technische universität
dortmund

fakultät für informatik

Masterarbeit

Energiegewahres Service-Level-Management in eingebetteten Betriebssystemen

Sebastian Engels

16. Februar 2016

Gutachter:

Prof. Dr.-Ing. Olaf Spinczyk

Dipl.-Inf. Markus Buschhoff

Lehrstuhl Informatik XII
Eingebettete Systeme
TU Dortmund

Inhaltsverzeichnis

1. Einleitung

Eingebettete Systeme, wie zum Beispiel ubiquitäre Systeme oder Wireless Sensor Networks sind kleinste Systeme, die in ihre Umgebung fest eingebunden sind und häufig Überwachungs-, Steuerungs- oder Regelfunktionen übernehmen. Die Einsatzmöglichkeiten solcher Systeme sind äußert vielseitig. So werden eingebettete Systeme zum Beispiel für die Internetanbindung von gewöhnlichen Geräten (Internet of Things), im Bereich der Fertigungstechnik und Logistik (Industrie 4.0) oder für die Vernetzung von Haustechnik und Haushaltsgeräten (Smart Home) genutzt.

Einsatzmöglichkeiten

Im Bereich des Ubiquitous Computing oder von drahtlosen Sensornetzwerken werden Systementwürfe oft von den ausgeprägten Ressourcenbeschränkungen solcher Systeme dominiert, beziehungsweise limitiert[21]. Insbesondere ist die verfügbare Energie bei diesen Anwendungen problematisch, da die Fortschritte der Batterietechnologien kleiner sind als bei anderen Systemkomponenten[7].

beschränkte Ressourcen

Eingebettete Systeme, die ausschließlich auf eine Verbindung mit einem Versorgungsnetz angewiesen sind, haben den Nachteil des begrenzten Einsatzbereiches, zum Beispiel im Freien. Ein alleiniger Batteriebetrieb würde bei den in großer Anzahl eingesetzten Systemen zu einem nicht vertretbaren Wartungsaufwand führen.

Diese Einschränkungen können jedoch durch das sogenannte Energy harvesting überwunden werden, bei welchem zusätzliche Energie aus der Umgebung des Systems gewonnen wird. Verfügbare Energiequellen sind zum Beispiel Sonne-, Wind- und Schwingungsenergie, die dem System über sogenannte Nanogeneratoren zugeführt wird. Die durch das Energy harvesting gewonnene Energie kann jedoch aufgrund verschiedener Faktoren, wie beispielsweise die Tageszeit oder Wetterumstände, stark schwanken. Es ist daher im Vorfeld nicht möglich ein bestimmtes Energieniveau zu garantieren, was mit unzumut-

Energy harvesting

baren Unsicherheiten für den Verwender einher gehen. Ein eingebettetes Betriebssystem sollte einen Mechanismus besitzen, um auf unterschiedliche Energieniveaus reagieren zu können und die Anwendungen des eingebetteten Systems dementsprechend steuern.
Diese Arbeit befasst sich mit den Möglichkeiten und Mechanismen, die dieses Problem eindämmen oder vollständig beseitigen können.

1.1. Struktureller Aufbau

Im Verlaufe dieser Ausarbeitung werde ich zunächst auf meine Motivation eingehen, die mich zur Verfassung dieser Masterarbeit bewogen hat. Darauf folgend werde ich darstellen, welche Problemstellung diese Arbeit behandelt, sowie die entsprechenden Ziele erläutern, die durch diese Arbeit erreicht werden sollen. Anschließend erfolgt in Kapitel 2 eine Einführung des Lesers in die notwendigen Grundlagen dieser Thematik. Hierzu erfolgt eine Vorstellung der genutzten Hardwareplattformen sowie des zugrunde liegenden Betriebssystems. Im Anschluss werden verwandte Forschungsarbeiten erläutert, die sich mit ähnlichen Thematiken beschäftigen. Dabei werden einerseits Ideenansätze für diese Arbeit vorgestellt, aber auch Unterschiede zum Ansatz dieser Arbeit hervorgehoben. In Kapitel 4 folgt der Entwurf von adaptiven Synchronisationsobjekten und der Modellierung von Service-Levels durch Zustandsautomaten. Auf der Grundlage dieses Entwurfs wird im anschließenden Abschnitt die Implementierung der vorgestellten Konzepte behandelt. Im Evaluationskapitel wird schließlich das Gesamtmodell bewertet. Dies beinhaltet vor allem eine detaillierte Analyse des entstehenden Energieverbrauchs der implementierten Betriebssystemkomponenten. Es werden weiterhin auch der zusätzliche Speicherverbrauch, die Generalisierbarbeit, sowie mögliche Einschränkungen und Grenzen dieses Ansatzes diskutiert. Diese Arbeit endet schließlich mit einer kurzen Zusammenfassung und einem Fazit, das auch einen Ausblick auf die zukünftige Entwicklung geben soll.

Grundlagen

Entwurf

Evaluation

Im Anhang dieser Arbeit erfolgt eine kurze Dokumentation der vorgenommenen Erweiterungen des Betriebssystems, welche eine einfache

Anhang

Weiterentwicklung ermöglichen soll. Zusätzlich sind weitere Tabellen aus Gründen der Übersichtlichkeit im Anhang abgebildet.

1.2. Motivation

Die in dieser Arbeit betrachteten Systeme beinhalten typischerweise eine Batterie als Energiespeicher und eine Möglichkeit des Energy harvestings, zum Beispiel durch eine Solarzelle. Eingebettete Systeme könnten somit ständig betrieben werden, ohne an eine Stromquelle angeschlossen zu sein, da sie Energie aus ihrer Umwelt gewinnen und überschüssige Energie speichern. Die Menge der durch das Energy harvesting zur Verfügung stehenden Energie ist oftmals schwer vorherzusagen und kann sich dramatisch aufgrund der Lage, Tageszeit, der Jahreszeit, den Wetterbedingungen sowie anderer Umweltfaktoren verändern.

Energie speichern

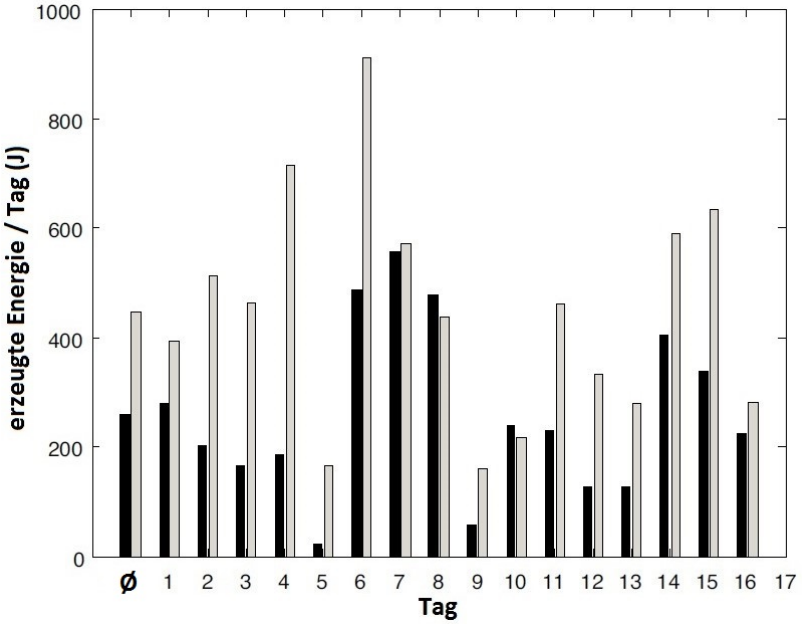

Abbildung 1.1.: Durchschnittlich erzeugte Energie zweier solarbetriebener Geräte über einen Zeitraum von 16 Tagen (aus [21]).

Die Abbildung 1.1 zeigt wie sich die unterschiedlich dargestellten Faktoren trotz ähnlicher Wetterbedingungen auf die Energieerzeugung auswirken können. In der Darstellung wird die täglich gewonnene Energie von zwei mobilen, solarbetriebenen Geräten während

Energieniveaus

eines Zeitraums von 16 Tagen beschrieben. Obwohl beide Geräte den gleichen allgemeinen Wettertrend zeigen, gibt es signifikante Unterschiede bei der Menge der gesammelten Energie. Diese Schwankungen können durch zusätzliche Effekte, beispielsweise dem spontanen Ausfall von Deckenbeleuchtungen, verstärkt werden.

Verwaltung Energie

Die Programmierung solcher ständig betriebener Systeme ist deshalb eine Herausforderung, da auf unterschiedliche Energieniveaus reagiert werden muss. Die Hauptmotivation für diese Arbeit basiert daher auf dem Wunsch nach einer Betriebssystemkomponente, die die Energieressourcen verwalten kann, um so nicht mehr auf externe Stromquellen angewiesen zu sein. Dadurch können langfristig Kosten eingespart, und der Einsatz von eingebetteten Systemen weiter ausgedehnt werden.

Ausführungsfrequenz

Hierbei soll nun nicht der klassische Ansatz eines minimalen Energieverbrauchs, erreicht durch das Deaktivieren von Systemkomponenten und Peripheriegeräten, verfolgt werden, sondern auch auf eine größere Energieverfügbarkeit reagiert werden. So soll als Reaktion auf sich ändernde Energieniveaus die Ausführungsfrequenz von energieintensiven Aufgaben angepasst werden. Dadurch soll ein reaktives System geschaffen werden, dass, sofern ausreichend Energie verfügbar ist, ein höheres Service-Level leistet, oder im Gegenzug, bei einem niedrigeren Energieniveau, durch ein geringeres Service-Level einen Systemausfall vermeidet. Weiterhin soll die Betriebssystemkomponente auch andere Faktoren, wie zum Beispiel die Tageszeit beachten und darauf reagieren können.

Overhead

Da eine zusätzliche Komponente auch immer Ressourcen verbraucht, wird die Frage nach dem verursachten Overhead[1] aufgeworfen. Hierbei ist nicht nur die verbrauchte Energie von Interesse, sondern beispielsweise auch der verbrauchte Speicher, da diese Ressourcen in einem eingebetteten System limitiert sind. Zusätzlich motiviert wird dieses Thema durch die Vision eines späteren proaktiven Systems, welches nicht nur aktuelle Faktoren in seine Entscheidung einfließen lässt, sondern aufgrund von in der Vergangenheit gesammelter Daten ein zukünftiges Systemverhalten abschätzt. Die hier entwickelten Komponenten können dann als wiederverwendbare Grundlage für ein

[1] Mehrverbrauch, der von der zusätzlichen Komponente verursacht wird.

solches System dienen, und somit vor allem den Energieverbrauch langfristig reduzieren.

1.3. Problemstellung

Nachdem zuvor die Motivation dieser Arbeit beleuchtet wurde, soll nun die Problemstellung des adaptiven Systems näher erläutert werden. Weiterhin wird die Art der genutzten Anwendungen des eingebetteten Systems für den weiteren Verlauf genauer definiert.

Im Bereich der eingebetteten Betriebssysteme ist das Deaktivieren von ungenutzten Peripheriegeräten ein Standardmechanismus für die Einsparung von Energie. Dieses Verfahren betrachtet vor allem die Nutzung eines Peripheriegerätes und nicht die Gesamtanwendung des eingebetteten Systems. Es ist allerdings auch wünschenswert, wenn die gesamte Anwendung den äußeren Bedingungen, beispielsweise der Energieerzeugung durch eine Solarzelle, angepasst wird.

Anpassung

Die im weiteren Verlauf betrachteten Anwendungen eines eingebetteten Systems bestehen typischerweise aus mehreren Tasks, beziehungsweise Threads T, welche zyklisch ausgeführt werden. Dies bedeutet, dass ein Thread t_i mit einer festgelegten Frequenz f_i immer wiederholend ausgeführt wird.

zyklische Threads

Eine beispielhafte Anwendung für ein solches eingebettetes System ist in Abbildung 1.2 skizziert. Die Anwendung besteht aus fünf Tasks, welche teilweise abhängig voneinander agieren. Zwei Tasks fragen mit den Frequenzen f_{M1} und f_{M2} Sensorwerte ab, die von einem weiteren Task mit der Frequenz f_F gefiltert werden. Ein weiterer Task sendet mit der Frequenz f_S die gefilterten Daten per Funk. Unabhängig von diesen Aufgaben ist ein Task für die Aktualisierung des Displays zuständig. Alle Tasks arbeiten mit unterschiedlichen Frequenzen, welche im Vorfeld festgelegt wurden. Die Anwendung wird somit durch die Threads $Display, Messung1, Messung2, Filter, Senden$ mit den zugehörigen Frequenzen $f_D, f_{M1}, f_{M2}, f_F, f_S$ definiert.

Beispielanwendung

Das Ziel eines energiegewahrenden, reaktiven Systems besteht darin, die Ausführungsfrequenzen der einzelnen Threads an äußere Bedingungen, zum Beispiel der vorhandenen Energie oder der Tageszeit, anzupassen. Ebenso sollen Threads und ihre genutzten Geräte voll-

Service-Level

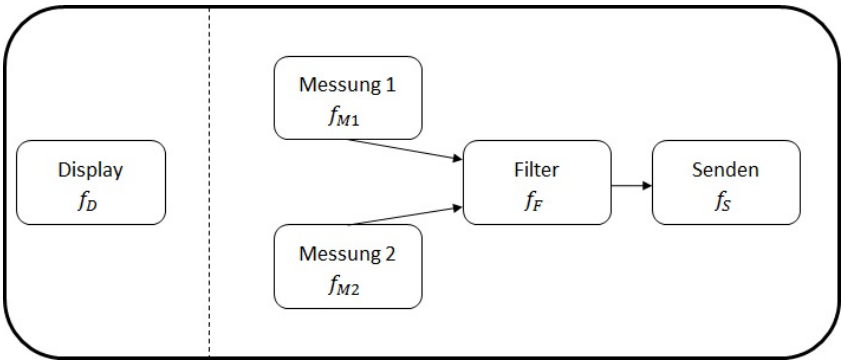

Abbildung 1.2.: Skizze einer Beispielsanwendung für ein eingebettetes System. Die einzelnen Threads arbeiten zyklisch mit den angegebenen Frequenzen.

ständig deaktiviert werden können. Dies führt zur Einführung von sogenannten Service-Leveln. Ein Service-Level SL_i lässt sich nun über die Menge der ausgeführten Threads T_{SL_i} sowie deren Frequenzen F_{SL_i} definieren.

Für die in Abbildung 1.2 dargestellte Anwendung lassen sich daher verschiedene Service-Level definieren. In Abbildung 1.3 sind exemplarisch am Beispiel eines Zustandsautomatens einige Service-Level für die Anwendung dargestellt worden. Abhängig von der vorhandenen Energie soll zwischen 07:00 und 17:00 Uhr ein höherer bzw. niedriger Service-Level gewählt werden. Außerhalb dieser Tageszeiten soll zu einem sehr niedrigen Service-Level gewechselt werden, um Energie einzusparen. In der Tabelle 1.1 sind die je nach Service-Level aktiven Threads und ihre Ausführungsfrequenzen aufgelistet.

Service-Level	aktive Threads	Frequenzen (Ausführungen pro Stunde)
SL_1	Display, Messung 1, Senden	$f_D = 0.5$, $f_{M1} = 0.25$, $f_S = 0.25$
SL_2	Display, Messung 1, Messung 2, Filter, Senden	$f_D = 1$, $f_{M1} = 0.5$, $f_{M2} = 0.5$, $f_F = 0.25$, $f_S = 0.5$
SL_3	Messung 1, Senden	$f_{M1} = 0.25$, $f_S = 0.25$

Tabelle 1.1.: Auflistung der je nach Service-Level aktiven Threads sowie ihre Ausführungsfrequenzen.

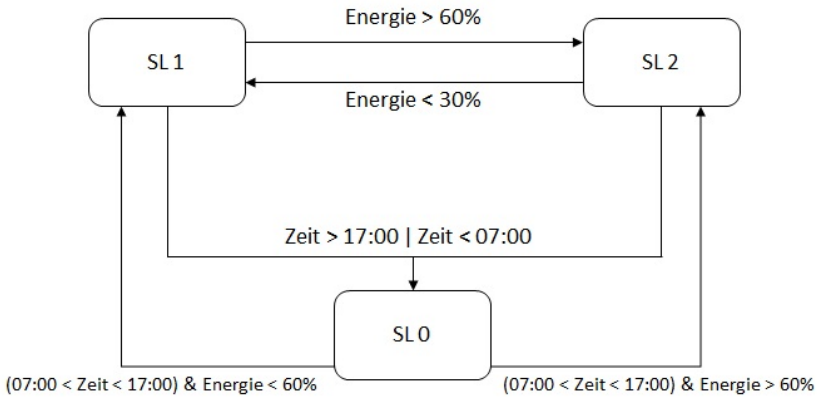

Abbildung 1.3.: Zustandsautomat für die Beispielanwendung des eingebetteten Systems. Abhängig von der verfügbaren Energie und Tageszeit soll zwischen verschiedenen Service-Levels gewechselt werden.

1.4. Ziele der Arbeit

neue Komponenten

Im Zuge dieser Arbeit sollen geeignete Betriebssystemkomponenten entworfen werden, die für ein reaktives System erforderlich sind und den ersten Schritt zu einem proaktiven System darstellen. Die Betriebssystemkomponenten sollen auf Faktoren wie unterschiedliche Energieniveaus oder Tageszeiten reagieren und so Ausführungsfrequenzen von Anwendungen steuern können. Die entsprechenden Frequenzen sowie die benötigten Bedingungen sollen vom Entwickler an einer zentralen Stelle im Quelltext vorgegeben werden können, damit die Anwendung übersichtlich gehalten werden kann.

Voraussetzungen

Dabei wird davon ausgegangen, dass bei dem zugrunde liegenden Betriebssystem klassische Strukturen wie Prozessverwaltung, Unterbrechungssynchronisation oder Interprozesskommunikation vorhanden sind. Hierzu gehören Komponenten wie ein Scheduler, zeitbasierte Synchronisationsobjekte oder Schutzkonzepte für kritische Quellcodeabschnitte. Im Rahmen dieser Arbeit sollen zusätzliche Komponenten im Bereich der Prozessverwaltung sowie Interprozesskommunikation entworfen werden, um die vorgestellte Problemstellung zu lösen. Hierzu gehören adaptive Synchronisationsobjekte sowie Verwaltungskomponenten mit denen Service-Level definiert und gesteuert werden können.

Für eine Evaluierung des Entwurfs, soll das Betriebssystem Kratos so modifiziert werden, dass Schnittstellen vorhanden sind, mit welchen es möglich wird, dass für Anwendungen Service-Level definiert werden können und diese aufgrund äußerer Umstände angepasst werden. Hierbei soll die bisherige Struktur nur minimal geändert werden, um das System bei Applikationen ohne die Nutzung von Service-Level nicht unnötig aufzublähen.

Overhead

Neben dem eigentlichen Entwurf dieser Komponenten beschäftigt sich diese Arbeit mit einer ausführlichen Overheadanalyse. Denn auch wenn auf höhere Energieniveaus reagiert wird, wäre es fatal, wenn die notwendigen Überprüfungen und Reaktionen die so entstandene, zusätzlich vorhandene Energie verbrauchen und somit kein Mehrwert erreicht werden kann.

2. Grundlagen

Das folgende Kapitel führt erforderliche Grundlagen ein, um die Herausforderungen, Problemstellungen und Lösungsansätze dieser Arbeit nachvollziehen zu können. Zunächst wird die zu Grunde liegende Hardware für die Entwicklung vorgestellt. Im Anschluss daran wird das vorhandene Betriebssystem erläutert, in welches die entwickelten Komponenten, die Kern dieser Arbeit waren, integriert wurden. Weiterhin wird die konkrete Problemstellung definiert und denkbare Lösungsansätze auf Basis des bisherigen Betriebssystems dargestellt. Der letzte Teil behandelt schließlich die Vorstellung von verwandten Arbeiten auf diesem Fachgebiet.

2.1. MSP430 FR5969, LaunchPad und Solar Doorplate

Zur Entwicklung und Evaluation der Betriebssystemkomponenten wurde in dieser Arbeit ein MSP430 LaunchPad von Texas Instruments[24] verwendet. Dieses nutzt als Mikrocontroller einen MSP430 FR5969[25]. Dieser Mikrocontroller zeichnet sich durch einen besonders geringen Energieverbrauch aus. Dadurch ist er vor allem für die Verwendung in batteriebetriebenen, eingebetteten Systemen geeignet. Im Folgenden wird zunächst der Mikroprozessor näher erörtert sowie auf die Plattform des LaunchPads eingegangen. Anschließend wird eine weitere Plattform, das Solar Doorplate[2], welches ebenfalls einen MSP430 nutzt, kurz vorgestellt.

2.1.1. MSP430 FR5969

Mikrocontroller

Beim MSP430 FR5969 (Abbildung 2.1) handelt es sich um einen 16-Bit RISC[1] Mikroprozessor, mit 2 KiB SRAM[2], 64 KiB FRAM[3] und einer Taktrate bis zu 24MHz. Er hat einen Betriebsspannungsbereich von 1.8 Volt bis 3.6 Volt. Besonders erwähnenswert ist die Verwendung von FRAM als nichtflüchtigen Speicher, anstatt der normalerweise genutzten EEPROM[4]- oder Flash-Speicher. Der Vorteil von FRAM gegenüber EEPROM- oder Flash-Speichern ist der geringere Energieverbrauch sowie deutlich höhere Zugriffsgeschwindigkeiten [23].

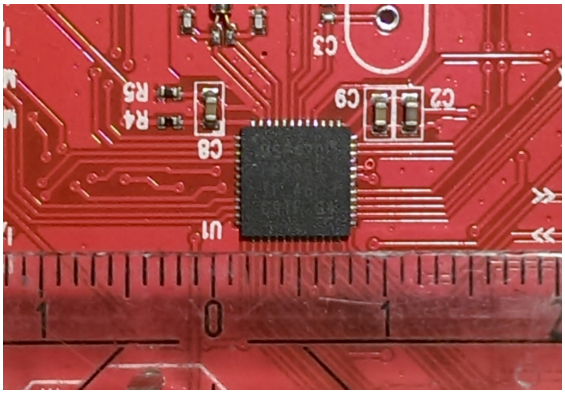

Abbildung 2.1.: MSP430 FR5969 Mikrocontroller im Vergleich zu einem Zentimetermaß. Der Mikrocontroller Texas Instruments eignet sich aufgrund seines sehr geringen Energieverbrauchs besonders für eingebettete Systeme.

Taktquellen

Der Mikrocontroller ist mit drei internen und zwei externen Taktquellen ausgestattet, sodass Peripheriekomponenten mit unterschiedlichen Taktraten betrieben werden können. Hierbei hervorzuheben ist insbesondere der interne DCO[5], ein ab Werk kalibrierter Oszillator, der den Systemtakt bereitstellt. Wahlweise können so Frequenzen von 1MHz, 2.7MHz, 3.5MHz, 4MHz, 5.3MHz, 7MHz, 8MHz, 16MHz, 21MHz und 24MHz realisiert werden. Die zwei weiteren in-

[1] Rechner mit reduziertem Befehlssatz, engl. Reduced Instruction Set Computer.
[2] Statischer RAM-Speicher, engl. Static Random Access Memory.
[3] Ferroelektrischer RAM-Speicher, engl. Ferroelectric Random Access Memory
[4] Nichtflüchtiger Speicher, engl. Electrically Erasable Programmable Read-Only Memory.
[5] Digital einstellbarer Oszillator, engl. Digitally Controlled Oscillator.

ternen Taktquellen sind der VLO[6] sowie der MODOSC[7], welche eine ungenauere Taktrate liefern. Die externen Oszillatoren LFXT[8] und HFXT[9] werden durch angeschlossene Quarze betrieben. Der LFXT ist ausschließlich für Uhrenquarze ausgelegt und dient der Realisierung einer RealTimeClock.

Interrupts

Für eine effiziente und schnelle Reaktion auf Ereignisse, ist der Mikrocontroller mit Interrupts ausgestattet. Unterbrechungen können zum Beispiel durch abgelaufene Timer verursacht werden. Dabei wird der aktuelle Programmfluss unterbrochen, der Programmzähler und die Status Register werden gesichert. Anschließend wird der entsprechende Eintrag der Interruptvektortabelle angesprungen, welcher die Unterbrechungsbehandlungsroutine einleitet. Die Interruptquellen des MSP430 können einzeln aktiviert sowie deaktiviert werden, sodass nur auf vom Benutzer gewünschte Ereignisse reagiert wird.

Timer

Um beispielsweise ein präemptives Scheduling zu nutzen, werden Timer benötigt, die periodisch in definierten Zeitabständen ein Interrupt auslösen und somit das Scheduling anstoßen. Bei Timern handelt es sich um Zähler mit einer einstellbaren Taktquelle, die ihren Zählerstand stets mit einer Vorgabe vergleichen und zum Beispiel beim Überschreiten dieser Vorgabe eine Unterbrechung auslösen. Der MSP430 besitzt zwei 16 Bit-breite Timer-Einheiten, die für Verzögerungen und das Scheduling genutzt werden können.

Energiesparmodi

Der MSP430 bietet unterschiedliche Energiesparmodi an, da ein energiesparendes Verhalten für batteriebetriebene Systeme besonders wichtig ist. Neben dem gezielten Deaktivieren von Komponenten kann das gesamte System in sogenannte low-power-modi versetzt werden. Je nach Modus werden so CPU, Oszillatoren oder FRAM deaktiviert, um Energie einzusparen. Eine Rückkehr zum normalen Betrieb wird wieder durch Interrupts, wie z.B. einen abgelaufenen Timer realisiert.

[6] Energiesparender Niederfrequenz-Oszillator, engl. Very-Low-Power Low-Frequency Oscillator.
[7] Modul-Oszillator, engl. Module Oscillator.
[8] Niederfrequenz-Oszillator, engl. Low Frequency Crystal.
[9] Hochfrequenz-Oszillator, engl. High Frequency Crystal.

2.1.2. MSP430 LaunchPad

Als geeignete Testumgebung wurde das LaunchPad (siehe Abbildung 2.2) von Texas Instruments [24] verwendet. Auf der Platine werden verschiedene Hardwarekomponenten zusammengefasst, wie der MSP430 FR5969 Mikrocontroller, zwei LEDs, ein Kondensator zur Spannungsversorgung, zwei Taster und ein Uhrenquarz für eine RealTimeClock. Weiterhin ist es möglich über Stiftleisten weitere Peripheriekomponenten, wie zum Beispiel ein Display, anzuschließen. Zusätzlich ermöglicht die Platine ein bequemes Debuggen und Aufspielen von Software per USB.

Peripherie ansteckbar

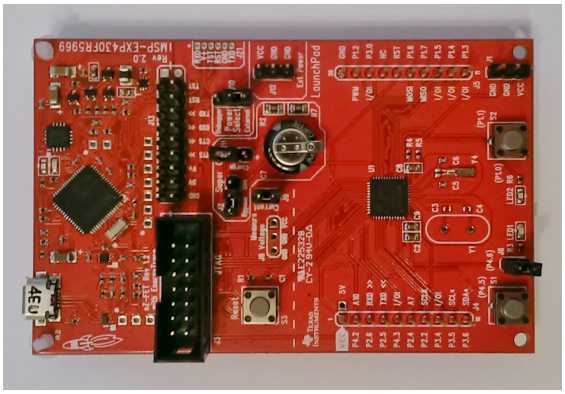

Abbildung 2.2.: Hardwareplattform LaunchPad von Texas Instruments mit dem MSP430 FR5969 Mikrocontroller.

2.1.3. inBin und Solar Doorplate

Das inBin (siehe Abbildung 2.3) ist eine Entwicklung des Fraunhofer Instituts für Materialfluss und Logistik (IML) [5]. Dabei handelt sich um eine Hardwareplattform, die an Transportkisten in Lagersystemen installiert wird und somit „intelligente Behälter“ ermöglichen soll. Die Plattform verfügt unter anderem über einen MSP430 FR5969 Mikrocontroller, ein e-paper-Display, ein Funkmodul sowie einem Kondensator, der über eine Indoor-Solarzelle geladen wird.

zusätzliche Hardwarekomponenten

Im Zuge der Projektgruppe 595 der TU Dortmund soll mittels des inBins ein Netzwerk von intelligenten Türschildern, sogenannten Solar Doorplates (siehe Abbildung 2.4), konzipiert werden. Die Türschilder sollen beispielsweise über das Display Auskunft über die Raumbelegung, aktuelle Veranstaltungen oder Belegungsdauer geben.

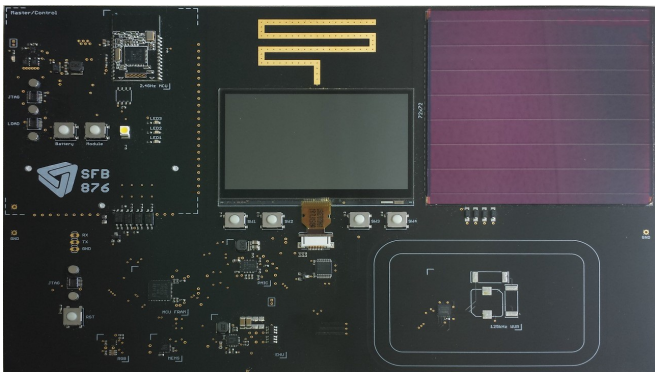

Abbildung 2.3.: Hardwareplattform inBin (aus [2]), entwickelt am Fraunhofer IML. Grundlage für die Solar Doorplates der PG595.

Abbildung 2.4.: Vision des Solar Doorplates (aus [27]), welches Auskunft über die Raumbelegung oder aktuelle Veranstaltungen geben soll.

2.2. Kratos

Weiterentwicklung OOStuBS

Im folgen Abschnitt erfolgt eine Vorstellung des Betriebssystem Kratos. Das Betriebssystem Kratos ist eine Weiterentwicklung des Betriebssystems OOStuBS, welches für verschiedene Lehrveranstaltungen entworfen wurde. Als Programmiersprache kommt überwiegend C++ zum Einsatz. Vereinzelt wird auch C sowie Assembler genutzt. Zunächst wird auf die genutzte Struktur des Betriebssystems eingegangen. Anschließend erfolgt eine Vorstellung von denkbaren Lösungsansätzen für die vorgestellte Problemstellung in Kratos. Dabei werden Schwachstellen dieser Ansätze erläutert, welche als Grundlage für den Entwurf von zusätzlichen Betriebssystemkomponenten dienen.

2.2.1. Struktur des Betriebssystems

Der strukturelle Aufbau des Betriebssystems ist in Abbildung 2.5 dargestellt. Das Betriebssystem Kratos besteht aus verschiedenen Schichten, welche wiederum aus unterschiedlichen Klassen bestehen.

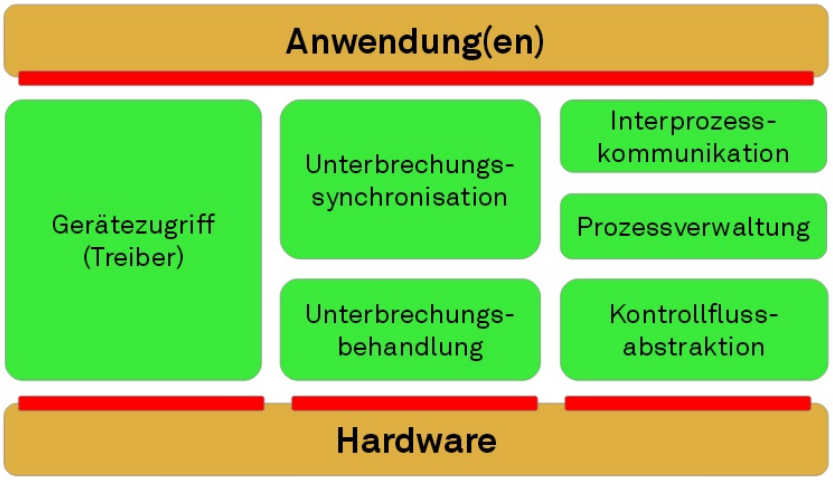

Abbildung 2.5.: Die Struktur des Kratos-Betriebssystems (aus [22]). Den Schichten können unterschiedliche Klassen zugeordnet werden, welche die Grundfunktionalitäten des Systems sicherstellen.

Interrupts

Im Bereich der Unterbrechungsbehandlung ist in Kratos eine Infrastruktur vorhanden, um auf mögliche Interrupts der verwendeten Geräte reagieren zu können. Dies geschieht, indem eine Unterbrechungsbehandlung an das entsprechende Gerätetreiberobjekt weitergeleitet wird. In diesem wird dann die entsprechende Behandlungsroutine ausgeführt. Ist für ein Gerät kein konkretes Treiberobjekt vorhanden, wird die Standardbehandlungsroutine ausgeführt.

Die Schicht oberhalb der Unterbrechungsbehandlung dient der Konsistenzsicherung zwischen den Anwendungskontrollflüssen und den Unterbrechungsbehandlungen. Die Beziehung zwischen Anwendungskontrollflüssen und Unterbrechungsbehandlungen ist asymmetrisch. Dies bedeutet, dass eine Unterbrechung an beliebiger Stelle des Anwendungskontrollflusses auftreten kann und diesen bis zur vollständigen Abarbeitung der Unterbrechung verdrängt (run-to-completion).

Prolog, Epilog

Ein Anwendungskontrollfluss kann Unterbrechungen ledig kurzzeitig sperren und ihre Behandlung somit verzögern. Die Synchronisation muss daher auf Seiten der Unterbrechung sichergestellt werden. Um

Unterbrechungen nicht unnötig lange zu sperren und Gefahr zu laufen, Unterbrechungsanforderungen zu verlieren, werden die Unterbrechungsbehandlungen in Kratos zweigeteilt. Zuerst wird bei gesperrten Unterbrechungen der Prolog abgearbeitet, welcher bei Bedarf noch einen Epilog zur weiteren Bearbeitung anfordern kann.

Eine weitere direkt auf der Hardware arbeitende Schicht dient der Abstraktion von Kontrollflüssen. Zweck dieser Schicht ist ein Wechseln zwischen unterschiedlichen Kontrollflussfäden, um eine Quasi-Parallelität zu ermöglichen. Hierzu müssen die aktuellen Registerwerte sowie der Stack Pointer der aktuellen Routine gesichert und durch entsprechende Werte der neuen Routine ersetzt werden. Aufbauend auf der Kontrollflussabstraktion verwaltet ein Round-Robin-Scheduler eine Liste von laufbereiten beziehungsweise aktiven Prozessen.

präemptiver Scheduler

Die Treiberschicht dient der direkten Verbindung der Hardware- mit der Anwendungsebene, welche die höchste Ebene in Kratos darstellt. Auf der Anwendungsebene werden die Aufgaben des eingebetteten Systems realisiert. Die Anwendungen werden dabei als eigenständige Klassen implementiert.

Treiber

Ein Zusammenspiel von Komponenten der verschiedenen Ebenen kann am Beispiel der Prozessverwaltung erläutert werden. Threads der Anwendungsebene haben die Möglichkeit einige Zeit zu pausieren, sich quasi „schlafen zu legen“. Hierzu bieten sogenannte Buzzer auf der Interprozesskommunikationsschicht sleep()-Methoden an. Dabei wird der Thread für die entsprechende Zeitspanne von der Liste der aktiven Threads des Schedulers entfernt. Die pausierenden Threads werden von einem sogenannten Bellringer verwaltet, der für die Reaktivierung, also dem „Wecken“ der Threads zuständig ist. Hierzu wird durch einen Timer alle 10 Millisekunden ein Interrupt ausgelöst. Durch eine Unterbrechungsbehandlung des Timertreibers wird der Bellringer auf Epilogebene aufgefordert, die Wartezeit der Threads zu verringern und zu überprüfen, ob Threads geweckt werden müssen. Weiterhin werden die Timerinterrupts vom Scheduler als Auslöser für einen Prozesswechsel genutzt.

Wecker

2.2.2. Lösungsansätze in Kratos

Die in Kapitel 1.3 beschriebene Problemstellung lässt sich in der bisherigen Kratos Implementierung nur bedingt lösen. Der verwendete Scheduler verwaltet die aktiven Threads und veranlasst alle 10 Millisekunden einen Threadwechsel. Threads ist es möglich, sich mit einem Buzzer eine bestimmte Zeitspanne schlafen zu legen und sich so von der Liste aktiver Threads kurzzeitig zu entfernen. Diese Zeitspanne ist jedoch fest im Quellcode verankert (siehe Quellcode 2.1) und für jeden entsprechenden Aufruf konstant. Änderungen während der Laufzeit des Systems sind somit nicht möglich.

feste Zeiten

```
1 void Thread::action()
2 {
3  Guarded_Buzzer buz;
4  while(1){
5    display.update();
6    buz.sleep(1000);
7  }
8 }
```

Quelltext 2.1: Fest verankerte Zeitspannen im Quelltext.

Die Pausenzeiten und damit verbunden die Ausführungsfrequenzen von Threads müssen daher vom Entwickler sorgfältig abgewogen werden. Eine zu hohe Frequenz könnte zu viel Energie verbrauchen und zu einem Systemausfall führen. Eine zu geringe Frequenz würde vorhandene Energie nicht nutzen. Eine Möglichkeit für den Entwickler wären Frequenzabschätzungen mittels Simulationen, welche im Vorfeld durchgeführt werden müssen. Allerdings ist der Erfolg solcher Simulationen stark abhängig von der Genauigkeit des verwendeten Modells. Da es kaum möglich ist, alle relevanten Faktoren in entsprechender Genauigkeit zu betrachten, besteht weiterhin die Gefahr von Systemausfällen bzw. Energieverschwendung. Zusätzlich könnte auch bei gut abgeschätzten Pausenzeiten der Threads nicht auf spontane Ereignisse, Umwelteinflüsse oder Modellabweichungen reagiert werden, sodass feste Buzzerzeiten keine zufriedenstellende Lösung bieten.

Simulation möglich

Eine denkbare Alternative stellen Fallunterscheidungen direkt innerhalb der Threads dar, um so die Pausenzeiten abhängig von System-

Fallunterscheidungen

variablen zu gestalten. Hierzu muss der Thread die entscheidenden System- und Sensorwerte abfragen und im Anschluss zum Beispiel mittels if-Abfragen eine Entscheidung bezüglich der Pausenzeiten treffen.

```
1  while(1){
2    display.update();
3    energy = powermanagement.getEnergy();
4    hour = time.getHour();
5    if(hour < 7 || hour > 17 )
6     buz.sleep(5000);
7    if(hour > 7 & hour < 17 & energy > 60)
8     buz.sleep(1000);
9    else
10    buz.sleep(2500);
11 }
```

Quelltext 2.2: Variable Pausenzeiten abhängig von Systemvariablen.

Ein Beispiel ist im Codeausschnitt 2.2 dargestellt. Es handelt sich um einen Thread zur Displayausgabe. Abhängig vom Energiewert und der Tageszeit soll zwischen einer höheren und niedrigeren Frequenz unterschieden werden. Allerdings existieren auch bei diesem Vorgehen einige Nachteile. Zum einen führt das Pollen[10] der System- und Sensorwerte, welches jeder Thread durchführen muss, zu einem hohen Overhead. Dies schlägt sich in einem erhöhten Energie- und Speicherverbrauch nieder. Weiterhin kann das Pollen auf der Anwendungsebene zu Problemen führen, wenn während des Pollens beziehungsweise vor dem Treffen einer Entscheidung, ein Kontextwechsel aufgrund des Schedulings oder Interrupts veranlasst wird. Es besteht die Möglichkeit, dass sich während eines Kontextwechsels Sensorwerte verändert haben, sodass eine Entscheidung aufgrund nicht mehr aktueller Werte getroffen wird. Im Quelltext 2.2 dürften daher zwischen den Codezeilen 3 und 10 keine Kontextwechsel vorliegen.

häufiges Pollen

Ein weiterer erheblicher Nachteil ist, dass die eingestellten Pausenzeiten von schlafenden Buzzern nachträglich nicht mehr verändert werden können. Daher ist es weder möglich, einen Thread vorzeitig

nachträglich nicht änderbar

[10] Eine Methode, den Status oder Wert eines Geräts bzw. einer Variablen mittels zyklischer Abfragen zu ermitteln.

zu wecken, wenn es zu einer Änderung des Systemzustands gekommen ist, noch diesen länger schlafen zu lassen. Daher führt auch dieser Ansatz zu keiner zufriedenstellenden Lösung des Problems.

3. Verwandte Arbeiten

Die Betrachtung der Energieressource im Bereich der eingebetteten Systeme ist in den letzten Jahren stets aktueller Gegenstand der Forschung. Sie ist aus unterschiedlichsten Zielsetzungen motiviert. Einerseits werden Ressourcenmodelle für den Energieverbrauch, die Energieerzeugung sowie Energiespeicherung formuliert. Ein weiteres Forschungsgebiete ist eine Verwaltung der vorhandenen Energiemodelle. Hierzu gehören beispielsweise Verfahren zur Energieeinsparung oder Adaptierungsstrategien.
Dieses Kapitel soll einen kurzen Überblick über die verschiedenen Arbeiten aus den unterschiedlichen Teilbereichen geben, die als Grundlage und Ideenquelle für die Arbeit dienten. Ebenso soll erläutert werden, wie sich diese Arbeit von den vorhandenen Ansätzen absetzt.

3.1. Ressourcenmodelle als Grundlage

In [19] wird ein umfangreicher Überblick über potentiell einsetzbare Energiequellen für eingebettete Systeme gegeben. Dabei wird vor allem auf die unterschiedlichen Modellierungsschwierigkeiten der einzelnen Energiequellen eingegangen. Ebenso werden für die einzelnen Modelle die charakterisierenden physikalischen Größen erläutert. Zusätzlich werden in anderen Arbeiten konkrete Energiequellen für das Energy harvesting genauer untersucht. So wird in [10] die Einsatzmöglichkeiten von Solarzellen zur Energiegewinnung anhand einer Beispielanwendung erläutert. Eine weitere Möglichkeit der Energiegewinnung wird in [18] vorgestellt. Hierbei wird mittels einer „piezoelektrischen Windmühle“ elektrische Energie aus der Windenergie gewonnen. Das Grundprinzip dabei ist das Auftreten von elektrischen Spannungen an Festkörpern, wenn diese elastisch verformt werden. In [11] wird zusätzlich eine Möglichkeit zur Energiegewinnung aus

Energiequellen

Vibrationen aus dem Umfeld des eingebetteten Systems beschrieben.

Energiespeicherung, Energieverbrauch

Ebenso gibt es im Bereich der Energiespeicherung eine Vielzahl von unterschiedlichen Modellen. Eine Übersicht wird dabei in [17] gegeben. Hierbei ist zu erwähnen, dass die vorgestellten Modelle auch nicht-lineare Effekte in Bezug auf die Energiespeicherung betrachten. Im Bereich der Modellierung und Abschätzung des Energieverbrauchs ist vor allem [3] zu nennen. In dieser Arbeit wurde ein energiegewahres Treibermodell für eingebettete Betriebssysteme entwickelt, welches den Energieverbrauch eines Peripheriegeräts je nach Nutzung abschätzen kann.

3.2. Theoretische Adaptierungsstrategien

Eine Vielzahl von Arbeiten in diesem Forschungsgebiet beschäftigt sich mit theoretischen Adaptierungsstrategien. Diese Adaptierungsstrategien verfolgen das Ziel, das Systemverhalten an äußere Umstände, vor allem an die verfügbare Energie, anzupassen.

Art der Anwendungen

Die meisten Arbeiten betrachten dabei eine ähnliche Art von Anwendungen. Eine Anwendung eines eingebetteten Systems besteht dabei in der Regel aus mehreren Tasks, die in bestimmten Frequenzen wiederholt ausgeführt werden [15] [13] [14] [16]. Darauf basierend werden in einigen Arbeiten Service-Level genutzt, die aus den entsprechenden Tasks und unterschiedlichen Ausführungsfrequenzen bestehen [13][14].

Systemmodelle

Weiterhin wurden in den meisten Arbeiten ein Systemmodell definiert, das aus unterschiedlichen Ressourcenmodellen besteht [15] [13] [14] [16]. In Abbildung 3.1 ist beispielhaft das in der Arbeit [14] verwendete Systemmodell dargestellt. Die Systemmodelle geben beispielsweise Vorhersagen für die zukünftig erzeugte sowie gespeicherte Energie an. Aufgrund diese Vorhersagen soll ein zukünftiges Systemverhalten bestimmt werden.

Die theoretischen Arbeiten zeichnen sich in der Regel durch eine offline Optimierung aus. Dies bedeutet, dass unterschiedliche Adaptierungsstrategien genutzt werden, um, aufgrund den Vorhersagen eines Systemmodells, eine Optimierung des Systemverhaltens zu erreichen. Dabei wird diese Strategie im Vorfeld und nicht zur Laufzeit des Sy-

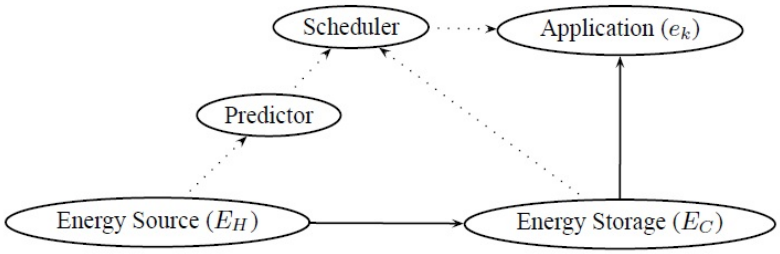

Abbildung 3.1.: Verwendetes Systemmodell in [14].

stems berechnet.

lineare Programmierung

Die Arbeit [16] behandelt eine Adaptionsstrategie, die die Parameter einer Anwendung, beispielsweise die Ausführungsfrequenzen, so verändern soll, dass diese an zeitlich unterschiedliche Energieniveaus angepasst werden. Hierbei wird davon ausgegangen, dass für eine Anwendung mit einer bestimmten Ausführungsfrequenzen ein Nutzen angegeben werden kann. Die unterschiedlichen Energieniveaus basieren dabei auf ein allgemeines Vorhersagemodell für Energie harvestende Systeme. Mittels der linearen Programmierung wurde ein Lösungsverfahren entwickelt, mit welchem der Gesamtnutzen, je nach Energieniveau, maximiert werden soll. Ein ähnlicher Ansatz, der ebenfalls die linearen Programmierung nutzt, wurde in [15] verfolgt.

dynamische Programmierung

Moser, Chen und Thiele beschreiben in ihrer Arbeit[14] eine weitere Möglichkeit, wie die Qualität des Services (quality of service) in einem eingebetteten Systems maximiert werden kann. Hierbei wird auch davon ausgegangen, dass Systeme mit einer Möglichkeit des Energy harvestings ausgestattet sind und Anwendungen aus periodischen Tasks bestehen. Das Ziel dieser Arbeit war es wieder, den geleisteten Nutzen des Gesamtsystems zu maximieren. Hierzu wurden Service-Level eingeführt, denen ein bestimmter Nutzen sowie ein Energieverbrauch zugeordnet wird. Anschließend ist es das Ziel für einen gewissen Zeitraum die gewonnene sowie gespeicherte Energie des Systems durch ein Systemmodell vorherzusagen. Dieses Systemmodell ist je nach Anwendung entsprechend zu wählen. Aufgrund der Vorhersagen soll eine Abfolge von Service-Leveln bestimmt werden, sodass der Gesamtnutzen in diesem Zeitraum maximiert wird. Zur

Lösung dieses Problems wurde ein rekursiver Algorithmus entworfen, der das Problemgebiet, bestehend aus der Menge der Service-Level und dem Zeitraum, schrittweise verkleinert. So können Lösungen für kleinere Teilproblemen bestimmt und anschließend eine optimale Lösung aus den optimalen Lösungen der Teilprobleme zusammensetzt werden. Hierzu wurde das Verfahren der dynamischen Programmierung verwendet.

Optimierungsprobleme

Bei den vorgestellten Arbeiten wurden klassische Optimierungsprobleme mit Nebenbedingungen, teilweise auch als Rucksackproblem bekannt, beschrieben und mittels unterschiedlicher Techniken gelöst. Generell sind auch anderen Lösungsansätze, wie beispielsweise Greedy-Algorithmen denkbar[12]. Der Rechenaufwand bei diesen Lösungsansätzen ist allerdings sehr hoch, sodass diese Vorgehensweisen nur im Vorfeld genutzt werden können, um eine grobe Vorausplanung für das Systemverhalten zu erstellen. Es ist jedoch weiterhin notwendig, dass zur Laufzeit des Systems Überprüfungen und Anpassungen vorgenommen werden.

3.3. Praktische Systemadaptierungen

Weiterhin existieren in diesem Forschungsbereich einige praktisch umgesetzte Adaptierungsverfahren. Dabei verfolgen viele entwickelte Adaptierungsstrategien, wie beispielsweise in [4] und [29], das primäre Ziel, durch Systemanpassungen Energie einzusparen und so eine konkrete Laufzeit des Systems zu gewährleisten. Dieses Ziel verfolgt auch TinyOS mit einem ereignisgesteuerten Ausführungsmodell. Das zugrunde liegende Prinzip ist, dass alle anstehenden Aufgaben schnellstmöglich bearbeitet werden sollen, um anschließend die Hardware in einen Energiesparmodus zu versetzen [9]. Im Bereich der eingebetteten, Energie harvestende Systeme ist allerdings ein weiteres Ziel, durch Systemanpassungen auch auf höhere Energieniveaus reagieren zu können.

Sprache und Laufzeitumgebung

Eine interessante Möglichkeit der Systemadaptierung bietet das System Eon[21]. Eon ist eine Kombination aus einer Programmierspra-

che und einer Laufzeitumgebung, welche eine Basis für entwickelte Programme bildet. Eon hat den Zeck, die Entwicklung von eingebetteten Systeme zu unterstützen, indem eine Anwendung aus mehreren Teilprogrammen unterschiedlicher Programmiersprachen, wie Java, C++ oder nesC, zusammengesetzt werden kann.

Bei Eon handelt es sich um eine sogenannte koordinative Sprache[6], die den Zweck erfüllt, den Datenfluss bzw. Datenaustausch zwischen verschiedene Teilkomponenten zu beschreiben. Eon verwaltet eine aus mehreren Teilprogrammen zusammengestellte Anwendung, indem es die Ausführung der einzelnen Teile steuert. Grundlage für Eon ist die Programmiersprache Flux[1], welche so erweitert wurde, dass auf unterschiedliche Energiezustände reagiert werden kann.

Dieser Ansatz bietet dem Entwickler eine hohe Abstraktionsebene, die das adaptive Verhalten, beispielsweise in Bezug auf Energiezustände, von der eigentlichen Programmlogik trennt. Eon unterstützt derzeit eine Reihe von verschiedenen Programmiersprachen (C und nesC) und Betriebssystemen (Linux und TinyOS), wodurch Eon leicht auf unterschiedlichen Hardware-Plattformen angewendet werden kann.

Mit Hilfe der Programmiersprache Eon können Adaptionsverhalten eines Systems bezüglich unterschiedlicher Energieniveaus beschrieben werden. Die Beschreibung erfolgt dabei zustandsbasiert. Dies bedeutet, dass für ein Programm verschiedene Ausführungsvarianten (Zustände) definiert werden können. Je nach Zustand werden unterschiedliche Programmkomponenten ausgeführt. Ebenso kann die Ausführungsfrequenz von Komponenten innerhalb eines Zustands variiert werden. In Abbildung 3.2 ist ein Beispiel für die Zustände eines Eon-Programms abgebildet. Im Basiszustand (Base State) ist ein Teilprozess des Programms abgeschaltet und ein weiterer Prozess wird mit einer festen Ausführungsfrequenz betrieben. In dem definierten HiPower-Zustand werden beide Prozesse ausgeführt, wobei der GPS-Prozess eine variable, stufenlose Ausführungsfrequenz hat. Es können somit beliebige Ausführungsraten zwischen 1 und 10 Stunden gewählt werden. Die Laufzeitumgebung von Eon dient als eine Art Verwalter zwischen den unterschiedlichen Teilprogrammen und dem Betriebssystem. Sie bestimmt den aktuellen Energieverbrauch sowie die momentane Energieerzeugung und steuert mit Hilfe der de-

zustandsbasierte Beschreibung

finierten Zustände die Ausführung der Teilprogramme. Dabei ist es das Ziel, dass der Energieverbrauch genauso hoch wie die aktuell erzeugte Energie ist. Größere Anpassungen des Energieverbrauchs können dabei mit über die Zustandswechsel realisiert werden, für feinere Anpassungen können die Ausführungsfrequenz verändert werden.

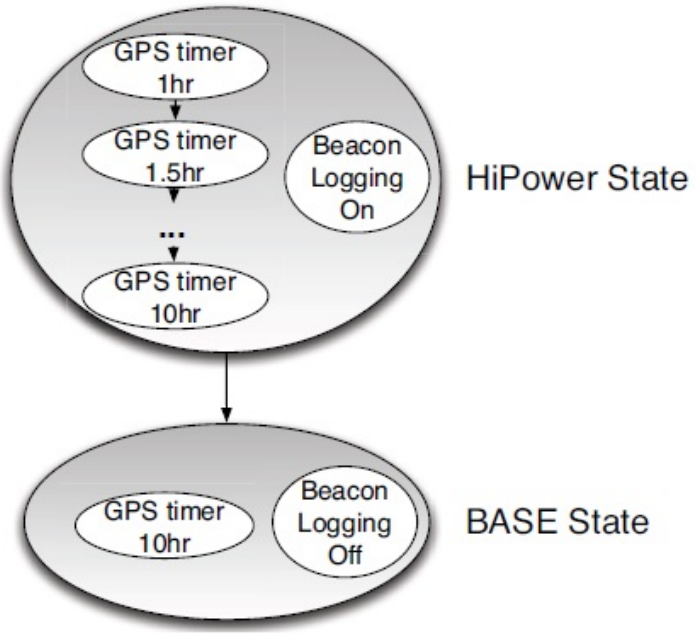

Abbildung 3.2.: Zustände eines Eon-Programms(aus [21]).

einfache Definition

Das Hauptziel von Eon war es, eine einfache Programmiersprache anzubieten, mit welcher Energieniveaus für verschiedene Programme definieren werden können. Im Zuge der Evaluierung konnte gezeigt werden, dass der Zeitaufwand für diese Definitionen deutlich geringer ist, als bei einer Umsetzung in C-Programmcode. Bei diesem Ansatz wurde allerdings davon ausgegangen, dass bereits ein Betriebssystemen (Linux oder TinyOS) vorhanden ist und verwendet werden kann.

Die verwandten Ansätze zeigen die Vielfältigkeit der Forschung im Bereich der eingebetteten Systeme. Die vorgestellten Arbeiten haben gezeigt, dass diese Arbeit bekannte grundlegende Ansätze verfolgt. Auch in anderen Arbeiten bestehen Anwendungen aus mehreren Tasks oder Threads, die gewissen Abständen wiederholt ausgeführt werden. Weiterhin ist es üblich, Service-Level zu definieren, die

bestimmte Tasks mit entsprechenden Ausführungsfrequenzen zusammenfassen.
Im Bereich der theoretischen Adaptierungsstrategien werden eher abstraktere Lösungsmöglichkeiten entwickelt, die aufgrund des hohen Rechenaufwands für eine offline Optimierung genutzt werden können. Das System Eon ist besonders interessant, da es Möglichkeit anbietet, während der Laufzeit Anpassungen vorzunehmen. Dabei nimmt Eon allerdings die Rolle eines Vermittlers zwischen Anwendung und Betriebssystem ein.
Im Zuge dieser Arbeit soll allerdings ein bestehendes Betriebssystem so erweitert werden, dass ein Vermittler nicht notwendig ist. Hierzu müssen Betriebssystemkomponenten entworfen werden, die es ermöglichen, für eine Anwendung verschiedene Zustände, sogenannte Service-Levels, zu definieren. Weiterhin soll der Entwickler in der Lage sein, die Übergänge zwischen den Service-Leveln bequem zu beschreiben.

4. Entwurf

Dieses Kapitel behandelt den Entwurf der erforderlichen Betriebssystemkomponenten, die nötig sind, um die in Kapitel 1.3 vorgestellte Problemstellung zu lösen.
Im ersten Teil dieses Kapitels werden die Ziele und Anforderungen an den Entwurf formuliert. Anschließend wird eine neue, notwendige Art von Synchronisationsobjekten eingeführt. Darauf aufbauend wird die Idee der automatenbasierten Darstellung von Service-Levels vorgestellt und genauere Funktionalitäten erläutert. Abschließend wird auf die Einbettung in ein Betriebssystem eingegangen.

4.1. Ziel und Anforderungen

Ein Ziel dieser Arbeit ist die Realisierung von Betriebssystemkomponenten um ein energiegewahres Service-Level-Management anbieten zu können. Ein Service-Level SL_i ist definiert über die Menge der aktiven, zyklisch ausgeführten Threads T_{SL_i} sowie deren Ausführungsfrequenzen F_{SL_i}. Das Betriebssystem soll die Threads T steuern, indem je nach Service-Level Threads aktiviert bzw. deaktiviert werden, und die Ausführungsfrequenzen der aktiven Threads bei Bedarf erhöht oder verringert werden können.

Service-Level

Hierzu benötigen die Betriebssystemkomponenten Kenntnis von allen wichtigen Faktoren, um zu entscheiden, welcher Service-Level aktuell geleistet werden kann. Im Gegensatz zum im Kapitel 2.2.2 vorgestellten Lösungsansatz mit Fallunterscheidungen innerhalb der Threads, gibt es eine zentrale Komponente, die alle relevanten Faktoren sammelt und speichert. Das Verhalten des Betriebssystems bezüglich der Entscheidung, welcher Service-Level aktuell geleistet werden kann, kann mit einem Automaten beschrieben werden. Der Entwickler gibt dabei Bedingungen vor, welche für ein Service-Level erfüllt sein müs-

Zustandsautomat

sen. Auch gibt er die entsprechenden Aktionen beim Übergang zu einem neuen Service-Level vor. Der Vorteil eines Automatenansatzes besteht in der einfachen (graphischen) Darstellbarkeit sowie einer hohen Vorstellbarkeit durch den Nutzer.

Anpassung der Frequenzen

Beim Übergang zu einem neuen Service-Level soll die Menge der aktiven Threads und deren Ausführungsfrequenzen angepasst werden. Weiterhin soll die Möglichkeit bestehen, zusätzliche Aktionen beim Übergang auszuführen, wie beispielsweise das Ein- oder Ausschalten von Peripheriekomponenten oder andere einmalige Anweisungen. Um Ausführungsfrequenzen entsprechend anpassen zu können, ist es notwendig, dass Threads früher geweckt werden können, wenn sich der Service-Level ändert. Hierzu wird eine neue Art von Synchronisationsobjekten benötigt, die zusätzliche Informationen wie bisher gewartete Zeit, insgesamt zu wartende Zeit sowie einen eindeutigen Identifier speichern können.

Schutz kritischer Abschnitte

Ein weiteres Ziel ist der Schutz kritischer Codeabschnitte , die unter anderem bei dem Aktualisieren von Sensorwerten und Systemvariablen, beim Treffen einer Service-Level Entscheidung oder beim Anpassen von Ausführungsfrequenzen auftreten können. Diese neuen Betriebssystemkomponenten müssen daher so in das bestehende System integriert werden, dass eine Konsistenzsicherung zwischen Anwendungskontrollflüssen, Unterbrechungsbehandlungen und den neuen Komponentenbehandlungen gewährleistet werden kann. Zusätzlich soll die Integration so geschehen, dass das Betriebssystem weiterhin konfigurierbar bleibt. Das bedeutet, dass das Betriebssystem auch ohne die zusätzlichen Komponenten betrieben werden kann. Zusammenfassend sind die konkreten Ziele und weitere Anforderungen an den Entwurf zur besseren Übersicht nochmals aufgelistet:

- Aktivierung bzw. Deaktivierung von Threads
- Anpassung von Ausführungsfrequenzen
- (graphische) Darstellbarkeit
- Rekonstruierbarer Kontrollfluss bei Interrupts
- Wiederverwendbarkeit

- Austauschbarkeit
- Konfigurierbarkeit
- Erweiterbarkeit
- Minimaler Overhead
- Einfache Implementierbarkeit
- Einfache Bedienbarkeit
- Minimalinvasiver Eingriff in Hinblick auf die Integration ins Betriebssystem

4.2. Adaptive Synchronisationsobjekte

Klassischerweise stehen in einem Betriebssystem Synchronisationsobjekte zur Verfügung, die es ermöglichen, dass Threads zeitweise nicht am Scheduling teilnehmen. Wie bereits in Kapitel 2.2.2 erläutert, stehen in der bisherigen Kratos-Implementierung Buzzer als Synchronisationsobjekte zur Verfügung. Mit diesen wird es Threads ermöglicht, sich für eine fest angegebene Zeitspanne schlafen zu legen.

keine Adaptierung möglich

Ein Problem dieser Implementierung ist, dass nachträgliche Adaptierungen der gewünschten Pausenzeiten nicht möglich sind, da ein Synchronisationsobjekt nur die zusätzliche Wartezeit bezüglich des vorherigen Objekts speichert. Bei der Verwendung von Service-Level muss es jedoch möglich sein, dass die Zeiten der Synchronisationsobjekte bei einem Wechsel des Service-Levels aktualisiert werden können. Ist die Wartezeit eines Threads durch die Aktualisierung der Zeiten abgelaufen, sollen dieser sofort wieder am Scheduling teilnehmen.

weitere Attribute

In einem eingebetteten Betriebssystem müssen daher Synchronisationsobjekte vorhanden sein, die so konstruiert sind, dass sie die insgesamt zu wartende Zeit, die bisher gewartete Zeit sowie einen Identifier verwalten. Letzterer dient bei der Aktualisierung der Wartezeiten der Identifikation der zugrundeliegenden Threads, welche sich hinter einem Synchronisationsobjekt verbergen.

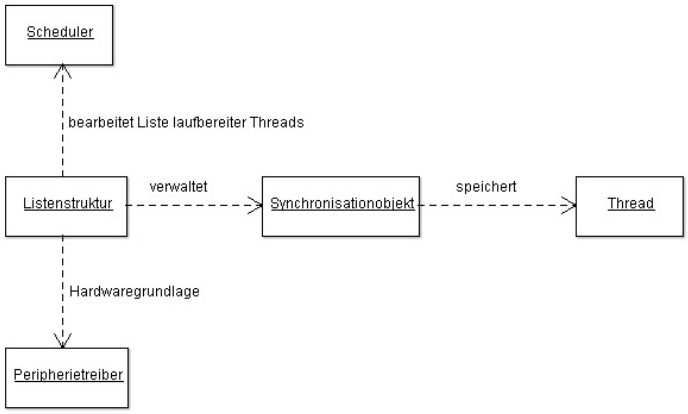

Abbildung 4.1.: Struktur der adaptiven Synchronisationsobjekte.

Struktur

Die Struktur des entstehenden Synchronisationskonzepts kann analog zu üblichen Strukturen entworfen werden. Eine konkrete Struktur ist in Abbildung 4.1 dargestellt. Als Grundlage wird ein Peripherietreiber benötigt, welcher in festen Zeitabständen Unterbrechungen auslösen kann. Dies kann beispielsweise durch die Verwendung eines Timer erfolgen. Die Synchronisationsobjekte müssen in Form von Methoden Möglichkeiten anbieten, mit denen Threads eine vorübergehende Pausierung auslösen können. Alle Synchronisationsobjekte werden dabei zum Beispiel in einer Listenstruktur von einer Betriebssystemkomponente verwaltet. Diese Komponente muss die Wartezeiten der Synchronisationsobjekte verändern können. Weiterhin muss diese Komponente überprüfen, ob durch die Aktualisierung die Wartezeit der Synchronisationsobjekte abgelaufen ist. In diesem Fall muss eine Mitteilung an den Scheduler erfolgen, dass die entsprechenden Threads wieder am Scheduling teilnehmen.

Auf die konkrete Umsetzung sowie Feinheiten bei der Implementierung der neuen Betriebssystemkomponenten wird in Kapitel 5 näher eingegangen.

4.3. Modellierung der Service-Level durch Zustandsautomaten

Wie bereits im vorherigen Kapitel erwähnt, ist es möglich, die Wechsel zwischen den Service-Levels mit einem Zustandsautomaten, einer Statemachine, zu beschreiben. Eine Statemachine lässt sich gut visuell darstellen, sodass auch komplexere Systeme verständlich erläutert werden können.

4.3.1. Theoretische Überlegung

In der Automatentheorie ist es üblich, einen Zustandsautomaten A über Tupel-Notation zu definieren[20]:

$$A =< \Sigma, S, s_0, \delta > . \tag{4.1}$$

Hierbei bezeichnet S die Menge der möglichen Zustände des Automaten, wobei s_0 den Startzustand des Systems definiert. Das Eingabealphabet Σ beschreibt eine Menge von potentiellen Ereignissen, die auf den Zustandsautomaten einwirken können. Über die Zustandsüberführungsfunktion δ werden die möglichen Übergänge zwischen den Zuständen beschrieben. Die Zustandsüberführungsfunktion δ bildet klassischer Weise die eine Kombination aus dem aktuellen Zustand $s \in S$ und dem Eingabealphabet Σ auf einen Folgezustand $s' \in S$ ab:

Überführungsfunktion

$$\delta : (s\ X\ \Sigma) \rightarrow s'. \tag{4.2}$$

Eine formale Definition der genutzten Statemachine ist analog möglich. Die Zustandsmenge S repräsentiert beziehungsweise beschreibt dabei die Menge der möglichen Service-Levels des Systems. Der Startzustand s_0 des Automatens gibt den initialen Service-Level des Systems beim Systemstart an. Die Grundlage für das Wechseln zwischen den Zuständen bildet das Eingabealphabet Σ. Bei der Statemachine sollen Bedingungen $B(V)$ bezüglich der entscheidungsrelevanten Systemvariablen V angegeben werden können, die zu einem Zustandswechsel führen sollen und somit Teil des Eingabealphabets Σ sind. Mögliche Variablen können die aktuelle Systemzeit, der Ladungszu-

Bedingungen, Systemvariablen

stand der Batterie, die aktuell erzeugte Energie einer Solarzelle oder der geschätzte momentane Energieverbrauch sein. Es ist anzumerken, dass eine Bedingung $B(V)$ nicht jede Variable $v \in V$ beachten muss. Weiterhin sollen spontan auftretende Ereignisse, im Weiteren als Events E bezeichnet, ebenfalls Zustandsübergänge in der Statemachine auslösen können. Events können unter anderem während der Behandlungsroutine eines Interrupts erzeugt werden. Beispielsweise wäre ein Event bei der Betätigung eines Tasters denkbar, um in einen Standby-Modus zu wechseln. Das Eingabealphabet Σ besteht somit aus der Menge der definierten Bedingungen $B(V)$ sowie möglichen Events E:

Events

$$\Sigma =< B(V), E > . \tag{4.3}$$

Die Zustandsüberführungsfunktion δ beschreibt auf der Grundlage des Eingabealphabets Σ die Wechsel zwischen den möglichen Zuständen. Ein Wechsel kann dabei durch die Erfüllung einer Bedingung $B(V)$, das Eintreten eines Events E oder einer Kombination aus beiden ausgelöst werden. In der Automatentheorie ist es üblich, dass der momentane Zustand $s \in S$ ein fester Bestandteil der Zustandsüberführungsfunktion δ ist. In diesem Kontext ist es allerdings wünschenswert, dass auch Zustandsübergänge unabhängig vom aktuellen Zustand möglich sind, wie in dem bereits erwähnten Beispiel eines Standby-Modus.

aktueller Zustand

Die Zustandsüberführungsfunktion δ bildet daher in diesem Kontext das Eingabealphabet Σ sowie Kombinationen aus dem aktuellen Zustand $s \in S$ und dem Eingabealphabet Σ auf einen Folgezustand $s' \in S$ ab:

$$\begin{aligned} \delta : (\Sigma) &\rightarrow s' \ , \\ (s \ X \ \Sigma) &\rightarrow s', \end{aligned} \tag{4.4}$$

Die genutzte Statemachine SM kann somit formal aus einem 4-Tupel $SM =< \Sigma, S, s_0, \delta >$ mit den vorangegangenen Festlegungen definiert werden.

4.3.2. Einbettung ins Betriebssystem

Die Statemachine soll die Ausführung von Threads steuern, indem die Ausführungsfrequenzen der Threads angepasst werden sowie Threads vollständig aktiviert beziehungsweise deaktiviert werden. Grundlage für die Statemachine bildet das zuvor vorgestellte Konzept der adaptiven Synchronisationsobjekte. Für eine Anpassung der Ausführungsfrequenzen muss die Statemachine so in das Betriebssystem integriert werden, dass ein Zugriff auf die Verwaltungskomponente der Synchronisationsobjekte möglich ist, um eine Anpassung der Zeiten auszulösen. Um zusätzlich ein Aktivieren und Deaktivieren von Threads zu ermöglichen, ist eine Verbindung zum Scheduler notwendig, sodass Threads zur Liste der aktiven Threads hinzugefügt oder gelöscht werden können.

Einbettung ins System

Die Statemachine hat unter anderem die Aufgabe, den aktuellen Systemzustand, also alle entscheidungsrelevanten Systemvariablen, abzuspeichern. Hierzu muss in der Statemachineklasse ein Variablenkonstrukt zur Verfügung stehen, in welchem der Entwickler alle entscheidenden Variablen zusammenfassen kann. Ebenso muss der Entwickler die Zustandsüberführungsfunktion definieren. Dies wird durch die Angabe einer Zustandsüberführungstabelle ermöglicht. Für ihre Angabe sollen Konstrukte vorhanden sein, mit denen eine bequeme und übersichtliche Angabe der Überführungstabelle möglich ist. Hierfür bieten sich beispielsweise Präprozessormakros an. In der Zustandsüberführungstabelle müssen Zustandsübergänge sowohl abhängig vom aktuellen Service-Level als auch Übergänge für alle Service-Levels beschrieben werden können. Hierzu muss die Statemachine den aktuellen Service-Level als Variable abspeichern. Weiterhin soll die Möglichkeit der Definition von Events bestehen, welche ebenfalls ein Teil der Zustandsübergangstabelle sein können.

Systemvariablen

Angabe als Tabelle

Wird beim Überprüfen der Statemachine festgestellt, dass der Service-Level gewechselt werden muss, soll automatisch eine Adaptierung der Zeiten der Synchronisationsobjekte veranlasst werden. Sind weitere Aktionen bei dem Übergang in ein neues Service-Level gewünscht, muss die Ausführung von entsprechenden Methoden für die einzelnen Service-Level sichergestellt werden.

Anpassung der Zeiten

Die vorgestellte Statemachine bietet dem Entwickler die Möglichkeit die Übergänge zwischen Service-Leveln bequem zu beschreiben. Um zur Laufzeit des Systems eine regelmäßige Überprüfung des Service-Levels zu gewährleisten, soll ein Thread verwendet werden. Dieser nutzt vorhandene Synchronisationsobjekte um zyklische Aktualisierungen der Systemvariablen und Überprüfungen der Statemachine zu veranlassen. Eine Aktualisierung von Variablen kann natürlich auch interruptbasiert realisiert werden. Da allerdings nicht alle Betriebssystemkomponenten Interrupts anfordern können, ist es notwendig, Variablenänderungen mittels zyklischem Abfragens zu ermitteln. Dieses Vorgehen wird als Polling bezeichnet. In Abbildung 4.2 ist die entstehende Gesamtstruktur der Statemachine skizzenhaft dargestellt.

Nutzung eines Threads

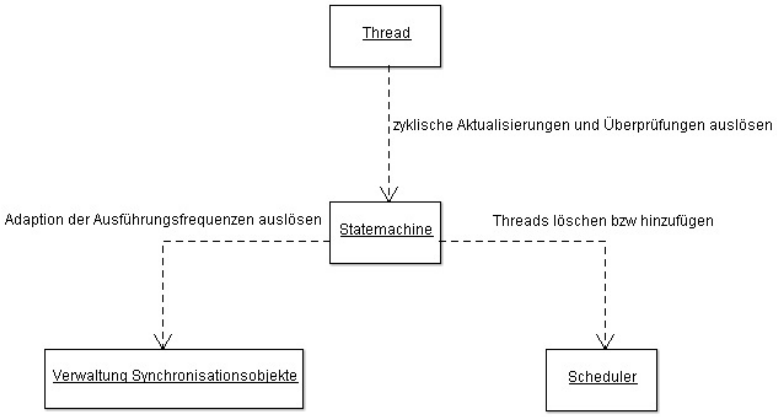

Abbildung 4.2.: Einbettung der Statemachine in ein Betriebssystem.

Ein weiterer zu beachtender Aspekt ist die Tatsache, dass während des Überprüfens der Statemachine oder dem Pollings von Systemvariablen kein Kontextwechsel durch den Scheduler geschehen darf. Um dies sicherzustellen, wechselt der Thread vor Aufruf der entsprechenden Funktionen auf die Epilogebene. Dies verhindert ein präemptives Scheduling während der kritischen Operationen. Alternativ wären interruptbasierte Aktualisierungen und Überprüfungen denkbar gewesen. So könnte ebenfalls ein Peripherietreiber genutzt werden, um die entsprechenden Methoden der Statemachine aufzurufen. Dies würde allerdings zusätzlichen Synchronisationsaufwand verursachen, da unterschieden werden muss, ob ein Interrupt eine Überprüfung der Statemachine signalisiert oder aus einem anderen Grund aufge-

Epilogebene

treten ist. Weiterhin wird durch die Verwendung eines Threads ein bekanntes Prinzip verwendet, welches allgemeingültig ist und leicht auf andere Systeme ohne entsprechendes Peripheriegerät adaptiert werden kann. Dies bedeutet, dass für die Verwendung der Statemachine nur sichergestellt sein muss, dass ein System mit adaptiven Synchronisationsobjekten zu Grunde liegt. Die genaue Realisierung dieser Objekte spielt dabei jedoch keine Rolle. Für weitere Einzelheiten der Implementierung der vorgestellten Komponenten wird auf das folgende Kapitel 5 verwiesen.

gängiges Prinzip

5. Implementierung

In diesem Kapitel wird die Implementierung der im Kapitel 4 vorgestellten Betriebssystemkomponenten erläutert. Hierzu wird zunächst auf die aspektorientierte Programmiersprache AspectC++ eingegangen, welche eine Erweiterung der Programmiersprache C++ ist und für eine Konfigurierbarkeit eines Betriebssystem genutzt werden kann. Anschließend wird die Implementierung der adaptiven Synchronisationsobjekte vorgestellt. Danach wird auf die Umsetzung der Statemachine eingegangen. Abschließend wird eine Beispielsanwendung erläutert, die das adaptive Verhalten der Statemachine zeigt.

5.1. Exkurs: AspectC++

Erweiterung von C++

Bei AspectC++ handelt es sich um eine aspektorientierte Erweiterung der Programmiersprache C++. Es ist vergleichbar mit AspectJ[1], aber aufgrund der Unterschiede zwischen Java und C++ in einigen Punkten völlig anders. Im folgenden Abschnitt soll ein kurzer Exkurs zu AspectC++ erfolgen, für weitere Informationen wird auf [28] verwiesen.

5.1.1. Motivation aspektorientierte Programmierung

Ziel von Aspekten

Die unterschiedlichen Aufgaben von Software können üblicherweise in zwei Bereiche unterteilt werden. Die funktionalen Anforderungen, die sogenannten Core-Level-Concerns, sind konkrete Anforderungen beziehungsweise Aufgaben, die an einer Stelle im Quelltext, zum Beispiel in einer Funktion, gekapselt werden können[8]. Hierfür wäre beispielsweise die Berechnung eines Wertes zu nennen. Bei den System-Level-Concerns handelt es sich um Anforderungen, die nicht gekapselt

[1] http://www.eclipse.org/aspectj/

werden können, da sie meist an mehreren Stellen im Quelltext implementiert werden müssen. Ein Beispiel hierfür wäre die Verwendung eines Loggers, dessen Aufrufe an unterschiedlichen Stellen erfolgen müssen. Die funktionalen Anforderungen ändern sich dabei jedoch nicht.

Klasse vs. Aspekt

Die Core-Level-Concerns werden typischerweise als Objekte beziehungsweise Klassen implementiert, während es sich bei den System-Level-Concerns um sogenannte Aspekte handelt. Da man in der Softwareentwicklung in der Regel Aspekte innerhalb von Klassen nutzen möchte, wird von verwobenen Anforderungen bzw. von Cross-Cutting Concerns gesprochen. Gelöst werden diese Anforderungen, indem der

Einweben

Aspektcode beim Compile-Vorgang erst in den eigentlichen Quelltext der anderen Objekte eingewoben wird. Im Anschluss daran erfolgt erst das Kompilieren des entstandenen C++ Codes.

Ein klassisches Anwendungsbeispiel im Bereich der Betriebssystemprogrammierung ist das Einweben von Aufrufen von plugin()-Methoden in leere Initialisierungsmethoden für Peripheriegeräte. So ist es möglich, dass eine Initialisierungsmethode, wie im folgenden Beispielquelltext 5.1, keine Anweisungen enthält, sondern ausschließlich über Aspekte mit Befehlen gefüllt wird.

```
1 /* Geraetetreiber-Initialisierungsfunktionen aufrufen
     */
2 static void init_devices()
3 {
4   // Wird ueber Aspekte mit Inhalt gefuellt
5 }
```

Quelltext 5.1: Leere Initialisierungsfunktionen im Originalquelltext.

Der einzuwebende Quelltext wird mittels sogenannter Advices definiert. Der folgende Beispielcode 5.2 führt zu einem Einweben der Anweisungen aus Zeile 10 in die Initialisierungsmethode aus 5.1.

```
1  #ifndef DMA_AH
2  #define DMA_AH
3
4  /* Activate DMA plugin */
5
6  #include "dma.h"
7
8  aspect ActivateDMA{
9   advice execution("void init_devices()") : after(){
10   dma.plugin();
11  }
12 }
13 ;
14
15 #endif
```

Quelltext 5.2: Aspekt zur Einwebung von Quellcode.

Über den Pointcut ”void initialize_devices()“ im Advicecode wird die Stelle definiert, an welcher eine Einwebung stattfinden soll. Das Schlüsselwort after() gibt dabei an, dass der Code direkt nach dem Methodenkopf eingewoben werden soll. Alternativ könnten auch die Schlüsselwörter before() oder around() verwendet werden.

Pointcut, Advice

Mit diesem Mechanismus wird Entwicklung und Programmierung von Betriebssystemen stark vereinfacht, da ein allgemeingültiger Quelltext erstellt werden kann und genauere hardwareabhängige Spezifizierungen hinterher mittels Aspekten erfolgen können. Ebenso können anwendungsspezifische Erweiterungen mit Aspekten realisiert werden.

5.1.2. Konfigurierbarkeit von Kratos

Alle Systemkomponenten von Kratos werden entweder als Klasse oder Aspekt implementiert, sodass eine Erweiterung des Betriebssystems ebenfalls durch zusätzliche Klassen oder Aspekte erfolgen kann.
Die Konfiguration der einzelnen Komponenten ist per graphischer Oberfläche möglich und in Abbildung 5.1 dargestellt. Die Komponenten, auch als Features bezeichnet, können auf der linken Seite der

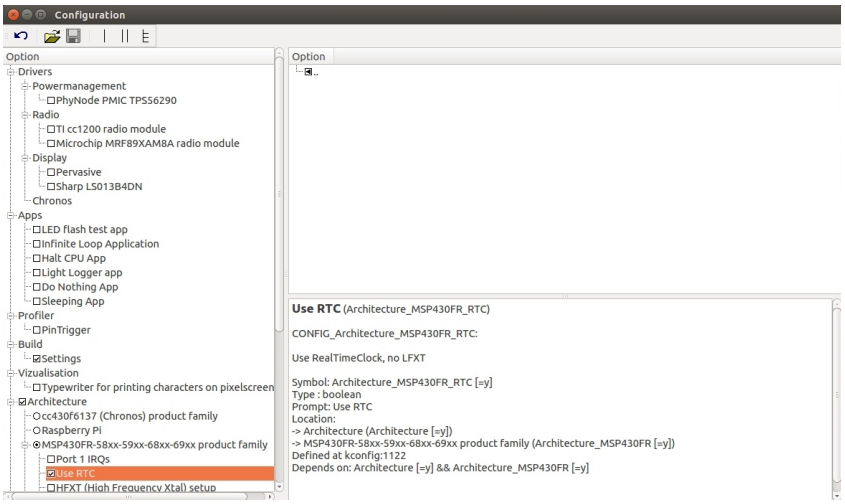

Abbildung 5.1.: Graphische Oberfläche zur Auswahl der Systemkomponenten.

Graphische Oberfläche

Oberfläche ausgewählt werden. Hierzu gehören unter anderem Zielarchitektur, Anwendungen, Peripheriekomponenten sowie entsprechende Treiberversionen. Auf der rechten Seite der graphischen Oberfläche befinden sich Parametrisierungsmöglichkeiten der ausgewählten Komponente, wie zum Beispiel Taktrate oder Variablenwerte bei Anwendungen. Durch das Abspeichern einer getroffenen Konfiguration wird eine Liste der entsprechenden Quelldateien erstellt. Ausgewählte Aspekte können so direkt in das System eingewoben werden. Gesetzte Parameterwerte werden in einer separaten Datei abgespeichert und als Präprozessormakros definiert.

.feature-Datei

Um eine Komponente konfigurierbar zu gestalten, muss eine zugehörige .feature-Datei im XML-Format erstellt werden. Der Quelltext 5.3 zeigt eine .feature-Datei für eine LED-Testanwendung. Über die Konfigurationsoberfläche kann bei diesem Beispiel die LED, über Angabe des Ports und Bits, ausgewählt werden. Beim Starten der graphischen Oberfläche wird das gesamte Quellcodeverzeichnis nach .feature-Dateien durchsucht und in die Sprache kconfig umgewandelt. Anschließend wir die Oberfläche mit dem Programm kconfig-qconf geladen.

Alle im Weiteren vorgestellten zusätzlichen Betriebssystemkomponenten werden mit .feature-Dateien ausgestattet, um die Komponen-

ten aus-, beziehungsweise abzuwählen. So soll eine maximale Konfigurierbarkeit des Systems erreicht werden.

```
<feature name="/Apps/LEDBLINK">
<dependencies>
 <dependsOn>/Architecture/MSP430FR</dependsOn>
</dependencies>
<in>
 <file>*</file>
</in>
<display>
 <prompt>LED flash test app</prompt>
  <help>Two threads concurrently toggling the LED by
      using XOR.</help>
</display>
<variables>
 <int name="LEDPORT" min="1" max="4" default="1">
  <display><prompt>LED Port Number</prompt></display>
 </int>
 <int name="LEDBIT" min="0" max="7" default="0">
  <display><prompt>LED Bit</prompt></display>
 </int>
</variables>
</feature>
```

Quelltext 5.3: .feature-Datei für eine LED-Testanwendung.

5.2. Implementierung der adaptiven Synchronisationsobjekte

Im folgenden Abschnitt wird die konkrete Implementierung der adaptiven Synchronisationsobjekte nach dem Entwurf in Kapitel 4.2 vorgestellt. Wie bereits in Kapitel 2.2.2 erläutert, stehen in der bisherigen Kratos-Implementierung sogenannte Buzzer als Synchronisationsobjekte zur Verfügung. Diese dienen dazu Threads einige Zeit schlafen zu legen und vorübergehend nicht mehr am aktuellen Scheduling teilzunehmen zu lassen. Hierfür kann die sleep(ticks)-Methode der Buzzerklasse genutzt werden. Dabei wird der aktuelle Thread in einem sogenannten Waitingroom für eine spätere Fortführung gespeichert. Als Zeitspanne wird eine Anzahl von Ticks angegeben, wobei

Buzzer

ein Tick ein Timerinterrupt darstellt, welcher alle 10 Millisekunden ausgelöst wird. Der Bellringer basiert auf einer Listenstruktur und verwaltet die erzeugten Buzzer. Die Buzzer werden im Bellringer nach der Wartedauer aufsteigend sortiert gespeichert, wobei ein Buzzer nur die zusätzliche Wartezeit bezüglich des vorangegangenen Buzzers abspeichert. Würde beispielsweise ein Thread mit dem Buzzer B_1 für eine Zeitspanne von 10 Ticks und ein weiterer Thread mit dem Buzzer B_2 für eine Dauer von 15 Ticks schlafen, so wäre der Buzzer B_1 mit einer Dauer von 10 Ticks das erste Listenelement des Bellringers. Der Buzzer B_2 würde mit einer Zeit von 5 Ticks als zweites Element gespeichert werden. Dieses Vorgehen ist besonders effizient, da bei einem Timerinterrupt nur die restliche Tickanzahl des ersten Listenelements verringert werden muss. Weiterhin wird bei jedem Timerinterrupt überprüft, ob die Tickanzahl eines Buzzers abgelaufen ist und dieser geweckt werden muss. Ist dies der Fall, wird der entsprechende Thread wieder in die Liste der laufbereiten Threads des Schedulers eingefügt. In Abbildung 5.2 ist die Struktur der bisher genutzten Komponenten skizziert.

effektive Speicherung

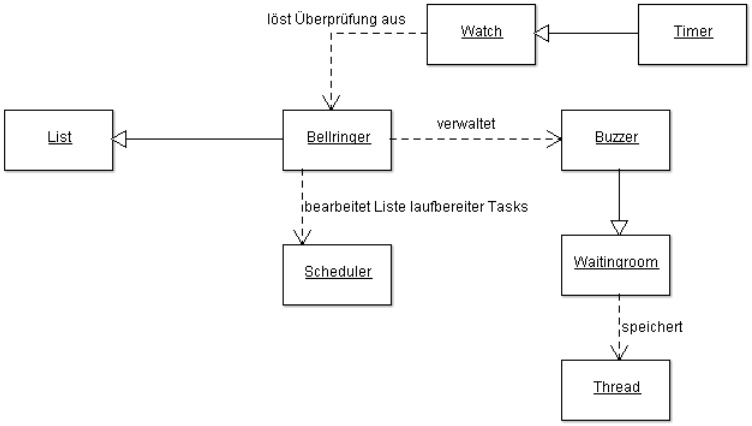

Abbildung 5.2.: Struktur der bisherigen Buzzerimplementierung.

Tickanzahl

Das Herunterzählen von Ticks ist besonders für größere Zeiträume kein effektives Vorgehen, da viele Interrupts benötigt werden. So werden beispielsweise für einen Zeitraum von 30 Minuten 180.000 Unterbrechungen durch den Timer benötigt. Da überwiegend solche längeren Zeiträume für die zyklischen Anwendungen benutzt werden, soll

die neue Art von Buzzer nicht mehr Timer-basiert umgesetzt werden. Eine Alternative bietet die RealTimeClock des MSP430 Launchpads, da bei dieser eine Alarmzeit minutengenau angegeben werden kann. Wird beispielsweise die Alarmzeit auf 15:15:00 Uhr gestellt, so wird nach dem Umspringen der Uhrzeit von 15:14:59 auf 15:15:00 Uhr ein Interrupt ausgelöst, der zur Signalisierung eines abgelaufenen Buzzers genutzt werden kann. Eine Ausweitung der Alarmzeiten auf den Sekundenbereich ist mit einer Anpassung der Startzeit der RealTimeClock möglich. Neue Buzzer können daher einen Wertebereich von 1 Sekunde bis zu 24 Stunden abdecken. Für Zeiten unterhalb einer Sekunde können weiterhin die vorhandenen Buzzer verwendet werden.

RealTimeClock

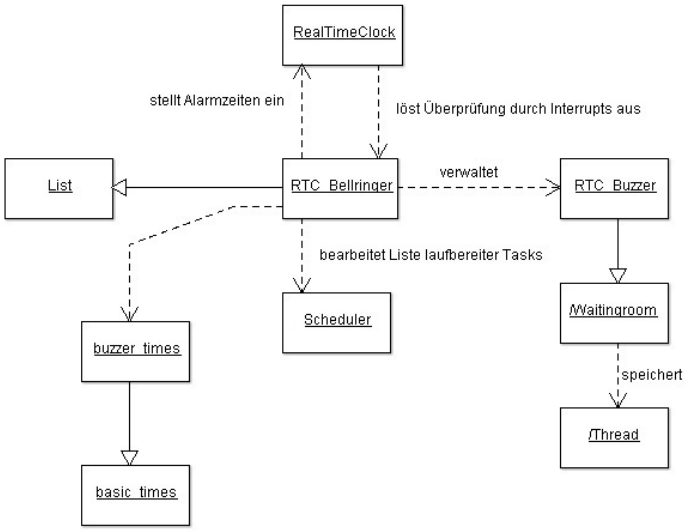

Abbildung 5.3.: Struktur der neuen Buzzerimplementierung.

Die Struktur der neuen Buzzerimplementierung wurde analog zur alten Struktur entworfen und ist in Abbildung 5.3 dargestellt. Es wurde ein Treiber für die RealTimeClock implementiert, die von der neuen Bellringerklasse RTC_Bellringer genutzt wird, um Alarmzeiten einzustellen. Durch die entstehenden Interrupts wird der RTC_Bellringer aufgefordert Buzzer auf den Ablauf ihrer Zeit zu überprüfen.

analoge Struktur

Der RTC_Bellringer verwaltet analog zum ursprünglichen Bellringer die neuen RTC_Buzzer in einer Listenstruktur. Hierbei bietet der RTC_Bellringer nun die Möglichkeit an, die Wartezeiten dieser Buz-

zer zu verändern. Aufgrund der zusätzlich eingeführten Variablen der Buzzer ist es nun möglich, Buzzer, deren Wartezeit durch die Aktualisierung abgelaufen ist, vorzeitig zu wecken. Um eine eindeutige Trennung zwischen den beiden Arten von Buzzern zu erlangen, wurde bewusst die zusätzliche Bellringerklasse eingeführt.
In den folgen Abschnitten wird die Programmierung des Treibers der RealTimeClock, der RTC_Buzzer sowie des RTC_Bellringers anhand von Quellcodeausschnitten genauer erläutert.

5.2.1. Treiber RealTimeClock

Die Grundlage für die adaptiven Buzzer ist die RealTimeClock des MSP430 LaunchPads. Sie ermöglicht die Einstellung diverser Optionen für die Erzeugung einer Unterbrechung. So können beispielsweise Unterbrechungen im Minuten- oder Stundentakt, zu Beginn eines neuen Tages oder zu einer angegebenen Alarmzeit erfolgen. Letztere Möglichkeit ist besonders für die Implementierung der adaptiven Buzzer interessant. Wie bereits im Vorfeld erwähnt, wird beim Umspringen der Uhrzeit auf die eingestellte Alarmzeit ein Interrupt ausgelöst, der eine Überprüfung von abgelaufenen Buzzern genutzt werden kann. Für die Angabe einer Alarmzeit stehen Register für die Minuten- und Stundenangabe zur Verfügung. Durch eine Anpassung der aktuellen Startzeit der RealTimeClock kann eine sekundengenaue Angabe der Alarmzeit erfolgen.

verschiedene Interrupts möglich

Da die RealTimeClock überall in Kratos ansprechbar sein soll, unter anderem für die Einstellung von Alarmzeiten und das Auslesen der Systemzeit, wird eine globale Instanz der Uhr angelegt. Im Konstruktor der RealTimeClock wird die Initialisierung des Geräts vorgenommen. Hierzu ist es notwendig, den Niederfrequenz-Oszillator LFXT zu aktivieren, über die Register der RealTimeClock eine Startzeit sowie ein Datum vorzugeben und schließlich über die Steuerungsregister die Uhr zu starten. Der Quellcode 5.4 gibt eine passende Initialisierungsmethode, analog zu den Treiberbeispielen aus [26], an.

globales Treiberobjekt

```
1  void RealTimeClock::init(){
2   WDTCTL = WDTPW | WDTHOLD;
3   PJSEL0 = BIT4 | BIT5;
4   PM5CTL0 &= ~LOCKLPM5;
5
6   // Configure LFXT 32kHz crystal
7   CSCTL0_H = CSKEY >> 8;
8   CSCTL4 &= ~LFXTOFF;
9   do
10  {
11    CSCTL5 &= ~LFXTOFFG;
12    SFRIFG1 &= ~OFIFG;
13  } while (SFRIFG1 & OFIFG);
14  CSCTL0_H = 0;
15
16  // Configure RealTimeClock
17  RTCCTL01 |= RTCHOLD;
18  //Time
19  RTCSEC = 0;
20  RTCMIN = 0;
21  RTCHOUR = 0;
22  //Date
23  RTCDOW = 0;
24  RTCDAY = 0;
25  RTCMON = 0;
26  RTCYEAR = 0;
27
28  //Start RealTimeClock
29  RTCCTL01 &= ~(RTCHOLD);
30 }
```

Quelltext 5.4: Notwendige Initialisierungsfunktionen der RealTimeClock.

Alarmzeit einstellen

Eine Hauptfunktionalität des Treibers stellt das Einstellen von Alarmzeiten dar. Hierzu steht die Methode setAlarm(int sec) zur Verfügung, die als Parameter die Gesamtsekundenanzahl der Alarmzeit übergeben bekommt. Aus der Sekundenanzahl kann anschließend die notwendige Startzeit der Uhr sowie die einzustellende Alarmzeit berechnet und über die entsprechenden Register gespeichert werden. Wenn die Uhr zusätzlich die Systemzeit angeben soll, ist es bei dem Einstellen einer neuen Startzeit notwendig, die bisherige Systemzeit zu speichern, um nach Ablauf des Alarms die Systemzeit durch Summierung von Alarmdauer und gespeicherter Systemzeit wiederherzustellen. Zur Speicherung der Systemzeit und des Datums wurden structs, bestehend aus int-Variablen, definiert, sodass entsprechende Attributvariablen zur Sicherung genutzt werden können.

Unterbrechungsbehandlung

Für eine entsprechende Behandlung der entstehenden Unterbrechungen ist es notwendig, dass das Betriebssystem Kratos die Unterbrechungsbehandlung an den Treiber der RealTimeClock weiterleiten kann. Hierzu ist es erforderlich, dass sich die RealTimeClock in die Abstraktion der Interruptvektortabelle, der Plugbox, einträgt. Der Aufruf dieser Methode kann, analog zum Beispielcode 5.2, mit Aspekten in eine Initialisierungsmethode der Plugbox eingewoben werden. Dies führt dazu, dass bei Eintreten eines Interrupts die Prologfunktion der RealTimeClock aufgerufen wird. In dieser Methode erfolgt die entsprechende Behandlung der Unterbrechung, wobei unterschieden wird, ob die Unterbrechung aufgrund einer periodischen Unterbrechung oder einer Alarmzeit erfolgte. Die Behandlungsroutine ist charakterisiert durch die Wiederherstellung der Systemzeit sowie Zurücksetzen der Alarmregister. Im anschließenden Epilog wird eine Überprüfung der Buzzer veranlasst. Die sich ergebende Unterbrechungsbehandlung ist im Quelltext 5.5 angegeben.

```
1  bool RealTimeClock::prologue(){
2   switch(RTCIV){
3    // Interval Alarm
4    case RTCIV_RTCTEVIFG:
5     // Register Reset
6     RTCCTL01 &= ~(RTCTEVIE | RTCTEVIFG);
7     // keinen Epilog anfordern
8     return false;
9    // Alarmzeit
10   case RTCIV_RTCAIFG:
11    // Register Reset
12    RTCCTL01 &= ~(RTCAIFG | RTCAIE);
13    // Zeiten wiederherstellen
14    resetTime();
15    aTime.min = 0; aTime.hour = 0; startSec=0;
16    // Epilog anfordern
17    return true;
18  }
19  // keinen Epilog anfordern
20  return false;
21 }
22
23 void RealTimeClock::epilogue(){
24  // Buzzer ueberpruefen
25  rtc_bellringer.check();
26 }
```

Quelltext 5.5: Unterbrechungsbehandlung der RealTimeClock.

5.2.2. RTC_Buzzer und RTC_Bellringer

Attribute der Buzzer

Die RTC_Buzzer dienen als Synchronisationsobjekte, mit denen sich Threads für eine bestimmte Zeit schlafen legen können. Hierfür können die Threads die implementierten sleep()-Methoden nutzen, wodurch das entsprechende Buzzerobjekt den aktuellen Thread, die zu wartende Zeit und optional noch einen Identifier für eine spätere Identifizierung speichert. Zusätzlich besitzen RTC_Buzzer noch zwei weitere Attribute, um die bisher gewartete Zeit sowie der Restzeit bezüglich eines vorherigen Buzzers speichern zu können. Bei Aufruf der sleep()-Methode eines Buzzers wird dieser vom RTC_Bellringer in die Liste der bisherigen Buzzer eingefügt. Der RTC_Bellringer ist für die Verwaltung der genutzten Buzzer und Einstellung der richtigen Alarmzeit zuständig. Wird mehr als ein Buzzer verwendet, muss der RTC_Bellringer entscheiden, welcher Buzzer zuerst abläuft und somit an erster Stelle in der Liste einsortiert werden muss. Sollte ein Buzzer aufgrund seiner geringeren Wartezeit einen anderen Buzzer als erstes Listenelement verdrängen, muss die Alarmzeit der RealTimeClock dementsprechend angepasst werden.

Angabe der Identifier

Um dem Entwickler eine bequeme und übersichtliche Möglichkeit zur Angabe von Wartezeiten zu bieten, wurden in der Basisklasse basic_times Präprozessormakros definiert, die eine Angabe der Identifier sowie eine Auflistung der Wartezeiten ermöglichen. Die relevanten Makros zeigt der Quelltext 5.6. Dabei ist hervorzuheben, dass für eine möglichst variable Definition von Identifier ein Template genutzt wird, welches eine generische Programmierung ermöglicht.
Mit Hilfe dieser Präprozessormakros können in der abgeleiteten Klasse buzzer_times konkrete Identifier und Wartezeiten festgelegt werden. Der Beispielcode 5.7 definiert die Identifier DEMO_SENSOR1, DEMO_THREAD1 sowie SM_THREAD vom Typ Ident. Eine entsprechende Definition von Wartezeiten für die Identifier in Abhängigkeit von zwei Service-Level ist in Quelltext 5.8 angegeben.

```
1  // Makros fuer die Zeittabelle
2  #define TIME_TAB \
3    int Buzzer_Times::getTime(Ident id)\
4    {\
5    if(!d_sm) return DEFAULT_TIME;
6  // ID des Timer // akt SL // entsp. Zeit
7  #define ID(c_id, c_sl, c_time) \
8    if(id == c_id && d_sm->getSL() == c_sl) return c_time
       ;
9  #define END_TIME_TAB return DEFAULT_TIME;};
10
11 // Default Zeit
12 #define DEFAULT_TIME 5
13 // Makro fuer Definition der genutzten Identifier
14 #define DEF_IDENT( E, e, ...) typedef enum E {
       eIdentVoid=0, __VA_ARGS__ } e;
15 template<typename E>
```

Quelltext 5.6: Präprozessormakros zur Definition von Identifier und Wartezeiten.

```
1  DEF_IDENT( Ident, ident, DEMO_SENSOR1, DEMO_THREAD1,
       SM_THREAD );
```

Quelltext 5.7: Definition von Identifier mittels Präprozessormakros.

```
1  TIME_TAB
2   // Identifier, Service-Level, Zeit
3   ID(DEMO_SENSOR1, 0, 600)
4   ID(DEMO_SENSOR1, 1, 30)
5   ID(DEMO_THREAD1, 0, 500)
6   ID(DEMO_THREAD1, 1, 20)
7   ID(SM_THREAD, 0, 100)
8   ID(SM_THREAD, 1, 50)
9  END_TIME_TAB
```

Quelltext 5.8: Definition von Wartezeiten für die Identifier mittels Präprozessormakros.

5.3. Implementierung der Statemachine

Struktur der Statemachine

Bei der Implementierung der Statemachine wurden bewusst nur die wichtigsten Grundfunktionalitäten umgesetzt, um den entstehenden Overhead bei der Speicherbelegung minimal zu gestalten. Daher beschränkt sich die Realisierung der Statemachine auf drei zu implementierende Klassen. Die genaue Struktur ist in Abbildung 5.4 dargestellt.

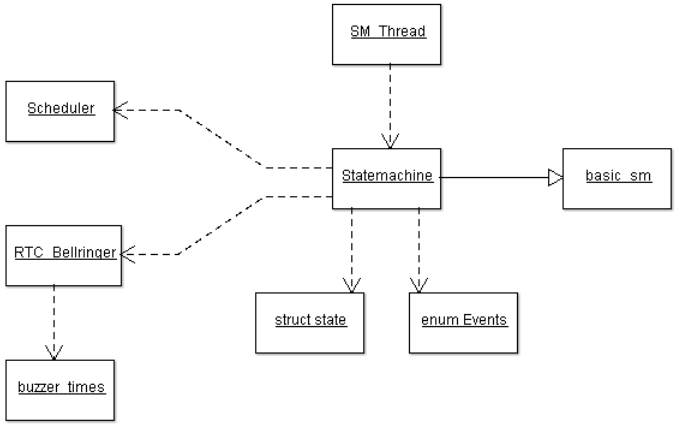

Abbildung 5.4.: Einbettung der Statemachine in das Betriebssystem.

Nutzung von Präprozessormakros

Die Basisklasse basic_sm dient analog zur Klasse basic_times der Definition von Präprozessormakros, die vom Entwickler zur Angabe von möglichen Events sowie der Zustandsübergangstabelle genutzt werden. Die Definition der Zustandsübergangstabelle erfolgt über die beiden Makros CON(curLevel, cond, eEvent, nextLevel) sowie CON_AS(cond, eEvent, nextLevel), denen verschiedene Funktionsparameter übergeben werden können. Hierzu gehört der aktuelle Service-Level, eine Bedingung für die Systemvariablen, ein notwendiges Event sowie der entsprechend neue Service-Level. Durch die beiden Präprozessormakros hat der Entwickler die Möglichkeit, Zustandsübergänge entweder unabhängig oder abhängig vom aktuellen Service-Level anzugeben.

Die von der Basisklasse abgeleitete Statemachine-Klasse dient der Definition eines konkreten Zustandsautomatens. Hierfür steht in der

Header-Datei der Klasse ein struct zur Verfügung, in welchem alle notwendigen Systemvariablen deklariert und abgespeichert werden können. Weiterhin wird in dieser Klasse mittels der Präprozessormakros der Basisklasse eine Zustandsübergangstabelle durch den Entwickler angegeben. Zusätzlich werden in der Statemachine-Klasse die Eintrittsfunktionen definiert, welche beim Wechsel in ein neues Service-Level ausgeführt werden. Hierzu kann sehr gut die aspektorientierte Programmierung eingesetzt werden, indem für jedes Service-Level leere Funktionen erstellt werden, die beliebig über Aspekte gefüllt werden können. Dies ermöglicht eine hohe Konfigurierbarkeit der Statemachine.

Angabe der Übergangstabelle

Die letzte zu implementierende Klasse ist der SM_Thread, welcher eine periodische Aktualisierung der Systemvariablen sowie eine regelmäßige Überprüfung der Statemachine sicherstellen soll. Hierzu wird ein RTC_Buzzer verwendet, da für diesen je nach aktuellen Service-Level unterschiedliche Ausführungsfrequenzen leicht definiert werden können. Im Quelltext 5.9 ist eine entsprechende Implementierung der wichtigsten Funktionen des SM_Threads angegeben.

periodische Aktualisierungen

```
1  void SM_Thread::action(){
2   Guarded_RTCBuzzer buz;
3   while(1){
4    check();
5    buz.sleep(SM_THREAD);
6   }
7   for(;;);
8  }
9
10 void SM_Thread::check(){
11  Secure sec;
12  update();
13  sm.checkState();
14 }
15
16 void SM_Thread::update(){
17  // ab hier ueber Aspekte fuellen
18 }
```

Quelltext 5.9: Implementierung der wichtigsten Funktionen des SM_Threads.

Epilogebene

Die check()-Methode veranlasst eine Aktualisierung der Systemvariablen sowie eine Überprüfung des Zustandsautomatens. Hierbei gewährleistet die vorherige Erstellung eines Secure-Objektes, dass diese Vorgänge auf der Epilog-Ebene ausgeführt werden. Dies verhindert, dass während Aktualisierungen oder Überprüfungen Prozesswechsel durch den Scheduler erfolgen können. Dies ist notwendig, da ein anderer Prozess die Systemvariablen verändern und den Systemzustand somit verfälschen könnte. Eine unterbrochene Überprüfung der Zustandsübergangstabelle könnte dann zu einem falschen Service-Level führen. Die update()-Methode des Threads ist leer und kann über Aspekte gefüllt werden. Eine beispielhafte Aktualisierung der gespeicherten Systemzeit wird durch den Aspekt des folgenden Quelltextes erreicht.

```
1  #ifndef ASPECT_TIME_AH
2  #define ASPECT_TIME_AH
3
4  #include "statemachine.h"
5  #include "SM_Thread.h"
6  #include "clocks/rtcb.h"
7
8  extern Statemachine sm;
9  extern RealTimeClock rtcb;
10
11 aspect Update_Time {
12  advice execution("void SM_Thread::update()") : after()
13  {
14   rtcb.returnTime(sm.systemstate.stime);
15  }
16 };
17
18 #endif
```

Quelltext 5.10: Aktualisierung der gespeicherten Systemzeit.

5.4. Beispielanwendung

Um die adaptiven Fähigkeiten des Systems zu testen, wurden unterschiedliche Beispielanwendungen implementiert. Eine Anwendung im Bereich des energiegewahren Service-Level-Management kann beispielsweise die Anpassung von Ausführungsfrequenzen an die erzeugte Energie des Systems sein. Um dies zu simulieren kann das MSP430 LaunchPad mit einem Lichtsensor ausgestattet werden. Dieser misst die Beleuchtungsstärke in Lux, welche die Energiegewinnung durch eine Solarzelle repräsentieren und als Systemvariable für die Statemachine dienen kann. Abhängig von dieser Systemvariablen kann die Ausführungsfrequenz eines Anwendungsthreads gesteuert werden.

Lichtwert als Energieniveau

Um einen zeitlichen Verlauf der erzeugten Energie darzustellen, werden die drei letzten abgefragten Sensorwerte des Lichtsensors als Systemvariablen abgespeichert. Basierend auf diesen Werten kann eine Zustandsübergangstabelle angegeben werden, die die Ausführungsfrequenz eines Anwendungsthreads verändert. Als Anwendungsthread kommt in diesem Beispiel ein Thread zum Einsatz, der eine LED des LaunchPads für eine Sekunde aufblinken lässt. Die Frequenzen des Blinkens sollen dabei abhängig vom aktuellen Service-Level gewählt werden.

Der Quellcode 5.11 zeigt den Aspekt, der für die Aktualisierung der Sensorwerte genutzt wird. Über das Treiberobjekt lightsensor wird der aktuelle Lichtwert abgefragt und anschließend in der Statemachine gespeichert. Ebenfalls erfolgt eine Verschiebung der älteren Sensorwerte.

```
#ifndef ASPECT_LIGHT_AH
#define ASPECT_LIGHT_AH

#include "statemachine.h"
#include "drivers/max44009.h"

extern MAX44009 lightsensor;

aspect Update_Light {
  advice execution("void QOS_Thread::update()") : after
      () {
   sm.systemstate.energy3 = sm.systemstate.energy2;
   sm.systemstate.energy2 = sm.systemstate.energy1;
   sm.systemstate.energy1 = (int)lightsensor.getLux();
  }
};

#endif
```

Quelltext 5.11: Aktualisierung der gespeicherten Lichtwerte des Lichtsensors.

Im Quellcode 5.12 ist die Definition der Zustandsübergangstabelle angegeben. Abhängig von den gespeicherten Sensorwerten wird zwischen drei Service-Leveln unterschieden. Die gewählten Werte wurden dabei empirisch ermittelt.

```
SL_CON_TAB
 CON(2, energy1 > 700 && energy2 > 700 && energy3 >
     700, NO_EVENT, 3);
 CON(1, energy1 > 500 && energy2 > 500 && energy3 >
     500, NO_EVENT, 2);
 CON_AS(energy1 < 300 && energy2 < 300 && energy3 <
     300, NO_EVENT, 1);
END_TAB
```

Quelltext 5.12: Definition der Zustandsübergangstabelle.

Um eine Adaption der Ausführungsfrequenz des LED_Threads zu erreichen, nutzt der Buzzer des Threads einen Identifier. Dadurch kann bei einem Wechsel des Service-Levels die Wartezeit des Buzzers angepasst werden. Im Quellcode 5.13 ist die entsprechende Tabelle der Klasse buzzer_times aufgelistet. Je nach Service-Level erfolgt ein

unterschiedliche Frequenzen

Blinken der LED alle 300, 30 oder 3 Sekunden. Außerdem findet bei dem niedrigsten Service-Level eine Anpassung der Aktualisierungs- und Überprüfungsfrequenz des SM_Threads statt.

```
1 TIME_TAB
2  // Identifier, Service-Level, Zeit
3  ID(LED_THREAD, 1, 300)
4  ID(LED_THREAD, 2, 30)
5  ID(LED_THREAD, 3, 3)
6  ID(SM_THREAD, 1, 100)
7  ID(SM_THREAD, 2, 5)
8  ID(SM_THREAD, 3, 5)
9 END_TIME_TAB
```

Quelltext 5.13: Definition der Wartezeiten für die genutzten Identifier.

Diese Beispielanwendung zeigt, dass die implementierten Betriebssystemkomponenten eine adaptive Anpassung von Ausführungsfrequenzen ermöglichen. Bei Veränderung der Lichtverhältnisse, beispielsweise durch zusätzliche Beleuchtung oder Abdeckung des Lichtsensors, passt sich die Blinkfrequenz an.

6. Evaluation

In diesem Kapitel werden die entworfenen Betriebssystemkomponenten hinsichtlich ihrer Energie- und Speicherverbräuche evaluiert. Hierzu wird zunächst das MIMOSA Messsystem erörtert, sowie das grundlegende Evaluationsverfahren vorgestellt. Anschließend erfolgt eine Darstellung, Erläuterung sowie Bewertung der gemessenen Energieverbräuche. Weiterhin wird der zusätzlich benötigte Speicher durch die Nutzung der Komponenten analysiert. Abschließend werden Einschränkungen, Grenzen sowie Nutzen der Zusatzkomponenten diskutiert.

6.1. Messsystem MIMOSA

Eine grundlegende Aufgabe für die Senkung des Energieverbrauchs im Bereich der eingebetteten Systeme ist es, Energiemodelle für Hard- und Softwarekomponenten zu erstellen, um mögliche Einsparpotenziale zu erkennen. Dies erfordert häufig eine genaue Messung des Energie- und Zeitverhaltens eines solchen Systems. Im folgenden Abschnitt wird MIMOSA, ein am Lehrstuhl für eingebettete Systeme der TU Dortmund entworfenes Messgerät zur integrativen Messung ohne Spannungsabfall, erläutert. MIMOSA stellt eine kostengünstige Messvorrichtung dar, die schnelle Spannungsregler und analoge Integrationsschaltungen verwendet.

Messungen notwendig

Die verbrauchte Leistung $P(t)$ wird in diesem Kontext definiert durch die Stromstärke $I(t)$ die bei einer konstanten Versorgungsspannung V_{Vers} in einem System fließt:

$$P(t) = V_{Vers} * I(t). \tag{6.1}$$

Der Energieverbrauch E ist demnach definiert durch die verbrauchte

Leistung $P(t)$ in einem bestimmten Zeitraum T:

$$E = \int^{T} P(t)dt = \int^{T} V_{Vers} * I(t)dt. \tag{6.2}$$

Es ist allerdings schwierig, die verbrauchte Leistung beziehungsweise Energie direkt zu messen. Es ist einfacher, zuerst die Faktoren Spannung, Stromstärke und Zeit zu bestimmen und daraus im Anschluss den Leistungs- beziehungsweise Energieverbrauch zu berechnen. Die Stromstärke kann wiederum auch nur indirekt gemessen werden, z.B. über einen Spannungsabfall V, der durch das Ohmsche Gesetz wie folgt beschrieben wird:

$$V = R * I(t). \tag{6.3}$$

Spannungsabfall über Shunt

Bei diesem Vorgehen wird ein sehr genauer ohmscher Widerstand R, auch Shunt genannt, verwendet, um an diesem Widerstand den Spannungsabfall zu messen und so den aktuellen Stromfluss zu berechnen. Hierbei ist darauf zu achten, dass der Widerstand groß genug ist, um einen messbaren Spannungsabfall zu generieren. Allerdings darf der Shunt nicht zu groß gewählt werden, da auch ein zu hoher Abfall das Messergebnis negativ beeinflussen kann.

Abtastrate kritisch

Ein weiterer zu beachtender Punkt ist die Abtastungsrate des Spannungsabfalls während des Messvorgangs. Um kurzzeitige Signaländerungen genau erfassen zu können, wird eine sehr hohe Abtastrate benötigt. Aus diesem Grund gibt es bei MIMOSA drei Integratoren, die jeweils versetzt den Spannungsabfall abtasten, integrieren und abschließend löschen. Durch die Verwendung von drei Integratoren und drei Messphasen ist eine lückenlose Abtastung und Berechnung des Energieverbrauchs möglich.

Aufbau

In Abbildung 6.1 ist der Aufbau des MIMOSA Messsystems dargestellt. Die untere Platine (gelb) bietet die Möglichkeit verschiedene Widerstände als Shunt auszuwählen. Die Platine versorgt das zu prüfende Gerät, hier das MSP430 LaunchPad (rot) mit einer konstanten Versorgungsspannung und generiert über den Shunt einen Spannungsabfall. Dieser Spannungsabfall dient als Eingangssignal des MIMOSA Messsystems (grün). Die Differenzspannung wird abwechselnd von

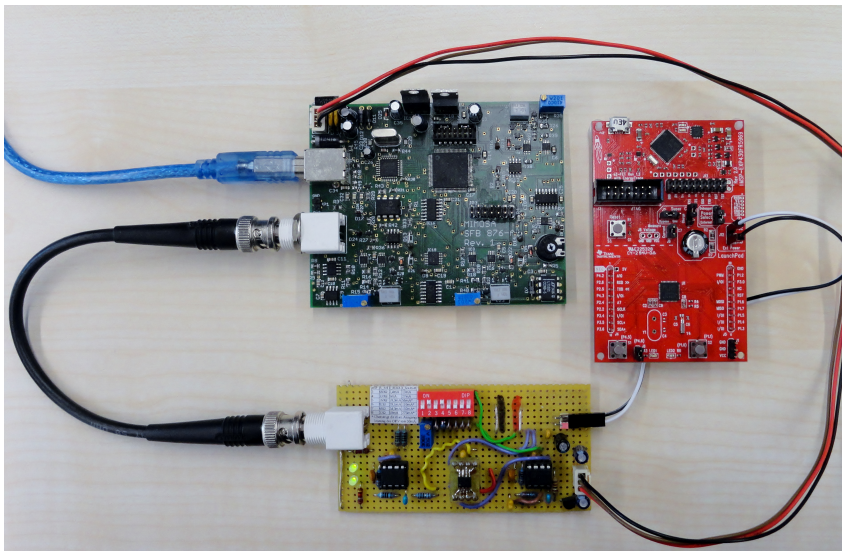

Abbildung 6.1.: Aufbau des MIMOSA Messsystems. Unten befindet sich eine Platine, die den notwendigen Spannungsabfall generiert und das zu prüfende Gerät mit Spannung versorgt. Das MIMOSA Messsystem überträgt die integrierten Daten an einen Computer.

einem der drei Integratoren bearbeitet. Per USB-Anschluss (blau) können die Messergebnisse an einen Computer übertragen werden, wo sie mittels einer graphischen Oberfläche veranschaulicht werden.

6.2. Evaluationsverfahren

Die Evaluierung des Energieverbrauchs der neuen Kratosversion erfolgt dreigeteilt, sodass die neu implementierten Funktionalitäten des Systems getrennt voneinander betrachtet werden können. Es wird daher der Energieverbrauch für das Überprüfen der Zustandsübergangstabelle, das Pollen von Systemvariablen sowie das Aktualisieren der Wartezeiten der RTC_Buzzer separat bestimmt und ausgewertet.

Evaluierung im Thread

Um den Energieverbrauch der einzelnen Komponenten zu ermitteln, wurde der in Kapitel 5 vorgestellte SM_Thread so modifiziert, dass bei der erstmaligen CPU-Zuteilung durch den Scheduler auf Epilog-Ebene gewechselt wird, um eine Prozessverdrängung des präemptiven Schedulings zu umgehen. Im Anschluss findet eine mehrmalige Ausführung der zu evaluierenden Funktionalität statt. Abschließend wird die Epilog-Ebene verlassen und ein Wechsel zum idle-Betrieb des Sy-

stems ausgeführt. Während der Messung ist es möglich, über den Reset-Taster des MSP430 LaunchPads einen Neustart des Systems auszulösen, welcher zu einer erneuten, mehrmaligen Ausführung der Funktionalität führt. Mit diesem Verfahren ist es möglich, mehrere Messergebnisse für eine entsprechende Funktion des Systems zu erhalten.

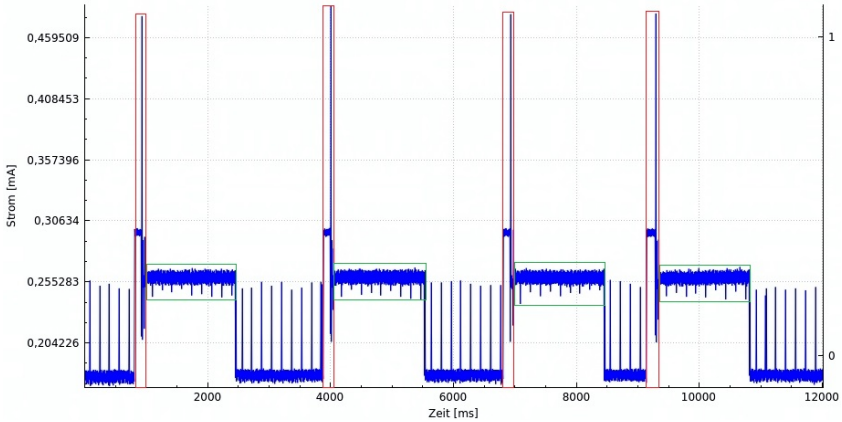

Abbildung 6.2.: Beispielmessung mit MIMOSA. Rot gekennzeichnet sind die Neustarts des Systems, grün markiert sind zu untersuchenden Ausführungen, in diesem Fall das Überprüfen der Zustandsübergangstabelle der Statemachine.

Messverläufe

In Abbildung 6.2 ist eine Beispielsmessung für das Überprüfen der Zustandsübergangstabelle der Statemachine dargestellt. Dabei werden auf der y-Achse die gemessenen Stromstärken in Milliampere angegeben. Rot gekennzeichnete Bereiche sind Neustarts des Systems, um erneut eine Überprüfungssequenz auszulösen. Die entsprechende Überprüfungssequenz ist grün markiert. Die MIMOSA Software bietet die Möglichkeit den relevanten Messbereich auszuwählen und den in diesem Bereich verursachten Energieverbrauch zu berechnen. Dies wird im Weiteren genutzt, um die Energieverbräuche der Überprüfungssequenzen zu ermitteln und anschließend einen Mittelwert zu bestimmen.

6.3. Messergebnisse der Energieverbräuche

Mit dem zuvor beschriebenen Verfahren können für die implementierten Komponenten bzw. Funktionalitäten entsprechende Messreihen

durchgeführt und Energieverbräuche bestimmt werden. Hierbei ist zu beachten, dass die ermittelten Verbräuche nicht nur von der untersuchten Komponente verursacht wurden. Es muss beachtet werden, dass das System auch im idle-Modus Energie benötigt. Diese benötigte Energie stellt daher ein Minimum an Energie dar, welche in jedem Fall verbraucht wird. Ein Aspekt hierbei sind beispielsweise die durch den Timer verursachten Interrupts. Auch wenn der SM_Thread auf Epilog-Ebene ausgeführt und somit ein präemptives Scheduling verhindert wird, werden dennoch weiterhin Interrupts durch den Timer verursacht, deren Unterbrechungsbehandlung ebenfalls Energie benötigt und somit mit den gemessenen Energieverbräuchen verrechnet werden muss.

grundlegender Verbrauch

Bezeichne $M(T)$ den gemessenen Energieverbrauch über einen Zeitraum T und $ID(T)$ den im idle-Modus benötigten Verbrauch für den Zeitraum T, so ist der durch die Komponente verursachte Energieverbrauch $EV(T)$ definiert durch:

$$EV(T) = M(T) - ID(T). \tag{6.4}$$

Im weiteren Verlauf werden daher die Energieverbräuche abzüglich des Verbrauchs des Systems im idle-Modus angegeben. Hierzu wurde im Vorfeld der Verbrauch von Kratos im idle-Zustand bestimmt, in welchem kein Anwendungsthread implementiert wurde, sodass das Betriebssystem nach Systemstart sofort in einen Energiesparmodus übergeht. Es konnte so ein idle-Verbrauch von 438 Mikrojoule pro Sekunde bestimmt werden.

6.3.1. Überprüfung der Zustandsübergangstabelle

Eine zu untersuchende Hauptfunktionalität ist die Überprüfung der Zustandsübergangstabelle der vom Entwickler definierten Statemachine. Das Ziel dieser Messungen ist es, eine Abschätzung des verursachten Energieverbrauchs abhängig von der Anzahl der Einträgen in der Zustandsübergangstabelle zu bestimmen. Hierzu wurden die Energieverbräuche für Zustandsübergangstabellen mit 4, 6, 8 und 10 Einträgen bestimmt. Grundlage für die Bedingungen sind drei Sy-

Abschätzung gesucht

stemvariablen, ein Energiewert (Typ int), die Anzahl der Systemticks (Typ unsigned long) sowie die Systemzeit (Struct aus drei int-Variablen).

Qualität Messdaten

Bevor ein Abschätzungsmodell bestimmt werden kann, ist ein weiterer notwendig zu beachtender Aspekt die vorherige Bewertung der Messdaten. Für die 4, 6, 8 und 10 Einträge in der Zustandsübergangstabelle wurden nach dem beschriebenen Prinzip jeweils vier Messwerte gewonnen und mit dem idle-Verbrauch des Systems verrechnet. Anschließend ist die Qualität dieser Daten hinsichtlich ihrer Streuung zu beurteilen.

					$\overline{x}$	A_{abs}
Dauer[ms]	1478	1483	1492	1488	1485,25	4,75
Energie[mJ]	0,4998	0,5026	0,5037	0,5014	0,5019	0,00126

Tabelle 6.1.: Auflistung der vier Messwerte bei vier Einträgen in der Zustandsübergangstabelle mit Angabe des arithmetischen Mittels sowie der Gesamtabweichung vom Mittelwert.

Abweichungen

In Tabelle 6.1 sind die Messwerte näher charakterisiert durch die Dauer der Messung in Millisekunden und den Energieverbrauch in Millijoule , für vier Einträge in der Zustandsübergangstabelle angeben. Ebenso wurde das arithmetische Mittel $\overline{x}$ und die absolute Gesamtabweichung A_{abs} vom Mittelwert $\overline{x}$ bestimmt, welche wie folgt definiert ist:

$$A_{abs} = \sum_{i=1}^{N} |(x_i - \overline{x})|. \tag{6.5}$$

Anhand dieser Kennzahlen ist es möglich, die Qualität der Messwerte zu bestimmen. Wird die absolute Gesamtabweichung vom Mittelwert A_{abs} als ein Fehlermaß angesehen, so beträgt der Fehler 1,28 Prozent bei der gemessen Dauer und 1 Prozent bei dem ermittelten Energieverbrauch im Bezug auf die Durchschnittswerte. Die Messwerte können daher als gut empfunden werden und sind für eine weitere Auswertung geeignet.

In Tabelle 6.2 sind die Durchschnittswerte, Gesamtabweichungen A_{abs} und prozentualen Fehlermaße für die weiteren Messreihen, abhängig von der Anzahl der Einträge in der Zustandsübergangstabelle darge-

stellt. Anhand der Werte für den prozentualen Fehler ist erkennbar, dass auch die Ergebnisse der weiteren Messreihen geeignet sind für eine weitere Auswertung. Die entstandenen Abweichungen sind auf Messungenauigkeiten sowie auf Fehler bei der Markierung der relevanten Messbereiche zurückzuführen.

Anzahl Einträge	4	6	8	10
Durchs. Dauer[ms]	1485,25	1711,5	1818	2355,75
Abweichung A_{abs}	19	28	36	31
Prozentualer Fehler	1,28	1,64	1,98	1,31
Durchs. Energie[mJ]	0,502	0,746	1,264	1,621
Abweichung A_{abs}	0,005	0,006	0,011	0,012
Prozentualer Fehler	1,00	0,90	0,93	0,74

Tabelle 6.2.: Auflistung der Durchschnittswerte, Gesamtabweichungen und prozentuale Fehlermaße abhängig von der Anzahl der Einträge in der Zustandsübergangstabelle.

Die berechneten Durchschnittswerte für den Energieverbrauch und die Dauer können nun genutzt werden, um ein Abschätzungsmodell zu bestimmen. Es wird erwartet, dass es sich um ein lineares, proportionales Modell handelt, sodass eine Verdoppelung der Einträge in der Zustandsübergangstabelle ebenfalls zu einer Verdoppelung des Energieverbrauchs führen wird. Der Grund für diese Annahme ist, dass für jeden Eintrag in der Zustandsübergangstabelle eine vergleichbare Menge an C++-Codes erzeugt wird (pro Eintrag je zwei if-Anweisungen).

proportionales Modell erwartet

In Abbildung 6.3 ist der gemessene Energieverbrauch gegenüber der Anzahl der Einträge in der Zustandsübergangstabelle graphisch dargestellt. Es ist anhand der Kennlinie erkennbar, dass es sich annähernd um ein lineares Modell handelt. Allerdings verhalten sich die Einträge in der Zustandsübergangstabelle und die Energieverbräuche nicht proportional zueinander. Dies wird besonders deutlich, wenn das Verhältnis von Energieverbrauch zur Anzahl der Einträge betrachtet wird. Dieses Verhältnis ist in Tabelle 6.3 aufgelistet.

nur lineares Modell

Um einen proportionalen Zusammenhang zwischen Energieverbrauch und Anzahl der Einträge in der Übergangstabelle herstellen zu können, müssten die berechneten Verhältnisse eine geringere Streuung aufweisen. Da dies allerdings nicht der Fall ist, kann von keinem pro-

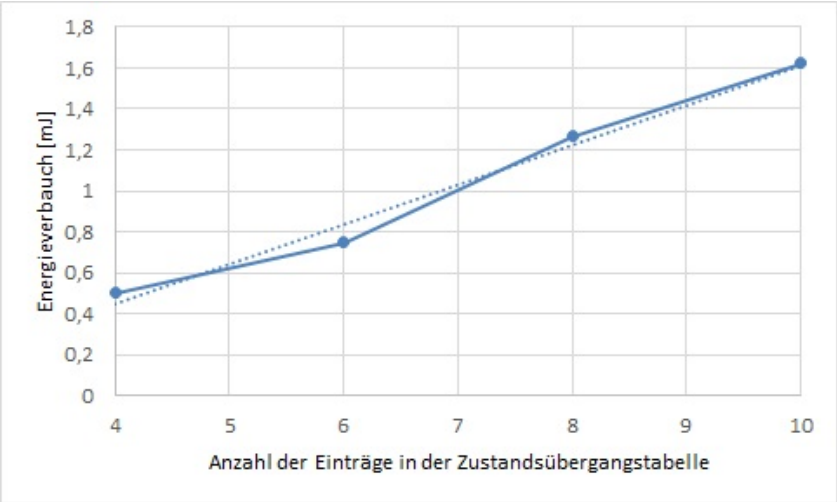

Abbildung 6.3.: Verhältnis zwischen Energieverbrauch und Einträgen in der Zustandsübergangstabelle. Anhand der Kennlinie ist erkennbar, dass es sich um ein lineares, jedoch nicht proportionales Modell handelt.

Anzahl Einträge	4	6	8	10
Durchs. Energie[mJ]	0,502	0,746	1,264	1,621
Verhältnis Energieverbrauch zu Anzahl Einträge	0,1254	0,1242	0,1579	0,1620

Tabelle 6.3.: Auflistung der Verhältnisse von Energieverbrauch zur Anzahl der Einträge. Um von einem proportionalen Verhältnis ausgehen zu können, müssen die Verhältnisse konstanter sein.

portionalen Zusammenhang ausgegangen werden. Da jedoch theoretisch ein solcher Zusammenhang erwartet wurde, muss dieser Sachverhalt genauer betrachtet werden. Eine Vermutung ist, dass beim Kompilieren des C++-Quellcodes Optimierungen durch den Compiler vorgenommen worden sind. Um dies zu überprüfen, muss das entstandene Betriebssystemimage zu Assemblercode dekompiliert werden. Dies ist mittels des Programms msp430-objdump möglich. Anhand des gewonnen Assemblerquelltextes kann die Anzahl der Assembleranweisungen für die entsprechende Überprüfungsmethode bestimmt werden.

Assemblercode

Die Anzahl der Assembleranweisungen sowie die sich daraus ergebenen Verhältnisse von Energieverbrauch zur Anzahl an Assembleranweisungen ist in Tabelle 6.4 dargestellt. Gegenüber den vorherigen

Verhältnissen hat sich die Streuung deutlich reduziert, die verbliebene Streuung ist auf Messfehler zurückzuführen. Dies bestätigt den Verdacht, dass durch die Optimierungen durch den Compiler verhältnismäßig weniger Assembleranweisungen bei weniger Einträgen in der Übergangstabelle benötigt werden. Dies erklärt den ebenfalls verhältnismäßig geringeren Energieverbrauch bei wenigen Einträgen.

Optimierungen durch Compiler

Anzahl Einträge	4	6	8	10
Anzahl Assembleranweisungen	59	87	147	190
Durchs. Energie[mJ]	0,502	0,746	1,264	1,621
Verhältnis zwischen Energieverbrauch und Assembleranweisungen	0,00850	0,00857	0,00859	0,00853

Tabelle 6.4.: Auflistung der Verhältnisse von Energieverbrauch zur Anzahl der Assembleranweisungen. Es liegen deutlich konstantere Verhältnisse mit geringer Streuung vor.

In Abbildung 6.4 ist nun der gemessene Energieverbrauch gegenüber der Anzahl der entstehenden Assembleranweisungen graphisch dargestellt. Im Gegensatz zur Abbildung 6.3 ist der proportionale Zusammenhang klar erkennbar.

proportionaler Zusammenhang

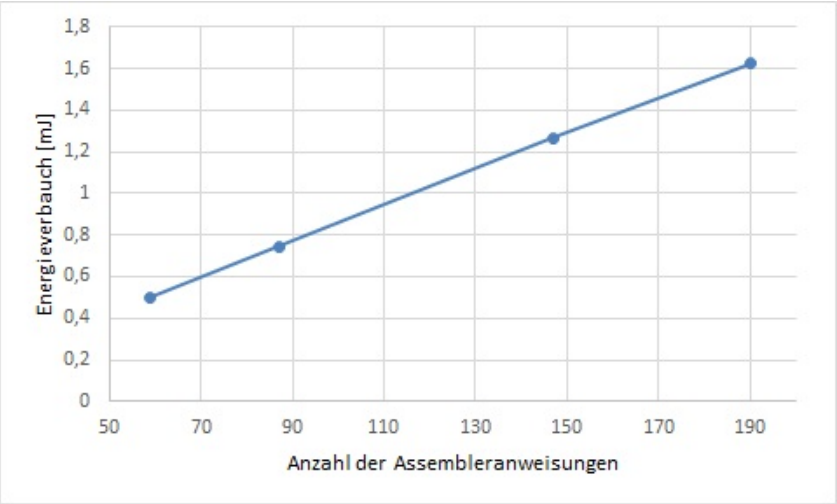

Abbildung 6.4.: Verhältnis zwischen Energieverbrauch und Anzahl der Assembleranweisungen. Es ist offensichtlich, dass ein proportionaler Zusammenhang vorliegt.

Die Optimierungen durch den Compiler ermöglicht leider nur eine begrenzte Abschätzung des Energieverbrauchs pro Eintrag in der Zu-

standsübergangstabelle. In den durchgeführten Messungen schwankte der entstehende Energieverbrauch für die Überprüfung zwischen 4,14 Nanojoule pro Eintrag (bei 4 Einträgen) und 5,4 Nanojoule pro Eintrag (bei 10 Einträgen).

6.3.2. Polling von Systemvariablen

Die zweite zu untersuchende Hauptfunktionalität ist das Polling der genutzten Systemvariablen der Statemachine. Das Ziel dieser Messungen ist es, analog zum vorherigen Abschnitt eine Abschätzung des verursachten Energieverbrauchs in Abhängigkeit von der Anzahl der zu aktualisierenden Variablen zu bestimmen. Hierzu wurden die Energieverbräuche für die Polling-Vorgänge mit 3, 6 und 9 Systemvariablen gemessen. Die Systemvariablen bestanden dabei aus den Variablentypen int, unsigned long und struct, wobei letzteres aus drei int-Variablen besteht.

Qualität Messdaten

Wie bei der Auswertung des Energieverbrauchs für das Überprüfen der Zustandsübergangstabelle soll zunächst die Qualität der Messdaten bestimmt werden. Es wurden für jeden Polling-Vorgang ebenfalls vier Messwerte aufgezeichnet und mit dem idle-Verbrauch des Systems verrechnet. Eine Bewertung der Daten soll ebenfalls anhand des arithmetischen Mittels $\overline{x}$ und der absoluten Gesamtabweichung A_{abs} vom Mittelwert $\overline{x}$ erfolgen.

In Tabelle 6.5 sind die Durchschnittswerte, Gesamtabweichungen A_{abs} und prozentuale Fehlermaße für die entsprechenden Messreihen abhängig von der Anzahl der Systemvariablen der Statemachine dargestellt. Anhand der prozentualen Fehler ist erkennbar, dass die Ergebnisse für eine weitere Auswertung geeignet sind, da der maximale Fehler 2,53 Prozent beträgt. Diese Abweichungen sind ebenfalls auf Messungenauigkeiten sowie auf Markierungsfehler bei der Auswahl der relevanten Messbereiche zurückzuführen.

geringe Fehler

Für ein Abschätzungsmodell wird erneut erwartet, dass es sich um ein linearer, proportionaler Zusammenhang zwischen der Anzahl der Systemvariablen und dem Energieverbrauch vorliegt. Um dies zu überprüfen wird analog zu den Ergebnissen des vorherigen Abschnitts das Verhältnis von Energieverbrauch zur Anzahl der Systemvariablen be-

proportionales Verhältnis

Anzahl Variablen	3	6	9
Durchs. Dauer[ms]	2414,25	4507	6227,75
Abweichung A_{abs}	25	50	17
Prozentualer Fehler	1,04	1,11	0,27
Durchs. Energie[mJ]	0,6798	1,2416	1,7283
Abweichung A_{abs}	0,0024	0,0313	0,0088
Prozentualer Fehler	0,36	2,53	0,51

Tabelle 6.5.: Auflistung der Durchschnittswerte, Gesamtabweichungen und prozentuale Fehlermaße abhängig von der Anzahl der Systemvariablen.

trachtet. Dieses Verhältnis ist in Tabelle 6.6 aufgelistet.

Anzahl Variablen	3	6	9
Durchs. Energie[mJ]	0,6798	1,2416	1,7283
Verhältnis Energieverbrauch zu Anzahl Variablen	0,2266	0,2069	0,1920

Tabelle 6.6.: Auflistung der Verhältnisse von Energieverbrauch zur Anzahl der zu aktualisierenden Systemvariablen. Aufgrund der größeren Schwankungen kann von keinem proportionalen Verhältnis ausgegangen werden.

Anhand der berechneten Verhältnisse ist festzustellen, dass kein proportionaler Zusammenhang vorliegt. Das Verhältnis von Energieverbrauch zu Anzahl der genutzten Variablen ist dabei für neun Variablen am geringsten. In Verbindung mit den zuvor evaluierten Erkenntnissen führt dies zu dem Verdacht, dass für eine größere Anzahl an Variablen ebenfalls eine Quellcodeoptimierung durch den Compiler möglich ist. Um dies zu überprüfen wird das Betriebssystemimage ebenfalls mit dem Programm msp430-objdump zu Assemblercode dekompiliert und anschließend die Anzahl der Assembleranweisungen bestimmt.

ebenfalls Optimierungen

Die Anzahl der Assembleranweisungen sowie die sich ergebenen Verhältnisse von Energieverbrauch zu Anzahl an Assembleranweisungen ist in Tabelle 6.7 dargestellt. Verglichen mit den zuvor berechneten Verhältnissen ist eine deutliche Reduzierung der Streuung zu beobachten. Die verbliebenen Abweichungen zwischen den Verhältnissen sind wiederum auf Messfehler zurückzuführen. Diesmal ist bei der An-

ebenfalls proportionaler Zusammenhang

Anzahl Variablen	3	6	9
Anzahl Assembleranweisungen	15	28	39
Durchs. Energie[mJ]	0,6798	1,2416	1,7283
Verhältnis Energieverbrauch zu Anzahl Assembleranweisungen	0,04532	0,04434	0,04431

Tabelle 6.7.: Auflistung der Verhältnisse von Energieverbrauch zur Anzahl der Assembleranweisungen. Es liegen deutlich konstantere Verhältnisse mit geringer Streuung vor.

zahl der Assembleranweisungen ein verhältnismäßig geringerer Anstieg an Anweisungen bei mehr Systemvariablen zu verzeichnen.

Durch die Optimierungen während des Kompiliervorgangs ist eine Abschätzung des Energieverbrauchs pro Aktualisierung einer Systemvariablen wieder nur begrenzt möglich. Bei den durchgeführten Messungen konnten durchschnittliche Energieverbräuche von 7,6 Nanojoule, 6,9 Nanojoule sowie 6,4 Nanojoule ermittelt werden.

6.3.3. Aktualisierungen der Buzzer

Die letzte zu untersuchende Hauptfunktionalität des Systems ist das Aktualisieren der eingestellten Zeiten der Buzzer. Es wurden die verursachten Energieverbräuche bei der Aktualisierung von 1, 2, 3 und 4 Buzzer gemessen, sodass im Anschluss eine Abschätzung des Energieverbrauchs, abhängig von der Anzahl der verwendeten Buzzer, bestimmt werden kann.

Anzahl Buzzer	1	2	3	4
Durchs. Dauer[ms]	13794,5	17057	19303,5	21550
Durchs. Energie[mJ]	6,956	8,650	10,115	11,580

Tabelle 6.8.: Auflistung der Durchschnittsdauer sowie des durchschnittlichen Energieverbrauchs abhängig von der Anzahl der verwendeten Buzzer.

Die aus den Messergebnissen berechneten Durchschnittswerte sind in Tabelle 6.8 dargestellt. Das Vorliegen eines linearen Zusammenhangs wird in der graphischen Darstellung der Messwerte deutlich

(siehe Abbildung 6.5). Auffällig ist, dass der Zusammenhang zwischen Energieverbrauch und der Anzahl der verwendeten Buzzer linear, allerdings nicht proportional ist. Im Gegensatz zu den vorherigen Auswertungen kann in diesem Kontext nicht von einem proportionalen Verhältnis ausgegangen werden. Beim Überprüfen der Zustandsübergangstabelle und Polling von Systemvariablen bestand ein direkter Zusammenhang zwischen der Anzahl der Einträge in der Zustandsübergangstabelle bzw. der verwendeten Systemvariablen und der resultierenden C++-Quellcodegröße.

andere Erwartungen

Bei den Aktualisierungen der eingestellten Zeiten der Buzzer sind Codeabhängigkeiten bezüglich der Anzahl der verwendeten Buzzer festzustellen. Es gibt jedoch auch gewisse Funktionen, wie das Auslesen der RealTimeClock und die Bestimmung der bisher gewarteten Zeit, die unabhängig von der Anzahl der genutzten Buzzer sind. Der Energieverbrauch für diese Funktionen ist demnach auch unabhängig von der Buzzeranzahl, sodass von keinem proportionalen Verhältnis ausgegangen werden kann.

unabhängige Funktionen

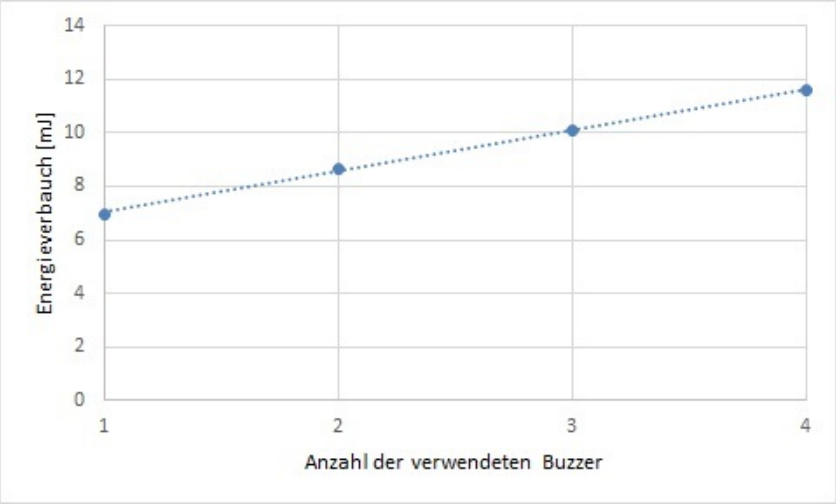

Abbildung 6.5.: Verhältnis zwischen Energieverbrauch und Anzahl der verwendeten Buzzer. Anhand der Kennlinie ist erkennbar, dass ein linearer Zusammenhang vorliegt.

6.3.4. Bewertung der Messergebnisse

Das durchgeführte Messverfahren stellte sich aufgrund der geringen Streuungen die beobachtet wurden, als geeignet heraus. Die Abweichungen zwischen den einzelnen Messwerten betrugen nur wenige Prozent und sind hauptsächlich auf Markierungsfehler bei der Auswahl der relevanten Messbereichen zurückzuführen.

Bei den Funktionalitäten der Statemachine wurden proportionale Zusammenhänge zwischen der Anzahl der Einträgen in der Zustandsübergangstabelle beziehungsweise der Variablenanzahl beim Polling und dem Energieverbrauch erwartet. In beiden Fällen konnten lineare Abhängigkeiten festgestellt werden. Proportionale Verhältnisse konnten jedoch erst bei der Berücksichtigung der Anzahl der Assembleranweisungen bestätigt werden. Ein Modell zur Abschätzung des Energieverbrauchs kann somit nur begrenzt angegeben werden. Für eine Anzahl x an Einträgen in der Zustandsübergangstabelle und y Variablen beim Polling kann der Energieverbrauch $E(x, y)$ in Nanojoule wie folgt geschätzt werden:

proportionale Verhältnisse

$$E(x, y) = 5,4x \ + \ 7,6y \quad [nJ]. \tag{6.6}$$

Für das Abschätzungsmodell wurden die verhältnismäßig höchsten Energieverbräuche gewählt. Falls eine Optimierung durch Compiler bei wenigen Einträgen in der Übergangstabelle oder bei mehreren Systemvariablen erfolgt, so weicht der Energieverbrauch nach unten ab.

Vergleich mit LED

Um den entstehenden Energieverbrauch besser bewerten zu können, soll das Verhältnis zu dem Leuchten einer LED betrachtet werden. Für eine LED des MSP430 LaunchPads wurde ein Energiebedarf von 11 Millijoule für eine Leuchtdauer von einer Sekunde ermittelt. Geschätzt benötigt eine Statemachine mit 10 Einträgen in der Übergangstabelle und 3 Variablen für einen Pollingvorgang und einer Überprüfung der Statemachine in etwa 77 Nanojoule. Dies bedeutet, dass nach dem vorliegenden Modell, die Statemachine in Vergleich zu einem ein-sekündigen Leuchtens der LED über 142.000 mal aktualisiert und überprüft werden könnte. Dieser Vergleich zeigt, dass der Energieoverhead durch Aktualisierungen und Überprüfungen der

Statemachine zu vernachlässigen ist.

Für die Aktualisierung der Buzzerzeiten wurden deutlich höhere Energieverbräuche gemessen. So hat die Aktualisierungen von einem Buzzer 2,31 Mikrojoule und die Anpassung von vier Buzzern 3,86 Mikrojoule verbraucht. Der im Vergleich zu den anderen Funktionalitäten hohe Energieverbrauch ist auf objekt-basierte Implementierung der Buzzer zurückzuführen. Bei der Aktualisierung werden Methoden der Buzzer, die alle ein eigenes Objekt darstellen, aufgerufen. Der Aufruf von Methoden anderer Objekte ist deutlich rechenintensiver als Methodenaufrufe des selben Objekts und erklärt damit den höheren Energieverbrauch. Im Verhältnis zum Energieverbrauch der LED ist dieser allerdings weiterhin als gering zu bezeichnen.

6.4. Speicherverbrauch

Es ist anzunehmen, dass durch die Verwendung der neuen Betriebssystemkomponenten der Speicherverbrauch deutlich erhöht wird. Neben der Ausführung der zusätzlichen Instruktionen benötigen die globalen Objekte Speicherplatz. Weiterhin nutzen die implementierten RTC_Buzzer mehrere int-Variablen als Attribute für die Speicherung von Wartezeiten. Bei der Verwendung dieser Buzzer, wie beispielsweise in dem SM_Thread, wird ebenfalls mehr Speicher benötigt, als bei der Verwendung der alten Buzzer. Zusätzlich benötigt der erstellte Treiber für die RealTimeClock Speicherplatz, um beim Einstellen einer Alarmzeit für einen RTC_Buzzer die alte Zeit zu speichern und so die bisher gewartete Zeit berechnen zu können.

erhöhter Verbrauch

Zur Bestimmung des tatsächlichen Bedarfs soll die Ursprungsversion von Kratos mit einer Version mit den RTC_Buzzern und mit einer Statemachine-Implementierung verglichen werden. Hierbei werden Versionen verwendet, die nur die entsprechend notwendigen Grundfunktionalitäten anbieten, um den Mehrbedarf, ausgehend von den zusätzlichen Komponenten, zu bestimmen. Die kompilierten Firmwareimages können dem Programm msp430-size übergeben werden. Dadurch wird die Belegung der .text-, .data- und .bss-Segmenten in

Auswertung der Segmente

Byte ausgegeben. Die Ausgabe wird in Abbildung 6.6 dargestellt.

```
> msp430-elf-size kratos.elf kratosRTCBuzzer.elf kratosStatemachine.elf
   text    data     bss     dec     hex filename
   2302     166     530    2998     bb6 kratos.elf
   3874     250     530    4654    122e kratosRTCBuzzer.elf
   4238     250     530    5018    139a kratosStatemachine.elf
> 
```

Abbildung 6.6.: Vergleich des Speicherverbrauch der Ursprungsversion von Kratos mit einer Version mit RTC_Buzzern und einer Statemachine-Implementierung.

Es ist zu erkennen, dass der Speicherverbrauch bei der Verwendung von RTC_Buzzer im Vergleich zur ursprünglichen Kratosversion deutlich gestiegen ist. Die Implementierung der neuen Buzzerart benötigt zusätzlich 1656 Byte Speicher, was einen Anstieg von ungefähr 55 Prozent entspricht. Die Grundfunktionalitäten für die Statemachine benötigen im Vergleich zur Buzzer-Implementierung 364 Byte. Dies ist ein Anstieg von ledig 7,8 Prozent. Es zeigt sich, dass bei der Implementierung der Buzzer Optimierungspotential besteht, zum Beispiel durch die Verwendung eines gemeinsamen Bellringers.

6.5. Generalisierbarbeit

Das Kapitel 5 und die Umsetzung der Beispielanwendung hat gezeigt, dass bei der Implementierung des entworfenen Modells und deren Anwendung keinerlei signifikante Schwierigkeiten aufgetreten sind. Daher wird davon ausgegangen, dass auch für weitere Anwendungen nach diesem Prinzip vorgegangen werden kann. Durch die Aufteilung einer Anwendung in mehrere einzelne Threads ist ein Entwurf von zustandsorientierten Service-Leveln intuitiv möglich. Die Implementierung der einzelnen Threads wird bis auf die Verwendung von adaptiven Synchronisationsobjekten nicht beeinflusst.

Konzept anwendbar

Neben der Definition der Systemvariablen und der Zustandsübergangstabelle ist vom Entwickler noch die Aktualisierung der Systemvariablen zu definierten. Da dies durch Polling möglich ist, können auch Systemkomponenten genutzt werden, die nicht zu Interrupts in der Lage sind. Für das Polling werden Treiberimplementierungen benötigt, die lediglich die Abfrage eines Wertes erlauben. Dies stellt somit keine besondere Einschränkung bei der Wahl der Systemvaria-

Interrupts nicht zwingend notwendig

blen dar. Da beim Wechsel des Service-Levels neben der Adaptierung der Buzzer noch weitere Aktionen angegeben werden können, ist ein vielseitiger Einsatz möglich.

Der zusätzliche Ressourcenaufwand durch die zusätzlichen Betriebssystemkomponenten beschränkt sich auf ein Minimum. Im Bereich des Energieverbrauchs ist der entstehende Overhead vor allem von den Funktionalitäten sehr gering (siehe Abschnitt 6.3.1 und 6.3.2). Der Mehrverbrauch an Speicher wird vor allem durch die Implementierung der adaptiven Buzzer verursacht. Der Speicherverbrauch für die Grundfunktionalitäten der Statemachine ist als gering anzusehen (vgl. Abschnitt 6.4).

Im Hinblick auf das zu Grunde liegende Betriebssystem ergeben sich für die Umsetzung eines Zustandsautomatens für die Steuerung von Service-Leveln nur bedingt Einschränkungen. Das Modell basiert lediglich auf der Verwendung von adaptiven Buzzern bei Implementierung der Threads.

Bei der Implementierung der Betriebssystemkomponenten wurden objektorientierte Mechanismen wie Vererbung verwendet. Um eine Definition von Events flexibel zu gestalten, wurden Templates verwendet. Für eine hohe Konfigurierbarkeit bei der Auswahl der Aktionen, bei einem Wechsel des Service-Levels, kann eine aspektorientierte Programmiersprache genutzt werden. Auf Kosten der Flexibilität und Konfigurierbarkeit ist grundsätzlich auch eine Implementierung ohne Templates und Aspekte möglich, beispielsweise durch festes Einprogrammieren im Quelltext.

aspektorientierte Programmierung

Insgesamt zeigt sich, dass sich das entwickelte Modell, unabhängig von der konkreten Implementierungsweise, eignet, um für Anwendungen Service-Levels zu definieren und ihre Ausführung zu steuern. Es kann auf Änderungen der Systemvariablen, die den Systemstatus repräsentieren, reagiert werden. So ist es beispielsweise möglich, auf geringe Energieniveaus reagieren zu können, um einen Ausfall des Systems zu verhindern.

6.6. Einschränkungen und Grenzen

Wie bereits im vorherigen Abschnitt erläutert, eignet sich das entworfene Konzept zur Nutzung von Service-Leveln innerhalb der Anwendung des eingebetteten Systems. Im folgenden Abschnitt soll kurz auf die sich ergebenden Einschränkungen dieses Konzepts eingegangen werden.

Eine große Einschränkung ist die notwendige Angabe einer Zustandsübergangstabelle. Der Entwickler muss konkrete Bedingungen für das Vorliegen eines Service-Levels benennen. Dies ist gerade bei Energieniveaus besonders schwierig, da ihre Bestimmung nicht trivial ist. Wahrscheinlich sind für ihre Angabe vorherige Simulationen notwendig, um abschätzen zu können, wie viel Energie in welchem Service-Level benötigt wird. Allgemein besteht ein direkter Zusammenhang zwischen der Güte der Bedingungen in der Zustandsübergangstabelle und der Adaptierbarkeit des Systems. Sind die Bedingungen für die Systemvariablen sinnvoll gewählt, kann das System die Ausführungsfrequenzen entsprechend adaptieren und beispielsweise so zusätzlich Energie nutzen. Sind allerdings die Bedingungen für die Systemvariablen schlecht formuliert, kann eine Anpassung des Systems das Gegenteil bewirken, indem ein höherer Service-Level geleistet wird, als es momentan möglich ist.

Angabe von Energieniveaus

Eine weitere Einschränkung entsteht durch das Polling von Systemwerten. Ebenso wie die Bedingungen der Systemvariablen muss die Ausführungsfrequenz des Pollings sinnvoll gewählt werden. Eine zu seltene Aktualisierung der Systemvariablen und Überprüfung der Übergangstabelle der Statemachine verhindert eine rechtzeitige Adaptierung des Systems. Im Gegenzug verursacht eine zu häufige Überprüfung einen zu hohen Overhead. Daher wäre ebenfalls für die Bestimmung der Ausführungsfrequenzen der Einsatz von Simulationen denkbar.

Frequenz entscheidend

7. Zusammenfassung und Fazit

Im Verlauf dieser Arbeit wurde für eingebettete Betriebssysteme ein Konzept entwickelt, welches es dem Entwickler ermöglicht, für eine Anwendung verschiedene Service-Level zu definieren und so die Ausführungsfrequenzen der einzelnen Threads zu verändern. Hierzu wurde das Prinzip der adaptiven Synchronisationsobjekte eingeführt. Ebenso wurde der entstehende Overhead durch die notwendigen Betriebssystemkomponenten untersucht.

Grundlagen

Vor dem Entwurf der notwendigen Betriebssystemkomponenten fand zunächst eine Einführung in die erforderlichen Grundlagen dieser Thematik statt. Diese bestanden aus den verwendeten Hardwareplattformen zur Entwicklung und Evaluation der Betriebssystemkomponenten sowie dem vorhandenen Betriebssystem Kratos. Weiterhin wurden in diesem Abschnitt Lösungsvorschläge für die Problemstellung in dem bisherigen Betriebssystem erläutert und aus den vorhandenen Schwächen Ansätze für Erweiterungen entwickelt.

Lösungsvorschläge in Kratos

verwandte Arbeiten

Im Anschluss wurden Forschungsarbeiten vorgestellt, deren Ansätze die Entwicklung der Betriebssystemkomponenten beeinflusst haben. Eine Abstraktion der Service-Level mittels Zustandsautomatens erwies sich dabei als geeignetes abstraktes Modell.

Im nachfolgenden Kapitel wurden zunächst die Ziele und Anforderungen an den Entwurf festgelegt. Anschließend wurden adaptive Synchronisationsobjekte eingeführt. Mit diesen wird es Threads ermöglicht, vorübergehend nicht mehr am Scheduling teilzunehmen. Das Besondere im Vergleich zu klassischen Synchronisationsobjekten ist, dass die Zeitspanne der Threads nachträglich verändert werden kann. Für eine Abstraktion der Service-Level wurde anschließend eine Statemachine eingeführt. In dieser können Service-Level anhand von

adaptive Synchronisationsobjekte

Statemachine

Systemvariablen definiert werden. Zusätzlich können für die Service-Levels entsprechende Ausführungsfrequenzen der Threads angegeben werden.

Evaluierung

Für eine Evaluation der neuen Betriebssystemkomponenten wurden diese für das Betriebssystem Kratos implementiert. Abschließend wurden diese Komponenten hinsichtlich des entstehenden Energieverbrauchs, des zusätzlichen Speicherverbrauchs, der Generalisierbarbeit sowie möglicher Einschränkungen und Grenzen bewertet.

7.1. Fazit

Konzept geeignet

Die Beispielanwendung in Kapitel 5.4 hat gezeigt, dass sich das vorgestellte System einer Statemachine für die Steuerung von Service-Level eignet. Bei dieser Anwendung wurde auf unterschiedliche Energieniveaus, repräsentiert durch die Sensorwerte eines Lichtsensors, mit einer Anpassung der Ausführungsfrequenzen eines Threads reagiert.

Optimierung möglich

Die Evaluation der erstellten Betriebssystemkomponenten hat gezeigt, dass eine grobe Abschätzung des Energiebedarfs aufgrund der Anzahl der Einträge der Zustandsübergangstabelle beziehungsweise der Anzahl der zu aktualisierenden Systemvariablen möglich ist. Im Idealfall findet eine Optimierung durch den Compiler statt, sodass der tatsächliche Energieverbrauch niedriger als der abgeschätzte Verbrauch ist. Bei den Aktualisierungen der Buzzer wurde ein deutlich höherer Verbrauch festgestellt, was auf die Verwendung von einzelnen Objekten als Buzzer zurückzuführen ist. Verglichen mit dem Energieverbrauch einer LED ist der verursachte Verbrauch durch die Aktualisierungen dennoch als gering einzustufen.

Alle vorgestellten Komponenten wurden in das Betriebssystem Kratos eingebunden und nutzen teilweise AspectC++ für eine bequeme Konfigurierbarkeit. Grundsätzlich ist das Modell jedoch weder an Kratos, noch an AspectC++ gebunden. Somit ist ein Einsatz auch in anderen eingebetteten Betriebssystemen möglich.

7.2. Ausblick

Durch diese Arbeit wurde die Grundlage für ein reaktives System geschaffen, mit welchem für eine Anwendung unterschiedliche Service-Level definiert werden können. Dies bildet die Grundlage für ein proaktives System, bei welchem eine Angabe konkreter Bedingungen für ein Service-Level durch den Entwickler nicht mehr notwendig ist. Bei einem proaktiven System sollen aufgrund vorheriger Werte Abschätzungen für ein zukünftig mögliches Service-Level getroffen werden.

proaktives System

Darüber hinaus bieten die adaptiven Buzzer weitere Möglichkeiten der Integration in das Betriebssystem an. Die bisherige Trennung zwischen RTC_Buzzer und den bisherigen Buzzern könnte abgeschafft werden. Damit wäre Optimierung des Energieverbrauchs des Systems möglich, da Unterbrechungen der Timer bei langen Wartezeiten vermieden werden können.

A. Statemachine in Kratos

In dem folgenden Abschnitt soll die Integration der Statemachine in Kratos erläutert werden. Hierzu wird zuerst auf die Einbettung der Dateien in die Ordnerstruktur des Quellcodeverzeichnisses eingegangen. Anschließend erfolgt eine kurze Beschreibung der anpassbaren Programmstellen.

A.1. Ordnerstruktur

Die Quelltextdateien der bisher vorhandenen Synchronisationsobjekte und ihren Verwaltungskomponenten wurden in dem Verzeichnis src\meeting gespeichert. Die neu erstellten adaptiven Synchronisationsobjekte wurden daher im selben Verzeichnis erstellt. Hierzu gehören folgende Dateien:

- basic_times.h
- buzzer_times.h
- buzzer_times.cc
- rtc_bellringer.h
- rtc_bellringer.cc
- rtc_buzzer.h
- rtc_buzzer.cc

Analog zu den bisherigen Buzzern wurden Guarded_RTCBuzzer erstellt. Die von diesem Buzzer angebotenen Methoden werden direkt auf die Methoden der Basisklasse RTC_Buzzer abgebildet, nur dass ihre Ausführung jeweils mit Hilfe eines Objekts der Klasse Secure geschützt wird.

Die Integration folgender Dateien erfolgt im Verzeichnis src\syscall:

- guarded_rtcbuzzer.h
- guarded_rtcbuzzer.cc

Die Quelltextdateien für den Peripherietreiber der RealTimeClock wurden, wie alle anderen Clocks des MSP430, in dem Verzeichnis src\arch\msp430fr\clocks erstellt. Hierzu gehören folgende Dateien:

- rtcb.ah
- rtcb.h
- rtcb.cc
- rtcb.feature

Für die Quellcodedateien der Statemachine wurde ein separater Ordner statemachine im Verzeichnis src\arch\msp430fr\erstellt. In diesem befinden sich alle Dateien die für die Statemachine nutzen notwendig sind. Hierzu gehören mindestens die Dateien:

- basic_sm.h
- SM_Thread.h
- SM_Thread.cc
- statemachine.h
- statemachine.cc
- statemachine.feature

Es ist sinnvoll in diesem Verzeichnis auch alle weiteren Dateien, beispielsweise Aspekte die in die update()-Methode des SM_Threads eingewoben werden sollen, gespeichert werden. Dies ermöglicht eine eindeutige Trennung zwischen den Dateien der Statemachine und dem restlichen System.

A.2. Benutzung der Komponenten

Nachdem alle Dateien entsprechend in das Quellcodeverzeichnis eingebunden wurde, können die Komponenten über die graphische Konfigurierungsoberfläche ausgewählt werden. Die Dabei rtcb.feature bindet dabei die RealTimeClock, den RTC_Bellringer sowie die entsprechenden Buzzer in das System ein. Eine Einbindung der Komponenten der Statemachine erfolgt über die .feature-Dabei statemachine.feature.

A.2.1. RTC_Buzzer

Die Benutzung der RTC_Buzzer unterscheidet sich kaum von der Nutzung der normalen Buzzer. In folgenden Quellcode ist der LED_Thread der Beispielsanwendung dargestellt. Es ist notwendig, dass die Header-Dateien des Buzzers (Zeile 5) und der Buzzerzeiten (Zeile 6) inkludiert werden. Letzte Datei dient der Verwendung von definierten Identifier bei den Buzzern (Zeile 29).

Die Definition der Identifier erfolgt in den Dateien buzzer_times.cc und buzzer_times.h. In der Header-Datei werden die genutzten Identifier mit einem Makro eingeführt (Zeile 6). Dabei kann ein Typ des Identifier angegeben werden.
Eine Angabe der entsprechenden Zeiten für die Identifier erfolgt in der Quellcode-Datei. Dabei ist eine einfache Auflistung mittels Präprozessormakros möglich (vgl. Zeile 9 - 14). Die Markos in den Zeilen 8 und 15 erstellen dabei den Methodenrumpf der entsprechenden Klasse.

```
1  #include <msp430.h>
2  #include "LED_Thread.h"
3
4  // Verwendung der RTC_Buzzer und Identifier
5  #include "syscall/guarded_rtcbuzzer.h"
6  #include "meeting/buzzer_times.h"
7
8  #define DeclareThreadWithSize(classname,instancename,
       stacksize) \
9    static char instancename##_stack[stacksize]
         __attribute__((aligned(0x02))); \
10   classname instancename(instancename##_stack +
         stacksize, stacksize);
11
12 /* 512 Byte Stack */
13 DeclareThreadWithSize(LED_Thread,ledThread,512);
14
15 // Testen mit LED
16 void LED_Thread::action(){
17  Guarded_RTCBuzzer buz;
18  //LED aus
19  P4DIR |= BIT6;
20  P4OUT |= BIT6;
21  while(1){
22   // LED an
23   P4DIR ^= BIT6;
24   P4OUT ^= BIT6;
25   buz.sleep(1);
26   //LED aus
27   P4DIR |= BIT6;
28   P4OUT |= BIT6;
29   buz.sleep(LED_THREAD);
30  }
31 }
```

Quelltext A.1: Quelltext des LED_Threads.

```
1  #ifndef Buzzer_Times_H
2  #define Buzzer_Times_H
3
4  #include "basic_times.h"
5
6  DEF_IDENT( Ident, ident, LED_THREAD, SMTHREAD );
7
8  class Statemachine;
9
10 class Buzzer_Times : public basic_times<Ident>
11 {
12  private:
13  Buzzer_Times(const Buzzer_Times& copy);
14  Statemachine *d_sm;
15
16  public:
17  Buzzer_Times(){};
18  int getTime(Ident id);
19  void setSM(Statemachine* sm);
20 };
21
22 extern Buzzer_Times buzzer_times;
23
24 #endif
```

Quelltext A.2: Header-Datei buzzer_times.h.

```
1  #include "basic_times.h"
2
3  #include "arch/msp430fr/statemachine/statemachine.h"
4
5
6  Buzzer_Times buzzer_times;
7
8  TIME_TAB
9   ID(LED_THREAD, 1, 300)
10  ID(LED_THREAD, 2, 30)
11  ID(LED_THREAD, 3, 3)
12  ID(SMTHREAD, 1, 30)
13  ID(SMTHREAD, 2, 3)
14  ID(SMTHREAD, 3, 3)
15 END_TIME_TAB
16
17 void Buzzer_Times::setSM(Statemachine *sm){
18  d_sm = sm;
19 }
```

Quelltext A.3: Quelltext-Datei buzzer_times.cc

A.2.2. Statemachine

Für eine Benutzung der Statemachine müssen vor allem die Dateien statemachine.h und statemachine.cc angepasst werden. In der Header-Datei können, analog zu den Identifiern, Events definiert werden (Zeile 9). Diese können einen Teil der Bedingungen für die Zustandsübergangstabelle sein. Weiterhin muss der Entwickler die genutzten Systemvariablen in einem struct definieren (Zeile 11 - 14). Diese werden im weiteren Verlauf von dem SM_Thread aktualisiert.
Die Angabe der Zustandsübergangstabelle erfolgt in der Datei statemachine.cc. Hierbei stehen verschiedene Makros zu Verfügung, je nachdem ob der aktuelle Service-Level Bestandteil der Bedingung sein soll. In den Zeilen 7 bis 13 sind Beispiele angegeben. Wird keine Variablenbelegung oder Event bei den Bedingungen verwendet, können die entsprechenden Parameter mit true beziehungsweise NO_EVENT gefüllt werden.
Um zusätzlich zu den Aktualisierungen der Buzzer noch weitere Aktionen ausführen zu können, kann mit den Makros in den Zeilen 19 und 20 auf weitere Methoden verwiesen werden. Ebenfalls wäre die Angabe einer konkreten Aktion (Zeile 18) möglich.
Die Initialisierungsfunktion der Statemachine startet den SM_Thread. Sind weitere Aktionen zu Beginn gewünscht, können diese im Aspekt startStatemachine.ah ergänzt werden.
Eine Anpassung des SM_Threads ist in der Regel nicht erforderlich. Für die Aktualisierung von Systemvariablen können Aspekte genutzt werden. Ein entsprechendes Beispiel wurde im Quellcode 5.11 angegeben.

```
#ifndef STATEMACHINE_H
#define STATEMACHINE_H
#include "guard/gate.h"
#include <msp430.h>
#include "basic_sm.h"
//Fuer Anwendungen (Threads)
#include "meeting/rtc_bellringer.h"
// Definition der vorhandenen Events
DEF_EVENT( Event, TestEvent );
// Variablen die den Systemzustand definieren
typedef struct {
 // Ergaenzen, je nach Anwendungen
 int energy1;
} state;

class Statemachine : public SM<Event>
{
private:
 // aktueller Service-Level
 int level;
 // Methoden fuer Service-Level
 void level1();
 void level2();
 void level3();

public:
 // Sytemvariablen
 state systemstate;
 // Konstruktor
 Statemachine() {level = 1; startStatemachine();}
 // get-Methode fuer Service-Level
 int getSL(){return level;}
 // Ueberpruefen des Service-Levels
 void checkStateWithEvent(Event eEvent);
 void checkState() {checkStateWithEvent(nEvent);}
 // Startfunktion der SM
 void startStatemachine();
 // Aufruf der Service-Level entsprechenden Methoden
 void actionOfState();
};
extern Statemachine sm;
#endif
```

Quelltext A.4: Header-Datei statemachine.h.

```
#include "statemachine.h"

extern RTC_Bellringer rtc_bellringer;
Statemachine sm;

// Definition der Zustanduebergangstabelle
SL_CON_TAB
 // Level, Bedingung, Event, Folgelevel
 CON(2, energy1 > 700, NO_EVENT, 3);
 CON(2, energy1 < 600, NO_EVENT, 1);
 CON_AS(true, TestEvent, 1);
 CON_AS(energy1 > 700, TestEvent, 3);
END_TAB


// Aufrufe fuer die entsprechenden Funktionen der
   Service-Level
SL_TO_ACTION
 FUNC_OF_STATE(1, level=1)
 FUNC_OF_STATE(2, level1())
 FUNC_OF_STATE(3, level2())
END_ACTION

void Statemachine::level1()
{
  // Wird ueber Aspekte mit Inhalt gefuellt
}
void Statemachine::level2()
{
 // Wird ueber Aspekte mit Inhalt gefuellt
}
void Statemachine::level3()
{
 // Wird ueber Aspekte mit Inhalt gefuellt
}

// Startfunktion der Statemachine
void Statemachine::startStatemachine()
{
 // Wird ueber Aspekte mit Inhalt gefuellt
}
```

Quelltext A.5: Quellcode-Datei statemachine.cc.

B. Messergebnisse

Im Zuge der Evaluation wurden die Energieverbräuche abzüglich des Verbrauchs des Systems im idle-Modus verwendet. Da der momentane idle-Verbrauch im Vergleich zu dem theoretisch möglichen Verbrauch sehr hoch ist, werden im Folgenden die gemessenen Energieverbräuche ohne eine Verrechnung mit dem idle-Verbrauch angegeben.

Dauer[ms]	1478	1483	1492	1488
Energieverbrauch[mJ]	1,148	1,153	1,158	1,154

Tabelle B.1.: Auflistung der gemessenen Verbrauchswerte für die Überprüfung der Zustandsübergangstabelle mit 4 Einträgen.

Dauer[ms]	1702	1713	1724	1707
Energieverbrauch[mJ]	1,489	1,498	1,504	1,494

Tabelle B.2.: Auflistung der gemessenen Verbrauchswerte für die Überprüfung der Zustandsübergangstabelle mit 6 Einträgen.

Dauer[ms]	1807	1831	1811	1823
Energieverbrauch[mJ]	2,051	2,071	2,059	2,062

Tabelle B.3.: Auflistung der gemessenen Verbrauchswerte für die Überprüfung der Zustandsübergangstabelle mit 8 Einträgen.

Dauer[ms]	2366	2342	2354	2361
Energieverbrauch[mJ]	2,661	2,642	2,654	2,659

Tabelle B.4.: Auflistung der gemessenen Verbrauchswerte für die Überprüfung der Zustandsübergangstabelle mit 10 Einträgen.

Dauer[ms]	2416	2425	2406	2410
Energieverbrauch[mJ]	1,74	1,742	1,735	1,737

Tabelle B.5.: Auflistung der gemessenen Verbrauchswerte für das Polling von 3 Systemvariablen.

Dauer[ms]	4504	4528	4485	4511
Energieverbrauch[mJ]	3,201	3,231	3,219	3,221

Tabelle B.6.: Auflistung der gemessenen Verbrauchswerte für das Polling von 6 Systemvariablen.

Dauer[ms]	6233	6227	6231	6220
Energieverbrauch[mJ]	4,458	4,461	4,463	4,455

Tabelle B.7.: Auflistung der gemessenen Verbrauchswerte für das Polling von 9 Systemvariablen.

Dauer[ms]	13804	13785
Energieverbrauch[mJ]	13,03	12,98

Tabelle B.8.: Auflistung der gemessenen Verbrauchswerte für die Aktualisierung von einem Buzzer.

Dauer[ms]	17122	16992
Energieverbrauch[mJ]	16,19	16,07

Tabelle B.9.: Auflistung der gemessenen Verbrauchswerte für die Aktualisierung von 2 Buzzern.

Dauer[ms]	19351	19256
Energieverbrauch[mJ]	18,645	18,515

Tabelle B.10.: Auflistung der gemessenen Verbrauchswerte für die Aktualisierung von 3 Buzzern.

Dauer[ms]	21580	21520
Energieverbrauch[mJ]	21,10	20,96

Tabelle B.11.: Auflistung der gemessenen Verbrauchswerte für die Aktualisierung von 4 Buzzern.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Burns, B.; Grimaldi, K.; Kostadinov, A.; Berger, E. D.; Corner, M. D.: *Flux: A language for programming high-performance servers.* 2006

[2] Buschhoff, M. and Lochmann, A.: *PG Solar Doorplate, Kurzvorstellung.* 2015. – URL `https://ess.cs.tu-dortmund.de/Teaching/PGs/solardoorplate/downloads.html`. – Zugriffsdatum: 13.01.2016

[3] Falkenberg, Robert: *Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme*, TU Dortmund, Diplomarbeit, 2014. – URL `https://eldorado.tu-dortmund.de/handle/2003/34377`

[4] Flinn, J.; Satyanarayanan, M.: *Managing Battery Lifetime with Energy-aware Adaptation.* 2004

[5] Fraunhofer-Institut für Materialfluss und Logistik IML: *InBin - Der intelligente Behälter.* – URL `http://www.iml.fraunhofer.de/de/themengebiete/automation_eingebettete_systeme/Produkte/IntelligenterBehaelter.html`. – Zugriffsdatum: 13.01.2016

[6] Gelernter, D.; Carriero, N.: *Coordination languages and their significance.* ACM, 1992

[7] Hu, S.; Yu, Y.; Xie, L.: *Comparing Power Management Strategies of Android and TinyOS.* 2011

[8] Jurack, S.: *Aspektorientierte Programmierung: AspectJ, AJDT.* 2006. – URL `https://swt.cs.tu-berlin.de/lehre/seminar/ss06/Jurack.pdf`. – Zugriffsdatum: 28.01.2016

[9] Li, S.; Sutton, R.; Rabaey, J.: *Low Power Operating System for Heterogeneous Wireless Communication System.* 2003

[10] Lin, K.; Hsu, J.; Zahedi, S.; Lee, D. C.; Friedman, J.; Kansal, A.; Raghunathan, V.; Srivastava, M. B.: *Heliomote: Enabling long-lived sensor networks through solar energy harvesting.* 2005

[11] Meninger, S.; Mur-Miranda, J. O.; Amirtharajah, R.; Chandrakasan, A. P.; Lang, J. H.: *Vibration-to-Electric Energy Conversion.* 2001

[12] Moser, C.; Chen, J.; Thiele, L.: *Reward Maximization for Embedded Systems with Renewable Energies.* 2008

[13] Moser, C.; Chen, J.; Thiele, L.: *Optimal Service Level Allocation in Environmentally Powered Embedded Systems.* 2009

[14] Moser, C.; Chen, J.; Thiele, L.: *Power Management in Energy Harvesting Embedded Systems with Discrete Service Levels.* 2009

[15] Moser, C.; Chen, J.; Thiele, L.: *Dynamic power management in environmentally powered systems.* 2010

[16] Moser, C.; Thiele, L.; Brunelli, D.; Benini, L.: *Adaptive Power Management in Energy Harvesting Systems.* 2007

[17] Perla, E.; Cathain, A.; Carbajo, R. S.; Huggard, M.; Goldrick, C.: *PowerTOSSIM z: Realistic Energy Modelling for Wireless Sensor Network Environments.* 2008

[18] Priya, S.; Chen, C.; Fye, D.; Zahnd, J.: *Piezoelectric Windmill: A Novel Solution to Remote Sensing.* 2005

[19] Roundy, S.; Steingart, D.; Frechette, L.; Wright, P. K.; Rabaey, J. M.: *Power sources for wireless sensor networks.* 2004

[20] Seifert, C. Instruments: *Automatentheorie.* – URL `http://homepage.ruhr-uni-bochum.de/christian.seifert-2/Automaten.pdf`. – Zugriffsdatum: 20.01.2016

[21] Sorber, J.; Kostadinov, A.; Garber, M.; Brennan, M.; Corner, M.; Berger, E.: *Eon: A Language and Runtime System for Perpetual Systems*. 2007

[22] Spinczyk, O.: *Vorlesung Betriebssystembau*. 2015. – (Vorlesung 1: Einführung)

[23] Texas Instruments: *MSP430 FRAM Quality and Reliability*. – URL `http://www.ti.com/lit/an/slaa526a/slaa526a.pdf`. – Zugriffsdatum: 13.01.2016. – (Rev. A)

[24] Texas Instruments: *MSP430FR5969 LaunchPad Evaluation Kit*. – URL `http://www.ti.com/ww/en/launchpad/launchpads-msp430-msp-exp430fr5969.html#tabs`. – Zugriffsdatum: 13.01.2016

[25] Texas Instruments: *MSP430FR59xx Mixed-Signal Microcontrollers*. – URL `http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf`. – Zugriffsdatum: 13.01.2016

[26] Texas Instruments: *MSP430FR59xx MSP430FR58xx Code Examples*. – URL `http://www.ti.com/tool/msp-exp430fr5969`. – Zugriffsdatum: 28.01.2016

[27] TU Dortmund, Fakultät Informatik, Lehrstuhl 12: *PG 595: Solar Doorplate*. – URL `https://ess.cs.tu-dortmund.de/Teaching/PGs/solardoorplate/index.html`. – Zugriffsdatum: 13.01.2016

[28] Urban, M. and Spinczyk,O.: *Documentation: AspectC++ Language Reference*. 2012. – URL `http://www.aspectc.org/doc/ac-languageref.pdf`. – Zugriffsdatum: 28.01.2016

[29] Zeng, H.; Ellis, C. S.; Lebeck, A. R.; Vahdat, A.: *ECOSystem: Managing Energy As a First Class Operating System Resource*. 2002