	Z10/00	
Z13		Z12/01	Z10/01	Z14/01	Z14/01	
Z14		Z16/11	Z16/11	Z16/11	Z16/11	
Z15		Z16/00	Z16/00	Z16/00	Z16/00	
Z16		Z17/00	Z17/00	Z17/00	Z17/00	
Z17		Z18/00	Z18/00	Z18/00	Z18/00	
Z18		Z3/11	Z1/11	Z5/11	Z5/11	

Dieses Zustandsdiagramm kann nun leicht in eine textuelle oder tabellarische Form umgesetzt werden. In Tabelle 11.5 ist die so genannte Schaltwerktabelle eingetragen, die zwei Tabellen enthält: Die Zustandsfolgetabelle und die Ausgangstabelle. Hierbei ist zu beachten, dass die Ausgangstabelle sich auf den aktuellen Zeitpunkt bezieht, während die Zustandsfolgetabelle den zukünftigen Zustand, also denjenigen, der bei einem Speicherereignis gespeichert wird, enthält.

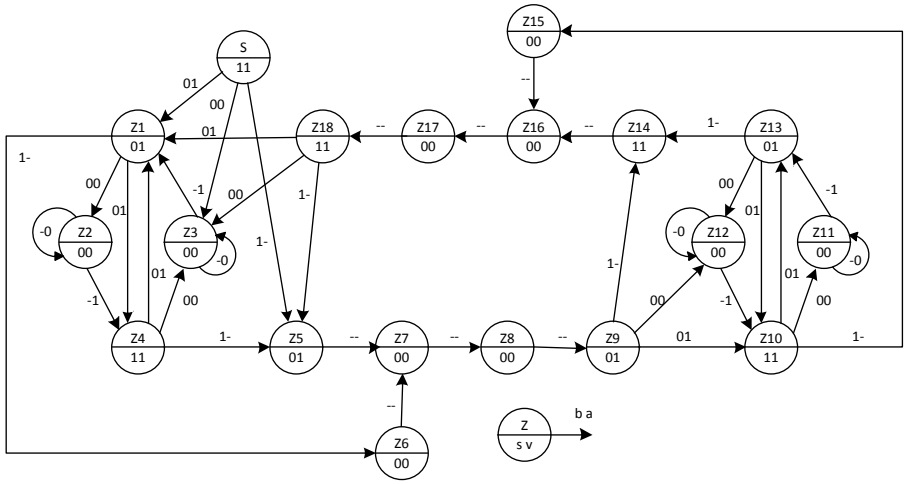


Bild 11.12 Zustandsdiagramm für HDB3, Sender

11.5 Medienzugangskontrolle (Layer 2a)

Für exklusive Leitungen zwischen zwei Kommunikationsteilnehmern ist ein Medienzugang sehr einfach, da das Medium vollständig in die Hoheit der beiden Teilnehmer fällt. In einer Vollduplex-Übertragung hat somit jeder eine komplette Sendeleitung exklusiv.

Die Situation ändert sich komplett, wenn ein Medium – z.B. aus Aufwandsgründen (Kanalauslastung, Kabelverlegung) oder aus technischen Gründen wie drahtlose Übertragung – nicht mehr exklusiv für einen Sender zur Verfügung steht. In diesem Fall muss der Zugang zum Medium geregelt werden, damit gegenseitige Störungen ausgeschlossen oder zumindest reduziert werden.

Übertragungsmedien wie das Vakuum oder Leitungen in Local Area Networks sind prinzipiell so ausgelegt, dass viele Teilnehmer – auch gleichzeitig – darauf schreibend zugreifen können. Mit dem Aufkommen dieser vernetzungen wurde dann auch die Einführung einer zusätzlichen Schicht im ISO/OSI-Layermodell notwendig, da zunächst nicht an nicht-exklusive Medien gedacht worden war.

Das Prinzip der Medienzugangskontrolle funktioniert insbesondere dann hervorragend, wenn die Laststruktur des einzelnen Teilnehmers so ausgelegt ist, dass Zugriffe nur selten vorkommen und nur kurze Zeit in Anspruch nehmen. Letzteres lässt sich wiederum erreichen, wenn hohe Datenübertragungsraten eingesetzt werden. Die erste Bedingung ist dagegen prinzipiell von der Übertragungsrate unabhängig und kann bei deutlicher Verletzung letztendlich zu einer Blockade des gesamten Netzes führen.

Die Behebung und/oder Vermeidung von Zugriffskonflikten geschieht über sehr unterschiedliche Methoden zur Zugriffskontrolle. Diese Unterschiede in der Methodik haben folgende Ursachen:

- Die typische Kommunikationslänge (Nutzerdaten) kann sehr unterschiedlich sein: Der Dialogverkehr benötigt viele, kurze Zugriffe, der Datenverkehr wie etwa Dateitransfer wenige, lange.
- Die Übertragungsverzögerung muss eventuell in eng gesteckten Grenzen gehalten werden (Optimierung in Richtung Echtzeitfähigkeit).
- Die (Netto-)Übertragungsrate soll maximiert werden (Auslastung).
- Die Fehlersicherheit der Übertragung ist die wichtigste Anforderung
- Die Signallaufzeiten sind – insbesondere bei Satellitenkommunikation – so lang, daß diese Zeit in dem Zugriffsverfahren zu berücksichtigen ist.
- Die maximale Zeit zur Übertragung einer Nachricht soll deterministisch bestimmt werden können (Echtzeitfähigkeit).

Um diesen vielen Anforderungen gerecht werden zu können, müssen mehrere, voneinander klar unterscheidbare Zugriffsverfahren definiert werden. Dies geschieht innerhalb der Schicht 2 des ISO/OSI-Layermodells, die aus diesem Grund – wie bereits angedeutet – in zwei Unterschichten unterteilt wird:

Die ‘tiefere’ Schicht wie als Medienzugriffskontrolle, engl. *Media Access Control*, MAC bezeichnet; diese Verwaltungsschicht regelt den Zugriff auf das Medium bei Mehrfachzugriffen, wobei die Zuteilungsstrategien im folgenden besprochen werden.

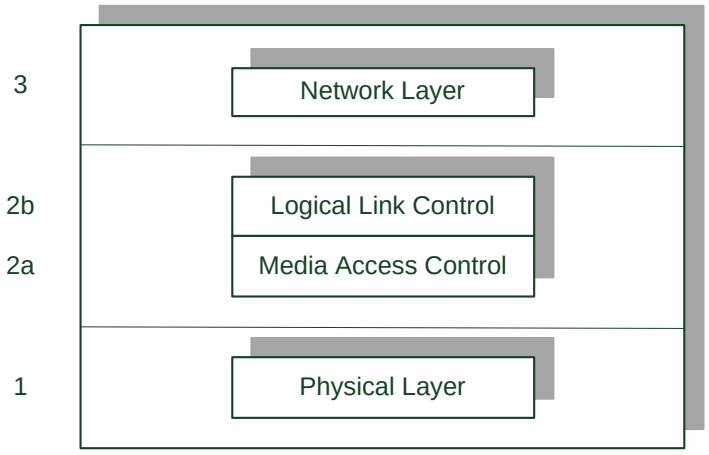


Bild 11.12 Mehrfachzugriffskontrolle im 7-Schichtenmodell

Die darüberliegende Subschicht bildet die logische Verbindungskontrolle, engl. *Logical Link Control*, LLC. Innerhalb dieses Teils der Schicht 2 werden Adressierung, Fehlerkontrolle etc. behandelt, d.h., es werden Verwaltungsbits zu den Nutzerdatenbits hinzuaddiert, während die MAC-Subschicht nur den Zugriff regelt. Viele Hardwarecontroller für den Netzzugriff integrieren daher die Unterschicht 2a in die Schicht 1, während für 2b Softwarelösungen existieren.

Wie den Medienzugriffsstrategien werden zwei große Prinzipien unterschieden: Die *Zuteilungsstrategien* verteilen die Sendeberechtigung auf einem unteilbaren Medium nach Kriterien wie Sendebereitschaft, Prioritäten, zyklische Methoden etc. Sie benötigen eine verteilte oder konzentrierte Zuteilungsinstanz, damit Rechen- und Netzübertragungszeit, bieten andererseits jedoch eine Deterministik in der maximalen Übertragungszeit.

Die *Zufallsstrategien* lassen den einzelnen Stationen die prinzipielle Freiheit des Sendebeginns. Kommt es dabei zu einer Kollision, so muss die Sendung sofort nach Detektierung unterbrochen und zu einem anderen Zeitpunkt wieder aufgenommen werden. Diese Strategieklasse benötigt sehr wenig (zentrale) Rechenkapazität, das Netz ist gewöhnlich mit höheren Gesamtübertragungsraten belastbar, kann jedoch keine Maximalübertragungszeit garantieren und sich letztendlich selbst blockieren.

11.5.1 Zuteilungsstrategien

Die bereits erläuterte Klasse der Zuteilungsstrategien kann weiterhin unterteilt werden in

- *feste Zuteilungsverfahren*, bei denen ein Teilnehmer zu festen Zeiten senden kann (Zeitmultiplex)
- *variable Zuteilungsstrategien* mit zentraler oder dezentraler Kontrolle der Zugriffsberechtigung.

Feste Zuteilungsverfahren werden z.B. in synchronen Nachrichtennetzen wie ISDN benutzt. Hier werden in einem 2 MBit/s-Kanal 30 Zeitschlitze für je ein Byte zur Verfügung gestellt, wobei jedem Benutzer dann 64 kBit/s Datenrate zur Verfügung stehen.

Das Verfahren benötigt praktisch keinen Overhead, so dass de facto die gesamte Bandbreite für nützliche Datenkommunikation verwendet werden kann. Die Datenrate lässt sich obendrein in gewissen Grenzen dadurch variieren, so dass die Einteilung zwar erhalten bleibt, aber einzelnen Verbindungen, insbesondere wenn weniger als 30 bestehen, mehr Zeitschlitze zur Verfügung gestellt werden.

Die zweite, in den letzten Jahren mit hoher Priorität weiter entwickelte feste Zuteilungsstrategie besteht in echtzeitfähigen Netzen, insbesondere mit kleiner Reaktions- bzw. Übertragungszeit. Hier werden Zeitscheiben für die Netzteilnehmer definiert, in denen sie nicht nur senden dürfen, sondern vielmehr senden *müssen*. Die Echtzeitfähigkeit und sogar die Ausfallerkennung sind auf diese

Weise gegeben, und die Korrektheit der Zuteilung wird auf die Synchronisation der (internen) Zeiten der Netzteilnehmer abgebildet.

Fazit: Der wesentlichen Einsatzfall der festen Zuteilungsstrategie liegt bei Übertragungen, die auf eine enge Begrenzung der Verzögerungszeiten hin optimiert sind: Digitale Sprachübertragung, echtzeitfähige Netzwerke im eingebetteten Bereich.

Innerhalb der festen Zuteilungsstrategien kann man weiterhin zwischen zentraler und dezentraler Zuteilung unterscheiden. Die *zentrale Zuteilung* durch Polling nutzt einen Masterrechner am Netz, der allen anderen ein Nachrichtenpaket als Anfrage zusendet. Diese werden dort aufgenommen und erlauben den Slavestationen, ihrerseits ein Datenpaket zu senden. Soll dies nicht geschehen, wird ein sogenanntes *Go Ahead*-Paket gesendet: Der Master betreibt seine Abfrage mit der nächsten Station.

Die Nachteile eines solchen Verfahrens sind augenscheinlich: Die Abfrage benötigt eine Zeit proportional zu der Anzahl der angeschlossenen Stationen, falls eine feste Strategie im Netz verankert ist. Hinzu kommt der Kommunikationsoverhead, da zur Aussendung eines Datenpakets zwei Sendungen notwendig sind und auch 'Nicht-Antworten' abgefragt werden. Das Zuteilungsverfahren mit zentraler Zuteilung ist allerdings prinzipiell echtzeitfähig, da eine maximale Reaktionszeit im Netz garantiert werden kann.

Strategien zur Optimierung des Netzdurchsatzes erfordern variable Zuteilungen. Hierzu ist eine adaptive Anpassung der Abfragesequenz möglich.

Die *dezentralen Zuteilungsprotokolle* bieten in Rechnernetzen, die Datenverkehr im Burstmodus aufweisen als dies bei der digitalen Sprachübertragung der Fall ist, eine wesentlich bessere Anpassung des Netzdurchsatzes an die theoretische Grenze. Hier kann man zwei verschiedene Verfahren grob voneinander unterscheiden:

- Dezentrale Zuteilung durch Reservierung
- Dezentrale Zuteilung durch Sendeberechtigung (Token)

Die *dezentrale Zuteilung durch Reservierung* benötigt einen zeitlichen Sendeabschnitt, in dem die einzelnen sendewilligen Stationen einen freien Sendeabschnitt reservieren können. Dieser Sendeabschnitt umfasst beim *Bit-Map-Protokoll* exakt n Bits bei n angeschlossenen Stationen. In diese Reservierungsbits kann die k-te Station ein Reservierungsbit an die k-te Stelle schreiben und erhält dann nach einer festgelegten, allen Stationen bekannten Reihenfolge einen Zeitschlitz zum Senden. Das Protokoll hat dann in etwa den Ablauf aus Bild 11.13.

Die Voraussetzungen für eine derartige Zuteilung bestehen in der Definiertheit der Reservierungszeitpunkte sowie in einer festen Konfiguration des Netzes. Ist das System nur gering belastet, werden praktisch nur Konkurrenzschlitze gesendet. Will die k-te Station nunmehr senden, so markiert sie das Bit k, muss aber mindestens die restlichen Bits des Reservierungsschlitzes noch abwarten, ehe sie senden darf. Die Wartezeit beträgt demnach

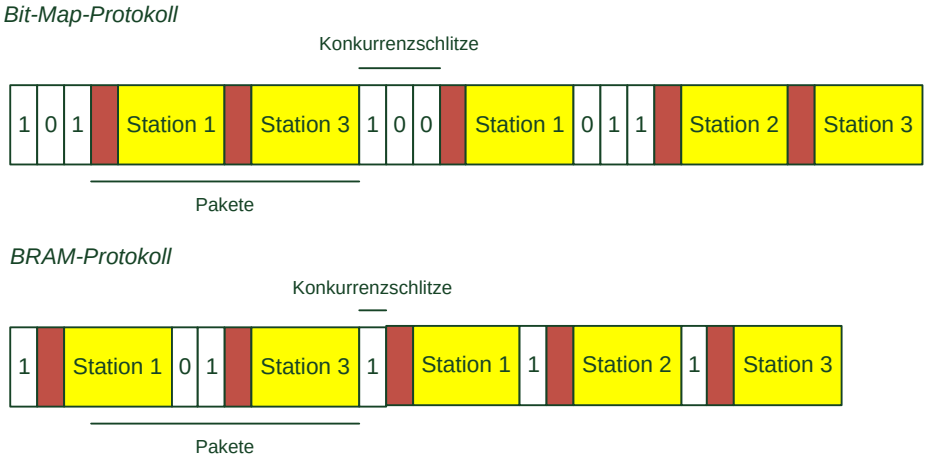


Bild 11.13 Dezentrale Zuteilungsprotokolle mit Reservierung

$$s_{nm} = \begin{cases} N - m & falls\ n < m \\ 2 \bullet N - m & falls\ n \geq m \end{cases}$$

so dass die mittlere Wartezeit ohne Datenübertragung

$$W_n = \sum_{m=1}^{N} p \bullet s_{nm} = p \bullet \left(\sum_{m=1}^{n-1} s_{nm} + \sum_{m=n}^{N} s_{nm} \right)$$

beträgt.

In der in Bild 11.13 bereits gezeigten Variation der Zuteilung, dem sogenannten ***BRAM-Protokoll*** (broadcast recognition access mechanism) wird die Reservierungsphase sofort unterbrochen, wenn eine Station die Sendeberechtigung erhalten will. Dadurch verringert sich die mittlere Wartezeit, wobei nun anscheinend die Stationen mit kleiner Nummer stark bevorzugt werden. Dies wird dadurch ausgeschaltet, indem die Reservierungsphase nach einer Sendung mit der nächstfolgenden Station weiterläuft, so dass keine Station mehr bevorzugt wird. Die mittlere Wartezeit bei geringer Belastung beträgt nunmehr

$$W_{BRAM} = 0{,}5 \bullet N + 0{,}5$$

und ist von n unabhängig. Abb. 8-2 zeigt ein Beispiel mit 3 Teilnehmern.

Das häufiger genutzte Verfahren zur Erteilung einer Sendeberechtigung verwendet Marken (*Token*). Der Inhaber dieser Marke – im Netz kann es eine oder mehrere solcher Marken geben – ist berechtigt, ein Datenpaket auf dem Netz zu senden, alle anderen Stationen dürfen keine Sendung tätigen. Nach Abschluss des Datenpakets

generiert die bisher sendeberechtigte Station eine neue Sendeberechtigung und gibt diese an die nächste Station weiter.

Wichtige Voraussetzung für die Sendezuteilung mittels Token ist die logische Strukturierung des Netzes als Ring, zumeist sogar unidirektional. Ferner muss eine ausgezeichnete Station eine Monitorfunktion übernehmen, die dafür sorgt, daß beispielsweise verlorengegangene Token wieder generiert werden usw.. Für die Sendezuteilung mittels eines Tokens existieren zwei Normen mit unterschiedlicher Struktur der Verbindung:

Im *Token-Bus-Verfahren* (IEEE 802.4) ist das Netz zwar physikalisch busförmig ausgeführt, logisch jedoch ein Ring. Dies bedeutet ein aufwendiges Verfahren zur Initialisierung, zum Einfügen von Stationen und zur Fehlerbehebung, bietet andererseits jedoch auch die Möglichkeit, den logischen Ring in eine komplexere Struktur zu überführen und so Prioritäten einzufügen. Bild 11.14 zeigt eine physikalische Bus- und logische Ringstruktur des Token-Bus.

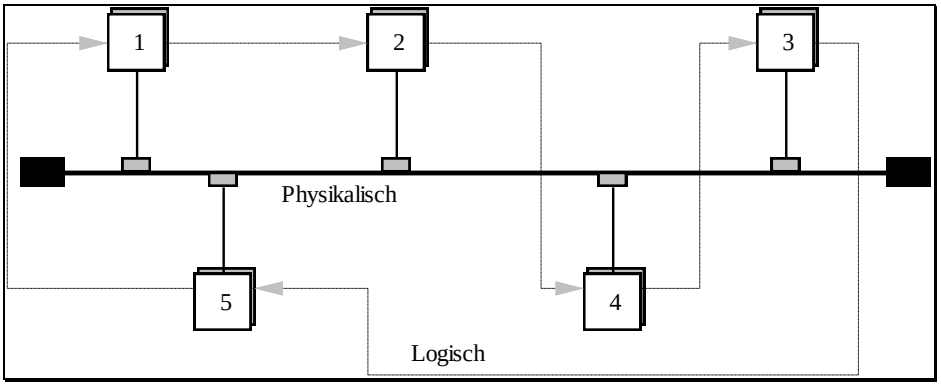


Bild 11.14 Struktur am Token Bus

Beim ***Token-Ring-Verfahren*** (IEEE 802.5) wird auch die physikalische Struktur als Ring ausgeführt, die im Prinzip auf miteinander verbundenen Punkt-zu-Punkt-Verbindungen besteht. Zwischen zwei Stationen wird in festgelegter Richtung der Freitoken übertragen, und es muss durch technische Maßnahmen gewährleistet werden, dass weder die Laufzeit drastisch gesteigert wird (etwa durch Zwischenspeichern der kompletten Sendung und erneutes Aussenden) noch die im Ring gespeicherte Bitmenge zu klein ist (für die Übertragung des Freitokens, z.B. 24 Bit beim Token Ring). Dies erfolgt durch Zwischenspeichern, aber sofortiges Weitersenden in den Knoten.

Die Sendezeit pro Freitoken wird auf z.B. 10 ms, durch das Netzwerkmanagement parametrisierbar, eingestellt. Im Anschluss muss die Sendestation diesen weitergeben, teilweise (4-Mbit/s-Version) nach Warten auf das Freiwerden des Rings, teilweise sofort (Early Token Release).

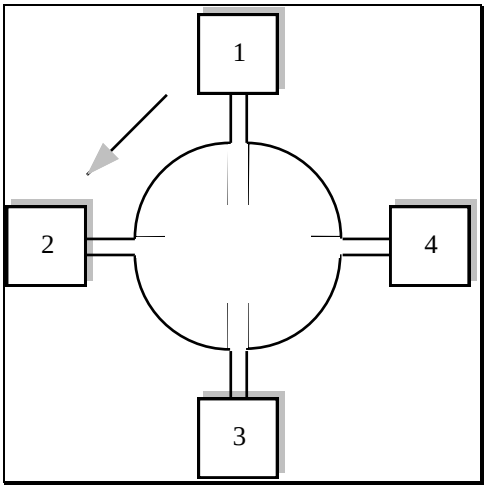


Bild 11.15 Token Ring Struktur

Eine der wichtigsten Probleme beim Token Ring ist die Fehlererkennung und –behebung insbesondere von Ringunterbrechungen. Diese Unterbrechungen erfolgen zumeist durch Rechnerausfall, selten durch Kabelausfall, und können durch einen Bypass behoben werden. Hierzu ist die Einrichtung eines Verkabelungszentrums notwendig, in dem die entsprechenden Schaltungen ausführbar sind.

11.5.2 Zufallsstrategien

Die Zufallsstrategien setzen an den Punkt an, wo eine Sendung beginnen soll. Der Sender versucht dann spontan, auf das Medium zuzugreifen, um die Daten unmittelbar zu transferieren. Der Zugriff wird natürlich sofort gestoppt, falls das Medium belegt ist (Kollisionsvermeidung).

Trotz dieser ersten Maßnahme kann es zu Datenkollisionen kommen, falls zwei Stationen zur gleichen Zeit ihre Sendung beginnen. Das Maß dieser Gleichzeitigkeit wird durch die räumliche Distanz der Stationen und die Signalgeschwindigkeit begrenzt. In diesem Fall muss die Kollision erkannt und durch geeignete Maßnahmen wieder behoben werden.

11.5.2.1 CSMA - Carrier Sense Media Access

Das Abhören eines Übertragungskanals zur Kollisionsvermeidung wird mit ***CSMA*** (Carrier Sense Media Access) bezeichnet. Dieses Verfahren kann in mehreren Untervarianten betrieben werden:

- Der (lesende) Zugriff auf das Übertragungsmedium kann zu jedem Zeitpunkt geschehen (reines CSMA) oder es kann nur zu bestimmten Zeitpunkten, namentlich zu Beginn von Zeitschlitzen, erfolgen (slotted CSMA). Slotted

CSMA-Netze haben einen bei Volllast gesehen höheren Ausnutzungsgrad, benötigen jedoch eine Zeitschlitzverteilung.

- Die Aussendung des Datenpakets erfolgt bei freiem Kanal mit der Wahrscheinlichkeit 1 (*nicht-persistentes CSMA*, persistent = beharrend), oder p (*p-persistentes CSMA*). Beim zweiten Verfahren wird mit der Wahrscheinlichkeit 1-p eine Zeit τ gewartet, die gerade so bemessen ist, dass ein Bit am Kanal entlanglaufen kann. Auf diese Weise wird ein vorher nicht-detektierter Sender bemerkt. Bei beiden Verfahren wird bei belegtem Kanal eine Zufallszeit T gewartet, bis der Sendewunsch erneut gestartet wird.

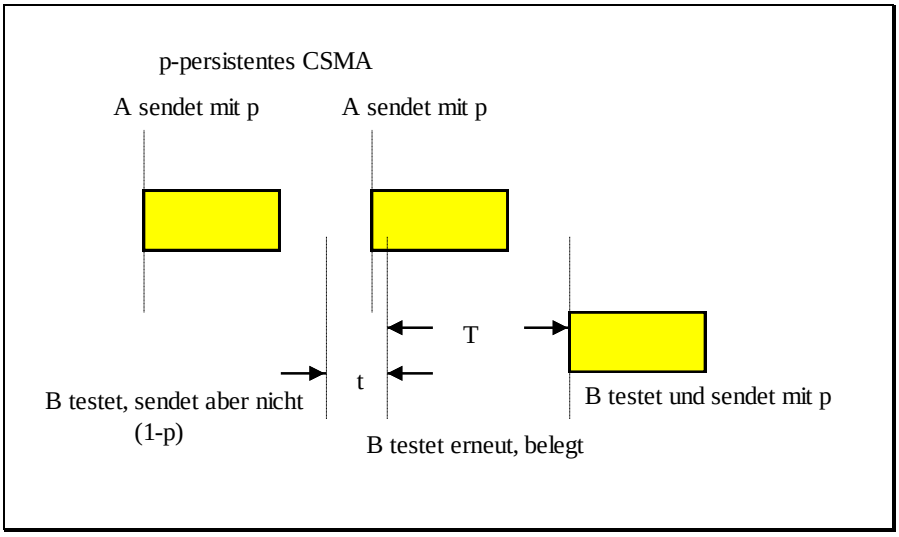


Bild 11.16 p-persistentes CSMA

Der Parameter p dient dabei der Durchsatzoptimierung.

11.5.2.2 CSMA/CD (CSMA with Collision Detection)

Beim Auftreten einer Kollision durch gleichzeitige Aussendung zweier Stationen wird im normalen CSMA-Verfahren das komplette Paket weitergesendet, und nur das Ausbleiben einer Quittung oder eine Negativ-Quittung ergibt eine Fehlerbehandlung durch Neusendung. Dieser Weg ist sehr zeitintensiv, so dass insbesondere bei stark belasteten Netzen nach anderen Wegen gesucht werden muss.

Die sendende Station muss zur Kollisionsdetektion lediglich ihr eigenes Signal am Netz lesen und mit dem gesendeten vergleichen. Entsteht hier eine Differenz, so liegt eine Kollision vor, die den Sender zum sofortigen Sendestopp mit einer anschließenden Zufallswartezeit veranlassen. Dies gilt in der Regel für beide Stationen (es gibt auch Abwandlungen hiervon!), so dass beide Sender sich zurück-

ziehen und neu übertragen. Hierdurch wird der Kanal nur für wesentlich kürzere Zeit mit unbrauchbaren Paketen belegt. Auch in diesem *CSMA/CD-Verfahren* (Carrier Sense Media Access with Collision Detection) unterscheidet man nicht-persistente und p-persistente CSMA/CD.

Die Detektierung der Kollision wird insbesondere durch die Erzeugung eines Störsignals (Jam-Sequence) auf dem Medium gesichert, das bei sicherer Detektierung von einer der beiden kollidierenden Stationen geschickt wird.

11.5.2.3 CSMA/CR (CSMA with Collision Resolution)

Verfahren, die nach dem Carrier-Sense Multiple Access with Collsion Resolution (CSMA/CR) Verfahren (auch als CSMA/CA, CSMA with Collison Avoidance, bezeichnet) arbeiten, nutzen hierfür eine Bitarbitrierung mit dominanten und rezessiven Bits aus. Beispielsweise wird die ‚0' als dominant, die ‚1' als rezessiv gewählt, so gewinnt bei der Kollision die ‚0', da die ‚1' gewissermaßen überstimmt wird.

Dies wird dahingehend ausgenutzt, dass bei einer gleichzeitigen Aussendung zweier Datenpakete an einem Medium nur dasjenige weitergesendet wird, das führend die meisten ‚0'-Werte besitzt, so dass es die erste Kollision gewinnt. Die zweite Transmission muss dann abgebrochen werden, weil das Rücklesen des eigenen Sendesignals einen abweichenden Wert geliefert hat. Diese Eigenschaft wird so ausgenutzt, dass zu Beginn eines jeden Pakets eine „Priorität" (in Form einer Sendeadresse) vorhanden ist.

Diese Form der Zuteilung hat einen wesentlichen Nachteil: Die Bits müssen sich, von beiden sendenden Stationen aus, auf dem Medium verteilen können, damit die Kollision noch während der Bitsendezeit erkannt wird. Bei einer Kabellänge von 30 m und einer Ausbreitungsgeschwindigkeit von $3 * 10^8$ m/s eine Ausbreitungsdauer von 100 ns, so dass für eine sichere Erkennung die Bitwechselfrequenz auf maximal 5 MHz begrenzt ist. Das Verfahren, z.B. bei CAN genutzt, ist damit auf kleiner Entfernung beschränkt.

Weiterhin ist mithilfe CSMA/CR keineswegs automatisch eine Echtzeitfähigkeit gegeben. Echtzeitfähigkeit ist nur die Priorität 0, die also führende Nullen bis zur Erkennungsgrenze (bei CAN: 11 oder 29 bit) besitzt, alle anderen müssen – bei entsprechenden Sendeaktivitäten der höheren Prioritäten – theoretisch beliebig lang warten.

11.5.3 Beispiele für Netzwerkstandards

Im den folgenden Abschnitten werden Beispiele für Netzwerke gegeben, die weit verbreitet sind, allerdings auch verschiedene Anwendungsgebiete abdecken. So gehört Ethernet zu den so genannten Local Area Networks (LAN), während I²C-Bus (Inter-IC-Bus) und das Serial Peripheral Interface zwei Standards darstellen, die auf der Ebene einer Halbleiterplatine arbeiten und zu den Controller Area Networks (CAN) gezählt werden. EIA-485 (ehemals RS-485) nimmt eine Zwi-

schenstellung ein, wird aber gerne in Feldbussystemen eingesetzt und zu den LANs gerechnet.

11.5.3.1 EIA-485

EIA steht als Abkürzung für *Electronic Industries Alliance*. EIA-485, vormals als RS-485 bezeichnet, verwendet ein Leitungspaar, um sowohl den invertierten und einen nichtinvertierten Pegel eines 1-Bit Datensignals zu übertragen. Am Empfänger wird aus der *Differenz* der beiden Spannungspegel das ursprüngliche Datensignal rekonstruiert. Das hat den Vorteil, dass sich Gleichtaktstörungen nicht auf die Übertragung auswirken und somit die Störsicherheit vergrößert wird. Im Gegensatz zu EIA-232 (vormals RS-232) sind so wesentlich längere Übertragungsstrecken und höhere Taktraten möglich.

EIA-485 wird aktuell von der TIA (Telecommunications Industry Association) gepflegt und als EIA/TIA-485 bezeichnet.

Bild 11.17 zeigt die Abhängigkeit von Reichweite und Bitfolgefrequenz bei EIA-485. Ein EIA485-Transmitter kann dabei 32 Empfänger – darunter sich selbst – mit elektrischer Leistung versorgen, um den Empfang zu gewährleisten. Bei dem Vorgängerstandard EIA-422, der ebenfalls differenziell arbeitet, sind 10 Empfänger möglich. Weiterhin kann der EIA-485 Standard im Multipoint-Modus, d.h. mit wechselnden Sendern betrieben werden, während EIA-422 im Point-to-Multipoint arbeitet.

Technische Einzelheiten

EIA-485 ist für eine zwei- und vieradrige Übertragung, also Halb- und Vollduplexbetrieb definiert. Meist wird jedoch nur ein Halbduplexbetrieb mit einem Adernpaar gewählt, diese Variante entspricht dann auch einem Netzwerkbetrieb ähnlich zu Ethernet.

Im Gegensatz zu anderen Bussen sind bei EIA-485 nur die elektrischen Schnittstellenbedingungen definiert. Das Protokoll kann anwendungsspezifisch gewählt werden, so dass EIA-485 allein als Standard keine Übertragung gewährleistet. Andererseits wurde dieser Standard als Grundlage für viele Feldbusstandards gewählt, folglich wird EIA-485 häufig eingesetzt.

Die beiden symmetrischen Leitungen der EIA-485/Schnittstelle arbeiten mit einem Differenz-Spannungspegel von mindestens +/-200 mV. Der Sender eines typischen 485-Bausteins verwendet eine Brückenschaltung, somit entspricht der Signalpegel beim Sender der Betriebsspannung des Treibers, z.B. +/- 5 V.

Wie bereits erwähnt sind mindestens 32 Teilnehmer an einem Adernpaar möglich; es existieren auch Bausteinvarianten mit 256 Teilnehmern, indem der Eingangswiderstand der Receiver auf den 8fachen Wert gesteigert wird. Bei einer maximalen Übertragungsrate von 10 Mbit/s sind 12 m Übertragungsreichweite möglich, bei 100 kbit/s maximal 1200 m.

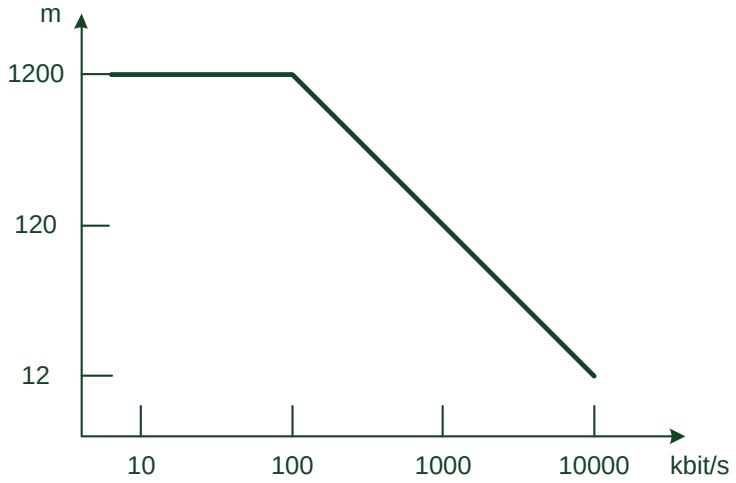


Bild 11.17 EIA-485 Reichweite/Geschwindigkeitsdiagramm

11.5.3.2 Ethernet

Der ursprüngliche Standard

Das ursprüngliche Ethernet (ca. 1980) verwendete das 1-persistente reine (nicht slotted) CSMA/CD-Verfahren, bei dem insbesondere Simulationen gezeigt hatten, dass es im Gesamtdurchsatz kaum zu übertreffen war. Der wesentliche Teil des Ethernet-Standards ist in IEEE 802.3 definiert. 1983 kamen Arbeiten um eine vereinfachte Verkableung (Cheapernet) hinzu.

Die Hardware von Ethernet:

- Koaxialkabel ('Yellow Cable') als Thick Wire bis 500 m, mit Repeater bis 2,5 km. Anschlüsse sind alle 2,5 m möglich und werden mittels eines Dorns im Kabel angebracht.
- Koaxialkabel als Thin Wire mit BNC-Buchsen oder selbstunterbrechenden Steckern (Cheapernet).

Die Codierung besteht im Bi Phase Mark (Manchester, → 11.4.5.5) mit physukalischen Signalwerten von 0 und -2V. Der als Kabelanschluss und Codierer dienende Transceiver enthält die Elektronik zum Senden, Empfangen und zur Kollisionserkennung. Prinzipiell können mehrere Stationen an einen Transceiver angeschlossen werden.

Bei einem erwünschten Sendezugriff wird der Kanal auf Aussendung abgetastet und bei Signalfreiheit noch mindestens 9,6 µs (96 Bits) gewartet; ist der Kanal belegt, wird zunächst auf die Unbelegheit gewartet. Nach Sendebeginn liest der Controller ständig am Medium mit. Bei Fehlererkennung wird noch 32 bis 48 Bits

weitergesendet, um mittels dieses Jam-Signals die Kollisionserkennung allen aktiven Teilnehmern mitzuteilen.

Die Wartezeit für eine erneute Aussendung ist in einem nicht-persistenten Algorithmus, dem BEB (Binary Exponential Backoff) festgelegt. Der Kanal wird dazu in Zeitschlitze der Länge 51,2 µs (512 Bits), innerhalb derer eine sendende Station auch beim größten Netz sicher erkannt wird eingeteilt. Jede sendebereite Station wählt sich genau einen Schlitz zum Senden aus. Tritt eine Kollision auf, wird die Anzahl der Schlitze verdoppelt, allerdings nicht über 10 Kollisionen hinweg, und ab der 16. Kollision wird der Prozess abgebrochen und der Misserfolg an höhere Schichten gemeldet. Ziel dieses Verfahrens ist die schnelle Bereinigung bei geringer Belastung und die sichere Kollisionsauflösung bei hohem Verkehrsaufkommen.

Die Ethernet-Pakete werden Frames genannt:

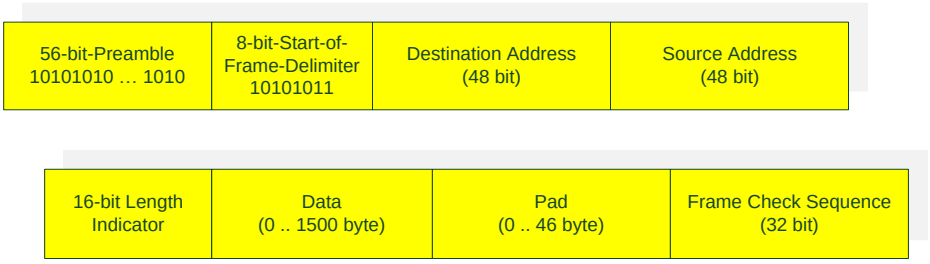


Bild 11.18 Rahmenformat Ethernet I

Die in diesem Rahmenformat angegebene 32-Bit-Prüfsumme (Frame Check Sequence) wird durch das Prüfpolynom für die CRC32 (→ 14.3.2.3)

$$CRC32 = x^{32} + x^{28} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

gebildet.

Weiterhin ist ein Pad-Feld angegeben. Dieses dient dazu, die Mindestpaketlänge (ohne die ersten 8 Bytes, also die Präambel und den Start-of-Frame-Delimiter) auf 64 anzuheben, um Kollisionen defintiv erkennen zu können.

Der Standard Ethernet II

Zwei wesentliche Änderungen wurden im Ethernet eingesetzt, um Ansprüchen aus der Praxis gerecht zu werden:

- Die Kollisiondomänen, d.h., die Bereiche, in denen eine Kollision auftreten kann, wurden systematisch verkleinert. Im geswitchten Ethernet, das im Vollduplex-Betrieb auf exklusiven Leitungen zwischen dem Switch und dem Endteilnehmer basiert, existieren keine Kollisionen auf Leitungsebene mehr.

- Das Rahmenformat wird geändert, um mehr Informationen darin unterzubringen.

Das Ethernet per exklusiver Verdrahtung zu einem Switch stellt aktuell den Standard dar. Hierdurch wird verhindert, dass es auf den Leitungen zu Kollisionen kommt, mit zwei Konsequenzen:

- Die Echtzeitfähigkeit eines Ethernet Typ II kann nunmehr dadurch gewonnen werden, indem die Switches selbst echtzeitfähig arbeiten und der Netzverkehr entsprechend eingeschränkt wird. In einem solchen Netz kann eine maximale Übertragungszeit, ggf. prioritätsgesteuert, garantiert werden.
- Ethernet II benötigt eine Flusskontrolle, da aufgrund der exklusiven Zuordnung der Leitungen Switches oder Empfänger überflutet werden können. Bezüglich des eben genannten Punktes *Echtzeitfähigkeit* ist diese Flusskontrolle unumgänglich, d.h., die echtzeitfähigen Switches, die insbesondere im Bereich der Automatisierungstechnik eingesetzt werden, müssen diese Funktionalität in jedem Fall unterstützen.

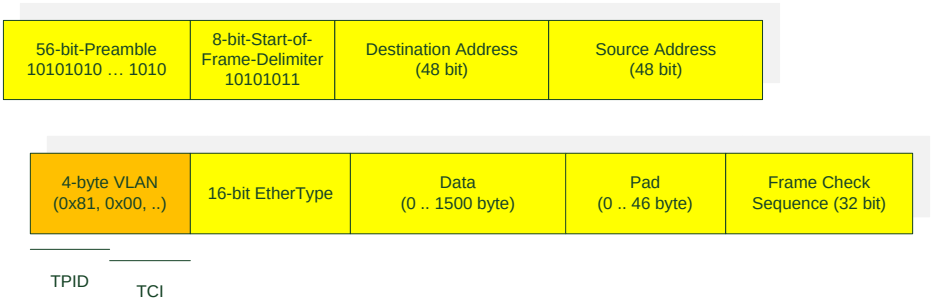


Bild 11.18 Aufbau eines Pakets Ethernet Typ II, hier mit VLAN-Tag

Die Präambel dieses Datenpakets besteht wie im Standard Ethernet I aus einer sieben Byte langen, alternierenden Bitfolge „10101010“, auf diese folgt der Start Frame Delimiter (SFD) mit der Bitfolge „10101011“ (Bemerkung: Im Ethernet wird das least significant bit – lsb – zuerst gesendet). Diese Sequenz hat historische Gründe, sie diente einst der Bit-Synchronisation der Netzwerkgeräte und war für all jene Geräteverbindungen notwendig, die die Bit-Synchronisation nicht durch die Übertragung einer kontinuierlichen Trägerwelle auch in Ruhezeiten aufrechterhalten konnten, sondern diese mit jedem gesendeten Frame wieder neu aufbauen mussten.

Die Bus-Netzwerkarchitekturen, die auf derartige Einschwingvorgänge angewiesen sind, werden heute kaum mehr verwendet, wodurch sich die Präambel, genauso wie das Zugriffsmuster CSMA/CD, die minimale und maximale Frame-Länge und der minimale Paketabstand IFG, nur aus Kompatibilitätsgründen in der Spezifikation befinden.

Das Übertragen der Ziel- und der Quell-Adresse (MAC-Adressen, je 6 Byte) entspricht dem Ethernet-I-Standard. Im Anschluss daran käme das Längenfeld, das die Länge des Pakets ohne den Header (Ziel- und Quell-MAC, Längenfeld selbst) und ohne die abschließende CRC beinhaltet. Werte von 46 (Mindestlänge) bis 1500 können hier eingetragen sein.

Ist die Länge > 1500, wird diese Zahl im Ethernet-II-Standard als Typfeld interpretiert, und die Länge wird durch die exakte Laufzeit bitgenau bestimmt. Einige Ethertypes sind vordefiniert, beispielsweise 0x0800 für ein IP-Paket, die undefinierten können für eigene Zwecke genutzt werden.

Steht in diesem Feld jedoch 0x8100, dann ist das das Kennzeichen für VLAN (TPID, Tag Packet Identifier). Die folgenden beiden Bytes stellen dann den Tag Control Identifier (TCI) dar, in dem verschiedene Informationen zusammengefasst sind (3 bit Priorität, 1 bit Kompatibilität, 12 bit VLAN identifier). Erst dann wird das Paket in gewohnter Weise, wieder mit dem Ethertype-Feld beginnend, wietergeführt. Das Paket wird in diesem Fall auf 1522 Bytes vergrößert, man spricht von einem Multi-Tagged-Paket.

Die Leitungscodierung im Ethernet II Standard erfolgt im 4B/5B-Verfahren (→ 11.4.6.1).

Multicasting im Ethernet

Innerhalb eines Netzes nach Ethernet ist es oftmals sehr hilfreich, entweder eine Broadcast- oder eine Multicast-Kommunikation (z.B. im Ethernet Powerlink, → 11.6.2) zu ermöglichen. Hierzu kann die Ethernet-Broadcastadresse (0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF) gewählt werden, oder für die Teilnehmern oder eine Untergruppe wird eine Multicast-Adresse aus den MAC-Adressen gebildet.

Eine Multicast-Adresse ist dadurch kenntlich gemacht, dass das Bit 0 des ersten Bytes auf ’1‘ gesetzt ist (Unicast: ’0‘). Um die Zugehörigkeit eines Ethernetknotens zu einer Multicast-(Ziel-)Adresse zu bestimmen, kann in den Ethernet-Controllern eine Hash-Summe geladen sein, aus der sich dann die Zugehörigkeit verifizieren oder falsifizieren lässt.

11.5.3.3 I²C-Bus

Der I²C-Bus, abkürzend für Inter-IC-Bus stehend, wurde in den frühen 1980er Jahren zur Kommunikation zwischen einem Mikrocontroller und Peripherie-ICs von Philips Semiconductor eingeführt. Es stellt ein typisches Bussystem auf Platinenebene dar. Technisch identisch ist das TWI (Two-Wire Interface).

Technische Beschreibung des I²C

Vom I²C-Bus liegen verschiedene Versionen und Geschwindigkeiten vor. Ursprünglich auf 100 kbit/s (als Brutto-Datenrate, *Normal Mode*) begrenzt wurde mit der Version 1.0 (1992) die maximale Geschwindigkeit auf 400 kbit/s (*Fast Mode*) erhöht. Version 2.0 (1998) erhöhte auf 3,4 Mbit/s (*High Speed Mode*), allerdings

mit verändertem Protokoll, V 3.0 (2007) ergänzte um den *Fast Mode Plus* mit 1 Mbit/s bei gegenüber dem normal und fast mode identischem Protokoll.

Zugleich wurden die Adressierungsräume erweitert. Der ursprünglichen Adressierung von 7 bit (= 128 Adressen, von denen 112 nutzbar waren) wurde eine 10-bit-Adressierung ab dem Fast Mode hinzugefügt, so dass maximal 1136 Teilnehmer (= 112 im 7-bit Modus plus 1024 im 10-bit Modus) angesprochen werden können.

I²C ist als *Master-Slave-Bus* konzipiert, d.h. zu einem Zeitpunkt hat nur der Master das Recht, die anderen Teilnehmer anzusprechen. Der Master sendet, und ein Slave reagiert darauf. Mehrere Master sind möglich (Multimaster-Mode). Die Buszuteilung (so genannte Arbitrierung), d.h. die Übergabe der Master-Eigenschaft, ist dabei per Spezifikation geregelt.

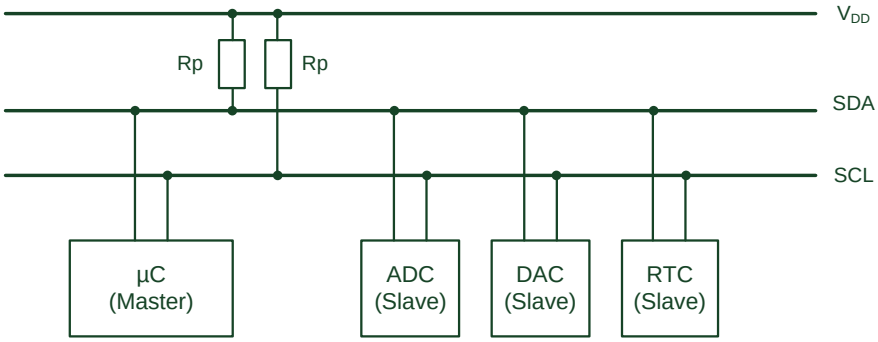


Bild 11.20 I²C-Bus mit einem Master (Mikrocontroller) und drei Slave-Teilnehmern (AD-Converter, DA-Converter, Real-Time Clock)

Der I²C-Bus benötigt lediglich zwei Signalleitungen: Den Takt (SCL, Serial Clock) und die Datenleitung (SDA, Serial Data). Beide sind im Ruhezustand mit den Pull-up-Widerständen Rp auf das Potential der Versorgungsspannung V_{DD} gezogen. Sämtliche am I²C-Bus angeschlossenen Geräte haben so genannte Open-Collector-Ausgänge, d.h., die Ausgangstransistorschaltung, die der bekannten Collector-Schaltung entspricht, weist keinen Collector-Widerstand auf (dieser ist durch Rp gegeben. Insgesamt entspricht die Schaltung einer Wired-AND-Schaltung: Zieht ein Ausgangstransistor eine Leitung auf Low, ist sie bei allen anderen Ein-/Ausgängen ebenfalls auf Low.

Der High-Pegel soll dabei mindestens $0{,}7 \times V_{DD}$ betragen, der Low-Pegel höchstens $0{,}3 \times V_{DD}$. Dieses Wired-AND-Prinzip bewirkt, dass ein nicht aktiver Ausgang (Transistor sperrt) als Eingang genutzt werden kann, indem die tatsächliche Spannung am Transistorausgang detektiert wird.

Übertragungsprotokoll

Der Bustakt wird immer vom Master ausgegeben. Für die verschiedenen Modi ist jeweils ein maximal erlaubter Bustakt vorgegeben, wobei in der Regel aber auch beliebig langsamere Taktraten verwendet werden können. Die verschiedenen Geschwindigkeiten wurden bereits dargestellt.

Wenn der Slave mehr Zeit benötigt, als durch den Takt des Masters vorgegeben ist, kann er zwischen der Übertragung einzelner Bytes den Clock auf low halten (*clock stretching*) und so den Master bremsen – allerdings nur auf Byte-Ebene, d.h. nach Übertragung eines Oktetts. Einzelbits sind nur gültig, wenn sich ihr logischer Pegel während einer Clock-High-Phase nicht ändert. Gewollte Ausnahmen davon sind das *Start*-, *Stop*- und *Repeated Start*-Signal. Das *Start*-Signal ist eine fallende Flanke auf SDA, während SCL high ist, das *Stop*-Signal ist eine steigende Flanke auf SDA, während SCL high ist. *Repeated Start* sieht genauso aus wie das *Start*-Signal.

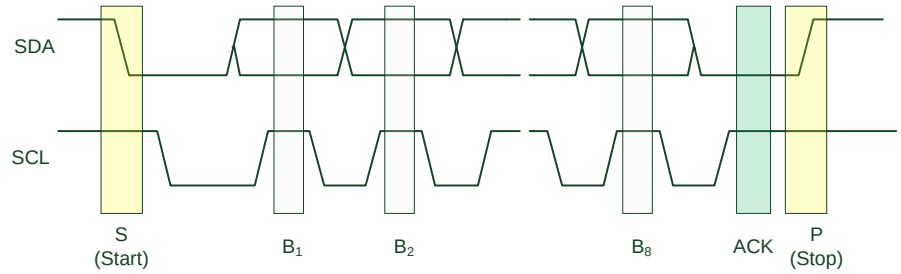


Bild 11.21 I²C-Bus Übertragungsprotokoll

Eine Dateneinheit besteht aus 8 Datenbits = 1 Oktett (welche protokollbedingt entweder als Wert oder als Adresse interpretiert werden) und einem ACK-Bit. Die Übertragung der Bits erfolgt immer in der Reihenfolge *Most Significant Bit First* (msb).

Das Bestätigungsbit (Acknowledge) wird durch einen Low-Pegel auf der Datenleitung von Seiten des Slaves während der neunten Takt-High-Phase (der Takt wird immer vom Master generiert wird) und als NACK (*not acknowledge*) durch einen High-Pegel signalisiert. Zugleich muss der Master auf der SDA-leitung ein High setzen, um das Wired-AND zu ermöglichen, und der Master liest dann diesen Wert ein.

Der Slave muss den L-Pegel an der Datenleitung anlegen bevor der Master das CLK-Signal auf High legt, andernfalls würden weitere eventuelle Teilnehmer ein „STOP-Signal“ lesen.

Adressierung und Übertragung

Eine Standard-I²C-Adresse ist das erste vom Master gesendete Byte, wobei die ersten sieben Bit als die eigentliche Adresse darstellen und das achte Bit (R/W-Bit) die Lese- oder Schreibrichtung festlegt (0 für schreiben, 1 für lesen). I²C nutzt

daher einen Adressraum von 7 Bit, was bis zu 128 Knoten auf einem Bus erlaubt (16 der 128 möglichen Adressen sind allerdings für Sonderzwecke reserviert).

Jeder I²C-fähige IC hat eine vom Hersteller festgelegte Adresse, von der bisweilen die untersten drei Bits (mit *Subadresse* bezeichnet) über drei Steuerpins festgelegt werden können. In diesem Fall können bis zu acht gleichartige ICs an einem I²C-Bus betrieben werden. Wenn nicht, müssen mehrere gleiche ICs mit getrennten I²C-Bussen angesteuert oder abgetrennt werden können.

Wegen Adressknappheit (!) wurde später eine 10-Bit-Adressierung eingeführt. Sie ist abwärtskompatibel zum 7-Bit-Standard durch Nutzung von 4 der 16 reservierten Adressen. Beide Adressierungsarten sind gleichzeitig verwendbar, was bis zu 1136 Knoten auf einem Bus erlaubt.

Der Beginn einer Übertragung wird mit dem *Start*-Signal vom Master angezeigt, dann folgt die Adresse. Diese wird durch das ACK-Bit vom entsprechenden Slave bestätigt. Abhängig vom R/W-Bit werden nun Daten Oktett-weise geschrieben (Daten an Slave) oder gelesen (Daten vom Slave), wobei der Takt immer vom Master gesetzt wird.

Das ACK beim Schreiben wird vom Slave gesendet und beim Lesen vom Master. Das letzte Byte eines Lesezugriffs wird vom Master mit einem NACK quittiert, um das Ende der Übertragung anzuzeigen. Eine Übertragung wird durch das *Stop*-Signal beendet. Alternativ kann auch ein *Repeated Start* am Beginn einer erneuten Übertragung gesendet werden, ohne die vorhergehende Übertragung mit einem *Stop*-Signal zu beenden.

> Anmerkung zum I²C-Bus: Dieses Kommunikationsmedium bietet eine hervorragende Möglichkeit, auf Platinenebene und –größe miteinander zu kommunizieren und einen Mikrocontroller mit weiteren Peripheriebausteinen zu versehen. Allerdings muss klar sein, dass I²C weder für große Entfernungen noch für große Übertragungssicherheit geeignet ist: Der Hauptvorteil von I²C liegt in der Effizienz dieses Standards.
>
> Im Multi-Mastermode, in dem also mehrere (potenzielle) Master am Bus aktiv sein können, wird der Bus anhand der ersten Bytes arbitriert, und zwar anhand einer CSMA/CR-Variante (→ 11.5.2.3) mit dominanter ‘0‘. Die Anzahl der Bytes/Oktetts, die für eine korrekte Adressierung und Befehlsübermittlung notwendig sind, muss daher für alle Kommunikationseinleitungen gleich sein, es sei denn, eine klare Unterscheidung ist bereits vorher möglich und der unterlegene Teilnehmer anhand der ersten Oktetts identifiziert.

11.5.3.4 Serial Peripheral Interface (SPI)

Neben dem I²C-Standard stellt das *Serial Peripheral Interface* (SPI) einen weiteren, weit verbreiteten Standard im Bereich der Vernetzung auf einer Platine dar. Dieser Standard stammt von der Firma Motorola, wird aber von vielen Herstellern angeboten.

SPI-Bus im Detail

Das Serial Peripheral Interface nutzt eine strenge Master/Slave-Kommunikation ohne Möglichkeit, den Master zu wechseln. Zur Kommunikation sind jeweils drei Leitungen notwendig, die die Teilnehmer gemeinsam nutzen:

- SCLK, Serial Clock zur Taktübermittlung (diese Leitung wird durch den Master gesetzt)
- MOSI, Master Out Slave In, zur Kommunikation in Richtung der Slaves, auch als SDO (Serial Data Out) bezeichnet.
- MISO, Master In Slave Out, zur Kommunikation in Richtung des Master, auch als SDI (Serial Data In) bezeichnet.

Zusätzlich müssen eine oder mehrere Selektierungsleitungen vorhanden sein, um den oder die Empfänger zu aktivieren. Diese Leitungen werden meist mit SS (Slave Select), CS (Chip Select), CE (Chip Enable) oder STE (Slave Transmit Enable) bezeichnet.

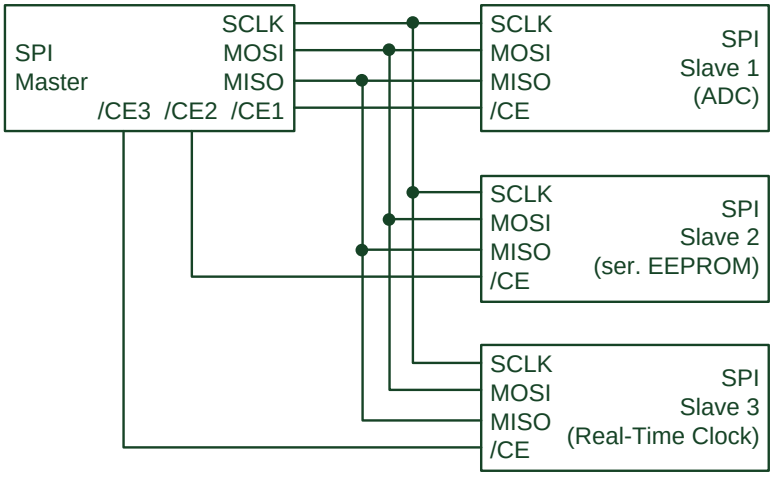


Bild 11.22a SPI in sternförmiger Verbindung

Die Kommunikation via SPI ist damit bidirektional (vollduplex). Bild 11.22a gibt die Verkabelung eines Masters mit drei Slaves, hier beispielhaft ein AD-Converter, ein serielles EEPROM und eine Real-Time Clock (RTC). Diese Form des Anschlusses wird meist gewählt, jeder Baustein ist separat ansprechbar, und in eingeschränktem Maße kann auch ein Multicasting erfolgen – wenn man auf die Antwort der Slaves verzichtet, da diese parallel zueinander, aber eben auf einer einzigen (Sammel-)Leitung gesendet werden.

Eine alternative Verdrahtung ist Bild 11.22b gezeigt. Hier werden die Teilnehmer in einer Kette miteinander verdrahtet, so dass die übertragenen Bits zunächst

zusammengestellt werden müssen und dann solange durch die Kette getaktet werden, bis sie ihr Ziel erreicht haben. Hierfür müssen alle Mitglieder in der Kette zugleich selektiert werden.

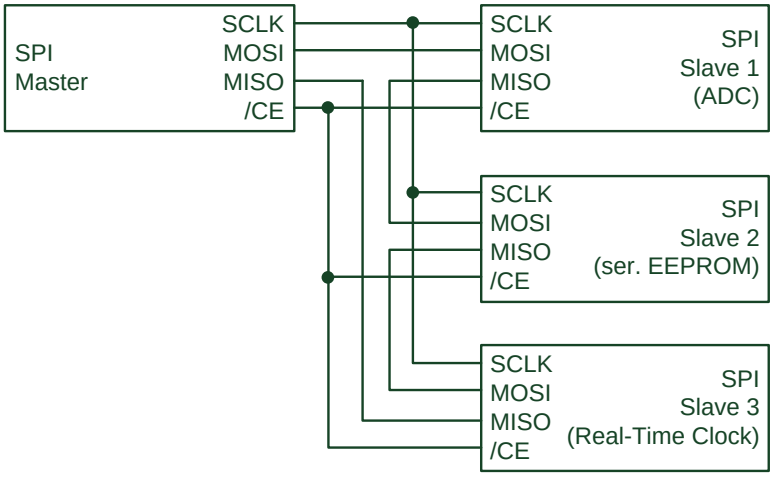


Bild 11.22b SPI in kaskadenförmiger Vernetzung

Protokollablauf

Es können theoretisch beliebig viele Teilnehmer an den Bus angeschlossen werden, wobei es exakt einen Master geben muss. Dieser Master erzeugt das Clock-Signal (SCK) legt fest, mit welchem Slave er kommunizieren will. Das geschieht über die Leitung „Slave Select“ (bzw. „Chip Enable“ etc.), die bei einem gegen Masse gezogenen Potenzial den jeweiligen Slave aktiviert.

Ein aktivierter Slave legt Daten im Takt des SCK-Anschlusses an MISO, so dass zugleich ein Byte empfangen und ausgesendet wird. Welche Daten das im Einzelfall sind, hängt von dem Baustein ab. Es kann sich im Allgemeinen nicht um eine unmittelbare Reaktion auf das aktuell übertragene Byte sein, meist ist es eine Reaktion auf die vorangegangenen Übertragungen. Einzelheiten hierzu müssen den jeweiligen Datenblättern entnommen werden.

In der Praxis haben sich für SPI vier verschiedene Modi durchgesetzt. Diese werden durch die Parameter Clock Polarität (CPOL) und Clock Phase (CPHA) festgelegt. Bei CPOL == 0 ist der Takt im Idle-Zustand Low, bei CPOL == 1 entsprechend High. CPHA gibt nun an, bei der wievielten Flanke die Daten übernommen werden sollen. Bei CPHA == 0 werden sie bei der ersten Flanke übernommen, nachdem SS/CE auf Low gezogen wurde, bei CPHA == 1 bei der zweiten. Bild 11.23 zeigt den Verlauf für CPOL/CPHA == 0/0 (Modus 0).

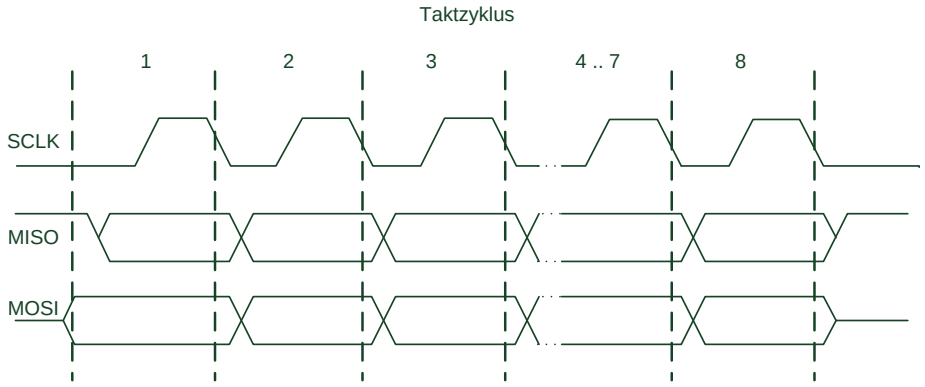


Bild 11.23 Zeitdiagramm SPI-Bus, Mode 0 (eine entsprechende Selektierungsleitung muss aktiv sein)

Zu beachten ist noch, dass der Slave bei CPHA == 0 seine Daten schon beim Aktivieren von SS/CE an MISO anlegt, damit der Master sie beim ersten Flankenwechsel übernehmen kann. Die verschiedenen Konstellationen für CPOL und CPHA werden auch als Modi bezeichnet:

Tabelle 11.6 Modi für Serial Peripheral Interface

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

11.6 Echtzeit-Netzwerke

Für die Definition eines echtzeitfähigen Netzwerks gilt analog zu den Definitionen 2.1 und 2.2, dass die maximale Übertragungsdauer von einem Knoten bis zum anderen deterministisch ist und somit auch bestimmt werden kann. Ein verteiltes, echtzeitfähiges System muss somit aus einem echtzeitfähigen Netzwerk und echtzeitfähigen, lokalen Teilsystemen bestehen. Im Folgenden sind einige Beispiele und Eigenschaften diskutiert.

11.6.1 Time-Triggered Protocol (TTP) und Byte Flight als Beispiele für echtzeitfähige Netzwerke

11.6.1.1 TTP/C

TTP/C (Time-Triggered Protocol Class C) stellt ein gutes Beispiel für ein Zeitgesteuertes Protokoll dar. Hier besitzen alle Knoten eine gemeinsame Zeit mit geringem Jitter. Dies wird durch spezielle Verteilung erreicht, z.B. durch IEEE-1588 (→ 11.6.3).

Jeder Knoten erhält nun die Tabelle mit den Sendezeiten – und zwar nicht nur seine eigenen, sondern auch die der anderen Knoten. Über diese Zeittabellen-gesteuerte Nachrichtensendung erhält jeder Knoten eine garantierte Sendemöglichkeit, außerdem können alle anderen Knoten die Betriebsfähigkeit des sendenden erkennen (und vor allem auch den Ausfall!).

Weitere Schutzmaßnahmen müssen in einem solchen Netzwerk getroffen werden, so ist z.B. der *Babbling Idiot* („quatschender Idiot“) auszuschließen. Hiermit wird ein Knoten bezeichnet, der am Netzwerk fehlerhaft ständig sendet und somit das gesamte Netzwerk stört. Die Sicherung gegenüber derartigen Störungen erfolgt mithilfe einer unabhängigen Kontrollinstanz am Knoten, und nur wenn der Knoten und die unabhängige Kontrollinstanz gleichzeitig senden wollen wird auch wirklich gesendet.

11.6.1.2 Byte Flight

Das *Byte Flight* Protokoll benötigt einen ausgezeichneten Sender, der über ein Zeitsignal eine gemeinsame Zeit verteilt. Diese gemeinsame Zeitbasis (Jitter: 100 ns) veranlasst die anderen Knoten nacheinander, Pakete zu senden oder ruhig zu bleiben. Dadurch wird es möglich, für eine begrenzte Anzahl von Sendungen einen exklusiven Zugriff zu gestatten.

Der Rest der Sendekapazität in einem Zeitschlitz wird nach dem CSMA/CR-Verfahren verteilt, sodass der Bus optimal ausgenutzt wird und zugleich (für eine begrenzte Anzahl von Daten) echtzeitfähig ist.

11.6.2 Ethernet Powerlink

Ethernet Powerlink arbeitet ein einem gemischten Polling- und Zeitscheibenmechanismus zur deterministischen Übertragung von Daten ein. Die zu erreichenden (und erreichten) Ziele sind:

- eine garantierte Übertragung von zeitkritischen Daten in sehr kurzen isochronen Zyklen mit konfigurierbarem Zeitverhalten
- eine zeitliche Synchronisation aller Netzwerkknoten mit sehr hoher Präzision im Sub-Mikrosekundenbereich

- eine Übertragung des weniger zeitkritischen Datenaufkommens im reservierten asynchronen Kanal

Erreichbar sind bei Ethernet Powerlink Zykluszeiten von unter 200µs und eine zeitliche Präzision (Jitter) von < 1µs.

Ethernet Powerlink spezifiziert außerdem auch ein an CANopen angelehntes Kommunikationsprotokoll zum Nutzdatenaustausch mit Knoten im Netzwerk. Beide Teile zusammen werden von einem Powerlink Protokollstack abgehandelt. Für diesen wird keine spezielle Hardware benötigt, d.h. Ethernet Powerlink setzt auf handelsüblicher Hardware auf. Die aktuelle Version ist Version 2 (EtherType 0x88AB).

Powerlink Datenformat

Jedes Powerlinkpaket ist in den Datenbereich eines normalen Ethernetpakets eingebunden und besteht aus einem Header und den eigentlichen Nutzdaten (siehe Bild 11.24). Der Powerlinkheader selbst setzt sich zusammen aus:

- 1 Bit Reserviert
- 7 Bit MessageType
- 8 Bit Zielknotennummer
- 8 Bit Quellknotennummer

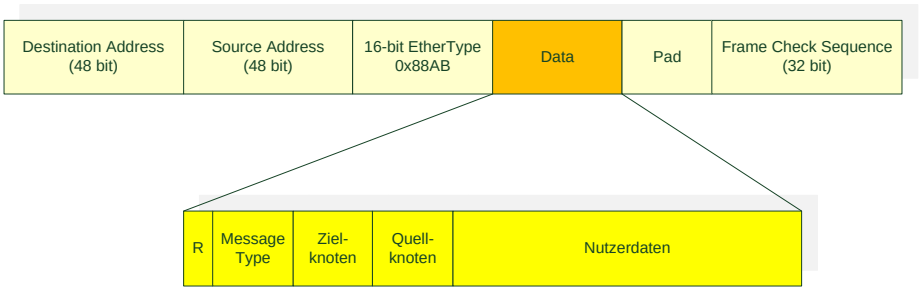


Bild 11.24 Ethernet Powerlink Datenpaket

Folgende Nachrichtentypen (MessageType) sind definiert:

Tabelle 11.7 Message Types für Ethernet Powerlink

MessageType	ID	Bezeichnung der Nachricht	Verwendung	Ethernet Transfertyp
SoC	0x01	Start of Cycle	Definiert den Start eines neuen Zyklus	Multicast
PReq	0x03	PollRequest	Erfrage zyklische Daten des CN	Unicast
PRes	0x04	PollResponse	Sende aktuelle zyklische Daten des CN	Multicast
SoA	0x05	Start of Asynchronous	Signalisiere den Start der asynchronen Phase	Multicast
ASend	0x06	Asynchronous Send	Senden von asynchronen Daten	Multicast

Kommunikation

Im Ethernet-Standard, der grundsätzlich nach dem CSMA/CD-Prinzip arbeitet, ist die Kollisionsvermeidung essentiell für eine Echtzeitkommunikation. Im Switched-Ethernet hingegen tritt das in den Hintergrund, weil die Netzwerkknoten bis zum Switch kollisionsfreie, exklusive Leitungen besitzen (bei Vollduplex).

Das Echtzeitproblem, also die garantierte Übertragung innerhalb einer definierten Zeitspanne, wird damit auf die Switches ausgelagert, weil hier via Scheduling die Meldungen verteilt werden. Die Lösung, die Switches nicht zu überfahren, ist recht einfach: Der Datenverkehr wird beschränkt.

Dazu wird die Datenübertragung durch einen speziellen Teilnehmer, den *Managing Node (MN)*, gesteuert. Die einzelnen Netzwerkteilnehmer, die *Controlled Nodes (CN)* dürfen nur dann senden, wenn sie dezidiert dazu aufgefordert wurden.

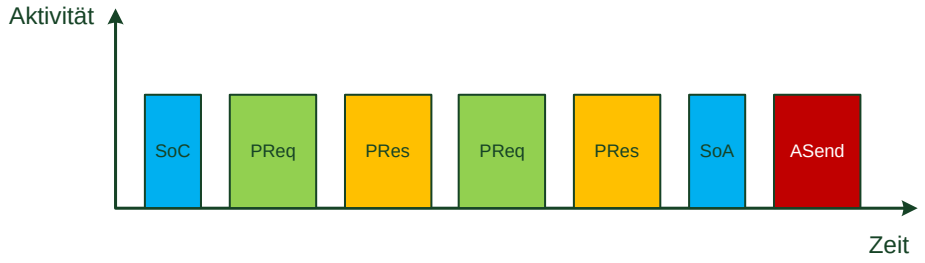


Bild 11.25 Zeitlicher Verlauf Ethernet Powerlink

Ein Zyklus beginnt mit der Nachricht *Start of Cycle (SoC)*. Anschließend wird jeder Knoten einzeln vom MN mit einem *Poll Request (PReq)* abgefragt, worauf der CN mit einem *Poll Response (PRes)* antwortet. Da die Antworten als Ethernet-Multicast gesendet werden, können anderen Powerlinkgeräte mithören. Somit ist Querverkehr zwischen den CNs möglich. Um die Zykluszeit klein zu halten, muss

nicht jedes Gerät in jedem Zyklus abgefragt werden (*Multiplexed Stations*). Die Antwortzeit eines Gerätes ($t_{PRes} - t_{PReq}$) ist ein wichtiges Qualitätsmerkmal, auch für solche Entscheidungen.

Nach Abschluss der zyklischen Phase beginnt die asynchrone Phase mit dem *Start of Asynchronous (SoA)*-Paket. In dieser Phase kann jeweils ein vom MN bestimmter CN nicht-zyklische Daten senden. Über spezielle Gateways lassen sich in der asynchronen Phase Daten aus einem normalen, nicht-deterministischen Netzwerk und dem Powerlinknetzwerk austauschen.

Objektverzeichnisse und Geräteprofile

Zwecks Organisation des Netzwerks und Kopplung zur Applikation werden bei Ethernet Powerlink alle Kommunikationsobjekte und alle Anwenderobjekte in einem Objektverzeichnis (OV) zusammengefasst. Das Objektverzeichnis ist im Powerlink-Gerätemodell das Bindeglied zwischen der Anwendung und der Kommunikationseinheit.

Jeder Eintrag im Objektverzeichnis steht für ein Objekt und wird durch einen 16-bittigen Index gekennzeichnet. Pro Index können wiederum bis zu 256 Subindizes enthalten sein. Dadurch können bis zu 65536 × 254 Nutzeinträge pro Gerät unterschieden werden, da die Subindizes 0 und 255 nicht frei verwendet werden können. In Profilen ist die Zuordnung von Kommunikations- und Geräteprofilobjekten zu einem jeweiligen Index genau definiert, und somit wird mit dem Objektverzeichnis eine eindeutige Schnittstelle zwischen der Anwendung und der Kommunikation nach außen definiert.

Für Geräteprofile, in denen die Funktionalität und der Aufbau des Objektverzeichnisses eingetragen sind, verwendet Ethernet Powerlink die Geräteprofile von CANopen.

11.6.3 IEEE-1588 – Precison Time Protocol als Grundlage der Zeitverteilung

Letztendlich steht und fällt die Echtzeitfähigkeit in Time-Triggered-Kommunikationssystemen mit der Verteilung einer gemeinsamen Zeit. Hier wurde bei IEEE ein präzises Zeitprotokoll definiert (Precision Time Protocol, IEEE-1588, [GM03] [NIST]), mit dessen Hilfe diese Verteilung erfolgen kann.

Die Verteilung erfolgt so, dass eine Clock in dem zu betrachtenden Netzwerk als Master bezeichnet wird. Diese Uhr soll möglichst genau sein, ggf. Anschluss an exakte Zeitgeber haben usw. Der Master sendet nun eine spezielle Meldung als Broadcast aus, die *Sync Message*. Diese Meldung enthält einen Zeitstempel, insbesondere eine Schätzung, wann sie auf dem Netzwerk sein wird.

Falls hohe Präzision gefordert (und möglich) ist, wird die Sync Message von einer zweiten Meldung, der *Follow-Up Message*, begleitet. Diese enthält dann die tatsächlich gemessene Zeit der Übertragung, also des physikalischen Zugriffs auf das Medium Netzwerk. Misst nun der Slave die Empfangszeit mit entsprechender

Präzision, kann er die interne Uhr auf den Master abstimmen – mit der Ausnahme, dass die Übertragungszeit nicht berücksichtigt wurde.

Diese Übertragungszeit kann ebenfalls bestimmt werden. Die Slaves, die diese Sendung empfangen haben, müssen nun mit allerdings geringerer Häufigkeit diese Prozedur wiederholen, indem sie wieder eine Sync Message und ggf. eine Follow-Up Message senden, nun nur an den Master adressiert. Hierin wird die Übertragungszeit der Master-Slave-Abstimmung ebenfalls übermittelt, und nun stehen beide Messungen, hin- und Rückweg, zur Verfügung.

Unter der Annahme, dass die Übertragung eine symmetrische Latenzzeit aufweist, kann nun also auch diese Zeit bestimmt werden. Die Synchronisation reicht hierdurch bis in den Sub-Mikrosekundenbereich zurück, allerdings müssen Router aufgrund ihrer langen Verzögerung ausgeschlossen werden (hierzu bietet IEEE-1588 allerdings ebenfalls Methoden an).

11.7 Beispiele für Feldbus-Systeme

11.7.1 Bitbus

Der *BITBUS* (IEEE-1118) ist ein offener und nicht proprietärer Feldbus. Ursprünglich wurde BITBUS 1984 von Intel spezifiziert und 1991 unter der Bezeichnung IEEE-1118 als internationaler Standard angenommen. Er baut auf zwei verbreiteten Standards als Grundlage auf. Die RS485 Schnittstelle (jetzt: EIA-485) wird als physikalische Verbindung zwischen den einzelnen Geräten verwendet. Auf der Softwareseite wird Synchronous Data Link Control (SDLC) verwendet.

11.7.1.1 Bus-Topologie

Innerhalb der Bus-Topologie können in einem Bus-Segment maximal 28 Teilnehmer miteinander verbunden sein. Beim Einsatz von Repeatern kann die Anzahl der angeschlossenen Geräte auf bis zu 250 erhöht werden. Falls mehr als zwei Repeater in Serie geschaltet sind, beträgt die Datenrate nur noch 62,5 kBit/s. Wird diese Geschwindigkeit verwendet, so können bis zu zehn Repeater hintereinander geschaltet sein. Jeder Repeater kann ein Bus-Segment mit einer Länge von 300 m bzw. 1200 m treiben, abhängig von der gewünschten Übertragungsgeschwindigkeit. Ein Repeater belastet den Bus wie ein gewöhnlicher Teilnehmer.

Der Bus muss an beiden Enden der Leitung mit einem 120 Ω Abschlusswiderstand versehen sein.

11.7.1.2 Elektrische Eigenschaften

Je nachdem zu welcher Länge der Bus ausgebaut wird, können unterschiedliche Datenübertragungsgeschwindigkeiten realisiert werden. Bei einer Buslänge von 300 m ist eine maximale Geschwindigkeit von 375 kBit/s möglich. Erstreckt sich

die Feldbusstrecke über eine Entfernung von 1200 m so sind noch Übertragungsraten von 62,5 kBit/s möglich.

Die Verkabelung erfolgt nach den Vorgaben der Spezifikation EIA-485. Es werden Twisted Pair Kabel zum Anschluss der einzelnen Geräte verwendet. Dabei bilden die Leitungen Data A und Data B ein verdrilltes Adernpaar und die optionalen Verbindungen RTS A und RTS B. Diese werden aber bloß in Segmenten benötigt die jenseits eines Repeaters liegen, wenn ein solcher zum Einsatz kommt. Durch Signalerde werden die Leitungen geschirmt.

Als Steckverbinder findet ein 9-poliger D-Sub Stecker Verwendung.

11.7.1.3 Bitübertragungsschicht (Layer 1)

Beim BITBUS unterscheidet man zwei Daten-Codierungstechniken, den *Synchron Mode* und den *Self Clock Mode*. Beim *Synchron Mode* wird noch ein weiteres Leitungspaar zur Übertragung des Synchronimpulses benötigt. Beim *Self Clock Mode* werden die Bits nicht nach dem Standard NRZ (*Non Return to Zero*) sondern nach NRZI (Non Return to Zero Insert) mit *Zero Bit Insertion* übertragen (→ 11.4.5).

11.7.1.4 Sicherungsschicht (Layer 2)

Pro Datenpaket können maximal 248 Bytes an Nutzdaten übertragen werden.

Der Adressraum beim BITBUS reicht von 0 bis 255 (hexadezimal: 0x00 bis 0xff). Jedem Teilnehmer ist eine eigene Adresse in Form einer Zahl von 1 bis 249 zugeordnet. Die Adressen 0 und 250 bis 255 sind reserviert und dürfen keinem Bus-Teilnehmer zugeordnet werden. 255 spricht in der alten BITBUS-Spezifikation die lokale Netzwerkkarte an. In der neueren IEEE 1118 Norm wird 255 als Broadcast-Adresse verwendet.

11.7.2 Modbus+

Das *Modbus*-Protokoll ist ein Kommunikationsprotokoll, das auf einer Master/Slave- bzw. Client/Server-Architektur basiert. Es wurde 1979 für die Kommunikation mit speicherprogrammierbaren Steuerungen ins Leben gerufen. In der Industrie hat sich der Modbus zu einem De-facto-Standard entwickelt, da es sich um ein offenes Protokoll handelt.

11.7.2.1 Bus-Topologie

Der Modbus definiert keine eigene Bustopologie, sondern es wird die serielle Schnittstelle (RS 232, jetzt EIA-232), EIA-485 oder Ethernet genutzt. Im ersten Fall ist die Bustopologie sternförmig, im zweiten und dritten entsprechend der eingesetzten EIA-485- bzw. Ethernet-Verkabelung.

11.7.2.2 Elektrische Eigenschaften

Entsprechen RS 232 / EIA-562, EIA-485 oder Ethernet.

11.7.2.3 Bitübertragungsschicht (Layer 1)

Entsprechen RS 232 / EIA-562, EIA-485 oder Ethernet.

11.7.2.4 Sicherungsschicht (Layer 2)

Im Fall einer RS 232 wird aufgrund der Punkt-zu-Punkt-Verbindung kein Buszugriff benötigt, in den anderen Fällen wird das jeweilige Protokoll genutzt.

Bei der Datenübertragung werden drei verschiedene Betriebsarten unterschieden:

- Modbus ASCII
- Modbus RTU
- Modbus TCP

Jeder Busteilnehmer muss eine eindeutige Adresse besitzen. Die Adresse 0 ist dabei für einen Broadcast reserviert. Jeder Teilnehmer darf Befehle über den Bus senden, in der Regel wird dies jedoch nur durch den Master ausgeführt.

ASCII-Modus

Im Modbus ASCII wird keine Binärfolge, sondern ASCII-Code übertragen. Dadurch ist es direkt für den Menschen lesbar, allerdings ist der Datendurchsatz im Vergleich zu RTU geringer.

Im ASCII-Modus beginnen Nachrichten mit einem vorangestellten Doppelpunkt, das Ende der Nachricht wird durch die Zeichenfolge Carriage return – Line feed (CRLF) markiert. Das erste Byte enthält zwei ASCII-Zeichen, die die Adresse des Empfängers darstellen. Der auszuführende Befehl ist auf den nächsten zwei Byte codiert. Über n Zeichen folgen die Daten. Über das gesamte Telegramm (ohne Start- und Ende-Markierung) wird zur Fehlerprüfung ein LRC ausgeführt, dessen Paritätsdatenwort in den abschließenden zwei Zeichen untergebracht wird.

Start	Adresse	Funktion	Daten	LR-Check	Ende
1 Zeichen (:)	2 Zeichen	2 Zeichen	n Zeichen	2 Zeichen	2 Zeichen (CRLF)

Bild 11.26 Paketrahmen für Modbus-Protokoll, ASCII-Modus

RTU-Modus

Modbus RTU (RTU: Remote Terminal Unit, entfernte Terminaleinheit) überträgt die Daten in binärer Form. Dies sorgt für einen guten Datendurchsatz, allerdings können die Daten nicht direkt vom Menschen ausgewertet werden, sondern müssen zuvor in ein lesbares Format umgesetzt werden.

Im RTU-Modus wird der Sendebeginn durch eine Sendepause von mindestens drei Zeichen Länge markiert. Die Länge der Sendepause hängt somit von der Übertragungsgeschwindigkeit ab. Das Adressfeld besteht aus acht Bit, die die Empfängeradresse darstellen. Der Slave sendet bei seiner Antwort an den Master eben diese Adresse zurück, damit der Master die Antwort zuordnen kann. Das Funktionsfeld besteht aus 8 Bit. Hat der Slave die Anfrage des Masters korrekt empfangen, so antwortet er mit demselben Funktionscode. Ist ein Fehler aufgetreten, so verändert er den Funktionscode, indem er das höchstwertige Bit des Funktionsfeldes auf 1 setzt.

Das Datenfeld enthält Hinweise, welche Register der Slave auslesen soll, und ab welcher Adresse diese beginnen. Der Slave setzt dort die ausgelesene Daten (z. B. Messwerte) ein, um sie an den Master zu senden. Im Fehlerfall wird dort ein Fehlercode übertragen. Das Feld für die Prüfsumme, die mittels CRC ermittelt wird, beträgt 16 Bit. Das Ende der Nachricht wird durch eine Sendepause von mindestens 1,5 Zeichen Länge markiert.

Start	Adresse	Funktion	Daten	CR-Check	Ende
Wartezeit (min. 3,5 Zeichen)	1 Byte	1 Byte	n Byte	2 Byte	Wartezeit (min 1,5 Zeichen)

Bild 11.27 Paketrahmen für Modbus-Protokoll, RTU-Modus

TCP-Modus
Modbus TCP ist RTU sehr ähnlich, allerdings werden TCP/IP-Pakete verwendet, um die Daten zu übermitteln. Der TCP-Port 502 ist für Modbus TCP reserviert. Modbus TCP befindet sich zurzeit in der Phase der Festlegung als Norm.

11.7.3 Local Operating Network (LON)

Local Operating Network (LON) ist ein Feldbus, der vorwiegend in der Gebäudeautomatisierung eingesetzt wird. Dieser wurde von der US-amerikanischen Firma Echelon Corporation um das Jahr 1990 entwickelt. Seit Dezember 2008 ist diese Technologie von der IEC und der ISO als internationale Norm anerkannt und in der Normenreihe 14908-x dokumentiert, nachdem sie bereits als Europäische Norm unter denselben Kennziffern geführt wurde.

11.7.3.1 Hardware

Hardwareseitiges Kernstück dieses Feldbussystems ist der *Neuron* (Chip). Der Neuron-Chip enthält drei 8 Bit Prozessoren (CPUs): Die Media Access CPU kontrolliert die physikalische Verbindung zum Netzwerk. Die Network-CPU ist für die Codierung und Decodierung der Netzwerknachrichten verantwortlich. Auf der Application CPU läuft die vom Anwender programmierte Software, welche die eigentliche „Intelligenz“ eines Knotens repräsentiert. Jeder Neuron-Chip enthält

eine weltweit einmalige, 48 Bit lange ID-Nummer (die Neuron-ID), mit deren Hilfe jeder Bus-Knoten im Netz eindeutig identifizierbar ist.

Das Kommunikationsprotokoll dieses Feldbusses wird als LonTalk-Protokoll bezeichnet. Das LonTalk-Protokoll definiert die Schichten 2 bis 7 des OSI-Referenz-Modells. Für die physikalische Schicht (Schicht 1 des OSI-Modells) stehen verschiedene Transceiver zur Verfügung, wie zum Beispiel leitungsgebundene Übertragung, Funk, Glasfaser aber auch Powerline-Kommunikation.

Die Datencodierung auf der physikalischen Schicht kann direkt durch den Neuron-Chip oder durch den Transceiver selbst gesteuert werden. Für die direkte Steuerung bietet das LonTalk-Protokoll die Betriebsart Direct Mode. Die Codierung der Daten erfolgt im Manchester-Code. Als Zugriffsverfahren wird ein modifiziertes p-persistentes CSMA (→ 11.5.2.1) mit optionaler Kollisionserkennung eingesetzt, wobei die Möglichkeit besteht, einzelne Nachrichten zu priorisieren. Im Special Purpose Mode können Transceiver mit eigener Signalverarbeitung angesteuert werden. In diesem Mode übernimmt der Transceiver selbst die Steuerung des Medienzugriffs. Grundsätzlich ist dieses Feldbussystem ein Multimastersystem.

Aus logischer Sicht kommunizieren die Knoten über Kommunikationsobjekte miteinander, sogenannter Network Variables (NV). Damit Knoten verschiedener Hersteller miteinander kommunizieren können, werden so genannte SNVTs (Standard Network Variable Types) definiert. Das sind Datentypen aus Anwendersicht, z. B. der Typ SNVT_temp_p, welcher eine Temperatur verkörpert. Die Organisation, welche das vorantreibt, ist die LonMark International.

11.7.4 Profibus

Der Profibus (Process Field Bus) wurde 1987 in seiner ersten Variante (FMS, Fieldbus Message Specification) definiert und 1993 um die wesentlich vereinfachte Variante DP (Decentralized Peripherals) erweitert. Die jetzt existierenden Varianten sind:

- PROFIBUS-DP (Dezentrale Peripherie) zur Ansteuerung von Sensoren und Aktoren durch eine zentrale Steuerung in der Fertigungstechnik. Hier stehen insbesondere auch die vielen Standarddiagnosemöglichkeiten im Vordergrund. Wietere Einsatzgebiete sind die Verbindung von „verteilter Intelligenz“, also die Vernetzung von mehreren Steuerungen untereinander (ähnlich PROFIBUS-FMS). Es sind Datenraten bis zu 12 Mbit/s auf verdrillten Zweidrahtleitungen und/oder Lichtwellenleiter möglich.
- PROFIBUS-PA (Prozess-Automation) wird zur Kontrolle von Messgeräten durch ein Prozessleitsystem in der Prozess- und Verfahrenstechnik eingesetzt. Diese Variante des PROFIBUS ist für explosionsgefährdete Bereiche (Ex-Zone 0 und 1) geeignet. Hier fließt auf den Busleitungen in einem eigensicheren Stromkreis nur ein begrenzter Strom, so dass auch im Störfall keine explosions-

fähigen Funken entstehen können. Ein Nachteil des PROFIBUS PA ist die relativ langsame Datenübertragungsrate von 31,25 kbit/s.

- PROFIBUS-FMS (Fieldbus Message Specification) war vor allem für den Einsatz in komplexen Maschinen und Anlagen gedacht, ist aber von DP abgelöst und heute nicht mehr Bestandteil der internationalen Feldbusnorm.

11.7.4.1 Topologie (für elektrische Variante nach EIA-485):

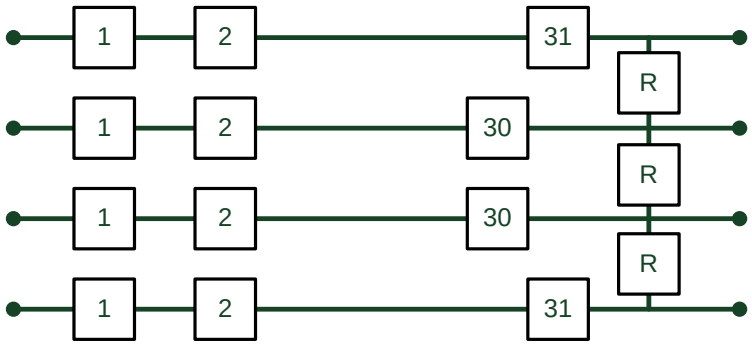


Bild 11.28 Topologie Profibus

- Linienförmig mit Stichleitungen (< 0.3m)
- Leitungslänge von Übertragungsrate abhängig; bei 93,75 kbit/s höchstens 1200m
- Teilnehmeranzahl auf 32 Teilnehmer pro Linie beschränkt
- Repeater (Leitungsverstärker) können Linien zusammenschalten
- Höchstens 3 Repeater zwischen 2 Teilnehmern
- Maximale Leitungslänge: 4.800m, maximale Teilnehmeranzahl: 122

11.7.4.2 Schichtenmodell nach ISO/OSI

Der Profibus (hier in der Variante DP) ist wie folgt in das Schichtenmodell nach ISO/OSI eingebettet:

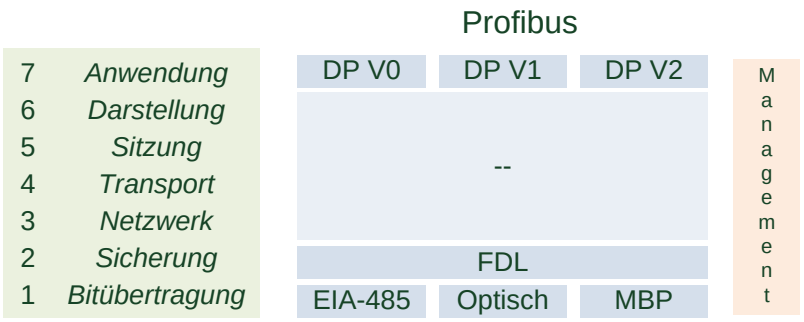


Bild 11.29 ISO/OSI-Schichtenmodell für PROFIBUS

11.7.4.3 Bitübertragungsschicht (Layer 1)

Bei der Bitübertragungsschicht sind drei verschiedene Verfahren festgelegt:

- Bei der elektrischen Übertragung nach EIA-485 werden verdrillte Zweidrahtleitungen mit einer Wellenimpedanz von 150 Ohm in einer Bustopologie eingesetzt. Zeichen werden über das Vorzeichen der Spannungsdifferenz identifiziert (bei EIA-232 bzw. -562 sind für log. 0 und log. 1 absolute Spannungspegel einzuhalten). Es können Bitraten von 9600 bit/s bis 12 Mbit/s eingesetzt werden. Je nach verwendeter Bitrate ist die Kabellänge zwischen zwei Repeatern auf 100 bis 1200 Meter beschränkt. Dieses Übertragungsverfahren wird vor allem beim PROFIBUS DP und FMS eingesetzt.
- Bei der optischen Übertragung über Lichtwellenleiter kommen Stern-, Bus-, und Ring-Topologien zum Einsatz. Die Distanzen zwischen den Repeatern können bis zu 15 km betragen. Die Ring-Topologie kann auch redundant ausgeführt werden.
- Bei der MBP (*Manchester Bus Powered*) Übertragungstechnik werden über dasselbe Kabel Daten und die Speisung der Feldgeräte übertragen. Die Leistung kann so begrenzt werden, dass auch ein Einsatz in explosionsgefährlicher Umgebung möglich ist. Dann spricht man von einem so genannten eigensicheren Bereich. Die Bustopologie kann bis zu 1900 Meter lang sein und lässt Abzweigungen zu den Feldgeräten mit maximal 120 Meter Länge zu. Die Bitrate beträgt hier fest 31,25 kbit/s. Diese Technologie ist speziell für den Einsatz in der Prozessautomation für PROFIBUS PA festgelegt worden.
- Bitcodierung (EIA-485): Non Return to Zero (NRZ)
- Bitsynchronisation: UART (1 Startbit (log. 0), 8 Informationsbits, 1 Paritätsbit (gerade Parität), 1 Stoppbit (log. 1))

- Idle-Time zwischen zwei Übertragungen: Nach Übertragung eines vollständigen Telegramms hat der Bus für 3 Zeichen (33 Bitzeiten) im Ruhezustand (log. 1) zu sein.

Für die Variante EIA-485 gelten folgende Verfeinerungen:

- Leitungsabschlusswiderstände Rt: 150 Ohm, zusätzliche Widerstände für definiertes Ruhepotential Rd: Pulldown (gegen Datenbezugspotential) und Ru: Pullup (gegen Versorgungsspannung)

Bild 11.30 Leitungsabschlusswiderstände im Profibus

- Wellenwiderstand: 100-150 Ohm bei f > 100kHz
- Adernquerschnitt: 0.22mm²
- Sub-D-Stecker (9polig)

Bild 11.31 Steckerdefinition Profibus

11.7.4.4 Sicherungsschicht (Layer 2)

Die Sicherungsschicht FDL (Fieldbus Data Link) arbeitet mit einem hybriden Zugriffsverfahren, das Token-Passing mit einem Master-Slave-Verfahren kombiniert.

In einem PROFIBUS-Netzwerk sind die Steuerungen oder Prozessleitsysteme die Master und die Sensoren und Aktoren die Slaves.

Es werden verschiedene Telegrammtypen verwendet, die durch den Startdelimiter (SD) unterschieden werden können (PROFIBUS DP):

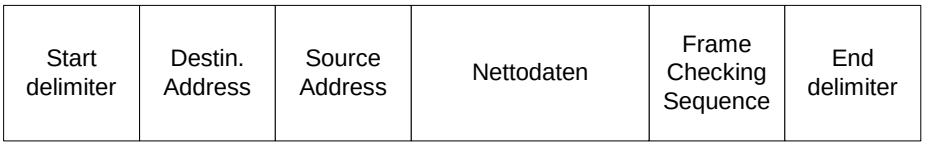


Bild 11.32 Telegrammformat (allgemein) Profibus
(ggf. kann hiervon abgewichen werden, so z.B. beim Paket SD == 0x68)

Startdelimiter	Bedeutung
0x01	Keine Daten
0x68	Daten variabler Länge
0xA2	Daten fester Länge
0xDC	Token (Sendeberechtigung)
0xE5	Kurzquittung

Bild 11.33 Start delimiter im Profibus

Der FCS wird durch einfaches Aufsummieren der Bytes innerhalb der angegebenen Länge berechnet. Ein Überlauf wird dabei ignoriert. Jedes Byte wird mit einer geraden Parität gesichert und asynchron mit Start- und Stopp-Bit übertragen. Folgende Eigenschaften sind im FDL-Protokoll noch implementiert:

- Dynamisches Hinzufügen und Entfernen von aktiven Stationen
- Fehlerhandling wie mehrfache und verloren gegangene Token ist eingeschlossen
- Aber: Zugriffsverwaltung belastet Übertragungskapazität
- Passive Teilnehmer quittieren und antworten

11.7.4.5 Anwendungsschicht

Die DP-Anwendungsschicht wurde in drei Schritten definiert. Das ursprünglich 1993 festgeschriebene DP-Protokoll wird heute umgangssprachlich als „DP-V0“ bezeichnet, die beiden Erweiterungen entsprechend „DP-V1“ und „DP-V2“. In den einzelnen Stufen wurden folgende Funktionen definiert:

- In DP-V0 der zyklische Austausch der Daten und Diagnosen. Geräte, die diesen Funktionsumfang unterstützen, finden vor allem in der allgemeinen Automatisierungstechnik und Maschinensteuerung Einsatz.
- In DP-V1 der azyklische Datenaustausch und die Alarmbehandlung. Geräte, die diese Erweiterungen unterstützen, finden sich vor allem in der Verfahrenstechnik.
- In DP-V2 der isochrone Datenaustausch, der Slave-Querverkehr und die Uhrzeitsynchronisation. Mit dieser Erweiterung wurde vor allem Anforderungen aus der Fertigungstechnik und Robotersteuerung Rechnung getragen.

Das PA-Protokoll wurde im Rahmen der Entwicklungsstufe DP-V1 definiert.

11.7.5 Interbus

1987 wurde der Interbus-S auf der Hannover-Messe unter erstmalig vorgestellt. Mittlerweile wurde die Namensgebung auf Interbus umgestellt, weiterhin gibt es einen Interbus Safety.

11.7.5.1 Topologie

Ein Interbus Netzwerk stellt topologisch eine aktive Ringstruktur dar. Da Hin- und Rückkanal jedoch in einem Anschlusskabel vereinigt sind und die Teilnehmer mind. 2 Anschlussklemmen besitzen (ankommend / abgehend), ergibt sich eine baumartige, physische Verkabelungsstruktur. Zum Schließen des Rings können alle Busteilnehmer ihre Ausgänge intern überbrücken, sollte kein weiterer Teilnehmer folgen. Bei Verzweigungen an so genannten Busklemmen wird der neue Zweig in den Hinkanal eingebunden und der Ring so erweitert. Sollte ein Teilnehmer durch Störung ausfallen, überbrückt der vorherige Teilnehmer seinen Ausgang um den Ring zu schließen und das System bis zum fehlerhaften Teilnehmer lauffähig zu halten.

Es gibt vier Ausprägungen in drei Hierarchieebenen in der baumartigen Verkabelung:

- der Fernbus

max 400m zw. zwei Teilnehmern, max 13km Gesamtlänge, Energieversorgung lokal am Teilnehmer

- Installationsfernbus

wie Fernbus, jedoch mit zentraler Energieversorgung

- der Lokalbus

zweigt über Buskoppler (Busklemmen) vom Fernbus ab, keine weitere Verzweigung möglich, zentrale Energieversorgung, kann einzeln vom Fernbus getrennt (abgeschaltet) werden

- Interbus-Loop

20cm – 20m zw. zwei Teilnehmern, max. 200m Gesamtlänge, Zweidrahtinterface für zentrale Energieversorgung und aufmodulierte Busdaten, Auflösung in nun auch physische Ringstruktur

11.7.5.2 Bitübertragungsschicht (Layer 1)

In der Bitübertragungsschicht (Schicht 1) wird eine NRZ-Codierung (Non Return to Zero, → 11.4.5) genutzt. Standardmäßig erfolgt die Datenübertragung mit 500 kBit/s. Genutzt werden Telegramme mit 13 Bit Länge (5 Bit Header, 8 Bit Daten). Zur Statusbestimmung werden in Übertragungspausen spezielle Header ohne Datenbits übertragen.

11.7.5.3 Sicherungsschicht (Layer 2)

In der Datensicherungsschicht (Schicht 2) des Interbus wird ein Summenrahmenverfahren eingesetzt. Ein Rahmen mit Datenslots für jeden Busteilnehmer wird erstellt und wie bei einem Schieberegister durch die Teilnehmer geschoben. Die Teilnehmer lesen dabei die Eingangsdaten in „ihrem“ Slot ein und speichern dafür ihre Ausgangsdaten. Durch eine Markierung am Ende des Rahmens (Loopback-Wort) erkennt der Master die Ankunft am anderen Ende des Rings und somit das Ende eines Zyklus.

Zur Erstellung des Rahmens bei der Initialisierung oder nach Fehlern fragt der Master alle Busteilnehmer in einem oder mehreren Identifikationszyklen ab. Diese antworten mit Identifikations- und Konfigurationsdaten. Danach folgen Datenzyklen zur Nutzdatenübertragung. Die Länge des Datenrahmens ergibt sich aus der Anzahl der Busteilnehmer sowie der Breite der jeweiligen Nutzdaten. Hinzu kommt ein 16 Bit langes Loopback-Wort, anhand dessen der Master das Ende eines Zyklus erkennt. Am Rahmenende wird eine 32 Bit lange Prüfsumme angehängt, um Datenfehler zu erkennen.

Die einzelnen Busteilnehmer werden nicht direkt adressiert, sondern indirekt durch ihre Position im Ring angesprochen. Eine Umsetzung auf logische Adressen erfolgt erst in Schicht 7.

Durch das Summenrahmenverfahren ergibt sich eine deterministische Laufzeit der Daten. Der Bus kann somit zur Steuerung zeitkritischer Regelungen eingesetzt werden. Das Lesen und Schreiben der Daten erfolgt außerdem stets durch alle Teilnehmer zum selben Zeitpunkt, so entstehen keine Inkonsistenzen.

Neben diesen zyklischen Daten (Prozessdaten) können auch azyklisch auftretende Daten größerer Menge (Parameterdaten) übertragen werden. Dazu besitzt jeder Teilnehmer in seinem Slot zusätzlich einen Bereich für solche azyklischen Daten, der im Regelfall leer bleibt. Die Übertragung der Prozessdaten und das deterministische Zeitverhalten werden hierdurch nicht beeinflusst. In Schicht 7 werden diese zwei Übertragungswege als Prozessdatenkanal und Parameterkanal bezeichnet. Ein eigenes Protokoll (PCP - Peripherals Communication Protocol) kümmert sich um die Aufteilung der oft umfangreichen Parameterdaten in einzelne Pakete,

die in mehreren Zyklen in den freien Bereichen der Slots übertragen und anschließend wieder zusammengesetzt werden.

11.7.6 CAN-Bus

Der CAN-Bus (Controller Area Network) ist ein asynchrones, serielles Bussystem und gehört zu den Feldbussen. Um die Kabelbäume (bis zu 2 km pro Fahrzeug) zu reduzieren und dadurch Gewicht zu sparen, wurde der CAN-Bus 1983 von Bosch für die Vernetzung von Steuergeräten in Automobilen entwickelt und 1987 zusammen mit Intel vorgestellt.

11.7.6.1 Topologie

Das CAN-Netzwerk wird als Linienstruktur aufgebaut. Stichleitungen sind in eingeschränktem Umfang zulässig. Auch ein sternförmiger Bus (Zentralverriegelung) ist möglich. Diese Varianten haben allerdings im Vergleich zum linienförmigen Bus Nachteile:

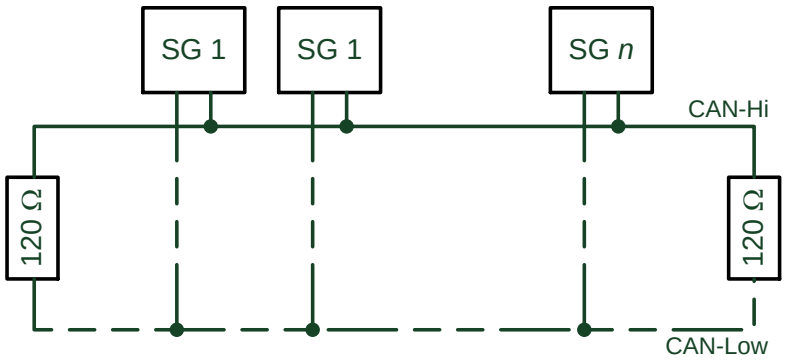


Bild 11.34 Linearer CAN-Bus

- Der sternförmige Bus wird meist von einem Zentralrechner gesteuert, da diesen alle Informationen passieren müssen, mit der Folge, dass bei einem Ausfall des Zentralrechners keine Informationen weitergeleitet werden können. Beim Ausfall eines einzelnen Steuergeräts funktioniert der Bus weiter.
- Stichleitungen und sternförmiger Bus haben den Nachteil, dass der Wellenwiderstand schwer zu bestimmen ist. Im schlimmsten Fall funktioniert der Bus nicht mehr.
- Der lineare Bus hat den Vorteil, dass alle Steuergeräte parallel an einer zentralen Leitung liegen. Nur wenn diese ausfällt, funktioniert der Bus nicht mehr. Diese Topologie wird häufig in Kraftfahrzeugen eingesetzt.

An jedem Busende muss sich ein Abschlusswiderstand von 120 Ω befinden.

11.7.6.2 Bitübertragungsschicht (Layer 1)

Im Falle von Kupferleitungen arbeitet der CAN-Bus bei höheren Datenraten (s. u.) mit Differenzsignalen. Die Differenzsignale werden normalerweise mit 2 oder 3 Leitungen ausgeführt: CAN_HIGH, CAN_LOW und optional CAN_GND (Masse). CAN_LOW enthält den komplementären Pegel von CAN_HIGH gegen Masse. Dadurch können Gleichtaktstörungen unterdrückt werden.

Die Übertragung der Daten erfolgt so, dass ein Bit, je nach Zustand, entweder dominant oder rezessiv auf den Busleitungen wirkt. Ein dominantes überschreibt dabei ein rezessives Bit. Das elektrische Verfahren ist damit ähnlich zu EIA-485, die Daten sind NRZ codiert.

Die maximale Teilnehmeranzahl auf physikalischer Ebene hängt von den verwendeten Bustreiberbausteinen (Transceiver, physikalische Anschaltung an den Bus) ab. Mit gängigen Bausteinen sind 32, 64 oder bis zu 110 (mit Einschränkungen bis zu 128) Teilnehmer pro Leitung möglich (Erweiterungsmöglichkeit über Repeater oder Bridge).

Seitens der Geschwindigkeit wird zwischen einem Highspeed- und einem Lowspeed-Bus unterschieden. Bei einem Highspeed-Bus beträgt die maximale Datenübertragungsrate 1 Mbit/s, bei Lowspeed 125 kbit/s.

Die maximale (theoretische) Leitungslänge beträgt z. B. bei 1 Mbit/s 40 m, bei 500 kbit/s sind 100 m möglich und bei 125 kbit/s 500 m. Diese Maximalwerte beruhen darauf, dass die Zeit, die ein Signal am Bus anliegt (Bitzeit, bit/Sekunde), umso kürzer ist, je höher die Übertragungsrate ist. Mit zunehmender Leitungslänge steigt jedoch die Zeit, die ein Signal braucht, bis es am anderen Ende des Busses angekommen ist (Ausbreitungsgeschwindigkeit). Daher darf die Zeit, die ein Signal am Bus liegt, nicht kürzer sein als die Zeit, die ein Signal braucht um sich auszubreiten.

Der Sender muss wiederum die eventuelle Buspegeländerung des/der Empfänger mitbekommen (→ Sicherungsschicht). Deshalb ist die max. Leitungslänge etwas komplexer zu berechnen. Es müssen Verzögerungszeiten auf der Leitung, des Transceivers (Sender und Empfänger), des Controllers (Sender und Empfänger), Oszillatortoleranzen und der gesetzte Abtastzeitpunkt (Sender und Empfänger) berücksichtigt werden.

Bit Stuffing

Bitstopfen (*bit stuffing,* → 11.4.3.1) kann die physikalische Länge eines Frames vergrößern. Bei bit stuffing wird nach fünf gleichpoligen Bits ein komplementäres Bit (sog. Stopfbit) in den logischen Bitstrom eingefügt. Bit stuffing wirkt auf Start of frame (SOF) bis einschließlich Prüfsummenfeld (CRC) von Daten- sowie Remote-Frames und dient der Nachsynchronisation der Teilnehmer innerhalb eines Frames.

11.7.6.3 Sicherungsschicht (Layer 2)

Der CAN-Bus arbeitet nach dem CSMA/CR (Carrier Sense Multiple Access / Collision Resolution) Verfahren (nicht zu verwechseln mit CSMA/CD wie bei Ethernet) (→11.5.2.3). In der Literatur wird das Verfahren oft als CSMA/CA (Collision Avoidance) benannt, was aber rein historische Gründe hat. Dabei werden Kollisionen beim Buszugriff durch die Arbitrierung oder Bit-Arbitrierung aufgelöst (siehe unten). Des Weiteren kommt zur Datensicherung die Zyklische Redundanzprüfung (ZRP bzw. engl. CRC) zum Einsatz. Zur fortlaufenden Synchronisierung der Busteilnehmer wird Bitstopfen (bit stuffing) verwendet. Der Bus ist entweder mit Kupferleitungen oder über Glasfaser ausgeführt. Der CAN-Bus arbeitet nach dem „Multi-Master Prinzip“: Mehrere gleichberechtigte Steuergeräte (= Busteilnehmer) sind durch eine topologische Anordnung (siehe unten) miteinander verbunden.

Der Buszugriff wird verlustfrei mittels der bitweisen Arbitrierung auf Basis der Identifier der zu sendenden Nachrichten aufgelöst. Dazu überwacht jeder Sender den Bus, während er gerade den Identifier sendet. Senden zwei Teilnehmer gleichzeitig, so überschreibt das erste dominante Bit eines der beiden das entsprechend rezessive des anderen, was dieser erkennt und seinen Übertragungsversuch beendet. Verwenden beide Teilnehmer den gleichen Identifier, wird nicht sofort ein Error-Frame erzeugt (siehe Frame-Aufbau), sondern erst bei einer Kollision innerhalb der restlichen Bits, was durch die Arbitrierung ausgeschlossen sein sollte. Daher empfiehlt der Standard, dass ein Identifier auch nur von maximal einem Teilnehmer verwendet werden soll.

Durch dieses Verfahren ist auch eine Hierarchie der Nachrichten untereinander gegeben. Die Nachricht mit dem niedrigsten Identifier darf immer übertragen werden. Für die Übertragung von zeitkritischen Nachrichten kann also ein Identifier hoher Priorität (= niedrige ID, z. B. 0x001; 0x000 für Netzmanagament - NMT) vergeben werden, um ihnen so Vorrang bei der Übertragung zu gewähren. Dennoch kann selbst bei hochprioren Botschaften der Sendezeitpunkt zeitlich nicht genau vorher bestimmt werden, da gerade in Übertragung befindliche Nachrichten nicht unterbrochen werden können und den Startzeitpunkt einer Sendung so bis zur maximalen Nachrichtenlänge verzögern können (nichtdeterministisches Verhalten). Lediglich die maximale Sendeverzögerung für die höchstpriore Nachricht kann bei bekannter maximaler Nachrichtenlänge errechnet werden.

Objekt-Identifier

Der Objekt-Identifier kennzeichnet den Inhalt der Nachricht, nicht das Gerät. Zum Beispiel kann in einem Messsystem den Parametern Temperatur, Spannung, Druck jeweils ein eigener Identifier zugewiesen sein. Die Empfänger entscheiden anhand des Identifiers, ob die Nachricht für sie relevant ist oder nicht.

Zudem dient der Objekt-Identifier auch der Priorisierung der Nachrichten. Die Spezifikation definiert zwei verschiedene Identifier-Formate:

- 11-Bit-Identifier, auch „Base frame format“ genannt (CAN 2.0A)
- 29-Bit-Identifier, auch „Extended frame format“ genannt (CAN 2.0B).

Ein Teilnehmer kann Empfänger und Sender von Nachrichten mit beliebig vielen Identifiern sein, aber umgekehrt darf es zu einem Identifier immer nur maximal einen Sender geben, damit die Arbitrierung funktioniert.

Der CAN-Standard fordert, dass eine Implementierung das „Base frame format“ akzeptieren muss, dagegen das „Extended frame format“ akzeptieren kann, es aber zumindest tolerieren muss.

Frame-Aufbau

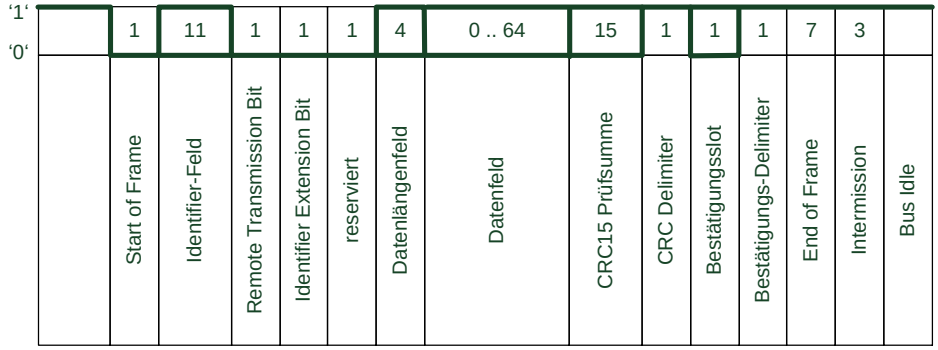


Bild 11.35 CAN-Datentelegramm im Base Frame Format

Die Kommunikation erfolgt mit Telegrammen. Innerhalb eines Telegramms gibt es Steuerbits und Nutzbits. Der genormte Aufbau eines solchen Telegrammrahmens wird als Frame bezeichnet.

Es gibt vier verschiedene Arten von Frames:

- Daten-Frame, dient dem Transport von bis zu 8 Byte an Daten
- Remote-Frame, dient der Anforderung eines Daten-Frames von einem anderen Teilnehmer
- Error-Frame, signalisiert allen Teilnehmern eine erkannte Fehlerbedingung in der Übertragung
- Overload-Frame, dient als Zwangspause zwischen Daten- und Remote-Frames

ACK-Slot

Der Acknowledge-Slot wird verwendet, um den Empfang eines korrekten CAN-Frames zu quittieren. Jeder Empfänger, der keinen Fehler feststellen konnte, setzt einen dominanten Pegel an der Stelle des ACK-Slots und überschreibt somit den rezessiven Pegel des Senders. Im Falle einer negativen Quittung (rezessiver Pegel) muss der fehlererkennende Knoten nach dem ACK-Delimiter ein Error-Flag

auflegen, damit erstens der Sender vom Übertragungsfehler in Kenntnis gesetzt wird und zweitens um netzweite Datenkonsistenz sicherzustellen. Wird der rezessive Pegel von einem Empfänger durch einen dominanten überschrieben, kann der Absender jedoch nicht davon ausgehen, dass das Telegramm von allen Empfängern erhalten wurde.

Datensicherung
Erkennt ein Empfänger eine Fehlerbedingung, sendet er einen Error-Frame und veranlasst so alle Teilnehmer, den Frame zu verwerfen. Sollten andere Teilnehmer diese Fehlerbedingung nicht erkannt haben, senden sie ihrerseits direkt im Anschluss ein weiteres Error-Frame. Damit wird eine weitere Sicherheitsfunktion des CAN-Protokolls möglich. Um zu vermeiden, dass einzelne Teilnehmer durch irrtümlich erkannte Fehlerbedingungen dauerhaft den Nachrichtentransport blockieren, enthält jeder Teilnehmer Fehlerzähler. Diese Zähler erlauben nach den Regeln der Spezifikation, einen fehlerhaft arbeitenden Teilnehmer in zwei Stufen des Betriebszustands vom Bus zu trennen, wenn er wiederholt Fehler erkennt, welche andere Teilnehmer nicht erkennen oder wiederholt fehlerhafte Frames versendet. Die Zustände nennen sich *error active* (normal), *error passive* (Teilnehmer darf nur noch passive - das heißt rezessive - Error-Frames senden) und *bus off* (Teilnehmer darf nicht mehr senden).

Der Sender wiederholt nach dem Error-Frame seine Datenübertragung. Auch der Sender kann durch die zuvor erwähnten Fehlerzähler vom Bus getrennt werden, wenn die Datenübertragung dauerhaft fehlschlägt. Verschiedene Fehlerfälle führen zu einer unterschiedlich großen Erhöhung des Fehlerzählers.

11.7.6.4 Anwendungsschicht

CANopen
CANopen ist ein auf CAN basierendes Schicht-7-Kommunikationsprotokoll, welches anfänglich in der Automatisierungstechnik verwendet wurde, mittlerweile aber vorwiegend in Embedded Systemen eingesetzt wird.

CANopen wurde vorwiegend von deutschen klein- und mittelständischen Firmen initiiert und im Rahmen eines Esprit-Projektes unter Leitung von Bosch erarbeitet. Seit 1995 wird es von der CAN in Automation gepflegt und ist inzwischen als Europäische Norm EN 50325-4 standardisiert. Der Einsatz erfolgt vorwiegend in Europa, gefolgt von Asien.

SafetyBUS p
SafetyBUS p ist ein auf CAN basierendes sicheres Kommunikationsprotokoll, welches hauptsächlich in der Automatisierungstechnik zur Übertragung sicherheitsgerichteter Daten verwendet wird. Alle Busteilnehmer sind 2- oder sogar 3-kanalig aufgebaut und prüfen die Datenintegrität. Das Übertragungsmedium selbst ist nicht sicher, die Sicherheit wird durch das SafetyBUS p eigene Datenprotokoll erreicht. Der SafetyBUS p kann bis SIL3 (→ 14.3.5) eingesetzt werden.

TTCAN
Time-Triggered Communication on CAN setzt auf dem CAN-Bus auf und ermöglicht über höhere Protokollebenen eine Echtzeitsteuerung.

11.8 Sichere Netzwerke

Sicherheit in Netzwerken – und in Rechner-basierten Systemen – ist ein großes Thema, das auch eingebettete Systeme umschließt. Hierbei wird Sicherheit in zweifacher Hinsicht gesehen:

- Betriebssicherheit (*Safety*) stellt diejenige Form der Sicherheit dar, die das äußere, umgebende System vor einer unsicheren Handlung des (eingebetteten) Systems schützt. Auf Netzwerke bezogen bedeutet dies, dass fehlerhafte oder fehlende Netzpakete nicht zu Fehlhandlungen führen dürfen.
- Angriffssicherheit (*Security* bezieht sich darauf, dass ein Rechnersystem alle unberechtigten Zugriffe abwehrt und somit den unbefugten Zugang zwecks Sabotage oder Spionage unterbindet. Auf Netzwerke bezogen bedeutet dies, dass eine Zugangsberechtigung geschaffen wird, die möglichst jedwedes Eindringen unterbindet – insbesondere auf dem Gebiet der drahtlosen Netzwerke eine schwierige Aufgabe.

In diesem Abschnitt sollen nun Grundprinzipien dargestellt werden, die das Design von betriebs- und angriffssicheren Netzwerken ermöglichen.

11.8.1 Betriebssichere Netzwerke (Safety)

Bezüglich der Betriebssicherheit in einem Netzwerk existieren drei wesentliche Aufgaben, deren Lösungsmöglichkeiten hier beschrieben werden sollen:

- Der Inhalt des übermittelteten Pakets muss unverfälscht sein bzw. eine Verfälschung muss erkennbar und ggf. sogar reparierbar sein.
- Der Ausfall eines Knotens muss sicher erkannt werden.
- Der Ausfall bzw. die Fehlfunktion eines Knotens darf nicht zum Ausfall des Netzes oder zu einer wesentlichen Beeinträchtigung des Netzbetriebs führen.

Diese drei Forderungen werden auf unterschiedlichen Ebenen behandelt.

Korrektheit des Paketinhalts
Die Korrektheit eines Paketinhalts lässt sich durch Checksummen gegen Einzel- und Mehrbitfehler sichern (→ 14.3.2). Möglich ist hier die Einführung einer Checksumme, bei Ethernet beispielsweise als Frame Check Sequence (FCS, → 11.5.3.2). Diese Checksumme beruht fast immer auf einer Cyclic Redundancy Checksum (CRC) mit einer entsprechenden Anzahl von Bits, z.B. CRC32 (→ 14.3.2.3).

Sollte dies nicht ausreichen – die Erkennung von Mehrbitfehlern ist hier problematisch –, dann kann als weitere Maßnahme der Pakteinhalt verdoppelt werden,

indem die nachricht zweimal enthalten ist. Hierdurch wird die Wahrscheinlichkeit eines unentdeckten Fehlers sehr weit gedrückt, so dass auch höchste Sicherheitsanforderungen erfüllt werden.

Ausfallerkennung eines Knotens
Die Erkennung des Ausfalls eines Knotens, der also komplett nicht mehr sendet, ist bei sicherheitskritischen, verteilten Applikationen wie z.B. Brake-by-Wire im Automobil oder Flugzeug zwingend erforderlich, um den verbleibenden Netzknoten die Möglichkeit zum Ausgleich zu geben. Die Einführung zeitgetakteter, obligatorischer Nachrichten ermöglicht die Ausfallerkennung, sofern alle Netzteilnehmer über die gleiche, miteinander synchronisierte Zeit verfügen (→ 11.6.3) und mithilfe der Zeittabellen wissen, wann welcher Netzteilnehmer zumindest ein „Hallo"-Paket senden muss.

Mit anderen Worten: Zur Ausfallerkennung sind grundsätzlich zeitgesteuerte Netzwerke in der Lage.

Verhinderung von Auswirkungen eines defekten Netzknotens
Um zu verhindern, dass der einfache Ausfall eines Netzknotens den kompletten Netzwerkverkehr verhindert (oder das Netzwerk in zwei Subnetze unterteilt), muss die Topologie entsprechend gewählt werden – dies ist aber Standard und in praktische allen Netzwerken, sieht man einmal von sehr einfachen ab, realisiert.

Eine wesentlich interessantere Aufgabe besteht darin, einen Störer (auch: babbling idiot, quatschender Idiot) an der Lahmlegung des Netzwerkverkehrs zu hindern. Hier sind zwei Ansatzpunkte möglich:

a) Die Netzknoten sind jeweils mit einem Switch (oder Router) verbunden, die einen Störer aufgrund bestimmter Eigenschaften, z.B. senden außerhalb der Tabellen-definierten Zeiten, entdecken und diesen Zweig abschalten. Dies bedingt die Einführung spezieller Switches.
b) Die Netzknoten selbst haben eine unabhängige Überwachungsinstanz, deren Aufgabe in der Überwachung der Netzwerkaktivitäten des Knotens besteht. Zu diesem Zweck führen diese Netzwerküberwachungseinheiten die Zeittabellen, die die Transmissionen aus diesem Knoten heraus definieren, mit. Netzwerkpakete werden dann nur gesendet, wenn beide Einheiten diese zeitgesteuert freugeben.

11.8.2 Angriffssichere Systeme

Die (Angriffs-)Sicherheit von Systemen ist generell stark abhängig von Angreifermodell, da sich die ergriffenen Maßnahmen stark daran orientieren. Im Grunde genommen ist es nur eine Maßnahme: Die Sendungen werden verschlüsselt (kryptographiert) übertragen.

Damit reduziert sich das Erreichen der Angriffssicherheit auf die Wahl des Schlüsselverfahrens und natürlich der Schlüssel. In Netzen werden oft symmetrische Verfahren wie DES (Data Encryption Standard, [Wiki_DES]), Triple DES oder

AES (Advanced Encryption Standard, [Wiki_AES]). Das trägt einen prinzipiellen Nachteil in sich: Der Schlüsselaustausch muss zuvor „per Hand“ erfolgen, oder der Algorithmus ist beim Schlüsselaustausch sehr unsicher. Der wesentliche Vorteil ist jedoch, dass der Rechenaufwand der Kryptographie wesentlich kleiner ist als bei asymmetrischen Verfahren, bei denen Ver- und Entschlüsselung auf verschiedenen Schlüsseln beruhen.

Die Situation wird in ad-hoc-Netzwerken, die sich sozusagen spontan bilden, noch verschärft; hier sind asymmetrische Verschlüsselungsverfahren zwingend notwendig (wegen des Schlüsselaustausches). Andererseits sind diese ad-hoc-Netzwerke wie im Fall der darhtlosen Sensornetzwerke (Wireless Sensor Network, WSN) oftmals mit Knoten versehen, die ihrerseits kaum Rechen- und Speicherkapazitäten in ausreichendem Maße zur Verfügung stellen können.

Dieses Gebiet ist somit ein Bereich, der noch klar in der Forschung steckt. Einen guten Überblick zu den Herausforderungen und den Ansätzen bieten [BHU+10] und [FH10].

12 Design verteilter Applikationen im Bereich Eingebetteter Systeme

12.1 Verteilte eingebettete Systeme

Verteilte Systeme sind – vereinfachend ausgedrückt – Systeme von mehreren, miteinander kommunizierenden Rechnern, die an einem Algorithmus oder Problem gemeinsam arbeiten. Die damit verbundene Mischung aus Freiheit der einzelnen Rechner – eine Kommunikation stellt immer eine wesentlich lockerere Kopplung dar – und zugleich der Kopplung an ein Problem erzeugt neue Aufgaben, die zur Systemerstellung gelöst werden müssen:

- Echtzeitverhalten des Netzwerks als Grundlage der Echtzeitfähigkeit des Systems.
- Zeitsynchronisation (einige Verfahren basieren darauf, eine gemeinsame Zeit im System zu haben)
- Synchronisation von beteiligten Rechnern bei gemeinsamen Aufgaben, einschließlich Datenübergabe und –rückgabe.

Diese Themen werden in den folgenden Abschnitten behandelt.

12.1.1 Echtzeitverhalten der Übertragung

Das Wesen der verteilten Systeme – die Einbindung und der Zugriff auf ein nichtexklusives Kommunikationsmedium – erfordert eine gesonderte Behandlung, bedingt eben durch die Nicht-Exklusivität. Ein derartiges System kann so ausgelegt sein, dass der jeweils lokale Teil auf Basis einer modifizierten Ereignissteuerung läuft, die Kommunikation ggf. jedoch entkoppelt davon.

Auf Seiten des Netzwerks muss ein deterministisches Verfahren zur Buszuteilung existieren, das zumindest für einen Satz von Nachrichten die echtzeitfähige Übertragung garantiert. Hier folgt eine kurze Diskussion der Zuteilungsverfahren (siehe auch 11.5.2):

- CSMA/CD (Carrier Sense Media Access with Collision Detection): Dieses bei Ethernet I verwendete Verfahren scheidet als Kandidat für Echtzeitnetzwerke aus, da der Zugriff probabilistisch ist und somit keine maximale Übertragungszeit garantiert werden kann – es sei denn, in einem besonderen Verfahren wird die Sendezeit zugeteilt. Das arbeitet dann deterministisch, das CSMA/CD-Verfahren ist damit aber durchtunnelt und somit hinfällig.

Bei Ethernet II (→ 11.5.3.2) werden jedoch exklusive Leitungen zwischen Netzteilnehmer und einem Switch geführt, so dass bei Vollduplexbetrieb keine

Konflikte mehr auftreten können. In diesem Fall ist die Echtzeitfähigkeit an die Flusskontrolle und das Switch-interne Scheduling gekoppelt.

- CSMA/CR (Carrier Sense Media Access with Collision Resolution) auch als CSMA/CA (Collision Avoidance) bezeichnet: Das Controller-Area Network (CAN) verwendet dieses Verfahren, bei dem bei einem Zugriff eine Kollision vermieden wird. Dies bedeutet, dass ohne weitere Maßnahmen die höchste Priorität garantiert übertragen wird, alle anderen aber wiederum keine Echtzeitfähigkeit besitzen..

Die besonderen Maßnahmen können die maximale Wiederholungsfrequenz betreffen. Durch diese Einschränkung könnte ein CSMA/CR-Netzwerk echtzeitfähig werden. Dadurch wäre ein Ereignis-gesteuertes Netzwerk tatsächlich möglich!

- Zuteilungsverfahren, insbesondere Zeit-basiert: TTP/C (Time-Triggered Protocol Class C) und Byte Flight: In diesen Zeit-gesteuerten Protokollen besitzen alle Knoten eine gemeinsame Zeit mit geringem Jitter. Dies wird durch spezielle Verteilung erreicht. Über eine Zeittabellen-gesteuerte Nachrichtensendung erhält jeder Knoten eine garantierte Sendemöglichkeit, außerdem können alle anderen Knoten die Betriebsfähigkeit des sendenden erkennen (und vor allem auch den Ausfall!).

Das Byte Flight Protokoll benötigt einen ausgezeichneten Sender, der über ein Zeitsignal eine gemeinsame Zeit verteilt. Diese gemeinsame Zeitbasis (Jitter: 100 ns) veranlasst die anderen Knoten nacheinander, Pakete zu senden oder ruhig zu bleiben. Dadurch wird es möglich, für eine begrenzte Anzahl von Sendungen einen exklusiven Zugriff zu gestatten.

Im Byte-Flight-Protokoll wird der Rest in einem Zeitschlitz nach dem CSMA/ CR-Verfahren verteilt, sodass der Bus optimal ausgenutzt wird und zugleich (für eine begrenzte Anzahl von Daten) echtzeitfähig ist.

12.1.2 Verteilung der Zeit in verteilten Systemen

Letztendlich steht und fällt die Echtzeitfähigkeit in Time-Triggered-Kommunikationssystemen mit der Verteilung einer gemeinsamen Zeit. Hier wurde bei IEEE ein präzises Zeitprotokoll definiert (Precision Time Protocol, IEEE-1588, [GM03] [IEE1588]), mit dessen Hilfe diese Verteilung durchgeführt werden kann.

Die Verteilung erfolgt so, dass eine Clock in dem zu betrachtenden Netzwerk als Master bezeichnet wird. Diese Uhr soll möglichst genau sein, ggf. Anschluss an exakte Zeitgeber haben usw. Der Master sendet nun eine spezielle Meldung als Broadcast aus, die *Sync Message*. Diese Meldung enthält einen Zeitstempel, insbesondere eine Schätzung, wann sie auf dem Netzwerk sein wird.

Falls hohe Präzision gefordert (und möglich) ist, wird die Sync Message von einer zweiten Meldung, der *Follow-Up Message*. Diese enthält dann die tatsächlich gemessene Zeit der Übertragung, also des physikalischen Zugriffs auf das Medium

Netzwerk. Misst nun der Slave die Empfangszeit mit entsprechender Präzision, kann er die interne Uhr auf den Master abstimmen – mit der Ausnahme, dass die Übertragungszeit nicht berücksichtigt wurde.

Diese Übertragungszeit kann ebenfalls bestimmt werden. Die Slaves, die diese Sendung empfangen haben, müssen nun mit allerdings geringerer Häufigkeit diese Prozedur wiederholen, indem sie wieder eine Sync Message und ggf. eine Follow-Up Message senden, nun nur an den Master adressiert. Hierin wird die Übertragungszeit der Master-Slave-Abstimmung ebenfalls übermittelt, und nun stehen beide Messungen, hin- und Rückweg, zur Verfügung.

Unter der Annahme, dass die Übertragung eine symmetrische Latenzzeit aufweist, kann nun also auch diese Zeit bestimmt werden. Die Synchronisation reicht hierdurch bis in den Sub-Mikrosekundenbereich zurück, allerdings müssen Router aufgrund ihrer langen Verzögerung ausgeschlossen werden (hierzu bietet IEEE-1588 ebenfalls Methoden an).

12.2 Kopplung der Applikationen im verteilten System

Ein allgemeines verteiltes System koppelt in der Regel Rechner zusammen, die nicht an einem Ort lokalisiert sind. Zumindest kann man diese Lokalität nicht voraussetzen, und meist sind die Standorte der Rechner nicht einmal bekannt. Zudem muss man mit einer Dynamik in der Zusammensetzung des verteilten Systems rechnen, da Funktionen verlagert, Ausfälle ausgeglichen und Daten aus Redundanzgründen mehrfach gehalten werden All dies hat durchaus Konsequenzen für das Design der Gesamtapplikation.

Für eingebettete Systeme jedoch gilt zumeist, dass diese – auch bei einem verteilten System – räumlich eng beieinander sind. Allein die physikalische Einbettung der Rechner in eine übergeordnete Maschine, etwa ein Industrieroboter, spricht für diese Annahme, und bei einer derartigen lokalen Anordnung sind Echtzeitbedingungen (→ 12.1.1) überhaupt möglich.

Ein weiterer Unterschied zwischen allgemeinen und eingebetteten verteilten Systemen dürfte sein, dass die eingebettete verteilte Applikation streng geplant sein wird, während man bei den allgemeinen Systemen eher von einem ”organischen“ Wachstum ausgehen sollte – soll heißen, dass das Systm kaum von Null an geplant ist.

Bei (allgemeinen) verteilten Systemen besteht die Kopplung in folgenden Verfahren:

- Übermittlung von Nachrichten (lose Kopplung)
- Remote Procedure Calls (RPC), Remote Method Invocation (RMI) und CORBA Middleware (Common Object Request Broker Architecture) (starke

Kopplung). Hierbei werden Prozeduren- bzw. Methodenaufrufe mit der Übermittlung von Daten (Parameter, Objektdaten) gekoppelt.

Diese beiden Methoden sind auch für eingebettete Systeme anwendbar, wie in den folgenden Abschnitten diskutiert wird.

12.2.1 Kopplung per Nachrichten

Die Kopplung per Nachrichtenaustausch ist vergleichsweise einfach, da für alle Seiten lediglich die Semantik der Nachrichten definiert werden muss. Es existiert hierfür aber noch eine weitergehende Methode.

In einem geplanten verteilten System können (Netz-)globale (auch als externe bezeichnete) Variable definiert werden, die eine andere Charakteristik als die Von-Neumann-Variable besitzen. Jeweils ein lokaler Rechner im verteilten System definiert eine solche externe Variable und dient als Datenquelle. Alle anderen Rechner im System stellen nur Senken für diese Variable dar.

Diese externen Variablen werden dann an die Zielrechner kommuniziert, z.B. im Broadcastverfahren. Durch die Planung des Systems können auch systemweit eindeutige Namen vergeben werden.

Die externen Variablen unterscheiden sich insoweit von den Von-Neumann-Variablen, dass sie strikt zwischen schreibend/lesend (auf dem erzeugenden, lokalen System) und nur lesend unterscheiden. Sie sind hervorragend zur Kopplung in einem verteilten System geeignet, soweit eine ausreichend zeitsynchrone Kommunikation im System möglich ist, d.h., alle betreffenden Empfänger erhalten die aktuelle Information ausreichend ”gleichzeitig“.

12.2.2 Ergänzungen zum Design Pattern (Software Events) für verteilte Systeme

Die zweite Art der Kopplung soll am Beispiel von Remote Procedure Calls (RPC) behandelt werden. Der Aufruf einer entfernten Prozedur unterscheidet sich in drei Formen von dem einer lokalen Prozedur:

- Die *Aufrufsemantik* muss definiert werden: Während bei lokalen Prozeduren das Prinzip ”exactly-once“ (eine Prozedur wird an einer Stelle exakt einmal aufgerufen) gilt, ist es bei RPCs entweder ”at-most once“ (maximal einmal) oder ”at-least once“ (mindestens einmal). Dies muss von Fall zu Fall entschieden werden: Im at-most-once-Prinzip verzichtet man auf einen Aufruf, falls das Ergebnis oder der Erfolg unbekannt ist. Der Grund dafür liegt darin, dass eine Wiederholung schädlich sein kann.
- Die Kommunikation der *Aufrufparameter* und des/der *Rückgabewert/e* muss geregelt werden, da die voneinander getrennte Rechner keinen gemeinsamen Adressraum haben und somit eine Specherschnittstelle (Stack) für die Kommunikation nicht möglich ist. Dieser Vorgang wird meist als Marshalling/Unmarshalling bezeichnet.

- Das zeitliche Verhalten der beiden Kommunikationspartner, insbesondere des aufrufenden Rechners, kann die Formen *synchron*, *asynchron* oder *verzögert synchron* annehmen.

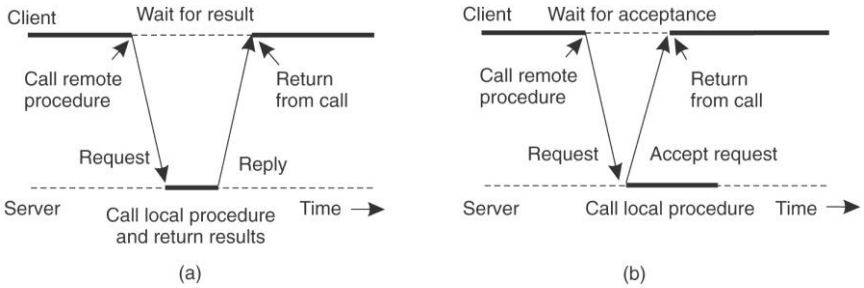


Bild 12.1 Remote Procedure Call
a) synchron (blockierend) b) asynchron (nicht-blockierend)

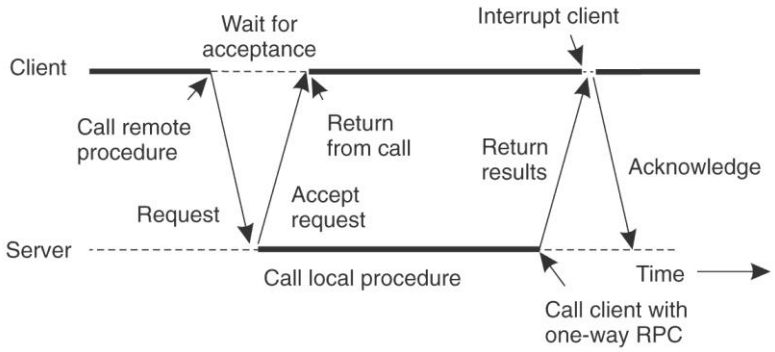


Bild 12.2 Remote Procedure Call, verzögert synchron

Bild 12.1 zeigt die Basisvarianten synchron (blockierend) und asynchron (nicht blockierend). Im synchronen Fall wartet der Aufrufer darauf, dass der aufgerufene, hier als Server bezeichnet, die Prozedur ausgeführt hat und die Ergebnisse zurücksendet. Dieses Verfahren ist einfach, bedeutet aber ein unnützes Warten.

In der asynchronen Variante entsteht die Wartezeit nicht (oder kaum). Der Aufrufer wartet lediglich auf den Eingang der Bestätigung, das Programm rechnet dann weiter. Das größte Problem hierbei ist der Erhalt der Ergebnisse (oder auch nur der Kenntnis, dass die Rechnung beendet ist.

Dieses Problem wird in der Variante, die in Bild 12.2 dargestellt ist, behoben. Hier wird die gesamte Kommunikation in zwei Teile aufgeteilt: Teil 1 entspricht der asynchronen Kommunikation aus Bild 12.1b, zusätzlich wird eine weitere, als Einweg-RPC bezeichnete Kommunikation durchgeführt, mit der das Ende der Rechnung und die Ergebnisse mitgeteilt werden.

Exakt diese Kommunikation, als verzögert synchron bezeichnet, kann im Threadingmodell nach Abschnitt 4.2 effizient unterstützt werden. Die Threads wurden in Definition 3.1 als asynchron gekoppelte, in sich geschlossene Programmteile bezeichnet, und diese asynchrone Kopplung bietet hier die Möglichkeit zum Aufbau einer verzögert synchron Kopplung. Zusätzlich können zur Identifikation von Kommunikationsschwierigkeiten *Timeouts* eingerichtet werden.

Realisierung von Timeouts

Um das diskutierte Designpattern (→ 4.2) perfekt einsetzen zu können, müssen also zwei Mechanismen geschaffen werden bzw. möglich sein: Die Einfügung von Timeouts im Programmfluss sowie die verzögert synchrone Kommunikation.

```
unsigned long ui32gSoftTimer;
unsigned long ui32gSoftTimerEvent, ui32gThreadID;

void interrupt vTimerISR()
{
   …
   if( ui32gSoftTimer != 0 )
   {
      ui32gSoftTimer--;
      if( ui32gSoftTimer == 0 )
      {
         iStoreNewEvent(ui32gSoftTimerEvent, ui32gThreadID );
      }
   }
   …
}
```

Bild 12.3 Implementierung eines Software-Timers für Timeout-Signalisierung

Timeouts lassen sich am Besten durch software-Timer implementieren. Hierzu ist ein Hardware-Timer notwendig, also ein Zähler von Takten, der auf eine Taktanzahl konfigueirt werden kann und von hier beginnend bis 0 herunterzählt. Beim Dekrementieren der Zahl 0 erfolgt ein so genannter Underflow, der einen Interrupt Request auslöst – die Zeit-gesteuerten Systeme werden so aufgebaut.

Der Software-Timer benutzt diese Basiszeit, auch als Tick bezeichnet, zur Implementierung einer Messzeit von ganzzahligen Vielfachen dieser Zeit. Bild 12.3 zeigt eine beispielshafte Implementierung, die eine globale Variable ui32gSoftTimer nutzt.

Dieser Code ist zwar nicht perfekt, er zeigt aber das Prinzip: Benötigt ein Thread einen solchen Timeout, muss er sich einen freien Software-Timer, also eine nicht genutzte Variable wie ui32gSoftTimer, suchen und diesen mit dem Zeitwert, der zu übermittelnden Nachricht und der Thread-ID initialisieren. Bei Erreichen des Nullwerts wird dann das Event an die Messagequeue gesendet, und der Scheduler muss es dem korrespondierenden Thread übermitteln.

Verzögert synchrone Kommunikation

Die verzögert synchrone Kommunikation stellt im Thread-Modell quasi eine Standard-Architektur dar, sofern zwischen zwei Threads kommuniziert wird. Genau das verbleibt als Aufgabe für die Softwareentwicklung: Da die Kommunikation verdoppelt ist – der ursprüngliche Auftraggeber wird per Kommunikation über die Erfüllung des Auftrags informiert – muss auch nach der zweiten Kommunikation ein Thread aktiviert werden. Hieraus ergibt sich folgender, in Bild 12.4 dargestellter Ablauf.

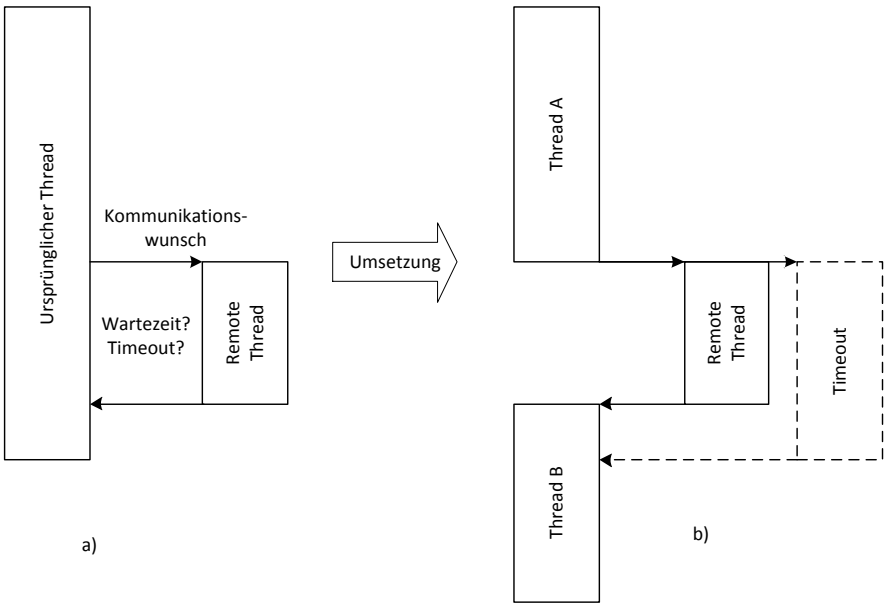


Bild 12.4 Verzögert synchrone Kommunikation mithilfe des Thread-Modells
a) ursprüngliches Softwaredesign b) neues Softwaredesign mit Threadpartitionierung

Die optimale Vorgehensweise besteht darin, den ursprünglichen Thread an der Kommunikationsstelle zu teilen und somit den Kommunikations-initiierenden Thread im Anschluss zu beenden. Die ’Ready’-Nachricht des Remote Thread startet dann den zweiten Teil des Threads, wobei durch einen eventuellen Timeout eine Blockade des Systems auszuschließen ist.

Abschnitt IV: Test und Verifikation

Zusammenfassung und Überblick zu Abschnitt IV

Eingebettete Systeme sind technische Systeme und somit Fehlern unterworfen. Bild IV.1 gibt einen Überblick zu den internen Fehlern (→ 14.2), die während des Designprozesses und der Laufzeit entstehen. Hierbei sind relevant:

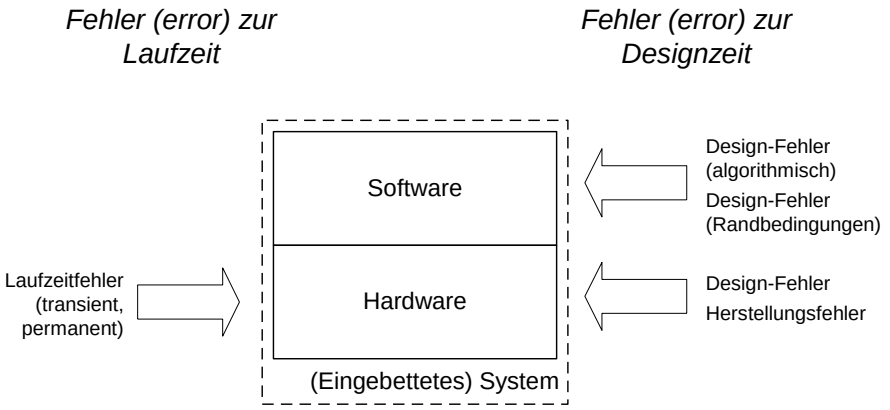


Bild IV.1 Überblick zu den entstehenden Fehlern in eingebetteten Systemen

- Die Laufzeitfehler, die in der Störung der Hardware bestehen. Gründe hierfür sind Höhenstrahlung, natürliche Rest-Radioaktivität, Elektromigration, Elektromagnetische Unverträglichkeit usw.. Die Fehler können transient oder dauerhaft sein.
- Die Designzeit-Fehler, die Hard- und Software gleichermaßen betreffen. Wesentliche Fehlerklassen sind dabei in der Hardware die eigentlichen Designfehler, also die Abweichungen vom intendierten Verhalten durch (menschliche) Fehler, und die Herstellungsfehler. Die Fehler in der Software betreffen algorithmisches Fehlverhalten sowie – wesentlich schwieriger zu detektieren und zu vermeiden – Fehler in den übrigen Systemparametern, namentlich im Abweichen vom intendierten Zeitverhalten.

Fehler in den eingebetteten Systemen müssen nun erkannt werden.Hierzu zählen Testverfahren, die online (→ 14.3),quasi-online (Built-In Self Test, BIST) oder offline(→ 15) durchgeführt werden können, im Fall derOnline-Verfahrens (also zur Laufzeit) auch gekoppelt mit einer Fehlerkorrektur (→ 14.3), sowie Verfahren zur formalen Verifikation (→ 16).

13 Softwaremetriken

Die Geschichte der modernen Naturwissenschaften ist geprägt von den Fortschritten in der Messtechnik. Ohne präzise Messungen könnten keine theoretischen Modelle entwickelt werden, die wiederum Vorhersagen treffen, deren Nachweis wiederum präzise Messtechnik voraussetzt. Mit anderen Worten: (Mess-)Technik und Theorie bedingen sich gegenseitig.

Eine Metrik im mathematischen Sinn ist dabei eine Funktion d (Distanz) auf einer Menge X, die folgende Eigenschaften hat:

$d: X \times X \rightarrow R$ mit

(1) $d(x, y) \geq 0$ und $d(x, y) == 0 \leftrightarrow x == y$

(2) $d(x, y) == d(x, y)$

(3) $d(x, y) \leq d(x, z) + d(z, y)$ für alle $x, y, z \varepsilon X$ (Dreiecksungleichung)

Was die Entwicklung und vor allem die Begutachtung von Software angeht, scheinen jedoch nur wenige und auch nur sehr unpräzise Metriken und Messmethoden vorhanden zu sein. Dabei wären für Reviews (→ 14.4), qualitative Beurteilungen oder auch nur Planungen von Tests (→ 15) quantitative Messmittel von großem Vorteil.

Ziel und Inhalt dieses Kapitels ist die Darstellung des aktuellen Stands der Softwaremetriken, verbunden mit einer Diskussion, wie diese Metriken produktiv eingesetzt werden können [FDS09][CL07]. In einer groben Einteilung können diese Softwaremetriken in

- Produkt-Metriken
- Prozess-Metriken
- Projekt-Metriken

eingeteilt werden. In diesem Kapitel soll dabei auf eine Prozess-Metrik (Functional Points) sowie auf verschiedene Produkt-Metriken, im Softwarebereich auch Code-Metriken genannt, eingegangen werden.

13.1 Prozess-Metriken

Eine der wichtigsten Fragen in Zusammenhang mit Softwareprozessen ist die Aufwandsabschätzung. Hierbei geht es darum, den Erstellungsaufwand inklusive des Testaufwands á priori zu schätzen, um auch die Projektkosten und die Wirtschaftlichkeit beurteilen zu können. Eines der bekanntesten Verfahren ist das Function-Point-Verfahren, das im Folgenden vorgestellt werden soll.

13.1.1 Einführung Function-Point-Verfahren [Wiki_FP]

Das *Function-Point-Verfahren* (auch FP-*Analyse* oder FP-*Methode*, kurz **FPA**) dient zur Bewertung des fachlich-funktionalen Umfangs eines informationstechnischen Systems, meist als Anwendung bezeichnet. Das Ergebnis einer Function-Point-Bewertung wird als *Functional Size* bezeichnet und in der Einheit *Function Points*, kurz fp, angegeben.

Die Functional Size dient dann als Basis für Aufwandsschätzung, Benchmarking und allgemein zur Ableitung von Kennzahlen zur Produktivität und Qualität. Eine Function-Point-Bewertung ist dabei unabhängig von der zu Grunde liegenden Technologie der Anwendung, ähnelt aber eher der Strukturierten Analyse (für imperative bzw. funktionale Programmentwicklung) als der Objekt-orientierten Analyse (OOA).

Der Bestimmung der Functional-Size liegt eine Zerlegung der funktionalen Anforderungen an eine Anwendung zugrunde. Die Zerlegung soll dabei in kleinste, für den Anwender sinnvolle Aktivitäten, die so genannten Elementarprozesse, erfolgen. Jedem Elementarprozess wird dann ein definierter Punktwert (→ 13.1.2.5) zugeordnet, gleiche Elementarprozesse werden nur einmal gewertet. Die Summe der Punktwerte aller Elementarprozesse ergibt die Functional Size.

13.1.2 Bestimmungsmethode

13.1.2.1 Anwendersicht

Die Bestimmung der Functional Size geht von der Anwendersicht (*User View*) aus. Der Begriff des Users in der FPA entspricht konzeptuell dem Akteur aus dem Requirement Engineering, z.B. bei UML (Unified Modelling Language). Ein User kann also eine natürliche Person, eine andere Software oder beispielsweise eine Maschine sein. Die Anwendersicht fokussiert sich darauf, dass bei der Bewertung nur diejenigen Funktionen der Software zu berücksichtigen sind, die der Unterstützung der jeweiligen Prozesse dienen.

13.1.2.2 Identifikation der Transaktionen

Definition 13.1:

Ein **Elementarprozess** ist definiert als die für den Anwender sinnvollste, kleinste Aktivität, die das System nach ihrer Ausführung in einem *konsistenten Zustand* lässt.

Diese Elementarprozesse werden nach drei Transaktionstypen unterschieden:

- Eingabe (*External Input*, EI)
- Ausgabe (*External Output*, EO)
- Abfrage (*External Inquiry*, EQ)

Entscheidend ist dabei der *Hauptzweck* des Elementarprozesses. Hierzu werden folgende Begriffe definiert:

Definition 13.2

Eine ***Eingabe*** (*External Input, EI*) hat als Hauptzweck die Pflege eines internen Datenbestandes und die Verarbeitung von Daten, die von außerhalb der Anwendung stammen.

Definition 13.3

Eine ***Ausgabe*** (*External Output, EO*) oder eine ***Abfrage*** (*External Inquiry, EQ*) haben jeweils den Hauptzweck der Präsentation von Informationen an der Anwendungsgrenze. Für eine *Ausgabe* ist zusätzlich gefordert, dass ihre Verarbeitungslogik mathematische Berechnungen oder Formeln, die Bildung abgeleiteter Daten, die Pflege eines internen Datenbestands oder eine Veränderung des Systemverhaltens beinhaltet.

Aus der Definition der *Transaktionen* ergibt sich, dass nur solche Elementarprozesse bewertet werden, die im Zusammenhang mit einem Datenfluss über die Anwendungsgrenze stehen.

Gleiche Transaktionen sollen nur einmal gewertet werden. Zwei Transaktionen gelten dann als gleich, wenn sie die gleichen Daten verwenden und die gleiche Verarbeitungslogik beinhalten, wobei auch kleinere Variationen nicht ausgeschlossen sind. Variationen gelten auf jeden Fall dann nicht mehr als *klein*, wenn den beiden Transaktionen zwei erkennbar unterschiedliche fachlich-funktionale Anforderungen zu Grunde liegen.

13.1.2.3 Identifikation der Datenbestände

Neben den Transaktionen bewertet die FPA auch die durch die Software verwalteten Datenbestände (*data functions*). Ein Datenbestand ist als eine Menge fachlich erkennbarer und logisch zusammengehöriger Daten definiert. Auch hier gilt, dass die Bewertung aus Anwendersicht erfolgen soll. Datenbestände werden weiterhin unterschieden in

- Interne Datenbestände (*Internal Logical File*, kurz ILF) und
- Externe Datenbestände (*External Interface File*, EIF).

Definition 13.4

Interne Datenbestände (*Internal Logical File, ILF*) sind solche Datenbestände, die innerhalb der bewerteten Anwendung gepflegt werden, auf denen also ein schreibender und lesender Zugriff stattfindet. ***Externe Datenbestände*** (*External Interface File, EIF*) werden von der bewerteten Anwendung nur referenziert oder gelesen, aber in einer anderen Anwendung gepflegt.

13.1.2.4 Funktionaler Hierarchiebaum

Nachdem die Transaktionen und die Daten im Einzelnen identifiziert worden sind, kann das Ergebnis u.a. in Form eines funktionalen Hierarchiebaums dargestellt werden. In dem folgenden Beispiel wurde das anhand der Diskreten Fourieranalyse (DFA), die bereits in einigen anderen Abschnitten erwähnt wurde, durchgeführt.

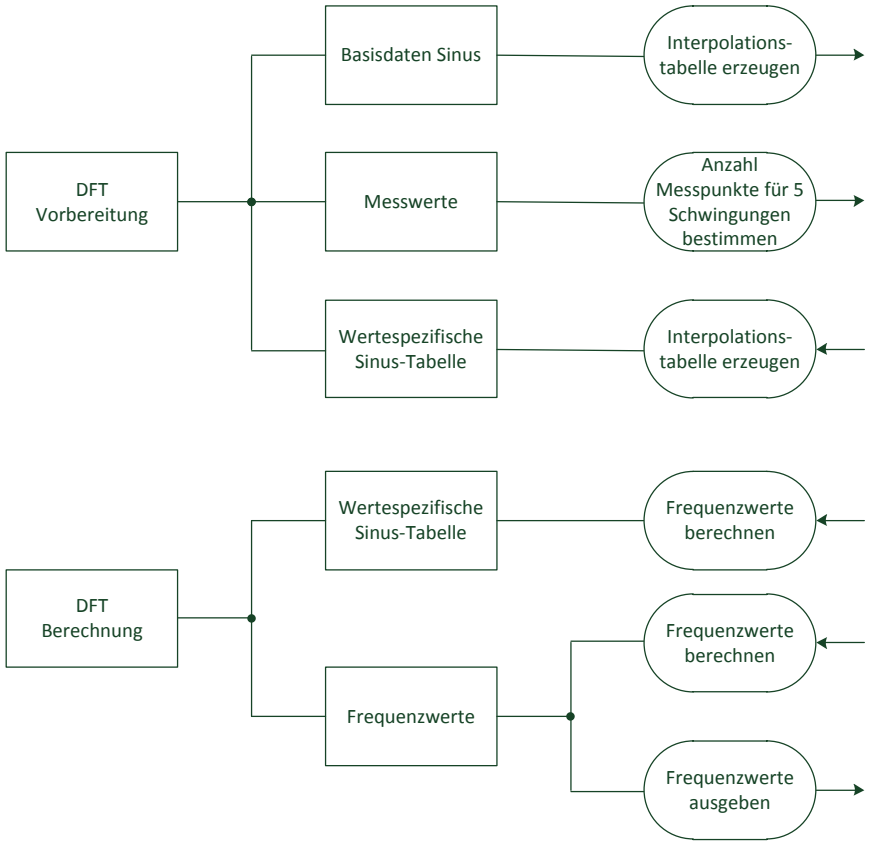


Bild 13.1 Funktionaler Hierarchiebaum am Beispiel DFT

13.1.2.5 Ermittlung der Functional-Size für eine Anwendung

Für die Zuordnung des Punktwerts zu Transaktionen und Datenbeständen gibt es Regeln, die so genannten *Komplexitätsregeln*. Der Punktwert für eine Transaktion ergibt sich aus der Anzahl der verwendeten Felder und der Zahl der in der Transaktion verwendeten Datenbestände. Für die Datenbestände wird der Punktwert aufgrund der Anzahl der enthaltenen Felder und der Anzahl von Feldgruppen bestimmt. Als Feldgruppe wird eine Menge fachlich zusammenhängender Daten-

felder verstanden, zum Beispiel die Menge der Felder *Anrede, Titel, Vorname und Nachname*, die zusammen den Namen einer natürlichen Person darstellen.

Die folgende Tabelle zeigt die für die einzelnen Transaktionstypen und Datenbestandstypen möglichen minimalen, mittleren und maximalen Punktwerte. Die letzte Spalte enthält Werte für die so genannte Schnellbestimmung (Rapid Prototyping). Dieses Verfahren liefert insbesondere für größere Projekte (> 100 Punkte) gute Werte, da sich die Differenzen zu den präziseren Schätzwerten ausgleichen.

Elementarprozess	Minimaler Punktwert	Mittlerer Punktwert	Maximaler Punktwert	Rapid Punktwert
Eingabe	3	4	6	4
Ausgabe	4	5	7	5
Abfrage	3	4	6	4
Interner Datenbestand	7	10	15	7
Externer Datenbestand	5	7	10	5

Tabelle 13.1 Punktewerte für die Berechnung der Functional Size

Die Summe der Punktwerte aller Transaktionen und Datenbestände ist dann die Functional-Size der Anwendung.

Beispiel DFT: Die Gesamtapplikation wurde in zwei Teile eingeteilt: Vorbereitung und Berechnung. Die Messwerte und die Basisdaten Sinus sind externe Datenbestände, denn sie werden in der DFT-Vorbereitung wie -Berechnung nur gelesen. Beide werden mit einem mittleren Punktewert von 7 bewertet. Die Frequenzwerte und die Werte-spezifische Sinus-Tabelle hingegen sind interne Datenbestände; sie werden mit jeweils 10 bewertet.

Weiterhin sind für die DFT-Vorbereitung 1 Input- und 2 Output-Transaktionen enthalten, die zusammen – bewertet mit jeweils dem mittleren Wert – 14 Punkte ergeben. Somit hat diese Applikation einen fp-Wert von 38.

Für die DFT-Berechnung und Ausgabe sind 2 Transaktionen, je einmal input und Output, vorgesehen, und sie wirkt lesend auf der Werte-spezifischen Sinustabelle und schreibend auf die Frequenzwerte. Dies ergibt zusammen 26 Punkte.

13.1.3 Functional-Size für Softwareanpassungen und -erweiterungen

Die bisherige Darstellung geht von einem vollkommen neu zu erstellenden Produkt aus. Diese Sichtweise ist natürlich für Projekte und Testverfahren korrekt, bei denen komplett neu begonnen wird. Häufig ist jedoch eine Weiterentwicklung geplant, so dass auch hierfür eine Functional-Size bestimmt werden muss. Hierzu gibt es einfache Regeln.

Die Functional-Size eines Neuentwicklungsprojekts wird mit dem Ergebnis der durch das Projekt gelieferten Anwendung gleichgesetzt.

Die Functional-Size eines Erweiterungsprojekts ergibt sich als Summe der Punktwerte aller neu hinzugefügten, geänderten und entfernten Transaktionen und Datenbestände. Hierfür wird – für ein Projekt – jede Transaktion und jeder Datenbestand maximal einmal als geändert gewertet, unabhängig vom tatsächlichen Umfang der konkreten Änderungen. Es wird also davon ausgegangen, dass Änderungen – aufwandstechnisch betrachtet – nur einmal durchgeführt werden, also nicht bis „zum, Schönsein geändert wird“. Die Zuordnung der Punktwerte zu den jeweiligen Transaktionen und Datenbeständen erfolgt dabei der oben beschriebenen Regeln für die Ermittlung der Functional-Size einer Anwendung.

Sowohl für Neuentwicklungs- als auch für Erweiterungsprojekte werden, soweit vorhanden, auch die Funktionen bewertet, die der Konvertierung der Daten aus früheren Versionen der Anwendung dienen.

Insgesamt bieten die Regeln eine praxisbewährte Methode zur Bewertung von Entwicklungs- und Erweiterungsprojekten.

13.2 Code-Metriken

Mit dem Aufkommen der funktionalen und imperativen Sprachen in den 1970er und 1980er Jahren entstanden auch die ersten Metriken, hier als prozedurale Metriken zusammengefasst. Diese Metriken wurden später erweitert und ergänzt.

13.2.1 Prozedurale Metriken

Prozedurale Code-Metriken können – bedingt durch die verschiedenen Strukturierungselemente in prozeduralen Sprachen – auf verschiedenen Ebenen angewendet werden:

- Prozedurebene (Funktion in C, Methode in C++)
- Modulebene (Datei in C, Klasse in C++)
- Programmebene (gesamtes Projekt)

Diese Metriken können nahezu einschränkungslos auch auf objektorientierte Software angewendet werden. Zu den wichtigsten Vertretern dieser prozeduralen Code-Metriken wählen Lines-of-Code und Cyclomatic Complexity nach McCabe).

13.2.1.1 Lines-of-Code (LOC)

Die LOC-Metrik stellt eine vergleichsweise einfache Maßzahl dar, um die Komplexität der Software zu bestimmen. Zunächst erscheint dieses Maß eher wenig aussagekräftig, da die einfachste Version wirklich nur die Anzahl der Gesamtzeilen im Code bestimmt; aus diesem Grund wird häufig noch detaillierter unterschieden:

- LOCphy: Die Gesamtanzahl der Zeilen (physical lines)
- LOCpro: Die Anzahl der Programmzeilen (program lines). Hierzu zählen auch Definitionen und Deklarationen
- LOCcom: Die Anzahl der Zeilen mit Kommentar (commented lines)
- LOCbl: Die Anzahl der Leerzeilen (blank lines). Leerzeilen innerhalb eines Kommentars zählen als Kommentar.

All diese Zahlenwerte können naturgemäß keinen semantischen oder auch nur syntaktischen Inhalt aufnehmen: Ein lineares Programm und ein Programm mit mehreren ineinanderliegenden Schleifen haben nach der LOC-Metrik die gleichen Komplexität, wenn die Zeilenzahlen identisch sind.

Allerdings sind andere Aussagen interessant. Das Verhältnis LOCcom/LOCphy fließt beispielsweise in den Maintability-Index (→ 13.2) ein und ist auch im Rahmen von Code-Reviews interessant; nach [CL07] soll dieses Verhältnis

$$\text{LOCcom/LOCphy} \approx 0{,}30 \,..\, 0{,}75$$

betragen, d.h. 30 bis 75 % aller Zeilen sollten Zeilen mit Kommentar sein. Weitere Empfehlungen [CL07] [Hol06] werden für LOCphy gegeben:

$$\text{LOCphy(Funktion)} \leq 40 \,..\, 60$$

$$\text{LOCphy(Datei)} \leq 1400$$

Wohlgemerkt: Dies sind Empfehlungen, die in Projekten oder bestimmten Anwendungsgebietet auch zu Regeln gemacht werden können.

13.2.1.2 Cyclomatic Complexity nach McCabe

Die *strukturierten Integrationstests* (SIT) wurden 1982 von Thomas J. McCabe eingeführt. Sie beruhen darauf, die minimal notwendige Anzahl von voneinander unabhängigen Programmpfaden zu bestimmen. Unabhängig ist dabei ein Programmpfad, wenn er nicht durch eine Linearkombination anderer Programmpfade darstellbar ist.

Thomas J. McCabe legte bei seiner Maßzahl (für die Anzahl der unabhängigen Programmpfade) den gerichteten Kontrollflussgraphen G einer Funktion zugrunde [McC76]. Ausgehend von einem gerichteten Kontrollflussgraphen G kann seine

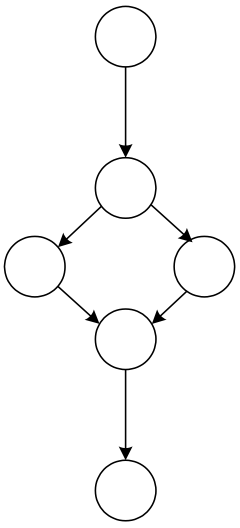
Komplexitätsmetrik, auch zyklomatische Komplexitât (Cyclomatic Complexity, CC, auch als v(G) bezeichnet) genannt, wie folgt bestimmtwerden:

$$v(G) = CC = E - N + 2$$
$$mit\, E = Anzahl\ der\ Kanten, N = Anzahl\ der\ Knoten \tag{13.1}$$

Mit *E* wird hierbei die Anzahl der Kanten (edges), mit *N* die Anzahl der Knoten (nodes). Diese Formel (13.1) stellt die Komplexität *einer* Prozedur dar, die also in dem einzigen Kontrollflussgraphen dargestellt ist. Sie kann auf mehrere Prozeduren (= Kontrollflussgraphen) erweitert werden, wenn *E* und *N* weiterhin die (Gesamt-)Anzahl der Kanten und Knoten angeben und mit *P* die Anzahl der Prozeduren (oder Funktionen) bezeichnet wird:

$$v(G) = CC = E - N + 2 \cdot P$$
$$mit\, E = Anzahl\ der\ Kanten, N = Anzahl\ der\ Knoten, \tag{13.2}$$
$$und\ P = Anzahl\ der\ Prozeduren\ (= Kontrollflussgraphen)$$

Gl. (13.1) ergibt sich aus (13.2) durch Setzen von $P = 1$. Bild 13.2 zeigt einen Graphen mit CC = 2;



```
int rundung( int a )
{
   int temp;
   temp = a%10;



   if( temp < 5 )
      a = a - temp;
   else
      a = a + 10 - temp;



   return( a );
}
```

Bild 13.2 Kontrollflussgraph einer Beispielfunktion mit v(G) = 2

Ein anderer Zugang zur Komplexitätsbestimmung von Prozeduren/Funktionen ist in Gl. (13.3) beschrieben. Betrachtet man nun einen einzigen Kontrollflussgraphen mit ausschließlich binären Verzweigungen (also Verzweigungen mit genau 2 Ausgängen), so gilt:

$$ECC = B + 1$$
$$mit\, B = Anzahl\, der\, binären Verzweigungen \tag{13.3}$$

ECC – die Extended Cyclomatic Complecity – gibt die untere Schranke für die Anzahl der im Code vorhandenen Wege an. Für den Test – eines der Hauptziele der CC-Metrik – gibt M zugleich die obere Schranke der Testfälle an, die zur vollständigen Kantenabdeckung notwendig sind.

Kritische Betrachtung der Cyclomatic Complexity
In die v(G) = CC gehen ausschließlich Verzweigungen – sei es als binäre und multiple Verzweigungen, sei es als Schleifen – ein. Linearer Code zwischen zwei Verzweiungen gehört zu einem Knoten und trägt somit nicht zur CC bei, obwohl auch hier in erheblichem Maße zur (eigentlichen) Komplexität des Programms beigetragen wird. Dieses Problem wird in Kapitel 14 nochmals aufgegriffen.

```
int k, fibo;
…
switch( k )
{
     case 0:  fibo = 0; break;
     case 1:  fibo = 1; break;
     case 2:  fibo = 1; break;
     case 3:  fibo = 2; break;
     case 4:  fibo = 3; break;
     case 5:  fibo = 5; break;
     case 6:  fibo = 8; break;
     case 7:  fibo = 13; break;
     case 8:  fibo = 21; break;
     case 9:  fibo = 34; break;
     case 10: fibo = 55; break;
     default: fibo = -1; break;
}
```

Bild 13.3 Beispielprogramm mit v(G) == 12 (Fibonacci-Zahlen)

Bei den Abfragen selbst werden Codesequenzen wie

`if( a > 10 )` und

`if( (a > 4  &&  b > a+1)  ||  (c < 0) )`

identisch behandelt, obwohl in der zweiten Abfrage sehr viel mehr Komplexität steckt. Dies wird in der ECC (bitte nicht mit der Error-Checking-and-Correction-Summe aus 14.3 verwechseln) berücksichtigt, indem die binären Entscheidungen `a > 4`, `b > a+1` und `c < 0` ebenfalls berücksichtigt werden.

Problematisch bleibt weiterhin die Bedeutung der konkreten CC- bzw. ECC-Werte. McCabe gab selbst an, dass ein v(G)-Wert von 10 nicht überschritten werden

sollte, weil dann die Komplexität des Programms zu groß wäre, um wirklich beherrschbar zu sein. Dieser Wert von 10 wird durchaus kritisiert, weil z.B. die in Bild 13.2 dargestellte Sequenz diesen Wert überschreitet – wobei das Programm keineswegs unübersichtlich wirkt. Der Zahlenwert allein ist somit offensichtlich nicht signifikant als „Komplexitätsmaß", wohl aber als Wert für die Anzahl der Testfälle, die zwecks vollständiger Codeabdeckung zu durchlaufen sind.

13.2.1.3 Halstead-Metriken

Maurice Howard Halstead [Hal77] verfolgte in seinen Arbeiten einen anderen Ansatz, verglichen mit der CC-Metrik. Das Ziel bestand darin, die wirkliche Komplexität z.B. beim Erfassen des Programms (z.B. für einen Code Review, → 14.3) abzubilden.

Die Halstead-Metrik bedient sich hierbei der Annahme, dass ausführbare Programmteile aus Operatoren und Operanden aufgebaut sind – eine Annahme, die durchaus zutreffend ist. Die Definition, was die zu betrachtenden Operatoren und Operanden sind, ist dabei eine der Aufgaben vor dem Einsatz einer Halstead-Metrik. Typischerweise wird folgendes vereinbart:

Operanden: Variablen, Konstanten, Datentypen, Funktionsaufrufe, Funktionsnamen

Operatoren: Schlüsselwörter, Operatoren der jeweiligen Sprache, logische und Vergleichsoperatoren, Klammern, Kommata usw..

Es werden dann für jedes Programm folgende Basismaße gebildet:

- Anzahl der verwendeten *unterschiedlichen* Operatoren (*n1*) und Operanden (*n2*), zusammen die Vokabulargröße *n*.
- Anzahl der *insgesamt* verwendeten Operatoren (*N1*) und Operanden (*N2*), zusammen die Implementierungslänge *N*.

Hieraus werden dann die Größen Halstead-Länge (*HL*) und Halstead-Volumen (*HV*) errechnet:

$$\begin{aligned} HL &= n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2 \\ HV &= N \cdot \log_2 n \end{aligned} \qquad (13.4)$$

Aus den Basisgrößen kann man verschiedene Kennzahlen berechnen, z. B.:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2} \qquad (13.5)$$

D (difficulty) steht für die Schwierigkeit, ein Programm zu schreiben oder zu verstehen. Hieraus wurden weitere Messgrößen abgeleitet:

$$E = D \cdot HV\,(Aufwand, Effort) \qquad (13.6)$$

$$B = E^{2/3} / 3000\,(Schätzung\ der\ Anzahl\ mi gelieferter\ Fehler) \qquad (13.7)$$

Die Halstead-Metrik liefert oftmals sehr gute Werte für die Komplexität von Programmen, was insbesondere im Hinblick auf Code Reviews wichtig ist.

13.2.2 Übergreifende Metriken

Die bislang diskutierten Metriken bewerten immer nur einen Teilaspekt des Softwarecodes. Zu jeder dieser Metriken lassen sich zudem auch Gegenbeispiele konstruieren, die die Metrik scheinbar ad absurdum führen – sprich: fragwürdige Ergebnisse liefern.

Diese Problematik soll in übergreifenden Metriken behoben werden.

13.2.2.1 Maintainability-Index

Der Maintability-Index *MI* wurde von der University of Idaho entwickelt. Er betrachtet die bislang vorgestellte Metriken CC, LOC und Halstead als strafend, so dass von einem positiven Startwert ausgehend durch schlechte Werte der Metriken der erhaltene Maintability-Index entsprechen gemindert wird. Die exakte Formel lautet für den *MIwoc* (Maintability Index without Comments):

$$MIwoc = 171 - 5{,}2 \cdot \ln(HV) - 0{,}23 \cdot CC - 16{,}2 \cdot \ln(LOCphy) \quad (13.8)$$

Berücksichtigt man nur Kommentare, so wird hierfür der Index *MIwc* (MI with Comments) wie folgt berechnet:

$$MIwc = 50 \cdot \sin(\sqrt{2{,}4 \cdot (LOCcom / LOCphy)}) \quad (13.9)$$

Der gesamte Maintability-Index ist dann die Summe der beiden Indizes:

$$MI = MIwoc + MIwc \quad (13.10)$$

Natürlich ist auch der Maintability-Index kritisch zu hinterfragen. So haben die Gleichungen (13.8) bis (13.10) einen stark empirischen Hintergrund; allein der bestmögliche Wert von 171(+50) überrascht doch sehr.

Weiterhin ist zu fragen, ob die Berücksichtigung des Kommentars wirklich zur Wartbarkeit beiträgt, zumindest in dieser Weise. Letztendlich kann ein auskommentierter Codebereich so den *MI* erhöhen, was nicht der Fall sein kann.

Dennoch: MI bzw. MIwoc sind die besten Metriken für die Wartbarkeit, die derzeit erhältlich sind. Programme mit Werten für MIwoc (Bestwert: 171) von über 85 gelten als gut wartbar, zwischen 65 und 85 als mäßig wartbar, unterhalb von 65 als schlecht wartbar. Es sei darauf verwiesen, dass MIwoc auch negative Werte annehmen kann.

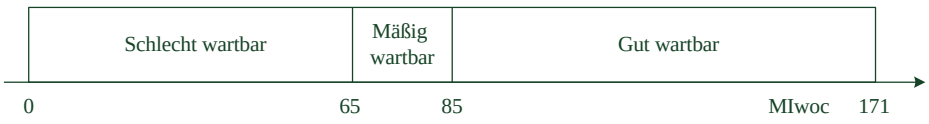


Bild 13.4 Klassifizierung des Maintability Index *MIwoc*

13.2.2.2 Maximale Schachtelungstiefe MaxND

Neben der Kritik an der Cyclomatic Complexity (und der Extended CC), dass linearer Code nicht in das Komplexitätsmaß eingeht, muss ebenfalls daran Kritik geübt werden, dass tief ineinander geschachtelte Kontrollstrukturen ebenfalls fehleranfällig sind – und lediglich linear beachtet werden.

Um hier einen besseren Überblick zu haben, wird die maximale Schachtelungstiefe MaxND eingeführt. Folgendes Beispiel zeigt zwei (unterschiedliche) Funktionen:

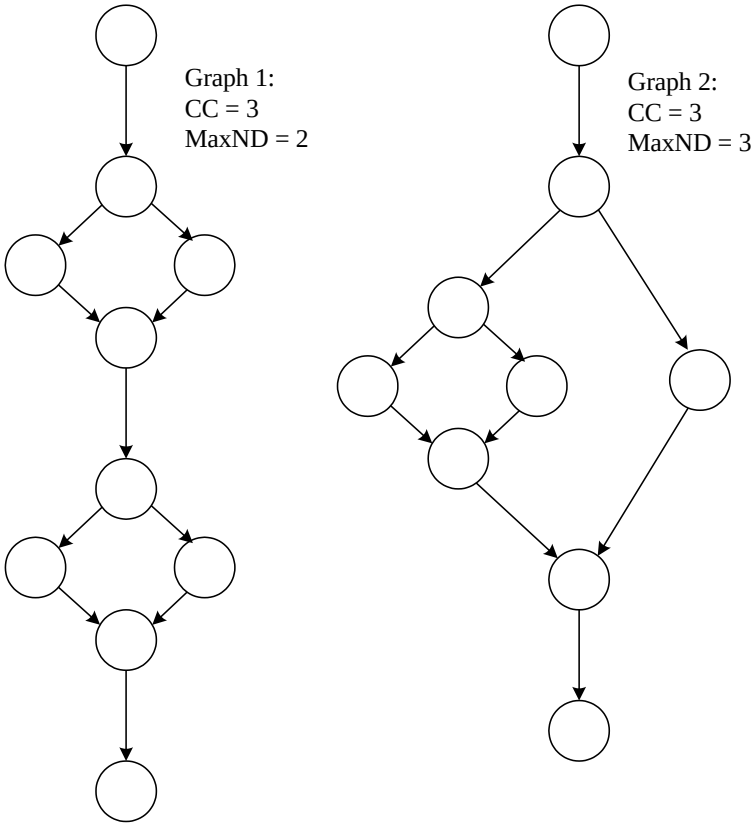


Bild 13.5 Funktionsgraphen mit unterschiedlichen MaxND-Werten [FDS09]

Obwohl die Codeabschnitte in Bild 13.5 gleich CC-Werte besitzen, unterscheidet sich der MaxND-Wert voneinander. Ursache hierfür ist die Struktur der Codeabschnitte, die im Graphen 1 eine Art Sequentialität, in Graphen 2 Parallelität aufweisen. In jedem Fall gilt die Relation

$$v(G) = CC \geq MaxND \tag{13.11}$$

In der Praxis zeigen die MaxND-Werte meist eine Beschränkung auf Werte ≤ 6.

13.2.2.3 Anzahl der Ubergabeparameter

Eine weitere Metrik, die eher weich formuliert ist, betrifft die Anzahl der Übergabeparameter bei Funktionen und Methoden. In [Mar09] wird eine Beschränkung auf Werte < 3 empfohlen.

Ein guter Grund hierfür ist die tatsache, dass bei Funktionen/Methoden alle Eingangsparameter und –kombinationen getestet werden müssen (→ 15.1, 15.2), so dass eine Beschränkung der Anzahl der Kombinationen dringend geboten erscheint. Andererseits kann man natürlich nicht die Algorithmen ändern, und eine Abbildung der vermiedenen Parameter auf (Datei- oder Objekt-)globale Daten ist natürlich auch keine Lösung.

13.2.3 Metriken und Zielgruppen

Bei dem aktuellen Stand der Forschung existiert keine universelle Metrik, wie aus den vorangegangenen Darstellungen deutlich geworden sein dürfte, auch nicht auf dem Gebiet der Codem
-Metriken. Somit stellt sich die Frage, wann welche Metrik anzuwenden ist.

Für das (höhere) *Management* werden Metriken vorzugsweise in verdichteter Form dargestellt. Die diversen LOC-Werte beispielweise ergeben für ein größeres Projekt eine Verteilung von Werten, die in Form von Häufigkeitsverteilungen gut darstellbar sind. Hierüber lassen sich gut allgemeine Qualitätsaussagen für das Projekt treffen.

Im *Projektmanagement* treten andere Fragestellungen in den Vordergrund. Im Anfangsstadium beispielsweise sind Aufwandsabschätzungen wichtig, die oft in Form von Function Points (FP) gegeben werden (→ 13.1). Zwischen der Codemetrik LOC und den Function Points gibt es aber einen zumindest mittelbaren Zusammenhang, der sich auch quantifizieren lässt. Nach [Jon00] gilt nämlich eine durchschnittlich Anzahl von LOC zu FP von:

Assembler:	320 LOC/FP
C:	128 LOC/FP
C++:	53 LOC/FP

(Anmerkung: Die Verhältniswerte LOC/FP werden in [Jon00] auch für weitere Sprachen geliefert)

Weiterhin sind im Projektmanagement mehrere Releases bzw. Versionen zu überwachen. Aus diesem Grund wird hier gerne die Fortentwicklung der Metriken für Funktionen, Module etc. beobachtet, auch, um wesentliche Eingriffe in die Entwicklung zu dokumentieren.

Besondere Wertschätzung besitzen Codemetriken auch bei Reviews (→ 14.4). Im Review-Prozess sollen gerade kritische Code-Abschnitte unter Augenschein genommen werden, und die Entscheidung, was als kritisch gilt, kann anhand einiger Code-Metriken gefällt werden.

Insbesondere der Wartbarkeitsindex MIwoc (→ 13.2.2.1) und die Verschachtelungstiefe MaxND (→ 13.2.2.2) können hier gute Anhaltspunkte liefern, falls deren Werte hoch sind bzw. herausragen.

Für den *Softwaretest* ist die zyklomatische Komplextät v(G) (bzw. CC) von hohem Interesse. CC repräsentiert die Anzahl der linear unabhängigen Pfade einer Funktion oder Methode. Daraus lässt sich der Testaufwand abschätzen, um eine 100prozentige Code Coverage zu erreichen.

Weiterhin sind Funktionen mit einer hohen Anzahl an Übergabeparametern (→ 13.2.2.3) und Funktionen mit einem hohen MaxND Kandidaten für größere Fehleranfälligkeit und somit für höheren Testaufwand.

14 Software- und Systemqualität

Eingebettete Systeme sind immer Bestandteil einer übergeordneten Maschine; Fehler in der Software dieser Systeme können also zu Schädigungen der Maschine und von Menschen führen. Dies allein ist sicher Motivation genug, in die Softwarequalität zu investieren.

Dies ist eine hehre Aufgabenstellung, die schnell formuliert und schwierig umzusetzen ist. Zunächst werden Begriffe erläutert und Definitionen gegeben. Speziell auf das Thema Zuverlässigkeit zugeschnitten ist der nächste Abschnitt, gefolgt von einem Kapitel zum anderen Blickwinkel: Die Sicht der Maschine (bzw. Maschinenbauer). Den Abschluss bildet ein Vorschlag für Codierungsregeln in Projekten mit sicherheitskritischer Software.

14.1 Beispiele, Begriffe und Definitionen

14.1.1 Herausragende Beispiele

Leider gibt es einige herausragende, sehr bekannte Beispiele dafür, dass ein Software-basiertes System nicht ordnungsgemäß funktioniert hat. Hierzu zählen die Bruchlandung eines Airbus A-320 auf dem Warschauer Flughafen am 14.09.1993 und der Absturz der Ariane-5 am 04.06.1996 in Kourou, Französisch-Guayana.

Beim Beispiel der Bruchlandung des Airbus A-320 war die Ursache eine fehlerhafte Bodenberührungserkennung im Flugzeug. Bedingt durch plötzlich auftretenden, starken Seitenwind setzte der Airbus mit nur einem Rad auf dem Boden auf, die Software erkannte dies nicht als Bodenkontakt an und schaltete nicht aus dem Flight Mode heraus. Die Piloten konnten somit keine Schubumkehr einschalten, das Flugzeug kam nur wenig gebremst von der Landebahn ab, fing Feuer, so dass 2 Menschen starben und 54 verletzt wurden.

Der Fehler lag in der Entscheidung der Konstrukteure und Software-Ingenieure, wie die Messungen der Bodensensoren interpretiert wurden. Der aufgetretene Fall war nicht abgedeckt, und somit kam es zum Unglück.

Im zweiten Fall musste die europäische Trägerrakete Ariane 5 bei ihrem Jungfernflug gesprengt werden, weil sie von ihrer geplanten Bahn stark abwich und in bewohntes Gebiet abzustürzen drohte. Die Ursache hier war ein nicht abgefangener Datenüberlauf bei der Berechnung der Flugbahn. Die Software war einfach von der Vorgängerrakete übernommen worden, bei der bewiesen werden konnte, dass dieser Überlauf niemals stattfinden konnte. Die Ariane 5 hingegen war schubstärker, und die Rakete erreichte Geschwindigkeiten, deren interne Darstellung 32767 (16 bit Integer mit Vorzeichen) überschritt. Der Datenunterlauf führte dann

zur Bahnabweichung und zur Sprengung. Ein Klassiker unter den Softwarefehlern, der mithilfe von Datenbereichskontrollen hätte abgefangen werden können.

Beide Fehler resultierten in Tod, Verletzung oder Gefährdung von Menschen sowie in erhebliche wirtschaftliche Verluste, Kriterien dafür, dass die Systeme sicherheitskritisch waren.

14.2 Grundlegende Begriffe und Definitionen

Als zentral in einem modernen Projekt wird heute die Softwarequalität erachtet. Dabei stellt sich natürlich die Frage, was darunter eigentlich zu verstehen ist:

Definition 14.1 [ISO/IEC 9126]:

Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Konkret wird die Beurteilung erst dann, wenn man sich auf die Qualitätsmerkmale bezieht. Diese stellen Eigenschaften einer Funktionseinheit dar, anhand deren ihre Qualität beschrieben und beurteilt wird. Allerdings enthalten sie keine Aussage über den Grad der Ausprägung. Beispielsweise existieren folgende **Softwarequalitätsmerkmale** (die im Übrigen miteinander in Wechselwirkung stehen oder voneinander abhängig sein können):

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Die nachfolgenden Definitionen stellen klar, was unter Softwarefehlern bzw. Fehlern allgemein verstanden wird. Hierbei wird zwischen tatsächlich auftretenden Fehlern, möglichen Fehlern und fehlerhaften Handlungen, die zu den beiden erstgenannten führen können, unterschieden:

Definition 14.2:

Failure (*Fehlverhalten, Fehlerwirkung, äußerer Fehler***)**: Hierbei handelt es sich um ein Fehlverhalten eines Programms, das während seiner Ausführung auch wirklich auftritt.

Definition 14.3:

Fault (*Fehler, Fehlerzustand, innerer Fehler*): Es handelt sich um eine fehlerhafte Stelle eines Programms, die ein Fehlverhalten auslösen kann.

Definition 14.4:

Error (*Irrtum, Fehlhandlung*): Es handelt sich um eine fehlerhafte Aktion, die zu einer fehlerhaften Programmstelle führt.

Daraus ergibt sich, dass Fehlhandlungen (*errors*) bei der Programmentwicklung oder durch äußere Einflüsse (z.B. Höhenstrahlung, Hardwareprobleme z.B. bei Flash-EEPROM-Zellen oder durch Bauteilestreuungen) zu Fehlern (*faults*) im Programm führen, die ihrerseits zu einem Fehlverhalten (*failure*) bei der Ausführung führen können. Hier soll die Qualitätssicherung entgegenwirken, und zwar sowohl konstruktiv als auch analytisch.

Um die Definitionen für *Validierung* und *Verifikation* zu verstehen, muss man den kompletten Designprozess betrachten (Bild 14.1). Aus einer informellen Problembeschreibung folgt eine formale Anforderungsdefinition, aus der heraus dann das eigentliche Rechnersystem (z.B. mit Mikroprozessor und Software) konstruiert wird. Die Übereinstimmung von Problem und Anforderungsbeschreibung ist sehr schwierig festzustellen, allein, weil die Problembeschreibung informell (und damit nicht maschinenprüfbar) ist. Dieser Vorgang wird Validierung genannt.

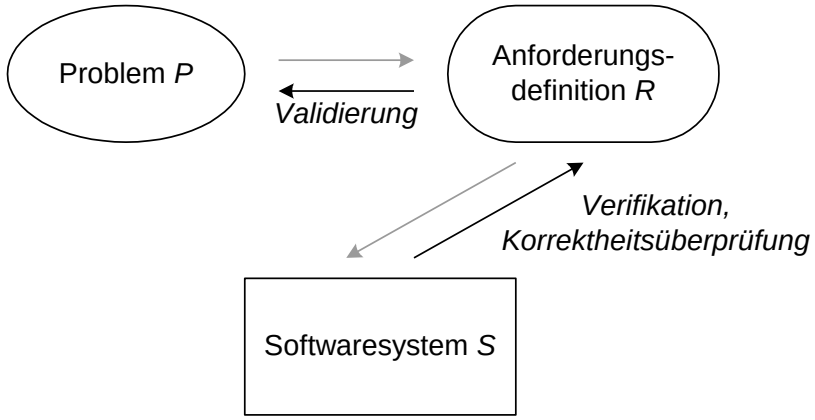


Bild 14.1 Einordnung der Begriffe Validierung und Verifikation

Die Verifikation hingegen in grundsätzlich durch formales Vorgehen lösbar, allerdings oft ebenfalls mit Schwierigkeiten. Hierzu sei einmal ein Software-basiertes System betrachtet: Eine logisch/arithmetische Anforderungsdefinition etwa in UML kann durch eine geeignete Software gegen ein daraus entstandenes Softwaresystem verifiziert werden (bzw. umgekehrt), mehr noch: Aus einer solchen Anforderungsdefinition kann mithilfe von Codegeneratoren das Softwaresystem sogar erzeugt werden.

Weitere Randbedingungen hingegen, wie sie z.B. in Form von zeitlichen Randbedingungen (Echtzeitsystem) vorliegen, können zwar formalisiert werden, sie sind

jedoch meist nicht funktional (also durch einen Compiler übersetzbar) und im Zielsystem nicht (oder zumindest nur unter weiteren Randbedingungen) formal prüfbar. Hier spielt auch die Systemkonzeption eine große Rolle (→ 3, 4).

Die formale Verifikation ist damit nur ein Bestandteil der Maßnahmen zur Erhöhung der Softwarequalität, der weitaus größere besteht in dem Testen.

14.3 Zuverlässigkeit

Von elektronischen Systemen wird ein hohes Maß an Zuverlässigkeit erwartet. Dieser Satz kann sicherlich als allgemein gültig angesehen werden, aber was ist *Zuverlässigkeit* eigentlich?

Definition 14.5:

Zuverlässigkeit (*reliability*) ist die Wahrscheinlichkeit, dass ein System seine definierte Funktion innerhalb eines vorgegebenen Zeitraums und unter den erwarteten Arbeitsbedingungen voll erfüllt, das heißt intakt ist und es zu keinem Systemausfall kommt.

Definition 14.6:

Die *Verfügbarkeit* (*availability*) eines Systems ist der Zeitraum gemessen am Anteil der Gesamtbetriebszeit des Systems, in dem es für den beabsichtigten Zweck eingesetzt werden kann.

Defintion 14.7:

Ein *Systemausfall* (*failure*) liegt vor, wenn ein System seine geforderte Funktion nicht mehr erfüllt.

Definition 14.8:

Ein *Risiko* ist das Produkt der zu erwartenden Eintrittshäufigkeit (Wahrscheinlichkeit) eines zum Schaden führenden Ereignisses und des bei Eintritt des Ereignisses zu erwartenden Schadensausmaßes.

Mit *Grenzrisiko* wird das größte noch vertretbare Risiko bezeichnet.

Hier sollte ganz deutlich sein, dass das, was noch zumut- oder vertretbar ist, durch die technologische Machbarkeit beeinflusst (bzw. definiert) wird. Dies kann beispielsweise so geschehen, dass eine neue Maschine (z.B. Flugzeug) zugelassen bzw. zertifiziert wird, wenn eine katastrophale Fehlersituation nur noch mit einer Wahrscheinlichkeit von 10^{-9} pro Betriebsstunde auftreten kann, integriert über alle Maschinen dieses Typs. Wie dies berechnet werden kann steht u.a. in den Normen zur Maschinensicherheit (→14.4).

Aus technischer Sicht benötigt man „nur“ ein System, bei dem die Ausfallwahrscheinlichkeit von z.B. 10^{-9} pro Betriebsstunde oder weniger, je nach gefordertem Sicherheitslevel, garantiert ist. Um dies zu erreichen müssen zunächst einmal die

Fehlerursachen, also die *Errors* (→ Definition 14.4) klassifiziert und analysiert werden.

14.3.1 Ursachen des Fehlverhaltens

Grundsätzlich existieren zwei Ursachen für ein Fehlverhalten eines Rechnersystems: Fehler in der Hardware und Fehler in der Software. Dies erscheint als wenig hilfreich, dennoch existieren zwischen Hard- und Softwarefehlern fundamentale Unterschiede.

Fehler im Softwareverhalten entstehen ausschließlich durch die Entwicklung der Software. Der/die EntwicklerIn begeht einen Irrtum (*error*), und zwar in der Formulierung der Anforderungen (Validierungsfehler) oder in der Umsetzung in die Software (Verifikationsfehler). Manche Fehler wirken auch nur indirekt, wie es z.B. bei Compilerfehlern der Fall ist: Hier wird ggf. korrekt geschriebene Software (in Hochsprache) in einer fehlerhafte Software (in Maschinensprache) übersetzt.

Die bekannten Gegenmaßnahmen sind Verifikation und Test (→ 14) mit allen Facetten sowie konstruktive Maßnahmen wie redundante, diversitär entwickelte Systeme (→14.3.2).

Die Fehler in der Hardware entstammen wiederum zwei unterschiedlichen Quellen. Zum einen wird die Hardware auch irgendwann als Design in Form einer softwareähnlichen Beschreibung erstellt und übersetzt und unterliegt damit den gleichen Fehlerquellen wie die Software – mit dem Unterschied, dass hergestellte Hardware meist automatisch durch vielerlei Tests aufgrund des Einsatzes in vielen verschiedenen Projekten durchläuft (während Software spezifisch getestet werden muss, weil sie „einmalig“ zum Einsatz kommt).

Zusammen mit den Herstellungsfehlern, die nicht mehr unbedingt systematisch sind, sondern statistisch verteilt sein können, können diese Fehler durch gleiche oder ähnliche Maßnahmen wie im Fall der Software gefunden werden.

Die andere Quelle für Fehler in der Hardware sind interne oder externe Prozesse, die den Betrieb stören. Hierzu zählen Elektromigration, „Weichwerden“ von Flash-Zellen aufgrund hoher Anzahl von Schreibzyklen, oder so genannte Soft Errors durch Höhenstrahlung oder Radioaktivität [Sie08]. Diese Fehler entstehen dynamisch und können somit nicht durch intensive Tests vor dem Betrieb gefunden werden. Die Wechselwirkung mit z.B. DRAM ist noch nicht restlos quantitativ geklärt [SPW09], grundsätzlich sind die Vorgänge jedoch bekannt und in Bild 14.2 qualitativ dargestellt.

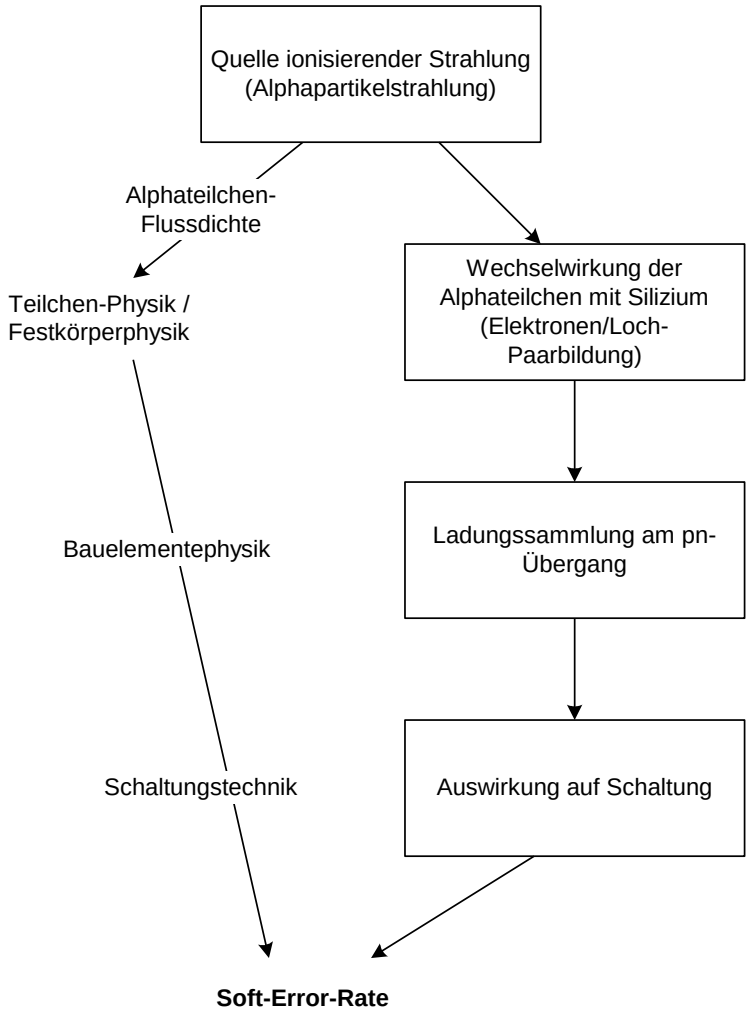


Bild 14.2 Entstehung von Soft Errors durch Höhenstrahlung

Gegen diese dynamischen Fehler wurde eine Reihe von konstruktiven Maßnahmen entwickelt, um sie zu erkennen bzw. sogar im Betrieb dynamisch zu korrigieren.

14.3.2 Konstruktive Maßnahmen

Eine der wichtigsten Fragen für die Konstruktion bzw. das Design sicherheitskritischer Maschinen ist diejenige nach konstruktiven Maßnahmen zur Vermeidung von Fehlern oder wenigstens Fehlerfolgen. Diese Art der Fehlertoleranz basiert

immer auf einer Form der Redundanz, d.h. zur Erkennung von Fehlern sind mehr Informationen als zum eigentlichen Betrieb notwendig, daher wird das System komplexer.

Der naheliegende und vor einigen Jahren auch fast ausschließlich genutzte Ansatz liegt dabei in der Erweiterung der Hardware um fehlererkennende Teile wie Paritätsbits, Prüfsummen, fehlererkennende bzw. –korrigierende Codes usw. Dieser Ansatz wird aktuell jedoch als zu einengend angesehen, so dass man sich nun um Mischformen bemüht.

14.3.2.1 Fehlererkennende und –korrigierende Codes

Die Speicherung und die Übertragung von Daten sind rein statistisch die Vorgänge, bei denen Daten mehrheitlich durch störende Einflüsse verändert werden können. Hier wurde eine Vielzahl von Codes entwickelt, um die Verfälschung zu entdecken (fehlererkennend) bzw. zu beheben (fehlerkorrigierend). Zu unterscheiden sind hierbei Wort- und Block-orientierte Codes.

14.3.2.2 Wort-orientierte Codes

Bei Wort-orientierten Codes wird dem Codewort, also z.B. einem Byte oder einem Doubleword, ein oder mehrere Bits hinzugefügt, um Informationen zur Fehlererkennung bzw. Fehlerbehebung zur Verfügung zu haben.

Paritätsbit

Die einfachste Variante ist das Paritätsbit. Es wird meist über ein Byte (8 bit) berechnet, und zwar als gerades (even) oder ungerades (odd) Paritätsbit. Gerade Parität heißt in diesem Fall, dass die Anzahl der Bits im Byte einschließlich des Paritätsbits, also gezählt über 9 Bits, durch 2 mit Rest 0 teilbar ist.

Ein derartiger Code mit Paritätsbit kann als (9,8,2)-Code bezeichnet werden: Die Anzahl der Bits insgesamt wird an erster Stelle, die der Nutzbits an zweiter Stelle geführt. Die letzte Stelle, hier eine 2, bezeichnet die Hamming-Distanz, d.h. den minimalen Abstand zwischen zwei gültigen Codes. Der (9,8,2)- bzw. abkürzend (9,8)-Code kann ausschließlich 1-Bit-Fehler erkennen (SED, Single Error Detection).

Hamming-Code

Der Hamming-Code weist einem zu schützenden Wort mehrere Paritätsbits zu, und zwar in einer Weise, die die verfälschte Bitstelle (einschließlich aller Prüfbits!) kennzeichnet. Hierzu sind natürlich mehrere Bits notwendig, und zwar abhängig von der Anzahl der zu schützenden Bits. Tabelle 14.1 gibt einen Überblick über die notwendigen Bits.

n Datenbits (Datenwort)	*k* Paritybits (Kontrollstellen)	*N* = n + k Gesamtlänge des Codewortes
1	2	3
4	3	7
11	4	15
26	5	31
57	6	63
120	7	127
247	8	255

Tabelle 14.1 Notwendige Anzahl von Schutzbits für Hammingcode

Die Berechnung und Bedeutung der Schutzbits lassen sich am Besten anhand Bild 14.3 darstellen. Hier sind die Paritätsbits p_1 .. p_5 sowie die Datenbits d_0 .. d_{18} in einer besonderen Reihenfolge, die sich beliebig erweitern lässt, aufgeführt. In der oberen Reihe werden Dezimalzahlen beginnend mit 1 gezählt, und immer, wenn die Dezimalzahl einer Zweierpotenz entspricht, wird darunter ein Paritätsbit, beginnend mit p_1, geführt, ansonsten ein Datenbit. Die Anzahl der notwendigen Paritätsbits muss der Relation

$$k > \log_2(n+k)$$

genügen.

2^0	2^1		2^2				2^3							2^4													2^5					
c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	c_{26}	c_{27}	c_{28}	c_{29}	c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	c_{36}	c_{37}	c_{38}
p_1	p_2	d_0	p_3	d_1	d_2	d_3	p_4	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	p_5	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	d_{17}	d_{18}	d_{19}	d_{20}	d_{21}	d_{22}	d_{23}	d_{24}	d_{25}	p_6	d_{26}	d_{27}	d_{28}	d_{29}	d_{30}	d_{31}

Bild 14.3 Hamming-Code (Beispiel: (38,32)-Hamming Code)

Die Paritätsbits werden nun wie folgt berechnet:

$$c_1 = p_1 = c_3 \oplus c_5 \oplus c_7 \oplus c_9 \oplus c_{11} \oplus c_{13} \oplus c_{15} \oplus c_{17} \oplus c_{19} \oplus c_{21} \oplus c_{23} \oplus c_{25} \oplus c_{27} \oplus c_{29} \oplus c_{31} \oplus c_{33} \oplus c_{35} \oplus c_{37}$$

$$c_2 = p_2 = c_3 \oplus c_6 \oplus c_7 \oplus c_{10} \oplus c_{11} \oplus c_{14} \oplus c_{15} \oplus c_{18} \oplus c_{19} \oplus c_{22} \oplus c_{23} \oplus c_{26} \oplus c_{27} \oplus c_{30} \oplus c_{31} \oplus c_{34} \oplus c_{35} \oplus c_{38}$$

$$c_4 = p_3 = c_5 \oplus c_6 \oplus c_7 \oplus c_{12} \oplus c_{13} \oplus c_{14} \oplus c_{15} \oplus c_{20} \oplus c_{21} \oplus c_{22} \oplus c_{23} \oplus c_{28} \oplus c_{29} \oplus c_{30} \oplus c_{31} \oplus c_{36} \oplus c_{37} \oplus c_{38}$$

$$c_8 = p_4 = c_9 \oplus c_{10} \oplus c_{11} \oplus c_{12} \oplus c_{13} \oplus c_{14} \oplus c_{15} \oplus c_{24} \oplus c_{25} \oplus c_{26} \oplus c_{27} \oplus c_{28} \oplus c_{29} \oplus c_{30} \oplus c_{31}$$

$$c_{16} = p_5 = c_{17} \oplus c_{18} \oplus c_{19} \oplus c_{20} \oplus c_{21} \oplus c_{22} \oplus c_{23} \oplus c_{24} \oplus c_{25} \oplus c_{26} \oplus c_{27} \oplus c_{28} \oplus c_{29} \oplus c_{30} \oplus c_{31}$$

$$c_{32} = p_6 = c_{33} \oplus c_{34} \oplus c_{35} \oplus c_{36} \oplus c_{37} \oplus c_{38}$$

Welche Grundregel liegt diesen Berechnungen zugrunde? Zunächst sind die Codebits c_j, die einem Paritätsbit entsprechen, mit einem als 2^k darstellbarem Index (k = 0, 1, …) versehen. Ein solches Codebit c_j wird dann so berechnet, dass eine XOR-Verknüpfung über alle rechts von diesem Codebit stehenden Codebits c_m (m > j) gebildet wird, wobei in der binären Codierung von m das k-te Bit ($j = 2^k$) auf '1' gesetzt ist.

Bei der Auswertung eines inklusive Paritätsbits gespeicherten oder übertragenen Werts wird anhand der Datenwerte die neuen Paritätsbits berechnet und mit den gespeicherten mittels XOR verknüpft. Der daraus entstehende Bitvektor, auch als Syndrom bezeichnet, bezeichnet dann die Code-Stelle (mit 1 beginnend), an der ein Bitfehler vorliegt, so dass das Bit korrigiert werden kann. Die wird mit Single Error Correction (SEC) bezeichnet.

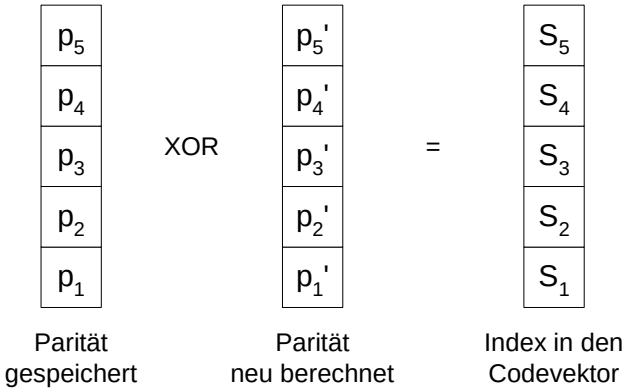


Bild 14.4 Auswertung des Hamming-Codes

Erweiterter Hamming-Code

Der vorgestellte Hamming-Code weist die Hamming-Distanz von 3 auf. Hierdurch kann jeder 1-Bit-Fehler korrigiert werden, 2-Bit-Fehler werden aber nicht unbedingt erkannt. Um dies zu erweitern ergänzt man den Hamming-Code um ein wieteres Paritätsbit zum erweiterten Hamming-Code (extended hamming code) mit der Hamming Distanz 4. Dieser Code kann dann 1-Bit-Fehler garantiert korrigieren (SEC) und 2-Bit-Fehler garantiert erkennen (Double Error Correction, DED).

Das zusätzliche Paritätsbit p_0 wird als gerade Parität über alle Codebits erzeugt und so mitgespeichert. Die folgende Tabelle gibt dann Aufschluss darüber, welche Fehler nach neuerlicher Berechnung aller Paritätsbits (aus den gespeicherten oder übertragenen Werten) wie zu erkennen sind:

Syndromvektor	zusätzliche Paritäts-Prüfung	Aktion des Decoders	Empfangenes Codewort
= **0**	0	kein Fehler	gültig
≠ **0**	1	korrigierbarer Fehler	ungültig
= **0**	1	korrigierbarer Fehler (im Parity-Bit)	ungültig
≠ **0**	0	erkannter, nicht korrigierbarer Fehler	ungültig

Tabelle 14.2 Auswertung des erweiterten Hamming-Codes

Anwendungen des (erweiterten) Hamming-Codes

Der erweiterte Hamming-Code wird z.B. als so genannte Forward Error Correction (FEC) für Halbleiterspeicher (SRAM, DRAM) genutzt. Mithilfe der zusätzlich gespeicherten Bits und einer entsprechende Logik können so 1-Bit-Fehler korrigiert und 2-Bit-Fehler sicher erkannt werden.

Aufgrund der Tatsache, dass bei kleiner Anzahl von Datenbits relativ viele Paritätsbits gespeichert werden müssen, wird der erweiterte Hamming-Code auf 32- oder 64-Bitebene verwendet. Hier kommen dann der erweiterte (39,32)- oder (72,64)-Hamming Code zum Einsatz.

14.3.2.3 Block-orientierte Codes

Ein anderer Ansatz zur sicheren Fehlererkennung besteht darin, auf größeren Blöcken zu arbeiten und dann bestimmte Codes – Prüfsummen – zu verwenden. In einem gewissen Maße können diese Prüfsummen in bestimmten Fällen auch zur Fehlerkorrektur verwendet werden, allerdings wird in den allermeisten Fällen lediglich die sichere Fehlererkennung von 1- und Mehr-Bit-Fehlern genutzt.

Das wohl wichtigste Verfahren ist die zyklische Redundanzprüfung (cyclic redundancy check, CRC) [Wiki_CRC]. Diese beruht auf der Polynomdivision eines Polynoms (entsprechend dem Datenstrom) durch ein defniertes Generatorpolynom. Bei dieser Polynomdivision bleibt ein Rest, der genau die CRC darstellt und mit versendet oder gespeichert wird.

Die Polynome werden über GF(2) – dem Galoisfeld der Ordnung 2 – gebildet. Ein solches Polynome über eine Variable x setzt sich aus der Summe der Potenzen über x zusammen, wobei die Koeffizienten nur die Werte 0 und 1 (wegen GF(2)) annehmen können. Ein Beispiel für ein solches Polynom ist: $x^3 + x + 1$.

Zur Durchführung der Division wird eine Polynom-Addition sowie -Subtraktion benötigt, die aber jeweils nur modulo 2 betrachtet wird unter somit der XOR-Operation entspricht. Somit ist $x^2 + x^2 = 0$, zugleich sind Addition und Subtraktion hierdurch identisch.

Die Polynom-Division wird dann wie die schriftliche Division mit den entsprechenden Regeln zu Addition/Subtraktion durchgeführt, und der dabei entstehende Rest ist dann die Prüfsumme.

Als Beispiel für die Polynom-Division sei das Polynom $(x^7 + x^5 + x^2 + 1)$ durch das (Generator-)Polynom $(x^3 + x^2 + 1)$ dividiert:

$$
\begin{array}{lll}
 & (x^7 + x^5 + x^2 + 1) : (x^3 + x^2 + 1) = x^4 & \\
- & \underline{(x^7 + x^6 + x^4)} & \\
 & (x^6 + x^5 + x^4) & + x^3 \\
- & \underline{(x^6 + x^5 + x^3)} & \\
 & (x^4 + x^3 + x^2 + 1) & + x \\
- & \underline{(x^4 + x^3 + x)} & \\
 & (x^2 + x + 1) & \text{Rest}
\end{array}
$$

Bild 14.5 Polynom-Division am Beispiel

Das Ergebnis ist also $x^4 + x^3 + 1$, der Rest ist $x^2 + x + 1$. Genau dieser Rest wird als Checksumme genommen.

Diese Polynom-Division wird nun so umgesetzt, dass der Binärwert des gesamten Blocks, für den die CRC berechnet werden soll, als Darstellung eines Polynoms gewertet wird. Das least significant bit (lsb) bekommt also die Wertigkeit x^0 im Polynom, das nächste x^1 usw. Das Generatorpolynom (Divisor) ist zwar frei wählbar, wird aber anhand zu erreichender Eigenschaften ausgewählt (siehe auch Tabelle 14.3).

Die Polynom-Division kann aufgrund der einfachen Operationen für Subtraktion und Addition vergleichsweise einfach in Hardware umgesetzt werden. Bild 14.6 zeigt dies für das Generatorpolynom $(x^8 + x^2 + x + 1)$, das für ISDN und ATM eingesetzt wird:

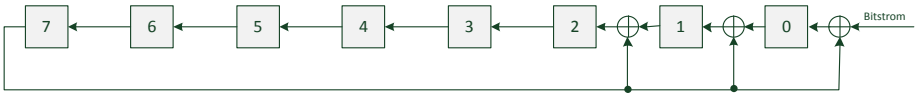


Bild 14.6 Berechnung der CRC-8 in Hardware

Für die Software-Implementierung wird das Generatorpolynom in einem Zahlenwert codiert. Diese Codierung besteht darin, für jeden Punkt der Rückkopplung (siehe Bild 14.6) – und damit für jede vorhandene Potenz im Generatorpolynom – eine ’1‘ im Zahlenwert zu codieren. Hierbei kann der höchste Zahlenwert fortgelassen werden, wobei dann die Anzahl der Bits angegeben werden muss, damit die Rückkopplung definiert ist.

Beispiel: Für die in Bild 14.6 dargestellte CRC-8 wird das Generatorpolynom durch 0x0107 bestimmt. Wird nur 0x07 angegeben, muss die Länge 8 bit angegeben werden.

Weitere Angaben, die für Algorithmen notwendig sind, betreffen

- die Vorbelegung der Checksumme (meist mit 0x00000000, bei CRC-32 im Ethernet aber 0xFFFFFFFF),
- die Rückgabe des Wertes in normaler oder invertierter Form,
- die Reihenfolge der Bits des Datenstroms (lsb first, msb first) und
- die Reihenfolge der Bytes der Checksumme (little endian, big endian, → 9.4.1).

Der Standard-Algorithmus zur Berechnung einer CRC ist in Bild 14.7 angegeben:

```
Schieberegister := 0000… (Startwert)
solange Bits im String verbleiben:
  falls das am weitesten links stehende Bit vom Schieberegister
        ungleich dem nächsten Bit aus dem String ist:
    Schieberegister := (Schieberegister linksschieben um 1, rechtes Bit 0)
                        XOR CRC-Polynom
  andernfalls:
    Schieberegister := Schieberegister linksschieben um 1, rechtes Bit 0
  nächstes Bit im String
Das Schieberegister enthält das Ergebnis.
```

Bild 14.7 Pseudoalgorithmus zur Berechnung einer CRC

```
#define G   0x04C11DB7   // Generator polynomial for CRC32
unsigned long crc32_basic( unsigned char *message, int len )
{
   int i = 0, j;
   unsigned long byte, crc = 0xFFFFFFFF;

   while( i < len )
   {
      byte = message[i];         // Get next byte.
      byte = reverse( byte );    // 32-bit reversal.
      for (j = 0; j < 8; j++)
      {  // Do eight times.
         if ((int)(crc ^ byte) < 0)
            crc = (crc << 1) ^ 0x04C11DB7;
         else
            crc = crc << 1;
         byte = byte << 1;   // Ready next msg bit.
      }
      i = i + 1;
   }
   return reverse(~crc);
}
```

Bild 14.8a C-Code zur Berechnung der CRC32, Basisversion

Der in Bild 14.8a dargestellte Implementierung des Algorithmus für CRC32 stellt eine Basisversion dar, die zum Einsatz zu ineffizient sein wird. Aus diesem Grund

ist eine effiziente Version in Bild 14.8b dargestellt, die bereits alle Spezialitäten wie die Umdrehung der Bitreihenfolge und die Invertierung der Summe eingebaut hat.

Zur Prüfung wird die Division erneut durchgeführt, jetzt mit dem angehängten Rest, und muss folglich 0 ergeben.

```
#define G   0x04C11DB7   // Generator polynomial for CRC32

int32 i32GetCRC( char *val, int32 len )
{
   int32 j, v = 0;

   for( j = 0; j < len; j++ )
   {
      v = ((v<<1) + val[j]) ^ (G & (v>>31));
   }

   return v;
}
```

Bild 14.8b C-Code zur Berechnung der CRC32

Für die praktische Implementierung haben sich folgende, in Tabelle 14.3 dargestellte Generatorpolynome als geeignet erwiesen. Allgemein gelten folgende Regeln zur Fehlererkennung:

1. Die Spalte *MHD* gibt die Minimale Hamming-Distanz an, die zwei Bitfolgen mit gültigem CRC-Wert unterscheidet. Ein CRC-Algorithmus kann also jeden Fehler erkennen, der innerhalb der angegebenen maximalen Länge weniger als *MHD* − 1 Bit-Positionen betrifft. Wird die maximale Länge überschritten, gibt es bei *jedem* CRC-Algorithmus zwei-Bit Fehler, die nicht erkannt werden
2. Ein beliebiges Generatorpolynom erkennt sämtliche Bündelfehler, die nicht länger als das Generatorpolynom sind – bis auf jenes, welches das gleiche Bitmuster hat wie das Generatorpolynom. Das beinhaltet natürlich auch 1-Bit-Fehler als Bündelfehler der Länge 1.
3. Ein Generatorpolynom, das durch x+1 teilar ist, erkennt jede ungerade Anzahl von Bitfehlern.
4. Es werden nur solche Zweibitfehler nicht erkannt, deren Abstand ein Vielfaches des Zyklus der Periode des längsten Bitfilters – die in Tabelle 14.3 angegebene *Länge* – ist. Bei optimal gewählten Generatorpolynomen vom Grad *n* mit gerader Anzahl von Termen ist dieser Abstand $2^{n-1} - 1$ bit.
5. Es lässt sich zeigen, dass alle Einbitfehler korrigiert werden können, wenn der Datenblock nicht länger als die *Länge* ist. Das folgt daraus, dass die Reste nach Division durch das Generatorpolynom alle verschieden sind - so weit man verschiedene Reste, von denen es höchstens 2^n gibt, haben kann. Allerdings

lassen unter Umständen 3-Bitfehler die gleichen Reste, so dass in diesem Fall eine Korrektur das Ergebnis noch mehr verfälschen kann. Allerdings sind 1- und 2-Bitfehler immer mit Sicherheit zu unterscheiden.

Name	Polynom	Länge	MHD	Anmerkungen
CRC-CCITT (CRC-4)	$x^4 + x + 1$	15		Identisch mit (15,11)-Hamming-Code
USB (CRC-5)	$x^5 + x^2 + 1$	31		Identisch mit dem (31,26) Hamming-Code
SD/MMC-Card (CRC-7)	$x^7 + x^3 + 1$	127	3	Identisch mit dem (127,120)-Hamming-Code
CRC-8 (ITU-T)	$x^8 + x^2 + x + 1$	127	4	ISDN Header Error Control
CRC-8 (SAE-J1850)	$x^8 + x^4 + x^3 + x^2 + 1$	255	3	Verwendet bei AES/EBU
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$			
CAN-CRC	$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$	127	6	
CRC-CCITT (CRC-16)	$x^{16} + x^{12} + x^5 + 1$	32767	4	Verwendet bei HDLC, X.25
CRC-32 (IEEE802.3)	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	$2^{32} - 1$	4	Verwendet bei Ethernet
CRC-64 (ISO 3309)	$x^{64} + x^4 + x^3 + x + 1$			

Tabelle 14.3 Auswahl einiger CRC-Definitionen

14.3.2.4 Fehlererkennung in Operationen

Wort- und Block-orientierte Codes verfolgen beide den Ansatz, statische Daten gegen Verfälschung – aus welchem Grund auch immer – zu schützen. Offen bleibt dabei jedoch die Frage, ob nicht auch Codes existieren, die auf vergleichbar einfache Weise Operationen auf Daten schützen können.

Ein triviales Beispiel hierfür ist eine Rotation der Bits eines Wortes. Wenn man alle Bits einbezieht, also auch ein ggf. genutztes Carrybit, dann bleibt bei der Rechts- wie Links-Rotation die Parität über alle Bits erhalten. Der Prognosewert der Parität der Ergebnisbits einer Rotation ist also gleich der Parität über die Quellbits.

Bei arithmetischen Operationen wird die Paritätsvorhersage schon wesentlich schwieriger, weil nunmehr zwei Operanden einwirken. Für die wichtigen Inkrement- und Dekrement-Operationen lassen sich allerdings Formeln angeben, die die Parität des Ergebnisses anhand des (einzig variablen) Operanden mit angemessenem Aufwand gestatten. Die hier angegebenen Formeln gelten für Inkrement/Dekrement auf 8-bit-Variablen, können aber auf größere Bitwerte erweitert werden.

$$p = b[0] \oplus b[1] \oplus b[2] \oplus b[3] \oplus b[4] \oplus b[5] \oplus b[6] \oplus b[7]; \quad (14.1)$$

$$\begin{aligned} p(neu) = p(alt) \oplus (not\, b[0] + (not\, b[2] * b[1] * b[0]) + \\ (not\, b[4] * b[3] * b[2] * b[1] * b[0]) + \\ (not\, b[6] * b[5] * b[4] * b[3] * b[2] * b[1] * b[0])); \end{aligned} \quad (14.2)$$

Das Dekrement erhält man nun, indem alle Eingangsoeranden b[x] invertiert werden. Dies kann z.B. innerhalb von Hardware ausgenutzt werden, um das Inkrement oder Dekrement auf ungerade Bitfehler zu kontrollieren.

Bewertet man nun ein XOR-Gatter mit der Kostenfunktion 3, ein OR sowie AND-Gatter mit jeweils 2 und eine Invertierung mit 1, so benötigt ist die Kostenfunktion des Inkrements 64, des Dekrements 78, die der Paritätsprüfung 76, mit Umschaltung zwischen Inkrement und Dekrement weitere 63, und die des Vergleichs (Fehlerprüfung) 38, so dass sich in etwa die gleichen Kosten für Berechnung und Paritätsprüfung ergeben.

Das Verhältnis wird allerdings deutlich besser zugunsten der Paritätsprüfung, wenn man zu 32-bit-Werten übergeht. Mitsamt allen Minimierungen ist der Overhead der Paritätsprüfung dann nur noch etwa 25%.

14.3.2.5 Einsatz redundanter Hardware (mit Fortführung des Betriebs)

Redundante Hardware kann im Wesentlichen durch Vervielfachung mit einem Mehrheitsentscheider erreicht werden. Dies wird auch als ”Voting“ bezeichnet, und bis auf den Entscheider selbst ist alles mehrfach ausgelegt.

Dieser Ansatz dient der Fehlererkennung mit Fehlerfolgenvermeidung und fortlaufendem Betrieb.

Der Vorteil dieses Ansatzes liegt darin, dass die gleiche Hardware kopiert wird. Das Fehlermodell geht davon aus, dass die Hardware aufgrund eines Defektes nicht funktioniert, nicht aufgrund eines konstruktiven Mangels. Die eigentliche Fehlertoleranz, d.h., die fehlervermeidende Reaktion, kann dann in Form dreier Varianten erfolgen:

- **Statische Redundanz**: Die Hardware bleibt immer erhalten, die Mitglieder stimmen laufend (an vorgesehenen Punkten) ab, und die Mehrheitsentscheidung gilt.

- **Dynamische Redundanz**: Bei Erkennen eines Fehlers wird die fehlerhafte Hardware rekonfiguriert, d.h., Reservekomponenten kommen zum Einsatz. Hier existieren z.B. Modellen für Prozessoren, Operationen (wie Addition) auf andere Einheiten (bzw. eine Sequenz davon) abzubilden.
- **Hybride Ansätze**: Die Mischung aus Mehrheitsvotum und Rekonfiguration stellt einen hybriden Ansatz dar, der zwar komplexer ist, aber natürlich die größte Flexibilität besitzt.

Genau genommen darf man das Fehlermodell der Hardware, dass diese zunächst fehlerfrei ist und keinen konstruktiven Mangel hat, natürlich nicht unbedarft übernehmen. So sind so genannte Chargenprobleme bekannt, d.h., eine Produktionscharge eines Hardwarebausteins zeigt den gleichen Mangel. Dies würde zu einem übereinstimmenden Verhalten mehrerer Komponenten im Betrieb führen mit dem Ergebnis, dass die Fehlertoleranz in eine Fehlerakzeptanz übergeht.

14.3.2.6 Einsatz redundanter Hardware mit Erreichen eines sicheren Zustands

Die Fortführung des Betriebs mithilfe einer laufenden Mehrheitsentscheidung ist in der Summe außerordentlich kostspielig. Einfacher ist die Fehlererkennung gekoppelt mit einem gezielten Erreichen eines sicheren Zustands; in diesem Fall reicht sogar ein doppelt augelegtes System, da es ja nur um die Erkennung des Ausfalls geht. Fehlt hingegen die Erreichbarkeit eines sicheren Zustands, muss auf Maßnahmen mit Fortführung des Betriebs zurückgegriffen werden.

Die Ausführung der beiden Systeme kann symmetrisch oder asymmetrisch erfolgen. Symmetrisch bedeutet, dass beide Systeme gleich ausgestattet sind und in gleichem, ggf. gegeneinander versetztem Takt laufen. Die Ergebnisse werden laufend miteinander verglichen, bei einer als signifikant deklarierten Abweichung wird das gesamte System in einen sicheren Zustand gefahren.

Ein Beispiel für die symmetrische Ausführung sind spezialisierte Mikroprozessoren im so genannten Lockstep-Betrieb („Gänsemarsch-Betrieb“) [IBM750]. Die Anzahl der zu vergleichenden Signale kann dabei rasch einige 1000 betragen, so dass der Einsatz an Siliziumfläche und Verlustleistung recht hoch ist. Weiterhin bleibt die Gefahr der so genannten Common Mode Failures [Wiki_CMF] vorhanden. Common Mode Failures liegen dann vor, wenn Fehler nicht statistisch unabhängig sind, also eine Form von kopplung haben. Der Ausfall der Energieversorgung für einen Dual-Core-Prozessor, der im Lockstep-Modus betrieben wird, ist so ein (drastischer) Common Mode Failure (oder Common Causation Failure.

Um solche Fälle von Common Mode Failures auszuschließen müssen konstruktive Maßnahmen ergriffen werden, man spricht dann von diversitärer Redundanz, wenn dies komplett erfolgt. Hierdurch ist dann eine Asymmetrei in den Hardwarekomponenten vorhanden, die soweit gehen kann, dass die überwachende Einheit wesentlich einfacher aufgebaut ist: Sie muss nur Fehler erkennen.

14.3.2.7 Einsatz redundanter Software mit Fortführung des Betriebs

Der mehrmalige Einsatz der gleichen Software ist zwecks Fehlertoleranz sinnlos, da Software nicht altert und somit keine neuen Fehler entstehen. Fehler sind von Beginn an enthalten, um hier fehlertolerant zu sein, müssen verschiedene Versionen verwendet werden.

Dies bedeutet einfach, dass mehrere unabhängige Designteams verschiedene Versionen herstellen müssen. Auch hier kann dann wieder zwischen statischer und dynamischer Redundanz unterschieden werden:

- **Statische Redundanz** (N-Version-Programming): Es werden mehrere Versionen durch verschiedene Entwicklungsteams erstellt, die dann real oder im Zeitscheibenverfahren nebeneinander laufen. Und definierte Synchronisationspunkte haben. An diesen Synchronisationspunkten werden die Ergebnisse verglichen und durch einen Voter bestimmt, welches Ergebnis das wahrscheinlich richtige ist (Mehrheitsentscheidung). Dieses Verfahren ist sehr aufwendig.
- **Dynamische Redundanz** (Recovery Blocks): Es wird eine permanente Fehlerüberwachung durchgeführt, um beim Erkennen eines Fehlers den entsprechenden Softwareblock gegen eine alternative Softwarekomponente auszutauschen.

14.3.2.8 Einsatz redundanter Software mit Erreichung eines sicheren Zustands

Die Hardwareredundanz ist vergleichsweise teuer, selbst für die Erkennung von Fehlern, da die Hardware auch hier zweifach ausgelegt werden muss und eben auch mit elektrischer Energie versorgt wird. Daher ist die Idee nahe liegend, die Funktionaseinheiten der (einzigen) CPU mithilfe zweier Durchläufe in Software zu erkennen.

Hierzu seien zwei Ansätze [Dea04][OMM02] kurz besprochen. Gemeinsam ist beiden, dass sie die Funktionseinheiten zur Datenverknüpfung mithilfe variierter Daten testen wollen.

In [Dea04] werden u.a. die Ansätze wift-R und Triple Redundancy vorgestellt:

Swift-R

Der Swift-R-Ansatz versucht, die entstehenden Soft Errors möglichst komplett zu eliminieren. Hierzu wird der Code zweifach kopiert und ausgeführt, und eine Mehrheitsentscheidung aus den drei Ergebnissen wird dann als das Ergebnis gewertet. Eine einfache Kopie hätte nur zur Folge, dass man Fehler detektieren, aber nicht zur Laufzeit beheben könnte.

Der kleine Codeabschnitt in Bild 14.9 zeigt, was diese Verdreifachung bedeutet. Jedes Register wird zweifach gespiegelt, z.B. speichern R2, R2’ und R2’’ jeweils den gleichen Wert, wenn keine Verfälschung entsteht. Die zusätzlichen Register müssen natürlich auf wirklich vorhandene abgebildet werden. Der Speicher gilt

hierbei als gesichert, z.B. durch ECC, und gleich nach der Ladeoperation (Zeile 2, Bild 14.9 b) werden die aus dem Speicher geholten Werte zweifach kopiert.

```
ld R4, [R2]   ; R4 = [R2] im Speicher

add R1, R3, R4; R1 = R3 + R4

                      a)

st [R1], R5
```

```
majority( R2, R2', R2'' );
ld R4, [R2];
mov R4', R4;      R4' = R4
mov R4'', R4;
add R1, R3, R4;
add R1', R3', R4';
add R1'', R3'', R4'';
majority( R1, R1', R1'' );
majority( R5, R5', R5'' );
st [R1], R5;
                          b)
```

Bild 14.9 Vergleich Original- (a) zu mittels Swift-R erweiterten Code (b)

Alle Rechenoperationen werden wirklich dreimal durchgeüfhrt, auf jeweils anderen Datenregstern, und zentraler Punkt ist dann die Mehrheitsentscheidung *majority()*, die den mehrheitlichen Wert in das ursprüngliche Register kopiert und damit die Fehlererholung (*Fault Recovery*) schafft. Dieser Code ist nun keineswegs vollkommen schützend, immerhin kann beim Laden oder Speichern ein Fehler auftreten, aber die Mehrheit der Fehlermöglichkeiten ist hierin erfasst.

Die Autoren in [Dea04] geben hierfür an, dass die Fehler für die Segmentverletzung (SEGV) auf 1,93 % und für stille Datenkorruption (SDC) auf 0,81 % sinken und damit etwa auf 1/10 des Ausgangsniveaus. Der Preis hierfür ist allerdings auf Seiten der Performance zu zahlen: Die Laufzeit für die Testprogramme steigt etwa um den Faktor 2.

Triple Redundancy (Trump)

Während der erste Weg eher naheliegend erscheint, existiert mindestens noch ein weiterer Weg, der die so genannten AN-Codes benutzt. Hier werden neue Datenwerte zweifach berechnet, einmal in der Originalversion, das zweite Mal multipliziert mit einem Faktor. Bei allen arithmetischen Verknüpfungen gilt nämlich das Distributivgesetz

$$A * (X + Y) = A * X + A * Y;$$

Wählt man nun für A einen Faktor $A = 2^n - 1$ (n eine natürliche Zahl), dann kann man auch garantieren, dass ein 1-Bit-Fehler in den multiplizierten Werten erkennbar ist, weil

$$C \pm 2^k \equiv \pm 2^k \,(\mathrm{mod}\, A) \neq 0 (\mathrm{mod}\, A)$$

gilt, wenn $A = 2^n$ ist. Mit anderen Worten: Wenn man zweifach rechnet, dann kann man nicht nur einen Fehler erkennen, man kann ihn auch mittels des folgenden Pseudocodes beheben:

```
original = <Rechnung>;
kopie = <Rechnung mit 3 mult.>

if( 3 * original != kopie )
   if( (kopie % 3) == 0 )
      original = kopie / 3;
   else
      kopie = 3 * original;
```

Bild 14.10 Pseudocode zur Fehlerbehebung in Trump, hier wurde A = 3 gewählt

```
ld R4, [R2]   ; R4 = [R2] im Speicher

add R1, R3, R4; R1 = R3 + R4



st [R1], R5
```

a)

```
recovery( R2, R2t );
ld R4, [R2];
mul R4t, R4, #3;  R4t = 3*R4
add R1, R3, R4;
add R1t, R3t, R4t;
recovery( R1, R1t );
recovery( R5, R5t );
st [R1], R5;
```

b)

Bild 14.11 Vergleich Original- (a) zu mittels Trump erweiterten Code (b)

In diesem Code, für den mit n = 2 der Wert A = 3 genutzt wurde, werden zwei Rechnungen durchgeführt und die Ergebnisse miteinander verglichen. Sind diese ungleich (Fehlerdetektierung), dann wird über die Dividierbarkeit der Kopie durch den Faktor 3 entschieden, welche Rechnung richtig war, und die Werte werden korrigiert. Dies führt dann zu der Codeerweiterung in Bild 14.11 (in Assembler):

Das Verfahren sind wesentlich einfacher aus als Swift-R, weist aber einige Lücken auf:

- Die Werte in den Registern sind durch die Anzahl der Bits begrenzt, man muss also darauf achten, dass die Multiplikation mit *A* überhaupt erlaubt ist und nicht zu Überläufen führt. Dies kann insbesondere bei Pointerarithmetik zu Problemen führen.
- Die AN-Codes sind nicht mit logischen Operationen wie UND, ODER, XOR anzuwenden.

Hieraus resultiert eine verminderte Anwendbarkeit. Während also die Laufzeit nur um 36 % steigt [Dea04], werden mit einer Restfehlerwahrscheinlichkeit von 7,39 % (SEGV) bzw. 4,88 % (SDC) nicht so gute Werte erzielt wie bei Swift-R.

ED^{4}I

Während in den bisher dargestellten Ansätzen die Mischung der Rechnungen innerhalb einer Funktion oder eines Thread erfolgte, kann man dies auch ohne

weiteres auf zwei Durchläufe verlagern. Dies wird in [OMM02] dargestellt, wobei die Autoren im zweiten Durchgang die Daten mit einem Faktor -1 .. -5 multiplizieren und dann auswerten, mit welcher Wahrscheinlichkeit Fehler gefunden werden.

Das Ergebnis ist, dass im Mitel die Multiplikation der Daten mit -2 die besten Resultate ergibt. Die Wahrscheinlichkeit, einen Fehler in der Hardware durch die Multiplikation zu verdecken, wird mit < 1 % angegeben. Die Wahrscheinlichkeit, diesen Fehler dann auch zu entdecken, beträgt im Mittel ca. 70 %, so dass mehrere Durchläufe zu einer hohen Wahrscheinlichkeit der Fehlerentdeckung führen.

Als Kritikpunkte an diesem Verfahren sind wiederum der mögliche Datenüberlauf zu nennen und die Unmöglichkeit, logische Operationen auf den variierten Daten mit entsprechend richtigem Ergebnis auszuführen.

14.3.3 Analytische Maßnahmen

Zurück zu den eigentlichen Systemen: Um bei komplexen Systemen die Zuverlässigkeit zu beurteilen muss man dieses in seine Einzelfunktionalitäten zerlegen. Die Zuverlässigkeit einer einzelnen Komponente sei dann bekannt und mit $R_i(t)$ mit $0 < R_i(t) < 1$ bezeichnet.

Die Kopplung der Systemkomponenten kann dann stochastisch abhängig oder unabhängig sein. Im einfacheren unabhängigen Fall müssen dann bei serieller Kopplung der Komponenten (heißt: das System fällt aus, wenn mindestens eine der Komponenten ausfällt) die Einzelwahrscheinlichkeiten multipliziert werden:

$$R_{seriell} = \prod_i R_i(t)$$

Bei paralleler Kopplung – in diesem Fall soll das System noch intakt sein, wenn mindestens eine Komponente intakt ist – ergibt sich die Zuverlässigkeit

$$R_{parallel} = 1 - \prod_i \left[1 - R_i(t) \right]$$

Bei stochastischer Abhängigkeit wird die Analyse entschieden komplexer, denn hier bewirken Einzelausfälle Kopplungen zu anderen. In diesem Fall kommen Analyseverfahren wie z.B. Markovketten zum Einsatz.

14.3.4 Gefahrenanalyse

Unter Gefahrenanalyse wird ein systematisches Suchverfahren verstanden, um Zusammenhänge zwischen Komponentenfehlern und Fehlfunktion des Gesamtsystems aufzudecken. Hierzu müssen noch einige Begriffe definiert werden:

Definition 14.9:

Als **Gefahr** (hazard) wird eine Sachlage, Situation oder Systemzustand bezeichnet, in der/dem eine Schädigung der Umgebung (Umwelt, Maschine, Mensch) möglich ist.

Eine Gefahrensituation ist also eine Situation, in der das Risiko größer als das Grenzrisiko ist. Die ursächlich zugrundeliegenden Fehler sollen nun zurückverfolgt werden, unabhängig davon, ob diese zufällig (Alterung) oder konstruktiv bedingt sind.

Definition 14.10:

Tritt eine Schädigung tatsächlich ein, so bezeichnet man dieses Ereignis als **Unfall** (accident).

Die systematischen Suchverfahren können nun prinzipiell überall ansetzen, in der Praxis wählt man jedoch einen der beiden Endpunkte. Man spricht dann von Vorwärts- bzw. Rückwärtsanalyse. Bekannt sind hierbei die Ereignisbaumanalyse (FTA, *Fault Tree Analysis*) und die *Failure Mode and Effect Analysis* (FMEA). Im letzteren Fall werden folgende Fragestellungen untersucht:

- Welche Fehler(-ursachen) können auftreten?
- Welche Folgen haben diese Fehler?
- Wie können diese Fehler vermieden oder das Risiko minimiert werden?

Die Fehlerliste führt dann zu einer Systemüberarbeitung, und die Analyse beginnt von vorne. Die FMEA hat folgende Ziele:

- Kein Fehler darf einen negativen Einfluss (auf redundante Systemteile) haben.
- Kein Fehler darf die Abschaltung der Stromversorgung eines defekten Systemteils verhindern.
- Kein Fehler darf in kritischen Echtzeitfunktionen auftreten.

Letztendlich ist dies auch Forschungsthema. So gibt es in Deutschland beispielsweise die Initiative ”Organic Computing“, die Methoden der Biologie nachzuvollziehen versucht.

14.3.5 Die andere Sicht: Maschinensicherheit

Letztendlich ist entscheidend, was die Anwender von Software-basierten Systemen haben wollen bzw. welche Eigenschaften sie garantiert haben wollen. Die Funktionalität einschließlich der Zuverlässigkeit ist nämlich entscheidend für die Sicherheit der Maschinen, in die diese Systeme eingebaut sind.

Die entscheidenden neuen Normen zur Maschinensicherheit sind DIN ISO 13849 (Maschinensicherheit, seit 2006 gültig) und DIN EN 61508 (Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, Oktober 2005). Diese beiden sind eng aufeinander bezogen und verwie-

sen gegenseitig. Tabelle 14.4 zeigt die so genannten Performance Level (PL) bzw. Security Integrity Level (SIL), die in den jeweiligen Normen definiert werden.

Wahrscheinlichkeit eines gefahrb. Ausfalls pro Stunde [1/h]	PL,–ISO 13849-1	SIL,–EN IEC 61508
$10^{-5} <$PDF$< 10^{-4}$	a	
$3x10^{-6} <$PDF$< 10^{-5}$	b	1
$10^{-6} <$PDF$< 3x10^{-6}$	c	1
$10^{-7} <$PDF$< 10^{-6}$	d	2
$10^{-8} <$PDF$< 10^{-7}$	e	3

Tabelle 14.4 Vergleich PL und SIL (PDF: Probability of dangerous failures per hour, auch PFH abgekürzt)

Interessant ist dabei die Sicht auf elektronische bzw. programmierbare elektronische Systeme. Programmierbare Hardware gilt dabei als Hardware. Wenn man nun ein sicheres System aufbauen will, müssen zusätzlich zu allen anderen Fehlern auch die Common Causation Failure (CCF), also die Fehler gleichen Ursprungs, beachtet werden.

Normalerweise reicht eine einfache Redundanz, also die Verdopplung der Hardware mit einer Entscheidungsinstanz aus, wenn es einen sicheren Zustand gibt. Hiermit ist gemeint, dass dieser sichere Zustand angenommen wird, wenn eine Hardware (Überwachung) eine entsprechende Situation detektiert. Die CCF entstehen nun durch Bausteinfehler, die gemeinsam in beiden Bausteinen sind. Die Maschinensicherheit fordert daher bei sicherheitskritischen Applikationen eine ”diversitäre Redundanz“, d.h. zwei verschiedene Bausteine mit zwei verschiedenen Konfigurationen (falls es sich um programmierbare Hardware handelt).

Die Software in derartigen Systemen muss entweder redundant diversitär aufgebaut sein – dies bedeutet, dass unterschiedliche Compiler eingesetzt und zwei verschiedene Versionen von unterschiedlichen Designteams erstellt werden müssen –, oder die Software muss in einem komplexen Prozess zertifiziert werden – oder auch beides.

14.4 Software-Review und statische Codechecker

Idealerweise hätte man gerne innerhalb der Entwicklungszeit die Möglichkeit, durch geeignete Tools die Fehler im Code aufzufinden und so zu eliminieren. Das Einzige, was hierzu erhältlich ist, ist im Software Review integriert. Dieser Software Review ist ein Teil des analytischen Prozesses, der alleine aufgrund der Trefferquote zwingend notwendig ist: 30 – 70 % aller Fehler werden in dieser Phase gefunden. Leider kostet ein solches Review, wird es ernsthaft betrieben, sehr viel Zeit.

Eine gewisse Hilfe sind die statischen Codechecker, die den Code analysieren und wertvolle Hinweise liefern. In [lint] kann z.B. ein von *lint* abstammender statischer Codechecker als Freeware-Tool gefunden werden.

Statische Codechecker können z.B. folgende Aktionen durchführen:

- Initialisation Tracking: Variablen werden darauf untersucht, ob sie vor der ersten lesenden Verwendung initialisiert wurden. Dies erfolgt auch über if/else-Konstrukte usw., so dass – im Gegensatz zu vielen Compilern – wirkliche Initialisierungsfehler gefunden werden.
- Value Tracking: Indexvariable für Arrays, mögliche Divisionen durch Null sowie Null-Zeiger stellen potenzielle Fehlerquellen im Programm dar. Sie werden ausführlich analysiert.
- Starke Typprüfung: Abgeleitete Typen (#typedef in C) werden darauf überprüft, dass nur sie miteinander verknüpft werden (und nicht die Basistypen). Wieterhin erfolgt eine sehr genaue Typprüfung, also z.B., ob Vergleiche zwischen int und short usw. geführt werden, und eine entsprechende Warnung wird ausgegeben.
- Falls es so genannten Funktionssemantiken gibt – das sind Regeln für Parameter und Rückgabewerte, etwa so, dass der erste Funktionsparameter nicht 0 sein darf – dann sind weitere Checks möglich.

Letztendlich erzwingt der Einsatz von statischen Codecheckern, dass sich der Entwickler sehr um seinen Sourcecode bemüht. Und genau das dürfte in Zusammenhang mit Codierungsregeln (→ 6.6) einen sehr positiven Effekt auf die Softwarequalität haben

15 Test und Testmetriken

In Kapitel 14 waren schon einige Elemente zum Test vorhanden: Die Methoden zur Softwareredundanz beispielsweise, die Fehler in der Hardware durch Zweitdurchläufe mit variierten Werten verwenden, stellen nichts anderes als In-Situ-Tests dar: Das Systemverhalten wird im Betrieb auf korrekte Funktionalität getestet.

Der fundmentale Unterschied zu den hier betrachteten Methoden besteht nun darin, dass im Fall des In-Situ-Tests nur sehr eingeschränkte Teile auf den Verlust ihrer korrekten Funktionalität getestet werden, während die hier besprochenen Tests das gesamte System erfassen, ohne dass überhaupt eine bislang vorhandene korrekte Funktionalität angenommen wird.

15.1 Testen (allgemein)

In der Praxis steuert tatsächlich alles zur Herstellung und Sicherung von Softwarequalität auf das Testen hin, dies erscheint als die ultimative Lösung zur Herstellung einer guten Systemqualität und Zuverlässigkeit. Eine gute Einführung in dieses überaus komplexe Thema ist in [Grü04a] [Grü04b] [Grü05a] [Grü05b] und [Grü06] gegeben.

Testen muss als destruktiver Prozess verstanden werden. Man versucht, die Software zu brechen, ihre Schwachpunkte zu finden, Fehler aufzudecken. Es ist natürlich sehr schwierig für den Entwickler, sein bislang konstruktive Sicht aufzugeben: Bislang war er/sie während des Designs und des Programmierens damit befasst, eine ordentliche Software herzustellen, so dass die destruktive Sicht sicherlich schwer fallen würde. Aus diesem Grund muss der Test von anderen, nicht mit der Entwicklung befassten Personen durchgeführt werden.

Um den Testprozess genauer zu beschreiben, wird er in 4 Phasen eingeteilt [Grü04a]:

- Modellierung der Software-Umgebung
- Erstellen von Testfällen
- Ausführen und Evaluieren der Tests
- Messen des Testfortschritts

15.1.1 Modellierung der Software-Umgebung

Eine der wesentlichen Aufgaben des Testers ist es, die Interaktion der Software mit der Umgebung zu prüfen und dabei diese Umgebung zu simulieren. Dies kann eine sehr umfangreiche Aufgabe sein:

- Die klassische Mensch/Maschine-Schnittstelle: Tastatur, Bildschirm, Maus. Hier gilt es z.B., alle erwarteten und unerwarteten Eingaben und Bildschirminhalte in dem Test zu organisieren. Einer der Ansätze hierzu heißt Replay-Tools, die Eingaben simulieren und Bildschirminhalte mit gespeicherten Bitmaps vergleichen können.
- Das Testen der Schnittstelle zur Hardware: Ideal ist natürlich ein Test in der Form ”Hardware in the loop“, d.h., die zu testende Hardware ist vorhanden und offen. Falls nicht, müssen hier entsprechende Umgebungen ggf. sogar entwickelt werden. Zudem gilt es, bei dem Test auch nicht-erlaubte Fälle einzubinden, d.h., es müssen Fehler in der Hardware erzeugt werden, insbesondere bei Schnittstellen.
- Die Schnittstelle zum Betriebssystem ist genau dann von Interesse, wenn Dienste hiervon in Anspruch genommen werden. Hier sind Fehlerfälle, z.B. in Form zu geringen Speicherplatzes auf einem Speichermedium oder Zugriffsfehlern, zu testen.
- Dateisystem-Schnittstellen gehören im Wesentlichen auch zum Betriebssystem, seien hier jedoch explizit erwähnt. Der Tester muss Dateien mit erlaubtem und unerlaubtem Inhalt sowie Format bereitstellen.

Letztendlich ist es der Phantasie und der Erfahrung des Testers zu verdanken, ob ein Test möglichst umfassend oder eben ein ”Schönwettertest“ ist. Beispielsweise müssen oft ungewöhnliche Situationen getestet werden, wie z.B. der Neustart einer Hardware während der Kommunikation mit externen Geräten.

15.1.2 Erstellen von Testfällen

Das wirkliche Problem der Erstellung von Testfällen ist die Einschränkung auf eine handhabbare Anzahl von Test-Szenarien. Hierbei hilft (zumindest ein bisschen) die so genannte *Test Coverage*: Man stellt sich die Frage, welche Teile des Codes noch ungetestet sind. Hierfür sind Tools erhältlich (bzw. in Debugging-Tools eingebaut), die den Sourcecode anhand der Ausführung kennzeichnet. Mit dem Ziel, die gewünschte Testabdeckung am Quellcode zu erreichen, wird der Tester daher Szenarien auswählen, die

- typisch auch für die Feldanwendung sind;
- möglichst ”bösartig“ sind und damit eher Fehler provozieren als die bereits zitierten ”Schönwettertests“;
- Grenzfälle ausprobieren

Bei der Testabdeckung gilt es noch zu überlegen, ob die Ausführung einer Sourcecodezeile überhaupt genügt. Hierzu werden noch Testabdeckungsmetriken dargestellt (→ 15.2)

15.1.3 Ausführen und Evaluieren der Tests

Zwei Faktoren beeinflussen die Ausführung des Tests, der manuell, halbautomatisch oder vollautomatisch sein kann: die Haftung bei Software-Fehlern und die Wiederholungsrate der Tests. Anwendungen mit Sicherheitsrelevanz etwa erzeugen einen erheblichen Druck in Richtung automatischer Tests, allein, um die exakte Wiederholbarkeit zu erreichen.

Derartige Wiederholungen können notwendig sein, wenn an anderer Stelle ein Fehler gefunden wurde, dessen Behebung nun auf Rückwirkungsfreiheit getestet werden soll (so genannte Regressionstests).

Nach Ausführung der Tests, was sehr gut automatisch durchführbar ist, müssen die Tests bewertet werden, was meist nicht automatisch durchzuführen ist. Zumindest müssen die Kriterien, wann ein Test bestanden ist und wann nicht, vorher fixiert werde, ansonsten droht ein pures ”Herumprobieren“. Last not least bleibt die Frage der Vertrauenswürdigkeit des Tests, denn ein ständiger Erfolg sollte Misstrauen erzeugen. Um dies zu prüfen, werden bewusst Fehler eingebaut (Fault Insertion oder Fault Seeding), deren Nichtentdeckung natürlich eine Alarmstufe Rot ergäbe.

15.1.4 Messen des Testfortschritts

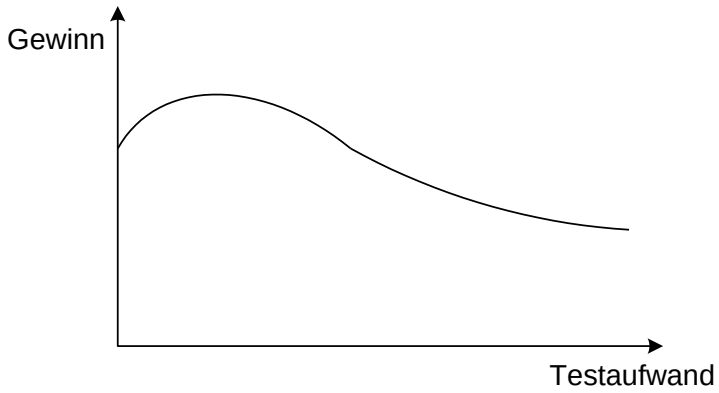


Bild 15.1 Gewinn versus Testaufwand

Ein Testprojekt sollte wie jedes andere Projekt genau geplant werden. Teil dieses Plans ist die Festlegung des Projektziels, etwa in der Form, wie viele unentdeckte Fehler die Software nach Testende noch haben darf. Art und Umfang der Tests werden sich nach dieser Größe richten, insbesondere darf nicht übersehen werden, dass sich der differenzielle Gewinn mit wachsendem Testaufwand wieder erniedrigt (Bild 15.1).

Um dies zumindest abschätzen zu können, ist das Wissen über die Komplexität des Codes wichtig. Eine passende Codemetrik ist die zyklomatische Komplexität (cyclomatic complexity) nach McCabe: Diese bestimmt die Anzahl der if-, while-do- und for-Kommandos im Code und damit die Anzahl der möglichen Verzweigungen. Tools hierfür sind (auch frei) verfügbar.

15.1.5 Code Coverage

Code Coverage steht unter Testern hoch im Kurs. Insbesondere im Modultest (→15.2) sollen sämtliche Codezeilen mindestens einmal durchlaufen werden, um keine prinzipiell unentdeckbaren Fehler im Programm zu haben – unentdeckbar deshalb, weil der Test an dieser Stelle niemals stattgefunden hat.

Der Umkehrschluss hingegen ist nicht zulässig: Nur weil im Test jede Codezeile durchlaufen wurde ist nicht gewährleistet, dass keine Fehler im Code mehr vorhanden sind. Gegenbeispiele hierfür sind schnell konstruiert:

```
unsigned long addiere( unsigned long summand1,
                       unsigned long summand2 )
{
      return( summand1 + summand2 );
}
```

Bild 15.2 Additionsfunktion mit Möglichkeit zum Überlauf

Die in Bild 15.2 gezeigte Funktion zum Addieren zweier Integerzahlen (im Format unsigned long) kann sehr einfach getestet werden: Es existiert hierin nur ein einziger Weg, der Test mit dem Ziel "Code Coverage = 100 %" braucht diese Funktion nur einmal aufzurufen. Dass als wesentlicher Fehler hier ein Datenüberlauf (→ 14.1.1) auftreten kann ist für Code Coverage uninteressant: Code Coverage adressiert den Kontrollfluss.

15.1.5.1 Definitionen zum Code Coverage

Um dies näher zu betrachten müssen einige Definitionen zum Code Coverage vorangestellt werden:

Defintion 15.1:

Eine **Condition** (**Bedingung**) ist ein nicht weiter teilbarer Boolescher Ausdruck, d.h. ein Ausdruck, der den Wahrheitswert *true* oder *false* annehmen kann (Beispiel x > 5).

Definition 15.2:

Eine **Decision** (*Entscheidung*) ist ein Boolescher Ausdruck, der aus *Conditions* und keinem, einem oder mehreren Booleschen Operatoren aufgebaut ist. Eine *Decision* ohne Booleschen Operator ist per Default eine Condition.

Definition 15.3:

Condition Coverage (oder auch **Predicate Coverage**) bezeichnet das Maß, wie viele der Booleschen Ausdrücke (auch Subausdrücke) in einer *Decision* ausgewertet wurden, und zwar sowohl nach *true* als auch nach *false*.

Definition 15.4:

Decision Coverage oder **Branch Coverage** ist das Maß dafür, wie viele Übergänge nach einer *Decision* getestet wurden. Äquivalent hierzu ist die Feststellung, dass alle bzw. wie viele Entscheidungen auf die Wahrheitswerte *true* und *false* abgefragt wurden. Hinzu werden noch alle Eintritts- und Austrittspunkte in einem Programm gezählt, die getestet wurden.

Definition 15.5:

Condition/Decision Coverage ist das Maß, wie viele Ein- und Austrittspunkte in einem Programm getestet wurden, wie viele *Conditions* in den *Decisions* getestet wurden, und wie viele *Decisions* im Programm getestet wurden.

Der 100%-Test der Condition/Decision Coverage bedeutet allerdings keinesfalls, dass alles im Kontrollflussverhalten getestet wurde. Hierzu muss diese Testbedingung nochmals verschärft werden, zur so genannten Modified Condition/Decision Coverage. Hierzu ist eine Zwischendefinition notwendig:

Definition 15.6:

Eine *Condition* wird als die *Decision* **unabhängig beeinflussend** bezeichnet, wenn es mindestens eine konstante Bedingung für alle anderen *Conditions* innerhalb der *Decision* gibt, so dass die unter Beobachtung stehende *Condition* das Ergebnis der *Decision* beeinflusst (also verschiedene Ergebnisse *true* und *false* ergibt).

Defintion 15.7:

Modified Condition/Decision Coverage (MC/DC) ist das Testmaß dafür, wie viele Ein- und Austrittspunkte in einem Programm getestet wurden, wie viele *Conditions* in den *Decisions* getestet wurden, und wie viele *Decisions* im Programm getestet wurden, und dass alle *Conditions* als *unabhängig beeinflussend* nachgewiesen wurden.

Diese modifizierte Definition bedeutet also, dass man nach unabhängigen Bedingungen sucht und dies auch nachweist.

Der Vollständigkeit halber noch einige weitere Definitionen:

Definition 15.8:

Function Coverage bezeichnet das Maß, wie viele Funktionen (Subroutinen, Coroutinen) wirklich im Test aufgerufen wurden.

Definition 15.9:

Statement Coverage bezeichnet das Maß, welcher Prozentsatz an Hochsprachenstatements (= Node) im Programm ausgeführt wurden.

15.1.5.2 Bemerkungen zum Code Coverage

Es erscheint nun als sehr sinnvoll, eine möglichst 100%ige Code Coverage und hierin vor allem eine 100%ige MC/DC zu erhalten. Hierbei gibt es aber durchaus Überraschungen, und einige Bemerkungen hierzu sind im Folgenden zusammengefasst [Büc10a][Büc10b].

Bemerkung 1: Nicht immer ist eine 100%ige Abdeckung möglich.

Folgendes Codebeispiel zeigt dies:

```
for( k = 0; k < 2; k++ )
{
    switch( k )
    {
        case 0:
            a = 100;
            break;
        case 1:
            a = 200;
            break;
        default:
            a = 300;
            break;
    }
}
```

Bild 15.3 Programmbeispiel ohne 100%iges Code Coverage

Es ist im Test unmöglich, den default-Zweig in dem Bild 15.3 zu erreichen, dementsprechend ist die Code Coverage nicht 100%. In diesem Fall handelt es sich aber um so genannten Deadcode, der nach Optimierung oftmals gelöscht ist.

Bemerkung 2: Die Anzahl der Testzweige ist nicht immer eindeutig.

Auch hier sei ein kurzes Beispiel angefügt:

```
switch( k )
{
    case 0:
    case 1:
    case 2:
        a = 200;
        break;
    default:
        a = 300;
        break;
}
```

Bild 15.4 Programmbeispiel mit Unsicherheit in der Anzahl der Wege

In diesem Beispiel sieht es so aus, als gäbe es nur 2 Wege (case 0/1/2 als einen, default als den anderen). Dies ist aber nicht der Fall, es sind 4 Wege, denn in der Übersetzung (→ 6.5.2) wird der Compiler die Weg 0, 1 und 2 durchaus in Teilen getrennt ausführen. Man muss also 4 Wege wirklich testen.

Bemerkung 3: Für die Code Coverage, präziser die Anzahl der Wege, ist es keineswegs egal, wie der Code formuliert ist.

```
if( (a || b) && (c || d) )
{
    x = 1;
}
else
{
    x = 0;
}
```

```
if( a )
    if( c )
        x = 1;
    else if( d )
        x = 1;
    else
        x = 0;
else if( b )
    if( c )
        x = 1;
    else if( d )
        x = 1;
    else
        x = 0;
else
    x = 0;
```

Bild 15.5 Programmbeispiel mit zwei unterschiedlichen Wegen bei gleicher Semantik

Das Beispiel in Bild 15.5 zeigt zwei Codefragmente, die den gleichen semantischen Inhalt besitzen. Im Sinn der zu durchlaufenden Wege besteht hier ein großer Unterschied; betrachtet man dies jedoch aus Sicht von MC/DC, haben beide

Codeabschnitte die gleiche Komplexität, den Test betreffend. Zum Nachweis, alle unabhängigen Wege durchlaufen zu sein, benötigt man in diesem Fall die gleiche Anzahl an Testfällen.

Hieraus ergibt sich auch, dass man sich das (Test-)Leben durch eine geschickte Programmierung nicht wirklich leichter machen kann.

Bemerkung 4: Es ist nicht ausreichend, die Testdaten, die zur Erreichung einer 100%igen Code Coverage führen sollen, allein aus dem Code heraus zu ziehen.

Diese Ableitung der Testfälle könnte automatisch erfolgen – Test und Entwicklung sollen aber streng voneinander getrennt sein. Der Test mit Code Coverage ist zwar ein "White-Box-Test" (→15.2), d.h., die Testfälle sollen aus der Kenntnis des Codes heraus erzeugt werden, aber ein einfacher Automatismus würde hier dazu führen, dass "vergessene" Fälle auch nicht getestet werden.

An dieser Stelle liegt ein wichtiges Interface zum erfolgreichen Test: Das Programm soll nicht nur in dem existierenden Algorithmus bestätigt werden, sondern es sollen auch Fälle erkannt werden, wo Teile im Algorithmus vergessen wurden. Dies führt zum Data Coverage im nächsten Abschnitt.

15.1.6 Data Coverage

Während Code Coverage auf den Test des Kontrollflussgraphen mit allen Verzweigungen hinzielt, wird mit Data Coverage – ein Thema und Begriff, das/der noch nicht etabliert ist – die Darstellung der Abhängigkeiten der Daten und die daraus resultierende Testdatengenerierung bezeichnet. Eine 100%ige Coverage wie im Codefluss kann hier natürlich nicht erreicht werden.

Als Beispiel soll ein Codeausschnitt aus dem Beispiel zur Diskreten Fourier Transformation (→ 9.2) mit in Bild 15.6 dargestelltem Codeausschnitt dienen. Die Zielvariable sind die Arrays i32CoeffA[] und i32CoeffB[], wobei hier allerdings nur eines betrachtet wird.

Die Frage, die durch Data Coverage beantwortet werden soll, ist, in welcher Weise die Zielvariablen (diese sind im Beispiel global) von den einzelnen Variablen abhängen, und zwar nicht nur die Tatsache, dass sie abhängen, sondern auch eine Klassifizierung.

Diese Klassifizierung, die in Bild 15.7 mit dargestellt ist, dient dann als Anhalt dafür, wie getestet werden soll, welche Variable beispielsweise besonders schützenswert sind, ggf. sogar im Code (Beispiel: Die Indizes zum schreibenden Zugriff) usw.

```
void vComputeDFT( uint16 ui16NumOfPoints, int16 *i16Value )
{
   uint16 k, m, ui16Index;
   int32 i32CoeffAHigh, i32CoeffBHigh;
   int32 i32CoeffATemp, i32CoeffBTemp;
   uint16 ui16CoeffALow, ui16CoeffBLow;

   for( k = 0; k < NUM_OF_COEFFICIENTS; k++ )
   {
      i32CoeffAHigh = 0;
      ui16CoeffALow = 0;
      i32CoeffBHigh = 0;
      ui16CoeffBLow = 0;
      i32CoeffATemp = 0;
      i32CoeffBTemp = 0;

      for( m = 0, ui16Index = 0; m < ui16NumOfPoints; )
      {
         i32CoeffATemp += *(i16Value+m) * i32SineTable[ui16NumOfPoints - ui16Index - 1];
         i32CoeffBTemp += *(i16Value+m) * i32SineTable[ui16Index];

         m++;
         ui16Index = (ui16Index + k) % ui16NumOfPoints;

         if( 0 == (m % SUM_MODULO) )
         {
            i32CoeffAHigh += (i32CoeffATemp >> SCALING_HIGH);
            ui16CoeffALow += (uint16)(i32CoeffATemp & MASKING_LOW);

            i32CoeffBHigh += (i32CoeffBTemp >> SCALING_HIGH);
            ui16CoeffBLow += (uint16)(i32CoeffBTemp & MASKING_LOW);

            i32CoeffATemp = 0;
            i32CoeffBTemp = 0;
         }
      }

      i32CoeffAHigh += (int32)(ui16CoeffALow >> SCALING_HIGH);
      i32CoeffBHigh += (int32)(ui16CoeffBLow >> SCALING_HIGH);
      i32CoeffA[k] = i32CoeffAHigh;
      i32CoeffB[k] = i32CoeffBHigh;
   }
}
```

Bild 15.6 Beispielcode DFT für Integer-Format

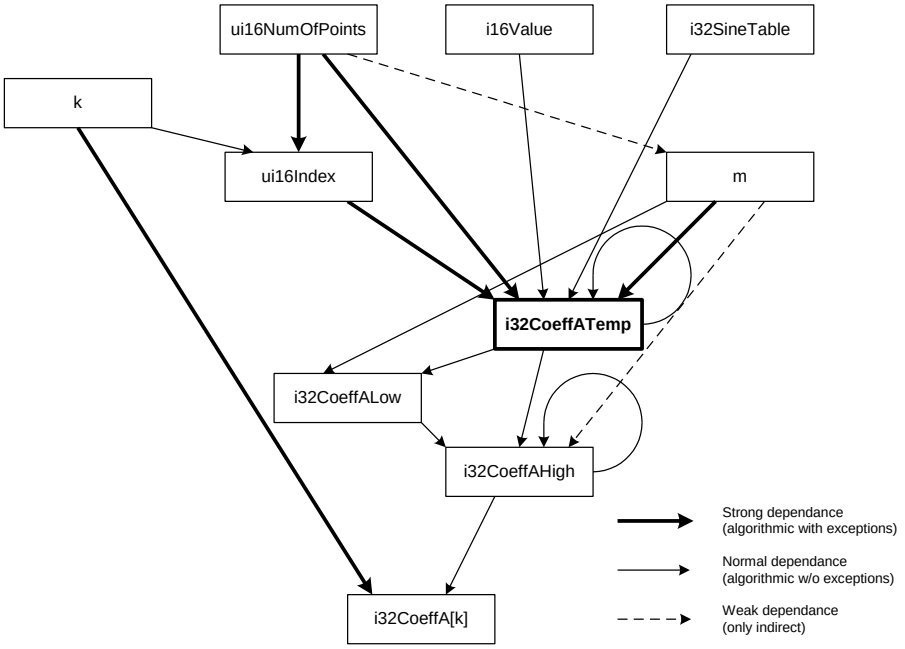


Bild 15.7 Data Dependancy Graph für DFT-Routine aus Bild 15.6

15.2 Unit- und Modultests

Die meisten Software-Entwicklungsmodelle unterscheiden zwischen Modultests, Integrationstests und Systemtests. Modultests sind dabei das erste und wirkungsvollste Instrument, denn durchschnittlich 65% aller nicht schon in Reviews abgefangener Software-Fehler werden hier gefunden.

Daneben wird auch von Unittests gesprochen. Eine Unit (wie auch ein Modul) ist so definiert, dass es sich hierbei um eine semantisch eng zusammengehörende Menge von Softwareelementen innerhalb eines größeren Projekts handelt, also beispielhaft eine Funktion, mehrere Funktionen (auch mit gemeinsamen, globalen Daten), eine Methode, ein Objekt usw.). Meist wird mit Unit eine kleinere Einheit als ein Modul beschrieben, aber dieser Unterschied ist nicht relevant.

Der wesentliche Unterschied zwischen Modul- und Unittest liegt darin, dass der Unittest von der gleichen Person oder Gruppe durchgeführt wird, die diese Unit auch entwickelt hat. Dies muss im Zusammenhang mit dem Software Engineering gesehen werden: Es hat sich in der Praxis als sehr günstig erwiesen, wenn vor der eigentlichen Entwicklung auch bereits die vorgesehenen Tests definiert werden – von den gleichen Personen. Das ist deshalb günstig, weil dann viele Randfälle wie

z.B. Überlauf bei arithmetischen Funktionen etc. beachtet werden und natürlich dann auch in die Implementierung einfließen.

Für den Modultest hingegen gilt die Trennung zwischen Entwicklungs- und Testgruppe, ansonsten gelten gleiche oder ähnliche Regeln für die Vorgehensweise, so dass hier nur vom Modultest gesprochen wird. Für diesen Modultest kann man verschiedene Strategien anwenden. Ein möglicher Weg kann der folgende sein:

1. Man teilt alle Eingangsgrößen (Variablen) in so genannte Äquivalenzklassen ein. Eine Äquivalenzklasse enthält all jene Eingangsgrößen oder Resultate eines Moduls, für die erwartet wird, dass ein Programmfehler entweder alle oder keinen Wert betrifft.

   Beispiel: Die Absolut-Funktion int abs(int) besitzt drei Äquivalenzklassen: negative Werte, die Null und positive Werte.

2. Aus jeder Äquivalenzklasse nimmt man nun zum Test des Moduls mindestens einen Vertreter. Im Testdesign werden die Eingangswerte, die Aktion und die erwarteten Ergebnisse festgelegt. Bei der Testdurchführung werden dann die erwarteten mit den tatsächlichen Ergebnissen verglichen, wobei ggf. ein Toleranzbereich zu definieren ist (z.B. bei Floating-Point-Zahlen).

Dieser Test orientiert sich nicht am inneren Design des Moduls und wird daher auch als ”Black-Box-Test“ bezeichnet. Wichtig ist dabei auch die Erkenntnis, dass ggf. auch Software zum Testen geschrieben werden muss, z.B. zum Aufruf, oder falls auf andere, noch nicht fertige oder nicht getestete Module zurückgegriffen wird. Im letzteren Fall werden die fehlenden Module durch so genannte Programmstümpfe (program stubs) ersetzt.

Der Test wird im Allgemeinen ergeben, dass keineswegs alle Codezeilen durchlaufen wurden. Um dies auch wirklich nachweisen zu können, werden Test-Coverage-Tools eingesetzt. Diese instrumentieren den Originalcode, d.h., sie fügen Code hinzu, der dem Tool den Durchlauf meldet. Nach dieser ersten Testphase werden also weitere Schritte folgen:

3. Der bisherige Test wird analysiert, und die Test Coverage wird bestimmt. Hieraus soll der Tester nun ableiten, mithilfe welcher Eingangswerte er weitere Teile durchlaufen und damit testen lassen kann. Der Test wird dann mit den neuen Werten weitergeführt, bis eine zufriedenstellende Test Coverage erreicht ist.

Diese Form des Tests wird ”White-Box-Test“ genannt, da nun die Eigenschaften des Quellcodes ausgenutzt werden.

Weiterhin entsteht die Frage nach dem Testsystem: Host- oder Target-Testing? Grundsätzlich heißt die Antwort natürlich Zielsystem, denn nur hier können versteckte Fehler wie Bibliotheksprobleme, Datentypabweichungen (wie viele Bits hat int?) usw. erkannt werden. Weiterhin können gemischte C/Assemblerprogramme tatsächlich nur dort getestet werden.

In der Praxis weicht man jedoch häufig auf Hostsysteme aus, weil diese besser verfügbar sind, Festplatte und Bildschirm haben, ggf. schneller sind usw.

15.3 Integrationstests

Der Test der einzelnen Module erscheint vergleichsweise einfach, da insbesondere die Modulkomplexität in der Regel noch begrenzt sein wird. Der nun folgende *Integrationstest* fasst nun mehrere (bis alle) Module zusammen, testet die Schnittstellen zwischen den Modulen und ergibt hiermit den Abschlusstest der Software, da der darauf folgende Systemtest auf das gesamte System einschließlich Hardware zielt.

15.3.1 Bottom Up Unit Tests

Die wohl sicherste Integrationsteststrategie besteht darin, keinen expliziten Integrationstest zu machen und stattdessen die Modultests entsprechend zu arrangieren. Dies wird als *Bottom Up Unit Test* (BUUT) bezeichnet.

Wie beim Black-Box-Modultest, auch als Isolationstest bezeichnet, werden die low-level-Module einzeln getestet, indem sie von einer Testumgebung (stubs, drivers) umfasst werden. Sind diese Module hinreichend getestet, werden sie zu größeren Modulen zusammengefasst und erneut getestet, wobei ”höhere“ Softwaremodule nur auf bereits getestete Module zurückgreifen dürfen.

Der Ansatz hört sich gut an, ist auch wirklich die sauberste Methode, hat aber auch Nachteile:

- Die Entwicklung wird erheblich verlangsamt, da Entwicklung und Test sozusagen Hand in Hand gehen müssen. Zudem ist eine erhebliche Menge an Code zusätzlich zu schreiben (stubs, driver).
- Folglich wird sich die BUUT-Methode auf kleinere Softwareprojekte beschränken.
- Das Softwareprojekt muss von Beginn an sehr sauber definiert sein, d.h., die Modulhierarchie muss streng gewährleistet sein.

15.3.2 Testabdeckung der Aufrufe von Unterprogrammen

Die zweite Methode zum Integrationstest besteht in einer möglichst hohen Abdeckung aller Unterprogrammaufrufe (call pair coverage). Messtechnisch wird der Code hierzu wiederum instrumentiert, d.h. mit zusätzlichem Code zur Messung der Abdeckung versehen. Es wird nun verlangt, eine 100% Call Pair Coverage zu erreichen.

Wird diese Abdeckung nicht erreicht, bedeutet dies, dass die erdachten Fälle zum Integrationstest nicht die volle Systemfunktionalität abdecken, und es muss nachgebessert werden.

15.3.3 Strukturiertes Testen

Die *strukturierten Integrationstests* (SIT) wurden 1982 von Thomas McCabe eingeführt. Sie beruhen darauf, die minimal notwendige Anzahl von voneinander unabhängigen Programmpfaden zu bestimmen. Unabhängig ist dabei ein Programmpfad, wenn er nicht durch eine Linearkombination anderer Programmpfade darstellbar ist.

Ausgangspunkt ist dabei ein Kontrollflussgraph G des Programms (Bild 15.8). Hierin werden die voneinander unabhängigen Programmpfade bestimmt, dies ergibt die so genannte zyklomatische Komplexität (→ 13.2.2.2). Es gilt hier die Formel

$$CC = E - N + 2$$

$$mit\, E = Anzahl\ der\ Kanten, N = Anzahl\ der\ Knoten$$

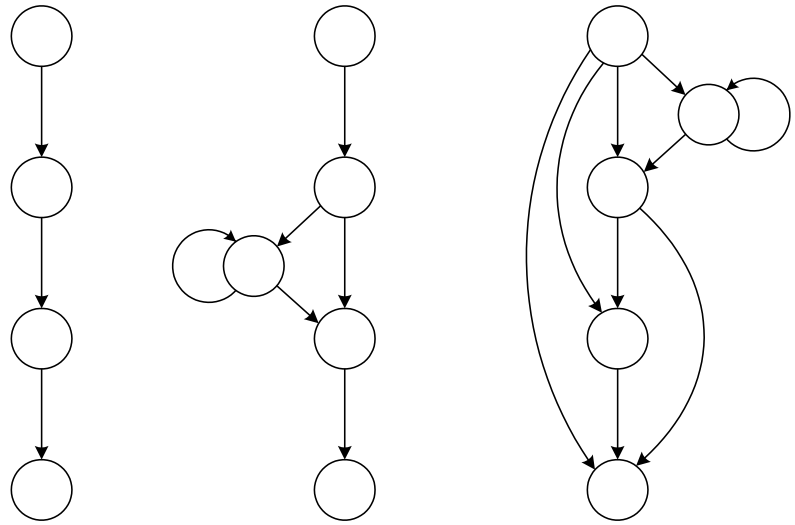


Bild 15.8 Kontrollflussgraphen mit den zyklomatischen Komplexitäten a) 1 b) 3 c) 6

Für den Integrationstest kann der Graph reduziert werden, denn hier sollen ja nur die Aufrufe der Unterprogramme getestet werden. Alle Programmpfade, die keinen solchen Aufruf enthalten, können somit ausgeschlossen werden, allerdings nur unter der Voraussetzung, dass das Dateninterface zu den Unterprogrammen aus-

schließlich über Parameter realisiert ist. In diesem Fall können folgende Operationen zur Reduktion durchgeführt werden:

1. Alle Knoten, die ein Unterprogramm aufrufen, werden markiert.
2. Alle markierten Knoten dürfen nicht entfernt werden.
3. Alle nicht markierte Knoten, die keine Verzweigung enthalten, werden entfernt.
4. Kanten, die zum Beginn einer Schleife führen, die nur unmarkierte Knoten enthält, werden entfernt.
5. Kanten, die zwei Knoten so verbinden, dass kein Alternativpfad für diese Verbindung mit markierten Knoten existiert, werden entfernt.

Der reduzierte Graph muss nun nur noch getestet werden.

15.4 Systemtests

Zum Schluss folgen die *Systemtests*: Sie beziehen sich auf das gesamte System, also die Zusammenfügung von Hard- und Software. Hierbei ist häufig Kreativität gefordert, denn dem Test fehlt ggf. die Außenumgebung.

Einige Möglichkeiten, wie Teiltests aussehen können, seien hier aufgezählt:

- **Belastungs- und Performancetests**: Diese stellen fest, wie das Verhalten unter erwarteter Last (Performancetest) bzw. unter Überlast (Belastungstest) ist. Was hierbei eine Überlast ist, ist wiederum nicht exakt definierbar, aber es gibt Anhaltspunkte. So können Eingaberaten höher sein als die Pollingrate bei Timer-triggered- bzw. Event-triggered-Systemen, Geräte, die das System beeinflussen, werden auf höchste oder niedrigste Geschwindigkeit gestellt usw.
- **Failover und Recovery Test**: Hier wird geprüft, wie sich verschiedene Hardwareausfälle bemerkbar machen, ob beispielsweise Daten verloren gehen, inkonsistente Zustände erreicht werden usw.
- **Ressource Test**: Die im Vordergrund stehende Frage ist hier, ob die Hardwareressourcen ausreichen. Beispiel ist hier der Hauptspeicher, wobei Stack und Heap spezielle Kandidaten sind, denn deren Verhalten ist zumeist unberechenbar. Bei beiden gilt: Großzügige Dimensionierung schafft Vertrauen.
- **Installationstests**: Installationstests verfolgen zwei Ziele: Die Installation der Software muss unter normalen wie abnormalen (zu wenig Speicher, zu wenig Rechte usw.) Bedingungen korrekt verlaufen, und die Software muss danach auch richtig lauffähig sein. Letzteres muss vor allem dann getestet werden, wenn es bereits eine Installation gab.
- **Security Testing**: Dieser Test betrifft die Sicherheit, d.h., inwieweit das System vor Hackern oder anderen Angreifer geschützt ist. Hierzu muss sich der Entwickler so verhalten wie ein Hacker und versuchen, in das System einzudringen.

16 Formale Verifkation

16.1 Einführung

In der Informatik und Softwaretechnik versteht man unter *formaler Verifikation* den mathematischen Beweis, dass ein Programm oder ein Hardwaredesign (also eine konkrete Implementation) der vorgegebenen Spezifikation entspricht. Man spricht dabei auch von *Korrektheit* (→ 16.1.1).

Solche Beweise werden mit Hilfe der Methoden der formalen Semantik geführt. Die Verifikation ist jedoch grundsätzlich nicht in jedem Fall möglich, wie u.a. der Gödelsche Unvollständigkeitssatz (→ 16.1.2) zeigt.

Da Beweise zur Verifikation zumeist außerordentlich groß und oft für den Menschen nicht intuitiv sind, werden interaktive oder automatisierte Theorembeweiser eingesetzt. Erstere basieren auf symbolischer Deduktion, während letztere spezielle Datenstrukturen verwenden. Während erstere zur Lösung sehr allgemeiner Probleme verwendet werden können, sind letztere nur in speziellen Bereichen (dann aber mit geringem Aufwand und geringen Vorkenntnissen) anwendbar.

Zur automatisierten Verifikation werden z. B. häufig Automatenmodelle eingesetzt. Für kleine Systeme mit endlicher Zustandsmenge (zum Beispiel im Hardwaredesign) werden dafür gerne Endliche Automaten eingesetzt, für parallele Prozesse finden Petri-Netze Verwendung. Aber auch andere Automaten können eingesetzt werden. Automaten sind die geeignetere Repräsentation der Problemstellung zum Zwecke der Analyse, da hier gute Algorithmen bekannt sind.

16.1.1 Korrektheit

Unter **Korrektheit** versteht man in der Informatik die Eigenschaft eines Computerprogramms, einer Spezifikation zu genügen. Spezialgebiete der Informatik, die sich mit dieser Eigenschaft befassen, sind die Formale Semantik und die Berechenbarkeitstheorie.

Nicht abgedeckt vom Begriff *Korrektheit* ist, ob die *Spezifikation* die vom Programm zu lösende Aufgabe korrekt beschreibt (dies wird Validierung genannt, → 14.2).

Definition 16.1:

Ein Programmcode wird bezüglich einer Vorbedingung P und der Nachbedingung Q **partiell korrekt** genannt, wenn bei einer Eingabe, die die Vorbedingung P erfüllt, jedes Ergebnis die Nachbedingung Q erfüllt. Dabei ist es noch möglich, dass das Programm nicht für jede Eingabe ein Ergebnis liefert, also nicht terminiert.

Definition 16.2:

Ein Code wird **total korrekt** genannt, wenn er partiell korrekt ist und zusätzlich für jede Eingabe, die die Vorbedingung P erfüllt, terminiert. Aus der Definition folgt sofort, dass total korrekte Programme auch immer partiell korrekt sind.

Der Nachweis der Korrektheit eines Programms kann jedoch nicht in allen Fällen geführt werden: das folgt aus dem Gödelschen Unvollständigkeitssatz (→ 16.1.2. Auch wenn die Korrektheit für Programme, die bestimmten Einschränkungen unterliegen, bewiesen werden kann, so zählt die Korrektheit von Programmen allgemein zu den nicht-berechenbaren Problemen.

16.1.2 Unvollständigkeitssatz von Gödel

Der Mathematiker Kurt Gödel wies mit seinem im Jahre 1931 veröffentlichten Unvollständigkeitssatz nach, dass man in (hier stets als rekursiv aufzählbar vorausgesetzten) Systemen wie der Arithmetik nicht alle Aussagen formal beweisen oder widerlegen kann. Sein Satz besagt:

> *Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.*

Eine einfache Formulierung des ersten Unvollständigkeitssatzes sowie des daraus unmittelbar folgenden zweiten Gödelschen Unvollständigkeitssatzes lautet:

> *In jedem formalen System der Zahlen, das zumindest eine Theorie der Arithmetik der natürlichen Zahlen enthält, gibt es einen unentscheidbaren Satz, also einen Satz, der nicht beweisbar und dessen Negierung ebenso wenig beweisbar ist. (1. Gödelscher Unvollständigkeitssatz).*

Daraus folgt unmittelbar, dass kein formales System der Zahlen, das zumindest eine Theorie der natürlichen Zahlen samt Addition und Multiplikation enthält, sich innerhalb seiner selbst als widerspruchsfrei beweisen lässt (2. Gödelscher Unvollständigkeitssatz).

16.2 Petri-Netze

Ein **Petri-Netz** ist ein mathematisches Modell von nebenläufigen Systemen. Es ist eine formale Methode der Modellierung von Systemen bzw. Transformationsprozessen. Die ursprüngliche Form der Petri-Netze nennt man auch Bedingungs- oder Ereignisnetz. Petri-Netze wurden durch Carl Adam Petri in den 1960er Jahren definiert. Sie verallgemeinern wegen der Fähigkeit, nebenläufige Ereignisse darzustellen, die Automatentheorie.

Ein Petri-Netz ist ein bipartiter und gerichteter Graph. Er besteht aus Stellen (*Places*) und Übergängen bzw. Transitionen (*Transitions*; Übergänge zur Verarbeitung von Informationen, ähnlich einem Schalter). Stellen und Transitionen sind

durch gerichtete Kanten verbunden. Es gibt keine direkten Verbindungen zwischen zwei Stellen oder zwei Transitionen.

16.2.1 Aufbau der Petri-Netze

Stellen (places) werden als Kreise, Transitionen (transistions) als Rechtecke dargestellt. Jede Stelle hat eine Kapazität und kann entsprechend viele Token (Marken bzw. Zeichen) enthalten. Ist keine Kapazität angegeben, steht das für unbegrenzte Kapazität – oder für eins. Jeder Kante ist ein Gewicht zugeordnet, das die *Kosten* dieser Kante festlegt. Ist einer Kante kein Gewicht zugeordnet, wird der Wert eins verwendet.

Sind alle Kapazitäten der Stellen und Gewichte der Kanten eines Petri-Netzes 1, so wird es auch als Bedingungs-, Prädikat- oder Ereignis-Netz bezeichnet.

Die Belegung der Stellen heißt *Markierung* und ist der Zustand des Petri-Netzes.

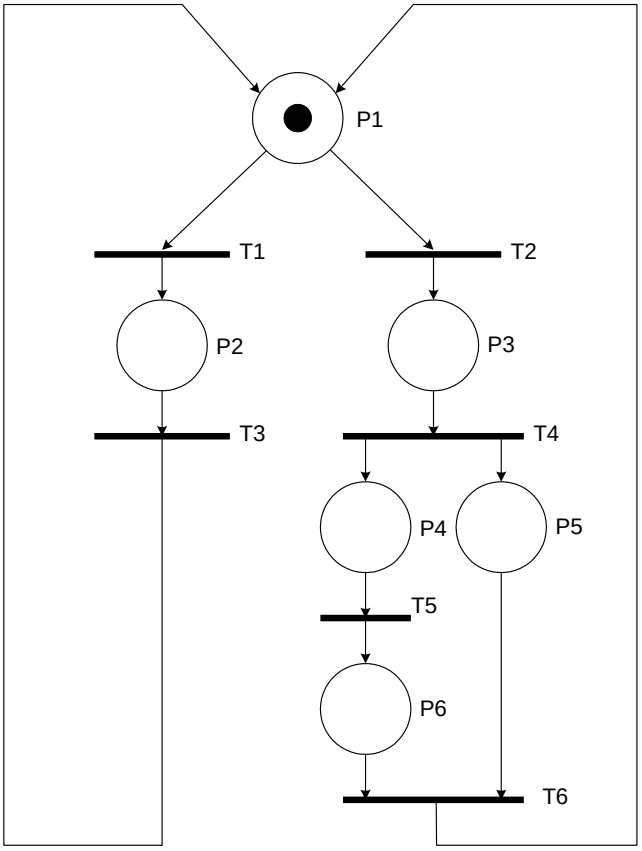


Bild 16.1 Beispiel für ein einfaches Petri-Netz

Transitionen sind *aktiviert* bzw. *schaltbereit*, falls sich in allen Eingangsstellen mindestens so viele Marken befinden, wie die Transitionen Kosten verursacht und alle Ausgangsstellen noch genug Kapazität haben, um die neuen Marken aufnehmen zu können. Schaltbereite Transitionen *können* zu einem beliebigen Zeitpunkt schalten. Beim Schalten einer Transition werden aus deren Eingangsstellen entsprechend den Kantengewichten Marken entnommen und bei den Ausgangsstellen entsprechend den Kantengewichten Marken hinzugefügt. Marken werden in einem Petri-Netz *nicht* bewegt. Sie werden entfernt und erzeugt!

Die Marken eines Petri-Netzes sind in ihrer einfachsten Form voneinander nicht unterscheidbar. Für komplexere, aussagekräftigere Petri-Netze sind *Markeneinfärbungen*, *Aktivierungszeiten* und Hierarchien definiert worden.

16.2.2 Wichtige Begriffe

Definiton 16.3: Lebendigkeit

Eine Transition heißt

- *tot*, falls sie unter keiner Folgemarkierung aktiviert ist.
- *aktivierbar*, falls sie unter mindestens einer Folgemarkierung aktiviert ist.
- *lebendig*, falls sie in jeder erreichbaren Markierung aktivierbar ist.

Ein Petri-Netz heißt

- *tot*, falls alle Transitionen tot sind.
- *todesgefährdet*, falls das Petri-Netz unter einer Folgemarkierung tot ist.
- *verklemmungsfrei* oder *schwach lebendig*, falls es unter keiner Folgemarkierung tot ist.
- *(stark) lebendig*, falls alle Transitionen lebendig sind.

Definition 16.4: Erreichbarkeit

Eine Markierung eines Petri-Netzes heißt *erreichbar*, falls es eine Schaltsequenz der Transitionen gibt, welche die Startmarkierung in diese Markierung überführt.

Definition 16.5: Konservativität

Ein Petri-Netz heißt *konservativ*, falls die (beliebig) gewichtete Summe der Marken konstant ist.

Definition 16.6: Beschränktheit

Ein Petri-Netz heißt *b-beschränkt*, wenn es eine Schranke *b* gibt, so dass nie mehr als *b* Marken in einer Stelle liegen.

Definition 16.7: Sicherheit

Ein Petri-Netz heißt *sicher*, falls es 1-beschränkt ist.

Definition 16.8: Konflikt

Es besteht ein Konflikt bei einer nicht nebenläufigen Aktivierung von zwei Transitionen. Im Vorbereich bedeutet das, dass zwei Transitionen die gleiche Marke benötigen, um zu schalten. Im Nachbereich sind es zwei Transitionen, die Marken erzeugen können, aber die Kapazität nicht für beide ausreicht.

Definition 16.9: Kontakt

Eine Transition hat *Kontakt*, wenn sie nicht schalten kann, weil sonst die Kapazität einer Stelle im Nachbereich überschritten würde.

16.2.3 Formale Definition

Ein Petri-Netz ist ein 6-Tupel (S, T, F, K, W, m_0). Durch das 3-Tupel (S, T, F) ist ein bipartiter und gerichteter Graph definiert.

- S, nichtleere Menge von Stellen $S = \{s_1,s_2,...,s_{|S|}\}$
- T, nichtleere Menge von Transitionen (Übergängen) $T = \{t_1,t_2,...,t_{|T|}\}$
- F, nichtleere Menge der Kanten (Flussrelation) $F \subseteq (S \times T) \cup (T \times S)$
- K, Kapazitäten der Plätze, Kapazitätsfunktion $K: S \rightarrow \aleph \cup \{\infty\}$
- W, Kosten der Kanten, Gewichtsfunktion $W: F \rightarrow \aleph$
- m_0, Startmarkierung $m_0: S \rightarrow \aleph$

Die Mengen der Stellen S und Transitionen T sind disjunkt Die aktuelle Markierung $m: S \rightarrow \aleph$ bezeichnet man als *Zustand des Petri-Netzes*. m(s) ist die Anzahl der Marken auf Stelle s.

Folgende (Teil-)Mengen sind für $t \in T$ definiert:

- *Vorbereich* $\bullet t = \{s \in S | (s, t) \in F\}$, also alle Stellen, von denen eine Kante zur Transition t führt,
- *Nachbereich* $t\bullet = \{s \in S | (t, s) \in F\}$, also alle Stellen, zu denen eine Kante von der Transition t aus führt.

16.2.4 Schaltbereitschaft

Eine Transition t heißt *aktiviert*, *schaltbereit* oder *hat Konzession*, falls gilt:

1. $\forall s \in \bullet t \setminus t\bullet: m(s) \geq W(s, t)$
2. $\forall s \in t\bullet \setminus \bullet t: K(s) \geq m(s) + W(t, s)$
3. $\forall s \in t\bullet \cap \bullet t: K(s) \geq m(s) - W(s, t) + W(t, s)$

16.2.5 Schaltvorgang

Eine aktivierte Transition t *kann* schalten. Falls sie schaltet, werden für alle Stellen s die Anzahl der Marken m‘(s) wie folgt neu berechnet:

$$m'(s) = \begin{cases} m(s) - W(s,t) & \text{falls } s \in \bullet t \text{ und } s \notin t\bullet \\ m(s) + W(t,s) & \text{falls } s \notin \bullet t \text{ und } s \in t\bullet \\ m(s) - W(s,t) + W(t,s) & \text{falls } s \in \bullet t \text{ und } s \in t\bullet \\ m(s) & \text{falls } s \notin \bullet t \text{ und } s \notin t\bullet \end{cases}$$

Aus einer Stelle s im Vorbereich der Transition t werden beim Schalten so viele Marken entnommen, wie es die Kosten W an der entsprechenden Kante (s, t) vorgeben. Entsprechend werden Marken im Nachbereich erzeugt. Eine Stelle s, die weder im Vor- noch im Nachbereich der Transition t liegt, wird nicht verändert, wenn die Transition t schaltet.

16.2.6 Inzidenzmatrix

Die Inzidenzmatrix C eines Petri-Netzes zeigt jeweils an, wie sich die Markenzahl einer Stelle s_i (dargestellt durch die Zeilen der Matrix) durch ein Schalten der Transition t_j (Spalten der Matrix) vergrößert oder verringert.

Sie ist definiert durch:

$$C_{ij} := \begin{cases} -W(s_i,t_j), & \text{falls } s_i \in \bullet t_j \text{ und } s_i \notin t_j\bullet \\ W(t_j,s_i), & \text{falls } s_i \notin \bullet t_j \text{ und } s_i \in t_j\bullet \\ W(t_j,s_i) - W(s_i,t_j), & \text{falls } s_i \in \bullet t_j \text{ und } s_i \in t_j\bullet \\ 0 & \text{sonst,} \end{cases}$$

für $1 \le i \le m$, $1 \le j \le n$. Hierbei ist zu beachten, dass Schlingen nur mit der Differenz ihrer beiden Kantengewichte, bei Gleichheit also gar nicht, das heißt nur als 0, in der Inzidenzmatrix C auftauchen.

16.2.7 Erweiterte Petri-Netze

Um mit Petri-Netzen genauere Modelle aufstellen zu können, wurden diese im Laufe der Zeit um neue Elemente erweitert. Daraus entstanden neue Klassen von Petri-Netzen, die einerseits mächtiger sind, andererseits aber schwerer oder gar nicht geschlossen analysiert werden können.

16.2.7.1 Prioritäten und hemmende Kanten

- Prioritäten werden als Zahlen, beginnend mit 1, neben einer Transition notiert. Wenn zwei zeitlose Transitionen aktiviert sind, schaltet die mit der höheren Priorität.
- Hemmende Kanten verbinden Stellen mit Transitionen. Wenn eine hemmende Kante durch eine Marke in ihrer Ausgangsstelle aktiviert ist, kann die verbundene Transition nicht feuern, auch wenn alle ihre anderen eingehenden Kanten aktiviert sind.

Durch Prioritäten und hemmende Kanten erreichen Petri-Netze die Mächtigkeit der Turingmaschine.

16.2.7.2 Zeiterweiterte Petri-Netze

Zusätzlich zu den zeitlosen Transitionen der klassischen Petri-Netze wurden Transitionen eingeführt, welche beim Schalten Zeit verbrauchen. Dabei unterscheidet man verschiedene Klassen von Netzen, je nachdem welche Art von zeitverbrauchenden Transitionen in ihnen vorkommt:

- SPN (Stochastic Petri Net): Jede Transition verbraucht Zeit. Die Zeit, die eine Transition beim Schalten verbraucht, ist eine Zufallsvariable und exponentialverteilt. Diese Klasse von Petrinetzen eignet sich nicht zur Modellierung von Synchronisation.
- GSPN (Generalized Stochastic Petri Net): Enthalten exponentialverteilte zeitbehaftete Transitionen und zeitlose Transitionen. Diese Klasse von Petri-Netzen lässt sich noch geschlossen analysieren. Warteschlangensysteme lassen sich als sehr einfache GSPNs darstellen.
- DSPN (Deterministic Stochastic Petri Net): Neben den exponentialverteilten zeitbehafteten Transitionen gibt es auch solche mit deterministischer Schaltzeit. Dieser werden als ausgefüllte Rechtecke gezeichnet. Die Komplexität von DSPNs ist ungleich höher als die von GSPNs: Nur wenn in einem DSPN nie mehr als eine deterministische Transition aktiviert sein kann, ist es überhaupt noch analysierbar. Dennoch sind DSPNs beliebt, da sich viele Sachverhalte mit ihnen sehr viel genauer modellieren lassen und gute numerische Simulationsmethoden existieren.
- Allgemeine Stochastische Petrinetze mit beliebigen Schaltzeitverteilungen.

16.2.7.3 Farbige Petri-Netze

Farbige oder colorierte Petri-Netze erweitern Marken um verschiedenen „Farben“, indem die Marken nun strukturiert und typisiert sind. Während Marken bei normalen Petri-Netzen nicht unterschieden werden können, ist dies durch die Färbung der Marken möglich.

16.2.7.4 Attributierte Petri-Netze

Dies stellt eine verallgemeinerte Form der Petri-Netze dar, bei der man Transitionen, Stellen und Konnektoren mit Attributen versehen kann.

- Die Attributierung der Konnektoren dient dazu, Daten und Objekte durch das Petrinetz zu transportieren.
- Die Attributierung der Stellen dient der Regulierung deren Kapazität.
- Die Attribute von Transitionen dienen der Zeitmodellierung, Verarbeitung der transportierten Daten sowie der bedingten Ausführung von Transitionen.

16.3 Formale Verifikation

Die formale Verifikation wird derzeit besonders auf dem Gebiet des Schaltkreisentwurfs eines mikroelektronischen Systems, also der Hardware, betrieben. 60-80% der Kosten für diesen Entwurf entfallen dabei auf die Verifikation. Dies bedeutet, dass die Produktivität der mikroelektronischen Industrie durch die Kosten für die Verifikation begrenzt ist (*verification gap*).

Für den Designer eines System-on-Chips (SoC) wird es zugleich immer schwieriger, festzustellen, ob sein Entwurf fehlerfrei ist, d.h. ob er eine korrekte und vollständige Spezifikation des Entwurfs angegeben hat und ob diese Spezifikation fehlerfrei in eine Implementierung umgesetzt wurde. Dies zu überprüfen ist die Aufgabe der *Entwurfsvalidierung* und der *Verifikation*.

16.3.1 Simulation

Das traditionelle Hilfsmittel zur Entwurfsvalidierung, d.h. der Prüfung, ob die Spezifikation korrekt und vollständig ist, ist die Simulation. Die in der Praxis auftretenden Schaltungen und Systeme sind jedoch viel zu groß, um auch nur annähernd alle möglichen Eingaben an das System zu simulieren. So ist es möglich, dass trotz aufwendiger und langwieriger Simulationen Entwurfsfehler oder *Bugs* übersehen werden. Dieser Fall tritt in der industriellen Praxis immer wieder ein und macht sogenannte *Redesigns* erforderlich.

Redesigns sind mit enormen Kosten verbunden. Je später ein Fehler erkannt wird, umso mehr Spezifikations- und Implementierungsschritte müssen neu durchgeführt werden. Allein die Maskenkosten für einen neuen Satz von Herstellungsmasken kosten bei aktueller Technologie mehrere Millionen €!

Die Simulation erfordert außerdem die Entwicklung von sogenannten *Test Benches*. Die Kosten für die Erstellung von *Test Benches* sowie die extrem langen Simulationszeiten führen dazu, dass man Simulationsmethoden heute an bestimmten Stellen des *Design Flows* durch *formale* Verifikationsmethoden ersetzt. Unter formaler Verifikation versteht man den Vorgang, mit mathematischen Methoden bestimmte Eigenschaften eines Systems exakt und vollständig nachzuweisen.

Doch hier ist Vorsicht angebracht: Die Validierung (→ 14.2) versucht, die Übereinstimmung der informellen Beschreibung der Aufgabenstellung mit der formalen Beschreibung des daraus resultierenden Systemmodells aufzuzeigen. Dies ist mathematisch nicht möglich, da die erste Beschreibung eben nur informell ist.

Hier bieten sich zwei Auswege an: Zum einen eben doch die Simulation, die möglichst frühzeitig z.B. unter Verwendung von *executable specifications* versucht, das gewünschte Verhalten zu testen, oder eine indirekte Methode, die aus der Extraktion von Eigenschaften aus der informellen Beschreibung und dm mathematischen Nachweis dieser im Modell besteht (property checking, → 16.3.3).

16.3.2 Equivalence Checking

Man möchte zeigen, dass die vorliegende Implementierung eines mikroelektronischen Systems, die nach einer langen Serie von Entwurfsverfeinerungen (Optimierungen) aus der ursprünglichen Spezifikation hervorgegangen ist, tatsächlich noch funktional äquivalent zur Spezifikation ist. Ein Addierer, der nach allen Regeln der Kunst optimiert wurde, muss nach wie vor das korrekte Additionsergebnis entsprechend seiner Spezifikation liefern. Dies zu überprüfen ist die Aufgabe des *Äquivalenzvergleichs* (*equivalence checking*).

Kommerzielle Werkzeuge zum *equivalence checking* werden inzwischen standardmäßig in den meisten industriellen *Design Flows* eingesetzt und haben etwa seit Mitte der 90er Jahre traditionelle Verfahren basierend auf Simulation ersetzt. Der Äquivalenzvergleich ist ohne Zweifel die erste formale Verifikationsmethode, die weltweit im großen Umfang industriell eingesetzt wird.

Folgende Methoden für den Äquivalenzcheck sind zurzeit gebräuchlich:

- Vergleich von Normalformen (Binary Decision Diagrams, BDD)
- Werkzeuge zur Erfüllbarkeitsprüfung (SAT)
- Werkzeuge zur automatischen Testmustererzeugung (ATPG, Automatic Test Pattern Generation)

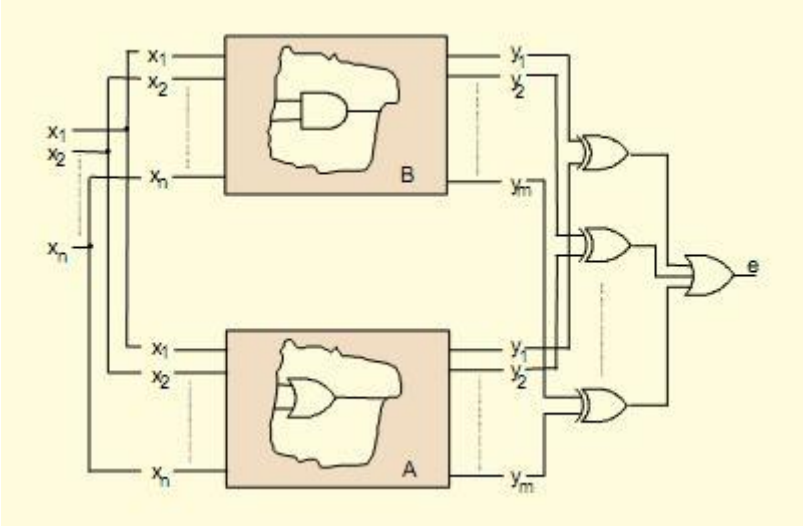


Bild 16.2 Zur formalen Verifikation von Schaltkreisen (Erfüllbarkeitsprüfung)

Bei der Erfüllbarkeitsprüfung (siehe Bild 16.2) geht es darum, Eingangsbedingungen zu finden, für die der Ausgang den Wert ’1’ annimmt. In diesem Fall wären die

Schaltungen nicht identisch. Erfüllbarkeitsprüfungen lassen sich auf kombinatorische Schaltungen anwenden.

Binary Decision Diagrams (BDD) sind azyklische, gerichtete Graphen, die eine Wurzel besitzen. Jeder Knoten im Graphen ist entweder ein innerer Knoten (dann hat ein zwei Ausgänge, einen für den Wert ’0’, einen für ’1’) oder ein Blatt (das keinen Ausgang hat und nur die Bezeichnungen ’0’ und ’1’ haben kann).

Bild 16.3 zeigt zwei scheinbar verschiedene BDDs für die Logikfunktion y = (a * b) + c.

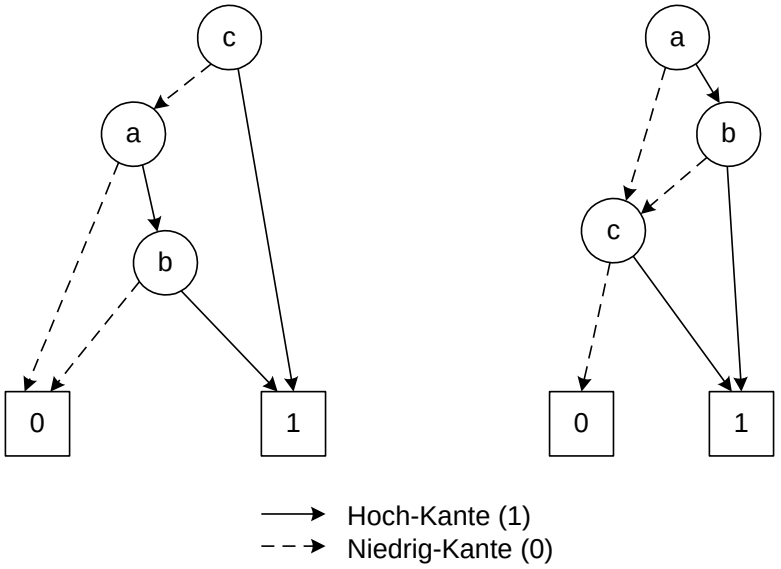


Bild 16.3 BDDs für (a * b) + c

Wie leicht zu erkennen ist: Die Reihenfolge der Variablen ist entscheidend für das „Aussehen“ des BDD. Wird der BDD für eine bestimmte Reihenfolge der Variablen definiert, spricht man von Ordered BDD (OBDD).

Die Aufgabe im Äquivalenzcheck liegt nun darin, die Äquivalenz für die BDDs zu beweisen. Dies kann je nach Größe sehr zeitaufwendig sein. Als Konsequenz beschränkt man sich auf lokale BDDs.

Die Nutzung von BDDs ist zunächst auch auf kombinatorische Probleme (ohne Speicher) ausgelegt, kann jedoch auch auf sequenzielle Probleme erweitert werden [FSF+10].

16.3.3 Property Checking

Neben der Äquivalenz von Spezifikation und Implementierung will man aber auch die Korrektheit der Spezifikation selbst überprüfen (Validierung). Dazu wird ein Satz von Eigenschaften festgelegt, von dem man erwartet, dass er durch die Spezifikation erfüllt wird. Hat man beispielsweise eine Steuerschaltung für eine Verkehrsampel entworfen, möchte man u.a. nachweisen, dass unter keinen Umständen in der Schaltung eine Signalbelegung auftreten kann, die dazu führt, dass die Ampeln für zwei sich kreuzende Straßen gleichzeitig „grün“ zeigen.

Den formalen Nachweis solcher Eigenschaften bezeichnet man oft als *property checking* oder *model checking*. *Property checking* führt auf noch weitaus komplexere algorithmische Probleme als *equivalence checking*, insbesondere, wenn nicht nur die rein logische, kombinatorische Schaltfunktion, sondern das sequenzielle Verhalten der Schaltung insgesamt und ihr Zusammenwirken mit anderen Komponenten eines Systems überprüft werden soll.

Durch große wissenschaftliche Fortschritte in den letzten Jahren ist auch im Bereich *property checking* inzwischen ein Stand erreicht, der diese Technik industriell einsetzbar macht (bei mikroelektronischen Schaltkreisen). Formales *property checking* wird zurzeit sehr erfolgreich in der Industrie verwendet, um die Blöcke (ca. 50.000- 100.000 Gatter) eines SoCs (bis 100 Mio. Gatter) formal zu verifizieren. Dadurch wird höchste Qualität bei den einzelnen Blöcken erzielt und die Systemverifikation, die das System als Ganzes betrachtet, massiv entlastet.

Literatur

[arm] http://www.arm.com/

[ASU99] *Alfred V. Aho, Rave Sethi, Jeffrey D. Ullman*, ”Compilerbau 1“. 2. Auflage, Oldenbourg Verlag, München, 1999.

[BBM00] *Benini, L.; Bogliolo, A.; De Micheli, G.*: A Survey of Design Techniques for System-Level Dynamic Power Management. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 3, pp. 299–316 (2000).

[BFS+03] *Blume, H., Feldkämper, H.T., von Sydow, T., Noll, T.G.*, ”Auf die Mischung kommt es an“. Elektronik 53(19) S. 54–64 und Elektronik 53(20) S. 62–67 (2003).

[BH04] *Beierlein, T.; Hagenbruch, O. (Hrsg.)*: ”Taschenbuch Mikroprozessortechnik”. Fachbuchverlag Leipzig im Carl Hanser Verlag, München Wien, 3., aktualisierte und erweiterte Auflage, April 2004. ISBN 3-446-22072-0

[BHF+02] *Blume, H., Hübert, H., Feldkämper, H.T., Noll, T.G.*, ”Model-based Exploration of the Design Space for Heterogeneous Systems on Chip”. Proceedings of the Workshop Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, April 2002.

[BHU+10] *Bohli, J.-M., Hessler, A., Ugus, O., Westhoff, D.*, „Security Solutions for Uplink- and Downlink-Traffic in Wireless Sensor Networks“. It – information technology 52(6) pp. 313-319 (2010).

[Bro+00] *Brooks, D. et.al.*: Power-Aware Microarchitecture. IEEE Micro Vol. 20, No. 6, pp. 26–44 (2000).

[Büc10a] *Frank Büchner*, ”Acht Irrtümer über CodeCoverage“. Embedded Software Engineering 5(1) S. 12-13 (2010).

[Büc10b] *Frank Büchner*, ”Fünf Irrtümer über CodeCoverage“. Embedded Software Engineering 5(2) S. 14 (2010).

[CKURS] http://www.hmh-ev.de/files/ckurs.pdf

[CL07] *Cullmann, Xavier-Noël, Lambertz, Klaus*, “Komplexität und Qualität von Software“. MSCoder 1/2007(2), pp. 36-43 (2007).

[CW02] *Cardoso, João M.P., Weinhardt, Markus*, “XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture”. Field-Programmable Logic and Applications FPL 2002.

[Dea04] *Alexander G. Dean*, ”Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers”. IEEE Micro 24(4), pp. 10-22 (2004).

[DW99] *Dehon, A., Wawrzynek, J.*: ”Reconfigurable Computing: What, Why and Implications for Design Automation”. Design Automation Conference DAC99, San Francisco, 1999.

[FDS09] *Fischer, Daniel, Drosdezki, Eugenia, Schuraev, Konstantin,* "Software-Metriken gezielt einsetzen". Proceedings Embedded Software Engineering Kongress 2009, pp. 346-354, Sindelfingen 2009.

[FH10] *Falk, R., Hof, H.-J.,* "Security Design for Industrial Sensor Networks". It – information technology 52(6) pp. 331-339 (2010).

[FSF+10] Görschwin Fey, André Süflow, Stefan Frehse, Rolf Drechsler, "Automatische formale Verifikation der Fehlertoleranz von Schaltkreisen". It+ti 52(4), S. 216-223 (2010)

[GM03] *Timo Gramann, Dirk S. Mohl*, "Precision Time Protocol IEEE 1588 in der Praxis". Elektronik 52(24), S. 86–94 (2003).

[Grü04a] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 1: Das Handwerk des Testens will gelernt sein, wird aber kaum gelehrt". Elektronik 53(22) S. 60 .. 72 (2004).

[Grü04b] Stephan Grünfelder, Neil Langmead "Den Fehlern auf der Spur. Teil 2: Modultests: Isolationstests, Testdesign und die Frage der Testumgebung". Elektronik 53(23) S. 66 .. 74 (2004).

[Grü05a] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 3: Automatische statische Codeanalyse". Elektronik 54(9) S. 48 .. 53 (2005).

[Grü05b] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 4: Integrationstests – das ungeliebte Stiefkind". Elektronik 54(13) S. 73 .. 77 (2005).

[Grü06] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 5: Systemtests – die letzte Teststufe ist alles andere als eine exakte Wissenschaft". Elektronik 55(14) S. 45 .. 51 (2006).

[Ha177] *Halstead, Maurice H.*: "Elements of Software Science". In: Operating and Programming Systems Series, Vol. 7, New York, Elsevier,1977.

[Hat95] *Les Hatton*, "Safer C: Developing Software for High Integrity and Safety Critical Systems". McGraw-Hill Professional, 1995.

[Hol06] *Gerad J. Holzmann*, "The Power of 10: Rules for Developing Safety-Critical Code". IEEE Computer 39(6), pp. 95–97, 2006.

[IBM750] https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/292763D22B80DFEF872570C1006DF928/$file/750GX_Lockstep_3-19-08.pdf

[IEE1588] http://ieee1588.nist.gov/

[intel] http://www.intel.com/

[Jon00] *Jones, Capers,* "Software Assessments, Benchmarks, and Best Practices". Addison-Wesley Professional, 2000. ISBN 978-0-201-48542-4

[KB94] Kowalk, W.P.; Burke, M.: Rechnernetze. Teubner-Verlag, Stuttgart, 1994. ISBN 3-519-02141-2

[Lance2] http://www.lancecompiler.com/

[Laplace] Laplace-Operator: http://www.iiit.uni-karlsruhe.de/download/Versuch8_Bildverarbeitung_06.pdf

[lint] http://www.splint.org/

[Man03] *de Man, H.,* "Designing Nano-Scale Systems for the Ambient Intelligent World". it – Information Technology 45(6), S.310-317 (2003).

[MarO9] *Martin, Robert C*, "CleanCode, A Handbook of Agile Software Craftsmanship", PearsonEducation, 2009

[McC76] *McCabe, Thomas J.*, "A Complexity Measure." IEEE Transactions on Software Engineering, Vol. SE-2,No.4, pp. 308-320, 1976.

[mips] http://www.mips.com/

[MNW03] *Martin, A.J., Nyström, M., Wong, C.G.*, "Three Generations of Asynchronous Microprocessors". IEEE Design&Test 20(6), pp. 9-17 (2003)

[MP01] *Mudge, T.*: Power: A First-Class Architectural Design Constraint. IEEE Computer Vol. 34, No. 4, pp. 52–58 (2001).

[Mül03] *Müller-Schloer, C. et.al*, "Organische Computersysteme". PARS Mitteilungen Nr. 20, ISSN 0177-0454, November 2003.

[NIST] http://ieee1588.nist.gov/

[OMM02] Nahmsuk Oh, Subhasish Mitra, Edward J. McCluskey, "ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions". IEEE Transactions on Computers Vol. 51, No. 2, pp. 180-200 (2002)

[pact] http://www.pactcorp.com

[Sch05] *Scholz, P.*: Softwareentwicklung eingebetteter Systeme. Springer Verlag Berlin Heidelberg New York, 2005.

[SFS+05] *Christian Siemers, Rainer Falsett, Reinhard Seyer, Klaus Ecker*, "Reliable Event-Triggered Systems for Mechatronic Applications". The Journal of Systems and Software 77, pp. 17–26 (2005).

[Sie04] *Christian Siemers*, "Prozessor-Technologie". tecCHANNEL-Compact, Verlag Interactive GmbH, München, Mai 2004.

[Sie05a] *Christian Siemers,* "Die Welt der rekonfigurierbaren Prozessoren, Teil 1: Lösungen auf dem Weg zum konfigurierbaren System-on-Chip". Elektronik 54(21) S. 42-48 (2005).

[Sie05b] *Christian Siemers,* "Die Welt der rekonfigurierbaren Prozessoren, Teil 2: Aktuelle Produkte und deren Zielmärkte". Elektronik 54(22) S. 42-48 (2005).

[Sie07a] *Christian Siemers*, ”Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 1“. ElektronikPraxis 43, Sonderheft 1/2007 Embedded Systems, S. 46-47 (2007)

[Sie07b] *Christian Siemers*, ”Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 2“. Embedded Software Engineering Report 2(2), S. 6, und http://www.ese-report.de/ (2007)

[Sie07c] *Christian Siemers*, ” Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 3“. Embedded Software Engineering Report 2(3), S. 3, und http://www.ese-report.de/ (2007)

[Sie08] *Christian Siemers*, ”Die unsichtbaren Störenfriede aus dem Kosmos“. ElektronikPraxis 44, Sonderheft Marktreport Embedded Systeme, S. 48-50, ESE-Report 3(1), S. 8 und http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/ implementierung/ articles/104812 (2008)

[Sie10] *Christian Siemers*, ”Der applikationsinterne Ereignis-Scheduler“. Embedded Software Engineering 5(2) S. 12 .. 13 (2010)

[SPW09] *Bianca Schroeder, Eduardo Pinheiro, Wolf-Dietrich Weber*, ” DRAM Errors in the Wild: A Large-Scale Field Study”. SIGMETRICS/Performance’09, Seattle, USA, 2009

[suif] SUIF compiler system: http://suif.stanford.edu

[SWM01] *Steinke, S.; Wehmeyer, L.; Marwedel, P.*: Software mit eingebautem Power-Management. Elektronik Vol. 50, H. 13, S. 62–67 (2001).

[SWW99] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

[TM05] *Emil Talpes, Diana Marculescu*, ”Toward a Multiple Clock/Voltage Island Design Style for Power-Aware Processors”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(5), pp. 591 – 603 (2005).

[Wiki_4B5B] http://de.wikipedia.org/wiki/4B5B-Code

[Wiki_AES] http://de.wikipedia.org/wiki/Advanced_Encryption_Standard

[Wiki_CMF] http://en.wikipedia.org/wiki/Common_mode_failure

[Wiki_CRC] http://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung

[Wiki_DES] http://de.wikipedia.org/wiki/Data_Encryption_Standard

[Wiki_FP] http://de.wikipedia.org/wiki/Function-Point-Verfahren

[Wiki_Ham] http://de.wikipedia.org/wiki/Hamming-Code

[WL01] *Weinhardt, M., Luk, W.*, ”Pipeline Vectorization“. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 2001, pp. 234-248.

[www_hh02] http://tech-www.informatik.uni-hamburg.de/lehre/ss2002/pc-technologie/docs/microprocessor-summary.pdf.

Sachwortverzeichnis

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z