technische universität
dortmund

Masterarbeit

Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme

ROBERT FALKENBERG
(GEB. ROBERT RAPCZYNSKI)

Dezember 2014

GUTACHTER:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Markus Buschhoff

Technische Universität Dortmund
Fakultät für Informatik
Eingebettete Systemsoftware (LS-12)
http://ls12-www.cs.tu-dortmund.de

ENTWURF EINES ENERGIEGEWAHREN TREIBERMODELLS FÜR EINGEBETTETE BETRIEBSSYSTEME

ROBERT FALKENBERG (GEB. ROBERT RAPCZYNSKI)

Masterarbeit

Eingebettete Systemsoftware (LS-12)
Fakultät für Informatik
Technische Universität Dortmund

Dezember 2014

Robert Falkenberg (geb. Robert Rapczynski):
Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme,
Masterarbeit, Dezember 2014

GUTACHTER:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Markus Buschhoff

ZUSAMMENFASSUNG

In eingebetteten Betriebssystemen ist die Kenntnis der genauen (momentanen) Leistungsaufnahme einzelner Peripheriekomponenten zumeißt unbekannt, sodass diese Informationen nicht in Verhaltensentscheidungen zur Laufzeit einfließen. Zudem fehlt für die Analyse der Systemsoftware hinsichtlich ihres Energieverhaltens ein Zusammenhang zwischen Peripheriezugriffen und dem damit verbundenem Verbrauch. Um diese Lücke zu schließen, wird in dieser Arbeit die Treiberschicht eines eingebetteten Betriebssystems dahingehend umgestaltet, dass der Energieverbrauch von Peripheriegeräten in jedem Betriebszustand abgefragt werden kann und Treiberzugriffe mit entsprechenden Kosten annotiert werden. Hierzu wird das Modell der **P**riced **T**imed **A**utomata (PTA) herangezogen und vollständig in die Treiberimplementierung integriert. Der Ansatz wird anhand von representativen Komponenten hinsichtlich seiner Exaktheit und seines Ressourcenverbrauchs evaluiert, sowie eine Methodik zur vereinfachten Treiberentwicklung nach diesem Modell vorgestellt.

DANKSAGUNG

Mit dieser Arbeit schließe ich mein Masterstudium in angewandter Informatik an der TU Dortmund ab. Diese Gelegenheit möchte ich dazu nutzen, allen Personen zu danken, die mich bei meinem Studium und der Abschlussarbeit begleitet und unterstützt haben. Besonders möchte ich mich für die Unterstützung durch meine Eltern bedanken, die mir ein sorgloses Studieren ermöglicht haben.

Außerdem möchte ich mich bei meinen Betreuern Prof. Dr.-Ing. Olaf Spinczyk und Dipl.-Inf. Markus Buschhoff für die Ermöglichung einer anspruchsvollen und interessanten Abschlussarbeit, die meine zwei Studienschwerpunkte (Informatik und Elektrotechnik) in einem Thema vereint, bedanken.

Abschließend möchte ich ganz besonders meiner Ehefrau Suzana den höchsten Dank dafür aussprechen, dass sie mich während des gesamten Studiums, sowie in der Vorbereitungs- und Bearbeitungszeit dieser Masterarbeit liebe- und verständnisvoll unterstützt hat.

INHALTSVERZEICHNIS

ABKÜRZUNGEN

I^2C	Synchrone integrierte Peripherieschnittstelle, engl. ***I**nter-**I**ntegrated **C**ircuit*.
ACLK	Engl. ***A**uxiliary **Cl**ock*.
AOStuBS	**A**spekt**o**rientiertes **Stu**denten **B**etrieb**s**system. ☞ Glossar.
API	Engl. ***A**pplication **P**rogramming **I**nterface*.
BMA	Bool'scher Mealy Automat, engl. ***B**oolean **M**ealy **A**utomaton*.
CMOS	Engl. ***C**omplementary **m**etal-**o**xide-**s**emiconductor*.
CPU	Hauptprozessor, engl. ***C**entral **P**rocessing **U**nit*.
CS	Chip-Auswahlleitung, engl. ***C**hip **S**elect*.
DCO	Interner digital einstellbarer Oszillator, engl. ***D**igitally **C**ontrolled **O**scillator*.
DFA	Deterministischer endlicher Automat, engl. ***D**eterministic **F**inite **A**utomaton*.
DMA	Speicherdirektzugriff, engl. ***D**irect **M**emory **A**ccess*.
DRAM	Dynamisches RAM, engl. ***D**ynamic **R**andom **A**ccess **M**emory*.
DUT	Prüflast, engl. ***D**evice **U**nder **T**est*.
E-PAPER	Elektronisches Papier, engl. ***E**lectronic **paper***.
EEPROM	Nichtflüchtiger Speicher, engl. ***E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory*.
eUSCI	Engl. ***E**nhanced **U**niversal **S**erial **C**ommunication **I**nterface*.
FRAM	Ferroelektrisches RAM, engl. ***F**erroelectric **R**andom **A**ccess **M**emory*.
FSM	Endlicher Automat, engl. ***F**inite **S**tate **M**achine*.
GLTS	Engl. ***G**uarded **l**abeled **t**ransition **s**ystem*.
HFXT	Externer Hochfrequenz-Oszillator, engl. ***H**igh **F**requency **Crys**tal*.
ISR	Interruptbehandlungsroutine, engl. ***I**nterrupt **S**ervice **R**outine*.
LCD	Flüssigkristallanzeige, engl. ***L**iquid **C**rystal **D**isplay*.
LED	Leuchtdiode, engl. ***L**ight-**E**mitting **D**iode*.
LFXT	Externer Niederfrequenz-Oszillator mit besonders geringem Energiverbrauch, engl. ***L**ow **F**requency **Crys**tal*.
MCLK	Engl. ***M**aster **Cl**ock*.

MCU	Mikrocontroller, engl. ***M**icro**c**ontroller **U**nit*.
MIMOSA	**M**essgerät zur **i**ntegrativen **M**essung **o**hne **S**pannungs-**a**bfall.
MISO	Master Datenausgangsleitung bei SPI, engl. ***M**aster **I**n **S**lave **O**ut*.
MODOSC	Interner Modul-Oszillator, engl. ***Mod**ule **Osc**illator*.
MOSFET	Feldeffekttransistor (mit isoliertem Gate), engl. ***m**etal-**o**xide-**s**emiconductor **f**ield-**e**ffect **t**ransistor*.
MOSI	Master Dateneingangsleitung bei SPI, engl. ***M**aster **O**ut **S**lave **I**n*.
NMI	Nicht maskierbare Unterbrechung, engl. ***N**on **M**askable **I**nterrupt*.
OMNeT++	C++ Simulationsframework für Netzwerkprotokolle o. ä., engl. ***O**bjective **M**odular **Net**work **T**estbed in C**++**.*
PC	Programmzähler, engl. ***P**rogram **C**ounter*.
PTA	Engl. ***P**riced **T**imed **A**utomaton*.
PWM	Pulsweitenmodulation, engl. ***P**ulse **W**idth **M**odulation*.
RAM	Speicher mit wahlfreiem Zugriff, engl. ***R**andom **A**ccess **M**emory*.
Remes	Engl. ***RE**source **M**odel for **E**mbedded **S**ystems.* ☞ *Glossar*.
RISC	Rechner mit reduziertem Befehlssatz, engl. ***R**educed **I**nstruction **S**et **C**omputer*.
ROM	Festwertspeicher, engl. ***R**ead-**O**nly **M**emory*.
RSSI	Indikator für die Signalstärke, engl. ***R**eceived **S**ignal **S**trength **I**ndication*.
RTC	Echtzeituhr, engl. ***R**eal **T**ime **C**lock*.
SCK	Taktleitung bei SPI, engl. ***S**erial **Cock**.*
SMCLK	Engl. ***S**ubsystem **M**aster **Cl**ock**k**.*
SP	Stapelzeigerregister, engl. ***S**tack **P**ointer*.
SPI	Synchrone integrierte Peripherieschnittstelle, engl. ***S**erial **P**eripheral **I**nterface*.
SR	**S**tatus **R**egister.
SRAM	Statisches RAM, engl. ***S**tatic **R**andom **A**ccess **M**emory*.
UART	Asynchrone integrierte Peripherieschnittstelle, engl. ***U**niversal **A**synchronous **R**eceiver **T**ransmitter*.
ULP	Mikrocontrollerserie von Texas Instruments mit sehr geringem Energieverbrauch, engl. ***U**ltra-**L**ow-**P**ower*.
UML	Einheitliche Modellierungssprache, engl. ***U**nified **M**odeling **L**anguage*.
Uppaal Cora	Engl. ***Upp**sala University and **Aal**borg University **C**ost **O**ptimal **R**eachability **A**nalysis.* ☞ *Glossar*.

VHDL	Hardwarebeschreibungssprache, engl. ***V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage*.
VLO	Interner Niederfrequenz-Oszillator mit besonders geringem Energiverbrauch, engl. ***V**ery-Low-Power **L**ow-Frequency **O**scillator*.
VTABLE	Tabelle virtueller Methoden, engl. *virtual method **table***.
XML	Engl. *E**x**tensible **M**arkup **L**anguage*.

active low	Beschreibung der Polarität eines digitalen Signaleingangs, dass seine namentliche Funktion bei einem 0-Pegel aktiviert und bei einem 1-Pegel deaktiviert wird. Dem entgegen bedeutet *active high* das Gegenteil.
AOStuBS	**A**spekt**o**rientiertes **Stu**denten **B**etrieb**s**system, welches für Forschungs- und Lehrzwecke am Lehrstuhl für Eingebettete Systeme [Tec14b] der TU-Dortmund verwendet wird.
AspectC++	Erweiterung der Programmiersprache C++ um Aspekte [Spi14]. Dadurch ist es möglich, querschneidende Belange in eingenständigen Modulen (Aspekten) zu kapseln, die beim Übersetzen automatisch an die gewünschten Stellen anderer Klassen und Methoden eingewoben werden. Siehe Kapitel 2.5.1.
C	Hardwarenahe Programmiersprache, die vor allem zur Systemprogrammierung verwendet wird.
C++	Objektorientierte Programmiersprache, die sich aus der Programmiersprache C entwickelt hat..
Energy harvesting	„Ernten“ von Energie aus der Umgebung. Beispiele hierzu sind Solarzellen zur Energiegewinnung aus Licht oder piezoelektrische Generatoren zur Spannungserzeugung durch Druck.
GNU Octave	Interpretierte Skriptsprache, besonders geeignet zur Lösung mathematischer Probleme, die größtenteils zum proprietären MATLAB kompatibel ist [Joh14, The14].
Hamming-Distanz	Anzahl der Bits, die sich beim Vergleich zweier Bitfolgen gleicher Länge unterscheiden. Mit anderen Worten: Anzahl der 1-Bits, die sich nach XOR-Verknüpfung zweier Bitfolgen ergibt.
InBin	Prototyp einer energiesparenden Baueinheit, entwickelt vom Fraunhofer IML in Dortmund, zur Anbringung an Transportkisten in einem automatisierten Lagersystem. Siehe Kapitel 2.4.2.
kconfig	Konfigurationssprache zur Auflistung und Formulierung von Features für den Linux Kernel. Die Features von Kratos werden in diese Sprache übersetzt, um eine einfache graphische Konfiguration des Systems mit vorhandenen Werkzeugen wie kconfig-qconf zu ermöglichen.

kconfig-qconf	Programm zur graphischen Konfiguration von kconfig-Dateien. Im Kontext dieser Arbeit wird es als Konfigurationsoberfläche für Kratos verwendet. Siehe Kapitel 2.5.4.
Kratos	Featureorientierte Erweiterung von AOStuBS für Forschungszwecke am Lehrstuhl für Eingebettete Systeme [Tec14b] der TU-Dortmund. Siehe Kapitel 2.5.
Ladungspumpe	Schaltung zum Erzeugen einer höheren Spannung aus einer gegebenen Spannung. Beispielsweise können N Kondensatoren in einer Parallelschaltung zunächst mit der Betriebsspannung aufgeladen werden, um anschließend in einer Reihenschaltung eine N-Fache Spannung anzubieten.
Launchpad	Evaluationsboard MSP-EXP430FR5969 von Texas Instruments. Siehe Kapitel 2.4.3.
Python	Programmiersprache, die üblicherweise nicht in Maschinencode übersetzt, sondern interpretiert wird.
Remes	Verhaltensbeschreibungssprache für eingebettete Systeme mit besonderer Berücksichtigung des Ressourcenverbrauchs (**RE**source **M**odel for **E**mbedded **S**ystems (Remes)) [SVP09]. Siehe Kapitel 3.2.
Shunt	Niederohmiger Widerstand zur Messung eines Stromflusses durch ein DUT. Siehe Kapitel 2.3.1.
TinyDB	Verteiltes Datenbanksystem für Sensornetzwerke. Siehe [MFHH05].
Uppaal Cora	Werkzeug zur kostenoptimalen Erreichbarkeitsanalyse in PTA. (**Upp**sala University and **Aal**borg University **C**ost **O**ptimal **R**eachability **A**nalysis (Uppaal Cora)). Siehe [Beh14].
Watchdog	Zu Deutsch „Wachhund“. Überwachungsschaltung innerhalb eines Mikrocontrollers, realisiert durch einen getakteten Zähler, der nach Ablauf einer zuvor definierten Zeitspanne einen kontrollierten Neustart (Reset) auslöst. Die Software muss durch regelmäßiges und rechtzeitiges Zurücksetzen des Zählers ihre korrekte Funktionalität bestätigen. Siehe Kapitel 2.4.1.8.

Teil I

WISSENSCHAFTLICHER TEIL

Dieser Teil beinhaltet den wissenschaftlichen Inhalt der Abschlussarbeit. Dokumentationen und größere Bilder sind im zweiten Teil gekapselt.

1 EINLEITUNG

In eingebetteten Systemen ist die zur Verfügung stehende Energie häufig ein limitierender Faktor für ihre Dimensionierung und ihren Funktionsumfang. Sie müssen mit der geringen Energie durch Energy harvesting (z. B. aus einer Solarzelle) arbeiten, oder beziehen ihre Versorgungsspannung aus einer Batterie, deren Austausch beispielsweise in Sensornetzen zu aufwendig wäre.

Insbesonders in verteilten Sensornetzen kann die Kenntnis des aktuellen Energiestatus dazu genutzt werden, energieeffiziente Pfade für den Datenaustausch zu finden und dadurch die Lebenszeit einzelner Knoten (und damit auch die Betriebszeit des gesamten Netzwerks) zu maximieren. Bei beweglichen Sensorknoten kann es zudem sinnvoll sein, die Messdaten zunächst zwischenzuspeichern, um sie erst bei einer günstigeren Funkverbindung unter Aufwendung einer geringeren Sendeleistung zu verschicken.

Betriebszeit maximieren

Diese Szenarien erfordern es, dass die Software des eingebetteten Systems für jede seiner Peripheriekomponenten Kenntnis über den Energieverbrauch des aktuellen Betriebszustands, sowie aller weiteren möglichen Modi besitzt. Obwohl der Betriebszustand durch die Aktivität der ausgeführten Software beeinflusst wird, lässt sich der resultierende Energieverbrauch nicht – oder nur mit Interpretationsaufwand – aus dem Programm ablesen. Dies gilt insbesonders, wenn die Peripheriegeräte ihren Betriebszustand ereignisbasiert verändern. Selbst wenn Ereignisse vorhersehbar sind (z. B. Abschluss einer ausgehenden Datenübertragung mit bekannter Länge), wird dieses Wissen nicht unmittelbar in das System implementiert, sondern findet nur bei der Gestaltung einer hinreichend schnellen Ereignisbehandlung Anwendung.

Quelltext: Energieverbrauch unbekannt

Hardwareseitig lässt sich der aktuelle Energieverbrauch durch die Integration von Messtechnik erfassen. Dieser Ansatz ist jedoch häufig nicht gewünscht, da er die Hardwarekosten erhöht und mit einem zusätzlichen Energieverbrauch des Systems zum Betrieb dieser Messkomponenten einhergeht. Ist die Auskunft der komponentenweisen Verbrauchsdaten erforderlich, vergrößert sich der erforderliche Schaltungsaufwand noch weiter.

externe Messtechnik

Zudem geht bei der Verwendung von integrierter Messtechnik der Energieverbrauch des Geräts weiterhin nicht aus dem Quelltext hervor. Eine statische Analyse (zur Übersetzungszeit des Systemsoftware) hinsichtlich des erwarteten Energieverbrauchs ist damit nicht realisierbar, da die Verbrauchswerte erst im laufenden System verfügbar sind.

rein softwarebasiert

Daher soll in dieser Arbeit ein softwarebasierter Ansatz zur Erfassung und Bereitstellung der gerätespezifischen Verbrauchseigenschaften entwickelt werden, der sowohl das laufende System ohne integrierte Messhardware mit allen Verbrauchsdaten versorgt, als auch die Grundlage für eine statische Analyse des Quelltextes hinsichtlich des Energieverbrauchs schafft.

1.1 MOTIVATION

Das Thema dieser Arbeit wird aus mehreren Richtungen motiviert. Aus Sicht der bereits genannten Anwendungsmöglichkeiten besteht der Wunsch, den Energieverbrauch von (Peripherie)-Geräten im Betrieb jederzeit abfragen zu können, um damit Aktivitäten effektiver hinsichtlich einer längeren Batterielaufzeit oder Verfügbarkeit planen zu können. Dadurch könnte die verbleibende Batteriekapazität rechnerisch und ohne Zusatzhardware ermittelt werden, sowie Prozessen und Geräten ein Anteil am Gesamtverbrauch zugeordnet werden. Ein weiteres Anwendungsmotiv ist die Ermöglichung einer statischen Analyse der Systemsoftware hinsichtlich ihres erwarteten Energiebedarfs. Dies erfordert jedoch eine feste Verknüpfung von Softwareaktionen und den damit verbundenen Konsequenzen auf den Energieverbrauch. Ist dieser Zusammenhang hergestellt, so rückt die Vision in greifbare Nähe, den Entwicklungsprozesses solcher Systeme zu beschleunigen, indem zeitraubende Testläufe zur Ermittlung des Verbrauchs und Bewertung der Implementierung eingespart werden können. Die Ergebnisse könnten stattdessen von einem Analysewerkzeug bei der Übersetzung des Systems errechnet werden.

Entwicklung von ES beschleunigen

Die Realisierbarkeit einer rein softwarebasierte Energieerfassung basiert auf der Annahme, dass Geräte in verschiedenen Betriebsmodi zwar unterschiedliche Mengen an Energie konsumieren, gleiche Modi jedoch stets den gleichen (oder hinreichend ähnlichen) Verbrauch aufweisen. Demnach können die Daten einmalig für alle Zustände und Aktionen ermittelt und in ein Softwaremodell eingepflegt werden. Die Wahl der passenden Stelle zum Einbinden dieser Verbrauchswerte führt zu dem nächsten Motiv.

wiederkehrender Verbrauch

Motiviert wird der Entwurf eines energiegewahren *Treiber*modells durch die Tatsache, dass der Gerätetreiber in der Schichtenarchitektur eines Betriebssystems dem assoziierten Gerät am nächsten ist und die meisten Informationen über das Gerät besitzt. Aus Sicht der übrigen Betriebssystemschichten stellt der Gerätetreiber eine funktionale Softwarerepräsentation des Geräts dar, die über eine individuelle Schnittstelle die Interaktion mit dem Gerät gestattet. Wird der Treiber jedoch dahingehend umgestaltet, dass er den Betriebszustand des Geräts (hinsichtlich des Energieverbrauchs) repräsentiert, können die zustandsbezogenen Verbrauchswerte in den Treiber eingepflegt und darin gekapselt werden. Der Treiber kann um eine *Power*-API erwei-

Abbildung des Gerätezustands

tert werden, die Auskunft über den aktuellen Modus erteilt, sowie die Kosten von Aktionen und Zuständen ausgibt. Die Bündelung im Treiber erlaubt eine hohe Konfigurierbarkeit und Wiederverwendbarkeit. Wird ein Gerät einschließlich Treiber aus dem System entfernt, müssen an den verbleibenden Komponenten und ihren Daten keine Änderungen vorgenommen werden, da jeder Treiber nur Verbrauchsdaten zu seinem Gerät enthält. Ferner können Geräte und Treiber an anderer Stelle wiederverwendet werden, ohne dass sie neu vermessen werden müssen.

wiederverwendbar

1.2 ZIEL

Diese Arbeit soll den ersten Schritt zur Realisierung der genannten Anwendungsfälle und Visionen darstellen und ein geeignetes Treibermodell liefern, dass sich allgemein anwenden lässt, eine einfache Vermessung der Hardware unterstützt, möglichst Ressourcen schont, hinreichend genau modelliert und schließlich dem System alle Verbrauchswerte zur Verfügung stellt. Dabei ist ersichtlich, dass die Qualität dieser Informationen im höchsten Maße von der Modelltreue und der Güte der Messwerte abhängig ist. Denn die beste Analyse ist nur so exakt, wie ihre zugrunde liegenden Daten. Deswegen beschäftigt sich diese Arbeit neben dem eigentlichen Entwurf eines geeigneten Modells und dessen Evaluation auch ausführlich mit Techniken zur Erfassung und Beurteilung von Verbrauchswerten. Da hierbei der Fokus auf besonders energiesparenden Peripheriegeräten liegt, treten Messungenauigkeiten und parasitäre Effekte selbst beim Einsatz von spezieller (noch prototypischer) Messhardware auf. Die Arbeit erfordert daher den Einsatz von interdisziplinärem Wissen aus Informatik und Elektrotechnik.

Güte der Messwerte

1.3 STRUKTUR DER ARBEIT

Der Inhalt der Arbeit gliedert sich in zwei Teile. Im ersten Teil befindet sich der wissenschaftliche Behandlung des Arbeitsthemas. Diese beginnt im Anschluss an die Einleitung mit den benötigten Grundlagen, die zur Nachvollziehung der einzelnen Entwurfs- und Implementierungsentscheidungen erforderlich sind. Anschließend werden verwandte Forschungsarbeiten vorgestellt, die sich mit ähnlichen Themen beschäftigen. Dabei werden einerseits die Einflüsse aus diesen Arbeiten herausgestellt, aber auch Unterschiede zum Ansatz dieser Arbeit hervorgehoben. Danach folgt der Entwurf des Treibermodells, sowie die Entwicklung der benötigten Zusatzkomponenten. Gemäß dieses Entwurfs wird im anschließenden Kapitel die Implementierung repräsentativer Gerätetreiber behandelt. Es werden Eigenarten der Geräte aufgezeigt, Hinweise zum Messverfahren gegeben und schließlich Automatisierungswerkzeuge vorgestellt. Bereits in diesen

Vorbereitungen

Entwicklung

beiden Kapiteln werden erste Evaluationen und Bewertungen vorgenommen, da sie Grundlage für Entwurfs und Implementierungsentscheidungen sind. Die Übergänge zwischen den Kapiteln sind dabei fließend und können nicht strikt getrennt werden, da es sonst zu Lücken in der Argumentation oder unnötiger Redundanz kommen würde. Im Evaluationskapitel wird schließlich das Gesamtmodell bewertet. Dies beinhaltet eine detaillierte Analyse der Modelltreue und Genauigkeit, sowie eine Diskussion über weitere Automatisierungsmöglichkeiten. Zusätzlich wird aufgezeigt, wie die Informationen aus dem Treibermodell bei der Codeanalyse verwendet werden können. Schließlich werden der Ressourcenbedarf des Modells, die Generalisierbarkeit und seine Einschränkungen aufgezeigt und bewertet. Der wissenschaftliche Teil endet mit einer Zusammenfassung, die auch das Fazit und einen Ausblick enthält.

fließende Evaluation

Dokumentation

Der zweite Teil der Arbeit dokumentiert die vorgenommenen Erweiterungen an den Werkzeugen und am Betriebssystem Kratos, die zur Realisierung dieser Arbeit notwendig waren. Zum Schluss folgen größere Graphiken und längere Quelltexte, die aus Gründen der Übersichtlichkeit dorthin ausgelagert wurden.

2 GRUNDLAGEN

Dieses Kapitel führt erforderliche Grundlagen, Werkzeuge und Arbeitsumgebungen ein, um die Herausforderungen und Lösungsansätze dieser Arbeit nachvollziehen zu können. Zunächst werden die Ursachen für den Energieverbrauch in eingebetteten Systemen sowie Methoden zur Modellierung ihres Verbrauchs erläutert. Danach werden das Messsystem und die Evaluationsplatformen mit ihren spezifischen Eigenschaften vorgestellt. Der letzte Teil behandelt schließlich das Betriebssystem, in welches der Entwurf dieser Arbeit exemplarisch integriert wird.

2.1 ENERGIEVERBRAUCH IN EINGEBETTETEN SYSTEMEN

Um die Leistungsaufnahme von Digitalschaltungen und eingebetteten Systemen verstehen zu können und so eine realitätsgetreue Modellierung des Energieverbrauchs durchzuführen, werden in diesem Abschnitt die Hauptursachen für die Entstehung der Verlustleistung solcher Systeme erläutert.

Aktuelle integrierte Digitalschaltungen werden als CMOS-Schaltungen entworfen. Das bedeutet, dass Logikelemente aus komplementär geschalteten MOSFETs konstruiert werden. Zur Veranschaulichung ist ein CMOS-Inverter in Abbildung 2.1a dargestellt [Göo8]. Darin werden jeweils ein P- und N-Kanal MOSFET in Serie geschaltet und ihre Gate-Anschlüsse mit einander verbunden. Je nachdem, ob eine positive oder negative Spannung U_E auf der Gate-Leitung anliegt, leitet entweder der untere oder der obere Transistor, sodass U_A entweder auf Masse- oder Versorgungsspannungsniveau liegt.

komplementäre MOSFETs

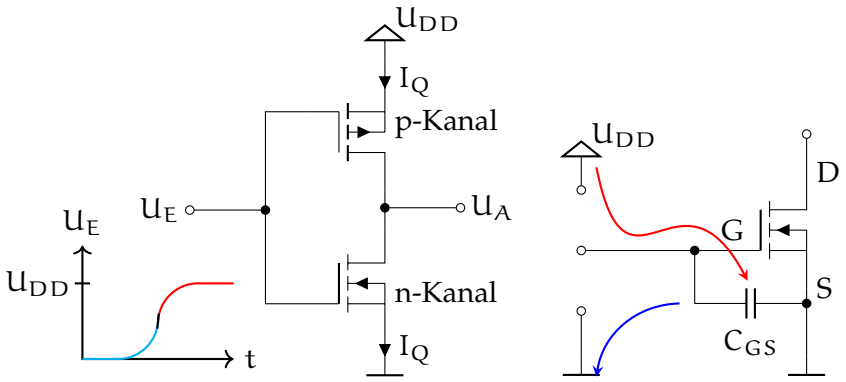
Beim Umschalten der Gatespannung kommt es allerdings wegen einer endlichen Flankensteilheit zu einer Übergangsphase, in der beide Transistoren teilweise leitfähig sind (in Abbildung 2.1a violett dargestellt). Dadurch fließt für kurze Zeit ein Querstrom I_Q durch den Inverter und erzeugt so Verlustleistung.

Querstrom

Die zweite Ursache für den Energieverbrauch ergibt sich aus parasitären Kapazitäten im einzelnen Transistor (Abbildung 2.1b). Im Gegensatz zur Basis bei Bipolartransistoren fließt bei MOSFETs kein Strom durch das Gate in die übrigen Teile des Transistors. Stattdessen genügt alleine das Anliegen einer Gatespannung zum Schalten des MOSFETs. Dadurch stellt das Gate jedoch eine Kondensatorfläche dar und erzeugt zusammen mit der Source-Fläche des Transistors eine parasitäre Kapazität C_{GS}.

kein Basisstrom

parasitäre Gatekapazität



(a) CMOS-Inverter.

(b) N-Kanal MOSFET mit eingezeichneter parasitärer Kapazität C_{GS}.

Abbildung 2.1: Ursachen für den Energieverbrauch in CMOS-Schaltungen. Links fließt beim Umschalten der Gatespannung U_E kurzzeitig ein Querstrom I_Q durch die Schaltung. Rechts führt das Umschalten des Gates zu einem Auf- und Entladen der parasitären Gatekapazität C_{GS} und verbraucht dadurch Energie.

Im Falle einer positiven Gatespannung wird diese Kapazität geladen und erzeugt einen kurzzeitigen Gatestrom (rot dargestellt). Beim Umschalten wird die gespeicherte Ladung wieder zur Masse abgeführt (blau), da der Kondensator kurzgeschlossen wird.

Beide Effekte skalieren proportional zur Umschalthäufigkeit und korrespondieren deswegen in getakteten Systemen unmittelbar mit der Taktfrequenz. In Mikrocontrollern (und auch anderen Digitalschaltungen) werden deswegen nicht benötigte Komponenten durch einfaches Abschalten ihres Arbeitstaktes deaktiviert, sodass ihr Energieverbrauch auf das Niveau vernachlässigbarer Leckströme abfällt.

Abschalten des Arbeitstaktes

Zusätzlich zur Taktrate hat die Betriebsspannung einen viel stärkeren Effekt auf die Leistungsaufnahme integrierter Schaltungen. Die Leistungsaufnahme $P = U \cdot I$ ergibt sich als Produkt aus Strom I und Spannung U. Nach dem Ohm'schen Gesetz

Einfluss der Betriebsspannung…

$$I = \frac{U}{R} \tag{2.1}$$

kann der Stromfluss durch die Betriebsspannung zusammen mit dem Leitungs- und Transistorwiderstand der Schaltung (R) dargestellt werden. Daraus resultiert eine Leistungsaufnahme der Schaltung

$$P = U \cdot \frac{U}{R} \sim U^2, \tag{2.2}$$

…ist quadratisch

die quadratisch mit der Betriebsspannung steigt. Halbleiterhersteller sind deswegen stets bemüht, die minimal erforderliche Betriebsspannung möglichst gering zu halten. Diesem Belang wirkt die Gatekapazität jedoch entgegen, da sie mit einer geringeren Betriebsspannung eine längere Ladezeit benötigt, bevor der Transistor vollständig

durchschaltet oder sperrt. Schlussendlich begrenzt dies die maximale Taktrate der Schaltung.

geringe Spannung = geringe Taktrate

Neben den vorgestellten Effekten verbrauchen noch weitere – für eingebettete Systeme typische – Schaltungen oder Elemente Energie: Spannungsregler wandeln den Spannungsüberschuss in Wärme um. Funkmodule benötigen Energie zum Betrieb von Verstärkern und zum Aussenden elektromagnetischer Strahlung. Referenzspannungsquellen werden mittels permanentem Querstrom durch eine Zenerdiode erzeugt (wichtig für AD-Wandlung). Licht- und Temperatursensoren verändern ihren Widerstand und können nur mit einem Stromfluss durch den Sensor eine messbare Größe (Spannungsabfall) liefern.

weitere Verbraucher

Für die Entwicklung energieeffizienter eingebetteter Systeme bedeutet dies die Auswahl einer möglichst geringen Betriebsspannung und *so niedrig wie nötig* getakteten Komponenten. Zudem sollten alle nicht benötigten Teile stillgelegt oder abgeschaltet werden.

2.2 ENERGIEMODELLE FÜR EINGEBETTETE SYSTEME

Zur Modellierung des Energieverbrauchs von eingebetteten Systemen bzw. ihrer einzelnen Komponenten gibt es unterschiedlichste Ansätze, die sich insbesondere in der Detailliertheit der Modellierung unterscheiden. Ansätze wie in [SKWM01] führen die Modellierung bis auf die Instruktionsebene von Mikroprozessoren durch und berücksichtigen das Umschalten jedes einzelnen Bus-Bits zwischen benachbarten Anweisungen. Die Erfassung der erforderlichen Daten ist mit umfangreichen Messungen verbunden, da die Hersteller solche Informationen in der Regel nicht zur Verfügung stellen.

Modell auf Instruktionsebene

Stattdessen vereinfachen einige Hersteller ihre Angaben sogar auf ein absolutes Minimum und geben zu ihrer Hardware lediglich Durchschnittswerte für wenige Anwendungsszenarien an.

Datenblatt enthält nur Durchschnittswerte

Deswegen ist ein häufig anzutreffender und naheliegender Ansatz in der Spanne der möglichen Granularität eine Orientierung an der Funktionalität der Komponente (siehe dazu das nächste Kapitel). Digitale Bauteile werden zumeißt als Zustandsautomat entwickelt und im Betrieb als solche angesteuert. Die Komponente kann sich in verschiedenen Zuständen befinden (z. B. *aus*, *bereit*, *arbeitend*) und damit verbundene Aufgaben erfüllen, wie das Durchführen einer Datenübertragung, Erfassen eines Messwerts oder einfaches Energiesparen.

Gerät als Zustandsautomat

In jedem Zustand und bei jedem Zustandsübergang kann die Komponente unterschiedliche Mengen von Systemressourcen in Anspruch nehmen, wie Speicher, Rechenzeit – aber auch Energie. Die Zustandsübergänge können auf verschiedenste Weise angestoßen werden: Ein Sensor kann durch einen *expliziten Befehl* mit der Erfassung eines Messwerts beauftragt werden; ein Mikrocontroller durch ein *externes Ereignis* aus dem Energiesparmodus geweckt werden; und ein Funk-

expliziter Zustandswechsel

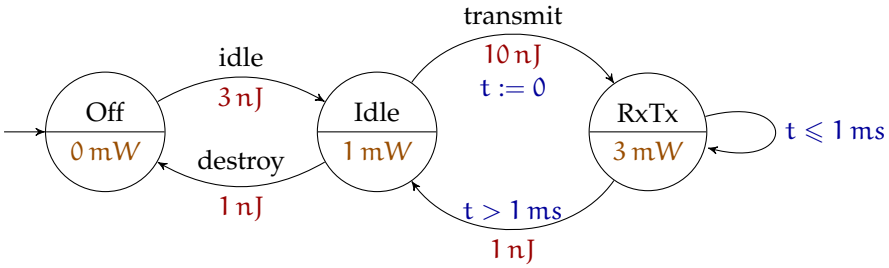


Abbildung 2.2: PTA einer fiktiven Kommunikationsschnittstelle. Zustände und Transitionen sind mit Kosten annotiert (orange, rot). Übergänge können durch Eintreten eines Ereignisses (schwarz) oder durch Erfüllen einer zeitlichen Bedingung (blau) ausgelöst werden.

zeitabhängiger Zustandswechsel

modul kann nach *Ablauf einer bestimmten Zeit* der Inaktivität selbstständig seine Sendeeinheit herunterfahren.

Ein häufig anzutreffendes theoretisches Modell zum Abbilden solcher Komponenten sind PTA [BFH+01], wie das folgende Kapitel zeigen wird. Dabei handelt es sich um endliche Automaten, engl. ***F**inite **S**tate **M**achines* (FSM), deren Zustände und Transitionen mit Kosten

PTA

annotiert sind. Zusätzlich sind PTA mit einer Menge von individuell löschbaren Stoppuhren ausgestattet und ermöglichen dadurch neben *expliziten* Zustandsübergängen die Formulierung *zeitabhängiger* Transitionen, sowie ihre Kombination.

Zur Veranschaulichung ist der PTA einer fiktiven Kommunikationsschnittstelle exemplarisch in Abbildung 2.2 dargestellt. Der Automat setzt sich aus drei Zuständen zusammen, dessen Transitionen durch die Ereignisse *idle*, *transmit*, *destroy* oder durch Erfüllen einer zeitabhängigen Bedingung (blau) ausgelöst werden. Die Transitionen und Zustände sind jeweils mit Kosten annotiert (rot, orange). Tritt im Zustand *Idle* das Ereignis *trasmit* auf, wird die Stoppuhr t zurückgesetzt und der Automat verbleibt für 1 ms im Zustand *RxTx* bevor er selbstständig zurück nach *Idle* wechselt.

PTA *Definition*

Formal ist ein PTA A definiert als Tupel (L, l_0, E, I, P) mit der endlichen Menge von Stoppuhren C und $\mathcal{B}(C)$ als Menge von Formeln, die in konjugierter Form paarweise beliebige natürliche Zahlen mit dem Wert der Stoppuhren vergleichen ($=, <, \geqslant, \ldots$) und damit die Transitionsbedingung formulieren [BFH+01]. Ferner bedeuten

- L Menge der Zustände,
- l_0 Startzustand, $l_0 \in L$,
- E Menge der Transitionen, $E \subseteq L \times \mathcal{B}(C) \times \Sigma \times \mathcal{P}(C) \times L$,
- $\mathcal{P}(C)$ Potenzmenge von C,
- I Menge der Invarianten zu den Zuständen (z. B. Mindestverweilzeit), $I : L \to \mathcal{B}(C)$,
- P Menge der Kosten, $P : (L \cup E) \to \mathbb{N}$.

Eine Transition $(l, b, \sigma, r, l') \in E$ beinhaltet somit Start- und Endzustand (l respektive l'), eintreffendes Ereignis σ, Transitionsbedingung b und die Menge der zurückzusetzenden Stoppuhren r.

Verzögerungskanten

Die Invarianten können als Verzögerunskanten angesehen werden, die den Automaten für die Dauer der erfüllten Transitionsbedingung im aktuellen Zustand festsetzen. Dem entgegen werden die Transitionen aus E verzögerungsfrei durchlaufen, wenn ein passendes Ereignis eintrifft und die Transitionsbedingung erfüllt ist.

Kosten, Kostenrate

Für Zustände und Kanten sind die Kosten unterschiedlich definiert. Sie beschreiben bei Kanten den Preis zum Passieren der Transition. Bei Zuständen wird hingegen eine Kostenrate angegeben, sodass sich die Kosten beim Verlassen des Zustands als Produkt aus Verweildauer und Kostenrate berechnen.

Überträgt man die abstrakten Kosten auf die Ressource *Energie*, so wird für das Passieren einer Transition ein Energieverbrauch und für jeden Zustand eine Verlustleistung annotiert.

Die Vorteile für die Verwendung eines etablierten theoretischen Modells liegen bei der Existenz von Werkzeugen, die eine Analyse der Automaten erlauben. So kann das Programm Uppaal Cora [Beh14] günstigste Wege in PTA unter Angabe von Nebenbedingungen berechnen.

2.3 MESSSYSTEM

Prototyp: MIMOSA

Zur Bestimmung des Energieverbrauchs wird im Rahmen dieser Arbeit das prototypische Messsystem MIMOSA [BGS14], eine Eigenentwicklung des Lehrstuhls für Eingebettete Systeme [Tec14b], verwendet. Unterstützend werden die Messungen mit einem digitalen Oszilloskop DSO7054A von Agilent Technologies verifiziert. Da die aufgezeichneten Werte von MIMOSA im Gegensatz zu einem Oszilloskop unterschiedlich interpretiert werden müssen, wird im Folgenden auf die Funktionsweise von MIMOSA eingegangen und die verwendete Methode der Leistungs- bzw. Energiemessung erläutert.

2.3.1 *MIMOSA*

Das am Lehrstuhl für eingebettete Systeme [Tec14b] entworfene **M**essgerät zur **i**ntegrativen **M**essung **o**hne **S**pannungs**a**bfall (MIMOSA)

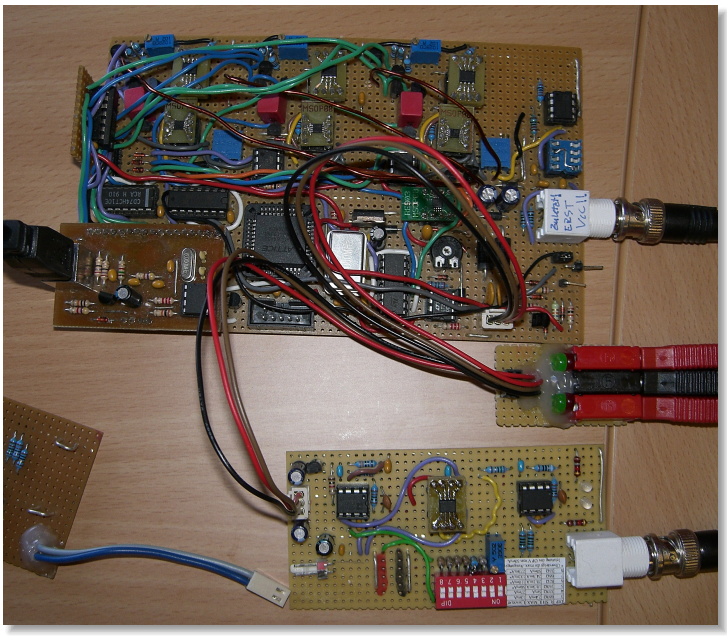


Abbildung 2.3: MIMOSA Prototyp: Unten im Bild befindet sich die Spannungsreglung mit Shunt zum Anschluss eines DUT (unten links). Die Platine im oberen Teil des Bildes führt die Integration, AD-Wandlung und die USB-Übertragung zum Computer durch.

Messung von Low-Power Geräten

(siehe Abbildung 2.3) ist speziell auf die präzise Energiemessungen von Low-Power Komponenten, wie eingebetteten Systemen oder einzelner Komponenten zugeschnitten. Es hebt sich durch zwei Kerneigenschaften von Messungen mit einem Oszilloskop und Energieerfassungswerkzeugen wie PowerScale [Hit14] ab:

Integration

- Durch die analoge Integration der Leistungsaufnahme zwischen den Abtastzeitpunkten werden auch kurze, impulsartige Veränderungen des Energieverbrauchs, die kürzer als die Abtastrate andauern, zuverlässig mit berücksichtigt.

Spannungsregelung

- Die besondere Spannungsregelung für das Messobjekt erlaubt den Einsatz von sehr großen Shunt-Widerständen, ohne dabei die Stabilität der Versorgungsspannung des Prüflings zu gefährden. Damit fällt eine höhere Spannung am Shunt ab und erlaubt so präzisere Messungen.

2.3.1.1 *Aufbau*

Der Aufbau vom MIMOSA ist in Abbildung 2.4 schematisch dargestellt. Im linken Teil der Schaltung wird das Messobjekt durch eine geregelte Spannungsquelle mit einer konstanten Betriebsspannung versorgt, unabhängig[1] von ihrem tatsächlichen Stromverbrauch. Der

1 Im Rahmen der Spezifikationen

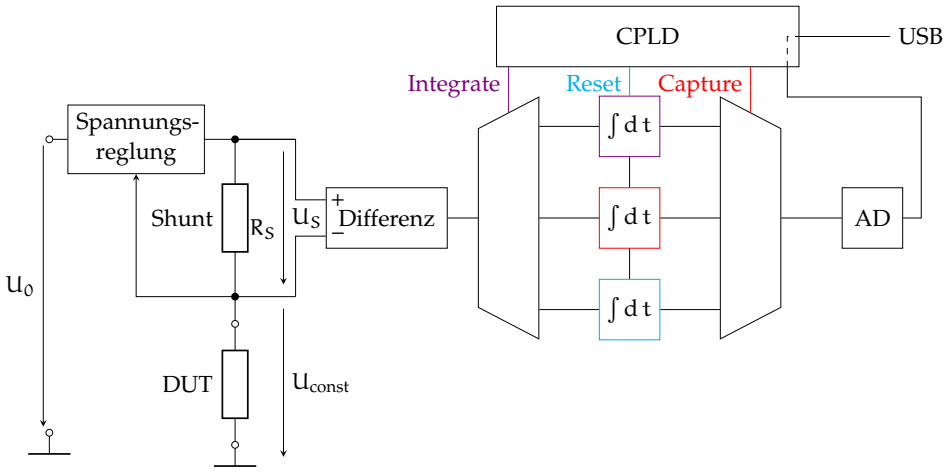


Abbildung 2.4: Schematische Darstellung von MIMOSA. Links erzeugt der Stromfluss durch das DUT einen Spannungsabfall U_s über dem Shunt. Die Differenzspannung wird einem von drei parallelen Analog-Integrieren zugeführt und anschließend abgetastet. Zur lückenlosen Energieerfassung durchlaufen alle Integrierer versetzt die Sequenz *integrieren*, *abtasten* und *löschen*.

gesamte Strom durchfließt dabei ebenfalls den Shunt R_s und erzeugt an dieser Stelle einen Spannungsabfall U_s, der nach dem ohmschen Gesetz

$$I_s = \frac{U_s}{R_s} \tag{2.3}$$

proportional zum Stromfluss I_s durch den Shunt R_s ist. Diese Spannung wird abwechselnd einer von drei analogen Integrationsstufen zugeführt. Die übrigen Integrationsstufen werden unterdessen entweder gelöscht oder abgetastet. Mit einer Frequenz von $f_s = 100\,\text{kHz}$, was auch der Abtastfrequenz von MIMOSA entspricht, durchläuft jeder Integrator jeweils versetzt die Folge *integrieren*, *abtasten* und *löschen*. Während der Phase *abtasten* wird die aus der Integration resultierende Spannung einem AD-Wandler zugeführt und schließlich per USB an einen Computer übertragen. Durch diese wechselseitige und kontinuierliche Integration ist eine lückenlose Erfassung des Energieverbrauchs zwischen zwei Abtastzeitpunkten möglich. Dieser wird als echter Mittelwert innerhalb eines Abtastintervalls erfasst.

3 Integrationsstufen

integrieren, abtasten, löschen

2.3.1.2 *Spannungsregelung*

Typischerweise wird zur Messung des Stromflusses durch ein Testobjekt ein niederohmiger Shunt-Widerstand in Serie geschaltet und der Spannungsabfall an diesem Widerstand gemessen. Der Shunt wird in diesen Fällen bewusst klein gewählt, da mit zunehmendem Stromfluss durch das DUT eine größere Spannung am Shunt abfällt. Wenn-

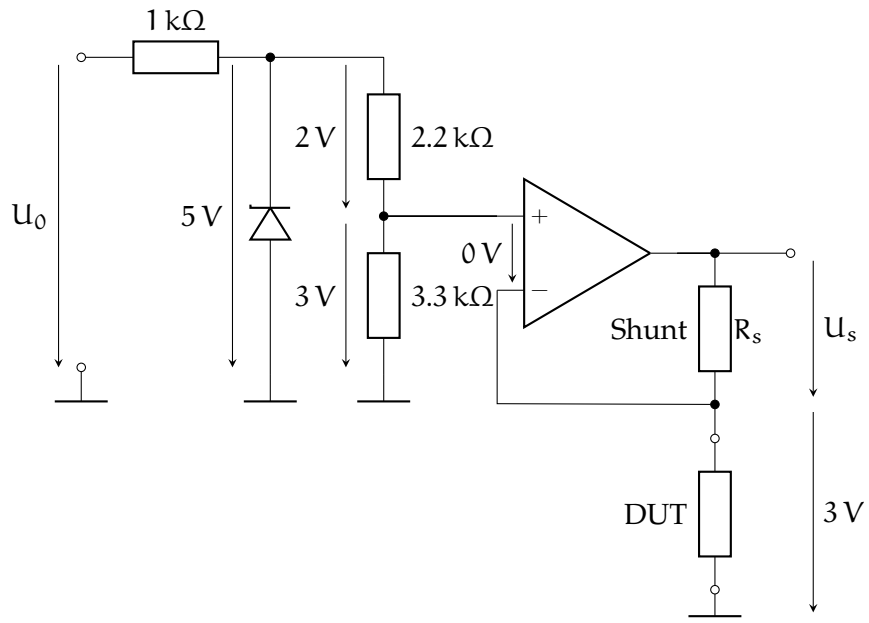


Abbildung 2.5: Vereinfachte Darstellung der Spannungsreglung von MIMOSA. Durch die Rückkopplung regelt der Operationsverstärker seine Ausgangsspannung so weit nach, bis sich eine Differenzspannung von 0 V zwischen beiden Eingängen einstellt. Damit fallen die 3 V Referenzspannung am ⊕-Eingang auch über dem DUT ab. Der Stromfluss durch das DUT ist mit dem Shunt-Strom identisch und führt am Shunt zu einem messbaren Spannungsabfall U_s.

gleich dies wünschenswert ist, um auf den Stromfluss rückzuschließen, bedeutet der Spannungsabfall U_s am Shunt eine Verringerung der Versorgungsspannung des DUT um U_s. Als Konsequenz kann ein zu groß gewählter Shunt die Stabilität des Prüflings gefährden, da insbesondere in Digitalschaltungen beim Unterschreiten der minimalen Versorgungsspannung im besten Fall eine Notabschaltung (Brown-Out) erfolgt; im schlimmsten Falle jedoch unvorhersehbares Verhalten auftritt.

großer Shunt gefährdet Stabilität

Ein zu niederohmiger Shunt hat hingegen den Nachteil, dass die Messung aufgrund des geringeren Spannungsabfalls einen verringerten Rauschabstand aufweist. Dadurch wirken sich eingefangene Störeinflüsse stärker auf die Messung aus, was insbesonders bei der Vermessung von energiesparenden Komponenten auffällt.

kleiner Shunt = geringer Störabstand

Aus diesem Grund wurde für MIMOSA eine intelligente Spannungsreglung entwickelt, die sowohl einen großen Versorgungsbereich (bezogen auf die Leistungsaufnahme des DUT) ermöglicht, als auch die Verwendung von verhältnismäßig hochohmigen Shunts gestattet. Herzstück dieser Spannungreglung (Abbildung 2.5) ist ein rückgekoppelter Operationsverstärker. An dessen ⊕-Eingang wird eine konstante Referenzspannung von 3 V angelegt, die sich durch den

besondere Spannungsreglung

Spannungsteiler von 2.2 kΩ und 3.3 kΩ aus der 5 V Präzisions-Spannungsreferenz (hier als Zenerdiode dargestellt) ergibt. Dadurch ist die Referenzspannung von den Versorgungsspannung U_0 unabhängig.

Durch die Rückkopplung des Operationsverstärkerausgangs an den ⊖-Eingang, regelt der Operationsverstärker seine Ausgangsspannung stets so weit nach, bis sich eine Differenzspannung von 0 V zwischen den beiden Eingängen einstellt.

Daraus folgt aufgrund der 3 V am ⊕-Eingang die identische Spannung am Zweig des ⊖-Eingangs. An diesen Zweig, der ausschließlich über den Shunt vom Operationsverstärkerausgang getrieben wird, wird das DUT angeschlossen und gewährleistet so eine konstante Versorgungsspannung. Wegen des hochohmigen ⊖-Eingangs fließt in den Operationsverstärker kein Strom hinein, sodass $I_{DUT} = I_s$, woraus nach dem Ohm'schen Gesetz (Gleichung 2.3) proportional der Spannungsabfall U_s folgt. Diese Spannung wird schließlich dem Integrationsteil von MIMOSA zur weiteren Erfassung zugeführt (Abbildung 2.4 rechts).

Bei diesem Aufbau kommt es im Falle einer erhöhten Leistungsaufnahme des Prüflings zu keinem Einbruch der Versorgungsspannung. Stattdessen erhöht der Operationsverstärker seine Ausgangsspannung, bis wieder 3 V am DUT abfallen. Deswegen können bei diesem Aufbau bedenkenlos Shunt-Widerstände in der Größenordnung von 100 Ω verwendet werden, während typische Shunts lediglich Werte von weniger als 0.1 Ω annehmen. Die genaue Dimensionierung des Shunts von MIMOSA ist in Anhang B aufgeführt und abhängig von der maximal erwarteten Leistungsaufnahme des DUT.

Kompensation erlaubt großen Shunt

2.3.1.3 *Integration*

Die genaue Erfassung kurzzeitiger Signaländerungen (ðhoher Frequenzen) erfordert eine sehr hohe Abtastrate f_s. Genauer gesagt können nach dem Abtasttheorem nur Signale mit Frequenzanteilen $< \frac{f_s}{2}$ aliasfrei abgetastet werden. Üblicherweise wird dies durch einen Tiefpassfilter vor der Abtastung realisiert, der Frequenzanteile oberhalb von $\frac{f_s}{2}$ unterdrückt. Dadurch fließen kürzere Signaländerungen nicht in das abgetastete Signal ein.

Abtasttheorem

Höhere Abtastraten stellen stärkere Anforderungen an die Messtechnik, was sich auch im Preis entsprechender Geräte wiederspiegelt. Sie ermöglichen damit auch die genaue zeitliche Lokalisierung von Signalveränderungen. In einigen Fällen, wie auch im Kontext dieser Arbeit, ist eine derart genaue zeitliche Zuordnung jedoch nicht notwendig. Vielmehr ist nur der exakte Signalmittelwert innerhalb eines hinreichend kurzen Messintervalls von Interesse, in den auch kürzeste Signalveränderungen mit einfließen.

nur geringe Abtastrate notwendig

Aus diesem Grund wird bei MIMOSA das Messsignal U_s vor der Abtastung in eine analoge Integrationsstufe geführt (Abbildung 2.4

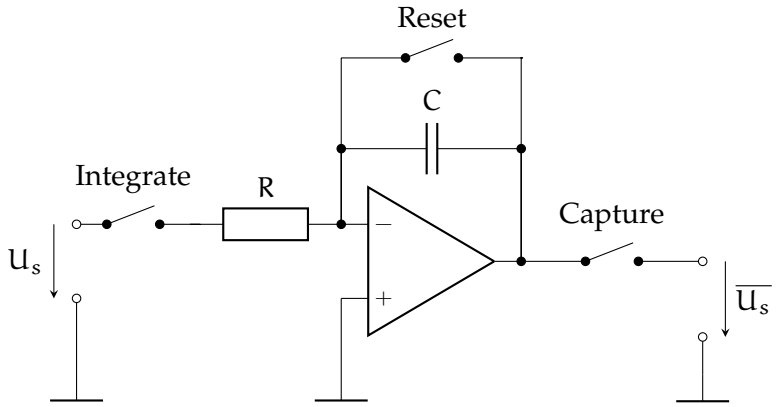


Abbildung 2.6: Analoge Integrierschaltung. Zum Integrieren wird der linke Schalter geschlossen und das Eingangssignal zum Operationsverstärker geführt. Beim Abtasten wird die Ausgangsspannung des Verstärkers an einen AD-Wandler durch schließen des rechten Schalter geleitet. Der mittlere Schalter entlädt den Kondensator und setzt den Integrierer damit zurück.

rechts). Schaltungstechnisch wird ein Integrierer durch einen Operationsverstärker mit einem Kondensator C in der Gegenkopplungsschleife realisiert (Abbildung 2.6). Das Ausgangssignal entspricht dem arithmetischen Mittel $\overline{U}_s$ im Intervall $[0, T]$ [TSG12]:

$$\overline{U}_s(T) = -\frac{1}{T}\int_0^T U_s(t)\,dt + \overline{U}_s(0). \tag{2.4}$$

Als Intervalllänge T ist bei MIMOSA eine Dauer von 10 µs festgelegt, woraus eine Abtastrate von 100 kHz folgt.

In Gleichung 2.4 ist zu erkennen, das die Ausgangsspannung des Integrierers um die Ausgangsspannung zum Zeitpunkt $T = 0$ verschoben ist. Aus diesem Grund muss das Integrationsglied nach jeder Messung zurückgesetzt werden, was durch Kurzschließen des Kondensators C realisiert wird. Da die Messung und das Löschen nicht verzögerungsfrei durchgeführt werden können, das Eingangssignal jedoch lückenlos und fehlerfrei erfasst werden soll, werden drei parallele Integrationsstufen reihum im Wechsel zum Integrieren, Auslesen oder Löschen angesteuert.

Integrierer löschen

Das integrierte Signal wird anschließend von dem AD-Wandler abgetastet. In diesem Fall ist jedoch kein separater Vorfilter notwendig, da die Integrationsstufe selbst einen aktiven Tiefpassfilter erster Ordnung darstellt (Abbildung 2.7). Die Grenzfrequenz dieser Schaltung berechnet sich nach [TSG12] als

$$f_g = \frac{1}{2\pi R_N C}, \tag{2.5}$$

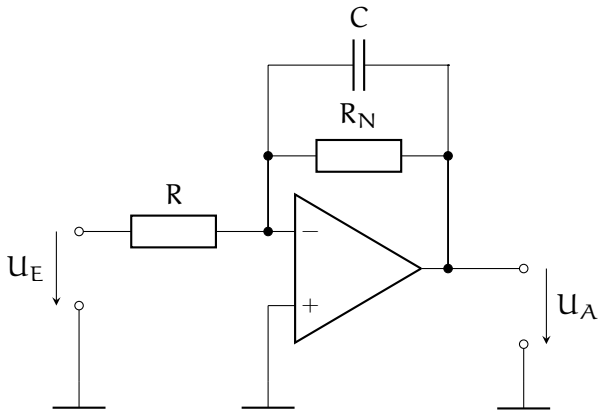


Abbildung 2.7: Aktiver Tiefpassfilter erster Ordnung. Frequenzanteile des Eingangssignals U_E oberhalb der Grenzfrequenz f_g (Gleichung 2.5) werden unterdrückt und fehlen im Augangssignal U_A.

wobei sich in diesem Fall aufgrund des fehlenden R_N und

$$f_g = \lim_{R \to \infty \Omega} \frac{1}{2\pi R_N C} = 0\,\text{Hz} \tag{2.6}$$

ein Tiefpassfilter ergibt, der alle Frequenzen oberhalb von 0 Hz unterdrückt und somit in jedem Fall die Aliasfreiheit sicherstellt. Die unterdrückten Frequenzanteile werden stattdessen bei der Integration mit berücksichtigt und fließen in den Mittelwert ein.

garantierte Aliasfreiheit

2.3.1.4 *Interpretation der Messwerte*

Da den Integratoren eine Spannung zugeführt wird, die proportional zur Leistungsaufnahme des DUT ist, lassen sich die digitalen Samples als mittlere Leistungsaufnahme seit dem letzten Sample (innerhalb der letzten 10 µs) interpretieren. Zwar können die resultierenden Samples schrittweise rechnerisch in die korrespondierende Leistungsaufnahme überführt werden, berücksichtigt dann jedoch keine Bauteiltoleranzen und Temperaturabhängigkeiten. Stattdessen kann eine Kalibrierungsmessung mit einem bekannten Messwiderstand R_{REF} als DUT, sowie eine Messung gänzlich ohne DUT durchgeführt werden, sodass sich für ein gemessenes Sample S folgende mittlere Leistungsaufnahme P ableiten lässt:

Temperaturabhängigkeit

$$P(S) = (S - S_\infty) \cdot \frac{P_{REF}}{S_{REF} - S_\infty} \tag{2.7}$$

$$= (S - S_\infty) \cdot \frac{U_{const}^2}{R_{REF}\,(S_{REF} - S_\infty)}, \tag{2.8}$$

mit dem Referenzwiderstand R_{REF}, der bei der Versorgungsspannung des DUT U_{const} zum Messwert S_{REF} führt, sowie dem Messwert S_∞

ohne angeschlossenem Lastwiderstand. Dies korrespondiert mit der Energieaufnahme

$$E(S) = \frac{1}{f_s} \cdot P(S) \tag{2.9}$$

innerhalb eines Messintervalls, bzw. mit dem Energieverbrauch

$$E(S_i, a, b) = \frac{1}{f_s} \cdot \sum_{i=a}^{b} P(S_i) \tag{2.10}$$

während einer längeren Aufzeichnungsperiode im Sampleintervall $[a, b]$ aus der Samplefolge S_i.

2.4 HARDWARE: DIE EVALUATIONSPLATFORMEN

Zur Entwicklung, Erprobung und Evaluation des Treibermodells wurden zwei Hardwareplattformen eingesetzt, deren Kern in beiden Fällen ein MSP430 FR5969 Mikrocontroller [Tex14d] ist. Die MCU ist aus der Mikrocontrollerserie von Texas Instruments mit sehr geringem Energieverbrauch, engl. ***U**ltra-**L**ow-**P**ower* (ULP) und ist somit besonders für eingebettete Systeme ausgelegt. Zudem sind die Plattformen mit besonders stromsparenden Sensoren, Kommunikationsschnittstellen und Ausgabegeräten ausgestattet. Aus diesem Grund eignen sich diese Plattformen hervorragend als Repräsentanten für die Gruppe energiegewahrer eingebetteter Systeme. Im Folgenden wird zunächst auf den Mikroprozessor eingegangen und seine für diese Arbeit relevanten Funktionen und Konfigurationsmöglichkeiten erläutert. Anschließend werden beide Plattformen kurz vorgestellt, da sie mit verschiedenen Peripheriegeräten ausgestattet sind und hierfür im Rahmen dieser Arbeit energiegewahre Treiber implementiert und evaluiert werden.

ULP *Mikrocontroller*

2.4.1 *MSP430 FR5969*

Beim MSP430 FR5969 (Abbildung 2.8) handelt es sich um einen 16-Bit RISC Mikroprozessor, mit 2 KiB SRAM, 64 KiB FRAM, einer Taktrate bis 24 MHz und einem Betriebsspannungsbereich von 1.8 V bis 3.6 V.

Eine Besonderheit dieses Mikrocontrollers ist neben der geringen Leistungsaufnahme die Verwendung von FRAM als nichtflüchtigen Speicher, anstatt der für Mikrocontroller üblichen EEPROM- oder Flash-Speicher. Die Speicherung der Information erfolgt durch Kondensatoren mit einem speziellen kristallinen (ferroelektrischen) Dielektrium zwischen den Kondensatorplatten [Tex14a]. Zum Beschreiben wird eine Spannung an den Kondensator angelegt, wodurch sich bewegliche Atome innerhalb des Kristallgitters verschieben. Diese Atome verbleiben auch nach dem Entfernen der Spannung in ihrer Position; das Kristallgitter bleibt polarisiert.

Polarisation des Kristallgitters

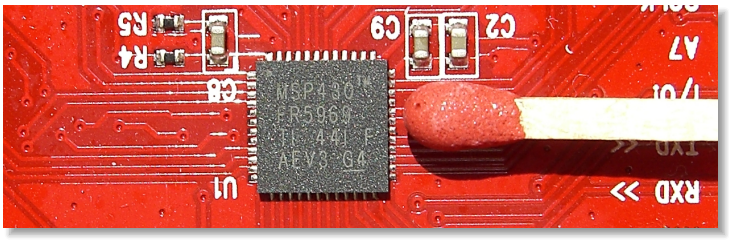


Abbildung 2.8: MSP430 FR5969 Mikrocontroller im Vergleich zu einem Streichholz. Der Mikrocontroller ist aus der ULP-Serie von Texas Instruments und eignet sich besonders für energiesparende eingebettete Systeme. Das in dieser Arbeit entwickelte Treibermodell wird anhand von Platformen mit dieser MCU evaluiert.

Zum Auslesen wird der Kristall erneut polarisiert. Je nach abgegebener Ladung wird durch Vergleich mit einem Referenzwert über den gespeicherten Zustand entschieden. Da auch beim Auslesen der Zustand des Kristalls verändert wird, erfordert analog zum DRAM jedes Auslesen ein Rückschreiben der Information in die Speicherzelle.

Der Vorteil von FRAM liegt beim geringeren Energieverbrauch im Vergleich zu EEPROM oder Flash-Speichern. Während bei FRAM die Betriebsspannung zum Polarisieren des Dielektriums genügt, sind bei EEPROM und Flash-Speichern zumeißt Ladungspumpen (zum Erzeugen einer höheren Spannung als der Betriebsspannung) erforderlich, um Elektronen durch eine Isolatorschicht hindurchtunneln zu lassen und dadurch die Information abzulegen. Ebenso erlaubt das im MSP430 enthaltene FRAM Lese- und Schreibzugriffe mit Geschwindigkeiten von 8 MHz, während die Zugriffsgeschwindigkeiten von Flash-Speicher ca. um den Faktor 100 geringer sind [Tex14a]. *FRAM vs. Flash*

Als Lebensdauer wird eine Größenordnung von 10^{15} Schreib- bzw. Löschzyklen angegeben [Tex14a]. Dieser Wert ist jedoch lediglich eine untere Schranke, da die Tests für die Lebensdauer solcher Speicher zum Zeitpunkt der Veröffentlichung von [Tex14a] noch andauerten und bislang noch keine Fehler aufgetreten waren. *hohe Lebensdauer*

Wie für Mikrocontroller üblich, sind in den MSP430 verschiedenste Peripheriekomponenten integriert. Die für diese Arbeit relevanten Komponenten werden im Folgenden kurz vorgestellt. Eine vollständige Liste der integrierten Peripherie kann aus dem Datenblatt [Tex14b, Tex14d] entnommen werden.

2.4.1.1 *Taktsystem*

Der MSP430 besitzt zur Taktversorgung seiner Komponenten drei zentrale Taktleitungen: MCLK, SMCLK und ACLK. Diese können wahlweise durch verschiedene – im Folgenden erläuterte – Oszilla-

toren direkt, oder jeweils in Zweierpotenzen von 1 bis 32 skaliert, getrieben werden.

Der Mikrocontroller ist mit drei internen und zwei externen Taktquellen ausgestattet, und erlaubt es dadurch, Peripheriekomponenten mit unterschiedlichen Taktraten zu betreiben. Der interne DCO ist ein ab Werk kalibrierter Oszillator, der in erste Linie den Systemtakt bereitstellen soll und liefert wahlweise Frequenzen von 1 MHz, 2.7 MHz, 3.5 MHz, 4 MHz, 5.3 MHz, 7 MHz, 8 MHz, 16 MHz, 21 MHz und 24 MHz.

DCO

Der VLO ist einer interne Taktquelle mit sehr geringer Energieaufnahme (100 nA) und stellt eine Frequenz von ca. 10 kHz bereit. Er ist für Einsatztzwecke ausgelegt, in denen es auf größte Energieeffizienz ankommt und keine exakte Taktrate erforderlich ist.

Schließlich hat der dritte interne Oszillator, MODOSC, eine besondere Funktion. Genauso wie VLO wird auch hier nur eine ungenaue Frequenz im Bereich von 4 MHz bis 5.4 MHz bereitgestellt, benötigt jedoch mit 25 µA relativ viel Energie. Diese Taktquelle wird im Falle des Ausfalls eines externen Oszillators automatisch eingeschaltet und stellt dadurch die Funktionalität der angeschlossenen Peripheriekomponenten (wenn auch nicht exakt getaktet) sicher.

MODOSC, VLO

Die beiden externen Oszillatoren LFXT und HFXT werden durch Quarze oder Quarzoszillatoren betrieben, die an den Mikrocontroller angeschlossen werden. Während HFXT durch Quarze im Frequenzbereich von 0 Hz bis 24 MHz betrieben werden kann, ist LFXT ausschließlich für günstige 32 768 Hz Uhrenquarze ausgelegt. Dafür zeichnet sich LFXT durch eine geringere Stromaufnahme von 180 nA bis 330 nA[2] im Gegensatz zu HFXT mit 75 µA bis 250 µA[3] aus.

Quarze

Das Taktsystem ist eng mit den Energiesparmodi des MSP430 (siehe Abschnitt 2.4.1.9) verknüpft. Doch zunächst müssen einige weitere Komponenten und Mechanismen erläutert werden.

2.4.1.2 *Interruptcontroller*

Um effizient und schnell auf externe oder interne Ereignisse reagieren zu können, ist der Mikrocontroller mit Interrupts ausgestattet. Quellen für Unterbrechungen können u. A. ein abgelaufener Timer, eine abgeschlossene Datenübertragung oder eine Zustandsänderung eines digitalen IO-Pins sein. Tritt ein Interrupt auf, werden der aktuelle Programmfluss unterbrochen, Programmzähler, engl. ***P**rogram **C**ounter* (PC) und **S**tatus **R**egister (SR) automatisch auf den Stack gesichert und ein Eintrag des Interruptvektors angesprungen. Letzteres ist ein spezieller Bereich im Speicher (beim MSP430 ab Adresse 0FFFEh), der zu jeder möglichen Unterbrechungsquelle eine korrespondierende Instruktion bereitstellt. Da eine Instruktion üblicherweise nicht zur Behandlung des Interrupts genügt, werden an der Stelle

Interruptvektor

2 Je nach angeschlossenen Kondensatoren.

3 Je nach Frequenz und angeschlossenen Kondensatoren.

lediglich Sprunganweisungen zu den tatsächlichen Interruptbehandlungsroutinen, engl. ***I**nterrupt **S**ervice **R**outine* (ISR) platziert.

Zum Abschluss der ISR muss die Assembler-Instruktion `reti` ausgeführt werden, was die gesicherten Register PC und SR wiederherstellt und den ursprünglichen Programmfluss wiederaufnimmt.

Die Interruptquellen können einzeln ein- und ausgeschaltet werden, sodass nur auf gewünschte Ereignisse reagiert wird. Bei mehreren gleichzeitigen Ereignissen wird die ISR mir der höchsten (vom Hersteller fest vorgegebenen) Priorität

feste Priorität

ausgeführt. Um verschachtelte Behandlungen zu vermeiden, werden während der Ausführung einer ISR alle Interrupts maskiert (d. h. deaktiviert).

Eine Ausnahme stellen nicht maskierbare Unterbrechungen, engl. ***N**on **M**askable **I**nterrupt* (NMI) dar und bilden die Kategorie der Unterbrechungen mit höchster Priorität für besonders kritische Ereignisse, wie der Ausfall eines Oszillators oder unerlaubte Speicherzugriffe. Deswegen können diese Ereignisse sogar eine ISR unterbrechen.

NMI*: kritisches Ereignis*

Die letzte Kategorie der Unterbrechungen bilden Reset-Ereignisse. Diese unterbrechen jegliche Ausführung, setzen den Mikrocontroller zurück und starten ihn (sofern möglich) neu. Auslöser solcher Ereignisse können ein Low-Pegel am Reset-Eingang des Mikrocontrollers, ein abgelaufener Watchdog (siehe weiter unten) oder ein Einbruch der Betriebsspannung sein.

Reset

2.4.1.3 *DMA Controller*

Mittels **D**irect **M**emory **A**ccess (DMA) können Kopieroperationen im Speicher effizient ohne den Einsatz der CPU durchgeführt werden. Dies bedeutet, dass die CPU während einer solchen Operation entweder andere Aufgaben abarbeiten kann, oder – viel wichtiger für eingebettete Systeme – während dieser Zeit in einen Energiesparmodus wechseln kann.

Der MSP430 besitzt drei unabhängig konfigurierbare DMA-Kanäle und kann Blöckgrößen von bis zu 65 535 Wörtern (= 131 070 B) verarbeiten. Die Übertragung kann durch verschiedenste Ereignisse ausgelöst werden, z. B. können eine abgeschlossene AD-Wandlung oder ein eingetroffenes Byte über eine Kommunikationsschnittstelle die Übertragung eines zuvor konfigurierten DMA-Kanals anstoßen. Der DMA-Controller kann dabei in vier verschiedenen Adressierungsmodi arbeiten:

DMA*-Kanäle*

FESTE ADRESSE NACH FESTER ADRESSE Jede Übertragung kopiert lediglich den Inhalt einer festen Adresse an eine andere feste Adresse. Dadurch kann z. B. das Resultat einer AD-Wandlung direkt auf das Ausgaberegister einer Kommunikationsschnittstelle kopiert werden.

FESTE ADRESSE NACH ADRESSBLOCK Hierbei wird der Inhalt einer festen Speicheradresse sequenziell in einen Speicherblock übertragen. Dadurch können z. B. eintreffende Bytes einer Kommunikationsschnittstelle in einem Puffer zwischengespeichert werden.

ADRESSBLOCK NACH FESTER ADRESSE Umgekehrt zum zuvor genannten Fall kann dieser Modus z. B. zur automatischen Übertragung eines Pufferinhalts über eine Kommunikationsschnittstelle verwendet werden.

ADRESSBLOCK NACH ADRESSBLOCK Mit diesem Modus kann das Kopieren größerer Speicherblöcke effizient gelöst werden, z. B. das Abspeichern von nachbearbeiteten Daten in den nichtflüchtigen Speicher (FRAM).

2.4.1.4 *Digitale IO-Pins*

Fast alle Anschlüsse des MSP430 können als digitale IO-Pins genutzt werden. Dies bedeutet, dass der Spannungspegel an einem Anschluss direkt auf ein Bit eines korrespondierenden Registers abgebildet wird bzw. umgekehrt, je nach konfigurierter Arbeitsrichtung. Jeweils acht IO-Pins werden zu einem sog. Port zusammengefasst und durch ein Register im Speicher des Mikrocontrollers repräsentiert. Sind IO-Pins als Eingang konfiguriert, können optional interne *pull-up* oder *pull-down* Widerstände hinzugeschaltet werden, um undefinierte Zustände durch „in der Luft hängende“ Signalleitungen auszuschließen. Da nach dem Neustart des Mikrocontrollers alle IO-Pins standardmäßig als Eingänge ohne *pull-up/-down* konfiguriert sind, müssen die Einstellungen bei der Initialisierung des Betriebssystems an die angeschlossenen Peripheriekomponenten angepasst werden. Insbesonders ungenutzte Anschlüsse sollten als Ausgang umkonfiguriert werden, da sich sonst ein höherer Energieverbrauch der MCU ergeben kann. Details hierzu sind in Anhang A aufgezeigt.

pull-up/-down

2.4.1.5 *Kommunikationsschnittstellen*

Für die Kommunikation mit externen Peripheriegeräten besitzt der Mikrocontroller zwei ***E**nhanced **U**niversal **S**erial **C**ommunication **I**nterfaces* (EUSCI). Diese können entweder als UART, I^2C oder SPI betrieben werden und unterscheiden sich im Kommunikationsprotokoll, sowie in der Anzahl der verwendeten Signal-, Takt- und Steuerleitungen. Um eine breite Vielfalt an Peripheriegeräten zuzulassen, sind die beiden EUSCI hochgradig konfigurierbar. Dazu gehören u. A. Wortlänge, Bitreihenfolge, Datenrate, Master-/Slave-Modus, optionale Signalleitungen, Interrupts und DMA.

UART, I^2C, SPI

2.4.1.6 *Timer*

Timer in Mikrocontrollern sind – stark vereinfacht – Zähler mit wählbarer und skalierbarer Taktquelle, die ihren Zählerstand stets mit dem Inhalt eines oder mehrerer spezieller Register vergleichen. Je nachdem, ob der Zählerstand geringer, gleich oder größer als der Registerwert ist, können bestimmte Ereignisse ausgelöst werden. So kann beispielsweise periodisch in definierten Zeitabständen ein Interrupt zum Aufruf des Schedulers (siehe Abschnitt 2.5) ausgelöst werden, oder ein PWM-Signal über einen IO-Pin ausgegeben werden. Umgekehrt können die Timer auch zur Zeitmessung bestimmter Ereignisse genutzt werden, indem der Zähler bei einem Zustandswechsel eines IO-Pins gestartet bzw. angehalten wird und der Zählerstand bei bekannter Taktrate die Ereignisdauer widerspiegelt.

Der MSP430 besitzt zwei Timer-Einheiten, deren Zähler jeweils 16 Bit breit sind. Das Betriebssystem Kratos verwendet einen Timer als systemweiten Zeitgeber zum Scheduling und zur Realisierung von Verzögerungen (z. B. die Methode `sleep(time_ms)`).

16 Bit Auflösung

2.4.1.7 *AD-Wandler*

Da eingebettete Systeme häufig Sensoren auslesen, die messbare Größe jedoch eine Analogspannung ist, sind Mikrocontroller zumeißt mit integrierten AD-Wandlern ausgestattet. Der MSP430 hat einen integrierten AD-Wandler mit bis zu 32 Eingangskanälen, wählbarer Taktquelle und einer konfigurierbaren Auflösung von 8-Bit, 10-Bit oder 12-Bit. Die Umwandlung benötigt je nach Auflösung 10, 12 bzw. 14 Takte pro Eingangskanal. Dadurch reduziert sich die maximale Abtastrate, je mehr Eingangskanäle gleichzeitig verwendet werden.

mehr Kanäle = geringere Abtastrate

Der AD-Wandler kann in verschiedenen Arbeitsmodi betrieben werden. So können einer oder mehrere Kanäle sequenziell einmalig, periodisch, oder nur bei ausgewählten Ereignissen umgewandelt werden und nach Abschluss wiederum einen Interrupt oder DMA-Transfer anstoßen.

Als Referenzgröße für die Umwandlung kann eine Spannung des internen Referenzmoduls (1.2 V, 2.0 V und 2.5 V) oder eine extern zugeführte Referenzspannung verwendet werden. Hierbei ist Anzumerken, dass die Erzeugung der internen Referenzspannung einen zusätzlichen Stromfluss zwischen 225 µA und 1.8 mA verursacht.

Stromverbrauch durch Referenzspannung

2.4.1.8 *Watchdog*

Beim Watchdog, zu Deutsch „Wachhund“, handelt es sich um einen getakteten Zähler, der nach Ablauf einer voreingestellten Zeit den Mikrocontroller kontrolliert zurücksetzt. Das Arbeitsprinzip beruht darauf, dass die vom Mikrocontroller ausgeführte Software periodisch ein Zurücksetzen des Zählerwerts veranlasst und dadurch signalisiert, dass das Programm nach wie vor korrekt arbeitet. Als Analo-

Totmannschalter

gie eignet sich ein Totmannschalter im Führerhaus eines Zuges, den der Zugführer regelmäßig betätigen muss, um einen automatischen Sicherheitshalt des Zuges zu verhindern. Hängt sich das Programm beispielsweise in einer Endlosschleife (oder einer anderen zu zeitintensiven Anwendung) auf, löst der Watchdog einen Neustart (Reset) des Systems aus.

Beim MSP430 ist zu beachten, dass der Watchdog nach dem Einschalten oder Neustart standardmäßig eingeschaltet ist und eine voreingestellte Ablaufzeit von ca. 32 ms hat. Besteht wie in Kratos kein Bedarf für einen Watchdog, so muss er rechtzeitig ausgeschaltet werden, da es sonst zu einem ständigen Neustart kommt.

2.4.1.9 *Energiesparmodi*

Weil der sparsame Umgang mit Energie in eingebetteten Systemen, insbesondere bei Batteriebetrieb, entscheidend für die Einsatzdauer und Verfügbarkeit ist, bietet der MSP430 verschiedenste Energiesparmodi an. Neben der Möglichkeit zum gezielten Ein- und Ausschalten interner Peripheriekomponenten kann der gesamte Mikrocontroller in verschiedene Tiefschlafmodi versetzt werden. Dabei wird die CPU angehalten und je nach Tiefe des Modus auch weitere Komponenten wie Oszillatoren oder FRAM stillgelegt. Zum Erwachen aus einem Tiefschlafmodus wird ein Ereignis (z. B. Ablauf eines Timers, externes Signal, etc.) benötigt, das einen Interrupt auslöst und dadurch das Anlaufen der CPU anstößt.

blockierte Tiefschlafmodi

Um zu verhindern, dass beim Übergang in den Energiesparmodus versehentlich der Takt für Komponenten deaktiviert wird, die später zum Erwachen des Prozessors notwendig sind, ist der MSP430 mit einem *clock request* System ausgestattet. Dadurch melden aktive Komponenten an das Taktsystem, durch welche Taktquelle sie aktuell versorgt werden. Die korrespondierenden Energiesparmodi werden dadurch blockiert und bei etwaiger Anforderung auf den nächsten erlaubten Modus umgelenkt. Beispielsweise wird in Kratos ein Timer zum Betrieb des Schedulers von SMCLK mit Takt versorgt. Obwohl in *idle*-Phasen der Befehl zum Übergang in den LPM4 abgesetzt wird, kann wegen der Abhängigkeit von SMCLK nur LPM1 betreten werden (vgl. Tabelle 2.1).

2.4.2 *InBin*

Lagersystems

Das InBin ist ein am Fraunhofer IML [Fra14] entwickelter Prototyp einer energiesparenden Platform, die an Transportkisten innerhalb eines automatisierten Lagersystems angebracht werden soll. Neben dem MSP430 FR5969 als zentrale Steuereinheit ist es ausgestattet mit einem E-PAPER-Display, um energieeffizient jederzeit ablesbare Informationen zur Transportkiste anzuzeigen, einem Funkmodul zur kabellosen Kommunikation und Konfiguration sowie einem Beschleu-

MODUS	EIGENSCHAFTEN	STROMAUFNAHME (16 MHz, 3 V, 25 °C)	AUFWACHZEIT (typ.)
AM	Aktivmodus (bei 75 % Hitrate im FRAM-Cache); Eingeschaltete Peripherie ist aktiv	~1420 µA	–
LPM0	CPU und MCLK werden angehalten, FRAM in Standby versetzt	~225 µA	0 µs
LPM1	Wie LPM0, FRAM wird ausgeschaltet, DCO wird angehalten, wenn nicht von SMCLK benötigt	~180 µA	6 µs
LPM2	Wie LPM1, zudem werden SMCLK und DCO angehalten	~0.9 µA	6 µs
LPM3	Wie LPM2, zudem wird die Gleichspannungsquelle für den DCO ausgeschaltet	~0.6 µA	7 µs
LPM4	Wie LPM3, zudem wird ACLK angehalten	~0.5 µA	7 µs
LPM3.5	Stillegung aller Taktquellen und Komponenten außer der RTC	~0.45 µA	250 µs
LPM4.5	Stillegung aller Taktquellen und Komponenten	~0.2 µA	250 µs

Tabelle 2.1: Betriebsmodi des MSP430 mit ihren korrespondierenden Stromaufnahmen und Aufwachzeiten nach [Tex14d].

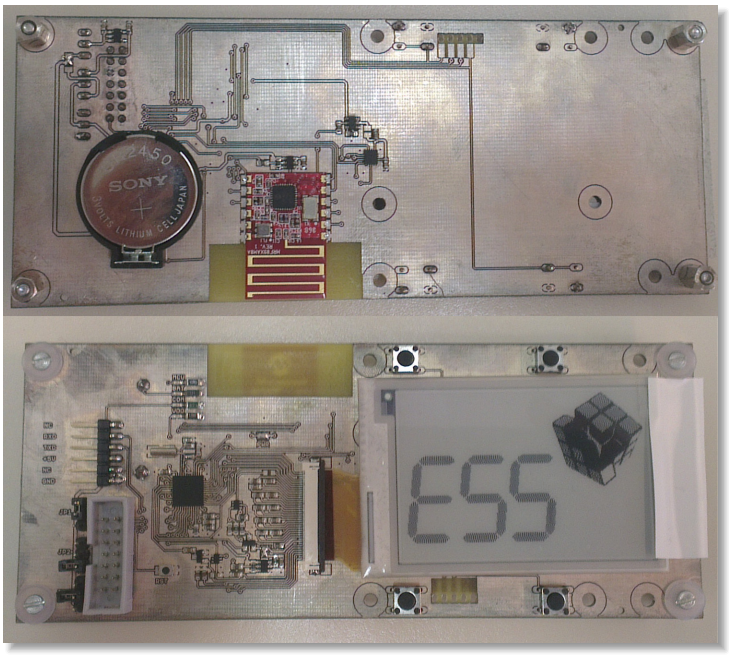


Abbildung 2.9: InBin Prototyp, entwickelt am Fraunhofer IML [Fra14]. Auf der Rückseite (oben) befinden sich Batterie, Funkmodul und Beschleunigungssensor; auf der Vorderseite (unten) sind die Programmierschnittstelle, der MSP430 und ein E-PAPER-Display angebracht.

nigungssensor zur Überwachung der Bewegung. Weitere Komponenten sind eine Batterie, Kontroll-LEDs, sowie vier Drucktaster. Sämtliche Peripheriekomponenten können über vorgeschaltete, vom Mikrocontroller gesteuerte Transistoren vollständig von der Versorgungsspannung getrennt werden, um Standby-Ströme zu vermeiden. Langfristig soll das InBin mittels Energy harvesting durch eine Solarzelle betrieben werden und überschüssige Energie in einem Akkumulator zwischenspeichern. Letztere Komponenten sind im verwendeten Prototypen noch nicht vorhanden, allerdings wurden bereits aus diesem Grund Bauteile mit besonders geringem Energieverbrauch verbaut.

Energy harvesting

2.4.3 *Launchpad*

Zur Erprobung und Evaluation von Schaltungen mit MSP430 bietet Texas Instruments ein Launchpad [Tex14c] an (siehe Abbildung 2.10). Dabei handelt es sich um eine Platine mit einem MSP430 FR5969 Mikrocontroller, dessen Anschlüsse größtenteils zu zwei Pfostenleisten geführt sind, was das Aufstecken von zusätzlichen Peripheriekomponenten erlaubt (wie z. B. rechts in Bild 2.10 das Display). Des Weiteren können zwei Taster und zwei LEDs frei verwendet, sowie ein hochkapazitiver Kondensator zur Spannungsversorgung optional hinzugeschaltet werden. Auf der Platine befindet sich zusätzlich ein Program-

Peripherie aufsteckbar

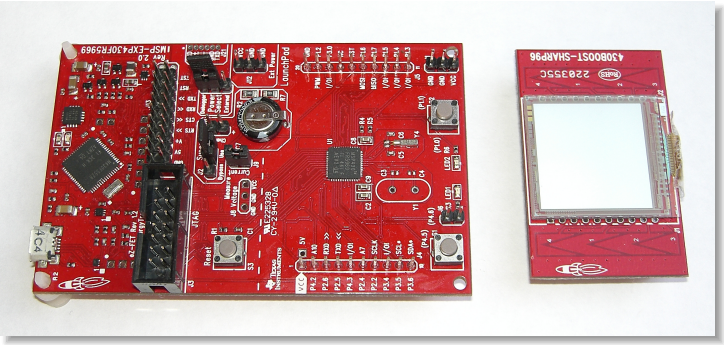


Abbildung 2.10: Launchpad von Texas Instruments [Tex14c]. Links im Bild ist die Hauptplatine mit MSP430 FR5969 Mikrocontroller (rechts) und integrierter Programmierschaltung zum Aufspielen der Firmware und Debuggen (links). Rechts befindet sich eine aufsteckbare Erweiterungplatine mit Display.

miermodul für den Mikrocontroller, welches das Debuggen und Aufspielen von Firmware direkt über USB ermöglicht. Um Energiemessungen nicht zu verfälschen, kann das Programmiermodul über eine Jumperleiste vollständig von der übrigen Schaltung getrennt werden.

Programmer

2.5 SOFTWARE: DAS BETRIEBSSYSTEM KRATOS

Das Betriebssystem Kratos ist eine featureorientierte Erweiterung von AOStuBS, einem für Lehr- und Forschungszwecke entworfenen Betriebssystem. Streng genommen handelt es sich wegen der hohen Konfigurierbarkeit um eine Betriebssystem Produktlinie. Das System ist in der Programmiersprache AspectC++ implementiert und erlaubt damit das aspektorientierte Hinzufügen von Erweiterungen, ohne Veränderungen am Quelltext des Gesamtsystems vornehmen zu müssen. Der Mechanismus hierzu wird im nachfolgenden Abschnitt durch einen kurzen Exkurs in AspectC++ vorgestellt. Anschließend werden die Struktur, Erweiterbarkeit und Konfigurierbarkeit von Kratos erläutert und anhand eines durchgängigen Beispiels – dem Einbau eines neuen Threads – demonstriert.

featureorientiert

2.5.1 *Exkurs: AspectC++*

AspectC++ ist eine Erweiterung der Programmiersprache C++ um den Mechanismus der Aspekte [pUS14]. Sowohl Klassen, als auch Aspekte, kapseln Funktionalität in zusammenhängenden Quelltextblöcken. Aspekte können allerdings, im Gegensatz zu Klassen, querschneidende Belange implementieren, da ihr übersetzter Quellcode seinen lokalen textuellen Zusammenhang aufgeben kann. Dies wird dadurch realisiert, dass der Aspektcode beim Übersetzen zunächst ge-

Klasse vs. Aspekt

zielt in den Quelltext anderer Module *eingewoben* wird und anschließend die Übersetzung von nun klassischem C++ Code folgt.

Einweben

Um die Stellen, an denen Code eingewoben werden soll zu definieren, werden Pointcuts verwendet. Dabei handelt es sich um Ausdrücke mit erlaubten Platzhaltern, die (vergleichbar zu regulären Ausdrücken) auf eine Menge von Typen, Attributen, Funktionen, Variablen oder Namensräumen innerhalb des übrigen Codes zutreffen. Ein Beispiel hierfür ist "% %Container::get%(...)", was auf alle Methoden zutrifft, die mit „get“ beginnen, eine beliebige Parameterliste und beliebigen Rückgabetyp besitzen und innerhalb einer Klasse stehen, deren Name mit „Container“ endet. Die Menge der zutreffenden Stellen, sog. Joinpoints, kann durch zusätzliche Bedingungen weiter eingeschränkt werden. Sollen im vorangegangenen Beispiel nur von einer Basisklasse namens „Base“ abgeleitete Klassen berücksichtigt werden, kann folgender Ausdruck verwendet werden:

```
"% %Container::get%(...)" && derived("Base")
```

Advice-Code

Der einzuwebende Code wird über Advices definiert. Beispielsweise führt der folgende Advice-Code dazu, dass bei der Ausführung der zuvor ausgewählten Methoden zunächst eine Ausgabe „Hallo Welt“ erfolgt:

```
advice execution("% %Container::get%(...)" && derived("Base")) :
    before() {
        cout << "Hallo Welt" << endl;
}
```

Neben dem Schlüsselwort `before()`, das den Advice-Code *vor* dem Code der Methode einwebt, können `after()` zum Einsetzen *nach* oder `around()` zum *Ersetzen* der Methode verwendet werden.

slice class

Ein weiterer Mechanismus ist `slice class` zum Erweitern von Klassen um Methoden, Attribute, etc. bis hin zu zusätzlichen Basisklassen. Soll z. B. jede Klasse, deren Name mit „Container“ endet, mit einem Zähler ausgestattet werden, der beim Aufruf der bereits oben erweiterten „get“-Methoden inkrementiert und mit ausgegeben wird, so kann folgender Ausdruck aus Quelltext 2.1 verwendet werden.

Natürlich ist dies nur ein sehr kompakter Einblick in die Möglichkeiten, die AspectC++ bietet, zeigt aber deutlich die Vorteile, die sich daraus ergeben: Funktionalitäten, die sich nicht auf eine Klasse beschränken lassen, sondern eine Modifikation an vielen verschiedenen Stellen des gesamten Systems erfordern würden, lassen sich mit Aspekten in einer übersichtlichen Einheit zusammenfassen. Sind die Erweiterungen nicht mehr gewünscht, wird der Aspekt beim Übersetzen ausgelassen. Auf diese Weise können Systeme nach dem „Baukastenprinzip“ erstellt und konfiguriert werden. So auch die Idee von Kratos.

Baukastenprinzip

```
slice class CountExtension {
private:
  int counter;
};

aspect GetLogger {
  advice "%Container" : slice CountExtension;
  advice execution("%%Container::get%(...)" && derived("Base"))
      : before() {
    cout << "Hallo Welt " << counter++ << endl;
  };
};
```

Quelltext 2.1: Beispiel für `slice class` in AspectC++ zum Erweitern von Klassen.

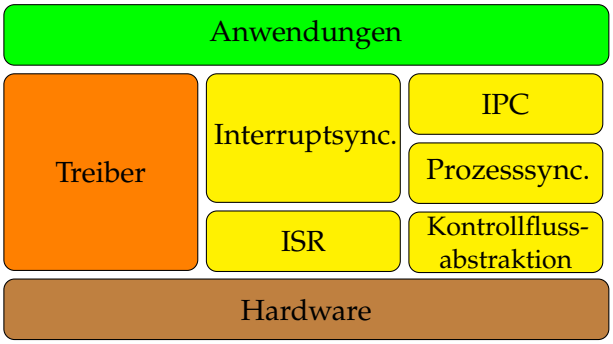


Abbildung 2.11: Schichtenstruktur von Kratos in Anlehnung an [Tec14a].

2.5.2 *Struktur*

Strukturell ist Kratos in Schichten organisiert (Abbildung 2.11), die sich wiederum aus mehreren Klassen zusammensetzen. Unmittelbar auf der Hardware aufbauend bietet Kratos Datenstrukturen und Methoden zum Einhängen von ISR an. Die darüber befindliche Schicht stellt eine geordnete Abarbeitung von Unterbrechungen sicher, da diese auch auftreten können, wenn eine ISR noch nicht abgeschlossen ist. Aus diesem Grund werden in Kratos die ISR in zwei Teile – den unmittelbar mit gesperrten Unterbrechungen abzuarbeitenden *Prolog* und den eventuell verzögerten *Epilog* – unterteilt.

Prolog, Epilog

Ebenfalls unmittelbar auf der Hardware arbeitend, ermöglicht die Kontrollflussabstraktion eine Unterbrechung einer aktuellen Routine und die (Wieder-)Aufnahme eines anderen Kontexts. Dies geschieht durch Sicherung der aktuellen Registerwerte einschließlich des Stapelzeigerregister, engl. ***S**tack **P**ointer* (SP) auf den Stack der aktiven Routine und einer Wiederherstellung der Werte aus dem Zielkontext.

Darauf aufbauend synchronisiert ein preemptiver zeitscheibenbasierter Scheduler die angemeldeten Prozesse. Er wird in Abständen

preemptiver Scheduler

von 10 ms durch eine Unterbrechung nach Ablauf eines Timers angestoßen und wechselt den Kontext auf den nächsten Prozess, der bereit ist.

Semaphoren, Wecker

Die IPC-Schicht verwaltet schließlich wartende bzw. schlafende Prozesse und stellt Wartemechanismen wie Semaphoren mit `wait()`- und `signal()`-Methoden, sowie Wecker (`Buzzer`) mit `sleep()`-Methoden bereit.

stapelbare Treiber

Die Ebene der Treiber zieht eine direkte Brücke zwischen der Hardware und den Anwendungen. Dabei können die einzelnen Treiberklassen einerseits direkt auf der Hardware arbeiten oder auf anderen Treibern aufbauen. Beispielsweise greift ein Treiber für ein Display, das über die SPI-Schnittstelle am Mikrocontroller angeschlossen ist, durch den SPI-Treiber auf das Display zu.

Zuletzt befinden sich auf der höchsten Ebene die vom Scheduler verwalteten Anwendungen, die die eigentlichen Aufgaben des eingebetteten Systems erfüllen. Sie werden als Klassen implementiert und verwenden die vom System bereitgestellten Schnittstellen, wie der nachfolgende Abschnitt anhand einer Beispielanwendung demonstriert.

2.5.3 *Erweiterbarkeit*

Da in Kratos die einzelnen Komponenten als Klassen oder Aspekte implementiert werden, erfolgt das Hinzufügen neuer, bzw. die Erweiterung existierender Module in gleicher Weise. Zur Veranschaulichung wird in den folgenden beiden Abschnitten begleitend der Einbau einer neuen Anwendung vorgeführt. In Quelltext 2.2 findet sich eine kompakte Kratos-Anwendung, die als Spezialisierung der Klasse `Thread` und implementierter virtueller Methode `action()` realisiert wird. Zeilen 20 und 26 zeigen den Zugriff auf existierende Systemkomponenten durch einfaches Hinzufügen der `.h`-Datei und Erzeugen eines entsprechenden Objekts. Komponenten, die Systemweit nur eine Objektinstanz benötigen, werden durch eine `extern`-Deklaration bekannt gemacht und in der `.cc`-Datei außerhalb einer Klasse instanziiert.

Prozesse erben von Thread

Umgekehrt erfordert der Einbau einer neuen Komponente häufig, dass sie auch von anderen Teilen des Betriebssystems verwendet wird. Dies lässt sich zwar durch Entwurfsmuster wie Observer realisieren, erzeugen in eingebetteten Systemen jedoch einen zu großen Overhead beim Verwalten der Instanzen. Insbesondere ist die Dynamik eines solchen Entwurfsmusters zur Laufzeit des Systems nicht notwendig, wenn die eingehängten Objekte bereits zur Übersetzungszeit bekannt sind und sich zur Laufzeit keine Änderungen ergeben.

Overhead durch Observer

Im Beispiel der Anwendung aus Quelltext 2.2 ist der Thread zwar Einsatzbereit und verwendet externe Komponenten; wird jedoch bisher an keiner Stelle im System bekanntgemacht oder verwendet. Um

```
/*********SleepingThread.h***********/
#ifndef SLEEPINGTHREAD_H
#define SLEEPINGTHREAD_H

#include "syscall/thread.h"

class SleepingThread : public Thread {

public:
    SleepingThread(void *tos) : Thread(tos) {}
    void action();
};

extern SleepingThread sleepingThread;

#endif // SLEEPINGTHREAD_H

/*********SleepingThread.cc***********/
#include "SleepingThread.h"
#include "syscall/guarded_buzzer.h"
#include "config.h"

/* Create thread instance */
DeclareThread(SleepingThread, sleepingThread, 256);

void SleepingThread::action() {
    Guarded_Buzzer buzz;

    buzz.set(CONFIG_SLEEP_DURATION_MS);
    while(1)
        buzz.sleep();
}
```

Quelltext 2.2: Beispielanwendung für Kratos. Anwendungen werden als Spezialisierung der Klasse `Thread` mit einer `action()`-Methode implementiert. Die Beispielanwendung erzeugt eine Instanz der Systemkomponente `Buzzer`, dessen `sleep()`-Methode periodisch aufgerufen wird. Das Schlafintervall wird durch ein Makro in der Datei `config.h` definiert.

```
#ifndef SLEEPINGTHREAD_AH
#define SLEEPINGTHREAD_AH

#include "SleepingThread.h"
#include "syscall/guarded_scheduler.h"

aspect StartSleepingThread {
  advice execution("void ready_threads()") : after() {
    organizer.Scheduler::ready(sleepingThread);
  }
};

#endif // SLEEPINGTHREAD_AH
```

Quelltext 2.3: Aspekt zum Einhängen der Anwendung aus Quelltext 2.2 in den Scheduler, damit der Thread gestartet werden kann.

den Thread zu starten, genügt es nicht, eine Instanz zu erzeugen. Vielmehr muss der Thread zusätzlich beim Scheduler angemeldet werden der seinerseits die `action()`-Methode startet, unterbricht und fortsetzt.

statischer Observer

Anstatt nun Änderungen am System vorzunehmen und den Thread manuell in die Liste des Schedulers einzutragen, wird dieser Belang als Aspekt implementiert (Quelltext 2.3). Dessen Pointcut richtet sich an die `ready_threads()`-Methode des Systems und erweitert diese um das Einhängen der neuen Anwendung. Dieser Mechanismus ist ein Schlüsselpunkt für die einfache Konfigurierbarkeit des Betriebssystems.

2.5.4 *Konfigurierbarkeit*

graphische Konfiguration

Die Konfiguration von Kratos geschieht bequem mit dem Programm kconfig-qconf und ist in Abbildung 2.12 dargestellt. Über die graphische Oberfläche können Komponenten (Features) ausgewählt und ihren Parametern feste Werte zugewiesen werden. Auf der linken Seite finden sich alle verfügbaren Features wie Anwendungen, Peripheriekomponenten und ihre Treiber oder die Zielarchitektur (hier Typ des Mikrocontrollers). Rechts finden sich die Parametrisierungsmöglichkeiten einer Komponente. Dort können Werte wie Taktrate, die verwendeten Anschlusspins oder, wie im Fall der Beispielanwendung aus Quelltext 2.2, das Schlafintervall festgelegt werden.

Konfigurationsparameter

Beim Speichern einer Konfiguration wird eine Liste der zugehörigen Quelldateien erzeugt. Durch die Verwendung von Aspekten werden die einzelnen Komponenten automatisch in das System eingewoben. Zusätzlich wird die Datei `config.h` generiert, die die gesetzten Parameterwerte als Präprozessormakros definiert, und so im Quelltext der Komponenten verfügbar macht.

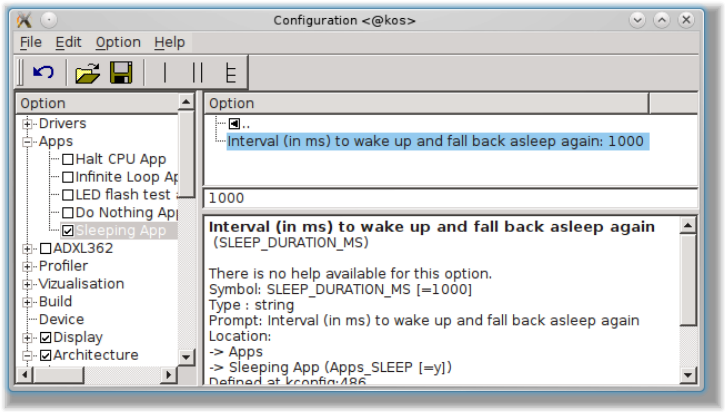


Abbildung 2.12: Konfigurationsoberfläche für Kratos mit dem Programm kconfig-qconf. Die gewünschten Features können bequem über die Oberfläche ausgewählt und ihre Parameter zugewiesen werden. Beim Speichern wird eine Liste der benötigten Quelldateien sowie eine Header-Datei mit Makros zu den gesetzten Parameter erzeugt, sodass das System übersetzt werden kann.

Da die Features und Parameter nicht automatisch aus dem vorhandenen Quelltext gewonnen werden, muss zu jeder Komponente, die konfigurierbar sein soll, eine `.feature`-Datei im XML-Format angegeben werden. Der Inhalt der zur Beispielanwendung gehörenden Datei ist in Quelltext 2.4 aufgelistet. Sie definiert neben der Position im Baum (Abbildung 2.12 links) auch den Namen und die Beschreibung der Komponente. Zudem werden Abhängigkeiten und Konflikte zu anderen Komponenten im Baum, sowie die zur Komponente gehörenden Dateien angegeben. Schließlich werden hier alle Parameter mit Typ, optionalem Standardwert und ihren Bezeichnungen sowohl in der graphischen Oberfläche, als auch in der Datei `config.h` festgelegt.

Beim Aufruf der Konfigurationsoberfläche wird zunächst mit speziellen Buildscripten die gesamte Verzeichnisstruktur von Kratos nach `.feature`-Dateien durchsucht, die Fragmente zusammengefügt und in die kconfig-Sprache umwandelt. Anschließend werden die Inhalte mit kconfig-qconf dargestellt und die Konfiguration kann durchgeführt werden.

feature-Datei

```
<feature name="/Apps/SLEEP">
  <display>
    <prompt>Sleeping App</prompt>
    <help>Thread which sleeps all the time an wakes up in
        specified intervals</help>
  </display>
  <dependencies>
    <dependsOn>/Architecture/MSP430FR</dependsOn>
  </dependencies>
  <in>
    <file>SleepingThread.h</file>
    <file>SleepingThread.cc</file>
    <file>SleepingThread.ah</file>
  </in>
  <variables>
    <string name="SLEEP_DURATION_MS" default="1000">
      <display>
        <prompt>Interval (in ms) to wake up and fall back
            asleep again</prompt>
      </display>
    </string>
  </variables>
</feature>
```

Quelltext 2.4: `.feature`-Datei im XML-Format für die Beispielanwendung aus Quelltext 2.2. In der Datei werden Abhängigkeiten von anderen Komponenten, Parameter und die zum Feature zugehörigen Übersetzungseinheiten angegeben.

3

VERWANDTE ARBEITEN

Die Modellierung vom Peripherieverhalten bzw. ihrem Energiebedarf, insbesondere für eingebettete Systeme, ist bis dato Gegenstand der Forschung. Sie ist teilweise aus unterschiedlichen Zielsetzungen motiviert. Einerseits wird das Verhalten der Geräte für eine automatische Treibersynthese in einer Beschreibungssprache formuliert. Andererseits beschäftigen sich Forscher mit der Aufstellung von Simulationsmodellen zu den einzelnen Komponenten (bzw. eines zusammengesetzten Gesamtsystems), um den Ressourcenverbrauch (Energie, Speicherverbrauch, etc.) bereits in der Entwurfsphase des Systems zu optimieren. Auch existieren Ansätze, die sowohl Simulationen und außenstehende Analysen ermöglichen, sowie dem System zusätzlich zur Laufzeit Informationen über den Energieverbrauch der Peripherie bereitstellen. Dieses Kapitel stellt verschiedene Arbeiten aus den drei Bereichen vor, die als Grundlagen und Ideenquellen für den Modellentwurf dieser Arbeit dienen und präsentiert, wie sich die Arbeit von den vorhandenen Ansätzen absetzt.

Ideenquellen, Abgrenzung

3.1 AUTOMATISCHE TREIBERSYNTHESE

Die automatische Synthese von Gerätetreibern wird aus verschiedenen Gesichtspunkten motiviert. Zunächst ist die Entwicklung von Treibern ein langwieriger und fehleranfälliger Entwicklungsprozess. Insbesondere in sicherheitskritischen Systemen, z. B. im Automobil oder Flugzeug, werden hohe Anforderungen an die Gerätetreiber (wie auch an die übrige Software) gestellt, da fehlerhafte Treiber zu katastrophalen Folgen führen können und dadurch Leib und Leben gefährden. Aber auch in weniger kritischen Anwendungsfällen sind korrekt arbeitende Treiber für einen effektiven Einsatz der Systeme unabdingbar. Versetzt ein Gerätetreiber beispielsweise in einem energiebeschränkten verteilten Sensornetz eine „stromhungrige“ Komponente nicht korrekt in den Energiesparmodus, entlädt sich die Batterie des Sensorknotens und er fällt vorzeitig aus. Doch sogar wenn jeder Treiber für sich korrekt implementiert ist, können Wechselwirkungen durch geteilte Ressourcen (z. B. gemeinsame Busse) den Energieverbrauch des Systems unnötig erhöhen. Mit dieser Problematik befasst sich die Arbeit von [BMM13].

sicherheitskritische Treiber

Durch den modularen Aufbau von Betriebssystemkomponenten und die Existenz mehrerer Abstraktionsebenen verteilt sich die Information über den aktuellen Systemzustand auf viele Teile des Systems. Jeder Teil kann nur basierend auf den lokal verfügbaren Infor-

verteilter Systemzustand

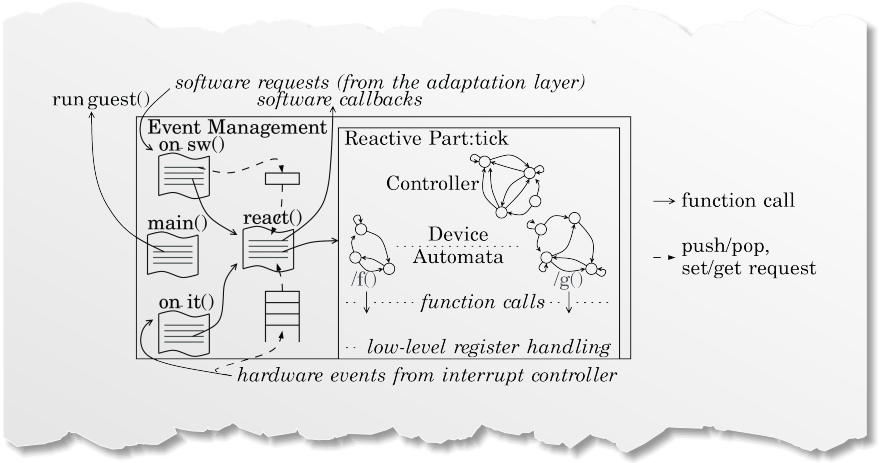


Abbildung 3.1: Auszug aus [BMM13]. Alternative Treiberschicht eines eingebetteten Betriebssystems, die energieintensive Wartezustände vermeiden soll. Gerätetreiber werden als Automaten modelliert, die ihren Bedarf geteilter Ressourcen an einen Controller melden (rechts). Ist eine Ressource belegt, werden die Transitionen abgelehnt oder verzögert. Die Synchronität wird durch Sequenzialisierung aller Soft- und Hardwareanfragen an die Treiberschicht sichergestellt (links).

mationen über die Art und Weise des Zugriffs auf andere Komponenten entscheiden. Es fehlt jedoch die globale Sicht, die zu besseren Entscheidungen führen könnte. Ziel der Arbeit von [BMM13] ist das vermeiden von leistungsintensiven Wartezuständen, die sich beispielsweise daraus ergeben können, dass ein Funkchip für eine bevorstehende Übertragung hochgefahren wird, der Transfer wegen eines blockierten Busses durch einen anderen Prozess verzögert wird.

Zur Lösung des Problems modellieren die Autoren die Gerätetreiber als Bool'sche Mealy Automaten, engl. *Boolean Mealy Automata* (BMA), deren Transitionen mit Bool'schen Ausdrücken annotiert sind, sodass Zustandsübergange nur beim Erfüllen der jeweiligen Bedingungen gestattet sind. Eingaben dieser Automaten sind Ereignisse aus der Soft- und Hardwareschicht des Systems, Ausgaben sind wiederum Ereignisse und C-Instruktionen. Durch die Bool'schen Ausdrücke sind die Automaten kontrollierbar, sodass Zustandsübergänge gezielt verhindert werden können. Ein zusätzlicher Automat dient als *Controller*, der global den Zustand geteilter Ressourcen wiederspiegelt, indem Bool'sche Variablen ereignisbasiert umgesetzt werden, z. B. ob ein Bus momentan belegt ist. Die Automaten werden in einer synchronen Sprache (Lustre) beschrieben und bei der Treibersynthese zu einer gemeinsamen C-Datei übersetzt.

BMA

globaler Controller

Treiber und Controller werden werden in einer eigenen Betriebssystemschicht gekapselt, wie der Auszug aus dem Artikel in Abbildung 3.1 verdeutlicht. Darin ist zu erkennen, dass alle treiberbezogenen

Ereignisse, seien es Softwareanfragen aus übergeordneten Ebenen (oben) oder Interrupts aus der Hardwareschicht (unten), zunächst in korrespondierenden Warteschlangen gesammelt und sequenziell abgearbeitet werden (links). Dabei arbeitet die Methode `react()` ein Ereignis aus einer der Warteschlangen ab und stößt Zustandsübergänge im Automatenmodell (rechts) an. Die Transitionen – sofern sie durchgeführt werden können – rufen korrespondierende Methoden auf, die auf niedrigster Ebene auf die Hardwarekomponente zugreifen und diese umkonfigurieren. Ist die Transition nicht gestattet (durch Nichterfüllung der Transitionsbedingung), wird der Übergang abgelehnt und die übergeordnete Schicht benachrichtigt, oder die Transition verzögert. Auf diese Weise kann erzwungen werden, dass Peripheriekomponenten im Energiesparmodus verbleiben, wenn die zur Bearbeitung notwendigen Ressourcen momentan nicht zur Verfügung stehen.

verzögerte Transitionen

Nicht auf den Energieverbrauch, dafür auf die Korrektheit und zusätzlich auf die Codegröße und Wiederverwendbarkeit von Treibern fokussiert sich die Arbeit von [TBB+13]. Auch hier sollen Treiber automatisch synthetisiert werden, jedoch aus der formalen Spezifikation sowohl des Gerätes, als auch der Anwendung, die das Gerät verwenden soll. Beide Modelle werden zusammengeführt und anschließend zu einem Programmquelltext übersetzt. Die getrennte Modellierung von Gerät und Anwendung bringt dabei viele Vorteile. Ein klassischer Gerätetreiber beinhaltet aus Wiederverwendbarkeitszwecken häufig mehr Funktionalitäten, als tatsächlich in einem konkreten Einsatzszenario benötigt wird. Dadurch können ungenutzte Codepassagen (sog. *dead code*) im übersetzten Programm vorkommen und vergrößern so unnötiger Weise den Speicherplatzverbrauch. Durch Einbezug der Anwendung auf Modellebene können nur die tatsächlich relevanten Teile der Gerätefunktionen – selbst im Falle eines vollständigen und allgemeinen Funktionsmodells – beim Übersetzen mit berücksichtigt werden.

automatische Synthese

Ein weiterer Punkt ergibt sich aus Verwendungsfehlern. Denn sogar ein fehlerfreier Gerätetreiber kann bei falscher Anwendung zu unvorhersehbarem Verhalten führen. Sind hingegen Gerätebeschreibung und Anwendung in einem formalen Modell gegeben und schließen beide durch Zusatzbedingungen eine Fehlbedienung aus, so können die Modelle nur erfolgreich zusammengeführt werden, wenn alle erforderlichen Bedingungen erfüllt sind. Ansonsten scheitert die Synthese.

vermeidet Verwendungsfehler

Zur Modellierung des Gerätes und der Anwendung verwenden die Autoren **G**uarded **l**abeled **t**ransition **s**ystems (GLTS). Dabei handelt es sich prinzipiell um Automaten, deren Transitionen – wie in der zuvor genannten Arbeit – mit Bedingungen annotiert sind und so Zustandswechsel sperren oder freigeben können. Die Beschreibung des Gerätes richtet sich dicht am Datenblatt des Herstellers und mo-

GLTS

delliert in den Automatenzuständen die Menge der Konfigurationseigenschaften. Bei den Transitionen handelt es sich entweder um kontrollierbare Zustandswechsel durch gezielte Ansteuerung des Gerätes, oder um nicht kontrollierbare Übergänge, ausgelöst durch Interrupts. Die Anwendung wird zunächst als Menge von Arbeitsmodi aufgefasst, wobei jeder Modus mit einer Menge von Konfigurationseigenschafen aus der Gerätebeschreibung assoziiert ist, die der jeweilige Modus erfordert. Diese Beschreibung wird dann automatisch in ein GLTS transformiert.

Zusammenführung mit Spieltheorie

Schließlich werden beide Beschreibungen mit Hilfe der Spieltheorie zu einem Gesamtsystem zusammengeführt. Ziel des „Spiels“ ist die Entwicklung einer Gewinnstrategie für jeden Anwendungsmodus die gewünschte Konfiguration des Geräts zu erreichen. Dabei kann die Anwendung nur die kontrollierbaren Zustandswechsel im Treiber anstoßen, während die Hardware als „Gegner“ die nicht kontrollierbaren Wechsel durchführt. Eine Gewinnstrategie ergibt sich daraus, dass der gewünschte Zustand sicher erreicht wird, unabhängig aller Aktionen, die die Hardware durchführt. Aus diesen Strategien soll schließlich der endgültige Programmcode generiert werden. Die Autoren behalten sich diesen Punkt jedoch für künftige Forschungsarbeiten vor.

Neben den beiden vorgestellten Arbeiten existieren noch viele weitere Ansätze zur automatischen Treibersynthese, wie u. A. [CGBRP11, WM03], die allesamt eines gemeinsam haben: Die Geräte werden, unabhängig von der gewählten Beschreibungssprache und Modellierungstiefe, als Automaten oder vergleichbare Transitionssysteme modelliert. Daher liegt die Schlussfolgerung nahe, dass sich Gerätetreiber vollständig durch Automaten beschreiben lassen und somit auch direkt in dieser Form implementiert werden können.

Beschreibung als Automat

3.2 RESSOURCEN SIMULATIONSMODELLE

Um den Ressourcenverbrauch von eingebetteten Systemen abzuschätzen oder zu optimieren, können in der Entwurfsphase Simulationen durchgeführt werden. Diese bilden das System teilweise oder vollständig in einem Modell nach und können automatisiert hohe Anzahlen an Testversuchen, mit jeweils unterschiedlichen Parametrisierungen durchführen. Die Granularität solcher Simulationen ist nicht fest vorgegeben, sondern wird für jeden Einsatzzweck passend gewählt.

Instruktionsebene

Zur Simulation des Energieverbrauchs von einem Mikroprozessor auf Instruktionsebene stellen [SKWM01] ein sehr feingranulares Modell vor, das sogar die Ladungsveränderungen auf Bus-Leitungen (vgl. Kapitel 2.1) zwischen zwei benachbarten Prozessorinstruktionen berücksichtigt und anhand der Hamming-Distanz parametrisiert. Ziel dieses Modells ist eine optimierte Anordnung von Instruktionen beim Übersetzungsprozess aus einer Hochsprache.

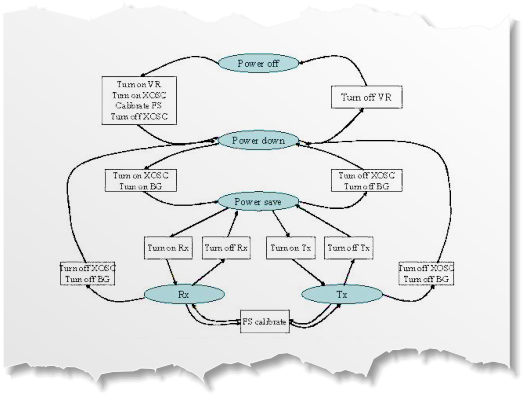


Abbildung 3.2: Originalgraphik aus [WY07]. Die logischen Energiezustände und Transitionen eines Funkchips werden in einem Simulator als FSM modelliert und jeder Zustand sowie jede Transition mit real gemessenen Kosten annotiert.

Andere Simulationsmodelle zielen hingegen auf die logische Sicht einzelner Systemkomponenten. In [WY07] wird ein automatenbasiertes Energiemodell für das Kommunikations-Subsystem eines Sensorknotens aufgestellt. Dazu werden die logischen Betriebszustände des Funkchips als Zustände einer FSM modelliert und die möglichen Zustandsübergänge als Kanten hinzugefügt (siehe Auszug aus dem Artikel in Abbildung 3.2). Durch Messung des Stromverbrauchs in den jeweiligen Zuständen (sowie bei den Zustandsübergängen) werden Zustände und Kanten mit entsprechenden Kosten annotiert. Dadurch werden höheren Schichten im Protokollstapel genauere Kriterien für energieeffizientes Schlaf-Scheduling zur Verfügung gestellt. Eine Simulation des Kommunikationsprotokolls kann damit optimale Parameter hinsichtlich minimaler Energieaufnahme bestimmen.

Schlaf-Scheduling

Auf gleiche Weise wird auch in [Wed10] ein Energiemodell für einen Funkchip (nRF24L01) aufgestellt. Die annotierte FSM wird in das bekannte Simulationsframework OMNeT++ implementiert, was anschließend die Originalsoftware[1] eines Sensorknotens ausführt. Dadurch kann die Firmware in sehr kurzen Entwicklungs- und Testschritten optimiert werden. Anzumerken ist, dass die Autoren beider Artikel bei der Untersuchung der Stromaufnahme von den jeweiligen Funkmodulen Abweichungen zum Datenblatt festgestellt haben. Aus diesem Grund ist eine Praxismessung für ein realistisches Energiemodell essenziell.

Abweichung vom Datenblatt

In [TRJ02] wird noch weiter von der konkreten Hardware abstrahiert und der Energiebedarf für die Ausführung von Systemdiensten erfasst. Auf dieser Ebene wird daher nicht nur der (hinter Diensten) versteckte Verbrauch der Peripherie erfasst, sondern auch die Kosten für Kontextwechsel, ISR, etc.. Die Bestimmung dieser Kosten

1 Sowohl die Firmware als auch OMNeT++ sind in der Programmiersprache C++ implementiert, sodass lediglich eine Adapterschicht erforderlich ist.

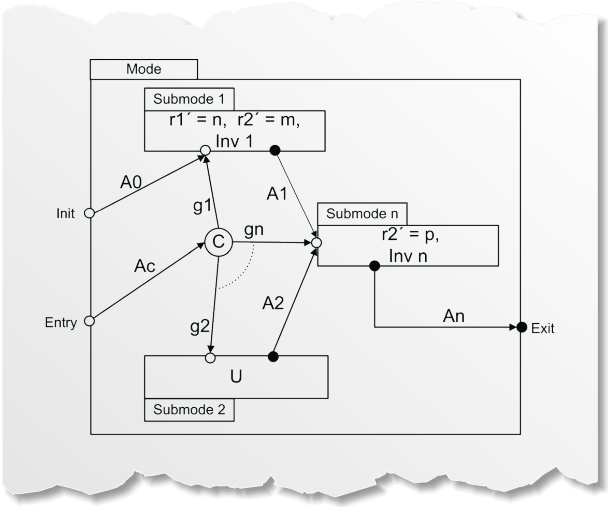


Abbildung 3.3: Auszug aus [SVP09]. In der Beschreibungssprache REMES werden Systemkomponenten in Form von Betriebsmodi in einem Transitionssystem beschrieben. Jeder (Sub-) Modus ist mit dem individuellen Ressourcenverbrauch (bzw. einer Verbrauchsrate) annotiert. Zustandsübergänge werden durch Erfüllen der annotierten Guard-Bedingung ausgelöst.

erfolgt durch wiederholte Aufrufe einzelner Systemdienste mit paralleler Messung und einer anschließenden Auswertung des Energie-Histogramms. Die Systemdienste werden mit allen tiefer liegenden Schichten bei der Messung als Blackbox betrachtet. Da allerdings stets das Gesamtsystem in einer festen Konfiguration vermessen wird eignet sich diese Abstraktionsebene nicht für die Konstruktion wiederverwendbarer Modelle.

Ressourcenmodell

Während sich die zuvor genannten Simulationsansätze ausschließlich auf den Energieverbrauch konzentrieren, wird in [SVP09] ein allgemeines Ressourcenmodell für verschiedenste Arten von Systemressourcen vorgestellt. Dies können neben Energie auch CPU-Last, Speicherverbrauch (ROM, RAM), IO-Ports oder Busse sein. Ziel sind Analysen der Machbarkeit und Optimierungen von Ressourcenanforderungen. Die Modellierung erfolgt in einer eigenen Beschreibungssprache (REMES), die das Verhalten einzelner Systemkomponenten in Form von Betriebsmodi wiederspiegelt. Zur Verdeutlichung ist ein Auszug aus der Veröffentlichung in Abbildung 3.3 dargestellt. Darin ist zu erkennen, dass ein Modus einem Transitionssystem mit definierten Einstiegs- (Init, Entry) und Ausstiegspunkten (Exit) entspricht, Unterzustände beinhalten kann und bedingte Zustandsübergänge ($A_0, \ldots, A_n$) besitzt. Die Transitionen $A=(g, S)$ beinhalten einen Guard g und eine Menge von Anweisungen S (wie das Setzen von Variableninhalten), die ausgeführt werden, sobald der Bool'sche Ausdruck in g wahr wird und die Transition durchgeführt wird. Zudem können konditionelle Verzweigungen (C mit $g_1, \ldots, g_n$) und Invarian-

ten (Inv 1, Inv n) für eine Mindestverweilzeit in einem Modus formuliert werden. In jedem (Sub-) Modus kann der individuelle Konsum beliebiger Ressourcen (r1, r2) angegeben werden, entweder als Direktwert oder als zeitabhängige Rate.

Die REMES Modelle werden systematisch (nach einem festen Schema) in PTA (siehe Kapitel 2.2) überführt, sodass mit etablierten Algorithmen und Werkzeugen Analysen durchgeführt werden können. Machbarkeitsanalysen entsprechen hierbei einem Erreichbarkeitsproblem bei gegebenen Ressourcen. Je nachdem, ob ein Zielzustand überhaupt erreicht werden kann oder in allen Fällen garantiert erreicht wird, spricht man von *weak feasibility* respektive *live feasibility*. Zur Bestimmung des optimalen und maximalen Ressourcenbedarfs kann das Werkzeug UPPAAL CORA verwendet werden.

Machbarkeitsanalyse

Bei der Betrachtung der vorgestellten Arbeiten ist auffällig, dass Ansätze, die die Systemkomponenten auf der Ebene logischer Betriebszustände betrachten, Transitionssysteme wie PTA o. ä. zur Modellierung verwenden. Aufgrund der einfachen formalen Struktur lassen sich sowohl schnelle Simulationen, als auch algorithmische Analysen durchführen. Allerdings beschäftigen sich die Ansätze ausschließlich mit externen (offline) Modellen, die als dedizierte Einheiten – unabhängig vom eigentlichen System – parallel mitentwickelt werden. Ändert sich die Implementierung des tatsächlichen Gerätetreibers, muss das Modell ggf. angepasst werden, und vice versa. Dieser Gedanke führt zusammen mit den Erkenntnissen aus der automatischen Treibersynthese zu den kombinierten Ansätzen.

parallele Mitentwicklung

3.3 KOMBINIERTE ANSÄTZE

Kompakte Energiesimulationsmodelle führen zu der Idee, diese Modelle nicht nur für eine Offline-Analyse zu verwenden, sondern direkt in das Zielsystem zu integrieren. Zwar muss das Modell ggf. vereinfacht werden, um ein ressourcenbeschränktes eingebettetes System nicht zu überfordern, allerdings erlaubt die so geschaffene online-Verfügbarkeit Entscheidungen anhand der tatsächlichen Energiesituation des Systems zu treffen. So motivieren die Autoren von [KPMB08], dass verteilte Datenbanksysteme wie TinyDB anhand der Energiesituation einzelner Sensorknoten ihre Abfragen hinsichtlich einer besseren Lastverteilung[2] optimieren können. Sie integrieren zusätzlich zum klassischen Gerätetreiber einen Zustandsautomaten, der dem Simulationsmodell des Gerätes entspricht und zur Laufzeit synchron durchlaufen wird. Dies geschieht durch manuelles Einpflegen der Transitionsaufrufe an korrespondierenden Stellen im Treiber. Im Simulationsmodell müssen jedoch durch Interrupts hervorgerufene Zustandsübergänge durch Ablauf einer vorgegebenen Zeitspanne ausgelöst werden. Der Simulator hat sonst keine Möglichkeit, den Zeit-

Lastausgleich

2 Lastverteilung im Sinne eines ausgewogenen Energieverbrauchs

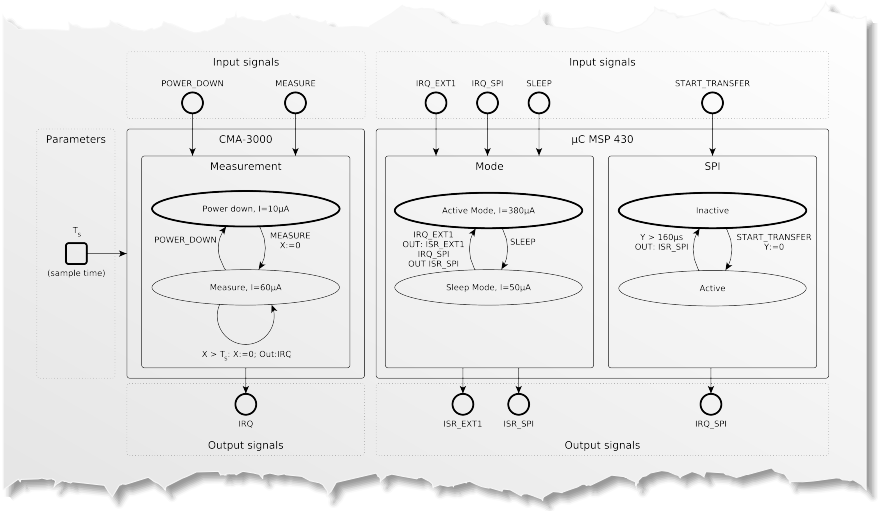


Abbildung 3.4: Aus [BGS12] entnommene Graphik. In dem Simulationsmodell werden die Betriebsmodi von Systemkomponenten als parametrisierte PTA beschrieben. Transitionen werden durch definierte Eingangssignale ausgelöst, die wiederum selbst Signale erzeugen können. Die Parametrisierung von Energie-/Leistungsaufnahme und Zeitschranken reduziert die Anzahl notwendiger Zustände. Für ein online-Modell werden durch Simulation szenariobasiert mittlere Verbrauchswerte bestimmt und in das System eingepflegt.

Zeitl. Abschätzung von Unterbrechungen

punkt eines Hardwareereignisses festzustellen. Online wird dieser Parameter ignoriert, da die Transition innerhalb einer ISR ausgelöst werden kann. Um die Anzahl der Zustände gering zu halten, können die annotierten Kosten parametrisiert werden, z. B. in Abhängigkeit des RSSI eines Funkmoduls, sodass nicht für jede mögliche Sendeleistungsstufe ein gesonderter Zustand existieren muss.

In [BGS12] wird die Reduktion des online-Modells weitergeführt. Dabei wird zunächst wie im letzten Ansatz ein OMNeT++ Simulationsmodell für die Betriebsmodi der Hardware in Form von PTA aufgestellt, deren Kosten und zeitabhängige Transitionen parametrisiert werden können. Zur Verdeutlichung ist ein Modell aus der Veröffentlichung in Abbildung 3.4 dargestellt. Es zeigt die komponentenweisen Betriebsmodi eines kleinen eingebetteten Systems, das Beschleunigungswerte mit unterschiedlicher Abtastrate erfasst. Die einzelnen Automaten beschreiben jeweils den Beschleunigungssensor (CMA-3000), Mikrocontroller (MSP430) und seine SPI-Schnittstelle. In der Simulation werden die Komponenten durch definierte Eingangssignale konfiguriert, was zu Transitionen im Modell führt. Die Transitionen können wiederum selbst Signale erzeugen und andere Zustandübergänge anstoßen.

kompaktes online-Modell

Zur Kompaktifizierung des online-Modells werden Simulationen für verschiedene grobgranulare Anwendungsszenarien durchgeführt

und die Leistungsaufnahme während dieser Szenarien gemittelt. Am Beispiel der dargestellten Abbildung können ausgewählte Abtastraten des Beschleunigungssensors und damit die Arbeitsgeschwindigkeit des Gesamtsystems als solche Szenarien angesehen werden. Die berechneten Mittelwerte werden in das System integriert und das aktuelle Anwendungsszenario zur Laufzeit mitgeführt. Durch eine *power-aware* API kann dann im laufenden System der aktuelle Energieverbrauch abgefragt werden.

Beide Ansätze zeigen Methoden zum Erzeugen von dicht verflochtenen offline- und online-Energiemodellen für eingebettete Systeme. Das Mittel zur abstrakten Beschreibung sind wiederholt PTA, wie auch die reinen Synthese- bzw. Simulationsmodelle aufzeigen. Allerdings stehen die online-Modelle – unabhängig von ihrer Granularität – als „Zusatzkomponenten“ neben den klassischen Gerätetreibern. Die Interaktion mit dieser Komponente muss manuell in den Treiber an den *passenden* Stellen eingepflegt werden. Die *passenden* Stellen sind allerdings insbesonders bei umfangreichen Gerätetreibern über ihren gesamten Quelltext verteilt und können leicht übersehen oder bei Modifikationen vergessen werden. Solche Fehler sind schwer zu entdecken, da eine vergessene oder falsche Transition zu keinem unmittelbar erkennbaren Fehlverhalten des Systems führen muss. Ein ungenau erfasster Energieverbrauch bleibt insbesondere bei der initialen Entwicklung unentdeckt, wenn Erfahrungswerte zu der Komponente noch fehlen.

Fehleranfälligkeit

Im Rahmen dieser Arbeit wird daher ein stärker integriertes Energiemodell entwickelt, das auch diese Problematik minimiert. Basierend auf PTA wird das Modell nicht als separate Einheit parallel zum Betrieb mitgeführt, sondern vollständig durch den Treiber realisiert. Die Treiberschnittstelle wird auf das Anstoßen von Transitionen (unter Angabe erforderlicher Parameter) reduziert und der Treiber selbst als Zustandsautomat implementiert. Der genaue Entwurf inklusive seiner Anforderungen wird im folgenden Kapitel beschrieben.

reduzierte Schnittstelle

4 ENTWURF

Dieses Kapitel behandelt den Entwurf des energiegewahren Treibermodells. Angetrieben durch Erkenntnisse aus verwandten Arbeiten wird hierzu ein automatenbasierter Ansatz für die Gestaltung des Treibers und seiner Schnittstelle gewählt. Im ersten Teil werden die Ziele für das Modell formuliert, sowie die Gründe herausgestellt, weswegen die klassische Implementierungsart von Treibern das Erreichen dieser Ziele erschwert. Ferner wird die Idee der automatenbasierten Treiberschicht als Lösungsansatz vorgestellt und die genauen Anforderungen zusammengefasst.

Anschließend behandeln die weiteren Abschnitte den konkreten Entwurf dieses Ansatzes am Beispiel des Betriebssystems Kratos. Dies umfasst neben der Entwicklung des eigentlichen Treibermodells auch Zusatzkomponenten, die für die Erstellung solcher Treiber erforderlich sind. Anhand von UML Klassendiagrammen wird das „Baukastenprinzip“ dieses hochgradig konfigurierbaren und erweiterbaren Entwurfs erläutert.

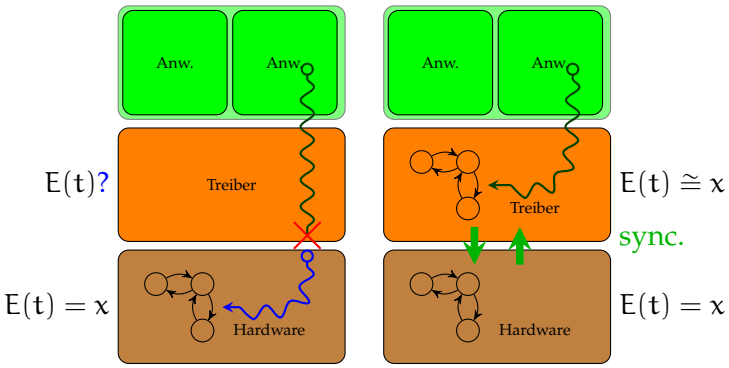
Im letzten Teil wird schließlich der typische Entwicklungsweg für automatenbasierte Treiber nach diesem Modell erklärt und die konkreten Einsatzstellen der Zusatzwerkzeuge aufgezeigt.

4.1 Ziel, Idee und Anforderungen

Ein Ziel dieser Arbeit ist die Bereitstellung vom Energieverbrauch einzelner Hardwarekomponenten eines eingebetteten Systems für das laufende Betriebssystem und die ausgeführten Prozesse. Die Software soll jederzeit Zugriff auf den aktuellen Betriebszustand, die aktuelle Leistungsaufnahme und den bisherigen Energieverbrauch einzelner Peripheriekomponenten erhalten. Letztere Punkte sollen auch für jeden beliebigen Betriebszustand (bzw. für den Übergang in diesen Modus) verfügbar sein. Wie bereits in der Einleitung erläutert, ist die Integration dieser Funktionaliät so nah wie möglich an der verursachenden Hardware – und somit im Gerätetreiber – anzusiedeln. Eine klassische Treiberschicht mit aufgabenorientierter Schnittstelle bietet zum Betriebsmodus und Energieverbrauch im Allgemeinen keine Informationen. Sie abstrahiert von den inneren Abläufen der Hardware und stellt einfache Methoden zur Verwendung der Komponenten bereit. Diese Abstraktion verwischt allerdings auch den verbundenen Energieverbrauch beim Aufruf der Funktionen. Daher ist das Treibermodell so umzugestalten, dass diese Beziehung wieder eindeutig ist.

Zugriff auf Betriebszustand

Abstraktion verschleiert Energieverbrauch



(a) Klassischer Treiber (b) Automatenbasierter Treiber

Abbildung 4.1: Im klassischen Treibermodell (links) reißt der Kontrollfluss beim Anstoßen der Hardware ab. Treiberfunktion erlauben keinen Rückschluss auf den Energieverbrauch. Rechts bildet ein automatenbasierter Treiber die Energiezustände der Hardware synchron nach. Dadurch kann der Verbrauch an den Softwareautomaten annotiert werden. Zudem ermöglicht der Automat eine Vervollständigung des Kontrollflusses.

Als weiteres Ziel soll der Entwurf zusätzlich zur Übersetzungszeit eine (genauere) statische Analyse hinsichtlich des Energieverbrauchs ermöglichen. Neben der unklaren Beziehung zwischen Treiberaufruf und Energieverbrauch liegt die größte Problematik beim Kontrollfluss. Dieser geht zwar aus dem Quelltext hervor und lässt sich vom der Anwendung bis hinab zum Aufruf einer Treiberfunktion verfolgen. Wird die Kontrolle jedoch an die Hardware abgegeben und mittels Semaphoren auf eine Rückmeldung via Interrupt gewartet, taucht die Ausführung der ISR nicht im Aufrufgraphen der Treiberfunktion auf (Abbildung 4.1a). Stattdessen wird der Kontrollfluss hinter dem blockierenden Semaphorenzugriff fortgesetzt; Wartezeit und Seiteneffekte auf den Energieverbrauch bleiben allerdings verschleiert.

abreißender Kontrollfluss

Im typischen Anwendungsfall lässt sich dieser Abriss vom Kontrollfluss korrigieren. Die Wartezeit und der damit verbundene Energieverbrauch sind häufig bekannt oder unter Angabe von Parametern berechenbar. Beispielsweise kann bei einer interruptbasierten Datenübertragung die Zeit zum Abschluss des Transfers aus der Datenmenge und Übertragungsrate abgeleitet werden. Das Treibermodell soll deswegen für diese Fälle die nötigen Informationen bereithalten, um den Kontrollfluss an der richtigen Stelle fortsetzen zu können.

Die aufgezeigten Ziele erfordern die Integration von Zusatzinformationen in den Gerätetreiber. Da klassische Treiber bereits zu stark von der Hardware – im Sinne des Energieverbrauchs – abstrahieren, soll im Folgenden die Treiberschicht von Kratos (als Repräsentant für ein eingebettetes Betriebssystem) grundlegend umgestaltet werden.

Umgestaltung der Treiberschicht

Inspiriert von der automatenbasierten Modellierung von Hardware, bzw. ihres Ressourcenbedarfs, wird nun der Ansatz eines automatenbasierten Treibermodells verfolgt. Die grundlegende Idee ist das synchrone Abbilden der Peripheriefunktionalität in Software (Abbildung 4.1b). Hierzu soll das etablierte Modell der PTA verwendet werden (siehe Kapitel 2.2 und 3), da es Möglichkeiten zur Kostenannotation (für Energie und Leistung) bietet und ereignis- wie auch zeitbasierte Zustandsübergänge beinhaltet. Letzteres ist insbesondere zur deterministischen Abbildung von Interrupts notwendig.

Kostenannotation

Der Umfang des Softwaremodells beschränkt sich allerdings auf eine Nachbildung der Leistungsaufnahme und keiner „Simulation" aller internen Abläufe, wie sie durch eine VHDL-Beschreibung der Komponente erfasst werden würde. Dies ist zum einen nicht notwendig und würde zum anderen das System mit der Nachbildung der Peripherie auslasten. Stattdessen soll der Entwurf einen ressourcenschonenden Ansatz zur Realisierung der Ziele bieten.

schonender Ressourcenumgang

Die eigentliche Funktionalität des Treibers wird in ereignisbasierten Zustandsübergängen realisiert. Sie sind – ebenso wie die Zustände – mit Kosten annotiert und verknüpfen somit den Treiberaufruf mit einem Energieverbrauch. Durch das Automatenmodell verändert sich die Semantik der Treiberschnittstelle weg von aufgabenorientierten Funktionsaufrufen hin zum einheitlichen Anstoßen von Zustandsübergängen in der Hardware. Mit der Schnittstellenerweiterung zur Abfrage des Ressourcenbedarfs von jeder Transition und jedes Zustands kann, zusammen mit dem aktuellen Gerätezustand, auch die momentane Leistungsaufnahme ausgegeben werden.

Semantik der Schnittstelle

Um die erforderliche Annotation durchführen zu können, müssen die Werte messtechnisch erfasst werden. Dazu muss jede Transition und jeder Zustand besucht werden und ihre Energie bzw. Leistungsaufnahme gemessen werden. Im Entwicklungsprozess wird der Treiber zunächst ohne Annotation implementiert und die Peripheriekomponente unter Verwendung dieses Treibers vermessen. Wird bei der Messung festgestellt, dass sich die Leistungsaufnahme nicht getreu des Softwaremodells verhält, muss die Implementierung angepasst und die Messung wiederholt werden.

Einschränkend wird im Rahmen des hier entworfenen Treibermodells eine konstante Betriebsspannung während der Entwicklungs- und späteren Einsatzzeit eines Treibers angenommen. Prinzipiell kann das Modell jedoch auch für unterschiedliche Betriebsspannungen erweitert werden. Der Energieverbrauch kann dazu anhand einer Parametrisierung zur Übersetzungszeit berechnet werden und erzeugt somit auch keinen zusätzlichen Aufwand im laufenden System. Allerdings erfordert die Parametrisierung des Energieverbrauchs deutlich umfangreichere Messungen und Verifikationen.

parametrisierte Betriebsspannung

Fasst man die zuvor genannten Ziele gemeinsam mit Anforderungen, die für den Entwurf jedes eingebetteten Systems gelten, zu einer

Liste zusammen, so ergeben sich folgende Anforderungen an den konkreten Entwurf:

- Nachbildung der Hardware-Leistungszustände im Treiber.
- PowerAPI zum Auslesen des aktuellen Zustands, sowie der annotierten Leistungsaufnahme bzw. des Energieverbrauchs von Zuständen respektive Transitionen.
- Rekonstruierbarer Kontrollfluss bei Interrupts.
- Erleichterte Vermessung der Hardware.
- Wiederverwendbarkeit.
- Austauschbarkeit.
- Konfigurierbarkeit.
- Erweiterbarkeit.
- Minimaler Overhead.
- Einfache Implementierbarkeit.
- Einfache Bedienbarkeit.
- Konfliktfrei zu weiteren klassischen Treibern im System.
- Minimalinvasiv bezüglich der Integration ins Betriebssystem.
- Problemlos stapelbar.

4.2 REALISIERUNG DER AUTOMATEN

Nachdem im letzten Abschnitt begründet wurde, dass zur Integration eines Energiemodells die Treiberschicht eines Betriebssystems so umgestaltet werden kann, dass einzelne Treiber als Zustandsautomaten implementiert werden, soll dieser Ansatz nun anhand eines Entwurfs für das eingebettete Betriebssystem Kratos realisiert werden.

Dazu wird das theoretisches Modell der PTA (Siehe Kapitel 2.2) zugrundegelegt und die Treiber nach diesem Konzept implementiert. Sie werden im Folgenden als DFA-Treiber (deterministischer endlicher Automat, engl. ***D**eterministic **F**inite **A**utomaton* (DFA)) bezeichnet.

ohne virtuelle Methoden

Beim Entwurf wird vollständig auf virtuelle Methoden, Funktionszeiger und Templates verzichtet. Bei Funktionszeigern und virtuelle Methoden sind die Speicheradressen des Aufrufs zur Übersetzungszeit noch nicht eindeutig festgelegt, sodass ihr Einsatz eine zuverlässige statische Analyse verhindern würden. Virtuelle Methoden erzeugen außerdem einen Laufzeitverlust durch Nachschlagen einer Implementierung in der Tabelle virtueller Methoden, engl. *virtual method **table*** (VTABLE) des Objekts.

ohne Templates

Der Verzicht auf Templates liegt hingegen darin begründet, dass Treiber und Schnittstelle einfache Erweiterungsmöglichkeiten für das Einhängen von Aspekten anbieten sollen, der Aspektweber jedoch im aktuellen Entwicklungsstand das Weben von Templates nicht unterstützt und mögliche Joinpoints stillschweigend ausgelassen werden [Spi14].

4.2.1 *Schnittstelle*

Die Treiberschnittstelle zur Ansteuerung der Peripherie beschränkt sich bei dem gewählten Automatenmodell auf das Anstoßen von Transitionen. Zur Veranschaulichung ist die Schnittstelle eines Beispieltreibers als Klassendiagramm in Abbildung 4.2 dargestellt und cyan hervorgehoben. Der Anstoß geschieht einheitlich über die Methode `passTransition(...)`, die für jede Transition mit einem Subtypen von `transition_t` überladen ist. Diese Subtypen repräsentieren die Menge der Transitionen E im Automaten und kapseln die zur Durchführung einer Transition erforderlichen Daten und Parameter, sowie Referenzen auf Speicherplatz für mögliche Rückgabewerte. Zusätzlich beinhalten sie den Energieverbrauch der Transition als öffentliches Attribut `transitionenergy` und bilden dadurch eine Teilmenge der Kosten P ab.

Transitionen kapseln Energieverbrauch

Um dem Betriebssystem den Status und die Leistungsaufnahme der Komponente verfügbar zu machen, beinhaltet die Schnittstelle zusätzliche Methoden zur Abfrage des aktuellen Zustands, der aktuellen Leistungsaufnahme, sowie der Leistungsaufnahme jedes beliebigen Zustands (Siehe Abbildung 4.2).

4.2.2 *Interne Struktur*

Intern wird der Treiber als Spezialisierung der Basisklasse `DFA_Driver` umgesetzt und speichert seinen aktuellen Zustand im Attribut `state`. Die Menge der Zustände L wird dabei durch ein `enum` realisiert, das alle Arbeitsmodi namentlich aufzählt. Schließlich beinhaltet das Feld `statepower[]` die Leistungaufnahme jedes Zustands an der zum `enum` korrespondierenden Stelle und vervollständigt somit die Menge der Kosten P.

Interaktion nur via passTransition()

Die funktionsbezogene Schnittstelle des Treibers wird ausschließlich durch die Methoden `passTransition(...)` abgedeckt. Bei der Implementierung ist jedoch darauf zu achten, dass `state` korrekt aktualisiert wird. Aus Effizienzgründen wird keine Kontrolle durchgeführt, ob eine angestoßene Transition im aktuellen Zustand zulässig ist. Wie im klassischen Treibermodell obliegt es der Verantwortung des Programmierers, den Treiber in korrekter Weise zu verwenden. Die Überprüfung kann jedoch bei Bedarf mit speziellen Aspekten und der `around()` Anweisung eingewoben werden.

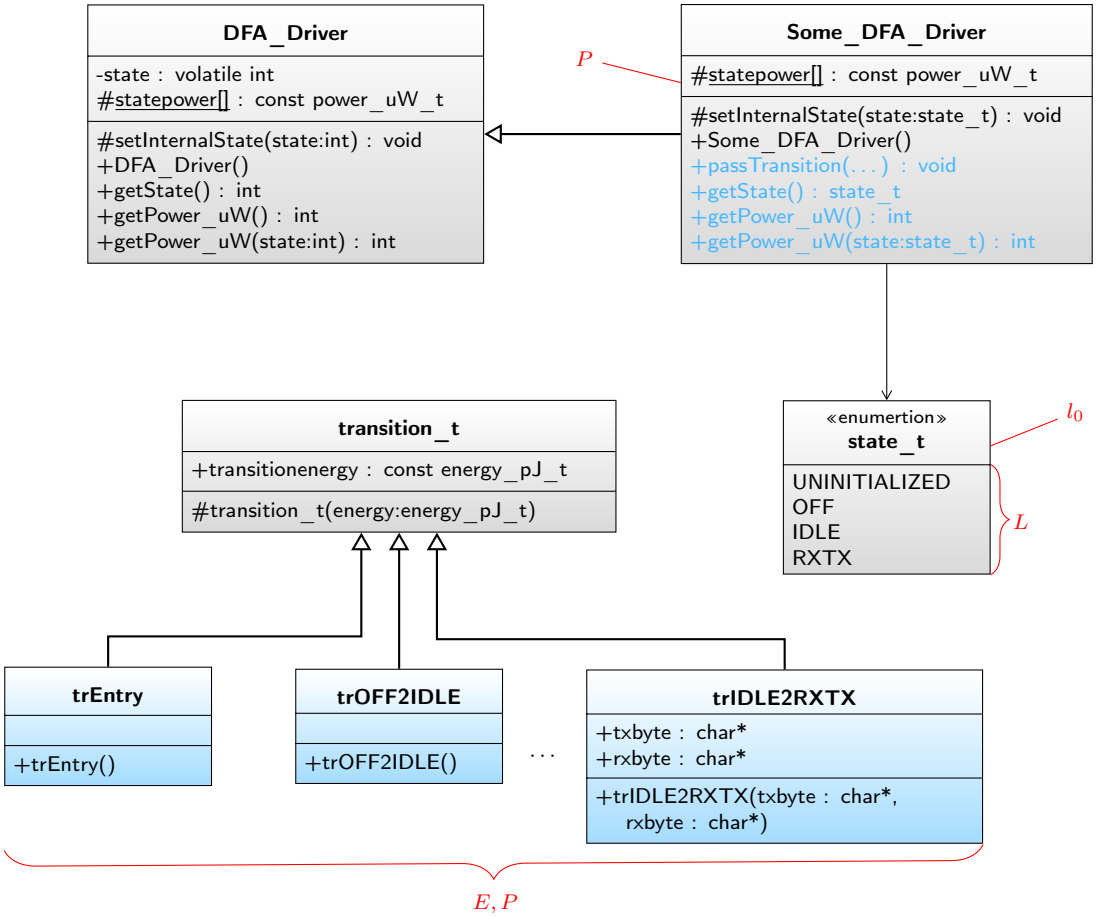


Abbildung 4.2: UML Klassendiagramm des DFA-Treibermodells zur Umsetzung von PTA in Kratos. Die Zustandsmenge wird in einem `enum` aufgelistet, die Transitionsmenge durch Spezialisierungen von `transition_t` repräsentiert. Die Schnittstelle beschränkt sich auf `passTransition(...)` und Statusabfragen für Zustand und Leistungaufnahme der Komponente.

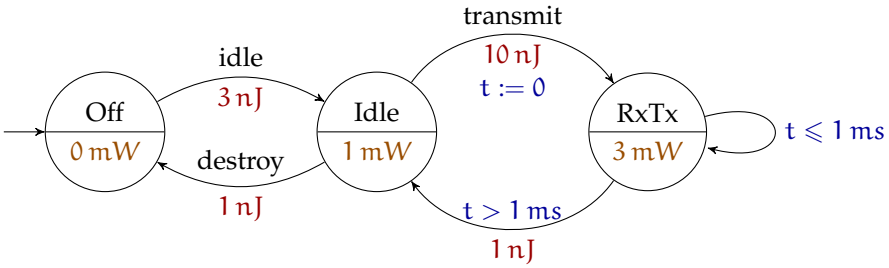


Abbildung 4.3: PTA einer fiktiven Kommunikationsschnittstelle. Zeitabhängige Transitionen manifestieren sich im Betriebssystem als explizite Zustandsübergänge, signalisiert durch Interrupts. Die zeitliche Abschätzung ist jedoch für die statische Analyse notwendig.

keine weiteren Einschränkungen

Die übrige Implementierung geschieht in gewohnter Weise durch Verwendung von zusätzlichen Attributen, Unterprogrammen, Systemkomponenten, Hardwarezugriffen oder weiteren Treibern.

4.2.3 *Zeitabhängige Transitionen*

In einem PTA können Transitionen durch äußere Ereignisse oder Erfüllen einer zeitlichen Bedingung ausgelöst werden. Im Treibermodell entspricht das äußere Ereignis offenkundig einem expliziten Anstoß einer Transition (durch Aufruf von `passTransition(...)`). Zeitbedingte Transitionen manifestieren sich bei dieser Modellierung allerdings auch als explizite Transitionen, wie folgendes Beispiel zeigt:

Betrachtet man die in den Grundlagen behandelte fiktive Kommunikationsschnittstelle (nochmals in Abbildung 4.3 gezeigt), verbleibt der Automat für 1 ms im Zustand *RxTx* und kehrt nach Abschluss der Übertragung in den Zustand *Idle* zurück. Diesen Übergang signalisiert die Schnittstelle dem System üblicherweise durch das Auslösen eines Interrupts. Somit kann der Zustand durch Aufruf von `passTransition(...)` in der ISR aktualisiert werden. Der aus abstrakter Sicht eindeutig zeitabhängige Übergang ist im laufenden Betriebssystem ein expliziter Methodenaufruf, ausgelöst durch eine Unterbrechung. Lediglich für die Codeanalyse wird eine zeitliche Angabe benötigt, da die ISR im Aufrufgraphen nicht auftaucht. Diese Verbindung kann jedoch mit Hilfe von außenstehenden Annotationen vervollständigt werden, wie in Kapiteln 5.4 und 6.3 noch gezeigt wird.

zeitabhängige Übergänge

Der Entwurf ermöglicht mit Unterstützung des Betriebssystems allerdings auch die Modellierung von zeitabhängigen Transitionen ohne Signalisierung, d. h. ohne Interrupt, wie es beispielsweise zum Modellieren von Tailstates erforderlich ist (siehe Kapitel 3 zu [PHZ+11]). Hierzu kann ein Hilfsthread gestartet werden, der die vorgegebene Verweilzeit schläft und anschließend die zeitabhängige Transiti-

Hilfsprozess

on auslöst. Wird in der Zwischenzeit eine andere Transition ausgelöst, kann der Thread abgebrochen und ggf. neu gestartet werden. Für die Codeanalyse unterscheiden sich solche Übergänge nicht von Interrupt-basierten Transitionen und werden gleichermaßen annotiert.

4.3 ZUSÄTZLICHE KOMPONENTEN

Bisher wurde ein automatenbasiertes Treibermodell entworfen, das den Energieverbrauch der getriebenen Komponenten im System verfügbar machen soll. Die Energie bzw. Leistungswerte sind allerdings fest in der Software eingebunden und müssen daher im Verlauf der Treiberentwicklung erst ermittelt werden. In diesem Abschnitt werden deswegen die entworfenen Zusatzkomponenten behandelt, die zur Implementierung und Evaluation von DFA-Treibern unverzichtbar sind.

Zunächst wird auf die Signalisierungskomponente (Trigger) eingegangen, die einem externen Messgerät Zustandsübergänge mitteilt und so die Akquirierung der notwendigen Verbrauchswerte ermöglicht. Im Anschluss wird der Energy Profiler behandelt, der im laufenden System den anfallenden Energieverbrauch aller gewünschten Komponenten erfasst. Er dient in erste Line zur Evaluation der Modelltreue der Treiber, kann allerdings auch vom Betriebssystem für Szenarien wie die Abschätzung der verbleibenden Batterielaufzeit verwendet werden.

aspektorientiert

Die beiden Zusatzkomponenten sind aspektorientiert entworfen und können in konfigurierbarem Umfang in das System eingewoben, bzw. vollständig ausgelassen werden.

4.3.1 *Trigger*

Signalisierung

Um den Energiebedarf einer Transition oder die Leistungsaufnahme eines Zustands zu bestimmen, ist eine Signalisierung (Trigger) an das Messgerät notwendig. Nur auf diese Weise kann die Messaufzeichnung in Abschnitte unterteilt und den einzelnen Ereignissen zugeordnet werden. Die Messungen können dadurch automatisiert über einen längeren Zeitraum durchgeführt und gemittelt werden, um zuverlässigere Resultate zu erhalten. In der frühen Entwicklungsphase ermöglicht die Markierung der Ereignisse eine Evaluation bezüglich einer hinreichend repräsentativen Modellierung des Treibers. Beispielsweise können Abschnitte ohne Zustandswechsel auf Leistungsschwankungen untersucht werden, die ggf. zusätzliche Zustände oder Transitionen erfordern.

Die Signalisierungsfunktion (`PinTrigger`) ist aspektorientiert entworfen, sodass sie sich auf Wunsch in einen beliebigen DFA-Treiber einweben und konfigurieren lässt. Abbildung 4.4 zeigt im UML Klassendiagramm die verwendete Struktur. Die Klasse `PinTrigger` (rechts)

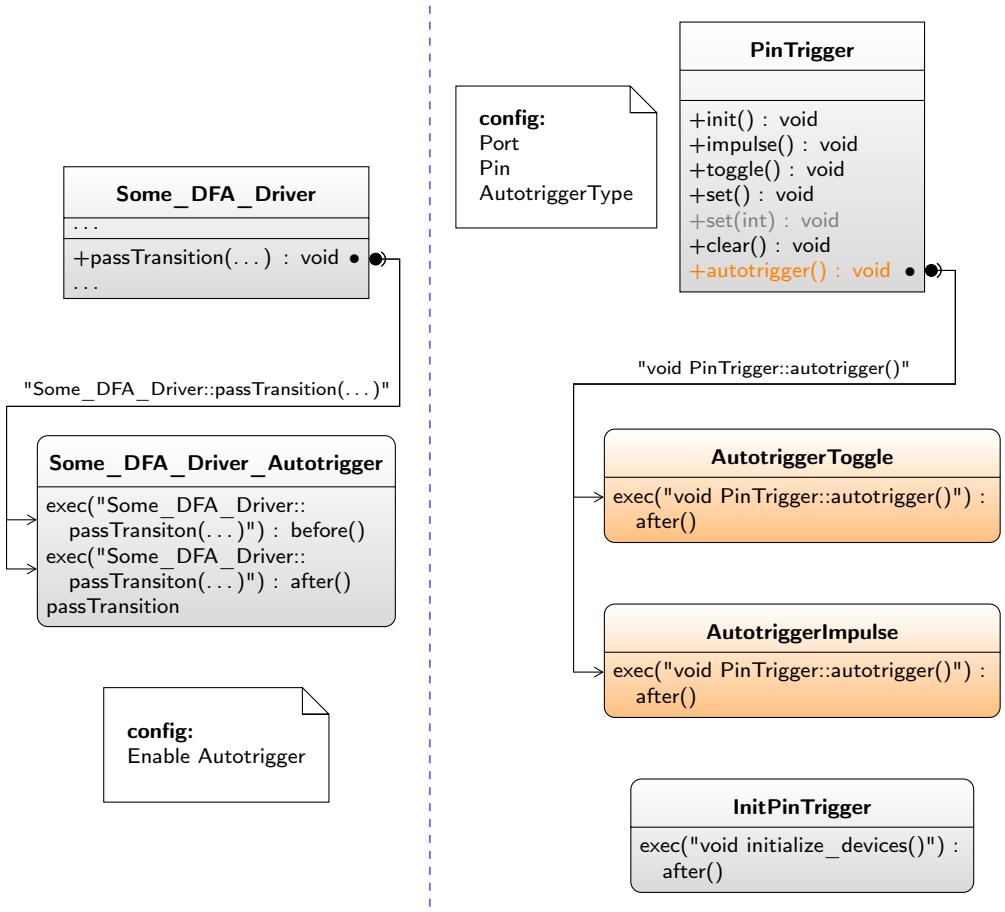


Abbildung 4.4: UML Klassendiagramm der Triggerkomponenten zum Signalisieren von Zustandsübergängen an die Messtechnik. Über einen Aspekt werden zu Beginn und Ende von Transitionen Aufrufe von `autotrigger()` eingewoben, die den Ausgangspegel eines IO-Ports verändern. Die gewünschte Signalisierungsart wird durch Auswahl einer der orange markierten Aspkete festgelegt.

implementiert die Signalisierung über einen IO-Port, dessen Ausgabewert über die bereitgestellten Methoden `set()`, `clear()`, etc. beeinflusst werden kann. Durch die grau eingefärbte Methode `set(value)` kann der Zustand oder die Transition in binärer Kodierung ausgegeben werden. Da MIMOSA im aktuellen Entwicklungsstand lediglich über einen 1 Bit Triggereingang verfügt, beschränkt sich die Implementierung auf das Steuern von einem Bit durch die übrigen Methoden.

Beschränkung auf 1 Bit

Zur Integration des Triggers in das Betriebssystem bzw. den Treiber muss zunächst die Klasse `PinTrigger` zusammen mit dem Aspekt `InitPinTrigger` beim Übersetzen des Betriebssystems berücksichtigt werden. Letzteres stößt die Initialisierung des gewünschten IO-Ports beim Systemstart an. Damit stehen die Methoden für die manuelle Verwendungen (insbesondere für Experimentierzwecke in der Entwicklungsphase von Treibern) zur Verfügung. Um jedoch auch automatisch bei Transitionen eine Signalisierung durchzuführen, wird beim Übersetzungsprozess zusätzlich ein zum Treiber gehörender Aspekt (hier `Some_DFA_Driver_Autotrigger`) einbezogen. Durch die einheitlich benannte Treiberschnittstelle kann mit sehr einfachen *pointcut-expressions* das Auslösen der Methode `autotrigger()` zu Beginn und Ende einer Transition eingewoben werden. Ebenso simpel gestaltet sich die Spezialisierung dieses Ausdrucks, wenn beispielsweise nur eine Teilmenge der Transitionen bei der Signalisierung berücksichtigt werden soll.

automatische Signalisierung

Bei der Konfiguration des Triggers kann neben des verwendeten IO-Ports bzw. Pins auch die Implementierung von `autotrigger()` ausgewählt werden. Durch Einbinden eines der in Abbildung 4.4 orange markierten Aspekte wird die Funktionalität durch Aufruf einer der übrigen Methoden von `PinTrigger` realisiert. Natürlich können auf diese Weise bei Bedarf beliebig umfangreiche Implementierungen eingehängt werden.

Autotrigger

Sobald die Implementierung eines Treibers und die Akquisition der Messdaten abgeschlossen sind, können die Triggerkomponenten rückstandsfrei aus dem System entfernt werden.

4.3.2 *Energy Profiler*

Zum softwarebasierten Aufzeichnen und Verfolgen des Energieverbrauchs im laufenden System wurde ein flexibler Energy Profiler entwickelt. In Anlehnung an die Namensgebung von AOStuBS und Kratos wird er als Stromzähler (`ElectricityMeter`) benannt. Er bildet für die Evaluation ein zentrales Werkzeug bei der Untersuchung der Modellgüte.

Stromzähler

Die Struktur ist in Abbildung 4.5 dargestellt. Cyan gekennzeichnet sind der DFA-Treiber, sowie der Aspekt `*_Energybilling`, der – an-

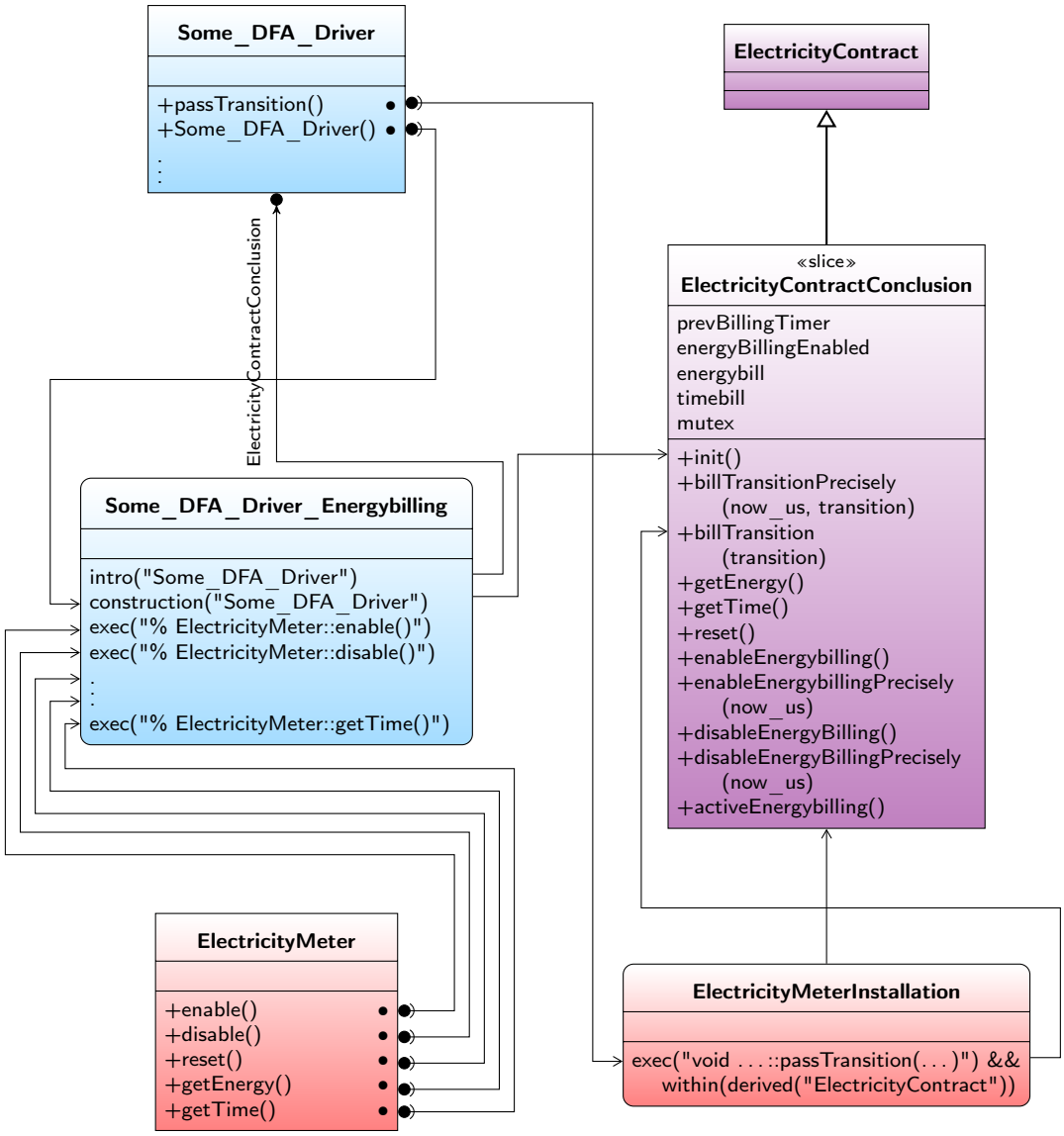


Abbildung 4.5: UML Klassendiagramm des Stromzählers zur Erfassung des Energieverbrauchs im laufenden System. Die rot und violett gekennzeichneten Komponenten sind einmalig im System vorhanden und implementieren die erforderliche Funktionalität. Die cyan markierten Teile können unter verschiedenen Namen mehrfach im System vorkommen. Dabei fügt der Aspekt `*_Energybilling` (cyan) auf Wunsch die Zusatzfunktionen (violett) zum Treiber hinzu und veranlasst eine Anmeldung am Stromzähler (rot).

schaulich ausgedrückt – für die Peripheriekomponente einen Stromversorgungsvertrag abschließt und den Stromzähler verbindet.

Der Vertragsschluss (`ElectricityContractConclusion` in violett) fasst zusätzliche Attribute und Methoden, die für die Abrechnung der Energie erforderlich sind, in einem *slice* zusammen. Dieser wird durch den Aspekt `*_Energybilling` zum Treiber hinzugefügt. Die Basisklasse `ElectricityContract` markiert den Treiber dabei als „Vertragskunden“, dessen Verbrauch vom Stromzähler erfasst werden soll.

Vertragskunden

Die rot gekennzeichneten Komponenten gehören zum zentralen Stromzähler mit der Klasse `ElectricityMeter` als Einstiegspunkt. Sie bildet die Schnittstelle zur Steuerung der Verbrauchsmessung (aktivieren, deaktivieren, zurücksetzen) und zur Ausgabe des Energieverbrauchs aller angemeldeten Komponenten. Initial handelt es sich bei den Methoden lediglich um definierte Joinpoints, an denen mit dem Aspekt `*_Energybilling` korrespondiere Methodenaufrufe aus dem *slice* `ElectricityContract` eingewoben werden. Beispielsweise wird dem Rückgabewert der Methode `ElectricityMeter::getEnergy()` das Resultat vom eingewobenen `Some_DFA_Driver::getEnergy()` hinzuaddiert. Mit diesem Mechanismus können beliebig viele Treiber gleichzeitig in die Stromzählung mit aufgenommen werden. Die Markierung als „Vertragskunden“ ermöglicht die Formulierung einer überschaubaren und zuverlässigen pointcut-expression zur Selektion aller abzurechnenden Treiber. Dies wird vom ebenfalls rot gekennzeichneten Aspekt `ElectricityMeterInstallation` genutzt und bewirkt, dass in alle Klassen, die von `ElectricityContract` erben, die Abrechnung des Energieverbrauchs bei der Ausführung von `passTransition(...)` eingewoben wird.

beliebig viele Teilnehmer

Wie der Trigger, erfordert auch der Stromzähler keine permanenten Veränderungen am Betriebssystem und kann somit inklusive seiner Abhängigkeiten vollständig aus dem System entfernt werden. Hierzu müssen beim Übersetzen die Klasse `ElectricityMeter`, sowie alle `*_Energybilling`-Aspekte ausgelassen werden, sodass in diesem Kontext lediglich die unveränderten DFA-Treiber im System verbleiben.

flexibel einbaubar

4.4 ENTWICKLUNGSABLAUF

Nachdem die einzelnen Bausteine des Treibermodells vorgestellt worden sind, soll nun auf den vorgesehenen Ablauf zur Entwicklung solcher DFA-Treiber eingegangen werden.

Es handelt sich hierbei um einen interativen Entwicklungsprozess, der in Abbildung 4.6 graphisch dargestellt ist. Im ersten Schritt werden die zur Funktionalität des Geräts benötigten Transitionen und die erwarteten Energiezustände festgelegt. Dies kann unter Zuhilfenahme des zugehörigen Datenblatts erfolgen, sowie Erfahrungen aus früheren Entwicklungen mit einfließen lassen. Wie sich im nächsten Kapitel noch zeigen wird, können Energieverbrauch und Leistungs-

Energiezustände festlegen

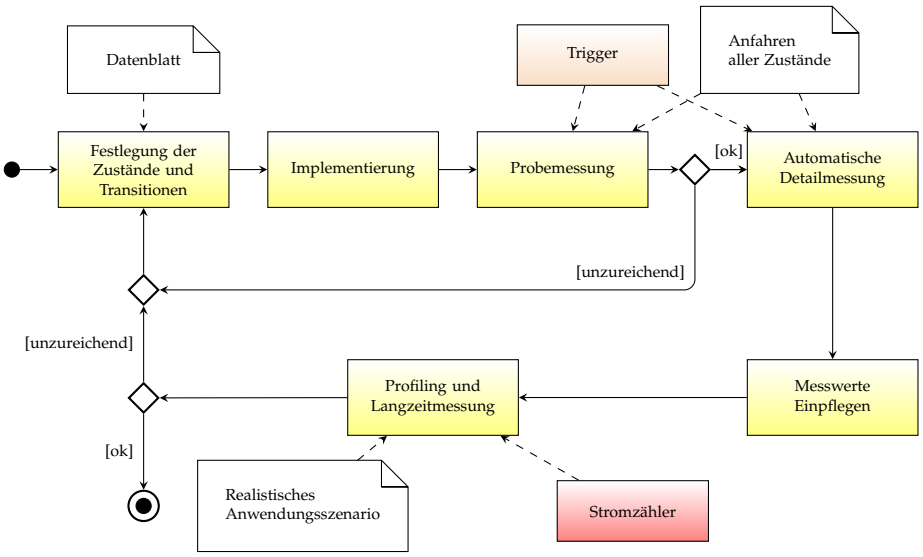


Abbildung 4.6: Entwicklungsablauf eines DFA-Treibers. Anhand des Datenblatts werden Zustände und Transitionen festgelegt und der Treiber nach dem entworfenen Modell implementiert. Probemessungen geben Anhaltspunkte darüber, inwiefern das modellierte Verhalten der Realität entspricht. Bei Abweichungen müssen Zustände und Transitionen angepasst werden. Ansonsten folgt die detaillierte Akquisition und Integration der Energie- und Leistungsaufnahme. Der gesamte Treiber wird schließlich in realistischen Anwendugsszenarien mit integriertem Profiler einer Langzeitmessung unterzogen. Je nach Resultat wird entweder die Modellierung angepasst oder die Treiberentwicklung abgeschlossen.

aufnahme im Datenblatt für den benötigen Umfang unzureichend dokumentiert sein. Daher sieht der Entwicklungsprozess im Anschluss an eine initiale Modellierung und Implementierung gemäß der zuvor entworfenen Schnittstelle, die Durchführung von Probemessungen vor. Hierbei wird der Treiber mit eingewobenem Trigger durch alle Transitionen geführt, während die Leistungsaufnahme der Peripheriekomponente messtechnisch aufgezeichnet wird. Ergeben sich in der Aufzeichnung Unterschiede zum erwarteten Verhalten – beispielsweise eine Leistungsschwankung in einem festen Zustand – so ist möglicherweise eine Erweiterung der Zustandsmenge erforderlich. Dadurch folgt eine weitere Iteration aus Implementierung und Probemessung.

Probemessungen

Ist die Implementierung augenscheinlich hinreichend genau, folgt die automatisierte Vermessung der Energie und Leistungsaufnahme. Um die Einflüsse von Rauschen und anderen Störungen auf die Messergebnisse zu minimieren, sollten alle Transitionen in häufiger Wiederholung durchlaufen und die jeweiligen Resultate gemittelt wer-

automatisierte Messung

den. Anschließend können die Ergebnisse in den Treiber eingepflegt werden.

Profiling und Langzeitmessung

Im letzten Schritt ist eine Langzeitmessung mit parallelem Profiling der Komponente vorgesehen, um die Modellgüte zu evaluieren. Dazu wird der Stromzähler in das System eingewoben und die Aufzeichnung des neuen Treibers aktiviert. Parallel zur Ausführung eines realistischen Anwendugsszenarios folgt eine Messung der Peripheriekomponente. Ergeben sich beim Vergleich von Profiling und Messung merkliche Unterschiede, muss die Modellierung des Treibers korrigiert werden. Ansonsten ist die Entwicklung abgeschlossen.

Um die einzelnen Arbeitsschritte zu unterstützen und beschleunigen, werden in Kapitel 5.4 Automatisierungswerkzeuge zur Treiberentwicklung behandelt, das wiederkehrende Aufgaben in Skripten abfertigen und dem Entwickler bei der Einhaltung des Treibermodells helfen.

5 IMPLEMENTIERUNG

Um das entworfene Treibermodell aus dem letzten Kapitel in das Betriebssystem integrieren und evaluieren zu können, müssen zunächst einige Treiber für repräsentative Geräte bzw. Komponenten implementiert werden. Dabei wird jeweils ein Treiber für ein Gerät aus folgenden (für eingebettete Systeme typischen) Kategorien entwickelt:

typische Geräteklassen in ES

integrierte peripherie Unter diese Kategorie fallen alle in einen Mikrocontroller integrierten Geräte und Schnittstellen, die als separate Komponente angesehen werden können. Dies umfasst Geräte wie z. B. Analog-Digitalwandler, Timer, wie auch SPI-, I^2C- und UART-Schnittstellen. Als Repräsentant dieser Kategorie wird die SPI-Schnittstelle nach dem entwickelten Treibermodell implementiert.

cpu Als zentrale Verarbeitungs- und Recheneinheit stellt die CPU eine eigene Kategorie dar. Auch sie trägt (wie sich später noch zeigen wird) einen nicht unerheblichen Teil zum Energieverbrauch des Gesamtsystems bei und verbraucht je nach aktuell aktivem Energiesparmodus unterschiedliche Mengen an Energie.

eingabegeräte/sensoren Sehr häufig werden eingebettete Systeme zur Perzeption ihrer Umgebung eingesetzt. Hierzu werden unterschiedlichste Sensoren genutzt, die eine physikalische Größe (z. B. Temperatur, Beschleunigung, etc.) in eine elektrische Größe (meistens eine Spannung) umwandeln und ggf. in einen digitalen Messwert überführen. Repräsentativ für diese Kategorie wird ein Treiber für den Beschleunigungssensor des InBin implementiert.

ausgabegeräte/akuatoren Komplementär zur letzten Kategorie sollen auch die erfassten Daten oder vorprogrammierte Abläufe an die Außenwelt mitgeteilt werden. Dies kann im einfachsten Falle durch Signalleuchten, jedoch auch durch verschiedenste Bildschirme oder Funkschnittstellen bis hin zum Ansteuern eines Motors erfolgen. Stellvertretend für diese Kategorie wird ein Treiber für das LCD zum Launchpad entwickelt.

Für die ausgewählten Komponenten werden die Gerätetreiber zunächst manuell gemäß des in Kapitel 4 entwickelten Treibermodells implementiert. Dabei wird auch auf spezifischen Eigenschaften der Geräte eingegangen und ihre Konsequenzen für die Erfassung und

Genauigkeit der Leistungs- bzw. Energieaufnahme begründet. Die aus der manuellen Implementierung gewonnenen Erkenntnisse werden in einem kurzen Zwischenfazit zusammengefasst und schließlich dazu verwendet, wiederkehrende Aufgaben in Skripten abzufertigen, um die Treiberentwicklung zu erleichtern und zu beschleunigen.

5.1 MANUELLE TREIBERIMPLEMENTIERUNG

Um einen Überblick über das Verhalten und den Energieverbrauch der zu modellierenden Peripheriekomponenten zu erhalten und anhand der Erfahrungen den Entwicklungsprozess automatisieren zu können, werden die Treiber zunächst manuell implementiert. Dabei wird durch wiederholte Messungen und Betrachtung des Energieverbrauchs überprüft, inwiefern das modellierte Verhalten mit der realen Leistungaufnahme korrespondiert.

Als Erstes wird ein Treiber für die interne SPI Schnittstelle des Mikrocontrollers implementiert, um die generelle Realisierbarkeit zu überprüfen. Im Anschluss wird das Modell auf die Energiesparmodi des Mikrocontrollers angewandt. Schließlich werden Gerätetreiber für externe Peripheriekomponenten, den Beschleunigungssensor des InBin und das Display des Launchpads, implementiert und analysiert.

5.1.1 *Integrierte Peripherie: SPI*

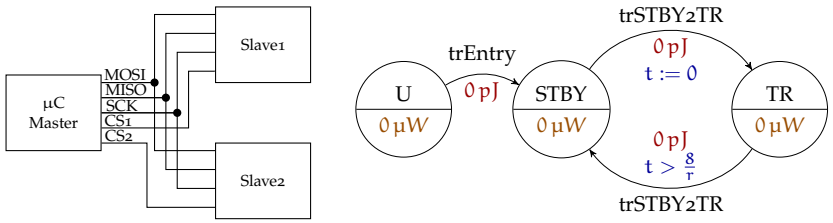
Die integrierte SPI Schnittstelle ist Bestandteil eines EUSCI innerhalb des MSP430 FR5969 (siehe Kapitel 2.4.1.5). Beim SPI handelt es sich um eine synchrone Kommunikationsschnittstelle mit einem *Master* und beliebig vielen *Slaves*. Abbildung 5.1a zeigt die Verdrahtung der Datenleitungen zwischen Teilnehmern. Die einzelnen Slaves werden vom Master über eine individuelle CS Leitung aktiviert, während die anderen Teilnehmer das Geschehen auf den Busleitungen ignorieren. Für die Datenübertragung gibt der Master auf der SCK einen Takt vor und schiebt sequenziell über ein Schieberegister ein Datenwort[1] über die MOSI Leitung zum Slave. Der ausgewählte Slave antwortet zeitgleich über die MISO Leitung und überträgt so sein Datenwort an den Master. Zur Realisierung von unidirektionalen Übertragungen wird das jeweils unnötige Datenwort ignoriert[2].

Master, Slave

Im MSP430 kann die Übertragung auf drei verschiedenen Wegen realisiert werden: Durch aktives Warten, über Interrupts oder über DMA. Beim aktiven Warten wird durch wiederholtes Abfragen eines Statusbits so lange gewartet, bis eine abgeschlossene Übertragung (Sendepuffer wieder bereit oder Datenwort vollständig empfangen)

3 Varianten zur Übertragung

1 Meistens ein Byte (8 Bit), kann jedoch konfiguriert werden.
2 Hierbei ist ggf. auf abweichende Angaben im Datenblatt einer Peripheriekomponente zu achten.



(a) SPI Busleitungen. (b) PTA zum SPI.

Abbildung 5.1: Links: Schaltplan einer SPI Anbindung mit zwei Datenleitungen (MISO, MOSI) und einer Taktleitung (SCK), sowie optionalen Selektionssignalen (CS). Rechts: PTA Modell zur SPI Schnittstelle mit den Zuständen **`UNINITIALIZED`**, **`STANDBY`** und **`TRANSFERING`**. Im ersten Entwicklungsschritt sind die Kosten noch unbekannt (deswegen auf 0 gesetzt).

signalisiert wird. Bei Interrupts kann die CPU während der Übertragung Energie sparen und wird durch das jeweils eingestellte Ereignis aufgeweckt. In beiden Fällen kann anschließend das nächste Datum übertragen werden. Für längere Datenfolgen bietet sich der DMA-Transfer (siehe Kapitel 2.4.1.3) an. Um sich bei der Modellierung ausschließlich auf die SPI Schnittstelle zu konzentrieren und den Einfluss des DMA-Controllers auszuschließen, wird ein automatenbasierter SPI Treiber auf Basis von Interrupts implementiert. *hier: Interrupts*

5.1.1.1 *Automatenmodellierung*

Gemäß des Enwicklungsablaufs aus Abbildung 4.6 werden zu Beginn die erwarteten Betriebszustände und Transitionen festgelegt. Beim SPI sind das zunächst der aktive Transfer `TRANSFERING` und ein „Bereitmodus“ `STANDBY`, der auf die nächste Übertragung wartet, währenddessen das SPI ruht. Um auch den nicht konfigurierten Zustand während des Systemstarts zu beschreiben, wird ein zusätzlicher Zustand `UNINITIALIZED` eingeführt. Es ergibt sich der in Abbildung 5.1b gezeigte Automat mit folgenden Transitionen:

ENTRY Automatischer Übergang von `UNINITIALIZED` zu `STANDBY` nach abgeschlossener Konfiguration des EUSCI.

STANDBY2TRANSFERING Explizite Datenübertragungsaufforderung. An die Transition sind zwei Speicheradressen für Sende- und Empfangsbyte geknüpft.

TRANSFERING2STANDBY Interruptbasierte Transition, die nach abgeschlossener Übertragung ausgelöst wird. Die Dauer zum Auslösen dieser Transition ist abhängig von der konfigurierten Übertragungsrate r in bit/s und berechnet sich als $t = \frac{8}{r}$.

5.1.1.2 *Programmierung*

Nach der Festlegung von Zuständen und Transitionen wird das Modell in Form einer C++ Klasse implementiert (Quelltext 5.1 und 5.2). Gemäß dem Entwurf wird die Treiberschnittstelle durch die Methoden `passTransition(...)` realisiert, die durch Übergabe eines Objekts des entsprechenden Transitionstypen ausgewählt werden. Die Transition `trSTANDBY2TRANSFERING` beinhaltet zwei Zeiger auf Übertragungsdaten und wird explizit vom Benutzer der Schnittstelle ausgeführt. Dem entgegen wird der Übergang `trTRANSFERING2STANDBY` auf Betriebssystemebene im Rahmen der ISR durchgeführt. Aus diesem Grund wird auch der zeitliche Parameter bei der Implementierung nicht berücksichtigt. Die Unterscheidung zwischen Benutzer- und Betriebssystemebene erfolgt anhand der Basisklasse des Transitionstypen und ist für die korrekte Synchronisierung des Stromzählers erforderlich (siehe Kapitel 5.2).

Transition enthält Nutzdaten

Da die Leistungsaufnahme und der Energieverbrauch zu diesem Zeitpunkt noch unbekannt sind, werden alle Werte auf $0\,\mu W$ bzw. $0\,pJ$ gesetzt.

Kosten zunächst auf o

5.1.1.3 *Probemessung*

Um die Übereinstimmung zwischen modellierten Zuständen und realem Energieverbrauch zu verifizieren wird die Implementierung mit eingewobenem Trigger mehreren Probemessungen unterzogen. Abbildung 5.2a zeigt eine MIMOSA Aufzeichnung einer SPI Übertragung von 2 Byte mit einer eingestellten Datenrate von $1\,Mbit/s$, wie sie das Display erfordert. Der Messaufbau beschränkt sich auf den MSP430 FR5969 des Launchpads ohne angeschlossenem Slave und keinen weiteren Peripheriegeräten. Von links nach rechts betrachtend steigt die Leistungsaufnahme zunächst an, da die CPU infolge eines abgelaufenen Timers aus dem Energiesparmodus (LPM1) geweckt wird. Während der Aktivphase zeigen die Triggerflanken (rot) einen Zustandswechsel und somit eine Übertragung an. Zunächst befindet sich das Gerät im Modus `STANDBY`, wechselt kurzzeitig in den Modus `TRANSFERING` (rotes Intervall) und anschließend infolge einer abgeschlossenen Übertragung wieder zurück. Der Ablauf wiederholt sich zur Übertragung des nächsten Bytes; danach wird die CPU wieder schlafen gelegt.

Durch die hohe Übertragungsrate befindet sich der Treiber lediglich für Dauer eines[3] MIMOSA-Samples (Abtastrate 100 kHz) im Übertragungszustand. Zwar findet während dieser Phase kein signifikanter Verbrauchssprung statt, allerdings kann ein geringfügiger Anstieg der Leistungsaufnahme in diesem Zustand aufgrund des Rauschens nicht abgelesen werden. Zur genaueren Untersuchung muss die Übertragungsrate signifikant verringert werden, sodass die restliche CPU

hohe Datenrate

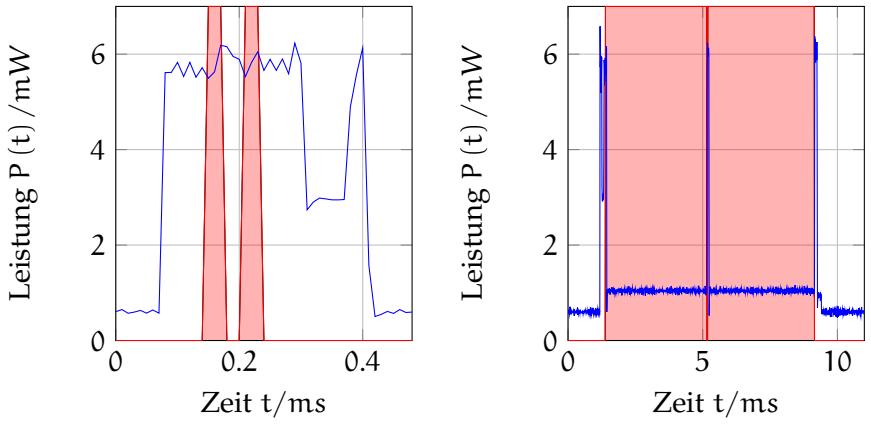
3 Der Übergang in den LPM1 und zurück streckt die Phase auf 3 Samples.

```
//...
namespace eUSCI_B {
class DFA_SPI : public DFA_Driver, //...
{
private:
  Guarded_Semaphore tx_sema;
public:
  struct State {
    enum state_t {
      UNINITIALIZED, STANDBY, TRANSFERING, _STATE_MAX
    };
  };
  struct trENTRY : public guarded_transition_t  {
    trENTRY() : guarded_transition_t(0) {}
  };
  struct trSTANDBY2TRANSFERING : public guarded_transition_t  {
    const unsigned char* tx_byte;
    unsigned char* rx_byte;
    trSTANDBY2TRANSFERING(const unsigned char* tx_byte, unsigned
        char* rx_byte) : guarded_transition_t(0), tx_byte(tx_byte
        ), rx_byte(rx_byte) {}
  };
  struct trTRANSFERING2STANDBY : public transition_t  {
    trTRANSFERING2STANDBY() : transition_t(0) {}
  };

  static const power_uW_t statepower[];
  GHOST int getPower_uW() { return statepower[getState()]; }
  GHOST int getPower_uW(int state) { return statepower[state]; }

  void passTransition(trENTRY trd);
  void passTransition(trSTANDBY2TRANSFERING trd);
  void passTransition(trTRANSFERING2STANDBY trd);
  //...
};}
```

Quelltext 5.1: Auszug der Implementierung vom SPI Treiber (`dfa_spi.h`). Vollversion in Quelltext C.1

```
//...
namespace eUSCI_B {
const DFA_Driver::power_uW_t DFA_SPI::statepower[] = {
  0,  // UNINITIALIZED
  0,  // STANDBY
  0   // TRANSFERING
};
void DFA_SPI::passTransition(trENTRY trd) {
  // Initialization
  //...
  setInternalState(State::STANDBY);
}
void DFA_SPI::passTransition(trSTANDBY2TRANSFERING trd) {
  setInternalState(State::TRANSFERING);
  UCB0IFG &= ~UCRXIFG;          //Reset RX-flag
  UCB0TXBUF = *(trd.tx_byte);   //Put TX-byte
  tx_sema.wait();               //Block

  if(trd.rx_byte != NULL)
    *(trd.rx_byte) = UCB0RXBUF;//Store RX-byte
}
void DFA_SPI::passTransition(trTRANSFERING2STANDBY trd) {
  setInternalState(State::STANDBY);
  tx_sema.Semaphore::signal(); //Unblock
}}
//...
```

Quelltext 5.2: Auszug der Implementierung vom SPI Treiber (`dfa_spi.cc`). Vollversion in Quelltext C.2



(a) SPI Bitrate: 1 Mbit/s.

(b) SPI Bitrate: 2 kbit/s.

Abbildung 5.2: Leistungsaufnahme des MSP430 bei der Übertragung von 2 Byte über SPI mit der Bitrate 1 Mbit/s und 2 kbit/s. Rote Flanken markieren Zustandswechsel des Treibers. Der Erhöhte Verbrauch während der Übertragung ist infolge der Übertragungsrate nur in (b) erkennbar (rot hinterlegt).

f_{SMCLK}	P_{SPI} (Messung)	P_{SPI} (Datenblatt)	ABWEICHUNG (Mess./Dat.)
16 MHz	340 µW	168 µW	2.02
8 MHz	160 µW	84 µW	1.90
4 MHz	86.7 µW	42 µW	2.04

Tabelle 5.1: Zusätzliche Leistungsaufnahme während einer SPI Übertragung im Vergleich zwischen Datenblatt [Tex14d] und Messung (Abbildung C.1) bei verschiedener Einspeisefrequenz von SMCLK. Der reale Verbrauch ist um ca. Faktor 2 höher als dokumentiert, skaliert aber gleichermaßen mit der Frequenz.

während der Übertragung für ein längeres Intervall schlafen gelegt werden kann. Abbildung 5.2b zeigt das gleiche Experiment mit einer Übertragungsrate von 2 kbit/s und sichtbaren Übertragungsphasen (rote Intervalle). An dieser Stelle zeichnet sich ein erhöhter Energieverbrauch infolge einer SPI Übertragung ab: Die CPU befindet sich sowohl während der weißen Phasen (links und rechts), als auch während der Übertragungen (rot) im LPM1, verbraucht in letzterer jedoch sichtbar mehr Energie. Diese Tatsache deckt sich auch mit dem im Datenblatt angegeben erhöhten Stromfluss während einer SPI Übertragung um 3.5 µA pro MHz [Tex14d]. Die Referenzgröße ist dabei nicht die Übertragungsrate, sondern der Einspeisetakt des EUSCI über SMCLK von 16 MHz. Hieraus folgt bei 3 V Betriebsspannung eine zusätzliche Leistungsaufnahme von 168 µW. Die Messung ergibt jedoch abweichend einen erhöhten Verbrauch um ca. 340 µW. Das gemessene Niveau verändert sich nicht bei anderen Übertragungsraten sondern skaliert – ebenso wie die Angabe aus dem Datenblatt – linear mit der Frequenz von SMCLK, sodass sich ein realer Verbrauch von ca. 21 µW pro MHz ergibt. Die entsprechenden Messaufzeichnugen sind Abbildung C.1 dargestellt und in Tabelle 5.1 zusammengefasst. Es zeigt sich, dass ein genaues Energiemodell unbedingt Verifikationsmessungen benötigt und das Datenblatt alleine nicht zum Erhalt korrekter Verbrauchswerte ausreicht.

niedrige Datenrate

Verbrauch skaliert mit SMCLK

Für die übrigen Zustände und Transitionen ergibt kein von 0 abweichender Verbrauch, da weder Messaufzeichnungen noch Datenblatt darauf hinweisen. Allerdings kann diese Schlussfolgerung nicht mit absoluter Sicherheit bestätigt werden, da SPI und aktive CPU nicht getrennt von einander vermessen werden können und ihre individuellen Verbrauchsprofile interferieren.

Interferenzen

5.1.2 *CPU*

Um den Energieverbrauch der CPU im System zu erfassen, soll auch ein Treiber hierfür entwickelt werden. Abgrenzend ist damit nicht

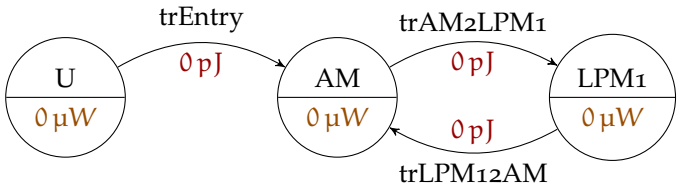


Abbildung 5.3: PTA Modell der CPU bezüglich ihrer Betriebsmodi mit den Zuständen `UNINITIALIZED`, `ACTIVEMODE` und `LOWPOWERMODE1`. Eine zeitliche Abschätzung des Übergangs von LPM1 nach AM ist nur möglich, wenn sich das dynamische Aufwachverhalten des Systems zeitabhängig beschreiben lässt, denn jeder Interrupt weckt die CPU aus dem Energiesparmodus.

Sonderrolle: CPU

etwa gesamte Mikrocontroller gemeint, sondern lediglich die ausführende Einheit des Mikrokontrollers mit ihren Energiesparmodi (siehe Kapitel 2.4.1.9). Die CPU stellt damit eine Sonderrolle für die Implementierung des Treibers dar, da für ihren Betrieb kein Treiber im klassischen Sinne erforderlich ist. Dennoch ist die CPU auch für einen beträchtlichen Anteil am Energieverbrauch eines Systems verantwortlich, sodass sie bei der Modellierung auch mit berücksichtigt werden sollte.

5.1.2.1 *Automatenmodellierung*

nur AM und LPM1

Wie bereits erläutert besitzt die CPU des MSP430 mehrere Betriebsmodi, wie Aktivmodus (AM) und Energiesparmodi (LPM0 ... LPM4, LPM3.5 und LPM4.5). In Kratos finden allerdings nur der AM und LPM1 Verwendung. Tiefere Schlafmodi können nicht verwendet werden, da der Scheduler durch einen Timer betrieben wird, der wiederum von SMCLK mit Takt versorgt wird. Das Taktsystem verhindert jedoch wegen der aktivierten Abhängigkeit von SMCLK einen Tiefschlafmodus, der diese Taktquelle abschalten würde. LPM0 findet ebenfalls keine Anwendung, da die im LPM1 einhergehende Reaktivierungszeit des FRAM (6 µs) in Kratos keine Nachteile bringt, LPM1 jedoch höheres Energiesparpotenzial bietet (vgl. Tabelle 2.1).

Damit setzt sich der Automat (Abbildung 5.3) aus zwei möglichen Betriebszuständen (`AM` und `LPM1`) zusammen, sowie dem zusätzlichen Pseudozustand `UNINITIALIZED` für den Zeitraum während der Initialisierungsphase des Betriebssystems. Die Anzahl der möglichen Transitionen beschränkt sich auf folgenden drei Übergänge:

ENTRY Automatischer Übergang von `UNINITIALIZED` zu `AM` nach Aufruf des Konstruktors.

AM2LPM1 Übergang in den Tiefschlafmodus. Die Transition wird in Kratos ausschließlich vom Scheduler ausgelöst, wenn die Warteschlange bereiter Prozesse leer ist.

```
aspect dfa_cpu {
  advice execution("void CPU::idle()") : before() {
    CONFIG_DFA_CPU_NAME.passTransition(DFA_CPU::trAM2LowPM1());
  };
  advice execution("% ...::prologue()") : after() {
    if(CONFIG_DFA_CPU_NAME.getState() == DFA_CPU::State::LowPM1)
      CONFIG_DFA_CPU_NAME.passTransition(DFA_CPU::trLowPM12AM());
  };
};
```

Quelltext 5.3: Einbindung des automatenbasierten Treibers für die CPU an korrespondierenden Stellen im Betriebssystem (`dfa-cpu.ah`). Der Übergang in den Energiesparmodus wird vor der Ausführung von `idle()` durchgeführt. Die Transition zurück in den Aktivmodus wird nach der Ausführung einer beliebigen `prologue()`-Methode im Rahmen einer Unterbrechungsbehandlung ausgelöst.

LPM12AM Erwachen aus dem Tiefschlafmodus durch einen beliebigen Interrupt. Eine zeitliche Abschätzung zum Auslösen dieser Transition kann nur erfolgen, wenn sich alle Unterbrechungsquellen zeitabhängig verhalten.

5.1.2.2 *Programmierung*

Die Realisierung des Automaten erfolgt wie beim SPI in Form einer Klasse gemäß dem Entwurf aus Kapitel 4. Da die CPU an sich keinen Treiber benötigt, beschränkt sich die Klassenimplementierung auf das Mitführen des aktuellen Zustands und Kapseln der jeweiligen Verbrauchswerte. Die Implementierung findet sich als vollständige Version im Anhang unter Quelltext C.3 und C.4.

Trickreich gestaltet sich hingegen die Einbindung dieses Treibers in das Betriebssystem. Da es ohne diesen Treiber auskommt, müssen die Transitionen mit Aspekten an den entsprechenden Schlüsselstellen eingewoben werden. Für den Übergang in den Tiefschlafmodus ist eine solche Methode `CPU::idle()`. Beim Erwachen müssen jedoch alle Interrupts berücksichtigt werden, sodass die zentrale ISR der Anlaufspunkt hierzu ist. Sie delegiert den Kontrollfluss an die zur Unterbrechung korrespondierende Behandlungsroutine, die sich in Kratos aus einem *Prolog* mit gesperrten Interrupts und einem optionalen *Epilog* mit reaktivierten Unterbrechungen zusammensetzt. Um die Transition so früh wie möglich nach dem Erwachen durchzuführen, ohne jedoch wichtige Aktionen im Prolog zu verzögern, wird die Transition in den Aktivmodus nach (`after()`) der Ausführung jedes Epilogs ausgelöst. Dies geschieht nur, wenn der Automat nicht bereits in diesem Zustand ist. Die Implementierung des Aspekts ist in Quelltext 5.3 abgebildet.

Treiber wird eingewoben

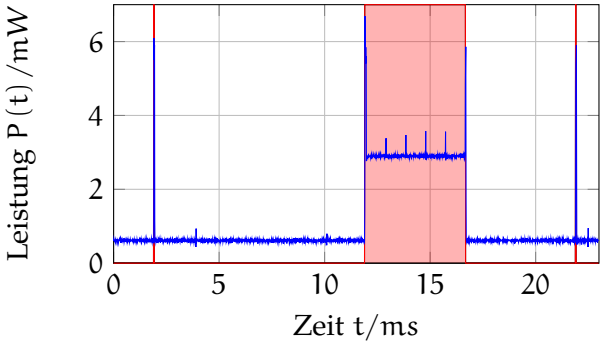


Abbildung 5.4: Leistungsaufnahme der CPU im Schlafmodus (weiß hinterlegt) und Aktivmodus (rot hinterlegt). Die Zustandswechsel werden durch den eingewobenen Trigger signalisiert. Während der längeren Aktivphase wird 5 Mal eine jeweils 1 ms aktiv wartende Methode (Zählschleife) aufgerufen. Die kurzen Spitzen bei 2 ms, 12 ms und 22 ms sind Timerinterrupts zum Aktualisieren der Systemzeit und Aufruf des Schedulers.

5.1.2.3 *Probemessung*

Zur Modellverifikation wird das laufende System mit eingewobenem Treiber und Trigger einer Messung unterzogen. Der Mikrocontroller wird dabei ohne zusätzliche Peripherie und mit einer Frequenz von 16 MHz betrieben. Abbildung 5.4 zeigt die Leistungsaufnahme der CPU bei der Ausführung einer Testanwendung. Dabei wird der Prozess zunächst schlafen gelegt, sodass das Betriebssystem die CPU in den Energiesparmodus versetzt. In Abständen von 10 ms erzeugt der Timer eine Unterbrechung, was die CPU aufweckt (Im Bild bei 2 ms, 12 ms und 22 ms). Dabei wird jeweils die Systemzeit aktualisiert (siehe Abschnitt 5.2.2.2) und der Scheduler aufgerufen, der ggf. wartende Prozesse reaktiviert. Sind keine Prozesse bereit, fällt die CPU wieder zurück in den LPM1. Bei der zweiten Unterbrechung im Bild wird der schlafende Prozess reaktiviert (rot hinterlegt). Während dieser Phase ruft der Prozess fünfmal (erkennbar an kleinen Peaks) eine *aktiv wartende* Methode auf, die jeweils 1 ms in einer Zählschleife verbringt. Dannach blockiert der Prozess und die CPU spart bis zum nächsten Interrupt Energie.

gleiche Aufgabe = gleicher Verbrauch

Auffällig zeigt sich, dass die CPU in Phasen gleicher Aufgabe (Zählschleife, Schlafen) jeweils eine annähernd konstante Leistungsaufnahme verzeichnet. Allerdings ergibt sich beim Vergleich verschiedener Aufgaben im Aktivmodus ein deutlicher Leistungsunterschied. So ergeben sich bei der Ausführung von Systemfunktionen (wie ISR, Scheduler) Spitzenwerte von ca. 6 m*W*, sowie erkennbare Spitzen beim fünffachen Aufruf der Verzögerungsmethode.

Diese Unterschiede sollen nun genauer untersucht werden, indem ein Testprogramm verschiedene Aufgaben in häufiger Wiederholung

DATENBLATT		MESSUNG	
Modus FRAM-Cache-Hitrate	Leistung (3 V,16 MHz)	Aufgabe	Leistung (3 V,16 MHz)
AM (0 %)	7.95 mW		
AM (50 %)	5.97 mW	`resume()`	5.7 mW
AM (66 %)	4.86 mW	leere Methode	4.9 mW
AM (75 %)	4.26 mW	`xor`	3.5 mW
AM (100 %)	2.19 mW	Zählschleife	2.9 mW
LPM1	0.54 mW		0.58 mW

Tabelle 5.2: Leistungsaufnahme des MSP430 FR5969 in verschiedenen Betriebsmodi laut Datenblatt [Tex14d] im Vergleich zu Messungen (Abbildung 5.4 und 5.5) bei 3 V und 16 MHz. Der dokumentierte Verbrauch deckt sich mit den Messergebnissen, ist im Aktivmodus jedoch stark von der Trefferate im FRAM-Cache abhängig.

unterschiedlicher Verbrauch je Aufgabe

ausführt. Vor und nach der Wiederholung wird der Prozessor mit einer Zählschleife aktiv gehalten. Zu den Aufgaben gehören jeweils 1000 Iterationen von `xor` Anweisungen (Abbildung 5.5 oben), Aufrufen einer leeren Methode (Abbildung 5.5 mitte) und Aufrufen der `resume()`-Methode vom Scheduler (Abbildung 5.5 unten). Letzteres veranlasst den Scheduler einen Kontextwechsel zum nächsten bereiten Prozess durchzuführen. Da es in diesem Experiment nur einen Prozess gibt, wird dieser direkt wieder reaktiviert. Die Triggerflanken stehen bei diesen Messungen nicht für Zustandswechsel im Automaten. Stattdessen markieren sie Start und Ende der Wiederholungsphasen, sodass die Iterationsintervalle rot hinterlegt erscheinen. Je nach Aufgabe varriiert in die Verlustleistung im Aktivmodus um ca. 3.4 μW. Beim Vergleich mit dem Datenblatt [Tex14d] ergeben sich klare Übereinstimmungen mit den gemessenen Werten, wie Tabelle 5.2 verdeutlicht. Die Verlustleistung ist dabei neben der Betriebsfrequenz auch von der Cache-Trefferrate beim Lesen von Instruktionen aus dem FRAM abhängig. Bei dem Cache handelt es sich um einen 2-Fach Satz-assoziativen Cache, bei dem jeder Satz aus einer 64 Bit langen Cachezeile besteht, die jeweils Platz für bis zu vier Maschineninstruktionen (je 16 Bit) bietet. Beim Zugriff auf das FRAM wird stets eine ganze Cachezeile befüllt, sodass die nachfolgenden drei Instruktionen keinen FRAM Zugriff benötigen, wenn kein Sprung oder Funktionsaufruf außerhalb der Cacheinhalte stattfindet. Beim Erreichen vom Zeilenende wird rechtzeitig die nächste Zeile geladen.

FRAM-Cache

Entsprechend erreicht eine Zählschleife oder die wiederholte Ausführung von `xor` eine höhere Trefferrate, als Aufrufe der stets gleichen leeren Methode, bzw. die vielfältigen Aufrufe innerhalb des Schedulers.

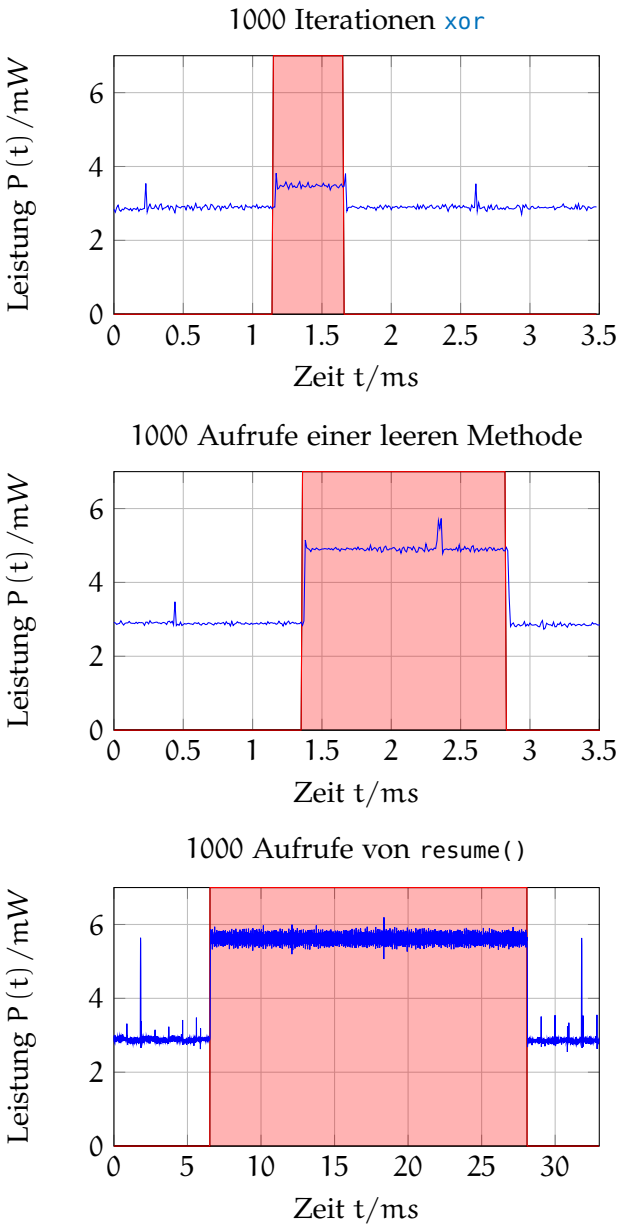


Abbildung 5.5: Leistungsaufnahme des MSP430 FR5969 während der 1000-fachen Ausführung der betitelten Instruktionen. Je nach Aufgabe schwankt die Leistungsaufnahme des Aktivmodus. Außerhalb der Iterationsintervalle befindet sich die CPU aktiv in einer Zählschleife (weiß hinterlegt).

Durch die Abhängigkeit von der Trefferrate im FRAM-Cache ergeben sich jedoch Konsequenzen für das Energiemodell. Die Rate kann ohne Messungen weder zur Entwicklungszeit noch zur Laufzeit zuverlässig ermittelt werden, da sie abhängig von der Programmstruktur und der Ausrichtung der Instruktionen ist. Erschwerend kommt hinzu, dass der Hersteller in den Datenblättern keine Auskunft über die konkrete Ersetzungsstrategie des FRAM-Caches erteilt, sondern beschreibt sie lediglich als „intelligente Logik".

Konsequenzen des Cache

Bei der Verwendung dieses Modells zur Simulation des Energieverbrauchs muss deswegen die Leistungsaufnahme im Aktivmodus anhand der erwarteten Trefferrate abgeschätzt werden. Um auch zur Laufzeit repräsentative Werte zu erhalten, sind finale Messungen des entgültigen Systems durchzuführen, sodass die mittlere Leistungsaufnahme des Aktivmodus – für diese konkrete Konfiguration – ermittelt wird.

5.1.3 *Ausgabegeräte/Akuatoren: Display*

Als Repräsentant für ein Ausgabegerät wird ein LC-Display gewählt, welches als Erweiterungsset für das Launchpad geliefert wird (siehe Abbildung 2.10 rechts). Dabei handelt es sich um das LS013B4DN04 von Sharp [Sha14] mit einer Auflösung von 96×96 Pixeln und einer Versorgungsspannung von 3 V bis 5 V. Es wird über SPI mit dem MSP430 des Launchpads verbunden und hat zusätzliche Signalleitungen für CS, sowie für das Ein-/Ausschalten der Bildschirminhalte (DISP). Um starre Pixel zu vermeiden, muss dem Display periodisch mit einer Frequenz von 1 Hz bis 60 Hz ein VCOM-Signal übergeben werden, was die Ladungspolarität innerhalb des Displays invertiert. Dieses Signal kann entweder über eine weitere dafür vorgesehene Signalleitung (EXTCOMIN) oder (wie hier gewählt) per Software im Rahmen einer normalen Datenübertragung mitgeteilt werden. Der Inhalt des Bildschirms kann pixelweise beeinflusst werden, wobei eine Datenübertragung immer mindestens eine ganze Pixelzeile beinhalten muss. Ein Bit steht dabei jeweils für ein Pixel, die insgesamt zu 12 Byte ($12 \cdot 8 = 96$) zusammengefasst werden. In der Präambel einer Übertragung wird zusätzlich die Zeilennummer angegeben, sodass Bildschirmzeilen die in beliebiger Reihenfolge und Anzahl übertragen werden können. Die übertragenen Pixel bleiben gespeichert, solange die Versorgungsspannung aktiviert ist.

Ladungspolarität

1 Byte = 8 Pixel

5.1.3.1 *Automatenmodellierung*

Da das Datenblatt keinen Zustandsautomaten für das Display angibt, und die Leistungsaufnahme nur als Durchschnittswerte in zwei Anwendungsszenarien angegeben wird, werden die Zustände und Transitionen anhand der Anforderungen aus der Anwendungsschicht

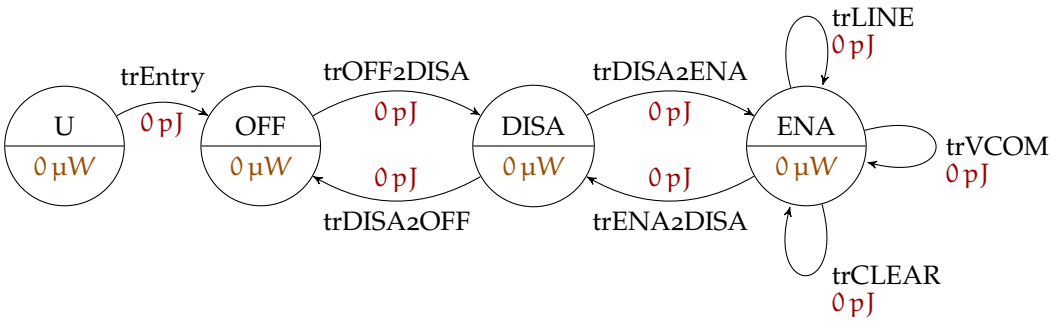


Abbildung 5.6: Automatenmodell der Betriebsmodi vom Sharp Display mit den Zuständen `UNINITIALIZED`, `POWEROFF`, `DISABLED`, `ENABLED`. Im letzten Zustand können Pixel, VCOM-Signal und ein Löschbefehl übertragen werden.

und der Funktionen des Displays aufgestellt. Dabei werden folgende Zustände identifiziert:

UNINITIALIZED Zustand des Displays beim Einschalten des Launchpad.

POWEROFF Versorgungsspannung des Displays ist ausgeschaltet.

DISABLED Die Versorgungsspannung zum Display ist eingeschaltet, das Display ist jedoch über die Signalleitung DISP deaktiviert. Zuvor übertragene Speicherinhalte bleiben erhalten, werden jedoch nicht dargestellt.

ENABLED Das Display ist eingeschaltet und stellt den Speicherinhalt sichtbar dar. Es muss periodisch ein VCOM ausgelöst werden, um starre Pixel zu vermeiden.

Der resultierende Automat ist in Abbildung 5.6 dargestellt und hat zu seinen Zuständen die folgenden Transitionen:

ENTRY Automatischer Übergang in `POWEROFF` bei Aufruf des Konstruktors.

POWEROFF2DISABLED Einschalten der Versorgungsspannung bei deaktivierter Bildschirmdarstellung.

DISABLED2POWEROFF Ausschalten der Versorgungsspannung, wobei die Speicherinhalt gelöscht wird.

DISABLED2ENABLED Darstellung der gespeicherten Bildinhalte auf dem Bildschirm.

ENABLED2DISABLED Darstellung deaktivieren, Bildinhalte verbleiben gespeichert.

ENABLED2ENABLED_LINE Übertragung einer Pixelzeile (96 Pixel). Die Transition beinhaltet eine Referenz auf die benötigten Pixeldaten (12 B) und den Index der Zielzeile. Das Display verbleibt im Zustand `ENABLED`.

ENABLED2ENABLED_VCOM Übertragung des VCOM-Signals. Das Display verbleibt im Zustand `ENABLED`.

ENABLED2ENABLED_CLEAR Übertragung der Befehls zum Löschen des Bildschirminhalts. Der Zustand bleibt unverändert.

5.1.3.2 *Programmierung*

Die Programmierung gestaltet sich auf gleiche Weise, wie zuvor das SPI und die CPU. Die einzige Transition, die Parameter benötigt ist `trENABLED2ENABLED_LINE` mit Pixeldaten und Zielzeile. Quelltext 5.4 zeigt die Implementierung der korrespondierenden Methode. Zeilen 1 bis 5 beschreiben den Transitionstypen, der die benötigeten Daten kapselt. Beim Aufruf der Transition in Zeile 7 wird ein vorallozierter Puffer für die Übertragung vorbereitet (Zeilen 8 bis 13). Anschließend wird das SPI exklusiv belegt (14), die Daten übertragen, der Bus wieder freigegeben und der Zustand aktualisiert. Da der Zustand unverändert bleibt ist letzteres nur aus Gründen der Vollständigkeit enthalten.

Selbst wenn das Display die einzige Komponente am SPI ist, muss der Bus während der Übertragung gesperrt werden. Grund dafür sind periodische Transitionen `trENABLED2ENABLED_VCOM`, die asynchron durch einen separaten Prozess ausgelöst werden.

konkurrierender Zugriff

5.1.3.3 *Probemessung*

Bei einer Probemessung wird das Display durch alle Transitionen geführt. Es ist dabei am Mikrocontroller vom Launchpad angeschlossen, während die Versorgungsspannung des Displays durch MIMOSA bereitgestellt wird. Damit wird nur der tatsächliche Verbrauch des Displays ohne Einfluss anderer Komponenten aufgezeichnet. Abbildung 5.7 zeigt die Resultate der Aufzeichnung. Im Gegensatz zu den bisherigen Aufzeichnungen zeigen Triggerflanken den Start und das Ende einer Transition an. Somit ist nicht ein bestimmter Zustand, sondern die Transitionsphase rot hinterlegt. Zeilenweise von links nach rechts wird das Display zunächst eingeschaltet, was unmittelbar den Prozess zum Auslösen von VCOM reaktiviert (zweites Intervall). Anschließend wird dem Display der Befehl zum Löschen des Bildschirminhalts gesendet. Dannach beginnt eine Übertragung von neuem Bildschirminhalt. Jedes rote Intervall steht dabei für eine übertragene Zeile. Aus Platzgründen ist die Aufzeichnung zu Beginn der vierten Zeile abgeschnitten. Schließlich wird das Display wieder deaktiviert.

separierte Spannungsversorgung

```
struct trENABLED2ENABLED_LINE : public transition_t  {
  const LinePixels& data;
  uint8_t line;
  trENABLED2ENABLED_LINE(const LinePixels& data, uint8_t line) :
      transition_t(0), data(data), line(line) {}
};

void Sharp_LS013B4DN04::passTransition(trENABLED2ENABLED_LINE trd
    ) {
  transmitLinebuffer.cmd = (1<<MODE_BIT) | vcom_frame.cmd;
  transmitLinebuffer.line = bitReverseTable[trd.line];
  for(unsigned int t=0; t<sizeof(transmitLinebuffer.pixels); t++)
      {
    transmitLinebuffer.pixels[t] = trd.data.pixels[t];
  }

  displayBus.wait();
  cs_high();
  displayBus.send((uint8_t*)&transmitLinebuffer, sizeof(
      transmitLinebuffer));
  cs_low();
  displayBus.signal();
  setInternalState(State::ENABLED);
}
```

Quelltext 5.4: Auszug aus der Implementierung des Automatenbasierten Displaytreibers. Es zeigt die Transition zum Übertragen einer Pixelzeile im Modus `ENABLED`. Vollversion in Quelltexten C.5 bis C.7

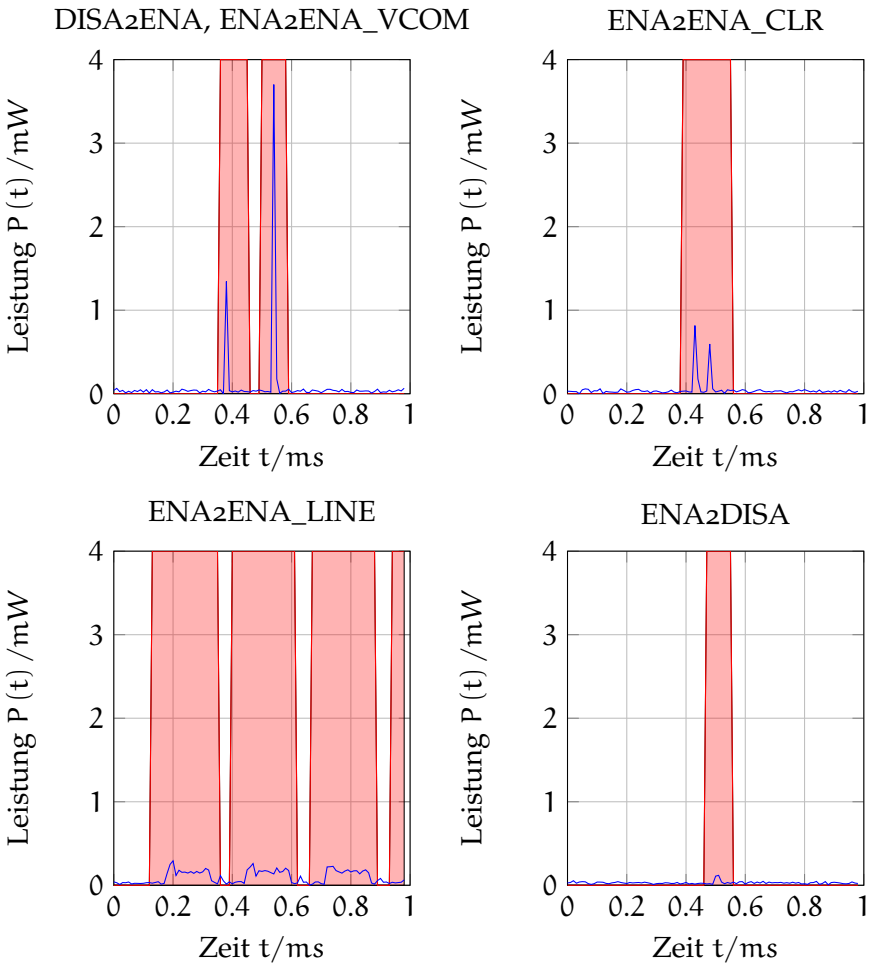


Abbildung 5.7: Leistungsaufnahme des Displays bei der Durchführung von Transitionen. Die Transitionsphase ist rot hinterlegt, Quell- und Zielzustand gehen aus dem betitelten Aktionen hervor.

Übereinstimmung mit Modell

Obwohl sich die Leistungsaufnahme während der Zustandsübergänge je Transitionsart deutlich unterscheidet, weist das Display während der Verweilzeit innerhalb der Zustände, unabhängig ob aus- oder eingeschaltet, keine erkennbare Verlustleistung auf. Die Aufzeichnungen zeigen eine deutliche Übereinstimmung zwischen modellierten Zuständen/Transitionen und realem Verbrauch. Es treten weder unerwartete Variationen während eines Zustands, noch ein abweichender Energiebedarf beim Vergleich der wiederholten Ausführung von Transitionen auf. Eine detaillierte Ermittlung der Werte findet sich Kapitel 6.1.2.1.

5.1.4 *Eingabegeräte/Sensoren: Beschleunigungssensor*

Stellvertretend für die Gruppe von Eingabegeräten und Sensoren wird ein energiegewahrer Treiber für den Beschleunigungssensor des InBin, einem ADXL362 von Analog Devices [Ana13], implementiert. Er zeichnet sich durch eine äußerst geringe Energieaufnahme, sowohl im Schlaf-, als auch im Aktivmodus aus. Zudem hat er einen Betriebsspannungsbereich von 1.6 V bis 3.5 V, eine einstellbare Abtastrate von 12.4 Hz bis 400 Hz und eine Anbindung über SPI mit empfohlenen Taktraten zwischen 1 MHz und 5 MHz.

5.1.4.1 *Automatenmodellierung*

Arbeitsmodi des Beschleunigungssensors

Der Beschleunigungssensor kann in verschiedenen Arbeitsmodi betrieben werden. Dazu gehört zunächst der klassische Anwendungsfall, in dem auf explizite Anforderung die aktuellsten Beschleunigungswerte ausgelesen werden. Der Sensor kann jedoch auch Unterbrechungen beim Überschreiten eines wählbaren Schwellwerts erzeugen.

Die Modellierung wird im ersten Versuch lediglich auf den klassischen Anwendungsfall beschränkt, da der dynamische Unterbrechungsmodus kein zeitlich vorhersagbares Verhalten aufweist. Für die statische Analyse könnte der Kontrollfluss zwar durch die bekannte Transition weiterhin vervollständigt werden, jedoch kein sinnvoller zeitlicher Parameter angegeben werden.

Für den gewählten Anwendungsfall gehen aus dem Datenblatt folgende Zustände hervor:

UNINITIALIZED Zustand bei der Einschaltphase des Systems.

POWEROFF Spannungsversorgung ist ausgeschaltet.

STANDBY Spannungsversorgung ist eingeschaltet, der Sensor befindet sich jedoch im Energiesparmodus und nimmt keine Daten auf.

MEASURE Der Sensor erfasst Beschleunigungswerte und speichert sie im internen Puffer. Sie können über SPI ausgelesen werden.

Anzumerken ist, dass der Zustand `POWEROFF` im InBin durch einen der Spannungsversorgung vorgeschalteten Transistor realisiert wird, der über den MSP430 gesteuert werden kann. Zu den Zuständen ergeben sich folgende Übergänge:

ENTRY Automatischer Übergang in den Modus `POWEROFF` beim Start des Systems.

POWEROFF2STANDBY Einschalten der Versorgungsspannung.

STANDBY2POWEROFF Ausschalten der Versorgungsspannung.

STANDBY2MEASURE Sensoraufzeichnung aktivieren.

MEASURE2STANDBY Sensoraufzeichnung deaktivieren.

MEASURE2MEASURE_READ Beschleunigungswerte auslesen. An die Transition ist eine Speicheradresse von einem 3-Byte-Tupel geknüpft, in das die Sensorwerte geschrieben werden sollen.

Die Programmierung folgt dem entworfenen Schema, welches zuvor demonstiert wurde (siehe Abschnitt 5.1.1.2, 5.1.2.2 und 5.1.3.2). Da sich hierbei keine Besonderheiten aufzeigen, wird nicht weiter auf den konkreten Quelltext eingegangen.

5.1.4.2 *Probemessung*

In einer Probemessung wird der Beschleunigungssensor durch die Zustände `POWEROFF`, `STANDBY` und `MEASURE` geführt und seine Leistungsaufnahme untersucht. Die Messaufzeichnung ist in Abbildung 5.8 dargestellt. Die orangene Kurve stellt die Leistungsaufnahme des gesamten InBin während des Experiments dar, worin die periodischen Scheduleraufrufe als kleinere Spitzen deutlich zu erkennen sind. Die ersten beiden großen Spitzen entstehen beim Einschaltvorgang des Beschleunigungssensors, ausgelöst durch Einschalten des Absperrtransistors[4] und das um 5 ms verzögerte Setzen in `STANDBY`. Die dritte signifikante Spitze entsteht beim Übergang in `MEASURE`. Ab diesem Zeitpunkt hebt sich auch die kontinuierliche Leistungsaufnahme leicht an.

Verbrauchsspitzen beim Einschalten

Zum Vergleich zeigt die blaue Kurve die alleinige Leistungsaufnahme des Beschleunigungssensors, gemessen an seiner Spannungsversorgung, während einer Wiederholung des gleichen Experiments. Da der Sensor von Beginn an durch MIMOSA mit Strom versorgt wird, zeigt sich keine Reaktion bei der ersten Spitze der orangenen Kurve. Beim Übersenden der Moduswechsel an den Sensor zeigen sich allerdings entsprechende Spitzen.

4 Dabei werden schlagartig Abblockkapazitäten von insgesamt 2.1 µF aufgeladen, was bei MIMOSA zu kurzem Nachschwingen führt.

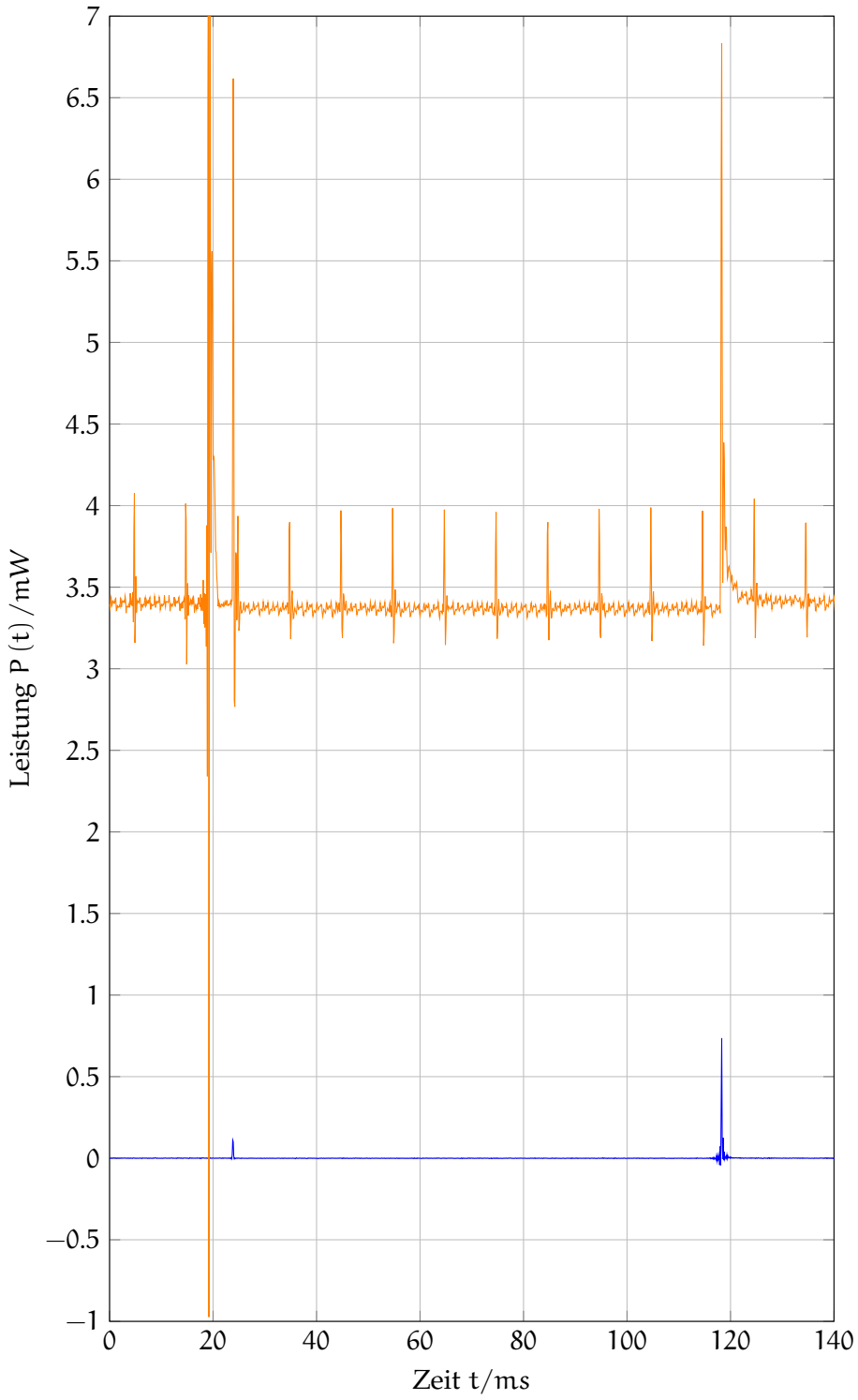


Abbildung 5.8: Orange: Leistungsaufnahme des InBin bei der Verwendung des Beschleunigungssensors, während die CPU in einer Zählschleife aktiv gehalten wird. Die drei großen Spitzen entstehen (von links) durch Zustandswechsel des Sensors beim *Einschalten der Versorgungsspannung*, Setzen in Modus *Standby* und Übergang in den *Messmodus*. Kleine Spitzen sind Scheduleraufrufe. Blau: Alleinige Leistungsaufnahme des Sensors, gemessen an der Versorgungsspannungsleitung.

Der Amplitudenunterschied beim Vergleich der Kurven ergibt sich aus dem Energiebedarf des MSP430 für den SPI Transfer (vgl. Abbildung 5.1a), der in der blauen Kurve nicht mitgemessen wird. Auffällig zeigt sich jedoch bei der Einzelmessung eine konstante Leistungsaufnahme von 0 mW, selbst *nach* dem Übergang in den Modus `MEASURE`. In der Gesamtmessung ergibt sich hier jedoch ein Anstieg um etwa 40 µW, was sich bei 3 V Betriebsspannung mit der Herstellerangabe von 13 µA für diesen Modus deckt.

Messparadoxon

Bei der Analyse dieser Unstimmigkeit fiel auf, dass der Sensor selbst ohne angeschlossene Versorgungsspannung arbeitet und währenddessen am Versorgungsspannungsanschluss eine Spannung von etwa 2.7 V anliegt. Dieser Pegel liegt nur an, wenn die CS-Leitung einen 1-Pegel (3 V) aufweist. Der Sensor wird somit durch parasitäre Stromabflüsse über diese Signalleitung mit Strom versorgt. Da die CS-Leitung active low ist, hat die Leitung nur während der SPI Übertragungsphase einen 0-Pegel. Durch die großzügigen Abblockkondensatoren am Versorgungsspannungseingang werden diese Phasen problemlos überbrückt. Während der Einzelmessung des Sensors wird die übrige Platine über das Programmiermodul mit einer Spannung von etwa 3.6 V versorgt, was unabhängig von der MIMOSA-Versorgung (3 V) eine Rückspeisung über CS auf 3.3 V zur Folge hat. Lediglich in Phasen der Übertragung werden die Kondensatoren durch MIMOSA nachgeladen.

Arbeitet ohne Versorgungsspannung

durch parasitären Stromfluss

Die Abblockkondensatoren und Leckströme verhindern auch ein Zurücksetzen des Beschleunigungssensors bei einem kurz andauernden Reset des Gesamtsystems. Daher erscheint in Abbildung 5.8 vor dem Einschalten der Sensorspannung noch der Energieverbrauch aus dem vorangegangenen Experiment im `MEASURE`-Modus. Aus diesem Grund wird 5 ms nach dem Einschalten der Versorgungsspannung explizit auf `STANDBY` gewechselt. Im Regelfall – wenn der Sensor längere Zeit sicher ausgeschaltet bleibt – ist dies nicht erforderlich, da beim Einschalten automatisch dieser Modus betreten wird.

Kondensator überbrückt Reset

Zur weiteren Analyse der Leckströme wurde MIMOSA um die Möglichkeit erweitert, die Versorgungsspannung stufenlos einstellen zu können (siehe Kapitel B.2). Eine detaillierte Untersuchung entfällt leider, da das InBin zwischenzeitlich bei anderen Forschungsarbeiten beschädigt wurde und aufgrund der zeitlichen Beschränkung nicht rechtzeitig repariert werden konnte. Die Ergebnisse zeigen jedoch bereits jetzt, dass insbesonders bei energiesparenden Komponenten auf Leckströme geachtet werden muss, um zuverlässige Ergebnisse zu erhalten. Gegebenenfalls müssen Komponenten bei der Messung vollständig galvanisch vom ansteuernden System getrennt werden.

```
#include <msp430.h>
#include "config.h"
#define GHOST __attribute((always_inline))
class PinTrigger {
public:
  GHOST void init() {/*...*/}
  GHOST void set() {/*...*/}
  GHOST void clear() {/*...*/}
  GHOST void impulse() {
    TRIGGEROUT |= (1 << CONFIG_TRIGGERBIT);
    TRIGGEROUT &= ~(1 << CONFIG_TRIGGERBIT);
  }
  GHOST void toggle() {
    TRIGGEROUT ^= (1 << CONFIG_TRIGGERBIT);
  }
  GHOST void autotrigger() {
    // to be filled by aspect
  }
};
extern PinTrigger pinTrigger;
```

Quelltext 5.5: Implementierung des Triggers. Alle Methoden sind `inline` und setzen lediglich IO-Ports.

5.2 IMPLEMENTIERUNG DER ZUSATZKOMPONENTEN

Um einen genaueren Überblick über die Arbeitsweise der Zusatzkomponenten (Trigger und Stromzähler) zu erhalten und Implementierungsentscheidungen begründen zu können, wird deren Implementierung kurz vorgestellt.

5.2.1 *Trigger*

minimale Wechselwirkungen

Da der Trigger nur während der Treiberentwicklungsphase beim Vermessen der Verbrauchswerte zum Einsatz komm, im endgültigen System jedoch nicht eingewoben wird, muss sein Einfluss auf das Systemverhalten auf ein absolutes Minimum beschränkt werden. Der Trigger setzt sich deswegen aus ausschließlich `inline`-Methoden zusammen, die lediglich den Zustand eines IO-Ports verändert (siehe Quelltext 5.5). Diese Zustandswechsel werden durch Aufrufe von `autotrigger()` vor und nach dem Auslösen jeder Transition eines ausgewählten Treibers eingewoben, wie Quelltext 5.6 für das Display demonstriert. Die Art der Signalisierung wird dabei zentral durch Auswahl eines entsprechenden Aspekts, z. B. Quelltext 5.7 festgelegt. Durch das Auflösen von `inline`-Funktionen ergibt sich bei `toggle()` ein Overhead von einer Maschineninstruktion (Quelltext 5.8) pro Einsatzstelle.

```
aspect Sharp_LS013B4DN04_Autotrigger {
  advice execution("void Sharp_LS013B4DN04::passTransition(...)")
       : before() {
    pinTrigger.autotrigger();
  }
  advice execution("void Sharp_LS013B4DN04::passTransition(...)")
       : after() {
    pinTrigger.autotrigger();
  }
};
```

Quelltext 5.6: Aspekt zum Einweben der Triggerfunktionalität in Zustandswechsel des Displaytreibers vor und nach Betreten jeder Transition.

```
aspect AutotriggerToggle {
  advice execution("void PinTrigger::autotrigger()") : before() {
    tjp->target()->toggle();
  }
};
```

Quelltext 5.7: Aspekt zum Setzen der Funktionalität von `autotrigger()` auf `toggle()`.

```
;...
4532:   f2 e2 02 02     xor.b   #8,     &0x0202 ;r2 As==11
;...
```

Quelltext 5.8: Durch `inline` löst sich die Methode `toggle()` zu einer Maschineninstruktion auf.

5.2.2 *Energy Profiler*

Der Stromzähler soll das online-Profiling von Treibern ermöglichen. Durch die aspektorientierte Struktur kann er den Energieverbrauch einer beliebigen Teilmenge von Systemtreibern „abrechnen“. Jeder berücksichtigte Treiber erhält hierfür zusätzliche Attribute, die jeweils den bisherigen Energieverbrauch, den Zeitpunkt der letzten Abrechnung und die Dauer des gesamten Berechnungszeitraums speichern.

Einweben der Abrechnung

Die Abrechnung erfolgt bei jeder Transition durch das Einweben eines Methodenaufrufs von `billTransition(...)` innerhalb der Treibermethoden `passTransition(...)`. Der Inhalt in Quelltext 5.9 dargestellt. Nach Auslesen der aktuellen Systemzeit (Zeile 28) werden zunächst die Leistungsaufnahme des bisherigen Zustands und seine Verweildauer multipliziert und gemeinsam mit der Transitionsenergie zur Stromrechnung addiert. Außerdem wird der Zeitpunkt der letzten Abrechnung aufgefrischt.

große Zahlentypen

Da als Datentypen für Systemzeit und Stromrechnung `unsigned long`, bzw. `unsigned long long` verwendet werden müssen, werden Multiplikationen und Additionen aus mehreren Maschineninstruktionen zusammengesetzt, die bei jeder Transition ausgeführt werden müssen. Ein alternativer Ansatz wäre die gebündelte Multiplikation erst beim Auslesen des Stromzählers durch `getEnergy()`. Allerdings benötigt dieser Weg zusätzliches RAM zum Speichern der individuellen Aufrufhäufigkeit jeder einzelnen Transition, sowie für jeden Zustand einen Zeitakkumulator zur Erfassung seiner Aufenthaltszeit. Nimmt man für dieses Szenario ein System an, das 4 Treiber à 3 Zustände und jeweils 6 Transitionen abrechnet, ergibt sich ein RAM Verbrauch von

$$4 \cdot \left(\overbrace{3 \cdot 8\,\mathrm{B}}^{64\text{ Bit je Zustand}} + \underbrace{6 \cdot 4\,\mathrm{B}}_{32\text{ Bit je Transition}} \right) = 192\,\mathrm{B}, \tag{5.1}$$

was bei 2 KiB (MSP430 FR5969) bereits 10.7 % des verfügbaren Arbeitsspeichers verbraucht! Aus diesem Grund wurde von dieser Variante abgesehen.

5.2.2.1 *Synchronisierung*

Da Transitionen ggf. konkurrierend durch mehrere Prozesse ausgelöst werden können, muss der Zugriff auf den Stromzähler synchronisiert werden, um falsche oder verschluckte Abrechnungen zu verhindern. Aus Effizienzgründen obliegt es dem Entwickler bei der Implementierung von `passTransition()` die Synchronität der Transitionen – sofern sie überhaubt benötigt wird – sicherzustellen. Durch das Einweben der Energieabrechnung vor (`before()`) diesen Methoden, entzieht sich der Stromzähler dem Einfluss von ggf. darin implementierter Synchronisierung. Aus diesem Grund existiert in der

```
#define GHOST __attribute((always_inline))
slice class
ElectricityContractConclusion : public ElectricityContract {
private:
  volatile Watch::time_t prevBillingTime;
  volatile bool energybillingEnabled;
  ElectricityMeter::energy_pJ_accumulator_t energybill;
  ElectricityMeter::time_us_t timebill;

  GHOST void init() {
    energybillingEnabled = false;
    prevBillingTime = 0;
    energybill = 0;
    timebill = 0;
  }
public:
  void
  billTransitionPrecisely(Watch::time_t now_us, DFA_Driver::
      transition_t transition) {
    DFA_Driver::power_uW_t powerdraw = getPower_uW();
    Watch::time_t deltaTime = now_us-prevBillingTime;
    energybill += deltaTime*powerdraw;
    energybill += transition.transitionenergy;
    timebill += deltaTime;
    prevBillingTime = now_us;
  }
  void
  billTransition(DFA_Driver::transition_t transition) {
    billTransitionPrecisely(watch.now_us(), transition);
  }
  GHOST ElectricityMeter::energy_pJ_result_t
  getEnergy() {
    Secure sec;
    return energybill;
  }
  /*...*/
};
```

Quelltext 5.9: Ausschnitt aus der `slice class ElectricityContract`, die zur Abrechnung der Energieaufnahme in den Gerätetreiber eingewoben wird.

synchrone Transition

Klasse `DFA_Driver` ein spezieller Transitionstyp `secure_transition_t`, der in solchen Fällen als Basisklasse des Transitionsparameters zu verwenden ist. Anhand dessen werden zu synchronisierende Abrechnungen gekennzeichnet und erkannt. Bei diesen Transitionen wird statt `billTransition(...)` ein *wrapper* eingewoben, der den Kontrollfluss während der Abrechnung auf Epilog-Ebene hebt. Konkurrierende Zugriffe werden dadurch sequenzialisiert. Transitionen, die direkt auf dieser Ebene durchgeführt werden sind ohnehin sequenzialisiert und dürfen diesen Spezialtypen deshalb nicht verwenden, um den Bereich nicht doppelt zu betreten.

Transition auf Prolog-Ebene

Gleiches gilt für Zustandswechsel auf der Prolog-Ebene, da diese nicht unterbrochen werden können und dadurch sequenzialisiert sind. Es ist jedoch zu beachten, dass die Synchronizität zwischen Transitionen auf Prolog-Ebene und den übrigen Ebenen dadurch nicht gewährleistet ist, da hierfür Interrupts während der Abrechnung gesperrt werden müssten. Bei den bisher implementierten Treibern ist nur bei der CPU ein Zustandswechsel auf Prolog-Ebene erforderlich gewesen, um auf alle Unterbrechungen reagieren zu können. Die Synchronizität ist in diesem Fall jedoch durch die Art der Implementierung gegeben, da die Transition im Prolog (beim Aufwachen der CPU) nur ausgeführt wird, wenn sie sich zuvor im Energiesparmodus befunden hat. Dieser wird jedoch erst nach der Abrechnung gesetzt.

Im allgemeinen Fall sollten Transitionen nur auf Epilog-Ebene oder tiefer ausgelöst werden, um lange Phasen mit gesperrten Unterbrechungen, wie sie zwangsläufig durch die Abrechnung auf Prolog-Ebene stattfinden, zu vermeiden.

5.2.2.2 *Zeiterfassungsproblem*

Beim Einsatz des Stromzählers für das Profiling von Treibern mancher Geräte, wie der CPU (siehe Abschnitt 5.1.2), kann es zu Ungenauigkeiten bei der Zeiterfassung kommen, sodass Anpassungen am Betriebssystem erforderlich sind. Die Problematik liegt in der interruptbasierten Erfassung der Systemzeit `system_ticks`. Sie wird in Intervallen von 10 ms inkrementiert, jeweils ausgelöst durch einen Timerüberlauf. Der Timer wiederum zählt in seinem internen Zählregister mit einer 5000-fachen Rate, sodass sich durch Kombination beider Zähler eine Systemzeiterfassung mit der Genauigkeit von 2 µs realisieren lässt. Die Methode ist in Quelltext 5.10 aufgelistet. Dabei muss sichergestellt werden, dass im Falle eines Timerüberlaufs *just* im Moment des Auslesens trotzdem ein korrekter Zeitwert ermittelt wird. Findet z. B. zwischen Zeilen 6 und 7 ein Überlauf des Timers statt, würde im Rahmen der ISR `system_ticks` aktualisiert, `ticks` verbliebe jedoch auf dem bisherigen Stand. Nach der Unterbrechung würde der Timerwert aus `TA0R` ausgelesen werden, der durch den Überlauf nun einen sehr kleinen Wert enthält. Ohne weitere Sicherheitsmaßnahmen würde der zusammengesetzte Zeitwert deutlich kleiner ausfallen (in

2 µs Genauigkeit

kritischer Überlauf

```
time_t now_us() {
  tick_t ticks;
  unsigned int timerValue;

  do {
    ticks = system_ticks;   //Systemticks lesen
    timerValue = TA0R;      //Timer-Zählregister lesen
  } while(ticks != system_ticks);

  if ((TA0CTL & TAIFG) != 0) {
    ticks++;
    timerValue = TA0R;
  }

  return ticks * (1000000L/ticktime)  //ticks nach us
  + timerValue *                      //timerwert nach us
  ((PRESCALER1_FACTOR * PRESCALER2_FACTOR * 1000000) /
   CONFIG_SMCLK_FREQUENCY);
}
```

Quelltext 5.10: Erweiterte Methode zum Auslesen der Systemzeit. Um die Präzision zu erhöhen setzt sich die Zeit aus `system_ticks` (Genauigkeit: 10 ms) und dem Timerwert `TA0R` (Genauigkeit: 2 µs) zusammen. Auf Benutzer- und Epilog-Ebene wird die Synchronität beider Werte durch die Schleife sichergestellt. Auf Prolog-Ebene wird durch Abfrage des Interruptflags `TAIFG` bei Bedarf eine Korrektur des Zählers durchgeführt.

der Größenordnung von 10 ms) und bei Differenzbildung mit einem *kürzlich zuvor* ausgelesenen Zeitwert zu einer *negativen* Dauer führen.

Zeitreisen?...

Aus diesem Grund werden zunächst beide Zählerstände nacheinander zwischengespeichert und anschließend in einem Schleifenfuß auf eine zwischenzeitliche Änderung von `system_ticks` geprüft. Bei einer Änderung werden beide Werte erneut ausgelesen. Da dieser Spezialfall mit der Wahrscheinlichkeit $\frac{1}{5000}$ sehr selten auftritt, kommt es in den meisten Fällen zu keinen Wiederholungen.

...nein!

Diese Methode funktioniert für Kontrollflüsse auf Benutzer- und Epilog-Ebene, da sie von Unterbrechungen verdrängt werden. Wird `now_us()` allerdings selbst innerhalb einer ISR (auf Prolog-Ebene) ausgeführt, so greift die schleifenbasierte Schutzmaßnahme für das zuvor genannte Beispiel nicht mehr. Dies ist beispielsweise der Fall, wenn die CPU eine Transition aus dem Tiefschlafmodus innerhalb der Prologe durchführt und die Transition im gleichen Zuge durch den optional eingewobenen Stromzähler abgerechnet wird. Zwar ist der Kontrollfluss hier nicht unterbrechbar, allerdings kann der Timer dennoch kurz vor dem Auslesen überlaufen, ohne dass der Inhalt von `system_ticks` aktualisiert wird. Die Behandlungsroutine für den Überlauf wird nämlich bis zum Abschluss der aktuellen ISR verzögert (siehe Kapitel 2.4.1.2). Um auch diese Fälle abzufangen, wird nach Zwischenspeichern beider Zähler auf einen eingetretenen Timerüberlauf kontrolliert, denn bei unterdrückten Interrupts setzt der MSP430 ein entsprechendes Bit im Kontrollregister des Timers. Bei Feststellung eines Überlaufs werden die zwischengespeicherten Werte korrigiert.

Sonderfall: Prolog

Zeitreisen im Prolog?...

...auch nicht!

Aus Sicht der Kontrollflüsse auf Benutzer- und Epilog-Ebene ist dieses Signalisierungsbit effektiv nie gesetzt, da es unverzüglich eine Unterbrechungsbehandlung auslöst und in diesem Zuge automatisch gelöscht wird.

5.3 ZWISCHENFAZIT

Die behandelten Abschnitte dieses Kapitels zeigen, dass sich Treiber verschiedener Geräteklassen auf einfache Weise nach dem entworfenen Modell implementieren lassen.

Mit Hilfe des eingewobenen Triggers und der Unterstützung des Messsystems können Transitionen auf Wunsch ohne größeren Aufwand markiert werden, und anhand dessen die Messaufzeichnungen in korrespondierende Intervalle aufgeteilt werden. Dies unterstützt den Entwickler bei der Analyse seiner Modellierung und erlaubt außerdem langfristige Messungen zum Erhalt genauerer Verbrauchsdaten durch Mittelwertsbildung. Bei den implementierten Treibern zeigt sich in den Probemessungen, dass sich das gewählte Modell der PTA mit annotierten Kosten an sowohl Zuständen, als auch Transitionen, mit dem realen Verhalten deckt.

Trigger erleichtert Messung

Es hat sich jedoch auch herausgestellt, dass die Verbrauchsangaben im Datenblatt des Geräteherstellers stellenweise unvollständig oder abweichend vom tatsächlichen Verbrauch sind. Dadurch sind genaue Messungen für ein exaktes Energiemodell unverzichtbar. Dies erfordert bei der Implementierung und insbesondere bei der Messung interdisziplinäres Wissen und Erfahrung aus Informatik und Elektrotechnik, um Messfehler durch interferierende Komponenten oder parasitäre Effekte zu vermeiden. Letztere fallen besonders bei energiesparenden Komponenten – wie die Probemessung des Beschleunigungssensors gezeigt hat – ins Gewicht.

Messungen sind unverzichtbar

Allerdings hat der Treiber für die CPU auch aufgezeigt, dass es Komponenten gibt, die ein dynamisches und komplexes Energieverhalten aufweisen können, welches sich nur mit einem erheblichen Analyse-, Mess- und Rechenaufwand exakt nachbilden lässt. Vor Allem wegen letzterem wird ein solches Modell hinfällig, da mögliches Energiesparpotenzial zur Laufzeit durch die Nachbildung des Energieverbrauchs aufgezehrt wird. In solchen Fällen muss eine vereinfachte Modellierung herangezogen werden, die mit Szenariospezifischen Durchschnittsdaten (wie auch in [BGS12] angewandt, siehe Kapitel 3.3) gefüllt wird. Die ermittelten Werte sind jedoch nicht auf andere Anwendungsfälle übertragbar, sondern müssen individuell neu bestimmt werden.

Die Implementierung der Treiber nach diesem Entwurfsmodell erfordert keine Änderungen am vorhanden Betriebssystem (hier am Beispiel Kratos). Lediglich für den Stromzähler musste die Methode zum Erfassen der Systemzeit verfeinert werden. Das Treibermodell kann parallel mit klassischen Treiberimplementierungen koexistieren. Außerdem kann auf die automatenbasierte Treiberschnittstelle eine Adapterschicht zur klassichen API aufgesetzt werden, sodass darüber liegende Implementierungen nicht angepasst werden müssen.

Treibermodell ist minimalinvasiv

Bei der manuellen Implementierung fällt auf, dass sie zwar sehr einfach nach diesem Modell umgesetzt werden kann, jedoch stellenweise mit aufwendigem und häufig wiederholendem Schreibaufwand bei der Neuimplementierung verbunden ist: Für jede Transition muss ein entsprechender Subtyp von `transition_t`, bzw. der Variante `secure_transition_t` erzeugt werden. Zudem müssen Header- und Implementierungsdatei synchron gehalten werden. Schließlich ist die Problematik des abreißenden Kontrollflusses bei Interrupts in Hinsicht auf die statische Analyse noch nicht behoben, da der Zeitparameter nicht in die Implementierung einfließt. Um diese Lücke zu schließen und eine schnellere Treiberentwicklung zu ermöglichen, werden im Folgenden Automatisierungswerkzeuge vorgestellt, die es erlauben, alle automatenbezogenen Informationen kompakt in einer Datei zu bündeln.

5.4 AUTOMATISIERUNGSWERKZEUGE

Aus den vorangegangenen Abschnitten dieses Kapitels geht hervor, dass sich die Implementierung der Treiber mit Werkzeugunterstützung beschleunigen lässt. Einerseits ergibt sich viel wiederholender Quelltext bei der Definition von Zuständen und insbesondere Transitionen. Andererseits kann auch die Erzeugung eines Evaluierungsprogramms teilweise automatisiert werden, indem alle Transitionen angefahren werden. Diese Werkzeuge sollen nun vorgestellt werden.

wiederholender Quelltext

5.4.1 *Rahmengenerator*

Kernstück des Treibermodells bilden Zustände und Transitionen, die die Energiemodi und Funktionen eines Gerätes beschreiben. Anstatt diese manuell und zerstreut in einem Programmquelltext niederzuschreiben, können die Informationen gebündelt in einer XML-Datei zusammengefasst werden. Abbildung 5.11 demonstriert die Automatendefinition des SPI. Die Definition beginnt mit der Angabe von Zuständen und ihrer Leistungsaufnahme. Anschließend werden Transitionen angegeben, die sich mindestens aus Quelle, Ziel, Energieverbrauch und Anwendungslevel zusammensetzen. Letzteres gibt an, aus welcher Unterbrechungsebene (Prolog, Epilog oder Anwendungsschicht) die Transition ausgelöst wird, um die korrekte Basisklasse des Transitionstypen zuzuordnen. Schließlich kann für zeitabhängige Zustandswechsel die Zeit zum Auslösen dieses Übergangs angegeben werden. Die Existenz dieses Parameters dient auch der Vervollständigung des Aufrufgraphen bei der statischen Analyse, die dazu natürlich alle solchen Definitionen mit berücksichtigen muss.

XML Beschreibung

Aus dieser XML-Definition erzeugt das hierfür entwickelte Python-Skript `dfa-driver-generator.py` den modellgetreuen C++ Rahmen. Dabei werden implizit der Zustand `UNINITIALIZED` und die Transition `trEntry` hinzugefügt. Da die Entwicklung eines Treibers ggf. mehrere Iterationen benötigt, erlaubt das Skript auch das nachträgliche Hinzufügen, Modifizieren und Entfernen von Definitionsteilen. Hierzu wird die XML-Datei entsprechend modifiziert und das Skript erneut gestartet. Falls dabei eine bereits frühere Implementierung existiert, wird der neue Rahmen unter einem abgeänderten Namen gespeichert und ein frei wählbares Programm zum Verschmelzen beider Dateien (hier `vimdiff`) gestartet.

Generierungsskript

Verschmelzen

5.4.2 *Testprogrammgenerator*

Durch die externe XML-Datei sind alle Zustände und Transitionen gegeben. Alternativ können diese Daten direkt aus dem Quelltext gewonnen werden, indem der Code nach entsprechenden Schlüsselwörtern durchsucht wird. Anhand dieser Informationen kann ein Pro-

```
<?xml version="1.0"?>
<data>
  <driver name="DFA_SPI">
    <states>
      <state name="STANDBY" power="0"></state>
      <state name="TRANSFERING" power="0"></state>
    </states>
    <transitions>
      <transition name="" energy="0">
        <src>STANDBY</src>
        <dst>TRANSFERING</dst>
        <level>user</level>
      </transition>
      <transition name="" energy="0">
        <src>TRANSFERING</src>
        <dst>STANDBY</dst>
        <level>epilogue</level>
        <timeout>8/CONFIG_eUSCI_B0_DFA_SPI_BITRATE</timeout>
      </transition>
    </transitions>
  </driver>
</data>
```

Quelltext 5.11: Automatendefinition in Form eine XML-Datei zur Vereinfachung der Implementierung. Die Angabe des *timeout* (Zeile 18) dient der Vervollständigung des Aufrufgraphen für die statische Analyse.

gramm generiert werden, dass alle Transitionen nacheinander durchläuft, um die entsprechenden Verbrauchswerte in einer Messung zu erfassen. Hierzu wurde ebenfalls ein Skript `genWalkthrough.py` entwickelt, das aus der Treiberspezifischen XML Datei ein Testprogramm generiert, das alle Transitionen enthält. Da die XML Beschreibung keine Informationen zu den benötigten Parametern und weiteren Nebenbedingungen enthält, werden allen Transitionen entsprechende Rümpfe vorbereitet, die mit den erforderlichen Parametern gefüllt werden können.

Testprogramm

6 EVALUATION

In diesem Kapitel wird das entworfene Treibermodell evaluiert. Dabei wird zunächst die Modellgenauigkeit bewertet, indem die Zuverlässigkeit der Messdaten bestimmt wird und die Genauigkeit des Profilings analysiert wird. Anschließend werden weitere Automatisierungsmöglichkeiten diskutiert, sowie aufgezeigt, wie die Informationen aus dem Treibermodell bei der Codeanalyse genutzt werden können. Das Treibermodell und die Zusatzkomponenten werden danach hinsichtlich ihres Ressourcenbedarfs analysiert und schließlich die Generalisierbarkeit und die Grenzen des Modells diskutiert.

6.1 MODELLGENAUIGKEIT

Um die Modellgenauigkeit zu evaluieren, sollen die Ergebnisse der softwarebasierten Energiemessung (durch den Profiler) mit dem tatsächlichen Verbrauchswerten in einem länger andauernden Experiment verglichen werden. Zuvor muss jedoch die Güte des noch prototypischen Messsystems (MIMOSA) bewertet werden, da sich im Verlauf der Messarbeiten einige Unstimmigkeiten ergeben haben und jene deswegen häufig wiederholt werden mussten.

Güte des Messsystems

Der zweite vorbereitende Schritt ist die Erfassung der eigentlichen Verbrauchsdaten zu den implementierten Peripheriegeräten. Diese werden durch mehrere Messaufzeichnungen durchgeführt, in denen Transitionen durch den Trigger markiert werden und dadurch automatisch der Mittelwert aus vielen gleichartigen Wiederholungen berechnet werden kann.

Güte des Treibermodells

6.1.1 *Zuverlässigkeit der Messdaten*

Im Verlauf der Messarbeiten mit MIMOSA ist mehrfach aufgefallen, dass die Messwerte bei Wiederholung des gleichen Experiments von einander abwichen. Es bestätigte sich die Vermutung, dass die Resultate abhängig von der Betriebstemperatur von MIMOSA sind. Der Prototyp beinhaltet keine speziellen Maßnahmen zur Kompensation dieser Temperaturdrifts. Da dieses Verhalten bisher nicht quantifiziert wurde, jedoch von großer Wichtigkeit zur Einschätzung der Messwertqualität dieser Arbeit sind, soll dies in Kurzform nun hier erfolgen.

Es werden zwei langfristige Aufzeichnungen nach einem Kaltstart von MIMOSA durchgeführt, ein Mal ohne Last und ein Mal mit der

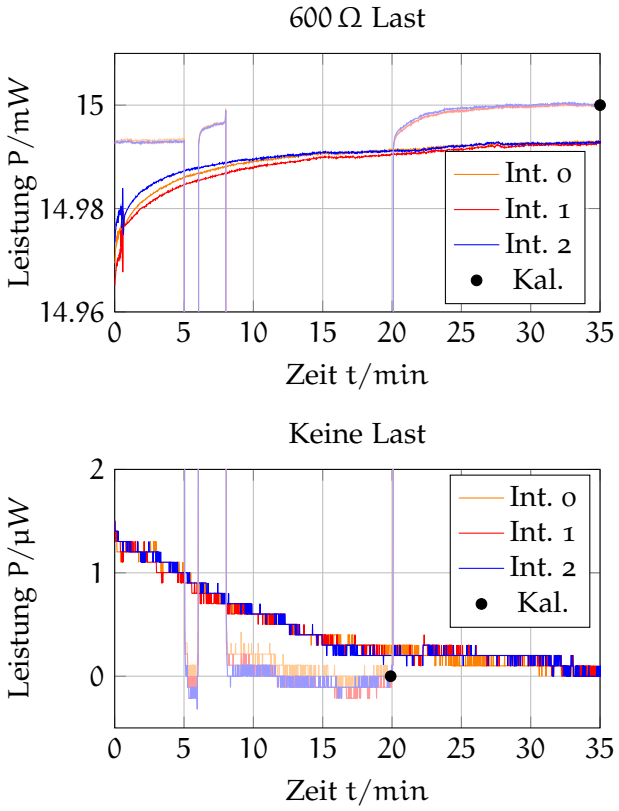


Abbildung 6.1: Temperaturverhalten von MIMOSA nach einem Kaltstart mit 600 Ω Last (oben) und ohne (unten) Last je Integrator. Die blassen Graphen zeigen die Fortsetzung (Warmstart) des obigen Experiments, bei dem zwischenzeitlich die Last entfernt wird. Die Kalibrierungspunkte für 0 mW und 15 mW bei 3 V sind schwarz gekennzeichnet.

konstanten Last von 600 Ω, was bei 3 V einer Verlustleistung von 15 mV entspricht. Pro Sekunde wird der Mittelwert der letzten Sekunde, getrennt nach den drei Integrationsstufen ausgegeben. Die Resultate dieser Aufzeichnungen sind in Abbildung 6.1 dargestellt. Dabei wurde der Wert des jeweiligen Integrierers an der Position des schwarzen Punktes als Referenzgröße für 0 mW bzw. 15 mW gewählt, da sich MIMOSA jeweils auf diesen Niveaus wiederholt einpendelt.

Temperaturdrift

Je nach Last zeigt sich ein Drift nach unten oder nach oben, der sich jeweils einem konstanten Niveau asymptotisch annähert (statischer Messfehler). Die einzelnen Integrationsstufen verhalten sich dabei gleichwertig, fächern sich aber unter Last in den ersten 15 min auf ein ca. 2 µW breites Intervall auf, welches sich jedoch mit der Zeit zunehmend verringert. Ursache für die Driftrichtung sind Temperaturabhängigkeiten verschiedener Bauteile auf der Platine. Beispielsweise verhält sich die Referenzspannungsquelle unabhängig von der angeschlossenen Last, während der treibende Operationsverstärker bereits bei 600 Ω Last merkbar warm wird. Die detaillierte Ursachen-

LAST	MAX. DIFFERENZ	
	ab 0 min	ab 30 min
$\infty\Omega$	1.50 µW	0.20 µW
600 Ω	28.03 µW	1.07 µW

(a) Temperaturdrift

LAST	σ_{NOISE}	t
$\infty\Omega$	**4.752 93 µW**	13.323 s
600 Ω	**9.606 15 µW**	8.80 s

(b) Grundrauschen

Tabelle 6.1: Genauigkeit der Messwerte von MIMOSA. Je nach Last und Betriebsdauer ergibt sich ein Temperaturdrift (links). Zudem ist auch das Grundrauschen lastabhängig (rechts).

analyse ist jedoch nicht Gegenstand dieser Arbeit und wird deswegen bewusst nicht weiter vertieft.

Festzuhalten bleibt jedoch, dass die Messabweichung dadurch lastabhängig ist, wie die Fortsetzung der 600 Ω-Messung aufzeigt. Sie ist in beiden Diagrammen blass eingezeichnet und stellt damit einen Warmstart von MIMOSA dar. Allerdings wurden bei 5 min und 8 min der Lastwiderstand für 1 min und 12 min entfernt. Beim Rückanbringen der Last fällt auf, dass sich die Schaltung jedes Mal neu einpendeln muss (vgl. 6 min und 20 min), da sie zwischenzeitlich abgekühlt ist (dynamischer Messfehler).

lastabhängiger Fehler

Für bestmögliche Messergebnisse sollte die Schaltung vor der Messung etwa für eine halbe Stunde warmlaufen, mindestens jedoch für 15 min. Um den Messfehler in Abhängigkeit von der gemessenen Last abschätzen zu können, werden die größten aufgetretenen Messwertdifferenzen ab 0 min bzw. 30 min Betriebszeit jeweils ohne und mit Last als Referenzgrößen genommen und linear interpoliert. Basierend auf den ermittelten Ausgangsgrößen in Tabelle 6.1a beträgt der abgeschätzte Messfehler ε dann

halbe Stunde Aufwärmzeit

$$\varepsilon_{t\geqslant 0\,min}(P) = 28.03\,\mu W \cdot P + 1.5\,\mu W, \tag{6.1}$$

$$\varepsilon_{t\geqslant 30\,min}(P) = 1.07\,\mu W \cdot P + 0.2\,\mu W. \tag{6.2}$$

Die Abschätzung aus Gleichung 6.2 kann jedoch nur für langfristig konstante Leistungsaufnahmen verwendet werden. Springt die Leistungsaufnahme des DUT, so ist wegen der Variation die Gleichung 6.1 für die Abschätzung zu verwenden.

Eine weitere Referenzgröße ist das Grundrauschen der Messwerte je angeschlossener Last. Aus Kalibrierungsmessungen von jeweils t Sekunden mit $\infty\Omega$ und 600 Ω geht die in Tabelle 6.1b aufgeführte Standardabweichung σ_{noise} für das Grundrauschen hervor.

Grundrauschen

Zum Schluss soll der Effekt der Dekalibrierung von MIMOSA aufgezeigt werden. Hierbei driften die Null-Niveaus der einzelnen Integrationsstufen auseinander, wenn MIMOSA über mehrere Stunden hinweg betrieben wird, wie in Abbildung 6.2 zeigt. Der Effekt tritt auch auf, wenn MIMOSA zwischenzeitlich ausgeschaltet wird. Daher

Dekalibrierung

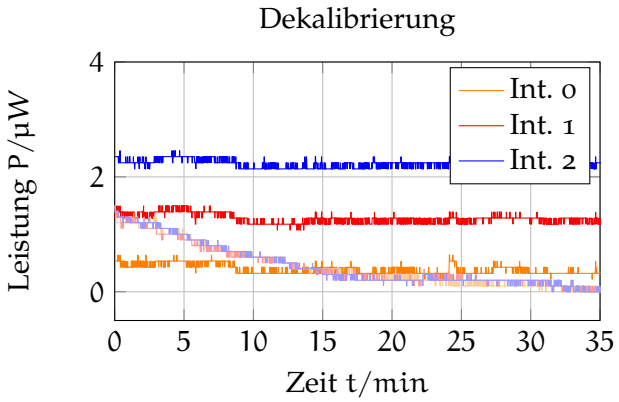


Abbildung 6.2: Dekalibrierung der Integrationsstufen nach der Durchführung von mehreren Messungen für die nachfolgenden Experimente. Integratoren pendeln sich auf einem neuen Niveau ein, verlassen diese Stufe jedoch nicht. Zur Orientierung ist der Verlauf eines Kaltstarts blass eingetragen.

ist es für die Messungen unabdingbar, stets vor einer Messaufzeichnung eine Kalibrierung durchzuführen (siehe Kapitel 2.3.1.4).

6.1.2 *Akquisition der Verbrauchsdaten*

Detailmessung von Display und CPU

Um das Treibermodell mit entsprechenden Verbrauchswerten zu füllen und anschließend bewerten zu können, müssen nun detaillierte Messungen für die zu untersuchenden Komponenten durchgeführt werden. Dies Umfasst das Display und die CPU. Da sich der Verbrauch des SPI bei hohen Übertragungsraten und kleinen Datenmengen nicht von der CPU separieren lässt, die CPU jedoch ohnehin auf szenariobasierte Durchschnittswerte angewiesen ist, fließt das SPI mit in die Durchschnittwerte der CPU ein. Gleiches würde auch auf andere integrierte Peripheriekomponenten innerhalb des Mikrocontrollers zutreffen. Aufgrund der Beschädigung des InBin bei parallelen Forschungsarbeiten konnte der Beschleunigungssensor nicht weiter berücksichtigt werden.

6.1.2.1 *Display*

Spannung: 3.2 V

Zur detaillierten Bestimmung der Verbrauchswerte des Displays werden mehrere Experimente unterschiedlicher Länge durchgeführt, die jeweils wiederholt Zustandsübergänge innerhalb des Displays auslösen. Das Display wird dabei von MIMOSA mit 3.2 V Betriebsspannung versorgt, während das Launchpad mit einer gleich hohen Batteriespannung gespeist wird. Die Höhe der Versorgungsspannung entspricht damit der gemessenen Spannung von zwei in Serie geschalteten Alkaline-Batterien.

Zur Plausibilitätsprüfung sind alle Experimente zusätzlich sowohl mit vertauschten Spannungsquellen, als auch mit ausschließlicher Versorgung durch MIMOSA wiederholt und gleichermaßen ausgewertet. Die Differenzen dieser Zusatzmessungen ergeben im Rahmen des Rauschens eindeutige Übereinstimmungen mit den Einzelmessungen des Displays – werden hier jedoch nicht weiter vertieft. Diese Ergebnisse bestätigen jedoch, dass im Display keine nennenswerten parasitären Effekte auftreten, wie sie etwa bei der Implementierung des Beschleunigungssensors (siehe Kapitel 5.1.4.2) aufgefallen sind.

Die ermittelten Verbrauchswerte für Zustände und Transitionen sind in Tabelle 6.2 angegeben. Der Energieverbrauch von Transitionen ist als arithmetisches Mittel $\overline{E}$ angegeben (siehe Tabelle 6.2a). Zusätzlich sind die mittlere Transitionsdauer $\overline{t}$, die Anzahl der berücksichtigen Transitionen des jeweiligen Typs n, sowie die Streuung des Energieverbauchs in Form der Standardabweichung σ dokumentiert.

Zur Bestimmung der zustandsbezogenen Leistungsaufnahme wird pro Verweilzeit in einem Zustand die mittlere Leistungsaufnahme während dieses Intervalls bestimmt (siehe Tabelle 6.2b). Der Durchschnitt dieser mittleren Intervallleistungen ist jeweils als arithmetisches Mittel $\overline{P}$ angegeben. Neben Intervallanzahl n und durchschnittlicher Intervalllänge $\overline{t}$ ist die Streuung der Intervallmittelwerte in Form der Standardabweichung σ_{outer} angegeben. Sie ist ein Maß dafür, wie stark die einzelnen Intervalle von einander abweichen. Zusätzlich ist jedoch auch die durchschnittliche *innere* Streuung $\overline{\sigma}_{inner}$ dokumentiert. Sie gibt an, wie stark die Leistungsaufnahme während der Aufenthaltsdauer in einem Zustand variiert. Große Werte sind hierbei ein Hinweis auf unzureichende Modellierung, können jedoch auch durch Betriebsrauschen dieses Zustands verursacht werden.

äußere und innere Streuung

Die Ergebnisse in Tabelle 6.2 zeigen jedoch äußerst plausible Resultate auf. Der deutliche Ausreißer bei `OFF2DISA` ergibt sich durch das Aufladen der Abblockkondensatoren auf der Displayplatine. Zur Bestimmung dieser Größe musste die Versorgungsspannung des Displays manuell an MIMOSA an- und abgesteckt werden, sodass die Anzahl der Wiederholungen für diesen Fall deutlich geringer ausfällt. Bei den übrigen Messwerten wurde die Aufzeichnung durch Berücksichtigung des Triggersignals aufgeteilt und automatisiert ausgewertet. Für Zustände und Transitionen, in denen die Versorgungsspannung abgetrennt ist, sind die Kosten jeweils auf 0 gesetzt, da hierbei keine sinnvolle Messung möglich ist. Schließlich zeigt die Betrachtung von $\overline{\sigma}_{inner}$, dass ein angeschlossenes Display das Grundrauschen erhöht (im eingeschalteten Zustand stärker). Eine Sichtung der Messaufzeichnungen bestätigt dies mit einer merkbar erhöhten Rauschamplitude. Ausreißer (d. h. Hinweise auf unzureichende Modellierung) sind nicht feststellbar. Dies wird auch durch die äußerst geringe Streuung der Intervallmittelwerte σ_{outer} bestätigt.

erneut Abblockkondensatoren

TRANSITION	$\overline{E}$	σ	$\overline{t}$	n
ENTRY	**0 nJ**	–	–	–
OFF2DISA	**528.235 nJ**	76.504 nJ	100 µs	4
DISA2OFF	**0 nJ**	–	–	–
DISA2ENA	**33.4066 nJ**	0.593 813 nJ	106.5 µs	200
ENA2DISA	**6.562 22 nJ**	0.513 448 nJ	100.85 µs	200
ENA2ENA_LINE	**3.8042 nJ**	1.315 55 nJ	316.824 µs	3168
ENA2ENA_VCOM	**74.083 nJ**	1.453 27 nJ	150.024 µs	2056
ENA2ENA_CLEAR	**22.4637 nJ**	0.716 583 nJ	176.3 µs	200

(a) Energieverbrauch je Transition.

ZUSTAND	$\overline{P}$	σ_{OUTER}	$\overline{\sigma}_{INNER}$	$\overline{t}$	n
U	**0 µW**	–	–	–	–
OFF	**0 µW**	–	–	–	–
DISA	**18.6954 µW**	0.8524 µW	12.4882 µW	99.645 ms	200
ENA	**19.2687 µW**	0.535 282 µW	41.3238 µW	100.479 ms	200

(b) Leistungsaufnahme je Zustand.

Tabelle 6.2: Messergebnisse der detaillierten Verbrauchsanalyse vom Display für Transitionen (oben) und Zustände (unten).

6.1.2.2 *CPU*

Wie bereits in Kapitel 5.1.2.3 festgestellt wurde, variiert die Leistungsaufnahme der CPU szenariobasiert, beispielsweise in Folge der Trefferrate im FRAM-Cache. Zur Bestimmung der Verbrauchswerte wird deswegen ein zum nachfolgenden Experiment (Abschnitt 6.1.3) ähnliches Testprogramm verwendet. Einschränkend werden in diesem Zusammenhang für die Transitionen in den Schlafmodus und zurück keine Energiemessungen durchgeführt. Dies liegt insbesondere darin begründet, dass keine exakte Markierung des Aufwachprozesses[1] durch den Trigger möglich ist, da die Signalisierung erst *nach* dem Erwachen erfolgt. Die Dauer zwischen dem Eintreten eines Unterbrechungsereignisses und der ersten ausgeführten Instruktion kann ebenfalls nicht auf diesem Wege ermittelt werden. Sie erfordert einen speziellen Messaufbau mit externer Unterbrechungsquelle, sowie ein Messgerät, das zwei Triggersignale mit gleichzeitiger Energiemessung verarbeiten kann. Zwar ließen sich diese Werte manuell mit einem Oszilloskop bestimmen, jedoch zeigt die Analyse der Zustandsübergänge keine Verbrauchsspitze, die über den Rahmen der *normalen* Leistungsaufnahme im Aktivmodus hinausgeht (vgl. Abbildungen 5.2b, 5.4 und Tabelle 5.2). Stattdessen fließen mögliche Verbrauchswerte mit in die szenariobasierten mittleren Leistungswerte der Zustände

Markierungsproblem

1 Umgekehrt gilt das Gleiche auch für die Transition in den Energiesparmodus.

ZUSTAND	$\overline{P}$	σ_{OUTER}	$\overline{\sigma}_{INNER}$	$\overline{t}$	n
U	**0 µW**	–	–	–	–
AM	**5283.08 µW**	56.759 µW	787.0 µW	0.310 026 ms	383
LPM1	**834.449 µW**	544.598 µW	303.846 µW	9.786 11 ms	357

Tabelle 6.3: Messergebnisse der detaillierten Verbrauchsanalyse der CPU im Hinblick auf das Experiment in Abschnitt 6.1.3. Da mit dem Trigger bei der CPU keine exakte Markierung von Start und Ende der Zustandsübergänge realisierbar ist, fließt ein kleiner Verschnitt mit in benachbarte Zustände ein.

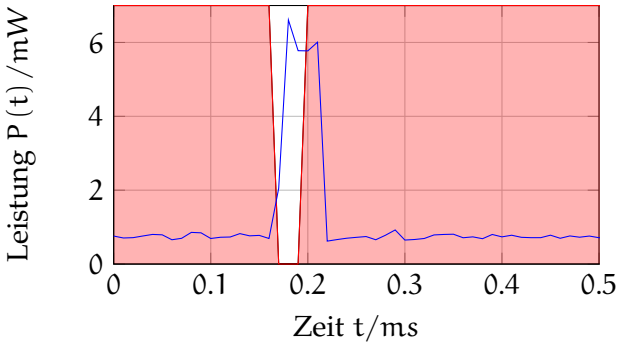


Abbildung 6.3: Ausschnitt aus einer Evaluationsmessung zur Bestimmung der CPU-Leistungsaufnahme im LPM1 (rot hinterlegt). Der Übergang in den Schlafmodus erfolgt erst nach der Signalisierung, sodass ein kleiner Verschnitt des Aktivmodus mit in Durchschnitt des LPM1 einfließt.

ein. Die Resultate dieser Messungen finden sich in Tabelle 6.3. Die Betriebsspannung wird zur Vereinheitlichung mit dem Display auf 3.2 V gesetzt. Als Referenzszenario wird die Leistungsaufnahme für den LPM1 während einer Schlafphase eines Threads gemessen und der Verbrauch des Aktivmodus bei der Übertragung via SPI bestimmt. Die Zustandswechsel werden durch den automatisch eingewobenen Trigger markiert, sodass jeweils nur die relevanten Abschnitte berücksichtigt werden. Abbildung 6.3 zeigt einen Ausschnitt aus zwei benachbarten Schlafintervallen, von denen insgesamt 357 gemittelt wurden. Der rote Bereich kennzeichnet dabei den vom Trigger markierten Schlaf-Abschnitt, der durch einen Timerinterrupt (Scheduler) unterbrochen wird. Hier zeigt sich die unscharfe Trennung der Zustandsübergänge bei der CPU, die zwar den Schlafmodus signalisiert, diesen Modus jedoch erst kurz danach betritt. Durch diese Unschärfe hebt sich der gemessene durchschnittliche Verbrauch im LPM1 von den im Bild erkennbaren 730 µW auf die in Tabelle 6.3 aufgeführten 834 µW. Damit steigt auch σ_{outer} signifikant an.

unscharfe Markierung = größere Streuung

Auf gleiche Weise, jedoch mit umgekehrter Richtung, wirkt sich die unscharfe Markierung auf die Bestimmung des durchschnittlichen Energieverbrauchs im Aktivmodus aus. Hier ergibt die Messung den Wert von 5283 µ*W*.

6.1.3 *Genauigkeit des Profilers*

Nachdem die Modellparameter für Display und CPU, sowie die Rahmenbedingungen von MIMOSA bekannt sind, soll nun die Erfassungsgenauigkeit des Profilers zur Laufzeit bestimmt werden. Dies gibt Aufschluss über die Modellgüte und zeigt auf, wie repräsentativ die errechneten Werte sind.

Es werden der Energieverbrauch von Display und CPU sowohl einzeln, als auch getrennt voneinander über einen längeren Zeitraum mitgezählt und parallel dazu durch MIMOSA (bei 3.2 V) vermessen. Das SPI wird aufgrund der vom Display geforderten Übertragungsrate von 1 Mbit/s und der damit vernachlässigbaren Zeitspanne für die Übertragung (vgl. Kapitel 5.1.1.3 und Abbildung 5.2a) nicht separat aufgeführt, sondern mit durch den Verbrauch des Aktivmodus der CPU abgedeckt.

Textausgabe auf Display

Das aufgespielte Testprogramm gibt fortlaufend eine Folge wechselnder Sätze in Textform auf dem Bildschirm aus. Ältere Zeichenfolgen werden dabei, wie bei einer Konsolenausgabe am Computer, nach oben geschoben (engl. *scrolling*). Der Bildschirm wird zusätzlich regelmäßig gelöscht, aus- und wieder eingeschaltet, sowie phasenweise in seinem Zustand belassen. Das Experiment wird periodisch angehalten und der aktuelle Stand des Stromzählers über UART an einen Computer übertragen. Gleichzeitig wird die Unterbrechung an MIMOSA signalisiert, das die Aufzeichnung während der Übertragung anhält und den bisherigen Zwischenstand als Sample ablegt. Eine erneute Signalisierung setzt das Experiment fort. Dadurch wirkt sich die Übertragung per UART nicht auf die Gesamtmessung aus.

zwei Etappen

Ferner ist das Experiment in zwei Abschnitte unterteilt: Zu Beginn (in den ersten 0.45 min) werden nacheinander drei Hauptszenarien für CPU und Display durchgeführt, bei denen der Bildschirm eingeschaltet wird und unverändert bleibt (0 min bis 0.175 min); fortlaufend auf das Display geschrieben wird (0.175 min bis 0.275 min); und der Bildschirm schließlich gelöscht und ausgeschaltet wird (0.275 min bis 0.445 min). In den drei Szenarien dominieren jeweils unterschiedliche Zustände und Transitionen und erlauben damit genauere Rückschlüsse auf die Ursache von Abweichungen. Jenseits von 0.445 min beginnt der zweite Abschnitt.

Der zweite Abschnitt des Experiments dauert ca. 30 min und beschreibt fortlaufend das Display. Pro Messpunkt werden drei Sätze

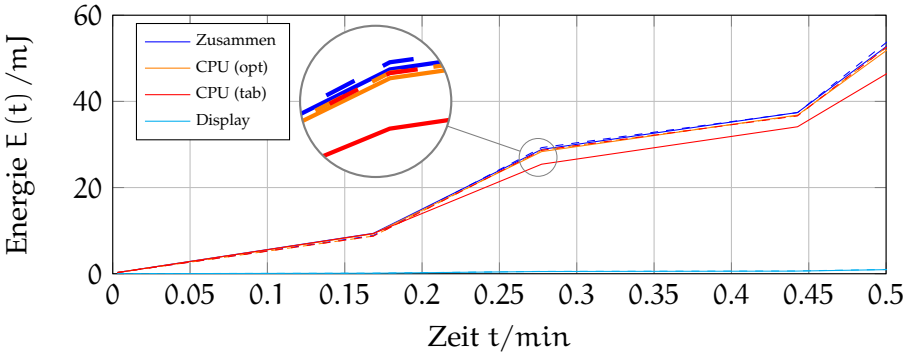


Abbildung 6.4: Energieverbrauch während eines Experiments mit Display und CPU, ermittelt durch den Profiler (durchgezogen) und gleichzeitige Referenzmessung mit MIMOSA (gestrichelt). Der geringe Displayverbrauch ist in Abbildung C.2 deutlicher aufgeführt.

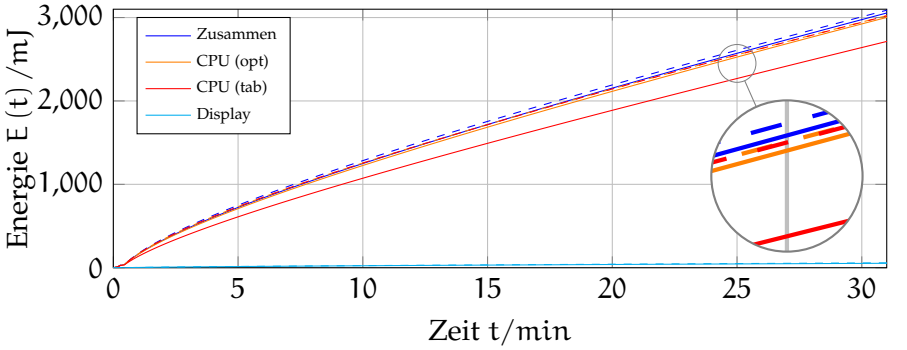


Abbildung 6.5: Fortsetzung des Experiments aus 6.4, ermittelt durch den Profiler (durchgezogen) und gleichzeitige Referenzmessung mit MIMOSA (gestrichelt). Der geringe Displayverbrauch ist in Abbildung C.3 deutlicher aufgeführt.

auf dem Bildschirm ausgegeben[2], wobei nach jedem Satz eine Schlafphase durchgeführt wird. Die Dauer dieser Schlafphasen wird pro Iteration ausgehend von 10 ms in Schritten von 10 ms bis 1 s erhöht. Dieser Abschnitt zeigt somit einen kontinuierlichen Übergang zwischen Schreib- und Konstantphase und soll die Modelltreue bei variierenden Gewichtungen evaluieren.

Die Ergebnisse der Aufzeichnungen sind in Abbildung 6.4 (erster Abschnitt) und Abbildung 6.5 (zweiter Abschnitt) aufgeführt. Abbildungen des zweiten Abschnitts enthalten im Folgenden stets auch den ersten Abschnitt in verkleinerter Form. Die jeweils untersuchten Komponenten werden durch Farben unterschieden (siehe Legende[3]).

2 Durch das Scrollverhalten wird stets der gesamte Bildschriminhalt mehrfach pro mehrzeiligem Satz aktualisiert.

3 Die Legende wurde absichtlich aus Platzgründen kompaktifiziert.

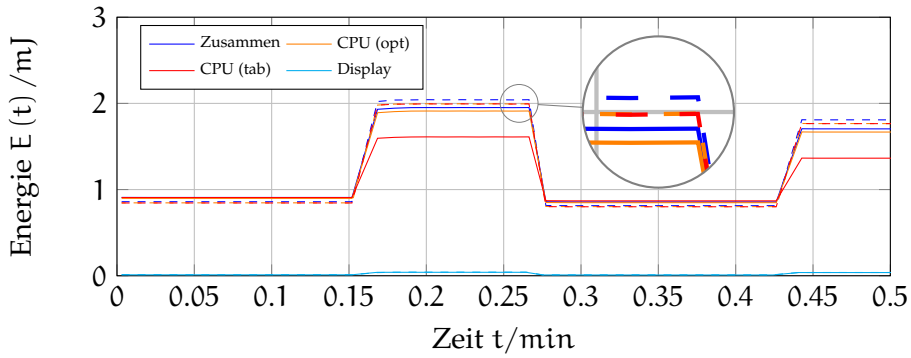


Abbildung 6.6: Zunahme des Energieverbrauchs zwischen den Messintervallen während des Experiments aus Abbildung 6.4. Resultate des Profilers sind durchgezogen, Messergebnisse durch MIMOSA sind gestrichelt. Die Fortsetzung befindet sich in Abbildung C.4.

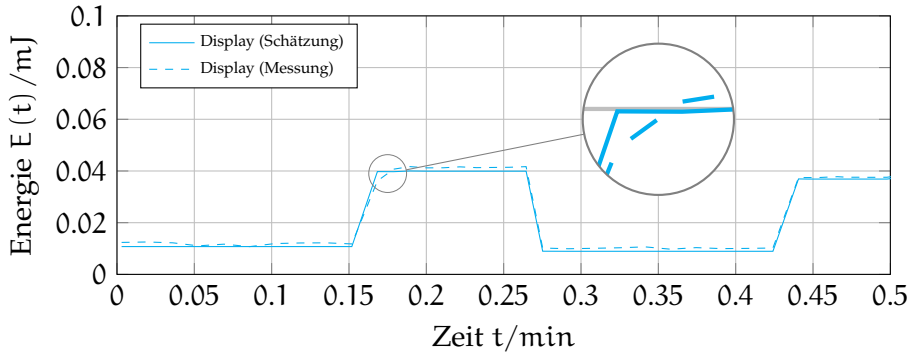


Abbildung 6.7: Zunahme des Energieverbrauchs aus Abbildung 6.6, skaliert für das Display. Die Fortsetzung befindet sich in Abbildung C.5. Die Kurze zur Messung ist von einem leichten Rauschen betroffen, das vom Display ausgeht.

Durchgezogene Graphen geben die geschätzten Werte aus dem Profiler an; gleichfarbig gestrichelte Linien stehen für den korrespondierenden realen Verbrauch (MIMOSA Messung).

CPU: *Abweichung in Aktivphase*

Bei der Betrachtung des CPU-Verbrauchs (rot) zeigt sich, dass geschätzte und gemessene Energie von einander abweichen. Die Abweichung baut sich jedoch fast ausschließlich während der Schreibphase des Displays auf. Während der ruhenden Phasen verlaufen die Kurven jedoch parallel. Alternative Darstellungen des ersten[4] Abschnitts in Abbildung 6.6 und 6.7 verdeutlichen diesen Sachverhalt. Die Graphik zeigt die Energie*zunahme* seit dem letzten Sample. Da offenbar während der CPU-lastigen Schreibphase ein zu geringer Wert

4 Der zweite Abschnitt ist wegen geringerer Relevanz in den Anhang ausgelagert: Abbildung C.4 und C.5.

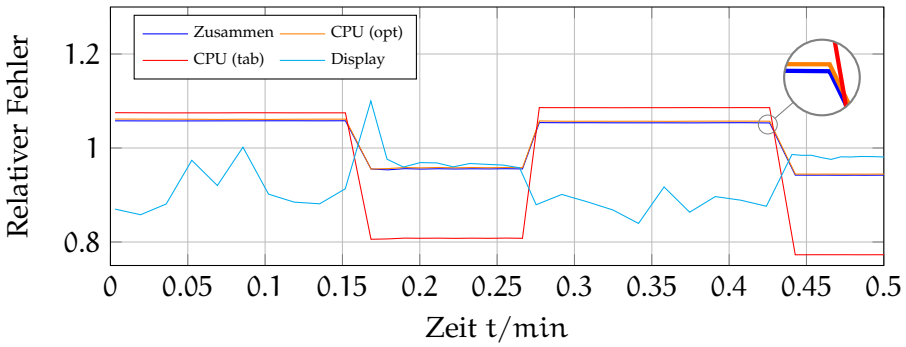


Abbildung 6.8: Relativer Schätzungsfehler gemäß Gleichung 6.3 in der ersten Phase des Experiments. Wegen der geringeren Leistungsaufnahme des Displays erscheint der Fehler stärker verrauscht.

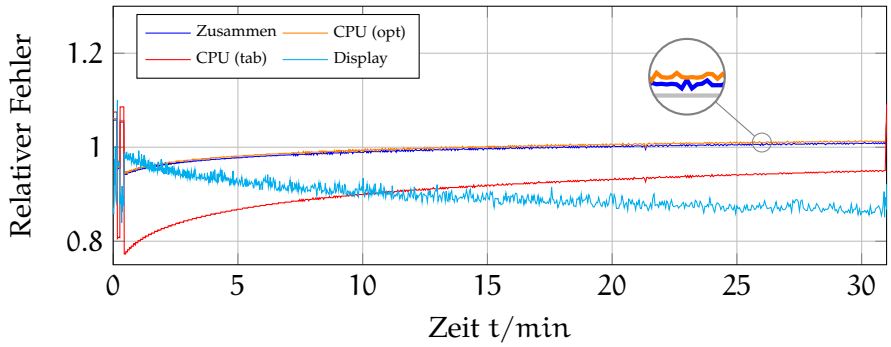


Abbildung 6.9: Relativer Schätzungsfehler gemäß Gleichung 6.3 in der zweiten Phase des Experiments.

geschätzt wird, scheint der ermittelte Wert für den Aktivmodus der CPU (5283 µW, vgl. Tabelle 6.3) zu klein zu sein. Durch weitere Versuchsläufe stell sich ein optimierter Wert von 6600 µW als repräsentativ dar (orange). Der Verlauf deckt sich in allen Phasen fast vollständig mit den Messergebnissen. Ursache hierfür ist der bereits festgestellte Verschnitt (Abbildung 6.3), da das beim Profiling berücksichtigte Intervall kürzer ausfällt, als die CPU tatsächlich aktiv ist. Insbesondere wird der Verschnitt zusätzlich durch die arithmetischen Operationen bei der Abrechnung der Transitionen gestreckt. Dieser Effekt beschränkt sich jedoch ausschließlich auf die CPU, da sie als einzige Systemkomponente ihren eigenen Verbrauch berechnen muss.

Ursache: Verschnitt

Beim Display (cyan) zeigt sich ebenfalls eine hohe Modelltreue, auch wenn eine geringe Unterschätzung des Energieverbrauchs stattfindet. Zur genaueren Untersuchung ist der abschnittsweise auftre-

tende Schätzungsfehler ε_{est} aller Aufzeichnungen in Abbildung 6.8 und 6.9 aufgeführt. Er berechnet sich nach

$$\varepsilon_{est}(n) = \frac{e(n) - e(n-1)}{m(n) - m(n-1)}, \tag{6.3}$$

wobei $e(n)$ und $m(n)$ den n-ten Schätzwert respektive Messwert aus den Aufzeichnungen wiedergeben. Beim Display bestätigt sich die stetige Unterschätzung, die in Ruhephasen um den Faktor 0.9 pendelt und während der Aktivphase bei ca. 0.96 liegt (Abbildung 6.8). Das gleiche Verhalten zeigt sich auch während der längeren Aufzeichnungsphase (Abbildung 6.8), bei der sich die Exaktheit mit abnehmendem Schreibanteil verringert – jedoch in Phasen intensiver Nutzung annähernd 1 erreicht. Ferner zeigt das Display im Vergleich zur CPU ein deutlich erhöhtes Rauschendes Fehlers. Die Standardabweichung dieser Streuung liegt jedoch bei ca. 0.578 µW und deckt sich deutlich mit dem zum Display ermittelten σ_{outer} aus Tabelle 6.2b.

sichtbares Rauschen beim Display

Die Analyse des CPU-Schätzungsfehlers lässt die letzte Aussage zunächst wie einen Widerspruch wirken. Denn obwohl hierbei ein sehr hohes σ_{outer} (56.8 µW bis 544.6 µW, vgl. Tabelle 6.3) vorliegt, wirkt der Fehler abschnittsweise annähernd konstant. Allerdings ergibt sich σ_{outer} im Aktivmodus durch die Betrachtung vieler Aktivitätsintervalle mit unterschiedlichster Leistungsaufnahme. Je nachdem, welche Arten von Instruktionen ausgeführt werden und wie hoch die Trefferquote im Cache ist, ergibt sich eine andere Leistungsaufnahme je Intervall (vgl. Tabelle 5.2). Beim Energiesparmodus resultiert diese Streuung aus dem Verschnitt und dessen Anteil an der (unterschiedlich lang andauernden) Schlafdauer. Wird hingegen – wie in dem Experiment – eine identische Abfolge von Instruktionen und Schlafmodi mehrfach durchlaufen, ergibt sich zwischen den Durchläufen nur eine minimale Streuung von 0.17 µW bis 4.5 µW[5], da die gleiche Instruktionsfolge jeweils gleich viel Energie verbraucht. Durch den höheren Absolutverbrauch wirkt sie sich zudem vernachlässigbar auf den Fehler aus. Im Display hingegen entsteht σ_{outer} aus einem Zufallsprozess im Display und ergibt beim Vergleich von zwei Intervallen unterschiedliche Werte (im Rahmen der Streuung). Zusätzlich ergibt sich ein geringer Absolutverbrauch, sodass die Streuung sichtbar auffällt.

Widerspruch?...

...nein

Bei der Betrachtung der gleichzeitigen Schätzung/Messung von CPU und Display (blau) zeigt sich, dass der Verlauf (aus offensichtlichen Gründen) durch die im Vergleich zum Display energieintensive CPU dominiert wird. Beim relativen Fehler überlagern sich die jeweiligen Schätzungsfehler, gewichtet mit dem relativen Anteil am gesamten Energieverbrauch (vgl. Abbildung 6.8 und 6.9 in orange und blau).

CPU dominiert

5 Abschnittwsweite errechnet aus den Messergebnissen in Abbildung 6.6 und 6.7.

KOMPONENTE	MIN.	MAX.
Zusammen	0.1 %	5.6 %
CPU (optimiert)	0.01 %	5.9 %
CPU (aus Tabelle)	1.3 %	18.6 %
Display	2.9 %	9.8 %

Tabelle 6.4: Minimal und maximal aufgetretener Fehler während des gesamten Experiments (kumuliert ab Beginn der Messung an: Gleichung 6.4).

6.1.4 *Bewertung*

Insgesamt ergibt sich experimentell eine hervorragende Übereinstimmung zwischen Online-Energiemodell und tatsächlichem Verbrauch. Sowohl in kurzfristigen, als auch langfristigen Zeitspannen bleiben Modell und Realität synchron. Tabelle 6.4 zeigt den minimal und maximal aufgetretenen Fehler während der Aufzeichnung. Dazu wird nicht der abschnittsweise Fehler aus Abbildung 6.8, 6.8 bzw. Gleichung 6.3 verwendet, sondern die absolute Abweichung von Beginn der Aufzeichnung an:

Modell bleibt realitätskonform

$$\varepsilon_{\text{total}[\%]}(n) = \left| \frac{e(n)}{m(n)} - 1 \right| \cdot 100. \tag{6.4}$$

Diese gibt Aufschluss darüber, wie zuverlässig die Online-Werte im langfristigen Einsatz sind. Mit den optimierten Werten für die CPU ergibt sich im Experiment beim gemeinsamen Profiling eine maximale Abweichung von 5.6 %. Das Display erzeugt hingegen eine maximale Abweichung von fast 10 %, kann jedoch auf gleichem Wege wie die CPU weiter optimiert werden, da die messtechnisch ermittelten Werte zu einer stetigen Unterschätzung des Verbrauchs führen (siehe Abbildung 6.8 und 6.9), die Unterschätzung jedoch in Ruhephasen besonders ausgeprägt ist.

Für die CPU verschlechtert sich die Qualität der Messwerte in Folge des Verschnitts signifikant, sodass es zu einer maximalen Abweichung von 18.6 % kommt. Zur Evaluation solcher Messwerte muss deswegen in zusätzlichen Probeläufen ein Vergleich aus realem und geschätztem Verbrauch durchgeführt werden.

Probeläufe sind unverzichtbar

Grundsätzlich hat sich die Akquisition des Verbrauchsdaten als besonders fehleranfällig herausgestellt und erfordert deswegen höchste Sorgfalt bei den Messarbeiten. Im Rahmen dieser Arbeit ist dies durch das prototypische Messsystem erschwert worden, da es in Folge des Temperaturverhaltens zu statischen und dynamischen Messfehlern kommen kann, sowie vor jeder Messung eine Neukalibrierung erforderlich ist. Durch den Einsatz des Triggers können aller-

dings eine Vielzahl von relevanten Intervallen gemittelt werden, um genauere Messwerte zu erhalten.

Insgesamt zeigt sich jedoch mit zuverlässigen Messdaten eine hohe Modelltreue über verschiedene Einsatzszenarien hinweg.

6.2 AUTOMATISIERBARKEIT

Testprogramm

Eines der Ziele dieses Treiberentwurfs ist die erleichterte Vermessung der getriebenen Hardwarekomponenten. Dies erfordert ein Testprogramm, das den Treiber (und damit auch das verbundene Gerät) durch seine Energiemodi führt. Die Erstellung dieses Testprogramms kann mit Hilfe von Skripten (teilweise) automatisiert werden (siehe Kapitel 5.4).

Da das Modell allerdings für parametrisierte Transitionen keine Zusatzinformationen (wie Standardwerte) enthält, obliegt es dem Entwickler, das Testprogramm mit diesen Daten zu vervollständigen und ggf. die Reihenfolge von Transitionen anzupassen.

automatische Durchlauffolge

Die automatische Erzeugung einer Durchlauffolge mag zwar auf den ersten Blick vielversprechend klingen, ist bei diesem Modell jedoch aufgrund der großen Vielfalt von Peripheriegeräten und ihrer spezifischen Eigenschaften nicht pauschal realisierbar. Einerseits kann es für die Messaufzeichnung sinnvoll sein, bestimmte Transitionen unmittelbar wiederholt durchzuführen, anstatt das Gerät durch alle Betriebsmodi zu schicken. Andererseits dürfen bestimmte Transitionen nicht zu häufig – aber auch nicht zu selten ausgeführt werden (z. B. das VCOM-Signal beim Display). Um auch diesen Punkt automatisieren zu können, müssten umfangreichere Modelle wie z. B. aus [TBB+13] herangezogen werden, die die vollständige Funktionalität einschließlich aller Zusatzbedingungen berücksichtigen.

Einschränkungen durch Messsystem

Die Automatisierbarkeit wurde im Rahmen dieser Arbeit zusätzlich stark vom Messsystem eingeschränkt, da es in seiner aktuellen Version nur ein Signalisierungsbit bereitstellt. Eine Ausweitung auf mehrere Bits ermöglicht eine Kodierung der genommenen Transition, sodass die Messaufzeichnung automatisch unterteilt und ausgewertet werden kann.

Die hier aufgeführten Messwerte sind deshalb durch wiederholte Messungen mit expliziter Signalisierung (siehe Trigger in Kapitel 4.3.1) an jeweils einer betrachteten Transition ermittelt. Alternativ sind Intervalle (sofern möglich) anhand ihrer charakteristischen Länge gefiltert, sodass mehrere Transitionen in einer Aufzeichnung untersuchbar sind.

6.3 CODEANALYSE

In Kapitel 4.1 wurde bereits erläutert, dass die statische Codeanalyse bei der Verwendung von klassischen Treibern an keiner Stelle

Informationen über den Energieverbrauch von Hardwareaktivitäten, bzw. deren momentane Leistungsaufnahme gewinnen kann. Dadurch kann keine Abschätzung hinsichtlich des zu erwarteten Energieverbrauchs erfolgen. Zudem ergibt sich bei interruptbasierten Peripheriegeräten ein Abriss des Kontrollflusses (Abbildung 4.1a), da die ISR nicht im Aufrufgraphen einer Treibermethode auftaucht. Dieser Abschnitt zeigt nun, dass diese Lücken mit dem entworfenen Treibermodell geschlossen werden können. Da die statische Codeanalyse ein äußerst umfangreiches Forschungsfeld ist, beschränkt sich die Betrachtung in dieser Arbeit auf Annotation und Vervollständigung des Aufrufgraphen, der aus dem *Repository* des `ag++`-Compilers generiert werden kann. Dadurch soll eine Grundlage für künftige Forschungsarbeiten geschaffen werden, die einen annotierten und vollständigen Kontrollfluss voraussetzen.

Repository

In Abbildung 6.10 ist exemplarisch ein Auszug zweier Aufrufgraphen bei der Verwendung des automatenbasierten SPI-Treibers dargestellt. Links im Bild greift ein Benutzerprozess (oder alternativ ein darauf aufbauender Treiber) auf die Adaptermethode `send(...)` zu, die eine Transition `trSTANDBY2TRANSFERING` anstößt. Diese wechselt den internen Zustand, initiiert die Datenübertragung und verharrt bis zum Abschluss des Transfers wartend auf eine Semaphore. Aus Sicht der Codeanalyse kann an dieser Stelle nicht abgeschätzt werden, zu welchem Zeitpunkt der Prozess fortgesetzt wird und `passTransition` zurückkehrt. Dies ist ausschließlich vom rechts abgebildeten – nicht zusammenhängenden – Kontrollfluss abhängig. Er wird in Folge der ISR ausgelöst, wenn die SPI-Übertragung abgeschlossen ist und gibt lediglich die gesperrte Semaphore wieder frei. Der Aufrufzeitpunkt der ISR kann bei der Analyse nicht ohne Zusatzinformationen ermittelt werden, da er aus dem Quelltext nicht hervorgeht.

An dieser Stelle werden die Transitionsinformationen und Zeitparameter des PTA relevant, die in der zugehörigen XML-Datei festgehalten sind (siehe Quelltext 5.11 und Kapitel 5.4.1). Daraus geht eindeutig hervor, dass bei der Ausführung von `trSTANDBY2TRANSFERING`[6] der Zustand `TRANSFERING` betreten wird und nach Ablauf von `<timeout>` die Transition `trTRANSFERING2STANDBY` wieder in `STANDBY` zurückkehrt. Mit diesen Informationen kann der Aufrufgraph um eine Verzögerungskante (rot) und eine gewöhnliche Kante (blau) ergänzt werden.

Zusätzlich enthält die XML-Datei die assoziierten Transitionskosten (Energie) und die Zustandskosten (Verlustleistung), die an den Aufrufgraphen annotiert werden können (orange). Somit hat die Codeanalyse beim Erreichen der genormten und einheitlichen Schnittstelle in Form von `passTransition(...)` alle erforderlichen Energie-, Leistungs- und Kontrollflussinformationen zur Verfügung und kann

einheitliche Schnittstelle erlaubt Kostenannotation

6 `<src>` und `<dst>` können aus der Bezeichnung des Transitionstypen ermittelt werden.

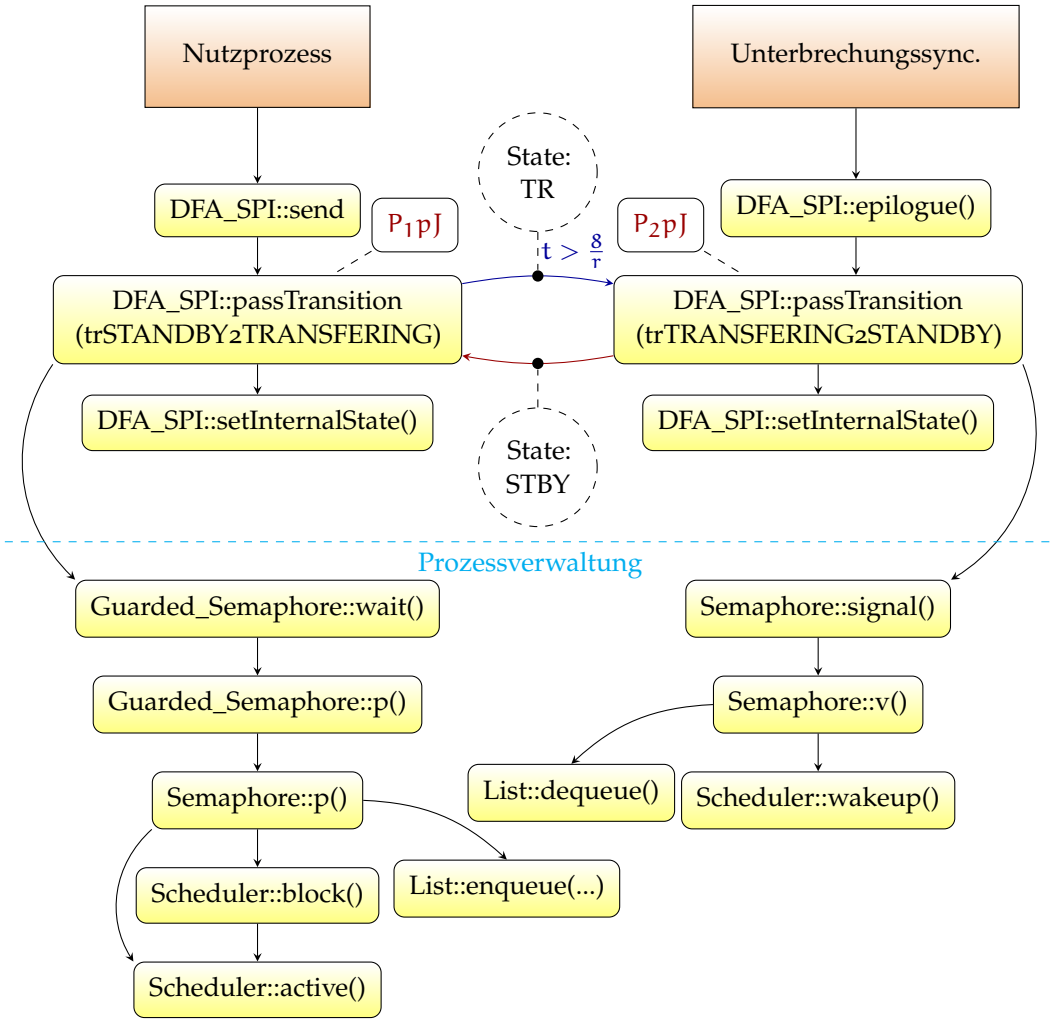


Abbildung 6.10: Aufrufgraphen zu den Methoden des SPI, links bei der Initiierung einer Datenübertragung und rechts bei der Behandlung der Unterbrechungsbehandlung. Anhand der XML-Definition kann der kausale Zusammenhang vervollständigt werden. Zudem können die Methoden der einheitlichen Treiberschnittstelle mit korrespondierenden Kosten annotiert werden.

darauf aufbauend Energieabschätzungen für das Programm durchführen.

6.4 RESSOURCENBEDARF

Die Evaluation des Ressourcenbedarfs gliedert sich in zwei Teile. Zunächst wird der Bedarf des reinen Treibermodells bestimmt. Anschließend werden die Einflüsse der Zusatzkomponenten ausgewertet.

6.4.1 *Treibermodell*

Grundsätzlich beansprucht die Implementierung von Treibern nach dem entworfenen Modell im Vergleich zur klassischen Herangehensweise kaum zusätzliche Ressourcen. Lediglich die Integration der Verbrauchswerte und das Mitführen des aktuellen Zustands benötigen etwas Speicherplatz, sowie die Ausführung weniger zusätzlicher Instruktionen.

Das Mitführen des Zustands geschieht in der Variablen `int state` und belegt somit pro Treiberinstanz 2 B RAM. Die Verbrauchswerte werden im `.text`-Segment abgelegt und belegen für den Treiber mit Index i

RAM Verbrauch durch Zustandserfassung

$$M^{\text{text}}_{\text{driver}(i)} = (|L_i| + |E_i|) \cdot 4\,\text{B}, \tag{6.5}$$

wobei $|L_i|$ die Zustandsanzahl und $|E_i|$ die Kantenanzahl der jeweiligen Treibers angibt. Das `.text`-Segment beinhaltet zusätzliche Instruktionen zum Setzen des aktuellen Zustands, sowie Methoden zum Auslesen des Zustands und seiner Leistungsaufnahme. Die Ausprägung ist jedoch Implementierungsspezifisch und schwankt aufgrund von Optimierungen beim Übersetzungsprozess.

ROM Verbrauch durch Power-API

Zur Einschätzung des tatsächlichen Bedarfs soll die automatenbasierte Implementierung des SPI (siehe Quelltext 5.1 und 5.2) mit einer äquivalenten klassischen Variante verglichen werden. Aus Kompatibilitätsgründen beinhaltet der automatenbasierte Treiber eine zusätzliche Adapterschicht zur klassischen Schnittstelle. Die kompilierte Firmware wird dem Programm `msp430-size` übergeben. Sie enthält jeweils eine der genannten Implementierungen. Ausgabe ist die Belegung von `.text`-, `.data`- und `.bss`-Segment in Byte:

Adapterschicht

```
> msp430-size kratosClassicSPI.elf kratosDFASPI.elf
   text    data     bss     dec     hex filename
   8140       0    1888   10028    272c kratosClassicSPI.elf
   8192       0    1890   10082    2762 kratosDFASPI.elf
```

Es ist zu erkennen, dass die automatenbasierte Implementierung im `.text`-Segment zusätzliche 52 B belegt. Nach Gleichung 6.5 belegen die Verbrauchswerte beim SPI mit 3 Zuständen und 3 Transitionen $M^{\text{text}}_{\text{driver}(i)} = 24\,\text{B}$. Die übrigen 28 B werden durch das Aktualisieren

KONFIGURATION	.TEXT	.BSS
Kein Profiling	0 B	0 B
Display Profiling	624 B	18 B
CPU Profiling	588 B	18 B
Gemeinsames Profiling	1160 B	36 B

Tabelle 6.5: Zusätzlicher statischer Speicherverbrauch durch eingewobenes Profiling in verschiedenen Konfigurationen für das `.text`- und `.bss`-Datensegment, ermittelt mit dem Programm `msp430-size`.

des Zustands in jeder Transitionsmethode belegt, sowie für die *getter* von Zustand und Listungsaufnahme benötigt. Die Differenz von 2 B im `.bss`-Segment kommt durch die Variable `int state` zu Stande.

kaum Belastung durch Treibermodell

Es zeigt sich, dass das reine Treibermodell (ohne die nachfolgenden Zusatzkomponenten) den Ressourcenverbrauch, im Sinne von Speicherbelegung und CPU-Belastung, nur minimal erhöht.

6.4.2 *Zusatzkomponenten*

Stromzähler benötigt mehr Ressourcen

Der Ressourcenbedarf des Profilers wirkt sich im Vergleich zum reinen Treibermodell stärker auf die Ressourcen aus. Zur Ermittlung des statischen Speicherverbrauchs wird wie im letzten Abschnitt das Programm `msp430-size` verwendet. Das Programm wird auf die übersetzte Firmware angewandt. Sie beinhaltet das Profiling des Displays, der CPU, beiden Geräten, bzw. keinem Gerät. Die jeweiligen Differenzen sind in Tabelle 6.5 aufgeführt. Bei einer Größe des FRAM von 64 KiB (vgl. Kapitel 2.4.1) nimmt das gemeinsame Profiling von Display und CPU etwa 1.8 % des Festspeichers in Anspruch. Zur groben Abschätzung des Speicherverbrauchs im `text`-Segment kann folgende Formel verwendet werden, die anhand der Zahlen in Tabelle 6.5 aufgebaut wurde:

$$M^{\text{text}}_{\text{prof}} \simeq n \cdot 470\,\text{B} + \sum_{i=1}^{n} |E_i| \cdot 22\,\text{B}, \tag{6.6}$$

wobei n für die Anzahl der betrachteten Treiber steht und $|E_i|$ die Anzahl der Transitionen von Treiber i angibt. Dies ist jedoch nur eine Richtgröße, da sich je Joinpoint unterschiedliche Optimierungen beim Übersetzen ergeben. Werden die Methoden `passTransition(...)` und `passTransitionPrecisely(...)` als `inline` deklariert, so ergibt sich folgende Abschätzung:

$$M^{\text{text}}_{\text{prof,inline}} \simeq n \cdot 350\,\text{B} + \sum_{i=1}^{n} |E_i| \cdot 152\,\text{B}. \tag{6.7}$$

ATTRIBUTNAME	TYP	GRÖSSE
`prevBillingTime`	`unsigned long`	4 B
`energybillingEnabled`	`bool`	2 B
`energybill`	`unsigned long long`	8 B
`timebill`	`unsigned long`	4 B
SUMME		18 B

Tabelle 6.6: Zusammensetzung des statischen RAM-Verbrauchs (`.bss`) beim Profiling eines Treibers. Die zugehörige Implementierung findet sich in Quelltext 5.9.

Der statische RAM-Verbrauch beträgt pro berücksichtigtem Treiber 18 B, was sich neben Resultaten aus Tabelle 6.5 auch aus den Attributen in Quelltext 5.9 ablesen lässt. Die genaue Zusammensetzung ist in Tabelle 6.6 aufgeführt und beansprucht pro Treiber ca. 0.9 % des Verfügbaren RAM-Speichers (2 KiB, vgl. Kapitel 2.4.1).

Der beim Ausführen einer Transition zusätzlich erforderliche Stapelspeicher entspricht 17 Registern = 34 B in beiden Varianten (ohne und mit `inline`). Dieser Wert kann durch Zählen der maximalen Anzahl von `push` bzw. `pushm`[7] Aufrufe innerhalb des Aufrufgraphen ab der Methode `billTransition(...)` ermittelt werden. Er beinhaltet zudem die zusätzlichen `push`-Aufrufe, die durch das Einweben der Abrechnung innerhalb von `passTransition(...)` entstehen. Dies entspricht ca. 1.7 % des Verfügbaren RAM-Speichers.

extra Laufzeit

Für zusätzliche Laufzeit für das Profiling des Energieverbrauchs primär durch die Methode `billTransitionPrecisely(...)` verursacht, die bei jeder Transition der berücksichtigten Treiber ausgeführt wird und Arithmetik (Addition und Multiplikation) auf großen Zahlentypen 4 B bzw. 8 B ausführt. Die übrigen Methoden schalten lediglich entweder die Bool'sche um, gebene die Zählerstände zurück oder rufen selbst die Abrechnungsmethode auf. Auf dem MSP430 FR5969 mit 16 MHz ergibt sich eine Ausführungsdauer der Abrechnung von ca. 58 µs, die mit Hilfe der internen Zeitangabe ermittelt wurde und zur Kontrolle mit MIMOSA bestätigt wurde.

Trigger: Verbrauch vernachlässigbar

Der Einfluss des Triggers wurde bereits bei der Implementierung ausgewertet (siehe Kapitel 5.2.1) und reduziert sich bei einer Signalisierung durch 1-Bit Flankenwechsel zu einer `xor` Instruktion pro Joinpoint oder Verwendungsstelle.

6.5 GENERALISIERBARKEIT

Wie die Implementierung der Treiber für vier verschiedene Geräteklassen gezeigt hat, treten an keiner Stelle signifikante Schwierigkei-

7 Sichert in einer Instruktion m Register auf den Stack.

ten bei der Umsetzung dieses Modells auf. Daher wird davon ausgegangen, dass sich auch für weitere Geräte dieser Klassen Treiber nach diesem Prinzip implementieren lassen. Dies liegt daran, dass alle bisher untersuchten Geräte prinzipiell zustandsbasiert arbeiten und sich derartige Automaten (oder eine Teilmenge davon) auch im Treiber abbilden lassen. Außerdem erfordert das entworfene Treibermodell keine strengen Regeln, die den Entwickler an der Implementierung eines Treibers hindern könnten. Lediglich die Treiberschnittstelle ist auf das Anstoßen von Transitionen vereinheitlicht worden, die jedoch Parameter und Rückgabewerte wie in klassischen Treiberimplementierungen erlauben, was somit keine echte Einschränkung darstellt. Um die Kompatibilität mit übergeordneten Schichten sicherzustellen, kann eine Adapterschicht zusätzlich die klassischen Treiberschnittstelle bereitstellen. Zudem wird die Verwendung des Gerätes erleichtert, da sein aktueller Betriebszustand jederzeit abgefragt werden kann. Der zusätzliche Ressourcenaufwand im Vergleich zum klassischen Treiber beschränkt sich auf ein absolutes Minimum (siehe Abschnitt 6.4.1), sodass es selbst in Systemen mit starken Ressourcenbschränkungen eingesetzt werden kann. Ist kein Online-Profiling notwendig, sondern nur eine statische Analyse gewünscht, so können die annotierten Verbrauchswerte (für Zustände und Transitionen) und ggf. sogar das Mitführen des Gerätezustands im laufenden System eingespart werden.

geringe Anforderungen an die Programmiersprache

Im Hinblick auf das umgebende Betriebssystem ergeben sich für die Umsetzung derartiger Treiber ebenfalls keine kritischen Einschränkungen. Das reine Treibermodell (ohne Zusatzkomponenten) benötigt weder Betriebssystemunterstützung, noch Aspekte oder Templates. Objektorientierte Mechanismen beschränken sich lediglich auf die Vererbung bei der Definition von Transitionen. Sofern nur die statische Analyse und keine Zusatzkomponenten erforderlich sind, können diese auch ohne Vererbung formuliert werden.

Die Zusatzkomponenten wie der Trigger und das Mitführen des Energieverbrauchs (Online-Profiler) erhöhen jedoch die Anforderungen. Zunächst muss das System den zusätzlichen Ressourcenbedarf (siehe Abschnitt 6.4.2) bewältigen. Zusätzlich erfordern beide Komponenten eine aspektorientierte Programmiersprache, um die gewonnene Übersichtlichkeit und Konfigurierbarkeit zu erreichen. Auf Kosten dieser Punkte sind sie grundsätzlich auch ohne Aspekte, z. B. durch Präprozessormakros oder festes Einprogrammieren realisierbar. Unverzichtbar für das Profiling ist jedoch die genaue Zeiterfassung zur Abrechnung der Energie, die sich auch aus der zustandsspezifischen Leistungsaufnahme und der Verweilzeit innerhalb eines Zustands ergibt. Zudem muss die CPU Additionen und insbesondere Multiplikationen großer Zahlentypen unterstützen.

Insgesamt zeigt sich, dass ein energiegewahres Treibermodell, unabhängig davon, ob es in genau der hier vorgestellten Implementie-

rungsweise und Ausprägung umgesetzt wird, zur langfristigen und rein softwarebasierten Energieverfolgung einer breiten Palette von eingebetteten Hardwarekomponenten eingesetzt werden kann, sowie auch die statische Codeanalyse hinsichtlich des erwarteten Energieverbrauchs vereinfacht.

6.6 EINSCHRÄNKUNGEN/GRENZEN

Die Analyse der Modellgüte in Abschnitt 6.1 hat deutlich gezeigt, dass die Exaktheit der Online-Schätzung (und damit gleichzeitig auch die Aussagekraft der auf den gleichen Daten arbeitende Codeanalyse) von der Qualität der Messdaten abhängig ist. Datenblätter stellen die Verbrauchsdaten nur grob dar und dienen daher nur als „Programmieranleitung“ und als Orientierungspunkt zur Festlegung von Zuständen und Transitionen. Die Akquisition der Verbrauchswerte stößt jedoch insbesondere bei besonders energiesparenden Komponenten an die Grenzen des verwendeten Messsystems. Auch mit gesteigerter Empfindlichkeit nimmt der Rauschabstand der Messdaten ab. Zudem fallen parasitäre Effekte wie Leckströme durch Busleitungen aufgrund des niedrigen Absolutverbrauchs stärker ins Gewicht. Für zuverlässigere Messwerte müssen solche Komponenten bei der Messung galvanisch vom treibenden System getrennt werden, was den Aufwand im Vergleich zur einfachen Messung an der Spannungsversorgungsleitung des Geräts wesentlich steigert. Integrierte Komponenten wie das SPI sind zudem nicht einzeln Messbar.

geringer Verbrauch = geringe Rauschabstand

Auch wenn sich durch (teil)generierte Testprogramme, vollständige Signalisierung und automatische Auswertung der markierten Messabschnitte der zeitliche Aufwand zur Ermittlung der Verbrauchsdaten verringern lässt, ist die Güte der erfassten Daten von der Erfahrung des Entwicklers und dem Detailgrad der Analyse abhängig.

Eine im Rahmen dieser Arbeit vorgenommene Einschränkung ist die Festlegung einer konstanten Versorgungsspannung (3.2 V). Die ermittelten Werte sind daher auch nur für diese Betriebsspannung gültig. Dies ist nicht nur für die Wiederverwendbarkeit des Treibers von Bedeutung, sondern auch für batteriebetriebene Systeme, deren Betriebsspannung durch Entladung der Batterie sinkt. Zwar können die Messwerte rechnerisch auf eine abweichende Spannung skaliert werden, es ist jedoch im Einzelfall zu prüfen, ob alle Werte gleichermaßen skalieren. Dies steigert jedoch den bereits ohnehin äußerst umfangreichen und fehleranfälligen Messaufwand und Analyseprozess um einen wesentlichen Faktor.

variable Betriebsspannung...

...erfordert aufwendige Analyse

Wie jedes Modell, hat auch dieser Entwurf seine Grenzen. In diesem Fall sind das hoch dynamische Systeme, bzw. Komponenten, deren Energiebedarf signifikant von äußeren Einflüssen abhängt und nicht zuverlässig vorhersagbar ist. Beispiel hierfür sind Servomotoren, wie sie kleinen humanoiden Robotern Anwendung finden. Um

unvorhersagbare Systeme

das Gleichgewicht zu halten, müssen die Servos unterschiedliche Beschleunigungseinflüsse ausgleichen, die u. A. von der Beschaffenheit des Untergrunds abhängig sind. Die Servos sind wiederum selbst Systeme mit einer Regelschleife, die dynamisch die Menge an Energie aufwenden, die zum Erreichen der aktuellen Sollgröße erforderlich ist – belastet durch die üblichen parasitären Effekte wie Überschwinger.

Solche Systeme lassen sich nur durch szenariospezifische Mittelwerte (z. B. Ausführung eines Schritts in der Ebene bei Holzboden und Geschwindigkeit v) erfassen, benötigen hierfür jedoch kein spezielles Treibermodell. Die Abbildung der gesamten Dynamik und alle verbundenen Parameter im Treiber treibt den Modellumfang stark nach oben, erschwert die Akquisition und Reproduzierbarkeit entsprechender Verbrauchswerte, und bleibt dennoch abhängig von unvorhersehbaren äußeren Einflüssen.

7 ZUSAMMENFASSUNG

Im Verlauf dieser Arbeit wurde ein energiegewahres Treibermodell für eingebettete Betriebssysteme entwickelt, das den aktuellen Betriebszustand des assoziierten Geräts in Software abbildet, sowie die mit dem Gerät verbundenen Kosten in Form von Leistungsaufnahme und Energieverbrauch für Zustände respektive Transitionen bereithält. Es ist motiviert durch das Energiesparpotenzial, das sich aus der Verfügbarkeit des Energieverbrauchs zur Entwicklungs- und Laufzeit ergibt. *Motivation*

Vor dem Entwurf des Treibermodells wurden zunächst die erforderlichen Grundlagen behandelt. Diese umfassten die Ursachen für den Energieverbrauch in integrierten Schaltungen, Methoden zur Modellierung des Verbrauchs, sowie die Funktionsweise des Messsystems, mit dem die Verbrauchswerte der besonders energiesparenden Geräte ermittelt worden sind. Außerdem wurden die Evaluationsplatformen und das umgebende Betriebssystem Kratos erläutert. Danach wurden ähnliche Forschungsarbeiten vorgestellt, deren Ansätze und Ideen die Entwicklung des Treibermodells inspiriert haben. PTA haben sich dabei als adäquates abstraktes Modell herausgestellt. *Grundlagen* *Betriebssystem*

Im nachfolgenden Kapitel wurden zunächst die Ziele und Anforderungen an das Treibermodell festgelegt und anschließend die Schnittstelle und interne Struktur zur Umsetzung der PTA in Software entworfen: Treiber nach diesem Modell werden als PTA entwickelt, deren Benutzungsschnittstelle sich auf das Anstoßen von Transitionen im Automaten vereinheitlicht. Die Transitionen sind dabei so zu wählen, dass sie sowohl die funktionalen Eigenschaften des Gerätes nach außen bereitstellen, als auch mit dem dabei entstehenden Stromverbrauch korrespondieren. Die Zustände des Automaten repräsentieren Betriebszustände des Gerätes (aus, bereit, aktiv,...), sind mit der entsprechenden Leistungsaufnahme verknüpft und können ausschließlich über Transitionen beeinflusst werden. *einheitliche Schnittstelle* *Kosten annotierbar*

Durch diese enge Korrespondenz zwischen Schnittstelle und Stromverbrauch des Geräts kann der Verbrauch des Geräts vollständig in Software nachgebildet werden. Die Verbrauchsdaten für Zustände und Transitionen werden hierzu direkt in den Treiber eingetragen und können über eine Power-API jederzeit abgefragt werden, die darüber hinaus auch Auskunft über den aktuellen Zustand des Geräts erteilen kann. Die einheitliche Schnittstelle ermöglicht es zusätzlich, dass die Verbrauchsdaten bei der statischen Codeanalyse mit einbezogen werden können. Der durch Unterbrechungen unvollständige *Power-API* *Kontrollfluss*

Kontrollfluss kann anhand der Automatendefinition unter Angabe einer zeitabhängigen Transition vervollständigt werden.

Zusätzlich zum eigentlichen Treiberentwurf wurden auch Zusatzkomponenten wie der Trigger zur Markierung von Zustandswechseln und der Energy Profiler (Stromzähler) zur Protokollierung des Energieverbrauchs entwickelt. Um maximale Konfigurierbarkeit zu Gewährleisten wurde dabei intensiv von aspektorientierter Programmierung Gebrauch gemacht. Schließlich wurde auch der typische Entwicklungsablauf für Treiber nach diesem Modell demonstriert.

Stromzähler

Im Implementierungskapitel wurde aus vier, für eingebettete Systeme typischen, Geräteklassen je ein Treiber nach dem entworfenen Modell umgesetzt. Dabei wurden gerätespezifische Besonderheiten vorgeführt und erste Probemessungen zur Überprüfung des Modells durchgeführt. Außerdem wurden Trigger und Stromzähler implementiert und Herausforderungen bei Synchronisierung und Zeiterfassung behandelt. Das Kapitel endete mit Werkzeugen, die eine Treiberentwicklung nach diesem Modell beschleunigen sollen.

Synchronisierung, Zeiterfassung

Schlussendlich folgte die Evaluation des entworfenen Treibermodells. Hierzu wurde zunächst die Genauigkeit des Messsystems untersucht, da hierbei Last- und Temperaturabhängikeiten aufgefallen sind. Anschließend wurden die Verbrauchswerte der Geräte unter Einsatz des Triggers ermittelt und deren Güte bewertet. Dannach wurden die Werte in die Treiber eingepflegt und in einem längeren Experiment ein Versuchslauf durchgeführt. Hierzu wurde der Energieverbrauch einzelner Treiber, sowie des Gesamtsystems rein Softwarebasiert anhand des Stromzählers ermittelt und mit den Ergebnissen einer gleichzeitig erfolgten Messung verglichen. Außerdem wurden weitere Automatisierungsmöglichkeiten diskutiert, sowie aufgezeigt, wie die Informationen aus dem Treibermodell bei der Codeanalyse genutzt werden können. Nach einer Auswertung des Ressourcenbedarfs für einergiegewahre Treiber und ihre Zusatzkomponenten wurde auf die allgemeine Anwendbarkeit des Modells eingegangen, sowie seine Grenzen aufgezeigt.

Temperaturabhängigkeit

Energiezählung in Software

Ressourcenbedarf, Grenzen

7.1 FAZIT

Die Evaluation der Beispieltreiber hat gezeigt, dass sich der Energieverbrauch mit hoher Genauigkeit in Software nachbilden lässt. Im Rahmen der Evaluation wurde eine maximale Abweichung in der Größenordnung von 5 % bis 10 % erreicht, die jedoch auch längerfristig stabil bleibt und nicht divergiert. Allerdings hat sich auch herausgestellt, dass komplexe Geräte wie die CPU aufgrund ihres Caches und der Komplexität nicht im vollständigen Umfang vom Treibermodell erfasst werden können, sondern auf szenariospezifische Mittelwerte zurückgegriffen werden muss. Dabei wird jedoch die Wiederverwendbarkeit der ermittelten Verbrauchsdaten eingebüßt. Eine

langfristig stabil

Caches, Komplexität

deutlich umfangreichere Modellierung solcher Geräte würde jegliches Einsparpotenzial durch den erhöhten Rechenaufwand aufzehren.

Rechenaufwand

Ferner wurde am Beispiel des SPI und des Beschleunigungssensors klar, dass die Komponenten einzeln vermessen werden müssen. Ansonsten können die Messergebnisse durch die Überlagerung eines anderen Verbrauchsprofils oder durch parasitäre Leckströme verfälscht werden. Die Evaluation hat bestätigt, dass eine Superposition des Verbrauchs einzelner Treiber problemlos möglich ist.

parasitäre Effekte
Superposition

Die Beispieltreiber und Zusatzkomponenten wurden in AspectC++ programmiert und in das Betriebssystem Kratos integriert. Dies erlaubt eine hochgradige Konfigurierbarkeit ohne den Einsatz von undurchsichtigen Makros oder dynamischen Entwurfsmustern, die viele unnötige Methodenaufrufe zur Laufzeit beinhalten. Zudem verzichtet der Entwurf vollständig auf den Einsatz von Templates.

AspectC++

Grundsätzlich ist das Treibermodell jedoch weder an Kratos, noch an AspectC++ gebunden und kann somit auch in anderen eingebetteten Betriebssystemen eingesetzt werden. Es benötigt keine invasiven Eingriffe in das Betriebssystem, kann parallel zu klassischen Treibern betrieben werden und bei Bedarf sogar an eine klassische Schnittstelle adaptiert werden. Zudem ist der zusätzliche Ressourcenbedarf des reinen Treibermodells ohne Zusatzkomponenten selbst auf Mikrocontrollern mit beschränktem RAM und ROM im Verlgeich zu klassischen Treiberimplementierungen vernachlässigbar klein. Diese Minimalkonfiguration kann herangezogen werden, wenn ausschließlich eine statische Analyse gewünscht ist, jedoch keine Zählung des Energieverbrauchs zur Laufzeit erfolgen soll. Das Profiling wirkt sich hingegen stärker auf die Systemressourcen aus und verbraucht auf dem MSP430 FR5969 für das Erfassen von zwei Geräten fast 2 % des verfügbaren Arbeitsspeichers, sowie zusätzliche 1.7 % während einer Transitionsabrechnung durch das Sichern von Registern auf dem Stapelspeicher. Auch der Festspeicher (hier FRAM) wird in diesem Szenario mit 1.8 % belastet. Zudem benötigt die Abrechnung jeder berücksichtigten Transition durch die Arithmetik mit großen Zahlen ca. 58 µs.

unabhängig von Kratos und AspectC++

Profiling beansprucht Ressourcen

Arithmerik großer Zahlen

Bei der Implementierung und Evaluation wurde ein einfaches Energiemodell mit konstanter Betriebsspannung gewählt. Das Treibermodell gestattet aber eine beliebige Parametrisierung der Verbrauchswerte. Allerdings muss an dieser Stelle warnend darauf hingewiesen werden, dass bereits Messungen mit konstanter Betriebsspannung äußerst mühsam und Fehleranfällig sind. Zwar entsteht dieser Aufwand infolge des noch prototypischen Messsystems und kann in künftigen Versionen durch höhere Automatisierung weiter reduziert werden. Dennoch steigt mit zunehmender Parametrisierung die Menge der Testfälle, die zur Überprüfung der Korrektheit notwendig sind, signifikant an.

konstante Spannung

Abschließend ist anzumerken, dass alle in Kapitel 4.1 gestellten Anforderungen durch das entworfene Treibermodell abgedeckt werden.

7.2 AUSBLICK

Durch diese Arbeit wurde die Grundlage zu einer Abschätzung des Energiebedarfs im Rahmen einer statischen Codeanalyse geschaffen. Künftige Forschungsarbeiten können aufbauend auf diesem Entwurf zunächst die Modellparameter in ein Analysewerkzeug einbinden, und dadurch die *Energie* als zusätzliches Optimierungskriterium bei der automatisierten Dimensionierung eingebetteter Systeme einfließen zu lassen.

automatische Dimensionierung von ES

Außerdem kann nun genau untersucht werden, wie groß das Energiesparpotenzial durch die Online-Verfügbarkeit der Verbrauchswerte ist. Hierzu kann beispielsweise ein verteiltes Sensornetz dienen, das mit Hilfe des Energiestatus seiner Sensorknoten einen Lastausgleich durchführt und dadurch die Verfügbarkeit steigert.

Energiesparpotenzial

Schließlich sollte der Ansatz der automatisierten Messung weiter verfolgt werden, da dieser Punkt noch wesentliches Optimierungspotenzial aufweist. Dies erfordert jedoch zunächst den Ausbau des Messsystems auf mehrere Trigger-Bits, um darin Transitionen und Zustände kodieren zu können. Auf diese Weise kann die Akquisition der Verbrauchswerte, bzw. ihre Auswertung, wesentlich beschleunigt werden.

Messsystem erweitern

Teil II

DOKUMENTATION & ANHANG

Dieser Teil dokumentiert die im Rahmen der Arbeit vorgenommenen Änderungen und Erweiterungen an Soft- und Hardware. Außerdem werden Quelltexte und große Bilder hier gebündelt.

A

KRATOS

Dieses Kapitel dokumentiert wichtige Modifikationen und Fehlerkorrekturen am Betriebssystem Kratos, die im Verlauf der Arbeit vorgenommen wurden.

TAKTSYSTEM

Die Initialisierung des LFXT war fehlerhaft, da die Einschwingphase des Quarzes nicht berücksichtigt wurde. Der MSP430 schaltet deswegen auf MODOSC um, dessen Frequenz höher (ca. 35 kHz bis 37 kHz) als die erwarteten 32 768 Hz ist. Da der UART mit diesem Takt versorgt wird, kommt es bei der Übertragung einzelner Bytes zu signifikanten Fehlern.

Der Fehler wurde wie folgt korrigiert: Sofern das Feature LFXT bei der Konfiguration von Kratos ausgewählt ist, wird nach einem Reset auf das korrekte Einschwingen von LFXT gewartet, bevor das Betriebssystem die Anwendungsprozesse startet.

AKTIVES WARTEN

Die Methode `Timer::delay_us(const unsigned int us)` für aktives Warten kurzer Zeitspannen hat die Anzahl der erforderlichen Zyklen vollkommen falsch berechnet. Die Anzahl der Zyklen wurde von `us/5` auf `us*3` korrigiert. Der Wert unterschätzt die erforderliche Zeitspanne um einen geringen Anteil, jedoch lässt sich die Multiplikation mit Faktor 3 effizient durch ein Shift nach links und eine Addition realisieren.

SPI UNTERBRECHUNGSVEKTOR

Für die interruptbasierte Datenübertragung via SPI war das Unterbrechungsereignis falsch konfiguriert. Es wurde eine Unterbrechung ausgelöst, sobald der Sendepuffer frei wurde (`UCB0IE = UCTXIE;`). Dieser wird sofort wieder frei, wenn das SPI im aktuellen Moment keine Datenübertragung ausführt. Da die Implementierung davon ausgeht, dass bei einer Unterbrechung je ein Byte gesendet und empfangen wird, kann bei geringen Datenraten ein veraltetes Datenwort aus dem Empfangspuffer ausgelesen werden. Das Unterbrechungsereignis wurde daher auf das Empfangen eines Datenworts korrigiert

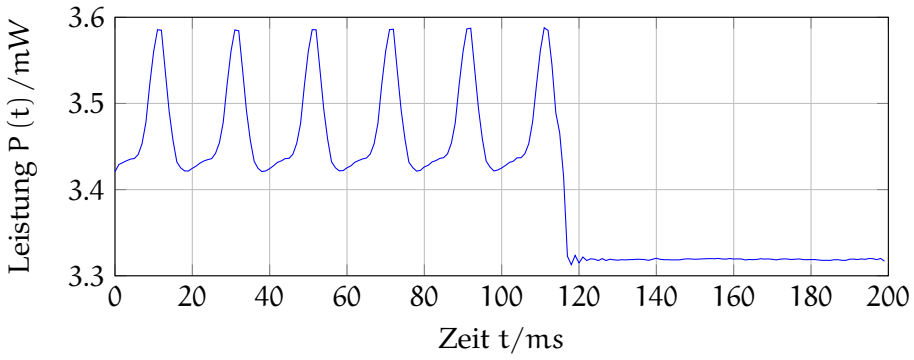


Abbildung A.1: Zusätzlicher Energieverbrauch durch eingefangenes Netzbrummen (50 Hz) über ungenutzte IO-Pins des MSP430. Ab 115 ms werden die IO-Ports als Ausgang konfiguriert – der Verbrauch sinkt und bleibt konstant.

(`UCB0IE = UCTXIE;`). Bei hohen Übertragungsraten fällt dieser Fehler nicht auf und blieb deswegen im Verborgenen.

UNGENUTZTE IO-PINS

Ungenutzte IO-Pins wurden bisher bei der Initialisierung des Mikrocontrollers nicht berücksichtigt und verblieben in der Standardkonfiguration. Diese werden nach einem Reset als Eingang ohne aktivierte *pull-up* oder *pull-down* Widerstände konfiguriert. Die hochohmigen Eingänge fangen Umgebungsstörungen ein und lösen im Mikrocontroller an den korrespondierenden Registern Bitwechsel aus. Diese führen zu einem signifikant erhöhten Energieverbrauch, wie Abbildung A.1 demonstriert. Am stärksten äußert sich das eingekoppelte Netzbrummen von 50 Hz. In einer abgeschirmten Kiste oder bei konfigurierten IO-Ports verschwindet diese Störung. Bei der Initialisierung von Kratos müssen deswegen alle ungenutzten IO-Ports entweder als Ausgang oder als Eingang mit aktivem *pull-up* oder *pull-down* Widerstand konfiguriert werden. Dies verhindert einen unnötigen Energieverbrauch und ermöglicht reproduzierbare Energiemessungen.

Können wie im InBin Peripheriegeräte über MOSFETs von der Versorgungsspannung getrennt werden, dann fängt das Gate des MOSFETs ebenfalls die genannten Störungen ein, falls das Gerät in der Konfiguration von Kratos entfernt wird und die korrespondierenden IO-Pins dadurch nicht konfiguriert werden. Als Folge wird das Gerät mit den Frequenzen der eingefangenen Störungen ein- und ausgeschaltet. Dadurch werden hinterliegende Abblockkondensatoren unnötig geladen und das Gerät verbraucht wider Erwarten Energie. Beim Beschleunigungssensor ist die besonders kritisch, da die Abblockkondensatoren großzügig dimensioniert sind und der Sensor beim Einschalten eine energieintensive Initialisierungsprozedur

(wahrscheinlich eine Kalibrierung) durchführt. Daher müssen solche „fliegenden“ Gates unbedingt vermieden werden, indem die verbundenen IO-Ports selbst bei auskonfiguriertem Gerät auf einen definierten Pegel gebracht werden.

SYNCHRONISIERUNG AM SPI-BUS

Um mehrere Komponenten an einem SPI zur ermöglichen, wurde die Implementierung des SPI um optionale Semaphoren erweitert (Auswählbar in der Konfiguration von Kratos), die ein exklusives Sperren des SPI vor Selektion (CS) des gewünschten Busteilnehmers gestattet.

TEXTAUSGABE FÜR PIXELBILDSCHIRME

Zur komfortablen und speichersparenden Ausgabe von Text auf dem Sharp Display des Launchpads wurde die Template-Klasse `Typewriter` entwickelt, die sich für beliebig dimensionierte Pixeldisplays verwenden lässt, welche ihre Pixel zeilenweise übermittelt bekommen.

Zwar bietet der MSP430 FR5969 hinreichend viel RAM zum Speichern von Textinhalt in Form von ASCII-Zeichen, jedoch nicht zum sinnvollen Zwischenspeichern der Bildschirmpixel (bei 96×96 Pixeln 1.2 kB), da dies bereits mehr als die Hälfte des RAM belegen würde. Stattdessen werden während der Übertragung der Pixelzeilen anhand des jeweils darzustellenden Abschnitts die Pixel aller betroffenen Zeichen dieser Zeile neu berechnet. Die Implementierung vom `Typewriter` findet sich in Quelltext C.8.

B

MIMOSA

Dieses Kapitel dokumentiert Modifikationen am Messsystem MIMOSA in Soft- und Hardware. Zudem stellt das Kapitel die GNU Octave-Skripte vor, die zur Auswertung der Messaufzeichnungen geschrieben und verwendet wurden.

B.1 SOFTWARE

Die MIMOSA-Software wurde um folgende Modi erweitert:

Monitormodus

Im Monitormodus werden die Messwerte nicht in einer Datei gespeichert, sondern der Durchschnitt der letzten 5000 Samples je Integrationsstufe aus der Konsole ausgegeben. Der Wert des Buzzers, sowie eine Laufnummer werden ebenfalls mit ausgegeben. Der Modus ermöglicht eine schnelle Überprüfung des Messaufbaus, sowie der Kalibrierung, ohne Speicherplatz zu verschwenden. Aufruf:

```
> ./mimosa -m
```

Temperaturmodus

Zur Beobachung und Auswertung des langfristigen Temperaturdrifts (siehe Kapitel 6.1.1), wurde ein Temperaturmodus implementiert. Dieser Modus speichert in Intervallen von <Zeit> Sekunden den Durchschnittswert der drei Integrationsstufen innerhalb jedes Intervalls in einer Datei `temperature-YYYYMMDD-HHMMSS.csv` und gibt sie zudem auf der Konsole aus. Aufruf:

```
> ./mimosa -t <Zeit>
```

Summationsmodus

Da die Messaufzeichnungen (`.csv`-Dateien) wegen der Abtastrate von 100 Hz bei längeren Aufzeichnungen große Mengen an Speicherplatz verbrauchen, wurde für langfristige Beobachtungen des Energieverbrauchs der Summationsmodus hinzugefügt. Dieser Modus speichert in der Datei `sum-YYYYMMDD-HHMMSS.csv` bei einer *steigenden* Flanke des Buzzers die Energie, die seit der letzten *fallenden* Flanke durch das DUT verbraucht wurde. Aufruf:

DIP	R_S	$I_S^{MAX}(U_S = 3\,V)$	U_S
1			3 V
2			0 V ··· 5 V
3	680 Ω	2.4 mA	
4	330 Ω	5.0 mA	
5	160 Ω	10.3 mA	
6	82 Ω	20.1 mA	
7	68 Ω	24.3 mA	
8	33 Ω	50.0 mA	

Tabelle B.1: Bedeutung der hinzugekommenen DIP-Schalter auf der Vorplatine von Mimosa (siehe Abbildung 2.3), für die Messbereichserweiterung (Schalter 3 bis 8) und Spannungserweiterung (Schalter 1 und 2). Im laufenden Betrieb dürfen niemals alle Schalter 3 bis 8, bzw. Schalter 1 und 2 gleichzeitig ausgeschaltet sein!

```
> ./mimosa -s
```

B.2 HARDWARE

Folgende Modifikationen wurden an der MIMOSA-Hardware durchgeführt:

Messbereichserweiterung

Um den Messbereich von MIMOSA in beide Richtungen zu erweitern, wurden zusätzliche Shunt Widerstände in die Vorplatine (Spannungsversorgung) eingesetzt, die sich durch kleine DIP-Schalter auswählen lassen. Die Platine mit den Schaltern ist in Abbildung 2.3 zu finden. Je nach Schalterstellung sind die in Tabelle B.1 aufgeführten Maximalströme I_S^{max} messbar. Die Angaben gelten jedoch nur bei $U_S = 3\,V$. Je kleiner der Maximalstrom, desto genauer können kleine Ströme gemessen werden. Im laufenden Betrieb dürfen *niemals* alle DIP-Schalter 3 bis 8 gleichzeitig ausgeschaltet sein!

Spannungserweiterung

Um auch andere Betriebsspannungen außer $U_S = 3\,V$ zu gestatten, kann die konstante Ausgangsspannung von MIMOSA nun auch stufenlos mit einem Präzisionspotenziometer im Bereich von 0 V bis 5 V eingestellt werden. Hierzu muss der DIP-Schalter 2 eingeschaltet, und Schalter 1 ausgeschaltet werden (siehe Tabelle B.1). Im laufenden Betrieb dürfen die DIP-Schalter 1 und 2 niemals gleichzeitig ausgeschaltet sein!

B.3 ANALYSEWERKZEUGE

Zur Auswertung und Visualisierung der MIMOSA Messaufzeichnungen wurden Skripte für GNU Octave entwickelt, von denen einige hier vorgestellt werden sollen:

Konvertierung

Ein Makefile konvertiert die `.csv`-Dateien im aktuellen Verzeichnis zu `.mat` Dateien. Zwar kann GNU Octave auch `.csv` lesen, jedoch deutlich langsamer als das `.mat`-Format. Dies fällt insbesondere bei Aufzeichnungen auf, deren Länge 30 s überschreitet. Die Konvertierung einer Datei wird durch ein eigenes Python-Skript `csv2mat.py` durchgeführt. Es ist zu beachten, dass die `.csv`-Datei dafür zunächst in den Arbeitsspeicher geladen wird.

Kalibrierung

`calibrate.m` berechnet zu einer Aufzeichnung den Durchschnittswert der drei Integrationsstufen. Optional kann ein Zeitintervall in Millisekunden angegeben werden. Die Durchschnittswerte zweier Referenzgrößen (hier wurden immer 600 Ω und ∞Ω verwendet) werden in das Skript `getPowerFactor.m` eingetragen, anhand dessen die weiteren Skripte ihre Resultate kalibrieren. Aufruf:

```
>> calibrate("Dateiname.mat", [<Start_ms>], [<Ende_ms>]);
```

Analyse

`analyzeVec.m` stellt eine Messaufzeichnung, getrennt nach Integrationsstufe, graphisch dar. Je nach Wert des Buzzers wird der Hintergrund weiß oder blau eingefärbt, sowie der Buzzerwert selbst mit in das Diagramm eingezeichnet. Optional kann ein festes Intervall in Millisekunden selektiert werden. Zur beschleunigten Anzeige kann auch die Abtastrate reduziert werden. Soll nicht reduziert werden, wird ein Faktor von 1 angegeben. Aufruf:

```
>> calibrate("Dateiname.mat", <Reduktionsfaktor>, [<Start_ms>],
    [<Ende_ms>]);
```

Zudem gibt das Skript die durchschnittliche Leistungsaufnahme und die Standardabweichung je Integrator, sowie für jedes durch die Buzzerflanken geteilte Zeitintervall die Durchschnittsleistung, die Standardabweichung, die Energie und die Intervalllänge aus.

Triggerauswertung

`triggerEvalVec.m` arbeitet ähnlich wie die Analyse, bestimmt jedoch die Gesamtstatistik zu Intervallen mit Buzzer = 1 und Buzzer = 0. Dies dient dazu, viele gleichartige Intervalle automatisiert auszuwerten. Optional kann auch hier das zu betrachtende Zeitintervall eingeschränkt werden. Im Quelltext können zudem Intervalle bestimmter Mindest- oder Höchstlänge gefiltert werden. Aufruf:

```
>> calibrate("Dateiname.mat", [<Start_ms>], [<Ende_ms>]);
```

C

ABBILDUNGEN

In diesem Abschnitt werden größere Graphiken und längere Quelltexte, die im Verlauf der gesamten Arbeit referenziert werden, in kummulierter Form dargestellt. Die Inhalte sind mit einer kurzen Beschreibung versehen und beinhalten einen Verweis auf das Kapitel, in dem auf die entsprechende Graphik bzw. Inhalte eingegangen wird.

Quelltext C.1: SPI Treiber (`dfa_spi.h`) ☞ Kapitel 5.1.1.2.

```
#ifndef DFA_SPI_H
#define DFA_SPI_H

#ifndef BV
#define BV(x) (1<<(x))
#endif

#include "drivers/dfa_driver.h"
#include <stddef.h>
#include "syscall/chardev.h"
#include "guard/gate.h"
#include "syscall/guarded_semaphore.h"

namespace eUSCI_B {

class DFA_SPI : public DFA_Driver, public CharDev, public
    Gate
{
private:
    Guarded_Semaphore tx_sema;

public:

    DFA_SPI();

    // PUBLIC DFA DEFINITIONS
    struct State {
        enum state_t {
            UNINITIALIZED,
            STANDBY,
            TRANSFERING,
            _STATE_MAX
        };
    };

    struct trENTRY : public guarded_transition_t  {
```

Quelltext C.1 (Forts.)

```
        trENTRY() : guarded_transition_t(0) {}
        //const energy_uJ_t transitionenergy;
    };

    struct trSTANDBY2TRANSFERING : public
        guarded_transition_t  {
        const unsigned char* tx_byte;
        unsigned char* rx_byte;
        trSTANDBY2TRANSFERING(const unsigned char* tx_byte,
            unsigned char* rx_byte) : guarded_transition_t(0),
             tx_byte(tx_byte), rx_byte(rx_byte) {}
    };

    struct trTRANSFERING2STANDBY : public transition_t  {
        trTRANSFERING2STANDBY() : transition_t(0) {}
    };

    static const power_uW_t statepower[];
    GHOST int getPower_uW() { return statepower[getState()];
        }
    GHOST int getPower_uW(int state) { return statepower[
        state]; }


    void passTransition(trENTRY trd);
    void passTransition(trSTANDBY2TRANSFERING trd);
    void passTransition(trTRANSFERING2STANDBY trd);
    // END DFA DEFINITIONS

    /* CharDev methods */
    void send(const unsigned char *addr, unsigned int size);
    void recv(unsigned char* addr, unsigned int size);

    /* Gate methods */
    void plugin();
    bool prologue();
    void epilogue();
};
}

extern eUSCI_B::DFA_SPI CONFIG_eUSCI_B0_DFA_SPI_NAME;

#endif // DFA_SPI_H
```

Quelltext C.2: SPI Treiber (`dfa_spi.cc`) ☞ Kapitel 5.1.1.2.

Quelltext C.2 (Forts.)

```
#include <msp430.h>
#include "device/plugbox.h"
#include <stddef.h>
#include "config.h"
#include "dfa_spi.h"

namespace eUSCI_B {

const DFA_Driver::power_uW_t DFA_SPI::statepower[] = {
#ifdef CONFIG_dfa_spi_BlindAccounting
    0,  // UNINITIALIZED
    0,  // STANDBY
    0   // TRANSFERING
#else
    999,  // UNINITIALIZED
    999,  // STANDBY
    999   // TRANSFERING
#endif
};

void DFA_SPI::passTransition(trENTRY trd) {
    /* UCB0CLK Pin 2.2 */
    P2SEL0 &= ~ BV(2);
    P2SEL1 |= BV(2);
    P2DIR  |= BV(2);

    /* UCB0SIMO Pin 1.6 */
    P1SEL0 &= ~BV(6);
    P1SEL1 |= BV(6);
    P1DIR  |= BV(6);

    /* UCB0SOMI Pin 1.7 */
    P1SEL0 &= ~BV(7);
    P1SEL1 |= BV(7);
    P1DIR  &= ~BV(7);
    P1OUT  &= ~BV(7);    //Pull-Down, verhindert Leckstrom
        bei ausgeschalteten Geräten
    P1REN  |= BV(7);

    UCB0CTLW0 = UCCKPH | UCMSB | UCMST | UCSYNC | UCMODE_0 |
        UCSSEL__SMCLK | UCSWRST;
    UCB0BRW = (CONFIG_SMCLK_FREQUENCY /
        CONFIG_eUSCI_B0_DFA_SPI_BITRATE);
    UCB0CTLW0 &= ~UCSWRST;

    setInternalState(State::STANDBY);
}
```

Quelltext C.2 (Forts.)

```
void DFA_SPI::passTransition(trSTANDBY2TRANSFERING trd) {
    setInternalState(State::TRANSFERING);
    UCB0IFG &= ~UCRXIFG;    //RX-Byte als abholt markieren
    UCB0TXBUF = *(trd.tx_byte);
    tx_sema.wait();

    if(trd.rx_byte != NULL)
        *(trd.rx_byte) = UCB0RXBUF;
}

void DFA_SPI::passTransition(trTRANSFERING2STANDBY trd) {
    setInternalState(State::STANDBY);
    tx_sema.Semaphore::signal();
}

////////////////////////////
////// CLASSIC API ////////
////////////////////////////

DFA_SPI::DFA_SPI()
{
    passTransition(trENTRY());
}

// Methods for CharDev for compatibility...
void DFA_SPI::send(const unsigned char *addr, unsigned int
    size) {
    trSTANDBY2TRANSFERING data(NULL, NULL);
    for(unsigned int i=0; i<size; i++)
    {
        data.tx_byte = &addr[i];
        passTransition(data);
    }
}

void DFA_SPI::recv(unsigned char *addr, unsigned int size) {
    trSTANDBY2TRANSFERING data(NULL, NULL);
    if(addr == NULL) return;
    for(unsigned int i=0; i<size; i++)
    {
        data.rx_byte = &addr[i];    //Setup Transition Data
            pointer buffer-addr
        passTransition(data);
    }
}

// Methods for Gate
void DFA_SPI::plugin()
```

Quelltext C.2 (Forts.)

```
{
    /* In Plugbox eintragen */
    plugbox.assign(Plugbox::UniSerial_B0, *this);
    //UCB0IE = UCTXIE;  //RXIE lieber, weil dann gibts
        interrupt wenn Transfer abgeschlossen, und nicht wenn
        TX-Puffer frei
    UCB0IE = UCRXIE;
}

bool DFA_SPI::prologue()
{
    return UCB0IV != 0;
}

void DFA_SPI::epilogue()
{
    passTransition(trTRANSFERING2STANDBY());
}

}

eUSCI_B::DFA_SPI CONFIG_eUSCI_B0_DFA_SPI_NAME;
```

Quelltext C.3: CPU Treiber (`dfa-cpu.h`) ☞ Kapitel 5.1.2.2.

```
#ifndef DFA_CPU_H
#define DFA_CPU_H

#include "drivers/dfa_driver.h"
#include "config.h"

class DFA_CPU : public DFA_Driver
{
public:
    struct State {
        enum state_t {
            UNINITIALIZED,
            AM,
            LowPM1,
            _STATE_MAX
        };
    };

    struct trENTRY : public transition_t {
        trENTRY() : transition_t(9) {}
    };
```

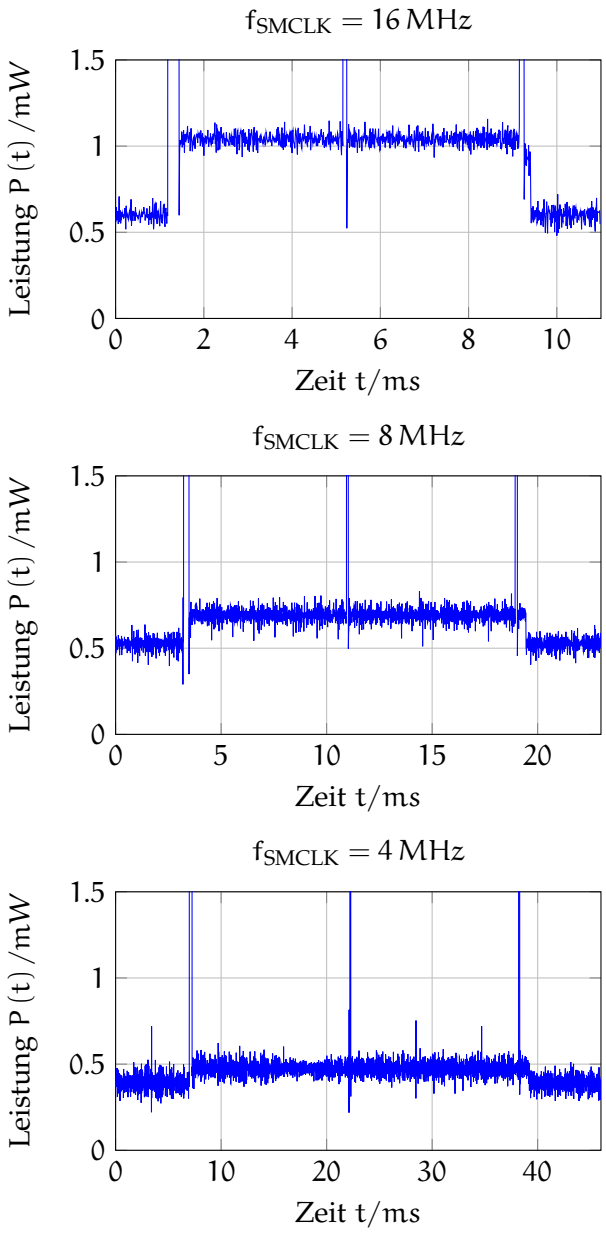


Abbildung C.1: Leistungsaufnahme vom MSP430 FR5969 bei der Übertragung von zwei Byte über SPI mit Bitrate 2 kbit/s für verschiedene Einspeisefrequenzen des EUSCI f_{SMCLK}. Die Leistungsaufnahme skaliert proportional mit f_{SMCLK}. ☞ Kapitel 5.1.1.3

Quelltext C.3 (Forts.)

```
    struct trAM2LowPM1 : public transition_t {
        trAM2LowPM1() : transition_t(0) {}
    };

    struct trLowPM12AM : public transition_t {
        trLowPM12AM() : transition_t(0) {}
    };


    static const power_uW_t statepower[];
    __attribute((always_inline)) inline int getPower_uW() {
        return statepower[getState()]; }
    __attribute((always_inline)) inline int getPower_uW(int
        state) { return statepower[state]; }

    DFA_CPU();

    void passTransition(trENTRY trd);
    void passTransition(trAM2LowPM1 trd);
    void passTransition(trLowPM12AM trd);

};

extern DFA_CPU CONFIG_DFA_CPU_NAME;     //see feature-file
    for defining the name

#endif // DFA_CPU
```

Quelltext C.4: CPU Treiber (`dfa-cpu.cc`) ☞ Kapitel 5.1.2.2.

```
#include "config.h"
#include "dfa-cpu.h"


const DFA_Driver::power_uW_t DFA_CPU::statepower[] = {
#ifdef CONFIG_DFA_CPU_BlindAccounting
    0,      // UNINITIALIZED
    1,      // AM
    1,      // LowPM1
#else
    2374,   // UNINITIALIZED
    2374,   // AM
    584,    // LowPM1
#endif
};
```

Quelltext C.4 (Forts.)

```
DFA_CPU::DFA_CPU()
{
    passTransition(trENTRY());
}

void DFA_CPU::passTransition(trENTRY trd) {
    setInternalState(State::AM);
}

void DFA_CPU::passTransition(trAM2LowPM1 trd) {
    setInternalState(State::LowPM1);
}

void DFA_CPU::passTransition(trLowPM12AM trd){
    setInternalState(State::AM);
}

DFA_CPU CONFIG_DFA_CPU_NAME;    //see feature-file for
    defining the name
```

Quelltext C.5: Display Treiber (`Sharp_LS013B4DN04.h`) ☞ Kapitel 5.1.3.2.

```
#ifndef SHARP_LS013B4DN04_H
#define SHARP_LS013B4DN04_H

#include <stdint.h>
#include "syscall/guarded_semaphore.h"
#include "Sharp_LS013B4DN04_VCOM_RefreshThread.h"
#include "drivers/dfa_driver.h"
#include "config.h"

/* Dimensions */
#define PIXELS_X 96
#define PIXELS_Y 96

/* Specifications */
#define fSCS 1              // Frame Frequency:         1Hz -
     60Hz
#define fSCLK 1000000       // SPI-Clock Frequency:     0.5
    MHz - 1MHz
#define fCOM 30             // COM Frequency:           0.5Hz
     - 30Hz

/* Special bits */
#define CLEAR_ALL_BIT 5     // M2
```

Quelltext C.5 (Forts.)

```
#define VCOM_BIT 6          // M1
#define MODE_BIT 7          // M0
#define TAILER8 0x00
#define TAILER16 0x0000

class Sharp_LS013B4DN04 : public DFA_Driver {
  friend class Sharp_LS013B4DN04_VCOM_RefreshThread;
private:
  static const uint8_t bitReverseTable[];
  typedef struct __attribute__ ((packed)) {
    uint8_t cmd;
    uint8_t line;
    uint8_t pixels[PIXELS_X/8];
    uint16_t tailer;
  } DataFrame_SingleLine;
  typedef struct __attribute__ ((packed)) {
    uint8_t cmd;
    uint8_t line;
    uint8_t pixels[PIXELS_X/8];
    uint8_t tailer;
  } DataFrame_FirstLine;
  typedef struct __attribute__ ((packed)) {
    uint8_t line;
    uint8_t pixels[PIXELS_X/8];
    uint8_t tailer;
  } DataFrame_IntermediateLine;
  typedef struct __attribute__ ((packed)) {
    uint8_t line;
    uint8_t pixels[PIXELS_X/8];
    uint16_t tailer;
  } DataFrame_LastLine;
  typedef struct __attribute__ ((packed)) {
    DataFrame_FirstLine first;
    DataFrame_IntermediateLine intermediate[PIXELS_Y/8 - 2];
    DataFrame_LastLine last;
  } DataFrame_AllLines;
  typedef struct __attribute__ ((packed)) {
    uint8_t cmd;
    uint8_t tailer;
  } DataFrame_ClearScreen;
  typedef struct __attribute__ ((packed)) {
    uint8_t cmd;
    uint8_t tailer;
  } DataFrame_ToggleVCOM;
  DataFrame_SingleLine transmitLinebuffer;
  DataFrame_ToggleVCOM vcom_frame;
  Guarded_Semaphore vcom_sema;
public:
```

Quelltext C.5 (Forts.)

```
typedef struct {
    uint8_t pixels[PIXELS_X / 8];
} LinePixels;
Sharp_LS013B4DN04();

// PUBLIC DFA DEFINITIONS
struct State {
  enum state_t {
    UNINITIALIZED, POWEROFF, DISABLED, ENABLED, _STATE_MAX
  };
};
struct trENTRY : public transition_t  {
  trENTRY() : transition_t(0) {}
};
struct trPOWEROFF2DISABLED : public transition_t  {
  trPOWEROFF2DISABLED() : transition_t(528235) {}
};
struct trDISABLED2POWEROFF : public transition_t  {
  trDISABLED2POWEROFF() : transition_t(0) {}
};
struct trENABLED2ENABLED_VCOM : public transition_t  {
  trENABLED2ENABLED_VCOM() : transition_t(74100) {}
};
struct trENABLED2ENABLED_LINE : public transition_t  {
  const LinePixels& data;
  uint8_t line;
  trENABLED2ENABLED_LINE(const LinePixels& data, uint8_t
      line) : transition_t(3800), data(data), line(line) {}
};
struct trENABLED2ENABLED_CLEAR : public transition_t  {
  trENABLED2ENABLED_CLEAR() : transition_t(12388) {}
};
struct trDISABLED2ENABLED : public transition_t  {
  trDISABLED2ENABLED() : transition_t(11081) {}
};
struct trENABLED2DISABLED : public transition_t  {
  trENABLED2DISABLED() : transition_t(1001) {}
};
static const power_uW_t statepower[];
__attribute((always_inline)) inline int getPower_uW() {
    return statepower[getState()]; }
__attribute((always_inline)) inline int getPower_uW(int
    state) { return statepower[state]; }

void passTransition(trENTRY trd);
void passTransition(trUNINITIALIZED2POWEROFF trd);
void passTransition(trPOWEROFF2DISABLED trd);
void passTransition(trDISABLED2POWEROFF trd);
```

Quelltext C.5 (Forts.)

```
  void passTransition(trENABLED2ENABLED_VCOM trd);
  void passTransition(trENABLED2ENABLED_LINE trd);
  void passTransition(trENABLED2ENABLED_CLEAR trd);
  void passTransition(trDISABLED2ENABLED trd);
  void passTransition(trENABLED2DISABLED trd);
  // END DFA DEFINITIONS
};
extern Sharp_LS013B4DN04 CONFIG_Sharp_LS013B4DN04_NAME;
#endif // SHARP_LS013B4DN04_H
```

Quelltext C.6: Display Treiber (`Sharp_LS013B4DN04.cc`) ☞ Kapitel 5.1.3.2.

```
#include "drivers.h"
#include "drivers/gpio.h"
#include "config.h"
#include "Sharp_LS013B4DN04.h"

#define displayBus CONFIG_Sharp_LS013B4DN04_COMM_NAME
//...
const uint8_t Sharp_LS013B4DN04::bitReverseTable[] = {
//...
};
const DFA_Driver::power_uW_t Sharp_LS013B4DN04::statepower[]
    = {
  0,      //UNINITIALIZED,
  0,      //POWEROFF, 0
  2,      //DISABLED, 2
  3       //ENABLED,  3
};
void Sharp_LS013B4DN04::passTransition(trENTRY trd) {
  setInternalState(State::UNINITIALIZED);
  transmitLinebuffer.tailer = TAILER16;
  vcom_frame.cmd = 0;
  vcom_frame.tailer = TAILER8;
  power_low();
  disp_low();
  cs_low();
  setOutput(CONFIG_Sharp_LS013B4DN04_POWER_CTRL_PORT,
      CONFIG_Sharp_LS013B4DN04_POWER_CTRL_PIN);
  setOutput(CONFIG_Sharp_LS013B4DN04_DISP_PORT,
      CONFIG_Sharp_LS013B4DN04_DISP_PIN);
  setOutput(CONFIG_Sharp_LS013B4DN04_CS_PORT,
      CONFIG_Sharp_LS013B4DN04_CS_PIN);
  setInternalState(State::POWEROFF);
}
```

Quelltext C.6 (Forts.)

```
void Sharp_LS013B4DN04::passTransition(trPOWEROFF2DISABLED
    trd) {
  power_high();
  setInternalState(State::DISABLED);
}

void Sharp_LS013B4DN04::passTransition(trDISABLED2POWEROFF
    trd) {
  power_low();
  setInternalState(State::POWEROFF);
}

void Sharp_LS013B4DN04::passTransition(trENABLED2ENABLED_VCOM
    trd) {
  vcom_frame.cmd ^= 1 << VCOM_BIT;
  displayBus.wait();
  cs_high();
  displayBus.send((uint8_t*)&vcom_frame, sizeof(vcom_frame));
  cs_low();
  displayBus.signal();
  setInternalState(State::ENABLED);
}

void Sharp_LS013B4DN04::passTransition(trENABLED2ENABLED_LINE
    trd) {
  transmitLinebuffer.cmd = (1<<MODE_BIT) | vcom_frame.cmd;
  transmitLinebuffer.line = bitReverseTable[trd.line];
  for(unsigned int t=0; t<sizeof(transmitLinebuffer.pixels);
      t++) {
      transmitLinebuffer.pixels[t] = trd.data.pixels[t];
  }
  displayBus.wait();
  cs_high();
  displayBus.send((uint8_t*)&transmitLinebuffer, sizeof(
      transmitLinebuffer));
  cs_low();
  displayBus.signal();
  setInternalState(State::ENABLED);
}

void Sharp_LS013B4DN04::passTransition(
    trENABLED2ENABLED_CLEAR trd) {
  DataFrame_ClearScreen clear_frame;
  clear_frame.cmd = (1 << CLEAR_ALL_BIT) | vcom_frame.cmd;
  clear_frame.tailer = TAILER8;
  displayBus.wait();
  cs_high();
```

Quelltext C.6 (Forts.)

```
  displayBus.send((uint8_t*)&clear_frame, sizeof(clear_frame)
      );
  cs_low();
  displayBus.signal();
  setInternalState(State::ENABLED);
}
void Sharp_LS013B4DN04::passTransition(trDISABLED2ENABLED trd
    ) {
  disp_high();
  vcom_sema.signal();
  setInternalState(State::ENABLED);
}
void Sharp_LS013B4DN04::passTransition(trENABLED2DISABLED trd
    ) {
  vcom_sema.wait();   //Sollte nie blockieren.
  disp_low();
  setInternalState(State::DISABLED);
}
Sharp_LS013B4DN04::Sharp_LS013B4DN04() : vcom_sema(0) {
  passTransition(trENTRY());
}
Sharp_LS013B4DN04 CONFIG_Sharp_LS013B4DN04_NAME;
```

Quelltext C.7: Implementierung vom VCOM Signalisierungsprozess für den Display Treiber (`Sharp_LS013B4DN04_VCOM_RefreshThread.cc`) ☞ Kapitel 5.1.3.2.

```
#include "Sharp_LS013B4DN04_VCOM_RefreshThread.h"
#include "Sharp_LS013B4DN04.h"
#include "syscall/guarded_buzzer.h"
#include "config.h"
#define disp CONFIG_Sharp_LS013B4DN04_NAME
DeclareThread(Sharp_LS013B4DN04_VCOM_RefreshThread,
    sharp_LS013B4DN04_VCOM_RefreshThread, 256);

void Sharp_LS013B4DN04_VCOM_RefreshThread::action() {
  Guarded_Buzzer buz;
  buz.set(CONFIG_Sharp_LS013B4DN04_VCOM_INTERVAL_MS);
  while(1) {
    disp.vcom_sema.wait();      //Wartet ob suspendiert
    disp.vcom_sema.signal();    //Sem. zuruecksetzen
    disp.toggle_VCOM();
    buz.sleep();
  }
}
```

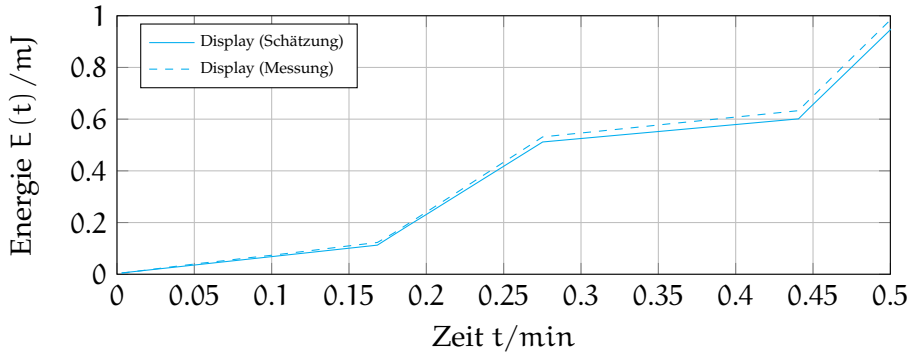


Abbildung C.2: Vergrößerte Darstellung des Energieverbrauchs vom Display während des ersten Teils vom Evaluationsexperiment (Abbildung 6.4). ☞ Kapitel 6.1.3.

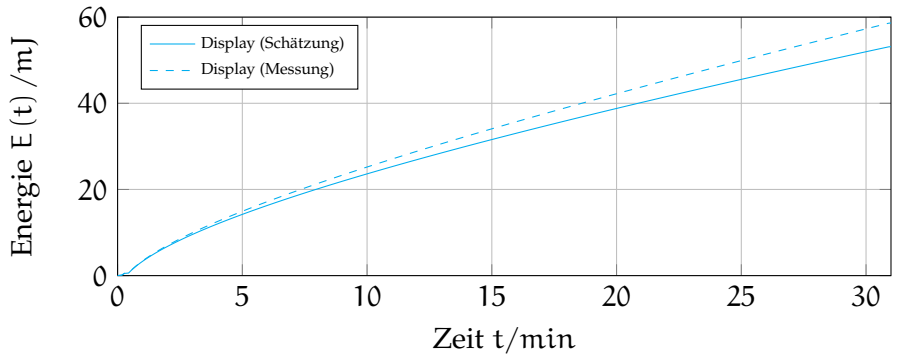


Abbildung C.3: Vergrößerte Darstellung des Energieverbrauchs vom Display während des zweiten Teils vom Evaluationsexperiment (Abbildung 6.5). ☞ Kapitel 6.1.3.

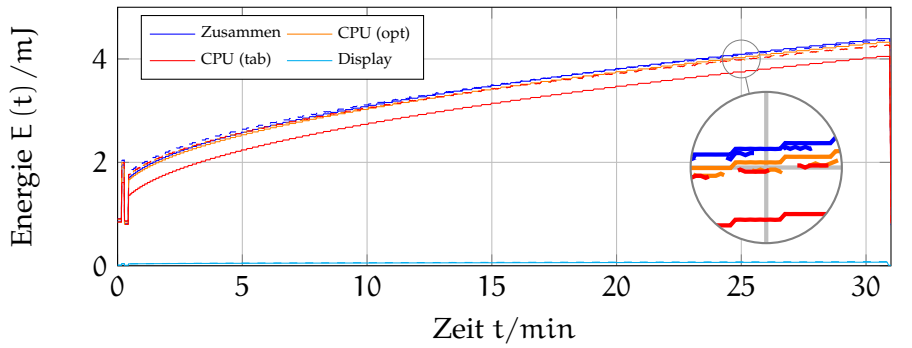


Abbildung C.4: Zweiter Teil von der Zunahme des Energieverbrauchs zwischen den Messintervallen während des Experiments aus Abbildung 6.5. Gestrichelte Graphen stehen für MIMOSA Messwerte, Ergebnisse des Profilers sind durchgezogen dargestellt. ☞ Kapitel 6.1.3.

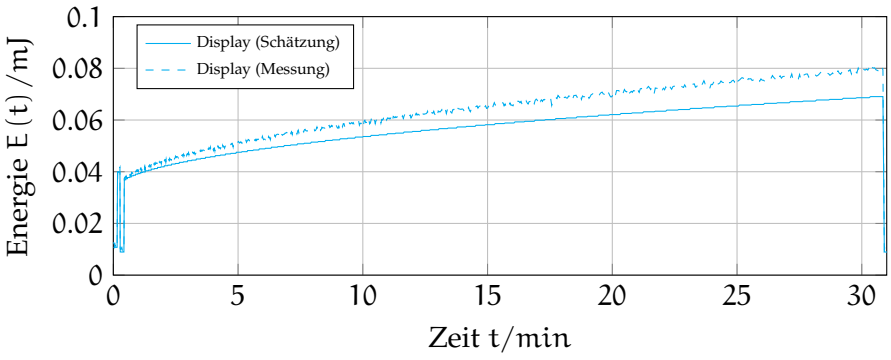


Abbildung C.5: Vergrößerte Darstellung der Zunahme vom Energieverbrauch aus Abbildung C.4, skaliert für das Display. ☞ Kapitel 6.1.3.

Quelltext C.8: Implementierung vom `typewriter.h` ☞ Kapitel A.

```
#include <stdint.h>
#include "config.h"
#include "drivers/dfa-drivers/display/Sharp_LS013B4DN04.h"
#include "syscall/guarded_semaphore.h"

//shortcut
#define twDisplay CONFIG_TYPEWRITER_DISPLAY

#define __FONTFILE(X) #X
#define _FONTFILE(X) __FONTFILE(X)
#define FONTFILE(FONTNAME) _FONTFILE(fonts/FONTNAME.txt)

#if CONFIG_TYPEWRITER_FONT == LCD4X6A
  #define TW_CHARACTER_WIDTH 4
  #define TW_CHARACTER_HEIGHT 6
#endif

#define TW_CHARACTERS_IN_LINE (DIM_X/TW_CHARACTER_WIDTH)
#define TW_LINE_COUNT (DIM_Y/(TW_CHARACTER_HEIGHT+
    CONFIG_TYPEWRITER_LINESPACE))

template<int DIM_X, int DIM_Y>
class Typewriter {
private:
  static const uint8_t font[128*6];
  uint8_t charbuffer[TW_LINE_COUNT][TW_CHARACTERS_IN_LINE];
  int xpos, ypos;
  Guarded_Semaphore mutex;

  void flush();
  void scroll();
```

Quelltext C.8 (Forts.)

```
public:
    Typewriter() : xpos(0), ypos(0), mutex(1) {}
    void clear();
    void beginning();
    void put(const char c);
};

extern Typewriter<CONFIG_DISPLAY_XPIXELS,
    CONFIG_DISPLAY_YPIXELS> typewriter;

template<int DIM_X, int DIM_Y>
const uint8_t Typewriter<DIM_X, DIM_Y>::font[128*6] = {
  #include FONTFILE(CONFIG_TYPEWRITER_FONT)
};

template<int DIM_X, int DIM_Y>
inline void Typewriter<DIM_X, DIM_Y>::scroll() {
  for(int y=1; y<TW_LINE_COUNT; y++) {
    for(int x=0; x<TW_CHARACTERS_IN_LINE; x++) {
      charbuffer[y-1][x] = charbuffer[y][x];
    }
  }
  for(int x=0; x<TW_CHARACTERS_IN_LINE; x++) {
    charbuffer[TW_LINE_COUNT-1][x] = 0;
  }
}

template<int DIM_X, int DIM_Y>
void Typewriter<DIM_X, DIM_Y>::flush() {
  Sharp_LS013B4DN04::LinePixels linePixels;
  unsigned int line_tw = 0;
  unsigned int line_px = 1;
  while(line_tw < TW_LINE_COUNT) {         // For each
      Typewriterline
    for(unsigned int ypos_px=0; ypos_px < TW_CHARACTER_HEIGHT
        ; ypos_px++, line_px++) {        // For each Pixelline
         in Typewriterline
      for(unsigned int xpos_octet=0; xpos_octet < DIM_X/8;
          xpos_octet++) {                          // For each
           Pixel-octet in Pixelline
        //Each iteration fills the pixel data of two
            character neighbours
        //because of their 4px width
        uint8_t octet;
        octet = font[charbuffer[line_tw][xpos_octet*2    ]*6
            + ypos_px] << 4;
```

Quelltext C.8 (Forts.)

```
        octet |= font[charbuffer[line_tw][xpos_octet*2 + 1]*6
            + ypos_px];
#ifdef CONFIG_DISPLAY_INVERTED
        linePixels.pixels[xpos_octet] = ~octet;
#else
        linePixels.pixels[xpos_octet] = octet;
#endif
      }
      twDisplay.transmit_line(linePixels, line_px);
    }
#if CONFIG_TYPEWRITER_LINESPACE > 0
    // Linespace
    for(unsigned int ypos_px=0; ypos_px <
        CONFIG_TYPEWRITER_LINESPACE; ypos_px++, line_px++) {
      for(unsigned int xpos_octet=0; xpos_octet < DIM_X/8;
          xpos_octet++) {
  #ifdef CONFIG_DISPLAY_INVERTED
        linePixels.pixels[xpos_octet] = 0xFF;
  #else
        linePixels.pixels[xpos_octet] = 0x00;
  #endif
      }
      twDisplay.transmit_line(linePixels, line_px);
    }
#endif
    // Next line
    line_tw++;
  }
}

template<int DIM_X, int DIM_Y>
void Typewriter<DIM_X, DIM_Y>::put(const char c) {
  mutex.wait();
  if(ypos == TW_LINE_COUNT) {
    scroll();
    ypos--;
  }
  charbuffer[ypos][xpos++] = c;
  if(xpos == TW_CHARACTERS_IN_LINE || c == '\n') {
    flush(ypos);
    xpos = 0;
    ypos++;
  }
  else if(c == '\r') {
    flush(ypos);
    xpos = 0;
  }
  mutex.signal();
```

Quelltext C.8 (Forts.)

```
}

template<int DIM_X, int DIM_Y>
void Typewriter<DIM_X, DIM_Y>::clear() {
  mutex.wait();
  xpos = 0;
  ypos = 0;
  for(int y=0; y<TW_LINE_COUNT; y++) {
    for(int x=0; x<TW_CHARACTERS_IN_LINE; x++) {
      charbuffer[y][x] = 0;
    }
  }
  flush();
  mutex.signal();
}

template<int DIM_X, int DIM_Y>
void Typewriter<DIM_X, DIM_Y>::beginning() {
  mutex.wait();
  xpos = 0;
  ypos = 0;
  mutex.signal();
}
```

LITERATURVERZEICHNIS

[Ana13] Analog Devices: *ADXL362 Data Sheet Rev B*. http://www.analog.com/static/imported-files/data_sheets/ADXL362.pdf. Version: Februar 2013

[Beh14] Behrmann, Gerd: *UPPAAL CORA*. http://people.cs.aau.dk/~adavid/cora/index.html. Version: Oktober 2014

[BFH+01] Behrmann, Gerd ; Fehnker, Ansgar ; Hune, Thomas ; Larsen, Kim ; Pettersson, Paul ; Romijn, Judi ; Vaandrager, Frits: Minimum-Cost Reachability for Priced Time Automata. Version: 2001. In: Di Benedetto, Maria-Domenica (Hrsg.) ; Sangiovanni-Vincentelli, Alberto (Hrsg.): *Hybrid Systems: Computation and Control* Bd. 2034. Springer Berlin Heidelberg, 2001. – DOI 10.1007/3–540–45351–2_15. – ISBN 978–3–540–41866–5, 147-161

[BGS12] Buschhoff, Markus ; Günter, Christian ; Spinczyk, Olaf: A unified approach for online and offline estimation of sensor platform energy consumption. In: *8th InternationalWireless Communications and Mobile Computing Conference (IWCMC)*, 2012, S. 1154–1158

[BGS14] Buschhoff, Markus ; Günter, Christian ; Spinczyk, Olaf: MIMOSA, a Highly Sensitive and Accurate Power Measurement Technique for Low-Power Systems. Version: 2014. In: Langendoen, Koen (Hrsg.) ; Hu, Wen (Hrsg.) ; Ferrari, Federico (Hrsg.) ; Zimmerling, Marco (Hrsg.) ; Mottola, Luca (Hrsg.): *Real-World Wireless Sensor Networks* Bd. 281. Springer International Publishing, 2014. – DOI 10.1007/978–3–319–03071–5_16. – ISBN 978–3–319–03070–8, S. 139–151

[BMM13] Berthier, Nicolas ; Maraninchi, Florence ; Mounier, Laurent: Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems. In: *ACM Transactions on Embedded Computing Systems (TECS)* 12 (2013), März, Nr. 1s, 39:1–39:26. – DOI 10.1145/2435227.2435235. – ISSN 1539–9087

[CGBRP11] Chen, Hui ; Godet-Bar, G. ; Rousseau, F. ; Petrot, F.: Me3D: A model-driven methodology expediting embedded device driver development. In: *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, 2011. – ISSN Pending, S. 171–177

[Fra14] Fraunhofer-Gesellschaft : *Logistik Beratung, Planung & Forschung - Fraunhofer IML*. http://www.iml.fraunhofer.de. Version: Oktober 2014

[Göo8] Göbel, Holger: *Einführung in die Halbleiter-Schaltungstechnik*. 3. bearb. und erw. Aufl. Berlin : Springer, 2008. – ISBN 978–3–540–69288–1

[Hit14] Hitex Development Tools GmbH: *PowerScale with ACM technology*. http://www.hitex.com/index.php?id=powerscale. Version: Oktober 2014

[Joh14] John W. Eaton: *GNU Octave*. https://www.gnu.org/software/octave/. Version: Dezember 2014

[KPMBo8] Kellner, S. ; Pink, M. ; Meier, D. ; Blass, E.-O.: Towards a Realistic Energy Model for Wireless Sensor Networks. In: *Fifth Annual Conference on Wireless on Demand Network Systems and Services*, 2008, S. 97–100

[MFHHo5] Madden, Samuel R. ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei: TinyDB: An Acquisitional Query Processing System for Sensor Networks. In: *ACM Trans. Database Syst.* 30 (2005), März, Nr. 1, 122–173. – DOI 10.1145/1061318.1061322. – ISSN 0362–5915

[PHZ+11] Pathak, Abhinav ; Hu, Y. C. ; Zhang, Ming ; Bahl, Paramvir ; Wang, Yi-Min: Fine-grained Power Modeling for Smartphones Using System Call Tracing. In: *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA : ACM, 2011 (EuroSys '11). – ISBN 978–1–4503–0634–8, 153–168

[pUS14] pure-systems GmbH ; Urban, Matthias ; Spinczyk, Olaf: *Documentation: AspectC++ Language Reference*. http://www.aspectc.org/doc/ac-languageref.pdf. Version: Oktober 2014

[Sha14] Sharp Microelectronics of the Americas: *Application Information for Sharp's LS013B4DN04 MemoryLCD*. http://www.sharpmemorylcd.com/resources/LS013B4DN01_Application_Info.pdf. Version: November 2014

[SKWMo1] Steinke, Stefan ; Knauer, Markus ; Wehmeyer, Lars ; Marwedel, Peter: An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In: *PATMOS*. Yverdon (Switzerland), September 2001

[Spi14] SPINCZYK, Olaf: *The Home of AspectC++*. http://www.aspectc.org/. Version: Oktober 2014

[SVP09] SECELEANU, C. ; VULGARAKIS, A. ; PETTERSSON, P.: REMES: A Resource Model for Embedded Systems. In: *14th IEEE International Conference on Engineering of Complex Computer Systems*, 2009, S. 84–94

[TBB+13] TANGUY, J. ; BECHENNEC, J.-L. ; BRIDAY, M. ; DUBE, S. ; ROUX, O.H.: Device driver synthesis for embedded systems. In: *IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, 2013. – ISSN 1946–0740, S. 1–8

[Tec14a] TECHNISCHE UNIVERSITÄT DORTMUND: *Betriebssystembau*. http://ess.cs.tu-dortmund.de/DE/Teaching/WS2013/BSB/. Version: Oktober 2014

[Tec14b] TECHNISCHE UNIVERSITÄT DORTMUND: *Lehrstuhl für Eingebettete Systeme - Arbeitsgruppe Eingebettete Systemsoftware*. http://ess.cs.uni-dortmund.de/DE/Home/index.html. Version: Oktober 2014

[Tex14a] TEXAS INSTRUMENTS: *MSP430 FRAM Quality and Reliability (Rev. A)*. http://www.ti.com/lit/pdf/slaa526. Version: Oktober 2014

[Tex14b] TEXAS INSTRUMENTS: *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family*. http://www.ti.com/lit/pdf/slau367. Version: Oktober 2014

[Tex14c] TEXAS INSTRUMENTS: *MSP430FR5969 LaunchPad Evaluation Kit*. http://www.ti.com/ww/en/launchpad/launchpads-msp430-msp-exp430fr5969.html#tabs. Version: Oktober 2014

[Tex14d] TEXAS INSTRUMENTS: *MSP430FR59xx Mixed-Signal Microcontrollers*. http://www.ti.com/lit/gpn/msp430fr5969. Version: Oktober 2014

[The14] THE MATHWORKS, INC.: *MATLAB and Simulink for Technical Computing*. http://de.mathworks.com/. Version: Dezember 2014

[TRJ02] TAN, T.K. ; RAGHUNATHAN, A. ; JHA, N.K.: Embedded operating system energy analysis and macro-modeling. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors.*, 2002. – ISSN 1063–6404, S. 515–522

[TSG12] Tietze, Ulrich ; Schenk, Christoph ; Gamm, Eberhard: *Halbleiter-Schaltungstechnik*. 14. überarb. und erw. Aufl. Heidelberg : Springer-Verlag GmbH, 2012. – ISBN 978–3–642–31025–6

[Wed10] Weder, A.: An Energy Model of the Ultra-Low-Power Transceiver nRF24L01 for Wireless Body Sensor Networks. In: *Second International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*, 2010, S. 118–123

[WM03] Wang, Shaojie ; Malik, S.: Synthesizing operating system based device drivers in embedded systems. In: *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, 2003, S. 37–44

[WY07] Wang, Qin ; Yang, Woodward: Energy Consumption Model for Power Management in Wireless Sensor Networks. In: *SECON '07. 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks.*, 2007, S. 142–151