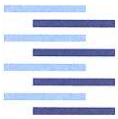

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Richard Günther

Vorausschauend-adaptives Energiemanagement auf einem energieautonomen eingebetteten System

Fakultät Technik und Informatik

Faculty of Engineering and Computer Science

Richard Günther

Vorausschauend-adaptives Energiemanagement auf einem energieautonomen eingebetteten System

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Mechatronik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Rasmus Rettig
Zweitgutachter: Prof. Dr. rer. nat. Thomas Lehmann

Eingereicht am: 15.04.2014

Richard Günther

Thema der Arbeit
Vorausschauend-adaptives Energiemanagement auf einem energieautonomen eingebetteten System

Stichworte
Energiemanagement, eingebettete Systeme, energieautonome Systeme, vorausschauend, adaptiv

Kurzzusammenfassung
In dieser Bachelorthesis wird ein vorausschauendes Energiemanagement auf einem autonom mit Energie versorgten eingebetteten System prototypisch umgesetzt. Es wird ein Algorithmus entwickelt, welcher gestützt von Wettervorhersagen die Energieaufnahme des eingebetten Systems vorausschauend beeinflusst.

Title of the paper
Predictive and adaptive powermanagement on an autonomously powered embedded system

Keywords
power management, embedded systems, autonomously powered systems, predictive, adaptive

Abstract
In this thesis a prototype for predictive power management on an autonomously powered embedded system is implemented. An algorithm is developed which predictively controls the power consumption of the embedded device using weather forecast data.

Inhaltsverzeichnis

1. Einführung

1.1. Motivation

Energiemanagement in eingebetteten Systemen ist besonders bei mobilen Anwendungen von zentraler Bedeutung. Die Funktionalität soll möglichst lange ohne Aufladung gewährleistet sein, aber dennoch die Funktionsfähigkeit so wenig wie möglich eingeschränkt werden.

Besonders kritisch ist das Energiemanagement, wenn der Füllstand des Akkus sehr niedrig ist. Ein hoher Energieverbrauch kann dann zur Folge haben, dass es zum kurzfristigen Ausfall des Gerätes kommt. Die Funktion kann erst wieder durch das Aufladen des Akkus hergestellt werden. Die Aufladung muss dabei jedoch immer aktiv vom Benutzer eingeleitet werden. Viele Geräte versuchen daher den Energieverbrauch bei niedrigem Akkustand zu reduzieren, indem z. B. weniger wichtige Funktionen deaktiviert werden.

Ein weiterer Aspekt kommt hinzu, wenn man ein System betrachtet, welches autonom mit Energie versorgt wird (z. B. Sonnen- oder Windenergie (*Energy Harvesting*)). Die Aufladung wird dann nicht aktiv vom Benutzer gesteuert, sondern ist von den Umweltbedingungen abhängig. Ein solches System könnte ausfallen, wenn sich eine ungünstige Konstellation aus niedrigem Akkustand und fehlender Energiezufuhr einstellt. Ein Ausfall ist hier jedoch wesentlich kritischer, denn eine manuelle Aufladung ist häufig nicht möglich.

Während die manuelle Aufladung durch einen Benutzer immer kontrolliert und mit einer gewissen Regelmäßigkeit durchgeführt wird, treten Umwelteinflüsse, wie Wind und Sonne, in Abständen auf, die teilweise durch den Zufall bestimmt werden. Durch Erfahrungen und Analysen aus der Meteorologie können zukünftige Wetterbedingungen mit einer begrenzten Genauigkeit vorhergesagt werden. Mit Hilfe dieser Vorhersagen kann abgeschätzt werden, wann die nächste Aufladung stattfindet und wie stark diese ausfallen wird. Damit ließen sich die Zeiten bestimmen, in denen sich möglicherweise ein kritischer Akkustand einstellt. Dadurch kann bereits im voraus Energie gespart werden, um diese Zeiträume zu überbrücken und so die Funktionalität durchgängig zu gewährleisten.

Im Rahmen dieser Arbeit wird ein System entwickelt, welches ein solches vorausschauendes Energiemanagement auf einem autonomen eingebetteten System umsetzt. Die Arbeit wird in Kooperation mit der Firma Garz & Fricke GmbH erstellt.

1.2. Zielsetzung und Aufgabenstellung

In dieser Arbeit wird ein vorausschauendes Energiemanagement-System auf einem autonom mit Energie versorgten und eingebetteten System realisiert. Dabei soll das autonome Gerät dynamisch den eigenen Energieverbrauch steuern und kritische Akkuzustände vermeiden. Die Regelung des Energieverbrauchs soll dazu nicht nur vom aktuellen Zustand des Energiespeichers ausgehen, sondern auch mit Hilfe der Vorhersage zukünftiger Energiespeicherzustände erfolgen. Dazu stützt sich das System auf Vorhersage-Daten über zukünftig verfügbare Energiemengen.

Zunächst wird der Stand der Technik von Energiemanagement-Methoden auf eingebetteten Systemen und vorausschauenden Regelungssystemen beschrieben.

Im Anschluss wird ein Algorithmus entwickelt, welcher ein vorausschauendes Energiemanagement auf einem eingebetteten System realisiert und dieser prototypisch in einer Software umgesetzt. Als Basis für die Umsetzung wird ein Linux-basiertes Betriebssystem verwendet, welches ebenfalls im Zuge dieser Arbeit auf dem eingebetteten Gerät in Betrieb genommen wird.

Abschließend wird ein prototypischer Aufbau des Gesamtsystems getestet und die Ergebnisse ausgewertet. Hierbei wird untersucht, wie leistungsfähig das entwickelte vorausschauendes Energiemanagement-System auf dem verwendeten eingebetteten Gerät ist.

2. Stand der Technik

Das folgende Kapitel beschreibt vorhandene Technologien, welche dem zu entwickelnden System als Grundlage dienen.

Zunächst werden prädiktive Regelsysteme und deren Anwendungsgebiete erläutert, sowie zwei konkrete Beispiele vorgestellt. Im zweiten Teil werden Energiemanagement-Mechanismen für Mikrocontroller und Computer im Überblick dargestellt, wobei der Fokus auf eingebettete Systeme gelegt wird.

2.1. Prädiktive Systeme

Vorausschauende Systeme werden im Bereich der Regelungstechnik angewendet, die sogenannte „modellbasierte prädiktive Regelung“ (Model based Predictive Control (MPC)). Prädiktive Regelungsalgorithmen verwenden ein Modell für das dynamische Verhalten des Prozesses im laufenden Betrieb des Reglers. Das Modell dient dabei zur Vorhersage des zukünftigen Verhaltens der Regelgrößen des Systems. [4, vgl. Vorwort]. Durch das Wissen über zukünftige Werte der Regelgrößen, kann die Stellgröße bereits im Voraus angepasst werden. Dadurch wird die Sollwertabweichung erheblich verbessert und Über- und Unterschwinger der Regelgröße weitgehend vermieden.

Die Genauigkeit der Vorhersage der Regelgrößen spielt dabei eine entscheidende Rolle, denn die Regelung stützt sich auf diese Werte, um die Stellgröße zu beeinflussen. Da die Genauigkeit der Vorhersage von der Güte des verwendeten Modells abhängt, werden prädiktive Regelungssystem häufig adaptiv ausgelegt. So können die Parameter des Modells im laufenden Betrieb verändert und auf den echten Prozess angepasst werden. So kann das System auch auf schleichende Veränderungen am Prozess reagieren, welche bei der initialen Auslegung des Modells nicht berücksichtigt wurden. Typisch hierfür sind Alterungs- oder Abnutzungsprozesse von Komponenten im zu regelnden Prozess.

Anhand der beiden folgenden Beispiele sollen typische Teilaufgaben und Randbedingungen verdeutlicht werden, die auch für die zu lösende Aufgabenstellung zutreffen.

2.1.1. Prädiktiver, selbstadaptiver Heizungsregeler

Eine typische Anwendung für eine prädiktive und adaptive Steuerung ist die Regelung einer Heizungsanlage. Eine solche intelligente Heizungsregelung ist inbesondere in großen Gebäuden von Bedeutung, um die Energieeffizienz zu steigern. Die Regelung des Heizsystems

hängt dabei von der aktuellen Außentemperatur, der Innentemperatur und Anlagen-internen Größen ab. Nachfolgend wird ein solcher Heizungsregler am Beispiel eines Produkts der Firma Siemens AG vorgestellt [24].

Ein herkömmlicher Heizungsregler kann nur auf die aktuellen Daten der Sensoren zurückgreifen. So kann es zu Überhitzungs- oder Unterkühlungsphasen kommen, da die Randbedingungen des Gebäudes nicht berücksichtigt werden oder das Heizungssystem verzögert reagiert. [24, S. 6] Durch Integration eines Modells des Gebäudes und durch Vorhersagen über zukünftiger Sensorwerte, kann der Regler die Heizung wesentlich effizienter steuern. So kann die Heizleistung bereits vor Erreichen der gewünschten Temperatur zurückgenommen werden, um beispielsweise das „Nachheizen“ der Anlage zu berücksichtigen. Durch Vorhersagen über den zukünftigen Verlauf der Außentemperatur kann weiterhin der optimale Ein- bzw. Ausschaltpunkt der Heizanlage berechnet werden. Auf diese Weise werden Sollwert-Abweichungen der Raumtemperatur minimiert. Die Abbildung 2.1 zeigt vereinfacht die Wirkungsweise des prädiktiven Heizungsreglers.

Die Energieeinsparung der prädiktiven Regelung gegenüber gut eingestellten klassichen Heizungsregelungen beträgt ca. 8-13 % bei vergleichbarer Regelgüte. Die Sollwert-Abweichungen der prädiktiven Heizungsregelung erzielen im Vergleich zu den Referenzanlagen ausgezeichnete Werte und tragen somit zum Komfort bei. [24, S. 14-15]

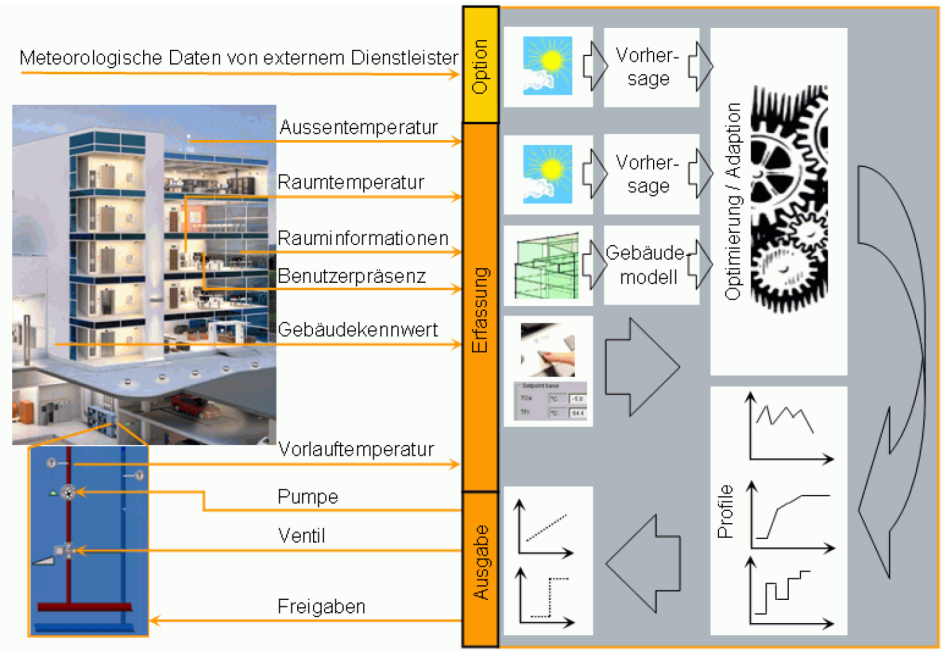

Abbildung 2.1.: Vereinfachte Darstellung der Wirkweise der prädiktiven Heizungsregelung [24, S. 7]

2.1.2. Vorausschauende Getriebesteuerung in Nutzfahrzeugen

Eine weitere Anwendung für eine vorausschauendes System ist die Steuerung von Automatikgetrieben in Nutzfahrzeugen. Dabei wird durch eine prädiktive Schaltstrategie insbesondere der Kraftstoffverbrauch und die Durchschnitts- bzw. Transportgeschwindigkeit des Fahrzeugs optimiert.

Die Firma ZF Friedrichshafen AG hat eine vorausschauende Schaltstrategie „PreVision GPS“ für das Nutzfahrzeuggetriebe „TraXon“ entwickelt [27]. Zur Berechnung der Schaltpunkte wird, zusätzlich zu aktuellen Getriebe- und Fahrzeugdaten, das zu befahrende Streckenprofil berücksichtigt. Unter Zuhilfenahme des GPS-Systems des Fahrzeugs, kann das System beispielsweise bevorstehende Steigungen erkennen und frühzeitig einen niedrigeren Gang wählen, sodass der Geschwindigkeitsverlust auf der Steigung minimiert wird. Des weiteren erkennt das System Streckenabschnitte, die vom Fahrzeug kraftstoffsparend durchrollt werden können. So wird insgesamt auch die Anzahl der Schaltvorgänge minimiert, was außerdem zu einer höheren Lebensdauer des Getriebes führt.

Durch dieses System steigt die Wirtschaftlichkeit eines Nutzfahrzeugs durch Kraftstoffeinsparung und Erhöhung der Transportgeschwindigkeit. [27]

2.2. Energiemanagement auf eingebetteten Systemen

Auf eingebetteten Systemen werden meist System On Chip (SoC)-Prozessoren eingesetzt, welche neben der CPU noch weitere Controller auf einem Chip vereinen. Ein solcher SoC hat häufig bereits ein integriertes Energiemanagement. Im folgenden wird der Stand der Technik zur Energieverwaltung auf Prozessoren und Peripherie-Geräten gemeinsam vorgestellt, da diese häufig auf einem SoC verschmelzen.

Die folgenden Mechanismen können für das Energiemanagement von eingebetteten Systemen verwendet werden:

- Power States (ACPI-States)
- Clock-Management
- Peripherie-Management (Abschaltung, Energiesparmodi, Clock-Gating)

2.2.1. Power States (ACPI-States)

Grundsätzlich besitzt ein Mikroprozessor verschiedene *Power States*, die den Aktivitätsgrad des Prozessors widerspiegeln. Diese werden häufig nach dem Advanced Configuration and Power Interface (ACPI)-Standard definiert und auch als *ACPI-States* bezeichnet. Die Standardisierung hat den Vorteil, dass das Betriebssystem beim Anfahren der verschiedenen Modi, unabhängig vom Prozessortyp, die zu erwartende Prozessoraktivität kennt. Die Abbil-

dung 2.2 gibt eine Übersicht über die verfügbaren ACPI-States eines modernen Intel Core 4th Generation Prozessors [13].

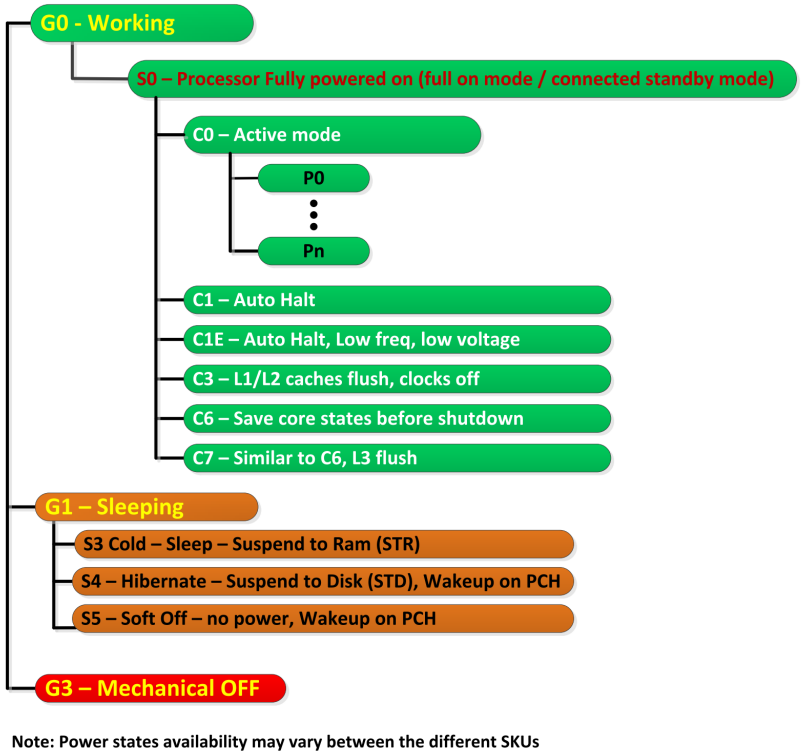

Abbildung 2.2.: ACPI-States eines Intel Core 4th Gen. Prozessors [13, S. 49]

Es werden aktive und passive globale System-Zustände unterschieden (ACPI S-State). Im Aktiv-Modus ist der Prozessor im „arbeitsfähigen“ Zustand und verarbeitet entweder aktiv Befehle oder wartet auf das Eintreffen neuer Befehle. Im passiven Modus (Energiesparmodus) sind weite Teile des Prozessors deaktiviert und es können keine Aufgaben bearbeitet werden, da der Prozessorkern abgeschaltet ist. Es werden unterschiedlich tiefe Energiesparmodi unterschieden, wobei im Normalfall mindestens *Suspend to Ram* und *Suspend to Disk* unterstützt werden. Der „Soft-Off“-Modus (S5) ist der Normalzustand für einen heruntergefahrenen Computer. In diesem Zustand wird der Prozessor nur durch Bestätigung des Einschaltknopfes wieder in einen Aktivmodus versetzt. An diesen System-Zustand sind nicht nur der Prozessorkern, sondern auch weitere integrierte Module eines SoC-Prozessors, wie z.B. Speicher- und Bus-Controller gekoppelt.

Die CPU ist nur im Systemzustand S0 aktiv und kann Instruktionen verarbeiten. Für den Zustand des Prozessors im Aktivmodus werden weitere ACPI-States unterschieden (C-State). Diese Beschreiben das Verhalten des Prozessors für die Zeiträume zwischen den Instruktionen und bei der aktiven Bearbeitung von Aufgaben. So kann sich der Prozessor automatisch in leichte Energiesparmodi versetzen, während er auf das Eintreffen neuer In-

struktionen wartet. Da die Leerlaufzeit zwischen zwei Instruktionen sehr kurz sein kann, muss der Wechsel der Modi sehr schnell sein, damit keine spürbare Latenz auftritt. Die C-State Modi unterscheiden sich gegenüber den S-State-Energiesparmodi dadurch, dass nur kleine Teile des Prozessors für kurze Zeit deaktiviert werden. Dabei gilt, je tiefer der Energiesparmodus, desto größer ist die Energieersparnis, doch umso länger dauert der Ein- und Austritt für diesen Zustand. Während der aktiven Bearbeitung von Instruktionen kann die CPU verschiedene *Performance-States* einnehmen, die Taktfrequenz und Spannungsversorgung des Prozessors regulieren, sodass der Energieverbrauch gesenkt werden kann.

2.2.2. Clock-Management

Ein Ansatzpunkt für das Energiemanagement eines laufenden Prozessors ist das Clock-Management. Üblicherweise besitzt ein SoC-Prozessor ein Clock Controller Module (CCM), welches die Hauptfrequenz des Rechenkerns steuert und Taktsignale für andere Module auf dem SoC zur Verfügung stellt. Da eine CPU die meiste Zeit nicht vollständig ausgelastet ist, muss diese nicht zwangsläufig mit der höchsten Frequenz arbeiten. Die Haupt-Frequenz der CPU kann daher bei geringer Auslastung gedrosselt werden, ohne signifikante Leistungseinbußen zu erzeugen. Zusätzlich kann auch die Versorgungsspannung des Prozessorkerns verringert werden. Dies hat zwei Vorteile: Zum einen wird der Energieverbrauch gesenkt, zum anderen reduziert sich die Wärmeentwicklung des Prozessors. Dieses Verfahren wird als Dynamic Voltage and Frequency Scaling (DVFS) bezeichnet. Mehrkernprozessoren können bei geringer Auslastung auch einzelne Kerne im laufenden Betrieb komplett abschalten bzw. in einen Schlafmodus versetzen. In Kombination mit DVFS ergibt sich damit eine sehr energieeffiziente Taktung der Prozessorkerne. Eine weitere Energieersparnis kann erreicht werden, indem Taktsignale für Module deaktiviert werden, welche sich im Energiesparmodus oder Idle-Zustand befinden (Clock-Gating).

2.2.3. Peripherie-Management

Das Energiemanagement der Peripherie-Geräte wird durch deren jeweilige Treiber-Software gesteuert. Die Geräte, welche Energiemanagment-Unterstützung im Treiber besitzen, können durch das Betriebssystem ausgeschaltet oder in einen Energiesparmodus versetzt werden. In diesem Zuge wird meist auch das entsprechende Taktsignal für diese Geräte abschaltet (s.O. Clock-Gating). Auf den meisten eingebetteten Systemen arbeitet der Prozessor außerdem mit einem Power Management Integrated Circuit (PMIC) zusammen, welcher die Spannungsversorgungen für das System verwaltet. So können Teile der Peripherie komplett von der Spannungsversorgung getrennt und die Aktivität auf der Hauptplatine auf ein Minimum beschränkt werden.

3. Analyse

Dieses Kapitel beschreibt das Einsatzszenario für vorausschauendes Energiemanagement auf einem eingebetteten System, welches als Anwendungsfall für diese Arbeit ausgewählt wurde. Es werden die Anforderungen analysiert, die an die einzelnen Komponenten des Gesamtsystems gestellt werden.

3.1. Szenario: Autonome Messstation

Im Bereich der Landwirtschaft können die Wetterbedingungen großen Einfluss auf die Entstehung von Pflanzenkrankheiten an Nutzpflanzen haben. Als Beispiel soll die Pflanzenkrankheit *Septoria-Blattdürre* dienen, deren Verursacher zur Gattung der Pilze gehören und die mit Fungiziden bekämpft werden kann. In der Regel entstehen jedoch bei der Verwendung von Fungiziden Resistenzen, sodass es nicht zielführend ist präventiv Bekämpfungsmittel auf das Feld zu bringen. Umweltfreundlicher ist die Pflanzenkrankheit nur dann zu bekämpfen, wenn akute Entstehungsgefahr besteht. [2, S. 105, S. 482ff]

Mit Hilfe einer autonomen Messstation sollen daher günstige Entstehungsbedingungen für diese Pflanzenkrankheit auf dem Feld erkannt werden, sodass Fungizide nur noch dann eingesetzt werden, wenn dies zwingend erforderlich ist.

Das System soll anhand von Wetterdaten erkennen, ob eine akute Gefahr für die Entstehung von *Septoria*-Pilzen besteht. Um Flexibilität und Mobilität zu gewährleisten, soll die Messstation autonom arbeiten. Die Energieversorgung soll durch ein Solarmodul erfolgen, ein Akkumulator dient als Energiespeicher, damit das Gerät auch über Nacht mit Energie versorgt werden kann. Mit Hilfe von Temperatur-, Luftfeuchte- und Blattnässe-Sensoren, werden die Umweltbedingungen erfasst und von einem Computer-Modul ausgewertet. Das System kann über ein integriertes Global System for Mobile Communication (GSM)-Modem eine Kommunikation über das Mobilfunknetz aufbauen, um eine Warnung abzusetzen. Die Abbildung 3.1 zeigt ein vereinfachtes Blockschaltbild des zu entwickelnden Systems.

Zur Erkennung von kritischen Wetterbedingungen werden im Laufe eines Tages regelmäßig eine Mindestzahl an Messungen von Temperatur, Luftfeuchte und Blattnässe durchgeführt. Sobald die Station günstige Bedingungen für die Entstehung des Pilzes feststellt, wird über das GSM-Modem eine Warnung gesendet. Zusätzlich zu der Mindestzahl, soll das Gerät versuchen weitere Messungen durchzuführen, wenn der Ladezustand des Akkus und der zu erwartende Energieertrag es zulassen. Die Planung dieser zusätzlicher Aktivitäten

soll vorausschauend erfolgen, um zu verhindern, dass durch die erhöhte Aktivität der Akku zu stark entleert wird. Dazu sollen Wettervorhersagen verwendet werden, welche das Gerät regelmäßig über das GSM-Modem einholt.

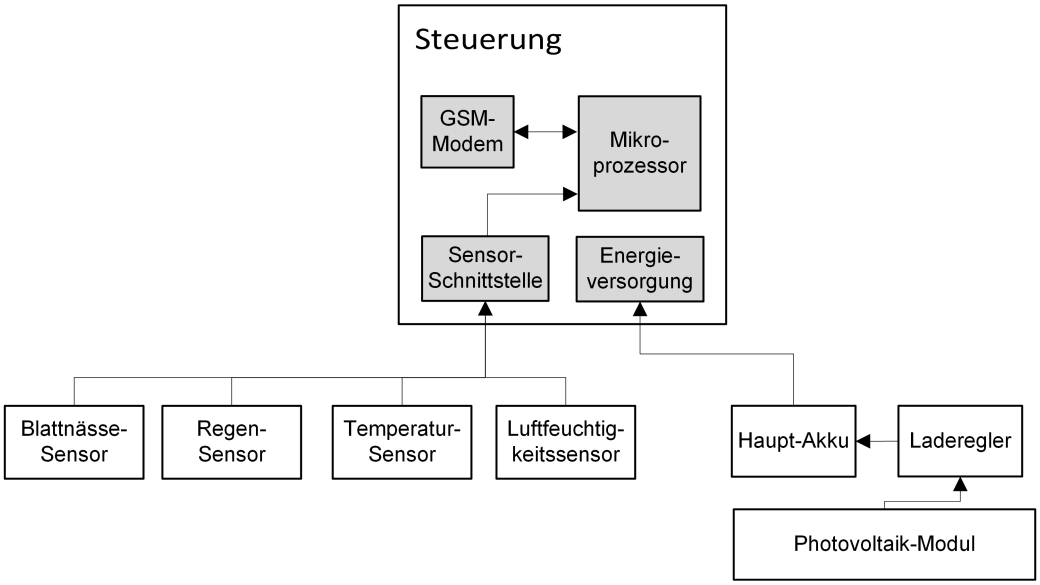

Abbildung 3.1.: Blockschaltbild autonome Messstation als Warnsystem für Pflanzenkrankheiten

3.2. Anforderungsanalyse

Nachfolgend werden die Anforderungen an die einzelnen Systemkomponenten analysiert und, wenn möglich, quantifiziert, die für die Entwicklung des zuvor beschriebenen Systems benötigt werden.

3.2.1. Anforderungen an die Hardware

Sensorik: Die Sensorik muss in der Lage sein, die für die Erkennung des Septoria-Pilzes erforderlichen Umweltbedingungen zu erfassen. Dazu zählt die Messung von Außentemperatur, Blattfeuchte, Niederschlag und Luftfeuchtigkeit. Die Sensoren müssen über eine kabelgebundene Schnittstelle verfügen, über welche die Messwerte an die Steuerung weitergeleitet werden können.

Energieversorgung: Die Energieversorgung soll aus einem Solarmodul, einem Laderegler und einem Akkumulator bestehen. Der Akku dient dabei als Zwischenspeicher und versorgt die Messstation in Zeiten, in denen keine Energie durch das Solarmodul bereitgestellt wird.

Liefert das Solarmodul genügend Energie, soll der Akku mit Hilfe eines Ladereglers aufgeladen werden. Die Kapazität des Akkus muss groß genug sein, um das gesamte System auch ohne Aufladung mindestens 3 Tage lang zu betreiben. Bei der Dimensionierung des Solarmoduls muss berücksichtigt werden, dass der Akku bei starker Sonneneinstrahlung innerhalb von 3 Tagen voll aufgeladen werden kann.

Steuerung: Als intelligente Steuerung für die autonome Messstation soll ein programmierbares Mikroprozessor- oder Computer-Modul verwendet werden. Hier müssen mindestens die Schnittstellen für serielle Kommunikation (*RS232*, *RS485*) vorhanden sein, an welche später die Sensoren oder Diagnose- und Auslesegeräte angeschlossen werden können. Zusätzlich werden auch analoge Eingänge (A/D-Wandler) benötigt, um Spannungen und Ströme innerhalb des Gesamtsystems zu überwachen, damit der aktuelle Ladezustand des Akkus beurteilt werden kann. Weiterhin muss das Modul ein GSM-Modem besitzen, über welches später die Messwerte übermittelt werden können. Zusätzlich zum Hauptakku sollte ein Backup-Akku für die Steuerung vorgesehen werden, falls durch einen Defekt die Hauptversorgung ausfällt. Damit die Steuerung Messungen zu exakten Zeitpunkten durchführen kann, muss diese eine auf eine Echtzeit-Uhr (Real Time Clock (RTC)) zurückgreifen können, die separat mit Energie versorgt wird. Die auszuwählende Steuerung sollte für den mobilen Einsatz geeignet sein und eine maximale Energieaufnahme von unter 3 W besitzen.

Die gesammelten Anforderungen an die Hardware werden in der Tabelle 3.1 zusammengefasst.

Teilsystem	**Anforderung**
Steuerung	• Schnittstellen: RS232, RS485, Analoge Ein-und Ausgänge • GSM-Modem • Überwachung von Akku- und Solarmodul-Spannung • Echtzeit-Uhr • Backup-Akku • Energieverbrauch maximal 3 W (Annahme)
Sensorik	• Sensor-Schnittstellen: RS232, RS485, Analog • Messung von Temperatur, Luftfeuchte, Blattnässe, Niederschlag
Energieversorgung	• Aufladung des Akkus innerhalb von 3 Tagen bei einer Sonnenleistung von 250 W/m^2 (Annahme) • Akku: Gewährleistung von mindestens 3 Tagen Betrieb ohne Aufladung (Annahme)

Tabelle 3.1.: Anforderungen an die Hardware des Systems

3.2.2. Anforderungen an die Software

Die Anforderungen an die Software werden in die drei Bereiche Betriebssystem, Energiemanagement-Software und Messsoftware unterteilt.

Betriebsystem: Es soll ein Linux-basiertes System verwendet werden, welches auf einer Yocto-Distribution aufbaut und mit einen Linux-Kernel der Version 3.10 arbeitet. Das Betriebssystem stellt die erforderlichen Schnittstellen für den Zugriff auf die Hardware bereit, sodass diese von der Anwender-Software verwendet werden können (Sensor-Schnittstellen, Echtzeituhr, GSM-Modem) Auch die Laufzeitumgebung für die Anwendungssoftware wird durch das Betriebssystem zur Verfügung gestellt. Das System muss mindestens ein Energiesparmodus unterstützen, welcher vom Benutzer angefahren werden kann. Weiterhin soll der Umfang des Betriebssystem so gering wie möglich gehalten werden.

Energiemanagement: Die Energiemanagement-Software reguliert den Energieverbrauch des eingebetteten Systems je nach Wetterbedingungen. Dazu muss die Software unterschiedliche Aktivitätslevel definieren, welche jeweils zu einem spezifischen Energieverbrauch führen. Es müssen Batterie- und Solarmodul-Spannung gemessen und ausgewertet werden. Die Software soll einen prädiktiven Algorithmus enthalten, der die Regulierung des Energieverbrauchs vorausschauend vornimmt. Dazu werden, neben den aktuellen gemessenen Werte, Wettervorhersage-Daten verwendet, um zukünftige Zustände von Batterie und Solarmodul vorauszuberechnen. Das Ziel des prädiktiven Energiemanagements soll sein, die maximal mögliche Anzahl an Messungen durchzuführen, wobei der Akkumulator zu keinem Zeitpunkt unter 25 % Restkapazität entladen wird.

Messsoftware: Die Messungen für die Erkennung von „Septoria“-Gefahr, werden durch die Messsoftware durchgeführt. Dazu müssen die Umweltbedingungen mit Hilfe der Sensorik erfasst und bewertet werden. Im Falle gefährlicher Bedingungen soll eine Warnung generiert und über das GSM-Modem weitergeleitet werden.

Die Tabelle 3.2 fasst die Anforderungen an die Software des Gesamtsystems zusammen.

Teilsystem	**Anforderung**
Betriebssystem	• Embedded Linux (Yocto) mit Linux-Kernel 3.10 • Bereitstellung der Schnittstellen für die Hardware (Treiber) • Bereitstellung einer Laufzeitumgebung für die Anwender-Software • mindestens ein Energiesparmodus verfügbar • Enthält nur notwendige Komponenten und Softwarepakete
Energiemanagment-Software	• Erfasst Akku-Ladezustand und Solarmodul-Zustand • Implementiert prädiktiven Energiemanagement-Algorithmus: - Restenergie im Akku immer über 25% - Maximal mögliche Messdichte
Mess-Software	• Messung und Bewertung der Wetterbedingungen • Generierung einer Warnung bei „Septoria“-Gefahr (über GSM)

Tabelle 3.2.: Anforderungen an die Software des Systems

4. Konzept

Die Konzepte für die Entwicklung des Systems werden im folgenden Kapitel erläutert. Es wird dazu das allgemeine Verhalten des Gesamtsystems beschrieben und der verwendete Energiemanagement-Ansatz vorgestellt, mit welcher der Energieverbrauch des Systems beeinflusst wird. Die Basis hierbei bilden die Energieflüsse zwischen Solarmodul, Akkumulator und eingebettetem System. Anschließend wird das Konzept für einen Algorithmus erläutert, der eine vorausschauende Steuerung des Energieverbrauchs realisiert.

4.1. Verhalten des Gesamtsystems

Wie bereits in Abschnitt 3.1 beschrieben, soll das System regelmäßige Aufgaben bzw. Messungen durchführen. Da sich die Wetterbedingungen nur langsam ändern, ist es nicht notwendig permanent Messungen durchzuführen. Es bietet sich daher an, die Station nach einer Messung in einen Energiesparmodus zu versetzen, bis die nächste Messung durchgeführt wird. Dadurch ergibt sich ein Wechsel zwischen aktiver Phase (Durchführung der Messung) und passiver Phase (Energiesparmodus). Je weiter nun die Aktivphasen auseinander liegen, desto länger befindet sich das Gerät durchschnittlich im Energiesparmodus und verbraucht weniger Energie. Demnach kann durch gezieltes Verschieben der aktiven Phasen der Energieverbrauch der Messstation reguliert werden.

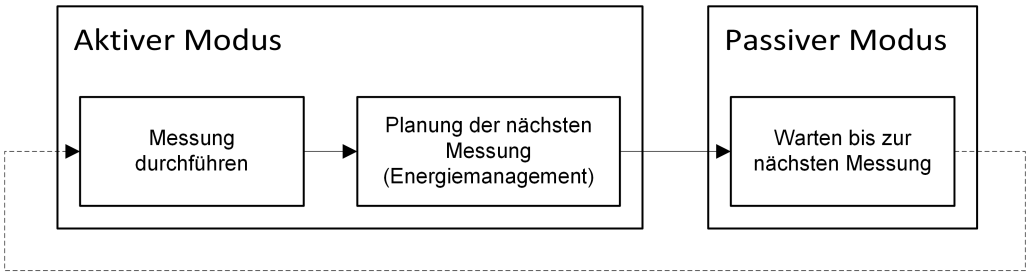

Abbildung 4.1.: Verhalten des Gesamtsystems: Wechsel zwischen Aktiv- und Passivphasen

4.2. Energiemanagement-Ansatz

Es wird ein Ansatz verwendet, der nur geringfügig von der verwendeten Hardware abhängig ist. Die verwendete Technik basiert auf der unterschiedlichen Stromaufnahme des Geräts

im aktiven und passiven Zustand (z. B. in einem Energiesparmodus). Da die meisten eingebetteten Systeme mindestens einen Energiesparmodus besitzen, kann diese Strategie auf vielen Geräten angewandt werden.

Die Energieaufnahme des eingebetteten System ist im passiven Zustand wesentlich geringer als im aktiven Zustand. Durch abwechselnde Aktiv- und Passivphasen stellt sich ein mittlerer Energieverbrauch ein. Wird das Verhältnis zwischen aktiver und passiver Zeit nun verändert, so ändert sich auch die mittlere Energieaufnahme des Geräts. Nachfolgend beschreibt ein Aktivitätslevel den Grad des Energieverbrauchs des Systems und ein Aktivitätszyklus die Dauer eines Zyklus aus Aktiv- und Passivphase.

Die folgende Abbildung 4.2 verdeutlicht den Aktivitätsgrad des Geräts über der Zeit. Dabei beschreibt der untere Graph (b) ein höheres Aktivitätslevel als der Graph (a), denn die Zeit im passiven Modus ist im Verhältnis zur Zeit im aktiven Modus wesentlich kürzer. Die Leistungsaufnahme kann auf diese Weise innerhalb eines bestimmten Bereichs nahezu stufenlos geregelt werden.

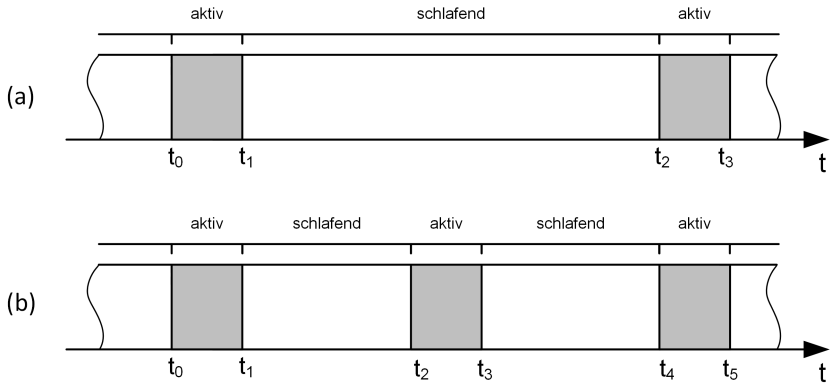

Abbildung 4.2.: Steuerung des Energieverbrauch durch abwechselnde Aktiv- und Passiphasen, mittlerer Energieverbrauch in (a) niedriger als in (b)

4.3. Energiefluss

Für die Entwicklung des Energiemanagements werden die Energieflüsse im Gesamtsystem untersucht. Dabei ist insbesondere die Berechnung von Energieverbrauch und Energiezufuhr von großer Bedeutung für die spätere Implementierung.

Das zu realisierende System soll autonom mit Sonnenenergie versorgt werden. Demnach stellt die Sonnenstrahlung die einzige Energiequelle dar, welche das System nutzen kann. Durch ein Solarmodul wird die Sonnenenergie in elektrische Energie umgewandelt und über einen Laderegler in den Zwischenspeicher (Akkumulator) geleitet. Das eingebette-

te System, welches die Messungen durchführt, stellt den Verbraucher des Systems dar und bezieht Energie aus dem Zwischenspeicher.

Die Abbildung 4.3 zeigt schematisch die Energieflüsse im Gesamtsystem.

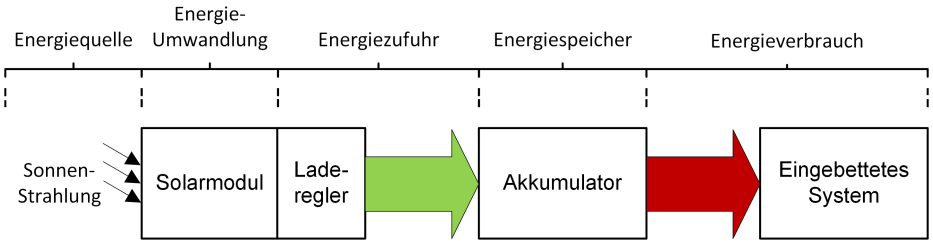

Abbildung 4.3.: Energiefluss im Gesamtsystem

4.3.1. Berechnung des Energieverbrauchs

Im Folgenden bilden eine Aktivphase und eine Passivphase gemeinsam einen Aktivitätszyklus mit der Zykluszeit t_{zyklus}. Dieser setzt sich aus der Zeit im Aktivmodus t_{aktiv} und der Zeit im Passivmodus t_{passiv} zusammen. Die Leistungsaufnahmen P_{aktiv} und P_{passiv} in den jeweiligen Zuständen werden als konstant angenommen.

Die Abbildungen 4.4 und 4.5 zeigen die Leistungsaufnahme des Geräts über der Zeit bei unterschiedlichen Aktivitätslevels.

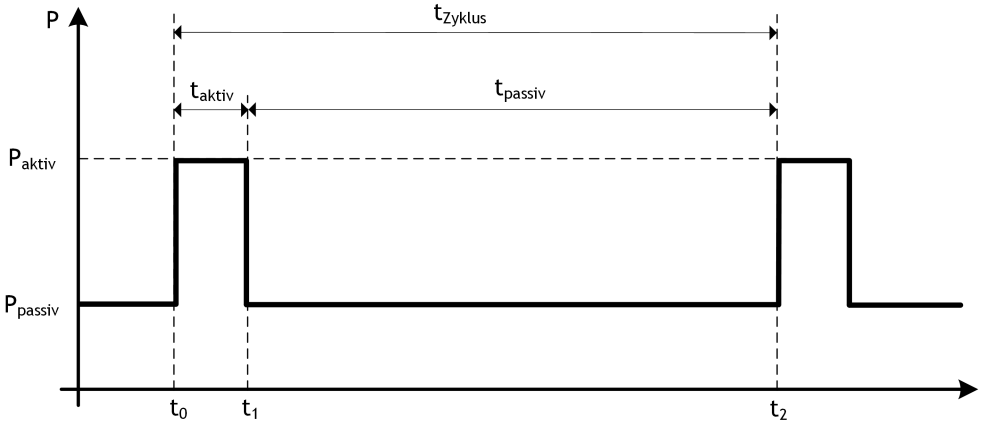

Abbildung 4.4.: Leistungsaufnahme über der Zeit bei einem niedrigen Aktivitätslevel

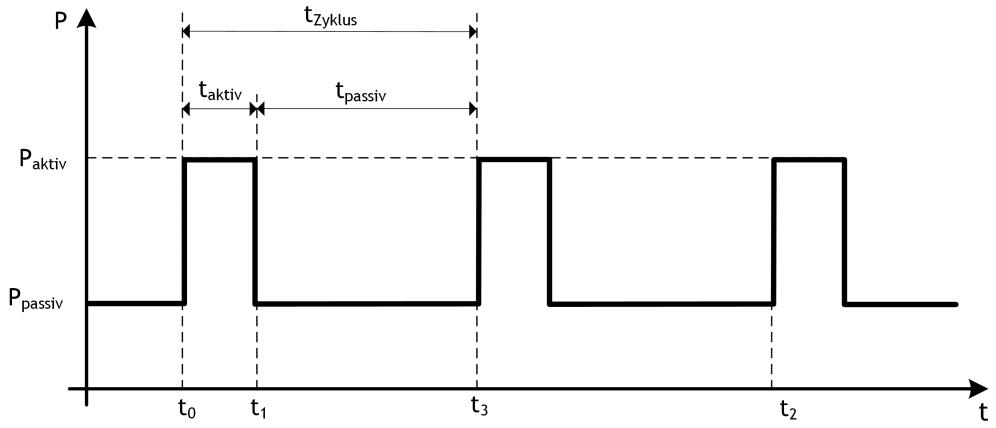

Abbildung 4.5.: Leistungsaufnahme über der Zeit bei einem hohen Aktivitätslevel

Die sich einstellende mittlere Leistungsaufnahme P_d für einen Aktivitätszyklus kann mit der Formel 4.1 berechnet werden:

$$P_d = P_{passiv} + (P_{aktiv} - P_{passiv})\left(\frac{t_{aktiv}}{t_{passiv} + t_{aktiv}}\right) \tag{4.1}$$

Aus der Formel 4.1 ist ersichtlich, dass das Verhältnis von Aktiv- und Passivzeit den Einfluss von P_{aktiv} auf die Durchschnittsleistung P_d reguliert. Dieser Zusammenhang ist linear, wie die Abbildung 4.6 zeigt.

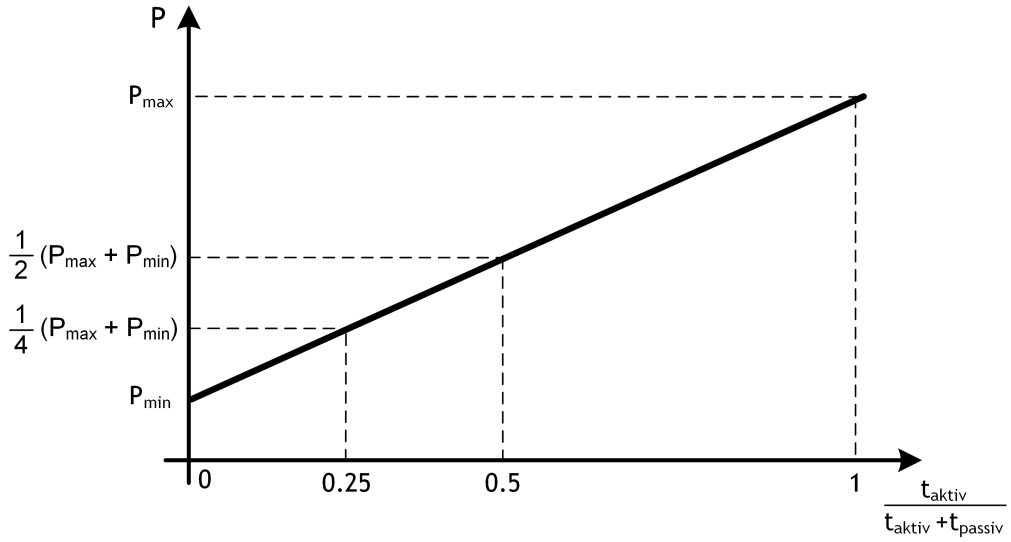

Abbildung 4.6.: Leistungsaufnahme über dem Verhältnis von Aktivzeit zur Zykluszeit

Da das Gerät in der aktiven Phase eine feste Folge von Aufgaben bearbeiten muss, wird die Aktivzeit für einen Zyklus als konstant angenommen. Die Zeit im Passivmodus bestimmt so allein den mittleren Energieverbrauch für einen Aktivitätszyklus. Die folgende Grenzwertbetrachtungen geben Aufschluss über die Randbedingungen, die für die Einstellung der Leistungsaufnahme L durch die Passivzeit beachtet werden müssen.

$$\begin{aligned} L &= \lim_{t_{passiv} \to \infty} P_{passiv} + (P_{aktiv} - P_{passiv}) \left(\frac{t_{aktiv}}{t_{passiv} + t_{aktiv}} \right) \\ &= P_{passiv} + (P_{aktiv} - P_{passiv}) \cdot 0 = \underline{\underline{P_{passiv}}} \end{aligned} \tag{4.2}$$

$$\begin{aligned} L &= \lim_{t_{passiv} \to t_{aktiv}} P_{passiv} + (P_{aktiv} - P_{passiv}) \left(\frac{t_{aktiv}}{t_{passiv} + t_{aktiv}} \right) \\ &= P_{passiv} + (P_{aktiv} - P_{passiv}) \cdot \frac{1}{2} = \underline{\underline{\frac{1}{2}(P_{aktiv} + P_{passiv})}} \end{aligned} \tag{4.3}$$

$$\begin{aligned} L &= \lim_{t_{passiv} \to 0} P_{passiv} + (P_{aktiv} - P_{passiv}) \left(\frac{t_{aktiv}}{t_{passiv} + t_{aktiv}} \right) \\ &= P_{passiv} + (P_{aktiv} - P_{passiv}) \cdot 1 = \underline{\underline{P_{aktiv}}} \end{aligned} \tag{4.4}$$

Wie die Gleichungen 4.2 - 4.4 zeigen, lässt sich die Leistungsaufnahme nur zwischen der maximalen Leistung P_{aktiv} und der minimal Leistung P_{passiv} einstellen. Ist die Passivzeit genauso groß wie die Aktivzeit, ist der Einfluss von P_{aktiv} genauso groß wie der Einfluss von P_{passiv} auf die mittlere Leistung.

Nachfolgend wird der funktionale Zusammenhang zwischen der Passivzeit und der sich einstellenden mittleren Leistungsaufnahme P_{Ziel} analyisert (Formel 4.5).

$$t_{passiv}(P_{Ziel}) = t_{aktiv} \left(\frac{P_{aktiv} - P_{passiv}}{P_{Ziel} - P_{passiv}} \right) - t_{aktiv} \tag{4.5}$$

Annahme: $t_{aktiv}, P_{aktiv}, P_{passiv} = const.$ und $P_{Ziel} := [P_{passiv}, P_{aktiv}]$

Der durch die Formel beschriebene Zusammenhang zwischen Passivzeit und mittlerer Leistungsaufnahme ist stark nichtlinear, wie in Abbildung 4.7 deutlich wird.

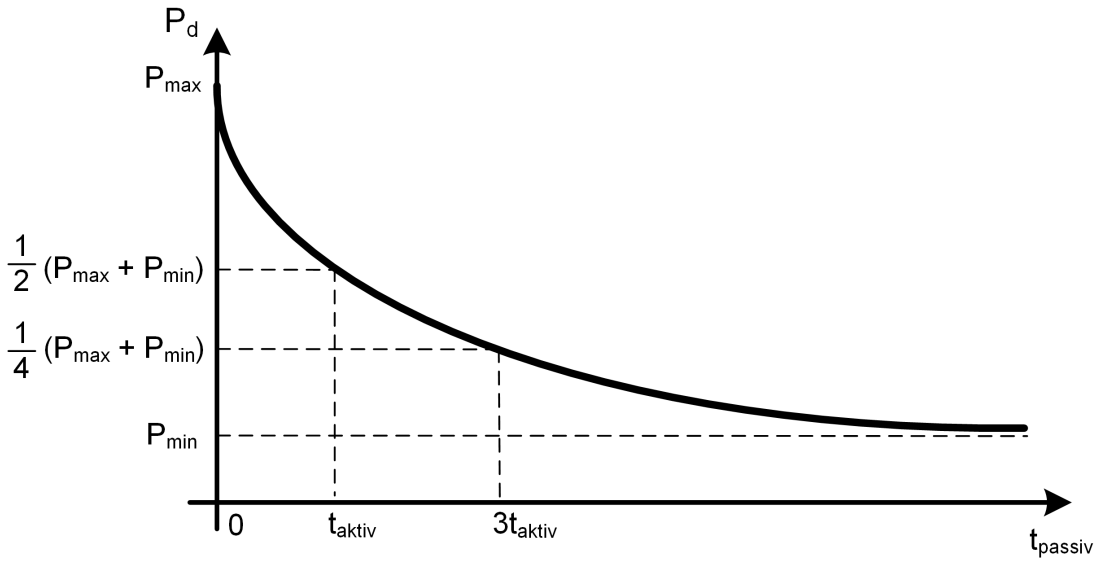

Abbildung 4.7.: Mittlere Leistungsaufnahme als Funktion der Passivzeit

Nachdem die mittlere Leistungsaufnahme für einen Aktivitätszyklus ermittelt wurde, kann nun die Energiemenge berechnet werden, die in diesem Zyklus verbraucht wird. Dazu wird eine Integration über der Zeit durchgeführt, mit der Anfangs- bzw. Endzeit des Zyklus als Grenzen.

Die Grafik 4.8 kennzeichnet die Fläche, welche durch die Gleichung 4.6 berechnet wird und die verbrauchte Energiemenge darstellt.

$$E_{Zyklus} = \int_{t_0}^{t_2} P_d \cdot dt \tag{4.6}$$

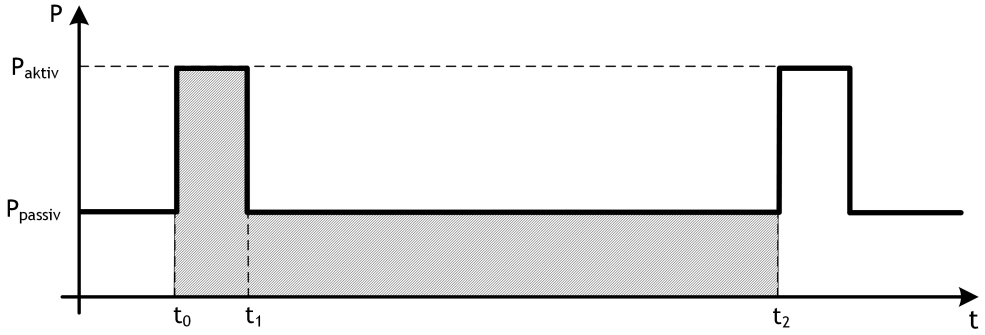

Abbildung 4.8.: Verbrauchte Energiemenge für einen Gesamtzyklus

4.3.2. Berechnung der Energieerträge

Ebenso wichtig wie die Betrachtung des Energieverbrauchs durch das Gerät, ist die Berechnung der Energiezufuhr durch das Solarmodul.

Die vom Solarmodul abgegebene Leistung hängt von der Intensität der Sonneneinstrahlung ab. Diese wiederum ist von der geografischen Position des Solarmoduls, dessen Himmelsausrichtung und der Jahreszeit abhängig.

Sind die technischen Daten des Solarmoduls bekannt, kann aus der Sonnenleistung die durch das Solarmodul abgegebe Leistung berechnet werden. Dazu werden die Daten über den Wirkungsgrad, sowie die wirksame Fläche des Solarmoduls benötigt. Mit der folgenden Formel 4.7 kann anschließend die vom Modul abgegebene Leistung berechnet werden [17, S. 32].

$$P_{Solar} = E_{Sonne} \cdot A_{Solar} \cdot \eta_{Solar} \tag{4.7}$$

Für die vorauschauende Steuerung des Energieverbrauchs, müssen Daten über zukünftige Energieerträge durch das Solarmodul berechnet werden. Dazu werden Wettervorhersagen über die zu erwartende Strahlungsleistung der Sonne verwendet. Die Vorhersage-Daten werden in gleichmäßigen Zeitintervallen angegeben, sodass Datenpaare aus Sonnenleistung und Uhrzeit vorliegen. Daraus lässt sich vereinfacht eine Stufenfunktion bilden, die den vereinfachten Verlauf der zu erwartenden Strahlungsenergie über der Zeit darstellt (s. Abbildung 4.9).

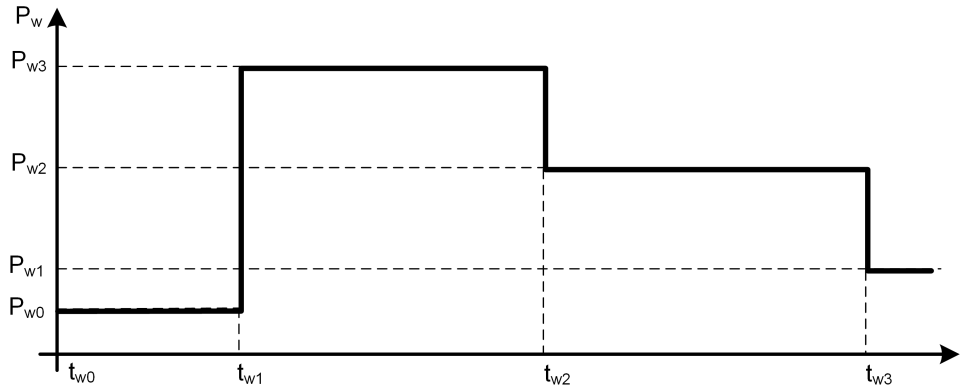

Abbildung 4.9.: Vorhersage-Daten über zukünftige Sonnenenergie als Stufenfunktion über der Zeit

Ähnlich der Berechnung der verbrauchten Energiemenge, kann auch die zu erwartende zu-

geführte Energiemenge durch Integration berechnet werden. Die Abbildung 4.10 zeigt die Fläche, welche die zu erwartende, durch das Solarmodul zugeführte, Energiemenge darstellt.

$$E_w = \int_{t_0}^{t_2} P_w \cdot dt \tag{4.8}$$

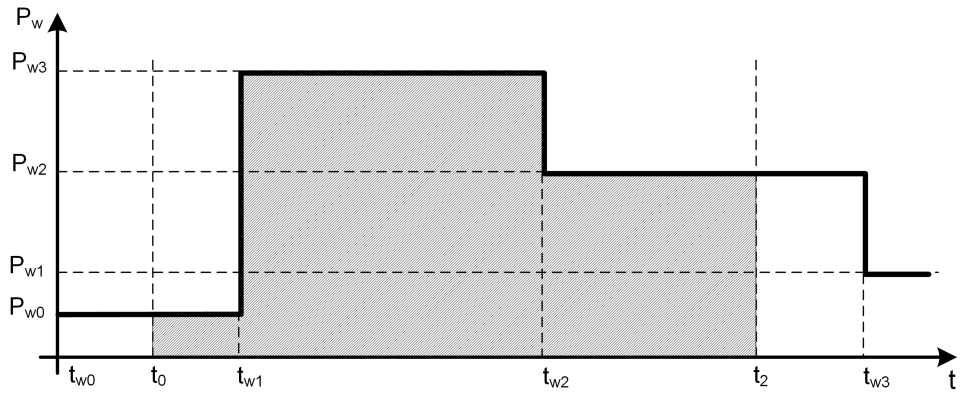

Abbildung 4.10.: Erwartete zugeführte Energiemenge für einen Gesamtzyklus

4.4. Vorausschauender Algorithmus

Um den Energieverbrauch des einbetteten Systems vorausschauend zu steuern, wird ein Algorithmus verwendet, der durch Vorhersage der Regelgröße auf das zukünftige Verhalten des Gesamtsystems schließt. Im zu entwickelnden System stellt der Akkuladezustand die Regelgröße dar. Das Ziel ist, das aktuelle Aktivitätslevel optimal zu wählen, sodass auch in Zukunft keine unerwünschten Entladezustände entstehen. Die Zeitspanne, für die zukünftige Werte vorausberechnet werden, wird als Prädiktionshorizont bezeichnet und beeinflusst die „Langfristigkeit“ der später durchgeführten Optimierung. Durch Analyse der vorausberechneten Werte können optimale Aktivitätslevel für den aktuellen Zustand und zukünftige Zustände berechnet werden.

Nachfolgend beschreibt ein *Systemzustand* die aktuellen Werte des Gesamtsystems (Systemzeit, Aktivitätslevel, Aktivitätszyklus (Zyklus aus Aktiv- und Passivphase), Energieverbrauch, Akkuladezustand)

4.4.1. Ablauf der Optimierung

Für die Optimierung des aktuellen Aktivitätslevels werden zunächst Akkuzustände für zukünftige Zeitpunkte, entsprechend dem Prädiktionshorizont, berechnet. Anschließend werden die berechneten Werte ausgewertet und geprüft, ob die vorausberechneten Zustände den Randbedingungen entsprechen. Entspricht ein Zustand nicht den Randbedingungen, so wird dieser optimiert und die Vorhersage wiederholt. Dies geschieht so lange, bis alle zukünftigen Zustände optimal sind oder eine weitere Optimierung nicht mehr möglich ist. In Abbildung 4.11 ist der Ablauf dargestellt, nach welchem das aktuelle Aktivitätslevel optimiert wird.

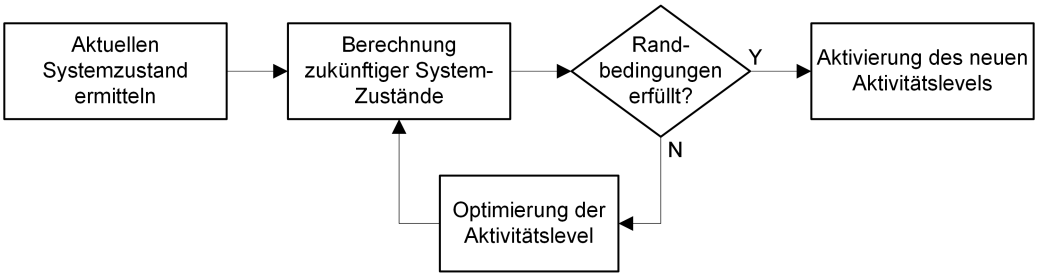

Abbildung 4.11.: Allgmeiner Ablauf der vorrauschauenden Optimierung des aktuellen Aktivitätslevels

4.4.2. Vorausberechnung zukünftiger Systemzustände

Da das System phasenweise aktiv ist, wird die Vorausberechnung des Akkuzustands nur für Zeitpunkte durchgeführt, zu denen das Gerät aktiv ist.

In der Abbildung 4.12 ist dargestellt, wie ausgehend vom aktuellen Zeitpunkt t_{S0} und dem aktuellen Akkuzustand S_0 ein zukünftiger Akkuzustand S_1 zum Zeitpunkt t_{S1} berechnet wird.

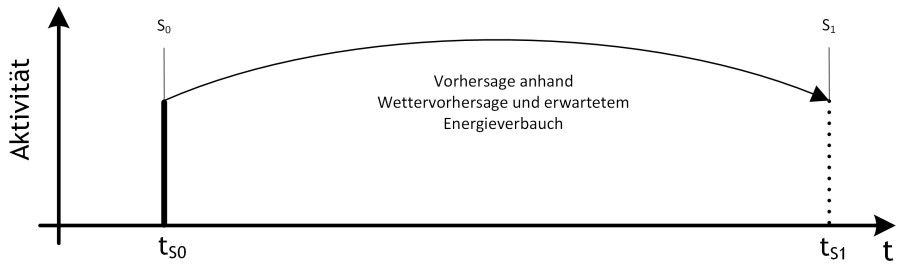

Abbildung 4.12.: Vorausberechnung eines Systemzustands in der Zukunft

Die Anzahl der zu berechnenden zukünftigen Werte hängt vom Prädiktionshorizont $t_{präd}$

und dem Aktivitätszyklus ab. Die Zeitspanne für einen Aktivitätszyklus ist im jeweiligen Aktivitätslevel definiert, wodurch die Anzahl an notwendigen Werten berechnet werden kann. Dabei muss berücksichtigt werden, dass sich die aktiven Zeitpunkte des Geräts verschieben, wenn das Aktivitätslevel gewechselt wird. Da die Vorausberechnungen nur für aktive Phasen vorgenommen werden, der Prädiktionshorizont jedoch auch in einer passiven Phase liegen kann, wird der nächste aktive Zustand berechnet, der nach dem Prädiktionshorizont folgt (s. Abb. 4.13).

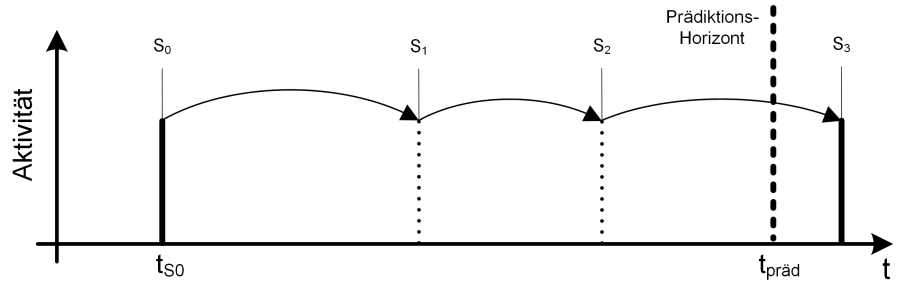

Abbildung 4.13.: Vorausberechnungen für Aktivzyklus-Zeitpunkte

Um zukünftige Ladezustände zu berechnen, muss die entnommene und zugeführte Energiemenge vorausberechnet werden. Dazu werden Modelle für die entsprechenden Komponenten des Systems gebildet, um deren Verhalten zu simulieren. Die Qualität der Berechnung ist dadurch an die Güte des verwendeten Modells gebunden. Anschließend wird mit Hilfe der Modelle das energetische Gesamtverhalten des Systems nachgebildet und ein daraus resultierender neuer Akkuladezustand zur nächsten Aktivphase berechnet.

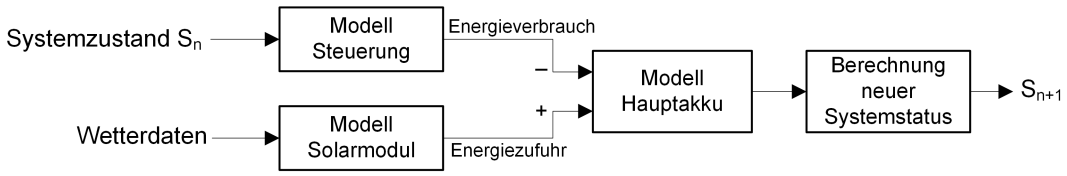

Abbildung 4.14.: Ablauf für die Vorausberechnung eines zukünftigen Systemzustands

Im Regelfall werden mehrere aufeinander folgende Akkuzustände berechnet. Für die erste Vorausberechnung dienen dabei die aktuellen Daten des Akkus als Grundlage. Die Berechnung des darauf folgenden Zustands hängt dann von den Ergebnissen der ersten Vorausberechnung ab. Die Berechnung eines Zustands hängt so immer von den Ergebnissen der vorherigen Berechnung ab. Das Aktivitätslevel für neue vorausberechnete Zustände wird dabei vom vorherigen Systemzustand übernommen.

Auf diese Weise können mehrere Zustände iterativ berechnet werden. In Abbildung 4.15 ist beispielhaft die Berechnung von 3 zukünftigen Systemzuständen, basierend auf dem aktuellen Zustand des Systems, dargestellt.

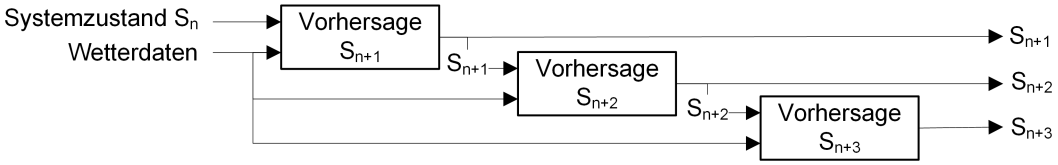

Abbildung 4.15.: Ablauf für die Berechnung von 3 zukünftigen Akkuzuständen

4.4.3. Optimierung des Aktivitätslevels

Mit Hilfe der vorausberechneten Werte soll nun vorausschauend das aktuelle Aktivitätslevel optimal eingestellt werden. Dazu ist zunächst die Definition von Randbedingungen nötig, welche einen Akkuzustand als optimal kennzeichnen. Zusätzlich muss berücksichtigt werden, dass im nächsten Aktivzyklus wieder eine Optimierung durchgeführt wird.

Als Randbedingung wird festgelegt, dass die Restkapazität im Akku 25% nicht unterschreiten darf. Dies ist eine Anforderung an den Algorithmus gemäß Abschnitt 3.2. Um einen Zustand zu optimieren, welcher nicht dieser Randbedingungen entspricht, muss die Energiemenge im Akku für diesen Zustand erhöht werden. Das wird erreicht, indem das Aktivitätslevel des *vorherigen* Systemzustands verringert wird. Dadurch wird der Energieverbrauch für den Zeitraum vor dem zu optimierenden Zustand verringert und es bleibt mehr Energie im Akku erhalten. Anschließend muss die Vorhersage für alle folgenden Zustände wiederholt werden, da sich die Berechnungsgrundlage verändert hat und sich auch die aktiven Zeitpunkte verschieben.

Ein besonderer Fall tritt ein, wenn das vorherige Aktivitätslevel nicht mehr optimiert werden kann, da es bereits auf den kleinstmöglichen Wert eingestellt ist. Um dennoch die verbleibende Restenergie im zu optimieren Zustand zu erhöhen, wird nun das Aktivitätslevel des *vor-vorherigen* Systemzustands verringert, auch wenn in diesem Zustand mehr als 25% Restenergie im Akku vorhanden ist. Dadurch ist im vorherigen Zustand bereits mehr Restenergie im Akku und so auch im zu optimierenden Zustand. Die Optimierung der einzelnen Aktivitätslevel wird so rekursiv durchgeführt, bis der aktuelle Zustand erreicht wurde oder alle berechneten Akkuzustände optimal sind. Durch diese Vorgehensweise erfolgt die Optimierung vorausschauend.

In der Abbildung 4.16 ist die rekursive Optimierung als Ablaufdiagramm dargestellt.

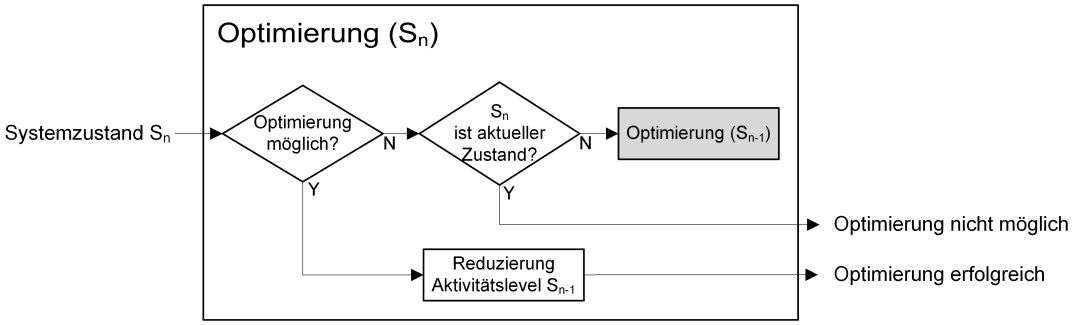

Abbildung 4.16.: Rekursive Optimierung des Aktivitätslevels des Zustands Sn

Das aktuelle Aktivitätslevel wird erst dann verändert, wenn alle vorausberechneten Zustände vollständig optimiert wurden und das minimale Aktivitätslevel verwenden. Treten dennoch unzulässige Akkuladezustände, ist es erforderlich das aktuelle Aktivitätslevel zu verringern, um die Restenergie für alle zukünftigen Zustände zu erhöhen.

In Abbildung 4.17 ist der Akkuladezustand und das Aktivitätslevel über der Zeit für einen einfachen Entladevorgang ohne Aufladung dargestellt. Die Abbildung beinhaltet 3 Graphen, welche 3 Phasen in der Optimierung der Aktivitätslevel darstellen. Graph (a) zeigt die unoptimierten vorhergesagten Zustände. Es folgt eine Zwischenphase (b), in welcher bereits einige zukünftige Systemzustände optimiert wurden. Der letzte Graph (c) stellt die abgeschlossene Optimierung des Aktivitätslevels dar.

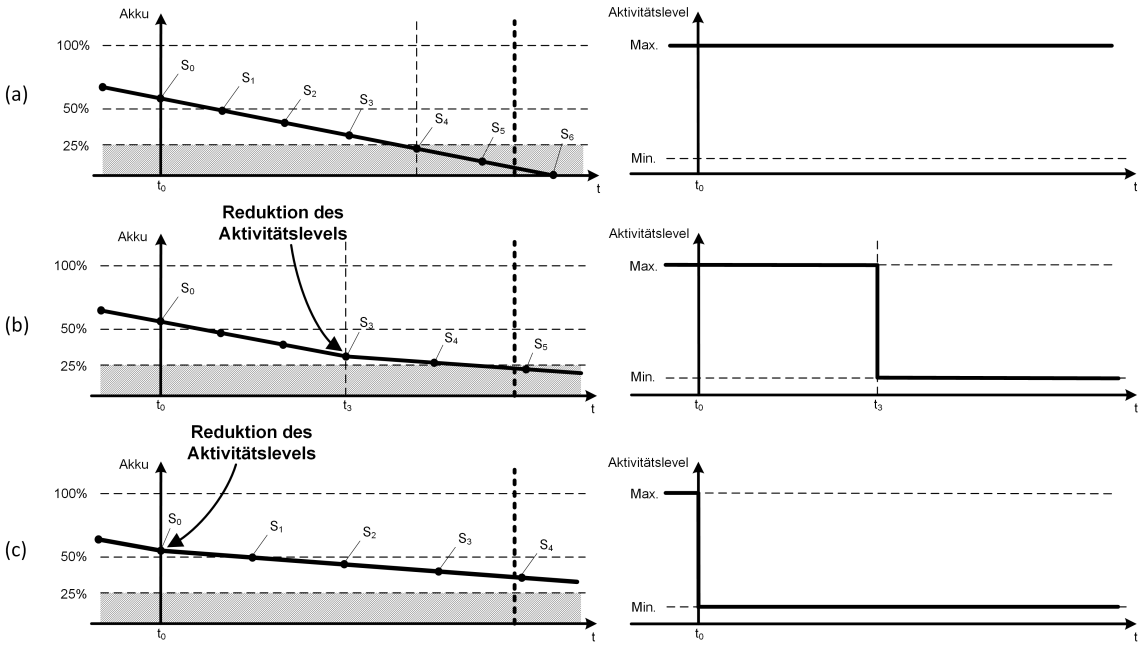

Abbildung 4.17.: Schematische Darstellung der rekursiven Optimierung des Aktivitätslevels

Nach der ersten Vorausberechnung haben alle zukünftigen Zustände das maximale Aktivitätslevel und damit den maximalen Energieverbrauch. Es ist erkennbar, dass der Zustand S_4 und alle darauf folgenden Zustände nicht der Randbedingung (Restenergie > 25%) entsprechen. Es muss demnach eine Optimierung vorgenommen werden. Um Zustand S_4 zu optimieren, wird nun das Aktivitätslevel von Zustand S_3 verringert, um im Zustand S_4 mehr Restenergie im Akku zu erhalten (2. Graph in Abbildung 4.17). Anschließend werden die berechneten Werte für S_4 und alle folgenden Zustände verworfen und die Vorausberechnung wiederholt. Auf diese Weise wird das geänderte Aktivitätslevel von Zustand S_3 berücksichtigt. In der neuen Vorausberechnung werden die auf S_3 folgenden Zustände das gleiche Aktivitätslevel wie S_3 annehmen.

Nachdem das Aktivitätslevel von Zustand S_3 so weit optimiert wurde, dass der minimale Wert erreicht ist, liegt der Zustand S_4 nun oberhalb der 25%-Grenze. Doch S_5 befindet sich immer noch unter der 25%-Grenze. Gleichzeitig haben alle 3 Zustände bereits das minimale Aktivitätslevel erreicht.

Um dennoch die Restenergie für Zustand S_5 zu erhöhen wird das Aktivitätslevel von Zuständen vor S_3 reduziert. So wird indirekt die Restenergie S_4 und S_5 erhöht. Auf diese Weise werden nun rekursiv alle vorherigen Zustände optimiert, bis der aktuelle Systemzustand erreicht und optimiert wurde oder alle berechneten Zustände der Randbedingung genügen. Der dritte Graph der Abbildung 4.17 zeigt die abgeschlossene Optimierung. Das aktuelle Aktivitätslevel wurde verringert, damit für alle zukünftigen Zustände die Randbedingung erfüllt ist.

5. Realisierung

Das folgende Kapitel beschreibt die Implementierung der zuvor erläuterten Konzepte. Dazu werden zunächst die, für die Realisierung verwendeten, Komponenten vorgestellt. Anschließend wird die Inbetriebnahme und Anpassung eines Linux-basieren Betriebssystems auf dem verwendeten eingebetteten System dokumentiert. Abschließend wird die Implementierung einer Anwendungssoftware beschrieben, welche den entwickelten Algorithmus prototypisch umsetzt.

5.1. Übersicht der verwendeten Komponenten

5.1.1. Eingebettetes System

Für die Steuerung steht das Computer-Modul „LIVIUS“ der Firma Garz & Fricke GmbH zur Verfügung [9]. Auf diesem Gerät sind Hardware-seitig mehrere Energiemanagement-Mechanismen integriert, wodurch es sich als intelligente Steuerung für eine autonome Messstation sehr gut eignet. Das Gerät ist in einem Hutschienen-Gehäuse mit der Schutzart IP20 untergebracht.

Abbildung 5.1.: Computer-Modul Garz & Fricke „LIVIUS“ [10]

Hardware-Spezifikation

Der Prozessor des Geräts ist ein Freescale i.MX35 SoC (ARM-Architektur), welcher mit einer Frequenz von 532 MHz arbeitet. Auf der Platine sind 128 MB DDR-RAM und 256 MB Flash-Speicher untergebracht. Es sind die Schnittstellen RS233, RS485, CAN sowie analoge und digitale Ein- und Ausgänge vorhanden. Die Energieversorgung des Geräts kann variabel mit 9 - 30 VDC erfolgen. Sollte die Spannungsversorgung ausfallen, übernimmt ein interner Lithium-Ionen-Akku die kurzfristige Energieversorgung. Weiterhin ist eine Echtzeituhr mit Alarm-Funktion vorhanden, die durch eine separate Lithium-Batterie versorgt wird, sodass auch bei ausgeschaltetem Gerät das Datum und die Zeit aktuell bleiben. Über ein integriertes GSM-Modem kann das System über das Mobilfunknetz kommunizieren.

Die wichtigsten technischen Daten des Computer-Moduls sind in Tabelle 5.1 im Überblick dargestellt:

CPU	
Typ	Freescale ARM1136JF-S™ i.MX35
Frequenz	523 MHz
Speicher	
RAM	128 MB DDR-RAM
Flash	256 MB NAND-Flash
Erweiterung	MicroSD(HC) Slot
Schnittstellen	
Netzwerk	10/100 MBit/s Ethernet
Mobilfunk	GE 865-QUAD GSM/GPRS Modem
Serielle Schnittstellen	RS-232 RS-485 (half/full duplex) CAN
Digital I/O	4x Digital Eingang 4x Digital Ausgang
Analog I/O	4x Analog Eingang (0-10 V, max. 20 mA) 2x Analog Ausgang (0-10 V)
Energieversorgung	
Versorgungsspannung	9 – 30 VDC
Leistungsaufnahme	typ. 140 mA @ 12 V DC (Angabe aus Datenblatt)
Besonderheiten	
Backup-Akku	Lithim-Ionen Akku 3.7 V / 1000 mAh
Echtzeituhr	NXP PCF8563 (Versorgung mit Lithium-Batterie)

Tabelle 5.1.: Technische Daten: Garz & Fricke LIVIUS [10]

Energiemanagement

In LIVIUS sind mehrere Hardware-seitig unterstützte Energiemanagement-Funktionen realisiert. Dazu zählen die Überwachung wichtiger Spannungen und Ströme, ein Tiefschlafmodus und entsprechende Mechanismen zum Aufwecken des Geräts.

Als Basisfunktionen kann das Gerät die interne Spannungsversorgung über den Backup-Akku deaktivieren, wodurch auch das GSM-Modem abgeschaltet wird. LIVIUS hat einen integrierten Laderegler für die Aufladung des Backup-Akkus, welcher den Ladevorgang bei Erreichen einer Akku-Spannung von 4.0 V automatisch startet. Dieses Verhalten kann durch das Betriebssystem unterbunden werden.

Über einen internen Analog/Digital-Wandler kann das System die folgenden Spannungen selbstständig überwachen: externe Spannungsversorgung, Strom der Versorgungsspannung, Backup-Akku-Spannung und Echtzeituhr-Batterie-Spannung.

Das System unterstützt einen „Off-Mode“ genannten Tiefschlafmodus. Dabei kann das Betriebssystem das Gerät sofort komplett abschalten. Die Stromaufnahme sinkt dabei unter 100 µA. Ein Aufwecken des Systems ist dann nur noch über einen Echtzeituhr-Interrupt, einen digitalen Eingang oder das Einstecken einer externen Energieversorgung möglich. Sobald das Gerät geweckt wird, durchläuft es einen normalen Systemstart.

Betriebssystem

Auf dem Gerät arbeitet aktuell ein Linux-System, basierend auf einer OSELAS®-Distribution [19], welches mit dem Linux-Kernel 2.6.33 arbeitet. Der Kernel beinhaltet mehrere Anpassungen, welche durch die Firma vorgenommen wurden, um die Hardware vollständig zu unterstützen. Als Bootloader dient eine, ebenfalls modifizierte Version von eCos „Redboot“ [6].

Im Zuge dieser Arbeit wird ein Linux-System auf dem Gerät in Betrieb genommen, da das vorhandene System veraltet ist und nicht alle Eigenschaften des Gerätes unterstützt.

5.1.2. Energieversorgung

Für die Energieversorgung der Messstation steht ein Dünnschicht-Solarmodul der Firma Koekraf GmbH zur Verfügung. Als Zwischenspeicher dient ein Panasonic LC-R127R2PG Blei-Akkumulator. Zwischen Solarzelle und Akku wird der Solar-Laderegler H-Tronic SL 53 geschaltet. Der Laderegler ist für die Aufladung von 12V Akku konzipiert und reguliert die Ladespannung. Dies ist notwendig, damit der Akku nicht „überladen“ wird.

Die wichtigsten technischen Daten der Energieversorgung im Überblick:

Solarmodul	
Produkt	Koekraf 6 W Solarmodul 12 V
Bauform	Dünnschicht (amorph)
Leistung	6 Wp
Fläche	ca. 0.146 m^2
Laderegler	
Produkt	H-Tronic SL 53
Betriebsspannung	12 - 15 VDC
Eigenverbrauch	< 0.15 W
Ladestrom	max. 4 A
Soloarmodul-Leistung	max. 53 W
Akku	
Produkt	Panasonic LC-R127R2PG
Technologie	Valve-Regulated Lead-Acid (VRLA)
Nennspannung	12 V (6 Zellen)
Kapazität	7.2 Ah

Tabelle 5.2.: Technische Daten der Komponenten der Energieversorgung [12][14][18]

5.1.3. Sensorik

Vorgesehen sind Sensoren zur Messung von Temperatur, Luftfeuchte, Blattnässe und Niederschlag. Für die Implementierung des Energiemanagements ist es jedoch nicht erforderlich echte Sensoren für eine Messung abzufragen. Die Durchführung der Messung wird daher in der Software durch einen, mit Zeitstempel versehenen, Eintrag in einer Log-Datei „simuliert“. Soll das System später mit echten Sensoren getestet werden, kann der Log-Eintrag durch den Aufruf einer Messfunktion ersetzt werden.

5.1.4. Gesamtsystem

Die vorhandenen Komponenten werden zu einem Gesamtsystem kombiniert, um eine autonome Messstation zu realisieren. Nachfolgend ist das detaillierte Blockschaltbild des Gesamtsystems mit allen verwendeten Komponenten und deren wichtigste Subkomponenten dargestellt (Abb. 5.2).

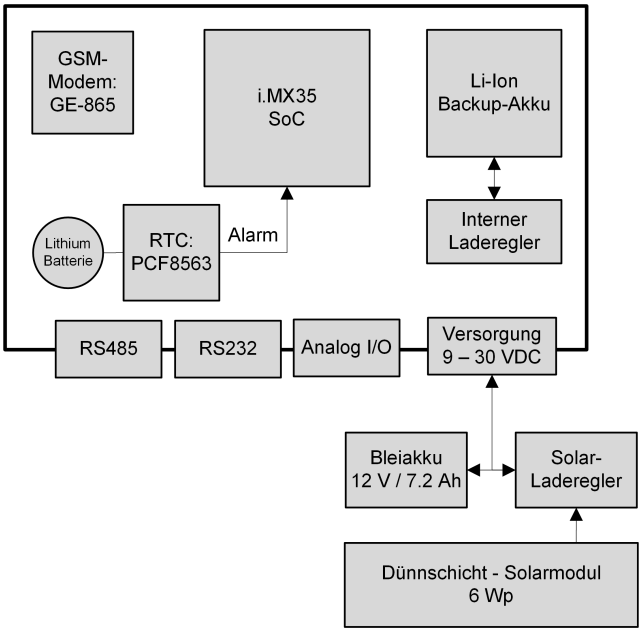

Abbildung 5.2.: Blockschaltbild des verwendeten Gesamtsystems

5.2. Betriebsystem

Auf dem eingebetteten Gerät wird ein Linux-basiertes Betriebssystem mit Hilfe des Yocto-Systems [16] generiert und in Betrieb genommen. Der verwendete Linux-Kernel hat die Version 3.10 und wird leicht angepasst. Dazu zählt das Hinzufügen der Hardware-Unterstützung für das Gerät, sowie Anpassungen und Erweiterungen an einzelnen Geräte-Treibern.

5.2.1. Yocto Embedded Linux

Das Yocto-Projekt ist ein *Open-Source*-System, dass Werkzeuge zur Erstellung von Linux-basierten Betriebssystemen zur Verfügung stellt. Es kommt die Version 1.5.1 (Yocto Dora) zum Einsatz.

Das Yocto-System stellt eine Vielzahl an Paketen zur Verfügung, aus denen das spätere Betriebssystem besteht. Für der Erstellung eines Betriebssystems können die Pakete beliebig ausgewählt werden, sodass nur die Features auf dem späteren System zur Verfügung stehen, die gewünscht sind. Bei einem Linux-System stellt der Linux-Kernel das wichtigste Paket des Systems dar, die anderen Pakete sind z. B. Bibliotheken, Programme und Laufzeitumgebungen. Die gewünschten Pakete werden in eine Konfigurationsdatei (*Image-Configuration*) eingetragen. Zusätzlich wird auch eine *Machine-Configuration* ange-

legt, welche Informationen über die Zielhardware, wie z. B. Prozessor-Architektur und -Typ enthält. Das Yocto-System benötigt diese Informationen, da es einen für die Ziel-Hardware geeigneten *Compiler* verwenden muss, um die Pakete zu übersetzen.

Anschließend wird das Betriebssystem mit dem Yocto-System vollständig generiert. Dabei werden alle Pakete (und von diesen abhängige Pakete) übersetzt, die in der Konfiguration ausgewählt wurden. Das Ergebnis stellt ein *Kernel-Image* und Dateisystem dar, welche auf den Flash-Speicher des Gerätes übertragen werden.

Durch ein im Bootloader hinterlegtes *Bootscript* werden dem Kernel entsprechende Parameter übergeben, um in das neu erstellte Yocto-Linux zu booten.

5.2.2. Anpassung des Linux-Kernels

Für die Inbetriebnahme müssen verschiedenen Anpassungen am Kernel vorgenommen werden, damit dieser mit der verwendeten Hardware funktioniert. Als Grundlage dient ein Freescale-Linux-Kernel mit der Version *3.10.17* [7]. Dabei handelt es sich um einen *Mainline*-Linux-Kernel, der Erweiterungen enthält, um die CPUs und Controller der Freescale i.MX-Prozessoren zu unterstützen.

Anpassungen an Garz & Fricke Bootloader

Die Firma Garz & Fricke verwendet für ihre Gerät einen eigenen Bootloader, der auf Redboot basiert. Zunächst muss der Kernel angepasst werden, um den Kernel mit diesem Bootloader verwenden zu können. Diese Anpassungen wurden im Rahmen dieser Arbeit durchgeführt, werden jedoch nicht weiter beschrieben, da sie für die Implementierung des Energiemanagements nicht von Bedeutung sind.

Hardware-Unterstützung

Zunächst muss die Hardware-Plattform in die Liste der Geräte eingetragen werden, die dem Kernel bekannt sind. Dazu wird ein neuer *Machine-Type* hinzugefügt, mit welchem der Kernel später erkennen kann, welche Hardware vorliegt. Mit diesem Machine-Type identifiziert der Kernel die vorliegende Hardware und wählt die entsprechend hinterlegte Konfiguration aus, um die Hardware zu initialisieren. Es muss daher auch eine neue Hardware-Konfiguration für das Geräts angelegt werden. Hierfür wird eine *Board-Configuration* verwendet. Die *Device-Tree*-Architektur wird nicht verwendet, da dies von der Firma Garz & Fricke nicht erwünscht ist. Die Board-Konfiguration enthält sämtliche Konfigurationen für die einzelnen Hardware-Geräte, welche auf der Plattform vorhanden sind.

Es wurden anschließend sämtliche verfügbaren Schnittstellen getestet, um die korrekte Initialisierung der Hardware und einen stabilen Betrieb des Gerätes zu verifizieren.

Treiberanpassungen

Neben kleineren Fehlerbehebungen und Erweiterungen an Geräte-Treibern, wurden auch mehrere Treiber neu hinzugefügt.

Serieller Treiber: Der vorhandene serielle Geräte-Treiber wurde erweitert, sodass das RS485-Bussystem im Halb- und Vollduplexmodus unterstützt wird. Dazu muss auf Treiber-Ebene eine zusätzliches Signal eingeführt werden, welches im Halbduplexmodus verwendet wird, um zwischen Sende- und Empfangsmodus der RS485-Schnittstelle umzuschalten. Beim verwendeten Gerät teilen sich die RS485- und die RS232- Schnittstelle einen Universal Asynchronous Receiver/Transmitter (UART) des Prozessors. Der jeweilige Betriebsmodus kann über einen General Purpose Input Output (GPIO) umgeschaltet werden. Dieses Signal muss ebenfalls in den Treiber integriert werden, damit der Betriebsmodus später durch die Software automatisch ausgewählt wird.

RTC: Die RTC ist über den Inter-Integrated Circuit (I2C)-Bus an den Prozessor angeschlossen. Die Uhr besitzt, neben der Bereitstellung der Uhrzeit und des Datums, auch einen programmierbaren Alarm. Beim Erreichen der Alarmzeit wird über einen Pin der RTC ein Signal erzeugt, welches den Prozessor „aufwecken“ kann. Der im Kernel vorhandene Treiber für die RTC ist in der Lage die Uhrzeit und das Datum der Systemuhr zu lesen und zu schreiben, unterstützt die Alarm-Funktion jedoch nicht. Da für die Realisierung des Gesamtsystems die Alarm-Funktion zum gesteuerten Aufwecken des Systems unbedingt erforderlich ist, wird der Treiber erweitert, sodass auch das Setzen und Lesen eines Alarms möglich ist. Die Alarm-Zeit der RTC kann Hardware-seitig auf Minuten genau eingestellt werden.

A/D- und D/A-Wandler: Im verwendeten Kernel sind keine Treiber für die auf LIVIUS vorhandenen A/D- bzw. D/A-Wandler vorhanden. Es existieren aber in der Firma bereits Treiber für die entsprechenden Geräte, welche das Industrial Input Output (IIO)-Subsystem verwenden. Diese Treiber wurden in den Kernel integriert, wobei der Treiber für den A/D-Wandler minimal angepasst werden musste, da dieser für einen etwas anderen A/D-Wandler des Herstellers programmiert wurde. Der Geräte-Treiber für den D/A-Wandler konnte ohne Anpassungen übernommen werden.

CPUidle: Linux bietet ein Subsystem (*CPUidle*), welches das Verhalten des Prozessors im *Idle*-Zustand steuert. Der verwendete Kernel besitzt jedoch keinen Treiber für den verwendeten IMX35-Prozessor. Damit der Prozessor im Idle nicht mit Höchstleistung läuft, wird ein einfacher CPUidle-Treiber für den IMX35 ergänzt. Dieser bringt den Prozessor im Idle-Zustand in den *WAIT*-Mode, einen leichten Energiesparmodus des Prozessors.

Die Änderungen und Anpassungen am Kernel werden in der folgenden Tabelle 5.3 kurz zusammengefasst.

Komponente	**Anpassung**
Allgemein	• Anpassung an Garz & Fricke Redboot Bootloader
Hardware-Unterstützung: GF LIVIUS	• Hinzufügung eines neuen Machine-Types • Board-Konfiguration für GF LIVIUS
Serieller Treiber	• Unterstützung von RS485 • Wechsel zwischen RS232 und RS485 Modus
RTC-Treiber	• Unterstützung zum Schreiben und Lesen des Alarms
ADC- und DAC-Treiber	• Hinzufügen eines IIO-Treibers für ADC und DAC
CPUidle	• Unterstützung von CPUidle für IMX35 Prozessoren

Tabelle 5.3.: Übersicht der Kernelanpassungen

5.3. Implementierung der Anwendungssoftware

Für die prototypische Implementierung des Algorithmus in einem Programm wird die Programmiersprache „C“ verwendet. Die implementierte Software besteht aus zwei Hauptbestandteilen. Der erste Teil beinhaltet Hilfsfunktionen für die Hardwarezugriffe, während der zweite Teil die eigentliche Umsetzung des Energiemanagements darstellt.

Der vollständige Quellcode der Anwendungssoftware befindet sich im Anhang A.2 und auf der CD-ROM (A.1). Die Applikation besteht aus den folgenden Dateien:

- pman.c
- pman.h
- sysfs_utils.c
- sysfs_utils.h
- weather.h

Die Datei *pman.c* enthält den Hauptteil der Applikation, die prototypische Implementierung des Algorithmus. Dort befindet sich die *main*-Funktion des Programms. In *sysfs_utils.c* sind Hilfsfunktionen zur Interaktion mit dem Linux-Sysfs und damit mit der Hardware des Geräts implementiert. In der Headerdatei *weather.h* sind unterschiedliche Arrays mit virtuellen Wetterdaten enthalten. Diese werden für die Prädiktion von zukünftigen Ladezuständen vom Hauptprogramm verwendet.

5.3.1. Entwicklungsumgebung

Als Entwicklungsumgebung für die Software wird ebenfalls das Yocto-System verwendet. Es wird ein neues Yocto-Paket angelegt, welches das neue Energiemanagement-Programm enthält. Das Paket wird außerdem zur *Image-Configuration* für das Gerät hinzugefügt, sodass es standardmäßig im Betriebssystem verfügbar ist.

Das Yocto-System stellt auch hier einen entsprechenden Kompiler zur Verfügung, welcher für die Zielhardware ausführbaren Maschinen-Code erzeugt. Ein zusätzlicher Vorteil bei der Verwendung des Yocto-Systems ist, dass die implementierte Software auch für andere Geräte mit anderer Hardware übersetzt werden kann. Dazu muss nur eine andere *Machine-Configuration* ausgewählt werden. Das Yocto-System wird dann automatisch die korrekten Einstellungen vornehmen.

5.3.2. Hardware-Zugriff

Für den Zugriff auf die Hardware des eingebetteten System wird das *Sysfs*, ein virtuelles Dateisystem unter Linux, verwendet. Es stellt Informationen der Hardware-Treiber und-Geräte in Form eines Dateisystems zur Verfügung. Treiber-Subsysteme werden als Ordner dargestellt und einzelne Eigenschaften oder Funktion von Geräten als Dateien. So wird durch einfache Dateimanipulation eine Interaktion mit der Hardware realisiert. Zusätzlich kann das *Sysfs* auch zur Konfiguration von Geräten und deren Treibern verwendet werden.

In der Datei *sysfs_utils.c* sind Funktionen implementiert, welche über das *Sysfs* entsprechende Dateien lesen oder beschreiben und auf diese Weise mit der Hardware des Systems interagieren. Als Grundlage für die Umsetzung dienen Funktionen, welche in der Dokumentation des IIO-Subsystems des verwendeten Linux-Kernels zu finden sind.

In der folgenden Übersicht sind die implementierten Funktionen aufgelistet, die mit der Hardware kommunizieren:

- Schreiben und Lesen von Zuständen der GPIOs
- Schreiben der Alarmzeit der RTC
- Lesen der Spannung eines Analog-Digital-Converter (ADC)-Kanals

Auf die RTC wird über das RTC-Subsystem von Linux zugegriffen. So können Zeit und Datum, aber auch Alarm und Konfiguration gelesen oder gesetzt werden. Es muss beachtet werden, dass die RTC die Angaben zu Zeit und Datum nur im UNIX-Format[1] liefert und verarbeitet.

[1] Das UNIX-Zeit-Format ist eine positive Zahl, die die vergangenen Sekunden seit dem 1. Januar 1970 00:00 Uhr zählt.

Der ADC ist über das IIO-Subsystem in das System integriert. Es können die Rohdaten des ADC direkt ausgelesen werden, dazu zählt das Ergebnis der Wandlung als 10-Bit-Wert und die Referenz-Spannung des niederwertigsten Bits. Aus diesen Werten kann eine Spannung berechnet werden, welche die am ADC angelegte Spannung abbildet.

5.3.3. Verwendete Strukturen

Die zentrale Komponente der Applikation ist eine C-Struktur, die Daten eines Systemzustands enthält. Die wichtigsten Elemente sind dabei das aktuelle Aktivitätslevel, die Systemzeit, die Spannung des Hauptakkus und die daraus berechnete Restenergie des Akkus. Außerdem enthält die Struktur zwei Zeiger auf Strukturen dieses Typs, wodurch mehrere Instanzen der Struktur zu einer verketteten Liste verknüpft werden können.

Für die einzelnen Aktivitätslevel wird ebenfalls eine Struktur verwendet. Diese beinhaltet das Level, die Dauer des Aktivitätszyklus und die Leistungsaufnahme des Geräts.

Die Daten der Wettervorhersagen werden ebenfalls in einer Struktur abgelegt, welche als Elemente eine Zeit im UNIX-Format und einen Wert für die vorhergesagte Sonnenleistung in mW/m^2 beinhaltet.

Die folgende Abbildung 5.3 zeigt die verwendeten Strukturen mit den jeweiligen Elementen und deren Datentyp.

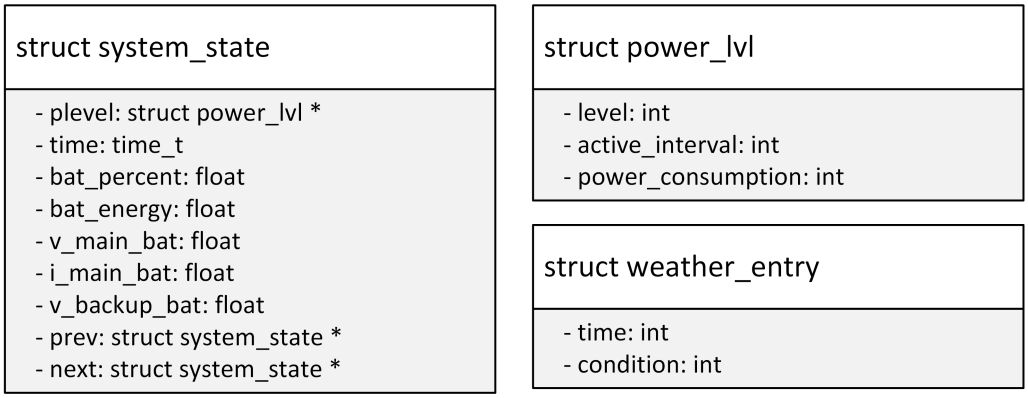

Abbildung 5.3.: Strukturen für Systemzustand, Aktivitätslevel und Wetterdaten

Alle Funktionen wurden so implementiert, dass zur Manipulation eines Systemzustands nur ein Zeiger auf den jeweiligen Zustand übergeben wird. Mehrere Systemzustände werden in einer verketteten Liste verwaltet, sodass Funktionen auch Zugriff auf weitere Elemente der Liste erlangen. So kann jede Funktion auch die gesamte Liste von Zuständen bearbeiten. Für die Organisation der Liste, wurden Funktionen für das Hinzufügen und Löschen von Elementen der Liste implementiert. Nur diese Listen-Funktionen initialisieren und löschen

Instanzen von *struct system_state*. Alle anderen Funktionen greifen nur auf die Werte der einzelnen Listenelemente zu.

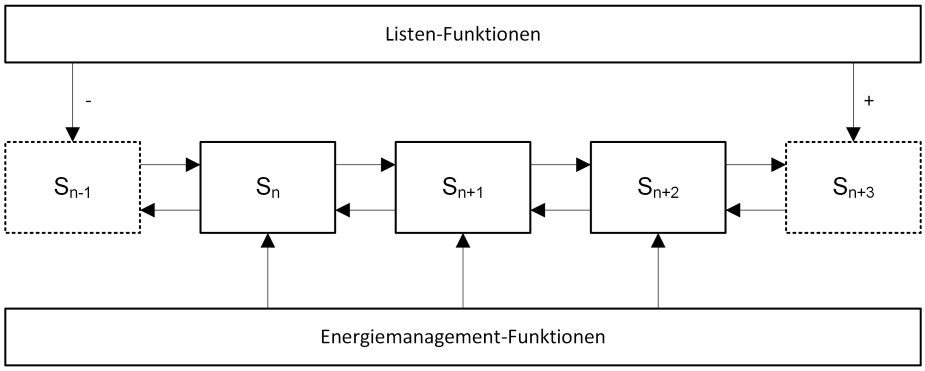

Abbildung 5.4.: Doppelt verkettete Liste von Systemzuständen

5.3.4. Umsetzung der Anwendungssoftware

Nachfolgend wird beschrieben, wie die in Abschnitt 4 entwickelten Ansätze programmiertechnisch umgesetzt wurden. Die exakten Abläufe sollen an dieser Stelle nicht erneut erläutert werden, sondern nur die für die Umsetzung verwendeten Mechanismen. Die Abläufe werden mit Hilfe von Struktogrammen verdeutlicht, wobei grau hinterlegte Anweisungsblöcke in einem weiteren Struktogramm dargestellt und erläutert werden.

Das implementierte Programm arbeitet ohne grafische Oberfläche und wird über die Kommandozeile bedient. Dabei können beim Start mehrere Kommandozeilenparameter angegeben werden, welche das Verhalten des Programms beeinflussen.

Main-Funktion

Die zentrale Funktion des Programms stellt die *Main*-Routine dar. Darin ist die Initialisierung der Variablen, die Verarbeitung der Kommandozeilen-Parameter und die Hauptschleife für die Anwendung enthalten.

In der Hauptschleife werden die Aktivitäten ausgeführt, welche die Funktionalität des Gesamtsystems realisieren. Dazu gehören die Ermittlung des aktuellen Systemzustands, die Durchführung einer Messung und die anschließende Optimierung des aktuellen Aktivitätslevels. Im Anschluss wird die Alarm-Zeit der RTC neu programmiert und das Gerät begibt sich in einen Energiesparmodus. Die Hauptschleife ist als Endlosschleife realisiert, da das Gesamtsystem durchgängig arbeiten soll. Sie wird jedoch in jedem Aktivzyklus nur einmal durchlaufen. Beim Eintritt in den Energiesparmodus werden sämtliche Prozesse des Systems gestoppt und erst nach der Reaktivierung durch den Alarm der RTC wieder an der

Stelle weitergeführt, an der diese angehalten wurden. In diesem Fall wird die Hauptschleife dann ein weiteres Mal ausgeführt bis wieder der Energiesparmodus aktiviert wird.

Die Abbildung 5.5 zeigt ein Struktogramm der *Main*-Routine. Die entsprechende Implementierung im Programm befindet sich auf Seite 90 (ab Zeile 1022) im Anhang A.2.

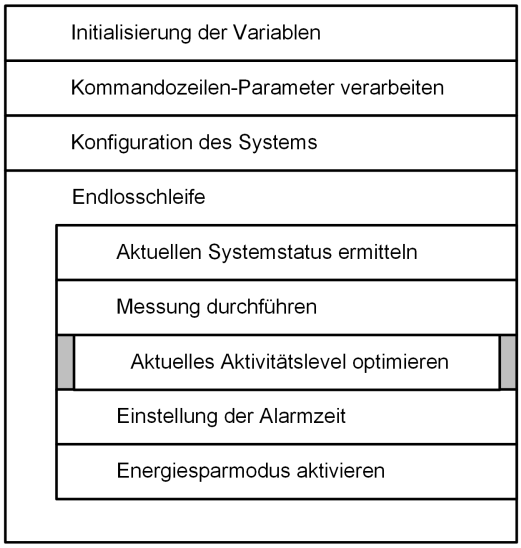

Abbildung 5.5.: Struktogramm der *Main*-Funktion der Anwendung

Optimierung des Aktivitätslevels

Der Anweisungsblock „Aktuelles Aktivitätslevel optimieren“ stellt die eigentliche Umsetzung des Energiemanagement-Algorithmus dar. Hier werden die Abläufe implementiert, welche bereits in Abschnitt 4.4.1 beschrieben sind.

Für die Optimierung des aktuellen Aktivitätslevels wird zunächst eine Vorhersage zukünftiger Ladezustände durchgeführt und die ermittelten Zustände zu einer Liste verkettet. Anschließend wird die Liste vom ersten Element beginnend durchlaufen und jeder Zustand mit einer Optimierungsfunktion geprüft bzw. angepasst, sofern dies notwendig ist. Das Ergebnis der Optimierung wird ausgewertet und entsprechend entweder der nächste Zustand betrachtet, die Prädiktion wiederholt oder die Optimierung beendet. Die Optimierung des Aktivitätslevels ist dann abgeschlossen, wenn die gesamte Liste durchlaufen wurde, ohne einen Zustand weiter optimieren zu müssen, oder eine weitere Anpassung nicht mehr möglich ist.

Das folgende Struktogramm in Abbildung 5.6 zeigt die implementierten Abläufe für die Einstellung des aktuellen Aktivitätslevels. Im Anhang A.2 befindet sich der C-Quellcode für die Funktion ab Seite 87 (Code-Zeile 850).

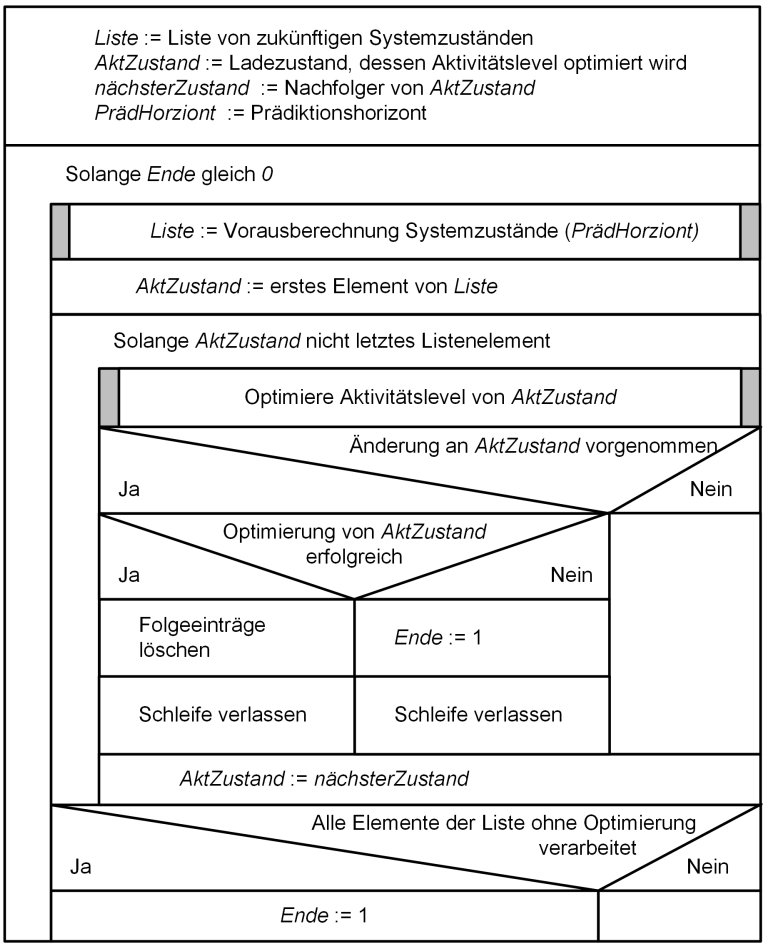

Abbildung 5.6.: Struktogramm für die Optimierung des aktuellen Aktivitätslevels

Vorausberechnung der Systemzustände

Die Vorausberechnungen finden im Anweisungsblock „Vorausberechnung zukünftiger Ladezustände“ statt. Die Funktion führt eine Prädiktion von zukünftigen Systemzuständen anhand von Wettervorhersage-Daten durch. Hierzu werden Modelle für das energetische Verhalten der einzelnen Komponenten des Systems verwendet (Energieverbraucher, Energieerzeuger, Zwischenspeicher). Die Anzahl der zu vorhersagenden Zustände richtet sich nach dem Prädiktionshorizont, welcher der Funktion ebenfalls übergeben wird. Es werden so lange Systemzustände vorhergesagt und deren Zykluszeiten addiert, bis die Summe größer ist als der Prädiktionhorizont. Die Umsetzung der Funktion befindet sich im Anhang A.2 auf Seite 84 ab Code-Zeile 669.

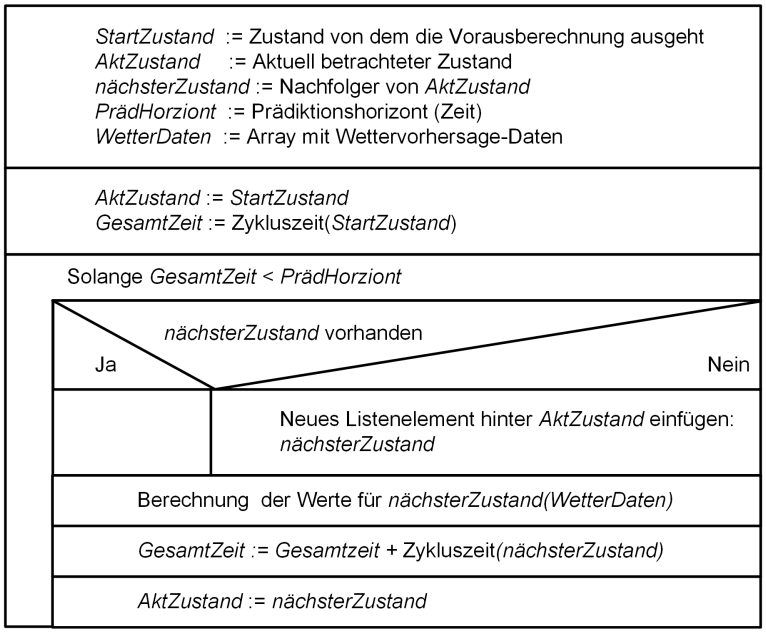

Abbildung 5.7.: Struktogramm für die Vorausberechnung zukünftiger Ladezustände

Rekursive Optimierung des Aktivitätslevels

Es wird nachfolgend die Implementierung der rekursiven Optimierungs-Funktion erläutert. In dieser Funktion ist das „Verhalten“ des Algorithmus implementiert. Die Entwicklung des Konzepts dieser Funktion ist im Abschnitt 4.4.3 zu finden.

Diese Funktion ist für die eigentliche Optimierung der Systemzustände verantwortlich und enthält die Randbedingungen, die einen Ladezustand als optimal oder ungültig kennzeichnen. Gemäß der Anforderungsliste wird eine Bedingung implementiert, die eine Untergrenze für die Restkapazität des Akkus von 25 % darstellt. Die Funktion betrachtet den Folgezustand des zu optimierenden Zustands und prüft, ob die Bedingung erfüllt ist. Ist die Randbedingung erfüllt, werden keine Aktivitätslevel geändert, die Funktion signalisiert „keine Änderung nötig“. Genügt der Folgezustand nicht der Randbedingung, wird das Aktivitätslevel des zu optimierenden Systemzustands reduziert. In diesem Fall wird ein Wert zurückgegeben, der signalisiert, dass ein Aktivitätslevel geändert wurde. Ist eine Reduktion des Aktivitätslevels nicht mehr möglich, werden rekursiv die Aktivitätslevel der vorherigen Systemzustände optimiert, auch wenn diese der Randbedingung entsprechen. Die Rekursion endet, wenn ein Systemzustand gefunden wurde, der weiter optimiert werden kann oder der aktuelle Systemzustand erreicht ist.

Die Abbildung 5.8 zeigt das Struktogramm der rekursiven Optimierungsfunktion. Der Quellcode für die Funktion befindet sich im Anhang A.2 auf Seite 88 ab Zeile 920.

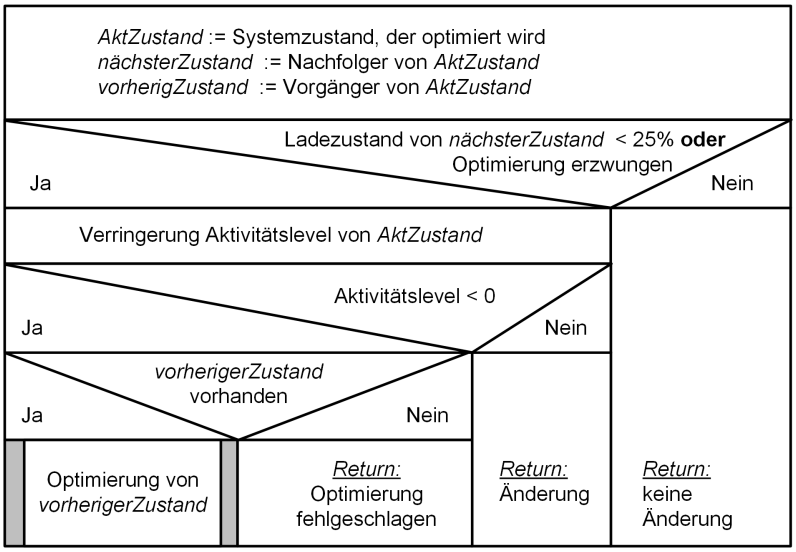

Abbildung 5.8.: Struktogramm für die rekursive Optimierung eines Systemzustands

5.3.5. Verwendung der Software

Um die Software zu starten, muss zunächst ein Login auf dem Gerät stattfinden. Dies kann entweder über den seriellen Debug-Port (RS232-1) über ein geeignetes Kabel oder durch eine Secure Shell (SSH)-Verbindung über das Netzwerk realisiert werden.

Das Programm gibt über die Standardausgabe Nachrichten aus, um den Benutzer über mögliche Fehler und Ereignisse zu informieren. Die Nachrichten werden mit unterschiedlichen Prioritäten ausgegeben, mit Hilfe von Kommandozeilenparametern kann die Anzahl der auszugebenden Nachrichten gesteuert werden. Bei einem Standardaufruf werden nur Fehlermeldungen und Warnungen ausgegeben.

Während das Programm ausgeführt wird, werden automatisch mehrere Log-Dateien erstellt. Darin wird der Systemstatus für jede aktive Phase festgehalten, um später analysieren zu können, welche Zustände das System einnimmt. Außerdem wird eine Datei erstellt, in welcher die Durchführung von Messungen mit entsprechendem Datum und Uhrzeit protokolliert wird.

Um die Software über die Kommandozeile zu starten, wird der Befehl „pman“ verwendet. Hierbei ist zu beachten, dass es sinnvoll ist, beim Start den Prozess von der aktuellen Login-Shell zu entkoppeln, da sonst das Programm beendet werden würde, sobald die Login-Shell geschlossen wird. In Linux kann dazu der *detach*-Parameter „&“ verwendet werden.

Beim Aufruf können zusätzliche Kommandozeilen-Parametern angegeben werden, die in der folgenden Übersicht mit ihrer jeweiligen Funktion dargestellt sind.

Parameter	Funktion
-h, –help	Zeigt eine Hilfe, sowie die verfügbaren Parameter an und beendet das Programm
-s, –status	Zeigt den aktuellen Status des Systems an und beendet das Programm
-v, –verbose	Erhöhung der Anzahl der auszugebenden Nachrichten
-q, –quiet	Verringerung der Anzahl der auszugebenden Nachrichten

Tabelle 5.4.: Kommandozeilenparameter der Anwendung

Die Parameter „-v“ und „-q“ können mehrfach angegeben werden, wodurch die Anzahl der Ausgaben stufenweise geändert wird.

Der folgende Beispielaufruf startet das Programm als entkoppelten Hintergrundprozess mit erhöhtem Verbositätslevel, sodass auch Nachrichten mit der Priorität „debug“ ausgegeben werden:

```
root@livius-0:~# pman -v &
```

5.4. Implementierung der Anwendungssoftware auf die vorhandene Hardware

Die entwickelte Anwendungssoftware muss an die Randbedingungen des Gesamtsystems und der verwendeten Hardware angepasst werden. Es werden entsprechende Modelle für die verwendeten Hardware-Komponenten ausgelegt, welche dann von der Software verwendet werden, um die Vorausberechnung von zukünftigen Systemzuständen durchzuführen. Weiterhin werden die Aktivitätslevel definiert, wobei unterschiedliche Aufteilungen der Zykluszeit verwendet werden, um verschiedene Aktivitätslevel-Kennlinien zu erstellen.

5.4.1. Modellierung des Gesamtsystems

Nachfolgend werden die Modelle beschrieben, die das energetische Verhalten des Gesamtsystems abbilden. Wie in der Abbildung 4.14 dargestellt, werden Modelle für das eingebettete System, das Solarmodul und den Akkumulator benötigt, um einen zukünftigen Systemzustand berechnen zu können.

Solarmodul

Das Solarmodul sorgt für die Energiezufuhr in den Akkumulator. Mit Hilfe von konkreten Wettervorhersagen für die Sonnenstrahlungsleistung kann der zukünftige Energieertrag durch das Solarmodul berechnet werden. Die Leistungsabgabe wird mit der Gleichung 4.7 berechnet.

Dazu wird zunächst der Wirkungsgrad des Solarmoduls aus den Nenndaten ermittelt. Die Nennleistung eines Solarmoduls wird im Normalfall unter Standardbedingungen (Standard Test Conditions (STC)) ermittelt. Die STC-Sonnenenergie beträgt 1000 W/m^2. Mit Hilfe der Solarmodul-Oberfläche und der angegebenen Nennleistung kann so der Wirkungsgrad des Moduls berechnet werden. [17, vgl. S. 31]

$$\begin{aligned} \eta_{STC} &= \frac{P_{STC}}{E_{STC} \cdot A} \\ &= \frac{6W}{1000\frac{W}{m^2} \cdot 0,146475m^2} \\ &= \underline{\underline{0,04096}} \end{aligned} \tag{5.1}$$

So ergibt sich die folgende Gleichung 5.2, mit der der Energieertrag des verwendeten Solarmoduls berechnet werden kann.

$$P_{Solar} = E_{Sonne} \cdot 0,146475m^2 \cdot 0,04096 \tag{5.2}$$

Daraus ergibt sich die in Abbildung 5.9 dargestellte Kennlinie für die Solarmodul-Leistung über der Sonneneinstrahlung.

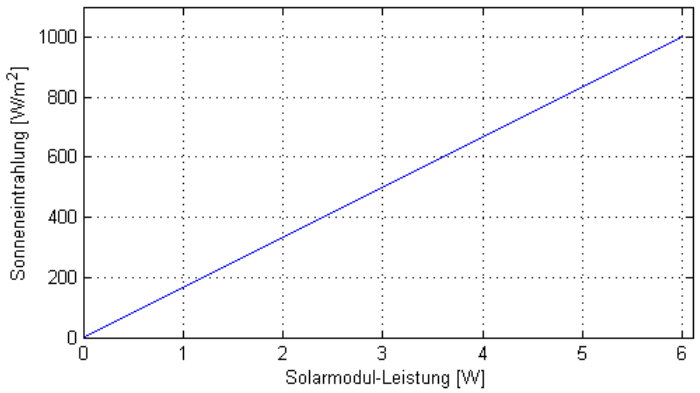

Abbildung 5.9.: Verwendete Kennlinie der Solarmodul-Leistung über der Sonneneinstrahlung

Akkumulator

Der Akkumulator dient dem Gerät als Zwischenspeicher für die Energie, welche vom Solarmodul erzeugt wird. Um vorauschauend zu agieren, muss im laufenden Betrieb der aktuelle Ladezustand des Akkus ermittelt werden können. Dafür wird die Spannung des Akkus gemessen und so ein Rückschluss auf dessen Restenergie gezogen.

Es wird eine lineare Akkumulatorkennlinie angenommen, welche von einer maximalen Spannung von 13,8 V und einer minimalen Spannung von 10,5 V ausgeht. Dabei wird nicht zwischen Auf- und Entladevorgängen unterschieden.

Die Abbildung 5.10 zeigt die Kennlinie für einem Entladestrom von 180 mA (maximale Stromaufnahme des eingebetteten Systems im Aktivmodus).

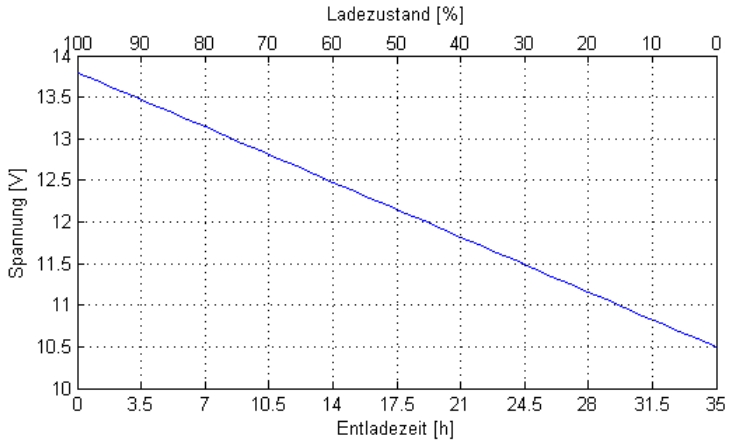

Abbildung 5.10.: Verwendete Entladungs-Kennlinie des Akkumulators

Um zukünftige Energiezustände des Akkus zu berechnen werden die Gleichungen 4.1 und 4.7 zur Berechnung des Energieverbrauchs durch das Gerät bzw. des Energieetrags durch das Solarmodul verwendet. Da das Aktivitätslevel für vorausberechnete Systemzustände bekannt ist, kann die Zeitspanne, in der Energie verbraucht und aufgeladen wird, bestimmt werden. Zusammen mit der jeweiligen positiven bzw. negativen Leistungaufnahme wird eine Energiebilanz gebildet, welche zur Restenergie des Akkus addiert wird (Gleichung 5.3).

$$\begin{aligned} E_{Akku} &= E_{Rest} + E_{Solar} - E_{Gerät} \\ E_{Akku} &= E_{Rest} + (P_{Solar} * t_{Zyklus}) - (P_{Gerät} * t_{Zyklus}) \end{aligned} \tag{5.3}$$

Eingebettetes System

Die Steuerung stellt den Energieverbraucher des Systems dar. Der Verbrauch wird dabei durch das Aktivitätslevel beeinflusst und ändert sich über der Zeit. Jedes Aktivitätslevel steht für eine spezifische mittlere Leistungsaufnahme, welche über den gesamten Aktivitätszyklus als konstant betrachtet wird. Das energetische Verhalten der Steuerung ist demnach in den unterschiedlichen Aktivitätsleveln abgebildet. Die Auslegung der Aktivitätslevel folgt in Abschnitt 5.4.2.

5.4.2. Auslegung der Aktivitätslevel

Die Aktivitätslevel dienen dem Algorithmus als Steuergröße für die Beeinflussung der Energieaufnahme und müssen daher sinnvoll ausgelegt werden. Dazu werden zunächst die einzuhaltenden Randbedingungen ermittelt und anschließend verschiedene Aktivitätslevel-Kennlinien definiert.

Randbedingungen und Annahmen

Die Randbedingungen werden entweder durch die Hardware vorgegeben oder sinnvoll angenommen. Die Verweildauer des Geräts im aktiven Modes wird auf 60 Sekunden festgelegt und für jedes Aktivitätslevel als konstant betrachtet. Innerhalb dieser Zeit wird später das Gerät die Messungen durchführen. Für die Auslegung ist weiterhin interessant wie groß die Schrittweite der Zeit im Passivmodus gewählt werden kann. Die verwendete RTC kann einen Interrupt zum Aufwecken nur mit einer Genauigkeit auf Minuten, nicht jedoch auf Sekunden auslösen. Daher ist die minimale Schrittweite für die Passivzeit 60 Sekunden. So ist auch die Zeit im Passivmodus mindestens 60 Sekunden lang. Als kürzester Aktivitätszyklus ergibt sich so aus der Zeit im Aktivmodus und der Zeit im Passivmodus 120 Sekunden, während die maximale Länge eines Gesamtzyklus mit 1h (3600 Sekunden) angenommen wird.

Wie im Abschnitt 4.4 ermittelt wurde, kann der Energieverbrauch nur zwischen der maximalen und minimalen Leistungsaufnahme gesteuert werden. Daher muss vor der Definition der Aktivitätslevel die Leistungsaufnahme der Steuerung im Aktiv- und Passivmodus gemessen werden. Das Gerät wird dazu an eine Spannungsversorgung angeschlossen und die Stromaufnahme jeweils für den Aktiv- und Passivmodus gemessen.

Die Tabelle 5.5 zeigt die Ergebnisse der Messungen der Energieaufnahme.

Modus	Spannung [V]	Strom [A]	Leistung [W]
Aktiv	13,8	0,1796	2,478
Passiv	13,8	0,0453	0,625

Tabelle 5.5.: Leistungsaufnahme des verwendeten Systems im Aktiv- und Passivmodus

Mit Hilfe der Leistungsaufnahmen und den zuvor ermittelten Randbedingungen für die Aktivitätszyklen kann nun der mit dem Gesamtsystem mögliche maximale und minimale Energieverbrauch berechnet werden. (Gleichung 5.4, 5.5):

$$P_{min} = 0,625W + (2,478W - 0,625W)\left(\frac{60s}{3600s}\right) = \underline{\underline{0,656W}} \tag{5.4}$$

$$P_{max} = 0,625W + (2,478W - 0,625W)\left(\frac{60s}{120s}\right) = \underline{\underline{1,551W}} \tag{5.5}$$

Die folgendende Tabelle 5.6 fasst die ermittelten Randbedingungen zusammen:

Parameter	Wert
Kleinster Aktivitätszyklus	120 s
Größter Aktivitätszyklus	3600 s (Annahme)
Zeit im Aktivmodus (konst.)	60 s (Anforderung)
Minimale Zeit im Passivmodus	60 s
Minimale Schrittweite Passivzeit	60 s (Vorgabe Hardware)
Maximale Leistungsaufnahme	1,550 W
Minimale Leistungsaufnahme	0,625 W

Tabelle 5.6.: Randbedingungen für die Auslegung der Aktivitätslevel

Definition der Aktivitätslevel

Mit den ermittelten Randbedingungen und den in Tabelle 5.5 gemessenen Werten, werden nun die Aktivitätslevel definiert. Die Anzahl der Level wird auf 11 (0 bis 10) festgelegt und es werden mehrere Aktivitätslevel-Kennlinien mit unterschiedlichen Aufteilungen der Zykluszeiten erstellt. So kann später die Auswirkung unterschiedlicher Kennlinien auf das Verhalten des Algorithmus ausgewertet werden.

Aktivitätkennlinie I:

Die erste Serie von Aktivitätsleveln verwendet einen linearen Verlauf der Zeit im Passivmodus. Als maximale Zykluszeit wird die Randbedingung von 1h ausgewählt, die minimale Zeit wird auf 180 Sekunden festgelegt. Zwischen diesen zwei Zeiten werden die anderen 9 Zykluszeiten weitestgehend gleichmäßig verteilt und die jeweilige Leistungsaufnahme berechnet. Für diese Serie bleibt das Gerät für 90 s statt für 60 s im aktiven Modus.

Level	Aktivitätszyklus [s]	Energieverbrauch [W]
0	3600	0,671
1	3258	0,676
2	2916	0,682
3	2574	0,689
4	2232	0,699
5	1890	0,716
6	1548	0,732
7	1206	0,762
8	864	0,817
9	522	0,943
10	180	1,550

Tabelle 5.7.: Definition der Aktivitätslevel-Kennlinie I

Die Abbildung 5.11 zeigt die Leistungsaufnahmen und Passivzeiten für die verschiedenen Aktivitätslevel.

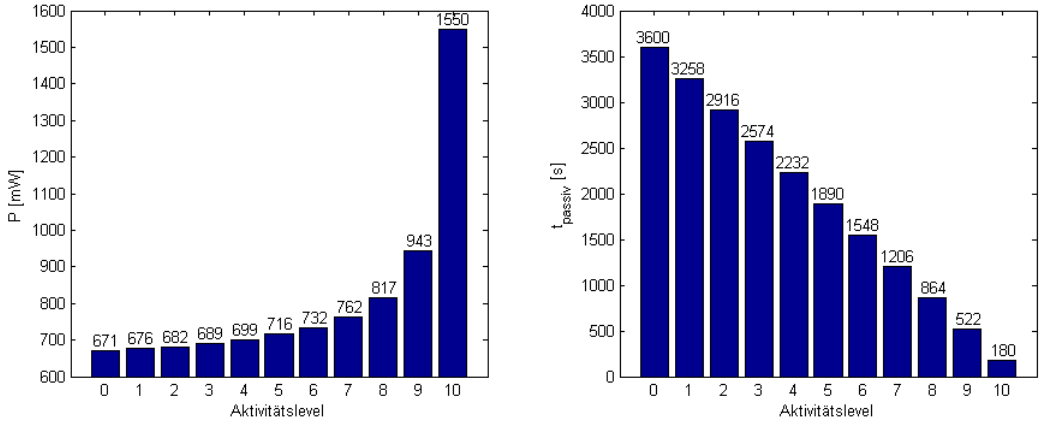

Abbildung 5.11.: Leistungsaufnahme und Passivzeit der Aktivitätslevel-Kennlinie I

Aktivitätkennlinie II:

Bei der Auslegung der zweiten Kennlinie wird versucht einen möglichst linearen Zusammenhang zwischen Aktivitätslevel und Leistungsaufnahme zu erzeugen. Durch den nichtlinearen Zusammenhang von Passivzeit und Energieverbrauch, in Kombination mit der Schrittweite der Passivzeit von 60 Sekunden, kann insbesondere für die hohen Aktivitätslevel jedoch kein linearer Zusammenhang erreicht werden. Hier wäre es notwenig die Passivzeit mit einer höheren Genauigkeit einzustellen. Daher wird in für die Aktivitätslevel 4 - 10 die minimale Schrittweite verwendet.

Level	Aktivitätszyklus [s]	Energieverbrauch [W]
0	3600	0,656
1	1449	0,702
2	813	0,762
3	564	0,822
4	480	0,856
5	420	0,889
6	360	0,934
7	300	0,995
8	240	1,080
9	180	1,242
10	120	1,551

Tabelle 5.8.: Definition der Aktivitätslevel-Kennlinie II

Nachfolgend ist der Verlauf von Leistungsaufnahme bzw. Passivzeit über den Aktivitätsleveln der Kennlinie II in Abbildung 5.12 dargestellt.

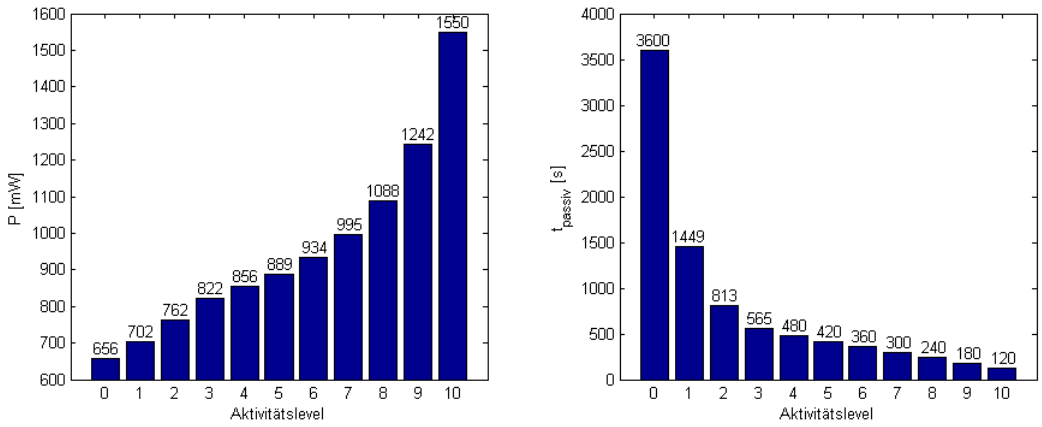

Abbildung 5.12.: Leistungsaufnahme und Passivzeit der Aktivitätslevel-Kennlinie II

Aktivitätkennlinie III:

Die dritte Definition von möglichen Aktivitätsleveln enthält wieder eine lineare Aufteilung der Zykluszeit. Die maximale Zeit wird, wie bei den vorherigen Serien, auf 1h begrenzt, jedoch wird nun die minimale Zeit auf 10 min (600 s). Dadurch ergibt sich insgesamt ein wesentlich geringerer Energieverbrauch des Systems.

In der folgenden Tabelle 5.9 werden die berechneten Werte für die Aktivitätkennlinie III dargestellt.

Level	Aktivitätszyklus [s]	Energieverbrauch [W]
0	3600	0,656
1	3300	0,659
2	3000	0,662
3	2700	0,666
4	2400	0,671
5	2100	0,678
6	1800	0,687
7	1500	0,699
8	1200	0,718
9	900	0,748
10	600	0,810

Tabelle 5.9.: Definition der Aktivitätslevel-Kennlinie III

Der Verlauf von Leistungsaufnahme bzw. Passivzeit für die Aktivitätslevel-Kennlinie III ist in Abbildung 5.13 dargestellt.

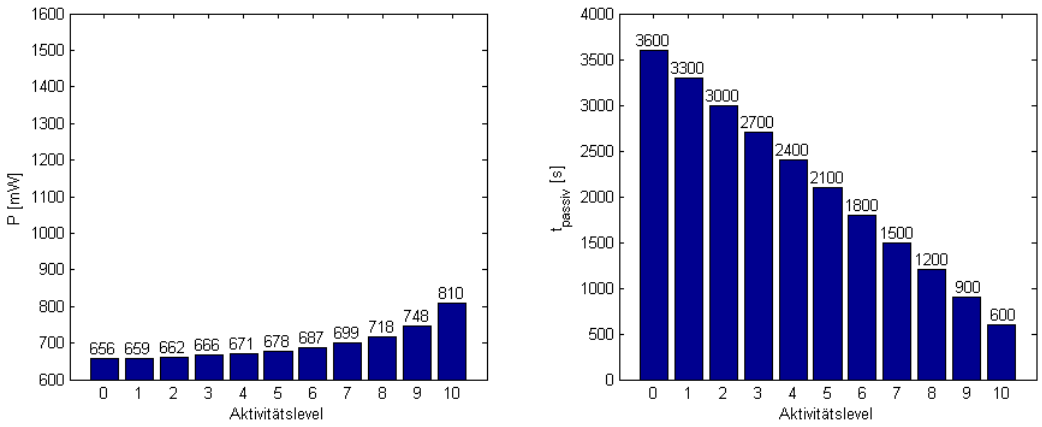

Abbildung 5.13.: Leistungsaufnahme und Passivzeit der Aktivitätslevel-Kennlinie III

6. Messungen und Auswertung

Nachdem ein funktionsfähiges System aufgebaut und die erforderliche Software umgesetzt wurde, wird nun das Verhalten des Gesamtsystems mit vorausschauendem Energiemanagement-Algorithmus analysiert. Dazu werden Szenarien entwickelt, welche den Algorithmus in unterschiedlichen Situationen testen. Mit Hilfe der von der Software erstellen Logdateien wird anschließend das Verhalten des Systems für die verschiedenen Szenarien ausgewertet.

6.1. Messungen

6.1.1. Messaufbau

Für die Messungen wird ein Messaufbau ohne Solarmodul verwendet. Um das Verhalten des Algorithmus unter unterschiedlichen Konditionen messen zu können, muss die Energiezufuhr durch das Solarmodul gesteuert werden können. Da eine Vorrichtung zur gezielten Bestrahlung des Solarmoduls nicht zur Verfügung steht, wird das Modul durch ein Labornetzgerät „simuliert“. Die Stromabgabe des Netzteils wird so begrenzt, dass die abgegebene Energie einem Wert entspricht, der mit der Leistung eines Solarmoduls vergleichbar ist. Durch Ein- bzw. Ausschalten des Labornetzgeräts kann nun die Energiezufuhr exakt gesteuert werden. Das Labornetzgerät wird auf eine Spannung von 13,8 V eingestellt und der Strom auf 0,2 A begrenzt. Dies entspricht einer Sonneneinstrahlung von 483 W/m^2 auf das vorhandene Solarmodul.

Zur Messung der Akkuspannung wird ein analoger Eingang am Gerät verwendet. Diese können mit Spannungswerten zwischen 0 V und 10 V beschaltet werden. Da die maximale Spannung des Akkus jedoch 13,8 V beträgt, wird ein Spannungsteiler mit einem Verhältnis von ca. 1:0,723 und einen daran angeschlossenen Impedanzwandler verwendet, um die Akkuspannung zu messen. Dadurch wird die maximale Spannung des Akkus von 13,8 V auf ca. 9,98 V herabgesetzt. In der Software wird der Spannungswert mit dem Kehrwert des Spannungsteilers multipliziert und somit die reale Akkuspannung berechnet. Der Impedanzwandler muss verwendet werden, da der Eingangswiderstand des A/D-Wandlers, gemessen an den Werten der für den Spannungsteiler verwendeten Widerstände, nicht hochohmig ist. Dies würde zu einem Abfall der zu messenden Spannung führen.

Der Messaufbau in Abbildung 6.1 wird für die Messungen verwendet.

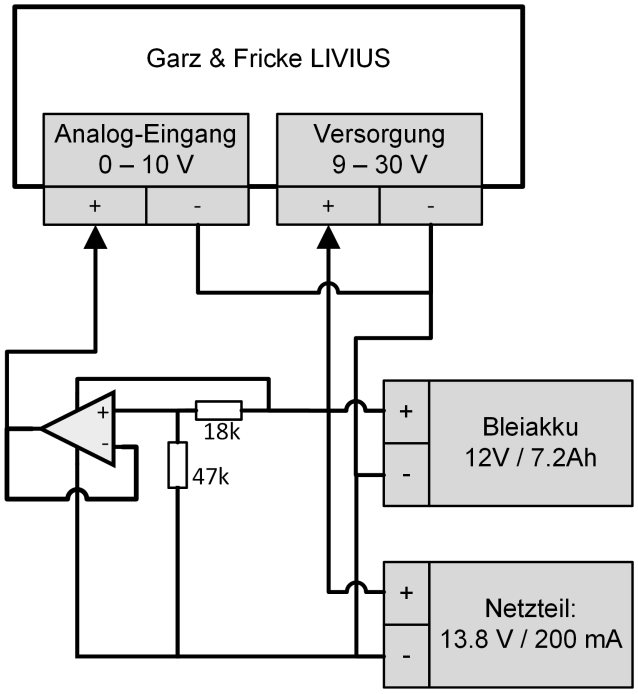

Abbildung 6.1.: Schaltplan des Messaufbaus

6.1.2. Durchführung der Messungen

Die Messungen werden mit unterschiedlichen Aktivitätslevel-Kennlinien, Wettervorhersagen und Energiezuführungen realisiert. Alle Messungen werden über einen Zeitraum von 4 Stunden und mit einem Prädiktionshorizont von 4 Stunden durchgeführt. Die Spannungs-Grenze des Akkus, welche nicht unterschritten werden soll, wird für jede Messung neu definiert. Die Wettervorhersagen für die Sonnenstrahlung entsprechen entweder der real möglichen Energiezufuhr von 483 W/m^2 oder 0 W/m^2. Diese binäre Aufteilung wurde gewählt, da mit dem vorhandenen Messaufbau auch nur eine binäre Energiezufuhr möglich ist. Weiterhin muss beachtet werden, dass immer Vorhersage-Daten für 8 Stunden vorhanden sein müssen, da das System für jeden Zeitpunkt während der Messung eine Prädiktion von 4 Stunden durchführt. Dazu werden zwei gleiche Vorhersage-Datensätze für 4 Stunden verwendet, welche hintereinander gereiht werden.

Es werden zwei Grundszenarien für die Messungen verwendet, um den Algorithmus zu analysieren.

Szenario 1: Es wird eine einfache Entladungs-Situation simuliert. Für die gesamte Messung ist keine Sonnenstrahlung vorhergesagt und es wird auch real keine Energie zugeführt. Dieses Szenario entspricht dem im Abschnitt 4 beschriebenen Beispiel. Anhand der Ergebnisse dieses Szenarios wird das vorausschauende Verhalten des Algorithmus unter vereinfachten Bedingungen analysiert, daher werden auch Prädiktions-Daten von markanten Zeitpunkten in einem Diagramm aufgezeichnet.

Szenario 2: Das zweite Szenario deckt innerhalb der Messperiode mehrere Kombinationen aus Wettervorhersage und realer Energiezufuhr ab. Auf diese Weise wird das Verhalten unter stark dynamischen Umwelteinflüssen analysiert. Die nachfolgenden Situationen werden in diesem Szenario abgedeckt:

- Energiezufuhr vorhergesagt und es findet real die exakt gleiche Energiezufuhr statt
- keine Energiezufuhr vorhergesagt und es findet keine reale Energiezufuhr statt
- Energiezufuhr vorhergesagt, doch es findet keine reale Zufuhr statt
- keine Energiezufuhr vorhergesagt, doch es findet reale Zufuhr statt

Entsprechend der Tabelle 6.1 wurden die Messungen mit den unterschiedlichen Szenarien und verschiedenen Aktivitätslevel-Kennlinien durchgeführt

Messung Nr.	Aktivitätslevel-Kennlinie	Szenario (1/2)	Start-Spannung	Grenz-Spannung
1	I	1	11.9752 V	11.820 V (40 %)
2	I	2	11.9412 V	11.919 V (43 %)
3	II	2	11.9380 V	11.886 V (42 %)
4	III	2	11.9177 V	11.899 V (42.4 %)

Tabelle 6.1.: Übersicht der durchgeführten Messungen und deren Randbedingungen

Die Software zeichnet selbstständig die Daten der einzelnen Systemzustände im Aktivmodus auf, dazu zählt die Systemzeit, das Aktivitätslevel, sowie die Akku-Spannung.

6.1.3. Messergebnisse

Dieser Abschnitt zeigt die Ergebnisse der Messungen, welche den Logdateien der Anwendungssoftware entnommen werden können. Für jede Messung ist jeweils die Akku-Spannung, das Aktivitätslevel, sowie die vorhergesagte und reale Energiezufuhr dargestellt. Über den jeweiligen Graphen sind die wichtigsten Kenndaten der Messung zusammengefasst.

Messung 1

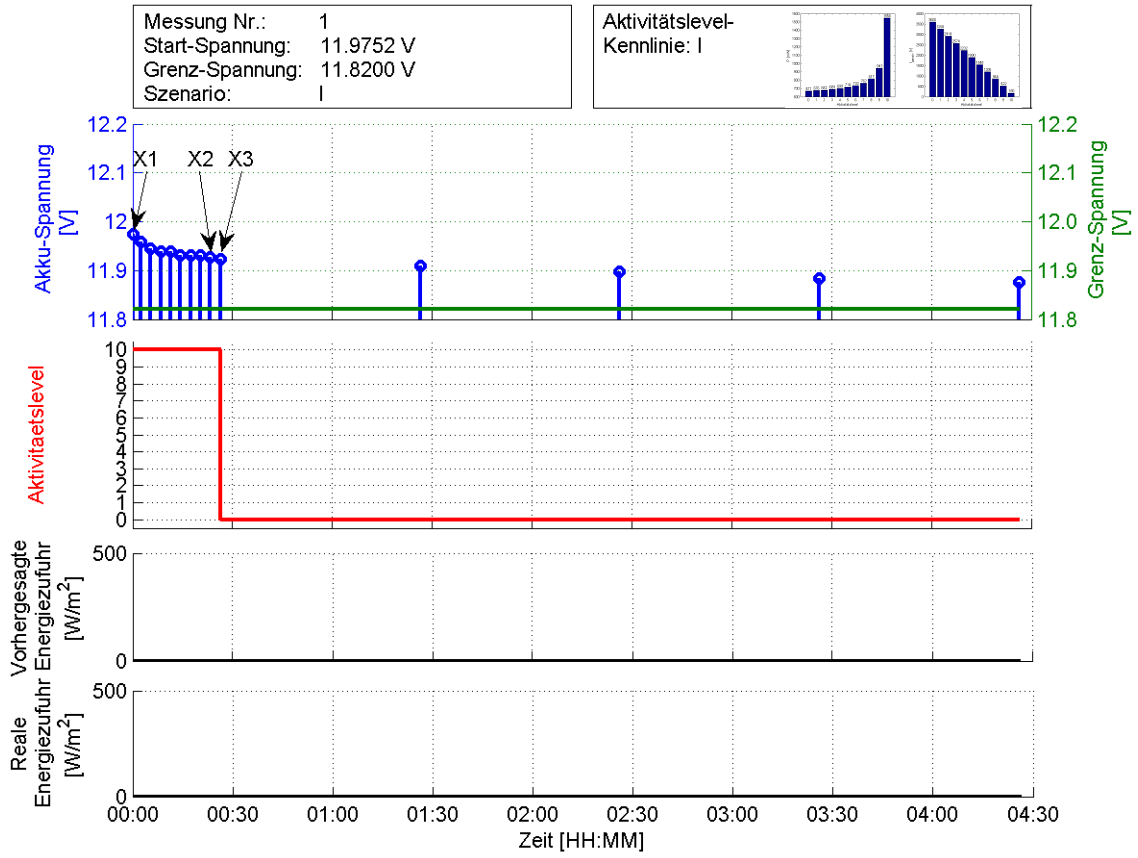

Abbildung 6.2.: Ergebnis der Messung 1

Für die Messung wird die Aktivitätskennlinie I verwendet (lineare Zykluszeit) und es wird bei einer Akku-Spannung von 11.9752 V gestartet. Die Grenz-Spannung wird auf 11.82 V festgelegt.

Das System startet im höchsten Aktivitätslevel (10) und schaltet nach ca. 25 Minuten auf das niedrigste Aktivitätslevel 0. Dieses Level wird dann bis zum Ende der Messung gehalten. Es ist außerdem erkennbar, dass die Akku-Spannung nach 4 Stunden die Grenz-Spannung nicht unterschritten hat, sondern weiter darüber liegt. Um die Entscheidung des Algorithmus zum Zeitpunkt X3 zu analysieren, werden die Daten der Prädiktionen betrachtet, die zu den Zeitpunkten X1, X2 und X3 berechnet wurden.

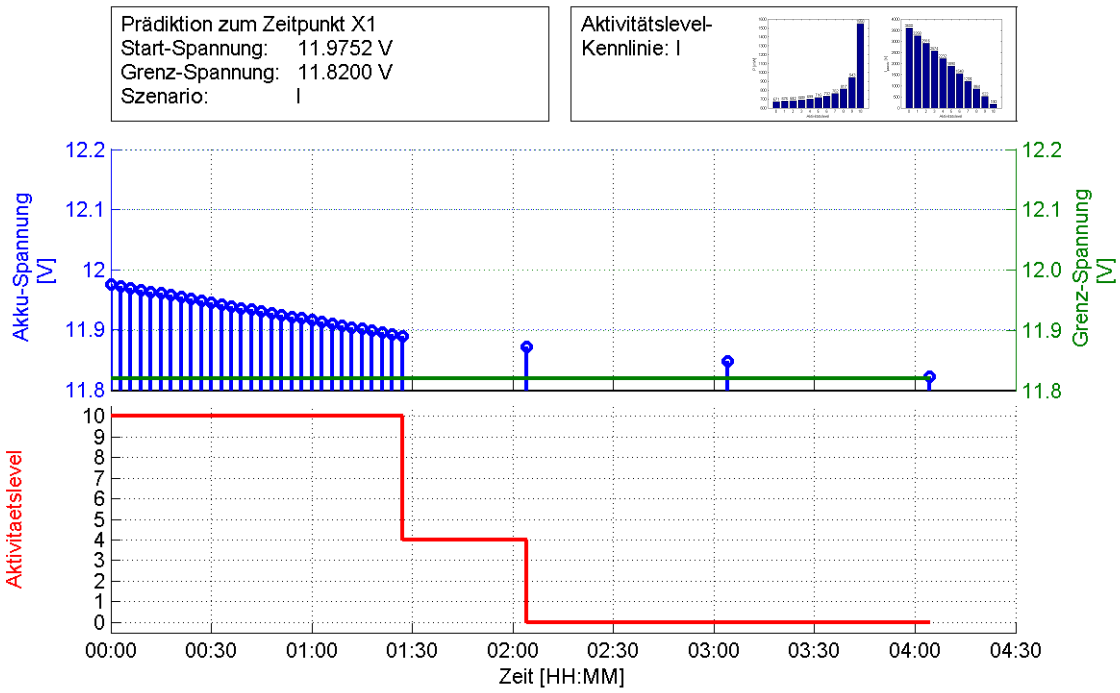

Abbildung 6.3.: Ergebnis der Prädiktion zum Zeitpunkt X1 (Messung 1)

Der Zeitpunkt X1 ist der Startpunkt der Messreihe. Anhand der Vorausberechnungen des Algorithmus für diesen Zeitpunkt (Abb. 6.3), muss nach 1,5 Stunden das Aktivitätslevel auf 4 und anschließend nach wiederum ca. 30 Minuten auf das Level 0 gewechselt werden, um nach 4 Stunden nicht unter die Grenz-Spannung zu fallen. In der Realität wechselt das System aber bereits nach ca. 30 Minuten auf das minimale Level. Das liegt daran, dass zu einem späteren Zeitpunkt X2 die Vorausberechnung wieder für 4 Stunden durchgeführt wird, jedoch zwischenzeitlich keine Aufladung stattfindet. Die Berechnung beginnt demnach bei einer niedrigeren Akku-Spannung, darf jedoch trotzdem nach 4 Stunden nicht die Grenz-Spannung unterschreiten. Daher muss in in der zweiten Vorausberechnung mehr Energie eingespart werden, als noch bei der ersten Berechnung.

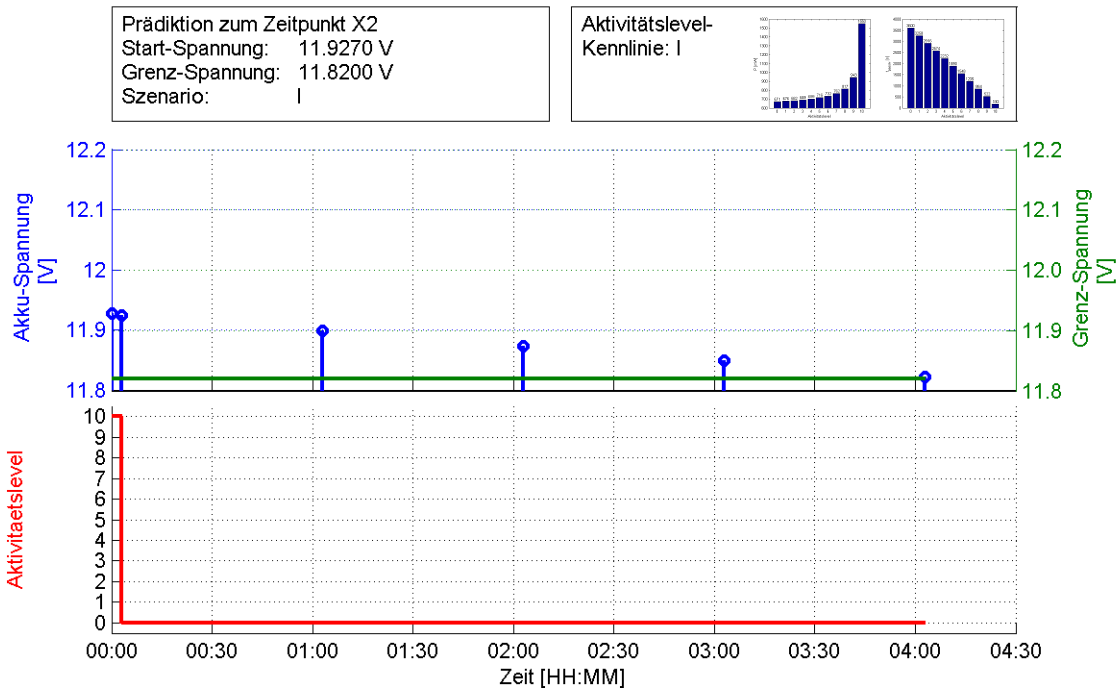

Abbildung 6.4.: Ergebnis der Prädiktion zum Zeitpunkt X2 (Messung 1)

Die Vorausberechnung zum aktiven Zeitpunkt X2, ergibt, dass das Aktivitätslevel beim nächsten Aktivzyklus auf den Wert 0 geändert werden muss, um innerhalb des Prädiktionshorizonts zulässige Akkuladezustände zu erhalten. Für den aktuellen Systemzustand kann jedoch noch das maximale Aktivitätslevel beibehalten werden (s. Abb. 6.4).

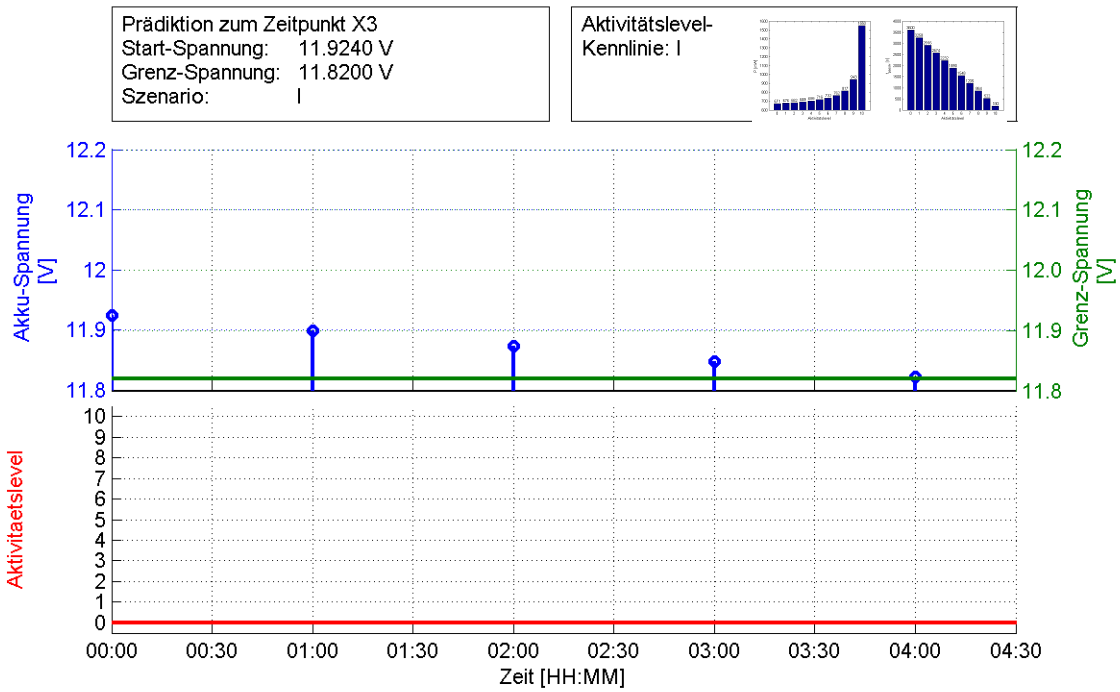

Abbildung 6.5.: Ergebnis der Prädiktion zum Zeitpunkt X3 (Messung 1)

Die Vorausberechnung zum Zeitpunkt X3 ergibt, dass es notwendig ist, von nun an nur noch das Aktivitätslevel 0 zu verwenden, um in 4 Stunden nicht unter die Grenz-Spannung zu fallen. Daher wechselt das System zu diesem Zeitpunkt auf das Aktivitätslevel 0.

Messung 2

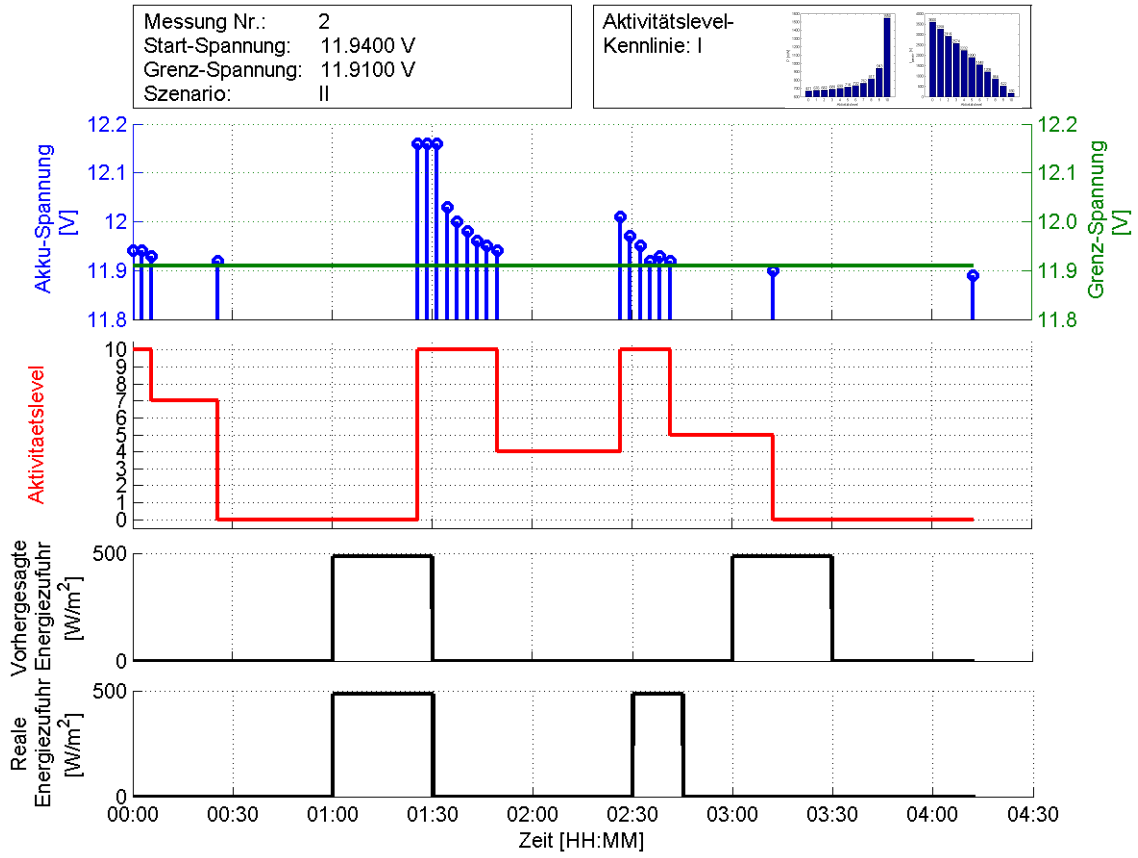

Abbildung 6.6.: Ergebnis der Messung 2

Es wird eine Aktivitätskennlinie mit einem linearen Verlauf der Zykluszeit und einem minimalen Zyklus von 3 Minuten verwendet. Die Abbildung 6.6 zeigt die aufgezeichneten Graphen für diese Messung. Der Algorithmus reduziert hier bereits nach kurzer Zeit das Aktivitätslevel, um nach ca. 30 Minuten auf das minimale Level zu wechseln. Durch die lange Passivzeit des minimalen Levels ist das System erst dann wieder aktiv, wenn die erste Aufladungsphase bereits fast abgeschlossen ist. Jedoch wird die höhere Akku-Spannung erkannt und das Aktivitätslevel auf den maximalen Wert eingestellt. Das Level wird für ca. 20 Minuten gehalten und anschließend auf ein mittleres Level gewechselt. Auf die nicht vorhergesagte Aufladung reagiert der Algorithmus mit einem sofortigen Wechsel auf das maximale Aktivitätslevel. Während der vorhergesagten Aufladung, welche jedoch nicht stattfindet, fällt der Akku-Spannungswert unter die Grenz-Spannung, worauf das Aktivitätslevel 0 aktiviert wird. Insgesamt ist das System innerhalb der 4 Stunden 21 Mal aktiv.

Messung 3

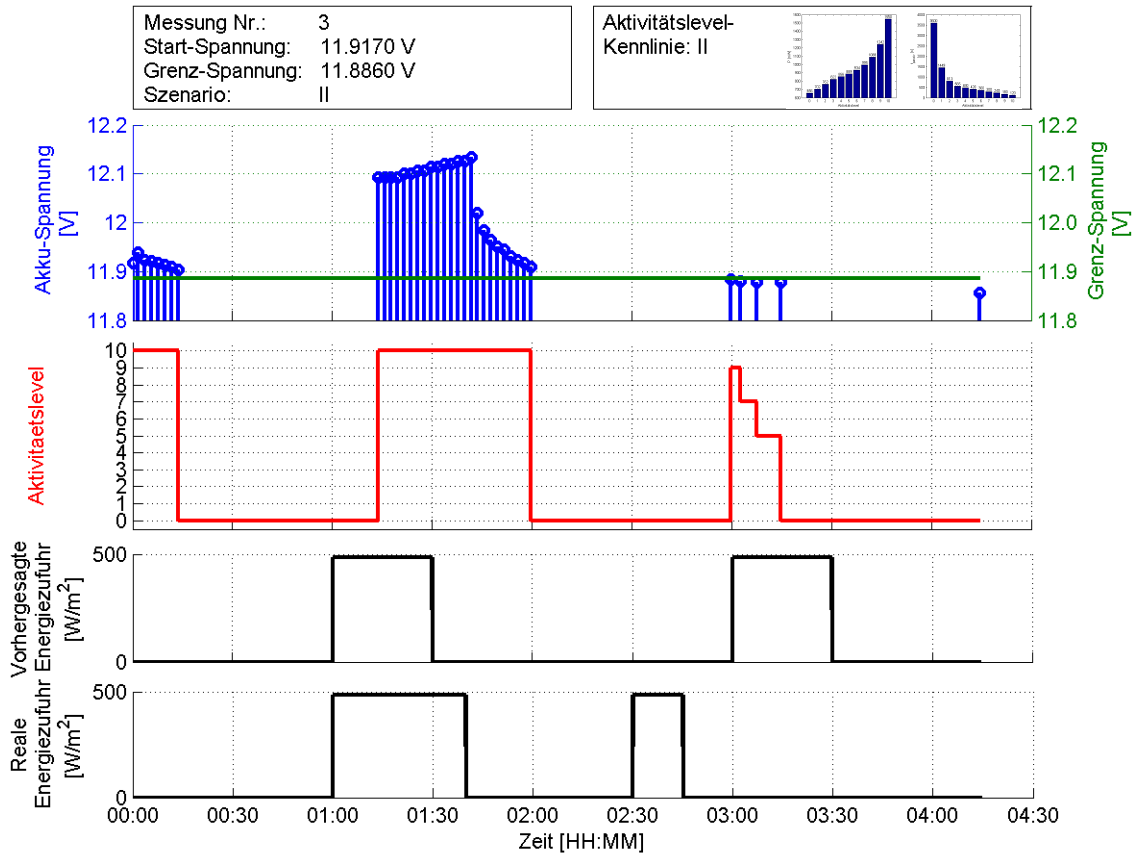

Abbildung 6.7.: Ergebnis der Messung 3

Für die nächste Messung (s. Abb. 6.7) wird die Aktivitätskennlinie II und eine leicht modifizierte reale Energiezufuhr verwendet. Die minimale Zykluszeit ist auf 2 Minuten festgelegt. Nach dem Start mit Aktivitätslevel 10 wechselt das System nach ca. 15 Minuten auf Level 0. Der nächste aktive Zeitpunkt befindet sich anschließend während der ersten vorhergesagten und realen Aufladung, was zu einem erneuten Wechsel auf das Level 10 führt. Das Aktivitätslevel wird auch nach der Aufladung weiter gehalten, bis nach ca. 20 Minuten wieder das minimale Level aktiv wird. Auf die unerwartete Aufladung reagiert der Algorithmus nicht, da sich das System während dieser Zeit im Passivmodus befindet. Der nächste Aktivzyklus beginnt zum Zeitpunkt der vorhergesagten, aber nicht erfolgten Aufladung. Die Akku-Spannung hat hier bereits die Grenz-Spannung erreicht. Da der Algorithmus jedoch mit einer Aufladung rechnet, wird ein hohes Aktivitätslevel gewählt. Die Level werden nun nach und nach reduziert, da die erwartete Aufladung nicht erfolgt, bis schließlich das minimale Aktivitätslevel aktiviert wird. Am Ende der Messung liegt die-Akku-Spannung unter der Grenz-Spannung. Das System führt in dieser Messung insgesamt 37 Messungen im Aktivmodus durch.

Messung 4

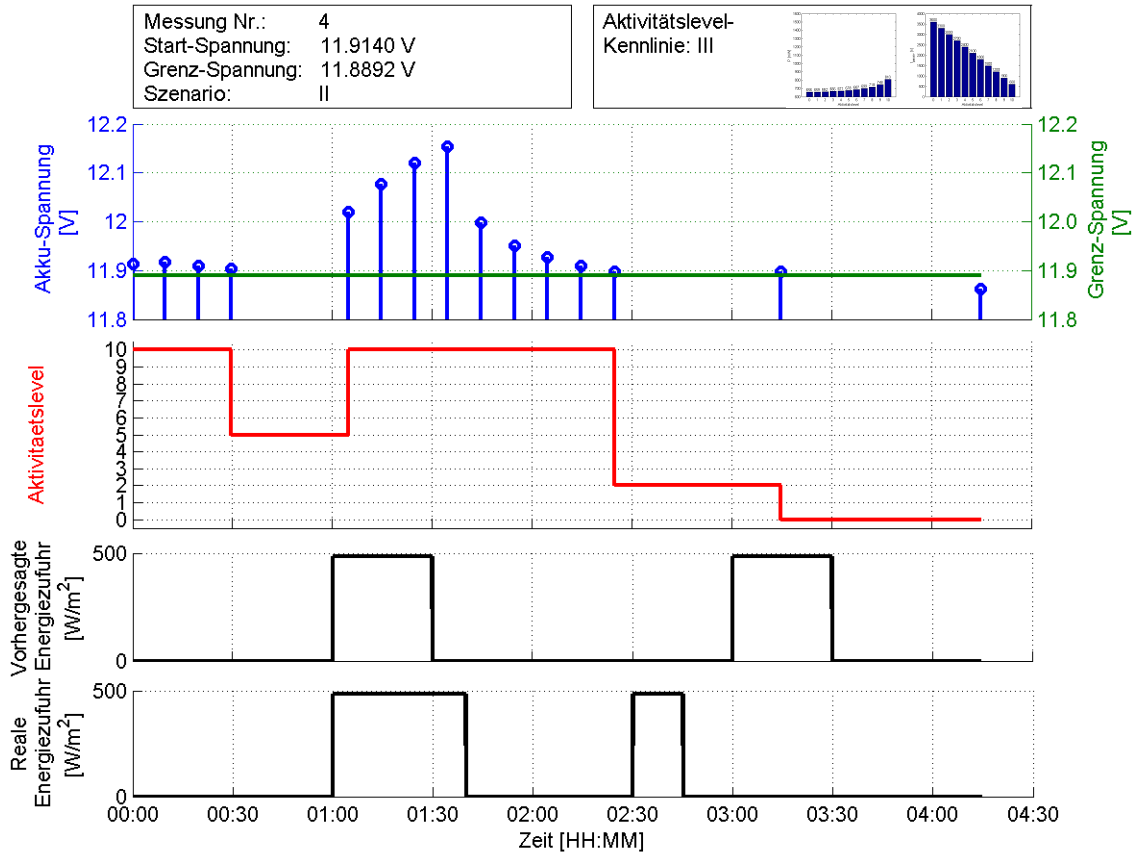

Abbildung 6.8.: Ergebnis der Messung 4

Die Aktivitätskennlinie III wird für die letzte Messung verwendet. Die Kennlinie hat, wie Kennlinie I, einen linearen Verlauf der Zykluszeit, verwendet aber eine wesentlich höhere minimale Zykluszeit von 10 Minuten. Für die Messung wird ebenfalls eine modifizierte reale Energiezufuhr verwendet. Die Abbildung 6.8 zeigt die zeitlichen Verläufe der aufgezeichneten Daten. Auch in dieser Messung wechselt der Algorithmus nach ca. 30 Minuten das Aktivitätslevel, es wird ein mittlerer Wert eingestellt. Während der ersten Aufladungsphase wird zurück auf das maximale Aktivitätslevel gewechselt, da die erhöhte Akku-Spannung erkannt wird. Das maximale Level wird auch nach der Aufladung weiter gehalten, bis es kurz vor der unerwarteten Aufladung auf einen niedrigen Wert eingestellt wird. Das System ist dann erst wieder zum Zeitpunkt der zweiten vorhergesagten Energiezufuhr aktiv und reagiert mit einem Wechsel auf Level 0. Innerhalb der Messperiode ist das System in dieser Messung 15 Mal aktiv.

6.2. Auswertung

6.2.1. Verhalten unter statischen Bedingungen

Das Verhalten unter statischen Bedingungen wird in Messung 1 untersucht. Es wird keine Energiezufuhr vorhergesagt und auch real keine Energie zugeführt. Mit Hilfe der Prädiktionsdaten für verschiedene Zeitpunkte wird die Wirkweise des Algorithmus verdeutlicht.

Die Messung ergibt, dass der Algorithmus vorausschauend das Aktivitätslevel frühzeitig reduziert, um am Ende der Messperiode die Grenzspannung nicht zu unterschreiten. Dabei wird das Aktivitätslevel früher reduziert als in der ersten Vorausberechnung ermittelt wurde, was mit der Verschiebung des Prädiktionshorizonts mit jedem aktiven Zyklus zusammenhängt. Es wird deutlich, dass die Optimierung für jeden aktiven Zeitpunkt neu durchgeführt wird, wobei sich die Berechnungsgrundlage und damit die Grundlage für die Auswahl eines Aktivitätslevels ändert.

Der Algorithmus arbeitet unter statischen Bedingungen sehr zuverlässig und die festgelegte Grenzspannung wird nicht unterschritten. Auffällig ist jedoch der große Sprung vom maximalen zum minimalen Aktivitätslevel.

6.2.2. Verhalten unter dynamischen Bedingungen

Das dynamische Verhalten des Energiemanagement-Algorithmus wird in den Messungen 2 bis 4 betrachtet. Es werden unterschiedliche Aktivitätslevel-Kennlinien für das gleiche dynamische Messszenario verwendet.

Auch in diesen Messungen kann die vorausschauende Reduktion des Aktivitätslevels beobachtet werden. Findet die vorhergesagte Energiezufuhr auch tatsächlich statt, werden die Level so gewählt, dass die Akku-Spannung nicht unter die Grenze fällt. Stimmt die Vorhersage jedoch nicht und wird insgesamt weniger Energie zugeführt, als vorhergesagt ist, so wählt der Algorithmus tendenziell zu hohe Aktivitätslevel. Dies führt dazu, dass am Ende der Messungen die Grenzspannung unterschritten wird. Es wurde eine deutliche Zwei-Punkt-Charakteristik festgestellt, die, je nach Länge der minimalen Zykluszeit, unterschiedlich stark ausgeprägt ist. Der Grund hierfür kann in der Implementierung des Algorithmus gefunden werden. Dieser wählt immer das maximal mögliche Aktivitätslevel, jedoch mit der Grenzspannung als Randbedingungen. So wählt der Algorithmus für energiereiche Perioden das maximale Aktivitätslevel, stellt dann aber fest, dass dies zu viel Energie verbraucht und schaltet zurück auf das minimale Aktivitätslevel, um noch bis zum Ende der Messung eine zulässige Akku-Spannung halten zu können.

Unter dynamischen Bedingungen ist das Verhalten des Algorithmus ebenfalls nachvoll-

ziehbar, doch können auch hier große Sprünge zwischen den Aktivitätsleveln beobachtet werden. Die Grenzspannung wird bei allen 3 Messungen am Ende unterschritten. Dies ist damit zu begründen, dass dem System weniger Energie als vorhergesagt zugeführt wurde.

6.2.3. Bewertung des Gesamtsystems

Die Messergebnisse zeigen, dass der entwickelte vorausschauende Energiemanagement-Algorithmus grundsätzlich funktioniert und den Energieverbrauch des eingebetteten Systems so steuert, dass der Akkumulator die festgelegte Spannungsgrenze weitestgehend nicht unterschreitet. Ist der Ladezustand des Akkus niedrig und wird eine längere Phase ohne Energiezufuhr erkannt, so wird das Aktivitätslevel reduziert, um Energie einzusparen. Sobald wieder ausreichend viel Energie im Akku zur Verfügung steht, wird das Aktivitätslevel erhöht. Das vorausschauende Verhalten des Algorithmus kann ebenfalls nachgewiesen werden, doch hängt dies stark von den gegebenen Umwelteinflüssen ab. Entscheidend für das Verhalten des Systems ist außerdem die verwendete Aktivitätslevel-Kennlinie.

Das System funktioniert sehr zuverlässig, wenn vorhergesagte und reale Energiezufuhr übereinstimmen, insbesondere wenn die Umweltbedingungen sich nicht ändern (s. Messung 1). Entsprechen die Vorhersage-Daten jedoch nicht der real zugeführten Energie so wählt das System zu hohe oder zu niedrige Aktivitätslevel, da die Optimierung direkt auf den Vorhersage-Daten basiert. Unerwartete Aufladungen werden vom System sehr schnell erkannt, sofern das Gerät in diesem Zeitraum einen aktiven Zyklus durchläuft. Das Aktivitätslevel wird in einem solchen Fall drastisch erhöht. Erfolgen jedoch Aufladungen, welche vorhergesagt waren, nicht, so wird dies nur indirekt festgestellt. Der Algorithmus plant dann hohe Aktivitätslevel, doch stellt fest, dass trotzdem immer weniger Energie zur Verfügung steht und ist daher gezwungen das Level zu verringern.

Die Ergebnisse der Messungen 2 bis 4 zeigen, dass die Aktivitätslevel-Kennlinie einen großen Einfluss auf das Verhalten des Systems hat. Allgemein können beim Wechsel der Aktivitätslevel sehr große Sprünge beobachtet werden, welche einer Zwei-Punkt-Steuerung zwischen maximalem und minimalem Level ähneln. Kürzere Zykluszeiten (Messung 3) führen dabei insgesamt zu häufigeren und größeren Sprüngen des Aktivitätslevels, aber auch zu einer hohen Messdichte. Sind die Zykluszeiten insgesamt länger (Messung 4), sind die Änderungen der Aktivitätslevel weniger stark ausgeprägt, doch die Grundtendenz zum Zwei-Punkt-Verhalten bleibt erhalten. Die großen Sprünge führen unter anderem auch dazu, dass das System auch zu Zeitpunkten inaktiv ist, zu denen eigentlich ausreichend viel Energie zur Verfügung steht, um eine Messung durchzuführen.

7. Zusammenfassung

Diese Arbeit zeigt, dass ein vorausschauendes Energiemanagement auf einem autonom mit Energie versorgten und eingebetteten System realisiert werden kann. Der implementierte Algorithmus steuert dazu dynamisch den Energieverbrauch des eingebetteten Systems in Abhängigkeit von Wettervorhersage-Daten und aktuellen Sensorwerten.

Das entwickelte Gesamtsystem arbeitet in Aktiv- und Passivphasen, welche zu unterschiedlichen Leistungsaufnahmen führen. Durch Änderung des Verhältnisses zwischen aktiver und passiver Zeit, wird die Leistungsaufnahme des Systems beeinflusst. Als Randbedingung für die dynamische Anpassung wird ein Spannungswert festgelegt, der eine Untergrenze für die Spannung des Akkumulators darstellt. Zusätzlich wird ein Prädiktionshorizont definiert, der festlegt, über welchen Zeitraum die vorausschauende Fähigkeit des Algorithmus wirkt. Eine Anwendungssoftware, die den Algorithmus implementiert, reguliert dann die Leistungsaufnahme, sodass das die Akkumulator-Spannung innerhalb des Prädiktionshorizonts nicht unter die festgelegte Schwelle fällt.

Unter statischen Umweltbedingungen reagiert das Energiemanagement-System zuverlässig und reduziert frühzeitig die Energieaufnahme, sodass die definierte Spannungsgrenze nicht unterschritten wird. Ist es jedoch dynamischen, von der Vorhersage abweichenden, Bedingungen ausgesetzt, sind die Reaktionen des Algorithmus tendenziell korrekt, führen aber unter Umständen zu unerwünschten Nebeneffekten. Insgesamt hat das Regelverhalten des Systems eine ausgeprägte „Zwei-Punkt“-Charakteristik. Dieses Verhalten wird durch kurze Zykluszeiten begünstigt. Die Definition der Aktivitätslevel stellt somit einen wichtigen Faktor bei der Auslegung des Systems dar. Dabei können sich die Kennlinien für verschiedene Anwendungszwecke unterschiedlich gut eignen.

Es konnte nachgewiesen werden, dass die entwickelte Software ein vorausschauendes Energiemanagement auf dem eingebetteten System umsetzt. Das grundlegende Verhalten des Algorithmus entspricht den Erwartungen, doch sollte für eine reale Applikation des Systems die Aktivitätslevel-Kennlinie sehr sorgfältig ausgelegt werden, da diese maßgeblich für das Gesamtverhalten des Systems verantwortlich ist.

8. Ausblick

8.1. Optimierung des Algorithmus

Das System reguliert in der aktuellen Umsetzung den Energieverbrauch über der Zeit ohne jegliche Einschränkungen. Für den realen Einsatz als Messstation können jedoch Randbedingungen gelten, die gewisse Aktivitätslevel zu bestimmten Uhrzeiten nicht zulassen (z. B. nachts). Außerdem könnte es notwendig sein, regelmäßig Messungen zu exakten Zeitpunkten durchzuführen. Der Algorithmus muss dies berücksichtigen, damit das Gerät zum gewünschten Zeitpunkt aktiv wird, obwohl das aktuelle Aktivitätslevel dies möglicherweise nicht vorsieht.

Modellierung des Systems

Die für die Implementierung gewählten Modelle der einzelnen Komponenten sind sehr einfach und stellen das reale Verhalten mit einer begrenzten Genauigkeit dar. Um die Vorausberechnungen und damit die Leistungsfähigkeit des Algorithmus zu verbessern, sollten exaktere Modelle verwendet werden. Es wurde bisher nur eine lineare Kennlinie für den Akkumulator verwendet, welche als Lade- und Entladekennlinie dient. Die realen Kennlinien für Aufladung und Entladung sind jedoch nicht linear und unterscheiden sich deutlich. Um den Ladezustand besser anhand der Akkuspannung zu ermitteln, könnte die Lade- und Entladekennlinie des verwendeten Akkumulators aufgezeichnet werden, um diese später für die Berechnung der Restkapazität zu verwenden. Auch für das Solarmodul sollte für unterschiedlich starke Sonneneinstrahlungen und unter realen Bedingungen eine Kennlinie erfasst werden.

Ein anderer Ansatz zur Verbesserung der Modellierung des Gesamtsystems wäre eine adaptive Auslegung der Modelle. Dazu müssten die vorausberechneten Werte eines Zustands mit den späteren realen Werten verglichen und anschließend die Parameter des Modell geändert werden. Das System würde so selbstständig während des Betriebs die Parameter der einzelnen Modelle immer weiter anpassen. Über einen längeren Zeitraum würden so Modelle vom System entwickelt werden, die das reale Verhalten des Systems sehr gut abbilden und eine genauere Prädiktion von zukünftigen Werten erlauben. Des Weiteren würden dann auch schleichende Prozesse, wie z. B. Alterungserscheinungen an Akkumulator und Solarmodul, berücksichtigt werden.

Gezielte Regelung der Aktivitätslevel

Wie die Messungen zeigen, sind die Wechsel zwischen den Aktivitätsleveln meist sehr groß, da der Algorithmus keine Begrenzung bei der Auswahl der Level verwendet. Um die dadurch entstehende „Zwei-Punkt"-Regelung zu verhindern, könnte der Wechsel der Level begrenzt werden. Noch vorteilhafter wäre die Implementierung einer echten Regelung der Aktivitätslevel. Durch entsprechende Einstellung der Regelparameter, welche auch adaptiv vom System angepasst werden könnten, könnte das Verhalten beim Wechsel der Aktivitätslevel beeinflusst werden.

8.2. Optimierung Software

Anwendungssoftware

Die implementierte Anwendungssoftware verwendet eine statische Wettervorhersage, welche in eine C-Header-Datei eingetragen ist. Diese statische Lösung ist für Testzwecke geeignet, weniger jedoch für einen späteren autonomen Einsatz des Systems. Die Anwendung könnte so erweitert werden, dass Wettervorhersagen für den exakten Standort der Messstation regelmäßig über das GSM-Modem eingeholt werden. Sind keine Vorhersage-Daten verfügbar, sollten Erfahrungswerte für die durchschnittliche Sonneneinstrahlung für den jeweiligen Monat im Jahr verwendet werden, um dennoch eine grundlegende Funktionalität zu gewährleisten. Solche Erfahrungswerte könnten auch durch die Anwendungssoftware selbst erstellt werden.

Die Anwendung sammelt während des Betriebs sehr viele Daten über die Umweltbedingungen und den aktuellen Systemzustand. Durch langfristige Analysen dieser Daten, können typische Verhaltensweisen des System erkannt und im Energiemanagement berücksichtigt werden. So könnte eine Jahres-Durchschnitts-Kurve für die Aktivitäten des Systems erstellt werden, durch die besonders kritische Zeiträume im Jahr bereits im Voraus erkannt und berücksichtigt werden können.

Betriebssystem

Um den Regelbereich für die Einstellung der Leistungsaufnahme zu erhöhen, muss der Unterschied zwischen Aktiv- und Passivphase so groß wie möglich sein. Das verwendete Betriebssystem unterstützt nur einen leichten Energiesparmodus. Das Linux-System stellt grundsätzlich auch effizientere Energiesparmodi zur Verfügung, doch fehlt die entsprechende Unterstützung für den verwendeten Prozessor. Durch Implementierung der Unterstützung von z. B. *Suspend-to-RAM* oder *Suspend-to-Disk* könnte der Energieverbrauch im Passiv-

modus stark verringert werden und damit der Regelbereich für die Leistungsaufnahme erhöht werden.

8.3. Einsatz optimierter Hardware

Bestimmung des Ladezustands des Akkumulators

Eine wichtige Aufgabe des System ist es, den Ladezustand des Akkumulators zu bestimmen. Es sind sehr genaue Werte notwendig, da der Akkumulator nur mit kleinen und langsamen Änderungen des Ladezustands reagiert. Die Spannungsmessung des Akkus könnte verbessert werden, indem ein eigens für diese Aufgabe zuständiger A/D-Wandler eingesetzt wird, welcher exakt auf den möglichen Spannungsbereich des Akkumulators abgestimmt ist. So würde die Auflösung des Wandlers optimal genutzt und das Ergebnis hätte eine sehr viel höhere Genauigkeit.

Die Ermittlung des Ladezustands über die Spannung des Akkus ist nicht sehr zuverlässig, da die Lade- und Entladekennlinien stark nichtlinear sind. Stattdessen könnte auch ein geeigneter Ladungszähler (*Coulomb-Zähler*) verwendet werden, um den Ladezustand zu bestimmen. Diese sind als Integrated Circuit (IC) verfügbar und können z. B. über eine serielle Schnittstelle mit dem eingebetteten System verbunden werden. Ein Coulomb-Zähler bestimmt den Ladezustand, indem die zugeführte und abgeführte Ladung gemessen wird. Mit dem Wissen über die Kapazität des Akkumulators kann so der Ladezustand berechnet werden. Der Vorteil dieser Methode ist, dass die Restenergie nicht durch einen Rückschluss über eine Kennlinie gezogen wird, sondern direkt die zugeführte und entnommene Ladungsmenge bestimmt wird.

Weckmechanismus für besondere Ereignisse

Um die Leistungsfähigkeit des Systems zu verbessern, muss verhindert werden, dass sich das System sich im passiven Modus befindet, obwohl gerade eine Aufladung stattfindet und genügend Energie zur Verfügung steht. Es könnte dazu zusätzlich eine Schaltung entwickelt werden, die das eingebettete System bei bestimmten Akku-Ladezuständen weckt, obwohl das aktuelle Aktivitätslevel dies nicht vorsieht. Ein Beispiel wäre ein Wecksignal beim Erreichen des Ladezustands von 100 % oder der zuvor festgelegten Spannungsgrenze. Durch den erzwungenen Aktivzyklus würde das System die Situation zu diesem Zeitpunkt neu bewerten und entsprechend reagieren können.

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACPI Advanced Configuration and Power Interface

CCM Clock Controller Module

DVFS Dynamic Voltage and Frequency Scaling

GPIO General Purpose Input Output

RTC Real Time Clock

ADC Analog-Digital-Converter

PMIC Power Management Integrated Circuit

IC Integrated Circuit

MPC Model based Predictive Control

UART Universal Asynchronous Receiver/Transmitter

I2C Inter-Integrated Circuit

IIO Industrial Input Output

SoC System On Chip

SSH Secure Shell

STC Standard Test Conditions

GSM Global System for Mobile Communication

VRLA Valve-Regulated Lead-Acid

Literaturverzeichnis

[1] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005.

[2] Horst Börner. *Pflanzenkrankheiten und Pflanzenschutz*. Springer, 2009.

[3] Christian W. Dawson. *Computerprojekte im Klartext*. Pearson Studium, 2003.

[4] Rainer Dittmar. *Modellbasierte prädiktive Regelung : eine Einführung für Ingenieure*. Oldenbourg, München, 2004.

[5] Prof. Dr. Dr. h.c. Günther Drosdowski, Dr. Werner Scholze-Stubenrecht, and Dr. Matthias Wermke, editors. *Das Fremdörterbuch*. Duden, 1997.

[6] eCos. Redboot: bootstrap environment for embedded systems. `https://sourceware.org/redboot/`. Zugriff: 13.04.2014.

[7] Freescale Conductor Inc. Freescale i.MX Linux Tree 3.10.17 beta. `http://git.freescale.com/git/cgit.cgi/imx/linux-2.6-imx.git/?h=imx_3.10.17_1.0.0_beta`. Zugriff: 08.01.2014.

[8] Freescale Conductor Inc. *i.MX35 (MCIMX35) Multimedia Applications Processor Reference Manual*. Rev. 3 edition, 2010.

[9] Garz & Fricke GmbH. LIVIUS: High-End M2M Hutschienen-Computer. `http://www.garz-fricke.com/livius_de.html`. Zugriff: 13.04.2014.

[10] Garz & Fricke GmbH. *Datenblatt Garz & Fricke LIVIUS Series*, 2010.

[11] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Der LaTeX -Begleiter*. Addison-Wesley, 2000.

[12] H-Tronic GmbH. *Anleitung Solarladeregler SL53 12 V / 4 A*.

[13] Intel Corporation. *Desktop 4th Generateion Intel Core Processor Family Datasheet*. Volume 1 edition, 2014.

[14] Koekraf GmbH. Spezifikation 6W Solarmodul 12V System. `http://www.koekraf.com/produkte/solar-wind-energie/duennschicht-solarmodule/6w-solarmodul-12v-system/spezifikationen/`. Zugriff: 03.04.2014.

[15] Otto Kruse. *Keine Angst vor dem leeren Blatt*. campus concret, 2000.

[16] Linux Foundation. Yocto Project. `https://www.yoctoproject.org/`. Zugriff: 13.04.2014.

[17] Konrad Mertens. *Photovoltaik*. Hanser Fachbuchverlag, 2013.

[18] Panasonic Corporation. *Datenblatt Panasonic LC-R127R2PG*.

[19] Pengutronix e.K. OSELAS® Services for Embedded Linux. `http://www.oselas.com/oselas/index_en.html`. Zugriff: 13.04.2014.

[20] Klaus Poenicke. *Wie verfaßt man wissenschaftliche Arbeiten?* Duden, 1988.

[21] J. A. Rossiter. *Model-based predictive control : a practical approach*. CRC Press, Boca Raton, 2003.

[22] Helmut Schellong. *Moderne C-Programmierung: Kompendium und Referenz (Xpert.press)*. Springer Vieweg, 2013.

[23] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux: Das Praxisbuch (X.systems.press)*. Springer, 2009.

[24] Siemens AG. *Desigo - Energieffiziente Applikationen: Prädiktiver und selbstadaptierender Heizungsregler (Applikationsdatenblatt)*. 2011.

[25] Joachim Wietzke. *Embedded Technologies: Vom Treiber bis zur Grafik-Anbindung (Xpert.press)*. Springer, 2012.

[26] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. O'Reilly Media, 2008.

[27] ZF Friedrichshafen AG. *Schaltstrategie mit Weitblick - PreVision GPS für TraXon-Nutzfahrzeuggetriebe von ZF (Presseinformation)*. 2012.

A. Anhang

A.1. CD

- **Bachelorthesis**
  - Bachelorthesis im PDF-Format
- **Energiemanagement Applikation**
  - Vollständiger Quellcode der Energiemanagement-Applikation „pman“
  - „Makefile“ zum Übersetzen der Applikation

A.2. Quellcode: Energiemanagement-Applikation

pman.c

```
/*****************************************************************************/
/*                                                                           */
/* pman - power management for Garz & Fricke LIVIUS                          */
/*                                                                           */
/* This program is free software; you can redistribute it and/or modify it   */
/* under the terms of the GNU General Public License version 2 as published  */
/* by the Free Software Foundation.                                          */
/*                                                                           */
/* This power management tool is meant to run on Garz & Fricke LIVIUS.       */
/* The tool provides an adaptive and predictive power management based on    */
/* weather forecast data.                                                    */
/*                                                                           */
/* Author:      Richard Guenther <richard.guenther@haw-hamburg.de            */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <syslog.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <float.h>
#include <sys/stat.h>
#include <getopt.h>

#include "pman.h"
#include "sysfs_utils.h"
#include "weather.h"

#define _GNU_SOURCE /* Must be defined to enable getopt_long functinality. */

/* default buffer sizes */
#define MAX_PATH_LENGTH     75
#define MAX_MSG_LENGTH      75

/* GPIOs */
#define RUN_VBAK5V      (1 * 32 + 11)
#define CHARGE_STAT1    (0 * 32 + 8)
#define CHARGE_DONE     (0 * 32 + 9)
#define CHARGE_DISABLE    (1 * 32 + 30)
#define VCC_CAN_EN      (2 * 32 + 2)
#define GSM_EN        (2 * 32 + 4)
#define VBATT_EN      (2 * 32 + 3)
#define VBAK5V_EN     (1 * 32 + 17)
#define KB_LED_EN     (1 * 32 + 10)

/* ADC channels */
#define ADC_CHANNEL_VIN       1
#define ADC_CHANNEL_BATTERY     5
#define ADC_CHANNEL_VIN_CURRENT   6
#define ADC_CHANNEL_RTC_BATTERY   7

/* battery data */
```

```
56  #define BAT_VOLTAGE_MAX     13.8  /* [V] */
57  #define BAT_VOLTAGE_MIN     10.5  /* [V] */
58  #define BAT_CAPACITY      7200  /* [Ah] */
59  #define BAT_ENERGY_MAX      86400 /* [Wh] */
60  #define BAT_PERCENT_BASE    (BAT_VOLTAGE_MAX - BAT_VOLTAGE_MIN) / 100
61
62  /* solar module data */
63  #define SOLAR_POWER_RATING    17.5    /* [V] */
64  #define SOLAR_SURFACE_AREA    0.146475  /* [m^2] */
65  #define SOLAR_MUE       0.04096
66
67  /* prediction defines */
68  #define PREDICTION_HORIZON    14400 /* [sec] */
69  #define LEVEL_CHANGE_THRESHOLD  42.4  /* [%] */
70
71  /* Log file paths */
72  #define LOG_PATH_STATES     "/home/root/current_state"
73  #define LOG_PATH_MEASUREMENT  "/home/root/measurements"
74  #define LOG_PATH_PREDICTION   "/home/root/prediction"
75
76  /* defines and function for printing messages */
77  #define V_INFO  3
78  #define V_DEBUG 2
79  #define V_WARN  1
80  #define V_ERROR 0
81
82  /* print function for verbose messages */
83  #define PRINT_INFO(verbosity_level, fmt, args...) do {    \
84    if (verbosity >= verbosity_level)         \
85      fprintf(stderr, "%s: " fmt, __progname, ## args);   \
86  } while (0)
87
88  /* print function for error messages */
89  #define PRINT_ERROR(errno, fmt, args...) \
90    fprintf(stderr, "%s: ERROR %d: %s: " fmt, __progname, errno, __func__, ## args);
91
92  /*****************************************************************************/
93  /*                                                                           */
94  /* global variables                                                          */
95  /*                                                                           */
96  /*****************************************************************************/
97  int verbosity = 1;      /* warnings enabled by default */
98  int logging = 0;      /* logging disabled by default */
99
100 /*****************************************************************************/
101 /*                                                                           */
102 /* struct definitions                                                        */
103 /*                                                                           */
104 /*****************************************************************************/
105 struct system_state {
106   struct power_lvl *    plevel;       /* power level */
107   time_t          time;       /* current system time (UNIX format) */
108   float         bat_energy;     /* available energy in main battery [Wh] */
109   float         bat_percent;    /* main battery state of charge [%] */
110   float         v_backup_bat;   /* backup battery voltage [V] */
111   float         v_main_bat;     /* main battery voltage [V] */
112   float         i_main_bat;     /* main battery current [A] */
113
114   struct system_state *   next;       /* pointer to the next system state */
115   struct system_state *   prev;       /* pointer to the previous system state */
```

```
116 };
117
118 struct power_lvl {
119   int   level;        /* level */
120   int   active_interval;  /* activity cycle interval */
121   int   power_consumption;  /* power consumption */
122 };
123
124 /*****************************************************************************/
125 /*                                                                           */
126 /* enumerations                                                              */
127 /*                                                                           */
128 /*****************************************************************************/
129
130 /* available power levels */
131 enum powerlevels {P0, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10};
132
133 /*****************************************************************************/
134 /*                                                                           */
135 /* tables                                                                    */
136 /*                                                                           */
137 /*****************************************************************************/
138 /*
139  * ----------------------------------------
140  * linear sleep time characteristic
141  * step: 342 s
142  * ----------------------------------------
143  */
144 struct power_lvl power_level_table1[] = {
145 /*  | Level | sleep | power |
146  *  |       | [sec] | [mW]  |*/
147   { P0,     3600,   671 },
148   { P1,     3258,   676 },
149   { P2,     2916,   682 },
150   { P3,     2574,   689 },
151   { P4,     2232,   699 },
152   { P5,     1890,   716 },
153   { P6,     1548,   732 },
154   { P7,     1206,   762 },
155   { P8,     864,    817 },
156   { P9,     522,    943 },
157   { P10,    180,    1550 },
158 };
159
160 /*
161  * ------------------------------------------------
162  * mostly linear power consumption characteristic:
163  * step: 0.085 mW
164  * ------------------------------------------------
165  */
166 struct power_lvl power_level_table2[] = {
167 /*  | Level | sleep | power |
168  *  |       | [sec] | [mW]  |*/
169   { P0,     3600  ,  656  },
170   { P1,     1449  ,  702  },
171   { P2,     813   ,  762  },
172   { P3,     565   ,  822  },
173   { P4,     480   ,  856  },
174   { P5,     420   ,  889  },
175   { P6,     360   ,  934  },
```

```
176   { P7,     300   ,  995  },
177   { P8,     240   ,  1088 },
178   { P9,     180   ,  1242 },
179   { P10,    120   ,  1550 },
180 };
181
182 /*
183  * ------------------------------------------------
184  * lower power consumption, linear sleep time characteristic
185  * step: 300 s
186  * ------------------------------------------------
187  */
188 struct power_lvl power_level_table3[] = {
189 /*  | Level | sleep | power |
190  *  |       | [sec] | [mW]  |*/
191   { P0,     3600,   656   },
192   { P1,     3300,   659   },
193   { P2,     3000,   662   },
194   { P3,     2700,   666   },
195   { P4,     2400,   671   },
196   { P5,     2100,   678   },
197   { P6,     1800,   687   },
198   { P7,     1500,   699   },
199   { P8,     1200,   718   },
200   { P9,     900,    748   },
201   { P10,    600,    810   },
202 };
203
204 /*****************************************************************************/
205 /*                                                                           */
206 /* Function:    get_next_active_time                                         */
207 /*                                                                           */
208 /* Description: Returns the next active time that is scheduled for the given  */
209 /*              system state.                                                */
210 /*                                                                           */
211 /* Input:       @state: pointer to the system state                          */
212 /*                                                                           */
213 /* Return:      @next_active_time: time value of next activity (UNIX format)  */
214 /*                                                                           */
215 /*****************************************************************************/
216 time_t get_next_active_time(struct system_state *state)
217 {
218   int i;
219   time_t next_active_time;
220
221   if(state == NULL) {
222     PRINT_ERROR(EINVAL, "invalid pointer to system state\n");
223     return -EINVAL;
224   }
225
226   /* get next active time from power level */
227   next_active_time = (state->time + state->plevel->active_interval);
228
229   PRINT_INFO(V_INFO, "next active time will be: %s", ctime(&next_active_time));
230
231   return next_active_time;
232 }
233
234 /*****************************************************************************/
235 /*                                                                           */
```

```
236 /* Function:    log_to_file                                                 */
237 /*                                                                          */
238 /* Description: writes a message to a file at the given path with a time    */
239 /*              stamp.                                                      */
240 /*                                                                          */
241 /* Input:       * message to write                                          */
242 /*              * path to the file                                          */
243 /*                                                                          */
244 /* Return:      On failure - negative errno return code                     */
245 /*              On success - '0' return code.                               */
246 /*                                                                          */
247 /****************************************************************************/
248 static int log_to_file(char *file_path, char *msg)
249 {
250   FILE *log_file;
251   time_t raw_time;
252   struct tm *ptm;
253 
254   char *log_msg = malloc(MAX_MSG_LENGTH + strlen(msg));
255   if (log_msg == NULL) {
256     PRINT_ERROR(ENOMEM, "log message memory allocation failed");
257     return -ENOMEM;
258   }
259 
260   time(&raw_time);
261   ptm = localtime(&raw_time);
262 
263   /* generate timestamp */
264   snprintf(log_msg, MAX_MSG_LENGTH, "[%.2d/%.2d/%d %.2d:%.2d:%.2d] ",
265            ptm->tm_mday, (ptm->tm_mon)+1, (ptm->tm_year)+1900,
266            ptm->tm_hour, ptm->tm_min, ptm->tm_sec);
267 
268   /* concatenate log message */
269   strcat(log_msg, msg);
270 
271   /* write message to logfile */
272   log_file = fopen(file_path, "ab+");
273   if(!log_file)
274     return -EIO;
275   fprintf(log_file, log_msg);
276   fclose(log_file);
277 
278   return 0;
279 }
280 
281 /****************************************************************************/
282 /*                                                                          */
283 /* Function:    log_system_state                                            */
284 /*                                                                          */
285 /* Description: writes data of the given system state to the specified log  */
286 /*              file path.                                                  */
287 /*                                                                          */
288 /* Input:       @file_path: path to the location of the log file            */
289 /*              @state    : pointer to the system state to log              */
290 /*                                                                          */
291 /* Return:      On failure - negative errno return code                     */
292 /*              On success - '0' return code.                               */
293 /*                                                                          */
294 /****************************************************************************/
295 static int log_system_state(char *file_path, struct system_state *state)
```

```
296 {
297   int ret;
298   time_t system_time, next_active_time;
299 
300   char *log_msg = malloc(MAX_MSG_LENGTH);
301   if (log_msg == NULL) {
302     PRINT_ERROR(ENOMEM, "log message memory allocation failed");
303     return -ENOMEM;
304   }
305 
306   system_time = state->time;
307   next_active_time = get_next_active_time(state);
308 
309   ret = log_to_file(file_path, "-------------------------------------------------\n");
310   /* system time */
311     snprintf(log_msg, MAX_MSG_LENGTH,  "system time: %s", ctime(&system_time));
312   ret = log_to_file(file_path, log_msg);
313 
314   /* power level */
315     snprintf(log_msg, MAX_MSG_LENGTH,  "power level: %d\n", state->plevel->level);
316   ret = log_to_file(file_path, log_msg);
317     snprintf(log_msg, MAX_MSG_LENGTH,  "  power consumption: %d\n", state->plevel->
          power_consumption);
318   ret = log_to_file(file_path, log_msg);
319     snprintf(log_msg, MAX_MSG_LENGTH,  "  active interval: %d\n", state->plevel->
          active_interval);
320   ret = log_to_file(file_path, log_msg);
321     snprintf(log_msg, MAX_MSG_LENGTH,  "  next active time: %s", ctime(&next_active_time));
322   ret = log_to_file(file_path, log_msg);
323 
324   /* main battery */
325     snprintf(log_msg, MAX_MSG_LENGTH,  "main battery percent: %.4f%%\n", state->bat_percent
          );
326   ret = log_to_file(file_path, log_msg);
327     snprintf(log_msg, MAX_MSG_LENGTH,  "main battery energy: %.4f mWh\n", state->bat_energy
          );
328   ret = log_to_file(file_path, log_msg);
329     snprintf(log_msg, MAX_MSG_LENGTH,  "main battery voltage: %.4f V\n", state->v_main_bat)
          ;
330   ret = log_to_file(file_path, log_msg);
331     snprintf(log_msg, MAX_MSG_LENGTH,  "main battery current: %.4f mA\n", (state->
          i_main_bat)*1000);
332   ret = log_to_file(file_path, log_msg);
333 
334   /* backup battery */
335   snprintf(log_msg, MAX_MSG_LENGTH,  "backup battery voltage: %.4f V\n", state->
        v_backup_bat);
336   ret = log_to_file(file_path, log_msg);
337   ret = log_to_file(file_path, "-------------------------------------------------\n");
338 
339   free(log_msg);
340   return ret;
341 }
342 
343 /*****************************************************************************/
344 /*                                                                           */
345 /* Function:    log_system_state_list                                        */
346 /*                                                                           */
347 /* Description: prints out the time of all list elements                     */
348 /*                                                                           */
```

```
349 /* Input:       @head: pointer to the head of the list                      */
350 /*                                                                          */
351 /* Return:      On failure - negative errno return code                     */
352 /*              On success - '0' return code.                               */
353 /*                                                                          */
354 /****************************************************************************/
355 int log_system_state_list(char *file_path, struct system_state *head) {
356     struct system_state *current;
357   time_t time;
358 
359   if(head == NULL) {
360     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
361     return -EINVAL;
362   }
363 
364   current = head;
365 
366     while (current != NULL) {
367     log_system_state(file_path, current);
368         current = current->next;
369     }
370 }
371 
372 /****************************************************************************/
373 /*                                                                          */
374 /* Function:    get_current_system_state                                    */
375 /*                                                                          */
376 /* Description: update values of the given system state by reading supply   */
377 /*              voltages and GPIOs                                          */
378 /*                                                                          */
379 /* Input:       @state: pointer to the system state to update               */
380 /*                                                                          */
381 /* Return:      On failure - negative errno return code                     */
382 /*              On success - '0' return code.                               */
383 /*                                                                          */
384 /****************************************************************************/
385 static int get_current_system_state(struct system_state *state)
386 {
387   float v_backup_bat, v_main_bat, i_main_bat, bat_percent, bat_energy;
388   float v_bat_1, v_bat_2;
389 
390   if(state == NULL) {
391     PRINT_ERROR(EINVAL, "invalid pointer to system state\n");
392     return -EINVAL;
393   }
394 
395   /* read ADC values */
396   v_backup_bat = sysfs_adc_read_channel(ADC_CHANNEL_BATTERY);
397   i_main_bat = sysfs_adc_read_channel(ADC_CHANNEL_VIN_CURRENT);
398 
399 
400   v_bat_1 = sysfs_adc_read_channel(ADC_CHANNEL_VIN);
401   sleep(1);
402   v_bat_2 = sysfs_adc_read_channel(ADC_CHANNEL_VIN);
403 
404   v_main_bat = (v_bat_1 + v_bat_2) / 2;
405 
406   /* calculate voltage/current (due to hardware) */
407   v_main_bat = v_main_bat * 1.386;
408   i_main_bat = i_main_bat / (0.022*100);
```

```
409
410   /* calculate percent and energy values */
411   bat_percent = ((v_main_bat - BAT_VOLTAGE_MIN) / (BAT_VOLTAGE_MAX - BAT_VOLTAGE_MIN)) *
          100;
412   bat_energy = BAT_ENERGY_MAX * (bat_percent / 100);
413
414   /* write new values*/
415   state->v_backup_bat = v_backup_bat;
416   state->v_main_bat = v_main_bat;
417   state->i_main_bat = i_main_bat;
418   state->bat_percent = bat_percent;
419   state->bat_energy = bat_energy;
420
421   return 0;
422 }
423
424 /*****************************************************************************/
425 /*                                                                           */
426 /* Function:    freeze_system                                                */
427 /*                                                                           */
428 /* Description: sets the device into "freeze" mode                           */
429 /*                                                                           */
430 /*****************************************************************************/
431 int freeze_system(void)
432 {
433   return system("echo freeze > /sys/power/state");
434 }
435
436 /*****************************************************************************/
437 /*                                                                           */
438 /* Function:    do_active_work                                               */
439 /*                                                                           */
440 /* Description: simulates active work by generating CPU load for 85 seconds  */
441 /*                                                                           */
442 /*****************************************************************************/
443 int do_active_work(void)
444 {
445   system("dd if=/dev/zero of=/dev/null &");
446   sleep(56);
447   system("killall dd");
448 }
449
450 /*****************************************************************************/
451 /*                                                                           */
452 /* Function:    set_power_level                                              */
453 /*                                                                           */
454 /* Description: writes the given power level to the given system state.      */
455 /*                                                                           */
456 /* Input:       @state: pointer to the system state to update                */
457 /*              @level: power level to write                                 */
458 /*                                                                           */
459 /* Return:      On failure - negative errno return code                      */
460 /*              On success - '0' return code.                                */
461 /*                                                                           */
462 /*****************************************************************************/
463 int set_power_level(struct system_state *state, int level)
464 {
465   if(state == NULL) {
466     PRINT_ERROR(EINVAL, "invalid pointer to system state\n");
467     return -EINVAL;
```

```
468   }
469
470   /* do not exceed array borders */
471   if (level > P10) {
472     level = P10;
473   }
474   else if (level < P0) {
475     level = P0;
476   }
477
478   PRINT_INFO(V_INFO, "setting power level to %d\n", level);
479
480   //state->plevel = &power_level_table1[level];
481   //state->plevel = &power_level_table2[level];
482   state->plevel = &power_level_table3[level];
483 }
484
485 /*****************************************************************************/
486 /*                                                                           */
487 /* Function:    gsm_startup                                                  */
488 /*                                                                           */
489 /* Description: boot the GSM modem by pulling the ignition pin to high for   */
490 /*              5 seconds                                                    */
491 /*                                                                           */
492 /* Input:       None                                                         */
493 /*                                                                           */
494 /* Return:      On failure - negative errno return code                      */
495 /*              On success - '0' return code.                                */
496 /*                                                                           */
497 /*****************************************************************************/
498 int gsm_startup(void)
499 {
500   int ret;
501
502   ret = sysfs_gpio_write_value(GSM_EN, 1);
503   sleep(5);
504   ret = sysfs_gpio_write_value(GSM_EN, 0);
505
506   return ret;
507 }
508
509 /*****************************************************************************/
510 /*                                                                           */
511 /* Function:    toggle_active_mode                                           */
512 /*                                                                           */
513 /* Description: apply a power level to the device. This will mainly toggle   */
514 /*              peripheral devices to reduce or gain power consumption       */
515 /*                                                                           */
516 /* Input:       @toggle: switch active mode on or off                        */
517 /*                                                                           */
518 /* Return:      On failure - negative errno return code                      */
519 /*              On success - '0' return code.                                */
520 /*                                                                           */
521 /*****************************************************************************/
522 int toggle_active_mode(int toggle)
523 {
524   int ret;
525
526   /* peripheral control switches */
527   ret = sysfs_gpio_write_value(CHARGE_DISABLE, !toggle);
```

```
528   ret = sysfs_gpio_write_value(VCC_CAN_EN, toggle);
529   ret = sysfs_gpio_write_value(VBATT_EN, toggle);
530
531   if(toggle) {
532     ret = gsm_startup();
533   }
534
535   return ret;
536 }
537
538 /*****************************************************************************/
539 /*                                                                           */
540 /* Function:    predict_next_system_state                                    */
541 /*                                                                           */
542 /* Description: Predicts values for the next system state.                   */
543 /*              A model of the solar panel and the main battery are used to  */
544 /*              calculate expected battery and main supply values for the    */
545 /*              next active time                                             */
546 /*                                                                           */
547 /* Input:       @current_state: pointer to the system state to start from    */
548 /*              @weather_data : pointer to the weather forecast data array   */
549 /*                                                                           */
550 /* Return:      On failure - negative errno return code                      */
551 /*              On success - '0' return code.                                */
552 /*                                                                           */
553 /*****************************************************************************/
554 int predict_next_system_state(struct system_state *current_state, struct weather_entry *
        weather_data)
555 {
556   struct system_state *next_state;
557   struct weather_entry *w_entry;
558
559   float bat_energy_charge = 0, bat_energy_discharge = 0;
560   float solar_power, time_interval;
561   float next_vbat, next_bat_energy, next_bat_percent, next_ibat;
562   int harvest_start_time, harvest_end_time;
563   int i;
564
565   if(current_state == NULL) {
566     PRINT_ERROR(EINVAL, "invalid pointers to system state\n");
567     return -EINVAL;
568   }
569
570   if(current_state->next == NULL) {
571     PRINT_ERROR(EINVAL, "invalid pointers to next system state (list corrupted)\n");
572     return -EINVAL;
573   }
574
575   if(weather_data == NULL) {
576     PRINT_ERROR(EINVAL, "invalid pointers to weather data array\n");
577     return -EINVAL;
578   }
579
580   next_state = current_state->next;
581   w_entry = weather_data;
582
583   /* update time for future system state */
584   next_state->time = get_next_active_time(current_state);
585
586   /*
```

```
587     * calculate energy harvest from solar panel:
588     * ========================================
589     */
590   next_bat_energy = current_state->bat_energy;
591   next_bat_percent = current_state->bat_percent;
592   next_vbat = current_state->v_backup_bat;
593
594   /* start and end time for energy harvesting */
595   harvest_start_time = current_state->time;
596   harvest_end_time = next_state->time - 1;
597
598   /* parse weather data array */
599   while((w_entry+1)->time != 0) {
600
601     /* interval is negative: exit */
602     if(w_entry->time > harvest_end_time) {
603       break;
604     }
605     /* interval is between two data entries */
606     else if ((harvest_start_time > w_entry->time) && (harvest_end_time < (w_entry+1)->time)
          ) {
607       time_interval = (float)(harvest_end_time - harvest_start_time);
608     }
609     /* complete data entry within the interval */
610     else if ((harvest_start_time == w_entry->time) && (harvest_end_time >= (w_entry+1)->
          time)) {
611       time_interval = (float)((w_entry+1)->time - w_entry->time);
612       harvest_start_time = (w_entry+1)->time;
613     }
614     /* interval: start time -> next data entry time */
615     else if((harvest_start_time > w_entry->time) && (harvest_start_time < (w_entry+1)->time
          ) && (harvest_end_time >= (w_entry+1)->time)) {
616       time_interval = (float)((w_entry+1)->time - harvest_start_time);
617       harvest_start_time = (w_entry+1)->time;
618     }
619     /* interval: last data entry time -> end time */
620     else if((harvest_start_time == w_entry->time) && (harvest_end_time < (w_entry+1)->time)
          ) {
621       time_interval = (float)(harvest_end_time - (w_entry->time));
622     }
623     /* data entry is not within the interval */
624     else {
625       if((w_entry+1)->time > harvest_start_time)
626         harvest_start_time = (w_entry+1)->time;
627       w_entry++;
628       continue;
629     }
630
631     /* calculate solar supply power */
632     solar_power = ((float)(w_entry->condition) * SOLAR_SURFACE_AREA) * SOLAR_MUE;
633
634     /* determine charging energy (integration) */
635     bat_energy_charge += (solar_power * (time_interval / 3600));
636
637     /* move to the next data entry */
638     w_entry++;
639   }
640
641   /*
642    * calculate changes on battery charge
```

```
643     * ====================================
644     */
645   bat_energy_discharge = ((float)(current_state->plevel->power_consumption * current_state
          ->plevel->active_interval)) / 3600;
646
647   PRINT_INFO(V_INFO, " total charge energy    : %.2f mWh\n", bat_energy_charge);
648   PRINT_INFO(V_INFO, " total discharge energy : %.2f mWh\n", bat_energy_discharge);
649   PRINT_INFO(V_INFO, " energy balance         : %.2f mWh\n", bat_energy_charge -
          bat_energy_discharge);
650
651   /* calculate values for the next state */
652   next_bat_energy += (float)(bat_energy_charge - bat_energy_discharge);
653
654   /* battery cannot carry more than max. capacity */
655   if(next_bat_energy > BAT_ENERGY_MAX)
656     next_bat_energy = BAT_ENERGY_MAX;
657
658   next_bat_percent = (next_bat_energy / BAT_ENERGY_MAX) * 100;
659   next_vbat = next_bat_percent * ((BAT_VOLTAGE_MAX - BAT_VOLTAGE_MIN) / 100) +
          BAT_VOLTAGE_MIN;
660
661   /* write new system state values */
662   next_state->bat_energy = next_bat_energy;
663   next_state->bat_percent = next_bat_percent;
664   next_state->v_main_bat = next_vbat;
665
666   return 0;
667 }
668
669 /*****************************************************************************/
670 /*                                                                           */
671 /* Function:    predict_multiple_system_states                               */
672 /*                                                                           */
673 /* Description: Predicts the future system state for the given time. For this */
674 /*              all other intermediate states between the start point and the */
675 /*              specified time need to be predicted                          */
676 /*                                                                           */
677 /* Input:       @start_state    : pointer to the system state to start from   */
678 /*              @prediction_time: time to predict the system state for       */
679 /*              @weather        : array with weather forecast data           */
680 /*                                                                           */
681 /* Return:      On failure - negative errno return code                      */
682 /*              On success - '0' return code.                                */
683 /*                                                                           */
684 /*****************************************************************************/
685 int predict_multiple_system_states(struct system_state *start_state, time_t prediction_time
        , struct weather_entry *weather_data)
686 {
687   struct system_state *current_state, *next_state;
688   int next_active_time;
689   time_t time_predicted;
690   int i, j;
691
692   if(start_state == NULL) {
693     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
694     return -EINVAL;
695   }
696
697   if(prediction_time <= 0) {
698     PRINT_ERROR(EINVAL, "invalid prediction horizon - must be positive integer\n");
```

```
699     return -EINVAL;
700   }
701
702   current_state = start_state;
703
704   /*
705    * This loop predicts future system states until reaching <prediction_time>
706    */
707   while(prediction_time > time_predicted) {
708
709     /* insert new element if list is too short */
710     if(current_state->next == NULL) {
711       add_list_element(current_state);
712     }
713
714     /* catch corrupted list */
715     if(current_state == NULL || current_state->next == NULL) {
716       PRINT_ERROR(EINVAL, "could not complete prediction. system state list ended
              unexpected\n");
717       return -EINVAL;
718     }
719
720     /* predict future system states */
721     predict_next_system_state(current_state, weather_data);
722     time_predicted = current_state->next->time;
723
724     /* move pointer to next list element */
725     current_state = current_state->next;
726   }
727
728   return 0;
729 }
730
731 /*****************************************************************************/
732 /*                                                                           */
733 /* Function:    pop_list_element                                             */
734 /*                                                                           */
735 /* Description: Removes the first element of the list and moves the head     */
736 /*              pointer to the next element                                  */
737 /*                                                                           */
738 /* Input:       @head: pointer to pointer to the head of the list            */
739 /*                                                                           */
740 /* Return:      On failure - negative errno return code                      */
741 /*              On success - '0' return code.                                */
742 /*                                                                           */
743 /*****************************************************************************/
744 int pop_list_element(struct system_state **head)
745 {
746     struct system_state *next_element = NULL;
747
748     if(*head == NULL) {
749     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
750     return -EINVAL;
751   }
752
753     next_element = (*head)->next;
754     free(*head);
755     *head = next_element;
756
757     return 0;
```

```
758 }
759
760 /*****************************************************************************/
761 /*                                                                           */
762 /* Function:    delete_list                                                  */
763 /*                                                                           */
764 /* Description: deletes the whole list                                       */
765 /*                                                                           */
766 /* Input:       @head: pointer to the head of the list                       */
767 /*                                                                           */
768 /* Return:      On failure - negative errno return code                      */
769 /*              On success - '0' return code.                                */
770 /*                                                                           */
771 /*****************************************************************************/
772 int delete_list(struct system_state **head)
773 {
774     struct system_state *next, *deleteMe;
775
776   if(*head == NULL) {
777     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
778     return -EINVAL;
779   }
780
781   deleteMe = *head;
782
783   while (deleteMe) {
784         next = deleteMe->next;
785         free(deleteMe);
786         deleteMe = next;
787     }
788     *head = NULL;
789
790   return 0;
791 }
792
793 /*****************************************************************************/
794 /*                                                                           */
795 /* Function:    add_list_element                                             */
796 /*                                                                           */
797 /* Description: Adds an element at the bottom of a doubly linked list        */
798 /*                                                                           */
799 /* Input:       @head: pointer to the head of the list                       */
800 /*                                                                           */
801 /* Return:      On failure - negative errno return code                      */
802 /*              On success - '0' return code.                                */
803 /*                                                                           */
804 /*****************************************************************************/
805 int add_list_element(struct system_state *head)
806 {
807   struct system_state *newState = malloc(sizeof(struct system_state));
808   struct system_state *current  = head;
809
810   if (newState == NULL) {
811     PRINT_ERROR(ENOMEM, "list element memory allocation failed");
812     return -ENOMEM;
813   }
814
815   if(current == NULL) {
816     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
817     return -EINVAL;
```

```
818   }
819
820   /* first insertion */
821   if(head->next == NULL) {
822     head->next = newState;
823     newState->prev = head;
824   }
825   /* find last element and insert the new one */
826   else {
827         while (1) {
828       if(current->next == NULL)
829       {
830         current->next = newState;
831         newState->prev = current;
832         break;
833       }
834             current = current->next;
835         };
836     }
837
838   /* pre set values */
839   set_power_level(newState, current->plevel->level);
840   newState->time = get_next_active_time(newState->prev);
841   newState->bat_energy = 0;
842   newState->bat_percent = 0;
843   newState->i_main_bat = 0;
844   newState->v_backup_bat = 0;
845   newState->next = NULL;
846
847     return 0;
848 }
849
850 /*****************************************************************************/
851 /*                                                                           */
852 /* Function:    optimize_system_state_predicted                              */
853 /*                                                                           */
854 /* Description: This function optimizes the power levels of a list of system  */
855 /*              states by looking at the next system states.                 */
856 /*                                                                           */
857 /* Input:       @head: Pointer to the head of the list of system states      */
858 /*                                                                           */
859 /* Return:      On failure - negative errno return code                      */
860 /*              On success - '0' return code if the plevel was not changed   */
861 /*              On success - '1' return code if the plevel was changed       */
862 /*                                                                           */
863 /*****************************************************************************/
864 int optimize_system_state_predicted(struct system_state *current_state, time_t
        prediction_time, struct weather_entry *weather_data)
865 {
866   struct system_state *current;
867   int finished = 0, optimized = 0;
868
869   if(current_state == NULL) {
870     PRINT_ERROR(EINVAL, "invalid pointer to list head\n");
871     return -EINVAL;
872   }
873
874   predict_multiple_system_states(current_state, prediction_time, weather_data);
875
876   while(finished == 0) {
```

```
877
878     optimized = 0;
879     /* generate future system state list */
880     predict_multiple_system_states(current_state, prediction_time, weather_data);
881
882     current = current_state;
883
884     /* check for system state to optimize */
885     while(current->next != NULL) {
886
887       optimized = optimize_power_level_recursive(current, 0);
888
889       /* optimization error */
890       if (optimized < 0) {
891         PRINT_INFO(V_WARN, "an error occurred while optimizing\n");
892         finished = 1;
893         break;
894       }
895       /* optimization successful */
896       else if(optimized == 1) {
897         delete_list(&(current->next));
898         break;
899       }
900       /* optimization reaches current state: cannot optimize any more */
901       else if (optimized == 2) {
902         PRINT_INFO(V_WARN, "opimization reached current system state\n");
903         if(current->plevel->level == 0 || current->plevel->level == 7)
904           finished = 1;
905         break;
906       }
907       /* no need to optimize this state - switch to the next */
908       current = current->next;
909     }
910
911     /* no state was optimized within the prediction horizon - we are done */
912     if(optimized == 0) {
913       finished = 1;
914     }
915   }
916
917   return 0;
918 }
919
920 /*****************************************************************************/
921 /*                                                                           */
922 /* Function:    optimize_power_level_recursive                               */
923 /*                                                                           */
924 /* Description: This function optimizes the power level of the given system  */
925 /*              state by looking at the next system state.                   */
926 /*                                                                           */
927 /* Input:       @state: Pointer to the system state                          */
928 /*                                                                           */
929 /* Return:      On failure - negative errno return code                      */
930 /*              On success - '0' return code if the plevel was not changed   */
931 /*              On success - '1' return code if the plevel was changed       */
932 /*              On success - '2' return code if state is current state and at */
933 /*                           power level 0                                   */
934 /*                                                                           */
935 /*****************************************************************************/
936 int optimize_power_level_recursive(struct system_state *state, int force)
```

```
937 {
938   int new_plevel, level_changed = 0;
939
940   if(state == NULL || state->next == NULL) {
941     PRINT_ERROR(EINVAL, "invalid pointer to system state\n");
942     return -EINVAL;
943   }
944
945   new_plevel = state->plevel->level;
946
947   /* ------------------------------------------------- */
948   /* prevent from going below 25% battery charge       */
949   /* ------------------------------------------------- */
950   if((state->next->bat_percent < LEVEL_CHANGE_THRESHOLD) || force == 1 ) {
951     new_plevel -= 1;
952
953     /* optimize previous power level if current cannot be optimized any more */
954     if (new_plevel < 0) {
955       if (state->prev != NULL) {
956         level_changed = optimize_power_level_recursive(state->prev, 1); // recursive!
957       }
958       else {
959         level_changed = 2;
960       }
961     }
962     else {
963       set_power_level(state, new_plevel);
964       level_changed = 1;
965     }
966   }
967
968   return level_changed;
969 }
970
971 /*****************************************************************************/
972 /*                                                                           */
973 /* Function:    print_system_status                                           */
974 /*                                                                           */
975 /* Description: displays the values of the given system status               */
976 /*                                                                           */
977 /* Input:       @state: pointer to the system state to display               */
978 /*                                                                           */
979 /*****************************************************************************/
980 int print_system_status(struct system_state *state)
981 {
982   if(state == NULL) {
983     PRINT_ERROR(EINVAL, "invalid pointer to system state\n");
984     return -EINVAL;
985   }
986
987   printf("\n");
988   printf("System status:\n");
989   printf("  - system time             : %s", ctime(&(state->time)));
990   printf("  - backup-battery          : %.2fV\n", state->v_backup_bat);
991   printf("  - main supply             : %.4fV at %.2f mA \n", state->v_main_bat, state->
        i_main_bat * 1000);
992   printf("Power level:\n");
993   printf("  - current level           : %d\n", state->plevel->level);
994   printf("  - activity interval       : %d seconds\n", state->plevel->active_interval);
995   printf("  - avg. power consumption  : %d mW\n", state->plevel->power_consumption);
```

```
996   printf("\n");
997
998   return 0;
999 }
1000
1001 /*****************************************************************************/
1002 /*                                                                           */
1003 /* Function:    show_help                                                    */
1004 /*                                                                           */
1005 /* Description: This function is called when the test is started with        */
1006 /*              parameter -h. It displays all parameter options specifically */
1007 /*              available for this test.                                     */
1008 /*                                                                           */
1009 /*****************************************************************************/
1010 void show_help()
1011 {
1012   printf("\n");
1013   printf("Power management tool for Garz & Fricke LIVIUS.\n");
1014   printf("\n");
1015   printf("Usage: pman [OPTIONS]\n");
1016   printf("  -h, --help            show this help and exit\n");
1017   printf("  -s, --status          print current system status\n");
1018   printf("  -v, --verbose         increase verbosity level\n");
1019   printf("  -q, --quiet           decrease verbosity level\n");
1020 }
1021
1022 /*****************************************************************************/
1023 /*                                                                           */
1024 /* Function:    main                                                         */
1025 /*                                                                           */
1026 /* Description: Entry point to the program. It parses all the command line   */
1027 /*              inputs, sets up the initial system state and runs the        */
1028 /*              infinite loop of the program.                                */
1029 /*                                                                           */
1030 /* Input:       None.                                                        */
1031 /*                                                                           */
1032 /* Return:      On failure - negative errno return code                      */
1033 /*              On success - '0' return code.                                */
1034 /*                                                                           */
1035 /*****************************************************************************/
1036 int main(int argc, char* argv[])
1037 {
1038   struct system_state *current_state = malloc(sizeof(struct system_state));
1039   struct weather_entry *weather_data_start;
1040   struct weather_entry weather_data[48];
1041   int opt, i, show_status=0;
1042   time_t prediction_time, next_active_time;
1043
1044   /* options definition */
1045   const char* const shortopts = "shvq";
1046   struct option longopts[] = {
1047     {"help"        , 0, NULL, 'h'},
1048     {"status"      , 0, NULL, 's'},
1049     {"verbose"     , 0, NULL, 'v'},
1050     {"quiet"       , 0, NULL, 'q'},
1051     {0, 0, 0, 0}      /* Array always terminates with a NULL option. */
1052   };
1053
1054   /* parsing options */
1055   while((opt = getopt_long(argc, argv, shortopts, longopts, NULL)) != -1) {
```

```
1056      switch(opt) {
1057      case 's':
1058        show_status = 1;
1059        break;
1060      case 'h':
1061        show_help();
1062        return 1;
1063        break;
1064      case 'v':
1065        verbosity += 1;
1066        break;
1067      case 'q':
1068        verbosity -= 1;
1069        break;
1070      case ':':
1071        printf("ERROR: Option needs a value.\n");
1072        show_help();
1073        return 1;
1074        break;
1075      case '?':
1076        printf("ERROR: Unknown option: %c\n", optopt);
1077        show_help();
1078        return 1;
1079        break;
1080      default:
1081        show_help();
1082        return 1;
1083      break;
1084      }
1085    }
1086
1087    /* always enable backup battery fall back */
1088    sysfs_gpio_write_value(VBAK5V_EN, 1);
1089    sysfs_gpio_write_value(KB_LED_EN, 1);
1090
1091    /* set values for the first list element */
1092    get_current_system_state(current_state);
1093    current_state->next = NULL;
1094    current_state->prev = NULL;
1095    current_state->time = time(NULL);
1096    set_power_level(current_state, P10);
1097
1098    /* print current status */
1099    if(show_status) {
1100      print_system_status(current_state);
1101      return 0;
1102    }
1103
1104    /* parse weather data array */
1105    for(i=0; i<48; i++) {
1106      weather_data[i].time = weather_forecast_testing_4h[i].time + time(NULL);
1107      weather_data[i].condition = weather_forecast_testing_4h[i].condition;
1108      //weather_data[i].time = weather_forecast_testing_night[i].time + time(NULL);
1109      //weather_data[i].condition = weather_forecast_testing_night[i].condition;
1110    }
1111
1112    weather_data_start = &weather_data[0];
1113
1114    /* main loop */
1115    while(1) {
```

```
1116
1117        /* update system state values and set preferred power level */
1118        set_power_level(current_state, P10);
1119        current_state->time = time(NULL);
1120        get_current_system_state(current_state);
1121        PRINT_INFO(V_DEBUG, "current time is          %s", ctime(&(current_state->time)));
1122        PRINT_INFO(V_DEBUG, "current battery voltage: %.4fV\n", current_state->v_main_bat);
1123
1124        /* -------------
1125         *  active mode
1126         * ------------- */
1127        toggle_active_mode(1);
1128
1129        /* do some task in active mode */
1130        log_to_file(LOG_PATH_MEASUREMENT, "measurement simulation\n");
1131        do_active_work();
1132
1133        /* ------------------------------------
1134         * this is where the prediction happens
1135         * ------------------------------------*/
1136        /* optimize current power level using prediction */
1137        prediction_time = time(NULL) + PREDICTION_HORIZON;
1138        optimize_system_state_predicted(current_state, prediction_time, weather_data_start);
1139
1140        PRINT_INFO(V_DEBUG, "switching to power level: %d\n", current_state->plevel->level);
1141
1142        log_system_state(LOG_PATH_STATES, current_state);
1143        log_system_state_list(LOG_PATH_PREDICTION, current_state);
1144
1145        /* delete predicted states */
1146        delete_list(&(current_state->next));
1147
1148        /* --------------
1149         *  passive mode
1150         * -------------- */
1151        toggle_active_mode(0);
1152
1153        next_active_time = get_next_active_time(current_state);
1154        PRINT_INFO(V_DEBUG, "next active time will be %s", ctime(&next_active_time));
1155
1156        /* setting next RTC wakeup and go to sleep */
1157        sysfs_rtc_set_wakealarm(next_active_time);
1158        freeze_system();
1159      }
1160
1161      return 0;
1162    }
```

pman.h

```
1  #ifndef __INCLUDE_PMAN_H
2  #define __INCLUDE_PMAN_H
3
4  extern char *__progname;
5  extern char *iio_dir;
6
7  #endif /* __INCLUDE_PMAN_H */
```

sysfs_utils.c

```
/* useful set of sysfs util functionality
 *
 * Copyright (c) 2008 Jonathan Cameron
 * Copyright (c) 2014 Richard Guenther
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 */

#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <stdint.h>
#include <dirent.h>
#include <errno.h>

#include "sysfs_utils.h"

/* Made up value to limit allocation sizes */
#define IIO_MAX_NAME_LENGTH 30
#define IIO_MAX_PATH_LENGTH 75

#define FORMAT_SCAN_ELEMENTS_DIR "%s/scan_elements"
#define FORMAT_TYPE_FILE "%s_type"

/* GPIO */
#define SYSFS_GPIO_PATH       "/sys/class/gpio"
#define SYSFS_GPIO_VALUE_PATH   SYSFS_GPIO_PATH "gpio%d/value"
#define SYSFS_GPIO_EXPORT_PATH    SYSFS_GPIO_PATH "export"
#define SYSFS_GPIO_UNEXPORT_PATH  SYSFS_GPIO_PATH "unexport"

/* ADC */
#define SYSFS_ADC_DEVICE      "/iio:device0"
#define SYSFS_ADC_CHANNEL_PATH    SYSFS_ADC_DEVICE "/in_voltage%d_raw"
#define SYSFS_ADC_SCALE_PATH    SYSFS_ADC_DEVICE "/in_voltage_scale"

/* RTC */
#define SYSFS_RTC_PATH        "/sys/class/rtc/rtc0/"

/* LED */
#define SYSFS_LED_PATH        "/sys/class/leds/"

char *iio_dir = "/sys/bus/iio/devices/";

inline int _write_sysfs_int(char *filename, char *basedir, int val, int verify)
{
  int ret=0;
  FILE *sysfsfp;
  int test;
  char *temp = malloc(strlen(basedir) + strlen(filename) + 2);
  if (temp == NULL)
    return -ENOMEM;
  sprintf(temp, "%s/%s", basedir, filename);
  sysfsfp = fopen(temp, "w");
  if (sysfsfp == NULL) {
    printf("failed to open %s\n", temp);
```

```
59      ret = -errno;
60      goto error_free;
61    }
62    fprintf(sysfsfp, "%d", val);
63    fclose(sysfsfp);
64    if (verify) {
65      sysfsfp = fopen(temp, "r");
66      if (sysfsfp == NULL) {
67        printf("failed to open %s\n", temp);
68        ret = -errno;
69        goto error_free;
70      }
71      fscanf(sysfsfp, "%d", &test);
72      fclose(sysfsfp);
73      if (test != val) {
74        printf("Possible failure in int write %d to %s%s\n",
75          val,
76          basedir,
77          filename);
78        ret = -1;
79      }
80    }
81  error_free:
82    free(temp);
83    return ret;
84  }
85
86  int write_sysfs_int(char *filename, char *basedir, int val)
87  {
88    return _write_sysfs_int(filename, basedir, val, 0);
89  }
90
91  int write_sysfs_int_and_verify(char *filename, char *basedir, int val)
92  {
93    return _write_sysfs_int(filename, basedir, val, 1);
94  }
95
96  int _write_sysfs_string(char *filename, char *basedir, char *val, int verify)
97  {
98    int ret = 0;
99    FILE  *sysfsfp;
100   char *temp = malloc(strlen(basedir) + strlen(filename) + 2);
101   if (temp == NULL) {
102     printf("Memory allocation failed\n");
103     return -ENOMEM;
104   }
105   sprintf(temp, "%s/%s", basedir, filename);
106   sysfsfp = fopen(temp, "w");
107   if (sysfsfp == NULL) {
108     printf("Could not open %s\n", temp);
109     ret = -errno;
110     goto error_free;
111   }
112   fprintf(sysfsfp, "%s", val);
113   fclose(sysfsfp);
114   if (verify) {
115     sysfsfp = fopen(temp, "r");
116     if (sysfsfp == NULL) {
117       printf("could not open file to verify\n");
118       ret = -errno;
```

```
119       goto error_free;
120     }
121     fscanf(sysfsfp, "%s", temp);
122     fclose(sysfsfp);
123     if (strcmp(temp, val) != 0) {
124       printf("Possible failure in string write of %s "
125         "Should be %s "
126         "written to %s\%s\n",
127         temp,
128         val,
129         basedir,
130         filename);
131       ret = -1;
132     }
133   }
134 error_free:
135   free(temp);
136
137   return ret;
138 }
139
140 /**
141  * write_sysfs_string_and_verify() - string write, readback and verify
142  * @filename: name of file to write to
143  * @basedir: the sysfs directory in which the file is to be found
144  * @val: the string to write
145  **/
146 int write_sysfs_string_and_verify(char *filename, char *basedir, char *val)
147 {
148   return _write_sysfs_string(filename, basedir, val, 1);
149 }
150
151 /**
152  * write_sysfs_string() - string write
153  * @filename: name of file to write to
154  * @basedir: the sysfs directory in which the file is to be found
155  * @val: the string to write
156  **/
157 int write_sysfs_string(char *filename, char *basedir, char *val)
158 {
159   return _write_sysfs_string(filename, basedir, val, 0);
160 }
161
162 /**
163  * read_sysfs_posint() - read positive int from sysfs
164  * @filename: name of file to read from
165  * @basedir: the sysfs directory in which the file is to be found
166  **/
167 int read_sysfs_posint(char *filename, char *basedir)
168 {
169   int ret;
170   FILE  *sysfsfp;
171   char *temp = malloc(strlen(basedir) + strlen(filename) + 2);
172   if (temp == NULL) {
173     printf("Memory allocation failed");
174     return -ENOMEM;
175   }
176   sprintf(temp, "%s/%s", basedir, filename);
177   sysfsfp = fopen(temp, "r");
178   if (sysfsfp == NULL) {
```

```
179     ret = -errno;
180     goto error_free;
181   }
182   fscanf(sysfsfp, "%d\n", &ret);
183   fclose(sysfsfp);
184 error_free:
185   free(temp);
186   return ret;
187 }
188 
189 /**
190  * read_sysfs_posint() - read float from sysfs
191  * @filename: name of file to read from
192  * @basedir: the sysfs directory in which the file is to be found
193  * @val: pointer to the variable in which the value is to be stored
194  **/
195 int read_sysfs_float(char *filename, char *basedir, float *val)
196 {
197   float ret = 0;
198   FILE  *sysfsfp;
199   char *temp = malloc(strlen(basedir) + strlen(filename) + 2);
200   if (temp == NULL) {
201     printf("Memory allocation failed");
202     return -ENOMEM;
203   }
204   sprintf(temp, "%s/%s", basedir, filename);
205   sysfsfp = fopen(temp, "r");
206   if (sysfsfp == NULL) {
207     ret = -errno;
208     goto error_free;
209   }
210   fscanf(sysfsfp, "%f\n", val);
211   fclose(sysfsfp);
212 
213 error_free:
214   free(temp);
215   return ret;
216 }
217 
218 /**
219  * sysfs_gpio_export() - export gpio
220  * @gpio: gpio number to export
221  **/
222 int sysfs_gpio_export(unsigned int gpio)
223 {
224   return write_sysfs_int("export", SYSFS_GPIO_PATH, gpio);
225 }
226 
227 /**
228  * sysfs_gpio_unexport() - unexport gpio
229  * @gpio: gpio number to unexport
230  **/
231 int sysfs_gpio_unexport(unsigned int gpio)
232 {
233   return write_sysfs_int("unexport", SYSFS_GPIO_PATH, gpio);
234 }
235 
236 /**
237  * sysfs_gpio_read_value() - read gpio value
238  * @gpio: gpio number to read value from
```

```
239  **/
240 int sysfs_gpio_read_value(unsigned int gpio)
241 {
242   int ret=0, result;
243   char *temp;
244
245   temp = malloc(strlen(SYSFS_GPIO_VALUE_PATH) + 2);
246   if (temp == NULL) {
247     printf("Memory allocation failed");
248     return -ENOMEM;
249   }
250   sprintf(temp, "gpio%d/value", gpio);
251
252   ret = write_sysfs_int("export", SYSFS_GPIO_PATH, gpio);
253   result = read_sysfs_posint(temp, SYSFS_GPIO_PATH);
254   ret = write_sysfs_int("unexport", SYSFS_GPIO_PATH, gpio);
255
256   free(temp);
257   if(ret != 0)
258     return ret;
259
260   return result;
261 }
262
263 /**
264  * sysfs_gpio_write_value() - write gpio value
265  * @gpio: gpio number to write value to
266  **/
267 int sysfs_gpio_write_value(unsigned int gpio, int value)
268 {
269   int ret, result;
270   char *temp;
271
272   temp = malloc(strlen(SYSFS_GPIO_VALUE_PATH) + 2);
273   if (temp == NULL) {
274     printf("Memory allocation failed");
275     return -ENOMEM;
276   }
277   sprintf(temp, "gpio%d/value", gpio);
278
279   ret = write_sysfs_int("export", SYSFS_GPIO_PATH, gpio);
280   ret = write_sysfs_int(temp, SYSFS_GPIO_PATH, value);
281   ret = write_sysfs_int("unexport", SYSFS_GPIO_PATH, gpio);
282
283   free(temp);
284   return ret;
285 }
286
287 /**
288  * sysfs_adc_read_channel() - read float value from adc channel
289  * @channel: iio channel number to read value from
290  **/
291 float sysfs_adc_read_channel(unsigned int channel)
292 {
293   int ret;
294   int raw_value;
295   float raw_scale;
296   char adc_channel_path[IIO_MAX_PATH_LENGTH];
297
298   snprintf(adc_channel_path, IIO_MAX_PATH_LENGTH, SYSFS_ADC_CHANNEL_PATH, channel);
```

```

  raw_value = read_sysfs_posint(adc_channel_path, iio_dir);
  ret = read_sysfs_float(SYSFS_ADC_SCALE_PATH, iio_dir, &raw_scale);
  if(ret != 0)
    return ret;

  return (raw_scale * raw_value);
}

/**
 * sysfs_rtc_read_time() - read time from rtc in UNIX format
 **/
int sysfs_rtc_read_time(void)
{
  return read_sysfs_posint("since_epoch", SYSFS_RTC_PATH);
}

/**
 * sysfs_rtc_read_wakealarm() - read wakealarm from rtc
 **/
int sysfs_rtc_read_wakealarm(void)
{
  return read_sysfs_posint("wakealarm", SYSFS_RTC_PATH);
}

/**
 * sysfs_rtc_reset_wakealarm() - reset the rtc wakealarm by setting it to 0
 * @channel: iio channel number to read value from
 **/
int sysfs_rtc_reset_wakealarm(void)
{
  return write_sysfs_int("wakealarm", SYSFS_RTC_PATH, 0);
}

/**
 * sysfs_rtc_set_wakealarm() - set the rtc wakealarm
 * @time: alarm time in UNIX format
 **/
int sysfs_rtc_set_wakealarm(int time)
{
  sysfs_rtc_reset_wakealarm();
  return write_sysfs_int("wakealarm", SYSFS_RTC_PATH, time);
}

/**
 * sysfs_rtc_set_wakealarm() - set the rtc wakealarm
 * @time: alarm time in UNIX format
 **/
int sysfs_led_set_brightness(char *led_name, int brightness)
{
  return write_sysfs_int(led_name, SYSFS_LED_PATH, brightness);
}
```

sysfs_utils.h

```
#ifndef __INCLUDE_SYSFS_UTILS_H
#define __INCLUDE_SYSFS_UTILS_H

char *iio_dir;
```

```
5
6  /* gpios */
7  int sysfs_gpio_export(unsigned int gpio);
8  int sysfs_gpio_unexport(unsigned int gpio);
9  int sysfs_gpio_read_value(unsigned int gpio);
10 int sysfs_gpio_write_value(unsigned int gpio, int value);
11
12 /* adc */
13 float sysfs_adc_read_channel(unsigned int channel);
14
15 /* rtc */
16 int sysfs_rtc_read_time(void);
17 int sysfs_rtc_set_wakealarm(int time);
18 int sysfs_rtc_read_wakealarm(void);
19 int sysfs_rtc_reset_wakealarm(void);
20
21 #endif /* __INCLUDE_SYSFS_UTILS_H */
```

weather.h

```
#ifndef __INCLUDE_WEATHER_H
#define __INCLUDE_WEATHER_H

/*
 * Using this header file to store weather forecast data to keep the main program clean
 * THIS IS FOR TESTING ONLY
 */

#include "pman.h"

struct weather_entry {
  int   time;       /* entry time */
  int   condition;      /* weather condition */
};

/* weather forecast data for 2 days */
struct weather_entry weather_forecast_real[] = {
  { 0, 0 },
  { 1200, 0 },
  { 2400, 0 },
  { 3600, 0 },
  { 4800, 0 },
  { 6000, 0 },
  { 7200, 0 },
  { 8400, 76000 },
  { 9600, 142000 },
  { 10800, 198000 },
  { 12000, 238000 },
  { 13200, 261000 },
  { 14400, 268000 },
  { 15600, 257000 },
  { 16800, 229000 },
  { 18000, 185000 },
  { 19200, 126000 },
  { 20400, 59000 },
  { 21600, 0 },
  { 22800, 0 },
  { 24000, 0 },
  { 25200, 0 },
  { 26400, 0 },
  { 27600, 0 },
  { 28800, 0 },
  { 30000, 0 },
  { 31200, 0 },
  { 32400, 0 },
  { 33600, 0 },
  { 34800, 0 },
  { 36000, 0 },
  { 37200, 76000 },
  { 38400, 142000 },
  { 39600, 198000 },
  { 40800, 238000 },
  { 42000, 261000 },
  { 43200, 268000 },
  { 44400, 257000 },
  { 45600, 229000 },
  { 46800, 185000 },
  { 48000, 126000 },
```

```
  { 49200, 59000 },
  { 50400, 0 },
  { 51600, 0 },
  { 52800, 0 },
  { 54000, 0 },
  { 55200, 0 },
  { 56400, 0 },
  { 0, 0},
};

/* weather forecast data for an 8h measurement */
struct weather_entry weather_forecast_testing_8h[] = {
  { 0, 483000 },
  { 1200, 483000 },
  { 2400, 483000 },
  { 3600, 0 },
  { 4800, 0 },
  { 6000, 0 },
  { 7200, 483000 },
  { 8400, 483000 },
  { 9600, 483000 },
  { 10800, 0 },
  { 12000, 0 },
  { 13200, 0 },
  { 14400, 483000 },
  { 15600, 483000 },
  { 16800, 483000 },
  { 18000, 0 },
  { 19200, 0 },
  { 20400, 0 },
  { 21600, 0 },
  { 22800, 0 },
  { 24000, 0 },
  { 25200, 0 },
  { 26400, 0 },
  { 27600, 0 },
  { 28800, 483000 },
  { 30000, 483000 },
  { 31200, 483000 },
  { 32400, 0 },
  { 33600, 0 },
  { 34800, 0 },
  { 36000, 483000 },
  { 37200, 483000 },
  { 38400, 483000 },
  { 39600, 0 },
  { 40800, 0 },
  { 42000, 0 },
  { 43200, 483000 },
  { 44400, 483000 },
  { 45600, 483000 },
  { 46800, 0 },
  { 48000, 0 },
  { 49200, 0 },
  { 50400, 0 },
  { 51600, 0 },
  { 52800, 0 },
  { 54000, 0 },
  { 55200, 0 },
  { 56400, 0 },
```

```
119   { 0, 0},
120 };
121
122 /* weather forecast data for an 4h measurement */
123 struct weather_entry weather_forecast_testing_4h[] = {
124   { 0, 0 },
125   { 600, 0 },
126   { 1200, 0 },
127   { 1800, 0 },
128   { 2400, 0 },
129   { 3000, 0 },
130   { 3600, 483000 },
131   { 4200, 483000 },
132   { 4800, 483000 },
133   { 5400, 0 },
134   { 6000, 0 },
135   { 6600, 0 },
136   { 7200, 0 },
137   { 7800, 0 },
138   { 8400, 0 },
139   { 9000, 0 },
140   { 9600, 0 },
141   { 10200, 0 },
142   { 10800, 483000 },
143   { 11400, 483000 },
144   { 12000, 483000 },
145   { 12600, 0 },
146   { 13200, 0 },
147   { 13800, 0 },
148   { 14400, 0 },
149   { 15000, 0 },
150   { 15600, 0 },
151   { 16200, 0 },
152   { 16800, 0 },
153   { 17400, 0 },
154   { 18000, 483000 },
155   { 18600, 483000 },
156   { 19200, 483000 },
157   { 20400, 0 },
158   { 21000, 0 },
159   { 21600, 0 },
160   { 22200, 0 },
161   { 22800, 0 },
162   { 23400, 0 },
163   { 24000, 0 },
164   { 24600, 0 },
165   { 25200, 0 },
166   { 25800, 483000 },
167   { 26400, 483000 },
168   { 27000, 483000 },
169   { 27600, 0 },
170   { 28800, 0 },
171   { 29400, 0 },
172   { 0, 0},
173 };
174
175 /* weather forecast data for an 4h measurement (night) */
176 struct weather_entry weather_forecast_testing_night[] = {
177   { 0, 0 },       /* 00:00 */
178   { 600, 0 },
```

```
179   { 1200, 0 },
180   { 1800, 0 },
181   { 2400, 0 },
182   { 3000, 0 },
183   { 3600, 0 },
184   { 4200, 0 },
185   { 4800, 0 },
186   { 5400, 0 },
187   { 6000, 0 },
188   { 6600, 0 },
189   { 7200, 0 },
190   { 7800, 0 },
191   { 8400, 0 },
192   { 9000, 0 },
193   { 9600, 0 },
194   { 10200, 0 },
195   { 10800, 0 },
196   { 11400, 0 },
197   { 12000, 0 },
198   { 12600, 0 },
199   { 13200, 0 },
200   { 13800, 0 },
201   { 14400, 0 },
202   { 15000, 0 },
203   { 15600, 0 },
204   { 16200, 0 },
205   { 16800, 0 },
206   { 17400, 0 },
207   { 18000, 0 },
208   { 18600, 0 },
209   { 19200, 0 },
210   { 20400, 0 },
211   { 21000, 0 },
212   { 21600, 0 },
213   { 22200, 0 },
214   { 22800, 0 },
215   { 23400, 0 },
216   { 24000, 0 },
217   { 24600, 0 },
218   { 25200, 0 },
219   { 25800, 0 },
220   { 26400, 0 },
221   { 27000, 0 },
222   { 27600, 0 },
223   { 28800, 0 },
224   { 29400, 0 },
225   { 0, 0},
226 };
227
228 #endif /* __INCLUDE_WEATHER_H */
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16 APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 15.04.2014	
Ort, Datum	Unterschrift