Ein ereignisbasiertes Betriebssystemkonzept für tief eingebettete Steuersysteme

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades eines

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Master of Science (M.Sc.)

Walther, Karsten

geboren am 24.3.1978 in Finsterwalde

Gutachter: Prof. Dr.-Ing. Jörg Nolte

Gutachter: Prof. Dr.-Ing. Rolf Kraemer

Gutachter: Prof. Dr.-Ing. Wolfgang Schröder-Preikschat

Tag der mündlichen Prüfung 14.4.2009

Inhaltsverzeichnis

1 Einleitung

> In the aggregate, PC microprocessors are responsible for less than 1% of all processors sold. Embedded processors outsell PC processors by more than 99%.
> – www.microcontroller.com –

1.1 Motivation

Tief eingebettete Systeme sind Bestandteil unseres Lebens, sie arbeiten meist versteckt z. B. in Klimaanlagen, Fahrzeugelektrik und Mobiltelefonen. Zur Zeit gibt es circa 100mal mehr eingebettete Systeme als Desktopsysteme [42] und es ist zu erwarten, dass dieses Verhältnis weiter ansteigt. Generell werden aus Kostengründen in eingebetteten Systemen sehr einfache Mikrocontroller mit niedriger Taktfrequenz und wenig Speicher eingesetzt. Daher sind nicht alle Programmiermodelle und -techniken, welche in leistungsfähigeren Systemen Stand der Technik sind, einsetzbar. Der Markterfolg hängt jedoch neben dem Preis auch von einer zeitigen Markteinführung und der Zuverlässigkeit des Produktes ab. Daher ist es wichtig, adäquate Konzepte und Werkzeuge bei der Entwicklung zu nutzen.

Die Herausforderung liegt darin, neue Konzepte zu entwickeln, mit denen sich extrem leistungsschwache Geräte schnell und einfach bei gleichzeitig hoher Qualität programmieren lassen. Das heißt, die Systeme müssen effizient mit ihren Ressourcen umgehen, zuverlässig funktionieren und leicht erweiterbar/wiederverwendbar sein. Mit der üblichen Programmierung auf einem niedrigen Abstraktionsniveau kann dieses Ziel nur sehr schwer erreicht werden.

Wie wichtig Abstraktionen sind, wird z. B. in [74] gezeigt. Dort wurde im Rahmen einer Lehrveranstaltung von verschiedenen Projektgruppen eine einfache aber nebenläufige Weichensteuerung implementiert. Von den Gruppen, welche das Taskmanagement in C implementieren mussten, schaffte es keine einzige, die Aufgabe in der vorgegebenen Zeit zu lösen. Von den Gruppen, welche auf eine ADA-Laufzeitumgebung mit Taskmanagement aufsetzen konnten, waren 75% erfolgreich.

Dennoch fehlt die Akzeptanz für Betriebssysteme bei den Entwicklern, da entweder neue Werkzeuge genutzt werden müssen oder die Betriebssysteme zu schwergewichtig sind. Die Entwickler programmieren daher sowohl den Anwendungsteil, als auch typische Betriebssystemfunktionen selbst. Dadurch werden typische Fehler bei der Ablaufplanung und bei der Synchronisation wiederholt gemacht. Weiterhin ist es nur sehr schwer möglich, solche Systeme zu portieren bzw. Kode wiederzuverwenden oder hoch komple-

xe Systeme zu verwirklichen, wie ein Antiblockiersystem oder eine Anlagensteuerung. Prinzipiell ist eine modellbasierte Entwicklung solcher Systeme zu empfehlen, da die resultierende Software eine höhere Qualität hat [32].

1.2 Bewertungskriterien für eingebettete Betriebssysteme

Eng verbunden mit der Frage nach dem Konzept zur Programmierung ist in eingebetteten Systemen das verwendete Betriebssystem. Dieses ist aufgrund der begrenzten Ressourcen die einzige Zwischenschicht zur Hardware und muss deshalb geeignete Abstraktionen und Mechanismen bereitstellen. Somit ist es der Schlüssel zum Erfolg für die Umsetzung eines Programmierkonzeptes, insbesondere da Anwendung und Betriebssystem oft miteinander verschmelzen. Die Entscheidung, ob ein Betriebssystem für eine Anwendung geeignet ist, kann an bestimmten Kriterien festgemacht werden. Im Folgenden sind die Kriterien aufgeführt, die für die Zielanwendungen am wichtigsten sind.

1.2.1 Modellbasierter Entwurf

Software für verschiedene Systeme und Anwendungen ist meist sehr verschieden, jedoch können sich Strukturen oder Mechanismen gleichen. Dies wird bei der modellbasierten Entwicklung von Anwendungen ausgenutzt.

Auf Modellebene lassen sich bestimmte Teile einer Anwendung, wie die Ausführungseinheiten oder die Schnittstellen, identifizieren und frühzeitig im Entwicklungsprozess Aussagen über Eigenschaften der Anwendung treffen, z. B. über die Einhaltung von Echtzeitanforderungen oder den Synchronisationsaufwand.

Modellbasierte Software besitzt implizit bestimmte Eigenschaften, ist beispielsweise frei von Nebenläufigkeitsproblemen wie Deadlocks [9]. Die Software muss deshalb nicht nachträglich auf diese Eigenschaften hin untersucht werden. Das steigert die Qualität und hat dazu geführt, dass der modellbasierte Entwurf in sicherheitskritischen Bereichen Stand der Technik ist [8, 94]. Aber auch bei anderen eingebetteten Systemen wird es in Zukunft unausweichlich sein, einen modellbasierten Ansatz zur Entwicklung zu nutzen, da dieser die Entwicklungszeiten verkürzt [26].

Der modellbasierte Entwurf hat noch einen weiteren Vorteil, den Wiedererkennungswert von Software bei den Entwicklern. Ist einem Programmierer das Modell bekannt, nach welchem eine Software entwickelt wurde, so kann er diese wesentlich schneller verstehen. Insbesondere bei größeren oder langfristigen Entwicklungen erweist sich das als Vorteil, da der Reibungsverlust zwischen den einzelnen Entwicklern reduziert wird [27]. Jedoch muss das genutzte Modell auch geeignete Abstraktionen bieten, um typische Anwendungen schnell und effizient implementieren zu können. Geeignete Abstraktionen finden sich zum Beispiel in domänenspezifischen Sprachen wie SDL (Structured Definition Language) [57] für Protokollbeschreibungen oder FUPs (Funktionspläne) [18] für Speicherprogrammierbare Steuerungen. Teilweise werden die Abstraktionen auch durch die verwendete Arbeitsumgebung vorgegeben, ein typisches Beispiel dafür ist Matlab Simulink [73].

Eng verbunden mit dem Modell ist die Laufzeitumgebung, auf die eine modellbasierte Anwendung abgebildet wird. Die Laufzeitumgebung sollte dabei die Nutzung der Modellabstraktionen unterstützen. In tief eingebetteten Systemen stellt aus Effizienzgründen das Betriebssystem bereits die Laufzeitumgebung dar.

1.2.2 Echtzeitfähigkeit

Der Begriff Echtzeitfähigkeit wird auf sehr unterschiedliche Art und Weise verwendet. Grundsätzlich gilt ein System jedoch als echtzeitfähig, wenn es innerhalb einer vorgegebenen Zeitschranke eine gestellte Aufgabe erfüllen kann [92]. Es wird zwischen harten und weichen Echtzeitsystemen unterschieden. Bei harten Echtzeitanforderungen ist das Nichteinhalten einer Zeitschranke aufgrund der möglichen Folgen (z. B. Gefährdung von Menschenleben) inakzeptabel. Bei weichen Echtzeitsystemen kann eine Verletzung der Zeitschranken hingenommen werden, wenn dies nicht zu oft geschieht. Ein Beispiel sind Aussetzer bei einer Videoübertragung. In [63, 69] und [92] ist eine ausführliche Darstellung von Echtzeitsystemen zu finden.

Die Echtzeitanforderungen sind von Anwendung zu Anwendung sehr verschieden. In einer Prozesssteuerung z. B. in einem Glasschmelzofen reicht eine Reaktionszeit von mehreren Sekunden oder Minuten. Ein Antiblockiersystem hingegen muss innerhalb von Millisekunden reagieren. Die Echtzeitfähigkeit eines Systems ist daher immer im Anwendungskontext zu bewerten. Die Abstraktionen, die ein Laufzeitsystem zur Verfügung stellt, können die statische Analyse der Echtzeitbedingungen stark unterstützen [14, 16]. Das verwendete Modell zur Programmierung sollte daher solche bieten.

1.2.3 Abarbeitungsmodell

Es besteht eine starke Beziehung zwischen der Echtzeitfähigkeit eines Systems und dem verwendeten Abarbeitungsmodell (Scheduling). Dieses legt die Granularität der ausführbaren/planbaren Einheiten des Systems fest und wie deren Ausführung sich gegenseitig überlappen kann. Mögliche planbare Einheiten sind z. B. Prozesse oder Funktionen. Das Abarbeitungsmodell dient somit dem Zerlegen der Anwendung in logische Blöcke und der Modellierung von Nebenläufigkeiten.

Wichtig ist die Möglichkeit, Algorithmen bei der Planung der ausführbaren Einheiten wechseln zu können, da es nicht **das** perfekte Abarbeitungsschema für alle Anwendungsfälle gibt. So wird in [20] gezeigt, dass statisch gesteuerte Verfahren besser für Steuerungsanwendungen sind, obwohl im allgemeinen Fall Earliest Deadline First (EDF) als optimales Verfahren dargestellt wird [17, 68]. Weitere Abarbeitungsverfahren sind beispielsweise Fixed Priority (FP), First Come First Served (FCFS), Time Triggered (TT) oder Varianten dieser Verfahren. In tief eingebetteten Systemen werden meist einfache Verfahren verwendet, da sie weniger systemseitigen Mehraufwand verursachen. Ein allgemeiner Überblick über Planungsverfahren wird zum Beispiel in [69] gegeben.

Grundsätzlich muss ein Betriebssystem die Verwendung verschiedener Abarbeitunsverfahren ermöglichen. Die Implementierung von planbaren Einheiten sollte dabei unabhängig vom verwendeten Abarbeitungsverfahren sein.

1.2.4 Synchronisation

Die Synchronisation ist ein großes Problem in vielen Systemen. Fehler aufgrund zeitlicher Abhängigkeiten treten meist nichtdeterministisch auf. Da solche Fehler schwer zu finden sind, muss ein Betriebssystem in geeigneter Weise die Synchronisation der Anwendung unterstützen. Im einfachsten Fall wird dies durch die Bereitstellung von Synchronisationsprimitiven wie Semaphoren, Mutex-Implementierung oder Spinlocks erreicht. Diese müssen jedoch vom Benutzer explizit und richtig genutzt werden. Besser ist es, wenn das verwendete Modell und die Laufzeitumgebung bereits dafür sorgen, dass die Anwendung synchronisiert ist bzw. Synchronisationspunkte automatisch erkannt und somit geschützt werden können.

1.2.5 Energiesparmechanismen

Betriebssystemseitige Energiesparmechanismen sind bei mobilen und batteriebetriebenen Geräten von existenzieller Bedeutung. So sollen beispielsweise Sensornetzwerke ohne externe Stromversorgung über Monate evtl. Jahre laufen. Aber auch in stationären Anwendungen mit externer Stromversorgung sind Energiesparmechanismen von Belang. Durch die Vielzahl der eingesetzten Systeme ergeben sich größere Energiesparpotenziale, wenn kleinere Steuerungssysteme mit weniger Rechenleistung genutzt werden können und diese nicht ständig unter Vollast laufen.

Gerade in Sensornetzwerkknoten ist der Mikrocontroller meist der Hauptstromverbraucher. Somit ist das Energiemanagement für diesen ein wesentlicher Faktor für die Überlebensdauer der Knoten. Das Betriebssystem muss daher in geeigneter Weise Energiesparmechanismen für die Anwendung zur Verfügung stellen oder selbsttätig durchführen.

1.2.6 Speichereffizienz

Ein großes Problem tief eingebetteter Systeme ist der typischerweise kleine Speicher. Dieser wird oft so klein wie möglich gewählt, da er die Kosten für das Gesamtsystem stark beeinflusst. Dies gilt insbesondere für Geräte, die in hoher Stückzahl produziert werden. Zudem hat ein größerer dynamischer Speicher auch einen höheren Stromverbrauch zur Folge. Tabelle 1.1 zeigt die Speicherausstattung einiger häufig verwendeter Mikrocontroller.

Die aufgeführten Mikrocontroller verfügen über Programmspeicher (ROM) von bis zu 512 KB. Es ist aber zu beachten, dass die Mikrocontroller teilweise über mehr Speicher verfügen als direkt adressierbar ist. Die Grenze bei der direkten Speicheradressierung von 16 Bit Mikrocontrollern liegt bei 64 KB. Der zusätzliche Speicher wird erst durch das Umschalten von Speicherbänken oder Speichersegmenten nutzbar, was aber nicht direkt von Hochsprachen wie C++ unterstützt wird.

Der Datenspeicher (RAM) auf den Mikrocontrollern ist meist noch wesentlich knapper. Die Gründe sind der Einfluß auf die Kosten eines Mikrocontrollers und der Energie-

Controller	Taktfrequenz	ROM	RAM
Texas Instruments MSP430	0,5 - 8 MHz	1-60 KB	128 Byte - 10 KB
Renesas M16C	20 - 32 MHz	24 - 512 KB	1 - 48 KB
Freescale HC(S)12	16 - 40 MHz	16-512 KB	2 - 32 KB
Atmel ATmega128	1 - 20 MHz	1-256 KB	128 Byte - 8 KB
Infineon C166	16 - 40 MHz	16-256 KB	1 - 11 KB

Tabelle 1.1: Speicherausstattung typischer Mikrocontroller

verbrauch. Die Speicherknappheit zeichnet die Zielsysteme dieser Arbeit aus und muss daher berücksichtigt werden.

1.2.7 Nutzung von Standardwerkzeugen

Für die Akzeptanz eines Systems unter den Entwicklern ist es von großem Vorteil, auf Standardwerkzeuge bei der Entwicklung zurückgreifen zu können. Diese gelten als zuverlässig und schützen vor schwer abschätzbaren Risiken. Insbesondere in sicherheitskritischen Anwendungen scheint es schwer vorstellbar, Systeme mit neuen Sprachen und Werkzeugen zu entwickeln. Für den Entwickler wäre vor allem das Erlernen einer neuen Programmiersprache ein erhebliches Hindernis bei der Programmierung und ruft daher Ablehnung hervor. Ein Ziel dieser Arbeit ist es daher, dass die Implementierung einer Anwendung in einer Standardprogrammiersprache möglich ist.

1.3 These

In dieser Arbeit wird ein Betriebssystemkonzept vorgestellt, mit welchem sich alle genannten Kriterien erfüllen lassen. Dadurch soll gezeigt werden, dass die aufgestellten Kriterien nicht nur einzeln erfüllbar sind, sondern auch gleichzeitig. Das wurde bisher nicht getan, wie in Kapitel 2 gezeigt wird. Die Arbeit hat einen integrativen Charakter und beschäftigt sich mit vielen Aspekten der Konzeption eines Betriebssystems. Das primäre Ziel ist, einen Gesamtansatz zu finden und nicht unbedingt neue Teillösungen zu entwickeln. So gibt es bereits eine lange wissenschaftliche Diskussion über die Vor- bzw. Nachteile von Thread-basierten und ereignisbasierten Systemen. Diese Diskussion ist ebenso Grundlage der Arbeit, wie die Vielzahl bereits implementierter Betriebssysteme für eingebettete Systeme und deren Konzepte. Dabei soll gezeigt werden, dass es eine Lücke bei der Programmierung tief eingebetteter Systeme gibt und das diese geschlossen werden kann. Letzteres erfolgt sowohl auf konzeptueller als auch auf praktischer Ebene. Im konzeptuellen Teil soll gezeigt werden, dass sich der ereignisbasierte Programmieransatz besser für die Programmierung tief eingebetteter Systeme eignet. Diese Aussage soll durch den praktischen Teil, in dem ein Referenzsystem implementiert wird, gestärkt werden.

Die gesamte Arbeit bezieht sich ausschließlich auf tief eingebettete Systeme, welche mit Low-End Mikrocontrollern mit wenigen KB Speicher ausgestattet sind. Das schließt jedoch nicht aus, dass das entwickelte Konzept später auch für leistungsfähigere Systeme geeignet ist.

1.4 Struktur der Arbeit

In Kapitel 2 werden existierende Plattformen zur Programmierung tief eingebetteter Systeme vorgestellt und anhand der in Abschnitt 1.2 genannten Kriterien bewertet. In Kapitel 3 wird das Ereignisflussmodell vorgestellt. Dieses erlaubt dem Programmierer eine einfache Modellierung seiner Anwendung und stellt gleichzeitig die Schnittstelle zum darunterliegenden System dar. In Kapitel 4 wird die konkrete Umsetzung, der im Ereignisflussmodell genutzten Abstraktionen, für das Reflex Betriebssystem gezeigt. Danach werden die Ablaufplanung und das Energiemanagement im Ereignisflussmodell in den Kapiteln 5 und 6 gesondert betrachtet. Abschließend werden Fallstudien in Kapitel 7 gezeigt und eine zusammenfassende Bewertung in Kapitel 8 gegeben.

2 Betriebssysteme für eingebettete Systeme

Tief eingebettete Systeme werden zumeist in echtzeitfähigen Steuerungsanwendungen und Sensornetzen eingesetzt. In beiden Anwendungsbereichen kommt ähnliche bzw. die gleiche Hardware zum Einsatz, wodurch die Anwendungen den gleichen Einschränkungen durch die Hardware unterliegen. Dennoch werden für beide Anwendungsbereiche unterschiedliche Betriebssysteme benutzt. Im eingebetteten Echtzeitbereich überwiegen Thread-basierte Betriebssysteme wie ECOS [72], FreeRTOS [41] oder Spring [95]. Im Bereich der Sensornetze, welcher sich seit der Jahrtausendwende entwickelt, werden vorwiegend ereignisbasierte Systeme, wie z. B. TinyOS [54] und Contiki [34], eingesetzt.

Dieses Kapitel stellt den Thread-basierten dem ereignisbasierten Ansatz gegenüber. Dazu erfolgt erst ein Exkurs in die wissenschaftliche Diskussion zu beiden Ansätzen, danach werden einzelne Vertreter vorgestellt und gezeigt inwieweit sie die in Abschnitt 1.2 aufgestellten Kriterien erfüllen. Darauf aufbauend wird dann begründet, warum der ereignisbasierte Ansatz besser zur Erfüllung aller genannten Kriterien geeignet ist.

2.1 Thread-basierte und ereignisbasierte Systeme im Vergleich

Seit den Anfangszeiten der modernen Computergeschichte (Mitte der 70 Jahre) werden zwei Ansätze zur Programmierung von Systemen diskutiert und verglichen: Der Thread-basierte und der ereignisbasierte Ansatz. Bereits im Jahr 1978 stellten Lauer und Needham fest, dass beide Ansätze wie Zwillinge sind [66]. Sie zeigten wie typische Muster bei der Programmierung auf den jeweils anderen Ansatz abgebildet werden können. Eine wichtige Erkenntnis ist zudem, dass sich konkrete Systeme oft nicht streng einordnen lassen, jedoch meist ein Ansatz überwiegt.

Thread-basierte Systeme sind dadurch gekennzeichnet, dass es pro Thread einen impliziten Ausführungszustand gibt, welcher durch den Stapel (Stack) des Threads beschrieben wird. Ein Thread hat eine beliebige Laufzeit und das System teilt den Threads die CPU zu. Threads können über viele verschiedene Arten kommunizieren, z. B. über gemeinsamen Speicher oder Nachrichten.

In ereignisbasierten Systemen gibt es einen einzigen Kontrollfluss und somit einen globalen Stapel. Die ausführbaren Einheiten werden in vielen Arbeiten als Tasks bezeichnet [34, 54, 98]. Das System führt die Tasks in Folge von Ereignissen aus. Die Tasks beenden sich nach einer Ausführung und geben eventuell benutzten Speicher auf dem Stapel frei (Run-to-completion-Semantik). Informationen über Ausführungen hinweg müssen expli-

zit gespeichert werden. Die Kommunikation zwischen Tasks erfolgt über Datenpuffer und Ereignisse.

Es gibt viele Arbeiten, welche darstellen, warum das eine dem jeweils anderen Konzept bei bestimmten Anwendungen überlegen ist. Ousterhout z. B. stellt in [80] dar, warum in einer Vielzahl von Anwendungen der ereignisbasierte Ansatz besser als der Thread-basierte ist. Hauptsächlicher Grund dafür ist seines Erachtens, dass Ereignisse die Probleme der Parallelisierung von Programmen (Synchronisation, Deadlocks, Debugging ...) verbergen. Nur wenn echte Parallelität (Mehrprozessorsysteme) vorliegt, wird seiner Ansicht nach ein Thread-basiertes System benötigt.

In [102] wird als Antwort auf Ousterhouts Aussagen gezeigt, warum im Serverbereich Threads besser als Ereignisse sind. Die Aussage ist jedoch nicht widersprüchlich zu denen Ousterhouts, da im Serverbereich Mehrprozessorsysteme eingesetzt werden. Als Lösung für die typischen Probleme Thread-basierter Systeme, wie. z. B. die Synchronisation und dem dynamisch wachsenden Stapel, werden vor allem Compiler-orientierte Ansätze angegeben. Kurioserweise wird als Beispiel die Sprache nesC [44] angeführt, in welcher es keine Thread-Abstraktion gibt. In nesC können atomare Sektionen gekennzeichnet werden, welche der Compiler dann automatisch schützt.

Allgemein werden ereignisbasierten Systemen folgende Vorteile gegenüber Thread-basierten zugeschrieben: Sie sind effizienter und leichtgewichtiger, erlauben feingranularere Nebenläufigkeiten und sind aufgrund der offensichtlichen Programmstruktur leichter zu synchronisieren bzw. zu analysieren. Thread-basierte Systeme haben weiterhin einen größeren dynamischen Speicherverbrauch, bedingt durch die Stapel der Threads. In [21] wird als Lösung vorgeschlagen, einen extra Stapel für das System zu benutzen, der bei der Unterbrechungsbehandlung genutzt wird. Dadurch können die Stapel für die Anwendungen kleiner ausgelegt werden. Andere Arbeiten beschäftigen sich mit der Vorabanalyse des Speicherbedarfs für den Stapel eines Threads, so dass möglichst wenig Speicher reserviert werden muss [62].

Nachteile von ereignisbasierten Systemen sind vor allem der Umgang mit blockierenden oder lang laufenden Funktionen, an den viele Programmierer nicht gewöhnt sind. Sie müssen explizit Wartezustände modellieren und können nicht einfach nur einen blockierenden Aufruf ausführen. Ein weiterhin häufig aufgeführter Nachteil von ereignisbasierten Systemen ist, dass diese nicht präemptiv arbeiten können [11, 66, 102]. Das ist jedoch nicht zutreffend, bereits in [7] wird gezeigt wie ein präemptives ereignisgetriebenes System mit einem Stapel funktionieren kann.

Es gibt aber auch Eigenschaften, die abhängig von der Anwendung vorteilhaft oder von Nachteil sind. So sind ereignisbasierte Systeme effizienter in Anwendungen mit aperiodischen Ereignissen oder periodischen mit geringer bis mittlerer Frequenz. In Anwendungen mit periodischen Ereignissen in hoher Frequenz hingegen sind aktiv abfragende Systeme von Vorteil, da nicht für jedes Ereignis eine Unterbrechungsbehandlung stattfinden muss.

Gegenwärtig gibt es Arbeiten, die versuchen, die Vorteile von beiden Ansätzen zu verbinden. In [11] treffen die Autoren die Aussage, dass dies speziell für eingebettete Systeme sinnvoll ist. Häufig besteht der Ansatz darin, ein ereignisbasiertes System um

Threading zu erweitern [2, 36, 105], dabei gehen jedoch Vorteile der ereignisbasierten Abarbeitung verloren. Aber es gibt auch Ansätze, welche versuchen, typische Probleme wie das der blockierenden Tasks rein ereignisgetrieben zu lösen [33, 35].

Weiterhin gibt es Ideen über die Weiterentwicklung der ereignisbasierten Programmierung hin zu objektbasierten Zustandsautomaten (**O**bject **S**tate **M**achines) [61]. Das vorgeschlagene Konzept sieht vor, den internen Objektzustand für die Ablaufsteuerung zu nutzen und das Objekt selbst zur Verknüpfung von Ereignissen und Tasks. Erforderlich für die Umsetzung ist ein System, welches dynamische Verknüpfungen von Ereignissen und Tasks erlaubt. Der Vorteil besteht laut den Autoren darin, dass nur die in einer Programmphase genutzten Zustandsinformationen gespeichert werden müssen. Das reduziert den Speicherbedarf von ereignisgetriebenen Systemen nochmals.

2.2 Bewertungsmaßstab

In den folgenden Abschnitten werden existierende Systeme vorgestellt und anhand der in Abschnitt 1.2 aufgestellten Kriterien bewertet. Grundsätzlich werden 3 Berwertungsstufen verwendet: erfüllt (+), teilweise erfüllt (o) und nicht erfüllt (-). Da die Kriterien jedoch recht unterschiedlich sind, wird im Folgenden die Bewertung im einzelnen erläutert.

Modellbasiert

- \- Es existiert eine rein funktionale Schnittstelle zum Kern.
- o Abstraktionen für die Programmierung werden vom Kern bereitgestellt.
- \+ Abstraktionen des Systems bilden die Grundlage der Struktur einer Anwendung.

Echtzeitunterstützung

- \- bietet keine Echtzeitunterstützung
- o bietet präemptive Abarbeitung, aber keine Unterstützung der Echtzeitanalyse
- \+ bietet präemptive Abarbeitung und unterstützt die Echtzeitanalyse

Abarbeitungsrahmenwerk

- \- Es wird nur ein Abarbeitungsschema angeboten.
- o Es werden mehrere Abarbeitungschemen angeboten oder es existiert zumindestens eine Schnittstelle, die den Austauch unterstützt.
- \+ Es werden mehrere Abarbeitungsschemen angeboten und der Anwendungskode kann unabhängig vom gewählten Schema implementiert werden.

Synchronisationsunterstützung

- \- Sie Synchronisation bleibt dem Anwender überlassen.
- o Es werden Synchronisationsmittel zur Verfügung gestellt.
- \+ Die Synchronisation wird weitgehend automatisch sichergestellt.

Energiesparmechanismen

- \- Energiesparmechanismen bleiben dem Anwender überlassen.
- o Es werden Mechanismen zum Energie sparen zur Verfügung gestellt.
- \+ Die Energiesparmechanismen arbeiten weitgehend implizit.

Speicherverbrauch

- \- Kode und Daten benötigen mehr als 256 KB im Speicher oder die Speicherverwaltung verlangt eine Hardwareunterstützung.
- o Kode und Daten benötigen weniger als 256 KB im Speicher oder die Speicherverwaltung stellt hohe Anforderungen.
- \+ Kode und Daten benötigen weniger als 64 KB im Speicher.

Standardwerkzeugkette

- \- Neue Werkzeuge sind notwendig, um Programme zu entwickeln.
- o nicht belegt
- \+ Die Werkzeuge zur Entwicklung existieren bereits.

2.3 Thread-basierte Systeme

In diesem Abschnitt werden Betriebssysteme für tief eingebettete Systeme vorgestellt, welche mit leichtgewichtigen Prozessen (Threads) arbeiten. Viele der Systeme werden auch für Echtzeitanwendungen benutzt und sind daher von besonderem Interesse. Die Beschreibung der Systeme erfolgt alphabetisch, da sie parallel entwickelt wurden und werden, somit kann keine eindeutige zeitliche Ordnung hergestellt werden.

2.3.1 AVRx

AVRx [4] ist eine Laufzeitumgebung für AVR-Mikrocontroller, die hauptsächlich für Robotikanwendungen genutzt wird. Der Anwendungskode wird in C geschrieben, das System selbst ist ausschließlich in Assembler programmiert, und stellt dem Benutzer 40 Systemaufrufe zur Verfügung. Die planbaren Einheiten sind leichtgewichtige Prozesse, die konzeptionell unendlich lange laufen. Der Ablaufplaner arbeitet präemptiv mit 16 Prioritätsstufen. Innerhalb einer Prioritätsstufe erfolgt die Abarbeitung kooperativ nach dem Round-Robin-Prinzip. In Bezug auf die Echtzeitfähigkeit unterstützt das System den Anwendungsprogrammierer durch Zeitgeber. Diese werden vom System bereitgestellt und gestatten die Aktivierung von Prozessen zu definierten Zeitpunkten.

Die Stärke des Systems ist das kleine Speicherabbild, welches je nach Konfiguration zwischen 1 KB und 1,4 KB groß ist. AVRx ist damit das kleinste der hier vorgestellten Systeme. Die größte Schwäche des Systems ist vor allem die schlechte Portierbarkeit und Erweiterbarkeit, aufgrund der Implementierung in Assembler. Außerdem ist die Funktionalität des Systems gering, die zur Verfügung gestellten Systemaufrufe dienen lediglich der Threadverwaltung und der Synchronisation mit Semaphoren. Die Speicherverwaltung (einschließlich Stapel), das Energiemanagement und die Ein-/Ausgabe muss komplett vom Anwendungsprogrammierer übernommen werden.

In der auf AVRx aufbauenden Arbeit [43] wird gezeigt, wie asynchrone Ausgabeoperationen effizient mit der Ausführung gewöhnlicher Tasks verwoben werden können. Typischerweise erfolgt die Übertragung von Nachrichten Byte-weise und für jedes Byte ist eine Unterbrechungsbehandlung nötig. Im vorgestellten Ansatz, den die Autoren “Software Thread Integration“ nennen, werden Nachrichten innerhalb eines lang laufenden Threads, z. B. zur Verschlüsselung von Nachrichten, mit abgearbeitet. Dafür wird im lang laufenden Thread vom Übersetzer Kode zur Übertragung eingewoben. Dies ist eine sehr aufwendige Technik und erfordert die Anpassung des genutzten Übersetzers. Jedoch wird ein wichtiges Problem adressiert, welches gerade in ereignisgetriebenen Systemen besteht: Der relativ große Aufwand für Unterbrechungsbehandlungen.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Energiesparen	Speicherverbrauch	Standardwerkzeugkette
-	o	-	-	-	o	+

Tabelle 2.1: Erfüllung der Kriterien in AVRx

2.3.2 ECOS

ECOS [72] ist ein verbreitetes Echtzeitbetriebssystem für eingebettete Systeme. Es bietet eine Unix-ähnliche Schnittstelle und stellt viele Mechanismen zur Echtzeitunterstützung

bereit, z. B. die Möglichkeit zur präemptiven Abarbeitung. ECOS ist für größere 16- und 32bit Mikrocontrollerplattformen portiert, welche in den meisten Fällen über mehrere Megabyte RAM und eine Speicherverwaltungseinheit (**M**emory **M**anagement **U**nit) verfügen.

Die planbaren Einheiten sind schwergewichtige Prozesse. Das verwendete Planungsschema ist wählbar. Zusammen mit der Möglichkeit der präemptiven Abarbeitung kann der Anwendungsprogrammierer seine Anwendung auf die entsprechenden Echtzeitanforderungen abstimmen, jedoch wird die Echtzeitanalyse nicht unterstützt.

Problematisch an ECOS sind die Ansprüche an die Hardware, welche für die in dieser Arbeit angestrebten Zielplattformen zu hoch sind. Die auf der Projektwebseite [37] als stabil markierten Portierungen nutzen allesamt einen 32bit Mikrocontroller/Mikroprozessor mit einer Speicherverwaltungseinheit.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerkzeugkette
-	o	+	-	-	-	+

Tabelle 2.2: Erfüllung der Kriterien in ECOS

2.3.3 Emeralds

Emeralds [108] ist ein Echtzeit-Mikrokern für tief eingebettete Steuersysteme, der Mitte der 90er Jahre entstand. Es ist ein sehr kompaktes System, der Kern benötigt auf einem Motorola 68040 nur etwa 13 KB Speicher. Es ist damit wesentlich kleiner als das zuvor beschriebene ECOS.

Das System stellt im Wesentlichen Dienste zur Prozessverwaltung, Synchronisation, Interprozesskommunikation, zeitgetriebenen Abarbeitung und zur Verwaltung von Unterbrechungen zur Verfügung. Treiber und Netzwerkkommunikation sind als eigenständige Prozesse ausgeführt.

Die Ablaufplanung erfolgt prioritätsgesteuert mit Hilfe eines 32bit Wertes und ist präemptiv. Empfohlen wird der Einsatz eines Schemas mit fixen Prioritäten. Jedoch wird auch ein Systemaufruf zur Änderung von Prioriäten zu Verfügung gestellt, welcher die Emulation von EDF-Abarbeitung (Earliest Deadline First) erlaubt.

Eine Besonderheit des Systems ist, dass die Treiber als Prozesse laufen. So brauchen nur angepasste Treiber für die jeweilige Anwendung geladen zu werden. Die Integration der Treiber aller verfügbaren Komponenten in den Kern würde unnötig Speicher verbrauchen.

Die Kommunikation zwischen Threads erfolgt über Nachrichten oder gemeinsamen Speicher. Nachrichten werden dabei an sogenannte Mailboxen geschickt. Diese können

entweder lokal sein oder sich auf anderen Knoten befinden. Die Besonderheit ist, dass die Abarbeitung der Nachrichten wahlweise ereignisgetrieben oder abfragend (Polling) erfolgen kann. Die ereignisgetriebene Abarbeitung wird für Prozesse empfohlen, welche selten oder aperiodisch Nachrichten erhalten.

Emeralds ist somit ein Thread-basiertes System, welches eine Nachrichtenschnittstelle zwischen Threads bietet. Jedoch verlangt die Implementierung das Vorhandensein einer MMU, welche auf Low-End Mikrocontrollern meist nicht vorhanden ist.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	+	-	-	-	-	+

Tabelle 2.3: Erfüllung der Kriterien in Emeralds

2.3.4 FreeRTOS und SafeRTOS

FreeRTOS [41] ist ein Betriebssystem für tief eingebettete Systeme. Die zur Verfügung gestellten Systemaufrufe dienen vor allem der Prozessverwaltung und Synchronisation.

SafeRTOS ist von FreeRTOS abgeleitet und besitzt die Zertifizierung nach DIN/IEC 61508 Stufe 3 [29]. Das entspricht einer Ausfallrate zwischen 10^-8 und 10^-7 pro Stunde. Im Wesentlichen wurde dies dadurch erreicht, dass Speicher im Gegensatz zu FreeRTOS nicht mehr dynamisch belegt wird und alle Systemaufrufe die übergebenen Parameter überprüfen.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	o	-	-	-	o	+

Tabelle 2.4: Erfüllung der Kriterien in FreeRTOS

Die Ablaufplanung in beiden Systemen erfolgt je nach Konfiguration präemptiv mit fixen Prioritäten oder kooperativ. Die planbaren Einheiten sind leichtgewichtige Prozesse. Ein großer Nachteil ist, dass die Prozesse gezielt für das gewählte Abarbeitungsschema implementiert werden müssen. Die Echtzeitfähigkeit begründet sich zudem im Wesentlichen darauf, dass eine präemptive Abarbeitung der Prozesse möglich ist.

FreeRTOS wurde auf viele verschiedene Plattformen portiert. Das wurde durch den geringen Assembleranteil im Kern und dem kleinen Funktionsumfang ermöglicht. Letzteres ist auch der große Nachteil, da dem Programmierer die Synchronisation seiner Anwendung, das Energiemanagement und die Ein-/Ausgabe gänzlich überlassen wird.

2.3.5 Mantis OS

Mantis [11] ist eine Laufzeitumgebung, welche speziell für Sensornetze entwickelt wurde. Die planbaren Einheiten des Systems sind leichtgewichtige Prozesse, welche präemptiv abgearbeitet werden. Es stehen insgesamt fünf Privilegstufen zur Auswahl. Wie schon bei AVRx werden Prozesse innerhalb einer Privilegstufe kooperativ abgearbeitet.

Die Entwickler stellen ihren Thread-basierten Ansatz dem ereignisgetriebenen von TinyOS [54] gegenüber. Ein grundlegendes Problem, dass sie und auch andere Gruppen [36, 65] bei TinyOS identifiziert haben, ist die nicht-präemptive Abarbeitung von Tasks. Werden lang laufende Tasks z. B. zur Kompression von Nachrichten verwendet, kann es zu kritischen Blockierungen anderer Tasks kommen. Die vorgeschlagene Lösung ist der Thread-basierte Ansatz.

Die Verwendung von Threads bringt jedoch Probleme mit sich. Das größte nach Meinung der Autoren ist der Platzbedarf für die Stapel der Prozesse. Es wird daran gearbeitet, diesen bei der Übersetzung des Programmes zu ermitteln und für jeden Prozess entsprechend Platz zu reservieren. Weiterhin wird vermutet, dass ereignisgetriebene Systeme Vorteile z. B. beim Energiemanagement bieten. Laut den Mantis Entwicklers sollte daher in Zukunft versucht werden, die Vorteile beider Ansätze in einem System zu vereinen.

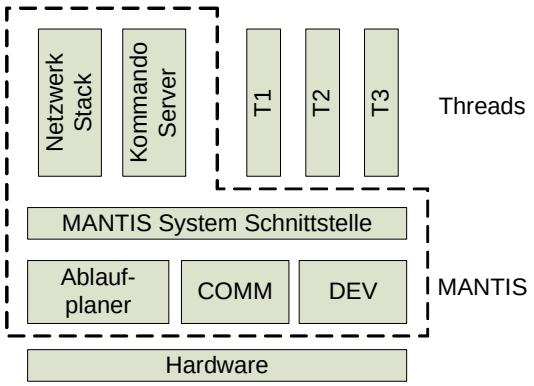


Abbildung 2.1: Architektur von Mantis (modifiziert entnommen aus [11])

Da Mantis für Sensornetzwerke entwickelt wurde, stellt das System bereits Unterstützung für die Kommunikation zur Verfügung. Im Wesentlichen ist dies eine pufferbasierte Schnittstelle zu den Kommunikationsgeräten. Bereits vorbereitete Netzwerkschichten sind auf der gleichen Ebene wie Anwendungsprozesse implementiert. Dadurch lassen sich

die Netzwerkschichten mit geringem Aufwand an die Anwendungsbedürfnisse anpassen. Die Architektur von Mantis ist in Abbildung 2.1 dargestellt.

Der Mantis Kern benötigt auf einem ATmega128 etwa 14 KB ROM und 500 Byte RAM. Die Anwendungen benötigen je nach Anzahl der verwendeten Threads und Puffer entsprechend mehr RAM. Das System ist zudem einfach portierbar, da es nur an sehr wenigen Stellen auf Assemblerkode zurückgreift.

Derzeit befindet sich die Möglichkeit zur dynamische Rekonfiguration über das Netzwerk in der Entwicklung. Das Energiemanagement muss derzeit durch den Anwendungsentwickler erfolgen, das System stellt dafür keine Unterstützung bereit.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	-	-	-	-	o	+

Tabelle 2.5: Erfüllung der Kriterien in Mantis

2.3.6 Open Ravenscar Kernel

ORK (Open Ravenscar Kernel) [28] ist eine Implementierung des Ravenscar-Profils [16] für Ada-Echtzeitsysteme. Ravenscar ist ein Systemprofil, welches höchste Zertifizierungsstufen für Echtzeitsysteme erreicht, z. B. DO-178B Level A [30], welche von der NASA und ESA für den Einsatz in Raketen verlangt werden [77].

Das Ravenscar-Profil legt fest, dass die Abarbeitung von Tasks deterministisch erfolgen muss. Dies wird im Wesentlichen durch Einschränkungen der Nebenläufigkeit erreicht.

In der Ravenscar-Stufe 0 sind Tasks nicht präemptiv und sowohl der Kern als auch die Tasks sind nicht unterbrechbar. Es werden auch Einschränkungen für den Anwendungskode vorgenommen, so dürfen bestimmte Sprachelemente von Ada wie `delay`[1] nicht verwendet werden. Die Einschränkungen begrenzen die Nebenläufigkeit, steigern aber die Analysierbarkeit der Anwendung. Das wurde in mehreren Arbeiten bestätigt, in welchen Ravenscar-Systeme formal verifiziert wurden [56, 77].

ORK implementiert auch andere Ravenscar-Profile für zeitgesteuerte und ereignisgetriebene Systeme, welche Unterbrechungen und eine präemptive Abarbeitung von Threads zulassen. Für die Echtzeitfähigkeit ist es in diesen Profilen zwingend notwendig, dass die Anzahl der Unterbrechungen begrenzt ist.

Bei der Ablaufplanung nutzt ORK zwei Listen, eine prioritätsbasierte für lauffähige Tasks und eine für wartende Tasks. Die Letztere ist nach der Reaktivierungszeit der Tasks geordnet. Die Speicherverwaltung erlaubt nur Speicheranforderungen zur Initialisierungszeit und keine Speicherfreigaben. Es handelt sich somit um eine eingeschränkte

[1] `delay` stoppt die Ausführung des laufenden Threads für eine angegebene Zeit

Freispeicherverwaltung. Die Entwickler halten eine vollständig dynamische Speicherverwaltung für nicht hinreichend analysierbar. Zur Speicherverwaltung gehört außerdem, dass der Stapel zur Laufzeit überwacht wird. Dazu wird bei Veränderung des Stapelzeigers geprüft, ob dieser weiterhin in einen vorher definierten Speicherbereich zeigt.

Weitere Besonderheiten des ORK Kerns sind, dass sowohl die Anwendungen als auch der Kern im priviligierten Modus laufen und die Synchronisation ausschließlich über das Sperren von Unterbrechungen erfolgt. Der ORK-Kern selbst hat eine Größe von 12 KB auf einem Sparc V7 Rechner, jedoch werden noch die ADA-Laufzeitbibliotheken benötigt. Die minimale Größe eines Programms ist daher 95 KB. ORK ist somit eine der kleinsten hart echtzeitfähigen Laufzeitumgebungen. Interessant sind vor allem die gezielten Einschränkungen der Nebenläufigkeit zur Sicherstellung der Analysierbarkeit.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	+	o	-	-	o	+

Tabelle 2.6: Erfüllung der Kriterien in ORK

2.3.7 OSEK/VDX

OSEK/VDX [79] ist eine allgemeine Schnittstellenspezifikation für ein Echtzeitbetriebssystem. Die Spezifikation soll dazu dienen, im Automobilbereich eine einheitliche Schnittstelle zwischen Anwendung und Betriebssystem zu schaffen. Für die Beschreibung der Schnittstellen von System und Anwendungen wird die Sprache OIL (OSEK Implementation Language) benutzt. Die Echzeitfähigkeit der zu implementierenden Anwendung wird als wichtigstes Merkmal angesehen. Zusätzlich sollen implementierende Systeme modular, flexibel und ressourcenschonend sein. Die kleinste Zielplattform sind 8-bit Mikrocontroller.

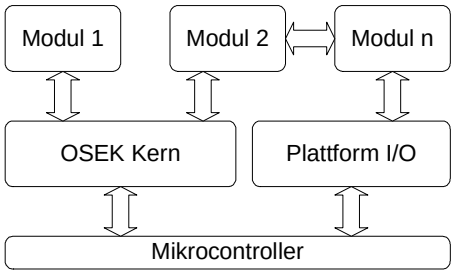


Abbildung 2.2: OSEK Modell (modifiziert entnommen aus [79])

Der Kern eines OSEK Systems stellt Funktionen zur Thread-Verwaltung, Synchronisation, Unterbrechungsbehandlung, Alarmierung, Intraprozesskommunikation und Fehlerdarstellung bereit. Für statisch zeitgesteuerte Systeme wird die erweiterte OSEKTime Schnittstelle definiert. Die Ein-/Ausgabe ist nicht Teil des Kerns, da die einzelnen Mikrocontroller sehr unterschiedlich sind. Die Ein-/Ausgabebibliothek existiert daher parallel zum Kern, wie in Abbildung 2.2 dargestellt ist.

	BCC1	BCC2	ECC1	ECC2
mehrfache Taskaktivierung	nein	ja	nein	nur für erweiterte Tasks
Anzahl aktiver Tasks	8		16	
mehrere Tasks pro Priorität	nein	ja	nein	ja
# Ereignisse pro Task	-		8	
# Prioritäten	8		16	
# Ressourcen	Ablaufplaner	Ablaufplaner + 7 Ressourcen		
interne Ressourcen	2			
Alarm	1			
Anwendungsmodi	1			

Tabelle 2.7: Konformitätsklassen in OSEK/VDX

Die Ablaufplanung erfolgt über statisch festgelegte Prioritäten. Dabei kann für eine Anwendung entschieden werden, ob die Abarbeitung nicht-präemptiv oder präemptiv erfolgen soll. Tasks werden in die Gruppen `basic` und `extended` eingeteilt. Bei der Basisvariante handelt es sich um Tasks mit Run-to-completion-Semantik. Die erweiterten können ihre Ausführung über den Systemaufruf `WaitEvent` unterbrechen. Einfache Tasks werden durch einen `schedule`-Aufruf zyklisch aktiviert, erweitertete Task können optional auch durch ein Ereignis aktiviert werden.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
o	+	o	-	-	+	+

Tabelle 2.8: Erfüllung der Kriterien in Osek

In OSEK/VDX sind Konformitätsklassen definiert, in denen Eigenschaften einer Anwendung sichergestellt werden müssen. Tabelle 2.7 zeigt die Anforderungen der einzelnen Konformitätsklassen. Diese unterscheiden sich hauptsächlich in der Planung der Tasks, z. B. ob mehrere die gleiche Priorität haben dürfen. Die Verwendung von erweiterten Tasks ist nur in den Konfigurationen ECC1 und ECC2 zulässig. Auffällig ist außerdem

die starke Beschränkung auf 8 bzw. 16 Tasks. Für die Sicherstellung der Echtzeitfähigkeit einer Anwendung sieht der OSEK/VDX Standard ein statisches System vor. Objekte werden ausschließlich während der Initialisierungsphase angelegt. Zusätzlich wird für die Ablaufplanung festgelegt, dass Prozesse eine statische Priorität besitzen.

2.3.8 PURE

PURE [10] ist kein Betriebssystem an sich, sondern ein Betriebssystembaukasten. Ziel der Entwickler ist es, ein für eine Anwendung angepasstes Betriebssystem generieren zu können. Dadurch kann eine sehr hohe Effizienz erreicht werden, da nicht benötigte Systemteile auch nicht im Kompilat enthalten sind. Zudem ist das enstehende System kleiner und lässt sich so leichter formal analysieren.

Der Kern ist je nach Einsatzszenario voll konfigurierbar. Abbildung 2.3 gibt einen Überblick.

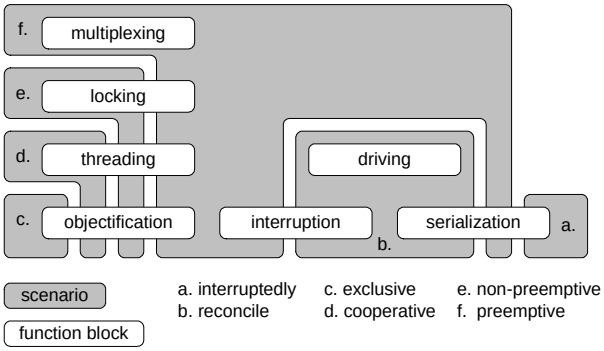


Abbildung 2.3: PURE Systemkonfigurationen (modifiziert entnommen aus [10])

Die minimale Konfiguration ist die für das `interruptedly`-Szenario. Dort werden lediglich grundlegende Funktionen zur Unterbrechungsbehandlung bereitgestellt. Mit diesen lassen sich anwendungsspezifische Behandlungsroutinen mit Unterbrechungen assoziieren. In der nächsten Stufe (`reconcile`) wird ein Pro-/Epilog Modell für die Unterbrechungsbehandlung eingeführt, in der die Behandlung der Unterbrechungen bei eingeschalteten Unterbrechungen serialisiert durchgeführt werden kann. In diesen Konfigurationen gibt es keine Thread-Abstraktion.

Für Systeme mit Thread-Abstraktion stehen die Konfigurationen `exclusive`, `cooperative`, `non-preemptive` und `preemptive` bereit. Ab der Konfiguration `non-preemptive` sind auch Funktionen zur Unterbrechungsbehandlung mit inbegriffen.

Threads stellen in PURE die planbaren Einheiten dar. Die Planungsstrategie ist frei wählbar, darin begründet sich auch der Anspruch auf die Echtzeitfähigkeit des Systems.

Die Implementierungssprache ist C++, wodurch das System leicht portabel ist. Unterstützte Plattformen sind z. B. x86, i860 ppc60x. Die Kodegröße des Kerns ist mit 812 Byte für das `interruptedly`-Szenario und 3642 Byte für das `preemptive`-Szenario relativ klein [10].

Interessant ist PURE vor allem wegen der Konfigurierbarkeit, aber auch im Hinblick auf die Trennung von funktionalem und nicht-funktionalem Kode [91]. PURE ist jedoch nicht auf tief eingebettete Systeme fokussiert, sondern viel weitreichender. Zudem beziehen sich viele in PURE verwendete Techniken auf Thread-basierte Systeme und sind daher nicht unbedingt für den Einsatz im ereignisbasierten Kontext geeignet.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	o	+	o	-	+	+

Tabelle 2.9: Erfüllung der Kriterien in PURE

2.3.9 RETOS

RETOS (Resilient Expandable and Threaded Operating System) [21] ist ein Betriebssystem speziell für den Einsatz in Sensornetzen. Die Nutzung von Threads als ausführbare Einheiten wird vor allem damit begründet, dass die präemptive Abarbeitung in ereignisgetriebenen Systemen nicht möglich ist. Die Entwickler halten somit blockierende Operationen in ereignisgetriebenen Systemen für schwer umsetzbar.

In RETOS gibt es neben den Stapeln für die Threads einen zusätzlichen für den Kern. Dieser wird für Unterbrechungsbehandlungen genutzt. Als Abarbeitungsschema kann zwischen Round Robin (RR), First Come First Serve (FCFS) und einer ereignisgewahren prioritätsbasierten Planung gewählt werden. Die letztere Variante ist präemptiv, wobei die Prioritäten der Threads dynamisch sind. Ein Thread bekommt eine höhere Priorität, wenn ein Zeitgeber für diesen abgelaufen ist und kann so laufende Threads unterbrechen.

Eine Besonderheit ist der Schutz des Systems gegen ein Überschreiben von Anwendungsseite. Der Schutz erfolgt sowohl statisch bei der Übersetzung, als auch durch dynamische Überwachung der Speicherzugriffe vom Anwendungskode. Dazu werden alle Zieladressen im Objektkode der Anwendung untersucht, sollten diese dynamisch berechnet werden, wird Kode eingeschoben, der die berechnete Adresse zur Laufzeit prüft. Dieser Ansatz ist stark maschinenabhängig und wirkt sich auch negativ auf das Laufzeitverhalten aus, da jeder dynamische Speicherzugriff zur Laufzeit überprüft wird.

Fest im System integriert ist weiterhin eine Netzwerkschicht, welche die Gerätetreiber, die Medienzugriffskontrolle und Verbindungtabellen der direkten Nachbarschaft beinhaltet. Darüber kann eine Wegewahlschicht (Routing) vom Programmierer eingesetzt werden, welche Systemrechte besitzt. Diese Architektur widerspricht dem Ansatz der anwendungsspezifischen Netzwerkarchitektur, welche von typischen Sensornetzplattformen verfolgt wird [60, 67].

Fälschlicherweise beanspruchen die Entwickler von RETOS in [21] für sich, das erste Betriebssystem für Sensorknoten implementiert zu haben, welches eine solche Viel-

zahl von Mikrocontrollern (Texas Instruments MSP430, Atmel ATmega128 und Chipcon CC2430) unterstützt. Jedoch waren sowohl TinyOS [54], Contiki [34], FreeRTOS [41] als auch Reflex [104] zum Zeitpunkt der ersten Veröffentlichung von RETOS im Jahr 2007 auf mindestens genauso vielen Mikrocontrollern lauffähig.

Obwohl dieses System eine Weiterentwicklung für den Sensornetzbereich sein soll, ist es eher als Rückschritt zu sehen. Zum einen wird die Ressourceneffizienz aufgegeben, eine minimale Anwendung benötigt bereits ca. 30 KB an Speicher. Zum anderen werden viele Vorteile, die aus dem Einsatz ereignisbasierter Systeme resultieren (z. B. einfache Synchronisation, Laufzeiteffizienz), nicht adressiert. Nicht zuletzt wurde der Energieverbrauch nicht betrachtet und auch kein Ansatz für das Energiemanagement in RETOS vorgestellt.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
-	-	-	-	-	o	+

Tabelle 2.10: Erfüllung der Kriterien in RETOS

2.3.10 Spring

Das Spring Betriebssystem wird seit Anfang der 90er Jahre entwickelt [78, 95, 96]. Es eignet sich insbesondere für den Einsatz in verlässlichen Systemen. Für diese ist es notwendig, eine Anwendung umfassend analysieren zu können. Das grundlegende Problem ist, dass bei Thread-basierten Systemen normalerweise die Abhängigkeiten zwischen den ausführbaren Einheiten (Präzedenzen) vom Anwender nicht mitmodelliert werden. Diese Abhängigkeiten ergeben sich aus der Interaktion von Threads und verlangen z. B., dass ein Thread A immer vor Thread B ausgeführt wird, damit es zu keiner Blockierung kommt. Die Entwickler des Spring Echtzeitbetriebssystems adressieren das Problem dadurch, dass Programme in SDL (Spring Description Language)[2] beschrieben werden. Diese Sprache erlaubt es, Abhängigkeiten von Prozessen, Ressourcen (Ports, gemeinsamer Speicher), Prozessgruppen und die Netzwerktopologie zu beschreiben.

Die Sprache SDL wird zur Beschreibung des Verhaltens und der Abhängigkeiten von Funktionen genutzt. Abbildung 2.4 zeigt die Programmierumgebung für Spring.

Im oberen Teil ist der Verlauf der Bearbeitung des Programmkodes zu sehen. Im mittleren wird die Verarbeitung der zugehörigen SDL-Beschreibung gezeigt und im unteren das Zusammenspiel von Analyse- und Generatorwerkzeugen. Zentraler Bestandteil ist die Datei `Full.db`, welche die gesamte Beschreibung des Programms und deren Abhängigkeiten enthält.

[2] nicht zu verwechseln mit der Structured Definition Language [57]

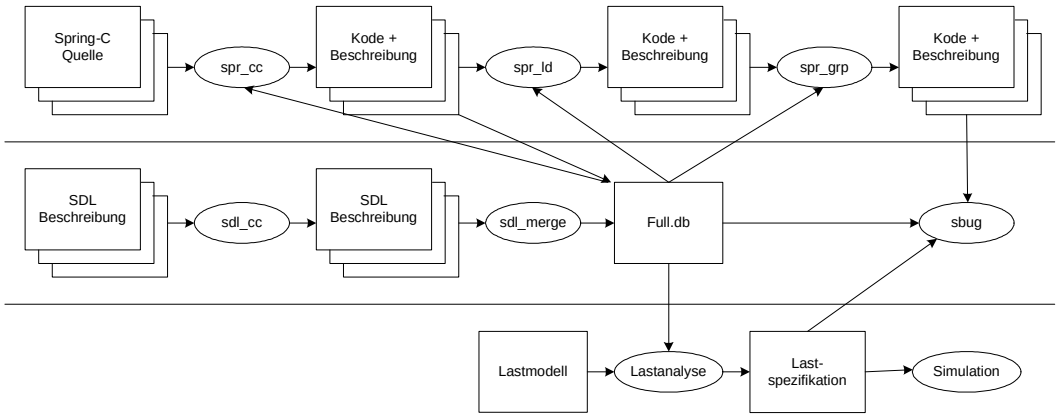


Abbildung 2.4: Die Spring Programmierumgebung (modifiziert entnommen aus [78])

Die Bearbeitung beginnt bei den SDL-Beschreibungen der einzelnen Quellen, diese werden kompiliert und allesamt der `Full.db` hinzugefügt. In der Beschreibung stehen Angaben darüber, ob eine Funktion z. B. unterbrechbar sein soll. Dieses Wissen wird zur parametrisierten Übersetzung der Quellen benutzt. Bei der Übersetzung werden gleichzeitig Laufzeitinformationen zu den Funktionen gewonnen, welche der Datei `Full.db` hinzugefügt werden. Vorläufiger Endpunkt der Übersetzung sind die einzelnen Prozesse (`spr_grp`).

Danach startet die Analyse der Daten. Es werden Simulationen durchgeführt, um die Last im System zu bestimmen. Auf Grundlage der Ergebnisse werden dann die Prozesse auf die verfügbaren Recheneinheiten aufgeteilt und die Systemabbilder für diese Knoten werden mit dem Werkzeug `sbug` erzeugt.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
+	+	+	+	-	o	-

Tabelle 2.11: Erfüllung der Kriterien in Spring

Das Spring System wurde in einer Reihe von Anwendungen eingesetzt. In [12] wird z. B. die Spring-basierte Steuerung eines Roboters zur Bestückung von Leiterplatten vorgestellt. Auch wenn sich mit Spring solche Systeme programmieren lassen, hat der Ansatz grundlegende Schwächen. So muss die SDL-Beschreibung einer Funktion mit deren Implementierung durch den Programmierer konsistent gehalten werden. Gerade im Bereich sicherheitskritischer Systeme ist das ein erheblicher Mangel. Außerdem greift das Spring-Rahmenwerk stark in den Übersetzungprozess ein, weshalb keine Standard-

werkzeuge benutzt werden können, sondern nur modifizierte Varianten.

Zusammenfassend ist Spring ein gutes Beispiel, wie viel Aufwand die Analyse echtzeitfähiger Thread-basierter Anwendungen erfordert. Der Hauptgrund sind die Abhängigkeiten von ausführbaren Einheiten untereinander, welche nur schwer automatisch zu ermitteln sind.

2.3.11 Andere Systeme

Es gibt noch eine Reihe weiterer Thread-basierter Echtzeitbetriebssysteme, z. B. μCOS [64], LynxOS [71] und QNX Neutrino [81] die hier nicht genauer betrachtet werden. Sie haben meist zu hohe Hardwareanforderungen und verfügen über keine wesentlich anderen Mechanismen als die bereits vorgestellten Systeme.

2.4 Ereignisbasierte Systeme

In diesem Abschnitt werden zuerst die ereignisbasierten Systeme Chimera [88, 98] und Boldstroke [94] vorgestellt, welche vorwiegend für Steuerungsaufgaben eingesetzt werden. Danach folgen die Betriebssysteme Contiki [34], SOS [49], TinyOS [54] und TinyGALS [22], die vorrangig für Sensornetzwerkknoten entwickelt wurden.

2.4.1 Chimera I & II

Chimera I [88] und Chimera II [98] sind Betriebssysteme für sensorbasierte Systeme. Beide Systeme basieren auf Port Based Objects (wörtlich: anschlussbasierte Objekte). Dies ist ein Anfang der 90er von Stewart [99] vorgeschlagenes formales Abarbeitungsmodell für objektbasierte Systeme.

In Abbildung 2.5 ist ein anschlussbasiertes Objekt dargestellt. Es verfügt über Ein- und Ausgänge zum Verbinden von Objekten zu einem Graphen. Jeder Ausgang speichert die zuletzt berechneten Daten. Diese werden über eine globale Zustandstabelle für andere Objekte verfügbar gemacht. Zusätzlich können Objekte private Zustände besitzen, welche nicht an andere Objekte propagiert werden. Diese sollen vor allem Geräteregister repräsentieren.

Aus den anschlussbasierten Objekten lassen sich komplette Anwendungen zusammensetzen. Dabei entspricht jedes Objekt einem Task und wird in jeder Runde einmal ausgeführt. Es wird davon ausgegangen, dass stets gültige Daten am Eingang vorliegen. Somit kann das Modell eigentlich nicht als ereignisbasiert bezeichnet werden, da die Funktionsblöcke global getaktet ausgeführt werden und die Ausführung unabhängig von dem Auftreten von Ereignissen ist. Für die Implementierung der Funktionsblöcke des Anwenders ist das jedoch unerheblich, da stets auf gültige Eingangsdaten zugegriffen werden kann. Dieses Abarbeitungsmodell wird z. B. auch in SPS-Steuerungen (**S**peicher **P**rogrammierbare **S**teuerung) benutzt [46].

Eine wichtige Erkenntnis Stewarts ist das „inside-out-method“-Paradigma [98]. Es wird nicht mehr das Betriebssystem von der Anwendung einbezogen, sondern das Betriebs-

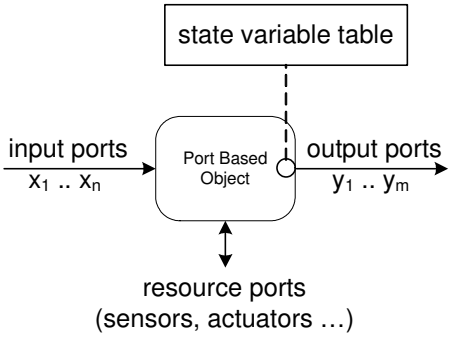


Abbildung 2.5: Port-Based-Object-Modell nach Stewart

system führt Teile der Anwendung nach Bedarf aus. Dadurch wird der Anwendungsprogrammierer von der Implementierung von Synchronisations-, Kommunikations- und Verbindungskode befreit. Die Port-Based-Objects enthalten nur noch den funktionalen Kode. Außerdem steigt die Analysierbarkeit des Gesamtsystems, da die Struktur der Anwendung bekannt ist.

Die Port-Abstraktion der Objekte führt außerdem zu einem hohen Grad an Modularität. Jegliche Tasks sind in diesem Modell frei kombinierbar, wenn die Datentypen der Ein-/Ausgänge kompatibel sind. Dieser Ansatz ist analog dem Hardwareentwurf, dort können Teilschaltungen kombiniert werden, wenn sie elektrisch kompatibel sind.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
+	+	-	+	-	+	+

Tabelle 2.12: Erfüllung der Kriterien in Chimera

Das Modell hat jedoch auch Grenzen. So verlangt das Synchronisationskonzept nach einem globalen Speicher für die Objektzustände, über welchen dann synchronisiert wird. Dadurch kann ein Objekt nicht mehrfach in einem System verwendet werden. Es ist auch nicht möglich, dass zwei Ausgänge auf einen Eingang verweisen. Das ist analog zu Hardware-Schaltungen, in denen nur mit Hilfe eines Tri-State-Treibers (Zustände 0,1 oder passiv) mehrere Ausgänge aufeinander geschaltet werden können. In Programmen wird ein solches Schema jedoch häufiger benutzt, z. B. wenn mehrere Objekte Nachrichten über einen Ausgabekanal ausgeben.

2.4.2 BoldStroke

Boldstroke ist eigentlich kein Betriebssytem, sondern ein Werkzeug, das Programme für eine Laufzeitumgebung übersetzt. Die Programmbeschreibung ist modellbasiert. Das zugrunde liegende Modell (im Folgenden Boldstroke-Modell genannt) soll die Kommunikation zwischen Diensten in verteilten eingebetteten Systemen vereinfachen. BoldStroke wird von Boeing für eingebettete Systeme in Flugzeugen verwendet [94]. Im Boldstroke-Modell werden komplexe Dienste als Komponenten aufgefasst, die über Kanäle miteinander kommunizieren. Die Kanäle übernehmen die Synchronisation und Pufferung beim Datenaustausch. Ein Ereignisdienst übernimmt die Verwaltung der Dienste. In Abbildung 2.6 ist dies dargestellt.

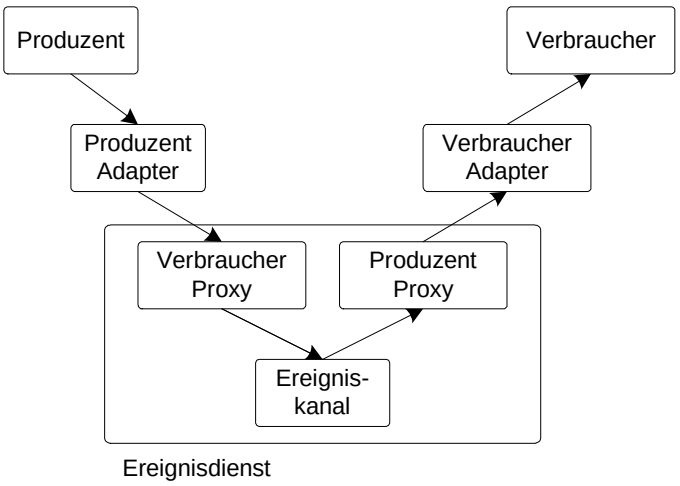


Abbildung 2.6: Das Boldstroke Modell

Es handelt sich um eine typische Broker-Architektur für Netzwerke. Für diese Arbeit interessant ist die Kapselung der einzelnen Dienste, welche nicht wie üblich auf Methodenaufrufen [107], sondern auf einem Austausch von Daten beruht. Der Grund ist, dass die meisten Anwendungen für Boldstroke datengetrieben sind. Liegen Daten vor, wird das als ein zu behandelndes Ereignis betrachtet. Über diese Art der Umsetzung lässt sich der Kontrollfluss vom Datenfluss abkoppeln. Das vereinfacht die Echtzeitanalyse, da der Kontrollfluss über eine übergeordnete Systeminstanz gesteuert wird und nicht durch den Anwendungskode. Das ist grundlegend für die Entwicklung sicherheitskritischer Flugzeugsoftware. Außerdem bemerken die Entwickler, dass durch das Boldstroke-Modell ein Maximum an Wiederverwendbarkeit für die einzelnen Komponenten erreicht wird.

Die Analysierbarkeit und Wiederverwendbarkeit waren bereits durch die Port-Based-Objects bekannt. Im Gegensatz dazu werden jedoch die Dienste im Boldstroke Modell nicht rundenweise sondern in Folge eines Datenaustausches ausgeführt. Datenfluss und Ereignisse sind also miteinander gekoppelt. Der Ablaufplaner arbeitet auf den Ereignissen, welche gemäß ihrer Priorität abgearbeitet werden. Die Wahl des konkreten

Abarbeitungsschemas ist dabei beliebig und beeinflusst nicht die Implementierung der Komponenten.

Durch die ereignisabhängige Ausführung wird die Last im System erheblich gesenkt, da nur benutzte Dienste ausgeführt werden. Um die potenziell asynchrone Abarbeitung zu ermöglichen, werden gepufferte Datenkanäle benötigt, welche in Boldstroke elementare Bestandteile des Modells sind. Sie werden zusätzlich genutzt, um entfernte und lokale Dienste auf gleiche Art und Weise anzusprechen.

Der wichtigste Grund für Boeing für die Benutzung des Modells ist die relativ einfache Analyse der verteilten Anwendungen, z. B. im Hinblick auf die Echtzeitfähigkeit. Das ist notwendig, da wie bereits erwähnt, das Modell zur Programmierung von höchst sicherheitskritischen Systemen in Flugzeugen genutzt wird. Weitere Vorteile bei diesem Modell sind Wiederverwendbarkeit und Erweiterbarkeit durch den Gebrauch von Komponenten. Damit werden für den Anwendungsprogrammierer eine Vielzahl von Problemen ausgeblendet. Für den Gebrauch in tief eingebetteten Systemen eignet sich Boldstroke dennoch nicht, da es die Anwendungen nicht auf Mikrocontroller sondern auf eine Java-Laufzeitumgebung abbildet.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
+	+	+	+	-	-	-

Tabelle 2.13: Erfüllung der Kriterien in BoldStroke

2.4.3 Contiki

Contiki [34] wurde als Betriebssystem für Sensornetzknoten entwickelt. Eines der Ziele bei der Entwicklung waren leichtgewichtige Mechanismen zum Laden von Programmkode auf die Sensorknoten zur Laufzeit. Außerdem beschäftigen sich die Entwickler mit der Implementierung von blockierenden bzw. lang laufenden Programmteilen in einem ereignisbasierten System.

Die ausführbaren Einheiten werden Prozesse genannt, es handelt sich dabei jedoch um passive Objekte ohne eigenen Stapel. Prozesse kommunizieren über Ereignisse miteinander, welche dem empfangenden Prozess vom System zugestellt werden (Abbildung 2.7). Es gibt synchrone und asynchrone Ereignisse. Erstere führen zur sofortigen Ausführung des empfangenden Prozesses, letztere werden vom System zur Verarbeitung in eine Warteschlange eingetragen. Der auslösende Prozess legt fest, ob das Ereignis synchron oder asynchron behandelt werden muss.

Neben den Ereignis-verarbeitenden Funktionen können Prozesse auch aktiv warten (pollen), diese Prozesse werden zyklisch vom System aufgerufen und dienen als Schnitt-

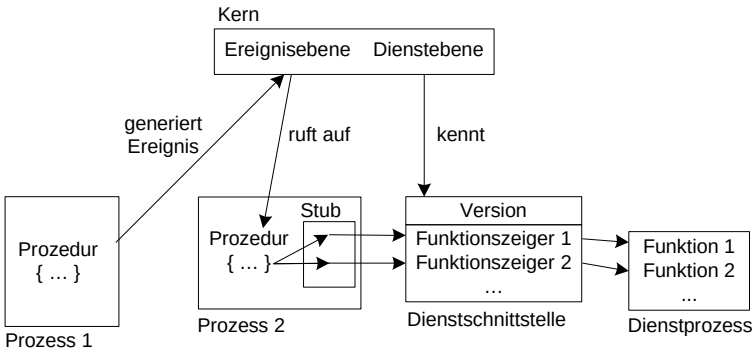


Abbildung 2.7: Prozesskommunikation in Contiki

stelle zwischen Unterbrechungsbehandlung und Prozessen. Die Unterbrechungsbehandlungsroutinen selbst können keine Ereignisse auslösen und Prozesse können sich nicht gegenseitig unterbrechen, dadurch ist das System implizit synchronisiert.

Weiterhin gibt es Dienstprozesse, welche allgemein verwendete Funktionen z. B. zur Ausgabeformatierung zur Verfügung stellen. Der Aufruf der Dienstfunktionen erfolgt über Aufrufweiterleitung (Stubs) und registrierte Dienstschnittstellen. Diese Indirektion erlaubt den Austausch und die Versionierung von Diensten. Außerdem müssen nur die genutzten Dienste in einem System geladen werden.

Auch die Anwendungsprozesse sind vom Basissystem im Speicher getrennt, wie in Abbildung 2.8 gezeigt wird. Somit ist es möglich, Anwendungen separat auf einen Sensorknoten zu laden. Jeder Prozess erhält einen Bereich im RAM und im ROM. Den bereitgestellten RAM verwaltet die Anwendung selbst, jedoch werden am Anfang des Bereichs für die Prozessverwaltung relevante Daten vom System abgelegt.

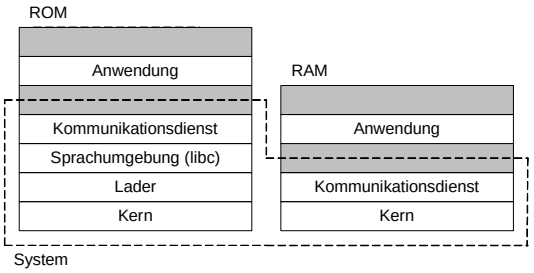


Abbildung 2.8: Speicherpartitionierung in Contiki

Contiki hat wie andere ereignisgetriebene, nicht präemptiv arbeitende Systeme das Problem der Implementierung von blockierenden Aufrufen. Funktionen, welche potenziell blockierende Aufrufe enthalten, müssen daher als Zustandsautomat modelliert werden und an der blockierenden Stelle in einen Wartezustand übergehen. Die Implementierung solcher Sachverhalte ist daher aufwendiger als in Thread-basierten Systemen.

Die Entwickler von Contiki schlagen sogenannte Protothreads [35] als Lösung vor. Das sind Funktionen, die automatisch verlassen werden, wenn sie blockieren müssen. Dabei wird die aktuelle Position der Ausführung im Kode als Wiedereinsprungspunkt gesichert. Beim nächsten Aufruf der Funktion startet diese an der gespeicherten Stelle. Protothreads dürfen keine Daten auf dem Stapel ablegen, da diese über die Aufrufe hinweg verloren gehen. Zudem ist es aufgrund der Stapelnutzung auch nicht möglich, sie in verschachtelten Funktionsaufrufen zu nutzen.

Als Beispiel für die Verwendung von Protothreads wird in [35] die Implementierung einer einfachen Medienzugriffsteuerung für ein drahtloses Netzwerk mit Schlaf- und Wachphase angegeben. In Listing 2.1 ist die Thread-basierte Implementierung der Steuerung zu sehen.

```
void radioSleepControl(){
    while(true){
        radio.on();
        wait(t_awake);
        timer1.set(t_sleep);
        if(!radio.idle()){
            timer2.set(t_wait);
            while( !radio.idle() &&
                   !timer2.timedOut() ){}
        }
        radio.off();
        while(!timer1.timedOut()){}
    }
}
```

Listing 2.1: Thread-basierte Implementierung einer blockierenden Funktion

Die Funktion durchläuft in einer Endlosschleife zuerst eine Wachphase und dann eine Schlafphase. Die Schlafphase wird immer dann betreten, wenn das Radio gerade nichts sendet oder empfängt (Zeile 6). Die Schlafphase wird verlassen, wenn der `timer2` ausläuft oder etwas empfangen wird (Zeilen 8 und 9). Die Funktion kann bei dem Aufruf von `timedOut()` als auch bei dem Aufruf von `idle()` blockieren.

Prinzipiell muss an den blockierenden Stellen in kooperativen ereignisgetriebenen Systemen der Ausführungblock verlassen werden, damit nicht das Gesamtsystem blockiert. Für die Wiederaufnahme der Arbeit an dieser Stelle muss vor dem Verlassen der aktuelle Zustand gesichert werden. Protothreads nehmen dem Programmierer diese Aufgabe ab. Es werden Makros verwendet, um den entsprechenden Kode einzusetzen. Die Implementierung als Protothread ist ähnlich der Thread-basierten Variante. Listing 2.2 zeigt die Implementierung des Beispiels aus Listing 2.1 mit einem Protothread.

```
void radioSleepControl(){
    PT_BEGIN
    while(true){
        radio.on();

        //einstellen der Zeit für die Wachphase
```

```
            timer1.set(t_awake);
            //warten aufs Ende der Wachphase
            PT_WAIT_UNTIL(timer1.timedOut())

            //einstellen der Zeit für die Schlafphase
            timer1.set(t_sleep);

            if(!radio.idle()){
                timer2.set(t_wait);

                //warten bis radio fertig oder timer2 abgelaufen
                PT_WAIT_UNTIL( radio.idle() ||
                               timer2.timedOut() )
            }
            radio.off();

            //warten aufs Ende der Schlafphase
            PT_WAIT_UNTIL(!timer1.timedOut())
        }
        PT_END
    }
```

Listing 2.2: Protothread-basierte Implementierung einer blockierenden Funktion

```
    void radioSleepControl(ProtoThread* pt){
        switch(pt->lc) {
        case 0:
            while(true){
                radio.on();
                timer1.set(t_awake);
                pt->lc = __LINE__ + 1;
        case __LINE__ :
                if(!timer1.expired())
                    return;
                timer1.set(t_sleep);
                if(!radio.idle()){
                    timer2.set(t_wait);
                pt->lc = __LINE__ + 1;
        case __LINE__ :
                if(!( radio.idle() ||
                       timer2.timedOut() ))
                    return;
                radio.off();
                pt->lc = __LINE__ + 1;
        case __LINE__ :
                if(!(timer1.timedOut()))
                    return;
            }
        }PT_END
    }
```

Listing 2.3: Expandierte Version der Protothread-Implementierung

Der Beginn und das Ende des Protothreads wird mit Makros (`PT_BEGIN` und `PT_END`) markiert und auch alle Stellen an den auf die Erfüllung einer Bedingung gewartet wird (`PT_WAIT_UNTIL`). Listing 2.3 zeigt die expandierte Version des Protothreads. Die ist im Wesentlichen ein durch eine `switch`-Anweisung beschriebener Zustandsautomat. Die Einsprungpunkte für den Zustandsautomaten erhalten durch das Makro `__LINE__` die Zeilennummer als Marke. Alternativ können in Contiki die genutzten Makros auch durch entsprechende Assembleranweisungen implementiert werden. Das ist etwas leichtgewichtiger aber schlecht portabel.

Die Protothreads in ihrer Umsetzung haben aber auch Nachteile. So funktionieren sie nur, wenn sie den Stapel nicht zur Speicherung von Variablen nutzen, die über mehrere Aufrufe hinweg genutzt werden sollen. Das kann jedoch verborgen für den Programmierer geschehen, auch wenn er keine Variablen explizit in der Protothread Funktion anlegt. Der Compiler kann temporär Variablen auf dem Stapel ablegen. Diese Variablen würden dann beim Verlassen des Threads nicht gesichert werden. Wird der Thread das nächste mal aktiviert, kann es zu einem Fehlverhalten kommen.

Das Konzept der Protothreads hilft, blockierende Operationen in ereignisgetriebenen Programmen darzustellen und löst damit ein vielfach angesprochenes Problem. Jedoch wird das Problem von lang laufenden Operationen z. B. Verschlüsselungsroutinen nicht gelöst. Dafür schlagen die Entwickler vor, präemptive Threads auf Nutzerebene zu implementieren. Mit diesen müssen langandauernde Operationen nicht mehr in mehrere Teilschritte unterteilt werden.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
o	-	-	+	-	+	+

Tabelle 2.14: Erfüllung der Kriterien in Contiki

2.4.4 SOS

SOS [49] ist ein Betriebssystem für Sensornetzknoten. Die Name ist übrigens kein Akronym, Deutungen reichen jedoch von “Save our Sensors“ bis “SOS Operating System“. Hauptziel des Projektes ist, ein rekonfigurierbares System für Sensornetzknoten zu schaffen. Der Ansatz ist die Implementierung eines nachrichtenbasierten Systems, in dem die Module lose gekoppelt sind und jeweils Nachrichtenbehandlungsroutinen implementieren.

Der Anwendungskode und Teile des Kerns sind in SOS in unabhängige Module unterteilt. Module sind eine Ansammlung von Funktionen. Darunter finden sich sowohl Behandlungsroutinen von Nachrichten, als auch private und öffentliche Funktionen. Die Behandlung der Nachrichten erfolgt über eine zentrale Funktion, welche die Nachrichten auf spezielle Behandlungsfunktionen verteilt. Öffentliche Funktionen können synchron von anderen Modulen aufgerufen werden.

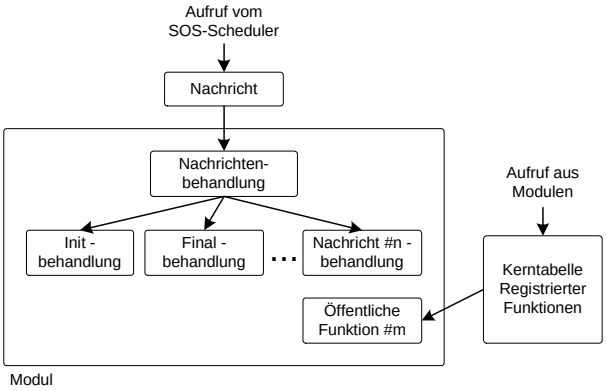


Abbildung 2.9: Struktur eines SOS Moduls (modifiziert entnommen aus [49])

Abbildung 2.9 zeigt ein SOS-Modul. Die zentrale Nachrichtenbehandlungsfunktion wird durch den Kern aufgerufen. Nach der Behandlung kehrt das Modul zum Kern zurück. Die Abarbeitung der Nachrichten erfolgt kooperativ und wird vom Kern verwaltet. Jedes Modul muss sogenannte `init` und `final` Nachrichten behandeln können. Diese werden dem Modul nach dem Laden und vor dem Entfernen desselbigen zugestellt. Die Nachrichten werden dazu in einer priorisierten Warteschlange gehalten. Jede Nachricht enthält die Adresse von Sender und Empfänger der Nachricht, ein Feld für den Nachrichtentyp (modulabhängig), den Puffer für die Daten und einen Zeiger auf den nächsten Puffer, siehe Abbildung 2.10.

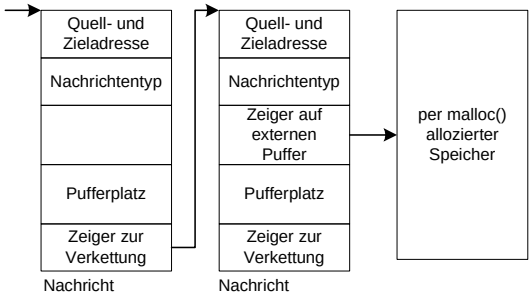


Abbildung 2.10: Nachrichtenstruktur in SOS (mod. entnommen aus [49]

Die Nachrichtenstruktur erlaubt es, beliebige Parameter über die Nachricht zu übergeben. Sollte der interne Puffer einer Nachricht nicht ausreichen, kann zusätzlich noch

externer Speicher referenziert werden. Die Besonderheit bei diesem ist, dass bei der Allokation eine Verantwortlichkeit eines Moduls (Sender oder Empfänger) für diesen Speicher festgelegt wird. Das verantwortliche Modul muss den Speicher nach der Benutzung freigeben.

Der Aufruf der registrierten Funktionen erfolgt nicht direkt, sondern immer über den Kern, welcher die Adressen von registrierten Funktionen in einer Tabelle hält. Ein Modul meldet bei seiner Initialisierung eigene Funktionen beim Kern an und teilt auch mit, welche Fremdfunktionen das Modul benutzt. Dieser Mechanismus wird vor allem für Kernmodule benutzt, z. B. zur Speicherverwaltung. Der Vorteil des indirekten Aufrufes ist die Entkoppelung von Modulen, welche dadurch separat ausgetauscht oder aktualisiert werden können. Das ist vorteilhaft in Sensornetzen, da die Menge der versendeten Daten bei einer Softwareaktualisierung sinkt. Die Dynamik der Verknüpfungen von Modulen, sei es über Nachrichten oder registrierte Funktionen, bringt aber auch Probleme mit sich. Die einzelnen Knoten in einem Netz sind z. B. nicht mehr als homogen anzusehen, da sich die Versionen der verwendeten Module auf den einzelnen Knoten unterscheiden können. Außerdem muss das System eine Reihe von Überwachungen zur Laufzeit vornehmen, um beispielsweise ungültige Verbindungen zwischen Modulen zu behandeln.

Das SOS Grundsystem inklusive Speicherverwaltung hat eine Kodegröße von ca. 20 KB auf einem Atmel ATmega128 und verbraucht etwa 2,5 KB RAM.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
o	-	-	-	-	+	+

Tabelle 2.15: Erfüllung der Kriterien in SOS

2.4.5 TinyOS

TinyOS [54] ist wahrscheinlich die meistgenutzte Plattform für Sensornetze. Das System ist ereignisbasiert und wurde vor allem mit der Prämisse entwickelt, möglichst klein zu sein. Zur Implementierung von Programmen wird die Sprache NesC [44] benutzt, welche Abstraktionen für die ereignisbasierte Programmierung zur Verfügung stellt. Die Programme werden erst zu C übersetzt und dieser Kode danach für die Zielplattform.

Eine Anwendung besteht aus Komponenten. Eine Komponente kann eine Implementierung sein und wird dann als Modul bezeichnet, oder eine Komponente ist eine Zusammenstellung von anderen Komponenten und wird dann Konfiguration genannt. In Abbildung 2.11 ist der Aufbau der Sense-and-Sense Anwendung SurgeC für TinyOS dargestellt. Die Anwendung wertet zyklisch einen Lichtsensor aus und übermittelt den Wert an eine Basisstation.

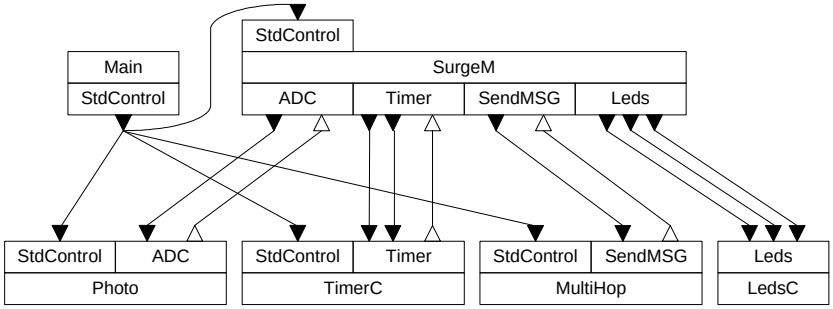


Abbildung 2.11: TinyOS Applikation SurgeC (modifiziert entnommen aus [44])

Die gezeigte Anwendung besteht aus den Modulen `Main`, `SurgeM`, `Photo`, `TimerC`, `Multihop` und `LedsC`. Bis auf `Main` implementieren alle Module Schnittstellen (oberer Kasten in den Modulen) direkt oder indirekt. `Main` und `SurgeM` nutzen außerdem noch Schnittstellen (unterer Kasten in den Modulen). Die Verbindungen zwischen den Modulen werden in der Konfiguration `SurgeC` festgelegt. Die Verbindungen können dabei asynchrone Ereignisschnittstellen sein (weißes Dreieck) oder synchrone Kommandos (schwarzes Dreieck). Listing 2.4 zeigt die Schnittstellendefinitionen der verwendeten Module `StdControl` und `Timer` sowie die Deklaration und Implementierung der Konfiguration `TimerC`.

```
interface StdControl {
    command result_t init();
}

interface Timer {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t fired();
}

configuration TimerC {
    provides {
        interface StdControl;
        interface Timer;
    }
}

implementation {
    components TimerM, HWClock;
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
    TimerM.Clk -> HWClock.Clock;
}
```

Listing 2.4: Beispiel einer TinyOS Komponente

TinyOS ist somit nicht rein ereignisgetrieben, da die Kommunikation zwischen den Modulen auch synchron erfolgen kann. Ereignisse werden nur an ausgewählten Stellen zur Entkopplung von Modulen verwendet. Es ist zu beachten, dass der Datenaustausch zwischen Modulen nicht mit Ereignissen verknüpft ist.

Ein großer Nachteil bei der Übersetzung von NesC Programmen ist, dass eine mehrfache Verwendung von Modulen schwierig ist. Das betrifft z. B. Treiber für seriellen Schnittstellen, wenn ein Mikrocontroller mehrere davon besitzt. Der Grund für die eingeschränkte Mehrfachverwendung ist die fehlende Kapselung der Module, wodurch keine parametrisierte Initialisierung möglich ist. Für jede serielle Schnittstelle eines Mikrocontrollers muss daher ein eigenes Modul implementiert werden. Bei Modulen, die dieses Problem nicht haben, kommt es bei einer Mehrfachverwendung zur Vervielfachung des Kodes, da jeder Modulbezeichner (Variablen, Ereignisse, Kommandos) mit dem Instanznamen erweitert wird.

Ablaufplanung in TinyOS

TinyOS 1.x arbeitet Tasks nach dem FCFS-Prinzip ab und benutzt dazu eine Tabelle fester Größe. In diese werden Zeiger auf parameterlose Funktionen eingetragen. Zusätzlich existiert noch ein Start- und ein End-Zeiger zur zyklischen Verwaltung der Tabelleneinträge. Ein Task kann mehrfach in der Warteschlange eingetragen sein. Dies hat jedoch mehrfach zu Systemabstürzen geführt[3]. In TinyOS 2.x wurde daher die Struktur geändert, die Planungstabelle hat genauso viele Einträge, wie es Tasks gibt. In jedem Eintrag steht die ID des nächsten auszuführenden Tasks. Die Start- und Endzeiger zeigen auf die Einträge des als nächstes auszuführenden Tasks sowie das Ende der ID Liste.

Der Vorteil der Implementierung ist der geringe Platzbedarf (3 Byte pro Task). Der Nachteil ist, dass die Tasks selbst entscheiden müssen, ob sie eventuell mehrfach ausgeführt werden müssen, da ein mehrfaches Anstoßen nicht vermerkt werden kann. Das verletzt wiederum den Grundsatz der ereignisgetriebenen Abarbeitung.

Neu in TinyOS 2.x ist auch, dass der Ablaufplaner durch ein austauschbares Modul implementiert wird. Jedoch wird nur eine FCFS Variante der Ablaufplanung bereitgestellt.

Ein oft angesprochenes Problem von TinyOS ist die rein kooperative Abarbeitung. Diese erschwert die Implementierung von lang laufenden Tasks und blockierenden Tasks. In einer Reihe von Arbeiten wurden daher Thread-Erweiterungen von TinyOS vorgeschlagen, welche das Problem beheben sollen.

In [105] führen die Autoren Fasern (fibers) ein. Eine Faser stellt einen Kontrollfluss dar. In einer Anwendung gibt es exakt eine Anwendungsfaser, die blockieren kann und einen eigenen Stapel besitzt. Alle anderen Fasern besitzen eine Run-to-completion-Semantik. Immer wenn die Faser mit dem eigenen Stapel blockiert, erfolgt ein Kontextwechsel und eine ereignisbasierte Faser wird abgearbeitet. Danach erfolgt wieder ein Wechsel zur blockierten Faser. Für den Systemteil von TinyOS sind für diesen Ansatz laut den Auto-

[3] www.tinyos.net TEP106

ren keine Änderungen nötig, jedoch wurden Betrachtungen zum globalen Energiemanagement ausgeklammert, welches durch den Ansatz defakto abgeschaltet wird. Außerdem muss beim Starten eines Tasks darauf geachtet werden, ob dies als normale TinyOS-Task oder als blockierende Faser geschehen soll. Die Transparenz an der aufrufenden Stelle geht damit verloren.

Ein recht ähnlicher Ansatz wird in [36] vorgestellt. Dort existiert auch exakt ein potenziell blockierender Thread. Jedoch gibt es einen zweiten Ablaufplaner im System, welcher in einem Thread TinyOS ausführt und in einem zweiten die potenziell blockierende Funktion. Der zweite Thread (von den Entwicklern SlaveThread genannt) wird immer aktiviert, wenn diesem von der TinyOS-Anwendung ein Signal zugestellt wird. Die Implementierung vernachlässigt aber wiederum die Energieverwaltung und ist nicht transparent für die verwendeten Module.

In [33] wird eine neue Implementierung der Ablaufplanung für TinyOS 2.x vorgestellt. Diese arbeitet mit fünf Prioritätsstufen, wobei für jede Prioritätsstufe ein extra Stapel verwendet wird. Die Prioritätsstufen sind:

1. hohe Priorität präemptiv
2. hohe Priorität nicht präemptiv
3. normale Priorität nicht präemptiv (Standard TinyOS Tasks)
4. niedrige Priorität nicht präemptiv
5. niedrige Priorität präemptiv

Die Semantik ist unkonventionell, so können Tasks der Stufe 1 alle anderen unterbrechen und Tasks der Stufe 5 von allen unterbrochen werden. Die Stufen 2 bis 4 hingegen Stellen eine 3-stufig prioritätsbasierte nicht-präemptive Verarbeitung dar. Die Entwickler begründen jedoch nicht, warum sie für jede Prioritätsstufe einen Stapel benötigen. Zudem muss bei der Implementierung eines Tasks bereits festgelegt werden, in welcher Prioritätsstufe dieser ausgeführt werden soll.

Umsetzung des Energiemanagements

TinyOS wurde speziell für Sensornetzwerkknoten entwickelt und muss daher das Energiemanagement unterstützen. Die Mechanismen von Version 1.x und 2.x werden aufgrund der großen Unterschiede getrennt betrachtet. In der Entwicklerdokumentation TEP112[4] sind weitere Informationen dazu zu finden.

1.x In dieser Version gab es noch kein einheitliches Konzept, sondern 2 konkurrierende Ansätze. Der erste wurde für die Telos Plattform, welche auf dem Atmel ATmega128 basiert, umgesetzt. Dort erfolgt ein expliziter Aufruf an das Betriebssystem, wenn das System in den Tiefschlafmodus gehen soll. Für den MSP430 ermittelt das System jedes Mal, wenn die Warteschlange des Ablaufplaners leer ist, in welchen Schlafmodus

[4] www.tinyos.net

gewechselt werden kann. In beiden Fällen überprüft das System alle Geräteregister und ermittelt aus deren aktuellem Zustand den tiefstmöglichen Schlafzustand.

Im ersten Ansatz wird es dem Benutzer überlassen, wann der Mikrocontroller schlafen gehen soll. Das birgt das Risiko, den falschen Zeitpunkt zu wählen und das System schlafen zu legen, obwohl noch eine Komponente aktiv ist. Dies führt unter Umständen zum Betreten des falschen Schlafmodus. Im zweiten Ansatz ist der Mehraufwand des Systems enorm. So werden jedesmal, wenn die Warteschlange leer ist, die Zustände aller Komponenten überprüft, um den niedrigst möglichen Schlafmodus zu ermitteln. Größtes Manko ist, wie die Entwickler feststellen, aber die fehlende Portabilität.

2.x In dieser Version von TinyOS wurde das Energiemanagement vereinheitlicht. Immer wenn die Warteschlange des Ablaufplaners leer ist, wird der tiefst mögliche Schlafzustand betreten. Es findet aber nur eine Neuberechnung dieses Zustandes statt, wenn Modulkonfigurationen sich ändern. Dazu muss jeder Treiber, wenn sich etwas Relevantes ändert, die Funktion `McuPowerState.update()` aufrufen, die ein `dirty-bit` setzt. Dieses Bit wird vom Ablaufplaner ausgewertet und bei Bedarf wird über eine hardwarespezifische Funktion der tiefste Schlafmodus ermittelt. Zusätzlich existiert für den Anwendungsentwickler noch die Möglichkeit, den tiefst möglichen Schlafmodus selbst festzulegen.

Das Schema entlastet den Programmierer von expliziten Aufrufen des Energiemanagements, er kann jedoch nicht direkt in bestimmte Schlafmodi wechseln, dazu müsste er selbst die komplette Weichenstellung im Vorfeld übernehmen, wobei andere Systemaktivitäten mit berücksichtigt werden müssen. Soll beispielsweise die serielle Schnittstelle (Funkmodul) deaktiviert werden, dann müssen erst alle noch ausstehenden Aufträge abgearbeit und alle möglichen Quellen neuer Aufträge blockiert werden.

Weiterhin ist für jede Portierung die Implementierung der Funktion nötig, welche den tiefsten Schlafmodus ermittelt. Diese überprüft jedoch alle Module unabhängig davon, ob diese jemals benutzt wurden oder nicht. Die Funktion ist somit sehr komplex.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
o	-	o	-	+	+	-

Tabelle 2.16: Erfüllung der Kriterien in TinyOS 2.x

Ein Vorteil von TinyOS ist die relativ große Komponentenbibliothek, welche durch die Vielzahl der Entwickler entstand. So gibt es viele Implementierungen von Netzwerkkomponenten, die der Anwendungsprogrammierer benutzen kann. Außerdem ist das System recht kompakt. Das Grundsystem benötigt etwa 3 KB ROM und 100 Byte RAM. Von Nachteil ist vor allem das begrenzte Abarbeitungsmodell, welches trotz austauschbarer

Abarbeitungsfunktion nicht ad hoc eine präemptive Abarbeitung ermöglicht. Außerdem kann durch die Nutzung der Sprache NesC keine Standardwerkzeugkette verwendet werden.

2.4.6 TinyGals

Bei der Programmierung mit TinyOS treten Probleme mit dem Mix von asynchroner und synchroner Kopplung von Komponenten auf [11, 65]. So hängt es vom Anwendungskontext einer Komponente ab, ob diese synchronisiert werden muss. Das TinyGals Projekt [22] adressiert dieses Problem. Die Grundidee ist, die Anwendung in grobgranulare Module aufzuteilen, welche asynchron gekoppelt sind, intern jedoch synchron arbeiten. Diese Strukturierung begründet auch den Namen **G**lobally **A**synchronous **L**ocally **S**ynchronous (Global Asynchron Lokal Synchron).

TinyGals basiert auf TinyOS. Verändert wurde vor allem die Ablaufplanung und es wurden neue Abstraktionen für die Strukturierung von Programmen eingeführt. Dennoch können original TinyOS-Komponenten weiter verwendet werden. Geschrieben werden die Anwendungen in GalsC [22].

Abbildung 2.12 zeigt die Darstellung einer TinyGals Anwendung. Diese besteht aus drei Modulen, die logische Einheiten bilden. Bestandteile der Module sind Eingänge, Ausgänge, Variablen, Komponenten und Verbindungen. Die Komponenten in den Modulen sind äquivalent zu den Komponenten von TinyOS.

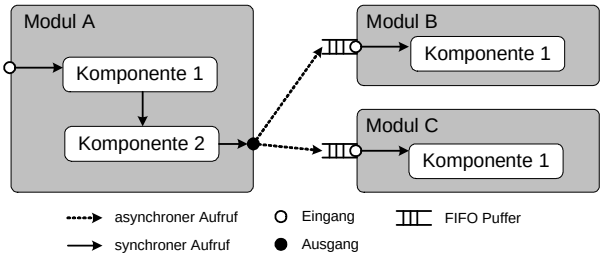


Abbildung 2.12: TinyGALS Applikation (mod. entnommen aus [22])

Die Interaktion von Komponenten innerhalb eines Moduls ist stets synchron. Die Kommunikation zwischen Komponenten in unterschiedlichen Modulen erfolgt über die Ein- und Ausgänge der Module und ist asynchron. Die Eingänge puffern die Ereignisse, wobei ausschließlich FIFO-Puffer verwendet werden. Das Schreiben in einen Puffer triggert dabei die Ausführung der assoziierten Komponente. Die Daten der Puffer werden vom Ablaufplaner gelesen und beim Aufruf der Komponente (entspricht einer Funktion) als Parameter übergeben. Datenfluss und Ereignisse sind so miteinander verknüpft.

Neu eingeführt werden synchronisierte Datenobjekte (TinyGuys). Diese werden für globale Variablen wie Knotenidentifikationsnummern benutzt. TinyGuys basieren in der Implementierung auf Stewarts Port Based Objects [98], welche globale Variablen zum Datenaustausch nutzen. Das Schreiben dieser Variablen ist asynchron, das Lesen synchron.

Beim Schreiben wird der neue Wert deshalb gepuffert und erst zwischen den Komponentenausführungen vom Ablaufplaner geschrieben. Dadurch kann das Lesen ohne das Setzen von Sperren (lock-free) durchgeführt werden.

Als eine der Stärken der benutzten Sprache GalsC wird hervorgehoben, dass die automatische Synchronisation beim Zugriff auf Variablen möglich wird. Der Grund ist die ausschließliche Kommunikation zwischen Modulen über Ein-/Ausgänge oder synchronisierte Variablen. Zusätzliche Einschränkungen gelten für Unterbrechungsbehandlungsroutinen, die dürfen nicht auf Modulvariablen zugreifen oder Komponenten aufrufen. Da für TinyGals nur eine nicht präemptive FCFS-Planung implementiert wurde, reichen die letzteren Einschränkungen bereits für die Synchronisation der Anwendung aus, da sich Komponenten nicht gegenseitig unterbrechen können.

Modellbasiert	Echtzeit	Abarbeitung	Synchronisation	Enerpiesparen	Speicherverbrauch	Standardwerk-zeugkette
+	-	o	+	+	+	-

Tabelle 2.17: Erfüllung der Kriterien in TinyGALS

2.5 Zusammenfassung der Systeme

In Tabelle 2.18 ist dargestellt, inwieweit die vorgestellten Systeme den in der Einleitung angesprochenen Anforderungen für tief eingebettete Steuersysteme entsprechen. Kein einziges dieser Systeme erfüllt alle Anforderungen.

	AVRx	BoldStroke	Chimera I & II	Contiki	ECOS	Emeralds	FreeRTOS	Mantis	ORK	OSEK	Pure	RETOS	SOS	Spring	TinyOS 2.x	TinyGals
Modellbasiert	-	+	+	o	-	-	-	-	o	o	-	-	o	+	o	+
Echtzeitunterstützung	o	+	+	-	+	+	o	-	+	+	o	-	-	+	-	-
Abarbeitungsrahmenwerk	-	+	-	-	+	-	-	-	+	o	+	-	-	+	o	o
Implizite Synchronisation	-	+	+	+	-	-	-	-	-	-	o	-	-	+	-	+
Energiesparmechanismen	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+
Speicherverbrauch	o	-	+	+	-	-	o	o	o	+	+	o	+	o	+	+
Standardwerkzeugkette	+	-	+	+	+	+	+	+	+	+	+	+	+	-	-	-

Tabelle 2.18: Eigenschaften existierender Betriebssysteme für eingebettete Systeme

Die Abstraktionen, welche in Boldstroke, Chimera und TinyGals genutzt werden, scheinen besonders geeignet für die Programmierung tief eingebetteter Systeme. Insbesondere die in Boldstroke und TinyGals vorgenommene Verknüpfung des Sendens von Daten mit der dadurch ausgelösten Ausführung der empfangenden Instanz ist vorteilhaft. So ist z. B. die implizite Synchronisation einer Anwendung möglich.

Auf Seiten der Echtzeitunterstützung sind vor allem die Ravenscar-Profile interessant. Die Umsetzung eines solchen Profils erleichtert die Echtzeitanalyse einer Anwendung. Verfahren die auf einer vollkommenen Spezifikation einer Anwendung hinsichtlich aller echtzeitrelevanten Eigenschaften beruhen, wie z. B. bei Spring, verursachen einen großen Aufwand für den Anwendungsentwickler. Zudem muss die Spezifikation stets konsistent mit dem Anwendungskode gehalten werden. AVRx und FreeRTOS sind nur begrenzt echtzeitfähig, da sie zwar präemptive Abarbeitung anbieten, aber die Echtzeitanalyse nicht unterstützen. Außerdem bieten sie nicht wie Boldstroke, ECOS, ORK und Spring die freie Wahl eines Abarbeitungsschemas.

Die implizite Synchronisation ist nur in fünf Systemen zu finden. Bis auf Spring sind diese ereignisbasiert. In Spring wird die implizite Synchronisation nur über eine exzessive Spezifikation der Anwendung erreicht. Lediglich in Boldstroke und TinyGals ist die Synchronisation auch potenziell mit präemptiver Abarbeitung implizit möglich.

In den meisten betrachteten Systemen gibt es kein Energiemanagement. In Thread-basierten Systemen liegt das daran, dass ein implizites Energiemanagement ohne Zusatzinformationen zur Anwendung nicht vollständig möglich ist. Das System kann z. B. nicht entscheiden, ob sich ein Thread in einer Warteschleife befindet oder Daten verarbeitet. Aber auch Systeme wie Contiki und Retos, welche für den Einsatz in Sensornetzen gedacht sind, besitzen selbst keine Energieverwaltung. Lediglich TinyOS und TinyGals implementieren wirkungsvolle Energiesparmechanismen.

Vom Speicherverbrauch für den Kode sind die meisten Systeme geeignet für tief eingebettete Systeme. Lediglich bei Boldstroke, ECOS und Emeralds ist das nicht der Fall. Die Thread-basierten Systeme haben jedoch alle das Problem der Verwaltung der Stapel und sind daher für Systeme mit wenig Speicher nur eingeschränkt tauglich.

Alle Systeme bis auf Boldstroke, Spring, TinyOS und TinyGals können mit Standardwerkzeugen übersetzt werden. Insbesondere bei Spring ist der Aufwand der Werkzeugkette erheblich, da diese auch auf Maschinenkodeebene arbeitet. Bei den anderen wird lediglich eine Abbildung auf eine Zielsprache (Java oder C) vorgenommen.

3 Das Ereignisflussmodell

Keines der in Abschnitt 2.3 und Abschnitt 2.4 vorgestellten Systeme erfüllt alle der in Abschnitt 1.2 aufgeführten Kriterien. In diesem Kapitel wird das Ereignisflussmodell vorgestellt, welches es ermöglichen soll, ein System zu implementieren, dass alle genannten Kriterien erfüllt. Das Modell stellt die grundlegenden Abstraktionen bereit, die eine Anwendung nutzen kann.

Abbildung 3.1 zeigt ein ereignisflussbasiertes System. Die ausführbaren Einheiten des Systems sind Aktivitäten und Unterbrechungsbehandlungen mit einer Run-to-completion-Semantik. Die Aktivitäten sind planbar und werden durch einen Ablaufplaner verwaltet. Unterbrechungsbehandlungen werden direkt in Folge von Unterbrechungen ausgeführt. Die ausführbaren Einheiten sind in Komponenten gruppiert, welche Ein- und Ausgänge besitzen. Ein- und Ausgänge verschiedener Komponenten können durch Ereigniskanäle verbunden werden. Eingänge haben Puffercharakter, sind mit einer Aktivität assoziiert und stoßen diese exakt einmal für jedes gespeicherte Datum an. Ausgänge sind Stellvertreter für Ereigniskanäle und dienen der Entkopplung von Komponenten.

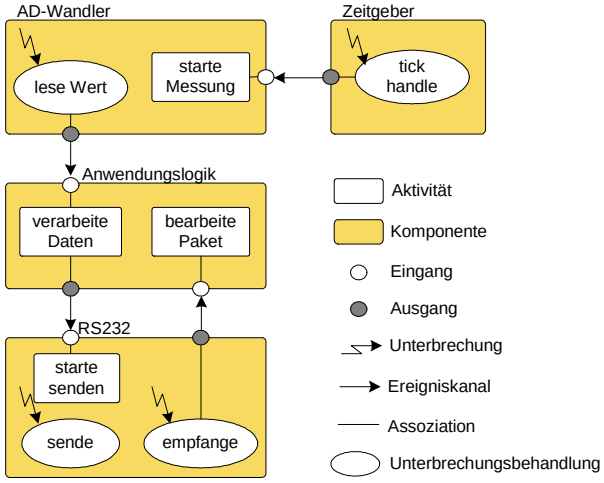


Abbildung 3.1: Ein Ereignisflussgraph

Die Kommunikation über die Ereigniskanäle ist asynchron und senderseitig initiiert. Aktivitäten und Unterbrechungsbehandlungen schreiben dazu auf einen assoziierten Ausgang. Das Schreiben wird als ein Ereignis aufgefasst. Das Ereignis wird vom Ereigniskanal zum verbundenen Eingang propagiert und dort für die asynchrone Abarbeitung gespeichert. Zusätzlich wird die assoziierte Aktivität getriggert und dadurch vom Ablaufplaner

zur Abarbeitung eingeplant. Sind mehrere Eingänge mit einer Aktivität assoziiert, so gilt eine Und-Semantik, dass heisst, nur wenn alle Eingänge Ereignisse enthalten, wird die Aktivität angestoßen. Initial werden Ereignisse durch Hardwareunterbrechungen ausgelöst.

Ein großer Vorteil des Modells ist, dass Aktivitäten nur dann ausgeführt werden, wenn gültige Daten an den zugehörigen Eingängen vorliegen. Wie in Datenflussgraphen gibt es daher keine Präzedenzprobleme. In Verbindung mit den nicht blockierenden Aktivitäten ist zudem das Gesamtsystem blockierungsfrei. Das ist möglich, da analog zu Datenflussmodellen der Weg der Daten beschrieben wird und somit der kausale Zusammenhang von ausführbaren Einheiten gegeben ist [59]. Eine ähnliche Art der Kopplung von Datenfluss mit Ereignissen wird z. B. in TinyGals [22] und Boldstroke [94] eingesetzt. Wichtig ist in jedem Fall die Atomarität des Speicherns von Daten mit dem Anstoßen einer ausführbaren Einheit. Dadurch braucht sich der Anwendungsprogrammierer nicht um die Pufferung der Ereignisse zu kümmern und der Zugriff auf die Ereigniskanäle ist synchronisiert.

Die Planung der Aktivitäten kann aufgrund der Run-to-completion-Semantik nach einem beliebigen Schema erfolgen. Da Aktivitäten zudem als Objekte aufzufassen sind, können sie einfach dem verwendeten Planungsschema entsprechend parametrisiert werden. Der Objektcharakter führt zudem dazu, dass auch Object State Machines [61] direkt umgesetzt werden können.

Das Ereignisflussmodell ähnelt von den in Abschnitt 2.4 vorgestellten ereignisbasierten Systemen/Modellen dem TinyGals Modell [22] am stärksten. Jedoch besitzt TinyGals keine Exactly-once-Semantik für die Triggerung von ausführbaren Einheiten nach einem Ereignis. Zudem sind dort die planbaren Einheiten Funktionen und es werden ausschließlich FIFO-Puffer als Eingänge für Komponenten verwendet. Außerdem ist der Einsatz präemptiver Planungsstrategien zwar prinzipiell möglich, verlangt aber nach Änderungen im System.

Es gibt auch starke Ähnlichkeiten zu BoldStroke, dort ist wie im Ereignisflussmodell die exactly-once Semantik bei der Triggerung umgesetzt. Jedoch wird bei BoldStroke das Ereignismodell auf eine Thread-basierte Java-Umgebung mit Broker-Architektur abgebildet. Dadurch ist BoldStroke zu schwergewichtig für den Einsatz in tief eingebetteten Systemen.

Im Vergleich zu Stewarts anschlussdefinierten Objekten ist das Ereignisflussmodell wesentlich leistungsfähiger, da eine ereignisbasierte Ablaufplanung integriert ist und nicht zyklisch alle Funktionen ausgeführt werden. Jedoch bleiben die wesentlichen Eigenschaften, wie die implizite Synchronisation und die Möglichkeit der Echtzeitanalyse von Anwendungen, erhalten.

Der wesentlichste Unterschied zu den Modellen von Contiki [34], TinyOS [54] und SOS [49] ist, dass diese nicht die Ereignisse mit dem Datenfluss koppeln. Durch die Kombination von ereignisbasierter Abarbeitung mit einem Datenflussschema ist es möglich, effiziente ereignisgetriebene Systeme zu entwickeln und dem Anwendungsprogrammierer eine einfache Datenflussabstraktion als Schnittstelle zur Verfügung zu stellen.

Im Folgenden werden zunächst die einzelnen Abstraktionen im Ereignisflussmodell

ausführlich betrachtet. Danach folgt die Diskussion, warum das Modell hilfreich für die Erfüllung der in Abschnitt 1.2 genannten Kriterien ist.

3.1 Die Aktivitäten

Die Aktivitäten stellen die planbaren Einheiten des Systems dar. Sie repräsentieren einfache Filter, komplexe Automaten oder auch Treiber. Aktivitäten sind im Wesentlichen passive Objekte, die eine `run()`-Methode besitzen, welche durch den Ablaufplaner aufgerufen wird. Die `run()`-Methode besitzt eine Run-to-completion-Semantik, weshalb nach der Ausführung eine Rückkehr zum aufrufenden Ablaufplaner erfolgt. Die Aktivitäten können auf die Variablen der umgebenden Komponente zugreifen oder auch selbst welche besitzen, da sie Objekte sind. So können Zustandsinformationen über mehrere Ausführungen hinweg gespeichert werden. Das führt dazu, dass Aktivitäten im Gegensatz zu gewöhnlichen Prozessen keinen eigenen Stapel benötigen. Während der Ausführung nutzen die Aktivitäten den Stapel des Systems. Dies ist eine große Erleichterung für die Implementierung von Anwendungen für tief eingebettete Systeme, da diese gemeinhin über wenig RAM verfügen.

Der Nachteil von passiven Objekten ist, dass die `run()`-Methode nicht blockieren darf, da so auch andere Aktivitäten blockiert werden können. Nur in präemptiven Systemen könnten niedrig priorisierte Aktivitäten blockieren und gleichzeitig höher priorisierte weiterlaufen. Das bringt jedoch die Gefahr von Deadlocks durch zyklische Abhängigkeiten mit sich, wenn z. B. durch falsche Parametrisierung eine Aktivität eine andere blockiert, welche ausgeführt werden müsste um die Blockierung aufzuheben. Zudem greift das implizite Energiemanagement nicht mehr.

Prinzipiell kann aber jede Aktivität, welche blockierende Aufrufe benutzt, als Zustandsautomat dargestellt werden. Abbildung 3.2 zeigt die Umsetzung einer typischen Funktion mit einem blockierenden Schreibzugriff als Zustandsautomat. Die Abarbeitung wird in mehrere Phasen aufgeteilt, wobei der Eintritt in eine neue Phase durch ein Ereignis (Erfolg, Fehler) ausgelöst wird.

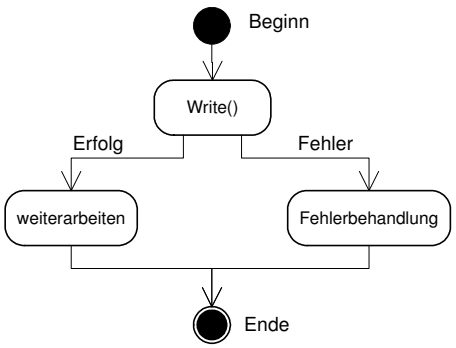


Abbildung 3.2: Umsetzung von blockierenden Schreibaufruf als Automat

Diese Art der Modellierung wird z. B. in SDL [57] benutzt. Die dort verwendeten

Prozesse blockieren ebenfalls nicht und vollziehen ausgelöst durch eine Nachricht eine Zustandsänderung. Daher können auch SDL-Prozesse leicht auf Aktivitäten abgebildet werden [48]. In TinyOS wird die Vorgehensweise Split-Phase-Operation genannt. Contiki erleichtert mit dem Protothread-Ansatz [35] dem Programmierer die Implementierung des Zustandsautomaten. Protothreads sind prinzipiell auch im Ereignisflussmodell nutzbar.

3.2 Unterbrechungen

Unterbrechungen sind die treibenden Elemente des Ereignisflusses. Sie sind die initialen Ereignisse einer Anwendung und werden entweder durch externe Bauelemente oder interne Komponenten des Mikrocontrollers ausgelöst. Sie können selbst weitere Ereignisse generieren, indem sie auf Ereigniskanäle schreiben. Unterbrechungen treten asynchron zum Kontrollfluss auf und werden unmittelbar von Unterbrechungsbehandlungsroutinen (Interrupt Handler) verarbeitet.

Unterbrechungsbehandlungsroutinen unterbrechen potenziell die Ausführung von Aktivitäten und können nicht geplant werden. Daher muss die Interaktion zwischen Unterbrechungsbehandlungsroutinen und Aktivitäten synchronisiert werden. Das ist implizit der Fall, wenn die Kommunikation von Unterbrechungsbehandlung zu Aktivität ausschließlich über Ereigniskanäle erfolgt. Im Gegensatz zu Contiki ist es so möglich, dass während einer Unterbrechungsbehandlung eine Aktivität angestoßen werden kann.

Wie die Aktivitäten sind Unterbrechungsbehandlungsroutinen passive Objekte und besitzen daher keinen eigenen Stapel, sondern benutzen den des Systems. Außerdem sind sie wie Aktivitäten blockierungsfrei.

Unterbrechungen sind i.d.R. unabhängige Ereignisse und können anhand des Musters, in dem sie auftreten, klassifiziert werden. Uhrenunterbrechungen sind streng periodisch, Ein-/Ausgabeunterbrechungen sind meist schubartig und durch Benutzer ausgelöste Unterbrechungen sind meist sporadisch. Manche Unterbrechungen werden nur in Folge der Ausführung von Aktivitäten ausgelöst, diese sind dann nicht unabhängig. Wichtig sind die Auftretensmuster für die Echtzeitanalyse, da in ereignisgetriebenen Systemen die Systemlast von ihnen abhängt [3].

3.3 Die Ereigniskanäle

Die Ereigniskanäle spielen eine zentrale Rolle im Gesamtsystem, sämtliche Interaktion zwischen Komponenten wird über sie abgewickelt. Die Kommunikation über die Kanäle wird senderseitig initiiert. Eine Unterbrechungsbehandlung oder eine Aktivität erzeugt ein Ereignis, indem sie Daten auf einen Kanal schreibt. Diese werden über den Kanal in einen Puffer geschrieben. Auf der empfangenden Seite kann die Aktivität die Daten aus dem Puffer auslesen, sobald sie ausgeführt wird. Der Ablauf ist beispielhaft in Abbildung 3.3 dargestellt.

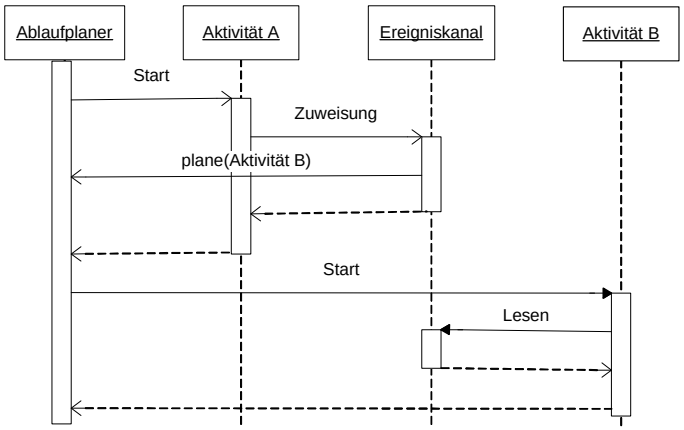


Abbildung 3.3: Kommunikationsablauf zwischen Aktivitäten

Die gerade in der Ausführung befindliche Aktivität A propagiert Daten/Ereignisse, indem sie einem Ereigniskanal Daten zuweist. Dieser veranlasst wiederum die Aufnahme der Aktivität B in die Ablaufplanung. Nachdem die Aktivität A ihre Ausführung beendet hat, wird die Aktivität B ausgeführt. Diese liest dann die zum Ereignis gehörenden Daten aus dem Puffer des Ereigniskanals. Das Lesen aus einem Puffer ist konzeptionell blockierungsfrei, da zum Ausführungszeitpunkt von B ein gültiger Wert in dem Puffer gespeichert ist. Somit ist im gesamten System kein aktives Warten auf Daten (Polling) notwendig.

Zu beachten ist, dass der Sender keine Rückmeldung darüber erhält, ob ein Ereignis erfolgreich verarbeitet wurde. Das muss entweder über einen separaten Rückkanal oder über spezielle Datentypen realisiert werden.

Die Ereigniskanalschnittstelle ist getypt, das heisst, Komponenten können dann miteinander verbunden werden, wenn der Typ der kommunizierten Daten von Sender und Empfänger gleich ist. Die Ereignisquelle (Aktivität oder Interrupthandler) benötigt daher kein Wissen über die nachfolgende Aktivität. Es besteht eine starke Analogie zu dem Entwurf in Hardware, bei dem zwei Elemente verbunden werden können, wenn sie elektrisch kompatibel sind. Dies garantiert auch in Software ein Höchstmaß an Wiederverwendbarkeit [26].

3.3.1 Ereigniskanalpuffer

Am Ende eines jeden Ereigniskanals wird ein Puffer benötigt, da die nachfolgende Aktivität asynchron ausgeführt wird. Der Zugriff auf einen Puffer ist stets atomar. Im Folgenden werden grundlegende Pufferarten beschrieben.

Einfacher Puffer

Dieser Puffer speichert genau einen Wert. Von der Semantik her ist im Puffer immer der aktuelle Wert. Wann immer dem Puffer ein Wert zugewiesen wird, wird die assoziierte Aktivität angestoßen.

Im Normalfall sollte ein Puffer abwechselnd beschrieben und ausgelesen werden. Jedoch kann es in manchen Anwendungen Überlastsituationen geben, in denen das nicht gewährleistet ist. Der Puffer kann in diesem Fall unterschiedlich reagieren. So kann er die assoziierte Aktivität nur anstoßen, wenn der Puffer seit dem letzten Beschreiben ausgelesen wurde. Das reduziert die Last im System, jedoch geht die Information verloren, wie oft ein Ereignis aufgetreten ist. Alternativ kann die Aktivität bei jedem Schreibvorgang getriggert werden. Der zuletzt geschriebene Wert wird in diesem Fall mehrfach gelesen und der zuvor geschriebene Wert geht verloren.

Die erste Variante ist sinnvoll bei der zustandsbasierten Steuerung über einen Schalter. Schalter haben die Eigenschaft zu prellen, wechseln also bei einer Betätigung mehrmals den Zustand, für die Steuerung ist jedoch nur die endgültige Schalterposition relevant. Daher kann der Verlust von Zwischenwerten in Kauf genommen werden.

Die zweite Variante kann z. B. bei der Aufnahme eines Audiosignals in Puls-Kode-Modulation sinnvoll sein. Bei dieser wird in diskreten Zeitschritten ein Signal abgetastet, wobei für den Hörer ein zeitlicher Fehler eher hörbar ist als ein falscher Pegel. Dies ist insbesondere so, da zeitlich nah beieinander liegende Pegel meist auch ähnliche Werte besitzen. Das heißt, ein Datenverlust kann eher in Kauf genommen werden als ein Ereignisverlust, da die zeitrichtige Wiedergabe entscheidend ist.

Begrenzter Puffer

Ein begrenzter Puffer kann n Werte speichern, welche nach dem Fifo-Prinzip geschrieben und gelesen werden. Der Puffer bringt für jeden gespeicherten Wert die mit ihm verbundene Aktivität genau einmal zur Ausführung. Sollte der Puffer voll sein, werden weitere Zuweisungen verworfen. Prinzipiell kann auch der Fifo-Puffer genau wie ein einfacher Puffer beliebige Datentypen speichern. Begrenzte Puffer können zum Puffern von schubartig generierten Daten verwendet werden.

Ein Problem ist, dass der Puffer den Speicherplatz für die Daten und deren Verwaltung bereitstellen muss. Werden viele Fifos genutzt, steigt der Speicherbedarf des Gesamtsystems dementsprechend stark an. Dies ist in den betrachteten Systemen ein kritischer Punkt, da der Speicher begrenzt ist. Zur Pufferdimensionierung in Datenflusssystemen gibt es bereits Arbeiten. In [106] wird z. B. dargelegt, wie die minimal benötigte Puffergröße in einem Datenflusssystem gefunden werden kann.

Warteschlange

Eine Warteschlange ist vom Konzept her unendlich, besitzt aber aus technischen Gründen eine endliche Länge. Sie arbeitet auf verkettbaren Elementen und benötigt daher nur einen Zeiger auf das erste Element. Warteschlangen können sehr lang werden, ohne dass

Speicherplatz verschwendet wird, insbesondere wenn in einem Ereignisfluss innerhalb eines Pfades an mehreren Stellen mehr als ein Datum gespeichert werden muss. Abbildung 3.4 zeigt beispielhaft einen solchen Graphen, umgesetzt mit Fifo-Puffern in Teil a.) und mit Hilfe von Warteschlangen in Teil b.). Bei dem Ansatz mit begrenzten Puffern muss jeder Puffer groß genug sein, um alle zu einem Zeitpunkt benötigten Elemente speichern zu können. Im Ansatz mit Warteschlangen, wird nur ein globaler Puffer benötigt, welcher alle zu einem Zeitpunkt benötigten Datenelemente speichern kann. Die Warteschlangen referenzieren lediglich die Elemente im globalen Puffer. Dadurch kann ein Datenelement zu unterschiedlichen Zeitpunkten von unterschiedlichen Warteschlangen referenziert werden. Durch den Einsatz von Warteschlangen kann somit der Speicherverbrauch reduziert werden.

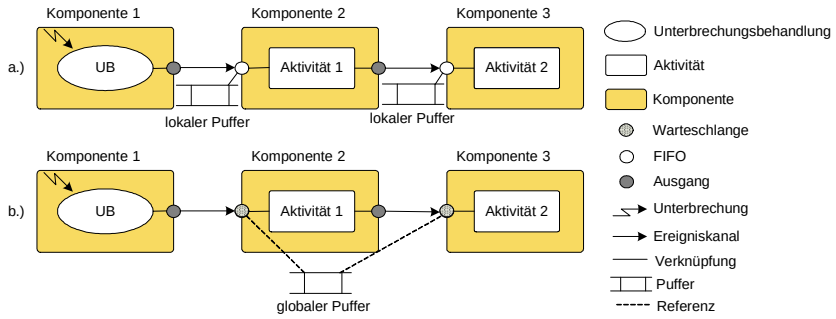


Abbildung 3.4: Begrenzte Puffer und Warteschlangen im Vergleich

Warteschlangen lösen noch ein weiteres Problem, welches bei begrenzten Puffern und dynamisch erzeugten Objekten in Überlastsituationen besteht. Wenn ein begrenzter Puffer voll ist und eine weitere Zuweisung vorgenommen wird, muss der Speicher des zugewiesenen Objekts wieder freigegeben werden. Das ist unter Umständen sehr aufwendig, da eventuell auch die Datenquelle für die Fehlerbehandlung darüber informiert werden muss. Mit Warteschlangen wird fehlender Pufferplatz bereits bei der Datenquelle erkannt, weil kein Datenobjekt erstellt werden kann. Das vereinfacht die Fehlerbehandlung.

Ereignispuffer

Einen Sonderfall bei den Ereigniskanälen stellt ein einfacher datenloser Ereignispuffer dar. Es handelt sich nicht um einen Puffer im eigentlichen Sinne, da keine Daten gespeichert werden. Er übernimmt jedoch wie die anderen Puffer das Anstoßen der assoziierten Aktivität, wenn ein Ereignis auftritt. Ein Ereignis ist in dem Fall ein Signal, welches dem Ereignispuffer angezeigt wird. Für jedes erhaltene Signal wird die assoziierte Aktivität genau einmal getriggert. Benutzt werden können solche Ereignisse z. B. für logische Taktsignale.

3.3.2 Vorverarbeitende Kanäle

Vorverarbeitende Kanäle reichen die Ereignisse nicht nur weiter, sondern arbeiten selbst auf den Daten. Solche Kanäle können miteinander oder mit einem Puffer verkettet werden. Vorverarbeitende Kanäle besitzen daher keinen Puffer und triggern keine Aktivität, da sie nicht die Endpunkte eines Ereigniskanals darstellen. Diese Kanäle sind sinnvoll, da sie helfen, die Anzahl der Planungsvorgänge zu reduzieren.

Da die Kanäle ihre Verarbeitung synchron zum Ereignis vornehmen und die Benutzung atomar ist, muss deren Komplexität beschränkt werden. Komplexe Funktionalität ist stets in Aktivitäten auszulagern, ansonsten können die Echtzeiteigenschaften einer Anwendung negativ beeinflusst werden.

Verteiler

In einem Ereignisfluss kann es Verzweigungen geben. Abbildung 3.5 zeigt eine Anwendung, in der ein AD-Wandler Werte aufnimmt, die von mehreren Komponenten verarbeitet werden. Eine Komponente konvertiert den Rohwert, die andere speichert diesen auf einem Datenträger. Für den AD-Wandler ist es dabei transparent, wie viele Komponenten den generierten Wert nutzen. Die Verteilung des Rohwertes wird von einem speziellen Kanal übernommen.

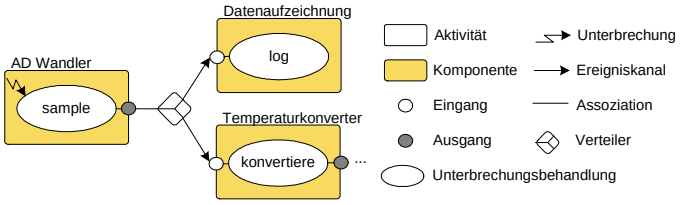


Abbildung 3.5: Verzweigung im Ereignisfluss

Der Vorteile ist, dass der AD-Wandler-Kode von der Anzahl der Empfänger unberührt bleibt. Zudem ist der Ansatz leichtgewichtiger als eine Variante, in der eine Aktivität die Verteilung der Daten übernimmt. In dieser würde zudem ein weiterer Ereignispuffer am Eingang der verteilenden Aktivität benötigt werden.

Filter

Ereigniskanäle können auch dazu verwendet werden, Ereignisse selektiv weiterzuleiten. Die mit dem Ereigniskanal assoziierte Aktivität wird dadurch nur bedingt getriggert und die Last im System kann durch die reduzierte Anzahl von Planungsvorgängen drastisch gesenkt werden.

Abbildung 3.6 zeigt eine Applikation, welche bei Überschreiten einer Temperatur einen Alarm auslösen soll. Die Komponenten AD-Wandler und die Temperaturkonverter werden wie im vorigen Beispiel benutzt. Nach der Temperaturkonversion leitet der Ereignis-

kanal nur Temperaturen über 85°C weiter. Die Aktivität, die den Alarm auslöst, wird nur beim Überschreiten der Temperaturschwelle angestoßen.

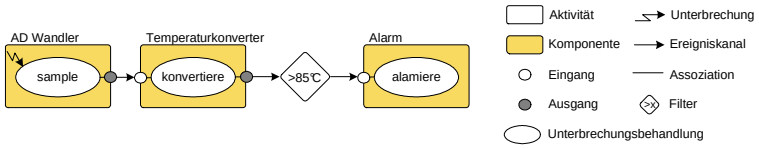


Abbildung 3.6: Filternder Ereigniskanal

Solche Filter können auch eingesetzt werden, um z. B. nicht relevante Bits aus einem Datum auszublenden oder um die Formatierung von Daten anzupassen. Filternde Kanäle reduzieren dann nicht nur die Last im System, sondern unterstützen auch die Wiederverwendbarkeit von Komponenten.

Taktteiler

Eine besondere Form eines Filters ist ein Taktteiler, dieser kann für datenlose Signale benutzt werden und leitet nur jedes n-te Signal weiter. Sinnvoll ist solch ein Taktteiler z. B. um aus einem Signal mit hoher Frequenz eins mit niedrigerer harmonischer Frequenz zu erzeugen.

In Abbildung 3.7 ist eine Uhrenimplementierung dargestellt, die eine Zeit in Sekunden zweistellig anzeigt. Die treibende Uhr läuft im Beispiel mit 37 KHz. Das Taktnetzwerk erzeugt daraus ein 1 Hz Signal und ein 0,1 Hz Signal.

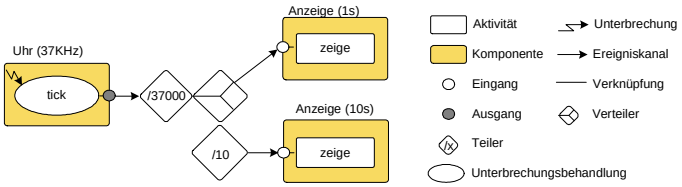


Abbildung 3.7: Einsatz eines Taktteilers

Der Vorteil ist offensichtlich, nur jeden 37000ten Takt ist ein Einplanung der Aktivität für die Sekundenanzeige notwendig. Außerdem kann durch solche Taktnetzwerke ein Zeitgeber mehrfach genutzt werden.

Globale Ereignisvariablen

In bestimmten Fällen kann es sinnvoll sein, nicht Daten zu propagieren, sondern nur die Information, dass sich Daten geändert haben. Das kann z. B. für globale Zustandsinformationen wie eine Routing-Tabelle genutzt werden. Der Vorteil ist, dass nicht bei mehreren Empfängern Daten kopiert und konsistent gehalten werden müssen. Die Lösung des Problems sind globale Ereignisvariablen, welche von der schreibenden Seite wie ein

Ereigniskanal benutzt werden, jedoch keine Aktivität anstoßen, sondern ein datenloses Ereignis generieren. Ähnliche Variablen werden auch in anderen Systemen benutzt, z. B. in TinyGals wo sie TinyGuys [22] genannt werden, diese senden jedoch keine Benachrichtigung an potenziell interessierte Komponenten. Sie werden als vorteilhaft angesehen für globale Informationen, die von mehreren Seiten gelesen aber nur von einer Seite beschrieben werden.

Abbildung 3.8 zeigt die Benutzung einer globalen Ereignisvariable. Sie speichert eine Temperaturtabelle, deren Werte durch verschiedene Komponenten benutzt werden. Diese sind wiederum nur an dem aktuellen Wert interessiert.

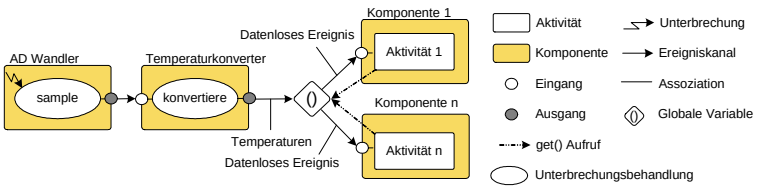


Abbildung 3.8: Darstellung einer synchronen Variable

Auf der schreibenden Seite verhält sich die globale Ereignisvariable wie ein datenbehafteter Ereigniskanal, dem Temperaturen zugewiesen werden können. Wurde die Tabelle aktualisiert, generiert die globale Ereignisvariable ein datenloses Ereignis, welches zu den angeschlossenen Komponenten propagiert wird. Das Lesen der Daten durch die getriggerten Aktivitäten erfolgt über den Aufruf der `get()`-Methode der globalen Ereignisvariable.

Eine globale Ereignisvariable enthält immer den aktuellen Wert. Eine Fifo- oder Queue-Semantik ist nicht sinnvoll, da es dadurch zu ungültigen Leseoperationen auf einem leeren Puffer kommen kann, was jedoch im Ereignisflussmodell nicht toleriert werden kann.

3.4 Die Komponenten

Komponenten sind in sich abgeschlossene Kodebereiche, die eine logisch zusammenhängende Aufgabe erfüllen, z. B. eine Zustandsmaschine. Komponenten verkürzen die Entwicklungszeit für Anwendungen, da grobgranulare Wiederverwendung für komplexe Anwendungsteile möglich wird. Eine Komponente interagiert ausschließlich über asynchrone Ereigniskanäle mit anderen Komponenten. Innerhalb einer Komponente gilt für die Aktivitäten eine Monitorsemantik, wodurch synchron auf Komponentenvariablen zugegriffen werden kann. Dieser Ansatz ist ähnlich dem von TinyGals [22], welcher für die Implementierung typischer Anwendungen geeignet zu sein scheint.

Die Monitorsemantik kann durch das Sperren der Unterbrechungen während der Ausführung einer Aktivität erweitert werden. Es ist jedoch im Einzelfall zu betrachten, ob für die gesamte Laufzeit einer Aktivität die Unterbrechungen gesperrt werden können.

Komponenten helfen vor allem bei der Strukturierung einer Anwendung. In einer Komponente können logisch zusammengehörige Teile zusammengefasst werden. Über die

standardisierten Ein- und Ausgänge kann eine Komponente in den Ereignisflussgraphen integriert werden.

Die Komponentenabstraktion unterstützt aber auch die Implementierung von zustandsbasierten Steuerungen mit mehreren Eingängen und Zuständen. In Abbildung 3.9 ist als Beispiel der Steuerungsautomat für ein bewegliches Solarpanel zu sehen. Das Panel bewegt sich von 8:00 Uhr morgens bis 8:00 Uhr abends stündlich für 20 Sekunden westwärts, um 10:00 Uhr abends dreht das Panel zurück auf die Ausgangsposition. Zusätzlich werden Endschalter genutzt, die ein Überdrehen des Panels verhindern sollen.

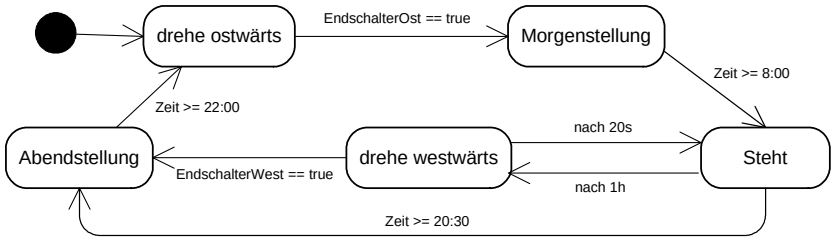


Abbildung 3.9: Zustandsautomat für einen beweglichen Solarkollektor

Abbildung 3.10 zeigt die Komponente, die für die Steuerung des Solarpanels genutzt wird. Es gibt zwei Aktivitäten, eine verarbeitet die Endschalterinformationen, die andere verarbeitet ein Zeitsignal. Beide Aktivitäten verändern bei Bedarf den Zustand der Komponente und schreiben neue Steuerinformationen für den Motor auf den Komponentenausgang.

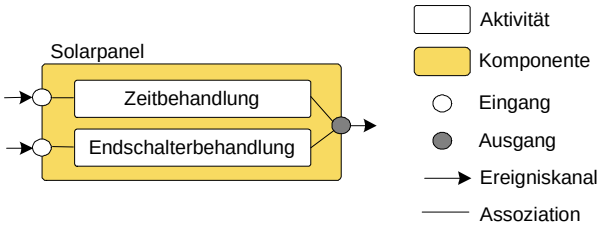


Abbildung 3.10: Ereignisflussgraph für einen beweglichen Solarkollektor

Mit Hilfe von Komponenten ist es möglich, den beiden Eingängen unterschiedliche Aktivitäten zuzuordnen und damit je nach Ereignis unterschiedliche Aktionen durchzuführen. Die Aktivitäten bearbeiten abhängig vom internen Zustand des Moduls das Ereignis, brauchen jedoch nicht zu evaluieren, welches Ereignis gerade aufgetreten ist. Im Beispiel bedeutet dies, dass in Folge einer Zeitänderung der Motor für die Bewegung angesteuert wird. Vollkommen parallel dazu kann es durch odometrische Fehler oder beim Zurückfahren zur Ursprungsposition dazu kommen, dass ein Endschalter erreicht wird. In diesem Fall wird durch eine extra Aktivität die Bewegung des Motors gestoppt. Zusätzlich wird eine Markierung gesetzt, die anzeigt, dass der Kollektor nun nur noch in die Gegenrichtung bewegt werden darf. Diese Markierung wird wiederum von der Ak-

tivität für den Bewegungsablauf benutzt und stellt somit einen gemeinsamen Zustand dar.

Die Komponentenabstraktion erlaubt die Zuordnung verschiedener Ereignisse zu verschiedenen Aktivitäten. Je nach Ereignis kann die entsprechende Reaktion erfolgen, ohne dass in den Aktivitäten evaluiert werden muss, welches Ereignis aufgetreten ist. Das wäre der Fall, wenn nur eine Aktivität alle Ereignisse für einen Zustandsautomaten behandelt. Zudem müsste dann entweder der Aktivität mitgeteilt werden, welches Ereignis aufgetreten ist, oder die Aktivität müsste die Eingänge abfragen. Beides ist ineffizient, da eine implizit vorhandene Information nicht genutzt wird.

3.5 Ablaufplanung

Im Ereignisflussmodell sind Aktivitäten die planbaren Einheiten. Das Modell legt jedoch nicht fest, nach welcher Strategie diese ausgeführt werden. Durch die Run-to-completion-Semantik der Aktivitäten kann ein einziger Stapel für die Abarbeitung der Aktivitäten genutzt werden. Im Folgenden wird gezeigt, wie verschiedene Ablaufplanungsverfahren für das Ereignisflussmodell umgesetzt werden können.

3.5.1 Abarbeitung mit einem Stapel allgemein

Die Ablaufplanung in den betrachteten Systemen erfordert ein Umdenken gegenüber leistungsstarken Computersystemen. Es werden keine Prozesse oder Threads abgearbeitet, welche einen eigenen Stapel und im Falle von schwergewichtigen Prozessen auch noch getrennte Adressräume besitzen. In tief eingebetteten Systemen steht dafür im Normalfall nicht genügend Speicher und auch keine Speicherverwaltungseinheit zur Verfügung.

Prinzipiell wird eine Ein-Stapel-Architektur dadurch ermöglicht, dass Aktivitäten immer komplett durchlaufen werden und konzeptionell blockierungsfrei sind. Daher ist die Ausführung einer Aktivität für den Ablaufplaner äquivalent zu einem Methodenaufruf. Der Vorteil bei dieser Vorgehensweise ist, dass die Notwendigkeit entfällt Platz für mehrere Stapel in dem knappen Speicher reservieren zu müssen, siehe Abbildung 3.11. Ein Ein-Stapel-System ist daher prädestiniert für den Einsatz in tief eingebetteten, stark speicherbeschränkten Systemen.

Die Unterbrechbarkeit von Aktivitäten durch andere Aktivitäten wird durch einen höheren Platzbedarf des Stapels erkauft, dieser ist jedoch wesentlich geringer als der für Thread-basierte Systeme. Zum einen muss nur auf dem einen systemweit genutzten Stapel Platz für Unterbrechungsbehandlungen bereitgehalten werden, zum anderen ist die maximale Stapeltiefe abhängig von der Anzahl der verwendeten Prioritätsstufen. Werden nur zwei Prioritätsstufen verwendet, können nur zwei Aktivitäten zu einem Zeitpunkt den Stapel nutzen. Die Aktivitäten einer Privilegstufe werden nach dem FIFO-Prinzip abgearbeitet.

In einem Thread-basierten System wird für jeden Thread genau ein Stapel benötigt. Auf jedem dieser Stapel muss sowohl Platz für die Aktivität selber, als auch für Un-

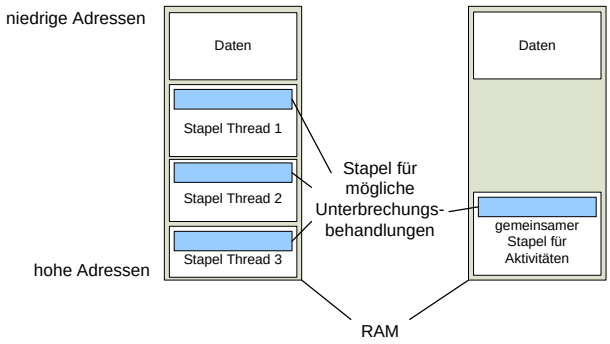


Abbildung 3.11: Speicherverbrauch RAM für Threads bzw. Aktivitäten

terbrechungen sein. Außerdem muss für jeden Thread extra ein Stapel dimensioniert werden.

3.5.2 Nicht-Präemptive Abarbeitung mit einem Stapel

Die nicht präemptive Abarbeitung mit einem Stapel ist trivial. Zu einem Zeitpunkt nutzt nur eine Aktivität den Stapel. Beendet sich die Aktivität, ist damit auch der Stapel wieder frei und die nächste Aktivität kann ausgeführt werden. Der Ablaufplaner wählt dazu eine Aktivität aus seiner Warteschlange.

Die Reihenfolge der Aktivitäten in der Warteschlange wird bestimmt durch das genutzte Abarbeitungsschema. Vorstellbar sind neben dem FCFS-Verfahren (First Come First Serve) auch prioritätsbasierte Verfahren, bei denen eine hochpriore Aktivität eine niederpriore auf der Ablaufplaner-Warteschlange verdrängen kann, jedoch nicht die aktuell laufende Aktivität unterbricht.

Weiterhin ist eine statisch zeitgesteuerte Abarbeitung möglich, bei der Aktivitäten nur ausgeführt werden, wenn Ereignisse für diese aufgetreten sind und ein bestimmter Zeitpunkt erreicht ist. Damit können auf einfache Art und Weise hart echtzeitfähige Systeme umgesetzt werden.

Im Ereignisflussmodell gilt, dass eine Aktivität genau einmal für jedes Ereignis ausgeführt wird. Da es durchaus dazu kommen kann, dass mehrere Ereignisse kurz hintereinander auftreten, muss eine Aktivität einen Zähler besitzen, der angibt wie oft sie noch ausgeführt werden muss. Ohne diesen Zähler müsste die Aktivität selbst durch Abfragen der Eingangspuffer feststellen, ob sie nochmals ausgeführt werden muss. Dieses Verfahren wird bei TinyOS angewandt.

3.5.3 Präemptive Abarbeitung mit einem Stapel

In einer Vielzahl von Veröffentlichungen für ereignisbasierte Systeme wird angenommen, dass diese nicht präemptiv sind [2, 11, 21, 85]. Die Systeme sind zwar meist so implementiert, es gibt allerdings keine grundlegende Notwendigkeit dafür, wie in [6, 7] gezeigt

wird. Das dort vorgestellte Protokoll MSRP (Minimal Stack Resource Policy) zeigt wie die präemptive Abarbeitung von Tasks mit nur einem Stapel umgesetzt werden kann.

Grundvoraussetzung für die präemptive Abarbeitung in einem Ein-Stapel-System ist das Vorhandensein von Prioritäten für die ausführbaren Elemente. Dies können fixe Prioritäten sein, wie im Falle der RMS-Ablaufplanung (Rate Monotonic Scheduling), aber auch dynamische Zeitschranken im Falle der EDF-Ablaufplanung (Earliest Deadline First).

Bei der präemptiven Abarbeitung befinden sich die Abarbeitungssrahmen von mehreren Aktivitäten gleichzeitig auf dem Stapel (Abbildung 3.12), jedoch nur von bereits vom Ablaufplaner aufgerufenen Aktivitäten. Diese Anordnung bedingt, dass unterbrochene Aktivitäten erst dann weiterlaufen können, wenn die unterbrechende Aktivität beendet ist. Das ist dadurch gegeben, dass sich die Priorität einer Aktivität nach deren Einplanung bis zum Ende der Ausführung nicht mehr ändert.

Der Platz zwischen den Rahmen der Aktivitäten auf dem Stapel wird für die Zuteilungsrahmen des Systems benötigt. Diese erlauben es, nachträglich eingeplante Aktivitäten zum richtigen Zeitpunkt zu starten.

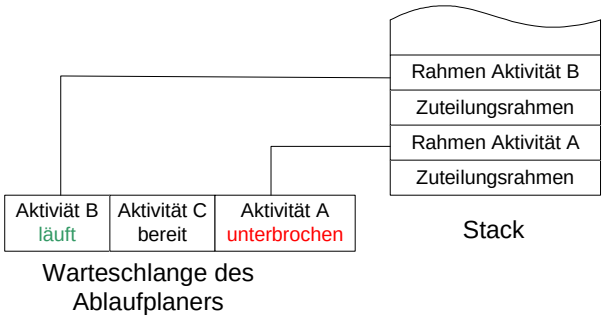


Abbildung 3.12: Präemptive Abarbeitung von Tasks

Im Beispiel 3.12 unterbricht Aktivität B die Aktivität A. Während der Ausführung von B tritt wiederum ein Ereignis für Aktivität C auf, welche dann eingeplant wird und aufgrund der höheren Priorität vor A in der Warteschlange des Ablaufplaners steht. Somit muss die Aktivität C gestartet werden, sobald B sich beendet. Das wird durch den eingefügten Zuteilungssrahmen erreicht. Diese werden sowohl bei allen Unterbrechungen als auch bei der Einplanung neuer Aktivitäten angelegt.

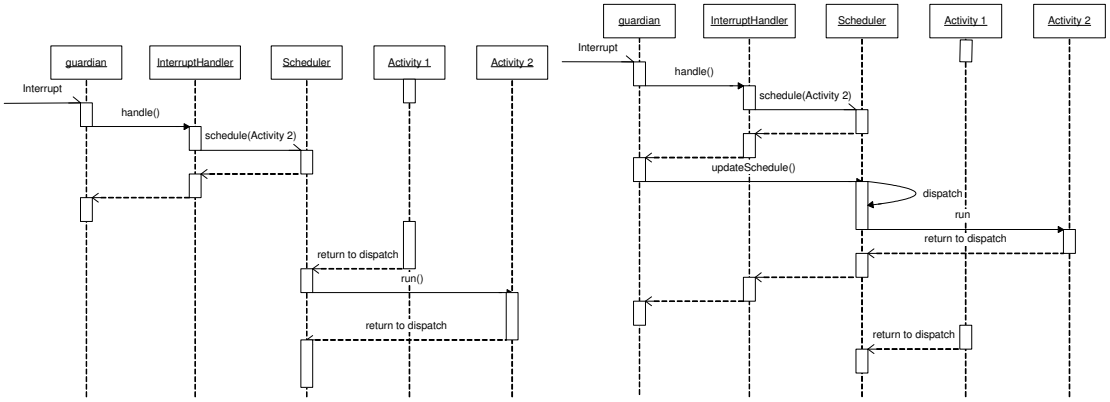
Systeminterne Abläufe

In Abbildung 3.13 ist der Ablauf einer nicht präemptiven und einer präemptiven Einplanung vergleichend dargestellt.

Im Teil a) ist der Ablauf eines nicht präemptiven Planungsvorganges dargestellt. Es ist zu sehen, wie Aktivität 1 durch ein Hardwareereignis unterbrochen wird, das Systemaktivitäten und die Planung der Aktivität 2 auslöst. Danach wird der Kontext der

Unterbrechung verlassen und Aktivität 1 läuft zu Ende. Nachfolgend wählt der Ablaufplaner die Aktivität 2 aus und startet diese.

Im Teil b) ist der Ablauf einer präemptiven Planung dargestellt. Bis zur Rückkehr von der Unterbrechung unterscheiden sich die Abläufe nicht. Danach findet jedoch keine Rückkehr zur Aktivität 1 statt, sondern der Ablaufplaner startet die Aktivität 2 sofort. Nach der Beendigung von Aktivität 2 kehrt der Kontrollfluss erst zum Ablaufplaner und dann zur unterbrochenen Aktivität 1 zurück.



(a) Nicht präemptive Abarbeitung zweier Aktivitäten

(b) Präemptive Abarbeitung zweier Aktivitäten

Abbildung 3.13: Vergleich der Abläufe bei nicht-präemptiver und präemptiver Aktivitätsabarbeitung

Wichtig bei diesem Verfahren ist, dass eine neu eingeplante hochpriore Aktivität erst nach dem Verlassen des letzten Kontextes, in dem eine Einplanung stattfinden kann, gestartet wird. Das ist notwendig, damit es im Falle von priorisierten Unterbrechungen nicht zu einer Prioritätsinversion der Aktivitäten und Unterbrechungsbehandlungen kommt. In Abbildung 3.14 ist eine solche dargestellt. Der Grund ist die präemptive Ausführung der Aktivität 1 direkt beim Planungsaufruf. Anfänglich tritt eine Unterbrechung mit geringer Priorität (Unterbrechung 1) auf. Während der Behandlung kommt es zu einer zweiten Unterbrechung mit höherer Priorität (Unterbrechung 2). Die zweite Unterbrechung löst wiederum ein Ereignis aus, in dessen Folge eine Aktivität eingeplant und sofort ausgeführt wird. Dadurch wird jedoch die zuvor angefangene Unterbrechungsbehandlung nicht beendet, und eine Aktivität unterbricht logisch gesehen eine Unterbrechungsbehandlung.

Solche Situationen ergeben sich auch, wenn innerhalb einer Unterbrechungsbehandlung mehrere Ereignisse ausgelöst werden oder die Planung einer hoch priorisierten Aktivität aus einer anderen Aktivität heraus unterbrochen wird.

Die Lösung für dieses Problem in präemptiven Systemen ist es, einen Zähler zu verwenden, in welchem die Zahl möglicher konkurrierender Planungsvorgänge vermerkt ist. Dieser Zähler wird immer dann inkrementiert, wenn eine Planung stattfindet oder eine Unterbrechung auftritt. Am Ende der Planung und nach Beendigung von Unterbre-

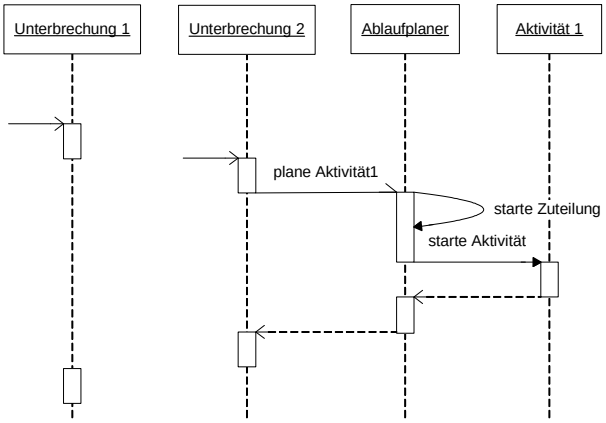


Abbildung 3.14: Prioritätsinversion bei präemptiven Systemen mit priorisierten Unterbrechungen

chungsbehandlungen wird der Zähler dekrementiert. Erreicht der Zähler wieder den Wert 0, wird vom System atomar die Zuteilung durchgeführt.

In nicht präemptiven Systemen gibt es das Problem der Prioritätsinversion prinzipiell nicht. Es kann zu keiner nebenläufigen Ausführung von Aktivitäten kommen, da diese ausschließlich in einer Endlosschleife des Systems aktiviert werden.

3.5.4 Verarbeitung lang laufender und blockierender Aktivitäten

Ein besonderes Problem ereignisgetriebener Systeme sind lang laufende und blockierende Funktionen [35, 65, 105]. Im Ereignisflussmodell können blockierende Aufrufe durch die Nutzung von Protothreads [34] dargestellt werden. Lang laufende Aktivitäten werden durch die präemptive Abarbeitung beherrschbar. So wird einer lang laufenden Aktivität einfach eine geringe Prioriät gegeben, dadurch kann diese von anderen Aktivitäten unterbrochen werden. Der Ansatz sorgt dafür, dass Aktivitäten nicht wie in Contiki und TinyOS kontextabhängig unterteilt werden müssen. Zudem werden unnötige Planungsvorgänge vermieden, da eine Aktivität nur einmal eingeplant und ausgeführt wird.

3.5.5 Diskussion

Die präemptive Abarbeitung mit einem Stapel verbindet in geeigneter Weise die Speichereffizienz der ereignisgetriebenen Abarbeitung mit den Möglichkeiten der Threadbasierten Programmierung.

Im Gegensatz zu anderen nicht präemptiven ereignisbasierten Systemen ist es möglich, lang laufende Aktivitäten auszuführen, ohne diese in mehrere getrennte Berechnungsschritte aufzuteilen. In [101] wird gezeigt wie die Aufteilung einer Schleife für TinyOS aussehen kann. Das Problem ist, dass die Aufteilung kontextabhängig ist und somit bei Benutzung der Komponente für die jeweilige Anwendung angepasst werden muss. Das

ist hinderlich für die Wiederverwendbarkeit von Komponenten. Bei präemptiver Abarbeitung im Ereignisflußmodell hingegen kann einer lang laufenden Aktivität einfach eine niedrige Priorität gegeben werden. Es ist zu beachten, dass dadurch auch die Anzahl von Planungsvorgängen reduziert wird, weil die betreffende Aktivität nur einmal pro Ereignis eingeplant wird.

Auch das Präzedenzproblem, welches in Thread-basierten Systemen auftritt, stellt sich trotz der präemptiven Abarbeitung nicht. Das ist auf die Erhaltung des rein ereignisbasierten Abarbeitungsschemas zurückzuführen und ist ein entscheidender Vorteil gegenüber den in [21, 36, 105] vorgeschlagenen Thread-basierten Verfahren. Solche Systeme können nicht entscheiden, ob ein Thread aktiv wartet oder normal arbeitet. Das wirkt sich auch auf das Energiemanagement aus, welches nicht implizit erfolgen kann. Der aktiv wartende Thread muss dem System mitteilen, dass er gerade blockiert ist. Das muss wiederum atomar mit der Überprüfung der Wartebedingung geschehen, damit das System nicht schlafen geht, obwohl die Wartebedingung gerade erfüllt wurde. Im Ereignisflussmodell muss eine Aktivität per Definition nicht aktiv warten, da sie nur ausgeführt wird, wenn alle Bedingungen erfüllt sind.

Die größte Einschränkung des Ereignisflussmodells ist die Run-to-completion-Semantik für ausführbare Einheiten. Diese Beschränkung lässt sich jedoch durch die Nutzung des Protothread Ansatzes [35] kompensieren. Außerdem vereinfacht die Run-to-completion-Semantik die Echtzeitanalyse [94, 98].

Eine weitere Einschränkung ist, dass Aktivitäten sich nicht zyklisch unterbrechen können. Das heißt, eine unterbrochene Aktivität kann nicht eine laufende unterbrechen, auch wenn die Priorität der unterbrochenen Aktivität dynamisch erhöht wird. Erst müssen die seit der Unterbrechung gestarteten Aktivitäten enden und den von ihnen genutzten Stapel dadurch wieder freigeben. Diese Einschränkung ist nur relevant in Systemen, in denen sich Prioritäten während der Ausführung einer Aktivität dynamisch ändern können. In allen anderen Fällen ist die Einschränkung sogar positiv, da unnötige Wechsel zwischen Aktivitäten verhindert werden [7]. Im Gegensatz dazu müssen unnötige Wechsel in Thread-basierten Systemen durch geeignete zusätzliche Maßnahmen [68, 80] verhindert werden.

3.6 Synchronisation

Die Synchronisation ist ein wesentlicher Aspekt bei der Entwicklung von Programmen. Sie sollte für den Anwendungsprogrammierer möglichst einfach sein, aber auch wenig Laufzeitaufwand und Mehraufwand beim Speicherverbrauch verursachen.

Die Abarbeitung eines Programmes erfolgt im Ereignisfluss mit einem Stapel, also in einem einzelnen Kontrollfluss. Demzufolge handelt es sich ausschließlich um Intraprozesssynchronisation [7]. Das führt dazu, dass nur blockierungsfreie Synchronisationsmechanismen wie z. B. Unterbrechungssperren verwendet werden können. Weiterhin muss beachtet werden, inwieweit die Synchronisation vom gewählten Abarbeitungssche-

ma abhängt. Das Ziel ist es, den Anwendungskode unabhängig vom gewählten Abarbeitungsschema zu halten.

Die Stellen, an denen synchronisiert werden muss, sind im Ereignisflussmodell durch den hierarchischen GALS-Aufbau (global asynchron lokal synchron) [23] vorgegeben. Zudem kann der Anwendungskode bis auf wenige Ausnahmen frei von Synchronisationsanweisung gehalten werden. Im Folgenden wird diskutiert, wo und wie im Ereignisflussmodell synchronisiert werden muss.

3.6.1 Synchronisation der Ereigniskanäle

Ereigniskanäle sind die Verbindung zwischen Komponenten, über sie werden Ereignisse ausgetauscht, die asynchron verarbeitet werden. Da verschiedene Komponenten auf die Kanäle lesend und schreibend zugreifen, muss der Zugriff synchronisiert erfolgen.

In Abbildung 3.15 sind verschiedene Szenarien gezeigt, wie Komponenten miteinander gekoppelt sein können. Es ist zu beachten, dass Ereigniskanäle erst die Daten puffern und dann die zugehörige Aktivität anstoßen. Dadurch ist für alle Abarbeitungsverfahren sichergestellt, dass auch Daten im Puffer stehen, wenn die angestoßene Aktivität ausgeführt wird . Eine Leseoperation kann somit keinen Schreibzugriff unterbrechen.

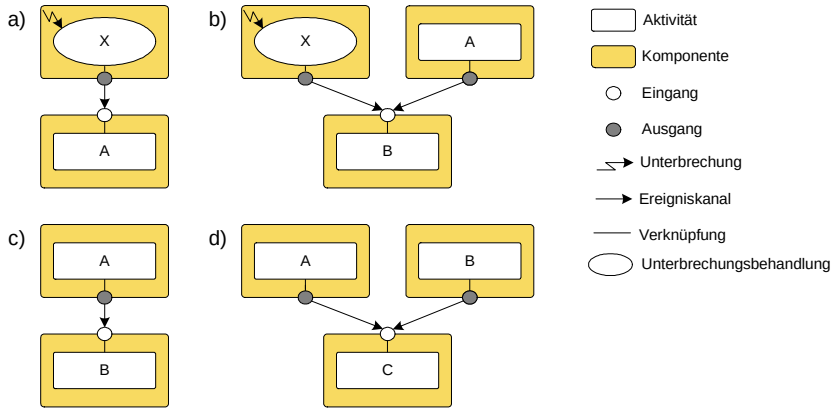


Abbildung 3.15: Szenarien zur Synchronisation im Ereignisflussmodell

In Beispiel a) wird während einer Unterbrechungsbehandlung auf den Ereigniskanal geschrieben. Diese Schreiboperation muss nicht geschützt werden, da die Unterbrechung nicht durch sich selbst oder die lesende Aktivität unterbrochen werden kann. Die Leseoperation hingegen kann durch eine Schreiboperation unterbrochen werden, da die Schreiboperation während einer Unterbrechungsbehandlung durchgeführt wird. Unter Umständen kann auch darauf verzichtet werden. Ist die Unterbrechung streng periodisch und die Aktivität wird stets innerhalb der Periodendauer der Unterbrechung ausgeführt, so ist keine Synchronisation notwendig.

In Beispiel b) muss zusätzlich zur Leseoperation das Schreiben von Aktivität A auf den Ereigniskanal geschützt werden. Der Grund ist, dass auch während der Behandlung

der Unterbrechung X der Ereigniskanal beschrieben werden kann. Da in diesem Fall die Aktivität A und die Unterbrechung X unabhängig voneinander sind, kann nicht davon ausgegangen werden, dass Aktivität A immer zwischen zwei Unterbrechungen von X ausgeführt wird.

In Beispiel c) ist es durch die Arbeitsweise der Ereigniskanäle unmöglich, dass die Leseoperation von Aktivität B die Schreiboperation unterbricht. Das ist selbst dann der Fall, wenn Aktivität B eine höhere Priorität als Aktivität A hat. Die Leseoperation hingegen kann durch eine Schreiboperation unterbrochen werden, wenn Aktivität A eine höhere Priorität als Aktivität B hat und präemptive Abarbeitung vorliegt. Unter Umständen kann durch eine Echtzeitanalyse dieser Fall ausgeschlossen werden.

In Beispiel d) ergeben sich vielfältige Möglichkeiten für die Prioritätsverteilung. Der Lesezugriff von Aktivität C muss dann geschützt werden, wenn Aktivität A oder B eine höhere Priorität haben und präemptive Abarbeitung vorliegt. Der Schreibzugriff einer Aktivität muss geschützt werden, wenn die jeweils andere schreibende Aktivität eine höhere Priorität hat.

Die bisherigen Beispiele zeigen bereits, dass die Wahl des Abarbeitungsschemas Einfluss auf die Synchronisation hat. Im Folgenden werden die Auswirkungen in komplexeren Applikationen gezeigt. Abbildung 3.16 zeigt beispielhaft einen Ereignisflussgraphen. In der Tabelle 3.1 ist für das Beispiel angegeben, welche Kanäle potenziell bei präemptiver und nicht präemptiver Abarbeitung geschützt werden müssen. Es werden im präemptiven Fall keine weiteren Annahmen über die Verteilung der Prioritäten getroffen. So wird die maximal notwendige Anzahl an Synchronisationspunkten deutlich.

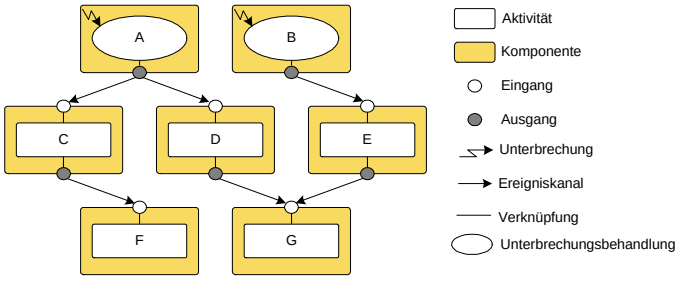


Abbildung 3.16: Anwendungsbeispiel mit Kennzeichnung aller Synchronisationspunkte

Die Zahl der zu synchronisierenden Operationen auf den Ereigniskanälen hängt stark davon ab, ob eine präemptive oder nicht präemptive Abarbeitung verwendet wird. Im präemptiven Fall müssen alle Leseoperationen geschützt werden, wenn nicht durch Analyse sichergestellt werden kann, dass dies nicht notwendig ist. Zudem müssen auch Schreibzugriffe geschützt werden, wenn mehrere Komponenten auf einen Ereigniskanalpuffer schreiben.

Allgemein hängt der Synchronisationsaufwand auch von den gewählten Prioritäten ab. Wird die Anzahl der verwendeten Prioritätsstufen in einer Anwendung minimiert, senkt das sowohl den Synchronisationsaufwand als auch den benötigten Speicherplatz

Puffer	nicht präemptive Abarbeitung	präemptive Abarbeitung
A lesen	X	X
A schreiben	-	-
B lesen	X	X
B schreiben	-	-
C lesen	-	X
C schreiben	-	-
D lesen	-	X
D schreiben	-	-
E lesen	-	X
E schreiben	-	-
F lesen	-	X
F schreiben	-	-
G lesen	-	X
G schreiben	-	X
Gesamt	2	12

Tabelle 3.1: Synchronisationspunkte bei nicht-präemptiver und präemptiver Abarbeitung

für den Stapel (siehe Abschnitt 3.5.3). Potenziell ergibt sich hier auch die Möglichkeit einer Werkzeugunterstützung, da die Synchronisationspunkte im Ereignisflussgraphen automatisch ermittelt werden können.

Für den Anwendungsprogrammierer erfolgt die Synchronisation in den Ereigniskanälen implizit. Das heisst, der funktionale Kode der Aktivitäten ist frei von Synchronisationsanweisungen.

Mögliche Synchronisationstechniken

Prinzipiell muss die potenziell unterbrechende Einheit (Aktivität oder Unterbrechungsbehandlung) bei einer zu schützenden Operationen von der Ausführung abgehalten werden. Dazu sind durch den Ein-Stapel-Ansatz ausschließlich Mechanismen zur Intraprozesssynchronisation erlaubt. Die einfachste Möglichkeit ist das globale Sperren aller Unterbrechungen [28]. Zudem sind blockierungsfreie Verfahren, die auf atomaren Compare-And-Swap-Instruktionen beruhen [52], aufgrund fehlender Maschinenbefehle nicht auf allen Mikrocontrollern umsetzbar.

Das Sperren der Unterbrechungen hat zwar systemweite Auswirkungen, die zu schützenden Operationen auf den Ereigniskanälen sind jedoch von kurzer Dauer. Zudem wird nach dem Puffern eine Aktivität angestoßen, wobei auf zentrale Datenstrukturen des Ablaufplaners zugegriffen wird. Dieser Vorgang muss wiederum atomar zur Einplanung erfolgen, wofür sich wiederum das Sperren der Unterbrechungen anbietet.

Die Synchronisation kann als ein Sonderfall des Priority Inheritance Protocol [93] angesehen werden, das Prioritätsinversion ausschließt. Beim Priority Inheritance Protocol bestimmt sich die Priorität, die ein Prozess bei dem Zugriff auf eine Ressource annehmen muss, durch die höchste Priorität eines anderen Prozesses, der potenziell auf die gleiche Ressource zugreifen will. Durch die Kopplung von Datenpufferung und Ablaufplanung im Ereignisflussmodell muss deshalb eine Aktivität bei dem Anstoßvorgang die höchste Priorität annehmen. Auch das kann wieder durch das globale Sperren der Unterbrechungen erreicht werden.

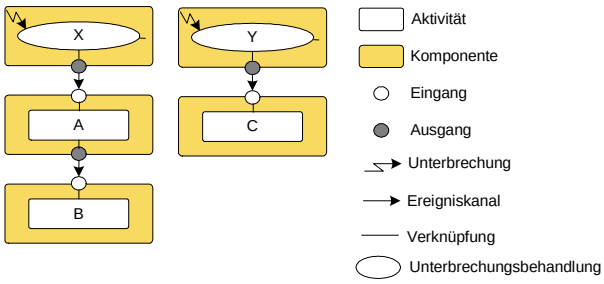


Abbildung 3.17: Beispielszenario für Prioritätsinversion durch partielle Sperren

Das partielles Sperren der Unterbrechungen bei Pufferzugriff nicht reicht, zeigt das Beispiel in Abbildung 3.17. In Abbildung 3.18 ist das Verlaufsdiagramm für eine Prioritätsinversion in der Beispielanwendung zu sehen. Die Aktivität B hat im Beispiel eine höhere Priorität als Aktivität A. Angenommen A liest von dem Ereigniskanal, und sperrt zum Schutz nur die Unterbrechung X. Wird dann während des Lesens die Unterbrechung Y ausgelöst, kommt es zur Prioritätsinversion. Die Unterbrechung Y triggert die Aktivität B, welche aufgrund der höheren Priorität gegenüber A sofort ausgeführt wird. Da die Leseoperation von A noch nicht abgeschlossen ist, bleibt die Unterbrechung X während der Ausführung von B unzulässigerweise gesperrt.

3.6.2 Synchronisation innerhalb von Komponenten

Im vorherigen Abschnitt wurde die Synchronisation bei der Kommunikation zwischen Komponenten über Ereigniskanäle betrachtet. Diese Art der Synchronisation entspricht der des Datenflussmodells. Einige Anwendungen lassen sich jedoch nur unzureichend oder umständlich allein mit den Mitteln des Datenflussmodells darstellen. Daher wurden Komponenten eingeführt, in denen es zu Nebenläufigkeit kommen kann.

Innerhalb von Komponenten können Aktivitäten und Unterbrechungsbehandlungsroutinen synchron auf Variablen und Methoden zugreifen. Dieser Zugriff muss jedoch geschützt erfolgen. Es werden dabei zwei mögliche Konfliktsituationen unterschieden, wie in Abbildung 3.19 zu sehen ist. Im Teil a) ist der nebenläufige Zugriff aus Aktivitäten heraus zu sehen, im Teil b) der nebenläufige Zugriff aus einer Aktivität und einer Unterbrechungsbehandlungsroutine. Im Folgenden wird gezeigt wie diese Konfliktsituationen verhindert werden können.

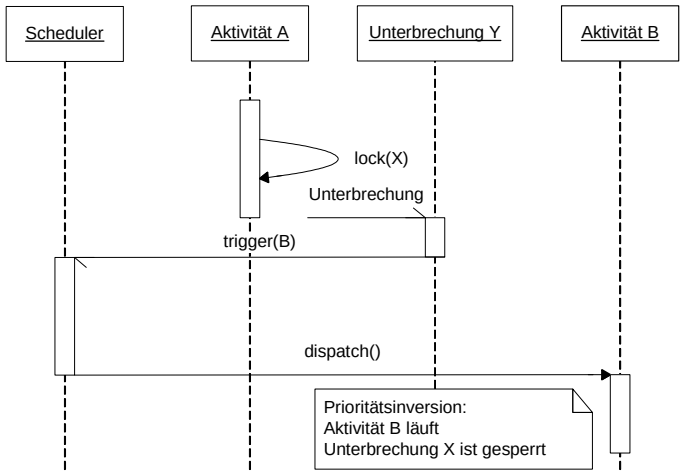


Abbildung 3.18: Beispiel für Prioritätsinversion durch partielles Sperren von Unterbrechungen

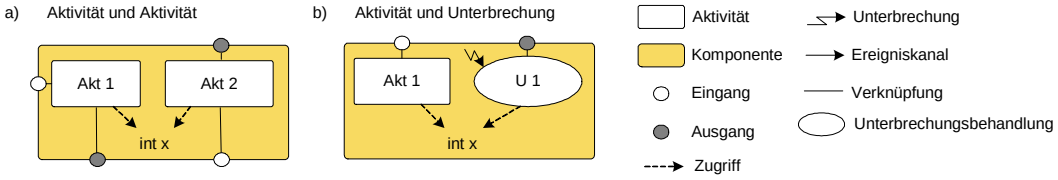


Abbildung 3.19: Synchronisationsszenarien innerhalb von Komponenten

Synchronisation zwischen Aktivitäten

Gilt für eine Komponente die Monitorsemantik, so finden keine nebenläufigen Zugriffe auf Variablen oder Methoden statt. Für Aktivitäten untereinander ist die Monitorsemantik gewahrt, wenn alle Aktivitäten einer Komponente nach dem Fifo-Prinzip ausgeführt werden. Für alle nicht präemptiven Abarbeitungsverfahren ist letzteres implizit gegeben.

Für die Planung nach festen Prioritäten ist die Fifo-Ordnung hergestellt, wenn alle Aktivitäten einer Komponente dieselbe Priorität bekommen. Dieses Verhalten gilt nur für Aktivitäten einer Komponente, Aktivitäten aus unterschiedlichen Komponenten können sich weiterhin unterbrechen. Bei der Abarbeitung nach dem EDF-Prinzip muss sichergestellt werden, dass die Zeitschranke (Deadline) für eine später geplante Aktivität auch später als die für eine bereits eingeplante Aktivität der Komponente ist. Da sich die Zeitschranke aus dem Zeitpunkt der Planung und der maximalen Reaktionszeit (Response Time) für diese Aktivität berechnet, muss die maximale Reaktionszeit für alle Aktivitäten einer Komponente gleich sein. Abbildung 3.20 zeigt das Verhalten an einem Beispiel. Da die Reaktionszeiten `Rx` und `Ry` für zwei unterschiedliche Aktivitäten gleich sind, besitzt die Zeitschranke der zuerst eingeplanten Aktivität den kleineren Wert.

Werden Prioritäten oder die maximal erlaubten Reaktionszeiten nicht wie beschrieben

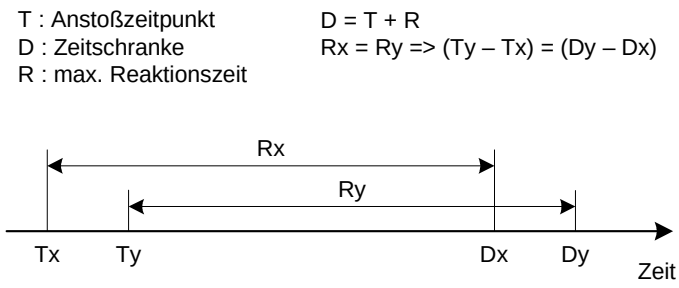


Abbildung 3.20: Sequentialsierung von Aktivitäten bei EDF-Abarbeitung

gewählt, muss eine feingranulare Synchronisation beim Zugriff auf interne Variablen der Komponente erfolgen. Diese feingranulare Synchronisation wird dadurch begünstigt, dass nur innerhalb einer Komponente auf diese Variablen zugegriffen werden kann. Zudem ist eine Komponente im Allgemeinen nicht sehr komplex.

Synchronisation zwischen Aktivität und Unterbrechungen

Für den Schutz gegen einen nebenläufigen Zugriff aus einer Unterbrechungsbehandlung heraus, muss die Unterbrechung während der Ausführung eines kritischen Abschnitts einer Aktivität gesperrt werden. Die Synchronisation erfolgt somit feingranular für den Zugriff auf die interne Variable der Komponente. Da Komponenten im Allgemeinen kompakt sind, scheint es möglich, dies automatisch durch ein Werkzeug vorzunehmen. Ähnliches wurde schon bei der Verwendung von GalsC [23] getan. Zudem ist diese feingranulare Synchronisation nur in Treiberkomponenten notwendig. Das heißt, der gewöhnliche Anwendungsprogrammierer, der die Treiber nur nutzt, kommt mit der Synchronisation zwischen Aktivitäten und Unterbrechungen nicht in Berührung.

3.6.3 Asynchrone Ausgaben

Wie Softwarekomponenten arbeiten einige Bausteine eines Mikrocontrollers ereignisgetrieben. Zum Beispiel kann eine Analog-Digital-Wandlung in Software angestossen werden. Ist diese fertig, wird dies über eine Unterbrechung angezeigt. Das Problem ist, dass nicht mehrere Aufträge gleichzeitig von dem AD-Wandler verarbeitet oder serialisiert werden können. In ereignisbasierten Systemen führt das zum sogenannten State Constraint Problem [55]. Die ansteuernde Softwareeinheit muss sich zwar nach der Ansteuerung des Bausteins beenden, darf aber nicht erneut ausgeführt werden, solange der Hardware-Baustein seine Arbeit nicht beendet hat.

In Abbildung 3.21 wird gezeigt, wie dieses Problem konzeptionell gelöst werden kann. Als Beispiel dient eine serielle Schnittstelle, die Pakete von der Anwendung erhält und nach dem Senden der Daten eine Unterbrechung generiert. Die Unterbrechungsbehandlung zeigt über ein Signal das Ende der Verarbeitung an. Dieses Signal und von der

Anwendung kommende Pakete werden auf Eingänge geschrieben, die beide mit der Aktivität `starte senden` verknüpft sind.

Abbildung 3.21: Konzeptionelle Lösung für das State-Constraint-Problem

Durch die UND-Semantik der Eingänge wird die Aktivität immer dann getriggert, wenn Daten anliegen und gerade keine Daten gesendet werden. Der Eingang für die Benachrichtigung, dass ein Paket gesendet wurde, muss initial einmal beschrieben werden, damit die Aktivität jemals angestoßen wird.

Unabhängig von der konkreten Umsetzung dieser Semantik ist der Anwendungskode weiterhin frei von Synchronisationsanweisungen. Das wird durch die im Ereignisfluss vorgenommene Verknüpfung von ereignisbasierter Abarbeitung und der Abstraktion eines Datenflusses möglich. Wäre das nicht der Fall, müsste die Aktivität bei jeder Sendeunterbrechung und jedem einkommenden Paket aufgerufen werden und prüfen, ob Daten versendet werden können.

3.7 Echtzeitunterstützung

Damit ein System hart echtzeitfähig ist, muss es deterministisch arbeiten. Das ermöglicht eine statische Analyse über die Einhaltung von allen geforderten Echtzeitbedingungen vor dem Einsatz des Systems. Das Ereignisflussmodell hilft bei dieser Analyse. Durch die Verwendung von Komponenten und verbindenden Ereigniskanäle ist die Struktur einer Anwendung bekannt. Zudem haben die Aktivitäten durch die Run-to-completion-Semantik definierte Start- und Endpunkte für jede Ausführung. Diese Eigenschaften werden z. B. in [97] an ein analysierbares System gestellt. Zusätzlich ist die Kopplung zwischen ausführbaren Einheiten bekannt und durch die reaktive Natur von Steuerungssystemen gerichtet [1]. Daher ist es auch allgemein möglich, Bearbeitungszeiten für Pfade zu untersuchen.

3.7.1 Abstraktionsebenen der Echtzeitanalyse

Der Ansatz zur Echtzeitanalyse im Ereignisfluss ist zweigeteilt. Zuerst werden die Laufzeiten der einzelnen Elemente (Aktivitäten, Unterbrechungsbehandlungen) und Systemvorgänge (Ablaufplanung, allgemeine Unterbrechungsbehandlung) getrennt voneinander ermittelt. Diese Zeiten werden dann in einer globalen Analyse auf dem Ereignisflussgraphen einer Anwendung benutzt. Dabei kommt zum Tragen, dass jegliche Bearbeitung in einer Anwendung initial von einer Unterbrechung ausgeht und aus der gegebenen

Strukturinformation hervorgeht, in welcher Folge weitere Aktivitäten ausgeführt werden können. Mit dieser gegebenen Strukturinformation ist es möglich, eine qualitativ hochwertige Echtzeitanalyse durchzuführen.

In Abbildung 3.22 ist zu sehen, wie ein Ereignisflussgraph für die globale Echtzeitanalyse annotiert werden kann.

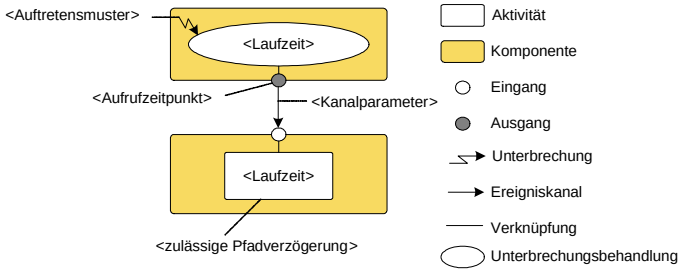


Abbildung 3.22: Ein für die Echtzeitanalyse annotierter Ereignisflussgraph

Die Annotationen sind in spitzen Klammern angegeben. Für die Unterbrechungen ist spezifiziert, in welchem Muster (streng periodisch, schubartig, ...) sie auftreten. Für die Ereigniskanäle wird eine Parametriesierung z. B. mit dem Teilerwert eines Taktteilers vorgenommen. Die Annotation des Takteilers gehört dabei ohnehin zum Entwicklungsprozess der Anwendung. Nur die Spezifikation der Unterbrechungen muss extra für die Echtzeitanalyse erfolgen.

Die Laufzeiten für die Aktivitäten und Ereigniskanäle geben an, wie lange diese minimal oder maximal brauchen, zudem wird kenntlich gemacht, wann während der Ausführung ein Ausgang benutzt wird. Die Laufzeiten können größtenteils automatisch ermittelt werden, da der Kode der Aktivitäten meist nicht sehr komplex ist.

Es ist zu beachten, dass der Anwendungskode ohne Annotationen auskommt. Dies ist vorteilhaft gegenüber Echtzeitsprachen wie Timber [13], bei denen der Quellkode stark mit Echtzeitannotationen durchsetzt ist.

Auf dem annotierten Ereignisflussgraphen kann die globale Echtzeitanalyse erfolgen. Diese berücksichtigt unter anderem das gewählte Ablaufplanungsverfahren. Die Ergebnisse können dann auch auf Ebene des Ereignisflussgraphen dargestellt werden, wie Abbildung 3.23 zeigt.

In dem Beispiel sind die Netzwerkschichten einer Sensornetzapplikation zu sehen. In den Aktivitäten und Unterbrechungsbehandlungsobjekten stehen die ermittelten maximalen Laufzeiten für eine Ausführung. Der Vorteil ist die direkte Darstellung der Ergebnisse auf der Abstraktionsebene, in der auch das System komponiert wird. Das macht es leicht, eventuelle Flaschenhälse oder Probleme zu erkennen. Im Beispiel wird ein TDMA-Medienzugriffsverfahren (Time Division Multiplexed Access) genutzt. Hat dieses eine maximal erlaubte Verzögerung von 1 ms, dann steht dies im Konflikt mit der Ausführungzeit der Kompressionstufe. Es muss also ein präemptives Abarbeitungsschema gewählt werden. In dem Ereignisflussgraphen können auch Bearbeitungszeiten für Pfade angegeben werden. Beispielhaft ist das für den kritischen Pfad von Zeitun-

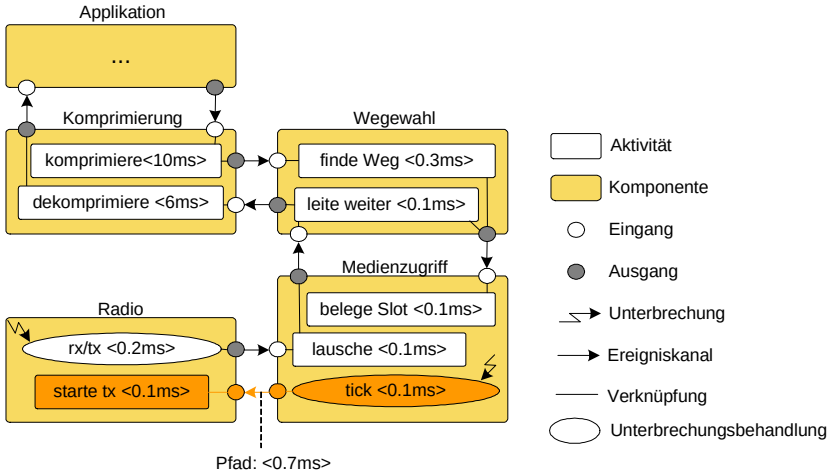


Abbildung 3.23: Darstellung von Echtzeitanalyseergebnissen im Ereignisflussgraph

terbrechung für das TDMA-Modul bis zum abgeschlossenen Start der Übertragung zu sehen. Es ist zu beachten, dass die Pfadverzögerung nicht einfach nur die Summe der Ausführungszeiten der enthaltenen Unterbrechungsbehandlung und der Aktivitäten ist. Die Verzögerung wird auch durch andere Unterbrechungen, Aktivitäten beeinflusst.

Die Darstellung der Ergebnisse der Echtzeitanalyse erfolgt direkt in der Modellebene und ist daher gut vom Programmierer auszuwerten. Eine solche Darstellung wird z. B. in [51] als unabdingbar für die Entwicklung robuster echtzeitfähiger Systeme angesehen.

3.7.2 Kriterien zur Echtzeitanalysierbarkeit

Neben der für den Entwickler günstigen Darstellung muss ein hart echtzeitfähiges System auch analysierbar sein. Entscheidende Kriterien für die Analysierbarkeit kritischer echtzeitfähiger Ada-Anwendungen finden sich zum Beispiel im Ravenscar-Profil [15]. Die Kriterien umfassen Eigenschaften, die ein hart echtzeitfähiges Programm aufweisen muss und welche Sprachkonstrukte es nutzen darf. Das Ravenscar-Profil wird zum Beispiel von der NASA für die Programmierung von Raketensoftware genutzt. Im OSEK Standard [79] werden solche Kriterien für OSEK-Systeme festgelegt. Ein wesentlicher Ansatz ist, dass Quellen für Nichtdeterminismus, wie z. B. die Möglichkeit einer Prioritätsinversion, von vornherein ausgeschlossen werden. Andere Systeme wie ECOS [72] oder QNX [81] beschränken ihre Echtzeitfähigkeit darauf, dass eine präemptive Abarbeitung möglich ist. Das ist aber weder eine notwendige noch hinreichende Bedingung für die Echtzeitfähigkeit eines Systems.

In [70] werden die folgenden Anforderungen als die wichtigsten des Ravenscar-Profils identifiziert.

1. Tasks werden statisch mit einer fixen Priorität (zur Vermeidung von Requeue Vorgängen) erzeugt

2. Die Task-Objekte bestehen für die gesamte Laufzeit
3. Task-Hierarchien (Tasks in Tasks) sind nicht erlaubt
4. Geschützte Objekte ermöglichen asynchrone Inter-Prozess-Kommunikation mit Monitorsemantik
5. Die Inter-Prozess-Kommunikation ist blockierungsfrei und verhindert Prioritätsinversion
6. Die Implementierung periodischer Prozesse ist möglich
7. Sporadische Prozessauslösungen durch Unterbrechungen sind möglich

All diese Anforderungen können im Ereignisflussmodell wie folgt erfüllt werden:
(1. und 2.) Tasks werden im Ereignisfluss durch Aktivitäten (Abschnitt 3.1) repräsentiert. Diese werden bei der Systeminitialisierung erstellt und bestehen für die gesamte Laufzeit des Systems. Bis auf die EDF-Ablaufplaner, sind die Prioritäten statisch. Aber selbst bei der EDF-Ablaufplanung kann es nicht zu einer Neueinordnung (Requeue) einer bereits eingeplanten Aktivität in der Warteschlange des Ablaufplaners kommen.
(3.) Alle Aktivitäten im System werden direkt vom Ablaufplaner (Abschnitt 3.5) verwaltet, es gibt daher keine Aktivitätshierarchie.
(4. und 5.) Die Ereigniskanalpuffer (Abschnitt 3.3) sind geschützt (Monitorsemantik) und erlauben die blockierungsfreie asynchrone Kommunikation von Aktivitäten und verhindern eine Prioritätsinversion.
(6.) Periodische Aktivitäten können durch Nutzung eines Takteinganges an einer Komponente implementiert werden. Zudem ist es durch die UND-Verknüpfung zweier Eingänge möglich, eine Aktivität in Abhängigkeit eines Taktsignals und dem Vorliegen von Daten auszuführen.
(7.) Die sporadische Prozessauslösung durch Unterbrechungen entspricht dem ereignisbasierten Abarbeitungsmodell.

Prinzipiell erfüllt das Ereignisflussmodell das Ravenscar-Profil und erlaubt somit die Modellierung analysierbarer Anwendungen. Das erlaubt die statische Echtzeitanalyse und folglich die Implementierung hart echtzeitfähiger Systeme.

4 Das Reflex System

Reflex (**R**eal-Time **E**vent **FL**ow **EX**ecutive) ist ein Betriebssystem, welches die grundlegenden Abstraktionen des Ereignisflussmodells implementiert. In Abbildung 4.1 ist das Architekturdiagramm für Reflex dargestellt. Die Implementierung teilt sich in drei Schichten, die unterste Schicht bildet der Betriebssystemkern, welcher die grundlegenden Mechanismen für Unterbrechungsbehandlungen, Energiemanagement und Ablaufplanung implementiert. Darauf aufbauend existiert eine Bibliothek, welche die Ereignisflusselemente bereitstellt. Der Anwendungskode benutzt ausschließlich diese zur Verfügung gestellten Abstraktionen. Eine Besonderheit sind Treiber, diese unterscheiden sich von dem normalen Anwendungsteil dahingehend, dass sie auch die Unterbrechungen direkt beeinflussen können.

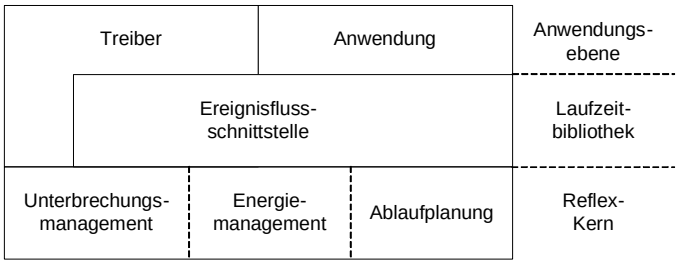


Abbildung 4.1: Die Architektur von Reflex

Im Folgenden wird die Umsetzung von den im Ereignisflussmodell genutzten Abstraktionen gezeigt. Die Ablaufplanung und das Energiemanagement werden in den Kapiteln 5 und 6 aufgrund ihrer Komplexität einzeln betrachtet.

4.1 Die Aktivitäten

Aktivitäten sind die planbaren Einheiten in einer Anwendung. In Reflex werden sie durch Objekte repräsentiert, welche vom jeweils verwendeten Ablaufplaner verwaltet werden (siehe Kapitel 5). Die allgemeine Aktivitäts-Schnittstelle wird in Abbildung 4.2 gezeigt.

Die `run()`-Methode: Diese Methode wird vom Ablaufplaner zur Aktivierung aufgerufen. Die Implementierung ist spezifisch für jede Aktivitätsklasse, daher ist die Methode abstrakt. Die Methode darf nicht blockieren, um die Run-to-completion-Semantik nicht zu verletzen.

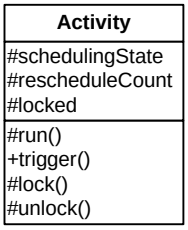


Abbildung 4.2: Aktivitätsabstraktion in Reflex

Die `trigger()`-Methode: Für jeden Aufruf der `trigger`-Methode wird die Aktivität genau einmal zur Abarbeitung eingeplant. Der Aufruf erfolgt durch einen Ereigniskanalpuffer. Wenn sich die Aktivität bereits in Ausführung oder auf der Warteschlange des Schedulers befindet, erfolgt eine erneute Einplanung erst nach der Ausführung der Aktivität.

Die `lock()`-Methode: Die Methode wird benutzt, um eine Aktivität temporär von der Planung auszunehmen. Das ist zum Beispiel sinnvoll, wenn ein Treiber ein Gerät einschaltet und asynchron über den Erfolg benachrichtigt wird. Eine erneute Ausführung der Aktivität in der diese Befehle an das Gerät sendet, sollte dann erst nach dem Start des Gerätes erfolgen. Die Planungssperre kann nur von der Aktivität selber gesetzt werden, die Entsperrung erfolgt an anderer Stelle beispielsweise einer Unterbrechungsbehandlung. Die Aktivität befindet sich zum Zeitpunkt der Sperrung in Ausführung und beendet sich nach dieser normal. Wenn andere Aktivitäten eine Aktivität sperren könnten, dann müsste die gesperrte Aktivität von der Warteschlange des Schedulers entfernt werden oder eine bereits laufende Aktivität abgebrochen werden. Insbesondere letzteres ist aufgrund der Ein-Stapel-Architektur nicht möglich.

Die `unlock()`-Methode: Die Methode signalisiert, dass eine Aktivität wieder eingeplant werden darf. Liegen bereits Planungswünsche vor, wird die Aktivität sofort in die Warteliste des Ablaufplaners eingefügt.

Die `rescheduleCount`-Variable: Die `rescheduleCount`-Variable zeigt an, wie oft die Aktivität eingeplant werden muss. Bei jedem Aufruf von `trigger()` wird der Wert um eins inkrementiert und nach jeder Abarbeitung der Aktivität um eins dekrementiert. Der Zähler stellt sicher, dass für jede Triggeroperation die Aktivität exakt einmal eingeplant wird. Der Zähler ist begrenzt aber in der Breite einstellbar, so kann für jede Anwendung sichergestellt werden, dass es nie zu einem Zählerüberlauf kommt.

Die `schedulingState`-Variable : Jede Aktivität hat einen Ausführungszustand. Welche Zustände möglich sind, hängt vom verwendeten Ablaufplanungsschema ab. Bei nichtpräemptiven Verfahren werden die Zustände `IDLE, RUNNING` und `SCHEDULED` benutzt. Bei präemptiven Planungsverfahren kommt der Zustand `INTERRUPTED` hinzu. Dieser wird

benötigt, um in der Zuteilungsschleife entscheiden zu können, ob die nächste Aktivität in der Warteschlange gestartet werden muss, oder ob der Kontrollfluss zu deren Ausführung zurrückkehren muss.

Die `locked`-Variable Ist diese Variable gesetzt, kann die Aktivität nicht neu eingeplant werden, stattdessen wird in der `trigger()`-Methode nur die `rescheduleCount`-Variable inkrementiert. Planungswünsche gehen dadurch nicht verloren.

4.2 Die Unterbrechungsbehandlungsroutinen

Unterbrechungen sind die initiale Quelle jeglicher Abarbeitung einer Anwendung. Bei der Behandlung von Unterbrechungen müssen benutzte Register vor der Behandlung gerettet und danach wieder hergestellt werden. In Abhängigkeit von dem verwendeten Abarbeitungsschema ist zudem eventuell auch der Eintritt in den Planungs-Monitor notwendig. Die Implementierung der eigentlichen Behandlungsroutine sollte davon jedoch nicht betroffen sein, sondern einzig und allein die spezifische Behandlung der aufgetretenen Unterbrechung durchführen.

Die Lösung ist, die Unterbrechungsbehandlung aufzuteilen. Abbildung 4.3 zeigt schematisch die Umsetzung in Reflex.

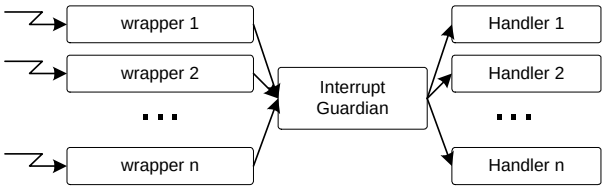


Abbildung 4.3: Indirekter Aufruf der Unterbrechungsbehandlungsroutinen

Nach dem Auftreten einer Unterbrechung wird die Ausführung an einer spezifizierten Adresse fortgesetzt. Die jeweilig ausgeführte `wrapper`-Routine vermerkt, welche Unterbrechung aufgetreten ist und springt zur allgemeinen Behandlung. In der allgemeinen Behandlung (`InterruptGuardian`) werden zunächst alle flüchtigen Register gesichert, danach findet ein Wechel zur Hochsprache statt. Dort wird die Synchronisation mit dem Ablaufplaner vorgenommen und die spezifische Behandlungroutine (`Handler`) wird aufgerufen. Eine ähnliche Art des Interruptmanagements wurde bereits in Systemen wie PURE [89] verwendet.

Die Basisklasse `InterruptHandler` registriert das Unterbrechungsbehandlungsobjekt zum Zeitpunkt der Initialisierung. Zusätzlich ist eine abstrakte `handle()`-Methode deklariert, die von den abgeleiteten Klassen implementiert werden muss.

4.3 Die Ereigniskanäle

Ereigniskanäle können, wie in Abschnitt 3.3 gezeigt, sehr verschieden sein. Dennoch sollen sie eine gleichartige Schnittstelle besitzen, sodass für die schreibende Seite transparent ist, welcher Art von Ereigniskanal Daten zugewiesen werden.

Die Schnittstelle wird in `Reflex` durch die abstrakten Klassen `Sink0` bis `SinkN` definiert. Diese Basisklassen deklarieren jeweils eine abstrakte Zuweisungsmethode. Bei `Sink0` heisst diese `notify()` bei den anderen `assign()`. Bei den Parameter-behafteten Klassen wird über Template-Parameter der Typ der Daten angegeben, die dem Ereigniskanal zugewiesen werden können. Durch die Template-basierte Umsetzung müssen die Typparameter von Schreiber und Ereigniskanal exakt übereinstimmen, die Kompatibilität der Typen reicht nicht aus. So ist z. B. die Klasse `Sink1<int>` nicht kompatibel mit der Klasse `Sink1<unsigned>` obwohl die Typen `int` und `unsigned` kompatibel sind. Die Abbildung 4.4 zeigt die Basisklassen für Ereigniskanäle und einige für Reflex implementierte Varianten.

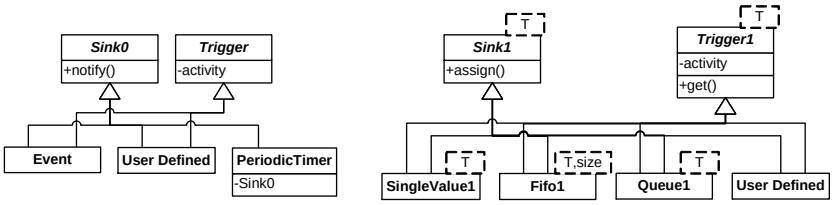


Abbildung 4.4: Ereigniskanalabstraktion in Reflex

Zusätzlich zur `Sink`-Schnittstelle implementieren Ereigniskanalpuffer noch die `Trigger`-Schnittstelle, welche die Endpunkte von Ereigniskanälen darstellen. Sie stoßen zum einen die assoziierte Aktivität an, zum anderen speichern sie die Daten für die asynchrone Verarbeitung. Die Daten werden über die Methode `get()` ausgelesen. Datenlose Ereigniskanalpuffer wie `Event` stoßen zwar die Aktivität an, speichern aber keine Daten.

Die Klasse `PeriodicTimer` stellt einen verarbeitenden Ereigniskanal dar, der als Taktteiler fungiert. Im Gegensatz zu den Puffern ist er nicht mit einer Aktivität assoziiert, sondern leitet jedes n-te Ereignis auf einen Ereigniskanal weiter.

4.4 Die Komponenten

Komponenten sind im Wesentlichen Container für die Komposition von Anwendungen. Jede Komponente kann Aktivitäten, Unterbrechungsbehandlungen, Funktionen und Variablen enthalten. In Reflex wird eine Komponente durch eine Klasse repräsentiert. Die Aktivitäten und Unterbrechungsbehandlungen sind Instanzenvariablen einer Komponente.

4.4.1 Schnittstellen

Eine Anwendung besteht aus untereinander verknüpften Komponenten. Für die Verknüpfung werden Ein- und Ausgänge benutzt, die die standardisierte Schnittstelle der Komponenten darstellen. Sie übernehmen eine Bindegliedfunktion, die es erlaubt, einen externen Ereigniskanal mit einem internen zu verbinden. Zudem stellen sie eine Schnittstelle zum System dar, da propagierte Ereignisse stets zu Systemaktivität führen.

Die Eingänge einer Komponente sind Ereigniskanalpuffer. Diese entkoppeln das Auftreten eines Ereignisses zeitlich von dessen Bearbeitung. Zudem besitzen sie eine einheitliche Schnittstelle, da sie von der Klasse `Sink` abgeleitet sind.

Die Ausgänge leiten ein in der Komponente generiertes Ereignis nach aussen und sind technisch Delegaten. Die Implementierung eines Ausganges vom Typ `Output1` ist in Listing 4.1 zu sehen. Die Klasse ist generisch und erhält als Template-Parameter den Typ der zu übermittelnden Daten (Zeile 1 und 17). Als Variablen enthält der Ausgang einen Zeiger auf das assoziierte Objekt (`object`) und einen Zeiger auf eine Funktion (`stub`) (Zeile 23 und 24). Diese werden in der `assign()`-Methode benutzt, um die `assign()`-Methode des verbundenen Ereigniskanals aufzurufen (Zeile 19). Ein Vorteil dieser Implementierung ist die Unabhängigkeit des aufrufenden Kodes in der Komponente von dem Typ des angeschlossenen Ereigniskanals. Gleichzeitig erlaubt diese Implementierung die Reduzierung virtueller Funktionsaufrufe bei der Weiterleitung von Ereignissen, da die aufrufende Funktion `invokeStub()` (Zeile 26-31) speziell für den angeschlossenen Ereigniskanal erzeugt wird. Der Übersetzer erzeugt die Funktion aufgrund der Zuweisung in Zeile 14 in der `connect()`-Methode.

```
template <typename ARG1T>
class Output1 : public Output {
    typedef void ( Stub_Fn )( void*, ARG1T );
public:
    Output1()
    {
        stub = &noOpStub<ARG1T>;
    }

    template <typename T>
    void connect(T* object)
    {
        this->object = static_cast<void*>(object);
        stub = &invoke_stub<T>;
    }

    void assign(ARG1T data)
    {
        (*stub)(object,data);
    }

protected:
    void* object;
    Stub_Fn* stub;
```

```
25
        template <class T>
        static void invokeStub (void* object, ARG1T data)
        {
            T* typedObject = static_cast <T*>(object);
30          typedObject->assign(data);
        }

        static void noOpStub(void* object, ARG1T data)
        {
35      }

    }
```

Listing 4.1: Implementierung eine Ausgangs

Ein Ausgang kann zu jeder Zeit innerhalb der Komponente beschrieben werden, selbst wenn der Ausgang nicht verbunden ist. Dadurch entfällt die Prüfung auf die Gültigkeit des Ausgangs. In statischen Systemen wird dieser zwar meist gültig sein, jedoch kann es bei der Fehlersuche oder bei einer dynamischen Rekonfiguration dazu kommen, dass Ausgänge nicht belegt sind. Die ständige Benutzbarkeit eines Ausganges wird durch die standardmäßig aufgerufene Funktion `noOpStub()` (Zeile 33-35) erreicht, die lediglich zum Aufrufer zurückkehrt. Die Funktion wird durch die Zuweisung im Konstruktur erzeugt (Zeile 7).

Ausgänge können zudem über die Basisklasse `Output` verwaltet werden. Das ist z. B. sinnvoll für das selektive Abschalten von Ausgängen oder die dynamische Neuverknüpfung.

4.4.2 Konfiguration einer Anwendung

Die Komponenten sind gleichzeitig die Abstraktion aus der eine Anwendung zusammengesetzt wird. In Abbildung 4.5 wird beispielhaft eine einfache Anwendung gezeigt. Die Anwendung besteht aus den drei Komponenten A, B und C. Die Komponenten A und B generieren dabei Ereignisse für C.

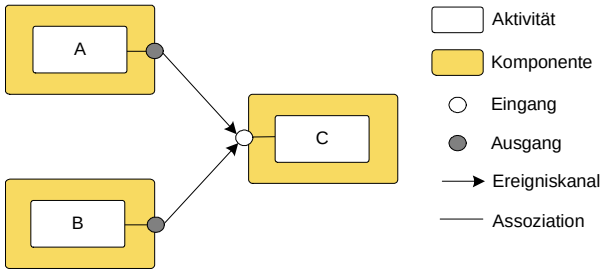


Abbildung 4.5: Beispielapplikation für die Konfiguration

In Listing 4.2 ist die dazugehörige Konfiguration der Anwendung zu sehen. Die Konfi-

guration besteht im Wesentlichen aus der Instanziierung (Zeilen 14 bis 16), der Verknüpfung (Zeilen 6 und 7) und der Parametriesierung (Zeile 10) der Komponenten. Die Komponenten sind Variablen in der `NodeConfiguration`-Klasse und werden im Konstruktur verknüpft und parametrisiert. Die Verknüpfungen bestehen zwischen Ausgängen und Eingängen, wobei einem Ausgang über die Methode `connectTo` bekanntgegeben wird, mit welchem Eingang er verknüpft ist. Zum Schluß wird im Beispiel noch der Komponente `b` eine höhere Priorität gegeben.

```
class NodeConfiguration {
public:
  NodeConfiguration()
  {
    //verknüpfen der Komponenten
    a.output1.connectTo(c.input1);
    b.output1.connectTo(c.input1);

    //parametriesieren der Komponenten
    b.setPriority(MAX_PRIORITY);
  }

  //instanziieren der Komponenten
  ComponentA a;
  ComponentB b;
  ComponentC c;
};
```

Listing 4.2: Konfiguration einer Anwendung

4.5 Synchronisation

Die Synchronisation der Anwendung erfolgt im Allgemeinen durch die Ereigniskanäle. Nur wenn Parallelitäten innerhalb einer Komponente vorliegen, z. B. durch die Verarbeitung einer Unterbrechung, muss explizit im Kode synchronisiert werden. Als Synchronisationsmittel werden auf den Mikrocontrollern die Unterbrechungen gesperrt, in Gastumgebungen wie z. B. Linux werden Markierungen (Flags) benutzt.

4.5.1 Ereigniskanäle

Für die Synchronisation der Ereigniskanäle werden in die Kanäle synchronisierende Elemente eingefügt. Diese sperren die Unterbrechungen und leiten das Ereignis weiter. Für eine vollständige Synchronisation wird ein solches Element am Anfang jedes Kanals eingefügt. Mit entsprechender Werkzeugunterstützung ist es zusätzlich möglich, automatisch in Abhängigkeit vom Abarbeitungsschema und den Prioritäten der Aktivitäten nur an den wirklich benötigten Stellen synchronisierende Elemente zu nutzen. Abbildung 4.6 zeigt ein Beispiel in dem selektive Synchronisation verwendet wird.

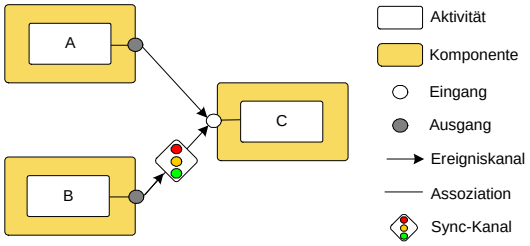


Abbildung 4.6: Einsatz des Synchronisationsfilters

Die Aktivitäten A und B schreiben beide auf den Eingang einer Ausgabekomponente. Es wird angenommen, dass die Schreiboperation von Aktivität A die von der Aktivität B unterbrechen kann, jedoch nicht umgekehrt. Daher muss nur die Schreiboperation von Aktivität B synchronisiert werden.

Der Vorteil dieser Lösung ist, dass nur dort synchronisiert wird, wo es notwendig ist. Das Sperren und Wiederherstellen des Unterbrechungsstatus benötigt Zeit, die insbesondere bei fein granularen Aktivitäten ins Gewicht fallen kann. Beispielsweise nimmt die Synchronisation auf Mikrocontrollern der MSP430-Reihe etwa 30 Takte in Anspruch. Der Nachteil der selektiven Synchronisation ist eine weitere Indirektionsstufe in den synchronisierten Ereigniskanälen. Die Indirektionsstufe kann jedoch in statischen Anwendungen potenziell durch den Übersetzer wegoptimiert werden.

4.5.2 Aktivitätssperren

Bei einer Aktivitätssperre wird die zugehörige Aktivität bis zur Entsperrung nicht neu eingeplant. Es ist nur vorgesehen, dass Aktivitäten sich selbst sperren, dadurch kommt es zu keiner Nebenläufigkeit beim Setzen der Aktivitätssperre. Die Aktivitäten befinden sich zum Zeitpunkt der Sperrung in Ausführung. Die aktuelle Ausführung wird nicht abgebrochen.

Das Sperren geschieht durch das Setzen einer Markierung (`locked`), diese wird bei der Einplanung durch den Ablaufplaner berücksichtigt. Da die Aktivität sich während des Setzens in Ausführung befindet, braucht diese Operation nicht synchronisiert zu werden.

Das Entsperren ist komplizierter, es muss festgestellt werden, ob die entsperrte Aktivität erneut eingeplant werden muss. Die Entsperrung erfolgt durch eine Unterbrechungsbehandlungsroutine oder eine andere Aktivität. Die entsperrte Aktivität kann sich zu diesem Zeitpunkt noch in Ausführung (aktuell unterbrochen) befinden oder befindet sich nicht auf der Warteliste. Listing 4.3 zeigt die Implementierung der `unlock()`-Methode für die präemptive EDF-Variante.

```
void EDFScheduler::unlock(EDFActivity* act)
{
    //Sperrung der Unterbrechungen für die gesamte Funktion
    InterruptLock lock;
5
```

```
        //entfernen der Sperre
        act->locked = false;

        //erneutes Einplanen der Aktivität, falls notwendig
10      if( ( act->status == EDFActivity::IDLE ) && act->rescheduleCount ) {

            enterScheduling();
            act->setDeadline();
            toSchedule.enque(act);
15          act->status = EDFActivity::SCHEDULED;
            leaveScheduling();

        }
    }
```

Listing 4.3: Methode zur Entsperrung einer Aktivität bei der präemptive EDF-Planung

Die Entsperrung der Aktivität ist eine kritische Operation und wird daher durch das Sperren von Unterbrechungen geschützt (Zeile 3). Eine erneute Einplanung (Zeilen 12 bis 16) der entsperrten Aktivität muss nur erfolgen, wenn sich diese nicht in Ausführung befindet und der `rescheduleCount` größer als null ist (Zeile 10). Der `rescheduleCount`-Wert wird jeweils nach einer Ausführung einer Aktivität dekrementiert. Befindet sich eine zu entsperrende Aktivität noch in Ausführung, dann wird die erneute Einplanung nach der Beendigung der aktuellen Ausführung vorgenommen. Die Einplanung bei einer Entsperrung unterscheidet sich von der normalen Einplanung nur darin, dass der `rescheduleCount` nicht erhöht wird, da kein neues Ereignis zur Einplanung geführt hat.

5 Das Abarbeitungsrahmenwerk

Im Ereignisflussmodell sind die Aktivitäten die planbaren Einheiten. Das Modell legt aber nicht fest, nach welcher Strategie die Planung abläuft. In diesem Abschnitt wird das Abarbeitungsrahmenwerk von REFLEX vorgestellt. Dieses erlaubt die von dem gewählten Ablaufplanungsschema unabhängige Implementierung von Aktivitäten aus Sicht des Anwendungsprogrammierers. Das ist für eine einfache Wiederverwendung von Aktivitäten bzw. Komponenten unabdingbar.

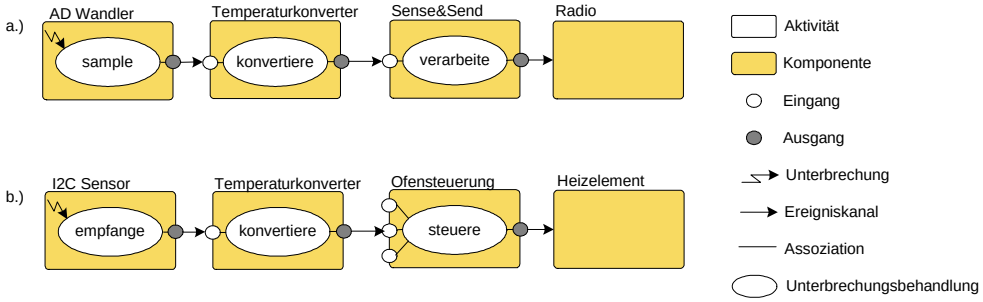


Abbildung 5.1: Verwendung einer Komponente in mehreren Kontexten

In Abbildung 5.1 wird als Beispiel eine Komponente zur Wandlung eines Rohwertes in eine Temperatur gezeigt. Diese Komponente wird in zwei unterschiedlichen Kontexten eingesetzt. Im Teil a) erfolgt der Einsatz in einer Sense-And-Send-Anwendung, wie sie in Sensornetzwerken vorkommen kann. Die Anwendung an sich ist zeitunkritisch. Im Teil b) der Abbildung wird diesselbe Komponente in der Steuerung eines Industrieofens eingesetzt. Für diese Anwendung ist es erforderlich, ein Temperaturprofil mit engen Grenzen zu durchlaufen, dadurch ergibt sich auch eine Zeitschranke für die Temperaturkonversion.

Die einfache Wiederverwendbarkeit ist insbesondere für Treiberkomponenten wichtig, da diese im Idealfall für eine Plattform nur ein einziges mal implementiert und dann in den unterschiedlichsten Anwendungen genutzt werden sollen. In anderen Szenarien ist es möglich, dass Anwendungen mit der Zeit wachsen und dadurch zeitliche Probleme bei der Ausführung verursachen. Dann ist es von Nutzen, die Prioritäten der betroffenen Komponenten ändern zu können, ohne deren interne Implementierung anzupassen.

Neben der Speichereffizienz muss bei der Implementierung die Laufzeiteffizienz betrachtet werden. Durch die Run-to-completion-Semantik werden viele Aktivitäten eine relativ kurze Laufzeit haben, wodurch konzeptbedingt der Mehraufwand für die Ablaufplanung steigt.

5.1 Struktur des Reflex-Abarbeitungsrahmenwerkes

In Abbildung 5.2 ist die Struktur des Rahmenwerkes zur Ablaufplanung zu sehen. Die Aktivitäten und auch der Ablaufplaner werden durch Konzepte [83] beschrieben. Die konkreten Ausprägungen implementieren diese Konzepte und erweitern sie gegebenenfalls. Ein Konzept ist eine Schnittstellenbeschreibung, jedoch wird nicht mit Ableitungen und Basisklassen gearbeitet. Der Grund dafür ist, dass in einer Anwendung nur ein Planungsschema verwendet werden kann und daher keine virtuellen Aufrufe notwendig sind.

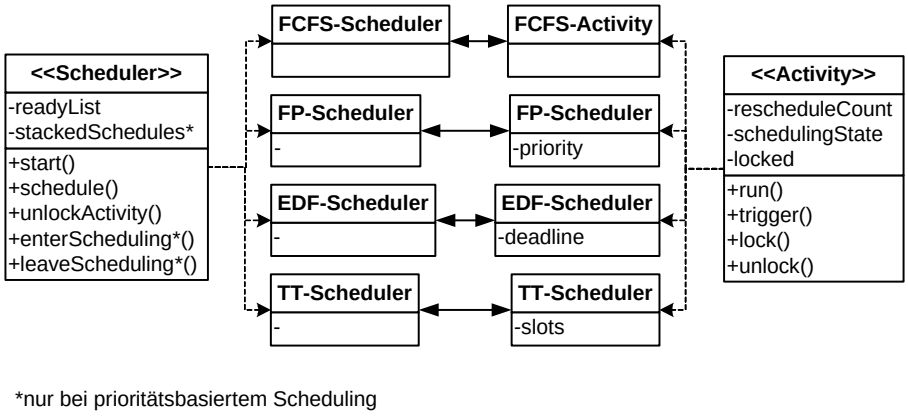


Abbildung 5.2: Konzeptbasierte Klassenstruktur für die Ablaufplanung

Über eine Typdefinition wird festgelegt, welche Ablaufplaner-Variante und Aktivitätsvariante in einem System benutzt wird. Überall dort, wo ein Ablaufplaner oder eine Aktivität benutzt wird, wird der Konzeptname benutzt. Leider bietet C++ keine direkte Unterstützung für die konzeptbasierte Programmierung, somit kann dem Übersetzer nicht kenntlich gemacht werden, dass eine Klasse ein Konzept implementiert. Dennoch kommt es zu einem Fehler beim Kompilieren, wenn das Konzept durch die Implementierung verletzt wird, da die Benutzung fehlschlägt.

Der konzeptbasierte Ansatz macht es möglich, sowohl den restlichen Systemkode (Energiemanagement, Synchronisation, Ereigniskanäle, Unterbrechungsbehandlung) als auch den Anwendungskode unabhängig vom verwendeten Ablaufplanungsschema zu implementieren. Zudem werden keine unnötigen Indirektionsstufen bei den Aufrufen der Planungsmethoden gebraucht.

Die Aktivitätsschnittstelle wurde bereits in Abschnitt 4.1 ausführlich dargestellt, daher folgt hier nur die Beschreibung der Ablaufplaner-Schnittstelle.

5.1.1 Das `Ablaufplaner`-Konzept

Das `Ablaufplaner`-konzept wird nur systemseitig verwendet. Keine der Funktionen oder Variablen kann direkt aus dem Anwendungskode heraus angesprochen werden.

Die `start()`-Methode:

Diese Methode wird nach der Initialisierung des Systems aufgerufen und stellt die `main()`-Funktion dar. Die Methode startet das System und kehrt nicht zurück.

Bei nicht präemptiven Ablaufplanern führt diese Methode die Aufrufschleife (dispatch-loop) aus und benachrichtigt das Energiemanagement, wenn keine Aktivität mehr ausgeführt wird. Im Falle von präemptiven Schedulern beschränkt sich die Funktionalität auf die Benachrichtigung des Energiemanagements.

Die `schedule()`-Methode:

`schedule()` wird aufgerufen, wenn eine Aktivität eingeplant werden muss. Der Aufruf erfolgt in der `trigger()`-Methode der Aktivität. Die `schedule()`-Methode fügt die übergebene Aktivität in die Warteschlange des Schedulers ein.

Die `enterScheduling()`-Methode:

Diese Methode ist zum Betreten des Planungs-Monitors notwendig. Bei nicht präemptiven Planungsverfahren wird diese Methode leer sein. Bei präemptiven Verfahren muss zusammen mit der `leaveScheduling()`-Methode sichergestellt sein, dass es zu keiner Prioritätsinversion bei der Ausführung von Aktivitäten kommt. Die `enterScheduling`-Methode wird sowohl am Anfang jedes von Software ausgelösten Schedule-Vorgangs als auch am Anfang jeder Unterbrechungsbehandlung aufgerufen.

Die `leaveScheduling()`-Methode:

Diese Methode wird nach dem Verlassen eines Kontextes aufgerufen, in dem kein `dispatch()` stattfinden durfte. Die Methode veranlasst die Zuteilung (Dispatching), falls dies notwendig ist.

Die `readyList`-Variable:

Hinter dieser Variable verbirgt sich eine Liste, die alle zur Ausführung eingeplanten Aktivitäten enthält. Die Aktivitäten sind gemäß der implementierten Planungsstrategie sortiert.

Die `stackedSchedules`-Variable:

Die Variable wird für den Planungs-Monitor benutzt und zeigt an, wie viele Kontexte aktuell betreten sind, in denen keine Zuteilung stattfinden darf. Nur wenn diese Variable 0 ist, darf `dispatch()` aufgerufen werden.

5.1.2 Die Warteschlange der Ablaufplaner

Wie bereits mehrfach erwähnt ist der Speicher in den betrachteten Systemen stark begrenzt. Somit ist es von großem Vorteil, wenn für die Ablaufplanung der benötigte Speicher a priori bereitsteht und nicht erst zur Laufzeit Speicher für das Einplanen einer Aktivität bereitgestellt werden muss.

Dafür enthalten Aktivitäts-Objekte einen Zeiger, der für die Verkettung innerhalb der Warteschlange (`readyList`) des Ablaufplaners benutzt wird. So ist bereits bei der Instanziierung einer Aktivität sichergestellt, dass diese später auch zu jedem Zeitpunkt zur Ausführung eingeplant werden kann.

5.2 Einfache unsortierte Abarbeitung

Eines der einfachsten Ablaufplanungsverfahren ist First Come First Serve (FCFS). Bei diesem werden die ausführbaren Einheiten bei der Planung in eine Liste eingehangen, welche dann von vorne nach hinten abgearbeitet wird. Die einzelnen ausführbaren Einheiten können sich dabei nicht gegenseitig unterbrechen.

Die Vorteile des Verfahrens sind die Einfachheit, der geringe Laufzeitaufwand, die Fairness und der geringe Platzbedarf des Stapels. Nachteilig ist, dass wichtige Aktivitäten unter Umständen lange auf ihre Ausführung warten müssen.

5.2.1 Verwaltung der Aktivitäten

Die Aktivitätsklasse für die FCFS-Planung implementiert das allgemeine Aktivitätskonzept. Für die Verwaltung in einer Warteschlange enthält die `FCFSActivity` zudem einen `next`-Zeiger auf eine Aktivität. Auf den `next`-Zeiger können nur die `FCFSActivity`-Basisklasse und der FCFS-Ablaufplaner zugreifen.

Der FCFS-Ablaufplaner verwaltet die Aktivitäten mit zwei Zeigern (`first` und `last`), welche die Enden der Warteschlange referenzieren. Durch die zwei Zeiger ist es möglich, die `schedule()`-Methode mit konstanter maximaler Laufzeit zu implementieren.

5.2.2 Die Abläufe im Ablaufplaner

Der FCFS-Ablaufplaner ist nicht präemptiv, daher kann die Zuteilung in der Endlosschleife der `start()`-Methode ausgeführt werden. Ein Planungs-Monitor ist nicht notwendig, weshalb die Methoden `enterScheduling()` und `leave-Scheduling()` leer sind. Listing 5.1 zeigt die Implementierung der `dispatch()`-Methode.

```
void FifoScheduler::dispatch()
{
        _interruptsEnable();
        first->run();           //Ausführen der Aktivität
        _interruptsDisable();

        first->rescheduleCount--;
```

```
        if((first->rescheduleCount) && (!first->locked)){ //nochmal einplanen
10          last->next = first;
            last = first;
            first = first->next;
            last->next = 0;
        }else{                                          //nicht nochmal einplanen
15          first->status = FifoActivity::IDLE;
            first = first->next;
        }
    }
```

Listing 5.1: Die `dispatch()`-Methode des FCFS-Ablaufplaners in Reflex

In dem Listing ist zu sehen, dass der Ablaufplaner dafür sorgt, dass während der Ausführung einer Aktivität die Unterbrechungen aktiviert sind (Zeilen 3-5). Außerdem ist die Auswertung der `rescheduleCount`-Variable und `locked`-Variable der Aktivität dargestellt (Zeile 9), anhand der entschieden wird, ob die Aktivität neu eingeplant werden muss.

5.3 Abarbeitung nach festen Prioritäten

Die statisch prioritätsbasierte Abarbeitung erlaubt die Anordnung von Aktivitäten gemäß ihrer Wichtigkeit auf der Warteschlange des Ablaufplaners. Dies verringert die Aktivierungszeit von Aktivitäten mit höherer Priorität, erhöht aber den Systemaufwand gegenüber der Abarbeitung nach dem FCFS-Prinzip. Die Prioritäten können können frei bestimmt werden oder gemäß einer Strategie wie z. B. bei dem RMS-Planungsverfahren (Rate Monotonic Scheduling) [68]. Bei diesem speziellen Verfahren verhält sich die Prioriät der ausführbaren Einheiten umgekehrt proportional zu ihrer Laufzeit. Dass heißt, Aktivitäten mit kurzer Ausführungzeit haben eine hohe Priorität.

Für Reflex wurden drei Varianten des FixedPriority-Ablaufplaners (FP) implementiert. Die einfachste arbeitet nicht präemptiv, lediglich die Warteschlange des Ablaufplaners ist nach Prioritäten sortiert. In den präemptiven Varianten können höher priorisierte Aktivitäten niedriger priorisierte unterbrechen.

Für alle Varianten wird derselbe Aktivitätstyp verwendet. Dieser besitzt wie die FCFS-Aktivitäten einen `next`-Zeiger. Zusätzlich enthält jede Aktivität eine Variable, welche die Priorität der Aktivität angibt. Diese Variable wird bei der Konfiguration der Anwendung gesetzt.

5.3.1 Nicht-präemptive FP-Ablaufplanung

Die Umsetzung ist analog dem `FCFSScheduler`. Jedoch wird in diesem Fall eine sortierte Liste benutzt, aus der jeweils die vorderste Aktivität bei der Zuteilung entnommen wird. Die gerade laufende Aktivität befindet sich nicht in der Liste, da sonst das Einsortieren neuer Aktivitäten erschwert wäre.

5.3.2 Einfache präemptive FP-Ablaufplanung

Diese Variante unterscheidet sich erheblich von der nicht präemptiven Variante. In der `start()`-Methode wird in der Endlosschleife keine Zuteilung mehr durchgeführt. Diese muss aufgrund des Ein-Stapel-Prinzips immer nach abgeschlossenen Einplanungen durchgeführt werden. Implementiert wird dieses Verhalten über den Zuteilungs-Monitor in den Methoden `enterScheduling()` und `leaveScheduling`, welche in Listing 5.2 dargestellt sind.

```
void FPScheduler::enterScheduling()
{
    InterruptLock lock; //enterScheduling muss atomar ausgeführt werden

    stackedSchedules++; //Sperren der Zuteilung von Aktivitäten

    //die laufende Aktivität als Unterbrochen markieren
    PriorityActivity* first = (PriorityActivity*)(readyList.first());
    if(first!=0){
        first->status = PriorityActivity::INTERRUPTED;
    }
}

void FPScheduler::leaveScheduling()
{
    stackedSchedules--; //Freigabe der Zuteilung von Aktivitäten

    if(stackedSchedules==0){ //Zuteilung kann durchgeführt werden
        dispatch();          //Starten der Zuteilung
    }
}
```

Listing 5.2: Implementierung des Zuteilungs-Monitors für das FP-Scheduling

Die Methoden nutzen im Wesentlichen die Variable `stackedSchedules`, welche den Wert 0 hat, wenn die Zuteilung durchgeführt werden kann. Der Zähler erlaubt es, verschachtelte Planungsoperationen durchzuführen.

In der `enterScheduling()`-Methode wird außerdem die gerade laufende Aktivität als unterbrochen markiert. Dadurch kann bei der Zuteilung festgestellt werden, ob eine Aktivität neu gestartet oder zu deren Ausführung zurückgekehrt werden muss. Listing 5.3 zeigt den Kode der Zuteilung.

```
void PriorityScheduler::dispatch()
{
    //starte Aktivitäten am Anfang der Warteschlange die nicht im
    //Zustand INTERRUPTED sind
    PriorityActivity* first = readyList.first();
    while( (first !=0 ) && (first->status != PriorityActivity::INTERRUPTED) ){

        first->status = PriorityActivity::RUNNING;
```

```
10          //ausführen der Aktivität mit freigeschalteten Unterbrechungen
            _interruptsEnable();
            first->run();
            _interruptsDisable();

15          //entfernen der Aktivität von der Warteschlange
            readyList.deque();
            first->status = PriorityActivity::IDLE;

            //erneute Einplanung, falls notwendig
20          if((first->rescheduleCount > 0) && (!first->locked)){

                first->rescheduleCount--;
                first->status = PriorityActivity::SCHEDULED;
                readyList.insert(first);
25          }

            first = readyList.first();
        }

30      //Rückkehr von der Zuteilung, wenn zu einer unterbrochenen Aktivität
        //zurückgekehrt wird, wird diese wieder als RUNNING markiert
        if(first && (first->status == PriorityActivity::INTERRUPTED) ){
                first->status = PriorityActivity::RUNNING;
        }
35  }
```

Listing 5.3: Die dispatch()-Methode für den preemptiven FP-Ablaufplaner in Reflex

Eine wesentliche Änderung gegenüber der nicht präemptiven Variante ist die `while`-Schleife (Zeile 5), in welcher alle Aktivitäten am Anfang der `readyList` bis zur ersten unterbrochenen Aktivität oder bis zum Ende gestartet werden. Im Unterschied zur nicht-präemptiven Implementierung wird außerdem eine Aktivität erst nach der Ausführung von der `readyList` entfernt, dies ist für die Ordnung der Aktivitäten notwendig. Am Ende der Funktion muss eine eventuell unterbrochene Aktivität wieder in den Zustand `RUNNING` versetzt werden, bevor zu dieser zurückgekehrt wird (Zeile 31).

5.3.3 Präemptiv mit minimalen Interruptblockierungszeiten

In der gezeigten präemptiven Variante der prioritätsbasierten Ablaufplanung ergibt sich ein Problem für echtzeitfähige Systeme. Die Unterbrechungen sind während der Einplanung neuer Aktivitäten gesperrt, die Dauer der Operation ist dabei abhängig von der Position in der `readyList`, an der eine Aktivität eingeordnet wird. Der Zeitaufwand steigt dabei linear mit der Position.

Die resultierende Zeit, in welcher die Unterbrechungen gesperrt sind, kann demzufolge recht lang werden. Das führt dann eventuell zu verlorenen Unterbrechungen. In diesem Abschnitt wird eine Variante der Planung nach fixen Prioritäten vorgestellt, in welcher die Blockierungszeiten begrenzt werden können.

Grundidee ist es, beim Einplanen Aktivitäten atomar in eine einfache unsortierte Liste einzuhängen und die Aktivitäten dann bei freigeschalteten Unterbrechungen in die `readyList` einzusortieren. In Abbildung 5.3 ist der Ablauf dargestellt.

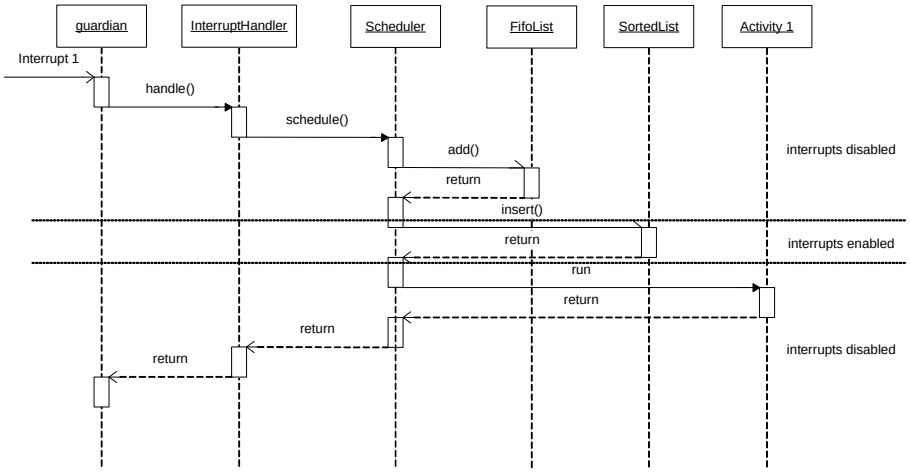


Abbildung 5.3: Ablauf der Planung mit zwei Listen

Das Einfügen in die `FifoList` erfolgt mit konstantem Aufwand, und die resultierende Zeit in welcher Unterbrechungen gesperrt sind kann begrenzt werden. Dies erhöht nicht nur die Reaktivität des Systems, sondern vereinfacht allgemein auch die Echtzeitanalyse [17]. Jedoch kommt es während der Einplanung zum mehrmaligen Sperren der Unterbrechungen, wenn mehrere Aktivitäten von der `FifoList` in die Warteschlange umgetragen werden. Zudem steigt der Aufwand durch das Einfügen jeder Aktivität in zwei Listen stark an. Die Auswirkung der zweiten Liste auf die Blockierungszeit für Unterbrechungen und die Gesamtlaufzeit der `schedule()`-Methode wird in Tabelle 5.1 deutlich. Sie zeigt die Zeiten für das Einplanen zweier Aktivitäten an, wobei die zweite Aktivität eine geringere Priorität als die erste hat. Die Werte geben die benötigten Takte für einen Freescale HC(S)12-Mikrocontroller an.

	Planung erste Aktivität		Planung zweite Aktivität	
	Gesamtdauer	Dauer Sperrung	Gesamtdauer	Dauer Sperrung
1-Listen Version	333	333	345	345
2-Listen Version	466	276	478	276

Tabelle 5.1: Vergleich der Laufzeiten für ein- und zwei-Listen FP-Planung

Die 1-Listen Variante ist prinzipiell schneller als die 2-Listen Variante, jedoch ist die Dauer der Sperrung von Unterbrechungen länger und steigt, wenn auch nur moderat (12 Takte pro Position), mit der Position, an der die Aktivität eingeordnet wird. Im Falle eines mit 8 MHz betriebenen HC(S)12 Mikrocontrollers beträgt die maximale

Blockierungszeit der 1-Listen Variante rund 42 μs + 1,5 μs pro Aktivität. Bei der 2-Listen Variante sind es konstant 35 μs. Der Mehraufwand der 2-Listen Variante ist bei der Einordnung einer Aktivität an der ersten Stelle in der Warteschlange am höchsten. Der relative Mehraufwand sinkt aber, je weiter hinten eine Aktivität eingeordnet wird.

Diese Ergebnisse zeigen, dass nur in Applikationen mit vielen Aktivitäten der Einsatz des 2-Listen Schedulers Vorteile bringen kann. Ansonsten unterscheiden sich die Länge der Blockierungszeiten nicht genügend und die 1-Listen Variante verursacht insgesamt weniger Aufwand.

5.4 Abarbeitung nach dynamischen Fristen

Das Earliest Deadline First Planungsverfahren (EDF) ist das optimale Verfahren in Bezug auf die Auslastung für harte Echtzeitsysteme [17, 68], theoretisch lässt sich eine Auslastung von 100% erreichen.

Für diese Aussage gelten jedoch drei wesentliche Einschränkungen. Erstens muss das System periodisch sein. Zweitens muss die Zeitschranke für die Abarbeitung eines Jobs gleich dem nächstmöglichen Zeitpunkt sein, zu welchem der Job erneut ausgeführt wird. Drittens wird der Planungsaufwand als vernachlässigbar klein angenommen. In tief eingebetteten ereignisgetriebenen Systemen werden diese Annahmen selten zutreffen. Ereignisse werden aperiodisch, eventuell schubartig auftreten. Insbesondere wird der Planungsaufwand bei der feingranularen Abarbeitung durch Aktivitäten nicht vernachlässigbar klein sein.

Wie bei dem FP-Ablaufplaner wurden drei Varianten implementiert. Eine nicht präemptive Variante, eine mit einer Warteschlange und eine mit zwei Warteschlangen. Die Implementierungen unterscheiden sich nur in wenigen Punkten von den FP-Varianten. Im Wesentlichen liegt der Unterschied im Vergleich der Prioritäten von Aktivitäten, bei der EDF-Planung werden Zeitschranken verglichen, bis zu denen die Aktivitäten vollständig ausgeführt werden müssen.

5.4.1 Dynamische Berechnung der Zeitschranke

Die Zeitschranke wird zum Zeitpunkt des Einplanens einer Aktivität wie in Formel 5.1 gezeigt berechnet. Die Zeitschranke ist der aktuelle Zeitpunkt plus einer für die Aktivität gegebenen Reaktionszeit.

$$T_{Zeitschranke} = T_{aktuell} + T_{Reaktionszeit} \tag{5.1}$$

Ein besonderes Problem in tief eingebetteten Systemen ist die Zeitbasis auf der die EDF-Planung beruht. In leistungsfähigeren Systemen ist die Zeitbasis meist die Realzeit. Die Darstellung der Realzeit ist in tief eingebetteten Systeme aufgrund der benötigten Bitbreite zu aufwendig. Bereits der Vergleich von 32 bit Werten muss auf vielen 8- und 16 bit Mikrocontrollern durch Softwarefunktionen implementiert werden, da kein entsprechender Maschinenbefehl zur Verfügung steht.

Es kann somit nur ein 16 bit Zähler genutzt werden, der als Maximalwert 65535 annehmen kann. Bei einer zeitlichen Auflösung des Zählers von 1ms kommt es alle 6,5 s zum Überlauf des Zählers. Aber selbst wenn ein 32 bit Zähler verwendet wird, kommt es ca. alle 5 Tage zum Überlauf. Bei typischen Anwendungen, die teilweise über Monate oder Jahre laufen, wird es demnach zu einem Überlauf der Zeitlinie kommen, welcher entsprechend behandelt werden muss.

Der Überlauf ist kritisch, da er die Ordnung der Aktivitäten durcheinander bringen kann und somit das Planungsschema verletzen würde.

5.4.2 Implementierung des Zeitschrankenvergleichs

Ein einfacher Lösungsansatz ist die Rückstellung der Systemzeit mit gleichzeitiger Anpassung aller Zeitschranken der zu diesem Zeitpunkt eingeplanten Aktivitäten. Problematisch ist dabei, dass der ganze Vorgang atomar sein muss und somit die Unterbrechungen gesperrt werden müssen. Die maximale Dauer dieser Operation ist abhängig von der maximalen Länge der Ablaufplaner-Warteschlange. Derart lange Sperren können in einem Echtzeitsystem nicht toleriert werden.

Wird nicht die Zeitschranke zweier Aktivitäten verglichen, sondern die Differenz zwischen der aktuellen Systemzeit und der jeweiligen Zeitschranke, kann der Überlauf des Zählers einfach toleriert werden. Dabei darf jedoch die erlaubte Antwortzeit nicht größer als der maximale Zählerwert sein, und es darf während der Ausführung zu keiner Verletzung einer Zeitschranke kommen.

Abbildung 5.4 zeigt ein Beispiel. Angenommen wird ein 4 bit breiter Zähler für die Systemzeit. Die aktuelle Systemzeit ist `11` und die Zeitschranke der Aktivität A ist `13`. Die Zeitschranke der Aktivität B wäre eigentlich `18`, ist durch den Überlauf bei der Berechnung jedoch `2` und liegt damit vor der Systemzeit. Die Differenzen zwischen den Zeitschranken der Aktivitäten und Systemzeit sind jedoch `2` für Aktivität A und `7` für Aktivität B. Dadurch hat Aktivität A aktuell die höhere Priorität, obwohl deren Zeitschranke den höheren Wert gegenüber Aktivität B hat.

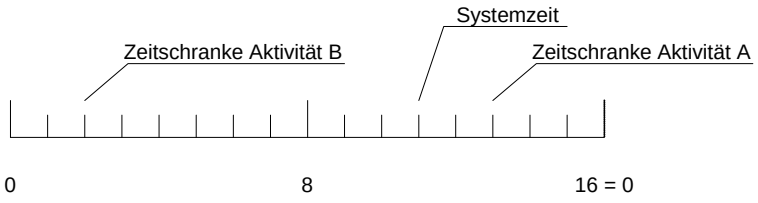


Abbildung 5.4: Beispielsituation für die Überlauf-tolerante Prioritätsberechnung

Es muss für dieses Verfahren durch eine statische Analyse im Vorfeld sichergestellt werden, dass keine Zeitschranke überschritten wird. Diese statische Analyse ist jedoch Teil des normalen Entwicklungsprozesses hart echtzeitfähiger Anwendungen.

Die Vergleichsfunktion wird sehr oft durchlaufen und sollte deshalb eine möglichst kurze Laufzeit haben. Listing 5.4 zeigt die Implementierung des Vergleichs in Reflex.

```
bool EDFActivity::lessEqual(EDFActivity& right)
{
    Time systemTime = clock.getTime();

    Time leftOffset = this->deadline - systemTime;
    Time rightOffset = right.deadline - systemTime;

    return (leftOffset <= rightOffset);
}
```

Listing 5.4: Vergleich von Deadlines in Reflex

Eine wesentliche Technik ist die Nutzung des Überlaufs auch bei den Subtraktionen (Zeile 5 und 6). Dabei werden die Differenzen von der Systemzeit zur Zeitschranke der jeweiligen Aktivität berechnet. Bei allen Zeitschranken, die vor der Systemzeit liegen, tritt ein Überlauf auf. Dies führt dazu, dass Aktivitäten mit Zeitschranken vor der Systemzeit einen höheren Vergleichswert bekommen als Aktivitäten mit einer Zeitschranke nach der Systemzeit. Die Ordnung von Aktivitäten mit Zeitschranken vor bzw. nach der Systemzeit bleibt erhalten. Wichtig ist die Benutzung eines vorzeichenlosen Datentyps für `leftOffset` und `rightOffset` und die Darstellung als 2er-Komplement auf Maschinenebene. Abbildung 5.5 zeigt die Berechnungsergebnisse für das Beispiel aus Abbildung 5.4.

Abbildung 5.5: Beispielvergleichs zweier Zeitschranken

Der Berechnung für einen 16 bit breiten Zähler benötigt beispielsweise auf einem MSP430 maximal 14 Takte und ist damit schneller als der einfache Vergleich von zwei 32 bit Werten, der 18 Takte benötigt. Außerdem wird wie erläutert das Überlaufproblem gelöst.

5.5 Statische Zeitgesteuerte Abarbeitung

Die statisch zeitgesteuerte Abarbeitung wird auch als Time Triggered Scheduling (TT) bezeichnet und häufig in sicherheitsrelevanten Systemen benutzt [50, 51]. Der wesentliche Vorteil ist die a priori Planung [86, 87], die sicherstellt, dass es im Betrieb nie zu einer

Überlastsituation kommt. Außerdem ist der Jitter bei der Aktivierung von Aktivitäten sehr klein, was sich günstig in Steuerungsanwendungen auswirkt [19].

In diesem Abschnitt wird dargestellt, wie eine zeitgesteuerte Abarbeitung auf das Ereignisflussmodell abgebildet werden kann. Die grundsätzliche Idee ist es, die Aktivierung einer Aktivität zusätzlich zum Auftreten des anstoßenden Ereignisses von der Systemuhr abhängig zu machen. Die zeitliche Reihenfolge der Abarbeitung ist bereits durch die Planung gegeben und spiegelt sich in der Warteschlange wieder. Beim Anstoßen einer Aktivität wird nicht die Warteschlange aktualisiert, sondern ein Zähler für die angestoßene Aktivität inkrementiert. Die Ausführung startet, wenn der Startzeitpunkt erreicht und der Zählerwert größer null ist.

5.5.1 Aufbau der Warteschlange

Im Gegensatz zu den anderen vorgestellten Verfahren wird eine indirekte Warteschlange benutzt, welche die statische Planung wiederspiegelt. Das heißt, nicht die Aktivitäten bilden die Liste, sondern es gibt spezielle Listenelemente, die Aktivitäten referenzieren. Dadurch kann eine Aktivität mehrfach in der Warteschlange referenziert werden. Der Grund ist, dass bei der statischen Zeitplanung die Ausführung der Aktivitäten über eine Hyperperiode verteilt wird [87]. Innerhalb einer Hyperperiode kann eine Aktivität mehrfach zur Ausführung kommen. Die Warteschlange ist als Ring ausgeführt, um die Hyperperioden zyklisch zu durchlaufen.

In Abbildung 5.6 ist der Ergeignisflussgraph einer Anwendung dargestellt, welche statisch zeitgesteuert ausgeführt werden soll. Die Planung für einen Durchlauf auf der Zeitachse und die zugehörige Warteschlange sind in Abbildung 5.7 zu sehen.

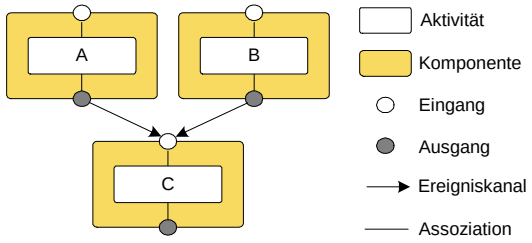


Abbildung 5.6: Beispielanwendung für die statisch zeitgesteuerte Abarbeitung

Es existieren drei Aktivitäten, A und B werden durch ein Hardware-Ereignis angestoßen und C durch ein Software-Ereignis. Die statische Zeitplanung im Beispiel teilt den einzelnen Aktivitäten einen oder mehrere Startzeitpunkte zu. Aktivität C muss mehrmals in einer Runde eingeplant werden, da pro Runde bis zu zwei Ereignisse für C generiert werden. Werden Aktivität A oder B auch mehrfach in einer Runde angestoßen, muss auch C entsprechend öfter ausgeführt werden.

Die Listenelemente benötigen Speicherplatz, welcher durch ein Feld zur Verfügung gestellt wird. Die Anzahl der Elemente kann pro Anwendung eingestellt werden. Jedes Element enthält sowohl einen Zeitstempel als auch einen Zeiger auf eine Aktivität. Ein

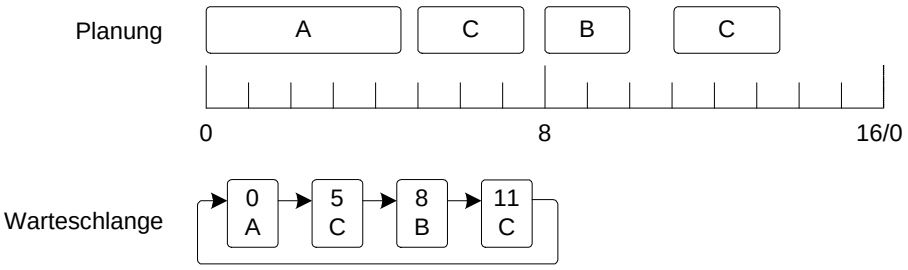


Abbildung 5.7: Statische Ablaufplanung einer Anwendung

Zeiger auf das nächste Element ist nicht notwendig, da dieses im nächsten Feldelement steht. Da die Planung statisch ist, kann die Warteschlange vorausberechnet und im Kodebereich abgelegt werden, wodurch weniger RAM benötigt wird. Das ist insbesondere dann sinnvoll, wenn lange Hyperperioden geplant werden müssen.

5.5.2 Die Aktivierungsschleife

Die Aktivierungsschleife ist wie bei den anderen nicht-präemptiven Verfahren eine Endlosschleife. Jedoch werden nicht alle noch ausstehenden Aktivitäten direkt nacheinader abgearbeitet, sondern vor der Ausführung einer Aktivität wird erst gewartet, bis der gewünschte Startzeitpunkt erreicht ist. Die Implementierung dieser Schleife in Reflex ist in Listing 5.5 zu sehen.

```
void TimeTriggeredScheduler::start()
{
    while(1){

        //Abfrage, ob Startzeitpunkt der nächsten Aktivität erreicht ist
        if(getReflex().clock.getTime() == current->startTime){

            //Abfrage, ob die Aktivität aktiviert werden muss
            if( (current->act->rescheduleCount > 0) &&
                !current->act->locked ) {

                current->act->rescheduleCount--;

                //starten der Aktivität
                _interruptsEnable();
                current->act->run();
                _interruptsDisable();

            }

            //gehe zur nächsten Aktivität in der Warteschlange
            current = current->next;
        }
```

```
25          //schlafenlegen des Mikrocontrollers bis zum nächsten Slot
            getReflex().powerManager.powerDown();

        }
    }
```

Listing 5.5: Aktivierungsschleife bei der statisch zeitgesteuerten Abarbeitung in Reflex

Der Ablaufplaner besitzt einen Zeiger auf ein Element der Warteschlange (`current`). Für dieses Element vergleicht er den Zeitstempel mit der aktuellen Zeit (Zeile 4). Ist der Wert gleich, wird überprüft ob die Aktivität ausgeführt werden muss. Dazu nutzt der Ablaufplaner die `rescheduleCount`-Variable und die `lock`-Markierung (Zeilen 9 und 10). Nachdem die Aktivität sich beendet hat, wird der Elementzeiger aktualisiert. Nach der Ausführung einer Aktivität oder bei dem Warten auf die nächste Zeitscheibe wird der Mikrocontroller in einen Schlafmodus versetzt.

Bei der zeitgesteuerten Abarbeitung ist die Bestimmung der Startzeiten innerhalb der Hyperperiode das größte Problem, Ansätze zur Lösung finden sich z. B. in [87]. Die Implementierung in Reflex erfordert zudem, dass die Slots in einer Hyperperiode eine Mindestlänge besitzen. Es darf unter Berücksichtigung aller durch Unterbrechungen verursachten Verzögerungen nie dazu kommen, dass der Aktivierungszeitpunkt für eine Aktivität verpasst wird. Das würde zur Inaktivität des System für eine gesamte Hyperperiode führen.

5.6 Speicherverbrauch für die Ablaufplanung

Das Abarbeitungsrahmenwerk für Reflex muss leichtgewichtig sein, da durch das Ereignisflussmodell relativ feingranulare Nebenläufigkeiten modelliert werden. Zudem steht auf den Zielplattformen meist nur wenig Speicher zur Verfügung.

Für Reflex wurden 8 Varianten der Ablaufplanung implementiert, welche unterschiedlich komplex sind. In Tabelle 5.2 sind die Größen des Objektkodes dieser Varianten für den MSP430 und HC(S)12 aufgeführt.

Für beide Architekturen wird trotz starker Unterschiede im Befehlssatz in etwa gleichviel Speicher für den Kode benötigt. Die statisch zeitgesteuerte Abarbeitung schneidet am besten ab, danach folgt die FCFS-basierte Abarbeitung und die prioritätsbasierten Varianten. Am aufwendigsten ist die Planung nach dem EDF-Schema. Aber selbst die komplexeste EDF-Variante mit 2 Listen benötigt zusammengenommen um die 800 Byte Speicherplatz. Somit können diese Verfahren von der Kodegröße her bedenkenlos in tief eingebetteten Systemen eingesetzt werden.

Bei dem Speicherverbrauch für die planbaren Einheiten (Aktivitätsobjekte) unterscheiden sich die Abarbeitungsschemen ebenfalls. Angegeben ist der Platzbedarf für den Basisklassenanteil eines Aktivitätsobjektes. Die FCFS- und FP-basierte Planung benötigt am wenigsten Platz in einer Aktivität, EDF benötigt doppelt soviel. Die Unterschiede zwischen den Mikrocontrollern kommen durch die 2 Byte Datenausrichtung auf dem MSP430 zustande. Die statisch zeitgesteuerte Abarbeitung stellt einen Sonderfall dar,

	TI MSP430			Freescale HC(S)12		
	Ablaufplaner	Aktivität		Ablaufplaner	Aktivität	
	ROM	ROM	RAM	ROM	ROM	RAM
statisch zeitgesteuert	292	52	8+6	357	91	7+6
FCFS	368	94	8	351	135	7
FP - nicht präemptiv	370	100	8	399	141	8
FP - präemptiv, 1 Liste	460	100	8	509	141	8
FP - präemptiv, 2 Listen	614	100	8	619	141	8
EDF - nicht präemptiv	426	138	16	403	193	16
EDF - präemptiv, 1 Liste	514	138	16	509	193	16
EDF - präemptiv, 2 Listen	658	138	16	617	193	16

Tabelle 5.2: Kodegröße für die implementierten Ablaufplaner

da der benötigte RAM neben der Anzahl der Aktivitäten noch von der Anzahl der benutzten Slots abhängt. Es muss die komplette Planungstabelle gespeichert werden. Da die Tabelle statisch ist, kann diese auf Mikrocontrollern mit von-Neumann-Architektur im ROM abgelegt werden.

	TI MSP430			Freescale HC(S)12		
	1.	2.	3.	1.	2.	3.
statisch zeitgesteuert	15	13	13	11	11	11
FCFS	116	112	73	48	53	22
FP - nicht präemptiv	114	123	72	81	101	22
FP - präemptiv, 1 Liste	238	248	72	197	208	22
FP - präemptiv, 2 Listen	381	391	72	332	343	22
EDF - nicht präemptiv	155	192	72	243	341	22
EDF - präemptiv, 1 Liste	307	344	72	434	520	22
EDF - präemptiv, 2 Listen	450	487	72	559	645	22

Tabelle 5.3: Laufzeitaufwand für die Ablaufplanung bei verschiedenen Verfahren

5.7 Systemlaufzeiten für die Ablaufplanung

Für den Vergleich der Ablaufplaner wurde der Laufzeitaufwand für Aktivitäten in unterschiedlichen Situationen ermittelt. Es wurden nacheinander 3 Ablaufplanungsoperationen aus einer laufenden Aktivität heraus durchgeführt. Die laufende Aktivität hat bei den priorisierenden Ablaufplaner-Varianten eine größere Priorität als die neu eingeplanten. Das verhindert eine präemptive Einplanung, wodurch nur die Zeit für die Einplanung ermittelt wird. Es werden zwei Aktivitäten eingeplant, eine davon mehrfach.

In Tabelle 5.3 sind die Laufzeiten in Takte für das Einplanen von Aktivitäten für die Mikrocontroller TI MSP430 und Freescale HC(S)12 angegeben.

Die Zeiten zeigen deutlich den unterschiedlichen Aufwand für die Planung bei den Verfahren. Die statisch zeitgesteuerte Abarbeitung ist am schnellsten, dort ist die Reihenfolge der Abarbeitung vorherbestimmt, lediglich ein Zähler muss inkrementiert werden. Schon wesentlich langsamer sind die nicht präemptiven Ablaufplaner. Die Laufzeiten unterscheiden sich vor allem bei längeren Warteschlangen, da dann bei FP und EDF der Einsortierungsaufwand steigt. Bei FCFS bleibt der Aufwand hingegen konstant, lediglich das Einfügen in eine leere Liste ist aufwendiger, da dann Start- und End-Zeiger gesetzt werden müssen.

Bei den präemptiven Varianten ist zudem deutlich der Unterschied zwischen der unterbrechbaren (2-Listen) und nicht unterbrechbaren Einplanung (1-Liste) zu sehen.

6 Das Energiemanagement

Das Energiemanagement ist eine wesentliche Aufgabe eines Betriebssystems für tief eingebettete Systeme. Das vorgestellte Ereignisflussmodell ermöglicht es, implizite Energiesparmechanismen zu implementieren. Dadurch ist es möglich, den funktionalen Kode einer Anwendung von dem Kode für das Energiemanagement zu trennen. Vor der Vorstellung des hier gewählten Ansatzes für das Energiemanagement werden die notwendigen technischen Grundlagen erläutert. Nach der Vorstellung des Konzeptes wird dann die Umsetzung gezeigt und eine Auswertung vorgenommen.

6.1 Grundlagen des Energiemanagements

Der zentrale Bestandteil eines tief eingebetteten Systems ist der Mikrocontroller, welcher meist auch der größte Energieverbraucher ist. Auf dem Mikrocontroller befinden sich der Rechenkern, der RAM, verschiedene IO-Module, sowie Module für die Takterzeugung und die Spannungsversorgung. Abbildung 6.1 zeigt schematisch den Aufbau eines Mikrocontrollers. Ein Mikrocontroller ist somit ein sogenanntes SOC (System on a Chip).

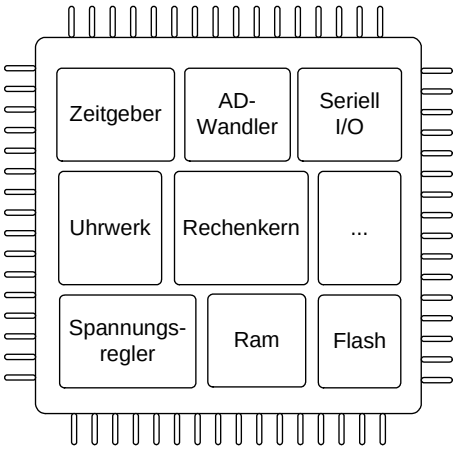


Abbildung 6.1: Aufbau eines Mikrocontrollers

Der Energieverbrauch von Mikrocontrollern setzt sich zusammen aus dem Verbrauch des Rechenkerns und dem Verbrauch der On-Chip-Module. Der Kern verbraucht die meiste Energie in dem System, jedoch nur, wenn er aktiv ist. Der zweitgrößte Verbraucher ist

typischerweise die Uhreneinheit, welche das systemweite Taktsignal generiert. Die Spannungsversorgung ist kein eigentlicher Verbraucher, jedoch treten dort Wandlungsverluste auf. Dabei steigen die absoluten Verluste mit der anliegenden Last.

Wichtig ist außerdem, dass sich der Stromverbrauch des Mikrocontrollers aus einem statischen und einem dynamischen Anteil zusammensetzt. Der statische Anteil beschreibt den Teil, der allein durch das Anlegen der Versorgungsspannung verursacht wird. Dieser Anteil umfasst im Wesentlichen Leckströme und den Stromverbrauch analoger Schaltungsteile. Dieser Verbrauch hängt quadratisch von der Spannung ab, kann aber ansonsten nicht durch den Anwender beeinflusst werden. Der dynamische Teil umfasst den Verbrauch, der durch Schaltungsaktivität entsteht. Dabei ist zu beachten, dass in CMOS (Complementary Metal Oxide Semiconductor) Schaltungen im Idealfall nur während der Umschaltvorgänge Strom fließt, wodurch der dynamische Anteil am Energieverbrauch höher ist als der statische. Dieses Verhalten weisen auch Low-End-Mikrocontroller auf, da sie meist die Technologiegrenzen in Sachen Geschwindigkeit nicht ausloten und dadurch die statischen Leckströme klein sind.

Formel 6.1 zeigt, wie sich die dynamische Leistungsaufnahme ($P_{dynamisch}$) zusammensetzt. Die Faktoren sind der Aktivitätsgrad (α), die kapazitive Last (C), die Versorgungsspannung (V) und die Taktfrequenz(f).

$$P_{dynamisch} = \alpha * C * V^2 * f \tag{6.1}$$

Die Einflussmöglichkeiten der Anwendung auf die Faktoren sind unterschiedlich. Der Aktivitätsgrad wird direkt durch die Anwendung bestimmt. Je aktiver diese ist, desto höher ist auch der Energieverbrauch. Die kapazitive Last wird im Wesentlichen von der Technologie bestimmt, jedoch können auf Mikrocontrollern ungenutzte On-Chip Module abgeschaltet werden, was die kapazitive Last verringert. Die Spannung kann meist aus technischen Gründen nicht verändert werden. Dann ist der quadratische Einfluss auf den Energieverbrauch bei Low-End Mikrocontrollern nicht nutzbar. Die Taktfrequenz wird entweder durch einen Quarz fest vorgegeben oder kann über einen DCO (Digitally Controlled Oszillator) eingestellt werden.

Im Folgenden werden Ansätze vorgestellt, welche den Energieverbrauch eines Mikrocontrollers senken können.

Partielles Abschalten bei Mikrocontrollern

Üblicherweise erlauben Mikrocontroller ein partielles An- und Abschalten. Als Techniken kommen dabei Clock- und Power-Gating zum Einsatz. In heute gängigen Mikrocontrollern wird vorwiegend Clock-Gating eingesetzt. Dabei wird das globale Taktsignal über einen Schalter an das jeweilige Modul geführt. Beim Abschalten wird der Schalter geöffnet und das Taktsignal erreicht nicht mehr das Modul, die Versorgungsspannung liegt jedoch weiter an. Das Verfahren ist sehr effektiv, da der dynamische Stromverbrauch in den verwendeten CMOS-Schaltungen überwiegt. Clock-Gating erlaubt zudem das Anschalten eines Moduls in einem Takt.

Mit geringeren Strukturbreiten bei der Chipherstellung steigt der statische Anteil am Stromverbrauch. Deshalb wird es zukünftig möglich sein, Module über Power-Gating abzuschalten. Zusätzlich zum Taktsignal wird dabei auch die Versorgungsspannung über einen Schalter geführt. Power-Gating kann auch in analog arbeitenden Modulen von Vorteil sein, da diese ständig Strom verbrauchen. Das Problem des Power-Gating sind die starken Lastschwankungen für die Versorgungsleitungen auf dem Mikrocontroller. Dadurch kommt es zu Umladevorgängen der Kapazitäten im Mikrocontroller und somit zu erhöhtem Energieverbrauch. Um Spannungsschwankungen und damit die Umladeverluste zu minimieren, kann es notwendig sein, Module stufenweise anzuschalten. Eng verbunden mit diesem Problem sind eventuelle Einschaltverzögerungen, hervorgerufen durch die Ladezeit für die kapazitive Last und Einschwingvorgänge in analogen Schaltungsteilen.

Ein weiteres Problem von Power Gating ist, dass Registerinhalte der Komponenten verloren gehen, wenn diese nicht mit Strom versorgt werden. Das heißt, Module müssen nach dem Anschalten stets wieder initialisiert werden. Zudem kann es sein, dass externe Anschlüsse (Pins) des Mikrocontrollers bei Power Gating während der Schlafphasen nicht getrieben werden, obwohl die Anschlüsse ständig einen definierten Zustand haben müssen.

RAM ist in tief eingebetteten Systemen nicht nur wegen der Kosten knapp, sondern auch weil er viel Energie verbraucht. Mittels Power-Gating könnten nicht benutzte Speicherbereiche abgeschaltet werden. In [47] wird ein solches Verfahren vorgestellt. Wenn der verwendete Mikrocontroller das teilweise Abschalten von RAM erlaubt, könnte das in der Speicherverwaltung genutzt werden, indem nur belegter Speicher auch aktiv ist.

Von der Benutzung unterscheiden sich Clock- und Power-Gating kaum. Beides sind sehr einfache und schnelle Verfahren, welche lokal auf Modulebene angewendet werden können, wenn das Modul gerade nicht benutzt wird.

Schlafmodi

Gängige Mikrocontroller unterstützen softwareseitige Energiesparkonzepte bereits durch die Bereitstellung von verschiedenen Schlafmodi. Tabelle 6.1 zeigt die Stromaufnahme in den verfügbaren Arbeitsmodi verschiedener Mikrocontroller.

Bei den Mikrocontrollern unterscheidet sich die Stromaufnahme in den einzelnen Modi erheblich. Jedoch gibt es bei allen einen Warte-Modus (Idle, Wait), in dem nur der Kern abgeschaltet wird. Bis auf den H8300 besitzen alle Mikrocontroller zudem einen Modus, in dem nur noch ein Zeitgeber und externe Unterbrechungen den Controller wecken können, der Quarz ist dabei abgeschaltet. Im tiefsten Schlafmodus ist nur noch die interne Spannungsversorgung aktiv und lediglich externe Signale können den Controller wecken. Die bereitgestellten Schlafmodi bieten ein hohes Potenzial für die Senkung des Energieverbrauchs.

Mikrocontroller	Modus	Eigenschaften	Stromaufnahme
Atmel ATmega 128 @ 3 V & 8 MHz	Active		8 mA
	Idle	$CLK_{Core} und CLK_{Flash}$ deaktiviert	4 mA
	ADC Noise Reduction	wie Idle CLK_{GPIO} deaktiviert	k.A.
	Power Down	ext. Oszillator deaktiviert nur externes aufwecken	0.3 μA
	Power Save	wie Power Down jedoch Timer0 aktiv	8 μA
	Stand By	wie Power Down Oszillator aktiv (fast wakeup)	0.6 mA
	Extended Stand By	wie Power Save Oszillator aktiv (fast wakeup)	k.A.
Freescale HCS12 @ 5 V & 16 MHz	Active		50 mA
	Wait	CLK_{Core} deaktiviert Module wahlweise aktiv	5-30 mA
	Pseudo Stop	CLK_{Core} und Module deaktiviert COP und RTI wahlweise aktiv	400-600 μA
	Stop	alles deaktiviert nur externes aufwecken	100 μA
Renesas H8/300 @ 5 V & 16 MHz	Active		36 mA
	Sleep	CLK_{Core} deaktiviert	24 mA
	Stand By	ext. Oszillator deaktiviert nur externes aufwecken	0.01 μA
Texas Instruments MSP430 @ 3 V & 5 MHz	Active		2.1 mA
	lpm0	CLK und MCLK deaktiviert	275 μA
	lpm1	lpm0 + DCO deaktiviert	k.A.
	lpm2	lpm0 + SMCLK deaktiviert	55 μA
	lpm3	lpm1 + SMCLK deaktiviert	8 μA
	lpm4	alles deaktiviert nur externes aufwecken	4 μA

Tabelle 6.1: Betriebsmodi ausgewählter Mikrocontroller

DVS und DFS

DVS (Dynamic Voltage Scaling) und DFS (Dynamic Frequency Scaling) sind Technologien, welche aus dem Bereich leistungsstarker Prozessoren kommen und meist parallel verwendet werden. Es wird ausgenutzt, dass die Versorgungsspannung mit der Taktfrequenz abgesenkt werden kann. Die Wirkung bezieht sich vor allem auf den statischen Stromverbrauch, da die Leckströme verringert werden. Der dynamische Stromverbrauch wird nur durch DVS gesenkt. Durch DFS verringert sich zwar die Stromaufnahme, jedoch verlängert sich dementsprechend die Rechenzeit für eine Aufgabe. Für DVS ergeben sich die Grenzen dadurch, dass die Versorgungsspannung nicht beliebig abgesenkt werden kann, da dann die Transistoren nicht mehr funktionieren.

Auch im Bereich von Mikrocontrollern können DVS und DFS genutzt werden, das bringt jedoch meist keinen Vorteil [24, 25, 58]. Der Grund ist, dass Mikrocontroller typischerweise nicht wie Desktop oder Server CPUs an der technologischen Leistungsgrenze betrieben werden. Der statische Anteil des Stromverbrauchs ist daher meist vernachlässigbar klein. Zudem werden viele Low-End-Mikrocontrollern mit einer fest vorgegebenen Versorgungsspannung betrieben. In [25] wird zudem gezeigt, dass bei Mikrocontrollern DVS durch die geringen Lasten einen großen Nachteil hat. Das ist der schlechte Wirkungsgrad der benötigten dynamischen Spannungsregler. Der Wirkungsgrad liegt im Bereich von 70% bei einer gut dimensionierten Schaltung. Festspannungsregler hingegen können selbst bei niedriger Last Wirkungsgrade von 95% erzielen. In [45] wird weiterhin festgestellt, dass DVS nicht für die feingranulare Steuerung geeignet ist, da das Auf-/Entladen der Schaltung den gewünschten Effekt zunichte macht. Zudem wird für die feingranulare Steuerung spezielle Hardware benötigt.

Neben den technischen Problemen hat DFS auch noch ein großen Einfluss auf das zeitliche Verhalten. Die Programmlogik muss dies berücksichtigen und wird dadurch komplexer. So müssen in Steuerungsanwendungen Funktionen oft in engen zeitlichen Grenzen ausgeführt werden [20], z. B. das streng periodische Auswerten einer Temperatur. Durch das Ändern der Taktfrequenz müssen ständig Anpassungen vorgenommen werden. Zudem erschwert eine schwankende Taktfrequenz die Echtzeitanalyse einer Anwendung.

Zusammenfassend erscheint die Nutzung von DFS und DVS für Low-End Mikrocontroller noch wenig sinnvoll. Jedoch sollte für jede Anwendung ermittelt werden, welches die geringst mögliche Taktfrequenz und Versorgungsspannung ist. Die ermittelte Kombination sollte in Verbindung mit den Schlafmodi genutzt werden [45].

Andere Ansätze

In [31] werden eine Reihe von energieeffizienten Verfahren zur Ablaufplanung zusammengefasst. Die Grundidee ist Tasks in einer für den Energieverbrauch optimalen Reihenfolge auszuführen und dadurch möglichst wenig zwischen Arbeitsmodi zu wechseln. Jedoch wird dazu Wissen über das zukünftige Verhalten der gerade laufenden Programme benötigt. In allgemeinen ereignisgetriebenen Systemen verhindert die sporadische Natur von

Ereignissen die effektive Nutzung. In statisch zeitgesteuerten Systemen hingegen ist es sinnvoll, die Aktivitäten so zu planen, dass der Energieverbrauch minimiert wird.

Weiterhin gibt es viele Arten des anwendungsseitigen Energiemanagements. So können beispielsweise Abtastraten verändert werden, um den Energieverbrauch zu senken [75]. In Sensornetzwerken haben die Netzwerkschichten einen großen Einfluss auf den Energieverbrauch. Dort können Medienzugriffsverfahren mit Schlafzyklen [82] oder energiegewahre Routingmechanismen [39] eingesetzt werden. Der Einfluss auf den Energieverbrauch ist teilweise erheblich. Das anwendungsseitige Energiemanagement ist jedoch nicht Gegenstand dieser Arbeit und wird daher nicht weiter betrachtet.

6.2 Das Energiemanagementkonzept

Das Ereignisflussmodell erlaubt die Implementierung einer für den Anwendungsprogrammierer weitgehend impliziten Energieverwaltung. Der Grund ist wiederum die Run-to-completion-Semantik der Aktivitäten und die ereignisgetriebene Aktivierung, die aktives Warten unnötig macht. Dadurch ist die Warteschlange des Ablaufplaners in einer Nulllastsituation automatisch leer. In diesem Kapitel wird gezeigt, wie dieses Verhalten für das Energiemanagement ausgenutzt werden kann.

6.2.1 Energiesparansatz

Der Ansatz zum Energiesparen ist, den Kern und die angeschlossenen Module des Mikrocontrollers so oft und so lange wie möglich zu deaktivieren. Daher wird, wann immer eine Nulllastsituation erreicht wird, der Mikrocontroller in den tiefst möglichen Schlafmodus versetzt. Im Wesentlichen bestimmen die zugelassenen Unterbrechungen, welche Schlafmodi des Mikrocontrollers genutzt werden können, da sie die initialen Lastverursacher in einem ereignisgetriebenen System sind. Die Kontrolle über die Unterbrechungen ermöglicht daher auch die Kontrolle über den Energieverbrauch. Wird eine Unterbrechung aktiviert oder deaktiviert, wird das dem System mitgeteilt, welches wiederum den tiefst möglichen Schlafmodus ermitteln kann. Es wird im Folgenden zwischen primären und sekundären Unterbrechungen unterschieden. Erstere werden durch externe Unterbrechungen ausgelöst, zweitere sind eine Folge von Softwareereignissen z. B. die Sendeunterbrechung einer seriellen Schnittstelle.

Primäre Unterbrechungen können nicht implizit abgeschaltet werden. Ob diese aktiv sein müssen, hängt von der aktuellen Programmphase ab. Es ist daher sinnvoll, eine Schnittstelle vom System zur Verfügung zu stellen, die das Umschalten von Programmphasen erlaubt. Für jede Unterbrechungsbehandlungsroutine kann dann im Ereignisflussgraphen annotiert werden, in welchen Programmphasen diese aktiv sein soll. So ergibt sich für den Anwendungsprogrammierer eine einfache Möglichkeit aus einer Komponente, z. B. einem standardisierten Schlaf-Wach-Zyklus, den Programmmodus zu wechseln, ohne dass dieser steuernden Komponente die einzelnen genutzten Unterbrechungen bekannt sein müssen.

Bei sekundären Unterbrechungen ist eine automatische Abschaltung möglich. Bei einer seriellen Schnittstelle kann die Sendeunterbrechung nur ausgelöst werden, wenn Daten zu übertragen sind. Somit kann die Sendeunterbrechung deaktiviert werden, wenn kein Datenpaket zum Versenden anliegt. Der Treiber kann daher selbst entscheiden, wann die Sendeunterbrechung abgeschaltet werden kann.

Zusätzlich zur Betrachtung der Unterbrechungen gibt es anwendungsseitige Bedingungen, die vom Energiemanagement berücksichtigt werden müssen. Beispielsweise kann es sein, dass die Aufwachzeit aus einem bestimmten Schlafmodus des Mikrocontrollers zu lang ist für eine Anwendung, weshalb es möglich sein muss, Sperrvermerke für diesen Schlafmodus zu setzen. Eine andere Bedingung ist die Dauer, für die der Mikrocontroller schlafen gelegt wird. Eigentlich alle Mikrocontroller besitzen einen Schlafmodus, in dem nur noch ein internes Taktsignal den Mikrocontroller aufwecken kann. Prinzipiell kann dieser Schlafmodus mit einer fest vorgegebenen Schlafdauer in Nulllastsituationen betreten werden. Jedoch führt das zu unnötigen Weckvorgängen. Es erscheint sinnvoller, für den Schlafmodus die Schlafdauer dynamisch bestimmen zu können.

6.2.2 Funktionsweise des Energiemanagements

In Abbildung 6.2 ist eine einfache Sense-and-Send-Anwendung aus dem Sensornetzbereich gezeigt, anhand derer exemplarisch die Abläufe des Energiemanagements dargestellt werden. Die Anwendung nimmt zyklisch eine Temperatur über einen AD-Wandler auf und propagiert diese über eine serielle Schnittstelle. Das Programm verfügt über eine separate Komponente, welche zwischen zwei Programmphasen umschaltet. In der Wachphase soll das Programm Daten aufnehmen und versenden und danach bis zur nächsten Wachphase schlafen. In der Schlafphase bleibt nur die Uhrenunterbrechung aktiv.

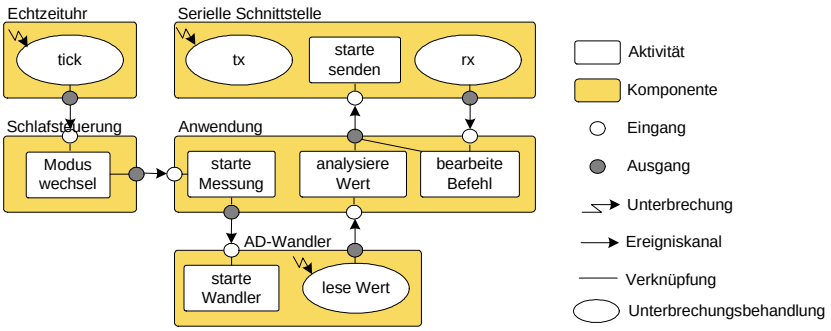


Abbildung 6.2: Beispielanwendung für das Energiemanagement

In Abbildung 6.3 ist der zeitliche Ablauf eines Schlaf-Wachzyklus vereinfacht dargestellt. Anfänglich weckt die Unterbrechung der Echtzeituhr den Mikrocontroller. Daraufhin wird die Komponente zur Steuerung der Schlaf-Wachphasen angestoßen. Diese wechselt in die Wachphase, dabei wird die Empfangsunterbrechung der Ein-/Ausgabekomponente aktiviert und die Zeit der Wachphase auf der Echtzeituhr eingestellt. Zusätzlich wird auch noch die Anwendung angestoßen, die wiederum eine Messung des AD-

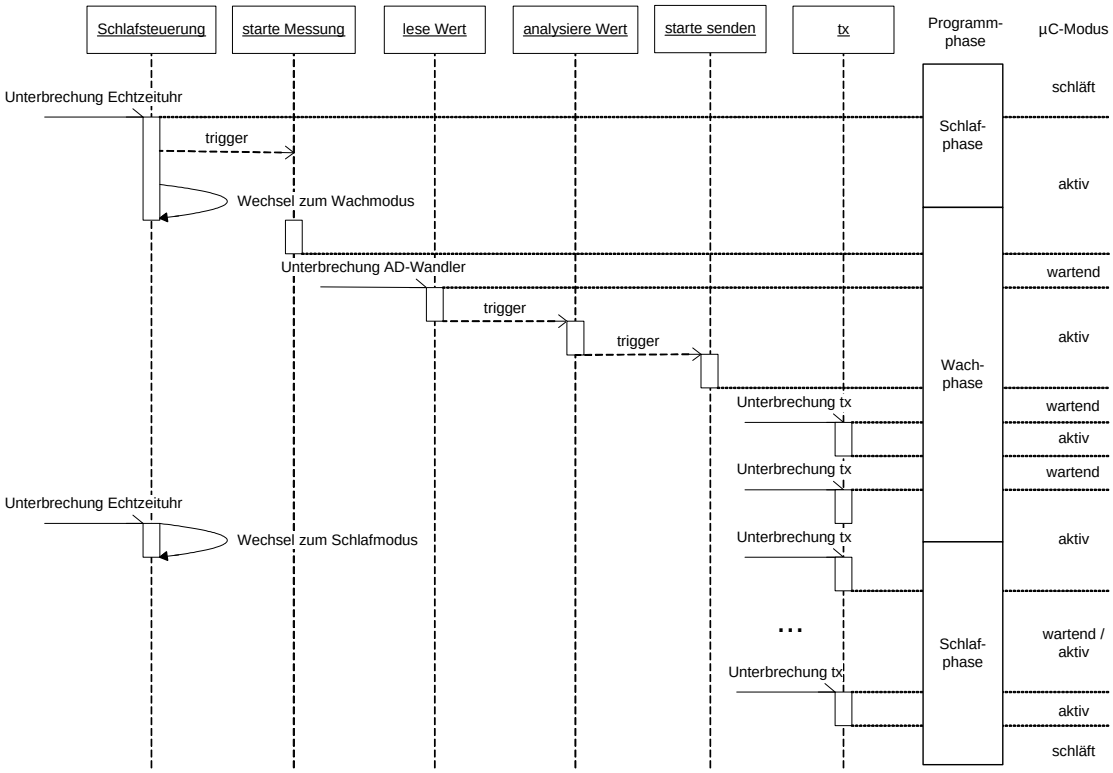


Abbildung 6.3: Beispielhafter Energiemanagementablauf

Wandlers initiiert. Im weiteren Verlauf kommt die Unterbrechung des AD-Wandlers und sendet einen Rohwert an die Anwendung. Diese sendet den gemessenen Wert über die serielle Schnittstelle. Während der Ausgabe werden immer wieder Nulllastsituationen erreicht, in denen der Mikrocontroller in einen leichten Schlafmodus versetzt wird (wartend). Außerdem tritt während des Sendens erneut die Echtzeitunterbrechung auf, woraufhin zur Schlafphase gewechselt wird. Der Sendevorgang wird aber erst noch abgearbeitet, bevor das System in den Tiefschlafmodus (schläft) wechselt.

Es ist zu beachten, dass in dem Beispiel der Kode der Anwendungskomponente frei von Direktiven zur Energieverwaltung ist. Nur die Treiber und die Komponente zur Steuerung des Schlaf-Wach-Zyklus benutzen Funktionen des Energiemanagements, sind aber unabhängig von der konkreten Anwendung. Außerdem muss die Komponente zur Steuerung der Schlaf-Wach-Phasen sich nicht mit der Anwendung synchronisieren. Diese Synchronisation übernimmt das System und sorgt im Beispiel dafür, dass vor Betreten des Tiefschlafmodus die Übertragung der Nachricht abgeschlossen wird.

6.2.3 Schaltbare Kanäle

Das bisher gezeigte Verfahren ermöglicht es, ausgehend von Unterbrechungen die Systemaktivität und damit den Energieverbrauch zu beeinflussen. In Anwendungen kann es

jedoch sein, dass nur bestimmte Verarbeitungsschritte während einer Programmphase nicht ausgeführt werden sollen, um Energie zu sparen. Dazu bietet es sich an, schaltbare Ereigniskanäle zu benutzen. Das sind Kanäle, die deaktiviert und aktiviert werden können. Nur wenn sie aktiviert sind, leiten sie Ereignisse weiter.

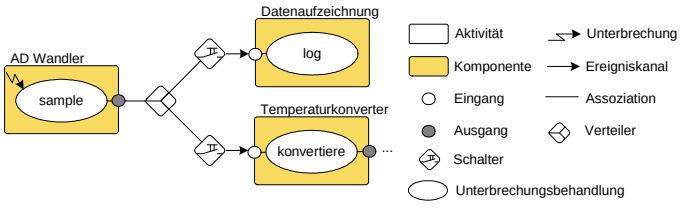


Abbildung 6.4: Schaltbarer Ereigniskanal

In Abbildung 6.4 ist ein Beispiel dargestellt. Ein AD-Wandler generiert Werte, die abhängig von der Anwendungsphase an verschiedene Komponenten weitergeleitet werden. Die schaltbaren Kanäle werden wie primäre Unterbrechungen vom Energiemanagement behandelt. Das heisst, wechselt die Anwendung in eine Phase, in der die Weiterleitung eines Ereignisses gewünscht ist, dann wird der Kanal durchgeschaltet, ansonsten gesperrt. Die schaltbaren Kanäle helfen so die Last und den Energieverbrauch im System zu reduzieren. Der Anwendungskode ist wiederum nicht davon betroffen.

6.3 Die Umsetzung

Abbildung 6.5 zeigt die Klassenstruktur für das Energiemanagement in Reflex. Die Klasse `EnergyManager` stellt das globale Energiemanagement dar und ermittelt, welcher Schlafzustand aktuell betreten werden kann. In der Klasse werden Objekte verwaltet, die aktiviert und deaktiviert werden können. Die verwalteten Objekte implementieren die `EnergyManageable`-Schnittstelle. In Nulllastsituationen wird die `powerDown()`-Methode der Klasse `EnergyManager` aufgerufen. Die Methode `switchMode()` schaltet zwischen den Arbeitsmodi einer Anwendung um.

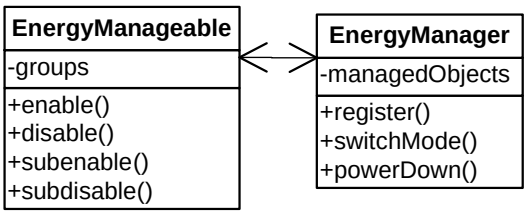


Abbildung 6.5: Struktur der Energieverwaltung

Die Basisklasse `EnergyManageable` stellt Methoden zum Aktivieren (`enable()`) und Deaktivieren (`disable()`) eines zu verwaltenden Objektes bereit. In den Methoden sind

die systemseitigen Aufgaben implementiert und es wird die objektspezifische Funktion zum An- und Abschalten aufgerufen (`subEnable()` bzw. `subDisable()`). Der Konstruktor der `EnergyManageAble`-Klasse meldet das Objekt beim Systems an.

6.3.1 Auswahl des Schlafmodus

Die Entscheidung, welcher Schlafmodus des Mikrocontrollers betreten wird, erfolgt über eine Tabelle von Zählern im `EnergyManager`-Objekt. Für jeden verfügbaren Schlafmodus eines Mikrocontrollers gibt es einen Eintrag. Jedes `EnergyManageAble`-Objekt besitzt die Variable `deepestAllowedSleepMode`, welche den tiefst möglichen Schlafmodus des Mikrocontrollers angibt, wenn das Objekt aktiv ist. In der `enable()`-Methode bzw. der `disable()`-Methode der Klasse `EnergyManageAble` wird jeweils der Zählerwert für diesen Schlafmodus in der Tabelle des `EnergyManager`-Objektes inkrementiert oder dekrementiert. Wird eine Nulllastsituation erreicht, durchläuft der `EnergyManager` die Tabelle mit Zählerwerten von vorne nach hinten, stößt er auf einen Eintrag ungleich null, so ist das der tiefste mögliche Schlafmodus. Der Zähler für den tiefsten Schlafmodus des Controllers wird mit „1“ initialisiert, die anderen mit „0“. So wird automatisch der tiefste Schlafmodus betreten, wenn alle Module des Mikrocontrollers abgeschaltet sind.

Listing 6.1 zeigt die notwendige Konfiguration und die Umsetzung. Für jeden Controller existiert eine Spezifikation der zur Verfügung stehenden Schlafmodi, im Beispiel sind die für den TI MSP430 aufgeführt (Zeilen 3-7). Jeder Treiber bestimmt während der Initialisierung den tiefstmöglichen Schlafmodus, welcher betreten werden darf, wenn der Treiber aktiv ist. Gezeigt wird das für den `TimerA`, welcher mindestens den Modus `LPM1` erfordert (Zeile 13). Bei der Aktivierung wird der entsprechende Zähler für diesen Modus inkrementiert (Zeile 20), danach wird die spezifische Anschaltroutine aufgerufen (Zeile 21).

```
//Controller.h
enum SleepModes {
    LPM0,
    LPM1,
    LPM2,
    LPM3,
    LPM4
};

//TimerA.cc
TimerA::TimerA()
{
    deepestAllowedSleepMode = LPM1;
}

//EnergyManageAble.cc
void EnergyManageAble::enable()
{
    if(!enabled){
        getReflex().energyManager.useCounts[deepestAllowedSleepMode]++;
```

```
            subEnable();
        }
    }
```

Listing 6.1: Bestimmung des tiefst möglichen Schlafmodus für einen Treiber

6.3.2 Programmphasensteuerung

In vielen Anwendungen sind je nach Programmphase unterschiedliche Geräte und Komponenten des Mikrocontrollers aktiv. Benötigt wird eine Schnittstelle, die es erlaubt, zwischen den Phasen einer Anwendung zu wechseln. Die besonderen Schwierigkeiten liegen darin, die Hardware-Unabhängigkeit der Programmlogik sicherzustellen und die Aktivierung und Deaktivierung von Komponenten mit dem System zu synchronisieren.

Die `EnergyManager`-Klasse bietet die Methode `switchMode()` zur gruppenorientierten Steuerung von Geräten an. Diese besitzt zwei Parameter. Eine Bitmaske gibt an, welche Gruppen umgeschaltet werden sollen und eine andere gibt für die betroffenen Gruppen an, ob sie aktiv oder inaktiv sein sollen. Listing 6.2 zeigt die Spezifikation der Gruppen und die Benutzung in der Konfiguration der Anwendung (`NodeConfiguration`).

```
//conf.h
typedef uint8 RunMode;
enum RunModes {
    //Definition der Gruppen für Programmteile und -phasen
    DCM_AWAKE = 0x1,
    DCM_SLEEP = 0x2,
    DCM_MASK = DCM_AWAKE | DCM_SLEEP,
    ANOTHER_GROUP = 0x4,
    ANOTHER_MASK = ANOTHER_GROUP
};

//NodeConfiguration.h
class NodeConfiguration : public System {
public:
    NodeConfiguration()
    {
        ...

        //Zuordnung der verwaltbaren Objekte zu Gruppen
        serial.rxiHandler.setGroups(DCM_AWAKE);
        timer.setGroups(DCM_AWAKE | DCM_SLEEP);

        adc.setGroups(ANOTHER_GROUP | DCM_AWAKE);

        ...
    }
};

//DutyCycleManager.h
void DCM::run()
```

```
{
    //umschalten der betroffenen Gruppen
    getReflex().energyManager.switchMode(DCM_MASK, DCM_SLEEP);
};
```

Listing 6.2: Nutzung von Gruppen beim Energiemanagement

Für jede Anwendung wird als erstes die Breite der Bitmaske über ein `typedef` für `RunMode` festgelegt (Zeile 2). Danach folgt die Zuweisung der Bits zu den Gruppen. Im Beispiel werden Gruppen definiert, die von einer allgemein verwendbaren Klasse zur Steuerung von Schlaf- und Wachphasen (DutyCycleManager-DCM) benutzt werden (Zeilen 5 und 6). Eine Maske dient zur Selektion der Gruppen, die von der DCM Steuerung genutzt werden. In der `NodeConfiguration` wird dann für benutzte Komponenten spezifiziert, in welchen Modi sie aktiv sein sollen (Zeilen 20-23). Es ist zu beachten, dass für getrennte Anwendungsteile unterschiedliche Gruppen und Modi definiert und parallel benutzt werden können. Im Beispiel nutzt die Schlaf- und Wachphasensteuerung die Gruppenumschaltung für die DCM-relevanten Gruppen (Zeile 33). Ist ein `EnergyManageable`-Objekt wie `adc` (Zeile 23) in mehreren Gruppen, dann wird es erst deaktiviert, wenn keine dieser Gruppen mehr aktiv ist.

6.3.3 Lokales Energiemanagement

Hinter dem lokalen Energiemanagement verbirgt sich das selbstständige bedarfsgesteuerte An- und Abschalten von Mikrocontrollerkomponenten oder externen Geräten durch den Treiber. In vielen Fällen kann der Treiber entscheiden, ob eine Komponente aktiv sein muss. So werden beispielsweise sekundäre Unterbrechungen nur benötigt, wenn Daten zum Verarbeiten anliegen. Aber auch primäre Unterbrechungen können abgeschaltet werden, wenn das Ereignis nicht weiter behandelt wird. Abbildung 6.6 zeigt eine Anwendung, die eine serielle Schnittstelle zum Senden von Daten nutzt. Die serielle Schnittstelle wird nur schreibend benutzt, der Empfang kann daher automatisch deaktiviert werden.

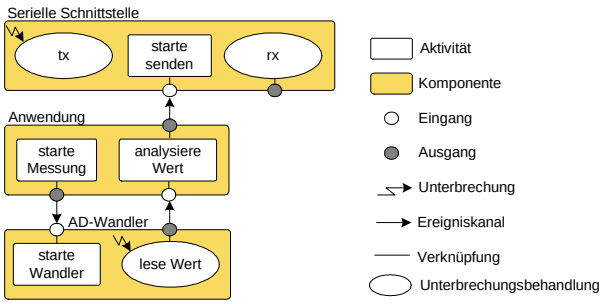


Abbildung 6.6: Nur in eine Richtung genutzte I/O-Schnittstelle

Da Komponenten dynamisch verknüpfbar sind, kann es auch zur Laufzeit dazu kommen, dass eine Unterbrechung nicht mehr genutzt wird. Die Lösung ist die Verknüpfung

des Komponentenausgangs, der durch die Behandlungsroutine benutzt wird, zu überwachen und entsprechend die Unterbrechung zu aktivieren oder zu deaktivieren.

Die Maßnahmen des lokalen Energiemanagements sind als Programmierrichtlinie zu verstehen. Da Treiber naturgemäß sehr unterschiedlich sind, wird in REFLEX auf weitere Abstraktionen und Schnittstellen für das lokale Energiemanagement verzichtet.

6.4 Evaluation

In diesem Abschnitt soll die Wirksamkeit der implementierten Energiesparmechanismen gezeigt werden. Trotz der weitgehend impliziten Wirkungsweise aus Sicht des Anwendungsprogrammierers soll das Energiemanagement effektiv sein.

6.4.1 Messaufbau

Der Energieverbrauch einer Schaltung kann nicht direkt bestimmt werden, sondern wird indirekt ermittelt. In Abbildung 6.7 ist der Versuchsaufbau dargestellt.

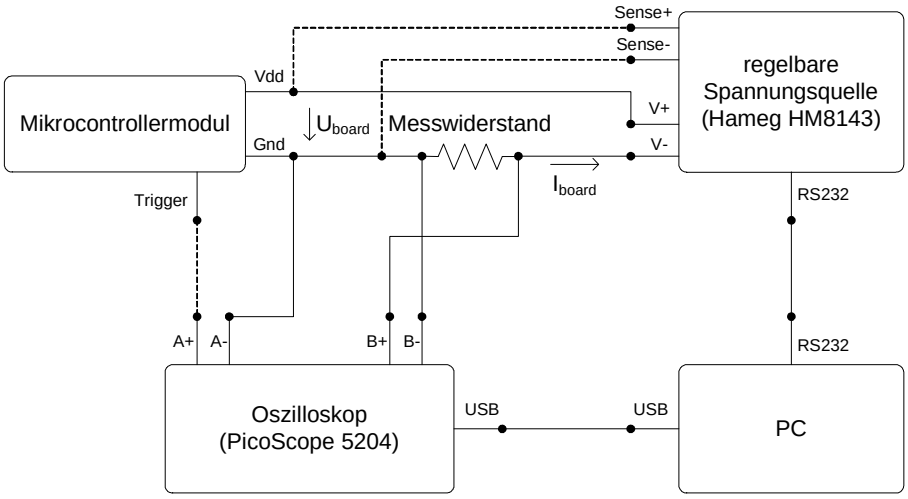


Abbildung 6.7: Messplatz für die Messungen zur Bestimmung des Energieverbrauchs

Neben dem jeweils betrachteten Mikrocontrollermodul wird ein Messwiderstand, ein Oszilloskop, ein Netzteil und ein PC verwendet. Der Energieverbrauch kann über die Formel 6.2 berechnet werden. Es ist das Integral der Leistungsaufnahme (U*I) über die Zeit.

$$E = \int U_{board}(t) * I_{board}(t)dt \tag{6.2}$$

Die Spannung U_{board} ist bekannt. Der Strom I_{board}, der in das Mikrocontrollermodul fliesst, wird indirekt über den Spannungsabfall am Messwiderstand mit bekannter Größe

gemäß Formel 6.3 bestimmt. Der Sense-Eingang des Netzgerätes und die Messeingänge des Oszilloskops sind hochohmig, daher fließt kein Strom über diese Anschlüsse. Das negative Vorzeichen des Spannungsabfalls über den Messwiderstand ist durch die Polung bei der Messung notwendig. Beide Oszilloskopkanäle müssen bei dem verwendeten Oszilloskop dasselbe Bezugspotential nutzen.

$$I_{board} = -U_{mess}/R_{mess} \tag{6.3}$$

Das Netzgerät hat einen Sense-Eingang, welcher es erlaubt, den Spannungsabfall über das Mikrocontrollermodul konstant zu halten. Die Spannung am Lastausgang des Netzgerätes schwankt je nach Lastsituation. Durch die Verwendung des Sense-Einganges kann der Messwiderstand größer gewählt werden. Dadurch steigt der Spannungsabfall über den Messwiderstand und somit die Messgenauigkeit. Ohne die Verwendung des Sense-Einganges schwankt zudem die am Mikrocontrollermodul anliegende Versorgungsspannung. Verwendet die untersuchte Plattform einen DCO (Digitally Controlled Oscillator) zur Takterzeugung, wie im Falle der TMoteSky Plattform [76], schwankt die Taktfrequenz mit der Versorgungsspannung. Das verfälscht die Messergebnisse und kann bei zu großen Schwankungen zum Versagen der Schaltung führen.

Optional ist die Nutzung eines Triggersignals vorgesehen. Das wird durch das auf dem Mikrocontroller laufende Programm erzeugt, wodurch eine exakte Bestimmung des Beginns einer Messung möglich ist. Der Nachteil ist, dass die Treiberstufen für Anschlüsse des Mikrocontrollers meist Leckströme aufweisen, welche die Messungen verfälschen. Jedoch ist dieser Einfluss konstant.

Der PC wird für die Ansteuerung des Oszilloskops und des Netzgerätes verwendet, zudem werden die vom Oszilloskop gemessenen Werte auf dem Rechner zur weiteren Verarbeitung gespeichert. Der Rechner ist galvanisch getrennt vom Mikrocontrollermodul, um die Messungen nicht zu verfälschen.

Plattformen

Als Plattformen für die verschiedenen Messungen kamen die TMoteSky-Plattform [76] und CardS12-Plattform [38] zum Einsatz. Erstere wurde als Prototyp für den Einsatz in Sensornetzwerken entwickelt und basiert auf einem Mikrocontroller der Texas Instruments MSP430-Reihe. Der Controller wird explizit für den Einsatz in Low-Power-Anwendungen beworben. Das implementierte Energiemanagement sollte daher von der Plattform profitieren können. Die CardS12 Plattform basiert auf dem Freescale HC(S)12 Mikrocontroller und ist für den Einsatz in Steuerungsanlagen konzipiert, in denen eine hohe Anzahl von Ein-/Ausgabesignalen benötigt wird. Der Controller stellt auch Energiesparmodi zur Verfügung, ist jedoch nicht so konsequent wie der MSP430 auf den Einsatz in Low-Power-Applikationen ausgelegt.

Der MSP430 verfügt über besondere Unterstützung für Low-Power Anwendungen und hat einen geringeren Stromverbrauch als der HC(S)12, besitzt aber auch weniger RAM. Er hat einen internen DCO (Digitally Controlled Oszillator) und kann daher ohne externen Quarz mit maximal 8 MHz bei 3 V betrieben werden.

Der HC(S)12 in der betrachteten Variante verbraucht relativ viel Strom, kann aber auch mit einer höheren Taktfrequenz als der MSP430 betrieben werden und hat eine höhere Treiberleistung an den I/O Pins (bis zu 25 mA pro Pin). Die Benutzung der Stromsparmechanismen ist einfach, es existieren spezielle Maschineninstruktionen zum Betreten der Stromsparmodi und jede interne Komponente kann einzeln aktiviert bzw. deaktiviert werden. Es kann auch festgelegt werden, ob diese sich automatisch abschalten, wenn der Sleep-Modus betreten wird. Für den Pseudo-Stop-Modus kann zusätzlich noch festgelegt werden, ob der externe Quarz mit der halben Spannung versorgt wird.

Mögliche Fehlerquellen

Die Messung weist naturgemäß Ungenauigkeiten auf. Die Ungenauigkeit des Oszilloskops liegt bei 3 %, die des Netzgerätes bei 0,2 %. Zusätzlich dazu ist der Messwiderstand abhängig von der Temperatur. Weitere Störungen ergeben sich durch Einkopplungen auf den Leitungen. Die mögliche zeitliche Auflösung ist abhängig von dem verwendeten Mikrocontrollermodul und wird im Wesentlichen durch die Größe der Kapazitäten an der Versorgungsleitung bestimmt, welche einen Glättungseffekt bei den Schwankungen der Versorgungsspannung hervorrufen.

6.4.2 Ermittlung der Einsparpotenziale

Für eine Bewertung der Wirksamkeit von Energiesparmechanismen wird zuerst für die betrachteten Plattformen der Energieverbrauch in den einzelnen Arbeitsmodi ermittelt. Dazu dient ein Programm, das die verschiedenen Arbeitsmodi durchläuft. In Abbildung 6.8 ist der Energieverbrauch für beide Plattformen dargestellt. Auf beiden Plattformen wurde jeweils nur ein Uhrenbaustein des Controllers genutzt, um zwischen den Arbeitsmodi zu wechseln.

Die Messungen zeigen, dass für beide Plattformen Unterschiede beim Energieverbrauch in den einzelnen Modi messbar sind, wobei jedoch die Auswirkungen der Schlafmodi auf Plattformebene unterschiedlich sind. Bei der CardS12-Plattform unterscheiden sich alle Arbeitsmodi im Verbrauch. Beim TMoteSky lassen sich die Modi lpm0 und lpm1 (von 0,35 s - 1 s) im Beispiel nicht unterscheiden, da aufgrund der Verwendung nur eines Uhrenbausteins die gleichen Teile des Controllers aktiv sind. Der Unterschied zwischen lpm2, lpm3 und lpm4 (ab 1s) ist nur bei Vergrößerung sichtbar, liegt aber schon im Bereich des Messfehlers.

Bei der CardS12-Plattform kann die Energieaufnahme von ca. 115 mW auf 70 mW gesenkt werden, bei der TMoteSky Plattform von 9 mW auf 0.1 mW. Die TMote-Platform verbraucht somit wesentlich weniger Energie und der relative Einfluss der Energiesparmodi des Controllers auf den Energieverbrauch des Boards ist größer als bei der CardS12-Plattform.

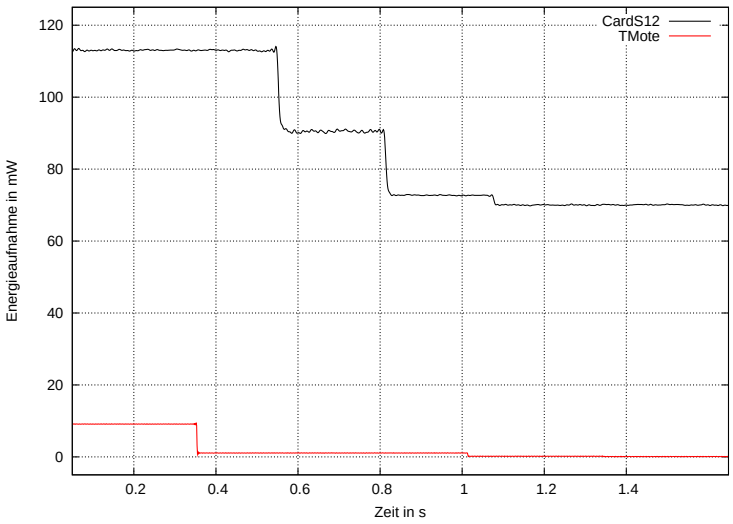


Abbildung 6.8: Energieverbrauch in den Schlafmodi in mW

6.4.3 Verhalten bei geringer Aktivität

Für energiegewahre Anwendungen, wie sie z. B. in Sensornetzen zur Anwendung kommen, kann ein geringer Aktivitätsgrad (Duty-cycle) angenommen werden. Als Beispielanwendung dient ein `HelloWorld`-Programm, dass jede Sekunde eine Nachricht über die serielle Schnittstelle ausgibt. Die Übertragungsrate beträgt 19200 Baud und es werden jede Sekunde 13 Zeichen übertragen, was etwa 7 ms in Anspruch nimmt.

Die Messungen wurden für beide Plattformen mit drei REFLEX-Varianten durchgeführt. Einmal wurde die Messung mit deaktiviertem Energiemanagement durchgeführt. Diese Messung dient vor allem als Referenz und zeigt wie sich die zusätzliche Aktivität (Senden der Nachricht) in einem aktiven System auswirkt. Die zweite Messung wurde mit einem einfachen Energiemanagement durchgeführt, welches in einer Nulllastsituation den Rechenkern deaktiviert. Dieser Schlafmodus kann bei allen Controllern betreten werden, ohne dass das Auswirkungen auf die Anwendung hat. Die dritte Messung wurde mit dem in Abschnitt 6.2 vorgestellten Energiemanagement durchgeführt. Abbildung 6.9 zeigt die Messergebnisse für die TMoteSky-Plattform bei 8 MHz Taktfrequenz und 2.2V Spannungsversorgung.

Den höchsten Energieverbrauch hat die Variante mit abgeschalteten Energiemanagement. Es ist zu sehen, dass bei aktiver CPU der Sendevorgang nicht zweifelsfrei anhand der gemessenen Daten identifiziert werden kann. Mit dem einfachen Energiemanagementschema sinkt der Energieverbrauch und die Sendephasen sind deutlich zu erkennen. Aufgrund der kurzen Dauer in Verbindung mit glättenden Effekten durch Kondensatoren steigt der Energieverbrauch in der Sendephase jedoch nicht auf das Niveau der Variante ohne Energiemanagement. Zudem wird zwischen dem Senden der einzelnen Bytes der Schlafmodus `lpm0` betreten. Durch die Nutzung des tiefst möglichen Schlafmodus ist nochmal eine deutliche Verringerung des Energieverbrauchs möglich. In den

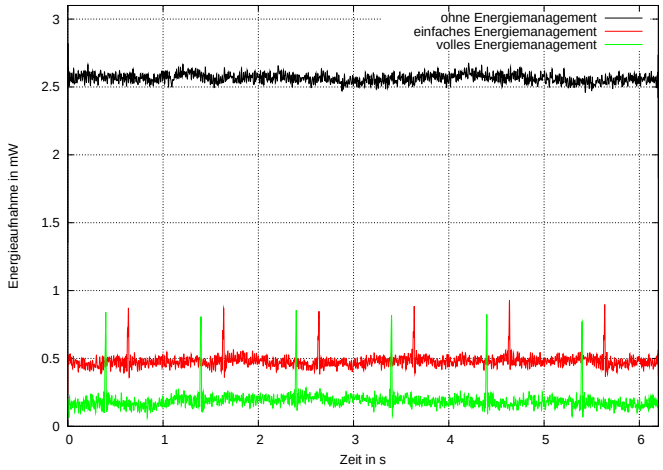


Abbildung 6.9: Energieverbrauch für HelloWorld auf der TMoteSky-Plattform

Sendephasen ist die Erhöhung des Energieverbrauchs etwa gleich zu der Erhöhung mit dem einfachen Energiemanagementschema.

In Abbildung 6.10 ist der Energieverbrauch der HelloWorld-Anwendung für die CardS12 Plattform zu sehen. Auch bei dieser unterscheiden sich die Energieverbräuche bei unterschiedlichem Energiemanagement deutlich. Im Vergleich zur TMoteSky-Plattform ist der Verbrauch aufgrund der höheren Versorgungsspannung und Taktfrequenz jedoch wesentlich höher. Zudem kann auf Boardebene der Energieverbrauch prozentual nicht so stark abgesenkt werden wie bei der TMoteSky-Plattform. Das liegt unter anderem daran, dass der externe Quarz nicht komplett abgestellt werden kann und sich noch Treiberbausteine für CAN-Bus und RS232 auf der Plattform befinden, welche ebenfalls nicht abgeschaltet werden können.

Bei den Messungen ist wie bei der TMoteSky-Plattform die Sendephase ohne eingeschaltetes Energiemanagement nicht erkennbar. Und auch die Verbesserungen beim Energieverbrauch durch die Energiesparmechanismen sind deutlich. Bei eingeschaltetem Energiemanagement sind neben den Sendephasen auch andere Aktivitäten erkennbar. Diese werden durch Unterbrechungen des verwendeten Zeitgebers ausgelöst. Letzterer kann nicht auf 1 s eingestellt werden, wodurch mehrfach Unterbrechungen zwischen den Sendephasen ausgelöst werden müssen.

Es fällt auf, dass in der Schlafphase der Energieverbrauch ca. 10 mW niedriger ist, als in der Messung in Abbildung 6.8. Das liegt daran, dass dort ein Triggersignal über einen I/O Pin generiert wurde. Die Treiberstufe war dadurch während der gesamten Messung aktiv, was erheblichen Einfluss auf die Energieaufnahme hat.

Tabelle 6.2 zeigt die durchschnittliche Leistungsaufnahme für verschiedene Konfigurationen. Die Werte zeigen den Einfluss des Energiemanagements auf den Energieverbrauch. Die größte relative Einsparung konnte bei 3 V auf dem TMoteSky erreicht werden, und senkt den Energieverbrauch auf ca. 3,7 % des Verbrauchs ohne Energiemanagement. Die geringste Einsparung wurde für das CardS12 Modul erreicht, dort konnte der

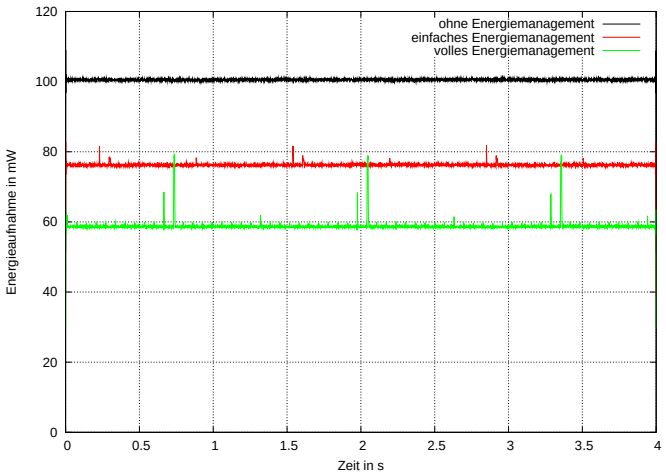


Abbildung 6.10: Energieverbrauch für HelloWorld auf der CardS12-Plattform

	Energiemanagement		
	ohne	einfach	voll
TMoteSky 2,2 V @ 8 MHz	0,19	0,48	2,56
TMoteSky 3 V @ 8 MHz	0,25	1,08	6,74
CardS12 5 V @ 16 MHz	100,5	76,0	59,3

Tabelle 6.2: Durchschnittliche Leistungsaufnahme für die `HelloWorld`-Anwendung

Energieverbrauch nur auf 59 % gesenkt werden. Diese höchst unterschiedlichen Einsparungen zeigen, dass es schwer möglich sein wird, den Effekt eines Energiemanagements im voraus zu bestimmen.

6.4.4 Einfluss von Taktfrequenz und Spannungsversorgung

Der Energieverbrauch eines Mikrocontrollers hängt in der Theorie linear von der Taktfrequenz und quadratisch von der Versorgungsspannung ab. Das ist jedoch eine idealisierte Formel, und spiegelt nicht die Gegebenheiten der einzelnen Plattformen wieder. Daher wurde für die `HelloWorld`-Anwendung untersucht, inwieweit sich Taktfrequenz und Versorgungsspannung auf den Energieverbrauch auswirken. Das Ergebnis ist in Abbildung 6.11 zu sehen.

In diesem Beispiel sollte sich die Taktfrequenz nur während der Sendephase auswirken, da die Takterzeugung in der Schlafphase inaktiv ist. Prinzipiell ist der Verbrauch bei 3 V höher als bei 2,2 V Spannungsversorgung. In der Schlafphase ist dieser Unterschied jedoch sehr gering. In der aktiven Phase unterscheiden sich die Ausschläge wie erwartet, bei 3 V @ 8 MHz ist der Ausschlag am höchsten, bei 2,2 V @ 1 MHz am geringsten. In Tabelle 6.3 ist der mittlere Energieverbrauch der einzelnen Varianten angegeben.

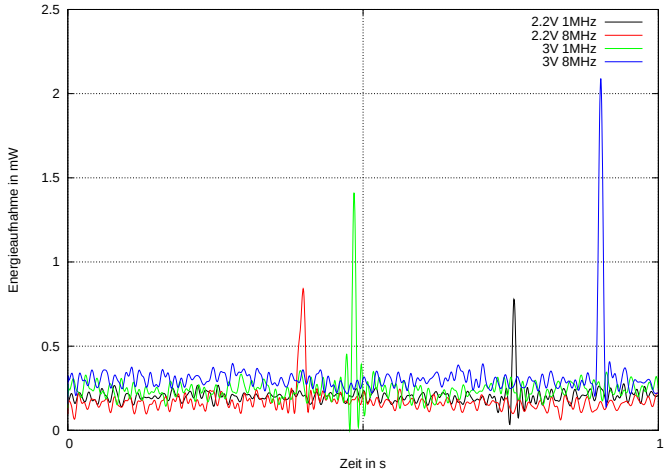


Abbildung 6.11: Einfluss von Taktfrequenz und Spannung auf den Energieverbrauch

	Reflex	
	1MHz	8MHz
TMoteSky 2,2V	0,19mW	0,17mW
TMoteSky 3V	0,25mW	0,31mW

Tabelle 6.3: Durchschnittliche Leistungsaufnahme für die `HelloWorld`-Anwendung

Durch die kurze Wachphase ist bei der 2,2 V Variante eigentlich kein Unterschied messbar. Die Abweichung von lediglich 0,2 mW ist hier nicht signifikant. Bei der 3 V Variante hingegen ist der Trend bereits ersichtlich. Für eine bessere Einschätzung wie sich Taktfrequenz und Versorgungsspannung in anderen Anwendungen auswirken, wurde die Messung nochmal mit deaktiviertem Energiemanagement durchgeführt. Der Rechenkern ist dadurch ständig aktiv. Das Ergebnis ist in Abbildung 6.12 dargestellt.

Die Messungen entsprechen nicht unbedingt dem erwarteten Ergebnis. Während die Erhöhung des Energieverbrauchs in etwa quadratisch zur Spannung erfolgt, wirkt sich die Taktfrequenz fast nicht aus. Der Grund ist im Wesentlichen, das im Kern kaum Umschaltvorgänge stattfinden, da sich die einzelnen Eingänge der kombinatorischen Blöcke nicht ändern. Lediglich in den Latch-Stufen (Speicher) finden Umschaltvorgänge statt. Diese Ergebnisse untermauern die These, dass Dynamic Frequency Scaling in tief eingebetteten Systemen nicht erfolgversprechend ist. Somit kann auch ein Energiemanagement wie das von Reflex, welches DFS nicht benutzt, effizient sein.

6.4.5 Vergleich mit TinyOS 2.x

Um die Effizienz des Energiemanagements in Reflex einschätzen zu können, ist noch ein Vergleich zu anderen Systemen notwendig. Als Referenzsystem bietet sich hier Ti-

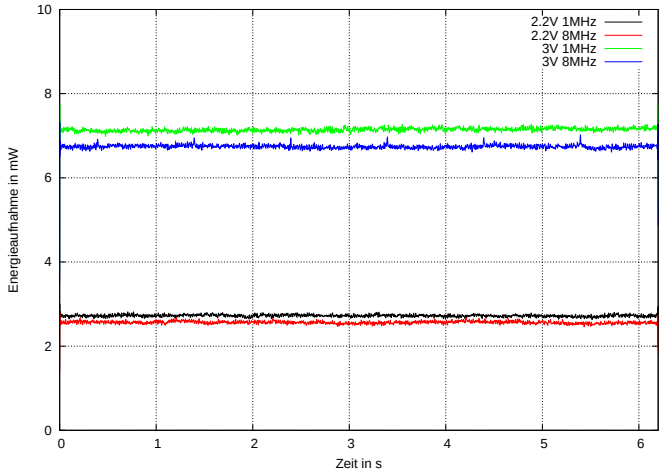


Abbildung 6.12: Einfluss von Taktfrequenz und Spannung auf den Energieverbrauch allgemein

nyOS 2.x an. Als Beispielanwendung wird wieder die `HelloWorld`-Anwendung genutzt, da diese Anwendung typische Verhaltensmerkmale einer Sense-and-Send-Anwendung für Sensornetze hat. Wie bei Reflex wurde bei der TinyOS Implementierung nur das implizite Energiemanagement genutzt. Abbildung 6.13 zeigt die Messergebnisse. Die Messungen wurden bei 1MHz Taktfrequenz durchgeführt, da dies die Standard Taktfrequenz von TinyOS ist.

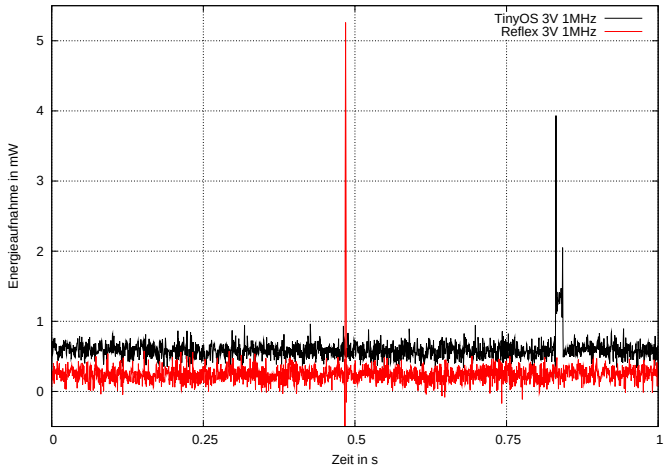


Abbildung 6.13: Energieverbrauch für HelloWorld für TinyOS und Reflex

Trotz gleicher Applikation unterscheiden sich die Messkurven deutlich. TinyOS verbraucht in der Schlafphase mehr Energie als Reflex. Das liegt daran, dass die serielle Schnittstelle unidirektional in Senderichtung genutzt wurde. In Reflex wird das berück-

sichtigt und die serielle Schnittstelle konnte automatisch deaktiviert werden, wenn nichts zu Senden anlag. Im Wesentlichen wird der höhere Energiebedarf bei TinyOS durch den ständig aktiven Baudratengenerator verursacht. Auch in der Wachphase unterscheiden sich beide Systeme. Bei Reflex ist die Wachphase nur so lang wie notwendig (5 ‰ Duty-Cycle) dafür ist der Energieverbrauch in der Wachphase größer. Der Grund dafür ist, dass nach Verlassen der Schlafphase im Gegensatz zu TinyOS der Baudratengenerator gestartet wird. Bei TinyOS ist die Wachphase länger als notwendig, das liegt daran, dass nicht sofort nach dem Beenden des Sendens der tiefstmögliche Schlafmodus berechnet wurde. Gut zu erkennen ist außerdem die aufwendige Berechnung des tiefstmöglichen Schlafmodus bei TinyOS am Ende der Wachphase.

	Reflex	TinyOS 2.x
TMoteSky 2,2 V @ 1 MHz	0,2 mW	0,31 mW
TMoteSky 3 V @ 1 MHz	0,25 mW	0,59 mW

Tabelle 6.4: Durchschnittliche Leistungsaufnahme für die `HelloWorld`-Anwendung

In Tabelle 6.4 ist die durchschnittliche Energieaufnahme für die `HelloWorld`-Anwendung in unterschiedlichen Konfigurationen zu sehen. Bei 2,2 V ist der Verbrauch unter TinyOS etwa 1,5 mal so groß wie unter Reflex. Bei 3 V ist der Verbrauch mehr als doppelt so groß. Der höhere Verbrauch begründet sich vor allem durch das Verhalten in der Schlafphase. Bei 3 V kommt der quadratische Einfluss der Spannung zum Tragen, da TinyOS nicht sofort in den Schlafmodus wechselt. Somit erhöht sich der Verbrauch in Relation zu Reflex gegenüber der 2,2 V Messung.

Prinzipiell ist somit gezeigt, dass das Energiemanagement von Reflex wirksam arbeitet. Die Ergebnisse zeigen aber auch, dass nicht ohne weiteres eine allgemeine Aussage über die Wirksamkeit in allen Anwendungen möglich sein wird. Es ist daher für die jeweiligen Anwendungen zu überprüfen, wie wirksam die Energiesparmechanismen sind, und welches Verhältnis von Spannung und Taktfrequenz am besten ist.

7 Fallstudien

Neben den in Abschnitt 1.2 aufgestellten Kriterien zur Bewertung des Ereignisflussmodells ist die Anwendbarkeit ein entscheidendes Merkmal. Im Folgenden soll durch Fallstudien gezeigt werden, dass diese gegeben ist. Zuerst wird eine selbst entwickelte komplexe Haussteuerung, welche als Demonstrator gedacht war, vorgestellt. Die Anlage ist seit mehreren Jahren in Betrieb. In den anderen zwei Fallstudien werden dagegen Konzeptimplementierungen vorgestellt, die jedoch das Potenzial des Ereignisflussmodells aufzeigen.

7.1 Haussteuerung

Parallel zur Entwicklung von Reflex enstand eine komplexe Haussteuerung, die sowohl das Heizsystem als auch die Alarmanlage eines Hauses steuert und seit mehreren Jahren in Betrieb ist. Den Kopf der Steuerung stellt ein CardS12-Mikrocontrollermodul [38] der Firma Elektronikladen dar. Der verwendete Mikrocontroller besitzt 4 KB RAM, 64 KB ROM und wird mit 8 MHz betrieben.

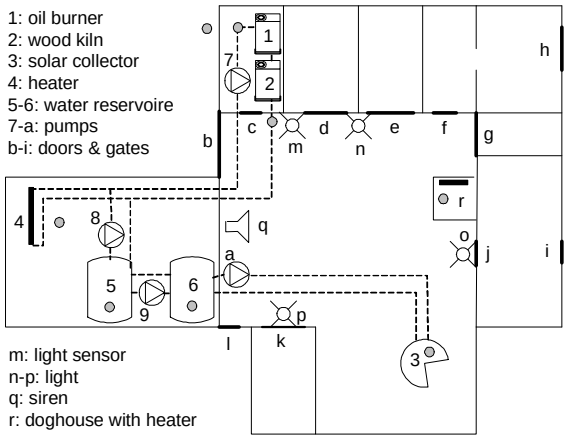


Abbildung 7.1: Grundriss Haus und Hof

In Abbildung 7.1 ist die Anlage schematisch dargestellt. Zum Heizungsteil gehören ein Ölbrenner (1), ein Holz-Kohle-Ofen (2) und ein Solarkollektor (3) zur Warmwassererzeugung. Über Pumpen (7-a) wird das Wasser zu den Heizkörpern (4) oder Warmwasserspeichern (5 und 6) gepumpt. Das gesamte System ist außerdem mit Temperatursensoren

versehen. Zum Alarmsystem gehören Kontakte an den Türen und Toren (b-i) des Grundstückes, sowie ein Lichtsensor(m), 3 Lampen (n-p) und eine Sirene (q). Zusätzlich wird auch noch die Hundehütte (r) mittels einer elektrischen Heizung frostfrei gehalten.

In Abbildung 7.2 ist der Ereignisflussgraph der gesamten Anwendung vereinfacht dargestellt. Es sind die Komponenten und die genutzten Ereigniskanäle zu sehen. Die hellen Komponenten sind Hardware-unabhängig, die dunklen sind Treiber für die verwendete Plattform.

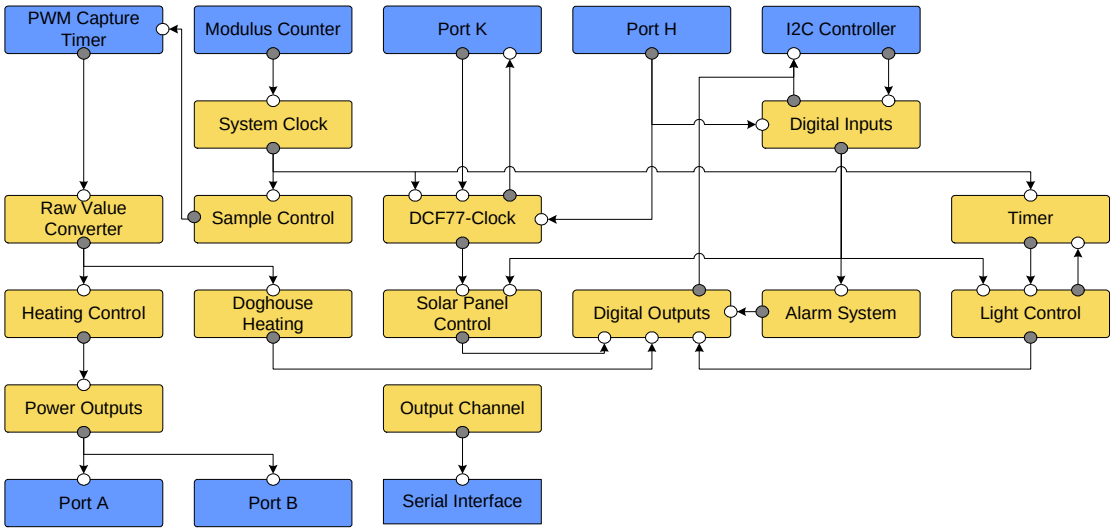


Abbildung 7.2: Ereignisflussgraph einer komplexen Haussteuerung

Eine einfache Komponente ist z. B. der `Raw Value Converter`, der den Duty-Cycle eines PWM-Signals (Pulsweiten Modulation) in eine Temperatur konvertiert. Dieser Wert stammt von den digitalen Temperatursensoren, welche über eine Multiplexerschaltung an den Mikrocontroller angeschlossen sind. Die Konversionskomponente ist von Anwendungsseite her zustandsfrei, da jeder Wert für sich betrachtet werden kann. Interessant ist, dass die Laufzeit relativ hoch ist, da Fließkommaoperationen genutzt werden, welche in Software implementiert sind.

Eine komplexe Komponente ist die `DCF77-Clock`, die mit einer externen Funkuhr kommuniziert und dafür einen Parallelport emuliert. Die Komponente enthält einen komplexen Zustandsautomat, die Aktionen sind jedoch von relativ geringer Laufzeit. Die Komponente hat die strengsten Echtzeitanforderungen in der Anwendung. Die externe Uhr erwartet Antwortlatenzen, die kürzer als 0.5 ms sind. Die von der Uhrenkomponente ermittelte Tageszeit wird sowohl von der Heizungslogik als auch im Alarmsystem verwendet. Die Echtzeitanforderungen der `DCF77-Clock`-Komponente verlangte eine Umstellung der Ablaufplanung von der einfachen unsortierten Abarbeitung zur prioritätsbasierten Abarbeitung während der Entwicklung. Der Grund war die steigende Komplexität der Anwendung und die daraus gestiegenen Latenzen für die Ausführung von Aktivitäten.

Komponente	ROM	RAM	max. Laufzeit
PWM Timer	1011	50	-
Sample Control	121	20	26
Raw-value Converter	3784	72	2811
Modulus Counter	161	6	-
Port AB	69	6	-
Port K	69	4	-
Port H	274	4	-
I2C Controller	1845	55	213
Serial Interface	959	19	192
Logical Clock	71	6	-
Digital Inputs	1755	50	982
DCF77	2210	100	2936
Heating Control	1797	48	2499
Solar Control	280	14	1348
Solar Tracking	1760	81	268
Alarmsystem	1074	18	117
Light Control	732	50	216
Power Outputs	1276	35	1243
Digital Outputs	1438	40	1131
Watchdog	286	8	25
Gesamt	23918	1906	—

Tabelle 7.1: Speicherverbrauch für die Haussteuerung

Die Heizungssteuerung an sich arbeitet bedarfsgetrieben, dass heisst, über die Rücklauftemperatur wird ermittelt, wie viel Wärme gerade im Haus absorbiert wird. Das funktioniert ähnlich zum Bajorath-Algorithmus [5] und hat eine Verringerung der Brennerstarts gegenüber vorlaufgetriebenen Heizsystemen zum Ziel. Das spart Energie, da der Ölbrenner in den Anlaufphasen einen schlechten Wirkungsgrad hat und die Vorlauftemperatur nur bei Bedarf hoch ist. Bei der konkreten Anlage muss zudem der Holz-/Kohleofen überwacht werden, über den zugeheizt werden kann, dieser ist jedoch nicht steuerbar. Als letzte Wärmequelle dient der Solarkollektor, welcher hauptsächlich zur Warmwassergewinnung genutzt wird. Zur Erhöhung des Wirkungsgrades wird der Kollektor der Sonne nachgeführt.

Der Alarmanlagenteil ist im Vergleich zur Heizungssteuerung weniger komplex. Für die einzelnen Türen kann eingestellt werden, wann signalisiert werden soll, dass diese offen sind. Zusätzlich wird, wenn eine Tür geöffnet wird und es dunkel ist, die Hofbeleuchtung

eingeschaltet. Nach dem Schließen der letzten Tür bleiben zwei der drei Lampen noch für 20 Sekunden an. Das ermöglicht zum einen, dass die Hofbeleuchtung bei Überqueren des Hofes und geschlossenen Türen an ist, zum anderen wird den Benutzern eine direkte Rückmeldung gegeben.

Eine Sonderstellung nimmt der Ausgabeteil ein, er ist in der Grafik nicht mit den Anwendungsteilen verbunden, da er von allen Teilen der Anwendung genutzt wird. Über die serielle Schnittstelle werden die Systeminformationen ausgegeben und bei Bedarf durch einen angeschlossenen Rechner dargestellt.

In Tabelle 7.1 ist der Speicherbedarf und die maximale Laufzeit der einzelnen Teile der Anwendung für diesen Mikrocontroller angegeben. Die Zahlen zeigen die unterschiedliche Komplexität der einzelnen Komponenten. Insgesamt benötigt die Anwendung etwa 24 KB ROM. Davon werden etwa 5 KB für die Fehlerausgabe und Kommunikation mit dem optional nutzbaren PC benötigt. Weitere 4 KB werden durch den Compiler für die Fließkommaberechnungen eingebunden. Nichtsdestotrotz wird nicht einmal die Hälfte des zur Verfügung stehenden Speichers (64 KB) benötigt. Für die Datenspeicherung werden knapp 2 KB benötigt, davon sind jeweils 512 Byte für Kommunikationspuffer und den Stapel reserviert.

Die Laufzeiten der Aktivitäten spiegeln die unterschiedliche Komplexität der Komponenten wieder. Interessant ist vor allem das Verhältnis zu den Systemlaufzeiten z. B. zur Ablaufplanung. Diese liegen je nach Planungsschema zwischen 11 und 645 Takten (siehe 5.3). Somit wird deutlich, dass im Ereignisflussmodell eine relativ feingranulare Aufteilung in Komponenten erfolgt.

Die Haussteuerung zeigt, dass es mit Reflex möglich ist, komplexe Steuerungssysteme für kleine leistungsschwache Systeme auf einem hohen Abstraktionsniveau zu implementieren. Weiterhin wird auch deutlich, wie wichtig z. B. das wählbare Ablaufplanungsschema ist, da das Ablaufplanungsverfahren noch während der Entwicklung umgestellt werden musste.

7.2 Reflex als Laufzeitumgebung für SDL-Programme

SDL (Specification and Description Language) [57] ist eine verbreitete formale Spezifikationssprache, die vorwiegend für die Beschreibung von Kommunikationsprotokollen genutzt wird. Um ein SDL-Programm ausführen zu können, wird eine Laufzeitumgebung benötigt. Typischerweise findet die Abbildung auf die Laufzeitumgebung durch eine Übersetzung der SDL-Beschreibung auf den Kode für das Zielsystem statt. Ein häufig verwendetes Werkzeug dafür ist der Telelogic Tau Compiler [100].

Im Folgenden soll exemplarisch an SDL gezeigt werden, wie Reflex als Laufzeitsystem für solche graphischen Sprachen genutzt werden kann.

7.2.1 Abbildung der SDL-Primitive auf das Ereignisflussmodell

Abbildung 7.3 zeigt die strukturellen Beschreibungselemente von SDL. Die oberste Hierarchiestufe stellt die Systembeschreibung dar, in der Blöcke miteinander verbunden wer-

den. Blöcke stellen logisch zusammengehörige Teile einer Anwendung dar und sind aus Prozessen aufgebaut, die wiederum miteinander verbunden werden. Über die Verbindungen werden Signale asynchron ausgetauscht. Ein Prozess ist im Wesentlichen ein Zustandsautomat. Ein Zustandsübergang wird meist durch ein empfangenes Signal ausgelöst oder ist unbedingt. Während des Zustandsübergangs können Prozeduren ausgeführt und Signale gesendet werden. Eine Prozedur ist dabei eine nicht blockierende Abfolge von Instruktionen mit definiertem Start und Endpunkt.

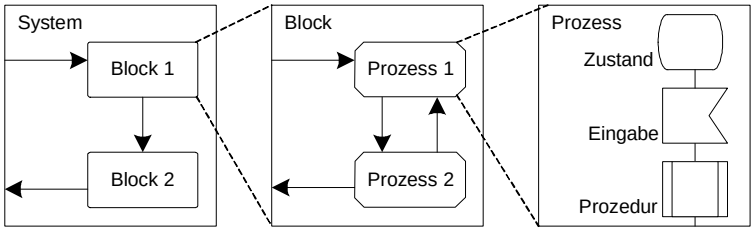


Abbildung 7.3: Beispielhafter SDL-Graph

Die strukturelle Beschreibung von SDL ähnelt der strukturellen Darstellung im Ereignisfluss. Dadurch ergibt sich eine einfache Möglichkeit zur Abbildung der Struktur. Blöcke können als Komponenten dargestellt werden, Prozesse als Aktivitäten und Signale als Ereignisse. Sowohl Signale als auch Ereignisse sind getypt.

Neben der Kommunikation der Signale gibt es in SDL synchrone entfernte Prozeduraufrufe. Diese müssen im Ereignisflussmodell mit Hilfe eines Zustandsautomaten dargestellt werden. Das kann zum Beispiel durch Nutzung des Protothread-Konzeptes [34] erfolgen. Diesem Ansatz kommt zu Gute, dass synchrone entfernte Prozeduraufrufe in SDL nur auf Prozessebene erlaubt sind, also nicht in verschachtelten Prozeduren aufgerufen werden können.

Zur Laufzeit ist es in SDL weiterhin möglich, dynamisch Prozesse zu erzeugen. Auch das ist in Reflex umsetzbar, eine Komponente muss dazu lediglich eine neue Aktivität instanziieren. Wahlweise kann der benötigte Speicherplatz von der Komponente vorgehalten oder eine Speicherverwaltung genutzt werden.

SDL ist somit gut auf das Ereignisflussmodell abbildbar. Die eventuell notwendige Priorisierung von Signalen und Prozessen wird nicht auf SDL-Ebene vorgenommen, dafür ist in SDL die Laufzeitumgebung zuständig. Im Ereignisfluss ist das direkt möglich, zudem gibt es dort die Möglichkeit Signale durch die Ereigniskanäle vorzuverarbeiten.

7.2.2 Erste Ergebnisse

In [103] wurde die Telelogic Tau Umgebung angepasst, um SDL-Programme in Reflex-Programme zu transformieren. In Tabelle 7.2 sind die erreichten Laufzeiten für die Kommunikation zwischen zwei Prozessen angegeben. Zum Vergleich sind die Laufzeiten für die Standardumgebung von Telelogic, welche auf ECOS basiert, angegeben. Die Imple-

mentierung unterscheidet sich im Wesentlichen bei der Signalübertragung, die Prozesse sind ansonsten identisch. Als Plattform kam ein Leon2 Mikrocontroller [84] zum Einsatz.

Anzahl der Signale	ECOS	Reflex
1000	0,12s	0,05s
3000	0,36s	0,14s
8000	0.98s	0,40s

Tabelle 7.2: Laufzeiten für die Versendung von Signalen (entnommen aus [48])

Die Reflex-basierte Variante ist in etwa doppelt so schnell wie die Ursprüngliche. Der Grund ist im Wesentlichen der, dass in der Standardumgebung eine einzige `switch`-Anweisung dafür genutzt wird, alle Signale zuzustellen. Diese `switch`-Anweisung wird größer und langsamer mit der Anzahl der verwendeten Signale. In Reflex hingegen werden Ereigniskanäle für die direkte Zustellung benutzt, wodurch ein Signal nicht erst an eine zentrale Stelle zur Auswertung übermittelt wird.

In Tabelle 7.3 ist der Speicherverbrauch für die SDL-Anwendung unter Reflex in Bytes angegeben. Das Grundsystem ist etwa 10 KB groß, wovon allein die Interruptvector-Tabelle 4 KB einnimmt, zudem wird etwa 1 KB für Fehlerausgaben benötigt. Die notwendige Integrationsschicht für SDL ist etwa 4,5 KB groß, dabei muss jedoch beachtet werden, dass noch nicht alle SDL-Konzepte umgesetzt wurden. Es fehlen z. B. die dynamische Prozesserzeugung, dynamische Signalpfade und entfernte Prozeduraufrufe.

	Kode	Daten	BSS
Betriebssystem	9852	272	148
Integrationsschicht	4592	236	16
SDL-Anwendungslogik	1172	472	160
SDL-Wrapper	1892	184	136
gesamt	20056	704	460

Tabelle 7.3: Speicherverbrauch der SDL-Anwendung (entnommen aus [48])

Speziell für die SDL-Anwendung wurden ca. 3 KB Kode generiert. Dieser setzt sich zusammen aus 1,2 KB für die Anwendungslogik und 1,9 KB für den Kapselungskode (Wrapper). Der Speicherverbrauch für Daten ist gering. Entscheidend für die Eignung ist jedoch der kleine Anteil der Daten für die Kapselung gegenüber dem Anteil der Anwendungsdaten.

Es ist ersichtlich, dass mit Reflex als effiziente Laufzeitplattform für SDL-Anwendungen verwendet werden kann. Zudem zeigen die Ergebnisse auch, dass Reflex, obwohl es für tief eingebettete Systeme entwickelt wurde, sich auch als Laufzeitumgebung in leistungsfähigeren Systemen eignet.

7.3 Integration in ein SPS-System

Speicherprogrammierbare Steuerungen (SPS) sind der Status quo in Anlagensteuerungen. Sie werden schon seit den 80er Jahren eingesetzt und zielen darauf ab, elektrische, mechanische bzw. pneumatische Anlagensteuerungen weitgehend durch Elektronik zu ersetzen. Man verspricht sich dadurch geringere Kosten, eine höhere Flexibilität und größere Zuverlässigkeit [46].

7.3.1 Programmierung von SPS-Systemen

Für den Entwickler werden mehrere einfache Sprachen zur Programmierung dieser Anlagen bereitgestellt. Prinzipiell gibt es in jeder Darstellung Ein- bzw. Ausgänge, welche über eine Funktion verknüpft sind. Es gibt zwei textuelle Sprachen, **Anweisungslisten** und **strukturierten Text**. Die erste Sprache ist eine Art Assembler, die zweite ist etwas abstrakter und erlaubt die Nutzung häufiger Anweisungselemente wie `if`-Bedingungen. Die anderen Sprachen sind alle graphisch. **Kontaktpläne** sind an Stromlaufpläne angeleht und erlauben eine Und-/Oder-Strukturierung von Schaltelementen. **Funktionsbausteine** sind Funktionen der boolschen Algebra, die Abarbeitung erfolgt zeilenweise von links oben nach rechts unten. Die sogenannte **Ablaufsprache** ist eine Petrinetzdarstellung und erlaubt beispielsweise eine einfache Fehleranalyse, wird aber selten verwendet.

Die komplexeste Sprache wird **Continous Function Chart** (kurz CFC) genannt und stellt eine Erweiterung der **Funktionsbaustein**-Sprache dar. Sie erlaubt den Einsatz komplexer Funktionsbausteine, welche in einer beliebigen anderen SPS-Sprache implementiert sind. CFC eignet sich daher besonders für die Strukturierung komplexer Anwendungen. CFC wird häufig verwendet, ist aber nicht im Standard EN61131 [18] für SPS-Steuerungen enthalten. CFC zeigt, dass mit wachsender Komplexität der Anlagen, ein Bedarf für die abstrakte Programmierung von SPS-Anlagen besteht. Eine bestehende Einschränkung ist beispielsweise, dass es keine lokalen Variablen in den Funktionsblöcken gibt. Dadurch muss ein Funktionsblock in abgeänderter Form implementiert werden, um ihn mehrfach zu nutzen. Aber auch bei der Abarbeitung gibt es Einschränkungen. Die meisten SPS-Systeme arbeiten streng zyklisch, das gesamte Programm wird in jedem Zyklus in einer festen Folge komplett abgearbeit.

7.3.2 Reflex als SPS-Laufzeitumgebung

Bis auf die Continous Function Charts sind alle SPS-Sprachen sehr einfach und die Logik kann auf Aktivitäten bzw. Komponenten abgebildet werden. Bei CFCs ist eine Abbildung von Funktionsblöcken auf Komponenten sinnvoll. Das Ereignisflussmodell könnte als Nachfolger von CFC fungieren. Zum einen kann bisher geschriebene Software automatisch ins Ereignisflussmodell überführt werden, zum anderen können neue Abstraktionen wie lokale Variablen und auch C++ als Programmiersprache genutzt werden.

Die bisherige streng zyklische Abarbeitung kann durch die Benutzung der statisch zeitgesteuerten Ablaufplanung umgesetzt werden. Da diese Abarbeitung jedoch nicht im Standard vorgeschrieben ist, können auch andere Planungsschemen verwendet werden.

Das senkt unter Umständen die Rechenlast des Systems erheblich. Zudem erfolgt die Abarbeitung in Reflex nicht interpretativ wie in SPS-Anlagen.

Eine interessante Variante der Nutzung von Reflex ist die als Subsystem. SPS-Anlagen sind meist hierarchisch aufgebaut, wobei die einzelnen Knoten über standardisierte Bussysteme miteinander kommunizieren. In [40] wurde eine solches Subsystem zur Steuerung eines Glasschmelzofens mit Hilfe von Reflex implementiert. In Abbildung 7.4 ist der Glasschmelzofen schematisch dargestellt.

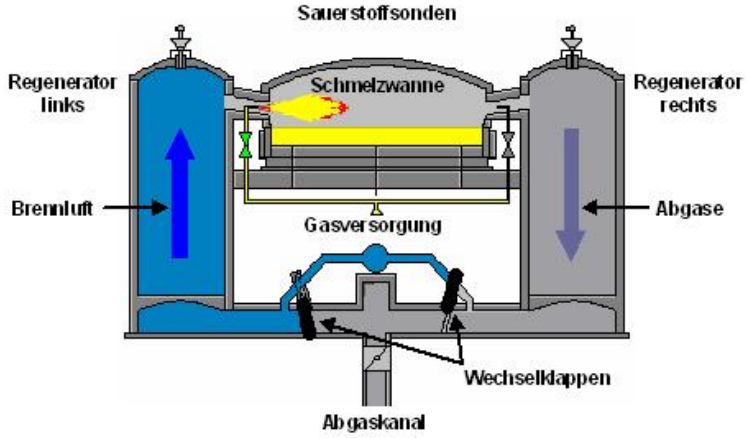


Abbildung 7.4: Schematische Darstellung eines Glasschmelzofens (entnommen aus [40])

Die Schmelzwanne wird wechselseitig (links/rechts) befeuert. Die Abluft erwärmt dabei die Regeneratoren, über die nach Umschaltung der Feuerseite die Zuluft strömt und dadurch vorgewärmt wird. Zum Umschalten müssen die Wechselklappen und die Brenner angesteuert werden. Die Temperaturüberwachung an verschiedenen Stellen erfolgt über verschiedene Temperaturfühler. Zusätzlich existieren noch analoge Sauerstoffsonden für die Messung des Sauerstoffgehaltes in der Ablauft. Das Subsystem übernimmt die Ansteuerung der Anlagenteile und übermittelt über eine Profibusanbindung den aktuellen Zustand an die Kopfstation. Diese sendet wiederum Befehle zum An- bzw. Abschalten der Feuerung.

In Abbildung 7.5 ist der Ereignisflussgraph der Schmelzwannensteuerung zu sehen. Diese ermittelt zyklisch die Temperaturen und Sauerstoffkonzentrationen, zudem werden Änderungen an den digitalen Eingängen registriert. Kommt ein Profibuspaket von der Kopfstation, wird dieses von dem `Profibus Paketverteiler` ausgewertet und an die lokal zuständige Komponente geleitet. Die Pakete sind in dem Fall Befehle, die ermittelten Daten der Kopfstation zu übermitteln oder die Digitalausgänge zu setzen. Sind Daten zu übermitteln, werden diese der Komponente `Profibus Paketaufbau` übergeben. Die packt die Daten in ein Profibuspaket und übergibt es der `Profibus Schnittstelle` zum Versenden. Für eine Nutzung in anderen Anlagen müssen lediglich die Ansteuerungskomponenten ausgetauscht werden.

Unter Reflex konnte für die Steuerung eines Glasschmelzofens ein einfaches Mikro-

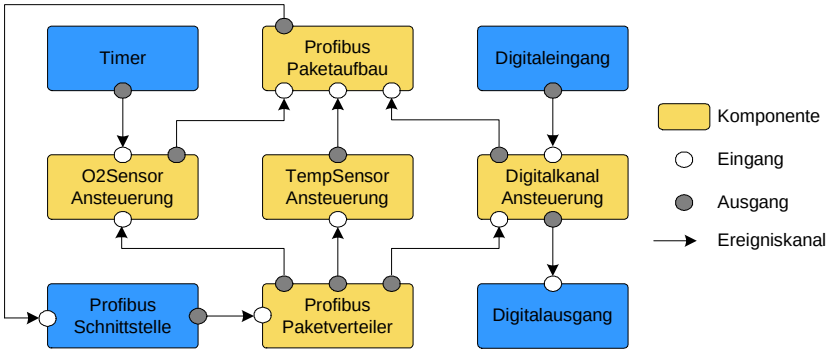


Abbildung 7.5: Ereignisflussdiagramm der Schmelzwannensteuerung

controllerboard auf Basis eines Atmel ATmega128 genutzt werden, welches über 64 KB ROM und 10 KB RAM verfügt und mit 16 MHz betrieben wird. Die Kodegröße beträgt etwa 13 KB und es wird 1 KB RAM benötigt.

Dieses Fallbeispiel zeigt, dass das Ereignisflussmodell in SPS-Anlagen einsetzbar ist und einen Fortschritt bei der Programmierung bedeuten kann, da neue Abstraktionen zur Programmierung zur Verfügung stehen.

8 Zusammenfassung

In dieser Arbeit wurde ein ereignisbasiertes Betriebssystemkonzept für tief eingebettete Steuersysteme vorgestellt. Das Konzept zeichnet sich dadurch aus, dass viele Aspekte, die bei der Programmierung von eingebetteten Systemen eine Rolle spielen, berücksichtigt wurden. Um dies zu erreichen, war eine breite Betrachtung des Themas und eine Identifikation von zu berücksichtigenden Kriterien erforderlich. Zudem fand eine eingehende Analyse existierender Ansätze und Systeme statt. Auf Grundlage dieser Betrachtungen wurde erst das Ereignisflussmodell und darauf aufbauend ein konkretes Betriebssystem entwickelt. Es fand zudem eine Evaluierung statt, die durch Fallstudien die Anwendbarkeit des Konzeptes belegt und eine Bewertung anhand der aufgestellten Kriterien ermöglicht.

Die Kernfrage, ob es ein System geben kann, dass alle aufgestellten Kriterien erfüllt, ist abschließend mit ja zu beantworten. Im Folgenden wird eine zusammenfassende Bewertung zur Erfüllung dieser Kriterien vorgenommen. Anschließend wird noch ein Ausblick gegeben, in welche Richtung das Konzept und auch Reflex weiterentwickelt werden können.

8.1 Erfüllung der Kriterien

Mit dem Ereignisflussmodell ist es möglich, ein System zu entwickeln, welches alle in Abschnitt 1.2 benannten Kriterien erfüllt. Tabelle 8.1 zeigt das Erreichte im Vergleich zu existierenden Ansätzen. Das Ereignisflussmodell ist der Schlüssel zur Erfüllung aller Kriterien.

Reflex ist modellbasiert, da es dem Anwendungsprogrammierer direkt die im Ereignisfluss genutzten Abstraktionen zur Verfügung stellt. Zudem ist die direkte Abbildunge anderer Modelle, wie beispielsweise SDL, auf das Ereignisflussmodell möglich. Durch die Fallstudien konnte auch gezeigt werden, dass die Abstraktionen des Ereignisflussmodells adäquat für das gewählte Einsatzgebiet sind. Nicht zuletzt kann das Ereignisflussmodell als eine Object-State-Machine verstanden werden, welche in [61] als ein vielversprechendes Programmiermodell für Sensornetze angesehen wird.

Die Echtzeitfähigkeit ist gegeben, da das System statisch analysierbar ist, was aus der Erfüllung des Ravenscar-Profils [15] resultiert. Außerdem ist die präemptive Abarbeitung möglich, was in vielen Arbeiten als maßgebliches Kriterium für die Implementierung echtzeitfähiger Systeme angesehen wird. Von den in Kapitel 2 betrachteten Systemen verfügen nur Thread-basierte über die Möglichkeit zur präemptiven Abarbeitung, obwohl bereits in [7] gezeigt wurde, dass dies auch in rein ereignisgetriebenen Systemen möglich ist. Der Thread-basierte Ansatz hat dabei den Nachteil, dass ohne weitere Ein-

	Reflex	AVRx	BoldStroke	Chimera I &II	Contiki	ECOS	Emeralds	FreeRTOS	Mantis	ORK	OSEK	Pure	RETOS	SOS	Spring	TinyOS 2.x	TinyGals
Modellbasiert	+	-	+	+	o	-	-	-	-	o	o	-	-	o	+	o	+
Echtzeitunterstützung	+	o	+	+	-	+	+	o	-	+	+	o	-	-	+	-	-
Abarbeitungsrahmenwerk	+	-	+	-	-	+	-	-	-	+	o	+	-	-	+	o	o
Implizite Synchronisation	+	-	+	+	+	-	-	-	-	-	-	o	-	-	+	-	+
Energiesparmechanismen	+	-	-	-	-	-	-	-	-	-	-	-	-	+	-	+	+
Speicherverbrauch	+	o	-	+	+	-	-	o	o	o	+	+	o	+	o	+	+
Standardwerkzeugkette	+	+	-	+	+	+	+	+	+	+	+	+	+	+	-	-	-

Tabelle 8.1: Eigenschaften existierender Systeme für Steuerungsanwendungen

schränkungen bzw. genauere Informationen über die Anwendung die Echtzeitanalyse durch Präzedenzprobleme erschwert wird.

Ein Abarbeitungsrahmenwerk zu implementieren ist in Thread-basierten Systemen Stand der Technik, jedoch nicht in ereignisgetriebenen Systemen. Das Ereignisflussmodell erlaubt die Implementierung eines Abarbeitungsrahmenwerks, bei dem der Anwendungskode bis auf die Konfiguration absolut unabhängig vom aktuell gewählten Abarbeitungsschema ist. Neben der Wahl zwischen einfacher unsortierter, statisch prioritätsbasierter und dynamisch Zeitschranken-basierter Abarbeitung ist es auch möglich, eine statisch zeitgesteuerte Abarbeitung vorzunehmen. Nicht zuletzt ist die Implementierung in Reflex leichtgewichtig genug für die anvisierten tief eingebetteten Systeme.

Mit dem Ereignisflussmodell ist die implizite Synchronisation einer Anwendung möglich. Der Grund ist, dass die Synchronisationspunkte direkt aus dem Ereignisflussgraphen abgeleitet werden können. Selbst ohne Werkzeugunterstützung kann eine Anwendung durch einfaches synchronisieren aller Ereigniskanäle fast vollständig synchronisiert werden. Lediglich wenn Nebenläufigkeiten zwischen Aktivitäten und Unterbrechungsbehandlungen auftreten oder bei bewussten Nebenläufigkeiten zweier Aktivitäten innerhalb einer Komponente muss von Hand synchronisiert werden. Aber selbst in diesen Fällen ist durch den klar abgegrenzten Einflussbereich - die Komponente - eine automatische Synchronisation möglich.

Auch das Energiemanagement kann weitgehend implizit erfolgen. Möglich ist das durch die einfache Erkennbarkeit von Nulllastsituationen. In Thread-basierten Systemen sind zusätzliche Informationen über die Threads erforderlich. Zudem kann in Reflex der tiefste mögliche Schlafzustand effizient ermittelt werden, wie in Abschnitt 6.3.1 gezeigt wurde. Außerdem existiert die Möglichkeit, gruppenweise die Komponenten der Anwendung zu verwalten. Dabei ist ein großer Vorteil, dass der funktionale Anwendungskode, wie schon bei der Wahl des Abarbeitungsschemas, nicht davon beeinflusst wird.

Obwohl das Ereignisflussmodell stark von der Hardware abstrahiert, lassen sich An-

wendungen effizient umsetzen. Reflex kann ohne Weiteres mit anderen sehr leichtgewichtigen Systemen wie z. B. TinyOS [54] konkurrieren. Ein Vorteil gegenüber Thread-basierten Systemen ist die Nutzung von nur einem Stapel, dessen maximale Tiefe sich zudem durch die Anzahl von verwendeten Privilegstufen beeinflussen lässt. In Thread-basierten Systemen ist die Stackverwaltung und Platzabschätzung ein nichttriviales Problem.

Es wurde gezeigt, dass das Ereignisflussmodell mit Standardwerkzeugen nutzbar ist. C++ bietet sich als objektorientierte Sprache an und erlaubt zudem die Template-basierte Programmierung. Die Entwickler von TinyOS [54] und TinyGals [22] hatten die Vermutung, dass für die Nutzung komplexer Modelle in tief eingebetteten Systemen eine domänenspezifische Sprache notwendig ist.

8.2 Ausblick

Für die weitere Entwicklung von Reflex ist eine graphische Entwicklungsumgebung unabdingbar, da ein Ereignisflussgraph am besten graphisch dargestellt wird. Mit dieser IDE sollte neben dem reinen Entwurf auch die Echtzeitanalyse möglich sein. Dazu müssen die bereits zur Echtzeitanalyse entwickelten Programme Laufzeitextraktor [90] und Real Time Simulator [53] vervollständigt und anschließend integriert werden. Das erste Programm erlaubt die weitgehend automatische Extraktion von Laufzeiten der Elemente eines Ereignisflusses. Das zweite Programm führt eine Echtzeitanalyse bzw. Simulation auf Modellebene aus. Eine weitere Möglichkeiten der IDE könnte auch die automatische Parametrisierung und Synchronisation auf Modellebene sein.

Auf Seiten der Werkzeugunterstützung kann auch darüber nachgedacht werden, wie Verfahren der Echtzeitanalyse auf die Analyse des Energieverbrauchs angepasst werden können. Diese Problematik ist eng verbunden mit der Echtzeitanalyse, da auch der Energieverbrauch lastabhängig ist. Zudem erlaubt die strukturelle Information des Ereignisflussgraphen eine genaue Abschätzung, wann welcher Teil eines Mikrocontrollers genutzt wird. So wird ein AD-Wandler nur aktiv sein, wenn dieser genutzt wird. Als Information wird lediglich benötigt, ob Ereignisse für den AD-Wandler vorliegen oder nicht.

Für die breite Verwendung von Reflex in sicherheitskritischen Systemen muss es zukünftig möglich sein, Komponenten gegen illegale Speicherzugriffe zu schützen. Konzeptionell ist durch den Ereignisflussgraphen einer Anwendung bekannt, welche Komponente auf welchen Speicher zugreifen kann. Dies muss aber noch durch geeignete Mechanismen im Laufzeitsystem sichergestellt werden, damit auch Komponenten Dritter gefahrlos benutzt werden können.

Das Ereignisflussmodell wurde zwar speziell für tief eingebettete Systeme entwickelt, jedoch erscheint auch eine Nutzung in leistungsfähigen Umgebungen sinnvoll. So ist es vorstellbar, Reflex als Laufzeitsystem für GUIs (Graphical User Interface) einzusetzen. Diese sind ebenfalls ereignisbasiert und könnten von einer leichtgewichtigen Laufzeitumgebung profitieren.

Eine weitere interessante Perspektive ist die Implementierung eines verteilten Lauf-

zeitsystems. Die vorgegebene Struktur eines Ereignisflussgraphen ermöglicht zum einen die transparente Weiterleitung eines Ereignisses zu anderen Knoten, zum anderen ist eine automatische Parallelisierung und dynamische Verteilung von Komponenten denkbar.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] A.Burg. The eventflow model - a concept for real-time control of intelligent autonomous systems. Techreport mip9713, TU Passau, 1997.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[3] K. Albers, F. Bodmann, and F. Slomka. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. *Proceeding of th 18th Euromicro Conference on Real-Time Systems (ECRTS)*, volume 0:pages 97–106, 2006.

[4] AVRX. *Projektwebseite http://www.barello.net/avrx [2008-09-09].*

[5] R. Bajorath. Hintergrundinformationen zur verfahrenstechnik. Technical report, Bajorath Systemhaus für Regelungstechnik und Hydraulik GmbH, 2005.

[6] T. Baker. Opening up ada-tasking. *In ACM Journal Ada Letters*, Volume X(Number 9):pages 60–64, 1990.

[7] T. P. Baker. Stack-based scheduling for realtime processes. *In Springer Journal Real-Time Systems*, volume 3(number 1):nages 67–99, 1991.

[8] M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni-Vincentelli. Efficient embedded software design with synchronous models. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT '05)*, pages 187–190, New York, NY, USA, 2005. ACM.

[9] M. Baleani, A. Ferrari, L. Mangeruca, A. L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct-by-construction transformations across design environments for model-based embedded software development. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1044–1049, Washington, DC, USA, 2005. IEEE Computer Society.

[10] D. Beuche, A. Guerrouat, H. Papajewski, W. Schroder-Preikschat, O. Spinczyk, and U. Spinczyk. On the development of object-oriented operating systems for deeply embedded systems - the PURE project. In *Proceedings of the Workshop on Object-Oriented Technology*, page 26, 1999.

[11] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, volume 10(number 4):pages 563–579, 2005.

[12] C. Bickford, M. Teo, G. Wallace, J. Stankovic, and K. Ramamritham. A robotic assembly application on the spring real-time system. *Proceedings of the second IEEE Real-Time and Embedded Technology and Applications Symposium*, volume 00, 1996.

[13] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Techreport, Oregon Graduate Institute School of Science & Engineering, 2002.

[14] M. Bordin and T. Vardanega. Automated model-based generation of ravenscar-compliant source code. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS '05)*, volume 00, pages 59–67, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[15] A. Burns. The ravenscar profile. *Ada Lett.*, XIX(4):49–52, 1999.

[16] A. Burns and A. J. Wellings. Restricted tasking models. In *IRTAW '97: Proceedings of the eighth international workshop on Real-Time Ada*, pages 27–32, New York, NY, USA, 1997. ACM.

[17] G. C. Buttazzo. Rate monotonic vs. edf: Judgment day. In *Proceedings of the 3rd International Conference on Embedded Software 2003*, pages 67–83, 2003.

[18] CENELEC. En norm für speicherprogrammierbare steuerungen 61131. Technical report, European Committee for Electrotechnical Standardization, 2004.

[19] A. Cervin. Improved scheduling of control tasks. *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, volume 00, 1999.

[20] A. Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, Apr. 2003.

[21] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th intl. conf.on Information processing in sensor networks (IPSN '07)*, Cambridge (USA), 2007.

[22] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: A programming model for event-driven embedded systems. In *Proceedings of ACM Symposium on Applied Computing*, pages 698–704, 2003.

[23] E. Cheong and J. Liu. galsc: A language for event-driven embedded systems. Technical Report UCB/ERL M04/7, EECS Department, University of California, Berkeley, 2004.

[24] Y. Cho and N. Chang. Memory-aware energy-optimal frequency assignment for dynamic supply voltage scaling. In *Proceedings of International Symposium on Low Power Electronics and Design (ISPLED '04)*, 2004.

[25] Y. Cho, Y. Kim, and N. Chang. Pvs: passive voltage scaling for wireless sensor networks. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISPLED '07)*, pages 135–140, New York, NY, USA, 2007. ACM Press.

[26] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems ISBN 1-58053-327-2.* Artech House, 2002.

[27] C. Davey and J. Friedman. Software systems engineering with model-based design. In *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[28] J. A. de la Puente, J. Zamorano, J. Ruiz, R. Fernández, and R. García. The design and implementation of the open ravenscar kernel. *In ACM Journal Ada Letters*, volume XXI(number 1):pages 85–90, 2001.

[29] DIN61508. *Normenreihe DIN EN 61508 (VDE 0803): Funktionale Sicherheit - Sicherheitssysteme (E/E/PES) , Deutsche Kommission Elektrotechnik Elektronik Informationstechnik, 2004.*

[30] DO-178B. *Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc. , 1992.*

[31] D.Rakhmatov and S.Vrudhula. Energy management for battery-powered embedded systems. In *ACM Transactions on Embedded Computing Systems Vol.2, No.3*, 2003.

[32] R. G. Dromey. Cornering the chimera. *In Journal IEEE Software*, volume 13(number 1):pp.33–43, January 1996. Showing the importance of methods/models for software engineering.

[33] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding preemption to tinyos. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92, New York, NY, USA, 2007. ACM.

[34] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (EmNets '04)*, 2004.

[35] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, Boulder, 2006.

[36] R. H. E. Trumpler. A systematic framework for evolving tinyos. In *Proc. of IEEE Workshop on Embedded Networked Sensors (EmNets)*, 2006.

[37] ECOS. *Projektseite http://ecos.sourceware.org/ [2008-09-09].*

[38] Elektronikladen. *CardS12 Prodktwebseite http://elmicro.com/de/cards12.html [2008-09-09].*

[39] S. C. Ergen and P. Varaiya. Energy efficient routing with delay guarantee for sensor networks. *In Kluwer Journal on Wireless Networks*, volume 13(number 5):pp.679–690, 2007.

[40] M. Fabisz. Seminar paper: Integration von reflex basierten steuerungs-/sensormodulen in industrielle sps-umgebungen, 2008.

[41] FreeRTOS. *Projektseite http://www.freertos.org/ [2008-09-09].*

[42] P. Frenger. Embed with forth. *In ACM Journal SIGPLAN Notices*, volume 39(number 8):pp. 8–11, 2004.

[43] P. Ganesan and A. G. Dean. Enhancing the avrx kernel with efficient secure communication using software thread integration. *In Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)*, volume 00:pp. 265, 2004.

[44] D. Gay, P. Levis, and R. von Behren. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design an Implementation*, June 2003.

[45] R. Ghattas and A. G. Dean. Energy management for commodity short-bit-width microcontrollers. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 32–42, New York, NY, USA, 2005. ACM Press.

[46] W. Giessler. *Simatic S7 SPS-Einsatzprojektierung und -Programmierung ISBN 3-8007-2492-8.* VDE Verlag, 2001.

[47] O. Golubeva, M. Loghi, M. Poncino, and E. Macii. Architectural leakage-aware management of partitioned scratchpad memories. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1665–1670, New York, NY, USA, 2007. ACM Press.

[48] G.Wagenknecht, D.Dietterle, J.-P.Ebert, and R.Kraemer. Transforming protocol specifications for wireless sensor networks into efficient embedded system implementations. In *Proc. of European Workshop on Wireless Sensor Networks*, 2006.

[49] C.-C. Han, R. S. Ram Kumar, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *In Proc. of the 3rd international conference on Mobile systems, applications, and services MobiSys*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[50] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Springer Lecture Notes in Computer Science*, volume 2211/2001:pp. 166–184, 2001.

[51] T. A. Henzinger and C. M. Kirsch. The embedded machine: predictable, portable real-time code. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 315–326, New York, NY, USA, 2002. ACM Press.

[52] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming PPOPP '90*, pages 197–206, New York, NY, USA, 1990. ACM.

[53] R. Herzog. Static real-time analysis for an event-driven operating system. Master's thesis, BTU Cottbus, 2006.

[54] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, November 12–15 2000.

[55] D. Holmes, J. Noble, and J. Potter. Technical report effective synchronisation of concurrent objects: Laying the inheritance anomaly to rest. Technical report, Department of Computing, Macquarie University, Sydney, 1998.

[56] D. J. Howe and S. Michell. An approach to formal verication of real time concurrent ada programs. *In ACM Journal Ada Letters*, volume XXIII(number 4):pp. 87–92, 2003.

[57] ITU-T. Specification and description language z.100. Technical report, Telecommunication Standardization Sector of ITU, 2002.

[58] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of Design and Automation Conference DAC04*, 2004.

[59] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *In Journal ACM Computing Surveys*, volume 36(number 1):pp. 1–34, 2004.

[60] R. Karnapke and J. Nolte. Copra - a communication processing architecture for wireless sensor networks. In *Euro-Par 2006 Parallel Processing*, pages 951–960. Springer, 2006.

[61] O. Kasten and K. Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 7, Piscataway, NJ, USA, 2005. IEEE Press.

[62] H. Kim and H. Cha. Multithreading optimization techniques for sensor network operating systems. In *Proc. of European Workshop on Wireless Sensor Networks (EWSN)*, pages 293–308, 2007.

[63] H. Kopetz. Software engineering for real-time: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 201–211, New York, NY, USA, 2000. ACM Press.

[64] J. J. Labrosse. *MicroC/OS-II: the real-time kernel ISBN 1578201039*. CMP Media, Inc., USA, 2002.

[65] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[66] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proc. of 2nd International Symposium on Operating Systems*, 1978.

[67] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, Berkeley, CA, USA, 2004. USENIX Association.

[68] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *In Journal of the ACM*, volume 20(number 1):pp. 46–61, 1973.

[69] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, ISBN 0130996513, Upper Saddle River, NJ, USA, 2000.

[70] K. Lundqvist and L. Asplund. A formal model of a run-time kernel for ravenscar. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 504, Washington, DC, USA, 1999. IEEE Computer Society.

[71] LYNX. *Projektseite http://www.lynuxworks.com/rtos/ [2008-09-09]*.

[72] A. J. Massa. *Embedded Software Development with eCos*. Prentice Hall, ISBN 0-13-035473-2, 2002.

[73] Matlab. *Produktwebseite http://www.mathworks.de [2008-09-09].*

[74] J. W. McCormick. We've been working on the railroad: a laboratory for real-time embedded systems. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 2005.

[75] C. Moser, L. Thiele, D. Brunelli, and L. Benini. Adaptive power management in energy harvesting systems. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 773–778, New York, NY, USA, 2007. ACM Press.

[76] MoteIV. *TMote Sky Datasheet , Moteiv Corperation, Ressource http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf [2008-09-09].*

[77] G. Naeser and K. Lundqvist. Component-based approach to run-time kernel specification and verification. *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS '05)*, 00:68–76, 2005.

[78] D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, C. C. Weems, W. Burleson, and J. Ko. The spring scheduling co-processor: Design, use, and performance. In *IEEE Real-Time Systems Symposium*, pages 106–111, 1993.

[79] OSEK/VDX. *Operating System Specification, Version 2.2.3, OSEK Group, February 2005.*

[80] J. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, ressource http://www.softpanorama.org/People/Ousterhout/Threads/, January 1996.

[81] QNX. *Projektseite http://www.qnx.com/products/neutrino_rtos/ [2008-09-09].*

[82] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves. Energy-efficient, collision-free medium access control for wireless sensor networks. *In Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys '03)*, 2003.

[83] G. D. Reis and B. Stroustrup. Specifying c++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM Press.

[84] G. Research. *LEON2 Processor Users Manual V1.0.23*, 2004.

[85] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. *In Proceedings of Euromicro Conference on Real-Time Systems (ECRTS '00)*, volume 00, 2000.

[86] K. Schild and J. Würtz. Off-line scheduling of a real-time system. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 29–38, New York, NY, USA, 1998. ACM Press.

[87] K. Schild and J. Würtz. Scheduling of time-triggered real-time systems. *In Springer Journal Constraints*, volume 5(number 4):pp. 335–357, 2000.

[88] D. E. Schmitz, P. K. Khosla, R. Hoffman, and T. Kanade. Chimera: A real-time programming environment for manipulator control. In *Proc. of 1989 IEEE International Conference on Robotics and Automation, Phoenix, Arizona*, pages pp. 846–852, 1989.

[89] F. Schon, W. Schroder-Preikschat, O. Spinczyk, and U. Spinczyk. Design rationale of the pure object-oriented embedded operating system. In *Proceedings of the IFIP Workshop on Distributed and Parallel Embedded Systems*, 1998.

[90] C. Schulze. Statische zeitanalyse von funktionen in echtzeitsystemen. Master's thesis, BTU Cottbus, 2008.

[91] M. Schulze. Maßgeschneidertes planungswesen in betriebssystemfamilien. Master's thesis, Universität Magdeburg, 2002.

[92] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *In Kluwer Journal on Real-Time Systems*, volume 28(number 2-3):pp. 101–155, 2004.

[93] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *In IEEE Journal on Transactional Computing*, volume 39(number 9):pp. 1175–1185, 1990.

[94] D. C. Sharp. Object-oriented real-time computing for reusable avionics software. In *ISORC '01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 185, Washington, DC, USA, 2001. IEEE Computer Society.

[95] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *In IEEE Journal on Software*, volume 8(number 3):pp. 62–72, 1991.

[96] J. A. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, and G. Wallace. The spring system: Integrated support for complex real-timesystems. *Real-Time Syst.*, 16(2-3):223–251, 1999.

[97] D. B. Stewart and G. Arora. A tool for analyzing and fine tuning the real-time properties of an embedded system. *IEEE Transactions on Software Engineering*, pages pp.311–326, April 2003.

[98] D. B. Stewart, D. E. Schmitz, and P. Khosla. The chimera ii real-time operating system for advanced sensor-based control applications. *In IEEE Transactions on Systems, Man, and Cybernetics*, volume 22(number 6):pp. 1282–1295, 1992.

[99] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.*, 23(12):759–776, 1997.

[100] Telelogic. *Telelogic Tau SDL Suite(2004), Produktseite http://www.telelogic.com/products/tau/sdl [2008-09-09].*

[101] TinyOS. *Lesson 2: Modules and the TinyOS Execution Model, webseite http://docs.tinyos.net/index.php/Modules_and_the_TinyOS_Execution_Model [2008-09-09].*

[102] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. of 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, 2003. USENIX.

[103] G. Wagenknecht. Effizientes implementierungsmodell von sdl-spezifikationen in eingebetteten systemen. Master's thesis, BTU Cottbus, 2005.

[104] K. Walther and J. Nolte. A flexible scheduling framework for deeply embedded systems. In *In Proc. of 4th IEEE International Symposium on Embedded Computing*, 2007.

[105] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, 2004.

[106] M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, 00:281–292, 2007.

[107] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *In ACM Transactions on Programming Languages and Systems*, volume 19(number 2):pp. 292–333, 1997.

[108] K. Zuberi and K. Shin. Emeralds: a microkernel for embedded real-time systems. *In Proceedings of the Second IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, 1996.