Embedded Systems Engineering

V1.01

Prof. Dr. Christian Siemers

TU Clausthal,
Institut für Informatik

Clausthal-Zellerfeld

Inhaltsverzeichnis

1 Einführung in eingebettete Systeme

Eingebettete Systeme (*embedded systems*) sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind [3]. 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 Vorlesung ist dem Design solcher Systeme gewidmet.

Die Vorlesung wurde so konzipiert, dass weder Software noch Hardware im Vordergrund stehen sollen. Es geht um (binärwertige) digitale Systeme, die programmierbar sind, 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 gemeint sein kann. Um hier Verwirrungen zu vermeiden, sei für diese Vorlesung 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.

Damit ist das Feld etwa abgesteckt, in dem sich die Vorlesung und dieses Skript bewegen. Es ist zugleich deutlich, dass es sich um ein interdisziplinäres Feld handelt, da viele Komponenten hier hineinspielen werden. In diesem ersten Kapitel werden einige Begriffe definiert, um für Einheitlichkeit zu sorgen.

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 [3]: (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 [3]. 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 [3]. 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 [3]. 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, sind nahezu immer als reaktives System ausgebildet. Interaktive Systeme sind zwar prinzipiell möglich, doch die Einbettung macht in der Regel eine Reaktivität notwendig. Die wichtigsten 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.

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. Echtzeitfähige eingebettete Systeme sind eine echte Untermenge der reaktiven Systeme, die ihrerseits eine echte Untermenge der eingebetteten Systeme darstellen (Bild 1.1).

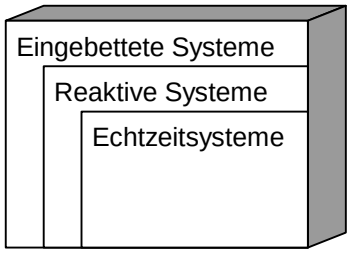

Bild 1.1 Klassifikation eingebetteter Systeme

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. [3] 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 herrschen, 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 [3]:

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

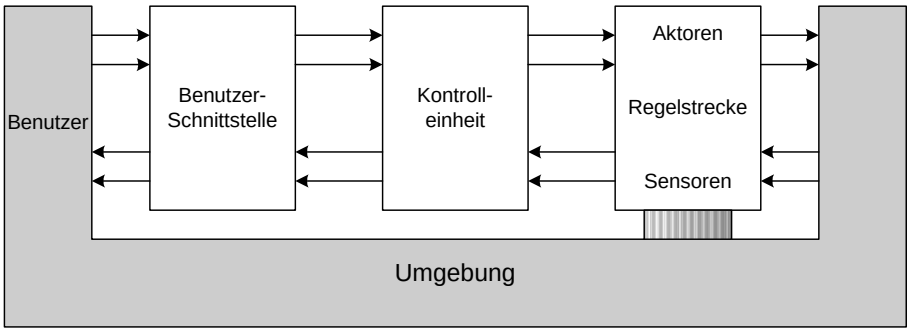

Bild 1.2 Referenzarchitektur eines eingebetteten Systems [3]

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.

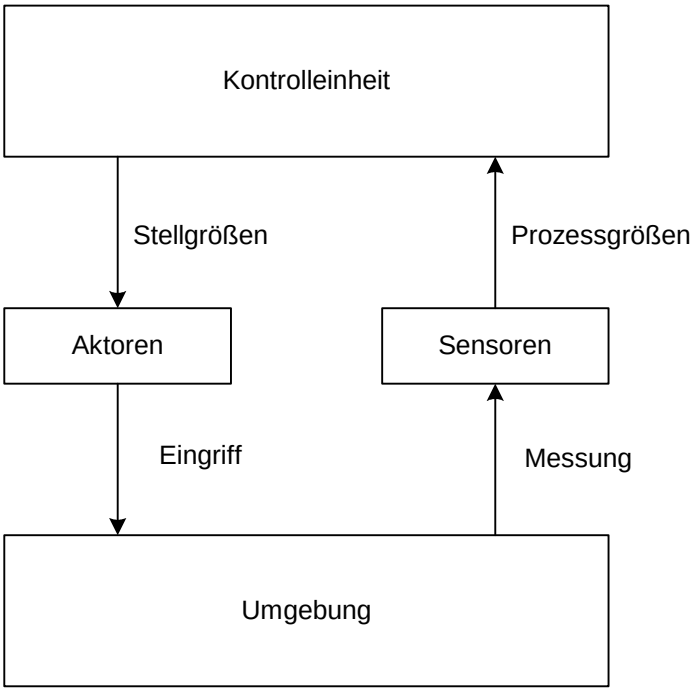

Bild 1.3 Wirkungskette System/Umgebung

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 Viel-

zahl 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 (→ 1.4).

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.

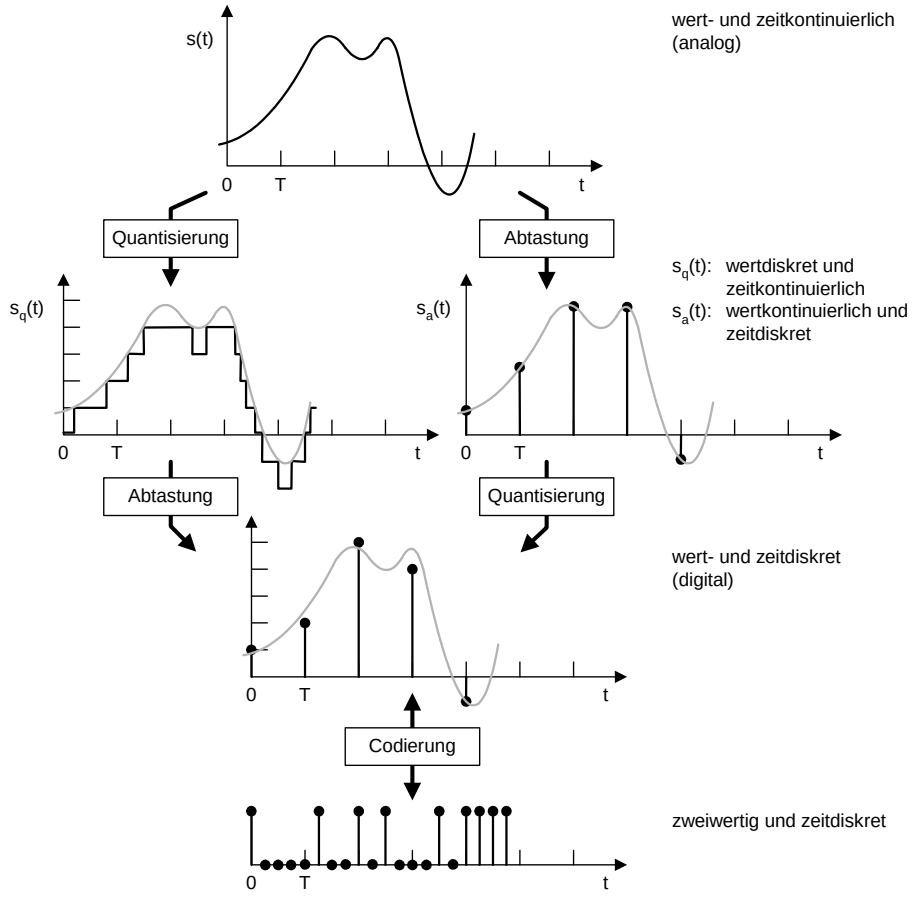

Bild 1.4 Vorgänge bei der AD-Wandlung

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 [3]:

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.

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.

Peripherie: Aktuatoren

Aktuatoren bzw. Aktoren verbinden den informationsverarbeitenden Teil eines eingebetteten Systems und den Prozess. Sie wandeln Energie z.B. in mechanischen 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 Wertelevant.

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.

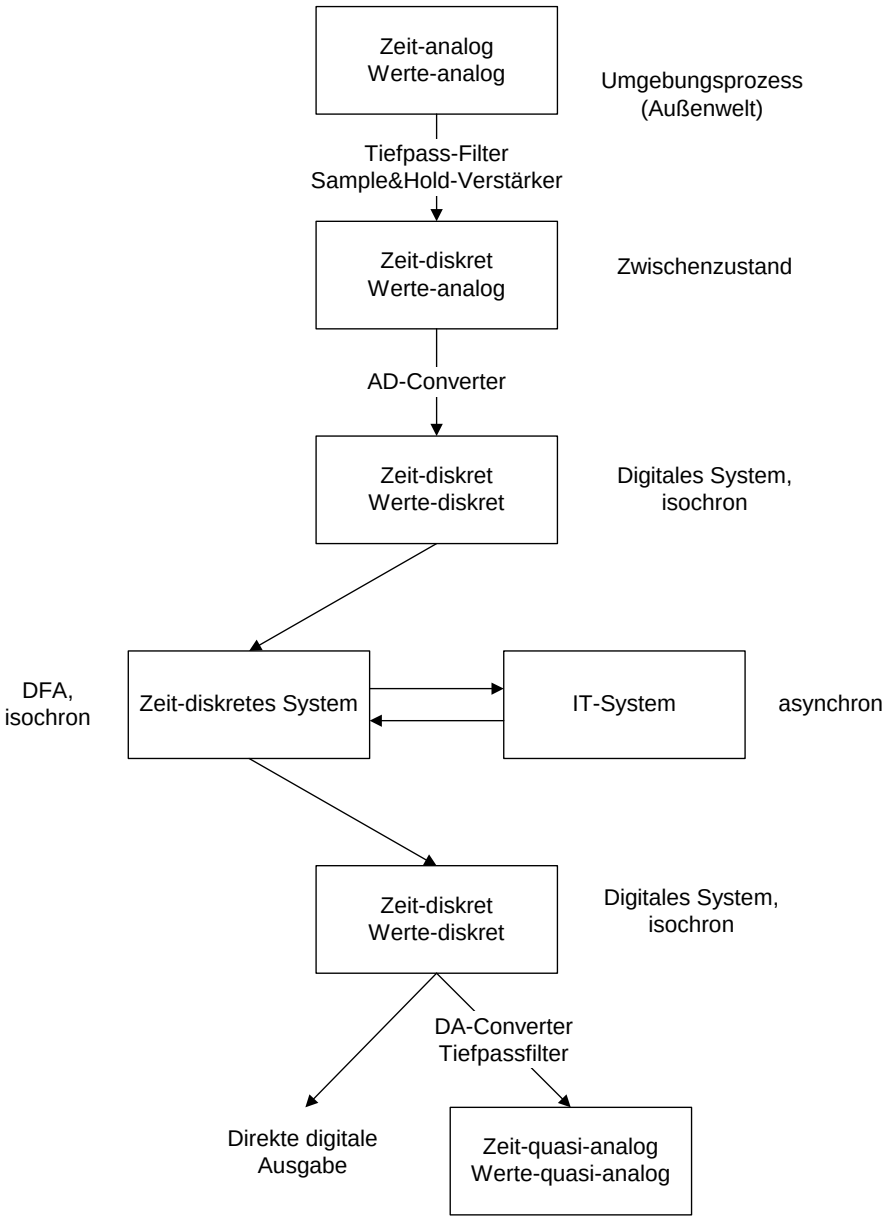

Bild 1.5 Übergänge zwischen den Zeitbindungen

Definition 1.11:

In *Zeit-diskreten Systemen* gilt, dass das System, beschrieben z.B. durch eine Funktion, von abzählbare 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.

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.

Die Zeit-unabhängigen Systeme werden häufig auch als informationstechnische Systeme (IT-Systeme) bezeichnet.

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 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 Anforderung 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.

1.4 Design Space Exploration

Bei all den bisher dargestellten Fakten sollte eines deutlich geworden sein: Ein eingebettetes System besteht im Kern aus einem programmierbaren Rechner (zumindest ist das die Regel), und die Wahl des Rechners sowie dessen Programmierung müssen sich Randbedingungen unterwerfen, z.B. der Erfüllung von zeitlichen Bedingungen.

Die zu erfüllenden Randbedingungen sind zwar formal definierbar, aber fast nie durch einen Compiler in Funktionalität umzusetzen. Zwar könnte man sich gerade dies bei Laufzeiten recht gut vorstellen, aber die Probleme sind sehr groß. Zudem gibt es kaum Programmiersprachen, die die Codierung zeitlicher Bedingungen erlauben. Auf diese Problematik wird in Kapitel 3 näher eingegangen.

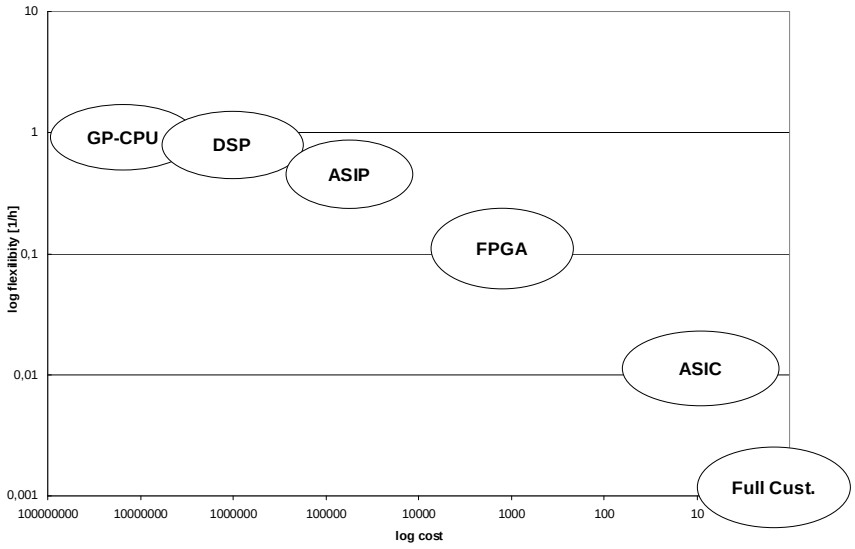

Bild 1.6 Kosten versus Flexibilität für verschiedene Implementierungsbasen

Hier soll etwas anderes dargestellt werden, eine Studie [18], die sehr viel Motivation liefert. Unter der Annahme, man hätte sehr viel Entwicklungszeit, könnten

verschiedene Lösungen für ein Problem erarbeitet und miteinander verglichen werden. Genau dies wurde in der Studie unternommen, und zwar für kommunikationsorientierte Algorithmen.

Bild 1.6 zeigt das wesentliche Ergebnis dieser Studie, die als Implementierungsbasis General-Purpose Prozessoren, Digitale Signalprozessoren (DSP), Applikations-spezifische Instruktionssatz-Prozessoren (ASIP), Field-Programmable Gate Arrays (FPGA, als spezifische Klasse der PLDs), Applikations-spezifische Integrierte Schaltungen (ASIC) und Fully-Customized Integrated Circuits, also vollkundenspezifische Schaltungen nimmt. Die Kosten, die hier als Produkt von Siliziumfläche, Verlustleistung und Ausführungszeit genommen werden, variieren über einen Bereich von 8 Zehnerpotenzen, bei den Werten zur Flexibilität, gemessen am Zeitverbrauch für eine Änderung, immerhin noch um 3 Zehnerpotenzen.

Dies sagt aus, dass die Implementierung die daraus resultierenden Nebenwirkungen in drastischem Maß beeinflusst. Bei programmierbaren Architekturen ergeben sich immerhin noch 4 Zehnerpotenzen. Leider muss man deutlich sagen, dass sich die Entwicklung für FPGAs deutlich von der für Mikrocontroller unterscheidet (typische Programmiersprache: VHDL versus C, Programmiermodell: parallel versus sequentiell). Die Studie ist damit wertvoll, aber sie entbindet noch nicht von der Last der unterschiedlichen Programmentwicklung.

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 [3]:

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*.

Ereignisgesteuerte 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.

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, ein Verfahren, das *Polling* genannt wird.

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.

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 zählendes 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):

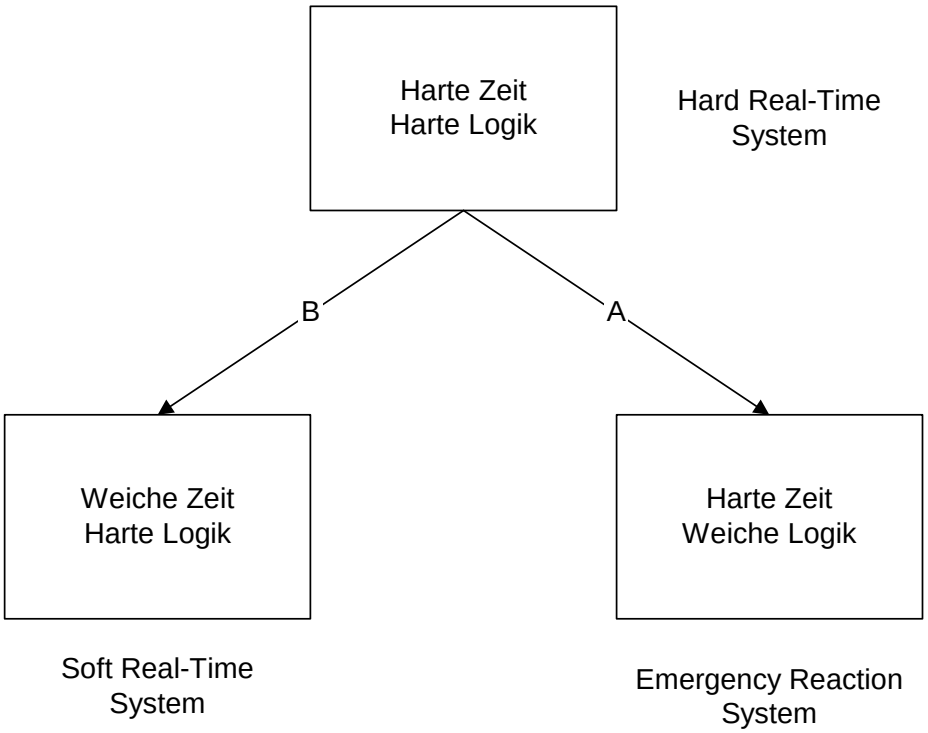

Bild 2.1 Darstellung verschiedener Applikationsklassen

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 [3]. 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 5 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, Multiprozessorcores, 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.

Ein Thread ist also ein Teil eines Prozesses, bestehend aus einem in sich geschlossenen Bearbeitungsstrang und einem recht minimalen eigenen Datenkontext. Letzterer wird benötigt, damit die Threads überhaupt parallel zueinander arbeiten können, und meist beschränkt sich dieser auf den Registersatz (des ausführenden Prozessors).

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.

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 unaufwändiger, 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. Wie dies im Fall eines Multithreading erfolgen kann, ist als Beispiel im Kapitel 5 gezeigt.

Bei diesen 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 5 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 Grundlegende Modelle für die Nebenläufigkeit

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 gegenseitige Gravitation und die der Sonne zu den Bahnen führt, und wird deshalb auch *Newtonsches Modell* genannt. Die 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 Zeit zur Bestimmung eines neuen Zustands im Empfänger kleiner als die Zeitspanne bis zum Eintreffen der nächsten Nachricht. Dies bedeutet, dass sich das gesamte System auf diese Meldungen synchronisieren kann.

3 Design von eingebetteten Systemen

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.

Als erstes wird hierzu 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*.

Im Anschluss daran werden grundsätzliche Lösungsansätze für das Zeitproblem und zur Verlustleistungsminderung besprochen. Den Abschluss dieses Kapitels bildet ein Blick auf mögliche Modellierungssprachen.

3.1 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 3.1 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

$$A \cdot T^k = \text{const(technology)} \text{ mit } k = 1..2 \qquad (3.1)$$

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

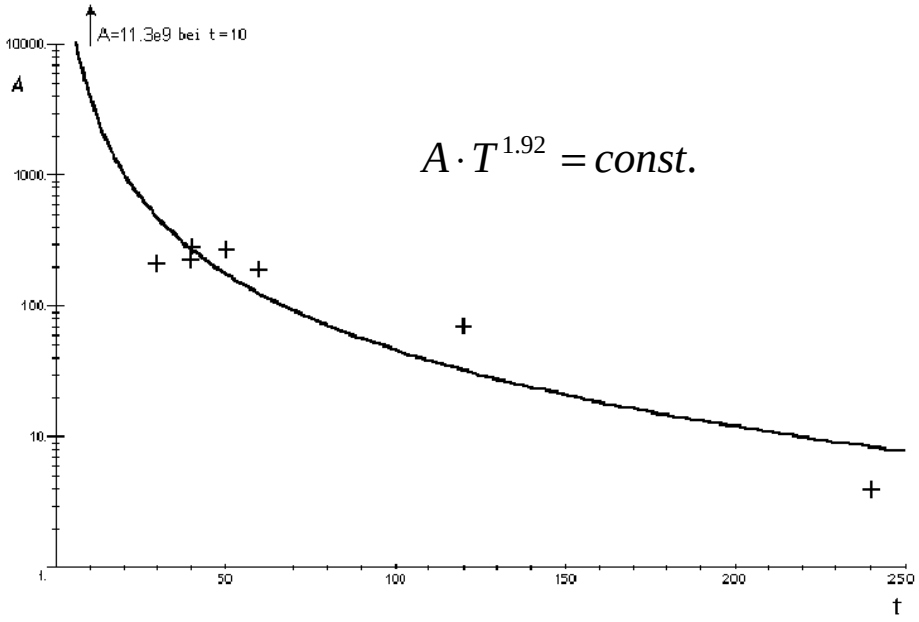

Bild 3.1 AT-Gesetz für 12-Bit-Addierer, verschiedene Implementierungsvarianten

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 3.1) optimal sind.

Definition 3.1:

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:

$$P_{total} = P_{SC} + P_{leakage} + P_{switching\_losses} \tag{3.2}$$

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.

$$P_{SC} = V \cdot I_{SC} \tag{3.3}$$

$P_{leakage}$ (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:

$$P_{leakage} = V \cdot I_{leakage} \tag{3.4}$$

$P_{switching\_losses}$ (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*

$$P_{switching\_losses} = \frac{C}{2} \cdot V^2 \cdot f \tag{3.5}$$

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. (3.5) 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

$$Q = C \cdot V = I \cdot t_{\min} = \frac{I}{f_{\max}} \tag{3.6}$$

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:

$$I = const. \cdot (V - V_{th})^{1,25} \tag{3.7}$$

Die maximal mögliche Frequenz ergibt sich durch Einsetzen von (3.7) in (3.6) und Auflösung nach $f_{\max}$. Hierbei kann eine weitere Näherung für den Fall angenommen werden, dass V von V_{th} weit genug entfernt ist:

$$f_{\max} = const\_1 \cdot \frac{(V - V_{th})^{1,25}}{V} \approx const\_2 \cdot V \quad (\textit{für}\ (V - V_{th}) \geq V_{th}) \tag{3.8}$$

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

$$P \cdot T^3 = const. \tag{3.9}$$

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 3.2 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 (3.5) den einzigen nennenswerten Beitrag zur Verlustleistung liefert, gilt dann zukünftig nicht mehr. Es kann sogar so sein, dass die statische Verlustleistung überwiegt.

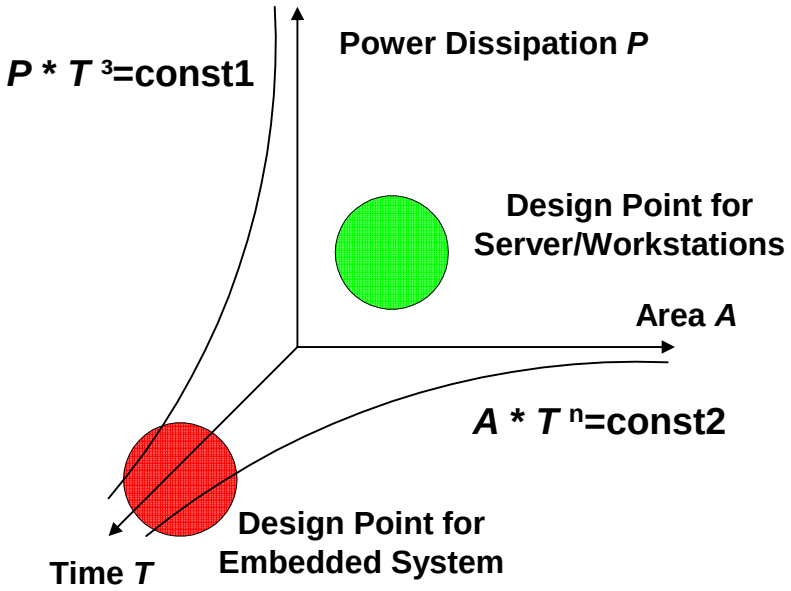

Bild 3.2 Zusammenhang zwischen *P*, *T* und *A*

3.2 Ansätze zur Erfüllung der zeitlichen Randbedingungen

Die im vorangegangenen Abschnitt angegebenen Formeln helfen nur bedingt, denn sie zeigen, welcher Gewinn möglich ist, aber nicht, wie dieser im Design zu erhalten ist. Zudem ist die Rolle der Zeit in den Formeln diejenige der ’Rechenzeit‘, also mit der Performance in Zusammenhang stehend.

Gerade in eingebetteten Systemen ist der entscheidende Zeitbegriff aber 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.

3.2.1 Zeit-gesteuerte Systeme (Time-triggered Systems)

Eine Möglichkeit, den realen Außenbezug im Sinne der Zeit 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 Prozesses müssen zur *Übersetzungszeit* 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 z.B. eine Task, 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 siehe Abschnitt 3.2.3) aufgerufen. Dies ist zwar auch eine Art Ereignis-Steuerung dar, sie ist aber geplant und zyklisch auftretend. In der Interrupt-Service-Routine werden dann aller Prozesszustände abgefragt und entsprechende Reaktionsroutinen aufgerufen.
- Beim kooperativen Systemaufbau ist jede Task verpflichtet, eine Selbst-Unterbrechung nach einer definierten Anzahl von Befehlen einzufügen. Diese Unterbrechung ist als Aufruf eines Schedulers implementiert, dieser ruft dann eine weitere Task 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. Es müssen folgende Ungleichungen gelten:

$$t_{cycle} \leq t_{critical} \tag{3.10}$$

$$\sum t_{task} \leq t_{cycle} \tag{3.11}$$

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.

Ungleichung (3.11) kann auch Tasks enthalten, die mehrfach berücksichtigt werden, weil sie beispielsweise mehrfach in einem Zyklus vorkommen müssen. Für

diese Tasks 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, sie kann ggf. sogar entfallen.

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).

3.2.2 Kombination mehrerer Timer-Interrupts

Grundsätzlich ist es natürlich möglich, mehrere 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. 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 zurückgestellt wird. Dies muss zwangsläufig in jeder Kombination möglich sein, da nichts vorbestimmbar ist.

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 3.3) 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 3.3), 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%).

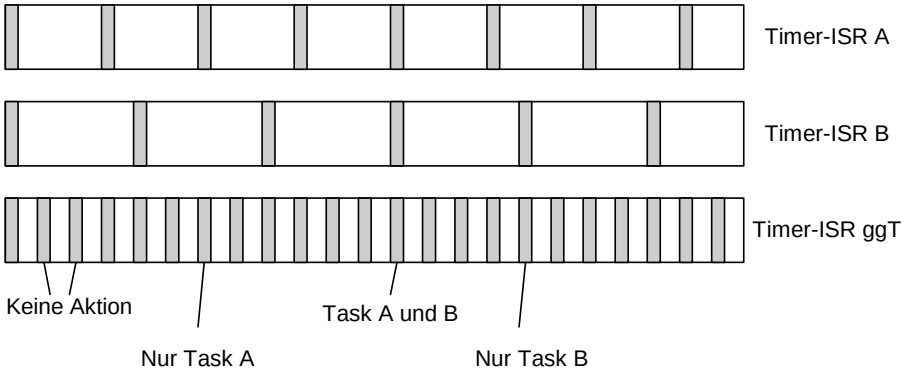

Bild 3.3 Zusammenfügen mehrerer Zeit-gesteuerter ISR zu einer Routine

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 3.3 dargestellt) hat.

3.2.3 Flexible Lösung durch Programmierbare 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 3.4 darstellt. Damit wäre 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. Diese Form der Lösung bleibt also z.B. den rekonfigurierbaren Prozessoren (Mikroprozessor + programmierbare

Logik) bzw. der Zusammenstellung solcher Komponenten auf Boardlevel vorbehalten.

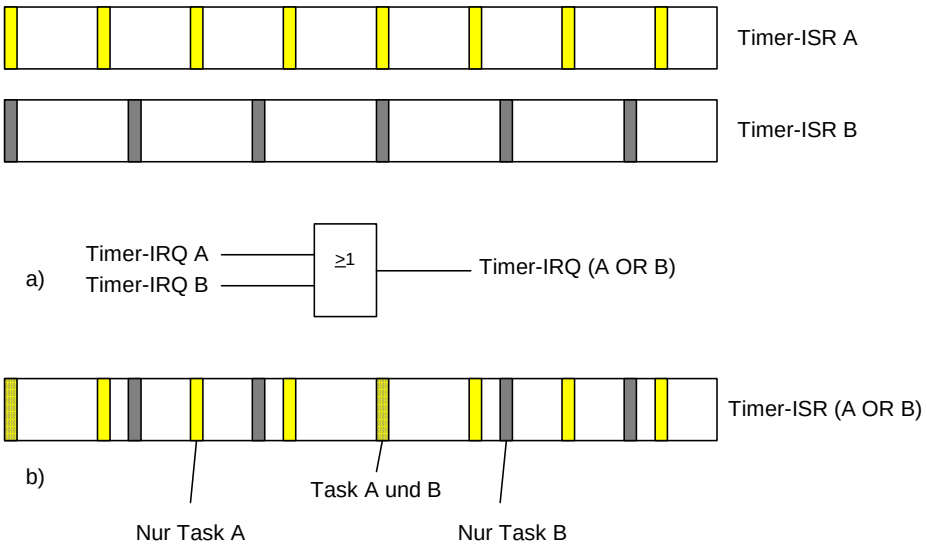

Bild 3.4 Zusammenfassung zweier Unterbrechungsquellen mittels Hardware
a) Verknüpfung der IRQ-Signale b) resultierendes Timingschema

3.2.4 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 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 priorisiert 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 garantiert werden. Nimmt man einmal an, dass

- keine ISR unterbrochen wird,
- jede ISR den IRQ vollständig behandelt,
- für jede ISR eine eigene Priorität (0 … k–1) gegeben wird (0 bedeutet dabei die höchste Priorität),
- für jeden IRQ eine maximale Frequenz des Auftretens und eine maximale Reaktionszeit gegeben ist und
- das Hauptprogramm jederzeit unterbrechbar ist,

dann gilt:

Für IRQ(i) sei F(i) die minimale IRQ-Folgezeit und T(i) die maximal zulässige Antwortzeit, A(i) die Bearbeitungszeit, alle Zeiten ausgedrückt in Prozessortakten bei angenommen 1 Instruktion pro Takt (IPC). A sei diejenige Zeit, die sich als KGV (kleinstes gemeinsames Vielfaches) aller minimalen Folgezeiten F(i) ergibt. Ferner sei f(i) = A/F(i) die maximale Anzahl der Auftritte pro Zeitintervall A. Jetzt müssen die Ungleichungen

$$\sum_{i=0}^{k-1} f(i) \cdot A(i) \le A \tag{3.12}$$

$$\forall n \in \{0,...k-1\}: \sum_{i=0}^{n-1} f(i) \cdot A(i) + \max_{\substack{j=n+1 \\ ...k-1}} (A(j)) + A(n) \le T(n) \tag{3.13}$$

gelten. (3.12) bedeutet dabei, dass die Summe aller im Zeitintervall 1 auftretenden IRQ-Bearbeitungszeiten dieses Intervall nicht überschreiten darf – eine vergleichsweise einfach zu realisierende Forderung. (3.13) 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. Hinzu kommt die eigene Bearbeitungszeit A(n).

Gl. (3.13) ist außerordentlich schwierig im Nachweis, oder sie bedeutet, dass das System planmäßig überdimensioniert werden muss. Insgesamt sind folgende Vor- und Nachteile für diese Form der Systemauslegung aufzuzählen:

+ 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 übrigbleibende 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.

3.2.5 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.

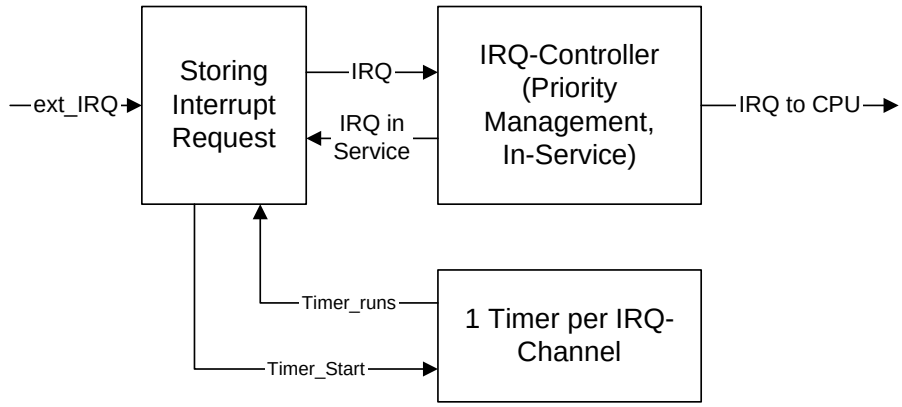

Bild 3.5 Modifizierter IRQ-Controller

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 jeglicher Interrupt nach Auftreten für eine bestimmte Zeit unterdrückt werden. Bild 3.5 zeigt das Blockschaltbild des hypothetischen IRQ-Controllers.

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 deterministischen Ereignisse, die auf diese Weise behandelt werden müssen; Ereignisse mit beliebigen Reaktionszeiten oder ’weichen‘ Behandlungsgrenzen existieren nicht.
– Die variierte Hardware ist derzeit nicht erhältlich.

3.2.6 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 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:

- Alle Ereignisse, die weiterhin zur ersten Kategorie gezählt werden, müssen in der Lage sein, diese Ereignisbehandlung zu unterbrechen, insbesondere höhere Priorität besitzen.
- 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 3.6 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 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.

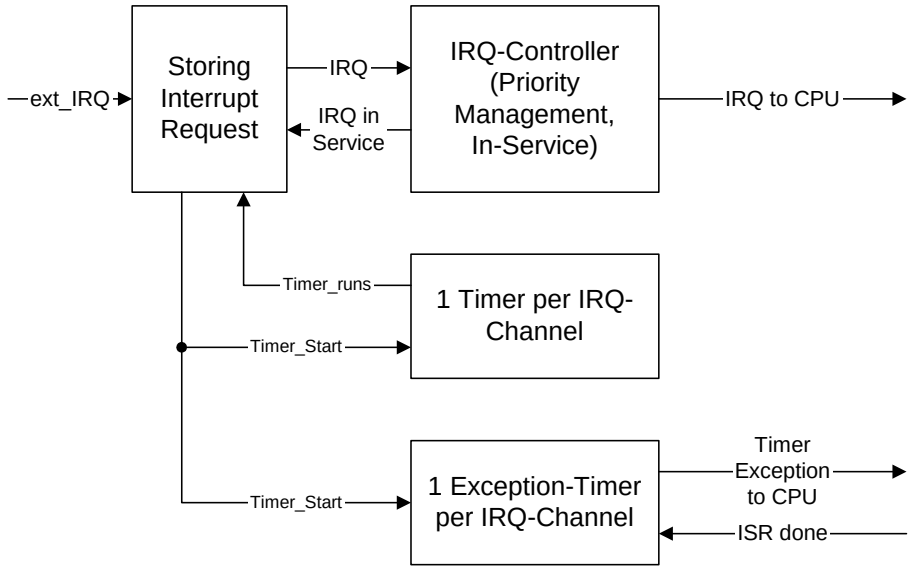

Bild 3.6 Erweiterter IRQ-Controller mit Time Exception

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.
– Die erweiterte Hardware ist derzeit nicht erhältlich.

3.3 Ansätze zur Minderung der Verlustleistung

Wie bereits in Abschnitt 3.1 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 [7], aber noch keinerlei analytische Lösungen. Im Folgenden sollen diese Ansätze kurz diskutiert werden.

3.3.1 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 [9].

Bild 3.7 zeigt einige Mikroprozessoren im Vergleich [9]. 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 3.1 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 3.7 zeigt allerdings nur die Hälfte der Wahrheit, indem kommerzielle Mikroprozessorprodukte miteinander verglichen werden.

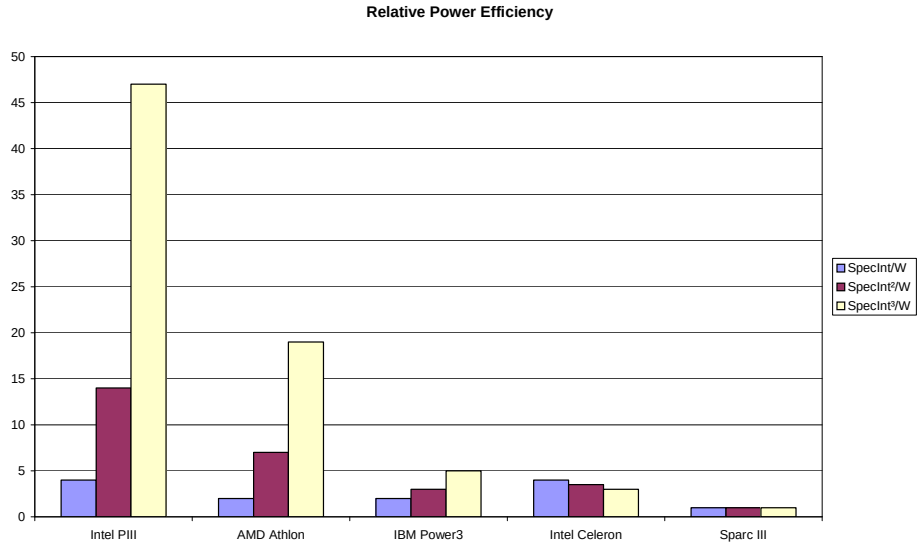

Bild 3.7 Relative Leistungseffizienz im Vergleich

In [8] 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 3.2 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.

3.3.2 Codierung von Programmen in besonders energiesparender Form

Vor einigen Jahren war ein Thema wie energiesparende Software undenkbar, mittlerweile hat es sich jedoch schon etabliert [10]: 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 3.8).

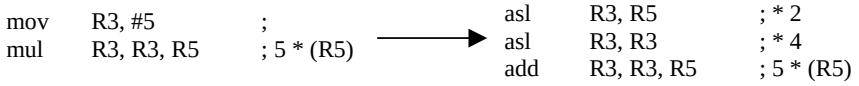

Bild 3.8 Umsetzung einer Multiplikation mit Konstanten in energetisch günstigere Form

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

3.3.3 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 3.9 zeigt ein Beispiel aus [7] für den Intel StrongARM SA-1100 Mikroprozessor.

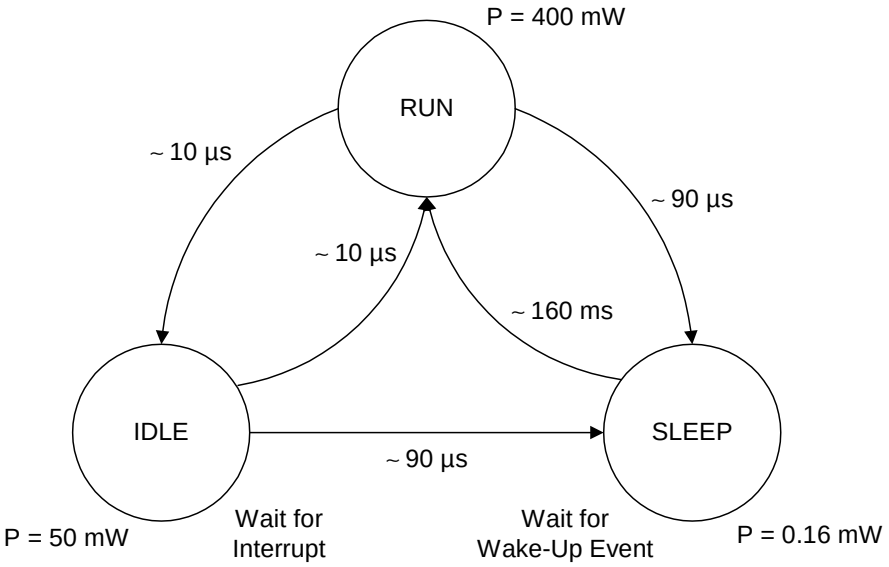

Bild 3.9 Power State Machine für SA-1100

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 3.9 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.

3.3.4 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 [19] wird ein derartiger Ansatz diskutiert, und zwar in einer vergleichsweise feinkörnigen Form.

Die Idee zielt eigentlich auf das Design superskalarer Prozessoren [20]. 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 [19] 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.

3.4 Modellierungs- und Programmiersprachen zur Einbeziehung der Randbedingungen

Bisher wurden einige Lösungswege im Sinn von ’Design Pattern‘ besprochen. Diese bedeuten, dass der/die Designer(in) das Problem in einer ingenieurmäßigen Lösung bearbeitet: Es wird nicht programmiert, es wird designed.

Im Zeitalter der Programmierbarkeit ist es natürlich von Interesse, die Randbedingungen auch innerhalb einer genormten Sprache formulieren zu können und vielleicht sogar in Funktionalität zu übersetzen. Letzteres ist derzeit noch ein Wunschtraum, aber ersteres geht natürlich bereits.

Im Folgenden soll der aktuelle Stand dargestellt werden, und zwar an zwei Beispielen: UML und SystemC. Hierzu werden einige Voraussetzungen definiert und anschließend die beiden Sprachen in Hinblick auf ihre Eigenschaften, Randbedingungen einzubeziehen, diskutiert.

3.4.1 Der Begriff Modellierungssprache

Um mit dem Begriff Modellierungssprache umgehen zu können, muss zwangsläufig zunächst der Begriff des *Modells* geklärt werden. Hierunter kann man landläufig die vereinfachte Abbildung der Realität verstehen. Eine präzisere Definition ist die folgende:

Definition 2.3:

Ein **Modell** ist die idealisierte Repräsentation eines Systems. Die Vollständigkeit und die Detailtreue eines Modells im Sinn einer Realitätsnähe werden durch die zu klärenden und zu untersuchenden Fragen, den Wissensstand und die Modellumgebung bestimmt.

Mit einem derartigen Modell verfolgt man natürlich Ziele, es ist kein Selbstzweck. Wie in der Definition schon dargestellt wurde: Die Realitätsnähe hängt tatsächlich von den zu klärenden Fragen und dem Wissensstand ab!

Bei den Modellen unterscheidet man weiterhin eine wichtige Unterart, die in folgenden Definitionen spezifiziert ist:

Definition 2.4:

Ein **formales Modell (formale Spezifikation)** ist ein Modell (eine Spezifikation) mit einer eindeutigen Interpretation.

Die formale Spezifikation übersetzt eine informelle oder teilformale Spezifikation in ein formales Modell. Informell bedeutet hierbei die Möglichkeit zur mehrdeutigen Interpretation, während sich teilformal auf Modellteile mit formaler sowie andere mit informeller Spezifikation bezieht, wobei das Gesamtmodell hieraus zusammengesetzt ist.

Die eindeutige Interpretation ist es, woran man in der Technik interessiert ist. Diese gewährt eine Möglichkeit zur kompletten Simulation seines Verhaltens. Hierzu wird ein Simulator benötigt:

Definition 2.5:

Ein **Simulator** ist eine Vorrichtung zur wirklichkeitsgetreuen Darstellung bestimmter Bedingungen und Verhältnisse in Auswertung eines Modells.

Die Kunst der Systementwicklung lässt sich sehr weitgehend auf die Kunst der Auswahl des richtigen Modells zurückführen, denn die Modelle haben sehr starken Einfluss auf die spätere Lösung. Als allgemeine Modellierungsprinzipien kann man folgendes festhalten:

- Jedes Modell kann in unterschiedlichen Präzisionsgraden ausgedrückt werden, und die Wahl des Präzisionsgrads muss anhand der darzustellenden und zu erforschenden Details ausgewählt werden.
- Jedes Modell sollte einen Realitätsbezug haben.
- Ein einzelnes Modell ist nie ausreichend, es sollten immer verschiedene Sichten modelliert werden.
- Jedes nichttriviale System wird am Besten durch eine kleine Menge fast unabhängiger Modelle angenähert.

Bei den formalen Modellen kann man dann noch die Teilmenge der *funktionalen Modelle* identifizieren, deren Eigenschaft darin besteht, dass die Beschreibung nicht nur das Modell wiedergibt, sondern durch einen geeigneten Übersetzungsprozess in ein funktionierendes System automatisch überführt werden kann. Dies führt dann zu den sogenannten Programmiersprachen, und derartige funktionale Modelle sind im Rahmen von Rapid Prototyping und Softwareentwicklung für Prozessor-basierte und struktural programmierbare Systeme denkbar.

Bei den allgemeinen formalen Modellen sind natürlich auch Randbedingungen einbeziehbar, die nicht in Funktionalität zu übersetzen sind (leider!). Es können beispielsweise Reaktionszeiten und benötigte Leistung modelliert werden, etwa für Teileinheiten, und daraus können Schlussfolgerungen gezogen werden. Für derartige Modelle gilt dann: Man trifft Annahmen und zieht aus den Ergebnissen, die aus der Modellsimulation entstehen, dann Schlussfolgerungen.

Allgemein kann man sagen, dass Modelle zu folgenden Zwecken entworfen und implementiert werden:

- Visualisierung
- Spezifizierung (Lasten-, Pflichtenheft, z.B. als Executable Specification ESpec)
- Konstruktion (z.B. Programmierung)
- Dokumentation

3.4.2 UML: Unified Modelling Language

UML ist eine Modellierungssprache und bedient sich Diagrammen, um ein System darzustellen. Ein Diagramm kann dabei als spezifische Projektion eines Systems bezeichnet werden, denn es stellt eine jeweilige Sicht dar. Zudem werden nicht Elektronik oder Software dargestellt, sondern *Systeme*.

Derartige softwareintensive Systeme zeichnen sich dabei durch eine Architektur aus. Für Architekturen (im informatorischen Sinn) gelten 5 Sichten: Die statische Entwurfs- oder Prozesssicht, die statische Anwendungsfallsicht, die statische Einsatzsicht, die statische Implementierungssicht und die dynamische Sicht. Es mag dabei überraschen, dass es so viele statische Sichten gibt, aber tatsächlich kann man ein System vielfach beleuchten (und modellieren).

Für die Modellierung der statischen Entwurfs- oder Prozesssicht dienen das *Klassendiagramm* und das *Objektdiagramm*. Hierbei werden entweder die Klassenbeziehungen (objektorientiertes Programmieren) oder die tatsächlich instanziierten Objekte dargestellt. Andere statische Sichten werden durch das *Komponentendiagramm* und das *Einsatzdiagramm* behandelt.

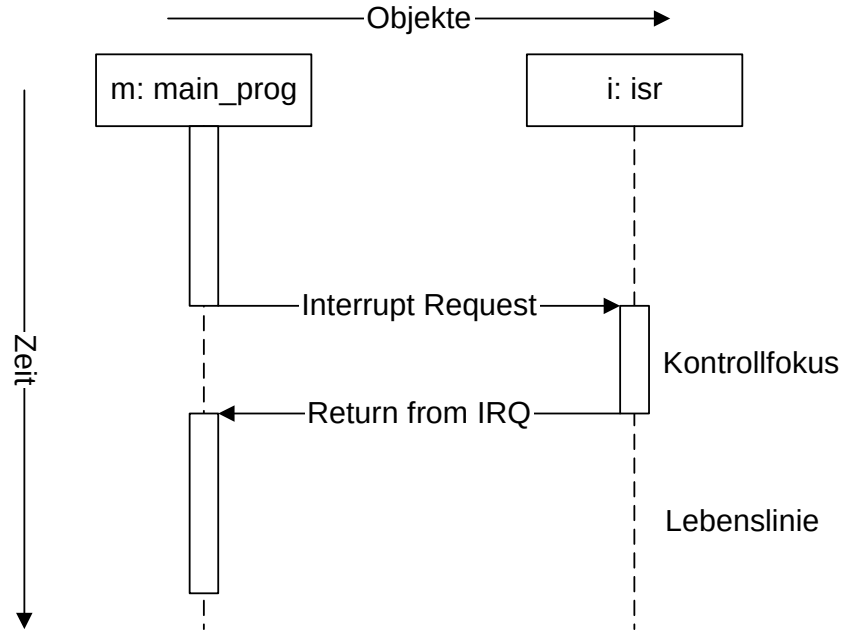

Bild 3.10 Sequenzdiagramm

An dieser Stelle sind natürlich die dynamisch modellierenden Diagramme interessant: *Interaktionsdiagramm*, *Zustandsdiagramm* und *Aktivitätsdiagramm*. Das

nachfolgend dargestellte *Sequenzdiagramm* ist eine spezielle Art des Interaktionsdiagramms. Es stellt die verschiedenen Objekte – hier mit Bezug auf diese Vorlesung je ein Objekt für die Haupt- und eines für die Interrupt-Service-Routine – nebeneinander dar und den Zeitverlauf nach unten. Wichtig ist, dass Ereignisse (synchrone Aufrufe, asynchrone Unterbrechungen) den Kontrollfokus jeweils verlagern können.

Dieses Sequenzdiagramm modelliert ein planbares System, also die Variante der Zeit-gesteuerten Systeme – unabhängig davon, ob die Implementierung nun durch Timer-Interrupt oder durch regelmäßigen Aufruf aus einem Programm heraus erfolgt.

Wie allerdings bereits eingehend diskutiert wurde, sind die planbaren Systeme mit einigen Nachteilen belegt, was u.a. zur Konzeption der modifizierten Ereignisgesteuerten Systeme geführt hat. Dies kann nur modelliert werden, wenn asynchrone Ereignisse (das in Bild 3.10 enthaltene Ereignis ist synchron) zur Verfügung stehen.

Hierfür stellt UML vier verschiedene Ereignisse bereit:

- *Signale*: Dies sind asynchrone Ereignisse, die von einem Ereignis ausgelöst und von dem anderen abgefangen werden. In diese Kategorie gehören die Exceptions, die Ausnahmen.
- *Aufrufe*: Aufrufe stellen synchrone Ereignisse dar, sie bestehen aus dem Aufrufen einer Operation.
- *Verstreichen von Zeit*: Das Verstreichen eines Zeitintervalls bzw. das Überschreiten einer Zeitmarke wird hierunter verstanden. Diese Ereignisse können sowohl synchron als auch asynchron aufgefasst werden.
- *Zustandsänderung*: Ein Änderungsereignis entsteht aus der dauernden Überprüfung von Bedingungen und daraus resultierenden booleschen Bedingungen. Diese Ereignisse sind synchron zum Erzeugerobjekt, jedoch asynchron zum Empfängerobjekt.

Um diese Ereignisse modellieren zu können, werden an den Übergängen zwischen Aktivitätszuständen Bedingungen formuliert. Innerhalb einer Workflow-Modellierung – eine beliebte Art der Modellierung – werden zwischen solchen Aktivitäten Übergänge beschrieben. Diese Übergänge können dabei unbedingt sein, verzweigend, zusammenführend, oder eben mit Bedingungen verknüpft. Bild 3.11 gibt ein kurzes Beispiel dafür.

Es sollte hierbei deutlich sein, dass dieses Aktivitätsdiagramm – das im Übrigen nur Aktivitätszustände und Übergänge enthält – das in 5.4.1 gegebene Beispiel modelliert. Schlussfolgerungen kann man daraus jedoch nur ziehen, wenn für die Signale IRQ_1 bis IRQ_3 ein bestimmtes Verhalten angenommen wird und natürlich die angenommenen Bearbeitungszeiten auch wirklich stimmen.

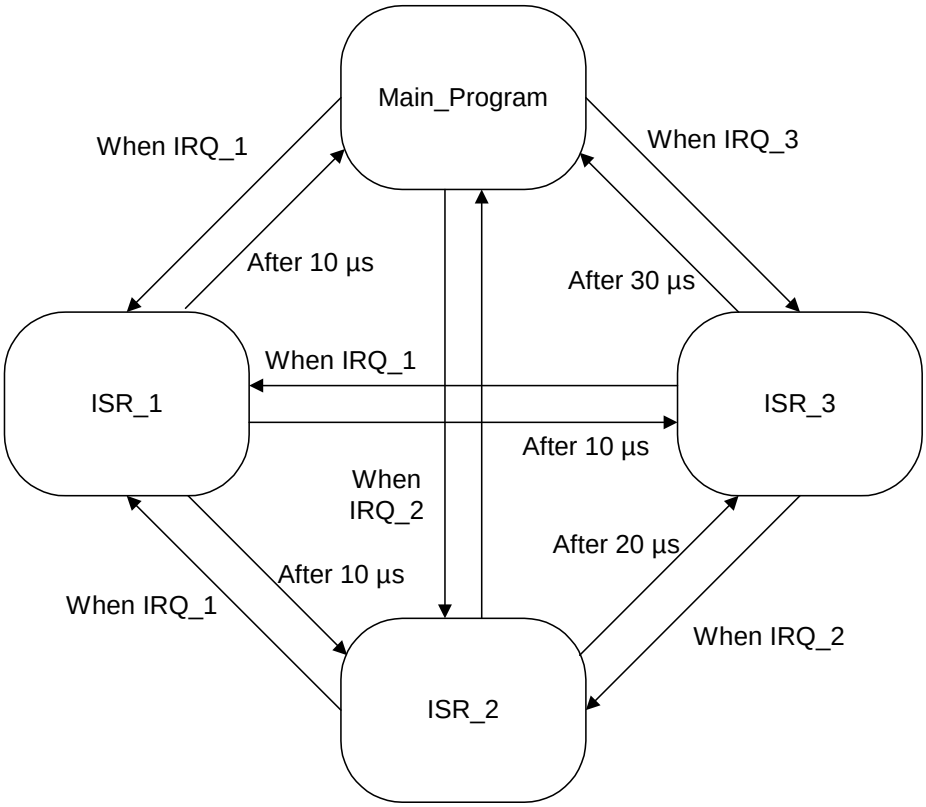

Bild 3.11 Aktivitätsdiagramm für asynchrone Unterbrechungen (Beispiel aus 5.4.1)

3.4.3 SystemC

SystemC wird derzeit als eine Sprache behandelt und entwickelt, die insbesondere zur Beschreibung von Hardware geeignet ist. Tatsächlich stellt SystemC lediglich eine Klassenbibliothek (für C++) und eine Entwicklungsmethodik dar, mit der ein Systementwickler in der Lage ist, *synchrone Software* zu entwickeln.

Gerade die Synchronität der Software ermöglicht eine Synchronisierung auf externe Ereignisse, z.B. einen Takt. Man beschreibt also einen Teil, oft als Thread bezeichnet, in Form von C- oder C++-Funktionen/Methoden, und das Interface nach/von außen ist synchronisiert.

Genau dies entspricht der Hardware-Entwicklung: In sich geschlossene, kleine Teile (= asynchrone Hardware) werden mit Hilfe von Registern (= Synchronisierung) nach außen sichtbar und können so zu großen Schaltungen zusammengebaut werden.

Scheinbar hat dies nichts mit dem Problem der Einbindung von Randbedingungen wie der Zeit in ein Systemdesign zu tun. Die Entwicklung von SystemC geht jedoch aktuell in die Richtung, dass mit V 3.0 ein virtuelles Echtzeit-Betriebssystem integriert werden soll. Damit ist zumindest denkbar, dass Compiler entstehen, die das zeitliche Gefüge eines Softwaresystems ebenfalls beherrschen.

Aktuell allerdings gilt, dass man auch in SystemC nur ein simulierbares Zeitgefüge einbauen kann. Dazu wird einfach angenommen, dass einzelne Laufzeiten bekannt sind, und aus SystemC heraus können Trace- bzw. Timingdiagramme erzeugt werden (z.B. Value Change Dump VCD). Diese Diagramme zeigen dann den zeitlichen Verlauf – in der Simulation.

3.5 Esterel als Beispiel für eine synchrone imperative Sprache

Bei dem Titel dieses Abschnitts empfiehlt sich gleich eine kurze Erläuterung der Begriffe imperativ und synchron:

Definition 3.2:

Eine Programmiersprache wird als *imperativ* bezeichnet, wenn ihre algorithmische Beschreibung auf der vollspezifizierten und vollständig kontrollierten Manipulation benannter Daten in einer schrittweisen Art basiert. Dies bedeutet, dass in der Programmcodierung alle Daten als Variablen benannt und komplett kontrollierbar sind. Dieses Paradigma entspricht am besten der menschlichen Denkweise.

Bekannte Vertreter für das imperative Programmierparadigma sind C und ADA.

Dies bedeutet, dass jede Datenmanipulation explizit beschrieben wird, auch in ihrem Weg, und die Datenzuweisungen sind sequenziell – beides aus der ”C“-Welt bekannte Tatsachen (mit allen Vor- und Nachteilen).

Das (perfekt) synchrone Modell für eine Sprache war bereits in Kapitel 2 erwähnt worden:

Definition 3.3: [3]

Eine Sprache wird als *synchron* bezeichnet, wenn sie das *perfekt synchrone Kommunikationsprinzip* verwendet. Hierbei handelt es sich um eine Hypothese, dass Kommunikation keine Zeit verbraucht. Das System arbeitet somit perfekt synchron zueinander.

Die Annahme, keine Zeit zu verbrauchen, ist in der Realität natürlich nicht einzuhalten. In Wahrheit ist dies die Annahme, dass das System zur Verarbeitung weniger Zeit benötigt als die Zeitspanne zwischen zwei aufeinanderfolgenden Signalen, die verarbeitet werden müssen [3]. Zur Kommunikation wird also nicht nur die eigentliche Übertragungszeit, sondern auch die Verarbeitungszeit gerechnet.

Esterel beschreibt nun das ”Zusammenleben“ von parallelen Prozessen. Zur Historie: Esterel ist eigentlich ein Höhenzug in Frankreich (bei Cannes), und der Name erinnerte die Entwickler an das französische Wort für Echtzeit (”temps réel). Esterel wurde in den 80er Jahren in Frankreich formal und praktisch entwickelt und ist nunmehr ein Produkt, dass industriell genutzt wird [21], z.B. in der Avionik- und Automobilindustrie.

Im Folgenden werden Eigenschaften und Ansätze dieser Sprache beschrieben, es wird jedoch keine Einführung in die Programmierung, Syntax etc. gegeben. Diese ist u.a. in [21] zum Download bereit.

3.5.1 Lösungsansätze zur Modellierung der Zeitspanne zwischen Ein- und Ausgabe

Grundsätzlich hat man verschiedene Möglichkeiten, Annahmen über das Zeitverhalten eines Programms zu machen. Wie bei den zeitgesteuerten Systemen (→ 3.2.1) bereits gezeigt wurde, ist es auch möglich, dem System ein Zeitverhalten einzuprägen. Esterel hingegen ist eine ereignis-orientierte Sprache, die für reaktive Systeme konzipiert und entwickelt wurde.

Damit muss das System auch so beschreibbar sein, dass es Ereignis-gesteuert arbeitet. Nun ist (oder wird) es Standard (sein), das System zunächst zu beschreiben, auf Konsistenz zu prüfen und anschließend erst zu implementieren. Dies bedeutet, dass Annahmen über die Zeitspanne zur Reaktion gemacht werden müssen, und hier bieten sich 4 Möglichkeiten an [3]:

1. Für jeden Zeitschritt wird die exakte Zeit angegeben, die für die Implementierung notwendig sein wird. Dies zwingt zu sehr frühzeitigen und aufwendigen Schätzungen, ein Vorgang, von dem gerade abstrahiert werden soll.
2. Jeder Systemreaktion wird eine konstante, endliche Zeitdauer zugeschrieben. Dies bedeutet, dass eine obere Grenze für die Reaktionszeit dar, auch für die Implementierung, die weder über- noch unterschritten werden darf. Somit könnte es auch passieren, dass Vorgänge künstlich verzögert werden müssen.

   Ein weiteres Problem ergibt sich, wenn man in der Modellierung eine Reaktion in eine Folge von Subreaktionen aufteilen will. In diesem Fall nimmt die modellierte Zeit linear zu, was sicher nicht der Realität entspricht.
3. Die nächste Alternative besteht darin, individuelle Zeiten für jede Reaktion anzunehmen. Dies ist sicher sehr flexibel und auch abstrakt, jedoch ist der Freiraum letztendlich so groß, dass kaum gesicherte Aussagen daraus gewonnen werden können.
4. Jeder Systemreaktion wird eine Zeitdauer von 0 Zeiteinheiten zugewiesen, und damit wird angenommen, dass die Systemreaktion stets schneller ist als die Rate der Ereignisse. Dass diese Annahme korrekt ist, dafür muss in einer späteren Phase der Entwicklung ein Nachweis geführt werden. Dies führt dann zur eingangs schon erwähnten perfekten Synchronie. Insbesondere werden hier-

durch keine künstlichen Verzögerungen notwendig, etwa, um die angenommene Zeit auch wirklich zu erreichen.

Esterel basiert auf der Hypothese, dass Signalaustausch und elementare Berechnungen keine Zeit benötigen und somit Systemausgaben mit ihren Eingaben synchron ablaufen. Hierdurch wird in Esterel ein nebenläufiges, deterministisches Verhalten möglich.

3.5.2 Determinismus

Ein Programm in Esterel verarbeitet Ströme von Ereignissen (events). Diese Ereignisse dienen zur internen und externen Kommunikation, und ein Ereignis kann aus mehreren Elementarereignissen (z.B. Signalen) bestehen, die ihrerseits nicht unterbrechbar sind. Die interne Verarbeitung ist schnell genug, dass während der Berechnungszeit keine weitere Eingabe eintritt. Dieses Berechnungsintervall wird *Moment* oder *Augenblick* (*instant*) genannt und ist (in der Annahme) unendlich kurz (*instantaneous*).

Nun ist Esterel so definiert, dass hiermit deterministische reaktive Systeme beschrieben werden sollen. Für ein einzelnes Programm gilt dann, dass dieses garantiert deterministisch ist, aber bei der Parallelkomposition mehrerer Programme kann es dennoch zu Nichtdeterminismus kommen.

Wie kann man sich das vorstellen? Angenommen, es existiert ein oder mehrere Ereignisse, auf die mehrere Esterel-Programme reagieren, und zwar mit Ausgaben in das gleiche Signal. Dann ist der Wert dieses Signals abhängig von der Reihenfolge, in der die Programme ausgeführt wurden (tatsächlich werden diese ja sequenziell hintereinander ausgeführt), und damit ist das Ergebnis nicht voraussagbar.

In Esterel ist es so gelöst, dass Algorithmen vor einer Übersetzung in reale Programme testen, ob es zu Nichtdeterminismus kommen kann.

3.5.3 Eigenschaften von Esterel

Das Grundprinzip der Esterel-Programme besteht darin, Zustandsmaschinen (Automaten) zu beschreiben, die bedingt durch externe Events einen Zustandswechsel durchführen und dabei neue Ereignisse aussenden. Diese Events sind nicht gespeichert vorliegend, sondern sind nach Bearbeitung wieder ”verschwunden“ (im Gegensatz zur Hardware-Implementierung bei endlichen Automaten).

Parallelität

Angenommen, es sind P1 und P2 zwei Esterel-Programme, dann ist auch P1 || P2 ein Esterel-Programm mit folgenden Eigenschaften:

- Alle Eingaben, die von der Systemseite empfangen werden, sind gleichermaßen für P1 und P2 verfügbar.

- Alle Ausgaben, die von P1 (bzw. P2) generiert werden, sind im gleichen Augenblick für P2 (bzw. P1) sichtbar (Prinzip der perfekten Synchronie).
- P1 und P2 werden nebenläufig ausgeführt, und ihre Parallelkomposition P1 || P2 terminiert, wenn beide (P1 und P2) terminieren.
- P1 und P2 teilen sich keine gemeinsamen Variablen.

Zwischen zwei verschiedenen Esterel-Programmen existieren keine gemeinsamen Variablen, und andere Konstrukte für den gezielten Austausch von Daten sind auch nicht zu verzeichnen. Aus diesem Grund muss die Kommunikation anders gereglt sein, hier durch ein *Broadcasting*: Jeder übertragene Wert steht sofort allen Empfängern zur Verfügung, ein gezieltes Senden an einen oder bestimmte Empfänger ist nicht möglich.

Deklarationen

Esterel ist keine vollständige Programmiersprache, sondern dient der Beschreibung der Prozesskontrolle. Die benötigten Datentypen, Konstanten, Signale, Funktionen etc. werden in einer Hostsprache (C oder Ada) implementiert und in Esterel importiert. Dementsprechend müssen diese Bestandteile auch deklariert werden.

Eine Besonderheit nehmen Signale ein: Sie dienen der Kommunikation per Broadcasting und können als Input, Output oder InputOutput (bidirektional) deklariert werden. Ferner besitzt jedes Signal zu jedem Zeitpunkt einen eindeutigen Status, entweder ”present“ oder ”absent“. Dieser Zustand wird nicht gespeichert und muss daher immer neu abgefragt werden.

Instruktionen

Im Instruktionsteil eines Esterel-Programms werden Anweisungen durch folgende Konstrukte aufgebaut:

- Deklarationen von lokalen Signalen und Variablen
- elementare Anweisungen wie Zuweisungen, Prozeduraufrufe
- Verzweigungen, Schleifen
- Eingaben, Ausgaben und Testen von Signalen
- zeitliche Anweisungen
- Ausnahmebehandlung

3.5.4 Kausalitätsprobleme

Die Prüfung eines Esterel-Programms auf Reaktivität und Determinismus ist keineswegs trivial, da es z.B. zu Verklemmungen aufgrund der gegenseitigen Kommunikation kommen kann. Beispielsweise können zwei Programme gegenseitig auf ein Signal des jeweils anderen warten, um fortzufahren.

Die Prüfung der logischen Korrektheit wird also auf die Prüfung der Signale und ihrer Beziehungen abgebildet. Signale haben einen von zwei Zuständen, wobei ”absent“ als Defaultstatus definiert ist. Im *Kohärenz-Prinzip* wird nun definiert, wann ein Signal ”present“ ist:

Definition 3.4 [3]:

Ein Ausgangssignal bzw. ein lokales Signal *x* ist zu einem Zeitpunkt präsent genau dann, wenn zu diesem Zeitpunkt die Anweisung ”emit x“ im Sichtbarkeitsbereich von *x* ausgeführt wird.

Nun kann es eben Zyklen geben, bei denen emit x genau dann ausgegeben wird, wenn x bereits präsent ist (so ist es natürlich trivial, in mehrfacher Form jedoch nicht):

```
signal x in
       present x then emit x end
end signal
```

Lässt man nun solche Zyklen zu, droht ein Deadlock. Lässt man andererseits überhaupt keine Zyklen zu, wäre dies eine zu starke Einschränkung. Also wird richtigerweise das Programm aufwendig geprüft (NP-vollständiges Problem). In der so genannten ”konstruktiven Semantik“ wird diese Prüfung vereinfacht.

4 Softwarequalität

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.

4.1 Beispiele, Begriffe und Definitionen

4.1.1 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.

4.1.2 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 4.1 [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 werden. 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 4.2:

Failure (*Fehlverhalten, Fehlerwirkung, äußerer Fehler***)**: Hierbei handelt es sich um ein Fehlverhalten eines Programms, das während seiner Ausführung auch wirklich auftritt.

Definition 4.3:

Fault (*Fehler, Fehlerzustand, innerer Fehler*): Es handelt sich um eine fehlerhafte Stelle eines Programms, die ein Fehlverhalten auslösen kann.

Definition 4.4:

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 4.1). 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.

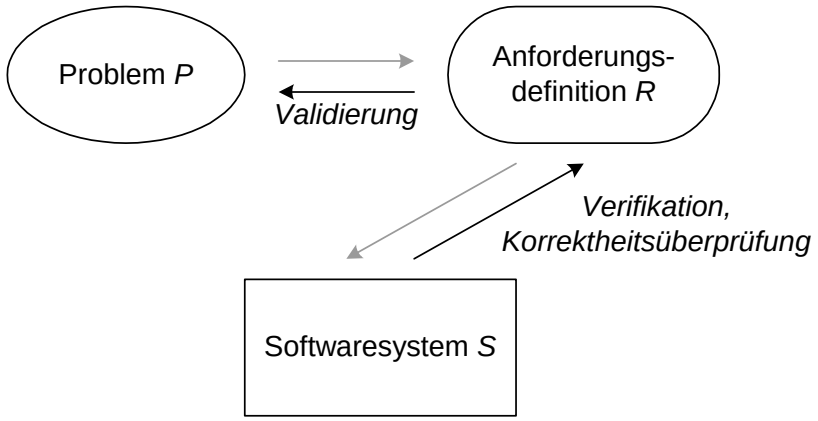

Bild 4.1 Einordnung der Begriffe Validierung und Verifikation

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 (→ 2.6, 5.1).

Die formale Verifikation ist damit nur ein Bestandteil der Maßnahmen zur Erhöhung der Softwarequalität, der weitaus größere besteht in dem Testen.

4.2 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 4.5:

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 4.6:

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 4.7:

Ein *Systemausfall* (*failure*) liegt vor, wenn ein System sein geforderte Funktion nicht mehr erfüllt.

Definition 4.8:

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 (→ 4.3).

4.2.1 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.

4.2.1.1 Einsatz redundanter Hardware

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.

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.

Um solche Fälle auszuschließen müssen konstruktive Maßnahmen ergriffen werden, die dann verschiedene Hardwarekomponenten miteinander verbinden.

4.2.1.2 Einsatz redundanter Software

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). Diese 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.

4.2.2 Analytische Maßnahmen

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.

4.2.3 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 4.9:

Als **Gefahr** (hazard) wird eine Sachlage, Situation oder Systemzustand bezeichnet, in der/dem eine Schädigung der Umgebung (Umwelt, Maschine, Mensch) möglich ist.

Ein 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 4.10:

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 (FTO, *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.

4.2.4 Software-Review und statische Codechecker

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 [24] 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 (→ 4.4) einen sehr positiven Effekt auf die Softwarequalität haben

4.2.5 Testen (allgemein)

In der Praxis steuert alles auf das Testen hin, dies erscheint als die ultimative Lösung. Ein gute Einführung in dieses überaus komplexe Thema ist in [25] – [29] 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 [25]:

- Modellierung der Software-Umgebung
- Erstellen von Testfällen
- Ausführen und Evaluieren der Tests
- Messen des Testfortschritts

4.2.5.1 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.

4.2.5.2 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 (→ 4.2.7)

4.2.5.3 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.

4.2.5.4 Messen des Testfortschritts

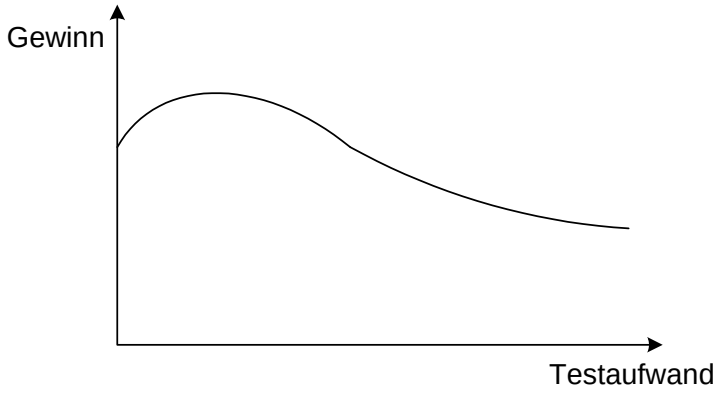

Bild 4.2 Gewinn versus Testaufwand

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 4.2).

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.

4.2.6 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.

Für einen 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 Ergebnisse 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.

4.2.7 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.

4.2.7.1 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.

4.2.7.2 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.

4.2.7.3 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.

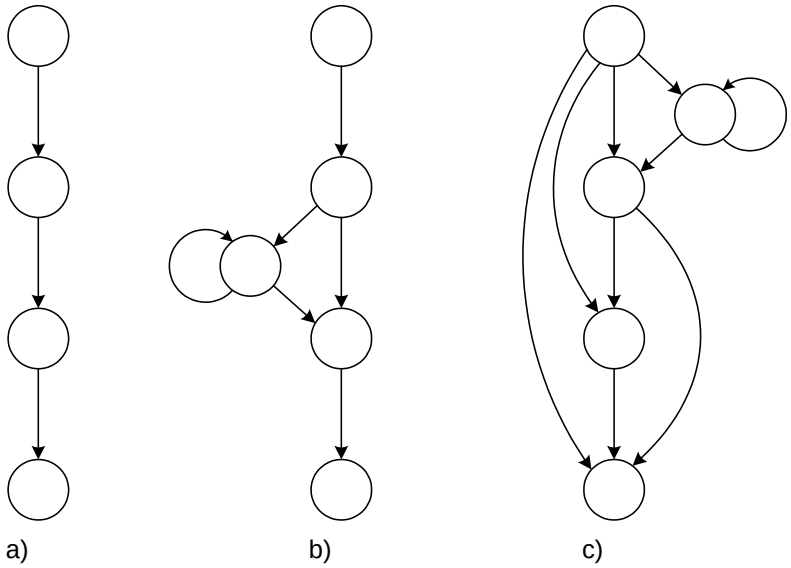

Bild 4.3 Kontrollflussgraphen mit den zyklomatischen Komplexitäten a) 1 b) 3 c) 6

Ausgangspunkte ist dabei ein Kontrollflussgraph des Programms (Bild 4.3). Hierin werden die voneinander unabhängigen Programmpfade bestimmt, dies ergibt die so genannte zyklomatische Komplexität. Es gilt hier die Formel

$$CC = E - N + 2$$

$$mit\ E = Anzahl\ der\ Kanten, N = Anzahl\ der\ Knoten$$

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 ausschließlich 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.

4.2.8 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.

4.3 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, voraussichtlich 2006 gültig) und DIN EN 61508 (Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, Oktober 2005). Diese beiden sind eng aufeinander bezogen und verweisen gegenseitig. Tabelle 4.1 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_r–ISO 13849-1	SIL_r–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 4.1 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.

4.4 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 [23] 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.

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.

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

for( k = 0, m = 0; k < 10; k++, m++ )
{
         if( 0 == array[m] )
             k = 0;
}
```

Beispiel 4.1 Implizit unbegrenzte `for`-Schleife (als Negativbeispiel)

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.

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 Datenobjekten 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) [24]. Über die Funktionalität dieser Programme wurde in Kapitel 4.2 bereits berichtet.

5 Design-Pattern für Echtzeitsysteme, basierend auf Mikrocontroller

5.1 Dynamischer 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 Tasks gelegt werden.

5.1.1 Klassifizierung der Teilaufgaben

Das hier dargestellte Designverfahren beruht darauf, die einzelnen Teilaufgaben (diese werden hier immer als Task 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 Tasks*: Fester Bestandteil dieser Teilaufgaben sind exakte Zeitabstände, in denen diese Tasks 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 Tasks*: Das Starten bzw. Wecken einer Task mit 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 Tasks 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 Tasks in dem System, die zwar keine scharfen Zeitbedingungen enthalten, im Ganzen jedoch

Zeitschranken einhalten müssen. Hiermit sind Tasks 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.

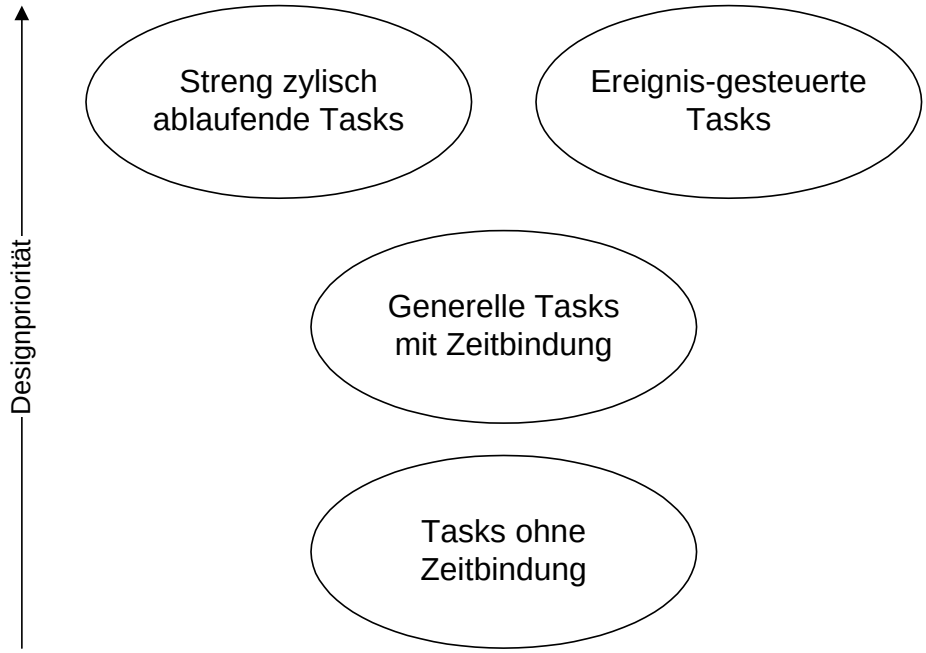

Bild 5.1 Taskklassen und Designprioritäten

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 eine Task, die überhaupt keine Zeitbindung besitzt – dies dürfte in der Praxis selten vorkommen – natürlich in die Klasse der generellen Tasks mit Zeitbindung sortiert werden. Diese Taskklasse ist in Bild 5.1 dargestellt, wurde jedoch nicht in die Klassifizierung aufgenommen, da sie irrelevant für das hier dargestellte Designprinzip ist.

Streng zyklisch ablaufende Tasks und Ereignis-gesteuerte Tasks sind in ihrer Designpriorität in etwa gleichzusetzen (→ Bild 5.1). In der Praxis kann die Implementierung auch sehr ähnlich sein, indem die zyklischen Tasks in Interrupt-Service-Routinen (ISR) mit Timersteuerung und die Ereignis-gesteuerten Tasks in

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

5.1.2 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 Tasks unbedingt verzichtet werden, denn dies kann zu großen Problemen bei der Bestimmung der WCET bis hin zur Unmöglichkeit führen. Diese Forderung führt zu einem sicheren Design, da sich Abhängigkeiten etwa in der Form, dass, falls Task 1 den maximalen Pfad durchläuft, Task 2 garantiert einen kleineren Pfad als seinen maximalen wählt, nur positiv auf die WCET des Gesamtsystems auswirken können.

Bild 5.2 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 Tasks, Timer-ISRs und ggf. Event-ISRs, beinhaltet die Organisation der Kommunikation zwischen den einzelnen Teilen sowie die Abstimmung des Zeitverhaltens. Als Kommunikation zwischen diesen Tasks ist ein nicht-blockierendes Semaphoren/Mailbox-System ideal: Semaphore, die seitens einer Task 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: Tasks 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.

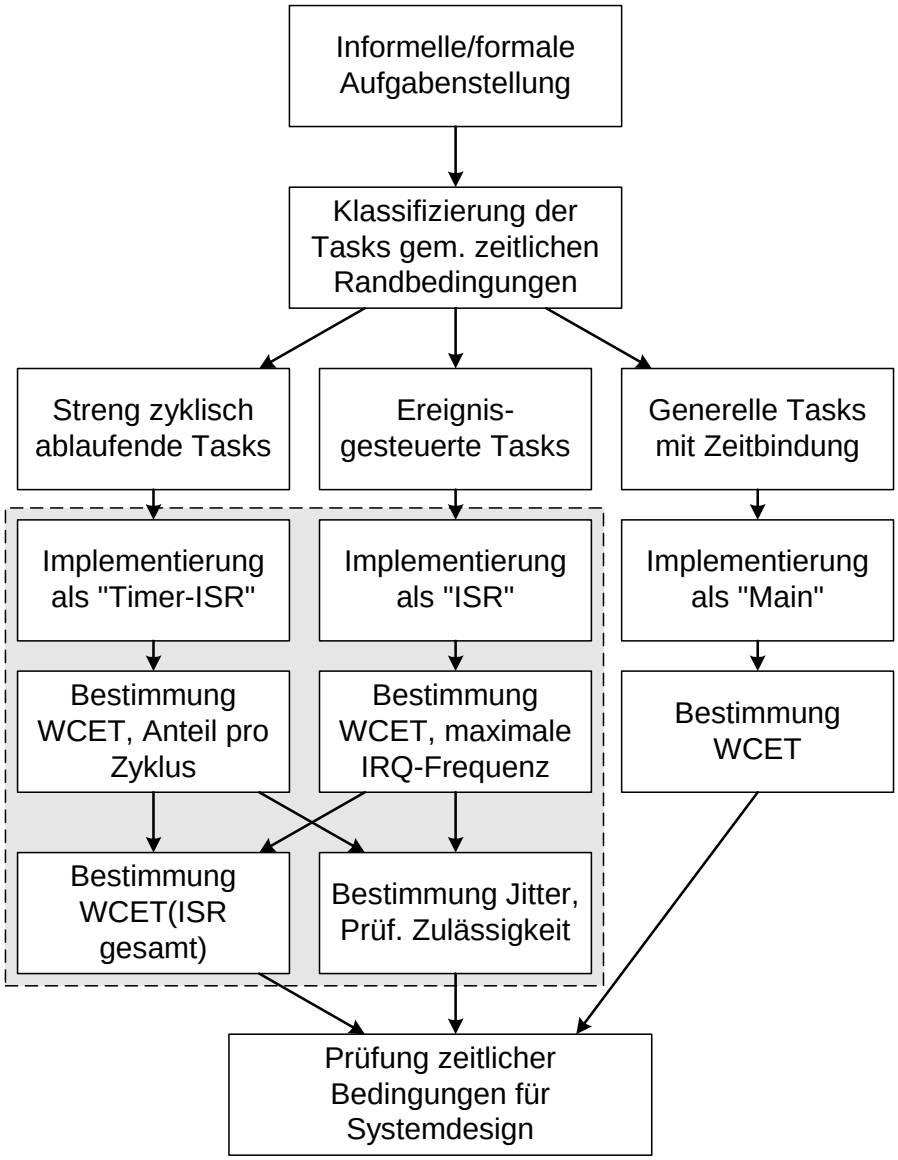

Bild 5.2 Gesamtablauf Systemdesign

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.

```
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 */
  globalMesswert = ADC_OUT;
  semaMess = 1;

  ...
}
```

b)

Bild 5.3 Nicht-blockierende Kommunikation zwischen Main- (a) und Interrupt-Routine (b)

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

- Wie beeinflussen ununterbrechbare Teile in der generellen Task bzw. einer ISR die Latenzzeiten der Interrupts? Die Beantwortung dieser Frage ist insbesondere für die zyklischen Tasks 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.

5.2 Komplett statischer Ansatz durch Mischung der Tasks

Ein in [16] dargestellter Ansatz verzichtet sowohl auf ein Scheduling durch ein Betriebssystem (→ Kapitel 6) 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.

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.

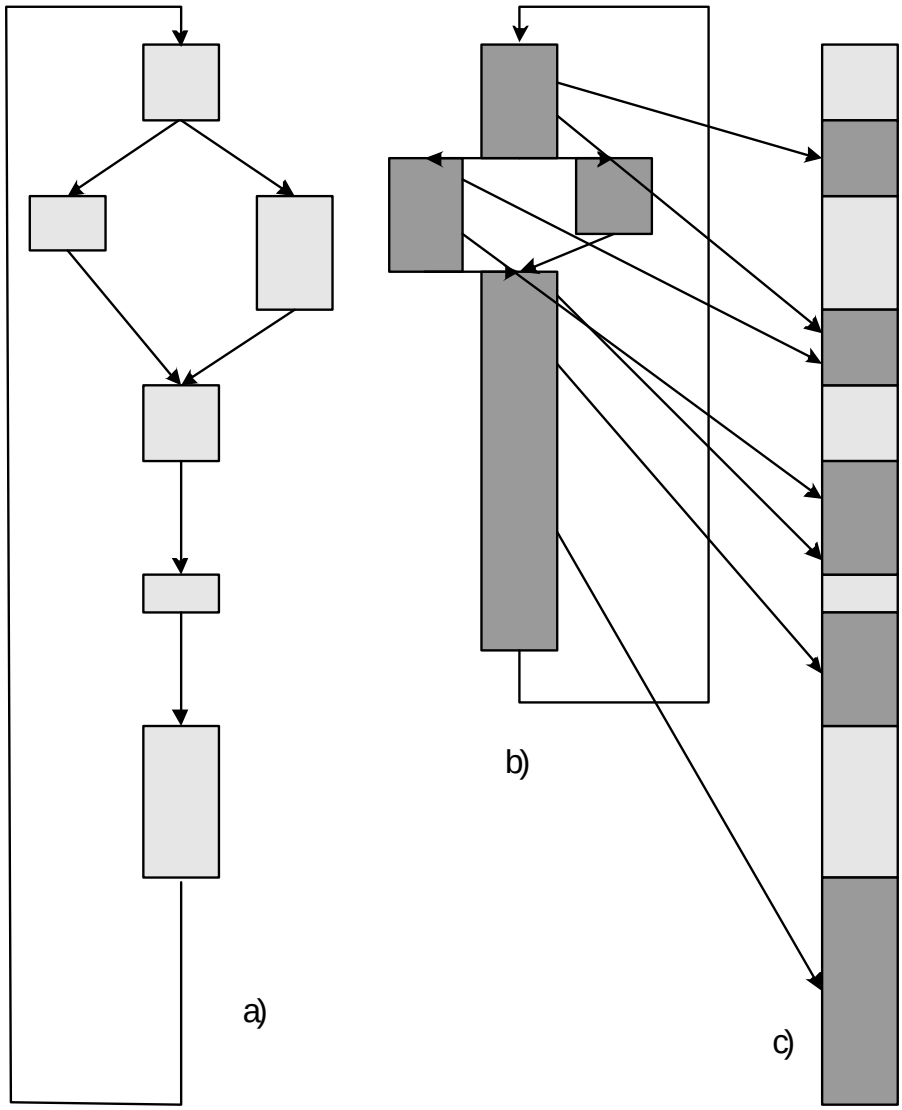

Bild 5.4 Mischung zweier Threads zwecks Software Thread Integration
a) Primärthread, zeitkritisch b) Sekundärthread, zeitunkritisch c) Thread Integration

Folgerichtig bemüht sich der Autor in [16] 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 5.4) 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 [16] 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.

5.3 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 5.5 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 5.5 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 das 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:

```
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 5.5 a) C-Sourcecode für ISR zur AD-Konvertierung mit Grenzwertvergleich

```
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 5.5 b) Assemblerübersetzung

- 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.

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 gerät. Wie Bild 5.6 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.

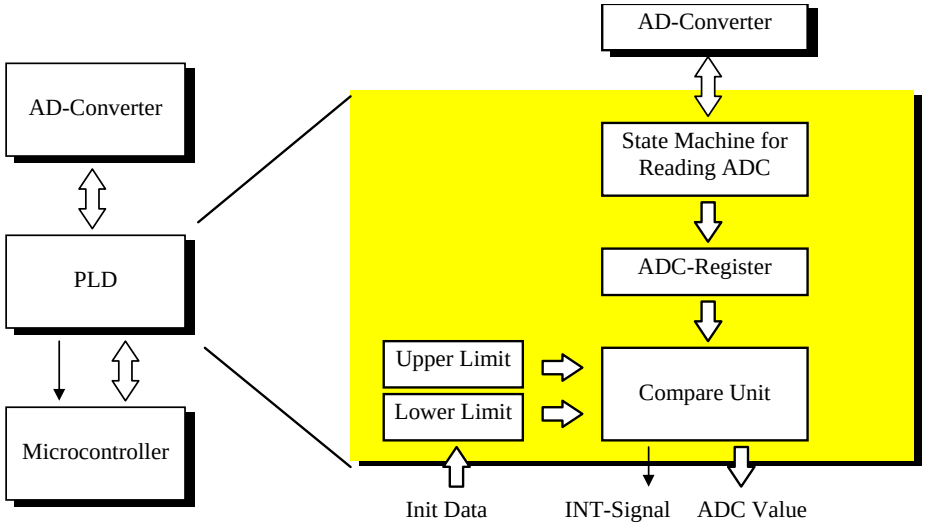

Bild 5.6 Implementierung der AD-ISR in PLD

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.

5.4 Zusammenfassung der Zeitkriterien für lokale Systeme

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 T_{Jitter} (auch als maximale Latenzzeit zu bezeichnen, siehe Definition 2.6) 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.

10 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}	< 10 Befehlszeiten	> ∑ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)
T_{Service}	> 50 % der gesamten Rechenzeit	< 1 % der gesamten Rechenzeit
T_{Reaction}	< 100 Befehlszeiten	> ∑ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)

Tabelle 5.1 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.

Tabelle 5.1 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 5.7).

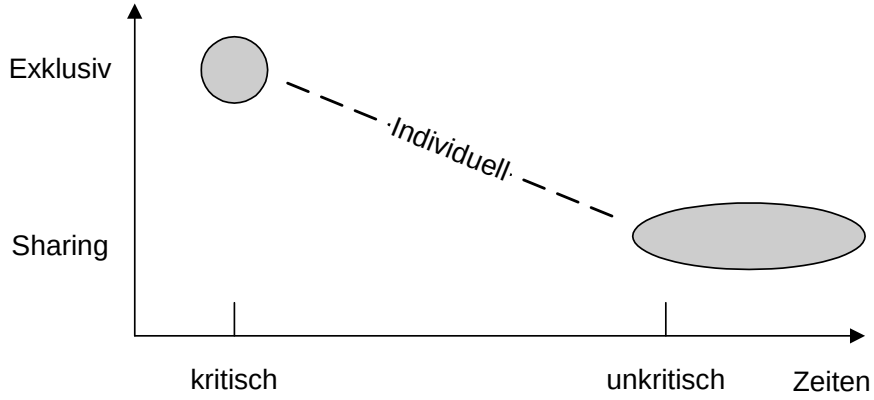

Bild 5.7 Designraum exklusiv/sharing für Systeme mit zeitlichen Randbedingungen

5.4.1 Vergleich Zeit-Steuerung und modifizierte Ereignis-Steuerung

Aus den Überlegungen zu den charakteristischen Zeiten kann man weiterhin eine Unterscheidung für Zeit-gesteuerte und modifizierte Ereignis-gesteuerte Systeme ableiten. Beide sind in der Lage, deterministische Echtzeitbedingungen zu erfüllen.

Der erste Vorteil für die modifizierten Ereignis-gesteuerten Systeme besteht darin, dass die Abfrage der Inputleitungen, die bei der Zeit-Steuerung im Pollingverfahren erfolgen muss, durch die Ereignissignale entfällt. Selbst wenn der Kontextwechsel in eine Interrupt Service Routine einen vergleichbaren Aufwand zum Polling darstellt, ist der Gewinn an Rechenzeit genau dann vorhanden, wenn die Ereignisse nicht regelmäßig kommen. Wie an einem kleinen Modell gezeigt werden wird, ist dieser Gewinn vergleichsweise gering.

Für den maximal zulässigen Jitter gilt, dass dieser nicht überschritten werden darf. Dies bedeutet, dass im Zeit-gesteuerten System entweder die Zykluszeit hiernach auszurichten ist, oder für die kritischen Teile muss eine Behandlung mehrfach innerhalb eines Zeitzyklus durchlaufen werden. Lösung 1 führt dazu, das System für alle Teile mit einer Zykluszeit zu versehen, die nicht überall benötigt wird, Lösung 2 erhöht die Pollingrate nur individuell, bedeutet aber auch mehr Programmaufwand intern.

Die Kombination der Einsparung von Rechenzeiten bei ereignislosen Abschnitten und der individuellen Anpassung der Latenzzeiten ergibt die wesentlichen Vorteile für modifizierte Ereignis-gesteuerte Systeme. Um dies zu quantifizieren, sei hier ein Modell angegeben:

- Es wird ein RISC-basiertes Mikroprozessorsystem gewählt, bei dem – vereinfachend – 1 Befehl/Takt als Geschwindigkeit angenommen wird.
- Jeder IRQ wird durch einen IRQ-Controller priorisiert, und der Sprung in die Interrupt Service Routine ISR wird mit 10 Takten veranschlagt (in der Reaktionszeit enthalten). Jeder höherpriorisierte IRQ (0: höchste Priorität) kann niedrigere ISR unterbrechen, das Hauptprogramm ist jederzeit unterbrechbar.
- Es werden 4 Zustände des Prozesses angenommen: Priorität 0 benötigt 100 Takte zur Bearbeitung, Priorität 1 200, Priorität 2 300 und Priorität 3 400 Takte (alles Maximalwerte). Für das Zeit-gesteuerte System sollen diese Zustände nacheinander abgefragt und bearbeitet werden, wobei die Abfrage 10 Takte beanspruchen soll.
- Für die Häufigkeit der IRQs wird angenommen, dass sie alle mit maximaler Frequenz von 10 kHz auftreten können. Zur Simulation wird eine Variation angenommen, also beim ersten Mal nach 100 µs, beim zweiten Mal nach 200 µs, dann nach 300 µs, wieder nach 10 µs usw.
- Als maximal zulässige Reaktionszeiten werden für Priorität 0 20 µs, für 1 50 µs und für die anderen 100 µs angenommen.
- Für das Verhältnis von ISR und Hauptroutinenrechenzeit wird ca. 1:1 angenommen, d.h., der Zyklus soll auf 2000 Takte ausgelegt werden. Für das Hauptprogramm wird gefordert, dass im zeitlichen Mittel ca. $2 * 10^6$ Instruktionen pro Sekunde ausführbar sind.

Für dieses Modell ergeben sich dann folgende, in Tabelle 5.2 wiedergegebene Schätzungen. Die darin vorhandenen Ergebnisse können so zusammengefasst werden:

- Das Verhältnis der Taktanzahlen, die dem Hauptprogramm zur Verfügung stehen, ist nahezu konstant, d.h., der Overhead für die Zeitsteuerung ist vergleichsweise gering. Es verbessert sich zwar noch weiterhin, wenn die IRQs noch sporadischer auftreten, dennoch dürfte der Effekt auf wenige Prozent begrenzt bleiben.
- Der Unterschied in der Auslegung der Betriebsfrequenz ist sehr groß. Die Ursache hierfür liegt in der geforderten maximalen Reaktionszeit. Die notwendige Frequenz kann für die Zeitsteuerung dadurch heruntergesetzt werden, dass die extrem kritischen Teile mehrfach vorkommen oder die Zykluszeit verringert wird. Letztere Maßnahme ist begrenzt (die Berechnungszeit der Prozess-gekoppelten Software setzt das Limit), ein häufigerer Timer-IRQ erzeugt mehr Overhead (Klammerwerte in Tabelle 5.2: Für Priorität 0 werden zwei Timer-IRQ pro 2000 Takte Zyklus erzeugt).

	Time-triggered	Modified Event-triggered
Anzahl Takte für Prozess-gekoppelte Software (auf 12000 Takte)	3300 (3420)	3120
Anzahl Takte Hauptprogramm (auf 12000 Takte)	8700 (8580)	8880
Relativer Gewinn	1	1.02 (1.035)
Maximale Latenzzeit [Takte]	0: 2000 (1000) 1: 2100 2: 2300 3: 2600	0: 0 1: 100 2: 300 3: 600
Mittlere Latenzzeit [Takte]	0: 1000 (500) 1: 1050 2: 1150 3: 1300	0: 0 1: 2,5 2: 22,6 3: 90
Maximale Reaktionszeit [Takte]	0: 2100 (1100) 1: 2300 2: 2600 3: 3000	0: 100 1: 300 2: 600 3: 1000
Resultierende Taktfrequenz [MHz]	105 (55) MHz (Reaktionszeit Priorität 0)	10 MHz (Reaktionszeit Priorität 3)

Tabelle 5.2 Taktzahlen und Operationsfrequenz im Modellsystem (Zahlen in Klammern: Erweitertes Timer-IRQ-System für Priorität 0 mit zwei Serviceroutinen pro Zyklus)

Als Fazit dieses Vergleichs bleibt an dieser Stelle festzuhalten, dass die (modifizierten) Ereignis-gesteuerten Systeme insbesondere Forderungen nach kurzen Reaktionszeiten wesentlich besser erfüllen können. Die Dimensionierung des Zeit-gesteuerten Systems ist in dem Modell gerade deshalb so hoch, weil die Reaktionszeit der höchsten Priorität zwar weit von der für die Befehlsbearbeitung entfernt ist, jedoch die Zykluszeit dieser Größe angepasst werden muss.

Eine Schätzung des Effekts durch Einführung von ’Modified Event-triggered with Exception Handling‘ kann für das Modell ebenfalls gegeben werden. Verringert man die Arbeitsfrequenz beispielsweise auf 8 MHz, so kann für aller Prioritäten die Echtzeitbedingung eingehalten werden, lediglich für Priorität 3 ist dies nicht immer möglich. Hier wird nun im Ausnahmefall (drohende Zeitüberschreitung) eine Notroutine angesprungen, die eine vorläufige Reaktion darstellt.

Das Down-Scaling in diesem Fall führt zu Einsparungen von ca. 20%. Dies ist im Einzelfall zu überprüfen und stellt lediglich eine erste Schätzung dar.

5.4.2 Übertragung der Ergebnisse auf verteilte Systeme

Das Wesen der verteilten Systeme – die Einbindung und der Zugriff auf ein nicht-exklusives 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:

- CSMA/CD (Carrier Sense Media Access with Collision Detection): Dieses bei Ethernet verwendete Verfahren scheidet aus, da der Zugriff probabilistisch ist und somit keine maximale Übertragungszeit garantiert werden kann.
- CSMA/CA (Carrier Sense Media Access with Collision Avoidance): 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/CA-Netzwerk echtzeitfähig werden. Dadurch wäre ein Ereignis-gesteuertes Netzwerk tatsächlich möglich!
- TTP/C (Time-Triggered Protocol Class C): In diesem Zeit-gesteuerten Protokoll 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!).
- 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 in einem Zeitschlitz wird nach dem CSMA/CA-Verfahren verteilt, sodass der Bus optimal ausgenutzt wird und zugleich (für eine begrenzte Anzahl von Daten) echtzeitfähig ist.

5.4.3 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, [17] [22]), 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*. 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).

6 Betriebssysteme als virtuelle Maschinen

Bei der Einführung der ersten Computer gingen die Gründerväter davon aus, dass man eine feste Hardware haben müsste, auf der dann eine Software laufen würde. Ein einfaches Prinzip lag den Rechnern zugrunde, sie waren in der Hardware so universell, dass jedes Programm ausführbar war (und ist), und die Software legte die Funktionalität fest.

Offenbar haben wir uns von diesem einfachen Standpunkt schon weit entfernt, denn jeder größere Rechner und in zunehmendem Maße auch die eingebetteten Systeme besitzen zwischen Hard- und Software noch ein Betriebssystem. Allein die Definition eines Betriebssystems zeigt, was dort eigentlich hinzugekommen ist: ”Ein Betriebssystem (Operating System) stellt das Bindeglied zwischen der Hardware eines Computers einerseits und dem Anwender bzw. seinen Programmen andererseits dar. Es umfasst Programme, die zusammen mit den Eigenschaften des Computers die Grundlage der möglichen Betriebsarten dieses Systems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen“ [9, Abschnitt 7.1].

Offenbar hat das Betriebssystem eine Art Oberhoheit, insbesondere über die Zeit. Gerade diese Größe wurde ja in den vorangegangenen Abschnitten als besonders kritisch dargestellt, Grund genug, sich Aufgaben, Architekturen und Angebot am Markt – Betriebssysteme betreffend – einmal näher anzuschauen.

6.1 Betriebssystem als Teil der Systemsoftware

Ein Betriebssystem zählt in jedem Fall zur Systemsoftware, zu der alle Programme, die eine effiziente und komfortable Nutzung eines Computers ermöglichen, zusammengefasst sind [9, Abschnitt 7.2]. Hierzu zählen z.B. auch Programmierumgebungen (Compiler, Linker, Debugger) und weitere Dienstprogramme, die häufig nicht vom Betriebssystem zu trennen sind, da sie dessen Schnittstellen nutzen. Mit einer großen Berechtigung kann das Betriebssystem damit als virtuelle Maschine aufgefasst werden, die dem Nutzer das native Interface der Hardware (Befehlssatz, Registermodell) sowie erweiterte Funktionen zur Verfügung stellt (Bild 6.1).

Welche Aufgaben sind konkret in der virtuellen Maschine verankert? Hierzu zählen die Erweiterung der Funktionalitäten, die Organisation, Steuerung und Kontrolle des gesamten Betriebsablaufs, die Verwaltung der Betriebsmittel und weitere Aufgaben, z.B. Protokollierung oder Nutzerverwaltung. Bild 6.2 zeigt die wesentlichen Säulen, die zusammen ein Betriebssystem bilden, und darin wird für Echtzeit- oder Real-Time-Betriebssysteme die Verwaltung der Zeit natürlich eine besondere Rolle spielen.

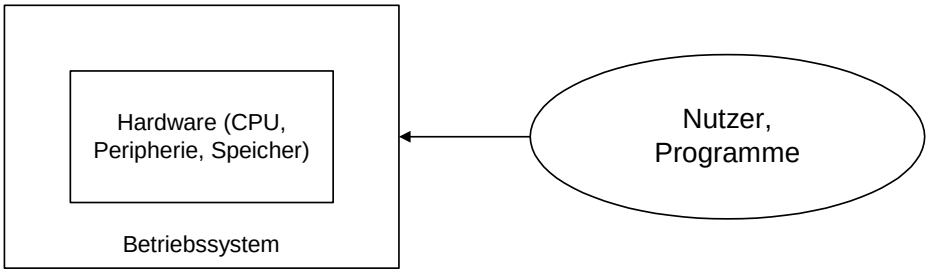

Bild 6.1 Betriebssystemsicht

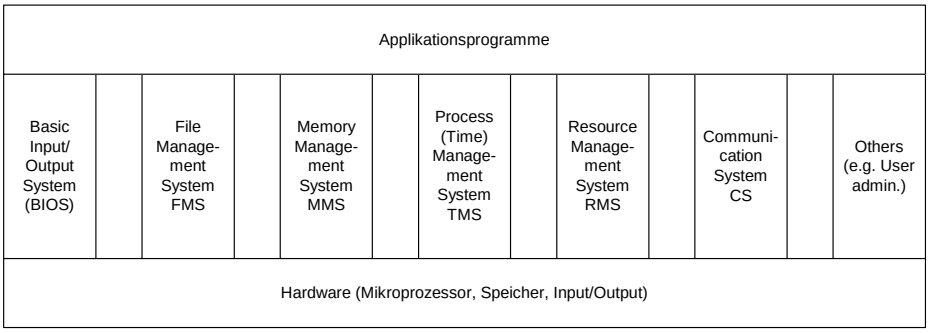

Bild 6.2 Betriebssystemaufgaben

Das BIOS und das File Management System, auch als Disk Operating System (DOS) bezeichnet, sind natürlich gute alte Bekannte. Sie bieten Methoden, um am Rechnersystem einfach auf den Input/Output-Bereich sowie den Massenspeicher zuzugreifen und zählen damit klassisch zu der Erweiterung der Funktionalität. Gleiches gilt für das Communication System CS, das aber erst in den letzten Jahren verstärkt aufgekommen ist. Hier muss klar festgehalten werden: In Zukunft wird es kaum noch Betriebssysteme im Embedded-Bereich geben, die keine Kommunikation mehr beinhalten.

Memory Management System MMS und Resource Management System RMS gehören zur Verwaltung der Betriebsmittel. Hierzu zählt auch die Ausgabe an Bildschirmen, häufig als Graphical User Interface GUI bezeichnet. Eigentlich könnte man auch das Process Management System TMS, das die Ressource Rechenzeit (Time) verwaltet, hier hinzuzählen, aber dieser Teil hat eine besondere Stellung, denn hier wird der Gesamtablauf kontrolliert.

6.2 Betriebssystemarchitekturen

Für ein einfaches Betriebssystem, das keine Kontrollfunktion im Sinne des TMS enthält, reicht es ggf. aus, eine Funktionssammlung anzubieten. Das Betriebssystem wird dadurch ausschließlich von Applikationen via API (Application Programmering Interface) aufgerufen. Dies reicht im Allgemeinen jedoch nicht mehr für Multiprocessing-fähige Systeme aus, denn in diesem Fall benötigt das Betriebssystem selbst Rechenzeit und eine entsprechende Architektur.

Die einfachste Architektur ist monolithisch aufgebaut (Bild 6.3a). Diese Architektur dürfte zwar die schnellste Variante sein, andere Forderungen wie Wartbarkeit, Modularisierung und Flexibilität sind so jedoch kaum zu erfüllen. Ein ebenfalls klassischer Aufbau ist der eines Schichtenmodells, bei dem exakt definierte Funktionen sowie Interfaces pro Schicht vorhanden sind. Dieses Modell (Bild 6.3b) ist vergleichbar mit dem ISO/OSI-Layermodell für Netzwerke, und hier wie dort offenbaren sich Nachteile, die sich in Rechenzeitbedarf oder Durchgriff durch die Schichten (quasi-Konsistenz, Treppenschichtenmodell) zeigen.

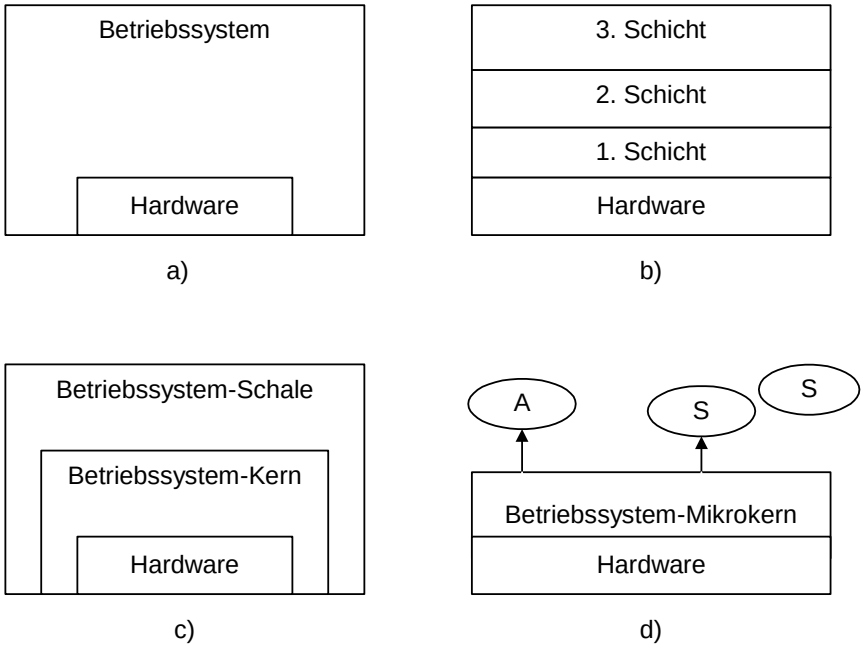

Bild 6.3 Betriebssystemarchitekturen
a) monolithischer Aufbau b) Schichtenmodell
c) Kern-Schalen-Modell d) Mikrokern-Modell

Modern sind Architekturen nach Bild 6.3c und 6.3d. Im Betriebssystemkern (kernel) sind alle wichtigen Funktionen, oftmals im privilegierten Prozessormodus ablaufend, vereint, ergänzende Funktionen, insbesondere ein API, bietet die Schale (shell) an. Die Schale ist damit prinzipiell austauschbar und kann den jeweiligen Gegebenheiten angepasst werden.

Noch modularer und austauschbarer ist der Mikrokernansatz (Microkernel, Bild 6.3d). Dieser bietet nur noch einen minimalen Funktionsumfang als Infrastruktur, z.B. Inter-Prozess-Kommunikation Inter-Process Communication, IPC), Prozessverwaltung, Speicherverwaltung, Scheduling und hardwarenahe E/A-Funktionen. Die eigentlichen Funktionen des Betriebssystems wie z.B. das FMS sind dann in Form von Systemprozessen angelegt – und dadurch frei austauschbar.

Was bedeutet die jeweilige Architektur in der Praxis? Zunächst wird man die Auswahl des Betriebssystems nicht nur an quantifizierbaren Aussagen wie Reaktionszeiten messen, sondern auch an Qualitätsaussagen. Modulare Systeme sind nicht nur wartbarer und skalierbarer, sie sind ggf. auch fehlertoleranter, d.h., Abstürze in Systemprozessen müssen nicht zugleich das gesamte System lahm legen. Auch der Nachteil soll nicht verschwiegen werden: Je modularer ein System ist, desto mehr Rechenzeit wird aufzuwenden sein, denn Schnittstellen und IPC kosten Rechenzeit.

6.3 Scheduling-Strategien

Der für Echtzeitbetriebssysteme wichtigste Teil folgt nunmehr mit dem Schedulingsystem. Um zu verstehen, worum es sich dabei eigentlich handelt, muss man die Begriffe Programm, Prozess und ggf. auch Thread näher erläutern. Mit einem *Programm* wird die statische Folge von Anweisungen in einer Programmiersprache unter Nutzung von Daten [9, Abschnitt 7.3] verstanden.

6.3.1 Grundbegriffe

Ein (sequenzieller) *Prozess* (Process) besteht hingegen aus einem *eigenen Adressraum* und einer *dynamische Folge von Aktionen*, die durch Ausführung eines Programms auf einem Prozessor zustande kommt. Der Prozess ist damit durch seinen zeitlich veränderlichen Zustand gekennzeichnet. Er wird durch das Betriebssystem als Folge eines Auftrags erzeugt, außerdem können Betriebssysteme die Erzeugung mehrerer Prozesse unterstützen.

Da ein Prozess immer einen großen Kontext (Dateisystem etc.) mit sich führt, entstand der Wunsch nach Leichtgewichtsprozessen, mit *Thread* (Programmfaden) bezeichnet. Hiermit wird ein sequenzieller Ausführungsfaden mit minimalem Kontext (Stack, Registersatz) innerhalb einer Ausführungsumgebung (Prozess) bezeichnet. Jeder Prozess hat damit mindestens einen Thread, mehrere Threads eines Prozesses teilen sich den gesamten Adressraum sowie weitere Betriebsmittel des Prozesses.

Mit einem Multithreading zielt man also darauf ab, einen Prozess selbst wenn möglich zu parallelisieren. Je nach Gesamtauslegung des Systems kann es nun von Vorteil sein, Teilaufgaben in Threads (besserer Datenaustausch, schnellere Umschaltung zwischen Programmteilen) oder in Prozessen (schärfere Trennung zwischen den Systemteilen) zu implementieren. Je nachdem, welche Formen durch das Betriebssystem unterstützt werden und wie die Inter-Prozess-Kommunikation ausgebaut ist, kann die Auswahl eines geeigneten Betriebssystems beeinflusst werden.

Der Begriff der Task wird häufig etwas unscharf verwendet, meist synonym zum Prozess, manchmal auch zum Thread. Aus diesem Grund wird *Multitasking* synonym zu *Multiprocessing* und *Multithreading* verwendet, falls nicht unterschieden werden muss.

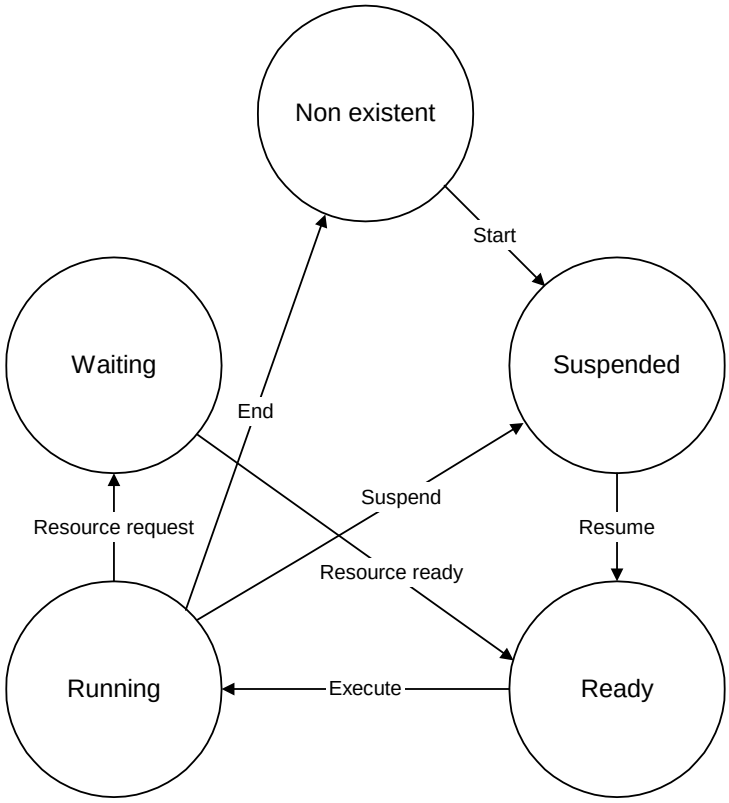

Bild 6.4 Beispiel für ein Prozesszustandsmodell

Neben den grundsätzlichen Fragen zur Unterstützung ist natürlich die Verwaltung von Prozessen bzw. Threads entscheidend für die Funktionalität des Betriebs-

systems. Dieser Teil wird als *Scheduler* bezeichnet, und Bild 6.4 gibt ein Beispiel für Prozesszustände, zwischen denen umgeschaltet werden kann.

In diesem Modell werden nur die Prozesse, die sich im Zustand *Ready* befinden, in Betracht gezogen, ausgeführt zu werden, und einer davon wird dann in *Running* versetzt. Wann genau kommt es nun dazu, dass das Betriebssystem überhaupt einen solchen Vorgang starten kann? Um es deutlich zu sagen, bislang sind alle Betriebssysteme Softwareprogramme, d.h., sie müssen ausgeführt werden, um eine Aktion zu starten.

Bei nicht-verdrängenden Strategien (non-preemptive) ist das Betriebssystem auf die Kooperation der Prozesse angewiesen. Der Aufruf *Suspend* bzw. *Resource request* wird von Prozess selbst durchgeführt, damit kann dann die Betriebssystem-software einen anderen Prozess schedulen.

Bei verdrängenden Strategien (preemptive) wird der Prozess an beliebiger Stelle unterbrochen, indem beispielsweise ein Timerinterrupt im Prozessor aufläuft. Als Folge dieses Interrupts wird das Betriebssystem aktiviert, und der Umschaltvor-gang läuft an. Hier sind zusätzlich natürlich auch Ereignissteuerung und kooperie-rendes Verhalten möglich.

6.3.2 Ansätze zum Scheduling

Im Allgemeinen sollte man von einem preemptiven System, also ohne eingebaute Kooperation ausgehen. Grundsätzlich muss man weiterhin beachten, dass nicht nur die Ressource Zeit, sondern ggf. auch andere, reale Ressourcen wie I/O-Ports oder Geräte auf ihre Verfügbarkeit hin überprüft werden müssen. Dies verschärft das Schedulingproblem noch weiter, sodass sich verschiedene Lösungsstrategien entwickelt haben. 4 verschiedene Schedulingklassen haben sich entwickelt:

Statische, Tabellen-gestützte Ansätze

In diesem Fall geht man davon aus, dass das System komplett zur Compilezeit bekannt ist. Es werden zur Laufzeit also keine neuen Tasks zugelassen, und damit kann während der Übersetzung berechnet werden, wann welche Task ablaufen bzw. starten soll.

Dieser statische Ansatz ist zwar unflexibel, garantiert aber die Multitaskingfähig-keit a priori. Insbesondere bei periodischen Abläufen kann dies erfolgreich einge-setzt werden. Bekannte Verfahren zum Scheduling, die hier eingesetzt werden sind:

- **Shortest Job First**, **Shortest Processing Time**: Dies erfordert die Kenntnis der anstehenden Rechenzeiten, der Scheduler wählt dann die Kurzläufe zuerst aus.
- **Terminabhängige Zuteilung**: Die nächste auszuführende Task wird durch das Kriterium **Earliest Deadline First** (Antwortzeitpunkt am nächsten) bestimmt.

Statische, Prioritäts-bestimmte Ansätze

Die Analyse, ob das System überhaupt ablauffähig ist, wird statisch durchgeführt. Es wird daraus allerdings keine Tabelle generiert, nach der verfahren wird, sondern es werden Prioritäten vergeben, und zur Laufzeit wird die jeweils höchste Priorität ausgeführt.

Durch diesen Ansatz wird nicht mehr eine exakte Reihenfolge festgelegt, sondern es werden Prioritäten fest vergeben (bei Rate Monotonic) oder dynamisch bestimmt. In jedem Fall bleibt die Anzahl der Tasks zur Laufzeit konstant. Beispiele hierfür sind:

- **Shortest Job First**, **Shortest Processing Time**: Wie bereits im Fall der Tabellensteuerung kann dieses Verfahren zur statischen Prioritätszuteilung genutzt werden, insbesondere bei periodischen Tasks. Im *Rate-Monotonic-Algorithmus* wird bestimmt, ob ein solches System überhaupt arbeiten kann. Gilt hier

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot \left(2^{1/n} - 1 \right) \tag{6.1}$$

$$\lim_{n \to \infty} \left(n \cdot \left(2^{1/n} - 1 \right) \right) = \ln 2 \tag{6.2}$$

  so kann das System garantiert gescheduled werden. C_i ist dabei die Ausführungszeit, T_i die Periode der Task i, und einer kürzeren Periode wird eine höhere Priorität zugeordnet. Die Grenze konvergiert für große n nach $\ln 2 \approx 0{,}69$ und stellt eine gültige, aber meist pessimistische Grenze dar. Es gilt die Grenze 1, wenn die Perioden ganzzahlig zur kleinsten Periode sind.

- **Terminabhängige Zuteilung**: Die Prioritäten und damit die nächste auszuführende Task werden durch das Kriterium **Earliest Deadline First** (Antwortzeitpunkt am nächsten) oder **Least Laxity First** (Differenz zum spättestmöglichen Startzeitpunkt) bestimmt.

Dynamisches, Plan-basiertes Scheduling

Um ein dynamisches Scheduling zu ermöglichen, das auch die Hinzunahme neuer Tasks gestattet, müssen diese neuen Tasks mit Informationen über ihren Rechenzeitbedarf sowie die erwarteten Antwortzeiten ausgestattet sein. Der Scheduler überprüft dann bei Start der neuen Tasks, ob und wie er diese einbauen kann. Wird sie garantiert, dann darf sie auch weiterlaufen. Bei Multiprozessorsystemen kann die Task im Negativfall auch an andere Prozessoren weitergegeben werden.

Dynamisches bestmögliches Scheduling

Diese Form des Scheduling wird in den meisten dynamischen Echtzeitbetriebssystemen verwendet, obwohl dieses Verfahren keine Echtzeitfähigkeit garantiert!

Eine neu hinzukommende Task wird also eingebaut, und es wird versucht, Wechselwirkungen zu den anderen Tasks möglichst zu vermeiden.

Man kann sich sicherlich vorstellen, wie dies im Fall eines Overload aussehen wird. Da in diesem Fall wohl jede Task einen Performanceverlust haben wird, natürlich nichts garantiert wird, ist dieser Ansatz für Echtzeit unbrauchbar.

Zusammenfassung

Man kann die unterschiedlichen Schedulingstrategien so zusammenfassen, dass statische Systeme, auch große, in den Griff zu bekommen sind. Für diese Systeme gibt es hervorragende Möglichkeiten der Berechnung, ein Überblick ist in [12, 13] gegeben.

Dynamische Systeme sind derzeit immer noch in einem Forschungsstadium.

7 Fallstudie: Verteiltes, eingebettetes System

Nach viel Theorie soll ein Beispiel (besser: eine Fallstudie) vieles von den dargestellten Dingen erläutern. Die Fallstudie basiert dabei auf realen Entwicklungen. Sie beinhaltet ein verteiltes System, eingebettet in eine Messumgebung.

7.1 Systemkonfiguration

Bild 7.1 zeigt den Aufbau des gedachten Systems. Aus technischen Gründen (Explosions-geschützter Bereich, gefährliche Spannungen, lange Übertragung per optischem Kabel etc.) wird die eigentliche Messeinheit (MessMonitor) von der Anzeige- und Bedieneinheit getrennt und durch einen lokalen Bus miteinander verbunden. Die dritte Einheit, das serielle EEPROM, beinhaltet Grunddaten des Geräts (z.B. Hersteller, Fabrikationsnummer) sowie Speicherplatz für Geräte-weite Daten (Einschaltwerte, Betriebsstundenzähler etc.).

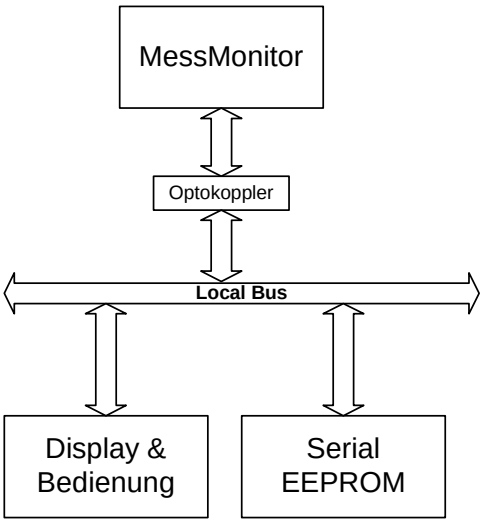

Bild 7.1 Systemkonfiguration im Fallbeispiel

Folgende generelle Aufgaben kommen auf dieses System zu:

1. Messwertaufnahme: Auf 4 Kanälen wird mit einer Abtastfrequenz von je 1 kSPS (Sample-per-Second) aufgenommen. Die Grundfrequenz des aufzunehmenden Signals sei 40 Hz, die Auflösung des AD-Wandlers 8 bit.

2. Bestimmung aller notwendigen Parameter zur Beurteilung der Güte der Signale. Hierzu zählen die tatsächliche Frequenz pro Kanal sowie die Über- bzw. Unterschreitung von gegebenen zulässigen Bereichen (für den Spitzenwert).
3. Anzeige der gemessenen Werte: Spitzenwert (pro Kanal), aktuelle Frequenz (pro Kanal), aktuelle Warnsituation.
4. Abfrage der Bedientasten, Anzeige des Menüs, Aufnahme neuer Grenzwerte.
5. Reprogrammierung bei Update der Software.

Ohne auf die Einzelheiten der Messwertauswertung etc. einzugehen soll dieses System konzipiert werden. Hierzu zählen im ersten Ansatz die Aufteilung der Aufgaben auf die beiden Rechner im Verbund sowie die Auslegung des lokalen Busses, im zweiten Ansatz dann die Konzeption der Software auf den Rechnern.

Grundsatz ist dabei, dass das System garantiert in Echtzeit arbeiten soll. Hierzu zählt, dass

- alle Messwerte garantiert verarbeitet werden und
- eine Alarmmeldung (mindestens zwei der vier Kurven sind ausgefallen, d.h., das Maximum ist < 10% vom zulässigen Minimalwert) in 4 ms bei der Anzeigeeinheit ist.

7.2 Auslegung des lokalen Busses

Geht man davon aus, dass das Gerät trotz seiner Verteilung in einem Gehäuse sitzt, fällt die Wahl für den lokalen Bus auf ein (sehr preiswertes) Inter-IC-Bussystem. Hier stehen mehrere zur Auswahl, der von Philips definierte I²C-Bus wird als Grundlage gewählt, wobei Multimasterbetrieb vorgesehen wird.

Aufgabe 1:

Suchen Sie die Definition des I²C-Bus und arbeiten Sie sie durch. Welche Adressierung finden Sie dort, gibt es Formatbeschränkungen wie etwa die Anzahl der gesendeten Bytes pro Paket? Wie ist der Zugriff bei Multimasterbetrieb geregelt? Ergibt sich hierdurch ein Echtzeitverhalten?

Aufgabe 2:

Wie steht es um die Unterstützung des I²C-Bus durch die Mikrocontroller der ATmega-Serie von Atmel? Klären Sie, wie dies durch die Hardware möglich ist und wie Sie dies in ein Programm mit Kommunikation durch Semaphoren (→ 5.1.2) abbilden können.

Die in Bild 7.1 gezeigten Optokoppler können die maximal zulässige Bitwechselfrequenz begrenzen. Während in der I²C-Bus-Definition 100 bzw. 400 kbit/s genannt sind, wird für diese Studie angenommen, dass die Transmissionsfrequenz auf 8 kbit/s reduziert werden muss.

Aufgabe 3:

Schätzen Sie in der angedachten Applikation ab, wie viele Datenpakete Sie pro Sekunde erzeugen und versenden müssen, wenn Sie

a) im MessMonitor nur messen

b) im MessMonitor auch auswerten und maximal pro Welle einen Statuswert erzeugen.

Wie regeln Sie jetzt das Echtzeitverhalten des Busses? Die Meldungen des MessMonitors müssen in jedem Fall sofort versendet werden, die der Display-Einheit können auf eine Lücke warten. Welche maximale Wartezeit ergibt sich dabei für eine Meldung des MessMonitors?

7.3 Architektur der Software

Nach reiflicher Analyse wird das System so ausgelegt, dass der MessMonitor nicht nur Messungen, sondern auch Auswertungen durchführt und diese an die Displayeinheit meldet. Tritt hierbei eine Situation auf, wo gewarnt oder alarmiert werden muss (was dies ist, sei hier undefiniert), wird die Meldung sofort gesendet, ansonsten wird nur eine Statusmeldung pro Kanal pro 2 Sekunden und eine Wertemeldung (Werte: Aktuelle Frequenz, Maximalwert) pro 10 Sekunden gesendet.

Aufgabe 4:

Stellen Sie einen Programmrahmen auf, mit dem die Zeitsteuerung für die Werte- und Nachrichtenübertragung ohne Störung anderer Routinen realisiert werden kann.

Literatur

[1] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

[2] *Beierlein, T.; Hagenbruch, O. (Hrsg.)*: Taschenbuch Mikroprozessortechnik. Carl Hanser Verlag München Wien, 2. Auflage, 2001.

[3] *Scholz, P.*: Softwareentwicklung eingebetteter Systeme. Springer Verlag Berlin Heidelberg New York, 2005.

[4] *Falk, H.; Marwedel, P.*: Source Code Optimization Techniques for Data Flow Dominated Embedded Software. Kluwer Academic Publishers Boston Dordrecht London, 2004.

[5] *Booch, G.; Rumbaugh, J.; Jacobson, I.*: Das UML-Benutzerhandbuch. Addison-Wesley München, 2. Auflage, 1999.

[6] http://www.systemc.org

[7] *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).

[8] *Mudge, T.*: Power: A First-Class Architectural Design Constraint. IEEE Computer Vol. 34, No. 4, pp. 52–58 (2001).

[9] *Brooks, D. et.al.*: Power-Aware Microarchitecture. IEEE Micro Vol. 20, No. 6, pp. 26–44 (2000).

[10] *Steinke, S.; Wehmeyer, L.; Marwedel, P.*: Software mit eingebautem Power-Management. Elektronik Vol. 50, H. 13, S. 62–67 (2001).

[11] *Uwe Schneider, Dieter Werner* (Hrsg.): ”Taschenbuch der Informatik“. Fachbuchverlag Leipzig im Carl Hanser Verlag, 3. Auflage,. München Wien 2000.

[12] http://www.faqs.org/faqs/realtime-computing/list/

[13] http://www.dedicated-systems.com/encyc/buyersguide/rtos/Dir228.html

[14] ”Special Issue on Real-Time Systems”. Proceedings of the IEEE 82(1), January 1994.

[15] ”Special Issue on Real-Time Systems”. Proceedings of the IEEE 91(7), July 2003.

[16] *Alexander G. Dean*, ”Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers”. IEEE Micro 24(4), pp. 10-22 (2004).

[17] *Timo Gramann, Dirk S. Mohl*, ”Precision Time Protocol IEEE 1588 in der Praxis”. Elektronik 52(24), S. 86–94 (2003).

[18] *Holger Blume, Hendrik T. Feldkämper, Thorsten von Sydow, Tobias G. Noll,* "Auf die Mischung kommt es an - Probleme beim Entwurf von zukünftigen Systems on Chip". Elektronik 53(19) S. 54 – 59, und Elektronik 53(20) S. 62 – 67 (2004).

[19] *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).

[20] *Christian Siemers,* "Prozessor-Technologie". tecCHANNEL-Compact, Verlag Interactive GmbH, München, Mai 2004.

[21] http://www.esterel-technologies.com/

[22] http://ieee1588.nist.gov/

[23] Gerad J. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code". IEEE Computer 39(6), pp. 95–97, 2006.

[24] http://www.splint.org

[25] 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).

[26] 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).

[27] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 3: Automatische statische Codeanalyse". Elektronik 54(9) S. 48 .. 53 (2005).

[28] Stephan Grünfelder, "Den Fehlern auf der Spur. Teil 4: Integrationstests – das ungeliebte Stiefkind". Elektronik 54(13) S. 73 .. 77 (2005).

[29] 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).

Sachwortverzeichnis

N

O

P

R

S

T