Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Jakub Zimny

Hard- und Softwareentwurf eines Steuergerätes für drahtlose Batteriezellensensoren auf Basis eines ARM-Controllers

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Jakub Zimny

Hard- und Softwareentwurf eines Steuergerätes für drahtlose Batteriezellensensoren auf Basis eines ARM-Controllers

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr.rer.nat. Jochen Schneider

Abgegeben am 24. Januar 2014

Jakub Zimny

Thema der Bachelorthesis

Hard- und Softwareentwurf eines Steuergerätes für drahtlose Batteriezellensensoren auf Basis eines ARM-Controllers

Stichworte

ARM Cortex-M3, Mikrocontroller, Platinenlayout, Hall-Sensor, drahtlose Übertragung, ISM Band 434 MHz, SD-Karte, Batteriesteuergerät

Kurzzusammenfassung

Diese Arbeit beschreibt die Hard- und Softwareentwicklung eines Batteriesteuergerätes auf Basis eines ARM Cortex-M3 Mikrocontrollers. Daneben sollen verschiedene Schnittstellen und Peripheriebausteine sowie die funksynchronisierte Strom- und Spannungsmessung auf der Platine integriert werden. Außerdem soll eine Spannungsversorgung mit einem erweiterten Bereich möglicher Versorgungsspannungen realisiert werden, um das Batteriesteuergerät aus einer möglichst großen Zahl verschiedener Batteriesysteme versorgen zu können.

Jakub Zimny

Title of the paper

Hard- and Software development of a Battery Control Unit for wireless Battery Cell Sensors based on an ARM-Controller.

Keywords

ARM Cortex-M3, Controller, PCB layout, Hall-Sensor, wireless data transmission, ISM 434 MHz, SD-Cart, Battery Control Unit

Abstract

This paper describes the Hard- and Software development of a Battery Control Unit based on an ARM-Controller. It includes several interfaces and peripherals and it is capable of a radio synchronised current and voltage measurement. Also there is an extended range of possible supply voltages for the operation with a maximised amount of different battery types.

Inhaltsverzeichnis

1 Einführung **5**
1.1 Einleitung **5**
1.2 Ziel dieser Arbeit **8**
2 Problemanalyse und neue Anforderungen **9**
2.1 Analyse des bestehenden Systems **9**
2.2 Anforderungen an das neue Batteriesteuergerät **13**
2.2.1 Betrieb an verschiedenen Batteriesystemen 13
2.2.2 Mikrocontroller 14
2.2.3 Schnittstellen und Peripheriebausteine 15
3 Konzept und Auswahl geeigneter Hardware **18**
3.1 Konzept **18**
3.2 LM3S9D92 Mikrocontroller **20**
3.3 Versorgungsspannung **25**
3.3.1 Die 5V-Versorgungsspannung 26
3.3.2 3,3V-Versorgungsspannung 31
3.3.3 Detektion der Spannungsversorgung 33
3.4 Display **36**
3.4.1 LCD-Modul 36
3.4.2 Push-Buttons 39
3.4.3 Generierung der Displayspannung 41
3.5 FTDI-Programer **43**
3.6 Schnittstellen **43**
3.6.1 RS232 44
3.6.2 SD-Karte 45
3.6.3 CAN-Bus 49
3.6.4 Ethernet 50
3.7 Strommessung **50**
3.8 Temperaturmessung **53**
3.9 Transceiver-Modul **55**
3.10 Status LEDs **55**
3.11 Schaltplan und Platinen-Layout **56**
4 Softwareentwicklung **57**
4.1 Programmieren des FTDI-Chips **57**
4.2 Konzept **58**
4.2.1 Zustandsautomat 58
4.2.2 Die Zustände 60
4.2.3 Interrupts 62
4.3 Spannungsmessung **63**
4.4 Strommessung **65**
4.4.1 Offsetkalibrierung 65
4.4.2 Strommessung 67
4.5 Speicherung der Daten **68**

4.5.1 SD-Karte ... 68
4.5.2 Zeitbasis ... 72
4.5.3 Datenformatierung ... 73
4.6 Steuerung des Batteriesteuergerätes ... 73
4.6.1 RS232 ... 73
4.6.2 Buttons ... 76
4.7 LCD-Display ... 77

5 Laborerprobung und Versuchsbetrieb ... 82
5.1 Übersicht des Batteriesteuergerätes ... 82
5.2 Versuchsaufbau ... 83
5.3 Ergebnis ... 85

6 Fazit und Ausblick ... 89
6.1 Zusammenfassung der erreichten Ziele ... 89
6.2 Probleme während dieser Arbeit ... 90
6.3 Ausblick ... 91
6.3.1 Kalibrierung des Hall-Sensors ... 91
6.3.2 Zukünftige Erweiterungen ... 92

7 Abbildungsverzeichnis ... 94

8 Tabellenverzeichnis ... 96

9 Literaturverzeichnis ... 97

10 Anhänge ... 100
10.1 Aufgabenstellung ... 100
10.2 Schaltpläne ... 103
10.3 Platinen Layouts ... 111
10.4 Quellcode ... 113
10.5 Messdaten des Versuchsbetriebes ... 181

1 Einführung

1.1 Einleitung

Weltweit benutzt die Menschheit als Energiequelle immer noch überwiegend die fossilen Brennstoffe. Kohle und Öl gehörten 2011 mit einem Anteil von 60,3% weltweit zu den am häufigsten verwendeten Brennstoffen [1]. Doch es ist allgemein bekannt, dass diese eines Tages erschöpft sein werden. Deswegen wird bereits seit Jahren die Forschung und Entwicklung vorangetrieben um die Effizienz der erneuerbaren Energiequellen zu erhöhen. Zu den erneuerbaren Energiequellen zählen u.a. Windenergie, Solarenergie, Geothermie und Energie der Wasserkraft. Diese „sauber“ erzeugte Energie kann z.B. in Batterien gespeichert und in vielen elektrischen Geräten verbaut werden. Zum Beispiel im weltweit beliebtesten Transportmittel, dem Auto.

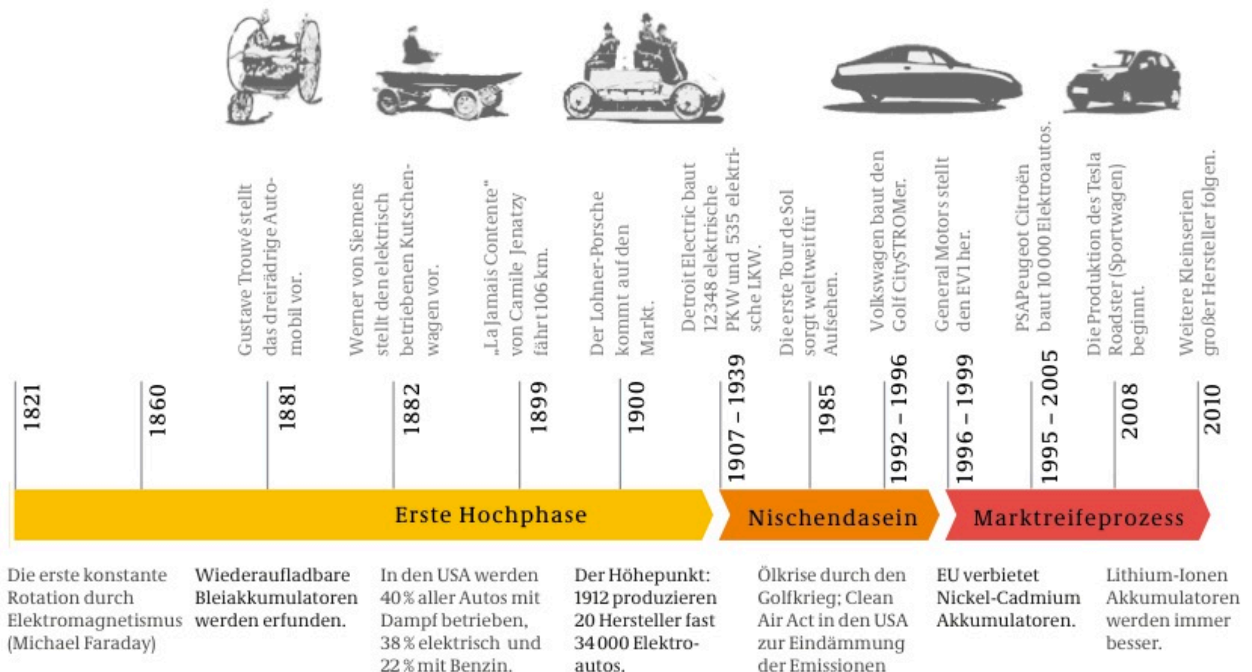


Abb. 1.1: Geschichte der Elektrofahrzeuge [2]

Historisch beginnt die Geschichte des Elektroautos 1881, als das das erste Elektrofahrzeug mit wiederaufladbarer Batterie von Gustave Trouvé vorgestellt wurde, 4 Jahre vor der Einführung des ersten Fahrzeuges mit Verbrennungsmotor von Karl Benz. Seitdem hat sich die Technik sehr stark verändert und Batterien sind in so gut wie allen Bereichen vertreten. Beispielsweise werden sie in Mobiltelefonen, MP3-Playern, Spielzeugen und vielen anderen Anwendungen und Geräten in Bereichen der Luft- und Raumfahrt sowie natürlich auch im Automobilbereich. Besonders in diesen Anwendungsgebieten, die einen sehr hohen Sicherheitsanspruch haben, müssen die Batterien sehr genau gemessen und ausgewertet werden. Durch diese Überwachung ist es möglich den Anwender vor einem drohenden Batterieausfall frühzeitig zu warnen und die Lebensdauer der Batterie zu erhöhen, da diese vor einer Tiefentladung geschützt werden kann.

Bezogen auf diese Punkte sind die wichtigsten Parameter einer Batterie der Ladezustand, auch State of Charge (SoC) genannt, sowie der Alterungszustand, State of Health (SoH). Um diese Parameter möglichst genau bestimmen zu können, muss die Batterie permanent überwacht und analysiert werden. Dazu werden leistungsfähige, spezialisierte Batteriesteuergeräte und Sensoren eingesetzt.

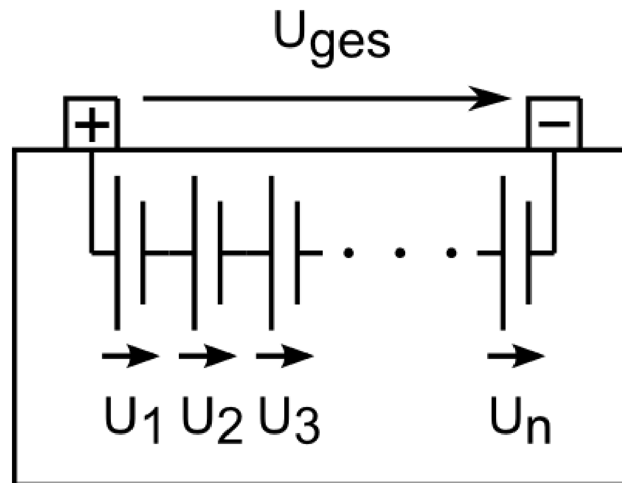


Abb. 1.2: Batterie mit n-Zellen [3]

Ein Ansatz dafür ist die Überwachung der Batteriespannung im Gesamten. Dadurch bekommt man allerdings nur den Mittelwert über alle Zellen in der Batterie und erkennt so eventuell schlechte Zellen, die beim Entladen die Mindestspannung unterschreiten und sich somit tiefentladen, nicht. Diese schwachen Zellen werden stärker belastet und altern deutlich schneller.

Das Forschungsprojekt BATSEN[1], das an der HAW Hamburg durchgeführt wird, widmet sich der Erforschung von drahtlosen Batteriesensoren. Genauer gesagt wird in diesem Projekt jede Zelle mittels eines Sensor überwacht und die Messdaten werden an ein Batteriesteuergerät übermittelt, welches dadurch das elektrische Gesamtbild der Batterie bestimmen kann.

[1] BATSEN „Drahtlose Zellsensoren für Fahrzeugbatterien“

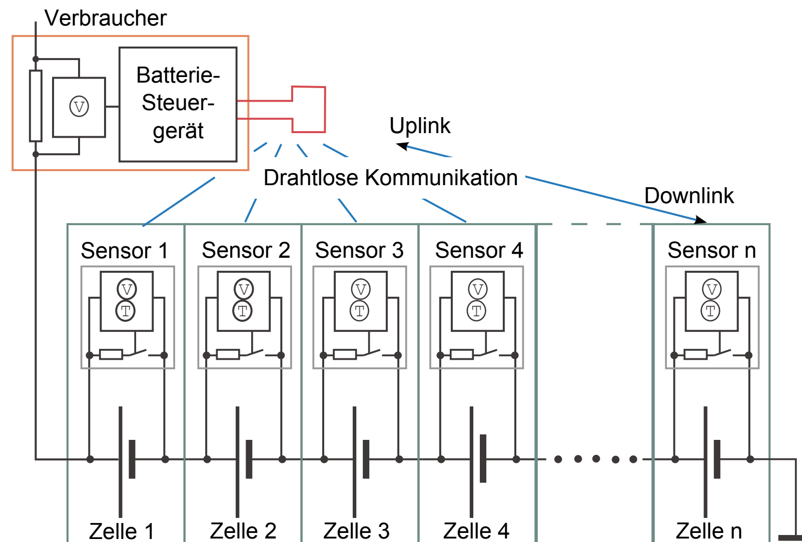


Abb. 1.3: Batterie mit n-Zellsensoren [3]

Die Abb. 1.3 veranschaulicht dieses Konzept. Für jede Zelle der Batterie wird mittels eines dedizierten Sensors die Zellspannung sowie die Zelltemperatur bestimmt und über Funk an das Batteriesteuergerät gesendet. Dieses Steuergerät misst zusätzlich noch den Stromfluss durch die Batterie. In Abb. 1.4 wird ein Spannungsverlauf beim Entladen einer solchen Batterie mit einer schwachen Zelle dargestellt. Die kritische Zelle verliert bereits vor den anderen Zellen ihre Spannung und wird somit die restliche Zeit, in der die Last noch aktiv ist, stärker belastet. Durch die Überwachung jeder Zelle können genau solche schwachen Zellen frühzeitig erkannt und getauscht werden, was letztendlich zu einer Erhöhung der Sicherheit sowie zu einer genaueren Vorhersage und Verlängerung der Lebensdauer der Batterie führt.

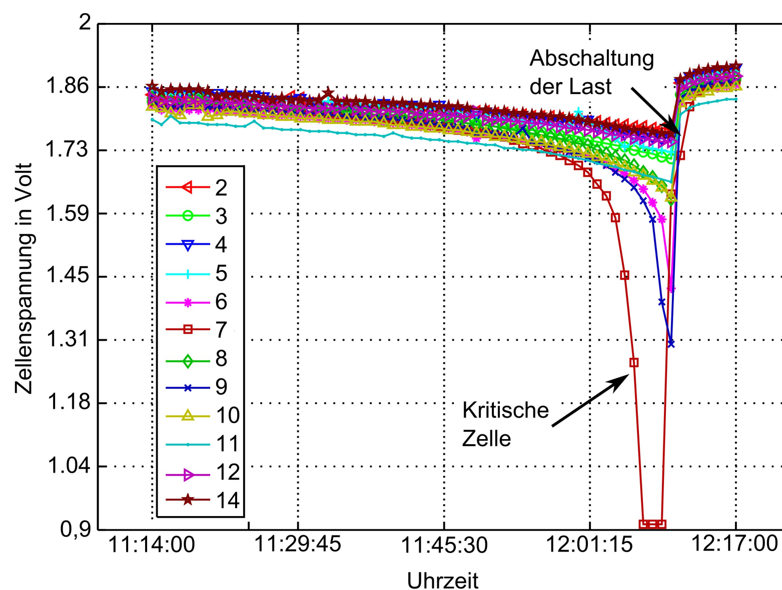


Abb. 1.4: Zellspannungsverlauf - kritische Zelle [4]

1.2 Ziel dieser Arbeit

Die bisher im Forschungsprojekt BATSEN eingesetzten Zellsensoren messen sowohl die Zellspannung als auch die Temperatur. Die Messdaten werden von den Sensoren dann über Funk im ISM Band 434MHz an ein zentrales Steuergerät übermittelt. Dieses Batteriesteuergerät besteht bisher aus einem Entwicklungsboard MSP430-168STK der Fa. Olimex [5]. Dieses Board ist, was seine Funktionalität und Schnittstellen nach außen betrifft, sehr eingeschränkt. Zudem werden, durch komplexere Algorithmen für Bestimmung des SoC und SoH, die Berechnungen auf dem Batteriesteuergerät komplexer sein und der Mikrocontroller wird voraussichtlich nicht mehr in der Lage sein, diese mit ausreichender Geschwindigkeit verarbeiten zu können.

Das gesetzte Ziel dieser Arbeit ist es, ein neues einheitliches Batteriesteuergerät auf Basis eines ARM-Mikrocontrollers aufzubauen und in Betrieb zu nehmen. Um dieses Ziel zu erreichen, existieren einige sehr wichtige Kriterien, die im Folgenden kurz dargestellt sind:

- ein erweiterter Bereich möglicher Versorgungsspannungen, damit das Batteriesteuergerät aus einer möglichst großen Anzahl verschiedener Batteriesysteme versorgt werden kann
- die Integration verschiedener Schnittstellen und Peripheriebausteine für einen möglichst flexiblen Einsatz in verschiedenen Systemen
- die Kompatibilität zu den bereits im BATSEN-Projekt vorhandenen Transceiver-Modulen und Zellsensoren
- der Entwurf und die Implementation einer modular aufgebauten Software, welche die Erweiterbarkeit des Batteriesteuergeräts gewährleistet

In den nachfolgenden Kapiteln wird zunächst eine Analyse der zu lösenden Probleme durchgeführt und die Konzepterstellung erläutert. Daraufhin wird auf die Auswahl und Integration der ausgewählten Hardware eingegangen und die implementierte Software beschrieben. Zum Abschluss dieser Arbeit werden basierend auf den Ergebnissen aus der Inbetriebnahme ein Fazit gezogen und mögliche Verbesserungsvorschläge bzw. Probleme während dieser Arbeit dargestellt.

2 Problemanalyse und neue Anforderungen

Nachfolgend wird das bereits bestehende System analysiert und es wird auf die Anforderungen an das neue Batteriesteuergerät eingegangen.

2.1 Analyse des bestehenden Systems

Das vorhandene System besteht aus dem Batteriesteuergerät, einem Transceiver-Modul sowie den Zellsensoren.

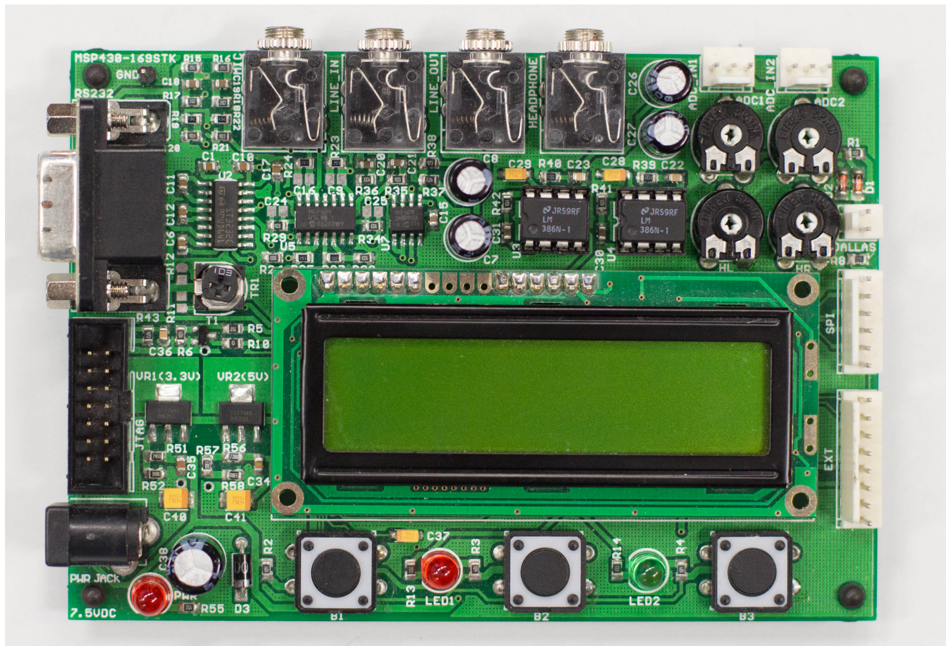


Abb. 2.1: Bisheriges Batteriesteuergerät

Abb. 2.1 zeigt das bisher verwendete Batteriesteuergerät. Es handelt sich dabei um ein Entwicklungsboard der Fa. Olimex mit der Bezeichnung MSP430-169STK [5]. Der auf diesem Board verwendete Mikrocontroller ist ein MSP430F169 der Fa. Texas Instruments [6] mit folgenden Eigenschaften:

- Ultra-Low Power Consumption:
  - 330 µA bei 1 MHz Taktfrequenz und 2,2V Versorgungsspannung
  - 1,1µA im Standby Mode
- 16-Bit RISC[2] Architektur
- max. 8MHz Taktfrequenz
- 60kB + 256B Flash Memory / 2kB RAM

[2] Reduced instruction set computing

- 48 GPIOs
- zwei 16-Bit Timer
- 12-Bit Analog/Digital-Umsetzer
- zwei serielle Schnittstellen, die als SPI, I^2C oder UART konfiguriert werden können

Zudem befindet sich auf dem Entwicklungsboard ergänzend zum Mikrocontroller:

- 8x8MB Flash Memory für Funktionen wie Data Logging
- 16x2 Liquid Crystal Display
- JTAG-, RS232- sowie SPI-Anschluss
- drei Status LEDs
- Eingangsspannungsregelung für den Bereich 7,5V-9V

Weitere vorhandene Hardware wie der Mikrofoneingang oder der Stereoeingang sind für die Funktionalität als Batteriesteuergerät im BATSEN-Projekt nicht weiter relevant.

Auffällig ist hier zunächst der sehr kleine Eingangsspannungsbereich von 7,5V-9V. Damit ist ein Betrieb an einer Fahrzeugbatterie, mit Spannungen von 12V oder größer, ausgeschlossen. Ebenso sind nur wenige Schnittstellen nach außen vorhanden. Es fehlt z.B. eine Schnittstelle zum CAN-Bus, des wichtigsten Bus-Systems im Automobilbereich. Zudem sind auf dem Entwicklungsboard nur 64MB Speicher vorhanden. Bei der verwendeten Formatierung der Daten, vgl. Kapitel 4.5.3, könnte man somit, bei einer Annahme von 1 Messung/sec nach (2.2), insgesamt ca. 51 Stunden lang Daten aufzeichnen. In der Regel werden die Messungen wesentlich schneller durchgeführt als hier angenommen, vgl. Burstmessung von Herrn Sassano [7].

$\boldsymbol{S_{max.}}$: $\boldsymbol{vorhandener\ Speicher}$

$\boldsymbol{S_{mess}}$: $\boldsymbol{\frac{Speicherverbrauch}{Messung}}$

$\boldsymbol{f_{mess}}$: $\boldsymbol{Taktrate\ der\ Speichervorgänge}$

$$\boldsymbol{t_{meas.} = \frac{S_{max.}}{S_{mess} \cdot f_{mess}}} \tag{2.1}$$

$$\boldsymbol{t_{meas.} = \frac{(64 \cdot 1024 \cdot 1024) Byte}{360 Byte \cdot \frac{1}{sec}} = 51{,}78\ Stunden} \tag{2.2}$$

Über eine Steckverbindung, zu sehen auf der rechten Seite in Abb. 2.1, wird der SPI-Bus des Mikrocontrollers, 5 GPIO-Pins des Mikrocontrollers sowie die GND und 3,3V Leitung nach außen geführt. Über diese Verbindung wird das Transceiver-Modul mit Spannung versorgt und via SPI-Bus mit Befehlen gesteuert.

Abb. 2.2: Transceiver Modul

Die Abb. 2.2 zeigt das zuvor erwähnte Transceiver-Modul, welches an das Batteriesteuergerät gesteckt wird. Als Transceiver-Chip wird der CC1101 Low-Power Transceiver der Fa. Texas Instruments verwendet [8]. Dieser kann in dem ISM-(Industrial, Scientific and Medical) sowie SRD-(Short Range Device)-Frequenzband im 315MHz, 434MHz, 868MHz und 915MHz Bereich betrieben werden. Im BATSEN-Projekt wird dieser im 434MHz ISM-Frequenzband betrieben.

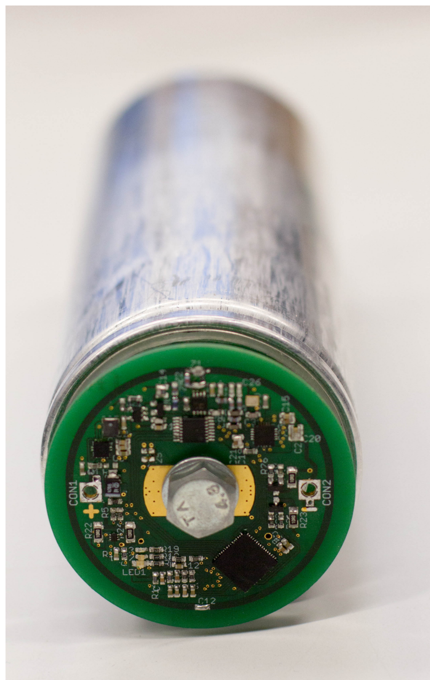


Abb. 2.3: Zellsensor der Klasse 3 an einer Batteriezelle [7]

Die Abb. 2.3 zeigt einen Zellsensor der Klasse 3, der zentral vom Batteriesteuergerät gesteuert wird und einen Down-[3] und Uplink[4] beherrscht. Dieser Sensor wird genutzt werden, um das in dieser Arbeit erstellte Batteriesteuergerät zu testen. Montiert ist dieser Sensor an einer Lithium-Eisenphosphat-Batteriezelle, kurz LiFePO4, der Fa. eec Repenning GmbH [9]. Eine solche Batteriezelle kann eine elektrische Ladung von bis zu 48Ah bei einer Spannung von 3,3V besitzen. Der Zellsensor wird direkt an die jeweilige Batteriezelle angeschlossen und von dieser auch mit Strom versorgt. Betrieben werden kann ein solcher Zellsensor in einem Spannungsbereich zwischen 0,6V und 5,5V [7].

Auf dem Zellsensor kommt als Mikrocontroller dem MSP430F235 der Fa. Texas Instruments zum Einsatz [10]. Dieser weist die folgenden Eigenschaften auf:

- Ultra-Low Power Consumption:
  - 270µA bei 1MHz Taktfrequenz und 2,2V Versorgungsspannung
  - 0,3µA im Standby Mode
- 16-Bit RISC Architektur
- max. 16MHz Taktfrequenz
- 16kB + 256B Flash Memory / 2kB RAM
- 12-Bit ADC

Die Unterschiede zwicshen dem Mikrocontroller des bisherigen Batteriesteuergerätes und des Sensors sind die niedrigere max. Taktfrequenz und der niedrigere Stromverbrauch.

Des Weiteren befindet sich auf dem Zellsensor ebenfalls ein CC1101 Transceiver, der für die Funkübertragung zwischen Transceiver-Modul und Zellsensor im Uplink sowie Downlink zuständig ist. Die Gründe für die Auswahl des Mikrocontrollers, sowie detaillierte Angaben zur Steuerung und zum Betrieb des Transceiver-Moduls und der Zellsensoren, sind in den Bachelorarbeiten von Herrn Sassano, Herrn Durdaut und Herrn Plaschke zu finden [7] [11] [12].

---

[3] Senden von Daten von einem Zellsensor zu dem Transceiver-Modul
[4] Senden von Daten von dem Transceiver-Modul zu einem Zellsensor

2.2 Anforderungen an das neue Batteriesteuergerät

2.2.1 Betrieb an verschiedenen Batteriesystemen

Wie bereits in Kapitel 1.2 kurz angesprochen, soll das Batteriesteuergerät einen möglichst großen Bereich an Versorgungsspannungen aufweisen, um an verschiedenen Batteriesystemen betrieben werden zu können. Um einen möglichst sinnvollen Spannungsbereich festlegen zu können, muss zunächst geklärt werden, wie die jeweiligen Nennspannungen der verschiedenen Batteriesysteme in den unterschiedlichen Einsatzumgebungen sind. Als zukünftige mögliche Umgebungen sind hier zu nennen der Automobil- und der Luftfahrtbereich.

	Nennspannung
Kraftfahrzeug	12V
Lastkraftwagen	24V
Flugzeug	28V

Tabelle 2.1: Nennspannungen verschiedener Bordnetze [13]

Die Tabelle 2.1 zeigt jeweils die Nennspannungen der Bordnetze von Kraftfahrzeugen, Lastkraftwagen sowie Flugzeugen. Da dies die wahrscheinlichsten Einsatzumgebungen für das Batteriesteuergerät sind, gilt es die Eingangsspannung des Batteriesteuergerätes so zu dimensionieren, dass die Funktionsfähigkeit über den gesamten Spannungsbereich zwischen 12V und 28V gegeben ist. Der mögliche Spannungsbereich sollte zudem noch den Einsatz einer kleineren Notfallbatterie berücksichtigen, aus der das Batteriesteuergerät versorgt werden kann. Wenn diese beispielsweise aus Nickel-Metallhydrid-Akkumulatoren mit 1,2V pro Zelle aufgebaut ist, würden bereits 10 Zellen in Reihe benötigt, um die notwendige Versorgungsspannung von 12V zu erreichen. Daher ist es erforderlich, die untere Grenze des möglichen Versorgungspannungsbereiches so niedrig wie möglich zu dimensionieren, um die Anzahl der Zellen und somit die Größe einer solchen Notfallbatterie möglichst klein halten zu können. Ausschlaggebend für die untere Grenze der möglichen Versorgungsspannung ist hier die min. Eingangsspannung des Spannungsreglers, der benötigt wird, um die 5V bzw. 3,3V für die Hardware auf dem Batteriesteuergerät zu generieren. Die genaue Erläuterung der für diese Arbeit gewählten Bausteine, die Funktionsweise dieser und der daraus resultierende mögliche Versorgungsspannungsbereich des Batteriesteuergerätes ist im Kapitel 3.3 nachzulesen.

2.2.2 Mikrocontroller

Wie bereits dem Ziel dieser Arbeit unter Kapitel 1.2 zu entnehmen ist, soll auf dem Batteriesteuergerät ein ARM-Mikrocontroller eingesetzt werden. Da bereits Voruntersuchungen durch Abschlussarbeiten von Herrn Schlüter [14] und Herrn Wisniewski [15], sowie Erfahrungen aus der Veranstaltung „Mikrocontrollertechnik“ an der Hochschule für Angewandte Wissenschaften vorliegen, wird als Mikrocontroller der Stellaris LM3S9D92, ein ARM Cortex-M3 Prozessor der Fa. Texas Instruments, verwendet [16].

	LM3S9D92	**MSP430F169**
Architektur	32-Bit RISC	16-Bit RSIC
Max. Taktfrequenz	80MHz	8MHz
Flash Memory	512kB	60kB + 256B
RAM	96kB	2kB
Stromverbrauch: Sleep-Mode/Typical/Max.	550µA/159mA/210mA	2µA/4mA/4,8mA

Tabelle 2.2: Vergleich LM3S9D92/MSP430F169 [16] [6]

Tabelle 2.2 verdeutlicht hier anhand der wichtigsten Parameter die direkten Unterschiede der beiden Mikrocontroller des bisherigen und des in dieser Arbeit entwickelten Batteriesteuergerätes. Zunächst ist der Geschwindigkeits- sowie Speichervorteil des LM3S9D92 zu erkennen. Weiterhin verfügt der ARM-Mikrocontroller über das ca. 8½-fache an Flash Memory und über 48 mal so viel RAM. Zudem ist die Rechengeschwindigkeit mit 80MHz, im Vergleich zum MSP430F169, zehnmal so schnell. Dies ist im Bezug auf die zukünftig größere Menge an Berechnungen sowie Daten, die der Mikrocontroller bewältigen muss, sehr vorteilhaft. Außerdem verfügt der LM3S9D92 noch über folgende Vorteile gegenüber dem MSP430F169, die nicht in dieser Tabelle erwähnt werden:

- StellarisWare Software: Für diesen Mikrocontroller sind bereits sehr viele umfangreiche Bibliotheken vorhanden, welche die Programmierung besonders erleichtern. Es müssen nicht mehr alle Register einzeln gesetzt und beschrieben werden, sondern nur noch die jeweilige Funktion mit den gewünschten Parametern aufgerufen werden. Dies erspart viel Zeit und ist sehr übersichtlich.

- Schnittstellen: Der LM3S9D92 verfügt über Schnittstellen wie CAN-Bus, USB und Ethernet, die dem MSP430F169 gänzlich fehlen.

Allerdings macht Tabelle 2.2 auch einen Nachteil deutlich. Der LM3S9D92 hat einen max. Stromverbrauch von 210mA, der MSP430F169 weist hingegen eine deutlich geringere max. Stromaufnahme von 4,8mA auf. Diese Stromverbrauchswerte gelten unter folgenden Bedingungen:

- LM3S9D92
  - **Sleep Mode:** Gesamte Peripherie ausgeschaltet, Taktfrequenz bei IOSC30kHz/64, was 468,75Hz entspricht und eine Umgebungstemperatur von 25°C
  - **Typical:** Gesamte Peripherie eingeschaltet, Taktfrequenz bei 80MHz und eine Umgebungstemperatur von erneut 25°C
  - **Max.:** Gesamte Peripherie eingeschaltet, Taktfrequenz bei 80MHz und eine Umgebungstemperatur von 85°C
- MSP430F169
  - **Sleep Mode:** Gesamte Peripherie ausgeschaltet, Taktfrequenz bei 32,768kHz und eine Umgebungstemperatur von 25°C
  - **Typical:** Gesamte Peripherie eingeschaltet, Taktfrequenz bei 8MHz und eine Umgebungstemperatur zwischen -40°C und +85°C
  - **Max.:** Gesamte Peripherie eingeschaltet, Taktfrequenz bei 8MHz und eine Umgebungstemperatur zwischen -40°C und +85°C

Als Bemerkung sei hier noch erwähnt, dass die Stromverbrauchswerte nicht unmittelbar der Realität entsprechen, da im Betrieb als Batteriesteuergerät nicht alle vorhandenen Peripheriebausteine zeitgleich aktiv sind. Außerdem lässt sich der Stromverbrauch deutlich durch die Optimierung der Software und die Abschaltung nicht genutzter Peripherie verringern. Beispielsweise erhält man durch die Abschaltung der Ethernet-Schnittstelle bereits eine Verringerung des benötigten Stromes um 58mA [16]. Da diese Untersuchungen und daraus resultierenden Optimierungen den Rahmen dieser Arbeit sprengen würden, wird hier nicht weiter darauf eingegangen.

2.2.3 Schnittstellen und Peripheriebausteine

Wie dem Ziel dieser Arbeit zu entnehmen ist, gehört die Integration möglichst vieler Schnittstellen zu den Anforderungen an den Neuentwurf des Batteriesteuergeräts. Hier gilt es zunächst herauszufinden, welche Schnittstellen den größten Vorteil bringen und am häufigsten verwendet werden.

Dabei sind fürs Erste die verwendeten und bewährten Schnittstellen aus dem vorherigen Batteriesteuergerät zu übernehmen. Diese sind im Einzelnen:

- **RS232-Schnittstelle:** Über die RS232-Schnittstelle wird das bisherige Batteriesteuergerät gesteuert. Dazu wird ein PC über einen seriellen COM-Port mit der RS232-Schnittstelle des Batteriesteuergerätes verbunden. Daraufhin können über ein Terminal-Programm wie z.B. HTerm [17] Befehle an das Batteriesteuergerät geschickt werden, die im Mikrocontroller interpretiert werden können um bestimmte Funktionen aufzurufen.

- **Hall-Sensor:** Für die Strommessung wird ein Hall-Sensor verwendet. Dieser wird an einen ADC-Kanal auf dem bisherigen Batteriesteuergerät angeschlossen.

- **Transceiver-Modul:** Das bereits in Kapitel 2.1 erwähnte Transceiver-Modul wird über eine Steckverbindung mit dem Batteriesteuergerät verbunden. Welche Bedingungen für die Implementierung des Interfaces zwischen Transceiver-Modul und Batteriesteuergerät beachtet werden müsen, wird in Kapitel 3.9 weiter erläutert.

Nun stellt sich die Frage, welche weiteren Schnittstellen in das neue Batteriesteuergerät integriert werden sollen. Diese Frage lässt sich schnell mit einem Blick in die Anforderungen dieser Arbeit in Kapitel 1.2 sowie einer Recherche zu den verwendeten Bussystemen und Schnittstellen in den Einsatzgebieten des Batteriesteuergerätes klären.

Aus der Zielsetzung wird ersichtlich, dass das Batteriesteuergerät in Zukunft eine große Menge an Daten produzieren wird. Dazu wird ein entsprechend großes Speichermedium benötigt. Eine Möglichkeit ist es, Speicher fest auf der Platine zu integrieren und die anfallenden Messdaten in diesem zu speichern, wie es bereits in der bisherigen Lösung auf Basis des Entwicklungsboards der Fa. Olimex, welches als bisheriges Batteriesteuergerät dient, realisiert. Nachteilig an dieser Variante ist es jedoch, dass nur ein Speicher mit fester Größe vorhanden ist. Dies schränkt die Flexibilität des Systems sehr stark ein und ist für zukünftige Implementierungen mit wachsenden Anforderungen nicht geeignet. Zudem sind die Daten nur auf dem Batteriesteuergerät gespeichert und müssen für eine externe Nutzung, z.B. Matlab an einem PC, erneut übertragen und ggf. formatiert werden. Daher bietet sich hier der Einsatz eines SD-Karten-Slots an. Eine SD-Karte ist flexibel im Einsatz und kann leicht an einen PC angebunden werden. Ebenso ist es möglich, die Größe des Speichermediums ohne größere Probleme anzupassen. So ist es heutzutage[5] kein Problem Speichergrößen von bis zu 64GB ohne Weiteres einsetzen zu können.

Die Automobil- und Luftfahrtbranche setzt seit vielen Jahren den CAN-Bus als Standard-Feldbussystem ein. Aus diesem Grund ist es obligatorisch dieses Bussystem in das neue Batteriesteuergerät zu integrieren. Auch wenn der CAN-Bus zur Zeit im BATSEN-Projekt keine Verwendung findet, so wird dieser mit Sicherheit in Zukunft genutzt werden.

Eine weitere zweckmäßige Schnittstelle für die Kommunikation mit externen Komponenten stellt ein Ethernet-Interface dar, da diese Schnittstelle sehr verbreitet ist und es somit ermöglicht wird das Batteriesteuergerät mit fast jedem PC zu verbinden oder dieses in ein Netzwerk zu integrieren. Über einen Ethernet-Anschluss können beispielsweise die Messdaten direkt ausgewertet werden oder Steuerbefehle an das Batteriesteuergerät gesendet werden.

---

[5] Stand: 2013

Zusammenfassend lassen sich nun die folgenden Punkte für die Auswahl der Schnittstellen des neuen Batteriesteuergerätes nennen:

- **RS232**
- **Hall-Sensor Anschluss**
- **CAN-Bus**
- **Ethernet**
- **SD-Karten-Slot**
- **Steckverbindung für das Transceiver-Modul**

Zudem werden noch weitere Peripheriekomponenten im Konzept des neuen Batteriesteuergerätes hinzukommen, die im Folgenden kurz aufgeschlüsselt werden:

- **Display mit Buttons:** Damit das Batteriesteuergerät auch unabhängig von einem PC gesteuert werden kann, wird ein Display zur Anzeige des aktuellen Status sowie Buttons zur Steuerung eines Menüs in das Batteriesteuergerät integriert.
- **FTDI-Programer:** Um die Programmierung und das Debuggen des Batteriesteuergerätes zu erleichtern, wird die FTDI-Programer-Schaltung aus der Arbeit von Herrn Schlüter übernommen [14].

Die Auswahl der geeigneten Hardware, die Implementation und Funktionsweise der hier genannten Schnittstellen und der Peripherie werden in Kapitel 3 erläutert.

3 Konzept und Auswahl geeigneter Hardware

Mit den aus Kapitel 2 gewonnenen Anforderungen lässt sich nun ein Konzept für das neue Batteriesteuergerät entwickeln.

3.1 Konzept

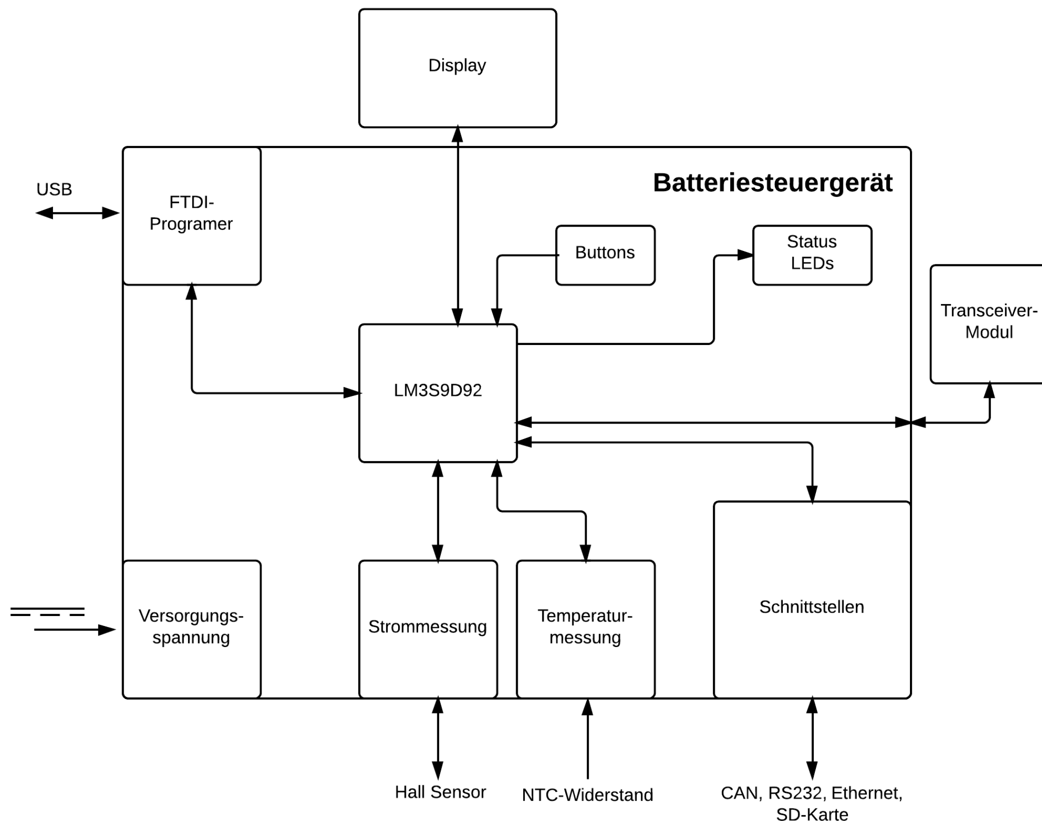


Abb. 3.1: Blockschaltbild Batteriesteuergerät

Die Abb. 3.1 zeigt das Konzept anhand eines Blockschaltbildes des Batteriesteuergerätes mit den in Kapitel 2.2 erwähnten Anforderungen. Die Verbindungen zwischen den einzelnen Modulen zeigen, ob es sich um eine bidirektionale oder unidirektionale Verbindung handelt. Zudem kann eine Verbindung auch einen Bus bzw. mehrere Leitungen beinhalten. Nachfolgend werden die einzelnen Blöcke genauer erläutert.

- **Versorgungsspannung:** Hinter diesem Block verbirgt sich die Schaltung für die Versorgungsspannung des Batteriesteuergerätes. Aus Gründen der Übersicht sind hier die Verbindungen zu den anderen Peripheriebausteinen nicht eingezeichnet.

- **LM3S9D92:** Dieser Block stellt den verwendeten Mikrocontroller dar. Alle vorhandenen Schnittstellen sowie Peripheriebausteine werden von dem Mikrocontroller gesteuert oder senden ihre Daten an diesen.

- **FTDI-Programer:** Die FTDI-Programer-Schaltung wurde aus der Arbeit von Herrn Schlüter übernommen und sorgt für eine einfachere Programmierung des Mikrocontrollers von einem PC via USB [14].
- **Strommessung:** Unter dem Block „Strommessung“ sind die Schaltung sowie die notwendige Hardware für die Anbindung eines Hall-Sensors auf dem Batteriesteuergerät zusammengefasst.
- **Temperaturmessung:** Das Modul „Temperaturmessung“ beinhaltet ebenfalls die Schaltung sowie die notwendige Hardware für die Einbindung eines NTC-Widerstandes für eine Temperaturmessung.
- **Display:** In dem Block „Display“ ist das LCD-Modul samt Daten-, Steuerungs- und Versorgungsspannungsleitungen zusammengefasst.
- **Status LEDs:** drei LEDs, die je nach Bedarf zur simplen Statusanzeige konfiguriert werden können.
- **Push-Buttons:** Es sind vier Steuerungs-Buttons für die Menüsteuerung des Batteriesteuergerätes vorgesehen. Zusätzlich gibt es noch einen Reset-Button, welcher einen Neustart des Batteriesteuergerätes auf Benutzeranforderung ermöglicht.
- **Schnittstellen:** Dieser Block fasst die Schaltung sowie alle dafür benötigten Bauteile für folgende Schnittstellen des Batteriesteuergerätes zusammen:
  - CAN
  - RS232
  - Ethernet
  - SD-Karte
- **Transceiver-Modul:** In diesem Modul befindet sich die Steckverbindung für das bereits in Kapitel 2.1 erwähnte Transceiver-Modul.

In den nachfolgenden Kapiteln werden zunächst die ausgewählten Bauteile und deren Integration in das neue Batteriesteuergerät erklärt und anschließend wird in Kapitel 4 auf die Softwarefunktionen für die jeweiligen Komponenten eingegangen.

3.2 LM3S9D92 Mikrocontroller

Der LM3S9D92 ARM-Mikrocontroller spielt, wie in Abb. 3.1 zu erkennen ist, eine zentrale Rolle auf dem Batteriesteuergerät. Es werden alle Peripheriebausteine sowie die Schnittstellen von dem Mikrocontroller gesteuert. Zudem wird die Verarbeitung der gemessenen Daten auf diesem Mikrocontroller erfolgen.

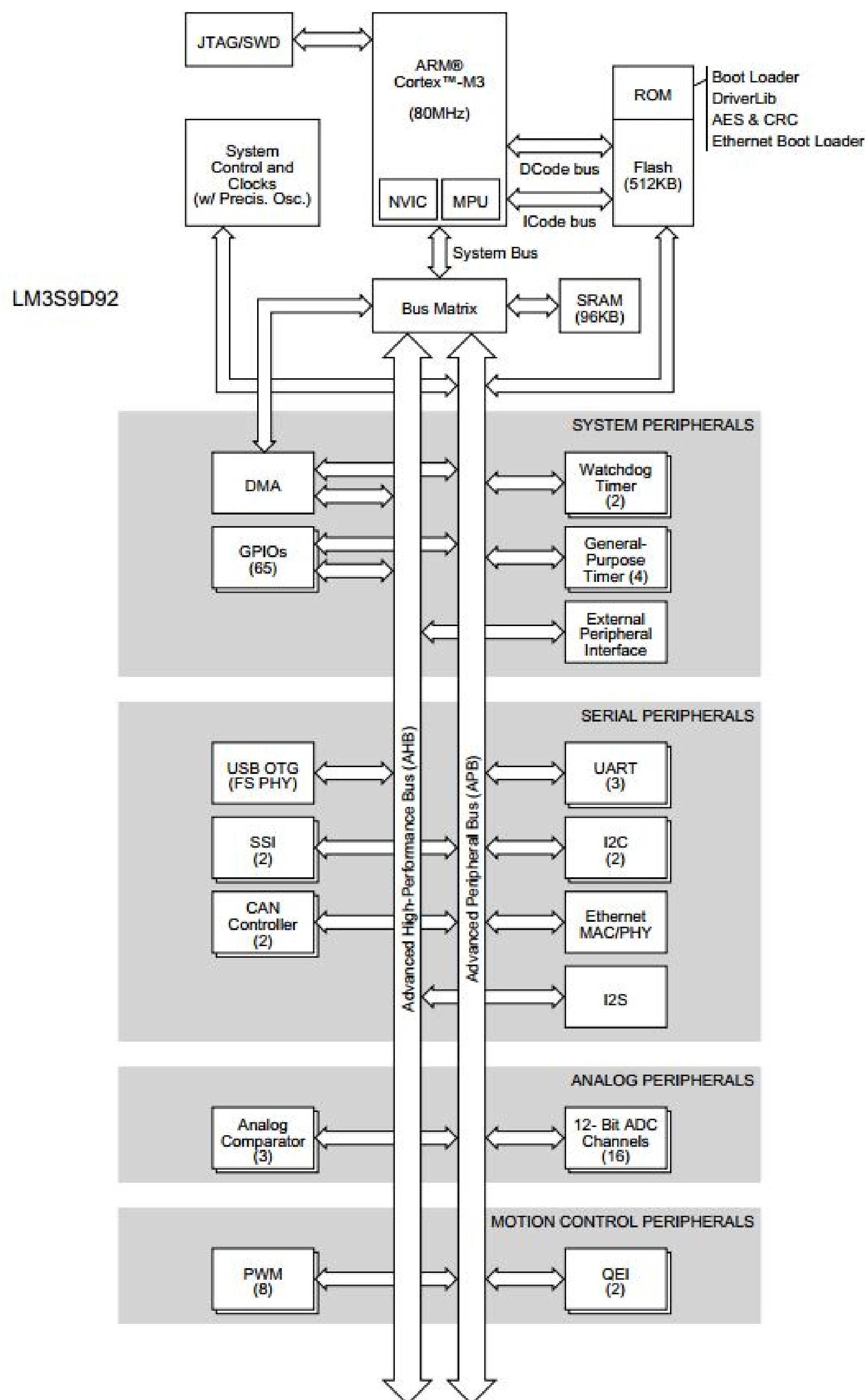


Abb. 3.2: High-Level Block Diagramm LM3S9D92 [16]

Die Abb. 3.2 veranschaulicht den internen Aufbau des LM3S9D92. Es sind die Schnittstellen sowie die Busverbindungen des Mikrocontrollers dargestellt. Alle Schnittstellen nach außen werden über den Datenbus an eine sogenannte Bus-Matrix geführt, welche die Daten an den Prozessor Cortex-M3 weiterleitet. Nachfolgend werden die benötigten Mikrocontroller-Schnittstellen bzw. Peripherieblöcke für die in Abb. 3.1 aufgeführten Module aufgelistet.

Versorgungsspannung:

- zwei GPIOs[6] (Versorgungsspannungsdetektion, vgl. Kapitel 3.3.3)

FTDI-Programer:

- JTAG/SWD

Strommessung:

- SSI

Temperaturmessung:

- ein 12-Bit-ADC-Channel

Display und Buttons:

- 16 GPIOs
- ein PWM-Output

Status LEDs:

- drei GPIOs

Schnittstellen:

- CAN-Controller
- ein UART (RS232)
- Ethernet MAC/PHY
- SSI (SD-Karte)

Transceiver-Modul:

- SSI
- sechs GPIOs

[6] General Purpose Input/Output

Nach dieser Auflistung der benötigten Peripherie sowie Schnittstellen des Mikrocontrollers wird bereits ein Problem deutlich. Der LM3S9D92 verfügt über zwei SPI[7] Schnittstellen, hier werden allerdings drei benötigt. Für die Lösung dieses Problem werden die SD-Karte und der ADC für die Strommessung an einem SPI-Bus gemeinsam betrieben. Die genaue Erläuterung dazu sowie zu den Anbindungen der anderen Module und Bauteilen an den Mikrocontroller ist in den jeweiligen Abschnitten in diesem Kapitel zu finden.

Der LM3S9D92 kann mit einer Taktfrequenz von max. 80MHz betrieben werden. Bereits in der Zielsetzung dieser Arbeit für das Batteriesteuergerät unter Kapitel 1.2 wurde erwähnt, wie wichtig die Rechengeschwindigkeit sein wird. Daher wird der Mikrocontroller auch mit seiner max. möglichen Taktfrequenz betrieben.

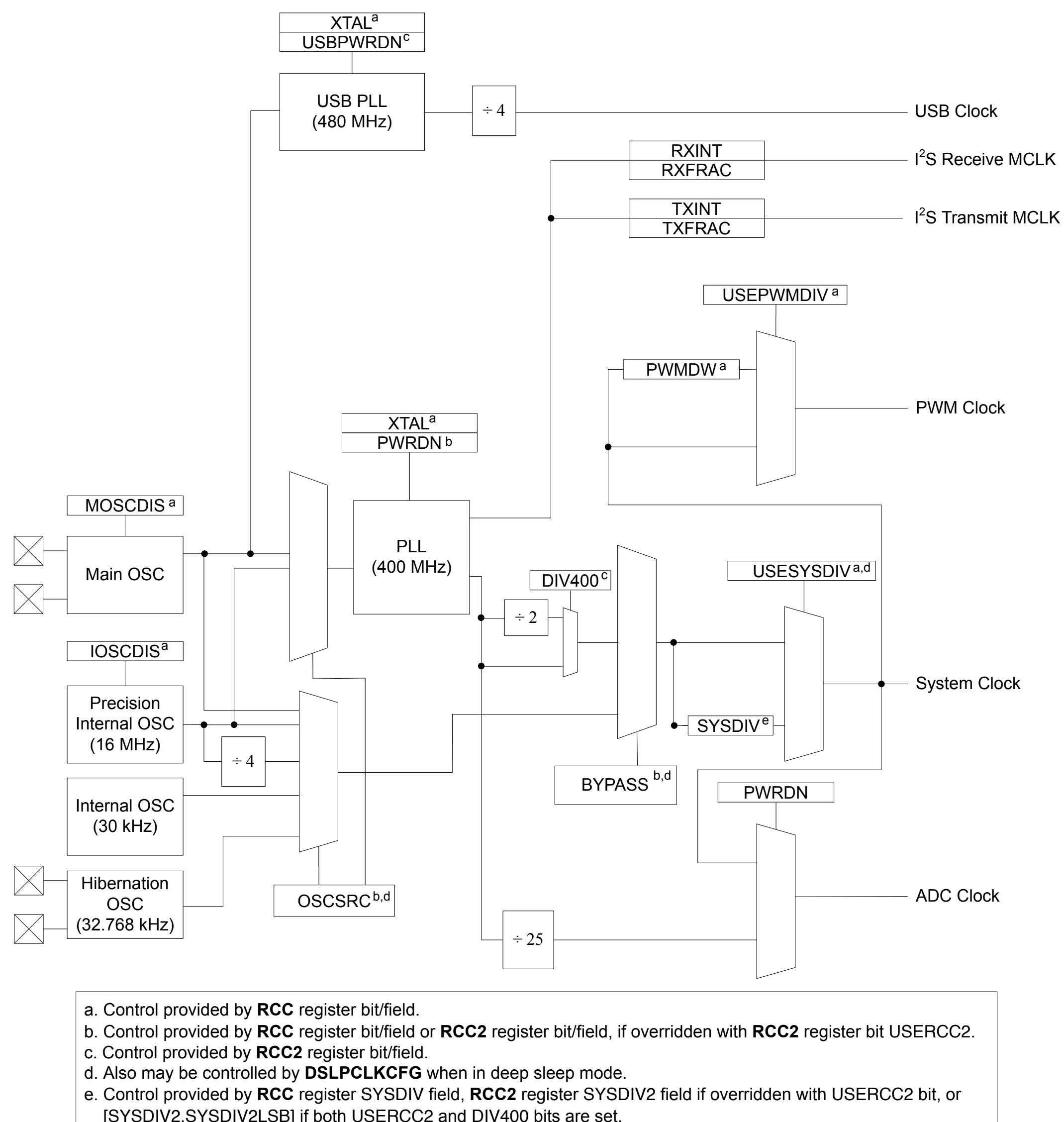


Abb. 3.3: Taktbaum LM3S9D92 [16]

[7] Auf dem LM3S9D92 Mikrocontroller ist der SPI-Bus als SSI benannt.

In Abb. 3.3 ist der interne Taktbaum des Mikrocontrollers zu sehen. Linksseitig in dieser Darstellung ist der Input für den Main OSC abgebildet, welcher der Taktgenerierung des Mikrocontrollers dient und mit 16MHz dimensioniert wurde. Da der Mikrocontroller auf dem Batteriesteuergerät aber mit 80MHz betrieben werden soll, muss der Systemtakt über die Phase-Locked-Loop Schaltung (PLL) generiert werden, zu finden mittig in Abb. 3.3. Die PLL läuft fest mit 400MHz. Sie wird zunächst durch einen festen Teiler von zweit auf 200MHz geteilt, bevor sie durch einen weiteren Teiler auf die gewünschte Frequenz gesetzt werden kann. Diese ist mit einem Teiler von 2,5 als Maximalwert 80MHz.

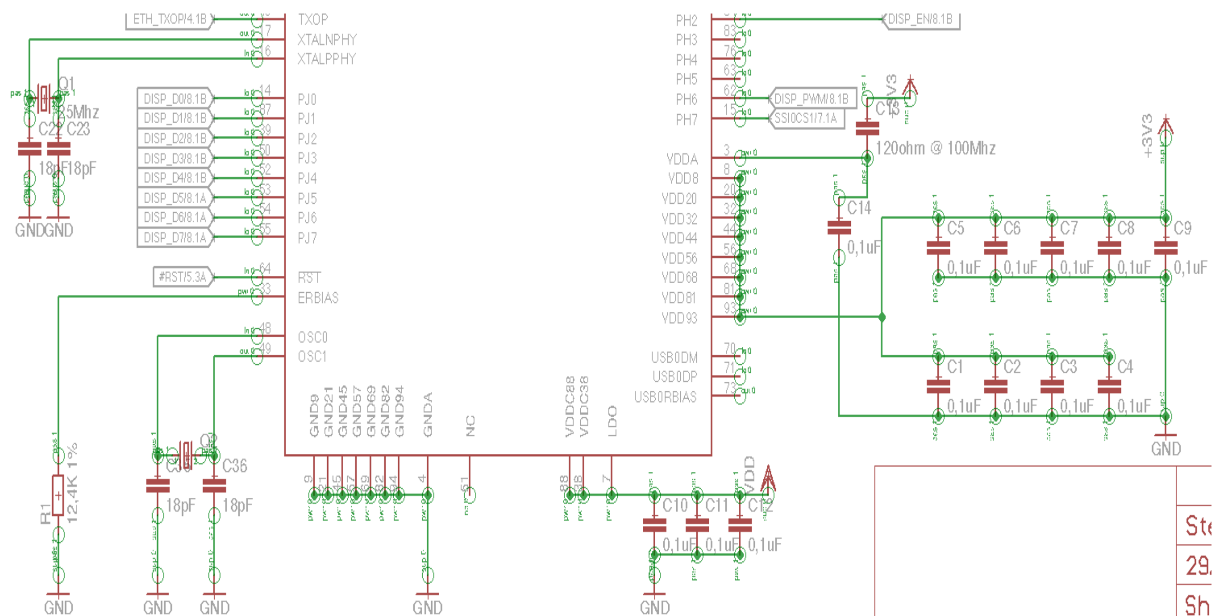


Abb. 3.4: Beschaltung LM3S9D92

Zur fehlerfreien Inbetriebnahme benötigt der Mikrocontroller eine korrekte Beschaltung. Dazu zählen, wie in Abb. 3.4 dargestellt, der Widerstand R1 (12,4kΩ, 1%), der intern vom Mikrocontroller für Ethernet PHY benutzt wird, ein 25MHz Oszillator, der ebenfalls für den Ethernet Controller benötigt wird, ein externer Oszillator, der hier auf 16MHz dimensioniert ist und an den Pins OSC0 und OSC1 angebunden wird sowie zahlreiche Kondensatoren, die der Stabilisierung der Versorgungsspannung dienen. Zusätzlich ist dem VDD-Pin ein analoger Tiefpassfilter vorgeschaltet, das das Rauschen für die Versorgungsspannung der analogen Funktionen, wie den internen ADC, minimiert [16].

	Min. Frequenz	**Max. Frequenz**
PLL wird benutzt	3,579545 MHz	16,384 MHz
PLL wird nicht benutzt	1 MHz	16,384 MHz

Tabelle 3.1: Zulässige Hauptoszillator Frequenzen

Die Tabelle 3.1 zeigt die zulässigen Frequenzen für den externen Oszillator in Abhängigkeit der Verwendung der PLL [16].

Wie bereits erwähnt, wird die Taktfrequenz auf 80MHz gesetzt. Diese ist nur mit Hilfe der PLL möglich und somit muss der externe Oszillator im Bereich 3,579545MHz und 16,384MHz liegen.

XTAL	Crystal Frequency (MHz)	PLL Frequency (MHz)	Error
0x04	3.5795	400.904	0.0023%
0x05	3.6864	398.1312	0.0047%
0x06	4.0	400	-
0x07	4.096	401.408	0.0035%
0x08	4.9152	398.1312	0.0047%
0x09	5.0	400	-
0x0A	5.12	399.36	0.0016%
0x0B	6.0	400	-
0x0C	6.144	399.36	0.0016%
0x0D	7.3728	398.1312	0.0047%
0x0E	8.0	400	-
0x0F	8.192	398.6773333	0.0033%
0x10	10.0	400	-
0x11	12.0	400	-
0x12	12.288	401.408	0.0035%
0x13	13.56	397.76	0.0056%
0x14	14.318	400.90904	0.0023%
0x15	16.0	400	-
0x16	16.384	404.1386667	0.010%

Abb. 3.5: Abweichung der PLL abhängig vom Main OSC [16]

Die Abb. 3.5 zeigt, weshalb die Wahl des Main OSC auf 16MHz gefallen ist. Bei dieser Frequenz des Main OSC arbeitet die PLL am genausten. Dies wäre auch mit den Frequenzen 4MHz, 5MHz, 6MHz, 8MHz, 10MHz und 12MHz möglich. Da der Zyklierprüfstand von Herrn Wisniewski [15] ebenfalls mit einer Main OSC Frequenz von 16MHz am LM3S9D92 betrieben wird, ist die Wahl aufgrund der Vereinheitlichung im BATSEN-Projekt ebenfalls auf 16MHz gefallen.

3.3 Versorgungsspannung

Als Versorgungsspannung benötigen die Bausteine des Batteriesteuergeräts die typischen Versorgungsspannungen von 5V und 3,3V. Dafür muss die Eingangsspannung, vgl. Abb. 3.6, herunter geregelt werden. Dies wird als Reihenschaltung zweier Regler realisiert. Zunächst wird eine 5V Spannung erzeugt, welche dann von einem weiteren Regler auf 3,3V herabgesetzt wird.

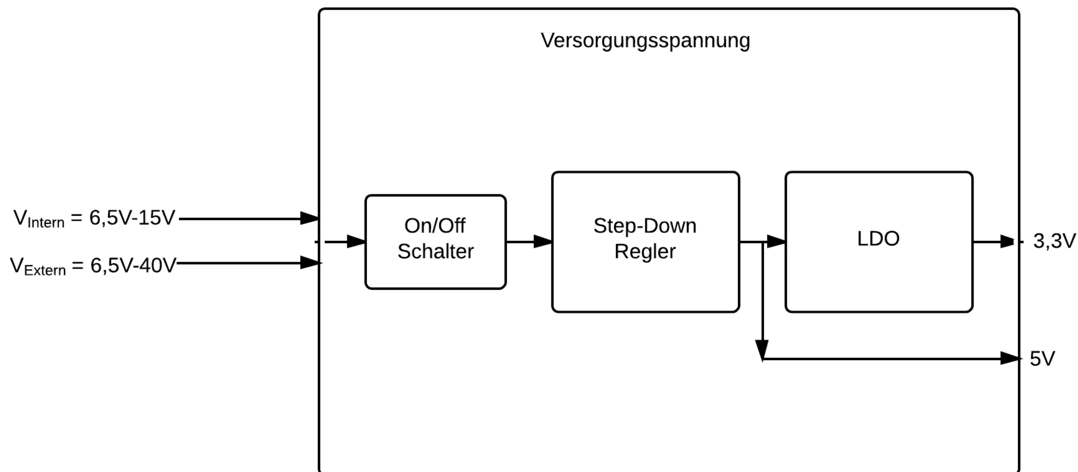


Abb. 3.6: Blockschaltbild Versorgungsspannung

Die Eingangsspannung für das Batteriesteuergerät kann über zwei verschiedene Anschlüsse erfolgen, einmal über den in Abb. 3.6 gezeigten Anschluss V_{Intern} und über den Anschluss V_{Extern}.Der Eingang V_{Intern} hat einen möglichen Spannungsbereich zwischen 6,5V und 15V und ist für einen Anschluss einer kleineren internen Batterie, z.B. einer Notfallbatterie, vorgesehen. V_{Extern} ist vorgesehen für einen Spannungsgenerator bzw. die Starterbatterie im Fahr- oder Flugzeug und verfügt über einen möglichen Spannungsbereich zwischen 6,5V und 40V. Dieser große Bereich ist notwendig, da die Bordnetze von Fahr- bzw. Flugzeugen in dem Spannungsbereich 12V bei Kraftfahrzeugen bis hin zu 28V bei Flugzeugen liegen können [13], vgl. Tabelle 2.1 in Kapitel 2.2.1.

Die zulässigen Eingangsspannungen von V_{Extern} sind begrenzt durch den Step-Down-Regler, Näheres dazu in Kapitel 3.3.1 und von V_{Intern} durch den Spannungsteiler sowie die max. Eingangsspannung am Mikrocontroller sind, erklärt in Kapitel 3.3.3.

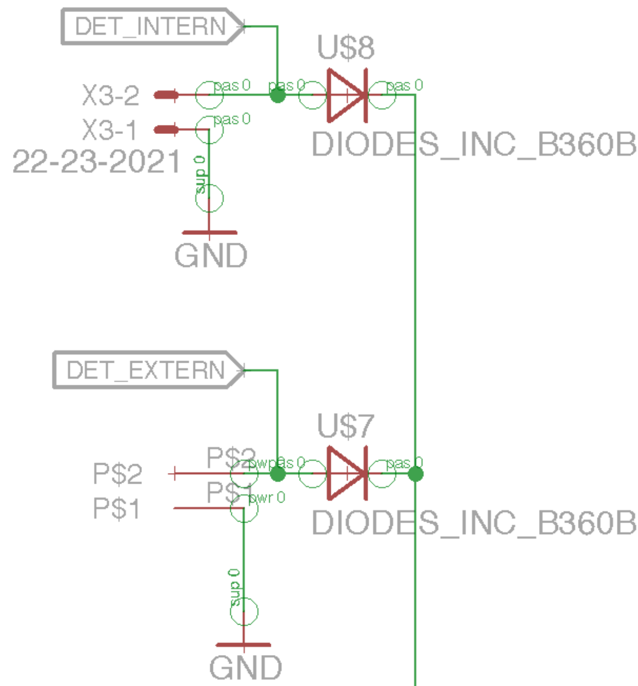


Abb. 3.7: Schaltplan der Anschlüsse der Versorgungsspannung

Für den Fall, dass zwei Spannungsquellen angeschlossen sind, ist am Eingang jeweils eine schützende Diode vorgesehen, um einen Kurzschluss der beiden Spannungsquellen zu verhindern (Abb. 3.7). Dazu müssen die Dioden eine Sperrspannung von mindestens 40V aufweisen, was der max. zulässigen Eingangsspannung des Batteriesteuergerätes entspricht. Die verwendeten Dioden der Fa. Vishay haben eine max. Sperrspannung von 60V und lassen einen Strom in Durchlassrichtung von bis zu 3A zu [18]. Liegt nun an beiden Eingängen eine Spannung an, sperrt die Diode des Einganges mit der niedrigeren Spannung und das Batteriesteuergerät wird aus dem Eingang mit der höheren Eingangsspannung versorgt.

3.3.1 Die 5V-Versorgungsspannung

Der Block „Step-Down-Regler“ in Abb. 3.6 sorgt für das Herunterregeln der Eingangsspannung auf 5V. Dazu gibt es günstige und ebenso einfach zu beschaltende Low-Dropout-Spannungsregler (LDO). Allerdings ist zu beachten, dass im ungünstigsten Fall eine Spannungsdifferenz von 40V zu 5V am Regler anliegt und dazu max. Ströme von bis zu 500mA fließen können. Der daraus resultierende Wirkungsgrad lässt sich für LDOs wie folgt errechnen:

$$\eta = \frac{U_{Out} \cdot I_{Out}}{U_{In} \cdot I_{In}} = \frac{5V \cdot 0,5A}{40V \cdot 0,5A} = 12,5\% \qquad (3.1)$$

Damit hätte ein LDO bei dieser Spannungsregelung einen Wirkungsgrad von ca. 12,5% und würde eine sehr hohe Abwärme produzieren. Dazu wäre, je nach Bauart des Gehäuses, ein Kühlkörper notwendig. Daher kommt für diese Regelung nur ein Step-Down-Regler in Frage.

Die grundlegende Funktionsweise eines Step-Down-Reglers lässt sich mit Abb. 3.8 erklären.

$\boldsymbol{U_E}$: ***Eingangsspannung BS***
$\boldsymbol{U_A}$: ***Ausgangsspannung Step − Down Regler***
$\boldsymbol{S}$: ***Schalter***
$\boldsymbol{D}$: ***Schutzdiode***
$\boldsymbol{L}$: ***Spule***
$\boldsymbol{C}$: ***Kondensator***

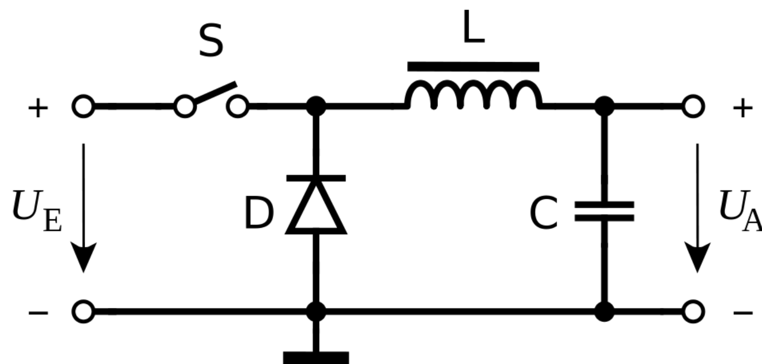


Abb. 3.8: Funktionsweise Step-Down Regler [19]

Zunächst lässt sich die Funktion in zwei Phasen einteilen. Einmal in die Phase t_{ON}, in welcher der Schalter S, ein Transistor, geschlossen ist und die Phase t_{OFF}, bei geöffnetem Schalter. Die Dauer der beiden Phasen hängt von der internen Taktfrequenz des Reglers ab.

- **t_{ON}:** In der t_{ON}-Phase ist die Spannung U_{Diode} über der Diode D gleich der Spannung U_E. Da U_A kleiner ist als U_E, wird der Ausgangskondensator C geladen und der Strom in der Spule L steigt linear an. Über eine Rückführung der Ausgangsspannung U_A zum Regler, wird durch eine interne Schaltung der Schalttransistor bei einem erreichten Schwellwert wieder ausgeschaltet.
- **t_{OFF}:** Wird der Schalter S nun geöffnet, dreht sich die Spannung an der Induktivität L um und der Strom fließt über die Diode und die Ausgangslast ab.

Diese beiden Phasen wiederholen sich mit der internen Taktfrequenz des Step-Down Reglers. Als besonders wichtig bei der Beschaltung eines solchen Reglers ist die korrekte Dimensionierung der Diode sowie der Induktivität.

Bei der Wahl des Step-Down-Reglers für das Batteriesteuergerät ist es wichtig, dass der Regler einen Eingangsspannungsbereich von min. 12V-28V aufweist. Wie in Kapitel 2.2.1 erwähnt, wäre es sehr von Vorteil, die untere Eingangsspannungsgrenze so niedrig wie möglich zu halten. Daher ist die Wahl auf den LM2672M-5.0 Step-Down Regler der Fa. Texas Instruments gefallen [20]. Dieser weist die folgenden Eigenschaften auf:

- Wirkungsgrad von bis zu 96%

- max. Ausgangsstrom von bis zu 1A
- Eingangsspannung 8V-40V (6,5V-40V bei einem Ausgangsstrom bis zu 0,5A)
- Ausgangsspannung: 5V
- 260kHz Fixed Frequency Internal Oscillator

Durch den hohen Wirkungsgrad von über 90% entsteht so signifikant weniger Abwärme als bei einem LDO und eine zusätzliche Kühlung durch einen Kühlkörper wird nicht benötigt. Zudem bietet dieser Regler einen Eingangsspannungsbereich von 8V-40V bei 1A Ausgangsstrom. Bei einem Ausgangsstrom von bis zu 0,5A liegt der Spannungsbereich sogar bei 6,5V-40V. Da das Batteriesteuergerät einen max. Stromverbrauch von 0,5A hat, kann dieser Bereich genutzt werden.

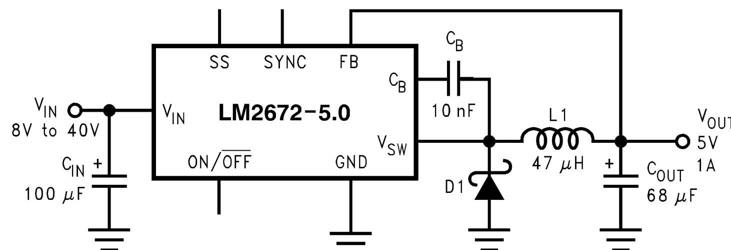


Abb. 3.9: Schaltplan LM2672-5.0 [20]

Für den hier eingesetzten LM2672M-5.0, Abb. 3.9, werden die benötigten Bauteile folgendermaßen dimensioniert [20]:

Bauteil	Wert
L1	47µF
C_{OUT}	100µF/10V
D1	1A/70V
C_{IN}	100µF/50V low ESR Elko
C_B	0,1µF/50V

Tabelle 3.2: Dimensionierung Bauteile für LM2672-5.0

In Abb. 3.10, Abb. 3.11 und Abb. 3.12 ist die Eingangsspannung (Channel #1, obere Linie, blau), direkte Ausgangsspannung am LM2672-5.0 (Channel #3, untere Linie, lila) sowie die durch den Ausgangskondensator C_{OUT} geglättete 5V Ausgangsspannung (Channel #2, mittlere Linie, türkis) zu sehen. Man erkennt deutlich die Schaltvorgänge im 260kHz-Takt des Step-Down-Reglers sowie die Anpassung des Tastgrades. Je höher die Eingangsspannung wird, desto kleiner wird der Tastgrad.

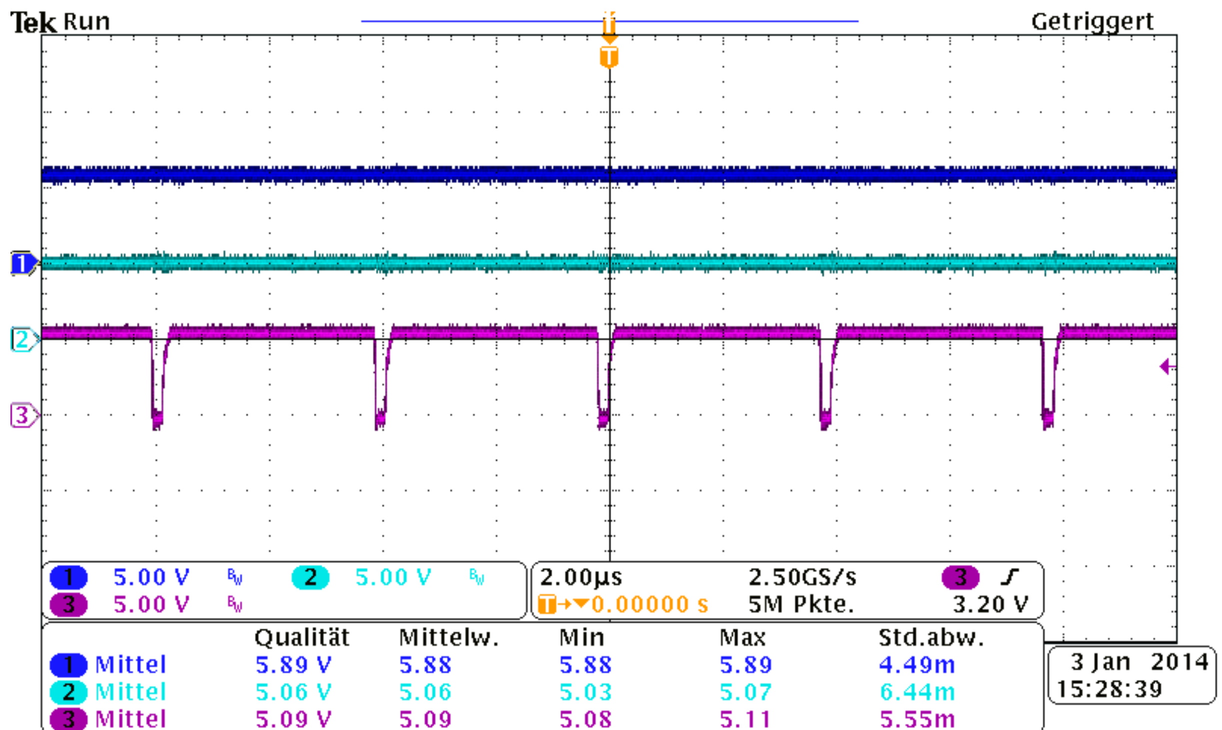


Abb. 3.10: Ausgangsspannung LM2672-5.0 bei 6V

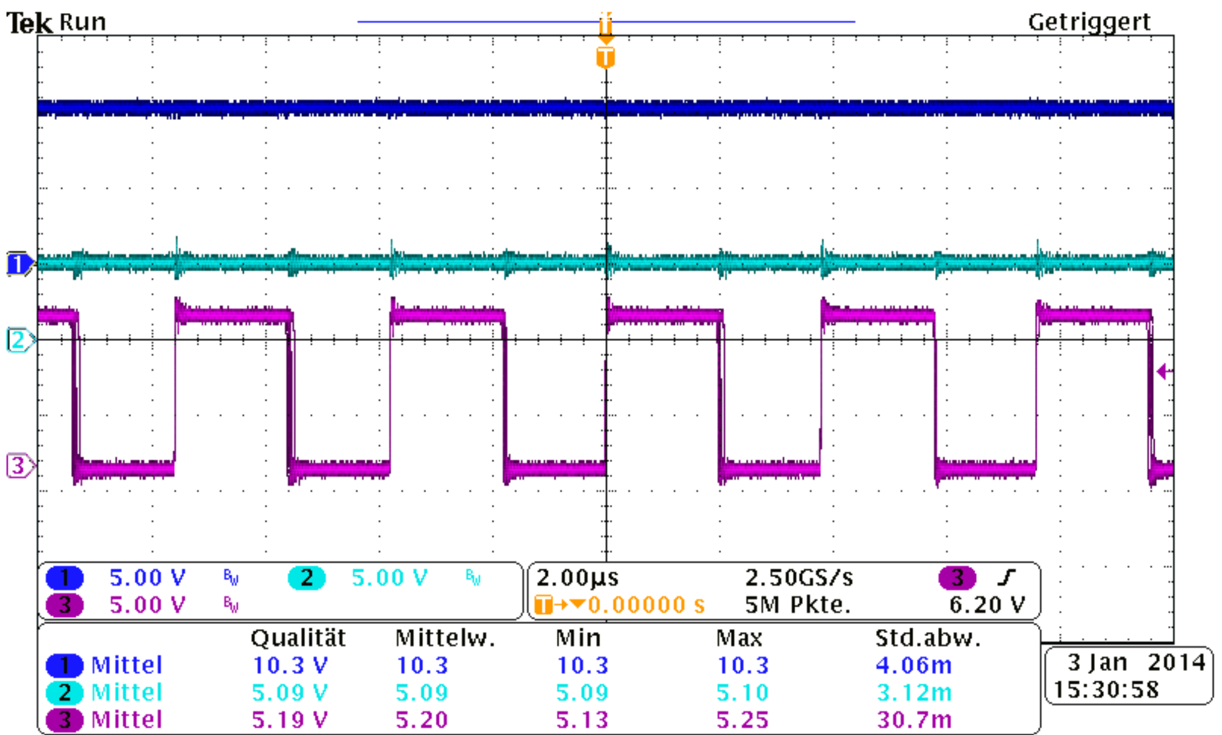


Abb. 3.11: Ausgangsspannung LM2672-5.0 bei 10V

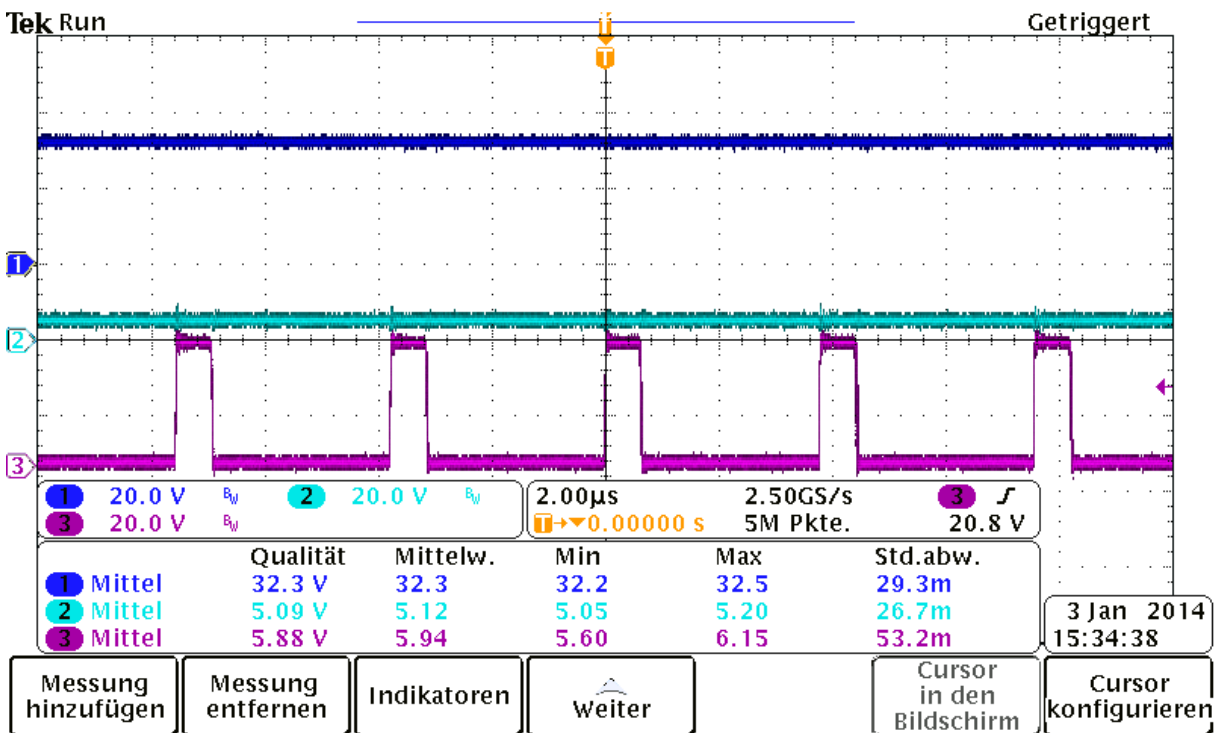


Abb. 3.12: Ausgangsspannung LM2672-5.0 bei 32V

3.3.2 3,3V-Versorgungsspannung

Wie bereits in Kapitel 3.3 erwähnt, benötigen die Bausteine des Batteriesteuergeräts zu der 5V-Versorgungsspannung eine weitere Versorgungsspannung von 3,3V. Diese wird durch einen Regler, der in Reihe zu dem Step-Down-Regler geschaltet ist, realisiert (Abb. 3.6).

$$\boldsymbol{\eta = \frac{U_{Out} \cdot I_{Out}}{U_{In} \cdot I_{In}} = \frac{3,3V \cdot 0,5A}{5V \cdot 0,5A} = 66,6\%} \tag{3.2}$$

Nach Formel (3.2) liegt der Wirkungsgrad somit bei 66,6%. Dabei entsteht zwar immer noch eine hohe Abwärme, diese lässt sich trotzdem ohne zusätzlichen Kühlkörper über das Gehäuse des Reglers sowie die Kupferleitungen auf dem Batteriesteuergerät abführen.

Die Wahl des LDOs ist auf den LM1117-3.3 Low-Dropout Linear Regler, ebenfalls von der Fa. Texas Instruments, gefallen [21]. Dieser wurde schon in dem vorherigen Batteriesteuergerät, dem Entwicklungsboard der Fa. Olimex, verwendet [5]. Dieser LDO besitzt folgende Spezifikation [22]:

- max. Eingangsspannung: 20V
- max. Ausgangsstrom: 800mA

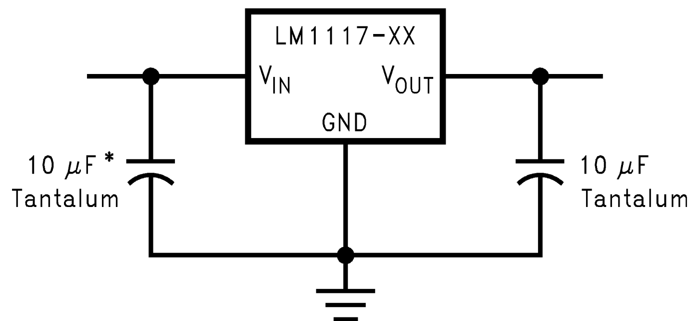


Abb. 3.13: Application Note LM1117-3.3 [22]

Für die Beschaltung des Reglers werden lediglich zwei Kondensatoren benötigt (Abb. 3.13). Der Eingangskondensator wird, wie in dem Datenblatt für die meisten Anwendungen empfohlen, mit 10 µF dimensioniert. Bei dem Ausgangskondensator ist für die Stabilität der Ausgangsspannung besonders wichtig, dass dieser einen äquivalenten Serienwiderstand, abgekürzt als ESR, von 0,3Ω bis 22Ω aufweist [22]. Der hier verwendete Kondensator verfügt über einen max. ESR von 2,2Ω [23].

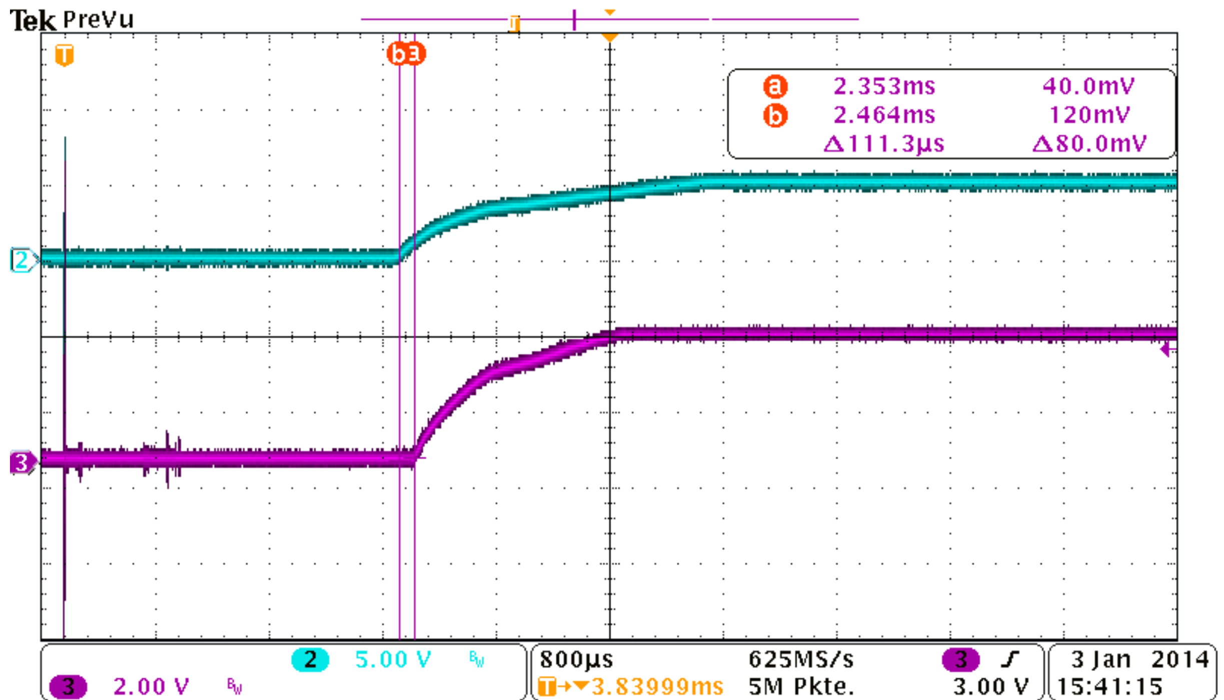


Abb. 3.14: Verzögerung der Regelung auf 3,3V

Die Abb. 3.14 zeigt jeweils die Ausgangsspannung des 5V-Reglers (Channel #2, oberer Verlauf, türkis) und 3,3V-Reglers (Channel #3, unterer Verlauf, lila). Da der LDO in der Reihenschaltung beider Regler der Hintere ist und eine Mindestspannung benötigt, beginnt dieser erst mit einer Verzögerung von ca. 110µs an arbeiten.

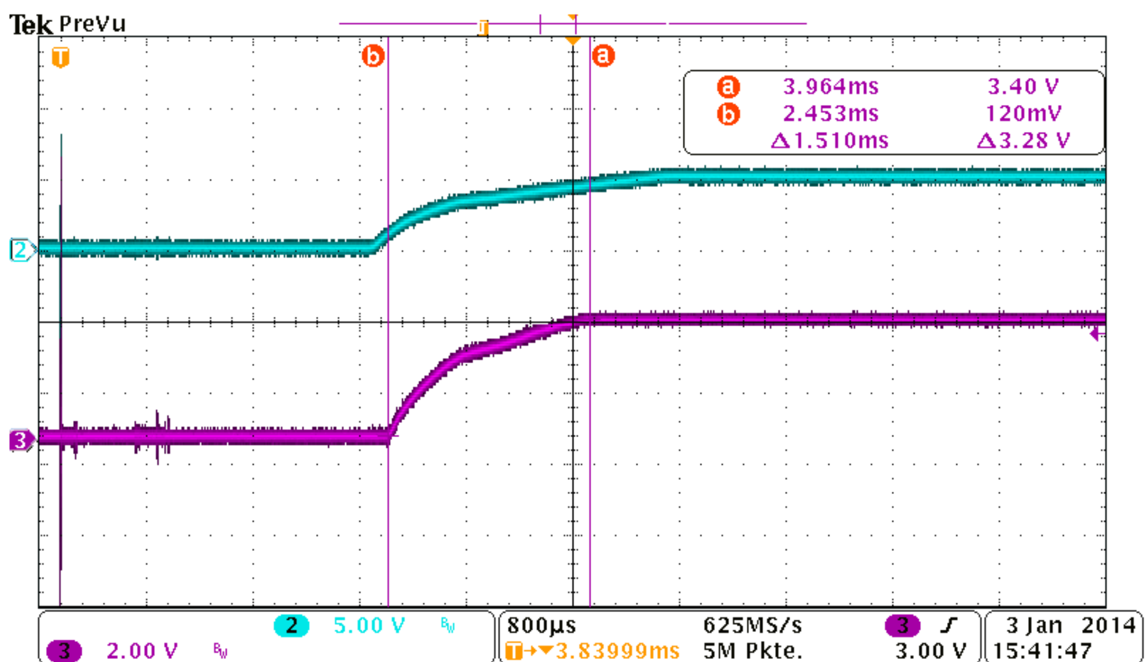


Abb. 3.15: 3,3V Startup Zeit 3,3V

Die Abb. 3.15 zeigt die Gesamtdauer der Spannungsreglung für die 3,3V-Spannungsversorgung an. Nach der Verzögerung zu dem Step-Down-Regler von ca. 110µs beginnt der LDO seine Ausgangsspannung auf 3,3V zu regeln und hat diese nach ca. 1,5ms erreicht. Zu bemerken ist hierbei noch, dass die 3,3V-Spannung schon erreicht wird, bevor der Step-Down-Regler seine 5V-Ausgangsspannung erreicht. Es kann also passieren, dass Bauteile mit 3,3V-Versorgungsspannung schon vor den Bauteilen mit 5V-Versorgungsspannung auf dem Batteriesteuergerät aktiv sind.

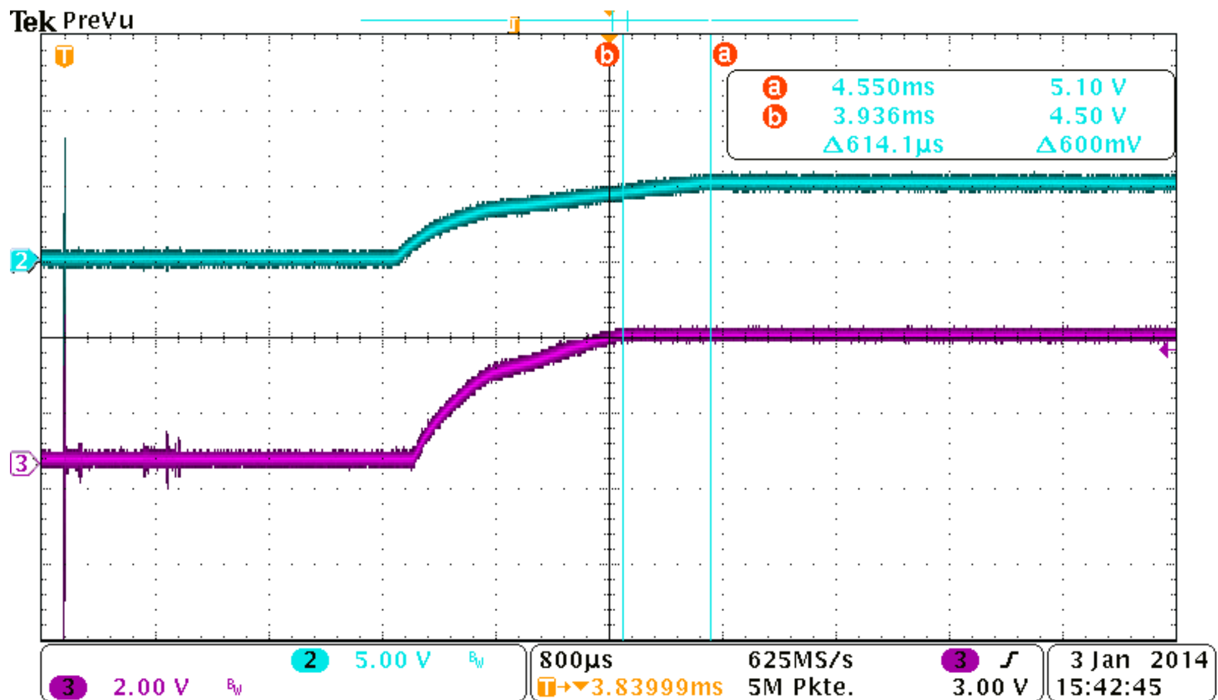


Abb. 3.16: Verzögerung Startup 5V

Der Step-Down-Regler erreicht nach weiteren 614µs, vgl. Abb. 3.16, seine 5V Ausgangsspannung und benötigt somit für den gesamten Startup-Vorgang ca. 2,2ms.

3.3.3 Detektion der Spannungsversorgung

Wie bereits in Kapitel 3.3 erwähnt, verfügt das Batteriesteuergerät über zwei Versorgungsspannungseingänge. Es kann entweder über eine Spannungsquelle bzw. die Starterbatterie oder über eine kleinere zusätzliche Batterie versorgt werden. Für eine Auswertung und Überwachung einer Batterie mit dem Batteriesteuergerät ist es wichtig zu wissen, aus welcher Quelle das Batteriesteuergerät versorgt wird. Dazu muss aus beiden Versorgungsspannungseingängen jeweils eine Leitung an den Mikrocontroller geführt werden. Da aber eine Versorgungsspannung im Bereich zwischen 6,5V und 40V liegen kann und der Mikrocontroller nur eine max. Eingangsspannung von 5,5V an den GPIO Pins hat, muss die Spannung angepasst werden. Die Spannung darf im Bereich zwischen 2,1V und 5V liegen, um von dem Mikrocontroller als „High“-Pegel interpretiert werden zu können.

Da die kleinere (Notfall-)Batterie einen geringeren Spannungspegel hat, ist hier ein einfacher Spannungsteiler mit zwei Widerständen ausreichend, um die Eingangsspannung in den vom Mikrocontroller spezifizierten Bereich herunter zu teilen (Abb. 3.17).

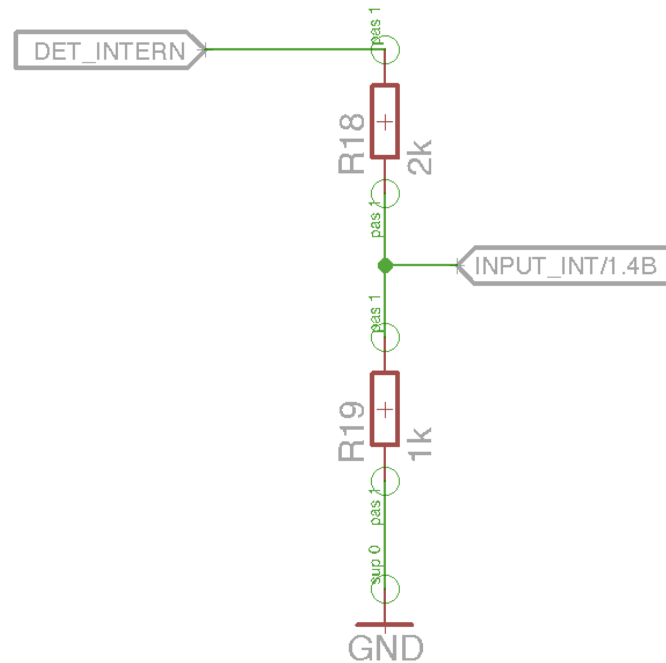


Abb. 3.17: Spannungsdetektion interne Batterie

$\boldsymbol{U_{\mu C}}$: ***Ausgangsspannung Spannungsteiler***
$\boldsymbol{U_E}$: ***Eingangsspannung Spannungsteiler (Spannung der Batterie)***

$$\boldsymbol{U_{\mu C} = \frac{U_E}{R_{18} + R_{19}} \cdot R_{19}} \quad (3.3)$$

$$\boldsymbol{U_{E,min} = \frac{U_{\mu C,min} \cdot (R_{18} + R_{19})}{R_{19}}} \quad (3.4)$$

$$U_{E,max} = \frac{U_{\mu C,max} \cdot (R_{18} + R_{19})}{R_{19}} \quad (3.5)$$

Somit ergibt sich nach (3.4) und (3.5) ein Spannungsbereich zwischen 6,3V und 15V für die Eingangsspannung an V_{Intern}, vgl. Kapitel 3.3. Wenn als (Notfall-)Batterie Nickel-Metallhydrid-Akkumulatoren mit 1,2V pro Zelle eingesetzt werden, sind min. sechs in Reihe geschaltete Zellen notwendig.

Für die Detektion der externen Batterie bzw. des Spannungsgenerators wird mit der Eingangsspannung ein MOSFET geschaltet, welcher die 3,3V-Versorgungsspannung auf einen Eingang des Mikrocontrollers schaltet, (vgl. Abb. 3.18). Für die Dimensionierung des MOSFET ist die Gate-Source-Spannung wichtig. Damit diese möglichst gering gehalten wird, muss zunächst ebenfalls ein Spannungsteiler eingesetzt werden, um die Eingangsspannung zu verkleinern.

$$\boldsymbol{U_{Gate,min} = \frac{U_{E,min} \cdot R_{45}}{(R_{45} + R_{26})} = \frac{6V}{3} = 2V} \qquad (3.6)$$

$$\boldsymbol{U_{Gate,max} = \frac{U_{E,max} \cdot R_{45}}{(R_{45} + R_{26})} = \frac{40V}{3} = 13,3V} \qquad (3.7)$$

Diese kann nach Formel (3.6) und (3.7), für eine Eingangsspannung zwischen 6V und 40V, in den Bereichen 2V und 13,3V liegen. Dazu eignet sich der 2N7002 MOSFET der Fa. Fairchild. Dieser hat eine maximale Gate-Source-Spannung von 20V sowie eine Drain-Source-Spannung von 60V [24]. Da hier nur 3,3V am Drain-Anschluss anliegen, ist dies zu vernachlässigen.

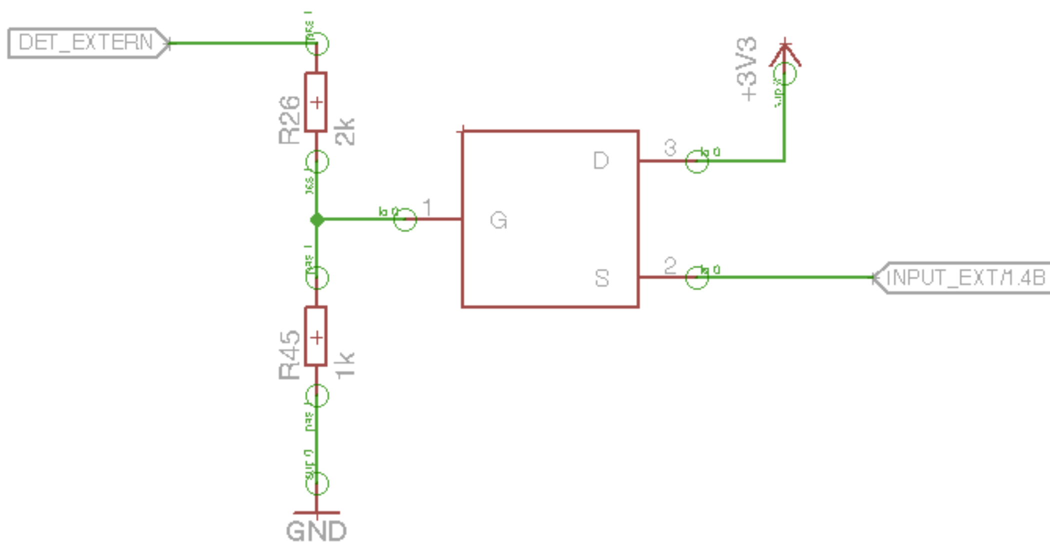


Abb. 3.18: Spannungsdetektion externe Batterie

Die Spannungsdetektion wird im Rahmen dieser Abschlussarbeit allerdings nicht weiter in der Software beachtet oder ausgewertet.

3.4 Display

3.4.1 LCD-Modul

Für eine Statusanzeige bzw. aktuelle Informationen des Batteriesteuergerätes und um die bereits erwähnte Steuerung über die vier Buttons (vgl. Kapitel 3.4.2 und Kapitel 4.6.2) zu ermöglichen und damit die Bedienung unabhängig von einem PC machen zu können, wird ein Display benötigt. Eingesetzt wird für diesen Zweck ein LCD-Modul der Firma Electronic Assembly mit der Modellbezeichnung EA W204B-NLW [25].

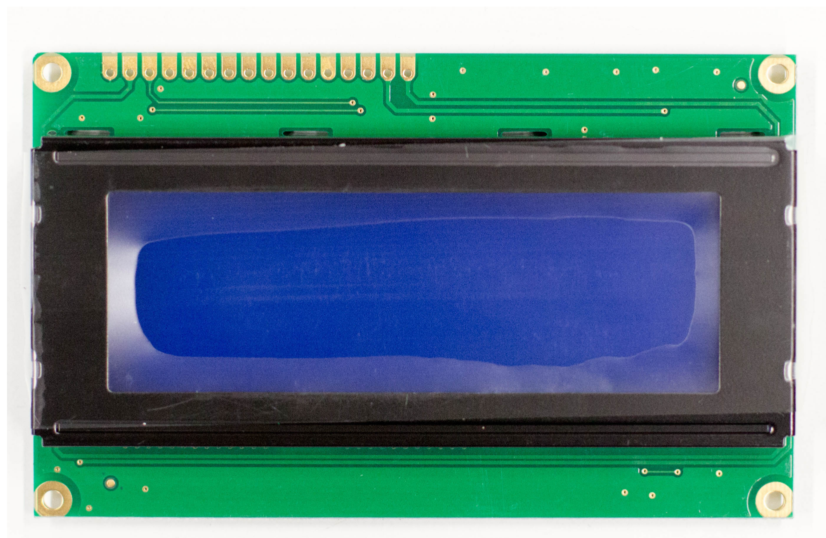


Abb. 3.19: Display EA W204B-NLW

Die Entscheidung für dieses Display wurde aufgrund von bereits vorhandener Vorarbeit in der Bachelorthesis von Herrn Wisniewski [15] sowie der damit verbundenen Vereinheitlichung der Steuerungssoftware im Projekt BATSEN getroffen. Zudem benötigt das Display durch den Einsatz von LEDs für die Hintergrundbeleuchtung nur einen max. Strom von 45mA [25].

Um dieses LCD-Modul mit dem Batteriesteuergerät verbinden zu können, werden 16 bzw. 12 Anschlüsse, abhängig davon ob 4-Bit-Modus oder 8-Bit-Modus verwendet wird, benötigt.

Pin	Funktion
1,16	GND
2	Versorgungsspannung 5V
15	LED-Beleuchtung 3V-3,6V
3	Displayspannung 0V-0,5V
4,5,6	RS-, R/W-, Enable-Pin
7-14	Data

Tabelle 3.3: Pinbelegung LCD-Modul

Das LCD-Modul wird in dem Batteriesteuergerät im 8-Bit-Modus betrieben und somit werden alle 16 Anschlüsse verwendet. Die Pins 4-14 werden direkt an den LM3S9D92 angebunden. Die Datenleitungen an den Pins 7-14 sind verbunden mit den GPIOs PJ0-PJ7 und den Pins 4-6 an den GPIOs PH0-PH2.

Für die LED-Beleuchtung (Pin 15) ist es notwendig, einen strombegrenzenden Widerstand auf dem Batteriesteuergerät einzubauen. Der max. Strom darf 45mA nicht überschreiten [25].

Die Dimensionierung für den Vorwiderstand R_{LED} (R15) ergibt sich aus:

$\boldsymbol{R_{LED}}$: $\boldsymbol{Vorwiderstand}$
$\boldsymbol{U_{LED}}$: $\boldsymbol{LED - Versorgungsspannung = 3,3V}$
$\boldsymbol{I_{max.}}$: $\boldsymbol{Max. LED - Strom = 45mA}$

$$R_{LED} = \frac{U_{LED}}{I_{max.}} \quad (3.8)$$

$$R_{LED} = \frac{3,3V}{45mA} = 73,3\Omega \quad (3.9)$$

Da im Labor die E-Reihe 24 zur Verfügung steht, wird ein 82Ω-Widerstand als Vorwiderstand gewählt.

Die Displaybeleuchtung an Pin 3 wird mittels einer PWM generiert, vgl. dazu Kapitel 3.4.3.

Durch den im LCD-Modul eingebauten internen Controller HD44780 ist die Steuerung einfach gehalten [25]. Im Speicher des Controllers sind 192 fest einprogrammierte Zeichen vorhanden, die mittels eines 8 Bit langen HEX-Wertes über die Datenbits DB0-DB7 übertragen werden und in ein gewünschtes Feld des Displays geschrieben werden können. Die Hex-Werte, siehe Abb. 3.20, sind identisch mit den ASCII-Werten.

ZEICHENSATZ

Upper 4 bit / Lower 4 bit	0000 ($0x)	0010 ($2x)	0011 ($3x)	0100 ($4x)	0101 ($5x)	0110 ($6x)	0111 ($7x)	1010 ($Ax)	1011 ($Bx)	1100 ($Cx)	1101 ($Dx)	1110 ($Ex)	1111 ($Fx)
xxxx0000 ($x0)	CG RAM (0)		0	@	P	`	p		ー	タ	ミ	α	p
xxxx0001 ($x1)	(1)	!	1	A	Q	a	q	。	ア	チ	ム	ä	q
xxxx0010 ($x2)	(2)	"	2	B	R	b	r	「	イ	ツ	メ	β	θ
xxxx0011 ($x3)	(3)	#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
xxxx0100 ($x4)	(4)	$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
xxxx0101 ($x5)	(5)	%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
xxxx0110 ($x6)	(6)	&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111 ($x7)	(7)	'	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
xxxx1000 ($x8)	CG RAM (0)	(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
xxxx1001 ($x9)	(1)	)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010 ($xA)	(2)	*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
xxxx1011 ($xB)	(3)	+	;	K	[	k	{	ォ	サ	ヒ	ロ	ˣ	万
xxxx1100 ($xC)	(4)	,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
xxxx1101 ($xD)	(5)	-	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
xxxx1110 ($xE)	(6)	.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
xxxx1111 ($xF)	(7)	/	?	O	_	o	←	ッ	ソ	マ	゜	ö	█

Abb. 3.20: Zeichensatz des LCD-Moduls [25]

Die Kommunikation zwischen dem LM3S9D92, dem LCD-Controller und dem Display findet immer nach einem festen Protokoll statt. Es werden zunächst die Daten auf den Bus (DB0-DB7), das Enable-Signal auf „High“ sowie das R/W- und RS-Bit auf den gewünschten Wert, welcher vom auszuführenden Befehl (Lesen/Schreiben/Registeransprache) abhängt, gesetzt [25]. Mit der darauffolgenden fallenden Flanke des Enable-Signals übernimmt der LCD-Controller die Daten und setzt diese auch um. Wenn z.B. ein Zeichen gesetzt werden soll, wird dieses bei der fallenden Flanke des Enable-Signals vom LCD-Controller an das Display geschickt.

3.4.2 Push-Buttons

Die bereits in Kapitel 3.4.1 erwähnten Push-Buttons, welche für die Steuerung des Displays notwendig sind, werden aus Gründen der Vereinheitlichung im BATSEN-Projekt und der bereits erfolgreichen Erprobung im Zyklierprüfstand von Herrn Wisniewski übernommen [15]. Wie dort bereits festgestellt, müssen die Taster entprellt werden. Eine Messung (Abb. 3.21) ergab, dass die Taster eine Prellzeit von ca. 1ms haben.

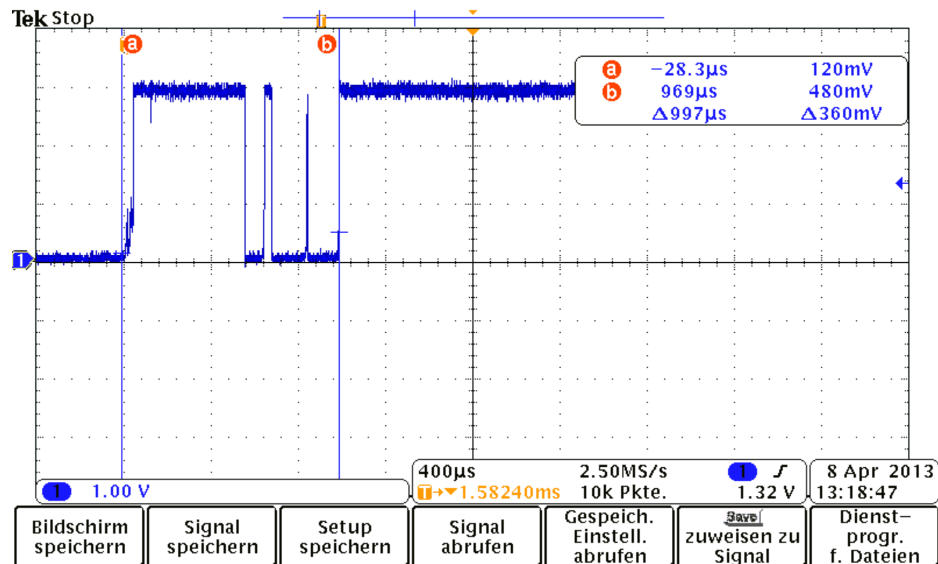


Abb. 3.21: Prellen der Taster [15]

Dieses Prellverhalten würde im Mikrocontroller unerwünschte Interrupts auslösen und so zu Fehlfunktionen im Programm des Batteriesteuergerätes führen. Um dieses Prellen zu unterdrücken, wird zunächst ein Tiefpass parallel zu den Buttons geschaltet. Dadurch wird es allerdings nicht mehr möglich sein, eine steile Signalflanke, die für eine Interruptauslösung benötigt wird, beim Betätigen eines Tasters zu generieren. Daher wird für die Generierung einer Signalflanke ein Schmitt-Trigger in Reihe hinter die Taster geschaltet.

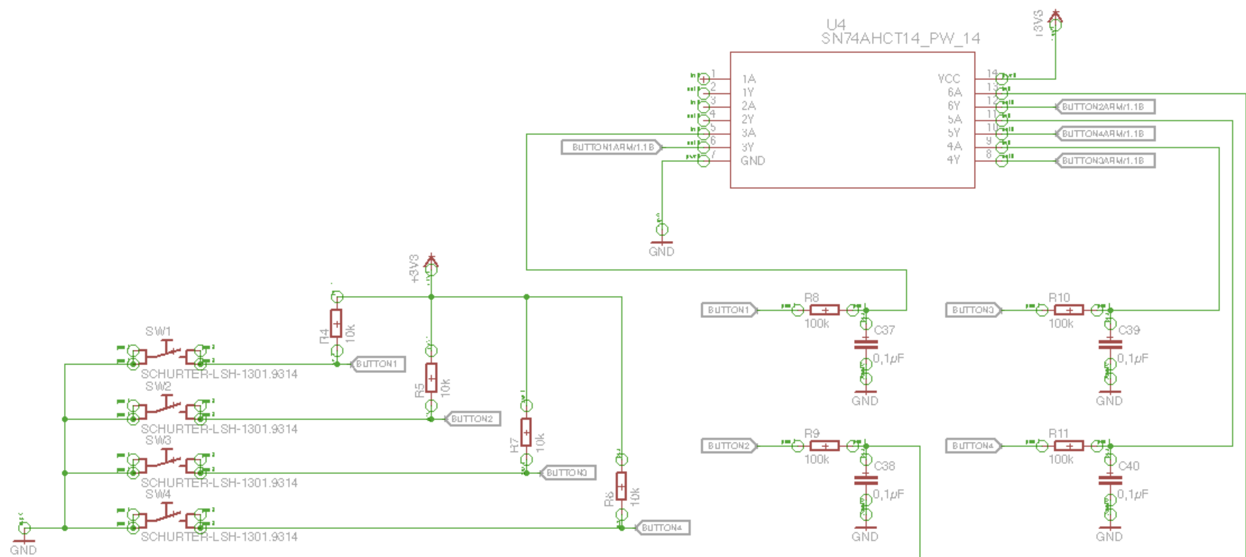


Abb. 3.22: Schaltplan Push-Buttons

Abb. 3.22 zeigt den Schaltplan und damit den generellen Funktionsaufbau der Push-Buttons. Bei Betätigen eines Tasters wird das jeweilige Signal auf das Potential der Masse gezogen, auch Pull-Down genannt. Dabei wird der Kondensator aus dem parallel geschalteten Tiefpass entladen. Um das Prellen zu verhindern, ist der Tiefpass großzügig mit einer Zeitkonstante von 0,01s dimensioniert (3.10).

$$\boldsymbol{\tau = R \cdot C = 100k\Omega \cdot 0,1\mu F = 0,01s} \qquad (3.10)$$

$$\boldsymbol{U(t) = U_{VCC} \cdot e^{-\frac{t}{\tau}} = 3,3V \cdot e^{-\frac{t}{0,01s}}} \qquad (3.11)$$

Der hier eingesetzte invertierende Schmitt-Trigger SN74AHC14 der Fa. Texas Instruments hat bei fallender Spannung eine Schaltschwelle zwischen 0,6V und 1,7V [26].

$$\boldsymbol{t = -\tau \cdot (ln(U(t)) - ln(U_{VCC}))} \qquad (3.12)$$

Damit muss nach Formel (3.12) der Schmitt-Trigger nach einer Zeit von min. 6,6ms und max. ca. 17ms schalten.

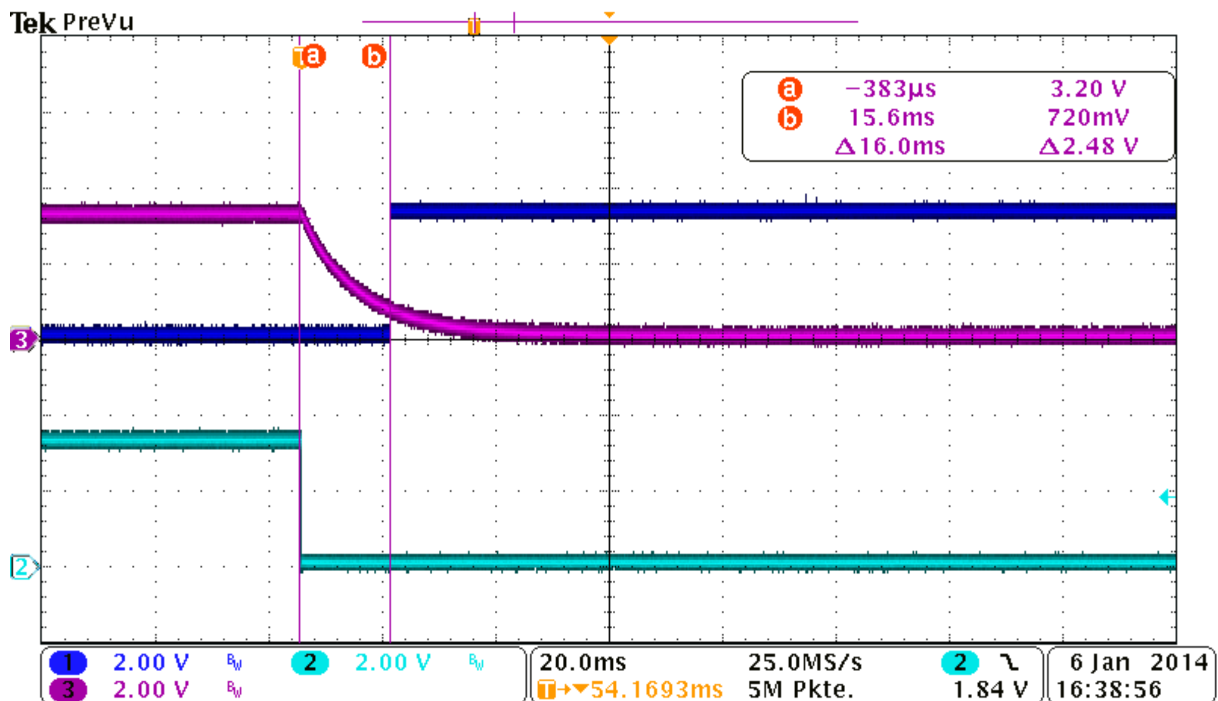


Abb. 3.23: Reaktionzeit bei Tastendruck

Die Abb. 3.23 zeigt die erwähnte Zeit zwischen dem Schließen des Schalters (Channel #2 türkis, untere Linie) und der Reaktion des Schmitt-Triggers (Channel #1 blau, mittlere Linie). Es dauert ca. 16ms, bis sich der Kondensator soweit entlädt (Channel 3 lila, obere Linie), dass die Schwellspannung des Schmitt-Triggers unterschritten wird und dieser schaltet. Diese Zeit resultiert, wie in Gleichung (3.12) berechnet, aus der Entladezeit des Kondensators und kann bei Bedarf angepasst werden, indem ein anderer Widerstands- bzw. Kapazitätswert auf der Platine eingelötet wird. Dabei muss allerdings stets das Prellen des Tasters beachtet werden; sobald die Lade- bzw. Entladezeit kürzer wird als die Prelldauer der Taster, gibt der Schmitt-Trigger mehrere Impulse, welche zu einer falschen Interruptauslösung im Mikrocontroller führen können.

3.4.3 Generierung der Displayspannung

Wie bereits in Kapitel 3.4.1 erwähnt, wird für die Displayspannung eine Spannungsversorgung zwischen 0V und 0,5V benötigt. Da der verwendete ARM Mikrocontroller interne PWM-Ausgänge hat, werden diese genutzt, um die benötigte Spannung zu generieren.

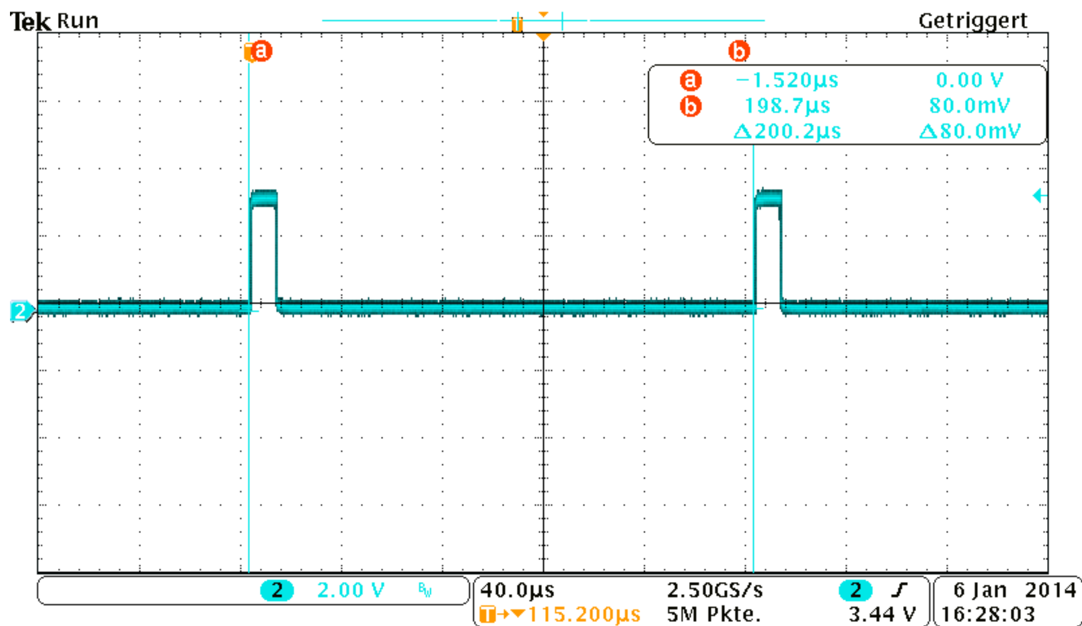


Abb. 3.24: PWM-Signal Periodendauer

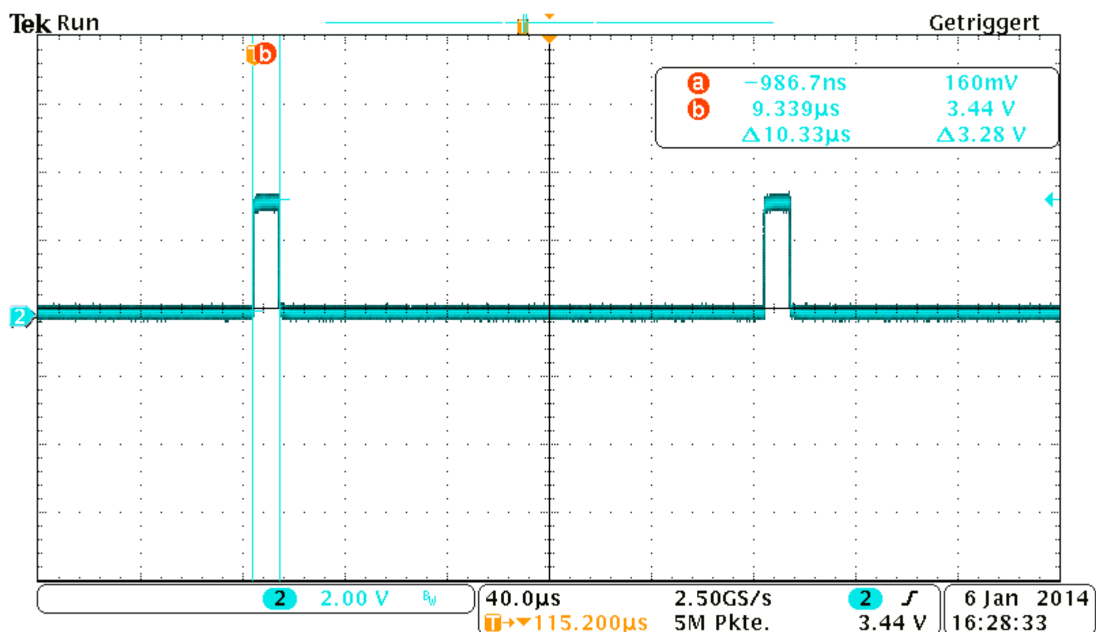


Abb. 3.25: PWM-Signal Tastgrad

Die Abb. 3.24 und Abb. 3.25 zeigen sowohl die Periodendauer, als auch den Tastgrad des im Mikrocontroller parametrisierten PWM-Signals. Bei einer Periodendauer von 200µs ergibt sich eine Frequenz von 5kHz. Durch die Impulsdauer von 10µs ergibt sich daraus ein Tastgrad von 0,05 bzw. 5%. Damit liegt der Mittelwert bei 3,3V•0,05, was 165mV entspricht und im vom Display zulässigen Bereich zwischen 0V und 0,5V liegt.

3.5 FTDI-Programer

Zur Programmierung des Mikrocontrollers steht eine JTAG-Schnittstelle zur Verfügung. Allerdings hat sich bereits in den Laborversuchen zu der Veranstaltung „Mikrocontrollertechnik“ sowie in der Abschlussarbeit von Herrn Schlüter [14] gezeigt, dass dies viel komfortabler mit Hilfe des FTDI-Chips funktioniert. Dieser stellt über ein In-Circuit-Debug-Interface eine Verbindung zwischen dem JTAG und einem COM-Port des PCs über den Universal-Serial-Bus (USB) her.

Es bringt den Vorteil des einfachen Debuggens von dem Batteriesteuergerät sowie eine einfache Programmierung vom PC aus über die USB-Schnittstelle. Die Beschaltung des FTDI-Chips ist von dem Evaluation-Kit EKI-LM3S9B92 nachempfunden und findet bereits bei dem Zyklierprüfstand von Herrn Wisniewski Verwendung [15].

Die Programer-Schaltung besteht aus einem EEPROM, der für die korrekte Funktion noch vor der Inbetriebnahme programmiert werden muss, vgl. Kapitel 4.1, dem FT2232D Chip sowie einigen Multiplexern. Für die genaue Erläuterung dieser Schaltung wird an dieser Stelle auf die Bachelorarbeit von Herrn Schlüter verwiesen [14].

3.6 Schnittstellen

Wie der Zielsetzung dieser Arbeit bereits zu entnehmen ist, sollen möglichst viele Schnittstellen in das Batteriesteuergerät integriert werden. Im Kapitel 3.1 werden die verwendeten Schnittstellen in Abb. 3.1 als ein Block „Schnittstellen“ zusammengefasst.

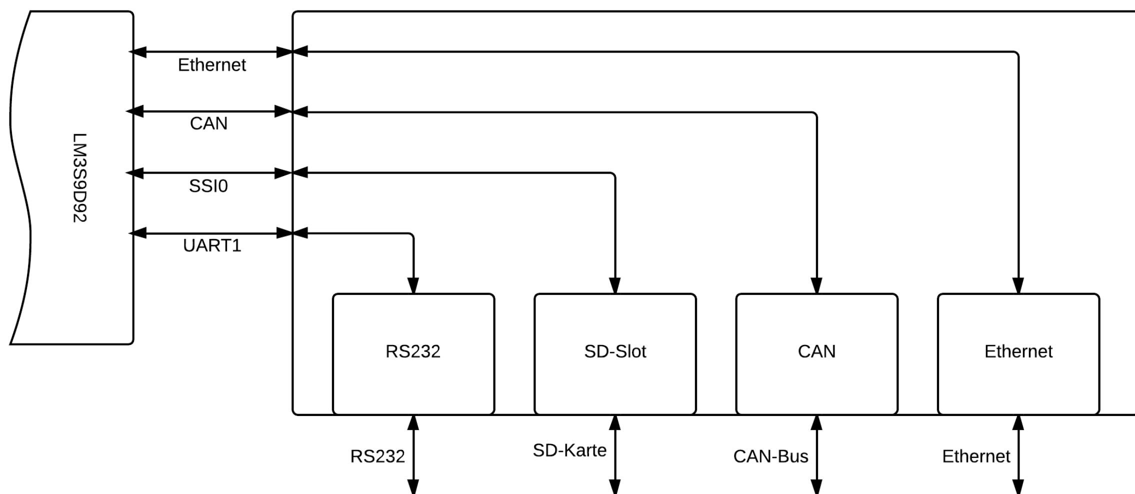


Abb. 3.26: Schnittstellen des Batteriesteuergerätes

Die Abb. 3.26 veranschaulicht dieses Subsystem genauer und zeigt wie die einzelnen Schnittstellen mit dem Mikrocontroller verbunden sind. In den nachfolgenden Kapiteln wird auf die Integration der jeweiligen Schnittstellen genauer eingegangen.

3.6.1 RS232

Der RS232 ist ein serieller Bus, der zwei Teilnehmer direkt miteinander verbindet. Dieses Interface wird verwendet, um das Batteriesteuergerät mittels eines PCs, unter Verwendung einer einfachen Terminalanwendung, steuern zu können. Dazu muss zunächst der PC über einen seriellen COM-Port an das Batteriesteuergerät angeschlossen und über ein Terminal-Programm, welches RS232-fähig ist, mit dem Batteriesteuergerät verbunden werden. Als Terminal-Programm wird für diese Arbeit das Programm HTerm verwendet [17]. Die Konfiguration des UART-Ports des Mikrocontrollers auf dem Batteriesteuergerät ist folgende:

- 115200 BAUD Übertragungsrate
- 8-Bit Datenlänge
- ein Stop-Bit
- keine Paritätsprüfung

Diese Konfiguration muss bei beiden Teilnehmern an dem RS232 Bus, in dem Fall der PC mit einem Terminal-Programm und der Mikrocontroller des Batteriesteuergeräts, konfiguriert werden.

Da die RS232-Signalpegel eine höhere Spannung haben als am Mikrocontroller zulässig, muss ein RS232-Transceiver als Pegelwandler in das Batteriesteuergerät integriert werden. Für diese Pegelwandlung wird der MAX3232IDWR der Fa. Texas Instruments eingesetzt.

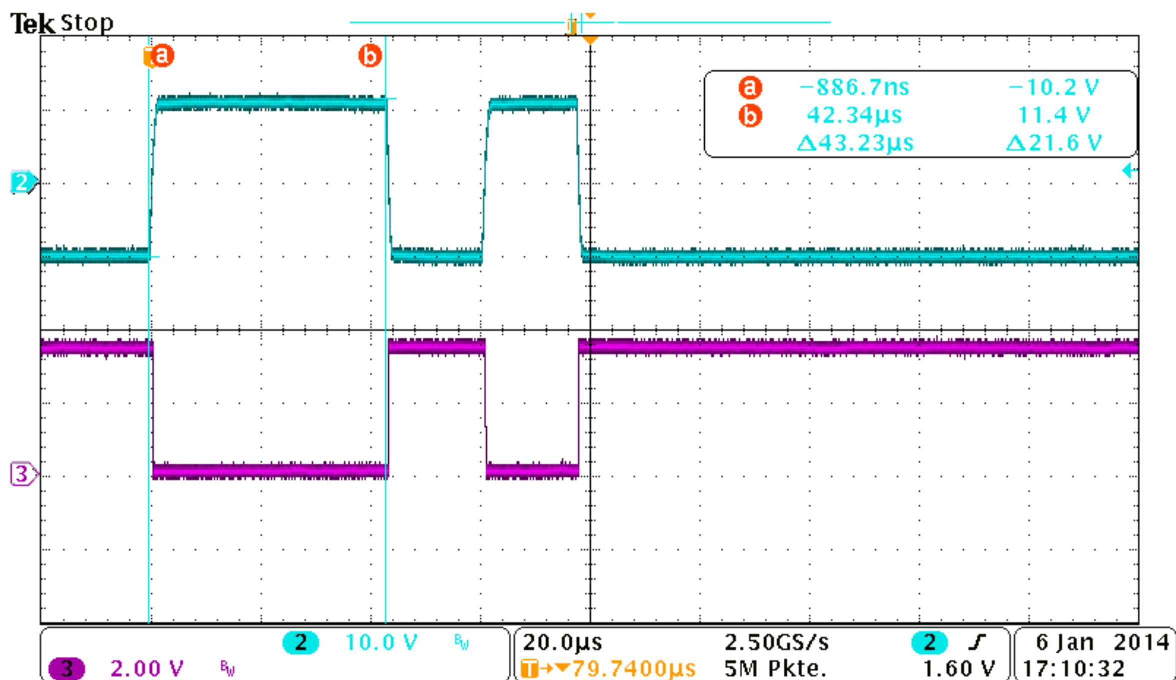


Abb. 3.27: Spannungspegel beim Senden über RS232

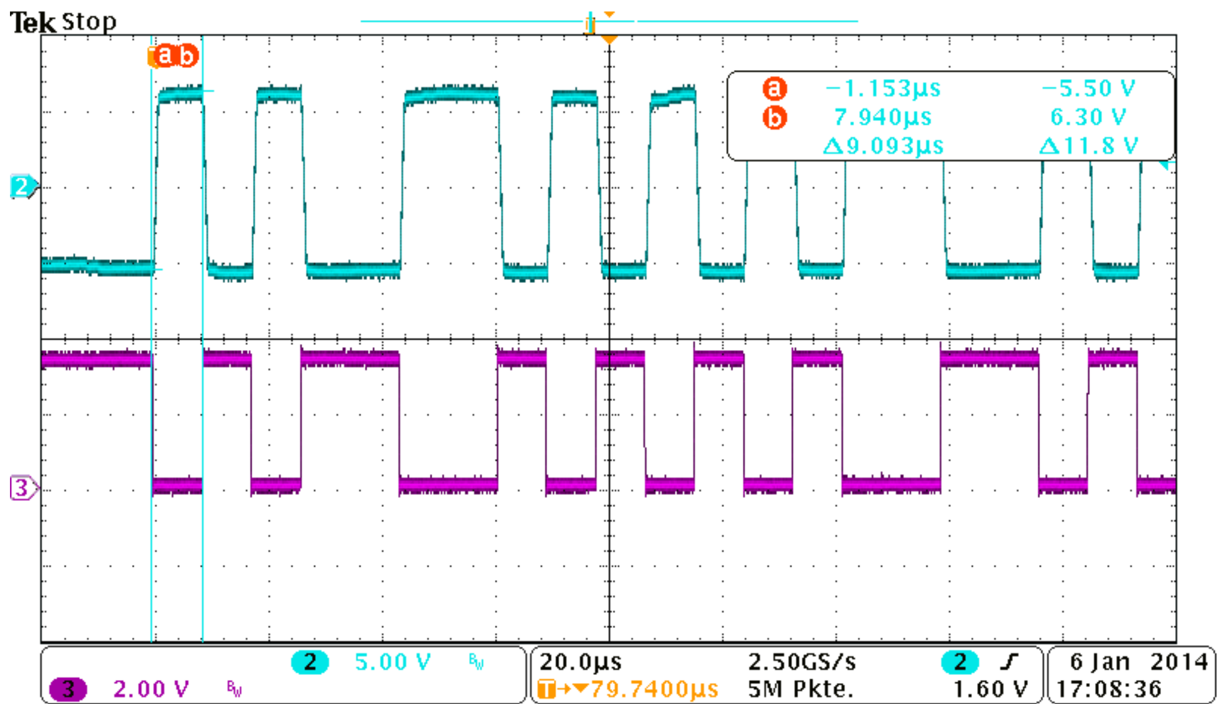


Abb. 3.28: Spannungspegel beim Empfangen über RS232

Die Abb. 3.27 und Abb. 3.28 zeigen jeweils die Spannungspegel beim Senden und Empfangen über den RS232 Bus. (Channel #3, untere Linie, lila) zeigt jeweils das vom Mikrocontroller generierte UART-Signal und (Channel #2, obere Linie, türkis) zeigt das vom Transceiver gewandelte Signal. Beim Senden hat das Signal einen Pegel von -10V als „Low“-Pegel und +11V als „High“-Pegel und beim Empfangen sendet der PC über den RS232 Bus mit einem Pegel von -5,5V als „Low“-Pegel und +6,3V als „High“-Pegel.

Der Anschluss an das Batteriesteuergerät für den RS232 erfolgt nicht wie gewöhnlich über einen SUB-D9-Stecker, sondern wird aus Platzspargründen über 2 einfache Pinheader nach außen geführt, an denen zukünftig beispielsweise ein Anschluss an ein externes Gehäuse eingebaut werden kann.

3.6.2 SD-Karte

Für die Aufzeichnung der Messdaten, wie z.B. die Daten der Strommessung über einen Hall-Sensor und Spannungsmessung über die Zellsensoren, ist ein Anschluss für eine SD-Karte eingebaut. Um Platz zu sparen, siehe Abb. 3.29, fiel die Entscheidung auf ein microSD-Kartenslot der Fa. Würth Elektronik.

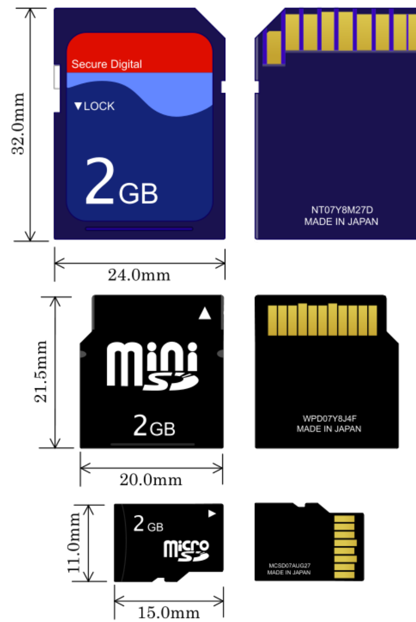


Abb. 3.29: Größenvergleich SD-Karten [27]

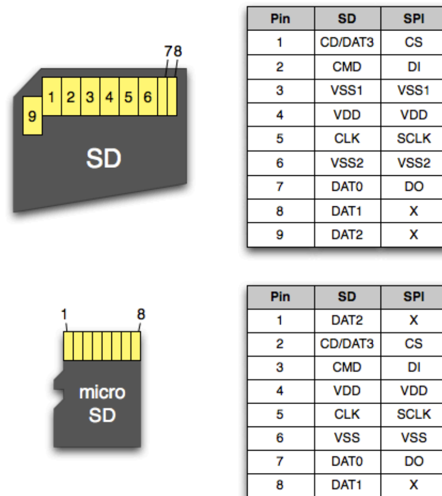
$$A_{SD} = 32mm \cdot 24mm = 768mm^2 \quad (3.13)$$

$$A_{miniSD} = 21{,}5mm \cdot 20mm = 430mm^2 \quad (3.14)$$

$$A_{\mu SD} = 15mm \cdot 11mm = 165mm^2 \quad (3.15)$$

Damit ist die microSD-Karte nach Formel (3.13) und Formel (3.15) um mehr als das vierfache kleiner bei gleicher Funktion. Elektrisch sowie bei der Ansteuerung der Karten gibt es zwischen den verschiedenen Kartengrößen keine Unterschiede (Abb. 3.30). Zum Zeitpunkt[8] der Entwicklung des Batteriesteuergerätes liegt die max. Speichergröße bei microSD-Karten bei 64GB.

[8] Stand: 2013



Pin	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	DO
8	DAT1	X
9	DAT2	X

Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

Abb. 3.30: SD-Karten Pinbelegung [28]

Als Dateisystem wird das FAT-Filesystem verwendet, für das bereits vorgefertigte Bibliotheken in der StellarisWare Software für den LM3S9D92 existieren. Je nachdem, wie die SD-Karte formatiert ist, werden hier die Formate FAT12, FAT16 sowie FAT32 unterstützt.

Gelesen sowie beschrieben wird die SD-Karte über den SPI-Bus des Mikrocontrollers. Der SPI-Takt wird hierfür mit 12,5MHz betrieben und ist damit auf den max. möglichen Takt gesetzt. Somit erzielen wir eine Übertragungsrate von 12,5MBaud/s, die im Falle vom SPI-Bus einer Nutzübertragungsrate von 12,5MBit/s entspricht. Dies ist der Fall, da in einem Datenblock 8 Bits übertragen werden, die alle Nutzdaten sind. Es gibt beispielsweise keine Stop- oder Startbits.

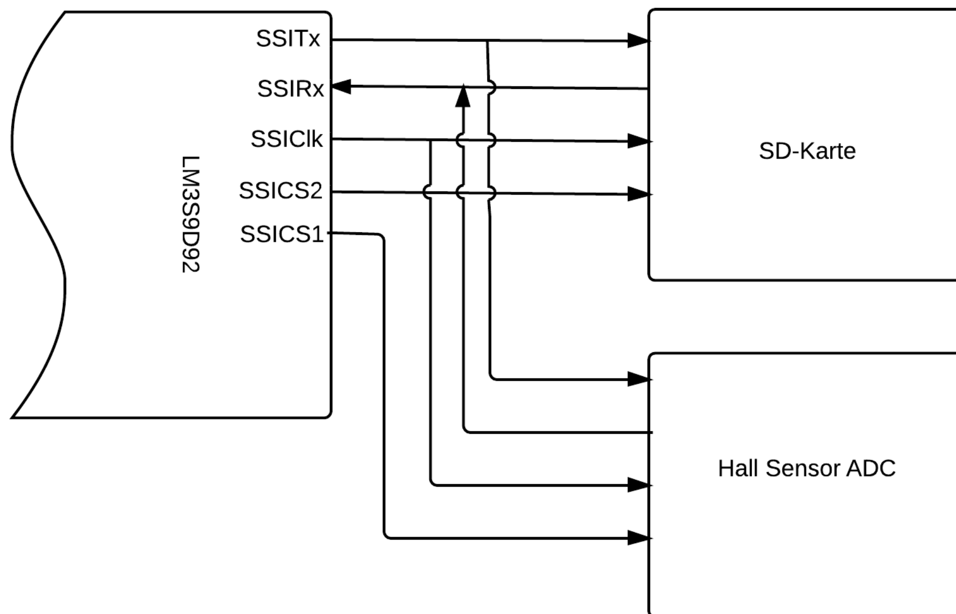


Abb. 3.31: Mehrfachverwendung des SPI-Busses

Wie in Kapitel 3.2 bereits aufgezeigt, stehen dem Batteriesteuergerät nur zwei SPI Anbindungen zur Verfügung. Da aber die SD-Karte, der ADC für den Hall-Sensor und das Transceiver-Modul jeweils über den SPI-Bus kommunizieren, wird ein SPI-Bus doppelt genutzt. Die SD-Karte und der ADC für den Hall-Sensor teilen sich einen BUS. Dazu muss, wie in Abb. 3.31 dargestellt, jeweils ein Chip-Select-Signal gesetzt werden, bevor die Kommunikation stattfinden kann. Mit dem Setzen des SSICS1 wird die Kommunikation mit dem ADC des Hall-Sensors freigeschaltet und mit dem SSICS2 wird die SD-Karte freigeschaltet. Es muss hierbei ausgeschlossen werden, dass beide CS-Signale gleichzeitig aktiv sind.

Als zusätzliche Funktion besitzt der SD-Kartenslot zwei weitere Pins, die der Detektion der SD-Karte dienen. Sobald eine SD-Karte in den Slot geschoben wird, öffnet sich der Schalter (Abb. 3.32) und der CARD_DET_PIN ist auf GND-Potential. Wenn allerdings die SD-Karte entfernt wird, schließt sich der Schalter und es liegen 3,3V am GPIO-Pin PA6 des Mikrocontrollers an (siehe Abb. 3.33). Diese Funktion wird in dieser Arbeit allerdings nicht in der Software implementiert, steht aber für zukünftige Versionen des Batteriesteuergerätes zur Verfügung.

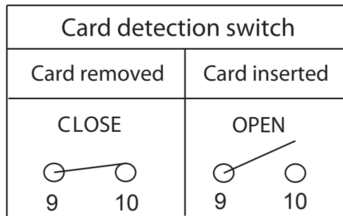


Abb. 3.32: Detektion SD-Karte

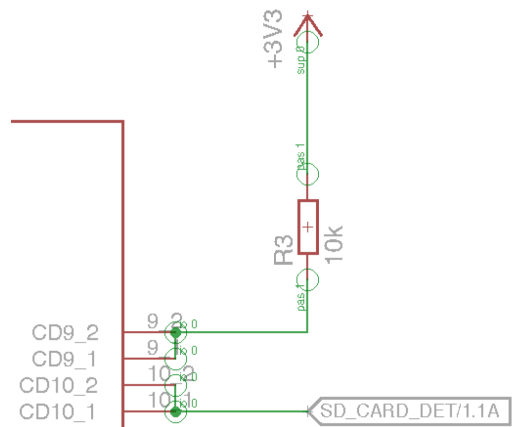


Abb. 3.33: Detektion SD-Karte Schaltplan

3.6.3 CAN-Bus

Der CAN-Bus ist ein serielles Bussystem, welches einen Standard im Automobilbereich bzw. in sicherheitsrelevanten Systemen, wie sie in der Flugzeug-, Raumfahrt- oder der Medizintechnik zu finden sind. Da das Batteriesteuergerät in Zukunft in diesen Bereichen zum Einsatz kommen kann, ist der CAN-Bus hardwareseitig implementiert worden. Als CAN-Transceiver wird ein SN65HVD1050 der Fa. Texas Instruments verwendet [29]. Dieser besitzt eine max. Übertragungsgeschwindigkeit von 1Mbit/s.

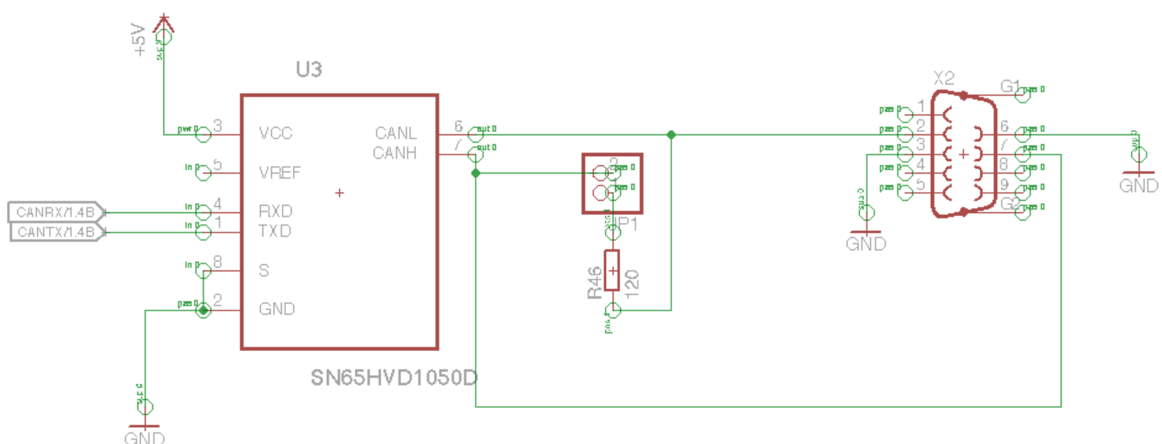


Abb. 3.34: Schaltplan CAN-Bus

Die Abb. 3.34 zeigt den Schaltplan des CAN-Busses auf dem Batteriesteuergerät. Die Versorgungsspannung des CAN-Transceivers beträgt 5V. Zudem besteht die Möglichkeit, den CAN-Bus mit einem 120Ω Widerstand (R46) über den Jumper (JP1) abzuschließen. Falls ein anderer Abschlusswiderstand benötigt wird, kann dieser anstelle des 120Ω Widerstandes eingelötet werden. Als Anschlussstecker für den CAN-Bus wird auf dem Batteriesteuergerät ein SUB-D9-Stecker eingesetzt.

3.6.4 Ethernet

Damit in der Zukunft eine Datenübertragung direkt über Ethernet zu ermöglichen und so z.B. Messdaten direkt über Internet/LAN verschicken zu können, ist ein Ethernet-Anschluss (RJ45-Stecker) auf dem Batteriesteuergerät bereits hardwareseitig vorgesehen. Der Mikrocontroller besitzt einen eigenen Ethernet-Controller, der den Media Access Controller (MAC) sowie das Network Physical Interface (PHY) implementiert hat. Diese entsprechen den OSI-Schichten 1 und 2. Als Übertragungsrate wird max. 100Mbit/s im 100BASE-TX Standard unterstützt [16].

Zusätzlich benötigt der Ethernet-Controller einen externen 25MHz Quarz zwischen den XTALNPHY und XTALPPHY Pins an des Mikrocontrollers. Verwendet wird der ABRACON ABM7 25MHz Quarz [30].

3.7 Strommessung

Damit, wie in der Einleitung dieser Arbeit bereits beschrieben, der Lade- sowie Alterungszustand erfasst werden kann, wird zusätzlich zu der Spannungsmessung der Batteriezellen eine Strommessung benötigt. Eine in der Automobilindustrie weit verbreitete und auch einfach umzusetzende Methode bietet ein Hall-Sensor. Legt man den zu messenden Leiter durch den Hall-Sensor, wird das Magnetfeld des Leiters gemessen. Dieses Magnetfeld liefert am Sensorausgang eine Spannung, welche proportional zum Produkt aus magnetischer Feldstärke und Strom ist.

Eingesetzt wird der Hall-Sensor DHAB S/24 der Fa. LEM [31]. Die Entscheidung auf diesen Sensor ist aufgrund der bereits vorhandenen positiven Erfahrungen im BATSEN-Projekt sowie der damit verbundenen Erprobung und Vereinheitlichung auf diesen Sensor gefallen. [15]

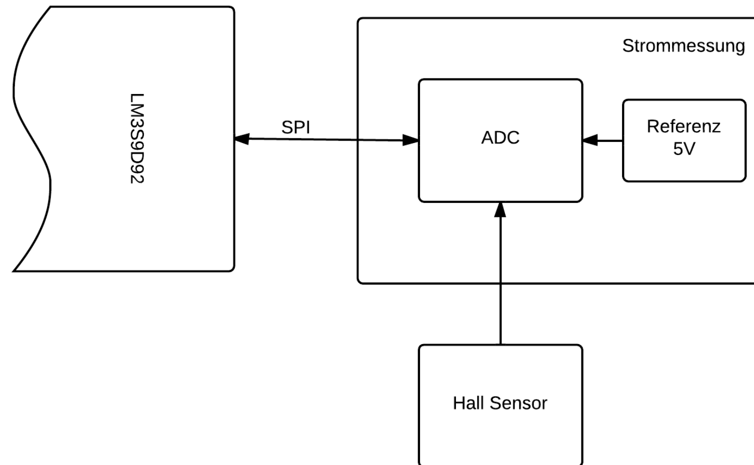


Abb. 3.35: Blockschaltbild Strommessung

Der Sensor hat zwei Kanäle, die eine Strommessung im Bereich ±75A bzw. ±500A ermöglichen. Als Spannungsversorgung benötigt der Hall-Sensor 5V. Die beiden Kanäle müssen für die Weiterverarbeitung auf dem Batteriesteuergerät mit einem ADC umgewandelt werden. Dazu wird ein externer 16-Bit ADC, AD7798 der Fa. Analog Devices, auf dem Batteriesteuergerät verwendet [32]. Damit dieser ein möglichst genaues Ergebnis liefert, wird als 5V-Referenzspannung für den ADC die Referenzspannungsquelle ADR4550 der Fa. Analog Devices, verwendet [33]. Diese bietet eine Ausgangsspannung von 5V mit einer max. Abweichung von 2mA [34]. Daraus folgt für den ADC eine Auflösung von 76,29µV/Bit. (3.17)

$$U_{Ref} = 5V$$
$$n = 2^{16}\,Bit$$

$$Auflösung = \frac{U_{Ref}}{n} \quad (3.16)$$

$$Auflösung = \frac{5V}{2^{16}Bit} = 76{,}29\mu V/Bit \quad (3.17)$$

Damit wird eine ausreichend gute Auflösung erreicht. Der am ADC entstehende Fehler durch Quantisierung ist hierbei zu vernachlässigen, da am Hall-Sensor, bedingt durch seine Funktionsweise, ein signifikant höherer Fehler entsteht.

Global Absolute Error (A)

Channel 1	Global Absolute Error (A)					
Temperature	-40	-20	0	25	65	125
Global Offset Error	± 0.68	± 0.58	± 0.48	± 0.35	± 0.55	± 0.85
Global Error @±40A	± 2.17	± 1.75	± 1.33	± 0.80	± 1.64	± 2.90
Global Error @±75A	± 3.25	± 2.65	± 2.05	± 1.30	± 2.50	± 4.30

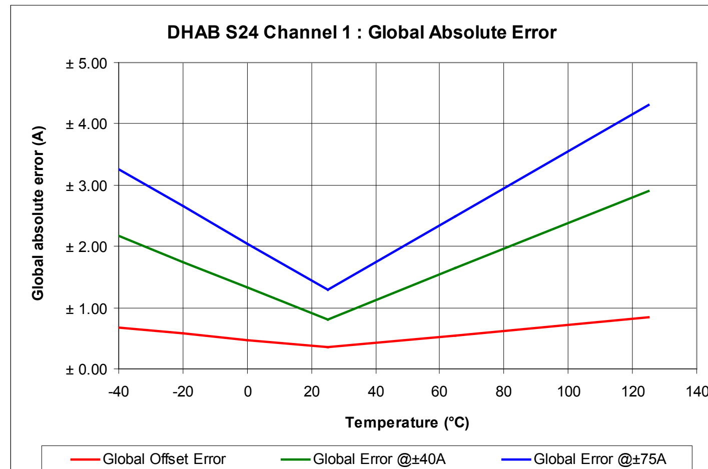


Abb. 3.36: Absoluter Fehler Hall-Sensor ±75A [31]

Global Absolute Error (A)

Channel 2	Global Absolute Error (A)					
Temperature	-40	-20	0	25	65	125
Global Offset Error	± 4.65	± 4.45	± 4.25	± 4.00	± 4.40	± 5.00
Global Error @±200A	± 10.88	± 9.38	± 7.88	± 6.00	± 9.00	± 13.50
Global Error @±500A	± 17.43	± 14.53	± 11.63	± 8.00	± 13.80	± 22.50

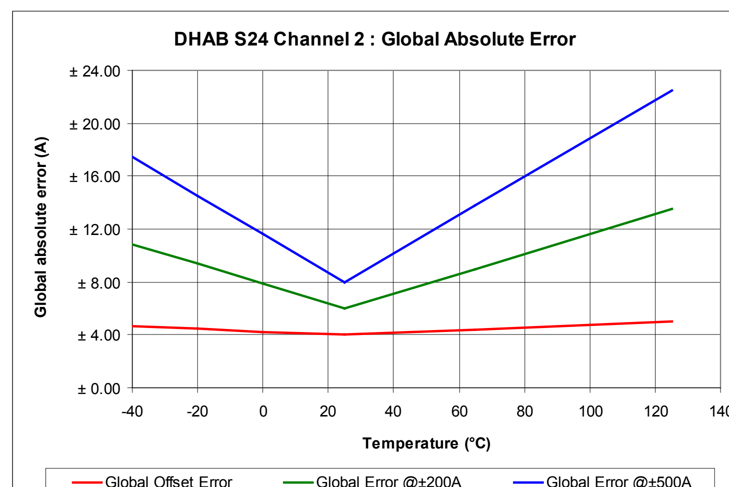


Abb. 3.37: Absoluter Fehler Hall-Sensor ±500A [31]

Die Abb. 3.36 sowie Abb. 3.37 zeigen die Gesamtabweichung der jeweiligen Kanäle. Sowohl der durch den zu messenden Leiter fließende Strom als auch die Umgebungstemperatur beeinflussen die Genauigkeit deutlich. Am genausten arbeitet dieser bei einer Temperatur von ca. 25°C. Sinkt die Temperatur, nimmt die Abweichung schneller zu als bei wärmeren Temperaturen.

Abweichung Ch1 75A	Abweichung Ch2 500A
$\boldsymbol{Abweichung_{25°C,40A} = \frac{0,8A}{40A} = 2\%}$	$\boldsymbol{Abweichung_{25°C,200A} = \frac{6A}{200A} = 3\%}$
$\boldsymbol{Abweichung_{25°C,75A} = \frac{1,3A}{75A} = 1,73\%}$	$\boldsymbol{Abweichung_{25°C,500A} = \frac{8A}{500A} = 1,6\%}$
$\boldsymbol{Abweichung_{125°C,40A} = \frac{2,9A}{40A} = 7,25\%}$	$\boldsymbol{Abweichung_{125°C,200A} = \frac{13,5A}{200A} = 6,75\%}$
$\boldsymbol{Abweichung_{125°C,75A} = \frac{4,3A}{75A} = 5,73\%}$	$\boldsymbol{Abweichung_{125°C,500A} = \frac{22,5A}{500A} = 4,5\%}$

Tabelle 3.4: Relative Gesamtabweichung vom Hall-Sensor in Abhängigkeit von Strom und Temperatur

Die Tabelle 3.4 stellt die min. sowie die max. relative Abweichung der einzelnen Kanäle in Abhängigkeit von der Umgebungstemperatur und dem gemessenen Strom dar. Es wird hier noch einmal deutlich, wie sehr die Temperatur einen Einfluss auf das Messergebnis hat. Aus diesen Gründen sollte man den Hall-Sensor möglichst bei einer Umgebungstemperatur von ca. 25°C betreiben. Falls das Batteriesteuergerät in ein Gehäuse eingebaut wird, muss dementsprechend für eine ausreichende Luftzirkulation gesorgt werden.

3.8 Temperaturmessung

Für die Batteriemessungen spielt die Temperatur der Zellen eine besondere Rolle. Die Temperatur beeinflusst die Leistung der Batterie und ist somit ausschlaggebend für die Berechnung des Lade- und Alterungszustandes. Die Temperaturmessung der Batteriezellen wird von den jeweiligen Zellsensoren übernommen und anschließend an das Batteriesteuergerät übermittelt [7].

Der Hall-Sensor ist, wie in Kapitel 3.7 festgestellt, sehr temperaturabhängig. Die Fehlerabweichung bei der Strommessung steigt bei Temperaturen unterhalb sowie oberhalb von 25°C an. Um den Fehler bei der Auswertung der Messdaten abschätzen und eventuell korrigieren zu können, ist eine Temperaturmessung notwendig.

Für diese zusätzliche Temperaturmessung wurde an dem Batteriesteuergerät ein Anschluss für einen NTC-Widerstand nach außen geführt.

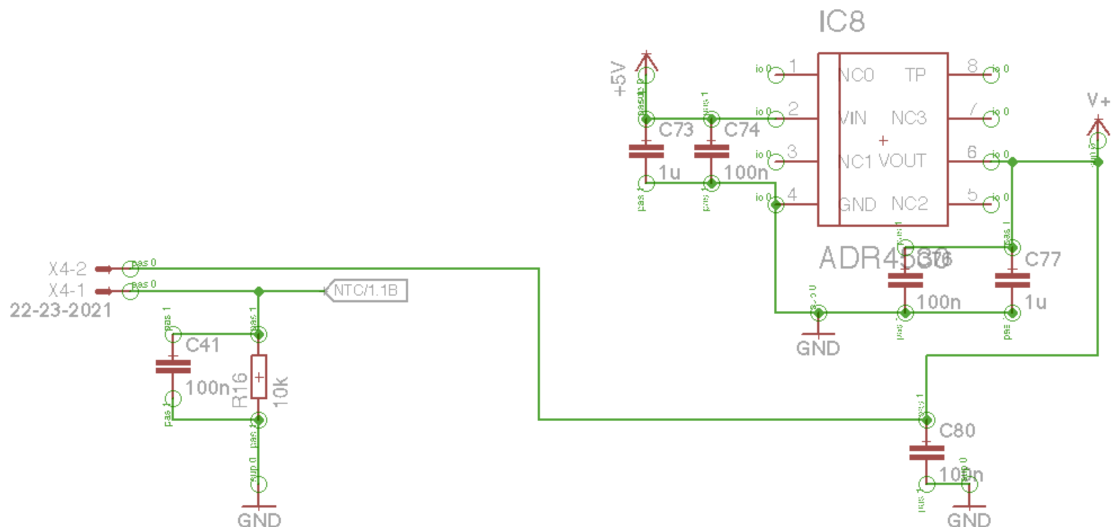


Abb. 3.38: Temperaturmessung an dem Batteriesteuergerät - Schaltplan

Abb. 3.38 zeigt den Schaltplan für die Temperaturmessung. Der NTC-Widerstand wird an den Pins X4-1 und X4-2 angeschlossen. Über einen Spannungsteiler zwischen dem NTC-Widerstand sowie dem Widerstand R16 verläuft die Leitung an den internen 12-Bit ADC des Mikrocontrollers am ADC-Input 0. Der Widerstand R16 ist als 10kΩ Widerstand für den B57703M der Fa. EPCOS vordefiniert [35]. Die Wahl auf diesen NTC-Widerstand erfolgte aufgrund der bereits vorhanden Vorentwicklung und erfolgreichen Implementierung in der Bachelorarbeit von Herrn Wisniewski [15]. Es kann je nach Bedarf auch ein anderer NTC-Widerstand verwendet werden; dazu muss allerdings je nach gewähltem NTC-Widerstand der Spannungsteiler angepasst werden.

Als Spannungsversorgung für den NTC-Widerstand wird die 3V-Referenzspannungsquelle ADR4530 der Fa. Analog Devices verwendet [34].

Die Softwareimplementierung sowie Inbetriebnahme dieser Temperaturmessung wird im Rahmen dieser Arbeit nicht durchgeführt, kann aber mit Hilfe der Dokumentation von Herrn Wisniewski schnell erfolgen.

3.9 Transceiver-Modul

Das bereits in Kapitel 2.1 erwähnte vorhandene Transceiver-Modul wird unverändert übernommen. Dafür muss lediglich die Steckverbindung von dem bisherigen Batteriesteuergerät auf das neue Layout übertragen werden. Dabei ist für die Übernahme der Abstand der einzelnen Pins sowie die Belegung mit den entsprechenden Funktionen zu beachten.

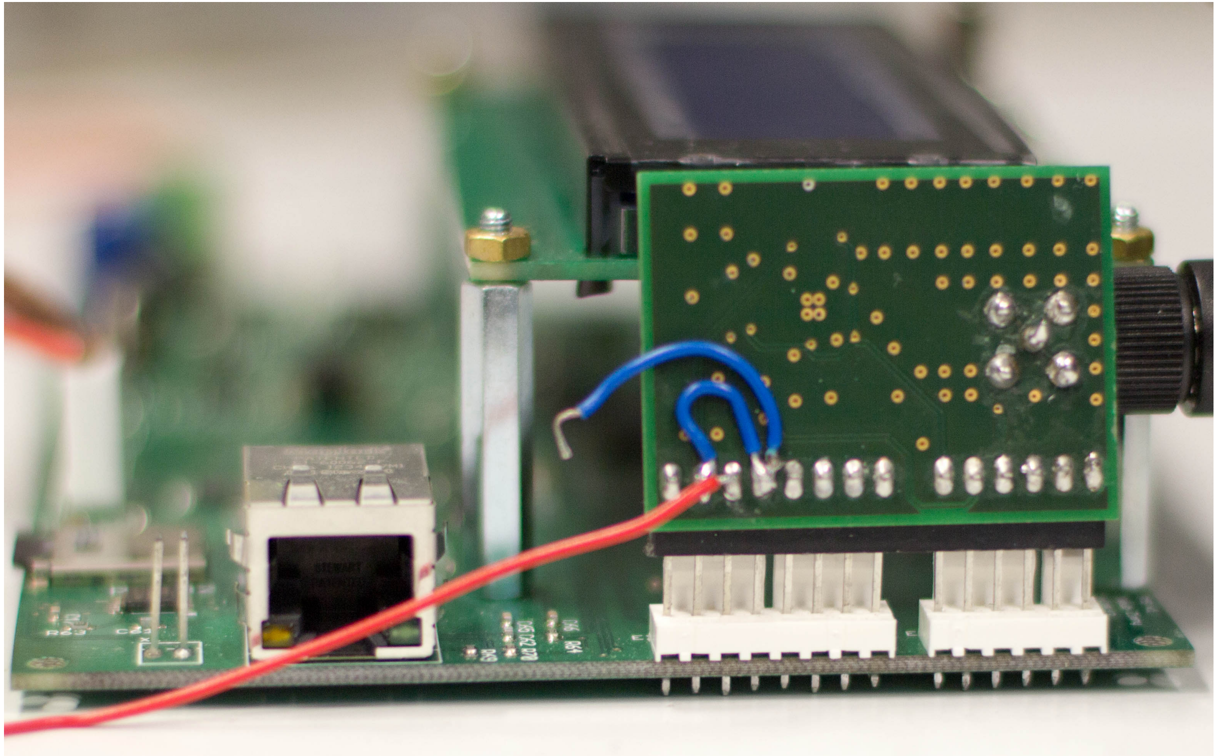


Abb. 3.39: Steckverbindung des Transceiver-Moduls

Die Abb. 3.39 zeigt die Steckverbindung für das Transceiver-Modul auf dem Batteriesteuergerät. Über diese wird das Transceiver-Modul mit 3,3V und GND versorgt. Die restlichen Pins sind als GPIOs an den LM3S9D92 Mikrocontroller an PD2-PD5 sowie an PE5 und PE6 geführt. Die genaue Pinbelegung ist im Schaltplan zu finden, siehe Anhang 10.2.

3.10 Status LEDs

Für eine beliebig konfigurierbare Statusanzeige sind drei LEDs im Batteriesteuergerät vorgesehen. Es handelt sich um 2mA LEDs in der Farbe Rot. Angeschlossen am Mikrocontroller sind sie am GPIO Port B an den Pins PB0-PB2.

3.11 Schaltplan und Platinen-Layout

Das Batteriesteuergerät (vgl. Anhang 10.3) wurde mit dem Programm EAGLE in der Version 6.4. lizenziert für die HAW Hamburg entworfen und weist folgende Eigenschaften auf:

- Größe von 100mm·160mm und entspricht damit dem Format einer Europakarte.
- Top- und Bottom-Layer vorhanden
- Verbindung der beiden Layer über 0,4mm Durchkontaktierungen (Vias)
- dicke der Leiterbahnen: 0,3mm

Aufgrund der Komplexität des Batteriesteuergerätes wurde auf das automatische Routing der Leiterbahnen verzichtet.

4 Softwareentwicklung

4.1 Programmieren des FTDI-Chips

Wie bereits in Kapitel 3.5 erwähnt, wird zum Programmieren sowie Debuggen des LM3S9D92 Mikrocontrollers das In-Circuit-Debug-Interface (ICDI) mit dem FTDI-Chip FT2232D verwendet. Dieser stellt eine Verbindung zwischen dem JTAG-Interface des Mikrocontrollers sowie einem COM-Port an einem PC über USB her.

Dafür muss zunächst auf dem PC der FT2232D-USB-Treiber installiert werden. Zu finden ist dieser unter www.ftdichip.com. Nach erfolgreicher Installation muss der EEPROM des FT2232D-Chips programmiert werden. Dazu wird von dem Hersteller des Chips ein Programm namens „FT-PROG“ zur Verfügung gestellt, ebenfalls zu finden auf der Webseite des Herstellers. Folgende Einstellungen sind in diesem Programm vorzunehmen, bevor der Chip programmiert wird:

- Typ FT2232D setzen
- 245 FIFO mit D2XX Direct Treiber konfigurieren
- Versorgungsspannung des FTDI-Chips auf „Self Powered“ stellen
- Verbindung auf USB to JTAG wählen

Nach erfolgreicher Programmierung kann nun das Batteriesteuergerät über USB an einen PC angeschlossen und programmiert werden.

4.2 Konzept

4.2.1 Zustandsautomat

Das Batteriesteuergerät wird mithilfe einer Zustandsmaschine gesteuert. Dabei wird alle 100ms der aktuelle Zustand überprüft und die entsprechenden Funktionen, die dem Zustand zugeordnet sind, werden ausgeführt. Um den Stromverbrauch zu reduzieren und um das Display nicht unnötig oft aktualisieren zu müssen, spielt bei der Zustandsmaschine der IDLE-Zustand eine wichtige Rolle. In diesem wird lediglich Zeit „verbraucht“, es wird also keine Funktion ausgeführt.

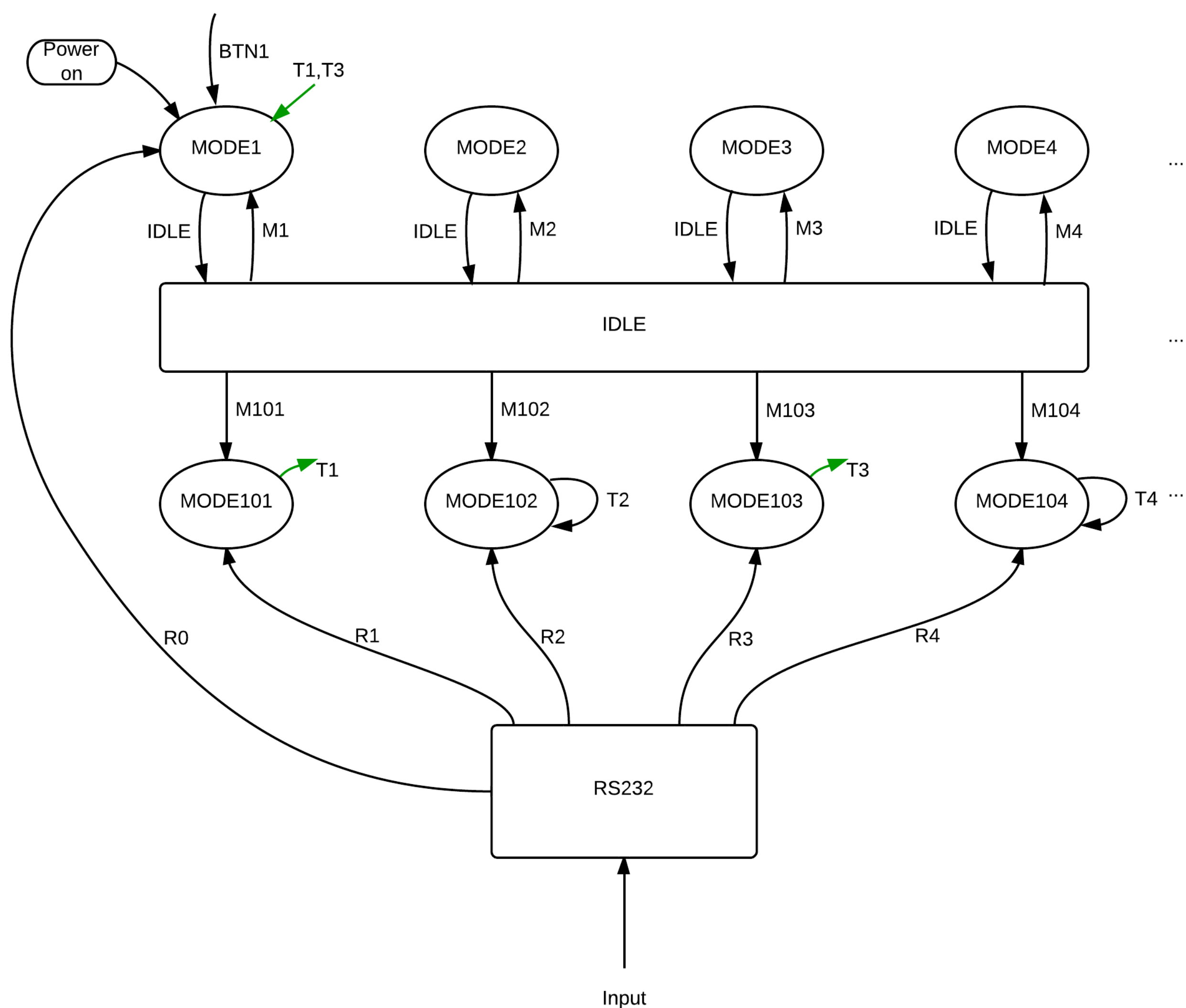


Abb. 4.1: Zustandsdiagramm

Die Abb. 4.1 zeigt das Zustandsdiagramm des Batteriesteuergerätes. Nachfolgend werden in Tabelle 4.1 die Transitionen des Zustandsdiagramms erklärt.

Transition	Bedingung
IDLE:	Nachdem alle Funktionen des jeweiligen Zustandes ausgeführt sind, wird der Zustand auf IDLE gesetzt.
M1:	Wenn state_aktuell = MODE2 und BTN3 oder Wenn state_aktuell = MODE4 und BTN4
M2:	Wenn state_aktuell = MODE3 und BTN3 oder Wenn state_aktuell = MODE1 und BTN4
M3:	Wenn state_aktuell = MODE4 und BTN3 oder Wenn state_aktuell = MODE2 und BTN4
M4:	Wenn state_aktuell = MODE1 und BTN3 oder Wenn state_aktuell = MODE3 und BTN4
M101:	Wenn state_aktuell = MODE1 und BTN2
M102:	Wenn state_aktuell = MODE2 und BTN2
M103:	Wenn state_aktuell = MODE3 und BTN2
M104:	Wenn state_aktuell = MODE4 und BTN2
T1:	Wenn alle Funktionen im Zustand M101 ausgeführt wurden, wird der Zustand auf Zustand M1 gesetzt
T2:	Zustand M102 bleibt solange aktiv, bis durch BTN1 zu Zustand M1 gewechselt wird.
T3:	Wenn alle Funktionen im Zustand M103 ausgeführt wurden, wird der Zustand auf Zustand M1 gesetzt
T4:	Zustand M104 bleibt solange aktiv, bis durch BTN1 zu Zustand M1 gewechselt wird.
BTN1:	Zustand M1 wird gesetzt, unabhängig des aktuellen Zustandes
R0:	Beim Empfang einer „0“ über RS232 wird der Zustand MODE1 gesetzt
R1:	Beim Empfang einer „1“ über RS232 wird der Zustand MODE101 gesetzt
R2:	Beim Empfang einer „2“ über RS232 wird der Zustand MODE102 gesetzt
R3:	Beim Empfang einer „3“ über RS232 wird der Zustand MODE103 gesetzt
R4:	Beim Empfang einer „4“ über RS232 wird der Zustand MODE104 gesetzt

Tabelle 4.1: Transitionen des Zustandsdiagrammes

4.2.2 Die Zustände

Nachfolgend werden in der Tabelle 4.2 die einzelnen Zustände näher erläutert.

Zustand	Beschreibung
MODE1 Auswahl: Einfache Strom- und Spannungsmessung	• Ausgabe des Menüs über RS232 und das Display • state_aktuell = MODE1 • state = IDLE
MODE2 Auswahl: Zyklische Strom- und Spannungsmessung	• Ausgabe des Menüs über RS232 und das Display • state_aktuell = MODE2 • state = IDLE
MODE3 Auswahl: Hall-Sensor kalibrieren	• Ausgabe des Menüs über RS232 und das Display • state_aktuell = MODE3 • state = IDLE
MODE4 Auswahl: Systemzeit anzeigen	• Ausgabe des Menüs über RS232 und das Display • state_aktuell = MODE4 • state = IDLE
MODE101 Einfache Strom- und Spannungsmessung	• Spannungsmessung • Strommessung • state = MODE1
MODE102 Zyklische Strom- und Spannungsmessung bei max. Messfrequenz für den Modus Einzelmessung im Zellsensor	• Spannungsmessung • Strommessung • Speicherung der Daten auf der SD-Karte
MODE103 Hall-Sensor kalibrieren	• Offsetkalibrierung des Hall-Sensors • state = MODE1
MODE104 Systemzeit anzeigen	• aktuelle Systemzeit auf Display anzeigen

Tabelle 4.2: Zustände des Zustandsdiagrammes

In den nachfolgenden Abbildungen werden die Ausgaben auf dem Display in den jeweiligen Zuständen angezeigt.

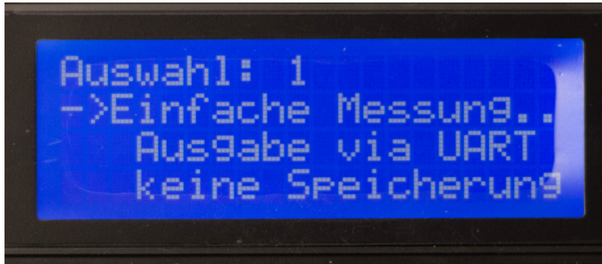


Abb. 4.2: MODE1 Displayausgabe

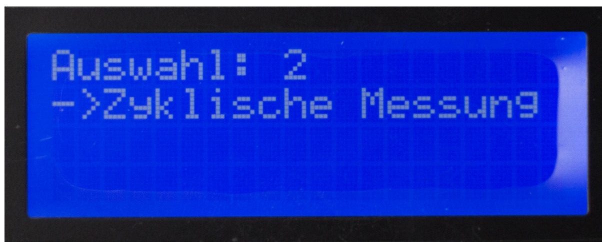


Abb. 4.3: MODE2 Displayausgabe

Abb. 4.4: MODE102 Displayausgabe

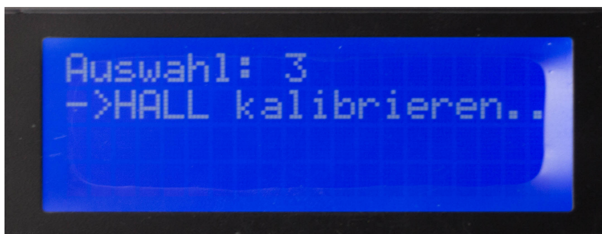


Abb. 4.5: MODE3 Displayausgabe

Abb. 4.6: MODE103 Displayausgabe

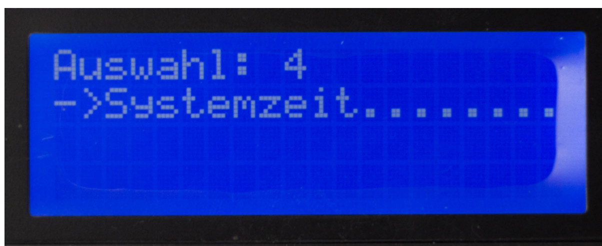


Abb. 4.7: MODE4 Displayausgabe

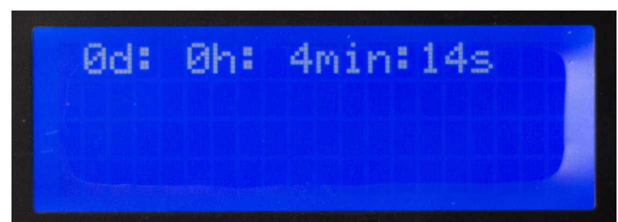


Abb. 4.8: MODE104 Displayausgabe

4.2.3 Interrupts

Die Software des Batteriesteuergerätes ist ausschließlich interruptgesteuert. Dem hier gewählten Mikrocontroller LM3S9D92 stehen dabei acht unterschiedliche Priorisierungen, mit 0 als höchste Priorität und 7 als niedrigste Priorität, der Interrupts zur Verfügung.

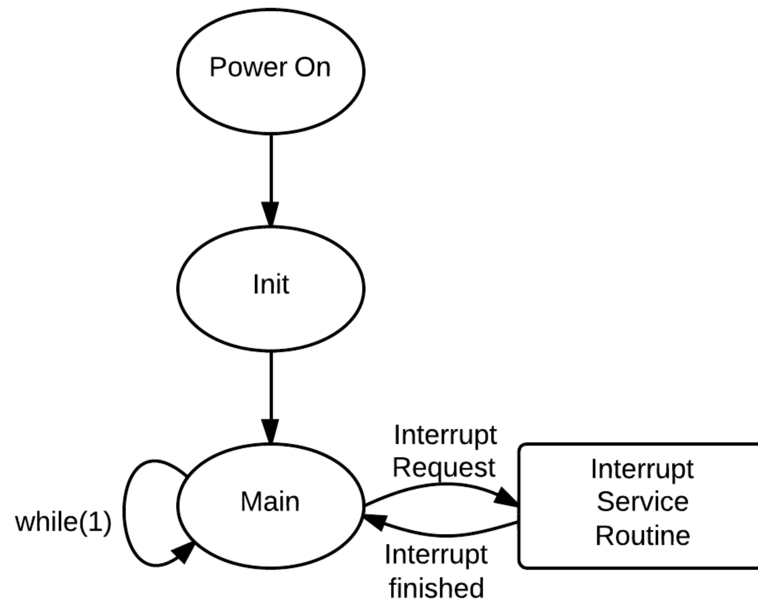


Abb. 4.9: Main-Schleife

Die Abb. 4.9 veranschaulicht den Startvorgang des Batteriesteuergerätes. Nach dem Einschalten des Gerätes erfolgt die Initialisierung der Peripherie. Anschließend läuft das Programm gezielt in eine Endlosschleife. Ab jetzt wird das Batteriesteuergerät ausschließlich mit Interrupts gesteuert. Die nachfolgende Tabelle stellt alle verwendeten Interrupts sowie deren Priorität dar.

Interruptquelle	**Interrupt Handler**	**Priorität**
Systick	SysTickHandler: • Wird alle 10ms aufgerufen • Wird benötigt für die Implementation der SD-Karte • Zählt globale Zeitvariablen hoch	Höchste Priorität – 0x00
GPIO Port C (Buttons)	GPIOPortCHandler: • Wird ausgelöst durch das Betätigen eines Buttons • Setzt die Zustände in Abhängigkeit des vorherigen Zustandes	2t-höchste Priorität – 0x20

UART1 (RS232)	UARTIntHandler: • Wird beim Empfangen von Daten über den RS232 Port aufgerufen • Setzt die Zustände in Abhängigkeit von den Empfangenen Daten	Gleiche Priorität wie Buttons 0x20
GPIO Port D (Transceiver-Modul)	GDO2IntHandler: • Wird am GDO2-Pin des Transceiver-Moduls ausgelöst • Verschiedene Funktionen wie Statusmeldungen	3t-höchste Priorität – 0x40
Timer1A (Statemachine)	Timer1IntHandler: • Wird alle 100ms ausgelöst • Überprüft den aktuellen Zustand und führt jeweilige Funktionen der Zustände aus	4t-höchste Priorität – 0x60

Tabelle 4.3: Übersicht alle Interrupts

Wie aus Tabelle 4.3 zu entnehmen ist, wird bei der Priorisierung der Schwerpunkt auf die Steuerung des Batteriesteuergerätes von außen gelegt. Wenn sich das System z.B. durch einen Übertragungsfehler bei dem Transceiver-Modul in einen Deadlock[9] verfängt, ist es weiterhin möglich, mit den Buttons bzw. über die RS232-Verbindung das System zu steuern.

4.3 Spannungsmessung

Für die Spannungsmessung der Batteriezellen muss der jeweilige Zellsensor angesteuert werden. Dazu muss das Batteriesteuergerät dem Transceiver-Modul die Befehle zur Spannungsmessung senden. Das Transceiver-Modul adressiert dann über Funk den jeweiligen Zellsensor und fordert diesen auf, eine Messung durchzuführen bzw. die Messdaten zu senden. Die Zellsensoren besitzen zwei verschiedene Messmethoden: die Burst- sowie die Einzelmessung. In dieser Arbeit wird zunächst die Einzelmessung implementiert, da der Aufwand für die Adaption der Burstmessung den zeitlichen Rahmen dieser Arbeit sprengen würde. Die genaue Funktionsweise der Zellsensoren ist nachzulesen in der Bachelorthesis von Herrn Durdaut und Herrn Sassano [11] [7].

[9] Das Batteriesteuergerät wartet auf das Transceiver-Modul, welches aber selbst wiederrum auf ein Signal seitens Batteriesteuergerät wartet

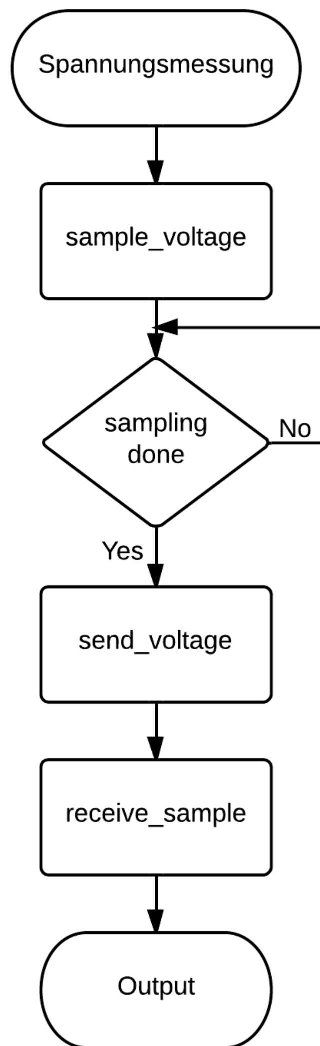


Abb. 4.10: Spannungsmessung

Der genaue Ablauf wird hier anhand von Abb. 4.10 verdeutlicht. Zuerst muss der Zellsensor aufgefordert werden, eine Spannungsmessung durchzuführen. Sobald diese fertig ist, wird mit der Funktion send_voltage der Zellsensor aufgefordert, die Messdaten zu senden. Die gesendeten Daten werden dann an dem Batteriesteuergerät mit der Funktion receive_sample empfangen und in einer Variable bzw. auf der SD-Karte gespeichert.

4.4 Strommessung

4.4.1 Offsetkalibrierung

Der hier benutzte Hall-Sensor, siehe Kapitel 3.7, besitzt sowohl eine Messungenaugkeit, bedingt u.a. durch die Umgebungstemperatur und den zu messenden Strom, sowie einen Offsetfehler. Die Messungenaugkeit lässt sich nicht so trivial kompensieren, da diese sehr variabel ist und durch viele Faktoren zusätzlich beeinflusst wird. Der Offsetfehler wird im ersten Anlauf durch eine einfache Offsetkalibrierung bei 0A behoben.

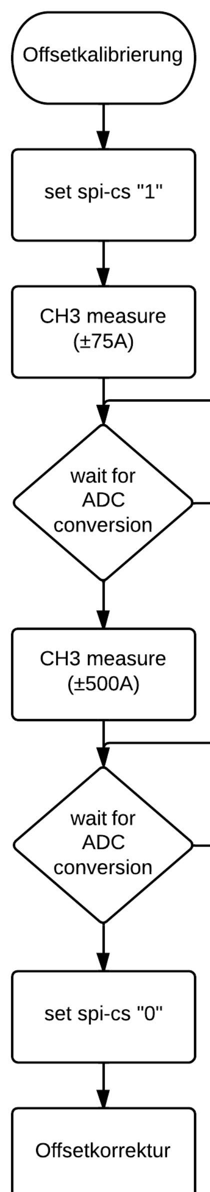


Abb. 4.11: Offsetkalibrierung Hall-Sensor

Die Abb. 4.11 zeigt den Ablauf der Offsetkalibrierung. Es wird zunächst das SPI-Chipselect-Signal auf „1“ gesetzt; damit wird die Leitung auf den „L“-Pegel gesetzt und der ADC wird zum aktiven Slave am SPI-Bus. Daraufhin wird die Hall-Spannung des ADC-Channels 3, 75A Bereich, als auch die des ADC-Channels 2, 500A Bereich, gemessen. Die Offsetkorrektur für beide Kanäle errechnet sich über folgende Formel:

$\boldsymbol{U_{Ref} = 5V}$
$\boldsymbol{ADC_{Aufl.} = 2^{16}\ Bit}$
$\boldsymbol{U_{Hall} : gemessene\ Hall - Spannung}$

$$\boldsymbol{ADC_{Step} = \frac{U_{Ref}}{ADC_{Aufl.}}} \quad (4.1)$$

$$\boldsymbol{U_{Offset} = ADC_{Step} \cdot U_{Hall} - 2,5V} \quad (4.2)$$

Die subtrahierten 2,5V in Formel (4.2) entsprechen $\frac{U_{Ref}}{2}$. Diese werden von dem Hall-Sensor gemessen, wenn kein Strom durch den zu messenden Leiter fließt.

4.4.2 Strommessung

Wenn der Hall-Sensor wie in Kapitel 4.4.1 korrekt kalibriert wurde, kann nun eine Strommessung mit den in Kapitel 3.7 genannten erwarteten Abweichungen stattfinden. Der Ablauf ist in Abb. 4.12 dargestellt.

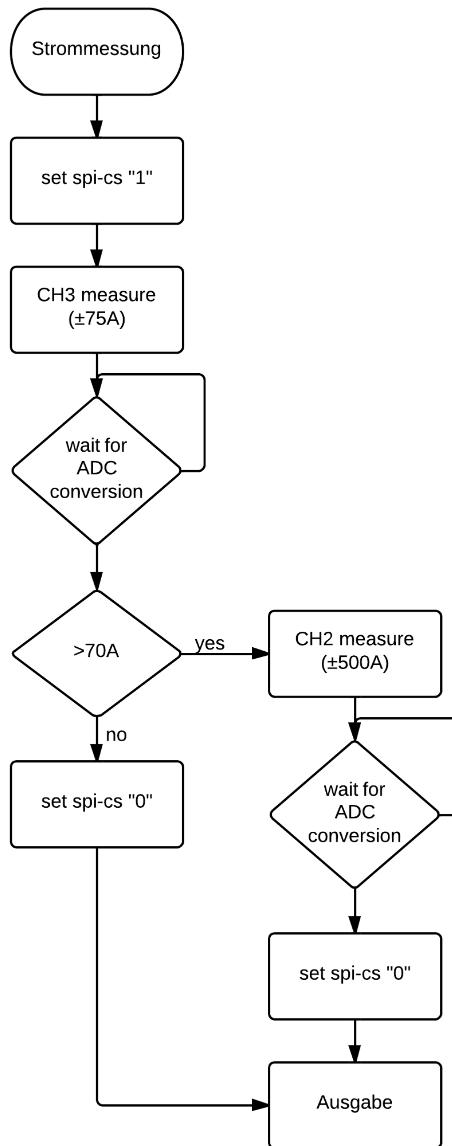


Abb. 4.12: Strommessung

Erneut muss das CS-Signal gesetzt werden, damit der ADC der aktive Teilnehmer am SPI-Bus wird. Daraufhin wird zunächst eine Messung über CH3 gestartet. Sollte der gemessene Strom über dem gewählten Wert von ±70A liegen, wird eine erneute Messung über den CH2 durchgeführt. Dieser verfügt über einen Messbereich von ±500A. Nachdem die Umwandlung der Messung am ADC abgeschlossen ist, wird das CS-Signal wieder zurückgesetzt und der SPI-Bus somit wieder für andere Teilnehmer freigegeben.

Diese Funktion gibt als Rückgabewert den aus dem Messwert errechneten Stromwert zurück. Die Berechnung erfolgt nach (4.5) [31].

$U_{Ref} = 5V;\ n = 2^{16} bit$
m_{adc} : $digitaler\ ADC - Wert$
U_{Offset}: $errechneter\ Hall - Offset$

$$V_{OUT} = \left(\frac{U_{Ref}}{n} \cdot m_{adc}\right) - U_{Offset} \qquad (4.3)$$

$$V_{OUT} = \left(\frac{5V}{2^{16}} \cdot m_{adc}\right) - U_{Offset} \qquad (4.4)$$

$V_{Out} = Ausgangsspannung\ am\ Hallsensor$

$V_C = Versorgungsspannung\ Hallsensor = 5V$

$G_{Ch1} = 0{,}0267 \frac{V}{A}\ ;\ G_{Ch2} = 0{,}004 \frac{V}{A}$

$$I_{meas.} = \left(V_{Out} - \frac{V_C}{2}\right) \cdot \frac{1}{G} \cdot \frac{5}{V_C} \qquad (4.5)$$

4.5 Speicherung der Daten

4.5.1 SD-Karte

Damit, wie in Kapitel 2.2.9 erwähnt, die Spannungs-, Strom- sowie Temperaturdaten auf der SD-Karte speichern zu können, ist ein standarisiertes Dateisystem auf der SD-Karte notwendig. Die Entscheidung ist auf das weit verbreitete FAT-Filesystem (FatFs) gefallen.

Für die korrekte Funktionsweise der SD-Karte und des FatFs muss alle 10ms eine FatFs-Funktion aufgerufen werden. Diese Funktion dekrementiert eine globale Variable, die als Timer für verschiedene weitere FatFs-Funktionen dient. Da die FatFs-Implementierung aus einem Beispielprogramm der StellarisWare-Software stammt, wird hier die Funktionsweise nicht weiter vertieft.

Der erwähnte Aufruf alle 10ms wird über den SysTick-Timer des LM3S9D92-Mikrocontrollers realisiert. Dieser ist so eingestellt, dass alle 10ms ein Interrupt auslöst wird. Über die durch den Interrupt aufgerufene Funktion (ISR – Interrupt Service Routine) wird die benötigte FatFs-Funktion aufgerufen, vgl. Abb. 4.16 in Kapitel 4.5.2.

Um den Zugriff auf die SD-Karte möglichst simpel zu gestalten, sind einfach zu benutzende Funktionen implementiert worden. Nachfolgend werden nur die wichtigsten Funktionen erläutert. In der Software sind allerdings noch weitere Funktionen implementiert, die bei Bedarf im Quellcode nachgelesen werden können.

Folgende Funktionen werden genauer erläutert:

- create_file
- add_to_file
- delete_file

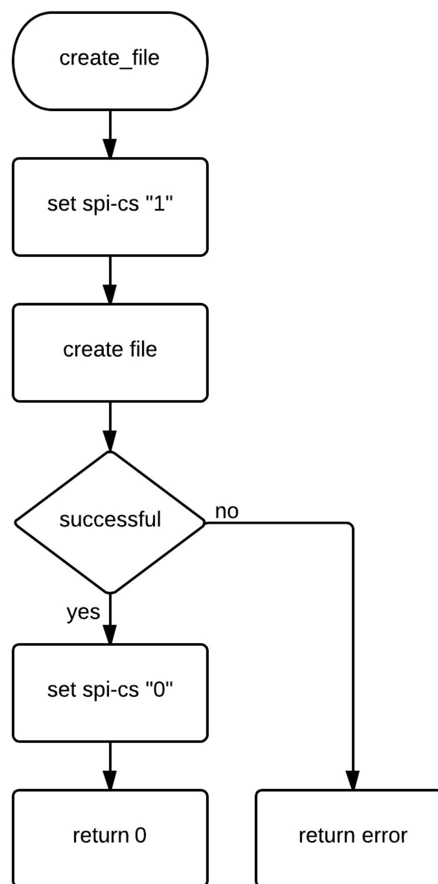


Abb. 4.13: SD-Karte create_file

Die Funktion „create_file“ benötigt als Parameter einen Dateinamen mit Endung. Nach der Konfiguration des SPI-Busses wird im festgelegten Verzeichnis [10] eine Datei mit dem gewählten Namen erstellt. Sollte bereits eine Datei mit diesem Namen existieren, wird sie überschrieben. Nach dem Erstellen wird geprüft, ob die Datei auch tatsächlich korrekt erstellt wurde. Bei einem Fehler wird dieser ausgegeben, ansonsten wird eine „0“ zurückgegeben.

[10] globale Variable im Quellcode, standardmäßig als „/“ (root Verzeichnis) festgelegt.

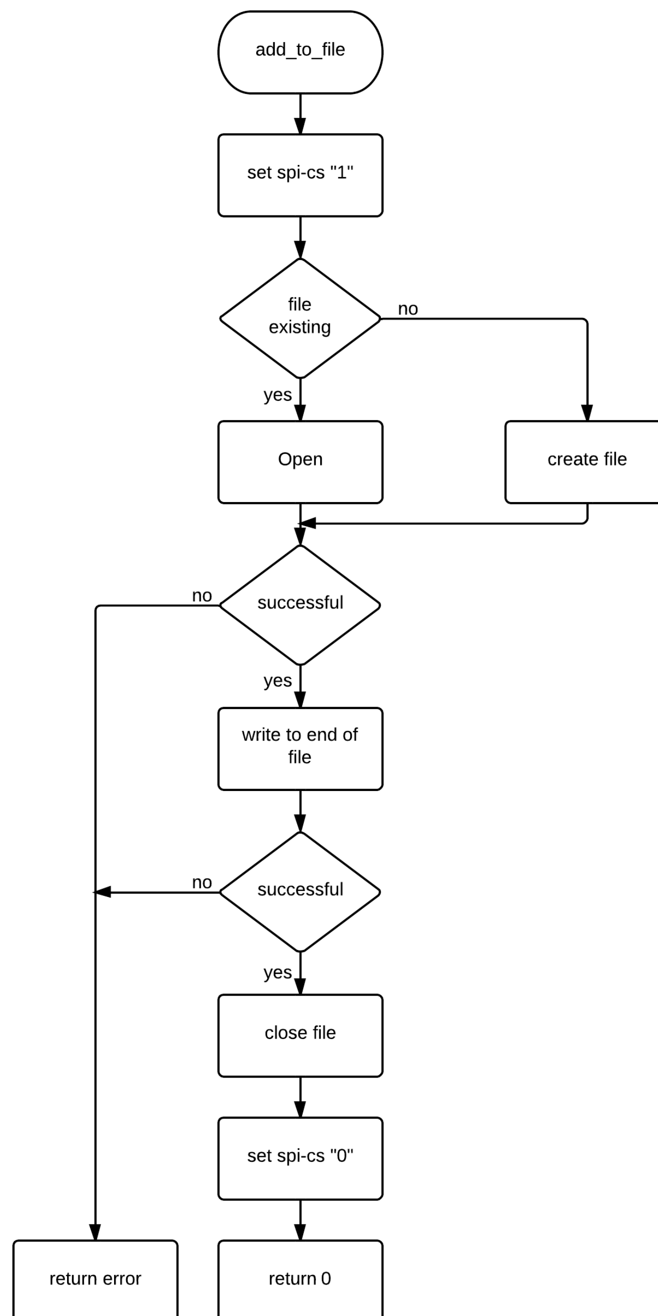


Abb. 4.14: SD-Karte add_to_file

Die „add_to_file“ Funktion benötigt als Parameter den Dateinamen mit Endung sowie einen String mit dem gewünschten Inhalt. Daraufhin wird erneut der SPI-Bus umkonfiguriert und das SPI-CS Signal wird gesetzt, wie es in dem Fall der „create_file“-Funktion ist. Daraufhin wird die Datei geöffnet bzw. erstellt, falls nicht vorhanden und der String wird an das Ende dieser Datei geschrieben. Jeweils nach dem Öffnen, Erstellen und Schreiben des Inhalts erfolgt eine Überprüfung, ob alles funktioniert hat. Bei einem Fehler wird dieser zurückgegeben und die Funktion wird beendet. Bei erfolgreichem Schreiben des Inhaltes in die Datei wird diese geschlossen und eine „0“ wird ausgegeben.

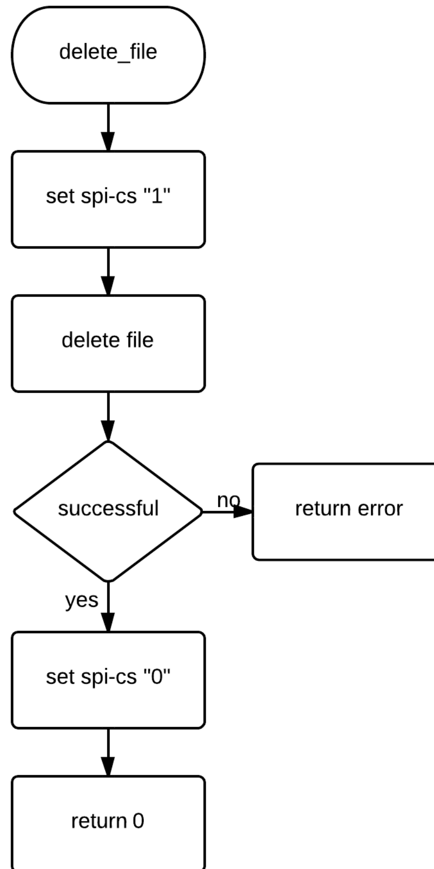


Abb. 4.15: SD-Karte delete_file

Zum Löschen einer Datei muss lediglich der Dateiname mit Endung an die Funktion „delete_file“ übergeben werden. Der SPI-Bus wird erneut zur Kommunikation zwischen dem Mikrocontroller und der SD-Karte rekonfiguriert und das CS-Signal wird gesetzt. Dann wird die Datei von der SD-Karte gelöscht. Sollte dabei ein Fehler auftreten, wird dieser zurückgegeben. Bei erfolgreichem Löschen gibt die Funktion ebenfalls „0“ zurück.

4.5.2 Zeitbasis

Für die Datenauswertung der Messergebnisse von den Zellsensoren sowie dem Hall-Sensor ist der zeitliche Zusammenhang besonders wichtig. Somit muss zu jeder Strom-, Spannungs- und Temperaturmessung der Zeitpunkt der Messung festgehalten werden. Dazu wird der in Kapitel 4.2.3 und Kapitel 4.5.1 erwähnte Interrupt verwendet. Bei dem Aufruf dieser Funktion werden jedes Mal mehrere globale Variablen, jeweils eine für Millisekunde, Sekunde, Minute, Stunde, Tag, Monat sowie Jahr hochgezählt (Abb. 4.16).

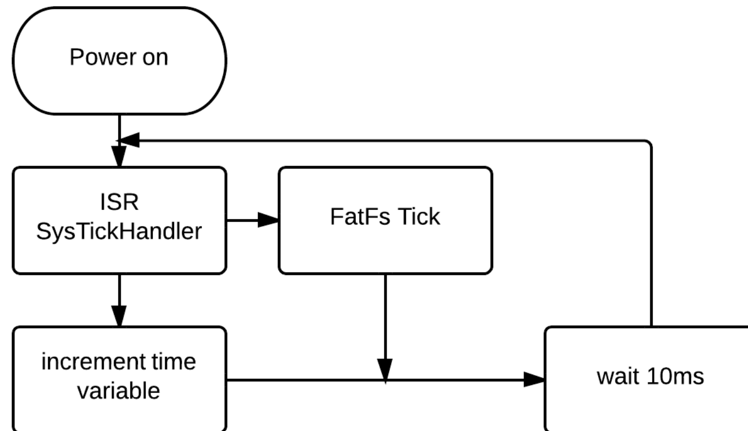


Abb. 4.16: Zeitbasis 10ms

Durch die Benutzung des FatFs-Interrupts, ist eine zeitliche Auflösung von 10ms möglich. Durch diese Methode fehlt allerdings eine Echtzeitangabe. Für die Lösung dieses Problems existieren zwei Lösungen. Entweder kann man z.B. über die RS232 Schnittstelle die aktuelle Zeit manuell definieren oder es muss in einer zukünftigen Version des Batteriesteuergerätes ein RTC-Chip inkl. Batterie implementiert werden.

4.5.3 Datenformatierung

Um die Messdaten der Zellsensoren und des Hall-Sensors auswerten zu können, müssen diese in einer vereinheitlichten Form in einer Datei stehen. Als geeignet hat sich folgendes Format erwiesen:

Messung;Sensor;Spannung;Strom;Temperatur;Tag:Stunde:min:sec:ms

Bedeutung der einzelnen Parameter:

- Messung: Eine Nummer, die kontinuierlich hochgezählt wird (10 Stellen reserviert)
- Sensor: Nummer des Sensors, mit dem die Spannungsmessung durchgeführt wurde (3 Stellen reserviert)
- Spannung: Spannungswert der von gemessenen Zelle in V (3 Nachkommastellen)
- Strom: Stromwert des Hall-Sensors in A (2 Nachkommastellen)
- Temperatur: gemessene Temperatur in °C (1 Nachkommastelle)
- Zeitangabe im Format: (TT:hh:mm:ss:ms)

Die Parameter sind jeweils mit einem Semikolon getrennt und können so leicht in ein Programm für weitere Auswertungen geladen werden. Durch diese Formatierung der Daten werden insgesamt 45 Zeichen pro Messung gespeichert. Ein Zeichen benötigt 8 Byte. Damit ergibt sich ein Speicherverbrauch von 360 Byte pro Messung.

4.6 Steuerung des Batteriesteuergerätes

Die Steuerung des Batteriesteuergerätes kann entweder über das LCD-Display mit Hilfe der 4 Buttons oder über den RS232 Anschluss mit einem PC gesteuert werden. Beide Modi werden in den folgenden Kapiteln näher erläutert.

4.6.1 RS232

Um das Batteriesteuergerät über den RS232 Anschluss zu steuern, wird ein PC benötigt, der über einen seriellen Port verfügt. Zudem wird ein Terminal-Programm benötigt. Während dieser Arbeit wurde das Programm HTerm genutzt [17].

Bevor man mit dem Batteriesteuergerät kommunizieren kann, muss das Terminal-Programm an die Einstellungen des Batteriesteuergerätes angepasst werden.

- 115200 BAUD Übertragungsrate
- 8-Bit Datenlänge
- 1 Stop-Bit
- keine Paritätsprüfung

Bei erfolgreicher Verbindung mit dem Batteriesteuergerät ist das Hauptmenü zu sehen (Abb. 4.17).

```
\nMenu:....\r
\nSend 1 for Mode1, Einfache Messung\r
\nSend 2 for Mode2, Zyklische Messung\r
\nSend 3 for Mode3, HALL kalibrieren\r
\nSend 4 for Mode4, Systemzeit anzeigen\r
\n
```

Abb. 4.17: Hauptmenü des Batteriesteuergerätes RS232

Die Steuerung über den RS232 Port in dem Batteriesteuergerät ist interruptgesteuert. Sobald von dem PC aus eine Eingabe gesendet wird, startet ein Interrupt Handler im Batteriesteuergerät.

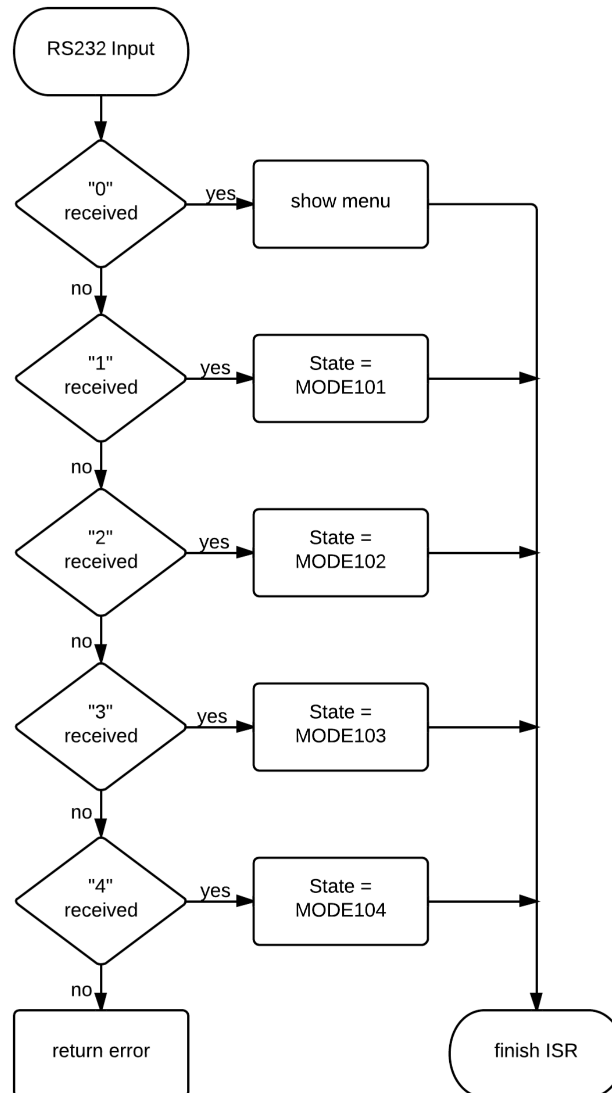


Abb. 4.18: RS232 Interrupt Handler

Abb. 4.18 veranschaulicht die Funktionsweise des Interrupt-Handlers. Es wird die empfangene Tasteneingabe ausgewertet und der entsprechende State wird gesetzt.

4.6.2 Buttons

Will man das Batteriesteuergerät unabhängig von einen PC steuern, so geht dies über die vier Buttons. Durch Betätigen des Buttons 4 bzw. 3, wird durch das integrierte Menü hoch- bzw. herunter geblättert. Die Anzeige auf dem Display ändert sich bei jedem Tastendruck und zeigt die aktuell angewählte Funktion an. Möchte man diese ausführen, muss der Button 2 betätigt werden. Zum Abbrechen der jeweiligen Funktion muss der Button 1 gedrückt werden. Damit kehrt man wieder zur Menüauswahl zurück. Die nachfolgende Abb. 4.19 verbildlicht diesen Ablauf des Interrupt-Handlers, der durch ein Drücken der Buttons aufgerufen wird.

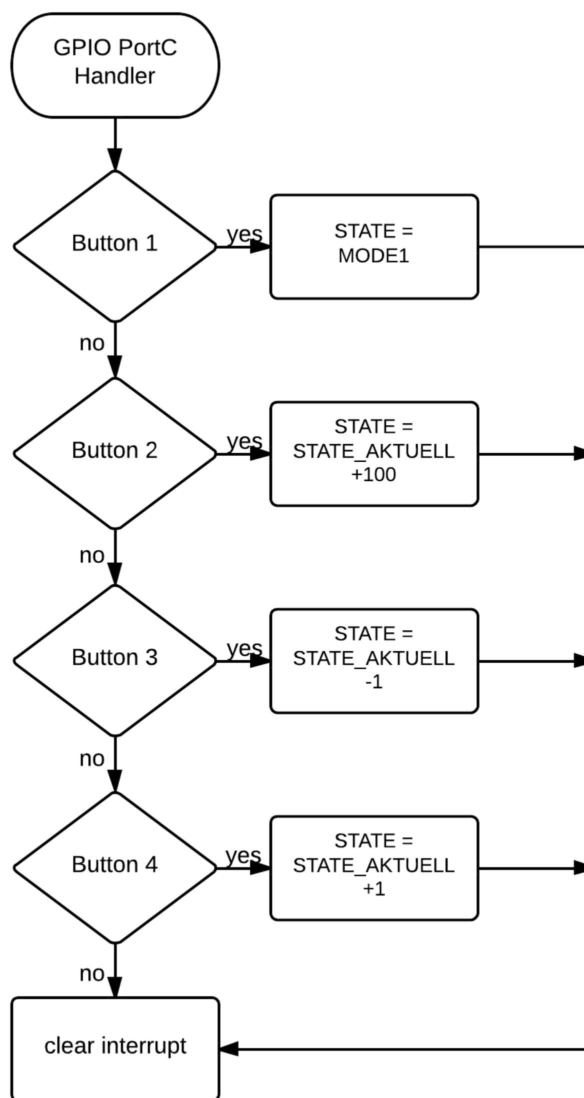


Abb. 4.19: Steuerung des Batteriesteuergerätes über die Buttons

4.7 LCD-Display

Wie bereits mehrfach erwähnt, wird ein LCD-Display zur Anzeige des aktuellen Status sowie zur Vereinfachung der Steuerung des Batteriesteuergerätes über die Buttons verwendet. Um die Steuerung des Displays möglichst erweiterbar und übersichtlich zu halten, ist ein modularer Aufbau der Steuerungssoftware notwendig. Dazu sind folgende Funktionen implementiert worden:

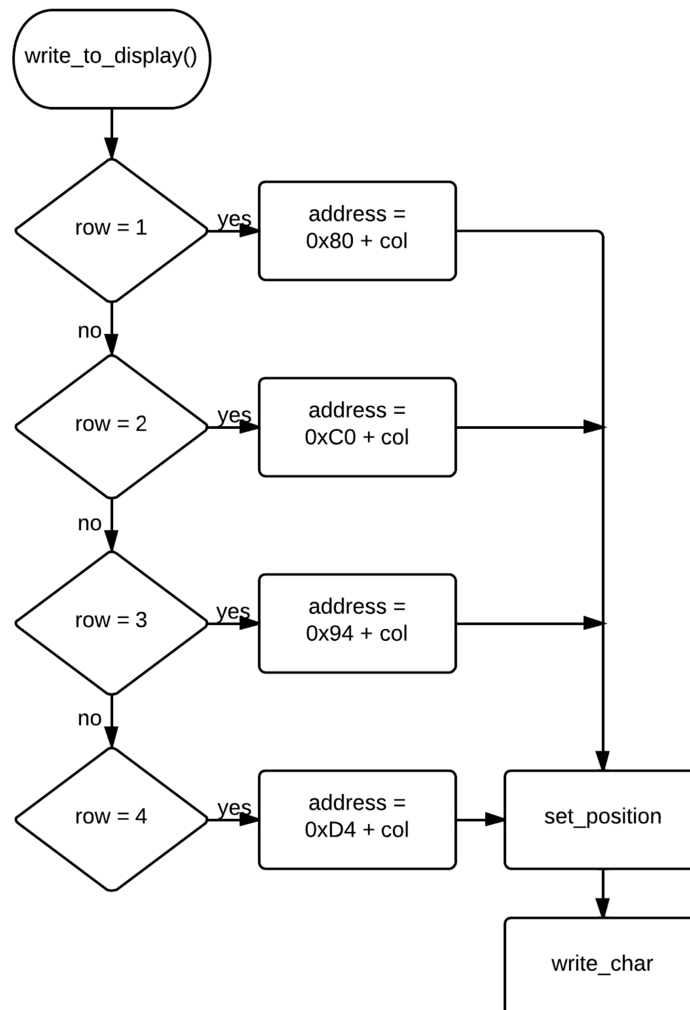


Abb. 4.20: write_to_display-Funktion

An die write_to_display-Funktion wird der zu schreibende Text als string, die Zeile als unsigned char sowie die Spalte als unsigned char übergeben. Aus der Zeile sowie der Spalte errechnet sich die genaue Position, an die der String geschrieben wird. Wenn der übergebene String nur ein Zeichen enthält, wird dieses Zeichen an die übergebene Position geschrieben. Falls der String mehrere Zeichen hat, werden diese einzeln nacheinander geschrieben. Dabei wird die Position nach jedem Zeichen um 1 erhöht. Durch die Größe des LCD-Displays sind maximal 20 Zeichen pro Zeile möglich [25].

Die genaue Position in der Zeile errechnet sich aus dem vorgegebenen Offset und der Spalte. Der vorgegebene Offset ist dem Datasheet zu entnehmen [25].

ZUORDNUNG DD-RAM ADRESSE ZU ZEICHENSTELLE IM DISPLAY

Displaytyp	Anfangs - Endadresse (HEX)				Bemerkung
	1.Zeile	2.Zeile	3.Zeile	4.Zeile	
1x8	$00-$07				
1x16	$00-$0F				MUX 1:8
1x16(8+8)	$00-$07				MUX 1:16 (linke Hälfte)
	$40-$47				(rechte Hälfte)
1x20	$00-$13				
1x40	$00-$27				
2x8	$00-$07	$40-$47			
2x12	$00-$0B	$40-$4B			
2x16	$00-$0F	$40-$4F			
2x20	$00-$13	$40-$53			
2x24	$00-$17	$40-$57			
2x40	$00-$27	$40-$67			
4x16	$00-$0F	$40-$4F	$10-$1F	$50-$5F	
4x20	$00-$13	$40-$53	$14-$27	$54-$67	Kontroller HD44780
	$00-$13	$20-$33	$40-$53	$60-$73	Kontroller KS0073
4x40	$00-$27	$40-$67	-	-	1. Kontroller (Enable 1)
	-	-	$00-$27	$40-$67	2. Kontroller (Enable 2)

Abb. 4.21: LCD-Display Adresse Offset [25]

Da es sich bei dem hier eingesetzten LCD-Display um ein 4x20-Zeichen-Modul mit dem HD44780-Controller handelt, müssen die Offsetwerte in der entsprechenden Zeile benutzt werden.

- 1. Zeile: 0x00 – 0x13
- 2. Zeile: 0x40 – 0x53
- 3. Zeile: 0x14 – 0x27
- 4. Zeile 0x54 – 0x67

Für das Setzen der Adresse muss zudem der DB7 Pin auf „1“ gesetzt werden. Daraus ergeben sich die in Abb. 4.20 gezeigten Werte.

- 1. Zeile: 0x80 – 0x93
- 2. Zeile 0xC0 – 0xD3
- 3. Zeile 0x94 – 0xA7
- 4. Zeile 0xD4 – 0xE7

Nach dieser Berechnung der genauen Position wird nun in der nachfolgenden set_position()-Funktion der Cursor an diese Stelle gesetzt.

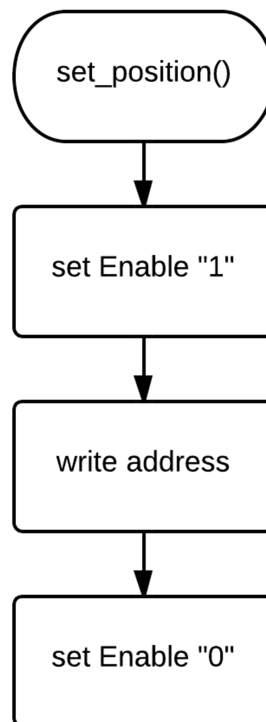


Abb. 4.22: set_position

Die Abb. 4.22 zeigt den Ablauf der set_position-Funktion. Zunächst wird das Enable-Signal auf „1“ gesetzt. Daraufhin wird die errechnete Adresse über die Pins DB0 – DB7 gesetzt. Mit der fallenden Flanke des Enable-Signals wird schließlich die Adresse übernommen und umgesetzt. Jetzt befindet sich der Cursor an der gewünschten Stelle und es können die Zeichen mit der nachfolgend erläuterten write_char-Funktion geschrieben werden.

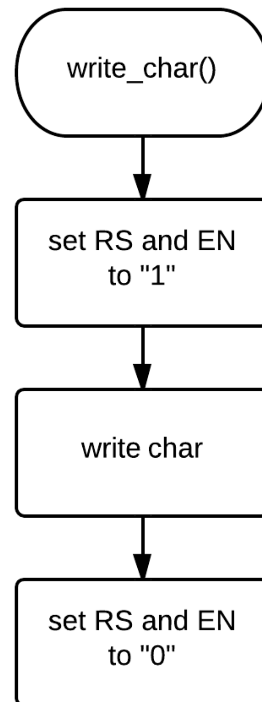


Abb. 4.23: LCD-Display write_char

Für das Schreiben eines Zeichens auf dem LCD-Display muss zunächst das Enable-Signal sowie das RS-Signal auf „1“ gesetzt werden. Daraufhin wird das Zeichen als 8-Bit-HEX-Wert über die Pins DB0-DB7 übertragen. Mit dem Zurücksetzen des Enable-Signals auf „0“ werden die Daten vom Kontroller auf dem LCD-Modul übernommen und das Zeichen wird auf dem Display an die gewählte Stelle gesetzt.

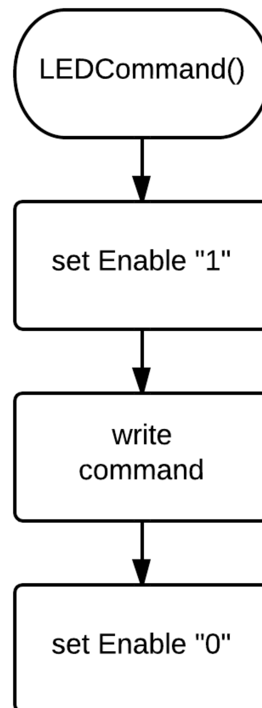


Abb. 4.24: LCD-Display LEDCommand

Zusätzlich zu den Funktionen, mit denen man Zeichen auf das Display schreiben kann, gibt es noch für das Schreiben von Befehlen an das LCD-Modul die LEDCommand-Funktion. Diese Befehle werden z.B. beim Initialisieren benötigt.

Abb. 4.24 zeigt den Funktionsablauf. Es wird das Enable-Signal auf „1“ gesetzt, dann wird der Befehl über die DB0 – DB7 Pins übertragen und anschließend übernimmt das LCD-Modul mit der fallenden Flanke die Daten.

5 Laborerprobung und Versuchsbetrieb

5.1 Übersicht des Batteriesteuergerätes

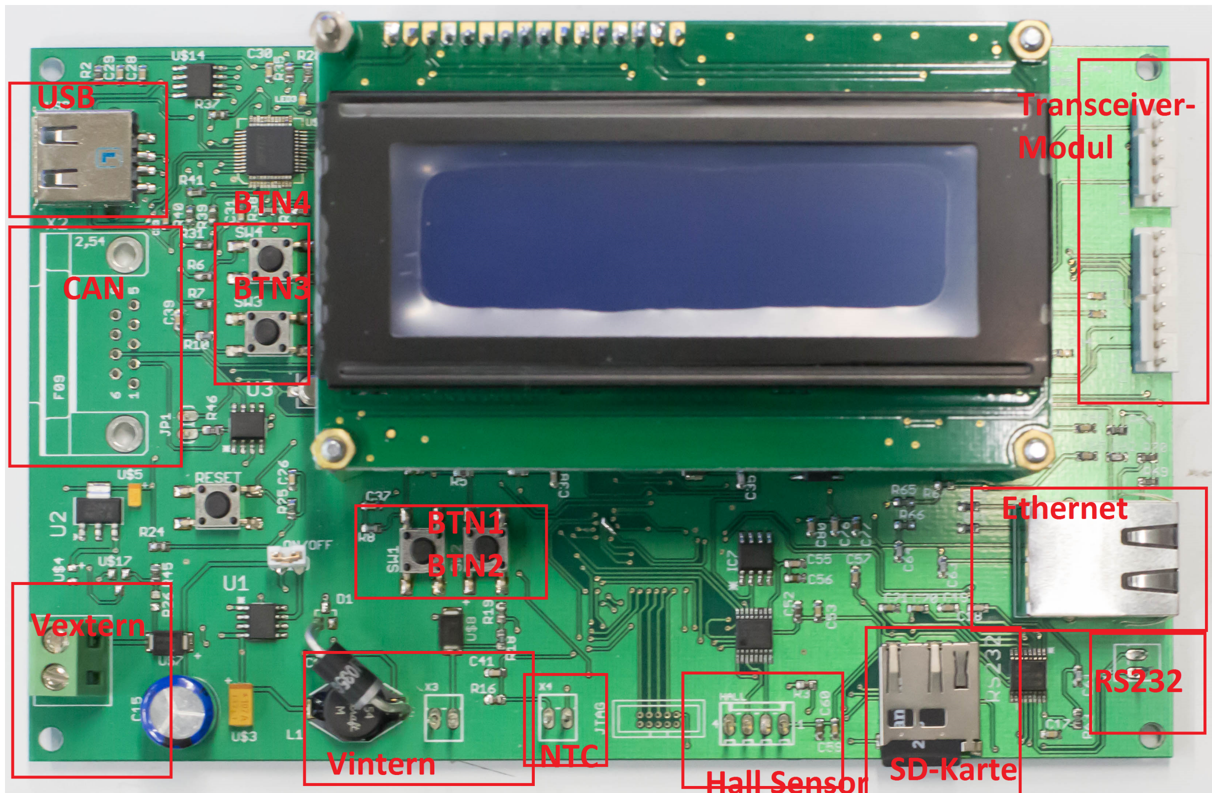


Abb. 5.1: Übersicht Batteriesteuergerät

Abb. 5.1 zeigt den ersten Prototypen des Batteriesteuergerätes mit den vorhandenen Schnittstellen sowie den Eingängen der Spannungsversorgung. Die Erläuterungen zu den jeweiligen Schnittstellen sind in Kapitel 3 zu finden.

Der nachfolgende Versuchsbetrieb wurde an dieser Platine durchgeführt.

5.2 Versuchsaufbau

Die Laborerprobung des Batteriesteuergerätes soll nachweisen, dass sowohl die Spannungsmessung durch einen Zellsensor der Klasse 3 als auch die funksynchronisierte Strommessung durch einen Hall-Sensor funktioniert und die Messdaten korrekt auf der SD-Karte abgespeichert werden. Der Funktionsnachweis findet, wie in der Aufgabenstellung zu dieser Arbeit beschrieben, mit generierten Signalen statt.

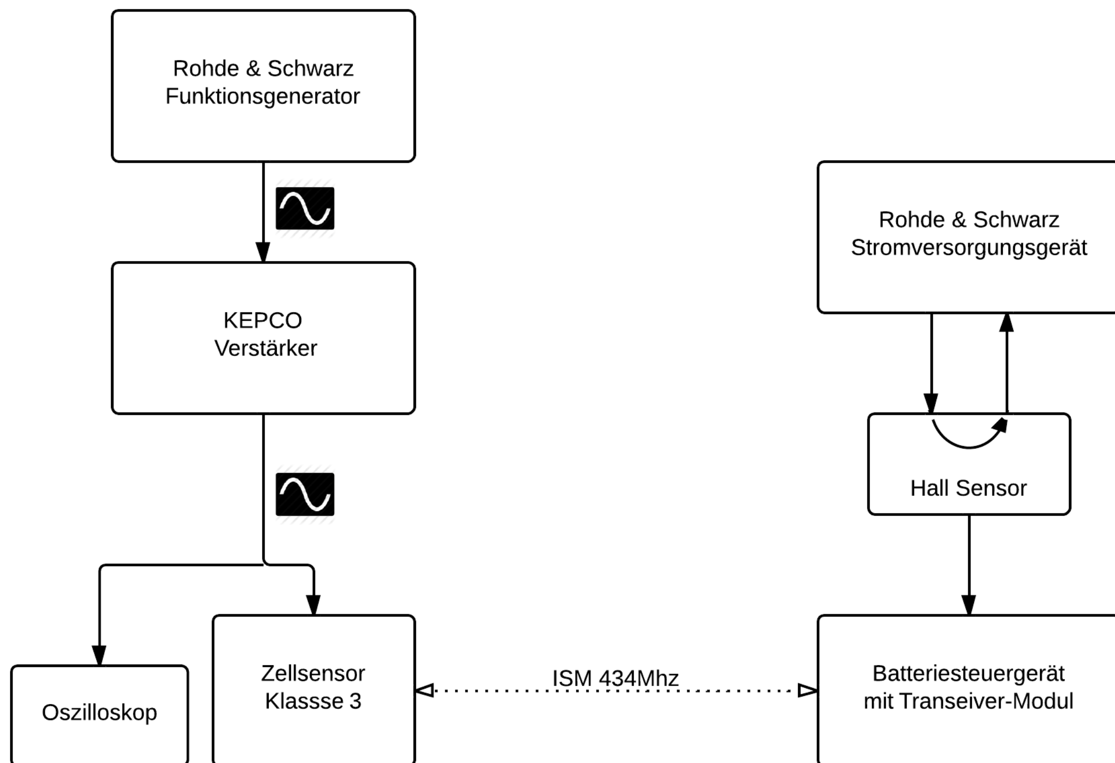


Abb. 5.2: Versuchsaufbau der Laborerprobung

Für diesen Funktionsnachweis wird der in Abb. 5.2 veranschaulichte Versuchsaufbau zusammengestellt. Für die Spannungsmessung des Zellsensors Klasse 3 darf die Spannung zwischen 0,6V und 5,5V liegen [7]. Damit die Spannungsmessung dynamisch ist und nicht nur ein konstanter Spannungspegel gemessen wird, erzeugt der Rohde&Schwarz-Funktionsgenerator ein Signal mit folgenden Parametern:

- Signalform: Sinus
- Frequenz: 1Hz
- Offset: -3V
- Amplitude: 1V

Da man mit dem Rohde&Schwarz-Funktionsgenerator keine Last, die der Zellsensor in diesem Fall darstellt, treiben kann, wird ein KEPCO Verstärker zwischen dem Funktionsgenerator und dem Zellsensor geschaltet. Dieser Verstärker ist allerdings ein invertierender Verstärker. Um am Ausgang des Verstärkers ein positives Signal generieren zu können, muss die Offsetspannung des Funktionsgenerators negativ sein. Mit dem eingestellten Verstärkungsfaktor „1“ ist die Ausgangsspannung des Verstärkers nun folgende:

- Signalform: Sinus
- Frequenz: 1Hz
- Offset: 3V
- Amplitude: 1V

Das daraus resultierende generierte Signal zur Laborerprobung ist sinusförmig mit einem Minimum bei 2V und einem Maximum bei 4V.

Damit bei der Auswertung das Messergebnis mit einem Referenzwert verglichen werden kann, wird dieses Signal zusätzlich mit dem „Tektronix MSO3034“-Oszilloskop gemessen und aufgezeichnet.

Um parallel zu der Spannungsmessung auch eine Strommessung durchzuführen, wird ein Rohde&Schwarz-Stromversorgungsgerät, sowie der bereits in Kapitel 3.7 erwähnte Hall-Sensor DHAB S/24 der Fa. LEM in dem Versuchsaufbau eingesetzt. Der durch diesen Hall-Sensor max. messbare Strombereich liegt bei ±500A. Um Ströme in diesem Bereich erzeugen und messen zu können, fehlen im Labor die notwendigen steuerbaren Stromquellen. Das hier eingesetzte Stromversorgungsgerät verfügt über einen max. Ausgangsstrom von 0,6A. Um diesen Strombereich noch etwas zu vergrößern, wird der zu messende Leiter vierfach durch den Hall-Sensor gewickelt. Daraus resultiert ein max. Strom von 0,6A•4=2,4A.

Für den Versuchsbetrieb wurde der Hall-Sensor bei 0A kalibriert. Das Batteriesteuergerät wird für diese Laborerprobung im Modus „MODE102 – Zyklische Messung“ betrieben, vgl. Kapitel 4.2.2.

5.3 Ergebnis

Zunächst ist in Abb. 5.3 das generierte Signal, aufgenommen mit dem „Tektronix MSO3034“-Oszilloskop, abgebildet. Als Minimum und Maximum sind 2,08V und 4,84V zu sehen. Erwartet werden, wie in Kapitel 5.2 beschrieben, 2V als Minimum und 4V als Maximum. Der Grund für diese Abweichung liegt am Verstärker. Dieser besitzt zur Einstellung des Offsets sowie der Spannungsverstärkung analoge Drehknöpfe. Dadurch lässt sich die Verstärkung „1“ sowie der Offset nicht exakt einstellen. Allerdings ist diese Abweichung für den Versuchsbetrieb nicht weiter störend, da der Zellsensor einen Versorgungsspannungsbereich von 0,6V-5,5V besitzt und somit dieser die Spannung problemlos messen kann.

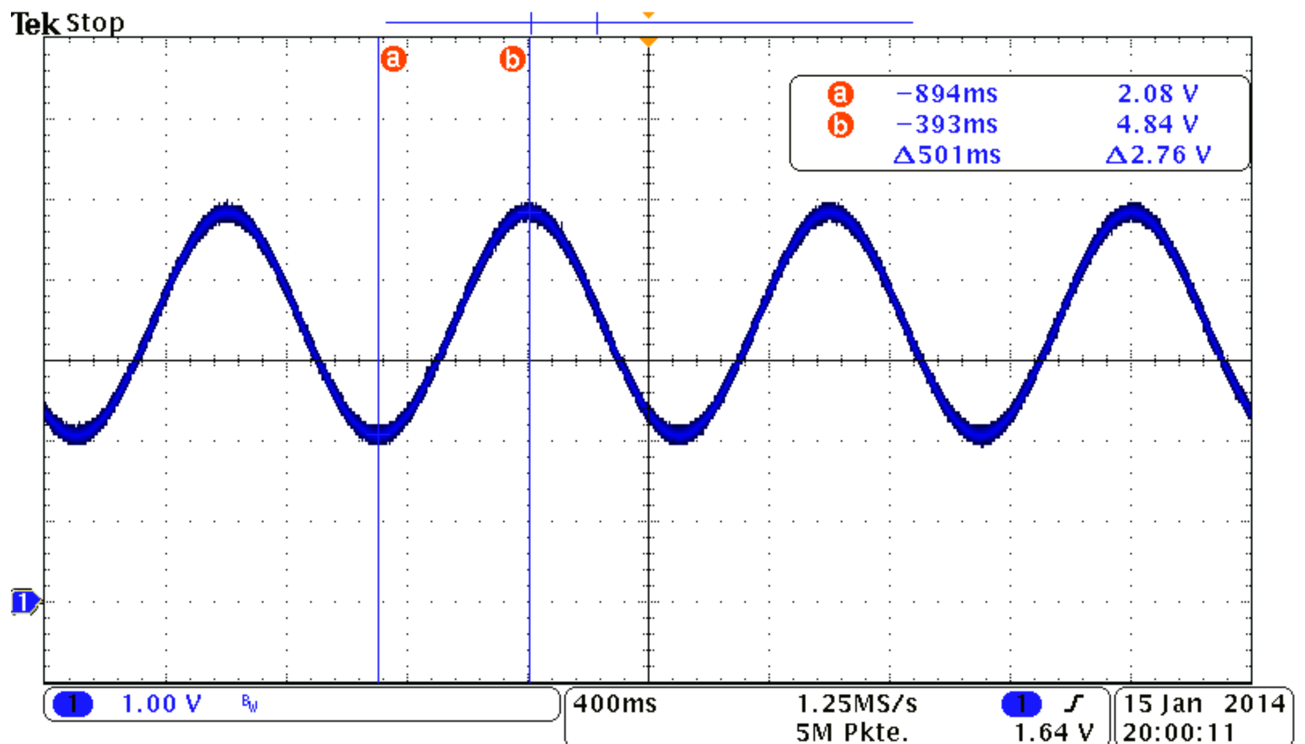


Abb. 5.3: Aufnahme des generierten Signals durch das Oszilloskop

```
 1;1;4.706;-0.01;0.0; 0d: 0h: 0min:12s:880ms
 2;1;4.706;-0.02;0.0; 0d: 0h: 0min:13s:40ms
 3;1;2.032;0.01;0.0; 0d: 0h: 0min:13s:260ms
 4;1;2.928;-0.01;0.0; 0d: 0h: 0min:13s:420ms
 5;1;4.524;-0.01;0.0; 0d: 0h: 0min:13s:570ms
 6;1;4.519;-0.00;0.0; 0d: 0h: 0min:13s:730ms
 7;1;3.446;-0.01;0.0; 0d: 0h: 0min:13s:890ms
 8;1;2.268;-0.03;0.0; 0d: 0h: 0min:14s:70ms
 9;1;2.041;-0.02;0.0; 0d: 0h: 0min:14s:230ms
10;1;3.169;0.01;0.0; 0d: 0h: 0min:14s:380ms
11;1;4.388;0.01;0.0; 0d: 0h: 0min:14s:540ms
12;1;4.617;0.02;0.0; 0d: 0h: 0min:14s:760ms
13;1;3.716;-0.05;0.0; 0d: 0h: 0min:14s:920ms
14;1;2.140;0.02;0.0; 0d: 0h: 0min:15s:80ms
15;1;2.170;-0.02;0.0; 0d: 0h: 0min:15s:230ms
16;1;3.257;-0.00;0.0; 0d: 0h: 0min:15s:390ms
17;1;4.427;-0.03;0.0; 0d: 0h: 0min:15s:550ms
18;1;4.600;0.02;0.0; 0d: 0h: 0min:15s:710ms
19;1;3.655;-0.01;0.0; 0d: 0h: 0min:15s:860ms
20;1;2.410;0.02;0.0; 0d: 0h: 0min:16s:20ms
21;1;2.006;0.01;0.0; 0d: 0h: 0min:16s:240ms
22;1;2.800;-0.01;0.0; 0d: 0h: 0min:16s:400ms
23;1;4.459;-0.01;0.0; 0d: 0h: 0min:16s:560ms
24;1;4.574;0.00;0.0; 0d: 0h: 0min:16s:720ms
25;1;3.575;-0.00;0.0; 0d: 0h: 0min:16s:870ms
26;1;2.365;0.01;0.0; 0d: 0h: 0min:17s:30ms
27;1;2.019;0.00;0.0; 0d: 0h: 0min:17s:190ms
28;1;2.861;0.07;0.0; 0d: 0h: 0min:17s:350ms
29;1;4.144;0.08;0.0; 0d: 0h: 0min:17s:500ms
30;1;4.672;0.16;0.0; 0d: 0h: 0min:17s:720ms
31;1;4.039;0.22;0.0; 0d: 0h: 0min:17s:880ms
32;1;2.325;0.25;0.0; 0d: 0h: 0min:18s:40ms
33;1;2.041;0.30;0.0; 0d: 0h: 0min:18s:200ms
34;1;2.933;0.36;0.0; 0d: 0h: 0min:18s:350ms
35;1;4.204;0.42;0.0; 0d: 0h: 0min:18s:510ms
```

Abb. 5.4: Gespeicherte Daten des Versuchsbetriebs

Die Abb. 5.4 zeigt einen Auszug der gespeicherten Daten auf der SD-Karte, die während des Versuchsbetriebs aufgenommen worden sind. Die Formatierung der Daten ist Kapitel 4.5.3 zu entnehmen. Zunächst ist durch diesen Versuchsbetrieb zu erkennen, dass die Zeit zwischen den einzelnen Messungen im Durchschnitt 150ms bis 160ms beträgt, was einer Abtastfrequenz von ca. 6Hz entspricht. Falls eine höhere Abtastrate benötigt wird, um beispielsweise ein kurzzeitiges Ereignis wie einen Motorstart überwachen zu können, muss der Burst-Modus benutzt werden, vgl. hierzu die Bachelorarbeit von Herr Sassano [7]. Nachfolgend wird aus dieser Datei eine Grafik generiert, die den Spannungs- sowie Stromverlauf darstellt.

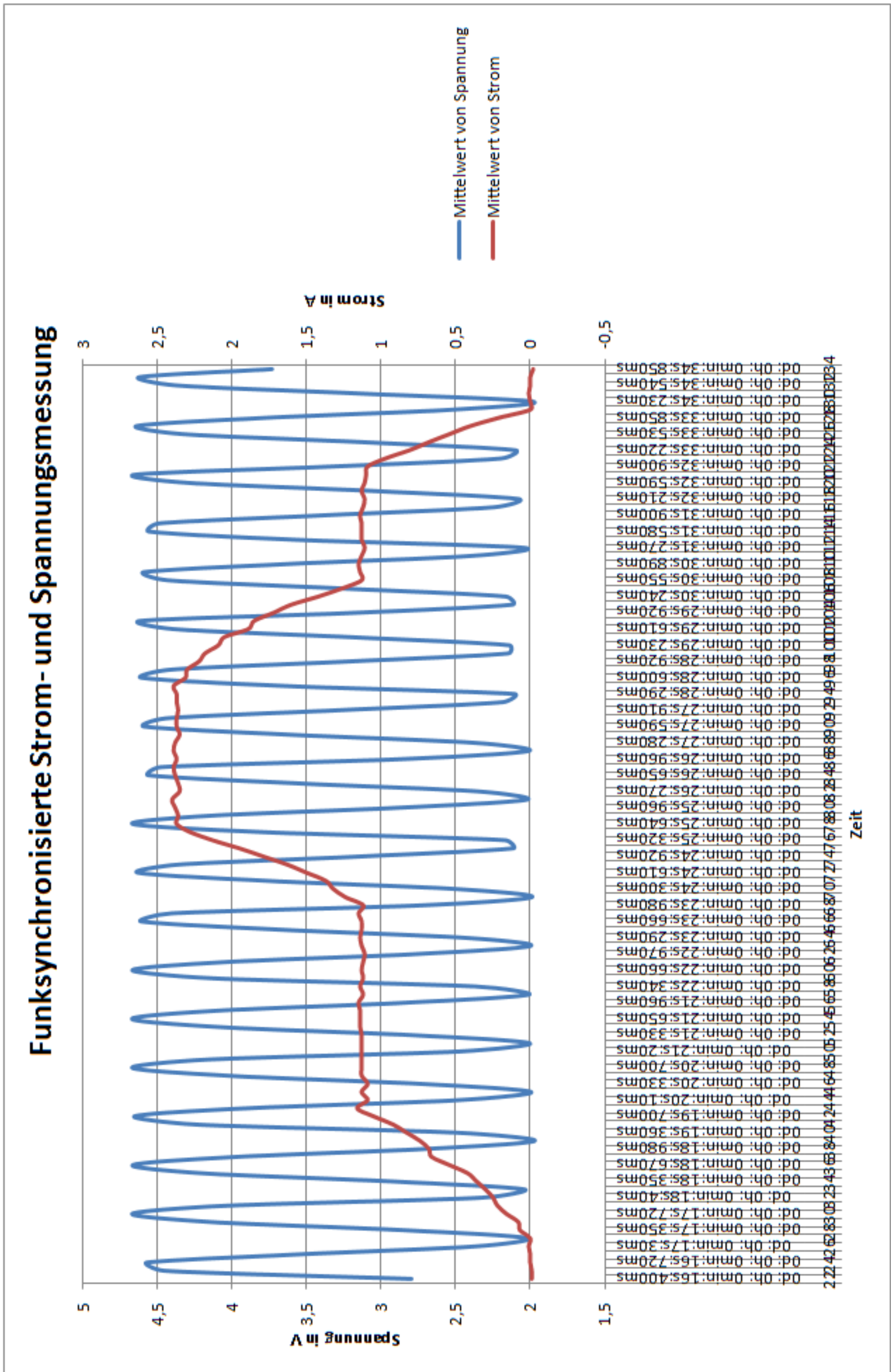


Abb. 5.5: Grafische Darstellung des Versuchsbetriebs

Die Abb. 5.5 stellt das Ergebnis des Versuchsbetriebs grafisch dar. Zunächst ist das Sinussignal (blaue Kurve) deutlich zu erkennen. Da der Sinus eine Frequenz von 1Hz hat und die Zellsensoren hier mit ca. 6Hz abtasten, kann eine genaue Nachbildung des Signals erfolgen. Ein direkter Vergleich zu dem Referenzverlauf, gemessen mit dem Oszilloskop, wird hier nicht durchgeführt. Dieser würde lediglich die Abweichung der Zellsensoren darstellen, was aber nicht Ziel dieser Arbeit ist.

Die rote Kurve stellt den gemessenen Stromverlauf dar. Wie bereits in Kapitel 5.2 erläutert, ist hierzu der stromdurchflossene Leiter viermal durch den Hall-Sensor gelegt worden, damit ein höherer Stromwert angezeigt wird. An dieser Stelle sei noch einmal darauf hingewiesen, dass der Stromfluss keinen Einfluss auf den Spannungsverlauf hat, da diese aus zwei verschiedenen Systemen kommen.

Nachfolgend sind die Werte der Strommessung erläutert:

- zum Startzeitpunkt der Messung fließt kein Strom durch den Leiter. (16s400ms)
- am Zeitpunkt 17s350ms wird der Ausgangsstrom des Stromversorgungsgerätes langsam auf 0,3A gedreht. Aufgrund des 4-fach durch den Hall-Sensor gelegten Leiters, wird ein Strom von ca. 1,2A gemessen.
- bei 23s980ms wird am Stromversorgungsgerät ein Ausgangsstrom von 0,6A eingestellt. Die Messung zeigt erneut den 4-fach erhöhten Wert von ca. 2,4A an.
- zum Zeitpunkt 28s600ms wird wieder ein Ausgangsstrom von 0,3A eingestellt. Der gemessene Wert zeigt 1,2A an.
- ab Zeitpunkt 34s230ms wird am Stromversorgungsgerät der Ausgangsstrom auf 0A gesetzt.

6 Fazit und Ausblick

6.1 Zusammenfassung der erreichten Ziele

In dieser Arbeit ist ein Neuentwurf des im BATSEN-Projekt verwendeten Batteriesteuergerätes auf Basis eines ARM-Controllers realisiert worden. Dazu wurde zunächst ein Konzept erstellt, welches die Anforderungen an das Batteriesteuergerät mit einbezog. Um dieses Konzept realisieren zu können, wurden die passenden Bauteile gewählt und das Platinenlayout erstellt. Für die Softwareentwicklung galt, es diese Hardware sinnvoll zu programmieren, um die Platine anschließend als Batteriesteuergerät in Betrieb nehmen zu können. Nachfolgend werden in Tabelle 6.1 noch einmal die gestellten Anforderungen sowie deren Lösungen zusammengefasst aufgelistet.

Anforderungen	Lösungen
ARM-Mikrocontroller soll eingesetzt werden.	ARM Cortex-M3 der Fa. Texas Instruments, Modellbezeichnung: LM3S9D92
Verschiedene Schnittstellen und Peripheriebausteine sollen integriert werden.	Folgende Schnittstellen und Peripheriebausteine sind vorhanden: • FTDI-Programer inkl. USB • RS232 • CAN • SD-Karte • Ethernet • Anschluss für Transceiver-Modul (SPI) • Hall-Sensor und 16-Bit ADC • LCD-Display mit Buttons zur Steuerung • Anschluss an NTC-Widerstand
Spannungsversorgung mit einem erweiterten Bereich möglicher Versorgungsspannungen	Das Batteriesteuergerät kann mit einer Spannung zwischen 6V und 40V versorgt werden.
Softwareentwurf für die Funkkommunikation mit dem Transceiver-Modul	Das Transceiver-Modul kann vollständig programmiert und gesteuert werden. Es wurde zudem die Einzelmessung an den

	Sensoren der Klasse 3 in Betrieb genommen und getestet.

Tabelle 6.1: Zusammenfassung der Anforderungen und Lösungen

Zusammenfassend lässt sich also sagen, dass die gesetzten Ziele alle erreicht wurden. Es kann über die Zellsensoren der Klasse 3 die Spannung gemessen werden, parallel dazu kann über einen externen Hall-Sensor der Stromfluss aufgenommen werden und die daraus resultierenden Daten können auf eine SD-Karte gespeichert werden. Die Steuerung erfolgt wie bereits bei dem Entwicklungsboard, welches als bisheriges Batteriesteuergerät diente, über den RS232-Anschluss. Zusätzlich gibt es nun die Möglichkeit der Steuerung über die eingebauten Buttons und das Display. Dadurch ist der Betrieb des Batteriesteuergerätes unabhängig von einem PC möglich.

6.2 Probleme während dieser Arbeit

Bei der Inbetriebnahme des Batteriesteuergerätes ist ein Kurzschluss mehrerer Vias unter dem LM3S9D92 Mikrocontroller festgestellt worden. Der Kurzschluss ist wahrscheinlich während des Lötvorgangs im Ofen entstanden. Durch diesen Kurzschluss sind folgende Leitungen kurzgeschlossen:

- SSI1TX – SPI Sendeleitung für das Transceiver-Modul
- NTC – Leitung für den NTC-Spannungsteiler
- GND

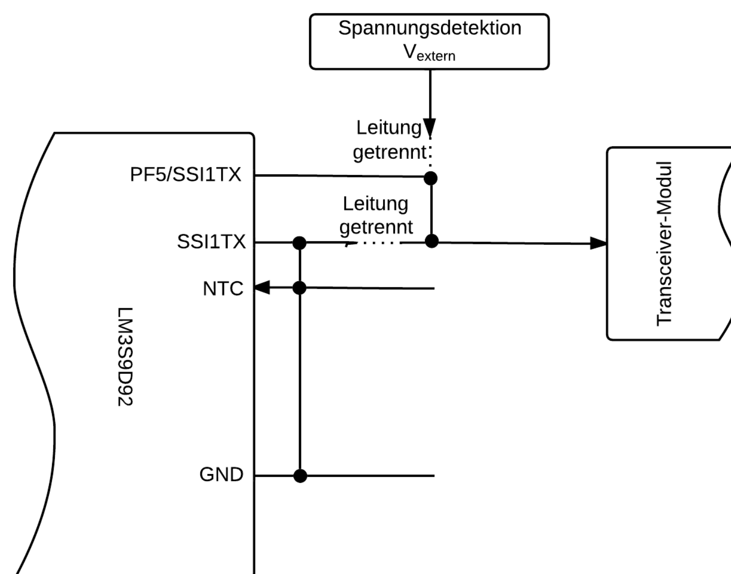


Abb. 6.1: Kurzschluss durch Via unter LM3S9D92

Um diesen Kurzschluss zu korrigieren, müsste der ganze Mikrocontroller entlötet werden, was diesen aller Wahrscheinlichkeit nach zerstören würde.

Als Lösung für die Sendeleitung des SPI wurde hier die SSI1TX-Funktion von dem Pin PF5 genutzt (Abb. 6.1). Dazu wurde eine Brücke zwischen den beiden Leitungen eingelötet. Die vorherige Funktion des Pins PF5, die externe Spannungsdetektion, ist somit auf dieser Platine nicht vorhanden.

Der NTC-Spannungsteiler kann wegen des Kurzschlusses mit GND nicht genutzt werden.

6.3 Ausblick

6.3.1 Kalibrierung des Hall-Sensors

Wie bereits in Kapitel 3.7 erwähnt, hat der Hall-Sensor eine Abweichung, die abhängig von der Temperatur sowie dem zu messenden Strom ist. Um diesen Fehler herausrechnen zu können, um einen möglichst genauen Wert zu erhalten, muss der Hall-Sensor komplett kalibriert werden.

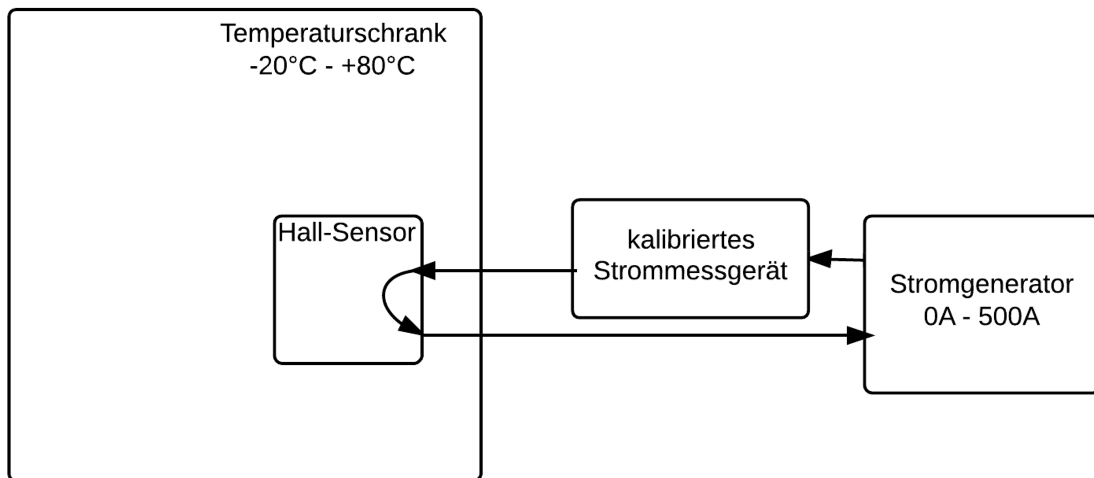


Abb. 6.2: Kalibrierung des Hall-Sensors

Eine Möglichkeit dieser Kalibrierung zeigt Abb. 6.2. Um das Temperaturverhalten des Hall-Sensors ermitteln zu können, muss dieser in einem Temperaturschrank bei verschiedenen Temperaturen Werte aufnehmen. Dabei muss bei jeder eingestellten Temperatur der gesamte Messbereich von ±500A gemessen werden. Parallel zu den Messungen des Hall-Sensors muss ein kalibriertes Messgerät den tatsächlichen Stromfluss ermitteln.

Wenn alle Messungen durchgeführt sind, müssen diese in einem Diagramm aufgetragen und mit den tatsächlichen Messwerten des kalibrierten Messgerätes abgeglichen werden. Diese Abweichungen können dann in der Software mit einprogrammiert werden, um so die Messungenauigkeit des Hall-Sensors zu minimieren.

Diese Kalibrierung konnte aufgrund des fehlenden Stromgenerators, der einen so hohen Strom generieren kann, nicht durchgeführt werden.

6.3.2 Zukünftige Erweiterungen

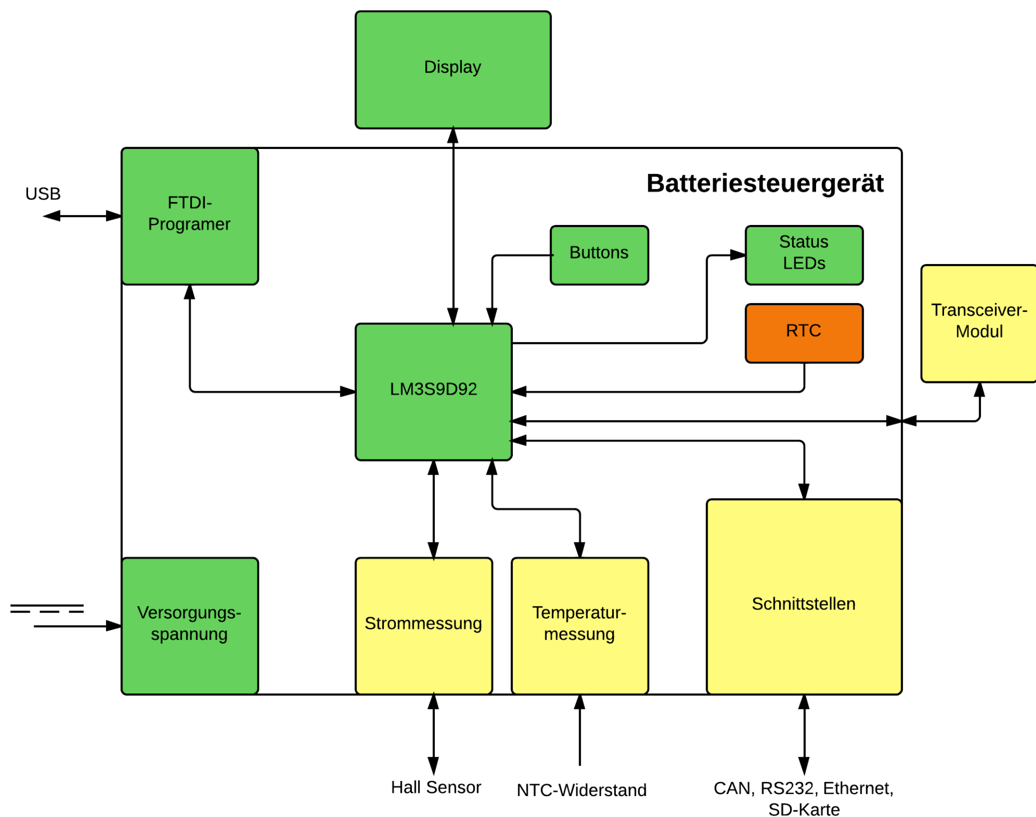


Abb. 6.3: Ausblick auf zukünftige Funktionen

Die Abb. 6.3 zeigt noch einmal die Gesamtübersicht des Batteriesteuergerätes. Alle grün eingefärbten Blöcke zeigen die bereits vollständig integrierten Funktionen. Die gelben Blöcke zeigen Module an, denen entweder die Software fehlt oder bei denen es in Zukunft noch Nachbesserungsbedarf gibt. Rot zeigt an, dass dieses Modul so nicht in dem Batteriesteuergerät vorhanden ist und in zukünftigen Modifikationen eingefügt werden kann. Nachfolgend eine genauere Erläuterung:

- Strommessung: Wie bereits in Kapitel 6.3.1 erläutert, sollte in Zukunft der Hall-Sensor komplett kalibriert werden. Diese Änderung kann rein softwareseitig integriert werden.

- Temperaturmessung: Da die Temperaturmessung in dieser Arbeit nur hardwareseitig implementiert worden ist, muss die Software noch geschrieben werden. Zu beachten ist hier, dass dies bei dem Prototypen des Batteriesteuergerätes nicht möglich ist. (vgl. Kapitel 6.2)

- Schnittstellen: Im Rahmen dieser Arbeit ist sowohl der Ethernet- als auch die CAN-Anschluss nur rein hardwareseitig integriert worden. Für die Inbetriebnahme dieser Schnittstellen wird die Software benötigt.

- Transceiver-Modul: Für das Transceiver-Modul ist die Hardware und Software komplett. Allerdings wurde für die Sensoren der Klasse 3 nur die einfache Spannungsmessung für die Inbetriebnahme programmiert. Hier kann noch der Burst-Modus umgesetzt werden.

- RTC: Ein Real-Time-Clock-Modul fehlt dem Batteriesteuergerät gänzlich. Um eine Echtzeit zu bekommen und diese den Messdaten zuordnen zu können, wird ein RTC-Chip benötigt. Dieser kann beispielsweise gesteuert werden über den I^2C-Bus, der im LM3S3D92 Mikrocontroller bereits vorahnden ist. Für die Stromversorgung dieses RTC-Chips wird zudem eine Batterie benötigt. Bei einer Modifikation des Batteriesteuergerätes ist diese Änderung dringend zu empfehlen.

7 Abbildungsverzeichnis

Abb. 1.1: Geschichte der Elektrofahrzeuge [2] 5
Abb. 1.2: Batterie mit n-Zellen [3] 6
Abb. 1.3: Batterie mit n-Zellsensoren [3] 7
Abb. 1.4: Zellspannungsverlauf - kritische Zelle [4] 7
Abb. 2.1: Bisheriges Batteriesteuergerät 9
Abb. 2.2: Transceiver Modul 11
Abb. 2.3: Zellsensor der Klasse 3 an einer Batteriezelle [7] 11
Abb. 3.1: Blockschaltbild Batteriesteuergerät 18
Abb. 3.2: High-Level Block Diagramm LM3S9D92 [16] 20
Abb. 3.3: Taktbaum LM3S9D92 [16] 22
Abb. 3.4: Beschaltung LM3S9D92 23
Abb. 3.5: Abweichung der PLL abhängig vom Main OSC [16] 24
Abb. 3.6: Blockschaltbild Versorgungsspannung 25
Abb. 3.7: Schaltplan der Anschlüsse der Versorgungsspannung 26
Abb. 3.8: Funktionsweise Step-Down Regler [19] 27
Abb. 3.9: Schaltplan LM2672-5.0 [20] 28
Abb. 3.10: Ausgangsspannung LM2672-5.0 bei 6V 29
Abb. 3.11: Ausgangsspannung LM2672-5.0 bei 10V 30
Abb. 3.12: Ausgangsspannung LM2672-5.0 bei 32V 30
Abb. 3.13: Application Note LM1117-3.3 [22] 31
Abb. 3.14: Verzögerung der Regelung auf 3,3V 32
Abb. 3.15: 3,3V Startup Zeit 3,3V 32
Abb. 3.16: Verzögerung Startup 5V 33
Abb. 3.17: Spannungsdetektion interne Batterie 34
Abb. 3.18: Spannungsdetektion externe Batterie 35
Abb. 3.19: Display EA W204B-NLW 36
Abb. 3.20: Zeichensatz des LCD-Moduls [25] 38
Abb. 3.21: Prellen der Taster [15] 39
Abb. 3.22: Schaltplan Push-Buttons 40
Abb. 3.23: Reaktionzeit bei Tastendruck 41
Abb. 3.24: PWM-Signal Periodendauer 42
Abb. 3.25: PWM-Signal Tastgrad 42
Abb. 3.26: Schnittstellen des Batteriesteuergerätes 43
Abb. 3.27: Spannungspegel beim Senden über RS232 44
Abb. 3.28: Spannungspegel beim Empfangen über RS232 45
Abb. 3.29: Größenvergleich SD-Karten [27] 46
Abb. 3.30: SD-Karten Pinbelegung [28] 47
Abb. 3.31: Mehrfachverwendung des SPI-Busses 48
Abb. 3.32: Detektion SD-Karte 49
Abb. 3.33: Detektion SD-Karte Schaltplan 49
Abb. 3.34: Schaltplan CAN-Bus 49
Abb. 3.35: Blockschaltbild Strommessung 51
Abb. 3.36: Absoluter Fehler Hall-Sensor ±75A [31] 52
Abb. 3.37: Absoluter Fehler Hall-Sensor ±500A [31] 52
Abb. 3.38: Temperaturmessung an dem Batteriesteuergerät - Schaltplan 54
Abb. 3.39: Steckverbindung des Transceiver-Moduls 55
Abb. 4.1: Zustandsdiagramm 58

Abb. 4.2: MODE1 Displayausgabe 61
Abb. 4.3: MODE2 Displayausgabe 61
Abb. 4.4: MODE102 Displayausgabe 61
Abb. 4.5: MODE3 Displayausgabe 61
Abb. 4.6: MODE103 Displayausgabe 61
Abb. 4.7: MODE4 Displayausgabe 61
Abb. 4.8: MODE104 Displayausgabe 61
Abb. 4.9: Main-Schleife 62
Abb. 4.10: Spannungsmessung 64
Abb. 4.11: Offsetkalibrierung Hall-Sensor 65
Abb. 4.12: Strommessung 67
Abb. 4.13: SD-Karte create_file 69
Abb. 4.14: SD-Karte add_to_file 70
Abb. 4.15: SD-Karte delete_file 71
Abb. 4.16: Zeitbasis 10ms 72
Abb. 4.17: Hauptmenü des Batteriesteuergerätes RS232 74
Abb. 4.18: RS232 Interrupt Handler 75
Abb. 4.19: Steuerung des Batteriesteuergerätes über die Buttons 76
Abb. 4.20: write_to_display-Funktion 77
Abb. 4.21: LCD-Display Adresse Offset [25] 78
Abb. 4.22: set_position 79
Abb. 4.23: LCD-Display write_char 80
Abb. 4.24: LCD-Display LEDCommand 81
Abb. 5.1: Übersicht Batteriesteuergerät 82
Abb. 5.2: Versuchsaufbau der Laborerprobung 83
Abb. 5.3: Aufnahme des generierten Signals durch das Oszilloskop 85
Abb. 5.4: Gespeicherte Daten des Versuchsbetriebs 86
Abb. 5.5: Grafische Darstellung des Versuchsbetriebs 87
Abb. 6.1: Kurzschluss durch Via unter LM3S9D92 90
Abb. 6.2: Kalibrierung des Hall-Sensors 91
Abb. 6.3: Ausblick auf zukünftige Funktionen 92

8 Tabellenverzeichnis

Tabelle 2.1: Nennspannungen verschiedener Bordnetze [13] 13
Tabelle 2.2: Vergleich LM3S9D92/MSP430F169 [16] [6].................... 14
Tabelle 3.1: Zulässige Hauptoszillator Frequenzen.................... 23
Tabelle 3.2: Dimensionierung Bauteile für LM2672-5.0.................... 28
Tabelle 3.3: Pinbelegung LCD-Modul 36
Tabelle 3.4: Relative Gesamtabweichung vom Hall-Sensor in Abhängigkeit von Strom und Temperatur.................... 53
Tabelle 4.1: Transitionen des Zustandsdiagrammes 59
Tabelle 4.2: Zustände des Zustandsdiagrammes.................... 60
Tabelle 4.3: Übersicht alle Interrupts 63
Tabelle 6.1: Zusammenfassung der Anforderungen und Lösungen 90

9 Literaturverzeichnis

[1] International Energy Agency Key World Energy Statistics 2013. (2013) www.iea.org. [Online]. http://www.iea.org/publications/freepublications/publication/KeyWorld2013.pdf

[2] Bundesministerium für Verkehr, Bau und Stadtentwicklung Elektromobilität. (2011, June) www.bmvbs.de. [Online]. http://www.bmvbs.de/SharedDocs/DE/Publikationen/VerkehrUndMobilitaet/elektromobilitaet-deutschland-als-leitmarkt-und-leitanbieter.pdf?__blob=publicationFile

[3] Matthias Schneider Karl-Ragmar Riemschneider, "Drahtlose Sensoren in den Zellen von Fahrzeug-Batterien," HAW-Hamburg, Mittweida, Konferenz 2011.

[4] S. Plaschke Diplomarbeit, "Experimentalsystem fu¨r drahtlose Batteriesensorik," HAW-Hamburg, Hamburg, 2008.

[5] Olimex MSP430-169STK. (2013, Dec.) www.olimex.com. [Online]. https://www.olimex.com/Products/MSP430/Starter/MSP430-169STK/

[6] Texas Instruments Datasheet MSP430F169. (2011, Mar.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/msp430f169.pdf

[7] Nico Sassano, Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung. Hamburg, Deutschland: HAW-Hamburg, 2013.

[8] Texas Instruments Datasheet CC1101. (2013, Nov.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/cc1101.pdf

[9] ecc Repenning GmbH Datasheet ECC-LFPP45 ECC-LFPP48. (2013, June) www.eccbatteries.com. [Online]. http://www.eccbatteries.com/files/datenblatt_lfpp_45-lfpp_48_06-2013.pdf

[10] Texas Instruments Datasheet MSP430F235. (2012, Dec.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/msp430f235.pdf

[11] Phillip Durdaut, Zellensensor für Fahrzeugbatterien mit Kommunikation und Wakeup-Funktion im ISM-Band bei 434 MHz. Hamburg, Hamburg, 2013.

[12] Stephan Plaschke, Experimentalsystem für drahtlose Batteriesensorik. Hamburg, 2008.

[13] o.V. Spannungsbereiche in verschiedenen Bordnetzen. (2013, Dec.) www.wikipedia.de. [Online]. http://de.wikipedia.org/wiki/Bordnetz

[14] Jan Schlüter Bachelorarbeit, Entwurf und Realisierung eines modular aufgebauten Experimentierboards für Mikrocontroller. Hamburg, Deutschland: HAW-Hamburg, 2013.

[15] Thomas Wisniewski Bachelorarbeit, Zyklierprüfstand für Batteriezellen mit Steuerung durch einen ARM-Controller sowie Messdatenverwaltung und Netzwerkanbindung. Hamburg, Deutschland: HAW-Hamburg, 2013.

[16] Texas Instruments Datasheet LM3S9D92. (2013, Apr.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/lm3s9d92.pdf

[17] Tobias Hammer HTerm. (2008, Nov.) www.der-hammer.info. [Online]. http://www.der-hammer.info/terminal/

[18] Vishay Datasheet B360B. (2012, Mar.) www.vishay.com. [Online]. http://www.vishay.com/docs/89122/b360b.pdf

[19] Walter Dvorak Abwärtswandler Schaltskizze. (2013, Dec.) Wikipedia. [Online]. http://upload.wikimedia.org/wikipedia/commons/thumb/f/f1/Buck_converter.svg/500px-Buck_converter.svg.png

[20] Texas Instruments Datasheet LM2672. (2013, Apr.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/lm2672.pdf

[21] Texas Instruments LM1117 800mA Low-Dropout Linear Regulator. (2013, Mar.) www.ti.com. [Online]. http://www.ti.com/product/lm1117-n

[22] Texas Instruments Datasheet LM1117. www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/lm1117-n.pdf

[23] AVX Datasheet AVX TRJ Kondensator. (12, 2013) www.avx.com. [Online]. http://www.farnell.com/datasheets/86282.pdf

[24] Fairchild Semiconductors Datasheet 2N7002. (1995, Nov.) www.fairchildsemi.com. [Online]. https://www.fairchildsemi.com/ds/2N/2N7000.pdf

[25] Electronic Assembly Datasheet EA W204B-NLW. (01, 2011) www.lcd-module.com. [Online]. http://www.lcd-module.com/deu/pdf/doma/4_20.pdf

[26] Texas Instruments Datasheet SN74AHCT14. (2003, July) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/sn74ahct14.pdf

[27] Wikipedia SD-Karten Vergleich. (2008, Nov.) Wikipedia. [Online]. http://upload.wikimedia.org/wikipedia/commons/thumb/6/67/SD_Cards.svg/500px-SD_Cards.svg.png

[28] SD-Karten Pinout. (2010, Jan.) www.elasticsheep.com. [Online]. http://elasticsheep.com/wp-content/uploads/2010/01/sd-card-pinout.png

[29] Texas Instruments Datasheet SNHVD1050 CAN Transceiver. (2010, Mar.) www.ti.com. [Online]. http://www.ti.com/lit/ds/symlink/sn65hvd1050.pdf

[30] Abracon Datasheet 25Mhz Quarz. (2010, Dec.) www.abracon.com. [Online]. http://www.abracon.com/Resonators/abm7.pdf

[31] LEM Datasheet DHAB S/24 Hall-Sensor. (2008, Feb.) www.lem.com. [Online]. http://www.lem.com/docs/products/dhab%20s24.pdf

[32] LOW NOISE, LOW POWER, 16-BIT, SIGMA DELTA ADC, Analog Devices AD7798 3-CHANNEL. (2013, July) www.analog.com. [Online]. http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad7798/products/product.html

[33] HIGH-ACCURACY 5.0V VOLTAGE REFERENCE, Analog Devices ADR4550 ULTRA-LOW-NOISE. (2012, Apr.) www.analog.com. [Online]. http://www.analog.com/en/special-linear-functions/voltage-references/adr4550/products/product.html

[34] Analog Devices Datasheet ADR4550/4530. (2012, Apr.) www.analog.com. [Online]. http://www.analog.com/static/imported-files/data_sheets/ADR4520_4525_4530_4533_4540_4550.pdf

[35] EPCOS Datasheet B57703M. (2013, Mar.) www.epcos.de. [Online]. http://www.epcos.de/inf/50/db/ntc_13/NTC_Probe_ass_M703.pdf

[36] Texas Instruments. (2013, Dec.) www.ti.com. [Online]. http://www.ti.com/product/lm3s9d92

[37] Texas Instruments LM2672 SIMPLE SWITCHER Power Converter High Efficiency 1A Step-Down Voltage Regulator. (2013, Apr.) www.ti.com. [Online]. http://www.ti.com/product/lm2672

10 Anhänge

10.1 Aufgabenstellung

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Hochschule für Angewandte Wissenschaften Hamburg
Department Informations- und Elektrotechnik
Prof. Dr.-Ing. Karl-Ragmar Riemschneider 22. Oktober 2013

Bachelorarbeit: Jakub Zimny

Hard- und Softwareentwurf eines Steuergerätes für drahtlose Batteriezellensensoren auf Basis eines ARM-Controllers

Motivation

Im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Forschungsvorhabens BATSEN (drahtlose Zellensensoren für Fahrzeugbatterien) und der Graduiertenschule 'Key Technologies for Sustainable Energy Systems in Smart Grids' soll der Zustand von Batterien durch Sensorik erfasst werden. Es werden Funksensoren untersucht, mit deren Hilfe Aussagen über den Lade- und Alterungszustand von Batterien möglich werden sollen. Die Sensoren selbst werden dabei im Inneren der jeweiligen Batteriezellen platziert. Dort messen sie Spannung und Temperatur und übertragen diese an ein Steuergerät außerhalb der Batterie. Dieses zentrale Steuergerät misst den Strom durch die Batterie, kombiniert diesen Wert mit den Daten der Sensoren und ermittelt daraus den ungefähren Lade- und Alterungszustand der Batterie. Für diese Aufgaben ist ein leistungsfähiger Mikrocontroller erforderlich.

Aufgabe

Herr Zimny erhält die Aufgabe eine einheitliche ARM-Mikrocontroller-Plattform als Grundlage für die verschiedenen Sensor-Klassen-spezifischen Batteriesteuergeräte aufzubauen und in Betrieb zu nehmen. Für die Plattform soll eine Platine mit einem ARM-Mikrocontroller der Fa. Texas Instruments entwickelt werden. Daneben sollen sollen verschiedene Schnittstellen und Peripheriebausteine auf der Platine integriert werden. Außerdem soll eine Spannungsversorgung mit einem erweiterten Bereich möglicher Versorgungsspannungen realisiert werden, um das Batteriesteuergerät aus einer möglichst großen Zahl verschiedener Batteriesystemen versorgen zu können. Die Plattform soll mit einem Empfänger bzw. Transceiver gekoppelt werden. Für die Funkkommunikation und die Datenaufzeichung soll Controllersoftware entworfen, implementiert und getestet werden. Ein Funktionstest mit den grundlegenden und einer komplexen Betriebsart der Sensoren soll erfolgen. Wesentlich ist der modulare Softwareaufbau, damit auch diese als Plattform für Erweiterungen dienen kann. Die Aufgabe gliedert sich wie folgt:

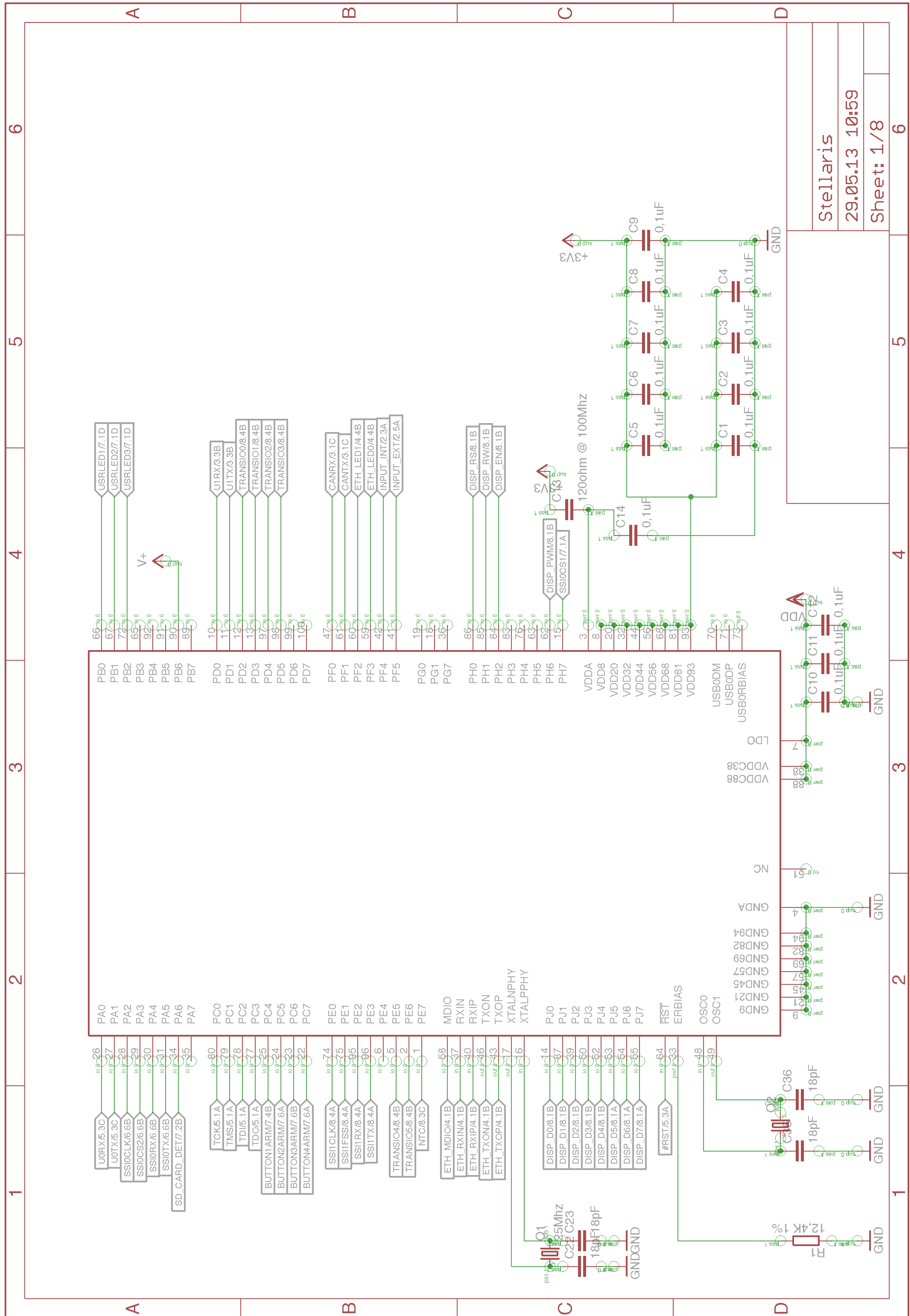
1) **Einarbeitung und Analyse der Rahmenbedingungen**
   - Einarbeitung in die Projektzielstellung
   - Einarbeiten in die Vorarbeiten innerhalb des Projektes, insbesondere in die funksynchronisierte Messung von Strom und Spannung
   - Erfassung von Anforderungen und Voruntersuchungen zu Bauelementen
2) **Konzepterstellung und Hardwareentwurf**
   - Festlegung des Controllers, der Schnittstellen und wesentlichen Module der Hardware
   - Erarbeitung des Schaltungskonzeptes
   - Versorgung der Platine und Konstruktion der Platine
   - Recherche und Auswahl von geeigneten Bauelementen
   - Entwurf des Schaltplans und Erstellen des Platinenlayouts
   - Aufbau und schrittweise Inbetriebnahme einiger Prototypen-Platinen

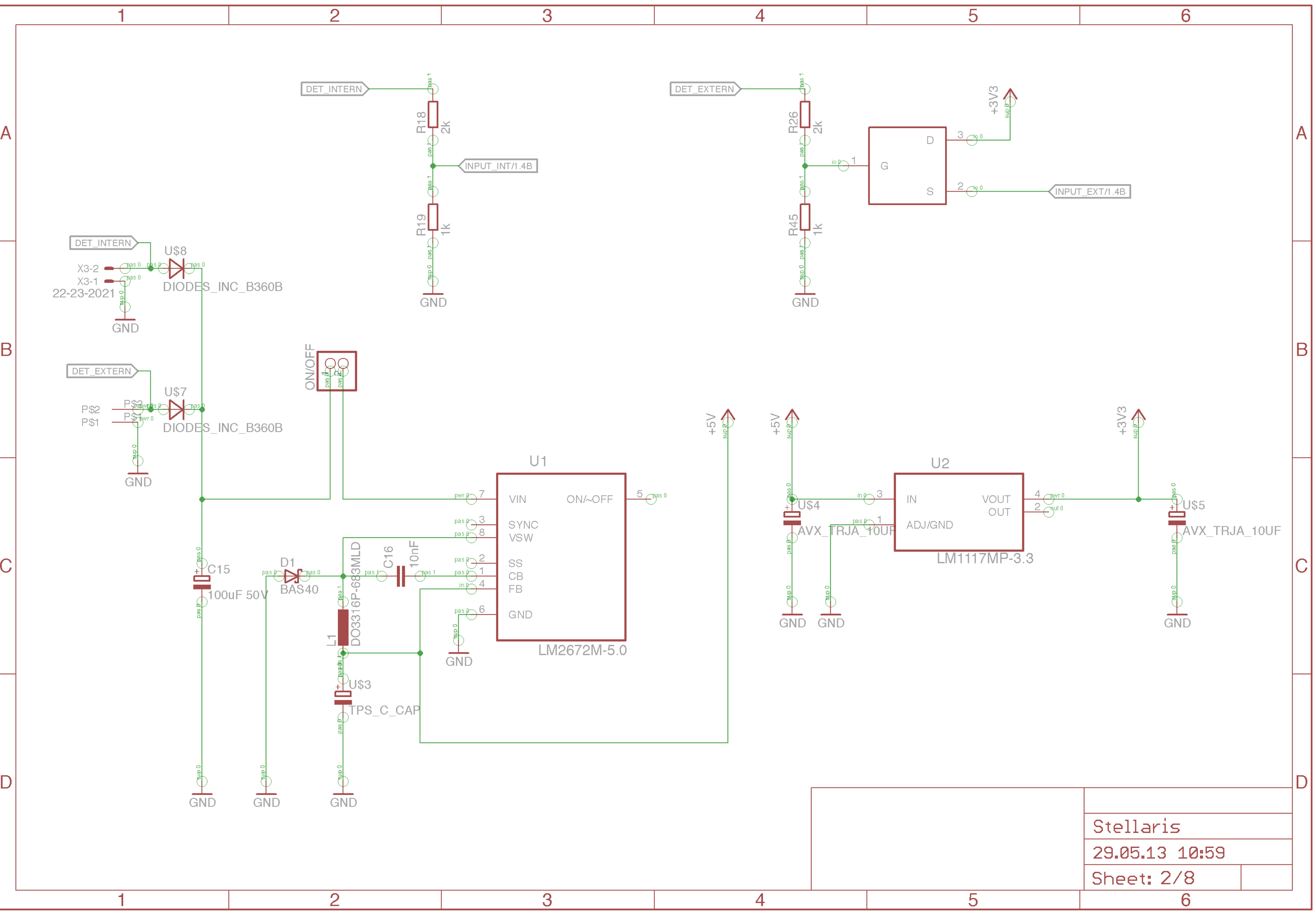
3) **Softwareentwicklung**
    - Erstellen von Software zur Demonstration der grundlegenden Hardwarefunktionen
    - Datenaufzeichnung auf SD-Karte und Zugang über LAN-Schnittstelle
    - Implementierung der Softwarefunktionen zum Betrieb der Platine als Batteriesteuergerät
    - Transceiverschnittstelle und Übertragungsprotokoll der Sensorklasse 3
    - Implementierung einer Zeitbasis
    - Einfache Kommunikation zum PC, Anzeige und Bedienfunktionen
    - Einbindung der Strommessung durch einen externen Hall-Sensor
4) **Funksynchronisierte Strom- und Spannungsmessung im Batteriesteuergerät**
    - Einarbeitung in das Prinzip der verteilten funksynchonisierten Messung von Spannung und Strom
    - Software-Implementierung der Sensor-Befehle für diesen Messmodus
    - Realisierung eines geeigneten Programmes zur Durchführung der funksynchonisierten Messung
5) **Laborerprobung und Versuchsbetrieb**
    - Inbetriebnahme im Labor und Test der Schaltung
    - Erstellen von Software zur Demonstration der grundlegenden Hardwarefunktionen
    - Implementierung der Softwarefunktionen zum Betrieb der Platine als Batteriesteuergerät für Klasse 3 Sensoren in Verbindung mit dem entsprechenden Receiver
    - Einbindung der Strommessung durch einen externen Hall-Sensor
    - Funktionsnachweise mit generierten Signale, anwendungsorientierte Demonstration
6) **Einordnung, Bewertung und Ausblick**
    - Zusammenfassung und Beurteilung der Lösung
    - Offene Punkte, Erfahrungen und Beobachtungen
    - Ausblick und mögliche Erweiterungen und Verbesserungen

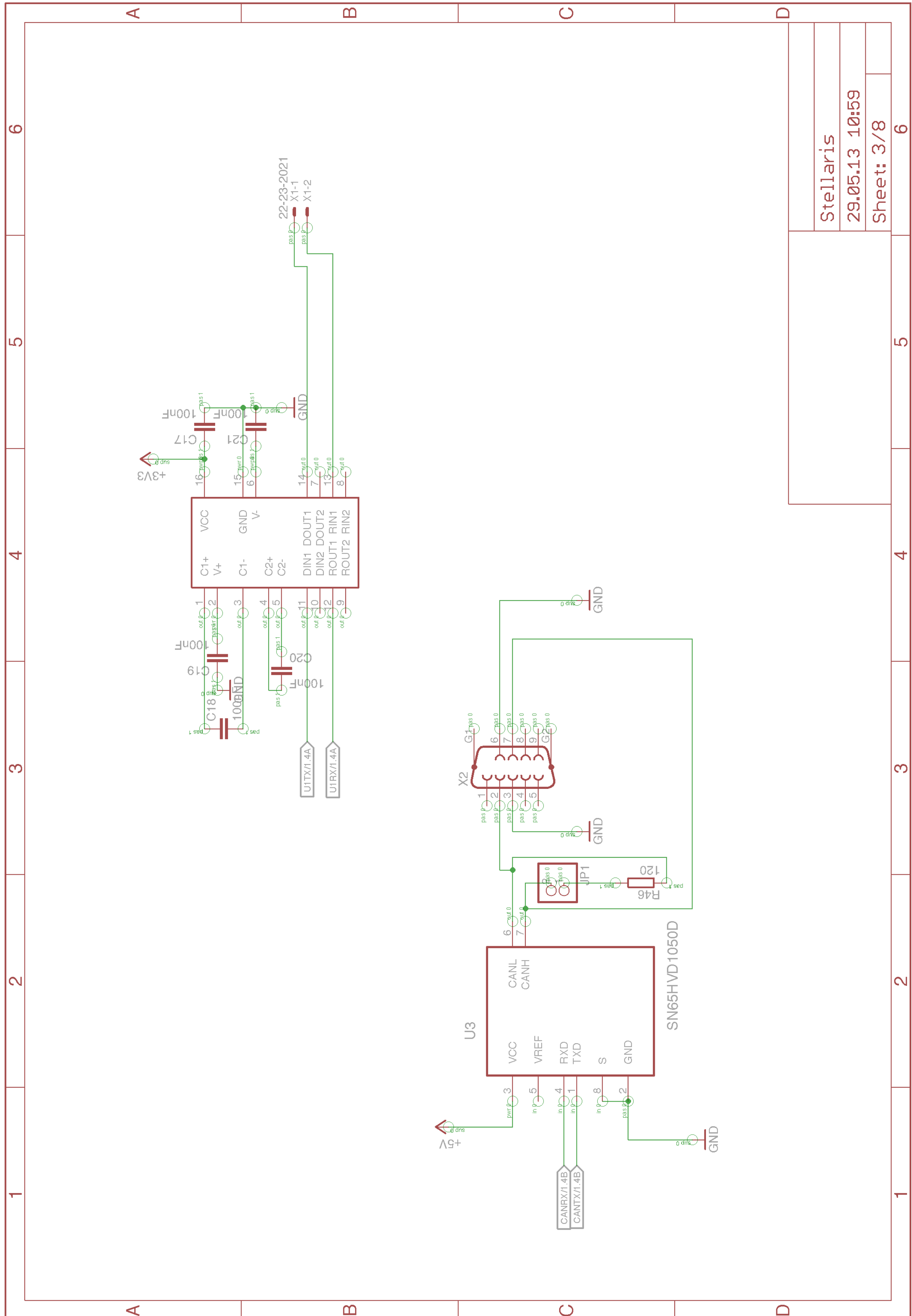
Dokumentation

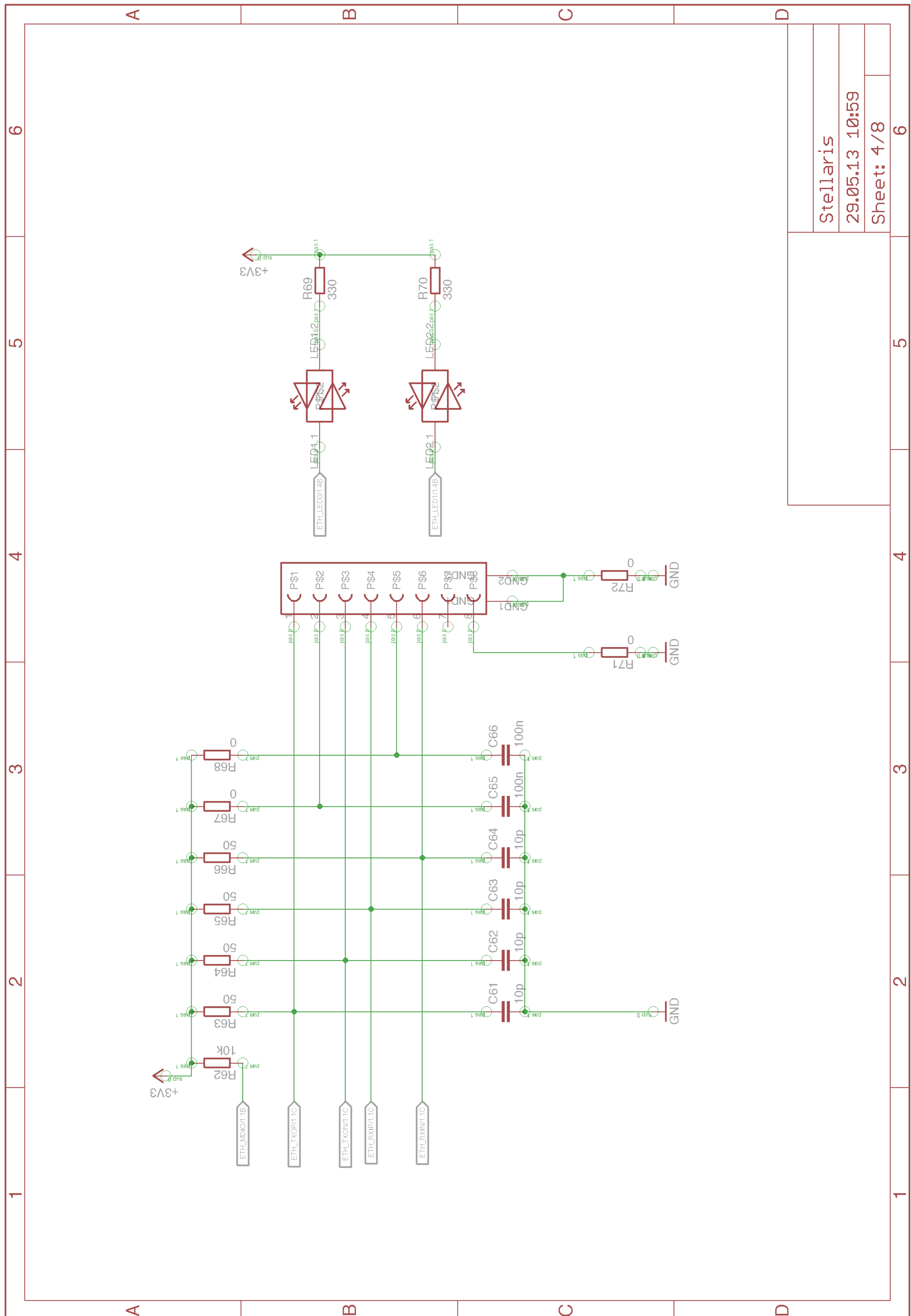
Die Fachliteratur, die Vorarbeiten und die kommerziellen Unterlagen sind zielgerichtet zu recherchieren. Die gewählte Lösung und die Funktionsweise sind gut nachvollziehbar zu dokumentieren. Die gesetzten Rahmenbedingungen, die Grundkonzeption, auftretende Probleme und wesentliche Folgerungen sollen beschrieben werden. Die Messergebnisse sind in exemplarischem Umfang zu erfassen und auszuwerten. Die realisierten Lösungen und die Ergebnisse sind kritisch einordnend zu bewerten. Ansätze für Verbesserungen und weitere Arbeiten sind zu nennen.

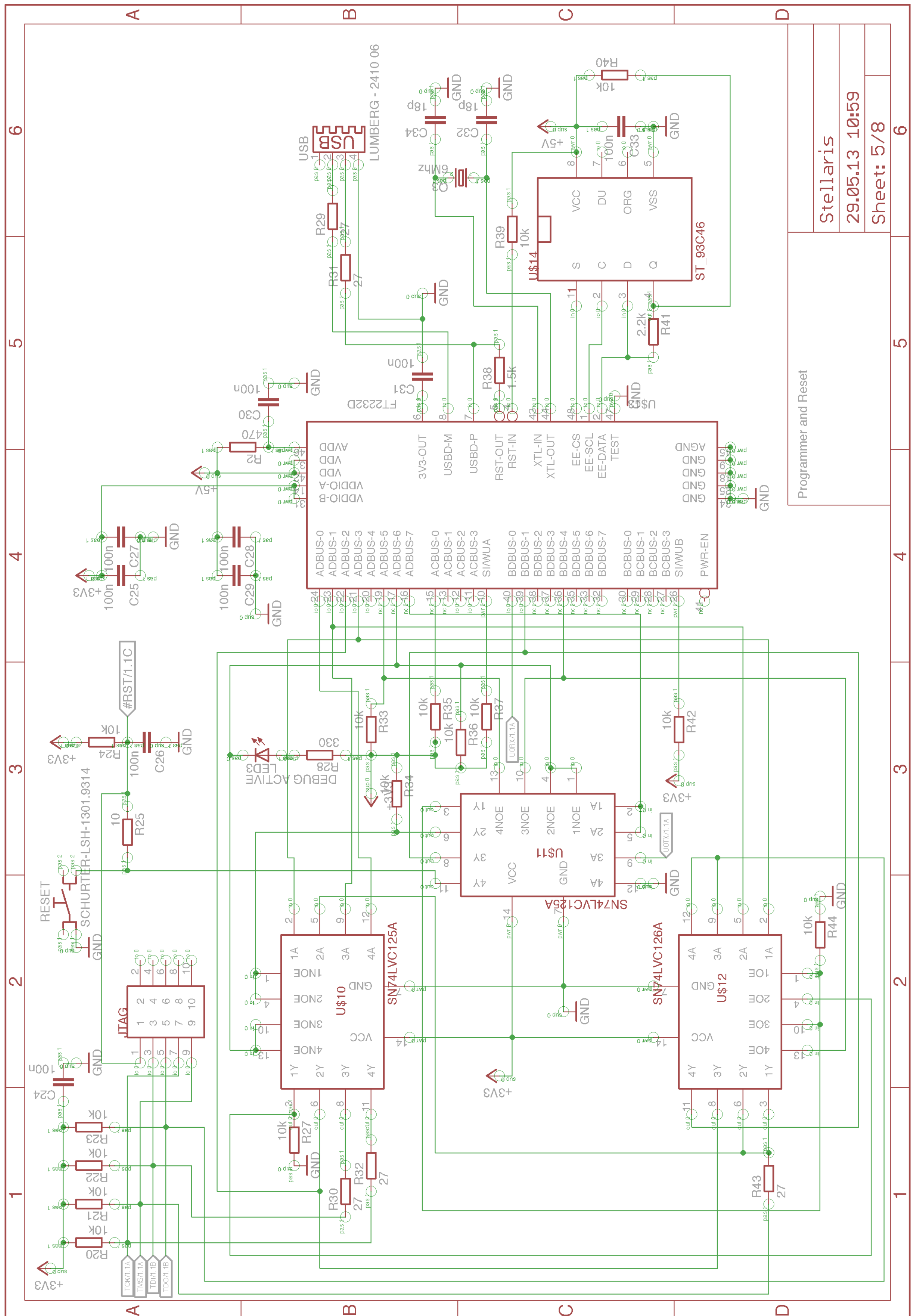
10.2 Schaltpläne

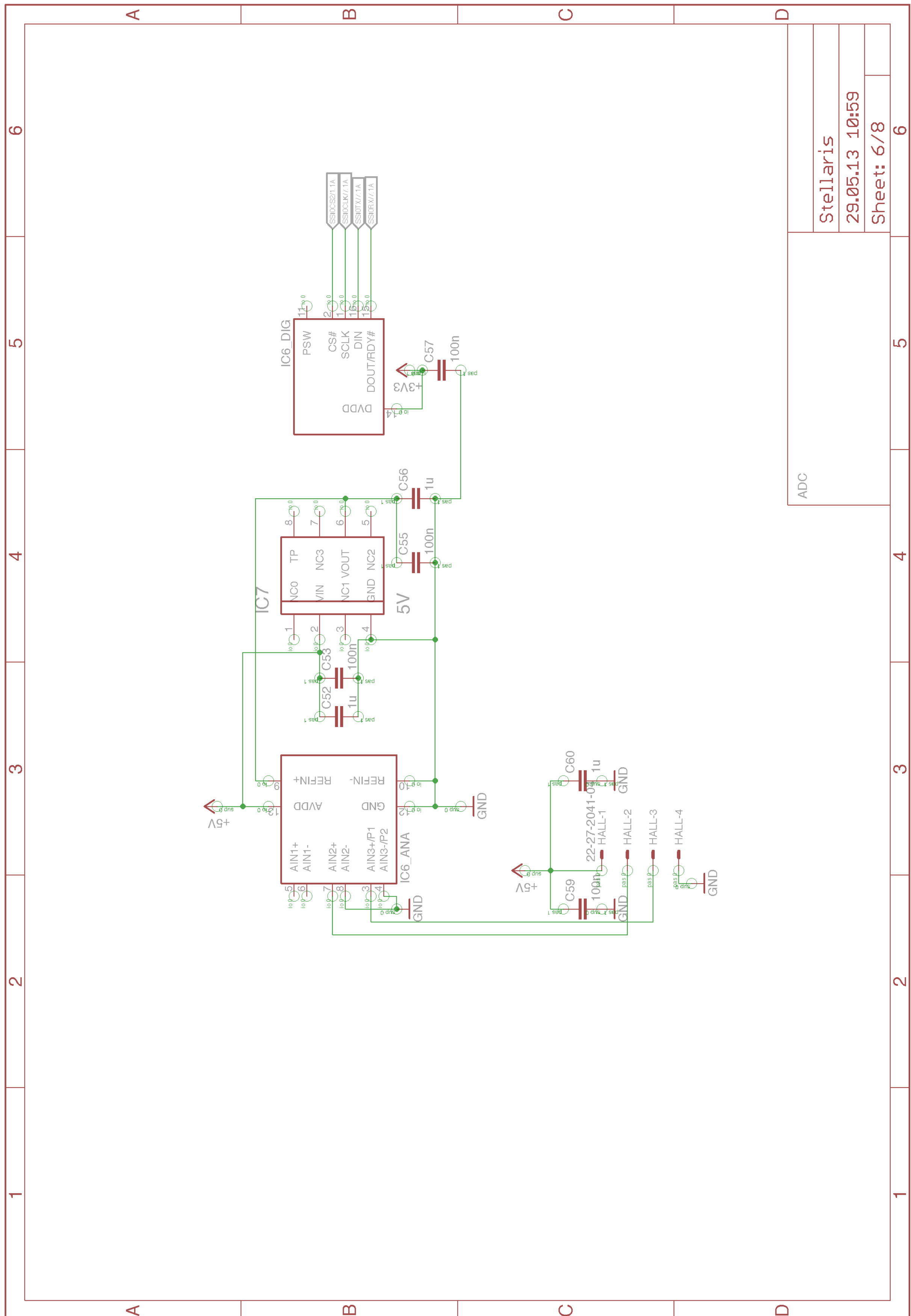


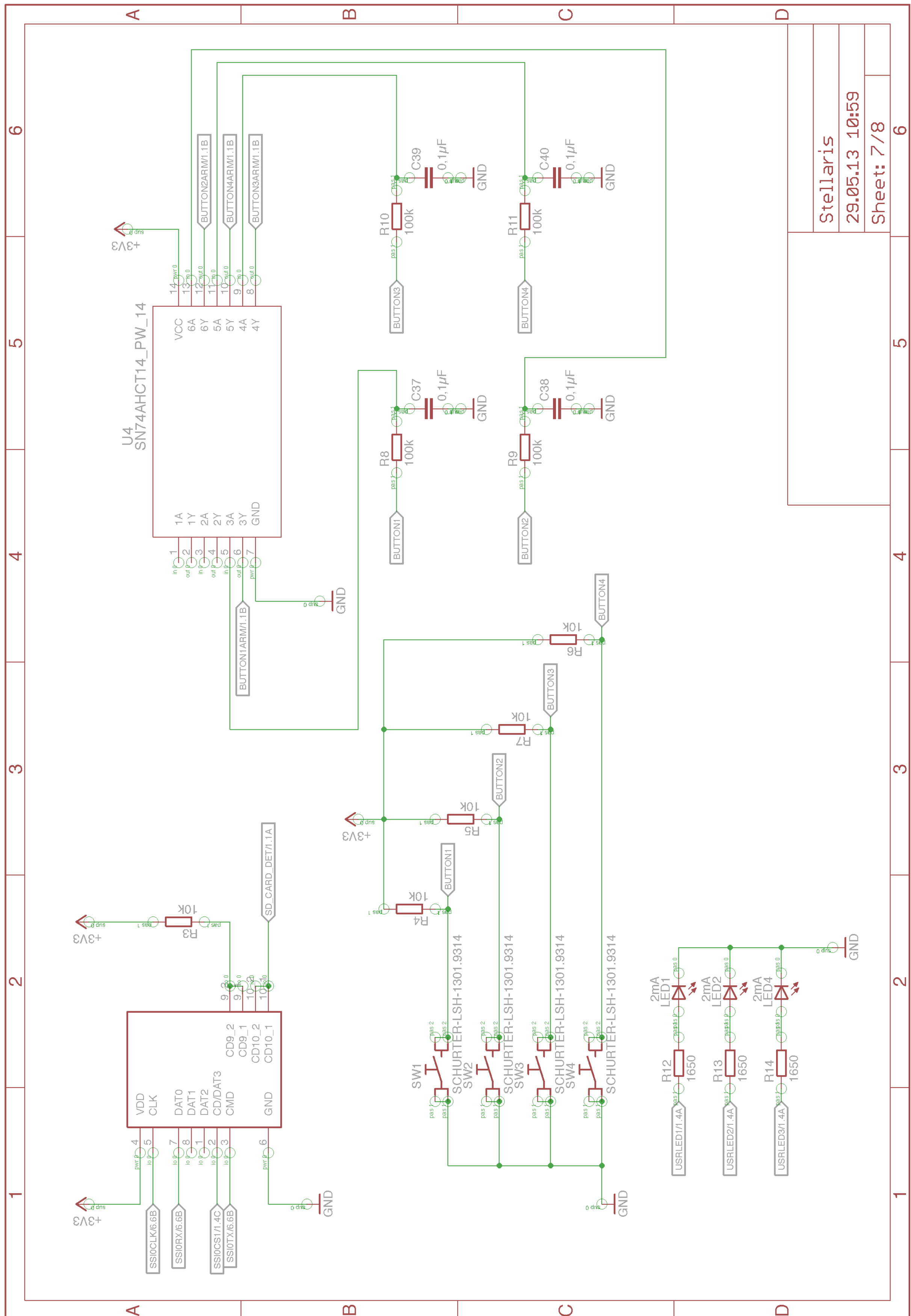

DET_INTERN
R18
2k
INPUT_INT/1.4B
R19
1k
GND
DET_EXTERN
R26
2k
R45
1k
D
G
S
+3V3
INPUT_EXT/1.4B
U$8
X3-2
X3-1
22-23-2021
DIODES_INC_B360B
U$7
P$2
P$1
ON/OFF
U1
VIN
ON/~OFF
SYNC
VSW
SS
CB
FB
GND
LM2672M-5.0
+5V
U2
IN
VOUT
OUT
ADJ/GND
LM1117MP-3.3
U$4
AVX_TRJA_10UF
U$5
C15
100uF 50V
D1
BAS40
DO3316P-683MLD
L1
C16
10nF
U$3
TPS_C_CAP
Stellaris
29.05.13 10:59
Sheet: 2/8

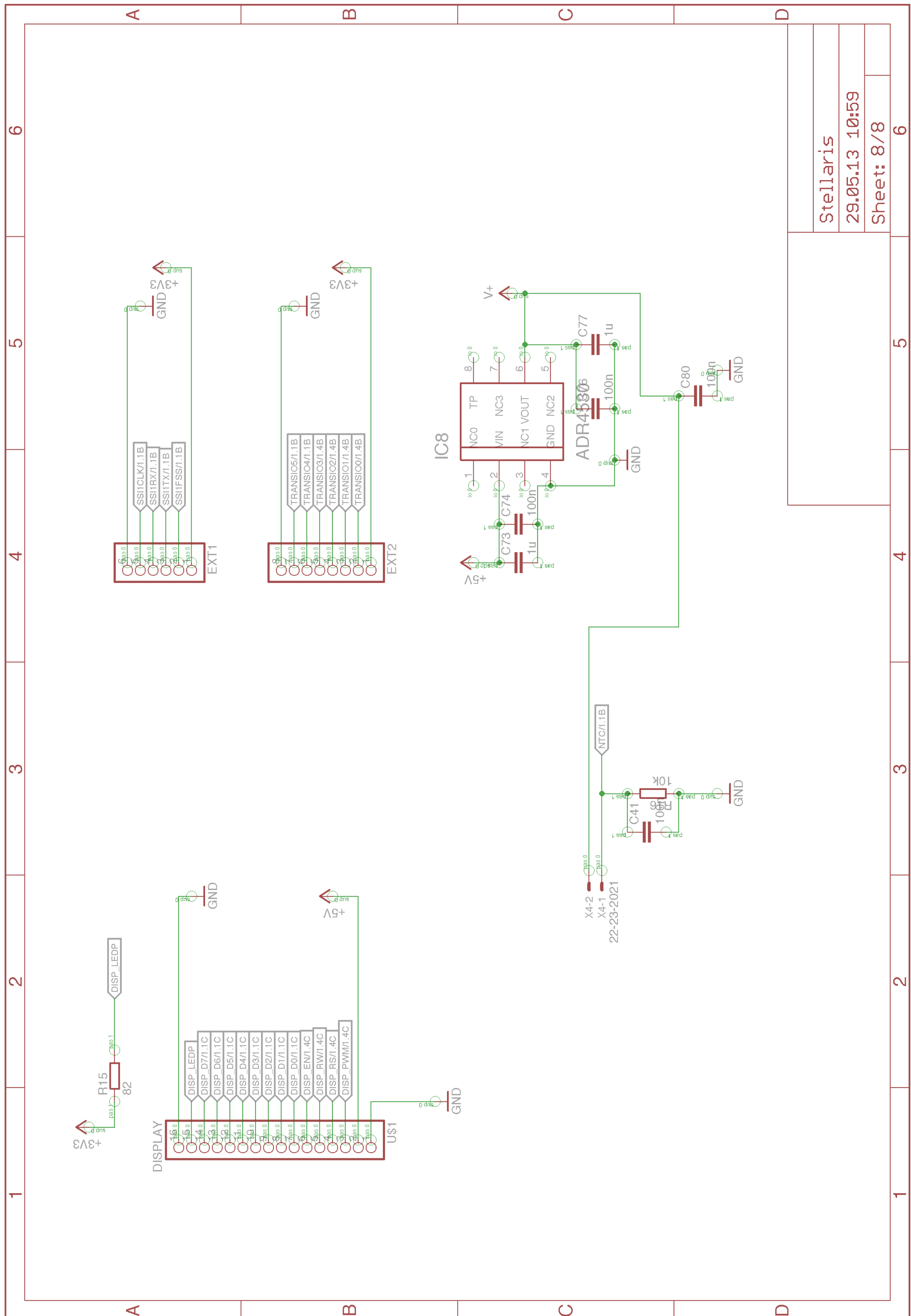

Stellaris
29.05.13 10:59
Sheet: 3/8
22-23-2021
X1-1
X1-2
GND
+3V3
100nF
C17
C21
C18
C19
C20
VCC
GND
V-
C1+
V+
C1-
C2+
C2-
DIN1
DIN2
ROUT1
ROUT2
DOUT1
DOUT2
RIN1
RIN2
U1TX/1.4A
U1RX/1.4A
X2
G1
G2
JP1
120
R46
U3
SN65HVD1050D
CANL
CANH
VREF
RXD
TXD
S
+5V
CANRX/1.4B
CANTX/1.4B


Stellaris
29.05.13 10:59
Sheet: 4/8
+3V3
R69
330
R70
330
GND
R72
0
R71
0
R68
R67
R66
R65
R64
R63
R62
10K
C66
100n
C65
100n
C64
10p
C63
10p
C62
10p
C61
10p

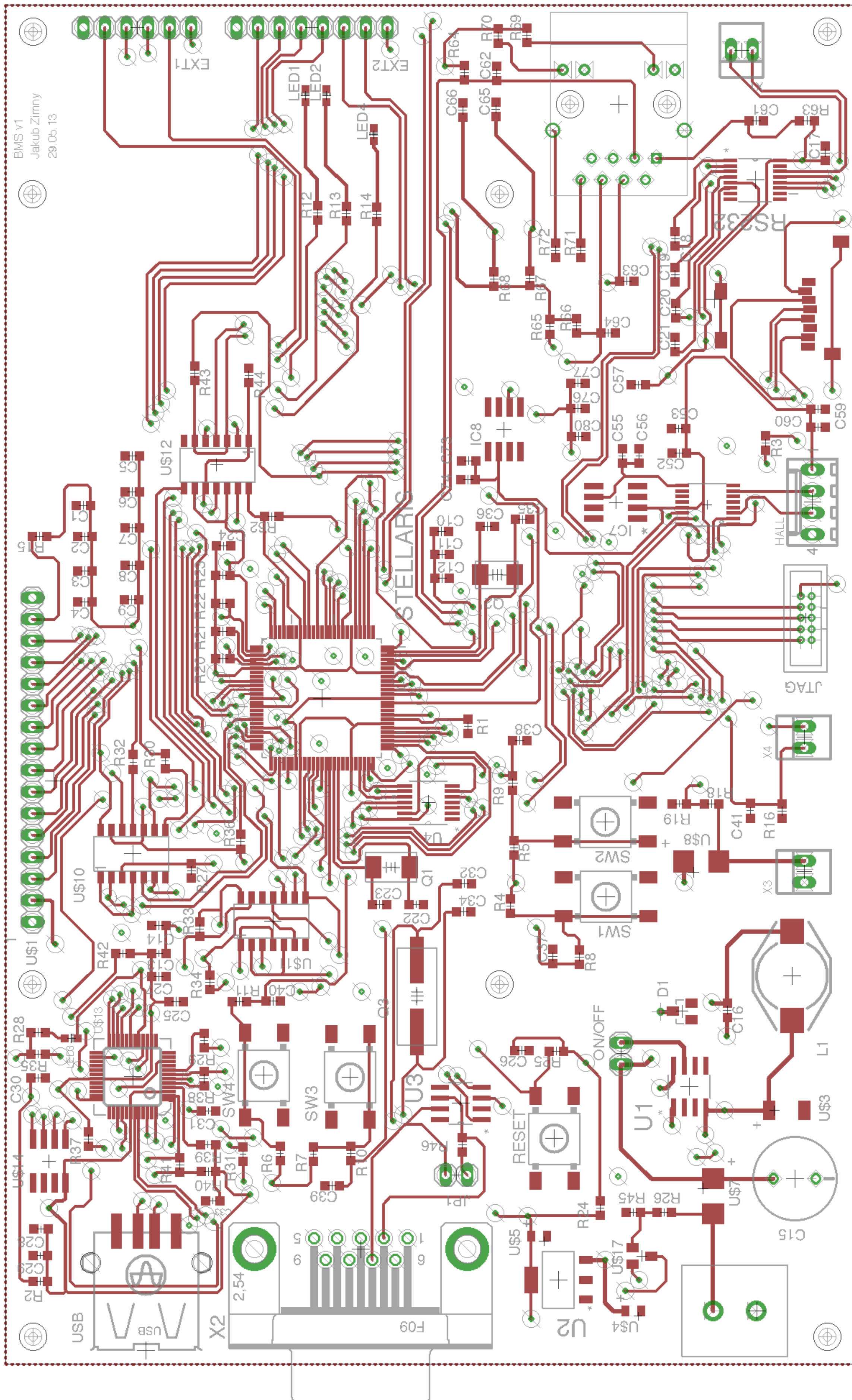

Stellaris
29.05.13 10:59
Sheet: 5/8
Programmer and Reset
FT2232D
ST_93C46
SN74LVC125A
SN74LVC126A
JTAG
RESET
SCHURTER-LSH-1301.9314
USB
LUMBERG - 2410 06
DEBUG ACTIVE
LED3
#RST/1.1C

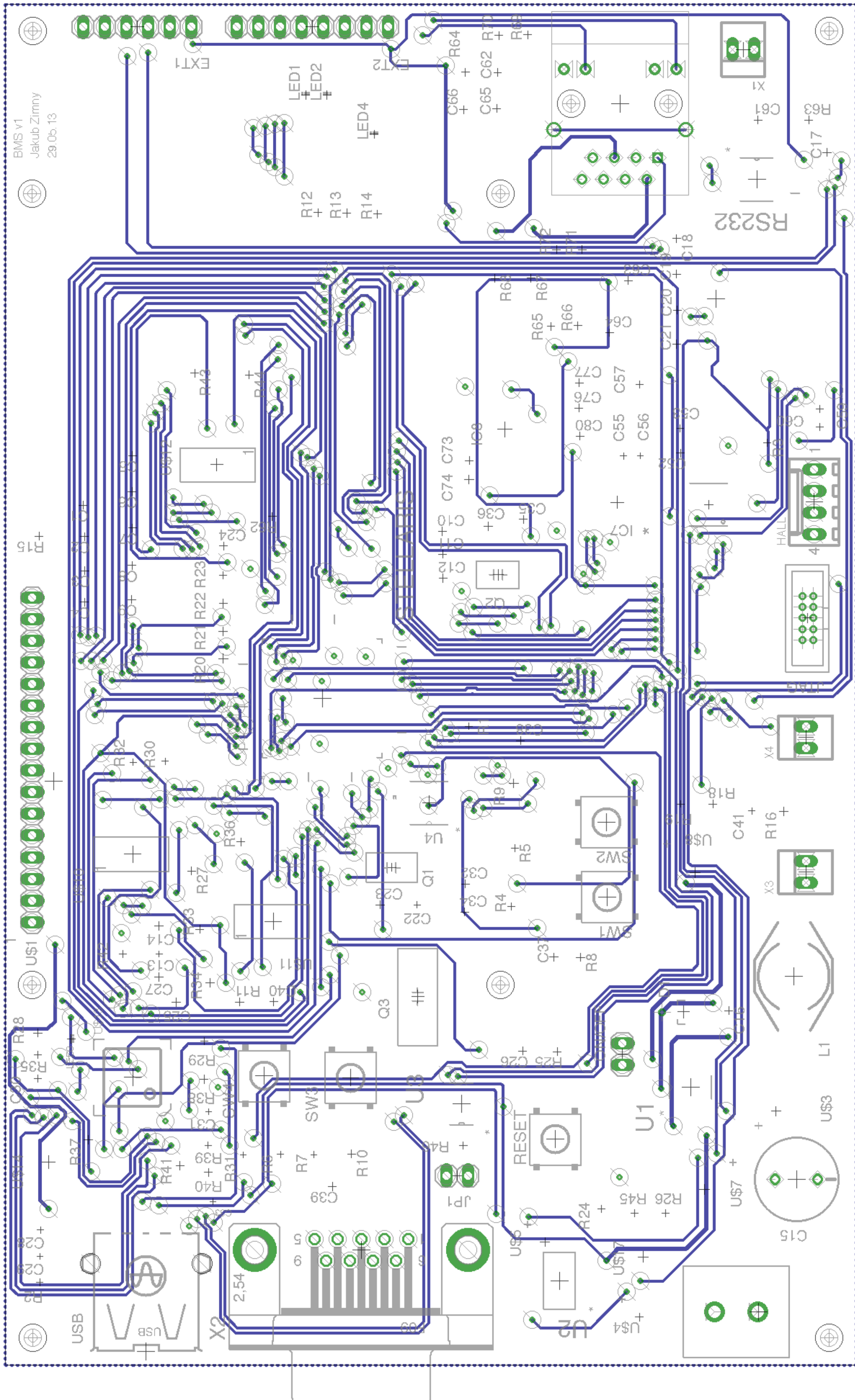

Stellaris
29.05.13 10:59
Sheet: 6/8
ADC
IC6_DIG
IC7
IC6_ANA
5V
+3V3
+5V
GND
C52
C53
C55
C56
C57
C59
C60
100n
1u
22-27-2041-04
HALL-1
HALL-2
HALL-3
HALL-4





10.3 Platinen Layouts





10.4 Quellcode

```
/*
 * ADCinit.c
 *
 *  Created on: 04.10.2013
 *      Author: Jakub Zimny
 */
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "header/myadc.h"



void ADCinit(void){
     //
         // Display the setup on the console.
         //
         UARTprintf("ADC Init...\n");

         //
         // The SSI0 peripheral must be enabled for use.
         //
         SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

         //Enable SSI0 Port
         //
         SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

         //
         // Configure the pin muxing for SSI0 functions
         // This step is not necessary if your part does not support pin muxing.
         //
         GPIOPinConfigure(GPIO_PA2_SSI0CLK);
         //GPIOPinConfigure(GPIO_PA3_SSI0FSS);
         GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_3);
         GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, 1);
         GPIOPinConfigure(GPIO_PA4_SSI0RX);
         GPIOPinConfigure(GPIO_PA5_SSI0TX);

         //
         // Configure the GPIO settings for the SSI pins.  This function also gives
         // control of these pins to the SSI hardware.  Consult the data sheet to
         // see which functions are allocated per pin.
         //
         GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 |
                        GPIO_PIN_5);

         //
         // Configure and enable the SSI port for SPI master mode.  Use SSI0,
         // system clock supply, idle clock level low and active low clock in
         // freescale SPI mode, master mode, 1MHz SSI frequency, and 8-bit data.
         // For SPI mode, you can set the polarity of the SSI clock when the SSI
         // unit is idle.  You can also configure what clock edge you want to
         // capture data on.  Please reference the datasheet for more information on
         // the different SPI modes.
         //
         SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
                            SSI_MODE_MASTER, 4000000, 8);

         //
         // Enable the SSI0 module.
         //
         SSIEnable(SSI0_BASE);

       // SSI0 Fifo l^schen
       adc_clear_fifo();

       // ADC reset
       ssi_cs(1);
       adc_write_read(0xFF);
```

```
        adc_write_read(0xFF);
        adc_write_read(0xFF);
        adc_write_read(0xFF);
        ssi_cs(0);




        // ca. 600 ms delay f¸r ADC reset
        delay(4000000);

        // unipolar mode, gain = 1, unbuffered, noref = 0x1000
        adc_set_config_reg(0x1000);

         // In den "power down" Modus gehen, um das "mode register" zu %ndern
        adc_set_mode_reg(0x600A);
        adc_set_offset_reg(0);  // Initialisierung des "offset registers" mit 0
        adc_set_fs_reg(0); // Initialisierung des "full scale registers" mit 0


        // Durchf¸hrung der internen Kalibrierung des ADCs
        adc_set_mode_reg(0x800A); // Internal zero offset calibration
        while((adc_status() & (1<<7)));
        adc_set_mode_reg(0xA00A); // Internal full scale calibration
        while((adc_status() & (1<<7)));

        // Pr¸fen, ob die Kalibierung erfolgreich war
        // (sich die Werte ge%ndert haben)
        if(adc_read_offset_reg() == 0)
        {
               UARTprintf("failed: offset calibration failed!\n");
        }
        if(adc_read_fs_reg() == 0)
        {
               UARTprintf("failed: full scale calibration failed!\n");
        }


          UARTprintf("ADC Init... done\n");

}
//1 == ADC;
void ssi_cs(int iCs){
      if (iCs == 1) {
        SSIDisable(SSI0_BASE);
        // Den Takt auf 4MHz stellen (hier ist noch Luft nach oben)
        SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3, SSI_MODE_MASTER,
4000000, 8);  // ADC=SSI_FRF_MOTO_MODE_3
        SSIEnable(SSI0_BASE);
        GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, 0x00);
      }
      else{
        GPIOPinWrite(GPIO_PORTA_BASE, GPIO_PIN_3, 0x08);
        SSIDisable(SSI0_BASE);
        // Den Takt auf 12,5MHz stellen (max)
        SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER,
12500000, 8);  // ADC=SSI_FRF_MOTO_MODE_3
        SSIEnable(SSI0_BASE);
      }


}


//Strommessung
float get_adc_current(){
    unsigned short usDataVol1, usDataVol2;
    float fTemp, fTemp1,fTemp2, fTemp3;

      //+-500A Ch
    usDataVol1 = adc_get_single_value_ch2();
    //+-75A Ch
    usDataVol2 = adc_get_single_value_ch3();

      fTemp = (5.00/65535.00*((float)usDataVol1)) - OffsetVoltage1;
      fTemp1 = (5.00/65535.00*((float)usDataVol2)) - OffsetVoltage2;
```

```
      //Current
      fTemp2 = (fTemp - 2.5)*(1.0/0.004);
      fTemp3 = (fTemp1 - 2.5)*(1.0/0.0267);

      //Wenn grˆfler 70A nimm +-500A range
      if (fTemp3>70) {
        return fTemp2;
      }
      else{
        return fTemp3;
      }
}

//HALL OFFSET
void get_hall_offset(){
    unsigned short usDataVol1, usDataVol2;
    float fTemp, fTemp1;

      //+-500A Ch
    usDataVol1 = adc_get_single_value_ch2();
    //+-75A Ch
    usDataVol2 = adc_get_single_value_ch3();

      fTemp = (5.00/65535.00*((float)usDataVol1));
      fTemp1 = (5.00/65535.00*((float)usDataVol2));

      OffsetVoltage1 = fTemp - 2.5;
      OffsetVoltage2 = fTemp1 - 2.5;

}


/*
 * Lˆschen des Hardware FIFOs vom SPI-Interface
 */
void adc_clear_fifo(void)
{

      unsigned long dummy;
      while(SSIDataGetNonBlocking(SSI0_BASE, &dummy));  // clear the fifo

}


/*
 * Lesen des Status Registers vom ADC.
 * Return: 1 Byte Status Register
 */
unsigned char adc_write_read(unsigned char data)
{

      adc_clear_fifo();

      unsigned long ans;
      SSIDataPut(SSI0_BASE, data);
      SSIDataGet(SSI0_BASE, &ans);

      return (unsigned char)(ans&0xff);

}
/*
 * Schreiben des Konfigurationsregsters
 * Das Register ist 2 Bytes grofl, es muss daher ein
 * zwei Byte short Wert ¸bergeben werden. Es erfolgt
 * keine Pr¸fung.
 */

void adc_set_config_reg(unsigned short value)
{
      ssi_cs(1);
      adc_write_read(0x10);
      adc_write_read((unsigned char)(value>>8));
      adc_write_read((unsigned char)(value&0xff));
      ssi_cs(0);
}
```

```
void adc_set_mode_reg(unsigned short value)
{
      ssi_cs(1);
      adc_write_read(0x08);
      adc_write_read((unsigned char)(value>>8));
      adc_write_read((unsigned char)(value&0xff));
      ssi_cs(0);
}

/*
 * Setzen des Fullscale Registers des ADC.
 * Es erfolgt keine Pr¸fung!
 */

void adc_set_fs_reg(unsigned short value)
{
      ssi_cs(1);
      adc_write_read(0x38);
      adc_write_read((unsigned char)(value>>8));
      adc_write_read((unsigned char)(value));
      ssi_cs(0);
}

/*
 * Setzen des Offsetregisters des ADC.
 * Es erfolgt keine Pr¸fung!
 */

void adc_set_offset_reg(unsigned short value)
{
      ssi_cs(1);
      adc_write_read(0x30);
      adc_write_read((unsigned char)(value>>8));
      adc_write_read((unsigned char)(value));
      ssi_cs(0);
}

unsigned char adc_status(void)
{
      ssi_cs(1);
      adc_clear_fifo();
      adc_write_read(0x40);
      unsigned char status_reg = adc_write_read(0x00);

      ssi_cs(0);
      return status_reg;
}

unsigned short adc_get_single_value(void)
{
      unsigned short data;

      // Configure unipolar mode, gain = 1, unbuffered, noref , CH1 = 0x1000
      adc_set_config_reg(0x1000);

      // set ADC to single conversion mode with f_ADC = 123 Hz = 0x2003
      // set ADC to single conversion mode with f_ADC = 470 Hz = 0x2001
      adc_set_mode_reg(0x2003);

      // wait for conversion to be done
      while((adc_status() & (1<<7)));

      data = (adc_read_data());

      return data;
}

unsigned short adc_get_single_value_ch2(void)
{
      unsigned short data;

      // Configure unipolar mode, gain = 1, unbuffered, noref , CH2 = 0x1001
      adc_set_config_reg(0x1001);

      // set ADC to single conversion mode with f_ADC = 123 Hz
```

```
    adc_set_mode_reg(0x2003);

    // wait for conversion to be done
    while((adc_status() & (1<<7)));

    data = (adc_read_data());
    //data = (adc_read_data()) + config.zerocurrentch1;

    return data;
}

unsigned short adc_get_single_value_ch3(void)
{
    unsigned short data;

    // Configure unipolar mode, gain = 1, unbuffered, noref , CH3 = 0x1002
    adc_set_config_reg(0x1002);

    // set ADC to single conversion mode with f_ADC = 123 Hz
    adc_set_mode_reg(0x2003);

    // wait for conversion to be done
    while((adc_status() & (1<<7)));

    data = (adc_read_data());
    //data = (adc_read_data())  + config.zerocurrentch2;

    return data;
}


/*
 * Auslesen der Daten vom ADC.
 * Es werden hierf¸r zwei Bytes gelesen und anschlieflend
 * zusammengesetzt. Diese Funktion wird nur intern
 * verwendet!
 */
unsigned short adc_read_data(void)
{
    unsigned short data;
    ssi_cs(1);
    adc_write_read(0x58);
    data = (adc_write_read(0x00))<<8;
    data |= adc_write_read(0x00);
    ssi_cs(0);
    return data;
}

/*
 * Auslesen des Fullscale Registers des ADC.
 * Dieses ist zwei Bytes grofl, die hier automatisch
 * beide gelsen und zusammen gesetzt werden.
 */
unsigned short adc_read_fs_reg(void)
{
    unsigned short data;
    ssi_cs(1);
    adc_write_read(0x78);
    data = (adc_write_read(0x00))<<8;
    data |= adc_write_read(0x00);
    ssi_cs(0);
    return data;
}

/*
 * Auslesen des Offsetregisters des ADC.
 * Dieses ist zwei Bytes grofl, die hier automatisch
 * beide gelsen und zusammen gesetzt werden.
 */
unsigned short adc_read_offset_reg(void)
{
    unsigned short data;
    ssi_cs(1);
    adc_write_read(0x70);
    data = (adc_write_read(0x00))<<8;
    data |= adc_write_read(0x00);
```

```
	ssi_cs(0);
	return data;
}
```

```
/*
 * myadc.h
 *
 *
 *      Author: Jakub Zimny
 */

#ifndef MAYADC_H_
#define MAYADC_H_

/*********************************************************************************
*
* Function Declarations
*
*********************************************************************************/

unsigned char adc_status(void);
unsigned short adc_read_config_reg(void);
unsigned char adc_write_read(unsigned char);
void adc_clear_fifo(void);
void adc_set_config_reg(unsigned short);
unsigned short adc_read_fs_reg(void);
unsigned short adc_read_offset_reg(void);
void adc_set_config_reg(unsigned short);
void adc_set_mode_reg(unsigned short);
void adc_set_fs_reg(unsigned short);
void adc_set_offset_reg(unsigned short);
unsigned short adc_read_data(void);
void ssi_cs(int iCs);
float get_adc_current();
void get_hall_offset();


extern void delay(unsigned long ticks);

void ADCinit(void);

unsigned short adc_get_single_value(void);
unsigned short adc_get_single_value_ch2(void);
unsigned short adc_get_single_value_ch3(void);



extern float OffsetVoltage1;
extern float OffsetVoltage2;
```

```
#endif /* MAYADC_H_ */
```

```
/*
 * buttons.c
 *
 *  Created on: 25.10.2013
 *      Author: Jakub Zimny
 */

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/timer.h"
#include "header/display.h"
#include "stdio.h"
#include "header/config.h"
#include "header/mycmdline.h"
#include "header/state.h"
#include "header/buttons.h"



void buttons_init(void)
{
      UARTprintf("Initializing Buttons...");
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
      //
    // Enable the GPIO that is used for the first push button
    //
    GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_4);


      //
    // Enable the GPIO that is used for the second push button
    //
      GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_5);


      //
    // Enable the GPIO that is used for the third push button
    //
      GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_6);



      //
    // Enable the GPIO that is used for the forth push button
    //

      GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, GPIO_PIN_7);



    //
    // Enable interrupts
    //

    GPIOIntTypeSet(GPIO_PORTC_BASE, GPIO_PIN_4,GPIO_RISING_EDGE);
    GPIOPinIntEnable(GPIO_PORTC_BASE, GPIO_PIN_4);
```

```
    GPIOIntTypeSet(GPIO_PORTC_BASE, GPIO_PIN_5,GPIO_RISING_EDGE);
    GPIOPinIntEnable(GPIO_PORTC_BASE, GPIO_PIN_5);


      GPIOIntTypeSet(GPIO_PORTC_BASE, GPIO_PIN_6,GPIO_RISING_EDGE);
      GPIOPinIntEnable(GPIO_PORTC_BASE, GPIO_PIN_6);


      GPIOIntTypeSet(GPIO_PORTC_BASE, GPIO_PIN_7,GPIO_RISING_EDGE);
      GPIOPinIntEnable(GPIO_PORTC_BASE, GPIO_PIN_7);

      IntEnable(INT_GPIOC);
      UARTprintf("Done\n ");

}

void GPIOPORTCHANDLER(void){
      IntDisable(INT_GPIOC);
      GPIOPinIntClear(GPIO_PORTC_BASE,GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 |GPIO_PIN_7);

      if (GPIOPinRead(GPIO_PORTC_BASE, GPIO_PIN_6) == 0x40) {
        BTN3Handler();
      }
      else if (GPIOPinRead(GPIO_PORTC_BASE, GPIO_PIN_5) == 0x20) {
        BTN2Handler();
      }
      else if (GPIOPinRead(GPIO_PORTC_BASE, GPIO_PIN_4) == 0x10) {
                BTN1Handler();
        }
      else if (GPIOPinRead(GPIO_PORTC_BASE, GPIO_PIN_7) == 0x80) {
                BTN4Handler();
        }
      IntEnable(INT_GPIOC);

}

void BTN1Handler(void){

      state = MODE1;
        }

void BTN2Handler(void){

      if (state_aktuell == MODE2) {
        //Hochz‰hlen des Files sowie Zur¸cksetzen des Counter f¸r Messungen
        ulCountSample=0;
        ulCountFile=ulCountFile + 1;
      }
      state = state_aktuell + 100;
        }

void BTN3Handler(void){

      if (state_aktuell == MODE1){
        state = MODE4;
      }
      else {
        state = state_aktuell - 1;
      }

}

void BTN4Handler(void){

      if (state_aktuell == MODE4){
        state = MODE1;
      }
      else {
        state = state_aktuell + 1;
      }

}
```

```
/*
 * buttons.h
 *
 *  Created on: 03.01.2014
 *      Author: Jakub Zimny
 */

#ifndef BUTTONS_H_
#define BUTTONS_H_


extern unsigned long ulCountSample;
extern unsigned long ulCountFile;
extern int state;
extern int state_aktuell;

void BTN1Handler(void);
void BTN2Handler(void);
void BTN3Handler(void);
void BTN4Handler(void);

#endif /* BUTTONS_H_ */
```

```
/*
 * CC1101.c
 *
 *  Created on: 28.10.2013
 *      Author: Jakub Zimny
 */
```

```
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "header/CC1101.h"
#include "header/state.h"
#include "driverlib/interrupt.h"

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/timer.h"
#include "header/display.h"
#include "stdio.h"
#include "header/config.h"
#include "header/mycmdline.h"
#include "header/state.h"



void cc1101_power_up_reset(void)
{
     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
     GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_1);

    // Manual power-on reset
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0x00);        // Chip enable
    // ca. 2ms
    delay(10000);
     GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0xFF);    // Chip disable

     delay(10000);
     GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0x00); // Chip enable

     delay(30000);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0xFF);         // Chip disable

    delay(30000);

    cc1101_reset();
}

void cc1101_set_tx(unsigned long brate) {    //26.03.13 NS

    cc1101_reset();         // Reset chip and go to idle state
    cc1101_config(brate); // Configure the transceiver for sending packets
}

void cc1101_set_rx(unsigned long brate) {    //26.03.13 NS

    cc1101_reset();         // Reset chip and go to idle state
    cc1101_config_rx(brate); // Configure the transceiver for sending packets
}

void cc1101_reset(void)
{
     SPI1Send(0x30);
     delay(500);
}

void cc1101_tx(unsigned long packetes_to_tx)
{
    // DATA packet length
```

```
     SPI1SendCC1101(CC1101_CR_PKTLEN, packetes_to_tx);

    // Enable CRC calculation when transmitting data
    //cc1101_enable_crc(); //Bei 0x04 kann Weg

    // TX state
    SPI1Send(CC1101_CS_STX);

    // Wait 5 ms for TX finished


    //delay_ms(5);

}

void cc1101_clear_tx_fifo(void)
{
     SPI1Send(CC1101_CS_SFTX);
}


void cc1101_fill_tx_fifo(unsigned long * buffer, unsigned long length)
{
    // Clear TX FIFO
    cc1101_clear_tx_fifo();

    // Transfer the bytes via SPI to the TX fifo of the transceiver
    cc1101_spi_write_register_burst(CC1101_ML_TXFIFO, buffer, length);
}

void set_spi_enable(int lowactive){
     if (lowactive == 0) {
       GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0x00);
     }
     else
     {
       GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0xFF);
     }
}

void cc1101_spi_write_register_burst(unsigned long address, unsigned long * buffer, unsigned
long count)
{
     unsigned long i;

     set_spi_enable(0); // Enable

    SPI1SendFifoTranseiver(address | CC1101_WRITE_BURST);   // Send address

    for (i = 0; i < count; i++) {

     SPI1SendFifoTranseiver(buffer[i]);                  // Send data

    }
    set_spi_enable(1);

}

void wait_for_ssi1(){
     // Wait until SSI1 is done transferring all the data in the transmit FIFO.
     //
     while(SSIBusy(SSI1_BASE))
     {
     }
}
/*
 Bei 80MHz Takt!
 10000000 ca 1,5s
 1000000 ca 150ms
 100000 ca 15ms
 10000 ca 1,5ms
 1000 150µs
 10 2µs
 1 600ns
 *
 */
```

```
void delay(unsigned long ticks){
      while (ticks) ticks--;
}

void SPI1SendFifoTranseiver(unsigned long Data){
      //Senden
      delay(5);
      SSIDataPut(SSI1_BASE, Data);
      wait_for_ssi1();
      delay(5);
}

void Enable_GDO2_IRQPIN(void){
      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
      GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_3);
      GPIOIntTypeSet(GPIO_PORTD_BASE, GPIO_PIN_3,GPIO_RISING_EDGE);
      GPIOPinIntEnable(GPIO_PORTD_BASE, GPIO_PIN_3);
      IntEnable(INT_GPIOD);
}

void Disable_GDO2_IRQPIN(void){
      GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_3);
      GPIOPinIntDisable(GPIO_PORTD_BASE, GPIO_PIN_3);
      IntDisable(INT_GPIOD);
}

void tx_packet(unsigned long packetes, unsigned long *txbuf) {

    cc1101_fill_tx_fifo(txbuf, packetes);
    GDO2_IRQ_MODE = 0x00;            // IRQ TX
    IRQ_TX_FLAG = 0;
    GPIOPinIntEnable(GPIO_PORTD_BASE, GPIO_PIN_3);
    cc1101_tx(packetes);
    while(!IRQ_TX_FLAG);
    IRQ_TX_FLAG = 0;

}

void cc1101_idle(void)
{
      SPI1Send(CC1101_CS_SIDLE);
}



void cc1101_enable_crc(void)
{
    unsigned long current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
    SPI1SendCC1101(CC1101_CR_PKTCTRL0, current | 0x04);
}

unsigned long cc1101_spi_read_register(unsigned long address)
{
    unsigned long x[1];
    set_spi_enable(0);
      delay(5);
      SSIDataPutNonBlocking(SSI1_BASE, (address | CC1101_READ_BURST));
      wait_for_ssi1();
      SSIDataPutNonBlocking(SSI1_BASE, 0x00);
      SSIDataGetNonBlocking(SSI1_BASE,&x[0]);
      wait_for_ssi1();
      x[0] &= 0x00FF;
      delay(5);
      set_spi_enable(1);
    return x[0];
}

unsigned long cc1101_spi_read()
{
    unsigned long x[1];
      SSIDataPutNonBlocking(SSI1_BASE, 0x00);
      SSIDataGetNonBlocking(SSI1_BASE,&x[0]);
      wait_for_ssi1();
    return x[0];
}
```

```
void cc1101_config(unsigned int brate) {
    // Rising edge on GDO0 when packet received and CRC check OK
    // (Deasserted when first byte is read from RX FIFO)
      SPI1SendCC1101(CC1101_CR_IOCFG0, 0x07);

      //Test1 = cc1101_spi_read_register(CC1101_CR_IOCFG0);
      //SPI1SendCC1101((CC1101_CR_IOCFG0 | CC1101_READ_BURST), 0x00);

    // Rising edge on GDO2 when sync word has been received
    SPI1SendCC1101(CC1101_CR_IOCFG2, 0x06);

    //Test1 = cc1101_spi_read_register(CC1101_CR_IOCFG2);
    //SPI1SendCC1101((CC1101_CR_IOCFG2 | CC1101_READ_BURST), 0x00);

    // The 4 SYNC bytes: 0x12 0x09 (repeated once)
    SPI1SendCC1101(CC1101_CR_SYNC1, 0x81);
    SPI1SendCC1101(CC1101_CR_SYNC0, 0x81);


    //Test1 = cc1101_spi_read_register(CC1101_CR_SYNC0);
    //SPI1SendCC1101((CC1101_CR_SYNC0 | CC1101_READ_BURST), 0x00);
    // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
    SPI1SendCC1101(CC1101_CR_PKTCTRL1, 0x0A);

    // No whitening, FIFO mode, CRC disable, Fixed packet length
    SPI1SendCC1101(CC1101_CR_PKTCTRL0, 0x04); //0x04 f¸r CRC!

    // Device Address
    SPI1SendCC1101(CC1101_CR_ADDR, 0xFF);

    //SPI1SendCC1101(CC1101_CR_ADDR | CC1101_READ_BURST, 0x00);


    // Channel 0
    SPI1SendCC1101(CC1101_CR_CHANNR, 0x00);

    // IF frequency: 152.34375 kHz
    SPI1SendCC1101(CC1101_CR_FSCTRL1, 0x06); //Ver‰ndern


    // Carrier frequency: 433.999969 MHz

    SPI1SendCC1101(CC1101_CR_FREQ2, 0x10);   //Ver‰ndern
    SPI1SendCC1101(CC1101_CR_FREQ1, 0xB1);   //Ver‰ndern
    SPI1SendCC1101(CC1101_CR_FREQ0, 0x3B);   //Ver‰ndern

    switch(brate) {

/*****************************************************************************************
        ** ‹bertragungstest
        ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************************/
        case 40: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8A);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*****************************************************************************************
        ** ‹bertragungstest
        ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000
kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

******************************************************************************************/
        case 60: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x2E);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
```

```
        }break;

/*****************************************************************************
        ** ‹bertragungstest
        ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************/
        case 80: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

/*****************************************************************************
        ** ‹bertragungstest
        ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************/
        case 100: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0xF8);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

/*****************************************************************************
        ** ‹bertragungstest
        ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************/
        case 120: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x2E);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

/*****************************************************************************
        ** ‹bertragungstest
        ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************/
        case 140: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x61);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

/*****************************************************************************
        ** ‹bertragungstest
        ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*****************************************************************************/
        case 160: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
```

```
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*****************************************************************************************
        ** ‹bertragungstest
        ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

******************************************************************************************/
        case 180: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0xC6);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

    }

    // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
    SPI1SendCC1101(CC1101_CR_MCSM0, 0x10);

    // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm ->
SmartRF Studio
    SPI1SendCC1101(CC1101_CR_FOCCFG, 0x16);     //Ver‰ndern

    //SPI1SendCC1101(CC1101_CR_FREND1, 0x00);



      set_spi_enable(0);
      delay(5);
      SSIDataPut(SSI1_BASE, 0x7E);
      wait_for_ssi1();

      delay(5);
      SSIDataPut(SSI1_BASE, 0x12);
      wait_for_ssi1();

      delay(5);
      SSIDataPut(SSI1_BASE, 0xc0);
      wait_for_ssi1();

      delay(5);
      SSIDataPut(SSI1_BASE, 0x02);
      wait_for_ssi1();

      delay(5);
      set_spi_enable(1);

    SPI1SendCC1101(CC1101_CR_FREND0, 0x11);    // 10 dBm output power
}

void cc1101_config_rx(unsigned int brate) {

      //SPI1Send(0x30);

    // Rising edge on GDO0 when packet received and CRC check OK
    // (Deasserted when first byte is read from RX FIFO)
      SPI1SendCC1101(CC1101_CR_IOCFG0, 0x07);

      //Test1 = cc1101_spi_read_register(CC1101_CR_IOCFG0);
      //SPI1SendCC1101((CC1101_CR_IOCFG0 | CC1101_READ_BURST), 0x00);

    // Rising edge on GDO2 when sync word has been received
    SPI1SendCC1101(CC1101_CR_IOCFG2, 0x06);

    //Test1 = cc1101_spi_read_register(CC1101_CR_IOCFG2);
    //SPI1SendCC1101((CC1101_CR_IOCFG2 | CC1101_READ_BURST), 0x00);

    // The 4 SYNC bytes: 0x12 0x09 (repeated once)
```

```
    SPI1SendCC1101(CC1101_CR_SYNC1, 0x81);
    SPI1SendCC1101(CC1101_CR_SYNC0, 0x81);


    //Test1 = cc1101_spi_read_register(CC1101_CR_SYNC0);
    //SPI1SendCC1101((CC1101_CR_SYNC0 | CC1101_READ_BURST), 0x00);
    // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
    SPI1SendCC1101(CC1101_CR_PKTCTRL1, 0x0A);

    // No whitening, FIFO mode, CRC disable, Fixed packet length
    SPI1SendCC1101(CC1101_CR_PKTCTRL0, 0x04); //0x04 f¸r CRC!

    // Device Address
    SPI1SendCC1101(CC1101_CR_ADDR, 0xFF);

    //SPI1SendCC1101(CC1101_CR_ADDR | CC1101_READ_BURST, 0x00);


    // Channel 0
    SPI1SendCC1101(CC1101_CR_CHANNR, 0x00);

    // IF frequency: 152.34375 kHz
    SPI1SendCC1101(CC1101_CR_FSCTRL1, 0x06); //Ver‰ndern


    // Carrier frequency: 433.999969 MHz

    SPI1SendCC1101(CC1101_CR_FREQ2, 0x10);   //Ver‰ndern
    SPI1SendCC1101(CC1101_CR_FREQ1, 0xB1);   //Ver‰ndern
    SPI1SendCC1101(CC1101_CR_FREQ0, 0x3B);   //Ver‰ndern

    switch(brate) {

/*******************************************************************************
        ** ‹bertragungstest
        ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*******************************************************************************/
        case 40: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8A);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*******************************************************************************
        ** ‹bertragungstest
        ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000
kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

********************************************************************************/
        case 60: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x2E);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*******************************************************************************
        ** ‹bertragungstest
        ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

********************************************************************************/
        case 80: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
```

```
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*************************************************************************************
        ** ‹bertragungstest
        ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
203.125000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*************************************************************************************/
        case 100: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8B);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0xF8);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*************************************************************************************
        ** ‹bertragungstest
        ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*************************************************************************************/
        case 120: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x2E);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*************************************************************************************
        ** ‹bertragungstest
        ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*************************************************************************************/
        case 140: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x61);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*************************************************************************************
        ** ‹bertragungstest
        ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*************************************************************************************/
        case 160: {
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x93);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;


/*************************************************************************************
        ** ‹bertragungstest
        ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth:
406.250000 kHz
        ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes

*************************************************************************************/
        case 180: {
```

```
            SPI1SendCC1101(CC1101_CR_MDMCFG4, 0x8C);
            SPI1SendCC1101(CC1101_CR_MDMCFG3, 0xC6);
            SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x33);
            SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);
            SPI1SendCC1101(CC1101_CR_MDMCFG0, 0x7A);
        }break;

    }

    // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
    SPI1SendCC1101(CC1101_CR_MCSM0, 0x10);

    // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm ->
SmartRF Studio
    SPI1SendCC1101(CC1101_CR_FOCCFG, 0x16);     //Ver%ndern

    //SPI1SendCC1101(CC1101_CR_FREND1, 0x00);

      SendBurst();
    SPI1SendCC1101(CC1101_CR_FREND0, 0x11);    // 10 dBm output power
}

void SendBurst(void)
{
      set_spi_enable(0);
      delay(5);
      SSIDataPut(SSI1_BASE, 0x7E);
      wait_for_ssi1();

      delay(5);
      SSIDataPut(SSI1_BASE, 0x12);
      wait_for_ssi1();

      delay(5);
      SSIDataPut(SSI1_BASE, 0xc0);
      wait_for_ssi1();

//    delay(5);
//    SSIDataPut(SSI1_BASE, 0x02);
//    wait_for_ssi1();

      delay(5);
      set_spi_enable(1);
}



void cc1101_config_no_packet_tx(void)
{
    // GDO2 Output Pin Configuration 2
    SPI1SendCC1101(CC1101_CR_IOCFG2, 0x2E);

    // GDO1 Output Pin Configuration 1
    SPI1SendCC1101(CC1101_CR_IOCFG1, 0x2E);

    // GDO0 Output Pin Configuration 0
    // Set for input data
    SPI1SendCC1101(CC1101_CR_IOCFG0, 0x2E);

    // RX FIFO and TX FIFO Thresholds
    SPI1SendCC1101(CC1101_CR_FIFOTHR, 0x07);

    // Sync Word, High Byte
    SPI1SendCC1101(CC1101_CR_SYNC1 , 0xD3);

    // Sync Word, Low Byte
    SPI1SendCC1101(CC1101_CR_SYNC0 , 0x91);

    // Packet Length
    SPI1SendCC1101(CC1101_CR_PKTLEN , 0xFF);

    // Packet Automation Control 1
    SPI1SendCC1101(CC1101_CR_PKTCTRL1, 0x04);

    // Packet Automation Control 0
    SPI1SendCC1101(CC1101_CR_PKTCTRL0, 0x32);
```

```
// Device Address
SPI1SendCC1101(CC1101_CR_ADDR, 0xFF);

// Channel Number
SPI1SendCC1101(CC1101_CR_CHANNR, 0x00);

// Frequency Synthesizer Control 1
SPI1SendCC1101(CC1101_CR_FSCTRL1, 0x0F);

// Frequency Synthesizer Control 0
SPI1SendCC1101(CC1101_CR_FSCTRL1, 0x00);

// Frequency Control Word, High Byte
SPI1SendCC1101(CC1101_CR_FREQ2, 0x10);

// Frequency Control Word, Middle Byte
SPI1SendCC1101(CC1101_CR_FREQ1, 0xB1);

//Frequency Control Word, LOW Byte
SPI1SendCC1101(CC1101_CR_FREQ0, 0x3B);

/**************************************************/
// Modem Configuration 4
// 250kBaud             ->  0xxD
// RX Filter 58.035714  ->  0xFx
SPI1SendCC1101(CC1101_CR_MDMCFG4, 0xFD);

// Modem Configuration 3
// 250kBaud -> 0x3B
SPI1SendCC1101(CC1101_CR_MDMCFG3, 0x3B);

// Modem Configuration 2
// OOK -> 0x30
SPI1SendCC1101(CC1101_CR_MDMCFG2, 0x30);

// Modem Configuration 1
// Channel spacing 199.951172kHz -> 0x22
SPI1SendCC1101(CC1101_CR_MDMCFG1, 0x22);

// Modem Configuration 0
// Channel spacing 199.951172kHz -> 0xF8
SPI1SendCC1101(CC1101_CR_MDMCFG0, 0xF8);
/**************************************************/

// Modem Deviation Setting
SPI1SendCC1101(CC1101_CR_DEVIATN, 0x15);

// Main Radio Control State Machine Configuration 2
SPI1SendCC1101(CC1101_CR_MCSM2, 0x07);

// Main Radio Control State Machine Configuration 1
SPI1SendCC1101(CC1101_CR_MCSM1, 0x30);

// Main Radio Control State Machine Configuration 0
SPI1SendCC1101(CC1101_CR_MCSM0, 0x18);

// Frequency Offset Compensation Configuration
SPI1SendCC1101(CC1101_CR_FOCCFG, 0x16);

// Bit Synchronization Configuration
SPI1SendCC1101(CC1101_CR_BSCFG, 0x6C);

// AGC Control 2
SPI1SendCC1101(CC1101_CR_AGCCTRL2, 0x03);

// AGC Control 1
SPI1SendCC1101(CC1101_CR_AGCCTRL1, 0x40);

// AGC Control 0
SPI1SendCC1101(CC1101_CR_AGCCTRL0, 0x91);

// High Byte Event0 Timeout 1
SPI1SendCC1101(CC1101_CR_WOREVT1, 0x87);

// High Byte Event0 Timeout 0
```

```
    SPI1SendCC1101(CC1101_CR_WOREVT0, 0x6B);

    // Wake On Radio Control
    SPI1SendCC1101(CC1101_CR_WORCTRL, 0x8F);

    // Front End RX Configuration 1
    SPI1SendCC1101(CC1101_CR_FREND1, 0x56);

    // Front End RX Configuration 0
    SPI1SendCC1101(CC1101_CR_FREND0, 0x11);

    // 10 dBm output power
    SPI1SendCC1101(CC1101_ML_PATABLE,0xc0);
}


void cc1101_tx_asynchronous_mode(void)
{
     SPI1Send(CC1101_CS_STX);
}

void wakeup_init(void){

     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
     GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_4);
     GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_2);
     //GDO2
     GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_3);
     GPIOIntTypeSet(GPIO_PORTD_BASE, GPIO_PIN_3,GPIO_RISING_EDGE);
     GPIOPinIntEnable(GPIO_PORTD_BASE, GPIO_PIN_3);
    //GDO0
    GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_4);
}

void cc1101_init_tx(void)
{
     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
     GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_3);
     GPIOPinIntDisable(GPIO_PORTD_BASE, GPIO_PIN_3);
}

void wakeup(void){
         cc1101_power_up_reset();
         cc1101_reset();
         cc1101_config_no_packet_tx();
         cc1101_tx_asynchronous_mode();
         delay(5000);

}

//Kommando zur einfachen Spannungsmessung
void sample_voltage_tx(void){
     unsigned long packet[4];

     cc1101_power_up_reset();

     packet[0] = BROADCAST;
     packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE;
     packet[2] = 0x00;
     packet[3] = 0x00;


     tx_data_length = 0;

     cc1101_set_tx(100);
     tx_packet(4, packet);


}

float sample_voltage_rx(void){
     unsigned long volt_msb=0, volt_lsb=0, ulVoltage=0;
     float fVoltage=0;

     GPIOPinIntDisable(GPIO_PORTD_BASE, GPIO_PIN_3);
     GPIOPinIntClear(GPIO_PORTD_BASE,GPIO_PIN_3);
```

```
	cc1101_power_up_reset();
	rx_data_length = 0;

	cc1101_set_rx(100);
	rx(6);

	while(!IRQ_RX_FLAG);
	IRQ_RX_FLAG = 0;

	volt_msb = (unsigned long)(( (packed[0] & 0x0F)  << 8) & 0x0F00);
	volt_lsb = (unsigned long)(( (packed[1] & 0xFF)  << 0) & 0x00FF);

	ulVoltage = volt_msb | volt_lsb;

	fVoltage = 5.0/4096*ulVoltage;


//	UARTprintf("%x\n",packed[0]); // Data MSB
//	UARTprintf("%x\n",packed[1]); // Data LSB
//	UARTprintf("%x\n",packed[2]); // BS
//	UARTprintf("%x\n",packed[3]); // Sensor ID
//	UARTprintf("%x\n",packed[4]); // 0
//	UARTprintf("%x\n",packed[5]); // 0

	return fVoltage;
}

void rx(int numb_of_pack) {

	SPI1SendCC1101(CC1101_CR_PKTLEN,numb_of_pack);

	SPI1Send(CC1101_CS_SFRX);	// Flush RX FIFO

	GDO2_IRQ_MODE = 0x01;		// IRQ RX
	GPIOPinIntClear(GPIO_PORTD_BASE,GPIO_PIN_3);
	GPIOPinIntEnable(GPIO_PORTD_BASE, GPIO_PIN_3);


	SPI1Send(CC1101_CS_SRX);	//start RX

}

void send_voltage(void){
	unsigned long packet[4];

	cc1101_power_up_reset();

	packet[0] = BROADCAST;
	packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE;
	packet[2] = 0x00;
	packet[3] = 0x00;

	tx_data_length = 0;

	cc1101_set_tx(100);
	tx_packet(4, packet);

}


void GDO2IntHandler(void){
	GPIOPinIntDisable(GPIO_PORTD_BASE, GPIO_PIN_3);
	GPIOPinIntClear(GPIO_PORTD_BASE,GPIO_PIN_3);

	if (GDO2_IRQ_MODE == 0x01) {
	  IRQ_RX_FLAG = 1;
	  delay(5000);

	  SPI1Send(CC1101_CS_SIDLE);	// Idle Mode

	  set_spi_enable(0);
	  delay(5);
	  SSIDataPut(SSI1_BASE, CC1101_ML_RXFIFO | CC1101_READ_BURST);
	  wait_for_ssi1();
	  packed[0] = cc1101_spi_read();
```

```
        packed[1] = cc1101_spi_read();
        packed[2] = cc1101_spi_read();
        packed[3] = cc1101_spi_read();
        packed[4] = cc1101_spi_read();
        packed[5] = cc1101_spi_read();

        delay(5);
        set_spi_enable(1);

      }

      if (GDO2_IRQ_MODE == 0x00) {
              IRQ_TX_FLAG = 1;
      }



}
```

```
/*
 * CC1101.h
 *
 *  Created on: 28.10.2013
 *      Author: Jakub Zimny
 */

#ifndef CC1101_H_
#define CC1101_H_



void cc1101_reset(void);

void cc1101_config(unsigned int brate);
void cc1101_config_no_packet_tx(void);
void cc1101_tx_asynchronous_mode(void);
void wakeup(void);
```

```c
void wakeup_init(void);
void cc1101_init_tx(void);
void cc1101_tx(unsigned long packetes_to_tx);
void cc1101_enable_crc(void);
void cc1101_spi_write_register_burst(unsigned long address, unsigned long * buffer, unsigned
long count);
void cc1101_idle(void);
unsigned long cc1101_spi_read_register(unsigned long address);
void cc1101_set_tx(unsigned long brate);
void sample_voltage_tx(void);
void send_voltage(void);
void SendBurst(void);
void SPI1SendFifoTranseiver(unsigned long Data);
void Enable_GDO2_IRQPIN(void);
void Disable_GDO2_IRQPIN(void);
void set_spi_enable(int lowactive);
void delay(unsigned long ticks);
void wait_for_ssi1(void);
void cc1101_config_rx(unsigned int brate);
void rx(int numb_of_pack);
unsigned long cc1101_spi_read(void);

extern void SPI1Send(unsigned long ulData);
extern void SPI1SendCC1101(unsigned long ulData1, unsigned long ulData2);
extern float get_adc_current();


//*******************************************************************************
//variables for Clock
//*******************************************************************************
extern unsigned long clock_msec;
extern unsigned long clock_sec;
extern unsigned long clock_min;
extern unsigned long clock_hour;
extern unsigned long clock_day;
extern unsigned long clock_month;
extern unsigned long clock_year;

extern volatile int rx_data_length;
extern volatile int GDO2_IRQ_MODE;
extern volatile char packed[6];
extern int state;
extern volatile int IRQ_TX_FLAG;
extern volatile int IRQ_RX_FLAG;


/*------------------------------------------------------------------------------
  Defines
 ------------------------------------------------------------------------------*/

/* Command Strobes (Table 42 in datasheet) */

#define cc1101_config_packet              0x30
#define CC1101_CS_SFSTXON           0x31
#define CC1101_CS_SXOFF             0x32
#define CC1101_CS_SCAL              0x33
#define CC1101_CS_SRX               0x34
#define CC1101_CS_STX               0x35
#define CC1101_CS_SIDLE             0x36
#define CC1101_CS_SAFC              0x37
#define CC1101_CS_SWOR              0x38
#define CC1101_CS_SPWD              0x39
#define CC1101_CS_SFRX              0x3A
#define CC1101_CS_SFTX              0x3B
#define CC1101_CS_SWORRST           0x3C
#define CC1101_CS_SNOP              0x3D

/* Configuration Registers */

#define CC1101_CR_IOCFG2            0x00     /* GDO2 output pin configuration */
#define CC1101_CR_IOCFG1            0x01     /* GDO1 output pin configuration */
#define CC1101_CR_IOCFG0            0x02     /* GDO0 output pin configuration */
#define CC1101_CR_FIFOTHR           0x03     /* RX FIFO and TX FIFO thresholds */
#define CC1101_CR_SYNC1             0x04     /* Sync word, high byte */
#define CC1101_CR_SYNC0             0x05     /* Sync word, low byte */
#define CC1101_CR_PKTLEN            0x06     /* Packet length */
```

```
#define CC1101_CR_PKTCTRL1          0x07     /* Packet automation control */
#define CC1101_CR_PKTCTRL0          0x08     /* Packet automation control */
#define CC1101_CR_ADDR              0x09     /* Device address */
#define CC1101_CR_CHANNR            0x0A     /* Channel number */
#define CC1101_CR_FSCTRL1           0x0B     /* Frequency synthesizer control */
#define CC1101_CR_FSCTRL0           0x0C     /* Frequency synthesizer control */
#define CC1101_CR_FREQ2             0x0D     /* Frequency control word, high byte */
#define CC1101_CR_FREQ1             0x0E     /* Frequency control word, middle byte */
#define CC1101_CR_FREQ0             0x0F     /* Frequency control word, low byte */
#define CC1101_CR_MDMCFG4           0x10     /* Modem configuration */
#define CC1101_CR_MDMCFG3           0x11     /* Modem configuration */
#define CC1101_CR_MDMCFG2           0x12     /* Modem configuration */
#define CC1101_CR_MDMCFG1           0x13     /* Modem configuration */
#define CC1101_CR_MDMCFG0           0x14     /* Modem configuration */
#define CC1101_CR_DEVIATN           0x15     /* Modem deviation setting */
#define CC1101_CR_MCSM2             0x16     /* Main Radio Cntrl State Machine config */
#define CC1101_CR_MCSM1             0x17     /* Main Radio Cntrl State Machine config */
#define CC1101_CR_MCSM0             0x18     /* Main Radio Cntrl State Machine config */
#define CC1101_CR_FOCCFG            0x19     /* Frequency Offset Compensation config */
#define CC1101_CR_BSCFG             0x1A     /* Bit Synchronization configuration */
#define CC1101_CR_AGCCTRL2          0x1B     /* AGC control */
#define CC1101_CR_AGCCTRL1          0x1C     /* AGC control */
#define CC1101_CR_AGCCTRL0          0x1D     /* AGC control */
#define CC1101_CR_WOREVT1           0x1E     /* High byte Event 0 timeout */
#define CC1101_CR_WOREVT0           0x1F     /* Low byte Event 0 timeout */
#define CC1101_CR_WORCTRL           0x20     /* Wake On Radio control */
#define CC1101_CR_FREND1            0x21     /* Front end RX configuration */
#define CC1101_CR_FREND0            0x22     /* Front end TX configuration */
#define CC1101_CR_FSCAL3            0x23     /* Frequency synthesizer calibration */
#define CC1101_CR_FSCAL2            0x24     /* Frequency synthesizer calibration */
#define CC1101_CR_FSCAL1            0x25     /* Frequency synthesizer calibration */
#define CC1101_CR_FSCAL0            0x26     /* Frequency synthesizer calibration */
#define CC1101_CR_RCCTRL1           0x27     /* RC oscillator configuration */
#define CC1101_CR_RCCTRL0           0x28     /* RC oscillator configuration */
#define CC1101_CR_FSTEST            0x29     /* Frequency synthesizer cal control */
#define CC1101_CR_PTEST             0x2A     /* Production test */
#define CC1101_CR_AGCTEST           0x2B     /* AGC test */
#define CC1101_CR_TEST2             0x2C     /* Various test settings */
#define CC1101_CR_TEST1             0x2D     /* Various test settings */
#define CC1101_CR_TEST0             0x2E     /* Various test settings */

/* Status Registers */

#define CC1101_SR_PARTNUM           0x30     /* Part number */
#define CC1101_SR_VERSION           0x31     /* Current version number */
#define CC1101_SR_FREQEST           0x32     /* Frequency offset estimate */
#define CC1101_SR_LQI               0x33     /* Demodulator estimate for link quality */
#define CC1101_SR_RSSI              0x34     /* Received signal strength indication */
#define CC1101_SR_MARCSTATE         0x35     /* Control state machine state */
#define CC1101_SR_WORTIME1          0x36     /* High byte of WOR timer */
#define CC1101_SR_WORTIME0          0x37     /* Low byte of WOR timer */
#define CC1101_SR_PKTSTATUS         0x38     /* Current GDOx status and packet status */
#define CC1101_SR_VCO_VC_DAC        0x39     /* Current setting from PLL cal module */
#define CC1101_SR_TXBYTES           0x3A     /* Underflow and # of bytes in TXFIFO */
#define CC1101_SR_RXBYTES           0x3B     /* Overflow and # of bytes in RXFIFO */
#define CC1101_SR_RCCTRL1_STATUS    0x3C     /* Last RC oscilator calibration results */
#define CC1101_SR_RCCTRL0_STATUS    0x3D     /* Last RC oscilator calibration results */

/* Single / Burst access */

#define CC1101_WRITE_BURST          0x40
#define CC1101_READ_SINGLE          0x80
#define CC1101_READ_BURST           0xC0

/* Memory locations */

#define CC1101_ML_PATABLE           0x3E
#define CC1101_ML_TXFIFO            0x3F
#define CC1101_ML_RXFIFO            0x3F

#define BROADCAST                                       0x00
#define COMMAND_DOWNLINK_BURST_MODE                     0x0C
#define BURST_FREQ_1000HZ           0x01
#define BURST_VALUES_700                                0x0E
```

```
#define COMMAND_DOWNLINK_SAMPLE_VOLTAGE                 0x03
#define COMMAND_DOWNLINK_SEND_VOLTAGE                   0x04


#endif /* CC1101_H_ */
```

```
/*
 * displayinit.c
 *
 *  Created on: 25.10.2013
 *      Author: Jakub Zimny
 */



/*********************************************************************************
*
* Own Includings
*
*********************************************************************************/


#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/timer.h"
#include "header/display.h"
#include "stdio.h"
```

```
#include "header/config.h"
#include "header/mycmdline.h"
#include "driverlib/pwm.h"

// Instance of display mode
static volatile display_handler_t handlerState;





void display_init(void)
{
      UARTprintf("Initializing Display...");
    //
    // Enable the peripherals used by this example.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);

    GPIOPinTypeGPIOOutput(LEDPORTCTRL, RS | RW | E ); // set RS | R/W | E as digital output
      GPIOPinTypeGPIOOutput(LEDPORTDATA, D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0); // set  D4 |
D5 | D6 | D7 as digital output

      GPIOPinWrite(LEDPORTCTRL, RS | RW | E, 0x00); // set RS | R/W | E to logic "0"
      GPIOPinWrite(LEDPORTDATA, D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , 0x00); // set  D4 | D5
| D6 | D7 to logic "0"


      //Function set: 8 bit Interface, 1 Zeilig
      LEDcommand(0x38);
      //Function set: 8 bit Interface, 1 Zeilig
      LEDcommand(0x38);
      //Display on/off
      LEDcommand(0x08);
      LEDcommand(0x0C);
      //Clear Display
      LEDcommand(0x01);
      //Entry Mode set: Increment, No shift
      LEDcommand(0x06);

      clear_display();

      delay(50000);

      write_to_display("*****INIT DONE******",0,0);
      write_to_display("********************",1,0);
      write_to_display("********************",2,0);
      write_to_display("********************",3,0);

      delay(50000);

      //
      // Enable Timer 1
      //
      TimerEnable(TIMER1_BASE, TIMER_A);

      UARTprintf("done\n");
}

//*****************************************************************************
//
// Clear the LED-Display
//
//*****************************************************************************
void clear_display(void){

LEDcommand(0x01);

}

//*****************************************************************************
//
// Write string to Display by row and column
//
```

```
//***************************************************************************
void write_to_display(char* inputText, unsigned char row, unsigned char col) {
      unsigned char address_d = 0;          // address of the data in the screen.

      //define address
      switch(row)
        {
                case 0: address_d = 0x80 + col;             // at zeroth row
                break;
                case 1: address_d = 0xC0 + col;             // at first row
                break;
                case 2: address_d = 0x94 + col;             // at second row
                break;
                case 3: address_d = 0xD4 + col;             // at third row
                break;
                default: address_d = 0x80 + col;     // returns to first row if invalid row
number is detected
                break;
        }

      //set position by address
      set_position(address_d);

      // Place a string, letter by letter.
      while(*inputText)
        write_char(*inputText++);
}

//***************************************************************************
//
// Set the position by address
//
//***************************************************************************
void set_position(unsigned char address) {

      GPIOPinWrite(LEDPORTCTRL, E, E);                          // Enable the Display
      GPIOPinWrite(LEDPORTDATA,  D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , address );

      delay(4500);

      GPIOPinWrite(LEDPORTCTRL, E, 0x00);                 // Disable the Display

      GPIOPinWrite(LEDPORTDATA, D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , 0x00);  // Reset pins
}

//***************************************************************************
//
// Write string to Display
//
//***************************************************************************
void write_char(unsigned char inputData) {


      GPIOPinWrite(LEDPORTCTRL, E | RS, 0xFF);                          // Enable the
Display and writeoperation
      GPIOPinWrite(LEDPORTDATA,  D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , inputData);

      delay(4500);

      GPIOPinWrite(LEDPORTCTRL, E | RS, 0x00);                  // Disable the Display


      GPIOPinWrite(LEDPORTDATA,  D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , 0x00);  // Reset pins

}

//***************************************************************************
//
// Send Command to Display
//
//***************************************************************************
void LEDcommand(unsigned char command){


      GPIOPinWrite(LEDPORTCTRL, E, E);                          // Enable the Display
      GPIOPinWrite(LEDPORTDATA,  D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 , command);
```

```
    delay(4500);

    GPIOPinWrite(LEDPORTCTRL, E, 0x00);                  // Disable the Display

    GPIOPinWrite(LEDPORTDATA, D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0, 0x00);     // Reset pins


}


//*****************************************************************************
//
// Initialize PWM for contrast setting of LED-Display
//
//*****************************************************************************
void pwm_init(void)
{

    UARTprintf("Initializing PWM...");

      unsigned long ulPeriod;
        //
        // Enable the peripherals used by this example.
        //
        SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
        GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_6);
        //
        // Set GPIO H6 as PWM pin.  It is used to output the PWM4 signal.
        //
        GPIOPinConfigure(GPIO_PH6_PWM4);
        GPIOPinTypePWM(GPIO_PORTH_BASE, GPIO_PIN_6);

        //
      // Compute the PWM period based on the system clock.
      // 16000 entspricht einer Periodendauer von 200us 16000*1/18MHz
      ulPeriod = 16000;

      //
      PWMGenConfigure(PWM0_BASE, PWM_GEN_2,
                                      PWM_GEN_MODE_UP_DOWN | PWM_GEN_MODE_NO_SYNC);
      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, ulPeriod);

      //
      // Set PWM4 to a duty cycle of 25%.
      //
      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ulPeriod / 20);

      //
      // Enable the PWM0 and PWM1 output signals.
      //
      PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT , true);

      //
      // Enable the PWM generator.
      //
      PWMGenEnable(PWM0_BASE, PWM_GEN_2);

      UARTprintf("done\n");
}
```

```
/*
 * display.h
 *
 *  Created on: 25.03.2013
 *      Author: Thomas W. und Jakub Zimny
 *
 */

#ifndef DISPLAY_H_
#define DISPLAY_H_

#define LEDPORTCTRL    GPIO_PORTH_BASE
#define LEDPORTDATA            GPIO_PORTJ_BASE

#define RS                             GPIO_PIN_0
#define RW                             GPIO_PIN_1
#define E                              GPIO_PIN_2

#define D0                             GPIO_PIN_0
#define D1                             GPIO_PIN_1
#define D2                             GPIO_PIN_2
#define D3                             GPIO_PIN_3
#define D4                             GPIO_PIN_4
#define D5                             GPIO_PIN_5
#define D6                             GPIO_PIN_6
#define D7                             GPIO_PIN_7


// Displaymodes
typedef enum
{
     START,
     CLOCK,
     MEAS,
     MEASSTOP,
     CELLA,
     CELLB,
     CELLC,
     CELLD,
     CURRENT,
```

```
    IP,
    CONF
} display_handler_t;

/*********************************************************************************
*
* Function Declarations
*
*********************************************************************************/


//*********************************************************************************
//Extern variables for Clock
//*********************************************************************************
extern unsigned long clock_msec;
extern unsigned long clock_sec;
extern unsigned long clock_min;
extern unsigned long clock_hour;
extern unsigned long clock_day;
extern unsigned long clock_month;
extern unsigned long clock_year;



void display_init(void);
void buttons_init(void);
void clear_display(void);
void write_to_display(char* inputText, unsigned char row, unsigned char col);
void set_position(unsigned char address);
void write_char(unsigned char inputData);
void LEDcommand(unsigned char command);
void pwm_init(void);

extern void delay(unsigned long ticks);


#endif /* DISPLAY_H_ */
```

```
/*
 * leds.c
 *
 *  Created on: 03.01.2014
 *      Author: Jakub Zimny
 */

#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "inc/hw_memmap.h"
```

```c
#include "driverlib/pin_map.h"
#include "inc/hw_types.h"

void led1_on_off(int on);
void led2_on_off(int on);
void led4_on_off(int on);

void led1_on_off(int on){
      if (on == 1) {
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_0);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
      }
      else{
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_0);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0x00);
      }
}

void led2_on_off(int on){
      if (on == 1) {
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_1);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
      }
      else{
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_1);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0x00);
      }
}
void led4_on_off(int on){
      if (on == 1) {
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
      }
      else{
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
        GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
      }
}
/*
 * main.c
 */
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "string.h"
#include "utils/uartstdio.h"

#include "inc/hw_ssi.h"
#include "driverlib/ssi.h"


/*********************************************************************************
*
* Own Includings
*
*********************************************************************************/
#include "utils/uartstdio.h"
#include "driverlib/rom_map.h"
#include "header/uart.h"
#include "header/mysdcard.h"
#include "header/myssi.h"
#include "header/myadc.h"
#include "header/relais.h"
```

```
#include "header/config.h"
#include "header/mycmdline.h"
#include "header/clocktimer.h"
#include "header/temperature.h"
#include "header/ethernet.h"
#include "header/control.h"
#include "header/mypwm.h"
#include "header/display.h"
#include "header/rtc.h"
#include "header/led.h"
#include "header/state.h"
#include "header/CC1101.h"


extern void delay(unsigned long ticks);
extern void UARTinit(void);
extern void mysdcardinit(void);
extern void SPIinit(void);
extern void ADCinit(void);
extern void init_clock_timer(void);
extern void display_init(void);
extern void pwm_init(void);
extern void buttons_init(void);
extern void cc1101_config(unsigned int brate);
extern void cc1101_reset(void);
extern void cc1101_power_up_reset(void);
extern void cc1101_config_no_packet_tx(void);
extern void cc1101_tx_asynchronous_mode(void);
extern void cc1101_config_no_packet ( void );
extern void wakeup_init(void);
extern void cc1101_init_tx(void);
extern void cc1101_tx(unsigned long packetes_to_tx);
extern void tx_packet(unsigned long brate, unsigned long *txbuf);
extern void cc1101_set_tx(unsigned long brate);
extern void Enable_GDO2_IRQPIN();

//Global Variables

/**********************************************
** Burst Mode
**********************************************/
unsigned long burst_freq = BURST_FREQ_1000HZ; // Burst Frequenz
unsigned long burst_values = 0;               // Anzahl der Burst Werte
unsigned long seq_number = 0;                // Z‰hlt die Seq.nummer
unsigned long tx_data_length;
unsigned long brate = 100;

//States
int state=IDLE;
int state_aktuell = MODE1;

//Variable f¸r Anz. Messungen in einer MEssung und Anz. der Messungen bzw. Files in die die
Messungen gespeichert werden.
unsigned long ulCountSample=0;
unsigned long ulCountFile=0;
float OffsetVoltage1;
float OffsetVoltage2;

volatile int IRQ_TX_FLAG = 0;
volatile int IRQ_RX_FLAG = 0;
volatile int GDO2_IRQ_MODE = 0;
volatile char packed[6];
volatile int rx_data_length = 0;




// Instanz der Konfiguration
config_t config;


// Define Command line pointer
tCmdLineEntry *g_psCmdTable;

tCmdLineEntry g_sMainCmdTable;
```

```
int
main(void)
{

    //
    // Set Clock to 80MHz PLL
    //
    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);


    //UART Init
    UARTinit();
    //SPIinit
    SPIinit();
    //ADCinit
    ADCinit();
    //DisplayTimer init
    init_clock_timer();
    //Display PWM
    pwm_init();
    //display init
    display_init();
    //Init BTNs
    buttons_init();
    //SD-Card init
    mysdcardinit();


    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
     GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_3);
     GPIOIntTypeSet(GPIO_PORTD_BASE, GPIO_PIN_3,GPIO_RISING_EDGE);
     GPIOPinIntEnable(GPIO_PORTD_BASE, GPIO_PIN_3);
     IntEnable(INT_GPIOD);

    // Interrupt Priority
     IntPrioritySet(INT_SYSCTL, 0x00); // Systick 1st Priority
     IntPrioritySet(INT_GPIOC, 0x20); //Buttons 2nd Priority
     IntPrioritySet(INT_UART1, 0x20); // RS232 2nd
    IntPrioritySet(INT_GPIOD, 0x40); //Transceiver Platine 4th
    IntPrioritySet(INT_TIMER1A, 0x60); //Statemachine 5th


    IntMasterEnable();
    state = MODE1;


    //
    // Loop forever
    //
    while(1)
    {

    }
}
```

```c
/*
 * sdCard.c
 *
 *  Created on: 14.10.2013
 *      Author: Jakub Zimny
 */

#include <string.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ssi.h"
#include <inc/lm3s9b92.h>
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include <driverlib/gpio.h>
#include "driverlib/rom.h"
#include "header/mycmdline.h"
#include "utils/uartstdio.h"
#include "utils/ustdlib.h"
#include "third_party/fatfs/src/ff.h"
#include "third_party/fatfs/src/diskio.h"


#include "header/mysdcard.h"
#include "header/myadc.h"
#include "header/temperature.h"
#include "header/clocktimer.h"
#include "header/config.h"
#include "header/control.h"
#include "header/display.h"
#include "stdio.h"
#include "utils/lwiplib.h"
#include "header/myssi.h"



//**********************************************************************************
//variables for Clock
//**********************************************************************************
unsigned long clock_msec;
unsigned long clock_sec;
unsigned long clock_min;
unsigned long clock_hour;
```

```
unsigned long clock_day;
unsigned long clock_month;
unsigned long clock_year;


//*****************************************************************************
//
// Defines the size of the buffers that hold the path, or temporary
// data from the SD card.  There are two buffers allocated of this size.
// The buffer size must be large enough to hold the longest expected
// full path name, including the file name, and a trailing null character.
//
//*****************************************************************************
#define PATH_BUF_SIZE   80

void SysTickHandler(void);

//*****************************************************************************
//
// This buffer holds the full path to the current working directory.
// Initially it is root ("/").
//
//*****************************************************************************
static char g_cCwdBuf[PATH_BUF_SIZE] = "/";

//*****************************************************************************
//
// A temporary data buffer used when manipulating file paths, or reading data
// from the SD card.
//
//*****************************************************************************
static char g_cTmpBuf[PATH_BUF_SIZE];


//*****************************************************************************
//
// The following are data structures used by FatFs.
//
//*****************************************************************************
static FATFS g_sFatFs;
static DIR g_sDirObject;
static FILINFO g_sFileInfo;
static FIL g_sFileObject;


//*****************************************************************************
//
// A structure that holds a mapping between an FRESULT numerical code,
// and a string represenation.  FRESULT codes are returned from the FatFs
// FAT file system driver.
//
//*****************************************************************************
typedef struct
{
    FRESULT fresult;
    char *pcResultStr;
}
tFresultString;

//*****************************************************************************
//
// A macro to make it easy to add result codes to the table.
//
//*****************************************************************************
#define FRESULT_ENTRY(f)        { (f), (#f) }

//*****************************************************************************
//
// A table that holds a mapping between the numerical FRESULT code and
// it's name as a string.  This is used for looking up error codes for
// printing to the console.
//
//*****************************************************************************
tFresultString g_sFresultStrings[] =
{
    FRESULT_ENTRY(FR_OK),
```

```
    FRESULT_ENTRY(FR_NOT_READY),
    FRESULT_ENTRY(FR_NO_FILE),
    FRESULT_ENTRY(FR_NO_PATH),
    FRESULT_ENTRY(FR_INVALID_NAME),
    FRESULT_ENTRY(FR_INVALID_DRIVE),
    FRESULT_ENTRY(FR_DENIED),
    FRESULT_ENTRY(FR_EXIST),
    FRESULT_ENTRY(FR_RW_ERROR),
    FRESULT_ENTRY(FR_WRITE_PROTECTED),
    FRESULT_ENTRY(FR_NOT_ENABLED),
    FRESULT_ENTRY(FR_NO_FILESYSTEM),
    FRESULT_ENTRY(FR_INVALID_OBJECT),
    FRESULT_ENTRY(FR_MKFS_ABORTED)
};

//*****************************************************************************
//
// A macro that holds the number of result codes.
//
//*****************************************************************************
#define NUM_FRESULT_CODES (sizeof(g_sFresultStrings) / sizeof(tFresultString))

//*****************************************************************************
//
// This function returns a string representation of an error code
// that was returned from a function call to FatFs.  It can be used
// for printing human readable error messages.
//
//*****************************************************************************
const char *
StringFromFresult(FRESULT fresult)
{
    unsigned int uIdx;

    //
    // Enter a loop to search the error code table for a matching
    // error code.
    //
    for(uIdx = 0; uIdx < NUM_FRESULT_CODES; uIdx++)
    {
        //
        // If a match is found, then return the string name of the
        // error code.
        //
        if(g_sFresultStrings[uIdx].fresult == fresult)
        {
            return(g_sFresultStrings[uIdx].pcResultStr);
        }
    }

    //
    // At this point no matching code was found, so return a
    // string indicating unknown error.
    //
    return("UNKNOWN ERROR CODE");
}

extern tCmdLineEntry g_sMainCmdTable;
extern tCmdLineEntry *g_psCmdTable;
//*****************************************************************************
//
// This is the table that holds the command names, implementing functions,
// and brief description.
//
//*****************************************************************************
tCmdLineEntry g_sBrowserCmdTable[] =
{
    { "help",   Cmd_help,      " : Display list of commands" },
    { "h",      Cmd_help,   "    : alias for help" },
    { "?",      Cmd_help,   "    : alias for help" },
    { "ls",     Cmd_ls,      "   : Display list of files" },
    { "chdir",  Cmd_cd,         ": Change directory" },
    { "cd",     Cmd_cd,      "   : alias for chdir" },
    { "pwd",    Cmd_pwd,      "  : Show current working directory" },
    { "cat",    Cmd_cat,      "  : Show contents of a text file (use only in Idle Mode)" },
    { "cre",    Cmd_cre,      "  : Create a file" },
```

```
    { "del",    Cmd_del,      "  : Delete a file" },
    { "exit",   Cmd_sdcard_exit,      " : Exit SD-Card browser" },
    { 0, 0, 0 }
};



// string for main location
 const char *g_cMainLocalBuf;

// Global location buffer
 char *g_cLocalBuf;

// TCP-Socket f¸r die aktuelle Steuerungs-Verbindung
int control_connection = 0;

// String buffer for print operations to UART and Ethernet
char print_buffer[80];

//define string for SD-Card location
char *g_cSdLocalBuf  = "SD-Card";

const char* TESTFILE = "testfile.txt";

//*****************************************************************************
//
// This function implements the "ls" command.  It opens the current
// directory and enumerates through the contents, and prints a line for
// each item it finds.  It shows details such as file attributes, time and
// date, and the file size, along with the name.  It shows a summary of
// file sizes at the end along with free space.
//
//*****************************************************************************
int
Cmd_ls(int argc, char *argv[])
{

    unsigned long ulTotalSize;
    unsigned long ulFileCount;
    unsigned long ulDirCount;
    FRESULT fresult;
    FATFS *pFatFs;

    //
    // Open the current directory for access.
    //
    fresult = f_opendir(&g_sDirObject, g_cCwdBuf);

    //
    // Check for error and return if there is a problem.
    //
    if(fresult != FR_OK)
    {
        return(fresult);
    }

    ulTotalSize = 0;
    ulFileCount = 0;
    ulDirCount = 0;

    //
    // Give an extra blank line before the listing.
    //
    UARTprintf("\n");

    //
    // Enter loop to enumerate through all directory entries.
    //
    for(;;)
    {
        //
        // Read an entry from the directory.
        //
        fresult = f_readdir(&g_sDirObject, &g_sFileInfo);

        //
```

```
    // Check for error and return if there is a problem.
    //
    if(fresult != FR_OK)
    {
           sprintf(print_buffer, "Error at reading File System Entry ");

           UARTprintf(print_buffer);                          // debug information
        if(control_connection){
                  ////telnet_write(print_buffer);
        }
           return(fresult);
    }

    //
    // If the file name is blank, then this is the end of the
    // listing.
    //
    if(!g_sFileInfo.fname[0])
    {
           break;
    }

    //
    // If the attribue is directory, then increment the directory count.
    //
    if(g_sFileInfo.fattrib & AM_DIR)
    {
        ulDirCount++;
    }

    //
    // Otherwise, it is a file.  Increment the file count, and
    // add in the file size to the total.
    //
    else
    {
        ulFileCount++;
        ulTotalSize += g_sFileInfo.fsize;
    }

    //
    // Print the entry information on a single line with formatting
    // to show the attributes, date, time, size, and name.
    //
    sprintf(print_buffer, "%c%c%c%c%c %u/%02u/%02u %02u:%02u %9u  %s\n",
                        (g_sFileInfo.fattrib & AM_DIR) ? 'D' : '-',
                        (g_sFileInfo.fattrib & AM_RDO) ? 'R' : '-',
                        (g_sFileInfo.fattrib & AM_HID) ? 'H' : '-',
                        (g_sFileInfo.fattrib & AM_SYS) ? 'S' : '-',
                        (g_sFileInfo.fattrib & AM_ARC) ? 'A' : '-',
                        (g_sFileInfo.fdate >> 9) + 1980,
                        (g_sFileInfo.fdate >> 5) & 15,
                         g_sFileInfo.fdate & 31,
                        (g_sFileInfo.ftime >> 11),
                        (g_sFileInfo.ftime >> 5) & 63,
                         g_sFileInfo.fsize,
                         g_sFileInfo.fname);

    UARTprintf(print_buffer);
    if(control_connection){
   ////telnet_write(print_buffer);
    }
}   // endfor

//
// Print summary lines showing the file, dir, and size totals.
//
  sprintf(print_buffer, "\n%4u File(s),%10u bytes total\n%4u Dir(s)",
        ulFileCount, ulTotalSize, ulDirCount);
  UARTprintf(print_buffer);
if(control_connection){
   ////telnet_write(print_buffer);
}
//
// Get the free space.
//
```

```
    fresult = f_getfree("/", &ulTotalSize, &pFatFs);

    //
    // Check for error and return if there is a problem.
    //
    if(fresult != FR_OK)
    {
      //UARTprintf(" Error at the end of the line ");             // debug information
        return(fresult);
    }

    //
    // Display the amount of free space that was calculated.
    //
      sprintf(print_buffer, ", %10uK bytes free\n", ulTotalSize * pFatFs->sects_clust / 2);

      UARTprintf(print_buffer);
    if(control_connection){
        ////telnet_write(print_buffer);
    }


    //
    // Made it to here, return with no errors.
    //
    return(0);
}
//*****************************************************************************
//
// This function implements the "cd" command.  It takes an argument
// that specifes the directory to make the current working directory.
// Path separators must use a forward slash "/".  The argument to cd
// can be one of the following:
// * root ("/")
// * a fully specified path ("/my/path/to/mydir")
// * a single directory name that is in the current directory ("mydir")
// * parent directory ("..")
//
// It does not understand relative paths, so dont try something like this:
// ("../my/new/path")
//
// Once the new directory is specified, it attempts to open the directory
// to make sure it exists.  If the new path is opened successfully, then
// the current working directory (cwd) is changed to the new path.
//
//*****************************************************************************
int
Cmd_cd(int argc, char *argv[])
{
    unsigned int uIdx;
    FRESULT fresult;

    //
    // Copy the current working path into a temporary buffer so
    // it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If the first character is /, then this is a fully specified
    // path, and it should just be used as-is.
    //
    if(argv[1][0] == '/')
    {
        //
        // Make sure the new path is not bigger than the cwd buffer.
        //
        if(strlen(argv[1]) + 1 > sizeof(g_cCwdBuf))
        {
              sprintf(print_buffer, "Resulting path name is too long\n");

            UARTprintf(print_buffer);
            if(control_connection){
                      ////telnet_write(print_buffer);
            }
            return(0);
```

```
        }

        //
        // If the new path name (in argv[1])  is not too long, then
        // copy it into the temporary buffer so it can be checked.
        //
        else
        {
            strncpy(g_cTmpBuf, argv[1], sizeof(g_cTmpBuf));
        }
    }

    //
    // If the argument is .. then attempt to remove the lowest level
    // on the CWD.
    //
    else if(!strcmp(argv[1], ".."))
    {
        //
        // Get the index to the last character in the current path.
        //
        uIdx = strlen(g_cTmpBuf) - 1;

        //
        // Back up from the end of the path name until a separator (/)
        // is found, or until we bump up to the start of the path.
        //
        while((g_cTmpBuf[uIdx] != '/') && (uIdx > 1))
        {
            //
            // Back up one character.
            //
            uIdx--;
        }

        //
        // Now we are either at the lowest level separator in the
        // current path, or at the beginning of the string (root).
        // So set the new end of string here, effectively removing
        // that last part of the path.
        //
        g_cTmpBuf[uIdx] = 0;
    }

    //
    // Otherwise this is just a normal path name from the current
    // directory, and it needs to be appended to the current path.
    //
    else
    {
        //
        // Test to make sure that when the new additional path is
        // added on to the current path, there is room in the buffer
        // for the full new path.  It needs to include a new separator,
        // and a trailing null character.
        //
        if(strlen(g_cTmpBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_cCwdBuf))
        {
              sprintf(print_buffer, "Resulting path name is too long\n");

            UARTprintf(print_buffer);

            if(control_connection){
                      ////telnet_write(print_buffer);
            }
            return(0);
        }

        //
        // The new path is okay, so add the separator and then append
        // the new directory to the path.
        //
        else
        {
            //
            // If not already at the root level, then append a /
```

```
            //
            if(strcmp(g_cTmpBuf, "/"))
            {
                strcat(g_cTmpBuf, "/");
            }

            //
            // Append the new directory to the path.
            //
            strcat(g_cTmpBuf, argv[1]);
        }
    }

    //
    // At this point, a candidate new directory path is in chTmpBuf.
    // Try to open it to make sure it is valid.
    //
    fresult = f_opendir(&g_sDirObject, g_cTmpBuf);

    //
    // If it cant be opened, then it is a bad path.  Inform
    // user and return.
    //
    if(fresult != FR_OK)
    {
      sprintf(print_buffer, "cd: %s\n", g_cTmpBuf);

        UARTprintf(print_buffer);

        if(control_connection){
       //telnet_write(print_buffer);
        }
        return(fresult);
    }

    //
    // Otherwise, it is a valid new path, so copy it into the CWD.
    //
    else
    {
        strncpy(g_cCwdBuf, g_cTmpBuf, sizeof(g_cCwdBuf));
    }


    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "pwd" command.  It simply prints the
// current working directory.
//
//*****************************************************************************
int
Cmd_pwd(int argc, char *argv[])
{

    //
    // Print the CWD to the console.
    //
      sprintf(print_buffer, "%s\n", g_cCwdBuf);

    UARTprintf(print_buffer);

    if(control_connection){
       //telnet_write(print_buffer);
    }


    //
    // Return success.
    //
    return(0);
```

```
}

//*****************************************************************************
//
// This function implements the "cat" command.  It reads the contents of
// a file and prints it to the console.  This should only be used on
// text files.  If it is used on a binary file, then a bunch of garbage
// is likely to printed on the console.
//
//*****************************************************************************
int
Cmd_cat(int argc, char *argv[])
{

    FRESULT fresult;
    unsigned short usBytesRead;

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      sprintf(print_buffer, "Resulting path name is too long\n");

        UARTprintf(print_buffer);

        if(control_connection){
       //telnet_write(print_buffer);
        }
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, argv[1]);

    //
    // Open the file for reading.
    //
    fresult = f_open(&g_sFileObject, g_cTmpBuf, FA_READ);

    //
    // If there was some problem opening the file, then return
    // an error.
    //
    if(fresult != FR_OK)
    {
        return(fresult);
    }


    //
    // Enter a loop to repeatedly read data from the file and display it,
    // until the end of the file is reached.
    //
    do
    {
        //
```

```
        // Read a block of data from the file.  Read as much as can fit
        // in the temporary buffer, including a space for the trailing null.
        //
        fresult = f_read(&g_sFileObject, g_cTmpBuf, sizeof(g_cTmpBuf) - 1,
                         &usBytesRead);

        //
        // If there was an error reading, then print a newline and
        // return the error to the user.
        //
        if(fresult != FR_OK)
        {
              sprintf(print_buffer, "\n");
            UARTprintf(print_buffer);
            if(control_connection){
                      //telnet_write(print_buffer);
            }
            return(fresult);
        }

        //
        // Null terminate the last block that was read to make it a
        // null terminated string that can be used with printf.
        //
        g_cTmpBuf[usBytesRead] = 0;

        //
        // Print the last chunk of the file that was received.
        //
        sprintf(print_buffer, "%s", g_cTmpBuf);

        UARTprintf(print_buffer);

        if(control_connection){
       //telnet_write(print_buffer);
        }

    //
    // Continue reading until less than the full number of bytes are
    // read.  That means the end of the buffer was reached.
    //
    }
    while(usBytesRead == sizeof(g_cTmpBuf) - 1);


    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "del" command.  It deletes a file
//
//*****************************************************************************
int
Cmd_del(int argc, char *argv[])
{

    FRESULT fresult;

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      sprintf(print_buffer, "Resulting path name is too long\n");

        UARTprintf(print_buffer);

        if(control_connection){
```

```
        //telnet_write(print_buffer);
         }
         return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, argv[1]);

    //
      // Delete File
      //
        fresult = f_unlink(g_cTmpBuf);

        //
        // Check if operation succeeded
        //

        if(fresult != FR_OK)
        {
            return(fresult);
        }



    //
    // Return success.
    //
    return(0);

}

//*****************************************************************************
//
// This function implements the "cre" command.  It creates a file
//
//*****************************************************************************
int
Cmd_cre(int argc, char *argv[])
{

    FRESULT fresult;
    FIL fnew;      /* new file object */

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(argv[1]) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      sprintf(print_buffer, "Resulting path name is too long\n");

         UARTprintf(print_buffer);

         if(control_connection){
        //telnet_write(print_buffer);
         }
         return(0);
    }
```

```
    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, argv[1]);

    //
    // Create the File
    //
    fresult = f_open(&fnew, argv[1], FA_CREATE_ALWAYS | FA_WRITE );

    //
    // Check if creation succeeded
    //

    if(fresult != FR_OK)
    {
        return(fresult);
    }

    /* Close opened files */
    f_close(&fnew);


    //
    // Return success.
    //
    return(0);

}

//*****************************************************************************
//
// This function implements the "help" command.  It prints a simple list
// of the available commands with a brief description.
//
//*****************************************************************************
int
Cmd_help(int argc, char *argv[])
{

    tCmdLineEntry *pEntry;

    //
    // Print some header text.
    //

    UARTprintf("\nAvailable commands\n");
    UARTprintf("------------------\n");

    if(control_connection){
      //telnet_write("Available commands\n");
      //telnet_write("------------------\n");
    }

    //
    // Point at the beginning of the command table.
    //
    pEntry = g_psCmdTable;

    //
    // Enter a loop to read each entry from the command table.  The
    // end of the table has been reached when the command name is NULL.
```

```
    //
    while(pEntry->pcCmd)
    {
        //
        // Print the command name and the brief description.
        //
      sprintf(print_buffer, "%s%s\n", pEntry->pcCmd, pEntry->pcHelp);

        UARTprintf(print_buffer);
        if(control_connection){
       //telnet_write(print_buffer);
        }

        //
        // Advance to the next entry in the table.
        //
        pEntry++;
    }


    //
    // Return success.
    //
    return(0);
}

int Cmd_sdcard_exit(int argc, char *argv[])
{

    // set Command line pointer to the beginning of the main command structure
    g_psCmdTable = &g_sMainCmdTable;

    // set location buffer to main
    g_cLocalBuf = (char*)g_cMainLocalBuf;

      return(0);
}

//*****************************************************************************
//
// This is the table that holds the command names, implementing functions,
// and brief description.
//
//*****************************************************************************
//extern tCmdLineEntry g_sCmdTable[];


//*****************************************************************************
//
// This function implements the "read_File" command.  It reads the contents of
// a file and prints it to the console.  This should only be used on
// text files.  If it is used on a binary file, then a bunch of garbage
// is likely to printed on the console.
//
//*****************************************************************************
int read_file(const char *filename)
{

    FRESULT fresult;
    unsigned short usBytesRead;

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
```

```
//
strcpy(g_cTmpBuf, g_cCwdBuf);

//
// If not already at the root level, then append a separator.
//
if(strcmp("/", g_cCwdBuf))
{
    strcat(g_cTmpBuf, "/");
}

//
// Now finally, append the file name to result in a fully specified file.
//
strcat(g_cTmpBuf, filename);

//
// Open the file for reading.
//
fresult = f_open(&g_sFileObject, g_cTmpBuf, FA_READ);

//
// If there was some problem opening the file, then return
// an error.
//
if(fresult != FR_OK)
{
    return(fresult);
}

//
// Enter a loop to repeatedly read data from the file and display it,
// until the end of the file is reached.
//
do
{
    //
    // Read a block of data from the file.  Read as much as can fit
    // in the temporary buffer, including a space for the trailing null.
    //
    fresult = f_read(&g_sFileObject, g_cTmpBuf, sizeof(g_cTmpBuf) - 1,
                     &usBytesRead);

    //
    // If there was an error reading, then print a newline and
    // return the error to the user.
    //
    if(fresult != FR_OK)
    {
          UARTprintf("\n");
        return(fresult);
    }

    //
    // Null terminate the last block that was read to make it a
    // null terminated string that can be used with printf.
    //
    g_cTmpBuf[usBytesRead] = 0;

    //
    // Print the last chunk of the file that was received.
    //

    UARTprintf("%s", g_cTmpBuf);


    //
    // Wait for the UART transmit buffer to empty.
    //

   #if defined(UART_BUFFERED)
          UARTFlushTx(false);
   #endif

//
// Continue reading until less than the full number of bytes are
```

```
    // read.  That means the end of the buffer was reached.
    //
    }
    while(usBytesRead == sizeof(g_cTmpBuf) - 1);

    /* Close opened files */
    f_close(&g_sFileObject);


    //
    // Return success.
    //
    return(0);
}

//*****************************************************************************
//
// This function implements the "read_config" command.  It reads the contents of
// a file and prints it to the console.  This should only be used on
// text files.  If it is used on a binary file, then a bunch of garbage
// is likely to printed on the console.
//
//*****************************************************************************
int read_into_buffer(const char *filename, char *buffer)
{

    FRESULT fresult;
    unsigned short usBytesRead;

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, filename);

    //
    // Open the file for reading.
    //
    fresult = f_open(&g_sFileObject, g_cTmpBuf, FA_READ);

    //
    // If there was some problem opening the file, then return
    // an error.
    //
    if(fresult != FR_OK)
    {
        return(fresult);
    }

    //
    // Enter a loop to repeatedly read data from the file and display it,
```

```
    // until the end of the file is reached.
    //
    char *p = buffer;
    do
    {
        //
        // Read a block of data from the file.  Read as much as can fit
        // in the temporary buffer, including a space for the trailing null.
        //
        fresult = f_read(&g_sFileObject, p, sizeof(p) - 1,
                         &usBytesRead);

        //
        // If there was an error reading, then print a newline and
        // return the error to the user.
        //
        if(fresult != FR_OK)
        {
               UARTprintf("\n");
            return(fresult);
        }

        //
        // Null terminate the last block that was read to make it a
        // null terminated string that can be used with printf.
        //
        p[usBytesRead] = 0;

        //
        // Print the last chunk of the file that was received.
        //
        //UARTprintf("%s", p);
        p += usBytesRead;

        //
        // Wait for the UART transmit buffer to empty.
        //
       #if defined(UART_BUFFERED)
               UARTFlushTx(false);
       #endif

    //
    // Continue reading until less than the full number of bytes are
    // read.  That means the end of the buffer was reached.
    //
    }
    while(usBytesRead == sizeof(p) - 1);

    /* Close opened files */
    fresult = f_close(&g_sFileObject);

    if(fresult != FR_OK)
    {
        return(fresult);
    }


    //
    // Return success.
    //
    return (0);
}



//
// This function implements the "delete_File" command.
// It simply delete a file on the SD Card, selected with its filename
//
int delete_file(const char *filename){


       FRESULT fresult;

         //
         // First, check to make sure that the current path (CWD), plus
```

```
        // the file name, plus a separator and trailing null, will all
        // fit in the temporary buffer that will be used to hold the
        // file name.  The file name must be fully specified, with path,
        // to FatFs.
        //
        if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
        {
             UARTprintf("Resulting path name is too long\n");
            return(0);
        }

        //
        // Copy the current path to the temporary buffer so it can be manipulated.
        //
        strcpy(g_cTmpBuf, g_cCwdBuf);

        //
        // If not already at the root level, then append a separator.
        //
        if(strcmp("/", g_cCwdBuf))
        {
            strcat(g_cTmpBuf, "/");
        }

        //
        // Now finally, append the file name to result in a fully specified file.
        //
        strcat(g_cTmpBuf, filename);

            //
            // Wait for the UART transmit buffer to empty.
            //
      #if defined(UART_BUFFERED)
             UARTFlushTx(false);
      #endif



    //
    // Delete File
    //
      fresult = f_unlink(g_cTmpBuf);

      //
      // Check if operation succeeded
      //

      if(fresult != FR_OK)
      {
          return(fresult);
      }


      //
      // Return success.
      //
      return(0);

}


//*****************************************************************************
//
// This function implements the "write_to_file" command.  It overrides the content of
// a file with new input.
//
//*****************************************************************************

int write_to_file(const char *filename, const char *write_Buffer){


      FIL fnew;       /* new file object */
      FRESULT fresult;
      unsigned short bw = 0;
```

```
        //
        // First, check to make sure that the current path (CWD), plus
        // the file name, plus a separator and trailing null, will all
        // fit in the temporary buffer that will be used to hold the
        // file name.  The file name must be fully specified, with path,
        // to FatFs.
        //
        if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
        {
             UARTprintf("Resulting path name is too long\n");
            return(0);
        }

        //
        // Copy the current path to the temporary buffer so it can be manipulated.
        //
        strcpy(g_cTmpBuf, g_cCwdBuf);

        //
        // If not already at the root level, then append a separator.
        //
        if(strcmp("/", g_cCwdBuf))
        {
            strcat(g_cTmpBuf, "/");
        }

        //
        // Now finally, append the file name to result in a fully specified file.
        //
        strcat(g_cTmpBuf, filename);

            //
            // Wait for the UART transmit buffer to empty.
            //
             #if defined(UART_BUFFERED)
                    UARTFlushTx(false);
             #endif


    //
    // Create File
    //

      fresult = f_open(&fnew, g_cTmpBuf, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);



      if(fresult != FR_OK)
      {
          return(fresult);
      }



      fresult = f_write(&fnew, write_Buffer, strlen(write_Buffer), &bw );

      if(fresult != FR_OK)
      {
          return(fresult);
      }


      /* Close opened files */
      f_close(&fnew);


      //
      // Return success.
      //
      return(0);

}


//*****************************************************************************
//
```

```
// This function implements the "add_to_file" command.  It appends new input to
// an existing file, selected by its filename. If file not exists, file is being created.
//
//*****************************************************************************

int add_to_file(const char *filename, const char *write_Buffer){


      FIL fnew;      /* new file object */
      FRESULT fresult;
      unsigned short bw = 0;


    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
    {
      UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, filename);

    //
    // Wait for the UART transmit buffer to empty.
    //
      #if defined(UART_BUFFERED)
      UARTFlushTx(false);
      #endif


      //
      // Open/Create File
      //

  fresult = f_open(&fnew, g_cTmpBuf, FA_OPEN_ALWAYS | FA_WRITE | FA_READ);


  if(fresult != FR_OK)
  {
      return(fresult);
  }

  // Set Pointer to end of File
  f_lseek(&fnew,fnew.fsize);

  // Write String to end of File
  fresult = f_write(&fnew, write_Buffer, strlen(write_Buffer), &bw );
  if(fresult != FR_OK)
  {
      return(fresult);
  }

  /* Close opened files */
  f_close(&fnew);
```

```
  //
  // Return success.
  //
  return(0);

}


//*****************************************************************************
//
// This function implements the "create_file" command.  It creates a file on
// the current working directory with specified filename.
//
//*****************************************************************************

int create_file(const char *filename)
{


     static FIL fnew;      /* new file object */
     FRESULT fresult;

    //
    // First, check to make sure that the current path (CWD), plus
    // the file name, plus a separator and trailing null, will all
    // fit in the temporary buffer that will be used to hold the
    // file name.  The file name must be fully specified, with path,
    // to FatFs.
    //
    if(strlen(g_cCwdBuf) + strlen(filename) + 1 + 1 > sizeof(g_cTmpBuf))
    {
     UARTprintf("Resulting path name is too long\n");
        return(0);
    }

    //
    // Copy the current path to the temporary buffer so it can be manipulated.
    //
    strcpy(g_cTmpBuf, g_cCwdBuf);

    //
    // If not already at the root level, then append a separator.
    //
    if(strcmp("/", g_cCwdBuf))
    {
        strcat(g_cTmpBuf, "/");
    }

    //
    // Now finally, append the file name to result in a fully specified file.
    //
    strcat(g_cTmpBuf, filename);

        //
        // Wait for the UART transmit buffer to empty.
        //
       #if defined(UART_BUFFERED)
               UARTFlushTx(false);
       #endif


//
// Create the File
//
  fresult = f_open(&fnew, filename, FA_CREATE_ALWAYS | FA_WRITE );

  //
  // Check if creation succeeded
  //

  if(fresult != FR_OK)
  {
      return(fresult);
  }
```

```
  /* Close opened files */
  f_close(&fnew);


      //
      // Return success.
      //
      return(0);
}




//*****************************************************************************
//
// Own variation of sdcard.c main function from stellarisware to implement an
// command line based file explorer for browsing the SD Card
//
//*****************************************************************************

int my_start_cmd_line(int argc, char *argv[])
{
    // set Command line pointer to the beginning of the SD-Card/Browser command structure
    g_psCmdTable = &g_sBrowserCmdTable[0];

    sprintf(g_cSdLocalBuf, "SD-Card: %s>", g_cCwdBuf);

    // set location buffer to Browser
    g_cLocalBuf = (char*)g_cSdLocalBuf;

      return(0);
}



//*****************************************************************************
//
// This is the handler for this SysTick interrupt.  FatFs requires a
// timer tick every 10 ms for internal timing purposes.
//
//*****************************************************************************
void
SysTickHandler(void)
{
    //
    // Call the FatFs tick timer.
    //
      TimerClock();
    disk_timerproc();
    //lwIPTimer(1);

}

void TimerClock(void)
{
        clock_msec += 10;
        if(clock_msec == 1000){
                clock_msec = 0;
                clock_sec++;
        }

          if (clock_sec == 60){
                clock_sec = 0;
                clock_min++;
          }
          if(clock_min == 60){
                clock_hour++;
                clock_min = 0;
          }
          if(clock_hour == 24){
                clock_day++;
                clock_hour = 0;
          }

          if((clock_month == 1) || (clock_month == 3) || (clock_month == 5) || (clock_month ==
```

```
7) || (clock_month == 8) || (clock_month == 10) || (clock_month == 12)){
               if(clock_day == 32) {
                      clock_month++;
                      clock_day = 1;
               }
          }
          if((clock_month == 4) || (clock_month == 6) || (clock_month == 9) || (clock_month ==
11)){
               if(clock_day == 31) {
                      clock_month++;
                      clock_day = 1;
               }
          }

          if(clock_month == 2){
               int d_temp = 29;
               if(clock_year%4 == 0)
                      d_temp = 30;
               if(clock_year%4 != 0)
                      d_temp = 29;
               if(clock_day == d_temp) {
                      clock_month++;
                      clock_day = 1;
               }
          }

          if(clock_month == 13){
               clock_year++;
               clock_month = 1;
          }
}

//*****************************************************************************
//
// Own Initialization of SSI0, SysTick, FatFs and SDcard
//
//*****************************************************************************

void mysdcardinit(void)
{
     // variable for FatFs results
    FRESULT fresult = FR_NOT_READY;

    UARTprintf("SD card initialization...\n");

    // Configure SysTick for a 100Hz interrupt.  The FatFs driver
    // wants a 10 ms tick.

    SysTickPeriodSet(SysCtlClockGet() / 100);
    SysTickEnable();
    SysTickIntEnable();

    fresult = f_mount(0, &g_sFatFs);

    if(fresult != FR_OK)
    {
        UARTprintf("f_mount error: %s\n", StringFromFresult(fresult));
    }
    // debugging stuff
    //else
    //{
    //    UARTprintf(" f_mount successful\n");
    //}

    // reset status flag
    fresult = FR_NOT_READY;


    // Open the current directory for access.
    fresult = f_opendir(&g_sDirObject, g_cCwdBuf);

    // Check for error and return if there is a problem.
    if(fresult != FR_OK)
    {
        UARTprintf(" f_opendir error: %s\n", StringFromFresult(fresult));
    }
```

```
    else
    {
        UARTprintf("done.\n");
    }

    // debugging stuff
      //else
    //{
    //    UARTprintf(" f_opendir successful\n");
    //}
}
```

```
/*
 * mysdcard.h
 *
 *  Created on: 21.01.2013
 *      Author: Thomas W. , Jakub Zimny
 */

#ifndef MYSDCARD_H_
#define MYSDCARD_H_

/*****************************************************************************
*
* Function Declarations
*
******************************************************************************/
int Cmd_cat(int argc, char *argv[]);
int Cmd_cd(int argc, char *argv[]);
int Cmd_help(int argc, char *argv[]);
int Cmd_ls(int argc, char *argv[]);
int Cmd_pwd(int argc, char *argv[]);
const char * StringFromFresult(FRESULT);
int Cmd_sdcard_exit(int argc, char *argv[]);


int add_to_file(const char *filename, const char *write_Buffer);
int delete_file(const char *filename);
int read_file(const char *filename);
int read_into_buffer(const char *filename, char *buffer);
int write_to_file(const char *filename, const char *write_Buffer);
int create_file(const char *filename);
void do_measure();

void SystickHandler(void);


void mysdcardinit(void);
int my_start_cmd_line(int argc, char *argv[]);
int Cmd_del(int argc, char *argv[]);
int Cmd_cre(int argc, char *argv[]);


#endif /* MYSDCARD_H_ */
```

```
/*
 * SPIinit.c
 *
 *  Created on: 01.10.2013
 *      Author: Jakub Zimny
 */


#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
```

```
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "header/spi.h"



//*****************************************************************************
//
// Configure SSI1 in master Freescale (SPI) mode.
//*****************************************************************************
void SPIinit(void)
{
    //
    // Display the setup on the console.
    //
    UARTprintf("SPI Init...\n");

    //
    // The SSI1 peripheral must be enabled for use.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI1);
    //Enable SSI1 Port
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    // VIA SHORT!!! This is only needed on prototype board!!
    //Enable F5 for SSI1_TX
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);


    //
    // Configure the pin muxing for SSI1 functions
    // This step is not necessary if your part does not support pin muxing.
    //
    GPIOPinConfigure(GPIO_PE0_SSI1CLK);
    GPIOPinConfigure(GPIO_PE1_SSI1FSS);
    GPIOPinConfigure(GPIO_PE2_SSI1RX);
    //GPIOPinConfigure(GPIO_PE3_SSI1TX);
    //Prototype!!!
    GPIOPinConfigure(GPIO_PF5_SSI1TX);

    //
    // Configure the GPIO settings for the SSI pins.  This function also gives
    // control of these pins to the SSI hardware.  Consult the data sheet to
    // see which functions are allocated per pin.
    //
    GPIOPinTypeSSI(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2);
    //Prototype!!!
    GPIOPinTypeSSI(GPIO_PORTF_BASE, GPIO_PIN_5);

    //
    // Configure and enable the SSI port for SPI master mode.  Use SSI1,
    // system clock supply, idle clock level low and active low clock in
    // freescale SPI mode, master mode, 1MHz SSI frequency, and 8-bit data.
    // For SPI mode, you can set the polarity of the SSI clock when the SSI
    // unit is idle.  You can also configure what clock edge you want to
    // capture data on.  Please reference the datasheet for more information on
    // the different SPI modes.
    //
    SSIConfigSetExpClk(SSI1_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER,
4000000, 8);

    //
    // Enable the SSI1 module.
    //
    SSIEnable(SSI1_BASE);
    UARTprintf("SPI Init... done\n");

}

void SPI1Send(unsigned long ulData) {
      set_spi_enable(0);
      delay(5);
      SSIDataPut(SSI1_BASE, ulData);
      wait_for_ssi1();
      delay(5);
      set_spi_enable(1);
```

```
}



void SPI1SendCC1101(unsigned long ulData1, unsigned long ulData2) {
      set_spi_enable(0);
      delay(5);
      SSIDataPut(SSI1_BASE, ulData1);
      wait_for_ssi1();
      SSIDataPut(SSI1_BASE, ulData2);
      wait_for_ssi1();
      delay(5);
      set_spi_enable(1);
}
void SPI0Send(unsigned long ulData) {
      //Senden
      SSIDataPut(SSI0_BASE, ulData);


}
```

```
/*
 * spi.h
 *
 *  Created on: 03.01.2014
 *      Author: Jakub Zimny
 */

#ifndef SPI_H_
#define SPI_H_

void SPI1SendCC1101(unsigned long ulData1, unsigned long ulData2);

extern void delay(unsigned long ticks);
extern void set_spi_enable(int lowactive);
extern void wait_for_ssi1(void);


#endif /* SPI_H_ */
```

```
//*****************************************************************************
//
// startup_ccs.c - Startup code for use with TI's Code Composer Studio.
//
// Copyright (c) 2009-2011 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
```

```
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 7243 of the EK-LM3S9B92 Firmware Package.
//
//*****************************************************************************

//*****************************************************************************
//
// Forward declaration of the default fault handlers.
//
//*****************************************************************************
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
static void IntDefaultHandler(void);
void UARTIntHandler(void);
void GPIOPORTCHANDLER(void);
void Timer1IntHandler(void);
void SysTickHandler(void);
extern void GDO2IntHandler(void);



//*****************************************************************************
//
// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****************************************************************************
extern void _c_int00(void);

//*****************************************************************************
//
// Linker variable that marks the top of the stack.
//
//*****************************************************************************
extern unsigned long __STACK_TOP;

//*****************************************************************************
//
// External declaration for the interrupt handler used by the application.
//
//*****************************************************************************
//extern void UARTIntHandler(void);

//*****************************************************************************
//
// The vector table.  Note that the proper constructs must be placed on this to
// ensure that it ends up at physical address 0x0000.0000 or at the start of
// the program if located at a start address other than 0.
//
//*****************************************************************************
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((unsigned long)&__STACK_TOP),
                                            // The initial stack pointer
    ResetISR,                               // The reset handler
    NmiSR,                                  // The NMI handler
    FaultISR,                               // The hard fault handler
    IntDefaultHandler,                      // The MPU fault handler
    IntDefaultHandler,                      // The bus fault handler
    IntDefaultHandler,                      // The usage fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // SVCall handler
```

```
    IntDefaultHandler,                      // Debug monitor handler
    0,                                      // Reserved
    IntDefaultHandler,                      // The PendSV handler
    SysTickHandler,                      // The SysTick handler
    IntDefaultHandler,                      // GPIO Port A
    IntDefaultHandler,                      // GPIO Port B
    GPIOPORTCHANDLER,                     // GPIO Port C
    GDO2IntHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                         // UART0 Rx and Tx
    UARTIntHandler,                      // UART1 Rx and Tx
    IntDefaultHandler,                      // SSI0 Rx and Tx
    IntDefaultHandler,                      // I2C0 Master and Slave
    IntDefaultHandler,                      // PWM Fault
    IntDefaultHandler,                      // PWM Generator 0
    IntDefaultHandler,                      // PWM Generator 1
    IntDefaultHandler,                      // PWM Generator 2
    IntDefaultHandler,                      // Quadrature Encoder 0
    IntDefaultHandler,                      // ADC Sequence 0
    IntDefaultHandler,                      // ADC Sequence 1
    IntDefaultHandler,                      // ADC Sequence 2
    IntDefaultHandler,                      // ADC Sequence 3
    IntDefaultHandler,                      // Watchdog timer
    IntDefaultHandler,                      // Timer 0 subtimer A
    IntDefaultHandler,                      // Timer 0 subtimer B
    Timer1IntHandler,                     // Timer 1 subtimer A
    IntDefaultHandler,                      // Timer 1 subtimer B
    IntDefaultHandler,                      // Timer 2 subtimer A
    IntDefaultHandler,                      // Timer 2 subtimer B
    IntDefaultHandler,                      // Analog Comparator 0
    IntDefaultHandler,                      // Analog Comparator 1
    IntDefaultHandler,                      // Analog Comparator 2
    IntDefaultHandler,                      // System Control (PLL, OSC, BO)
    IntDefaultHandler,                      // FLASH Control
    IntDefaultHandler,                      // GPIO Port F
    IntDefaultHandler,                      // GPIO Port G
    IntDefaultHandler,                      // GPIO Port H
    IntDefaultHandler,                      // UART2 Rx and Tx
    IntDefaultHandler,                      // SSI1 Rx and Tx
    IntDefaultHandler,                      // Timer 3 subtimer A
    IntDefaultHandler,                      // Timer 3 subtimer B
    IntDefaultHandler,                      // I2C1 Master and Slave
    IntDefaultHandler,                      // Quadrature Encoder 1
    IntDefaultHandler,                      // CAN0
    IntDefaultHandler,                      // CAN1
    IntDefaultHandler,                      // CAN2
    IntDefaultHandler,                      // Ethernet
    IntDefaultHandler,                      // Hibernate
    IntDefaultHandler,                      // USB0
    IntDefaultHandler,                      // PWM Generator 3
    IntDefaultHandler,                      // uDMA Software Transfer
    IntDefaultHandler,                      // uDMA Error
    IntDefaultHandler,                      // ADC1 Sequence 0
    IntDefaultHandler,                      // ADC1 Sequence 1
    IntDefaultHandler,                      // ADC1 Sequence 2
    IntDefaultHandler,                      // ADC1 Sequence 3
    IntDefaultHandler,                      // I2S0
    IntDefaultHandler,                      // External Bus Interface 0
    IntDefaultHandler                       // GPIO Port J
};

//*****************************************************************************
//
// This is the code that gets called when the processor first starts execution
// following a reset event.  Only the absolutely necessary set is performed,
// after which the application supplied entry() routine is called.  Any fancy
// actions (such as making decisions based on the reset cause register, and
// resetting the bits in that register) are left solely in the hands of the
// application.
//
//*****************************************************************************
void
ResetISR(void)
{
    //
    // Jump to the CCS C Initialization Routine.
```

```
    //
    __asm("    .global _c_int00\n"
          "    b.w     _c_int00");
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives a NMI.  This
// simply enters an infinite loop, preserving the system state for examination
// by a debugger.
//
//*****************************************************************************
static void
NmiSR(void)
{

    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives a fault
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
FaultISR(void)
{

    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives an unexpected
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
IntDefaultHandler(void)
{
    //
    // Go into an infinite loop.
    //
    while(1)
    {
    }
}
```

```
/*
 * statemachine.c
 *
 *  Created on: 25.10.2013
 *      Author: Jakub Zimny
 */


#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
```

```
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "string.h"
#include "utils/uartstdio.h"
#include "stdio.h"

#include "utils/uartstdio.h"
#include "driverlib/rom_map.h"
#include "header/uart.h"
#include "header/mysdcard.h"
#include "header/myssi.h"
#include "header/myadc.h"
#include "header/relais.h"
#include "header/config.h"
#include "header/mycmdline.h"
#include "header/clocktimer.h"
#include "header/temperature.h"
#include "header/ethernet.h"
#include "header/control.h"
#include "header/mypwm.h"
#include "header/display.h"
#include "header/rtc.h"
#include "header/led.h"
#include "header/state.h"
#include "driverlib/timer.h"



int add_to_file(const char *filename, const char *write_Buffer);
void uart_print_menu();
void write_measured_data_to_sd(float Voltage, float Current);
void init_clock_timer();



//*****************************************************************************
//
// StateMachine
//
//*****************************************************************************
void Timer1IntHandler(void){

    float fCurrent=0, fVoltage=0;
    char sTemp[50], sClock[50];



    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

      switch(state){
      case IDLE:
        state = state;
        break;
      case MODE1:
        uart_print_menu();
        sprintf(sTemp,"Auswahl: %d          ",state);
        write_to_display(sTemp,0,0);
        write_to_display("->Einfache Messung..",1,0);
        write_to_display("   Ausgabe via UART ",2,0);
        write_to_display("   keine Speicherung",3,0);
        state_aktuell = MODE1;
        state = IDLE;
        break;
      case MODE2:
        uart_print_menu();
        sprintf(sTemp,"Auswahl: %d          ",state);
```

```
      write_to_display(sTemp,0,0);
      write_to_display("->Zyklische Messung ",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      state_aktuell = MODE2;
      state = IDLE;
      break;
    case MODE3:
      uart_print_menu();
      sprintf(sTemp,"Auswahl: %d          ",state);
      write_to_display(sTemp,0,0);
      write_to_display("->HALL kalibrieren..",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      state_aktuell = MODE3;
      state = IDLE;
      break;
    case MODE4:
      uart_print_menu();
      sprintf(sTemp,"Auswahl: %d          ",state);
      write_to_display(sTemp,0,0);
      write_to_display("->Systemzeit........",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      state_aktuell = MODE4;
      state = IDLE;
      break;
    case MODE101:
      sample_voltage_tx();
      fCurrent = get_adc_current();
      delay(50000);
      send_voltage();
      tx_data_length = 0;
      fVoltage = sample_voltage_rx();
      sprintf(sTemp,"%.2f;",fCurrent);
      UARTprintf("%s A ", sTemp);
      sprintf(sTemp,"%.3f;",fVoltage);
      UARTprintf("%s V\n", sTemp);
      write_to_display("Sampling Voltage    ",0,0);
      write_to_display("                    ",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      state = MODE1;
      break;
    case MODE102:
      //Messungen Z‰hlen
      ulCountSample = ulCountSample + 1;

      //Sample Voltage
      sample_voltage_tx();
      //Get Current
      fCurrent = get_adc_current();
      delay(50000);
      send_voltage();
      tx_data_length = 0;
      fVoltage = sample_voltage_rx();
      write_measured_data_to_sd(fVoltage, fCurrent);
      sprintf(sTemp,"EMPFANGEN           ");
      write_to_display(sTemp,0,0);
      write_to_display("                    ",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      //ca 900ms
      //delay(6000000);
      break;
    case MODE103:
      get_hall_offset();
      UARTprintf("HALL CALIBRATED\n");
      write_to_display("HALL CALIBRATED     ",0,0);
      write_to_display("                    ",1,0);
      write_to_display("                    ",2,0);
      write_to_display("                    ",3,0);
      delay(10000000);
      state = MODE1;
      break;
    case MODE104:
```

```
        sprintf(sClock,"%2dd:%2dh:%2dmin:%2ds   ", clock_day,clock_hour,clock_min,clock_sec);
        write_to_display(sClock,0,0);
        write_to_display("                    ",1,0);
        write_to_display("                    ",2,0);
        write_to_display("                    ",3,0);
        //state_aktuell = MODE104;
        break;
      default:
        state = MODE1;
        break;
      }




}



void uart_print_menu(){
      UARTprintf("Menu:...\n");
      UARTprintf("Send 1 for Mode1, Einfache Messung\n");
      UARTprintf("Send 2 for Mode2, Zyklische Messung\n");
      UARTprintf("Send 3 for Mode3, HALL kalibrieren\n");
      UARTprintf("Send 4 for Mode4, Systemzeit anzeigen\n");

}

void write_measured_data_to_sd(float Voltage, float Current){
      char sTemp[50], sd_card[100], file[20];
      float Temperature_Dummy = 0.0;
      unsigned long Sensor_Dummy = 1;

      sprintf(sTemp,"%10d;",ulCountSample);
      strcpy(sd_card, sTemp);
      sprintf(sTemp,"%d;",Sensor_Dummy);
      strcat(sd_card,sTemp);
      sprintf(sTemp,"%.3f;",Voltage);
      strcat(sd_card,sTemp);
      sprintf(sTemp,"%.2f;",Current);
      strcat(sd_card,sTemp);
      sprintf(sTemp,"%.1f;",Temperature_Dummy);
      strcat(sd_card,sTemp);
      sprintf(sTemp,"%2dd:%2dh:%2dmin:%2ds:%2dms\n", clock_day,clock_hour,clock_min,clock_sec,
clock_msec);
      strcat(sd_card,sTemp);

      UARTprintf("%s\n",sd_card);


      sprintf(sTemp,"%d",ulCountFile);
      strcpy(file, "Messung");
      strcat(file, sTemp);
      strcat(file, ".txt");
      add_to_file(file, sd_card);
}

void init_clock_timer(){

      UARTprintf("Initializing Timer...");

         SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

         TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
         //100ms
         TimerLoadSet(TIMER1_BASE, TIMER_A, 8000000);

         IntEnable(INT_TIMER1A);
         TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

    UARTprintf("done\n");
}
```

```
/*
 * state.h
 *
 *  Created on: 25.10.2013
 *      Author: Jakub Zimny
 */

#ifndef STATE_H_
#define STATE_H_

//*******************************************************************************
//variables for Clock
//*******************************************************************************
extern unsigned long clock_msec;
extern unsigned long clock_sec;
extern unsigned long clock_min;
extern unsigned long clock_hour;
extern unsigned long clock_day;
extern unsigned long clock_month;
extern unsigned long clock_year;

extern unsigned long tx_data_length;
extern int state;
extern int state_aktuell;
extern float OffsetVoltage1;
extern float OffsetVoltage2;
extern unsigned long ulCountSample;
extern unsigned long ulCountFile;


extern void get_hall_offset();
extern void cc1101_idle();
extern float get_adc_current();
extern void wakeup(void);
extern void sample_voltage_tx(void);
extern void send_voltage(void);
extern void delay(unsigned long ticks);
extern float sample_voltage_rx();
extern void led1_on_off(int on);
extern void led2_on_off(int on);
extern void led4_on_off(int on);


#define IDLE 0
#define MODE1 1
#define MODE2 2
#define MODE3 3
#define MODE4 4

#define MODE101 101
#define MODE102 102
#define MODE103 103
#define MODE104 104



#endif /* STATE_H_ */
```

```c
/*
 * UARTinit.c
 *
 *  Created on: 30.09.2013
 *      Author: Jakub Zimny
 */
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include <stdio.h>
#include <math.h>
#include "string.h"
#include "header/state.h"
#include "header/uart.h"


//*****************************************************************************
//
// Send a string to the UART.
//
//*****************************************************************************
void
UARTSend(const unsigned char *pucBuffer, unsigned long ulCount)
{
    //
    // Loop while there are more characters to send.
    //
    while(ulCount--)
    {
        //
        // Write the next character to the UART.
        //
        UARTCharPutNonBlocking(UART1_BASE, *pucBuffer++);
    }
}



void UARTIntHandler(void)
{
    unsigned long ulStatus;
    unsigned char c;

    //
    // Get the interrrupt status.
    //
    ulStatus = UARTIntStatus(UART1_BASE, true);


    //
    // Clear the asserted interrupts.
    //
    UARTIntClear(UART1_BASE, ulStatus);



    //
    // Loop while there are characters in the receive FIFO.
    //
    while(UARTCharsAvail(UART1_BASE))
    {
        //
        // Read the next character from the UART and write it back to the UART.
        //
        //UARTCharPutNonBlocking(UART1_BASE,
        //                          UARTCharGetNonBlocking(UART1_BASE));
       c = UARTCharGetNonBlocking(UART1_BASE);
       // Char = "1"
```

```
      if ((char) c == 0x31)
      {
        state = MODE101;

      }
      //Char "0"
      else if ((char) c == 0x30){
        state = MODE1;
      }
      // Char "2"
      else if ((char) c == 0x32){
                      state = MODE102;
              }
      // Char "3"
      else if ((char) c == 0x33){
                      state = MODE103;
              }
      // Char "4"
        else if ((char) c == 0x34){
                              state = MODE104;
              }
      else
      {
        UARTprintf("Error!   \n");
      }


    }
}




void UARTinit(void) {

    //
    // Enable the peripherals used by UART
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    //
      // Set GPIO D0 and D1 as UART pins.
      //
      GPIOPinConfigure(GPIO_PD0_U1RX);
      GPIOPinConfigure(GPIO_PD1_U1TX);
      GPIOPinTypeUART(GPIO_PORTD_BASE, GPIO_PIN_0 | GPIO_PIN_1);

      //
      // Configure the UART for 115,200, 8-N-1 operation.
      //
      //UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 115200, (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioInit(1);
      //
      // Enable the UART interrupt.
      //
      IntEnable(INT_UART1);
      UARTIntEnable(UART1_BASE, UART_INT_RX | UART_INT_RT);




      UARTprintf("\nUART init done ...\n");
      UARTprintf("Send 0 for menu...\n");

}


/*
 * uart.h
```

```
 *
 *  Created on: 03.01.2014
 *      Author: Jakub Zimny
 */

#ifndef UART_H_
#define UART_H_

/*********************************************************************************
*
* Function Declarations
*
**********************************************************************************/

extern unsigned short adc_get_single_value_ch2(void);
extern int add_to_file(const char *filename, const char *write_Buffer);
extern int create_file(const char *filename);
extern int delete_file(const char *filename);
extern int read_file(const char *filename);
extern int write_to_file(const char *filename, const char *write_Buffer);
extern void write_to_display(char* inputText, unsigned char row, unsigned char col);
extern int state;


void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount);
void UARTIntHandler(void);
void UARTinit(void);



#endif /* UART_H_ */
```

10.5 Messdaten des Versuchsbetriebes

1;1;4.706;-0.01;0.0; 0d: 0h: 0min:12s:880ms
2;1;4.706;-0.02;0.0; 0d: 0h: 0min:13s:40ms
3;1;2.032;0.01;0.0; 0d: 0h: 0min:13s:260ms
4;1;2.928;-0.01;0.0; 0d: 0h: 0min:13s:420ms
5;1;4.524;-0.01;0.0; 0d: 0h: 0min:13s:570ms
6;1;4.519;-0.00;0.0; 0d: 0h: 0min:13s:730ms
7;1;3.446;-0.01;0.0; 0d: 0h: 0min:13s:890ms
8;1;2.268;-0.03;0.0; 0d: 0h: 0min:14s:70ms
9;1;2.041;-0.02;0.0; 0d: 0h: 0min:14s:230ms
10;1;3.169;0.01;0.0; 0d: 0h: 0min:14s:380ms
11;1;4.388;0.01;0.0; 0d: 0h: 0min:14s:540ms
12;1;4.617;0.02;0.0; 0d: 0h: 0min:14s:760ms
13;1;3.716;-0.05;0.0; 0d: 0h: 0min:14s:920ms
14;1;2.140;0.02;0.0; 0d: 0h: 0min:15s:80ms
15;1;2.170;-0.02;0.0; 0d: 0h: 0min:15s:230ms
16;1;3.257;-0.00;0.0; 0d: 0h: 0min:15s:390ms
17;1;4.427;-0.03;0.0; 0d: 0h: 0min:15s:550ms
18;1;4.600;0.02;0.0; 0d: 0h: 0min:15s:710ms
19;1;3.655;-0.01;0.0; 0d: 0h: 0min:15s:860ms
20;1;2.410;0.02;0.0; 0d: 0h: 0min:16s:20ms
21;1;2.006;0.01;0.0; 0d: 0h: 0min:16s:240ms
22;1;2.800;-0.01;0.0; 0d: 0h: 0min:16s:400ms
23;1;4.459;-0.01;0.0; 0d: 0h: 0min:16s:560ms
24;1;4.574;0.00;0.0; 0d: 0h: 0min:16s:720ms
25;1;3.575;-0.00;0.0; 0d: 0h: 0min:16s:870ms
26;1;2.365;0.01;0.0; 0d: 0h: 0min:17s:30ms
27;1;2.019;0.00;0.0; 0d: 0h: 0min:17s:190ms
28;1;2.861;0.07;0.0; 0d: 0h: 0min:17s:350ms
29;1;4.144;0.08;0.0; 0d: 0h: 0min:17s:500ms
30;1;4.672;0.16;0.0; 0d: 0h: 0min:17s:720ms
31;1;4.039;0.22;0.0; 0d: 0h: 0min:17s:880ms
32;1;2.325;0.25;0.0; 0d: 0h: 0min:18s:40ms
33;1;2.041;0.30;0.0; 0d: 0h: 0min:18s:200ms
34;1;2.933;0.36;0.0; 0d: 0h: 0min:18s:350ms
35;1;4.204;0.42;0.0; 0d: 0h: 0min:18s:510ms
36;1;4.668;0.54;0.0; 0d: 0h: 0min:18s:670ms
37;1;3.986;0.66;0.0; 0d: 0h: 0min:18s:820ms
38;1;2.695;0.68;0.0; 0d: 0h: 0min:18s:980ms
39;1;1.973;0.74;0.0; 0d: 0h: 0min:19s:200ms
40;1;2.509;0.83;0.0; 0d: 0h: 0min:19s:360ms
41;1;4.240;0.92;0.0; 0d: 0h: 0min:19s:520ms
42;1;4.656;1.05;0.0; 0d: 0h: 0min:19s:700ms
43;1;3.906;1.16;0.0; 0d: 0h: 0min:19s:860ms
44;1;2.485;1.09;0.0; 0d: 0h: 0min:20s:10ms
45;1;1.998;1.13;0.0; 0d: 0h: 0min:20s:170ms
46;1;2.731;1.09;0.0; 0d: 0h: 0min:20s:330ms
47;1;4.025;1.13;0.0; 0d: 0h: 0min:20s:490ms
48;1;4.672;1.13;0.0; 0d: 0h: 0min:20s:700ms
49;1;4.158;1.13;0.0; 0d: 0h: 0min:20s:860ms
50;1;2.477;1.13;0.0; 0d: 0h: 0min:21s:20ms
51;1;2.007;1.13;0.0; 0d: 0h: 0min:21s:180ms
52;1;2.797;1.13;0.0; 0d: 0h: 0min:21s:330ms
53;1;4.086;1.14;0.0; 0d: 0h: 0min:21s:490ms
54;1;4.673;1.14;0.0; 0d: 0h: 0min:21s:650ms
55;1;4.114;1.14;0.0; 0d: 0h: 0min:21s:810ms
56;1;2.832;1.15;0.0; 0d: 0h: 0min:21s:960ms

57;1;2.012;1.12;0.0; 0d: 0h: 0min:22s:190ms
58;1;2.374;1.14;0.0; 0d: 0h: 0min:22s:340ms
59;1;4.132;1.12;0.0; 0d: 0h: 0min:22s:500ms
60;1;4.670;1.13;0.0; 0d: 0h: 0min:22s:660ms
61;1;4.043;1.12;0.0; 0d: 0h: 0min:22s:820ms
62;1;2.755;1.11;0.0; 0d: 0h: 0min:22s:970ms
63;1;2.000;1.13;0.0; 0d: 0h: 0min:23s:130ms
64;1;2.450;1.14;0.0; 0d: 0h: 0min:23s:290ms
65;1;3.689;1.13;0.0; 0d: 0h: 0min:23s:440ms
66;1;4.612;1.13;0.0; 0d: 0h: 0min:23s:660ms
67;1;4.406;1.15;0.0; 0d: 0h: 0min:23s:830ms
68;1;2.710;1.12;0.0; 0d: 0h: 0min:23s:980ms
69;1;1.991;1.24;0.0; 0d: 0h: 0min:24s:140ms
70;1;2.510;1.32;0.0; 0d: 0h: 0min:24s:300ms
71;1;3.761;1.37;0.0; 0d: 0h: 0min:24s:450ms
72;1;4.633;1.50;0.0; 0d: 0h: 0min:24s:610ms
73;1;4.355;1.63;0.0; 0d: 0h: 0min:24s:770ms
74;1;3.145;1.79;0.0; 0d: 0h: 0min:24s:920ms
75;1;2.117;1.95;0.0; 0d: 0h: 0min:25s:140ms
76;1;2.172;2.13;0.0; 0d: 0h: 0min:25s:320ms
77;1;3.796;2.28;0.0; 0d: 0h: 0min:25s:490ms
78;1;4.667;2.37;0.0; 0d: 0h: 0min:25s:640ms
79;1;4.159;2.36;0.0; 0d: 0h: 0min:25s:800ms
80;1;2.883;2.38;0.0; 0d: 0h: 0min:25s:960ms
81;1;2.025;2.40;0.0; 0d: 0h: 0min:26s:110ms
82;1;2.349;2.35;0.0; 0d: 0h: 0min:26s:270ms
83;1;3.555;2.36;0.0; 0d: 0h: 0min:26s:430ms
84;1;4.565;2.38;0.0; 0d: 0h: 0min:26s:650ms
85;1;4.480;2.39;0.0; 0d: 0h: 0min:26s:810ms
86;1;2.848;2.37;0.0; 0d: 0h: 0min:26s:960ms
87;1;2.012;2.39;0.0; 0d: 0h: 0min:27s:120ms
88;1;2.385;2.38;0.0; 0d: 0h: 0min:27s:280ms
89;1;3.625;2.35;0.0; 0d: 0h: 0min:27s:440ms
90;1;4.591;2.37;0.0; 0d: 0h: 0min:27s:590ms
91;1;4.445;2.37;0.0; 0d: 0h: 0min:27s:750ms
92;1;3.300;2.36;0.0; 0d: 0h: 0min:27s:910ms
93;1;2.186;2.37;0.0; 0d: 0h: 0min:28s:130ms
94;1;2.112;2.37;0.0; 0d: 0h: 0min:28s:290ms
95;1;3.654;2.39;0.0; 0d: 0h: 0min:28s:440ms
96;1;4.608;2.31;0.0; 0d: 0h: 0min:28s:600ms
97;1;4.412;2.30;0.0; 0d: 0h: 0min:28s:760ms
98;1;3.231;2.22;0.0; 0d: 0h: 0min:28s:920ms
99;1;2.152;2.18;0.0; 0d: 0h: 0min:29s:70ms
100;1;2.145;2.09;0.0; 0d: 0h: 0min:29s:230ms
101;1;3.209;2.05;0.0; 0d: 0h: 0min:29s:390ms
102;1;4.398;1.89;0.0; 0d: 0h: 0min:29s:610ms
103;1;4.613;1.85;0.0; 0d: 0h: 0min:29s:770ms
104;1;3.180;1.72;0.0; 0d: 0h: 0min:29s:920ms
105;1;2.123;1.60;0.0; 0d: 0h: 0min:30s:80ms
106;1;2.167;1.42;0.0; 0d: 0h: 0min:30s:240ms
107;1;3.270;1.26;0.0; 0d: 0h: 0min:30s:400ms
108;1;4.436;1.13;0.0; 0d: 0h: 0min:30s:550ms
109;1;4.592;1.14;0.0; 0d: 0h: 0min:30s:710ms
110;1;3.633;1.15;0.0; 0d: 0h: 0min:30s:890ms
111;1;2.404;1.13;0.0; 0d: 0h: 0min:31s:110ms
112;1;2.048;1.11;0.0; 0d: 0h: 0min:31s:270ms
113;1;3.577;1.13;0.0; 0d: 0h: 0min:31s:420ms
114;1;4.559;1.13;0.0; 0d: 0h: 0min:31s:580ms
115;1;4.492;1.13;0.0; 0d: 0h: 0min:31s:740ms
116;1;3.392;1.14;0.0; 0d: 0h: 0min:31s:900ms

117;1;2.241;1.12;0.0; 0d: 0h: 0min:32s:50ms
118;1;2.080;1.11;0.0; 0d: 0h: 0min:32s:210ms
119;1;3.057;1.13;0.0; 0d: 0h: 0min:32s:370ms
120;1;4.297;1.11;0.0; 0d: 0h: 0min:32s:590ms
121;1;4.650;1.10;0.0; 0d: 0h: 0min:32s:750ms
122;1;3.341;1.09;0.0; 0d: 0h: 0min:32s:900ms
123;1;2.197;0.97;0.0; 0d: 0h: 0min:33s:60ms
124;1;2.102;0.81;0.0; 0d: 0h: 0min:33s:220ms
125;1;3.112;0.68;0.0; 0d: 0h: 0min:33s:380ms
126;1;4.337;0.54;0.0; 0d: 0h: 0min:33s:530ms
127;1;4.640;0.40;0.0; 0d: 0h: 0min:33s:690ms
128;1;3.776;0.21;0.0; 0d: 0h: 0min:33s:850ms
129;1;2.524;0.00;0.0; 0d: 0h: 0min:34s:70ms
130;1;1.989;0.00;0.0; 0d: 0h: 0min:34s:230ms
131;1;3.154;0.01;0.0; 0d: 0h: 0min:34s:380ms
132;1;4.379;0.00;0.0; 0d: 0h: 0min:34s:540ms
133;1;4.623;-0.00;0.0; 0d: 0h: 0min:34s:700ms
134;1;3.734;-0.02;0.0; 0d: 0h: 0min:34s:850ms
135;1;2.490;-0.03;0.0; 0d: 0h: 0min:35s:10ms
136;1;1.995;-0.00;0.0; 0d: 0h: 0min:35s:170ms
137;1;2.694;-0.01;0.0; 0d: 0h: 0min:35s:330ms
138;1;4.015;0.01;0.0; 0d: 0h: 0min:35s:550ms
139;1;4.670;0.03;0.0; 0d: 0h: 0min:35s:710ms
140;1;3.684;-0.02;0.0; 0d: 0h: 0min:35s:860ms
141;1;2.428;-0.01;0.0; 0d: 0h: 0min:36s:20ms
142;1;2.002;0.00;0.0; 0d: 0h: 0min:36s:180ms
143;1;2.778;0.02;0.0; 0d: 0h: 0min:36s:330ms
144;1;4.072;-0.02;0.0; 0d: 0h: 0min:36s:510ms
145;1;4.672;0.01;0.0; 0d: 0h: 0min:36s:670ms
146;1;3.988;0.01;0.0; 0d: 0h: 0min:36s:830ms
147;1;2.675;-0.01;0.0; 0d: 0h: 0min:37s:50ms
148;1;1.989;-0.00;0.0; 0d: 0h: 0min:37s:210ms
149;1;3.005;-0.01;0.0; 0d: 0h: 0min:37s:360ms
150;1;4.276;-0.02;0.0; 0d: 0h: 0min:37s:520ms
151;1;4.655;0.02;0.0; 0d: 0h: 0min:37s:680ms
152;1;3.870;-0.01;0.0; 0d: 0h: 0min:37s:840ms
153;1;2.638;0.01;0.0; 0d: 0h: 0min:37s:990ms
154;1;2.010;-0.03;0.0; 0d: 0h: 0min:38s:150ms
155;1;2.578;0.02;0.0; 0d: 0h: 0min:38s:310ms
156;1;3.832;-0.03;0.0; 0d: 0h: 0min:38s:530ms
157;1;4.650;-0.01;0.0; 0d: 0h: 0min:38s:690ms
158;1;3.816;0.00;0.0; 0d: 0h: 0min:38s:850ms
159;1;2.542;-0.00;0.0; 0d: 0h: 0min:39s: 0ms
160;1;1.986;-0.03;0.0; 0d: 0h: 0min:39s:160ms
161;1;2.635;0.01;0.0; 0d: 0h: 0min:39s:320ms
162;1;3.942;-0.00;0.0; 0d: 0h: 0min:39s:470ms
163;1;4.662;-0.03;0.0; 0d: 0h: 0min:39s:630ms
164;1;4.244;-0.03;0.0; 0d: 0h: 0min:39s:790ms
165;1;2.987;0.02;0.0; 0d: 0h: 0min:40s:10ms
166;1;2.056;-0.00;0.0; 0d: 0h: 0min:40s:170ms
167;1;2.687;0.00;0.0; 0d: 0h: 0min:40s:330ms
168;1;4.005;-0.02;0.0; 0d: 0h: 0min:40s:480ms
169;1;4.669;0.00;0.0; 0d: 0h: 0min:40s:640ms
170;1;4.187;-0.01;0.0; 0d: 0h: 0min:40s:800ms
171;1;2.919;-0.00;0.0; 0d: 0h: 0min:40s:950ms
172;1;2.028;-0.02;0.0; 0d: 0h: 0min:41s:110ms
173;1;2.322;-0.01;0.0; 0d: 0h: 0min:41s:270ms
174;1;3.521;0.00;0.0; 0d: 0h: 0min:41s:490ms
175;1;4.551;0.00;0.0; 0d: 0h: 0min:41s:660ms
176;1;4.152;-0.04;0.0; 0d: 0h: 0min:41s:840ms

177;1;2.783;-0.04;0.0; 0d: 0h: 0min:42s:30ms
178;1;1.975;-0.02;0.0; 0d: 0h: 0min:42s:210ms
179;1;2.828;-0.00;0.0; 0d: 0h: 0min:42s:370ms
180;1;4.288;-0.02;0.0; 0d: 0h: 0min:42s:530ms
181;1;4.647;0.01;0.0; 0d: 0h: 0min:42s:680ms
182;1;3.828;0.01;0.0; 0d: 0h: 0min:42s:840ms
183;1;2.571;-0.00;0.0; 0d: 0h: 0min:43s:60ms
184;1;1.986;0.01;0.0; 0d: 0h: 0min:43s:220ms
185;1;3.113;-0.00;0.0; 0d: 0h: 0min:43s:380ms
186;1;4.351;0.01;0.0; 0d: 0h: 0min:43s:540ms
187;1;4.631;0.00;0.0; 0d: 0h: 0min:43s:690ms
188;1;3.757;0.04;0.0; 0d: 0h: 0min:43s:850ms
189;1;2.500;-0.01;0.0; 0d: 0h: 0min:44s:10ms
190;1;1.989;0.01;0.0; 0d: 0h: 0min:44s:160ms
191;1;2.682;-0.01;0.0; 0d: 0h: 0min:44s:320ms
192;1;3.975;-0.01;0.0; 0d: 0h: 0min:44s:540ms
193;1;4.667;-0.00;0.0; 0d: 0h: 0min:44s:700ms
194;1;3.723;0.00;0.0; 0d: 0h: 0min:44s:860ms
195;1;2.461;0.01;0.0; 0d: 0h: 0min:45s:20ms
196;1;1.997;-0.01;0.0; 0d: 0h: 0min:45s:170ms
197;1;2.754;-0.02;0.0; 0d: 0h: 0min:45s:330ms
198;1;4.047;0.00;0.0; 0d: 0h: 0min:45s:490ms
199;1;4.673;-0.00;0.0; 0d: 0h: 0min:45s:640ms
200;1;4.150;0.00;0.0; 0d: 0h: 0min:45s:800ms
201;1;2.874;0.03;0.0; 0d: 0h: 0min:46s:20ms
202;1;2.021;-0.04;0.0; 0d: 0h: 0min:46s:180ms
203;1;2.788;-0.03;0.0; 0d: 0h: 0min:46s:340ms
204;1;4.098;0.00;0.0; 0d: 0h: 0min:46s:500ms
205;1;4.674;0.01;0.0; 0d: 0h: 0min:46s:650ms
206;1;4.087;-0.03;0.0; 0d: 0h: 0min:46s:810ms
207;1;2.802;-0.03;0.0; 0d: 0h: 0min:46s:970ms
208;1;2.013;0.00;0.0; 0d: 0h: 0min:47s:120ms
209;1;2.407;0.01;0.0; 0d: 0h: 0min:47s:280ms
210;1;3.636;-0.03;0.0; 0d: 0h: 0min:47s:500ms
211;1;4.594;-0.01;0.0; 0d: 0h: 0min:47s:660ms
212;1;4.050;-0.02;0.0; 0d: 0h: 0min:47s:840ms
213;1;2.745;-0.01;0.0; 0d: 0h: 0min:48s: 0ms
214;1;1.987;-0.00;0.0; 0d: 0h: 0min:48s:160ms
215;1;2.610;0.00;0.0; 0d: 0h: 0min:48s:310ms
216;1;3.907;0.01;0.0; 0d: 0h: 0min:48s:470ms
217;1;4.658;-0.01;0.0; 0d: 0h: 0min:48s:630ms
218;1;4.260;0.03;0.0; 0d: 0h: 0min:48s:780ms
219;1;2.998;-0.02;0.0; 0d: 0h: 0min:49s: 0ms
220;1;2.063;0.01;0.0; 0d: 0h: 0min:49s:160ms
221;1;2.656;-0.01;0.0; 0d: 0h: 0min:49s:320ms
222;1;3.975;0.01;0.0; 0d: 0h: 0min:49s:480ms
223;1;4.667;-0.02;0.0; 0d: 0h: 0min:49s:640ms
224;1;4.213;-0.05;0.0; 0d: 0h: 0min:49s:790ms
225;1;2.949;0.01;0.0; 0d: 0h: 0min:49s:950ms
226;1;2.041;0.01;0.0; 0d: 0h: 0min:50s:110ms
227;1;2.310;-0.01;0.0; 0d: 0h: 0min:50s:270ms
228;1;3.511;0.01;0.0; 0d: 0h: 0min:50s:490ms
229;1;4.546;-0.00;0.0; 0d: 0h: 0min:50s:650ms
230;1;4.161;-0.02;0.0; 0d: 0h: 0min:50s:800ms
231;1;2.866;-0.01;0.0; 0d: 0h: 0min:50s:960ms
232;1;2.020;-0.01;0.0; 0d: 0h: 0min:51s:120ms
233;1;2.360;-0.02;0.0; 0d: 0h: 0min:51s:270ms
234;1;3.574;-0.01;0.0; 0d: 0h: 0min:51s:430ms
235;1;4.575;0.03;0.0; 0d: 0h: 0min:51s:590ms
236;1;4.462;-0.02;0.0; 0d: 0h: 0min:51s:750ms

237;1;3.330;-0.03;0.0; 0d: 0h: 0min:51s:970ms
238;1;2.205;0.01;0.0; 0d: 0h: 0min:52s:130ms
239;1;2.396;0.00;0.0; 0d: 0h: 0min:52s:280ms
240;1;3.649;0.03;0.0; 0d: 0h: 0min:52s:440ms
241;1;4.598;-0.01;0.0; 0d: 0h: 0min:52s:600ms
242;1;4.429;0.02;0.0; 0d: 0h: 0min:52s:750ms
243;1;3.265;0.02;0.0; 0d: 0h: 0min:52s:910ms
244;1;2.175;-0.00;0.0; 0d: 0h: 0min:53s:70ms
245;1;2.128;0.00;0.0; 0d: 0h: 0min:53s:230ms
246;1;3.176;-0.02;0.0; 0d: 0h: 0min:53s:450ms
247;1;4.379;0.00;0.0; 0d: 0h: 0min:53s:630ms
248;1;4.396;-0.02;0.0; 0d: 0h: 0min:53s:790ms
249;1;3.014;0.01;0.0; 0d: 0h: 0min:53s:940ms
250;1;2.062;0.00;0.0; 0d: 0h: 0min:54s:100ms
251;1;2.260;0.00;0.0; 0d: 0h: 0min:54s:260ms
252;1;3.431;0.01;0.0; 0d: 0h: 0min:54s:410ms
253;1;4.514;-0.01;0.0; 0d: 0h: 0min:54s:570ms
254;1;4.542;-0.01;0.0; 0d: 0h: 0min:54s:730ms
255;1;3.477;-0.01;0.0; 0d: 0h: 0min:54s:950ms
256;1;2.295;-0.02;0.0; 0d: 0h: 0min:55s:110ms
257;1;2.296;-0.02;0.0; 0d: 0h: 0min:55s:270ms
258;1;3.503;-0.02;0.0; 0d: 0h: 0min:55s:420ms
259;1;4.543;-0.00;0.0; 0d: 0h: 0min:55s:580ms
260;1;4.507;0.00;0.0; 0d: 0h: 0min:55s:740ms
261;1;3.423;0.02;0.0; 0d: 0h: 0min:55s:890ms
262;1;2.260;-0.00;0.0; 0d: 0h: 0min:56s:50ms
263;1;2.062;0.00;0.0; 0d: 0h: 0min:56s:210ms
264;1;3.011;0.00;0.0; 0d: 0h: 0min:56s:430ms

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) 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, 24. Januar 2014

Ort, Datum

Unterschrift