Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterthesis

Nico Sassano

Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen

Fachhochschule Westküste
Fachbereich Technik

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Nico Sassano

Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen

Masterthesis eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
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.-Ing. Alfred Ebberg

Abgegeben am 14. August 2015

Nico Sassano

Thema der Masterthesis

Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen

Stichworte

Batteriezellsensor, Hardwareentwicklung, funksynchronisierte Messung, verteilte Signalverarbeitung, elektrochemischen Impedanzspektroskopie, Goertzel-Algorithmus, drahtlose Kommunikation

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen. Dabei wird eine $LiFePO_4$-Batteriezelle mit einem Wechselstrom angeregt. Der fließende Strom wird dabei durch das Messsystem erfasst. Die durch den Strom verursachte Wechselspannung wird von einem Zellsensor gemessen. Die synchrone Messung von Strom und Spannung wird dabei über ein spezielles proprietäres Funkprotokoll realisiert. Durch die Phasenverschiebung von Stom und Spannung bei unterschiedlichen Frequenzen, lässt sich das Impedanzspektrum der Batteriezelle berechnen.

Nico Sassano

Title of the paper

Development of a measuring system for radio-synchronised electrochemical impedance spectroscopy on battery cells

Keywords

Battery Cell Sensor, hardware development, radio-synchronised measurement, distributed signal processing, electrochemical impedance spectroscopy, Goertzel-Algorithm, wireless communication

Abstract

This thesis discusses the development of a measuring system for radio-synchronised electrochemical impedance spectroscopy on battery cells. Therefore, a $LiFePO_4$-battery cell is being stimulated by an ac current and measured by the measuring system. A cell sensor is measuring the ac voltage which has been caused by the ac current. The synchronous measurement of current and voltage is realized by a proprietary wireless protocol. The impedance spectrum is calculated through the phase shiftment of the current and the voltage that occur within different frequencies.

Inhaltsverzeichnis

1. Einleitung **8**
1.1. Stand der Technik . . . 9
1.2. Motivation und Zielsetzung . . . 10

2. Theoretische Grundlagen **12**
2.1. Die Lithium-Batterie . . . 12
2.1.1. Aufbau einer Lithium-Ionen Zelle . . . 14
2.1.2. Elektrochemischer Aufbau . . . 17
2.1.3. Batteriealterung . . . 24
2.1.4. Zustandsbestimmung einer Zelle . . . 26
2.2. Grundlagen der elektrochemischen Impedanzspektroskopie . . . 28
2.2.1. Anwendungsbereiche und Funktionsprinzip der EIS . . . 28
2.2.2. Mathematische Grundlagen zur Impedanzspektroskopie . . . 30
2.2.3. Darstellungsformen der komplexen Impedanz . . . 33
2.2.4. Informationen des Impedanzspektrums . . . 35
2.2.5. Anregung der Impedanzspektroskopie . . . 37
2.3. Batteriemodelle . . . 38
2.3.1. Batteriemodell nach Randles . . . 39
2.3.2. Optimiertes Batteriemodell . . . 44
2.3.3. Alterung am Batteriemodell . . . 45
2.4. Reale Batteriealterung . . . 47

3. Analyse und Neukonzeption des Messsystems **48**
3.1. Aktueller Stand des Messsystems . . . 48
3.1.1. Zellsensor . . . 49
3.1.2. Batteriesteuergerät . . . 53
3.2. Analyse der messtechnischen Anforderungen . . . 54
3.2.1. Analyse kommerzieller EIS-Labormessgeräte . . . 54
3.2.2. Messfrequenzbereich . . . 54
3.2.3. Impedanzbereich . . . 56
3.2.4. Spannungsbereich . . . 57
3.2.5. Phasengenauigkeit . . . 57
3.3. Synchronisation des Messsystems . . . 58
3.3.1. Synchronisation der Messwerte . . . 58

3.3.2. Synchronisation zwischen den Sensoren 62
3.4. Größe der Messamplitude 63
3.4.1. Impedanzmessung an Batterien 63
3.5. Neukonzeption des Messsystems 67
3.5.1. Messkonzeption für eine hochauflösende Messung 67
3.5.2. Messkonzeption für eine Messung mit analoger Vorverarbeitung . . 76
3.5.3. Vergleich der vorgestellten Konzepte 78
3.6. Berechnung des Impedanzspektrums durch verteilte Signalverarbeitung . . 80
3.6.1. Diskrete Fourier-Transformation - DFT 81
3.6.2. Fast Fourier-Transformation - FFT 82
3.6.3. Goertzel-Algorithmus 84
3.7. Abschätzung und Zusammenfassung der Analyse 88

4. Neuentwicklung und Redesign der bestehenden Hardware 90
4.1. Entwicklung des Batteriesteuergeräts 90
4.1.1. Basismodul des Batteriesteuergeräts 91
4.1.2. Transceiver-Erweiterungsmodul 92
4.1.3. Strommess-Erweiterungsmodul 93
4.1.4. Aufbau des Batteriesteuergeräts 94
4.2. Redesign des Zellsensors 95
4.2.1. Entwicklung der analoge Vorverarbeitung 95
4.2.2. Modifikationen des Schaltungsentwurfs 99
4.2.3. Mechanische- und elektrische Verbindung zwischen Zellsensor und Batteriezelle 103
4.2.4. Platinenlayout des Zellsensors 104
4.3. Hochauflösende AD-Wandler-Erweiterungsplatine 106

5. Softwareentwurf des Messsystems 107
5.1. Zellsensor Software 107
5.1.1. Speicheroptimierung der Messwerte 108
5.1.2. Dynamische Laufzeitermittlung 109
5.1.3. Steuerung der analogen Vorverarbeitung 112
5.2. Batteriesteuergerät Software 116
5.2.1. Funktionsübersicht des Batteriesteuergeräts 117
5.2.2. Taktaussendung und synchrone Strommessung 119
5.3. Implementierung des Goertzel-Algorithmus 120
5.3.1. Implementierung auf dem Zellsensor 120
5.3.2. Test der implementierten Filterstruktur 123

6. Inbetriebnahme und Erprobung der Impedanzspektroskopie 124
6.1. Inbetriebnahme der entwickelten Hardware 125
6.1.1. Inbetriebnahme des Batteriesteuergeräts 126

6.1.2. Inbetriebnahme des Zellsensors 128
6.2. Impedanzmessungen 138
6.2.1. Vermessung einer Testimpedanz 138
6.2.2. Untersuchung der Synchronität des Messsystems 143
6.3. Erprobung der Impedanzspektroskopie 147
6.3.1. Impedanzmessung einer Batteriezelle 148
6.3.2. Messungen an gealterten Batteriezellen 150
6.3.3. Impedanzmessung bei verschiedenen Ladezuständen 154
6.3.4. Impedanzmessung bei verschiedenen Temperaturen 156
6.4. Auswertung und Bewertung der Messergebnisse 157

7. Zusammenfassung und Ausblick 159
7.1. Zusammenfassung der Arbeit 159
7.2. Ausblick 160

Tabellenverzeichnis 163

Abbildungsverzeichnis 164

Literaturverzeichnis 170

A. Hochauflösende AD-Wandler Erweiterungsplatine 178

B. Spannungsversorgungs-Erweiterungsmodul 181

C. Simulation der Schleifenantenne 184

D. Alterung einer Batteriezelle 197

E. Impedanzmessung 199

F. Befehlsübersicht Batteriesteuergerät 203

G. Befestigungsadapter 207

H. PC-lauffähiges Goertzel-Testprogramm 210

I. Aufgabenstellung 213

J. Schaltplan 216

K. Platinenlayout 231

L. Quellcode des Messsystem in C 239

Listings **240**

M. Quellcode Matlab Auswertung **409**

N. Verzeichnisstruktur des Datenträgers **414**

1. Einleitung

Aktuell wird versucht, sich von der Abhängigkeit der fossilen Brennstoffen zu lösen. Besonders die sinkenden Erdölvorräte, der damit steigende Treibstoffpreis und der Klimawandel beschleunigen momentan den Willen, nach alternativen Antrieben zu suchen.

Dabei erlebt das Elektroauto in den letzten Jahren eine neue Renaissance. Gefördert von der Politik, der Industrie und der Gesellschaft, wird der Elektroantrieb von Fahrzeugen als zukunftsweisend angesehen. Besonders die Politik ist bemüht, eine ressourcenschonende und umweltfreundliche Alternative zum Verbrennungsmotor auf die Straßen zu bringen. So plant die Bundesregierung aktuell, dass bis zum Jahr 2020 eine Million Elektrofahrzeuge in Deutschland fahren sollen. Diese Zahl soll bis ins Jahr 2030 sogar bis auf sechs Millionen Fahrzeuge steigen [6].
Nach Vorstellungen der Bundesregierung, sollen die Autos mit erneuerbaren Energien geladen werden und fahren somit praktisch ohne Schadstoffausstoß [6]. Dazu hat die Regierung zusammen mit Partnern aus der Industrie die Nationale Plattform Elektromobilität (NPE) als ein Beratungsgremium der Bundesregierung zur Elektromobilität gegründet. Dieses Gremium soll der Bundesregierung Empfehlungen für die Umsetzung der Elektromobilität aussprechen. Ein Problem der Elektromobilität ist momentan die mangelnde Reichweite der Elektrofahrzeuge und die enormen Anschaffungskosten aufgrund der teuren Batterietechnik. Die Regierung versucht, durch Steuerbefreiung und Sonderparkflächen einen Kaufanreiz zu bieten. Dennoch scheuen sich die meisten Autofahrer vor der Anschaffung eines Elektroautos wegen der begrenzten Leistungsfähigkeit und Lebensdauer der Batteriezellen.

Deshalb will die Bundesregierung die Forschung und Entwicklung von Batterien zukünftig mit zwei Milliarden Euro fördern, da selbst die neuesten Batteriegenerationen zu schwer sind, um sie wirtschaftlich und mit der erforderlichen Reichweite in Fahrzeuge einzubauen [6]. Die Entwicklung neuer Batterietechnologien kann aber nur als langfristiges Ziel angesehen werden. Als mittelfristiges Ziel wird von der NPE die Forschung an effizienten Elektroniksystemen angegeben. Dabei sei der Schlüssel zu einer intelligenten und nachhaltigen Mobilität unter anderem das Forschungsfeld der Sensorik und des Batteriemanagements [15]. Es sollen also aktuelle Batteriesysteme in Kombination mit moderner Elektronik effizient ausgenutzt und somit die Lebensdauer der Batterie gesteigert werden.

1.1. Stand der Technik

An der Hochschule für Angewandte Wissenschaften (HAW) Hamburg gibt es eine Arbeitsgruppe, die sich mit der Entwicklung von intelligenter Überwachungssensorik von Batteriezellen beschäftigt. Dabei wird nicht nur die Batterie als Ganzes überwacht, sondern es wird eine Einzelzellüberwachung realisiert. Eine Besonderheit dieser Sensoren ist, dass es sich dabei um eine drahtlose Überwachung handelt. So wird der Batteriesensor von der Zelle versorgt, die er überwachen soll. Die Kommunikation und der Datenaustausch mit einem externen Steuergerät erfolgt dabei über eine Funkschnittstelle. Dies bringt enorme Vorteile mit sich, wie einen geringeren Verkabelungsaufwand, die Verhinderung eines Sensorausfalls durch Kabelbruch und besonders den Vorteil der Potenzialtrennung zwischen den einzelnen Batteriezellen. Die Abbildung 1.1 verdeutlicht dieses Prinzip der Einzelzellüberwachung und der zentralen Steuerung durch ein externes Batteriesteuergerät.

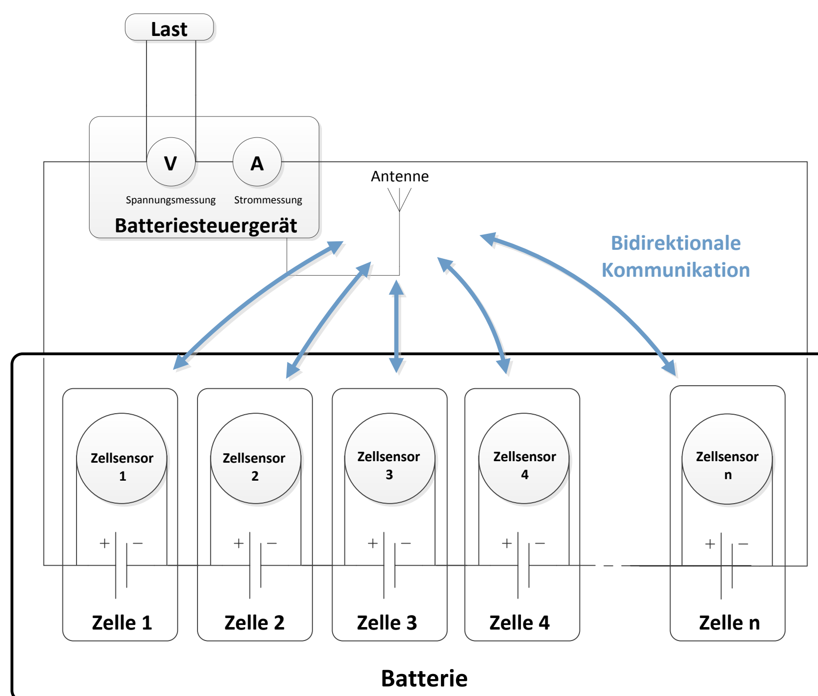


Abbildung 1.1.: Prinzip der Zellüberwachung und des Batteriemanagements (ent. aus [62])

Mittels dieser drahtlosen Sensorik können von jeder einzelnen Zelle die Temperatur und Spannung erfasst und überwacht werden. Mit diesen Parametern lässt sich schon eine Zustandsabschätzung der überwachten Batteriezelle machen. Dennoch lassen sich dadurch keine genauen Zustandsbestimmungen der Zelle machen, da diese ausgesprochen schwierig sind [68]. So kann der aktuelle Ladezustand nur näherungsweise anhand der gemessenen Spannung abgeleitet werden. Dabei fließen aber eine Menge Faktoren mit ein, die eine

genaue Bestimmung erschweren.
Eine Bestimmung des Alterungszustands der Zelle ist durch diese Parameter nur bedingt möglich. Doch gerade in Anwendungsbereichen wie der Elektromobilität ist es wichtig, das Altern einer Batteriezelle möglichst frühzeitig zu erkennen, bevor ihre Leistung nachlässt und somit den zuverlässigen Betrieb der gesamten Batterie verhindert.

Für eine genauere Ladungs- und Alterungsbestimmung einer Batteriezelle sind verschiedene Verfahren bekannt. Eines dieser Verfahren ist die elektrochemische Impedanzspektroskopie (EIS). Bei der EIS handelt es sich um ein Messverfahren, welches die komplexe Impedanz der Batteriezelle vermisst und daraus auf den Ladungs- und Alterungszustand schließen kann. Diese Messmethode findet sich heute schon in zahlreichen Anwendungsgebieten wieder und ist in der professionellen Batteriediagnostik ein häufig angewandtes Verfahren. Eine Vermessung der komplexen Impedanz ist momentan aber nur mit großen und teuren Labormessgeräten möglich, da es sich dabei um große Leistungselektronik und empfindliche Messtechnik handelt.

Ideal wäre, wenn sich die EIS nicht nur im Labor mit hohem Messaufwand, sondern auch innerhalb des Fahrzeuges an jeder einzelnen Batteriezelle durchführen lassen würde. Dazu muss die eingesetzte Elektronik aber hinsichtlich der Größe optimiert werden. Ein Teil der Forschungsgruppe der HAW Hamburg verfolgt dieses Ziel, die Funktionalität der EIS-Messung direkt auf den Batteriezellen zu integrieren. Dabei soll die Leistungselektronik und die Messelektronik voneinander getrennt werden. Durch eine Optimierung der Messelektronik soll diese komplett auf die drahtlosen Zellsensoren integriert werden.

An diesem Punkt gliedert sich diese Arbeit in das Forschungsprojekt ein. Es soll ein Messsystem, wie es in Abbildung 1.1 gezeigt wurde, entwickelt werden mit dem es möglich ist, die EIS-Messung an jeder einzelnen Zelle durchführen zu können. Dadurch soll das Gesamtziel des Forschungsprojekts, die Erhöhung der Betriebssicherheit und der Lebensdauer der Batteriezelle, erreicht werden.

1.2. Motivation und Zielsetzung

Die Diagnostik der Zellparameter ist für sehr viele Anwendungen von Interesse. Durch die Integration eines Zellsensors, der die EIS durchführen kann, würde diese Diagnostik enorm erleichtert werden. Durch eine genauere Abschätzung des Lade- und Alterungszustandes jeder einzelnen Zelle lässt sich die gesamte Batterie effizienter betreiben und senkt somit die Kosten für Wartung und die Neuanschaffung kompletter Batterien, die durch den Ausfall einer Zelle nicht mehr ihre gewünschte Leistung bringen.
Das Ziel dieser Arbeit soll es also sein, ein Messsystem, bestehend aus Batteriesteuergerät

und Zellsensoren zu entwickeln, mit dem es möglich sein soll, die EIS-Messung durchführen zu können. Dieses Messsystem kann auf der Basis von Vorarbeiten der Arbeitsgruppe entwickelt werden. Obwohl diese sich bereits mit den Untersuchungen der EIS-Messung beschäftigten haben, konnte bisher noch keine vollständige Impedanzspektroskopie durchgeführt werden.

Da es sich bei dem Messsystem um ein drahtlos kommunizierendes Konzept handelt, ist es neben der Integration der Messtechnik auf den Zellsensoren auch eine Herausforderung, die Synchronisation der Messwerte zwischen den Stromwerten des Batteriesteuergeräts und den Spannungswerten der Zellsensoren herzustellen. Für die Impedanzmessung ist das Verhältnis von Spannung und Strom und deren Phasenlage enorm wichtig, weshalb diese phasenrichtig gemessen werden müssen. Daher ist besonders auf eine synchrone Strom- und Spannungsaufnahme zu achten. Dies ist bei der Konzeptionierung des Messsystems immer zu beachten.

Aus den Anforderungen des Messsystems lassen sich Fragen ableiten, die diese vorliegende Arbeit beantworten soll.

• Ist es möglich, die gesamte Messelektronik, die für die Erfassung der EIS nötig ist, auf einem kleinen Zellsensor unterbringen?

• Können die Messwerte der Strom- und Spannungsmessung, die für die EIS-Messung nötigen sind, über eine drahtlos kommunizierende Schnittstelle soweit synchronisieren werden, dass eine phasenrichtige Aufnahme von Strom- und Spannung in ausreichender Genauigkeit möglich ist?

• Wenn eine ausreichende Synchronisierung der Messwerte gelingt, ist die Frage ob sich mit dem Messsystem EIS-Messungen durchführen lassen, die mit einem herkömmlichen Labormessgerät vergleichbar sind?

• Gelingen Messungen mit dem entwickelten Messsystem, so ist zu überprüfen, ob diese präzise genug sind, um Veränderungen der Parameter der Alterung, des Ladezustands oder der Temperatur zu detektieren.

Die Lösungen müssen im Laufe dieser Arbeit erarbeitet und anschließend überprüft werden. Gelingt die Integration der EIS-Messmethode auf den Zellsensoren, wäre dies ein großer Fortschritt für die Arbeitsgruppe und würde dem Ziel, einer effizienteren Nutzung der Batteriezelle und damit eine längere Betriebsdauer zu erreichen, ein großes Stück näher kommen.

2. Theoretische Grundlagen

In diesem Kapitel sollen die relevanten Grundlagen für das Verständnis der elektrochemischen Impedanzspektroskopie und deren Auswertung dargestellt werden. So werden der grundlegende chemische Aufbau sowie die elektrochemischen Vorgänge der Batterie erläutert und anhand eines bekannten Batteriemodells analysiert.
Des Weiteren wird in diesem Kapitel ein Überblick über den im Impedanzspektrum steckenden Informationsgehalt gegeben.

Grundlagen der Elektrochemie

In dieser Arbeit werden ausschließlich Lithium-Ionen-Akkumulatoren verwendet. Daher werden im Folgenden die grundlegenden Themen der Elektrochemie für Lithium-Ionen-Akkumulatoren erklärt, die zum Verständnis der elektrochemischen Impedanzspektroskopie nötig sind.

2.1. Die Lithium-Batterie

Seit den 60er Jahren des vergangenen Jahrhunderts wurde versucht, Batterien auf der Basis von Lithium aufzubauen. Der Vorteil der Lithium-Technologie ist die hohe Spannungslage und die hohe Energiedichte. Zudem hat sie flache Entladungskurven und ein sehr gutes Lagerverhalten mit einer geringen Selbstentladung.
Zunächst wurden damit Primärzellen hergestellt, d.h. diese waren nicht wiederaufladbar. Diese Primärzellen bestanden aus Lithium-Metall für die Anode und aus Braunstein (MnO_2) als Kathode [68]. Wiederaufladbare Lithium-Batteriezellen waren zu dieser Zeit aus Sicherheitsgründen nicht verfügbar. Grund hierfür war die starke Reaktionsfähigkeit von Lithium mit Wasser, wobei reaktiver Wasserstoff entsteht.
Erst 1991 brachte Sony eine Sekundärbatterie auf Basis von Lithium auf den Markt. Dabei wurde aber nicht mehr reines Lithium als Kathodenmaterial verwendet, sondern Lithium-Cobalt-Dioxid, kurz $LiCoO_2$. Als Anodenmaterial wurde zunächst amorpher Kohlenstoff verwendet, später dann Graphit. Diese Art der Lithiumzellen sind heute noch eine gängigen sekundären Batterie auf dem Markt. Vor allem im mobilen Sektor, wie Mobilfunkgeräten

oder Notebooks wird diese Batterietechnologie aufgrund ihrer hohen Energiedichte und des geringen Gewichts eingesetzt. Im Laufe der Zeit kamen eine Vielzahl von Zellen auf den Markt, bei denen unterschiedliche Aktivmaterialien für die Lithium-Ionen-Zelle eingesetzt wurden. Eine Auswahl der Materialien und deren unterschiedlich liegenden Potenziale zeigt die Abbildung 2.1.

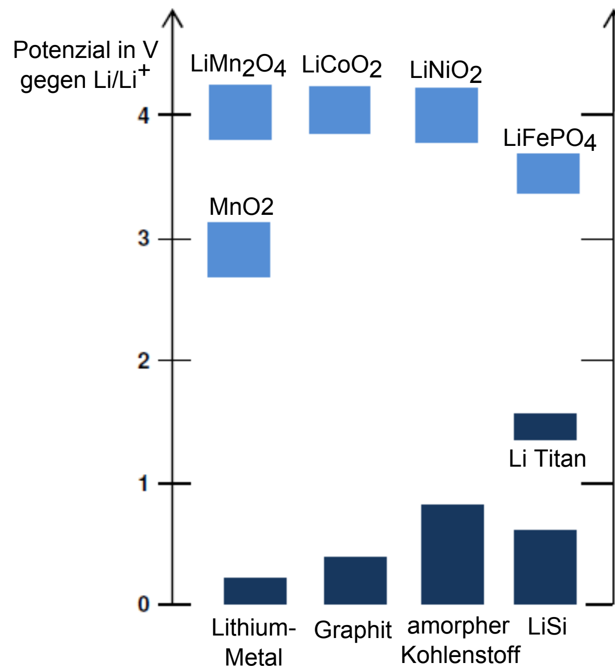


Abbildung 2.1.: Potenzialbereich verschiedener Aktivmaterialien (nach [72])

Je nach Anwendungsbereich werden unterschiedliche Kombinationen für das Anoden/Kathoden Material verwendet. Unterschiede bestehen dabei zum einen in der möglichen Zellspannung und zum anderen in der spezifischen Energie, d.h. wieviel Ladung pro kg die Zelle enthält. Ein weiterer Unterschied ist, ob die Zelle eher für den Hochstrombetrieb oder für Hochenergie-Anwendungen genutzt wird. Dabei wird meist das Kathodenmaterial geändert, für das Anodenmaterial wird derzeit überwiegend Graphit eingesetzt.
Im Kleingeräte-Markt, in dem vor allem mobile Geräte im Vordergrund stehen, wird heutzutage meist $LiCoO_2$ als Kathodenmaterial verwendet. Für Anwendungen im Hochstrombereich, wie Traktionsbatterien, wird Lithium-Eisen-Phosphat ($LiFePO_4$) eingesetzt. Diese sind wegen ihrer guten Hochstromfähigkeit und besonders wegen ihrer Sicherheit sehr gut für den Einsatz in der Elektromobilität geeignet. In dieser Arbeit werden deshalb alle Anforderungen und Messungen an $LiFePO_4$ Zellen durchgeführt.

2.1.1. Aufbau einer Lithium-Ionen Zelle

In diesem Abschnitt soll ein Überblick über den Aufbau einer Lithium-Ionen Zelle gegeben werden. Die Lithium-Ionen Batterie lässt sich prinzipiell wie in Abbildung 2.2 darstellen, bestehend aus zwei Elektroden, einem zwischen den Elektroden liegenden Separator, dem Elektrolyten und den Stromableitern.

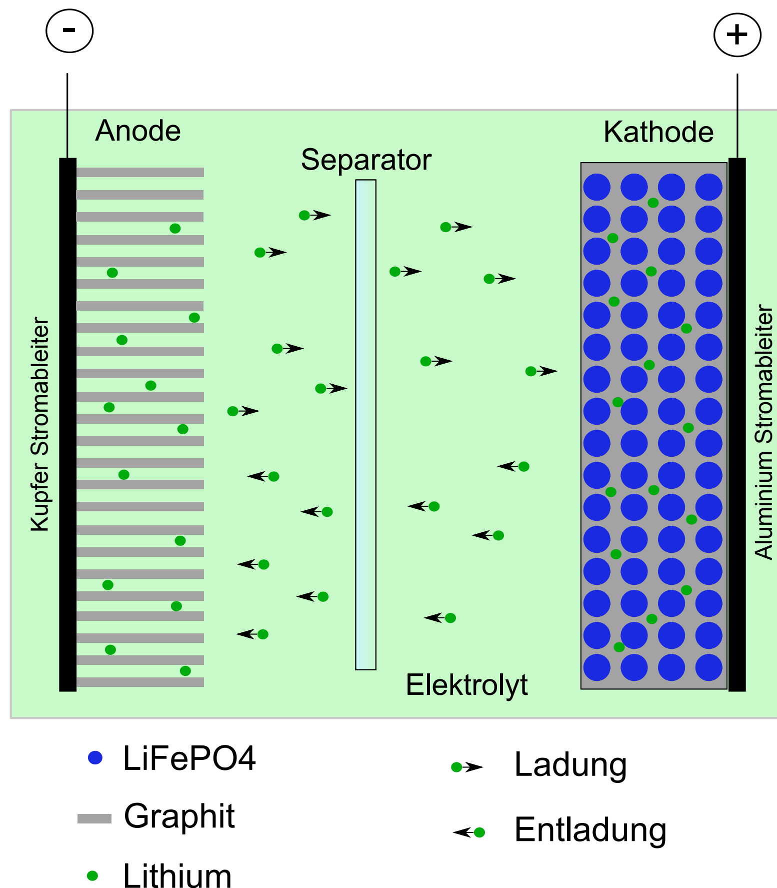


Abbildung 2.2.: Prinzipieller Aufbau einer $LiFePO_4$ Zelle (nach [72] [61])

Als Stromableiter wird meist Kupfer und Aluminium verwendet. Dabei wird an der negativen Elektrode Kupfer verwendet. An der positiven Seite kann kein Kupfer verwendet werden, da es hier zu starken Korrodierungen kommen kann [68]. Deshalb kommt hier das etwas schlechter leitende Aluminium als Ableiter zum Einsatz.
Als Elektrodenmaterial können verschiedene Materialien zum Einsatz kommen. Von der Auswahl dieser Materialien hängt u.a. auch die Spannung der Zelle, ihre Kapazität und die Sicherheit ab.

Kathode

Als Kathodenmaterial kommen in der Lithium-Ionen-Technologie meist Metalloxide mit eingelagertem Lithium zum Einsatz. Ein verbreitetes Kathodenmaterial ist Lithium-Cobalt-Dioxid $LiCoO_2$. Es besitzt eine mittlere Entladespannung von 3,9 V. Ein großes Problem dieses Materials ist die Gefahr von exothermen Reaktionen oberhalb von 4,4 V oder bei hohen Temperaturen. Deshalb ist beim Einsatz dieses Kathodenmaterials zwingend eine elektronische Überwachung der Zellen notwendig, um ein Überladen oder ein Überhitzen zu verhindern.

Ein weiteres Kathodenmaterial ist lithiiertes Manganoxid $LiMn_2O_4$. Dieses besitzt eine mittlere Entladespannung von 4,0 V. Der Vorteil dieses Materials ist sein gutes Sicherheitsverhalten bei hohen Temperaturen. Nachteil dabei ist aber eine bis zu 20 % geringere Energiedichte als bei $LiCoO_2$. Eingesetzt wird dieses Material vor allem im gemischten Zustand mit $LiCoO_2$ im Mittelstrombereich.

Sehr gute Hochstromeigenschaften hat das Lithium-Nickel-Oxid $LiNiO_2$. Es hat dabei eine mittlere Entladespannung von 3,8 V und eine sehr gute Energiedichte. Es wäre somit sehr gut geeignet für Traktionsbatterien. Der große Nachteil diese Kathodenmaterials ist das sehr kritische Sicherheitsverhalten. Daher wird es nicht für Traktionsbatterien eingesetzt.

Im Gegensatz zu den zuvor aufgeführten Kathodenmaterialien handelt es sich bei Lithium-Eisen-Phosphat $LiFePO_4$ um kein Oxid, sondern um ein Phosphat. Es wird also kein Sauerstoff freigesetzt, was dieses Kathodenmaterial sicher und thermisch stabil macht. Das Material besitzt eine hohe Kapazität in mAh/g und eine hohe Leistungsdichte, jedoch eine mittlere Spannung von nur 3,4 V. Die Energiedichte ist daher geringer als bei den zuvor vorgestellten Kathodenmaterialien. Die hohe Leistungsdichte und die guten Sicherheitseigenschaften machen dieses Kathodenmaterial zu einem sehr guten Kandidaten für den Einsatz in Fahrzeugbatterien.

Die in dieser Arbeit verwendeten Batteriezellen verwenden alle als Kathodenmaterial $LiFePO_4$.

Tabelle 2.1.: Übersicht der Kathodenmaterialien (nach [68])

	$LiCoO_2$	$LiMn_2O_4$	$LiNiO_2$	$LiFePO_4$
Mittlere Spannung	3,9 V	4,0 V	3,8 V	3,4 V
Kapazität in mAh/g	150	120	170	160
Energiedichte	+	-	++	- -
Sicherheit	-	+	-	++

Anode
Bei der Markteinführung der Lithium-Ionen Technologie wurde zunächst amorpher Kohlenstoff als Anodenmaterial verwendet, welcher eine Speicherkapazität von etwa 200 mAh/g aufweist [68]. Aufgrund der höheren Speicherkapazität von ca. 372 mAh/g [68] und dem ähnlich guten Sicherheitsverhalten für Lithium-Ionen wurde dann kurze Zeit später Graphit eingesetzt. Dieses wird auch noch heute verwendet.

Elektrolyt
Der Elektrolyt dient als Leitmedium für den Ionentransport zwischen den beiden Elektroden. Zum größten Teil besteht der Elektrolyt aus Lösungsmitteln und verschiedenen Leitsalzen. Im Gegensatz zu Blei- oder Nickelbatterien können bei Lithium-Ionen-Batterien keine wässrigen Elektrolyten eingesetzt werden, da Wasser sich bereits bei Spannungen unter 3 V zersetzt. Statt dessen werden organische Elektrolyte eingesetzt. Deren Nachteil ist allerdings eine deutlich schlechtere Ionenleitfähigkeit als bei wässrigen Elektrolyten [68].

Separator
Der Separator dient der elektrischen Isolation zwischen den beiden Elektroden. Er stellt also sicher, dass es zu keinem Kurzschluss zwischen Anode und Kathode kommt. Der Separator besteht aus einer hochporösen Plastikfolie, die aus Polyethylen (PE) oder auch aus Polypropylen (PP) besteht. Neben der elektrisch isolierenden Eigenschaft muss der Separator aber leitfähig für den Ionentransport zwischen Anode und Kathode sein. Dies erreicht er durch eine hohe Porosität.
Neben diesen Eigenschaften dient der Separator auch als Sicherheitselement innerhalb der Zelle. Bei zu hohen Temperaturen schmilzt er und unterbricht somit die Ionenleitfähigkeit, weshalb kein Strom mehr fließen kann. Dieser Sicherheitsmechanismus wird auch als “shut down“-Mechanismus bezeichnet [51].

SEI-Schicht
Die sogenannte SEI-Schicht (Solid Electrolyte Interface) ist eine Grenzschicht, die sich zwischen der negativen Elektrode und dem Elektrolyten ausbildet. Sie ist notwendig, um die Anodenseite von dem Elektrolyten zu trennen, da der Elektrolyt an der katalytisch wirkenden Kohlenstoffanode ansonsten fortlaufend zersetzt werden würde. Die SEI-Schicht bildet sich während der ersten Zyklen als Zersetzungsprodukt des Elektrolyten und wirkt als inertisierende Schutzschicht. Allerdings erschwert sie auch die Ionenleitfähigkeit der Zelle. Die unkontrollierte Ausbildung der SEI-Schicht ist, neben der fortlaufenden Schädigung der Elektroden, maßgeblich für die Alterung der Zelle verantwortlich [61]. Ziel für die Herstellung der Zellen ist es also, eine möglichst kontrollierte und dünne SEI-Schicht zu erzeugen, um die Ionenleitfähigkeit langfristig zu erhalten. Dies kann durch entsprechende Zugabe von Salzen zum Elektrolyten sowie die gezielte Steuerung der ersten Zellzyklen erreicht werden [68]. Über den Einfluss der SEI-Schicht auf die Alterung der Batteriezelle wird in Abschnitt 2.1.3 nochmals näher eingegangen.

2.1.2. Elektrochemischer Aufbau

In diesem Abschnitt soll der elektrochemische Aufbau der Zelle anhand von einzelnen Spannungen erläutert werden. Dabei lässt sich eine elektrochemische Zelle als Zusammenschaltung verschiedener Spannungsquellen beschreiben [68].
Zum einen ist dies die eigentliche Ruhespannung U_0 der Batteriezelle. Zum anderen sind es Spannungen, die innerhalb der Zelle auftreten, wenn diese mit dem Strom I durchflossen wird. Diese Spannungen nennen sich Überspannung η_i [68] und werden durch Spannungsabfälle innerhalb der Zelle verursacht. Sie lassen sich in verschiedene einzelne Überspannungen einteilen. Dabei tragen alle einzelnen Überspannungen eines elektrochemischen Gesamtprozesses zur Gesamtspannung U der Zelle bei und können mit der nachfolgenden Formel zusammengefasst werden [24]:

$$U = U_0 \pm \sum_i \eta_i \tag{2.1}$$

Diese Überspannungen werden im Falle des Ladens der Zelle zusammen mit der Ruhespannung addiert. Im Falle einer Entladung werden diese von der Ruhespannung abgezogen. Vereinfacht gesagt, tragen die Überspannungen beim Laden einer Zelle zu einer Spannungserhöhung und beim Entladen zu einer Spannungsreduzierung der Gesamtspannung U bei [68]. In Abbildung 2.3 ist dieses Prinzip vereinfacht dargestellt. Dabei ist der Fall einer Ladung der Zelle gezeigt. Im umgekehrten Fall, also einer Entladung, drehen sich die Spannungsrichtungen der Überspannungen um. Dabei werden dann die Überspannungen von der Ruhespannung abgezogen.

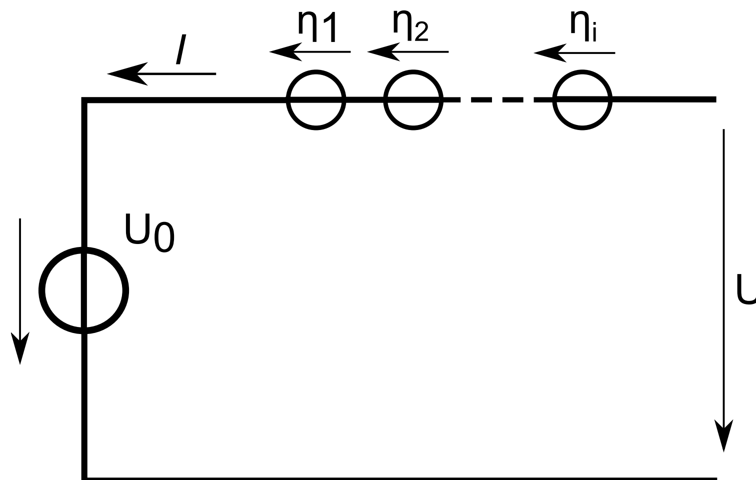


Abbildung 2.3.: Überspannungen an einer Zelle im Ladefall (nach [68])

Die Ursache dieser Überspannungen ist die begrenzte Leitfähigkeit der Materialien und der elektrochemischen Vorgänge innerhalb der Zelle. Fließt durch diese ein Strom, so entsteht

daran eine Überspannung. Nach [68] lassen sich die einzelnen Überspannungen einer Zelle vereinfacht in die folgenden Typen einteilen:

- Ohmsche Überspannung
- Durchtrittsüberspannung
- Diffusionsüberspannung

Im Folgenden werden diese Überspannungen näher betrachtet. Ziel dabei ist es, mit Hilfe der einzelnen Überspannungen ein Modell der Zelle aus elektrischen Elementen zu erstellen und den Aufbau zu verstehen.
Solch ein Batteriemodell, wie es aus der Literatur bekannt ist, soll eine quantitative Auswertung des Batteriezustandes ermöglichen. Dabei wird das Modell meist aus einfachen elektronischen Grundbausteinen wie Widerstand, Induktivität und Kapazität aufgebaut, können aber auch aus komplexen Bausteinen bestehen.

Ohmsche Überspannung

Die ohmsche Überspannung entsteht an den ohmschen Widerständen der Zelle. Diese setzen sich aus dem Widerstand der metallischen Ableiter, dem Widerstand des Elektrolyten und dem ohmschen Widerstand des Aktivmaterials zusammen. Der Großteil dieses Widerstandes wird durch den Elektrolyt verursacht [68], weshalb dieser hier als R_E (Elektrolyt-Widerstand) bezeichnet wird.

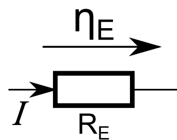


Abbildung 2.4.: Ohmsche Überspannung am Batteriemodell

Der ohmsche Widerstand steht in starker Abhängigkeit mit dem Alterszustand der Zelle [68]. So verändert sich mit zunehmendem Alterungszustand nicht nur die Leitfähigkeit des Elektrolyten, sondern es verändert sich vor allem der Widerstand des Aktivmaterials, verursacht durch eine Erhöhung des Übergangswiderstandes von der Elektrode zum Elektrolyten durch die Bildung der SEI-Deckschicht und der Beschädigung des Aktivmaterials [68].

Durchtrittsüberspannung

Eine weitere Überspannung ist die sogenannte Durchtrittsüberspannung. Diese wird durch den vom Strom I durchflossen Gleichstromwiderstand R_{ct} (charge transfer Widerstand) verursacht, der in Reihe mit dem Widerstand R_E liegt [42].

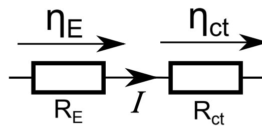


Abbildung 2.5.: Durchtrittsüberspannung am Batteriemodell

Durch das Anlegen einer Wechselspannung bzw. eines Wechselstromes wird die Zelle im Rhythmus der angelegten Frequenz angeregt. Dies bedeutet, dass die chemischen Vorgänge innerhalb der Zelle der Anregefrequenz folgen. Bei einer Anregefrequenz oberhalb von 1 Hz können diese nur noch einen quasistationären Zustand einnehmen [24]. Dies bedeutet, dass die Hin- und Rückreaktionen nicht mehr gleich schnell ablaufen können und es zu einer Verschiebung kommt. Es resultiert daraus eine Verschiebung des Ladungsdurchtritts i an den Elektroden. Dieses Verhalten lässt sich mittels des nichtlinearen Widerstandes R_{ct} beschreiben. Der Strom, der durch den Widerstand fließt, folgt der Kennlinie der Butler-Volmer-Gleichung 2.2 [24]. Sie beschreibt diese Verschiebung des Ladungsdurchtritts in Abhängigkeit des Durchtrittsfaktors α und der Austauschstromdichte i_0.

$$i = i_0 \left[\underbrace{exp\left(\frac{\alpha \cdot n \cdot F}{R \cdot T}\eta\right)}_{\text{Hinreaktion}} - \underbrace{exp\left(-\frac{(1-\alpha) \cdot n \cdot F}{R \cdot T}\eta\right)}_{\text{Rückreaktion}} \right] \tag{2.2}$$

Der Durchtrittsfaktor α gibt die Symmetrie der Hin- und Rückreaktion an. Das α kann dabei einen Wertebereich von 0 bis 1 annehmen.
n gibt die Anzahl der an der Reaktion beteiligten Elektronen an, F ist die faradaysche Konstante[1], R ist die allgemeine Gaskonstante[2] und T gibt die absolute Temperatur in Kelvin an.

[1] F = 96485.3365 C/mol
[2] R = 8,3144621 $J/(mol \cdot K)$

Im Folgenden wird die Butler-Volmer-Kennlinie für $\alpha = 0.5$ gezeigt. Dies bedeutet, dass die Hin- und Rückreaktionen gleich schnell ablaufen.

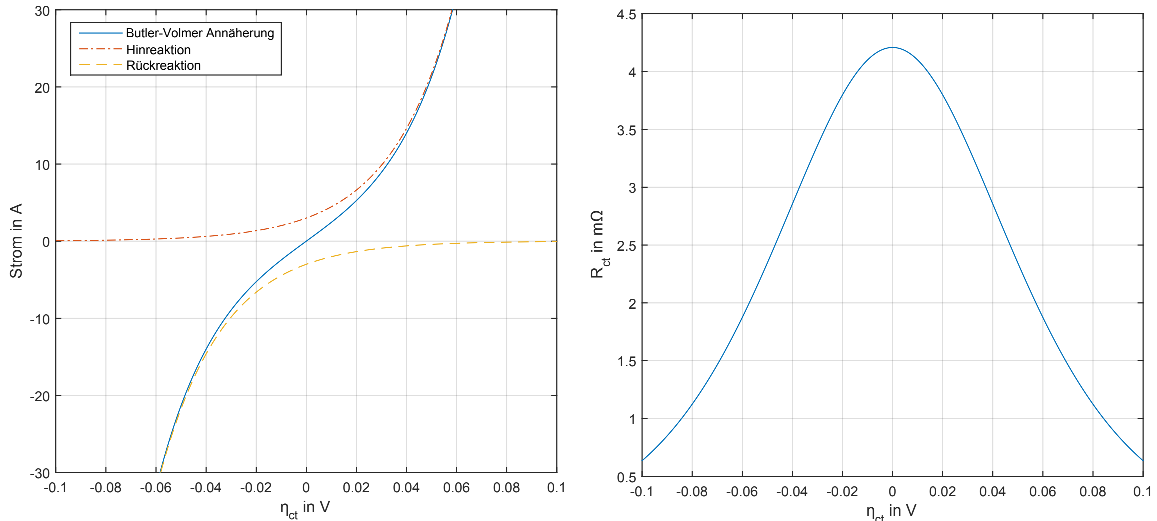


Abbildung 2.6.: Im linken Bild ist die Butler-Volmer-Annäherung mit i_0 = 3 A, $\alpha = 0.5$, T = 293 K und n = 2 zu sehen.
Im rechten Bild ist der Durchtrittswiderstand in Abhängigkeit der Überspannung η_{ct} zu sehen (nach [42]).

Im linken Teil der Abbildung 2.6 ist zum einen die Strom-/Spannungskennlinie der Butler-Volmer-Kennlinie zu sehen. Man erkennt eine starke Nichtlinearität zwischen der Überspannung η_{ct} und dem Strom.
Im rechten Teil der Abbildung ist der nichtlineare Widerstand R_{ct}, in Abhängigkeit der Überspannung η_{ct}, zu sehen. Dabei erkennt man, dass der Widerstand R_{ct} für große Ströme kleiner wird.

Doppelschichtkapazität

Beeinflusst wird die Durchtrittsüberspannung durch die Doppelschichtkapazität an den Elektroden. Wird eine Metallelektrode in einen Elektrolyten getaucht, entsteht an der Oberfläche der Elektrode eine Reaktion. Dabei entsteht die elektrochemische Doppelschicht an der Phasengrenze zwischen der Elektrode und dem Elektrolyten. Es stehen sich also zwei unterschiedlich geladene Potenziale gegenüber.

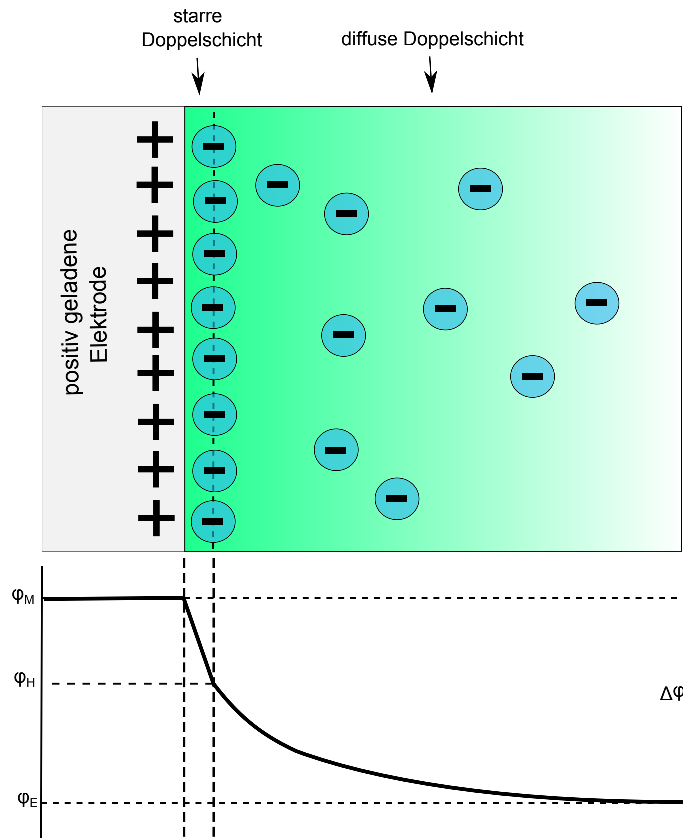


Abbildung 2.7.: Doppelschichtmodell mit Potenzialverlauf (nach [24])

φ_M ist das Potenzial der Elektrode
φ_H ist das Potenzial an der äußeren Helmholtzschicht
φ_E ist das Potenzial des Elektrolyten
$\Delta\varphi$ ist die Galvanispannung

An der Grenze der Doppelschicht bildet sich eine Kapazität mit der Ladung Q. Diese Ladung lässt sich in erster Näherung als normaler Plattenkondensator beschreiben. Dabei ist

die Ladung der Platten

$$Q = C \cdot U$$

Die Spannung U ist dabei der Potenzialunterschied $\Delta\varphi$ zwischen der Elektrode φ_M und dem Elektrolyten φ_E. Dieser wird beschrieben durch die Galvanispannung $\Delta\varphi$, wie sie auch in Abbildung 2.7 zu sehen ist.

$$\Delta\varphi = \varphi_M - \varphi_E$$

Berechnung der Doppelschichtkapazität C_{dl} mit der Galvanispannung:

$$C_{dl} = \frac{Q}{\Delta\varphi - \varphi_{Null}} \tag{2.3}$$

Durch das Verhältnis der großen Ladung Q, verursacht durch die große Oberfläche der Aktivmasse, zur kleinen Galvanispannung $\Delta\varphi$, entsteht eine sehr große Kapazität. Diese Kapazität verursacht zwar keine Überspannung, dennoch beeinflusst die Doppelschichtkapazität die Überspannung am Widerstand R_{ct}.
Diese Kapazität liegt im verwendeten Batteriezellenmodell parallel zum Widerstand R_{ct} und bildet somit ein RC-Glied.

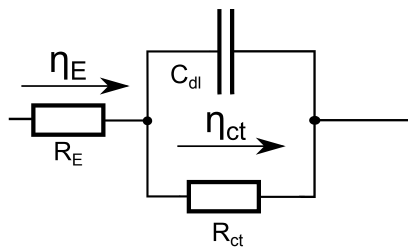


Abbildung 2.8.: Ohmsche Überspannung: Doppelschichtkapazität

Für Metallelektroden im Elektrolyten kann für eine erste Abschätzung der Kapazitat etwa 30 μF pro 1 cm^2 Elektrodenoberfläche angenommen werden [19].

Diffusionsüberspannung am Batteriemodell

Die Diffusionsüberspannung entsteht durch den Konzentrationsunterschied der Ionen zwischen der Elektrode und dem Elektrolyten. Diese Spannung hängt also von der vorhandenen Ionenladung innerhalb der Batteriezelle ab. In dem gewählten Batteriemodell lässt sich die Diffusionsüberspannung mit der Warburg-Impedanz Z_W darstellen. Die Warburg-Impedanz ist eine komplexe Impedanz, welche die Diffusion der einzelnen Ladungsträger innerhalb des Elektrolyten beschreibt. Beschreiben lässt sich die Impedanz nach Formel 2.4 [46].

$$Z_W = \sigma\,(\omega)^{-1/2} \cdot (1 - j) \tag{2.4}$$

Dabei wird σ als Warburg-Koeffizient bezeichnet, der hier nur der Vollständigkeit halber aufgeführt wird [46].

$$\sigma = \frac{R \cdot T}{n^2 F^2 A \sqrt{2}} \cdot \left(\frac{1}{C_0 \sqrt{D_0}} + \frac{1}{C_R \sqrt{D_R}} \right) \left[\frac{\Omega}{\sqrt{sek.}} \right] \tag{2.5}$$

Die folgende Abbildung zeigt die Warburg-Impedanz als Bode- sowie als Nyquist-Plot.

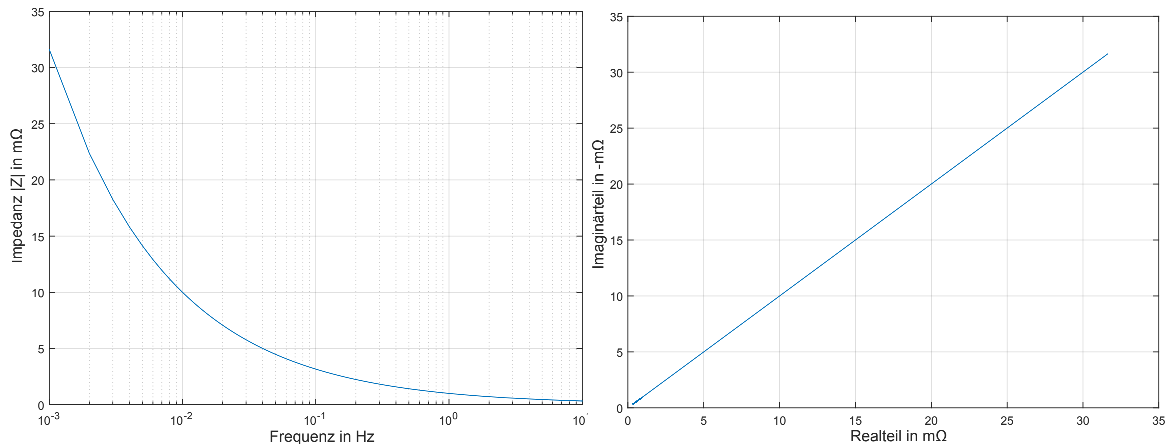


Abbildung 2.9.: Im linken Bild ist der Betrag der Impedanz Z logarithmisch über der Frequenz dargestellt. Im rechten Bild ist der Imaginärteil in Abhängigkeit des Realteils aufgetragen

In der linken Abbildung ist zu sehen, dass die Impedanz mit steigender Frequenz stark abnimmt, d.h. dass deren Einfluss erst mit niedriger Frequenz eintritt. In der rechten Abbildung ist die für die Warburg-Impedanz typische Steigung von 45 ° (für σ = 1) zu sehen. Die Warburg-Impedanz fügt sich im Batteriemodell in Reihe mit dem Widerstand R_{ct} ein [42]. Abbildung 2.10 zeigt das Batteriemodell mit allen drei beschriebenen Überspannungen.

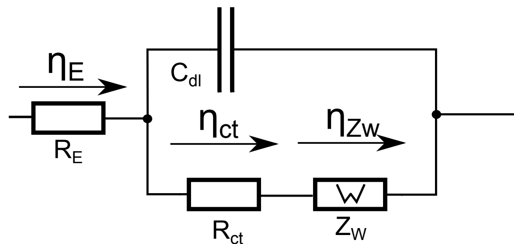


Abbildung 2.10.: Vollständiges Batteriemodell der drei Überspannungen

2.1.3. Batteriealterung

Die Alterung einer Batteriezelle wird überwiegend durch chemische Prozesse hervorgerufen. Dabei kommt es zu einer Erhöhung des Innenwiderstands der Zelle. Ebenfalls kommt es zu einem irreversiblen Kapazitätsverlust der Batteriezelle. Dabei gibt es grundsätzlich zwei verschiedene Arten, wie die Batteriezelle altern kann. Zum einen ist es die normale kalendarische Alterung der Zelle. Diese kalendarische Alterung wird im Wesentlichen durch das kontinuierliche Wachstum von SEI-Deckschichten verursacht. Zum anderen wird die Zelle durch ihre Nutzung gealtert. Dabei entstehen mechanische Belastungen, durch die Veränderungen des Aktivmaterials. Dadurch entstehen irreversible Kapazitätsverluste an der Batteriezelle, die zur Alterung führen [68] [61].

Kalendarische Alterung durch SEI-Wachstum

Die Hauptursache für die Alterung ist die Erhöhung des Innenwiderstands, hervorgerufen durch die Deckschichtbildung der SEI-Schicht [25]. Wie in Abschnitt 2.1.1 beschrieben, bildet sich die SEI-Deckschicht bereits bei der Herstellung der Zelle an der negativen Elektrode und wächst mit der Zeit weiter an. Dies vermindert die Ionen-Leitfähigkeit zwischen Aktivmaterial und Elektrolyt, weshalb der Innenwiderstand der Zelle mit zunehmender Alterung steigt. Abbildung 2.11 zeigt das Prinzip des SEI-Wachstums und den daraus erhöhten Widerstand für den Lithium Transport.

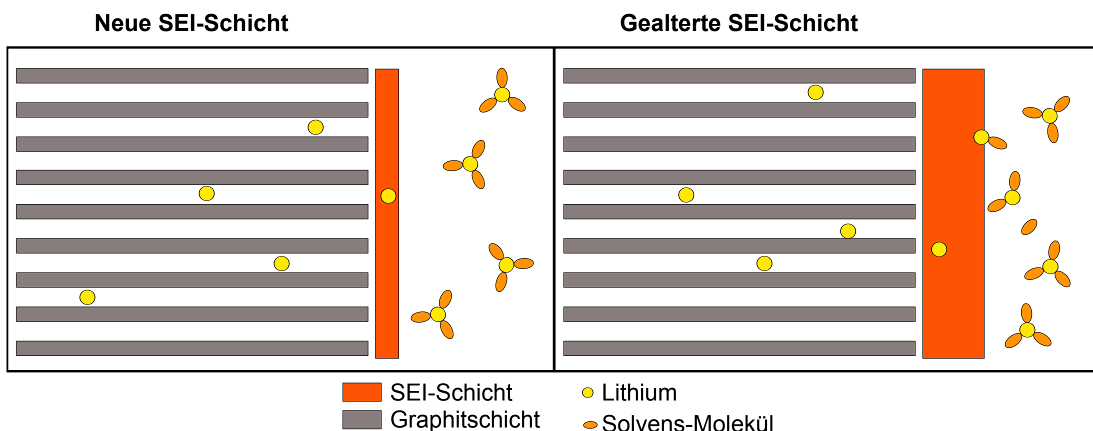


Abbildung 2.11.: Prinzip SEI-Wachstum (nach [61])

In der Abbildung 2.12 (entnommen aus [26]) ist eine Elektrodenmikroskopaufnahme von neuem und altem $LiFePO_4$ Material gegenübergestellt. Dieses Material wurde für mehrere Monate gelagert. Deutlich zu sehen ist die Veränderung der Oberfläche des gealterten Materials. Laut [17] entspricht die SEI-Schicht der weißen Oberflächen, die in der Abbildung zu sehen sind.

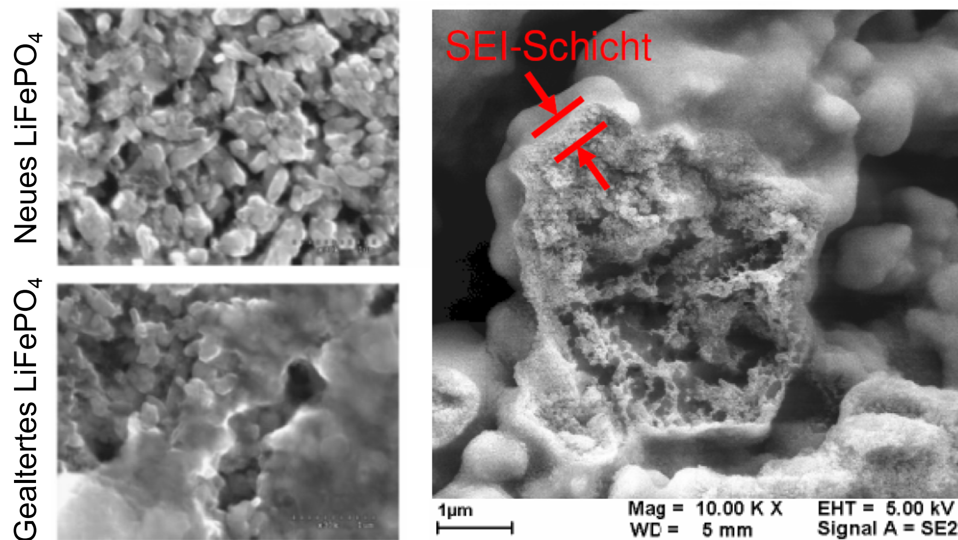


Abbildung 2.12.: SEI Wachstum am gealterten $LiFePo_4$ Material (ent. aus [26] [45])

Neben dem kontinuierlichen Wachstum der SEI-Schicht, wird das Wachstum durch verschiedene Faktoren teilweise stark beschleunigt. So ist die Lagerung oder der Betrieb der Batteriezelle in hohen Temperaturbereichen (>50°C) [26] für ein beschleunigtes Wachstum der SEI-Schicht verantwortlich und fördert somit die Alterung der Batteriezelle.

Alterung durch Zyklierung

Ein weiterer Alterungseffekt tritt durch den normalen Betrieb der Zelle auf. Durch die Zyklierung der Zelle werden Lithium-Ionen innerhalb der Graphitschicht ein- und ausgelagert. Dieses Ein- und Auslagern führt zu einer Volumenänderung des Aktivmaterials. Durch die mechanische Belastung kommt es zu einem Aufbrechen der Graphitstruktur, in der sich wiederum eine Deckschicht bildet. Durch diese Reaktion reduziert sich schließlich die aktive Elektrodenoberfläche, was einen Kapazitätsverlust zur Folge hat.

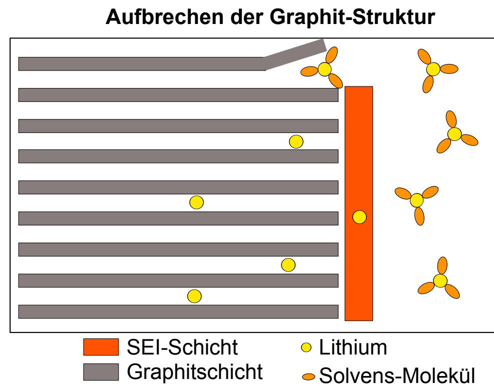


Abbildung 2.13.: Aufbrechen der Graphit-Struktur (nach [26])

Des Weiteren kommt es durch die Volumenänderung zu Kontaktverlusten zwischen dem Aktivmaterial und dessen Stromableiter. Somit wird auch an dieser Stelle die Leitfähigkeit verschlechtert und der Widerstand der Zelle steigt an.

2.1.4. Zustandsbestimmung einer Zelle

In diesem Abschnitt sollen einige Begriffe erklärt werden, die im Zusammenhang mit der Zustandsbestimmung der Batteriezelle auftauchen.

Elektrische Ladung einer Zelle

Bei der Kapazität einer Zelle ist zwischen der Nennkapazität und der momentan tatsächlich vorhandene Kapazität zu unterscheiden. Die Nennkapazität C_N ist die von Hersteller angegebene Kapazität. Die Kapazität C_m ist die im momentan tatsächliche vorhandene Kapazität, wie in Abbildung 2.14 zu sehen ist. Diese ist, bedingt durch Alterung der Zelle, immer kleiner als die Nennkapazität.

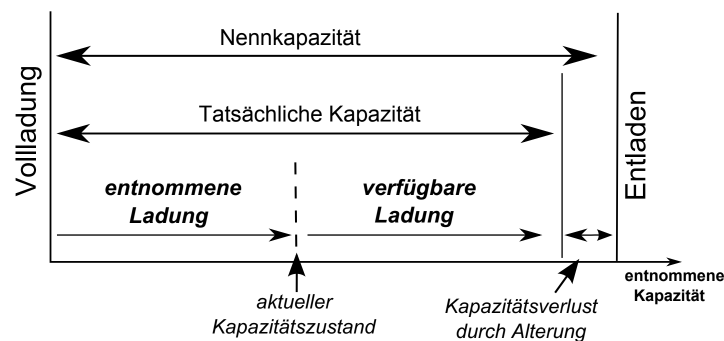


Abbildung 2.14.: Tatsächliche Kapazität einer Zelle (nach [68])

Gesundheitszustand einer Zelle - State of Health und End of Life

Der State of Health (SoH) gibt als Quotient aus Ist-Kapazität und Nennkapazität eine Aussage über den Grad der Alterung einer Batteriezelle [47]. Dabei wird eine neue Zelle mit einem SoH von 100% bewertet. Mit fortschreitender Alterung fällt der SoH bis zu einem Wert von 0%. Der Bereich, an dem die Batteriezelle noch ca. 60-80%[3] ihrer Nennkapazität besitzt, wird auch der End of Life (EoL) genannt. Ab diesem Bereich fällt die noch vorhandene Kapazität der Batteriezelle mit jedem Zyklus stark ab [68], bis der Zustand von 0% SoH erreicht ist. Der Gesundheitszustand der Zelle lässt sich anhand der noch vorhandenen Kapazität C_m und der Nennkapazität C_N berechnen.

$$\mathrm{SoH} = \frac{C_m}{C_N} \tag{2.6}$$

Ladezustand einer Zelle - State of Charge und Depth of Discharge

Der Ladezustand einer Zelle wird als State of Charge (SoC) bezeichnet. Dieser gibt den aktuellen Ladezustand der Zelle an, im Verhältnis zu ihrer Nennkapazität. Der Ladezustand wird dabei im Bereich vom 0 % für eine vollkommen entladene Zelle bis zu 100 % für eine voll geladene Zelle angegeben.

$$\mathrm{SoC} = \frac{C_N \text{ - Entnommene Ladung}}{C_N} \tag{2.7}$$

Eine alternative Angabe für den Ladezustand einer Zelle wird durch die Entladungstiefe, welche auch "Depth of Discharge (DoD)"genannt wird, angegeben. Bei einer voll geladenen Batteriezelle wird der DoD zu 0 % angegeben. Im Gegenzug wird eine komplett entladene Zell mit dem DoD von 100 % angegeben. Diese Angabe ist somit der Gegenteil des SoC. Der DoD lässt sich wie folgt bestimmen [61]:

$$\mathrm{DoD} = 100\,\%\text{- SoC} \tag{2.8}$$

[3] Dabei gibt die Literatur unterschiedliche Angaben an

2.2. Grundlagen der elektrochemischen Impedanzspektroskopie

2.2.1. Anwendungsbereiche und Funktionsprinzip der EIS

Anwendungsbereiche

Die Impedanzspektroskopie ist ein bekanntes Messverfahren zur Untersuchung des frequenzabhängigen Impedanzverhaltens von unterschiedlichen Systemen [61]. Es findet sich ein breites Anwendungsfeld für diese Art der Untersuchung.
Eingesetzt wird die Impedanzspektroskopie, neben der Materialforschung, auch in der Medizintechnik [42]. Einen intensiven Einsatz findet die Impedanzspektroskopie in der Elektrochemie. Dabei wird diese Messmethode zur Analyse von galvanischen Zellen genutzt. Neben der Untersuchung von Brennstoffzellen [71] kann die Impedanzspektroskopie auch an Batterien durchgeführt werden. Mittels dieses Impedanzspektrums lassen sich Zustandsbestimmungen wie Alterung und Ladezustand der Batterie ermitteln und bewerten [68]. Auch können elektrochemische Vorgänge innerhalb der Batteriezelle untersucht und beobachtet werden [42]. Es ist also ein sehr vielfältiges Verfahren, dass heutzutage auch immer mehr Beachtung in der Forschung und der Diagnostik findet.

Funktionsprinzip

Zur Bestimmung des Impedanzspektrums wird das zu untersuchende System mit Hilfe eines periodischen Strom- bzw. Spannungssignals angeregt. Meist wird dazu ein sinusförmiges Anregesignal verwendet. Das zu untersuchende System antwortet bei einer Anregung mit Spannung mit einem, um den Phasenwinkel φ verschobenen Strom.

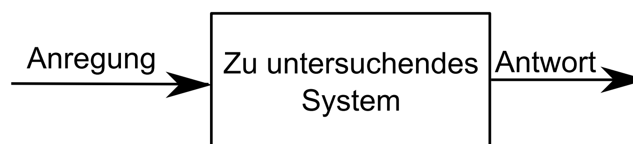


Abbildung 2.15.: Angeregtes System

Bei der Anregung mit einem Stromsignal wird die Antwort eine zum Anregesignal verschobene Spannung sein. Durch die Phasenverschiebung $\Delta\varphi$ zwischen Anregung und Antwort sowie der Amplitudenhöhe der beiden Signale lässt sich nun die komplexe Impedanz des zu vermessenden Systems berechnen. Eine solche exemplarische Phasenverschiebung zwischen Anregung und Antwort zeigt die Abbildung 2.16.

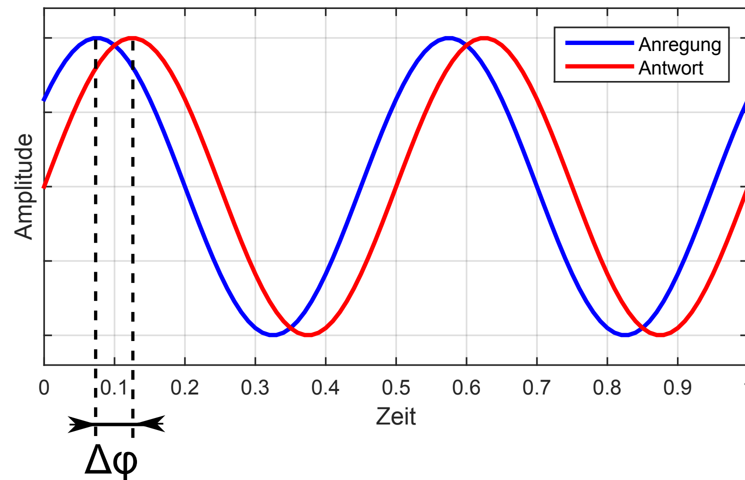


Abbildung 2.16.: Typische Phasenverschiebung zwischen Anregung und Antwort

Bei einem frequenzabhängigen System wird sich die Phasenverschiebung $\Delta\varphi$ und die Amplitudenhöhe von Strom und Spannung in verschiedenen Frequenzbereichen ändern. Durch diese Änderung bei unterschiedlichen Frequenzen erhält man bei jeder Frequenz eine unterschiedliche komplexe Impedanz. Teilt man die komplexen Impedanzen, die bei verschiedenen Frequenzen gemessen wurden, in ihre Real- und Imaginärteile auf, so erhält man das Impedanzspektrum.

2.2.2. Mathematische Grundlagen zur Impedanzspektroskopie

Das Ziel der elektrochemischen Impedanzspektroskopie ist die Ermittlung des komplexen Innenwiderstandes $\underline{Z}$ der zu vermessenden Batteriezelle bei verschiedenen Frequenzen. Als komplexer Widerstand $\underline{Z}$ wird der Quotient der komplexen Spannung $\underline{U}$ und des komplexen Stroms $\underline{I}$ eines Zweipols bezeichnet.

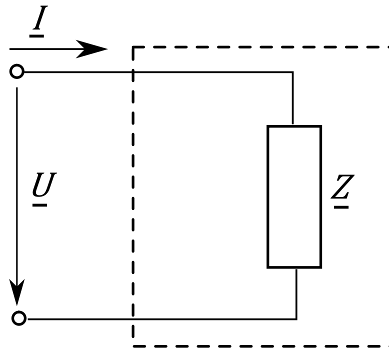


Abbildung 2.17.: Zweipol

Die Gleichung 2.9 zeigt den Zusammenhang zwischen komplexer Impedanz $\underline{Z}$, der komplexen Spannung $\underline{U}$ und des komplexen Stroms $\underline{I}$.

$$\underline{Z}\left(\omega_0\right) = \frac{\underline{U}\left(\omega_0\right)}{\underline{I}\left(\omega_0\right)} \tag{2.9}$$

Die Kreisfrequenz ω_0 ist dabei wie folgt definiert:

$$\omega_0 = 2 \cdot \pi \cdot f_0 \tag{2.10}$$

$\underline{U}\left(\omega_0\right)$, $\underline{I}\left(\omega_0\right)$ sowie $\underline{Z}\left(\omega_0\right)$ sind somit kontinuierliche Funktionen der Frequenz f_0. Im Folgenden wird nun davon ausgegangen, dass es sich um eine komplexe Sinusspannung $\underline{U}\left(\omega_0\right)$ mit der Phasenlage φ_u und um einen komplexen Strom $\underline{I}\left(\omega_0\right)$ mit der Phasenlage φ_i handelt. Im eingeschwungenen Zustand des Systems lassen sich für Spannung und Strom die zeitabhängigen Werte wie folgt darstellen:

$$u\left(t\right) = \widehat{u} \cdot sin\left(\omega_0 t + \varphi_u\right) \tag{2.11}$$

$$i\left(t\right) = \widehat{i} \cdot sin\left(\omega_0 t + \varphi_i\right) \tag{2.12}$$

Das Zeitsignal von Spannung und Strom ist hier in Abbildung 2.18 dargestellt. Dabei beschreibt der Phasenverschiebungswinkel $\Delta\varphi$ die Differenz der Phasenlagen der Spannung und des Stromes.

$$\Delta\varphi = \varphi_u - \varphi_i \tag{2.13}$$

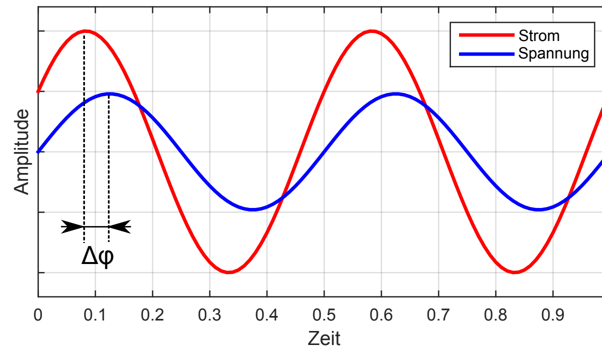


Abbildung 2.18.: Strom-Spannungs-Zeitsignale einer komplexen Impedanz

Durch die Tatsache, dass der Strom in der exemplarischen Abbildung 2.18 der Spannung voreilt, kann daraus geschlossen werden, dass es sich bei dem vorliegenden System um ein überwiegend kapazitives handelt. Bei einem überwiegend induktiven System wäre der umgekehrte Fall vorhanden, der Strom würde der Spannung nacheilen [18]. Da es sich bei der elektrochemischen Impedanzspektroskopie aber überwiegend um ein kapazitives System handelt, wird für die Beschreibung der mathematischen Grundlagen auch ein solches gewählt.

Um nun leichter rechnen zu können, werden die Spannung und der Strom, wie in der Wechselstromtechnik üblich, als komplexe Zahlenwerte dargestellt. Zunächst wird die Spannung in ihren Real- und Imaginärteil aufgespalten.

$$Re\left\{U\left(\omega_0\right)\right\} = \widehat{u} \cdot cos\left(\omega_0 t + \varphi_u\right) \tag{2.14}$$

$$Im\left\{U\left(\omega_0\right)\right\} = \widehat{u} \cdot sin\left(\omega_0 t + \varphi_u\right) \tag{2.15}$$

Diese Aufspaltung lässt sich ebenfalls mit dem Strom durchführen.

$$Re\left\{I\left(\omega_0\right)\right\} = \widehat{i} \cdot cos\left(\omega_0 t + \varphi_i\right) \tag{2.16}$$

$$Im\left\{I\left(\omega_0\right)\right\} = \widehat{i} \cdot sin\left(\omega_0 t + \varphi_i\right) \tag{2.17}$$

Durch das Zusammenfassen von Real- und Imaginärteil ergibt sich nun der komplexe Spannungs- bzw. der komplexe Stromwert.

$$\underline{u}\left(t\right) = \underbrace{\widehat{u} \cdot cos\left(\omega_0 t + \varphi_u\right)}_{\text{Realteil}} + \underbrace{j \cdot \widehat{u} \cdot sin\left(\omega_0 t + \varphi_u\right)}_{\text{Imaginärteil}} \tag{2.18}$$

$$\underline{i}\left(t\right) = \underbrace{\widehat{i} \cdot cos\left(\omega_0 t + \varphi_i\right)}_{\text{Realteil}} + \underbrace{j \cdot \widehat{i} \cdot sin\left(\omega_0 t + \varphi_i\right)}_{\text{Imaginärteil}} \tag{2.19}$$

Mit Hilfe der komplexen Zahlen lässt sich nun ein Zeigerdiagramm erstellen, welches die

Spannung und den Strom in der komplexen Zahlenebene darstellt. Die absolute Lage der Zeiger ist dabei nicht erforderlich, wichtig ist nur die Differenz des Phasenverschiebungswinkels $\Delta\varphi$ in der Zahlenebene. In Abbildung 2.19 nimmt die Spannung die Lage des Nullphasenwinkels ein, d.h. $\varphi_u = 0$

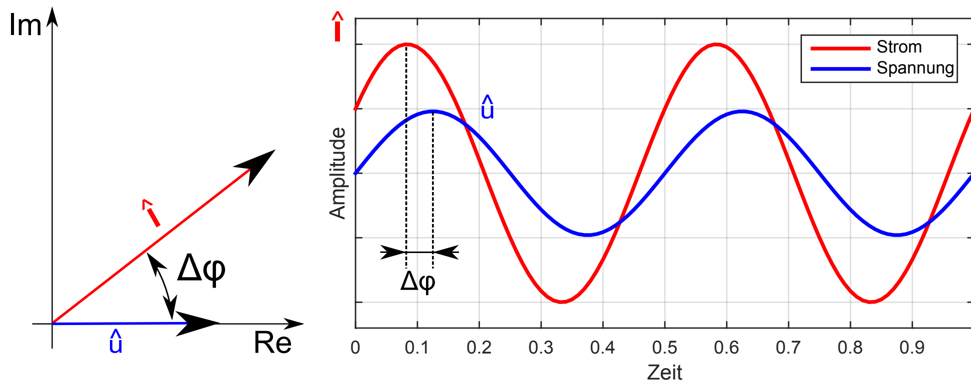


Abbildung 2.19.: Strom-Spannungs-Zeigerdiagramm

Mittels der Eulerschen Beziehungen lässt sich die trigonometrische Form in die Exponentialform umrechnen. Dabei ergeben sich für die Spannung und für den Strom die folgenden Gleichungen:

$$\underline{U}\,(t) = \widehat{u} \cdot e^{j\omega_0 t + \varphi_u} \tag{2.20}$$

$$\underline{I}\,(t) = \widehat{i} \cdot e^{j\omega_0 t + \varphi_i} \tag{2.21}$$

Durch die Überführung der trigonometrischen Form in die Exponentialform lässt sich nun auf einfache Weise der komplexe Wechselstromwiderstand berechnen. Diese Definition entspricht der des Gleichstromwiderstandes mit der zusätzlichen Information der Phasenlage [18].

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{U \cdot e^{j(\varphi_U)}}{I \cdot e^{j(\varphi_I)}} = \frac{U}{I} \cdot e^{j(\varphi_U - \varphi_I)} = |Z| \cdot e^{j\Delta\varphi_Z} \tag{2.22}$$

Dabei beschreibt $|Z|$ den Betrag und $\Delta\varphi_Z$ die Phase des komplexen Wechselstromwiderstands.

2.2.3. Darstellungsformen der komplexen Impedanz

Um die gemessenen Impedanzen darzustellen, gibt es zwei Darstellungsformen. Dies ist zum einen die Darstellung im Bode-Diagramm, welche häufig in der Elektrotechnik Einsatz findet. Zum anderen gibt es die Darstellungsform im Nyquist-Diagramm. Nachfolgend werden beide Darstellungsarten kurz vorgestellt.

Bode-Diagramm

Beim Bode-Diagramm werden der Betrag und die Phasen der Impedanz in Abhängigkeit der Frequenz dargestellt. Diese werden jeweils einzeln in unterschiedlichen Abbildungen dargestellt.
Typischerweise wird die Frequenzachse in der Bode-Form logarithmisch auf der Abszisse aufgetragen, dadurch lassen sich kleine Frequenzen wie auch große Frequenzen übersichtlich darstellen. Der Betrag der Impedanz wird linear aufgetragen. In Abbildung 2.20 ist der Impedanzverlauf einer $LiFePO_4$-Batteriezelle zu sehen.

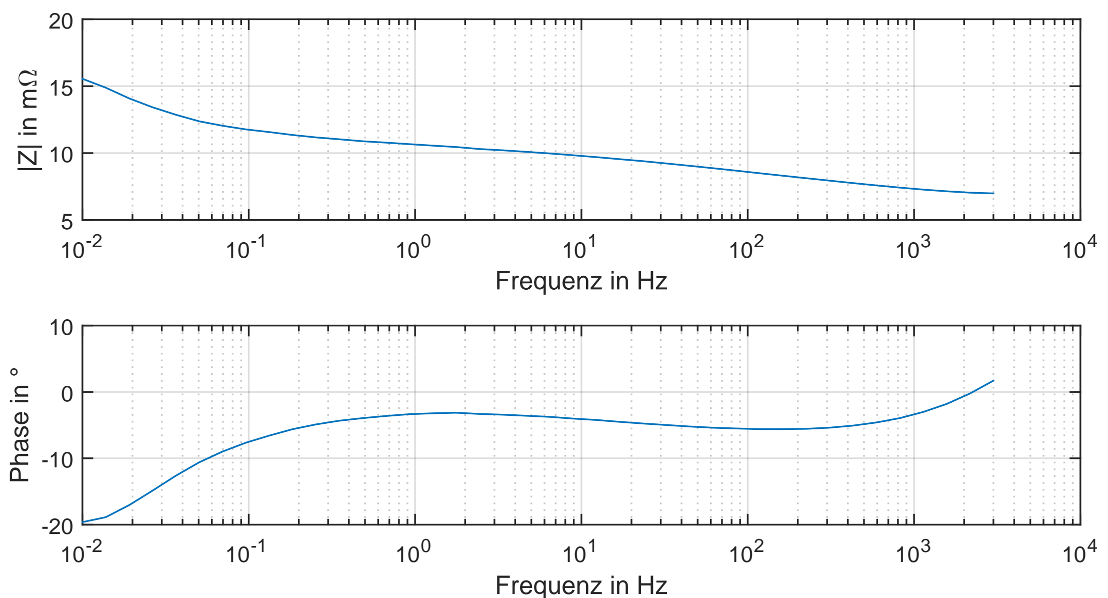


Abbildung 2.20.: Bode-Diagramm einer Impedanzspektroskopie einer $LiFePO_4$-Zelle

Nyquist-Diagramm

Eine in der Elektrochemie übliche Darstellungsform ist das Nyquist-Diagramm. Bei dieser Art der Darstellung wird die komplexe Impedanz in ihren Real- und Imaginärteil aufgespalten und jeweils auf einer Achse dargestellt. Meist wird im elektrochemischen Bereich die Ordinate umgekehrt dargestellt, da die meisten chemischen Systeme kapazitive Eigenschaften haben und somit das Diagramm einfacher zu lesen ist. Daher wird auch in dieser Arbeit diese Darstellungsform gewählt. Deshalb ist, wenn von einer Erhöhung der Impedanz gesprochen wird, von einem steigenden kapazitiven Anteil die Reden.

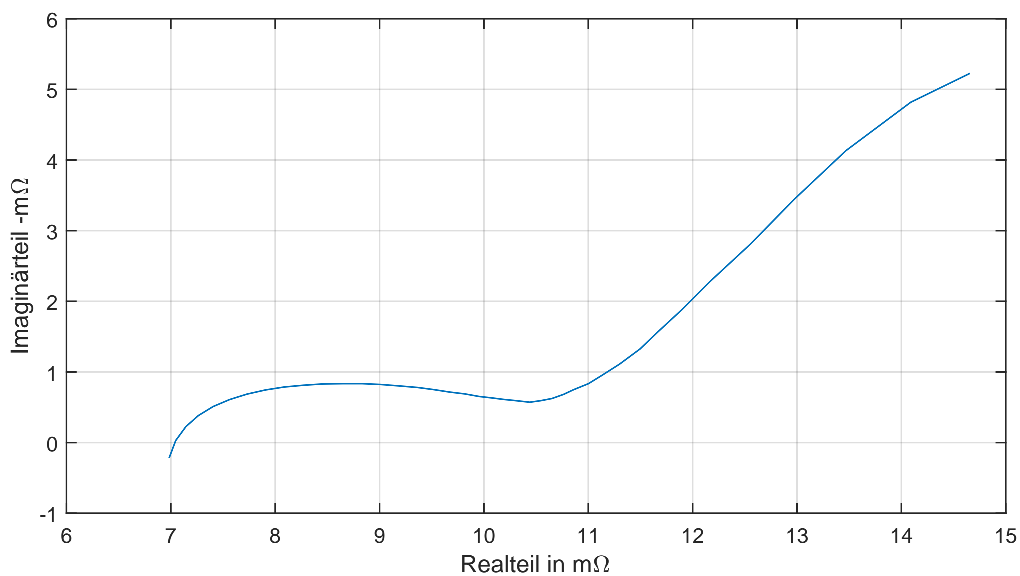


Abbildung 2.21.: Nyquist-Diagramm einer Impedanzspektroskopie

Ein großer Vorteil der Darstellung im Nyquist-Diagramm ist, dass man aus dem Impedanzspektrum schon sehr leicht in erster Näherung auf das Ersatzschaltbild des gemessenen Systems schließen kann. Darauf wird in Abschnitt 2.3 näher eingegangen.

Ein großer Nachteil dabei sind die fehlenden Frequenzangaben innerhalb des Diagramms. Meist werden einzelne Punkte innerhalb des Spektrums mit einer Frequenz beschriftet, um einen ungefähren Überblick über das Frequenzspektrum zu geben.

2.2.4. Informationen des Impedanzspektrums

Das charakteristische Impedanzspektrum einer Batteriezelle wird typischerweise im Frequenzbereich von wenigen mHz bis zu mehreren kHz aufgenommen. Dabei treten bei den verschiedenen Frequenzen unterschiedliche Reaktion an der zu vermessenden Batteriezelle auf. Allgemein kann die elektrochemische Impedanzspektroskopie in vier Bereiche aufgeteilt werden, wie in Abbildung 2.22 zu sehen ist. Im ersten Bereich treten induktive Effekte der Batterie zum Vorschein. Diese treten erst bei Frequenzen im kHz Bereich auf, weshalb sie in der Literatur auch gern als Hochfrequenzwiderstand benannt werden [29]. Da die induktiven Anteile für die Bestimmung der SoC und SoH wenig interessant sind, findet dieser Bereich meist wenig Beachtung. Der Bereich II beschreibt den Ladungsaustausch zwischen den Elektroden und dem Elektrolyt. Im Bereich III lassen sich Aussagen über die Stofftransporte bzw. die Diffusion der Ladungsträger innerhalb des Elektrolyts treffen. Der Bereich IV zeigt die Diffusionen außerhalb des porösen Plattenmaterials. Da diese Messungen erst ab dem unteren mHz bis μHz Bereich anfangen und somit die Messzeiten für diesen Bereich sehr hoch sind, wird dieser in der bekannten Literatur und auch in der vorliegenden Arbeit nicht weiter betrachtet.
Es werden die Bereiche I, II und III untersucht und im Folgenden näher erläutert.

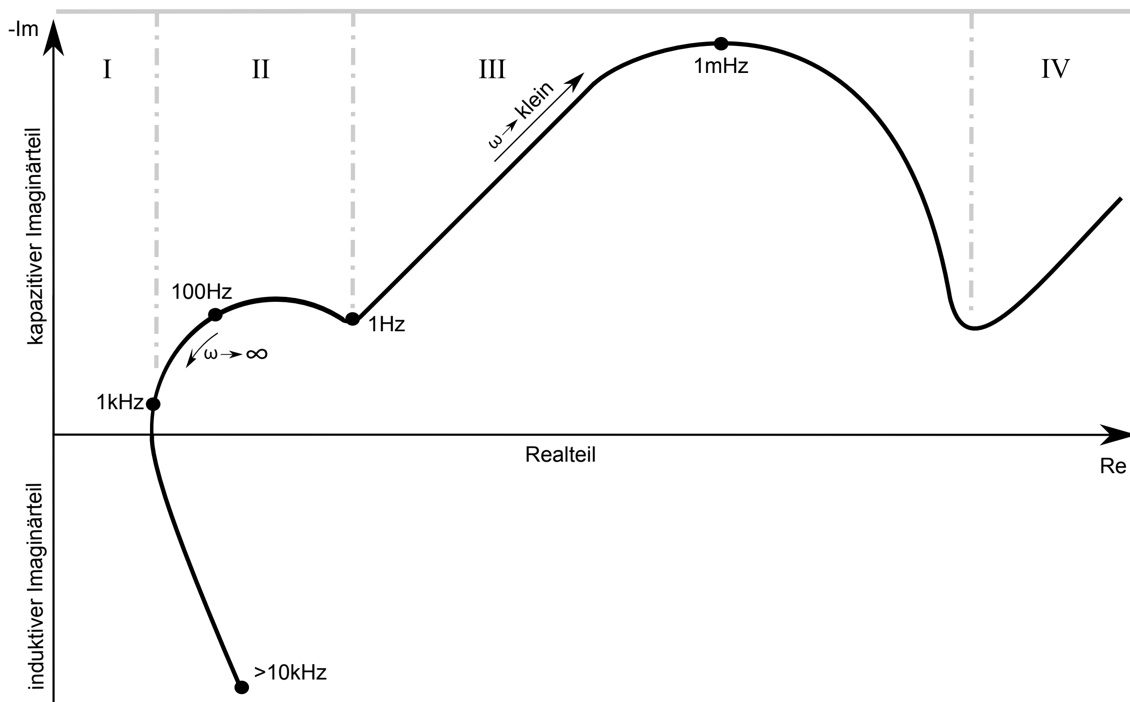


Abbildung 2.22.: Nyquist-Diagramm (nach [63])

Bereich I: Durchtrittspunkt und induktiver Bereich

Der induktive Bereich tritt erst bei Messfrequenzen im kHz Bereich auf. Der Punkt, an dem das Verhalten vom kapazitiven in den induktiven Teil wechselt, nennt sich Durchtrittspunkt [61]. Dieser Durchtrittspunkt, an dem der Imaginärteil der komplexen Impedanz zu Null wird, entspricht dem Innenwiederstand der Zelle. Dieser setzt sich in erster Linie aus dem Widerstand der Stromleiter, des Aktivmaterials und des Elektrolyten zusammen. Er stellt also einen sehr interessanten Punkt für den Alterungszustand der Batteriezelle dar, da hier unter anderem Rückschüsse auf die SEI-Deckschichten auf den Elektroden gezogen werden können. Der weitere induktive Anteil dagegen wird durch den geometrischen Aufbau der Batteriezelle beeinflusst und kann je nach Batterietyp stark variieren [57]. Konkrete Aussagen über den Zustand der Batteriezelle lassen sich durch den induktiven Anteil nicht machen, da dieser hauptsächlich durch die Induktivitäten der Messkabel beeinflusst wird. Dies ist der Grund, weshalb dieser Bereich in der Literatur wenig betrachtet wird und eine Messung über diesen Bereich nicht sinnvoll ist.

Bereich II: Ladungsaustausch zwischen den Elektroden und dem Elektrolyt

Dieser Bereich erstreckt sich typischerweise von 1 Hz bis hin zum Durchtrittspunkt bei einigen kHz. Dieser Teil des Spektrums beschreibt den Ladungsaustausch zwischen den Elektroden und dem Elektrolyten. In diesem Bereich kann also eine Aussage über den Ladungsdurchtritt an der Doppelschicht, wie er im Abschnitt 2.1.2 beschrieben wurde, erfolgen.

Bereich III: Diffusionsast

Im Diffusionsast sind Transportprozesse wie Diffusion und Ionenwanderung für das Verhalten der Batteriezelle verantwortlich [63] [61]. In diesem Bereich können Rückschlüsse auf den Ladezustand der Zelle gemacht werden. Dabei sind Messungen bis in den mHz Bereich notwendig, die aufgrund ihrer langen Periodendauer sehr zeitintensiv sind.

2.2.5. Anregung der Impedanzspektroskopie

Die Anregung des zu vermessenden Systems kann über verschiedenste Arten erfolgen. Der einfachste Fall ist die monofrequente Anregung, also der Anregung mit einer einzelnen Frequenz. In der Literatur werden aber auch andere Anregungsarten vorgestellt. So gibt es auch die Multi-Frequenz-Anregung, bei der das System mit mehreren Frequenzen gleichzeitig angeregt wird oder auch die Anregung mittels eines Frequenzsweeps. In der vorliegenden Arbeit wird aber ausschließlich mit einer monofrequenten Anregung gearbeitet, weshalb die anderen Anregearten hier nicht weiter betrachtet werden.

Bei der monofrequenten Anregung lassen sich grundsätzlich zwei verschiedene Arten unterscheiden. Dies ist zum einen die Anregung mit Hilfe eines Spannungssignals und zum anderen die Anregung mittels eines Stromsignals. Aus der Elektrochemie sind diese beiden Arten der Anregung als Untersuchungsmethode bereits bekannt. Dabei wird die Anregung mittels Spannung als potentiostatisches Untersuchungsverfahren und die Anregung mittels Strom als galvanostatisches Untersuchungsverfahren bezeichnet.

Die Messungen in dieser Arbeit sollen mit dem galvanostatischen Verfahren erfolgen. Ein großer Vorteil dieser Anregung ist, dass damit sehr einfach ein Arbeitspunkt für die Messung eingestellt werden kann. Dazu wird ein Gleichstrom angelegt, der von einem Wechselstrom überlagert wird. Die Einstellung eines Arbeitspunktes ist für die Vermessung der Impedanzspektroskopie wichtig, da es sonst zu Fehlmessungen kommen kann. Dies wird im Laufe dieser Arbeit nochmals näher betrachtet.

2.3. Batteriemodelle

Die Modellierung einer Batteriezelle ist sehr komplex. In der Literatur existieren viele Modellansätze, die das Verhalten einer Batteriezelle wiedergeben. Die verschiedenen Modellarten können dabei in drei Hauptkategorien eingeteilt werden.
Die erste Modellart ist das physikalisch-chemische Batteriemodell. In [60] werden diese physikalisch-chemischen Batteriemodelle als Modelle beschrieben, die auf mikroskopischer Ebene ablaufen, d.h. es werden reale Prozesse in der Zelle nachgebildet und versucht, das tatsächliche Zellverhalten zu simulieren. Man spricht dabei auch von Bottom-Up-Modellen. Eine weitere Methode ist das Modellieren mit elektrischen Ersatzschaltmodellen. Mittels dieser Ersatzschaltungen lässt sich das dynamische Verhalten einer Batteriezelle sehr gut annähern. Da diese Modelle aus elektronischen Bauteilen bestehen, sind diese weniger rechenaufwendig und somit gut für Simulationen geeignet.
Unter die dritte Modellkategorie lassen sich alle weiteren Modelle zusammenfassen. Diese sind zumeist rein mathematische Modelle.

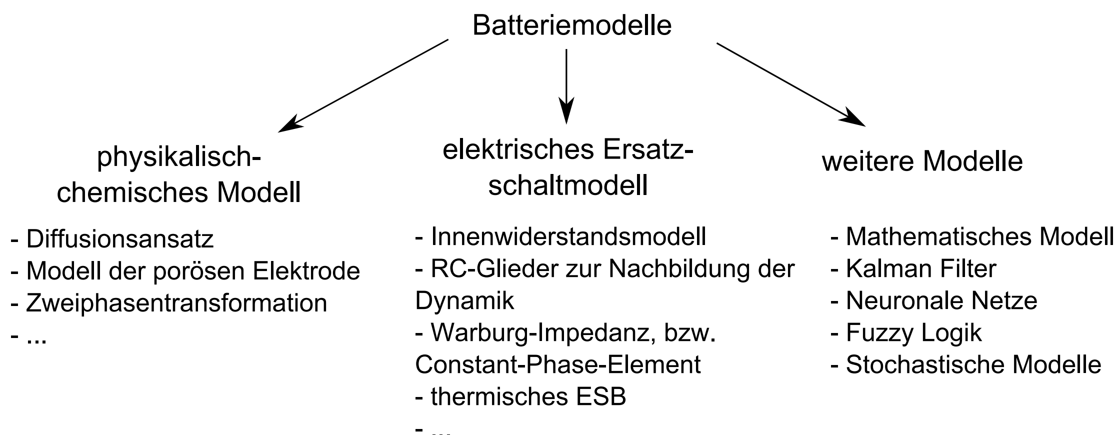


Abbildung 2.23.: Verschiedene Batteriemodelle (nach [16])

Da die Nachbildung einer vollständigen Batteriezelle mit all ihren Eigenschaften extrem komplex und rechenaufwendig wäre, muss das Modell nach dem Einsatzzweck ausgewählt werden. Es hat sich gezeigt, dass durch die Modellbildung mittels elektrischer Schaltungen die Charakteristik von Batteriezellen recht gut angenähert werden kann. Der Vorteil dabei ist, dass das komplexe Verhalten teilweise durch einfache Modelle mit relativ wenig Bauelementen näherungsweise abgebildet werden kann.
Bei der Modellierung einer Batteriezelle ist zwischen der Genauigkeit und dem Rechenaufwand zu unterscheiden. Soll die Genauigkeit erhöht werden, steigt gleichzeitig auch die Rechenzeit für das Modell. Für die Modellierung des dynamischen Verhaltens genügen meist wenige elektronische Bauelemente. Dabei löst man sich von der ursprünglichen Zellchemie und versucht, die Effekte auf diskrete Bauelemente zu übertragen. Das Thevenin-Theorem

[39] besagt, dass das Verhalten komplizierter linearer Netzwerke auch durch einfache Zweipole mit wenigen diskreten Elementen zurückzuführen ist [60]. Eingesetzt werden dabei üblicherweise Widerstände, Kapazitäten, Induktivitäten sowie Strom- und Spannungsquellen. Im Folgenden wird das Batteriemodell nach Randles[4] näher betrachtet. Dieses Modell wurde bereits in Abschnitt 2.1.2 als Ansatz für den elektrochemischen Aufbau verwendet.

2.3.1. Batteriemodell nach Randles

Randles unternahm 1947 theoretische Untersuchungen des Stromdurchgangs durch eine Elektrode [58]. Dabei entstand ein lineares Batteriemodell. Dieses Modell beschreibt schon die wesentlichen Merkmale, die für die Interpretation der elektrochemischen Impedanzspektroskopie wichtig sind.
Abbildung 2.24 zeigt hier das einfache Batteriemodell nach Randles. Dieses besteht aus einem parallel geschalteten RC-Glied mit einem in Reihe geschalteten Widerstand.

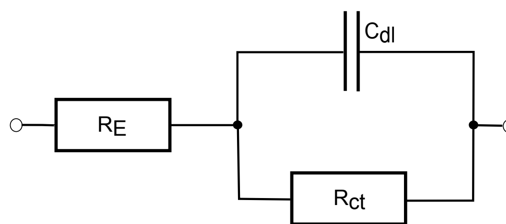


Abbildung 2.24.: Einfaches Batteriemodell nach Randles [58]

Dabei repräsentiert der vorgeschaltete Widerstand R_E den ohmschen Widerstand des Elektrolyten, den Übergang von den Elektroden zum Elektrolyt und den Widerstand zwischen Stromsammler und Aktivmaterial [68].

R_{ct} ist der nichtlineare Durchtrittswiderstand zwischen Elektrode und Elektrolyt. Zusammen mit der Kapazität C_{dl}, die die Doppelschichtkapazität repräsentiert, bilden diese das RC-Glied im Batteriemodell.

[4]John Edward Brough Randles, Elektrochemiker

Die Impedanz des RC-Glieds errechnet sich wie folgt

$$\frac{1}{Z_{RC}} = \frac{1}{R_{ct}} + j\omega C_{dl} \tag{2.23}$$

$$\frac{1}{Z_{RC}} = \frac{1 + j\omega R_{ct} C_{dl}}{R_{ct}} \tag{2.24}$$

Die Gleichung 2.23 wird nun konjugiert komplex erweitert, um diese in ihren Real- und Imaginärteil aufspalten zu können.

$$\frac{1}{Z_{RC}} = \frac{(1 + j\omega R_{ct} C_{dl})\,(1 - j\omega R_{ct} C_{dl})}{R_{ct}\,(1 - j\omega R_{ct} C_{dl})} \tag{2.25}$$

$$\frac{1}{Z_{RC}} = \frac{1 + \omega^2 R_{ct}^2 C_{dl}^2}{R_{ct} - j\omega R_{ct}^2 C_{dl}} \tag{2.26}$$

$$Z_{RC} = \frac{R_{ct} - j\omega R_{ct}^2 C_{dl}}{1 + \omega^2 R_{ct}^2 C_{dl}^2} = \frac{R_{ct}}{1 + \omega^2 R_{ct}^2 C_{dl}^2} - j \cdot \frac{\omega R_{ct}^2 C_{dl}}{1 + \omega^2 R_{ct}^2 C_{dl}^2} \tag{2.27}$$

Die Gesamtimpedanz des einfachen Batteriemodells ergibt sich dann zu

$$Z_{ges} = R_E + Z_{RC} \tag{2.28}$$

$$Z_{ges} = R_E + \frac{R_{ct}}{1 + \omega^2 R_{ct}^2 C_{dl}^2} - j \cdot \frac{\omega R_{ct}^2 C_{dl}}{1 + \omega^2 R_{ct}^2 C_{dl}^2} \tag{2.29}$$

Die in Abbildung 2.25 zu sehende Impedanz Z des einfachen Batteriemodells nach Randles ist hier beispielhaft bei einer beliebigen Kreisfrequenz ω und ihrem Phasenwinkels α aufgezeichnet. Beim Abfahren des gesamten Frequenzbereichs entwickelt sich das Nyquist-Diagramm mit der charakteristischen RC-Ortskurve.

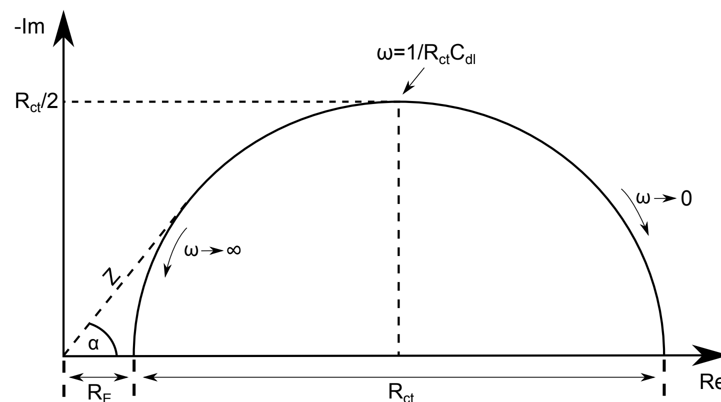


Abbildung 2.25.: Nyquist-Diagramm der Elektrodenimpedanz (nach [24])

Mit der Abbildung 2.25 und der Gleichung 2.29 können erste Abschätzungen für die Parameter der Widerstände R_E und R_{ct} für das einfache Batteriemodell erfolgen. In Abbildung 2.26 ist ein Vergleich zwischen einer realen Impedanzmessung und der Annäherung durch das einfache Batteriemodell zu sehen.

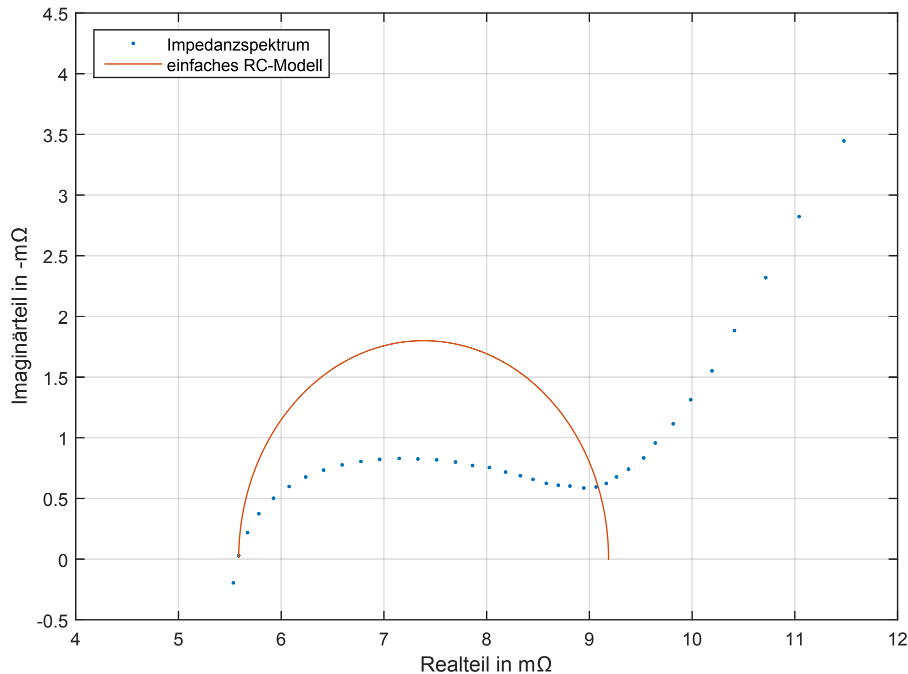


Abbildung 2.26.: Erste Abschätzung der Parameter für das Batteriemodell aus einem beispielhafen Impedanzspektrum

Die Werte für die Widerstände R_E und R_{ct} können aus der realen Impedanzmessung einfach abgeschätzt werden. Bestimmt wurden die Werte hier zu $R_E = 5.58\,\Omega$ und $R_{ct} = 3.6\,m\Omega$. Zu erkennen ist aber auch, dass das einfache Batteriemodell nur eine erste Annäherung an das Impdanzspektrum geben kann. Eine Näherung ist lediglich am Durchtrittspunkt an der Ordinate und dem Halbkreis, der den Ladungsaustausch zwischen den Elektroden und dem Elektrolyt beschreibt, möglich. Eine Annäherung an den Diffusionsast ist mit dem einfachen Batteriemodell nicht möglich. Für eine genauere Annäherung an den Diffusionsast ist eine Erweiterung des Batteriemodells um ein RC-Glied notwendig.

Erweitertes Batteriemodell

Durch das Hinzufügen eines weiteren RC-Glieds, lassen sich die dynamischen Vorgänge der zu modellierenden Batteriezelle näher beschreiben. So lässt sich die Annäherung an den Diffusionsast durch ein weiteres RC-Glied erreichen.

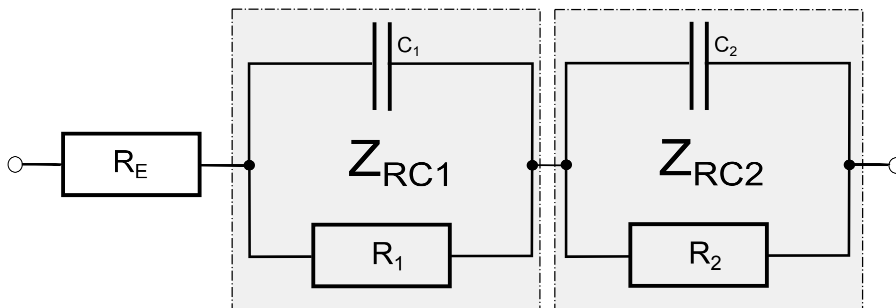


Abbildung 2.27.: Erweitertes Batteriemodell nach Randles [58]

Um das Verhalten der Schaltung bezüglich ihrer Impedanz zu untersuchen, wird diese auch wieder in ihre Real- und Imaginärteile aufgespalten. Die Schaltung besteht dabei aus zwei in Reihe geschalteten Impedanzen und dem reellen Widerstand R_E.
Für die Impedanz der RC-Glieder kann Gleichung 2.27 verwendet werden.

$$Z_{RC1} = \frac{R_1 - j\omega R_1^2 C_1}{1 + \omega^2 R_1^2 C_1^2} = \frac{R_1}{1 + \omega^2 R_1^2 C_1^2} - j \cdot \frac{\omega R_1^2 C_1}{1 + \omega^2 R_1^2 C_1^2} \tag{2.30}$$

$$Z_{RC2} = \frac{R_2 - j\omega R_2^2 C_2}{1 + \omega^2 R_2^2 C_2^2} = \frac{R_2}{1 + \omega^2 R_2^2 C_2^2} - j \cdot \frac{\omega R_2^2 C_2}{1 + \omega^2 R_2^2 C_2^2} \tag{2.31}$$

Die Gesamtimpedanz des erweiterten Batteriemodells ergibt sich zu

$$Z_{ges} = R_E + Z_1 + Z_2 \tag{2.32}$$

$$Z_{ges} = R_E + \frac{R_1}{1 + \omega^2 R_1^2 C_1^2} + \frac{R_2}{1 + \omega^2 R_2^2 C_2^2} - j \left(\frac{\omega R_1^2 C_1}{1 + \omega^2 R_1^2 C_1^2} + \frac{\omega R_2^2 C_2}{1 + \omega^2 R_2^2 C_2^2} \right) \tag{2.33}$$

Es ergeben sich im Nyquist-Diagramm zwei Halbkreise, die ineinander übergehen. Dadurch erhält man die weitere Annäherung an den Diffusionsast, die in Abbildung 2.28 zu sehen ist.

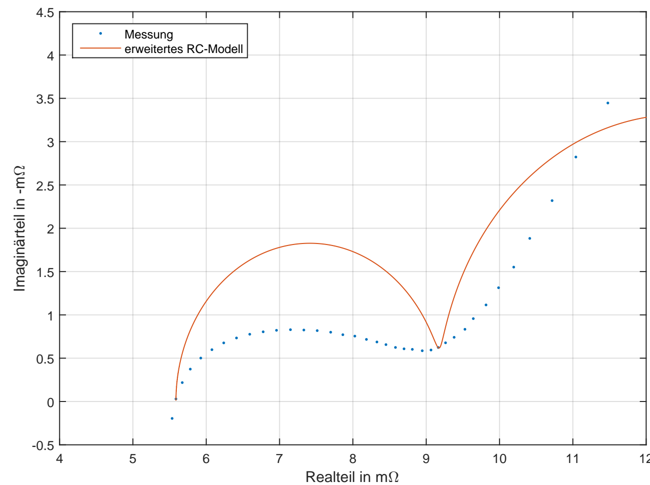


Abbildung 2.28.: Nyquist-Diagramm des erweiterten Randles Batteriemodells

Wie schon am Anfang des Abschnitts erklärt, ist bei der Modellierung eines Batteriemodells immer zwischen der Genauigkeit und des Rechenaufwands zu unterscheiden. Dadurch ist eine genauere Annäherung an das reale Impedanzverhalten mit einer steigenden Anzahl der RC-Glieder zu erreichen. Dies bedeutet aber auch ein rechnerisch aufwendigeres Modell. In [16] werden verschiedene Batteriemodelle mit mehreren RC-Gliedern vorgestellt. In Abbildung 2.29 ist der Unterschied zwischen einem Batteriemodell mit zwei RC-Gliedern und dem Modell mit vier RC-Gliedern dargestellt. Deutlich ist dabei die bessere Annäherung an das Impedanzspektrum zu erkennen.

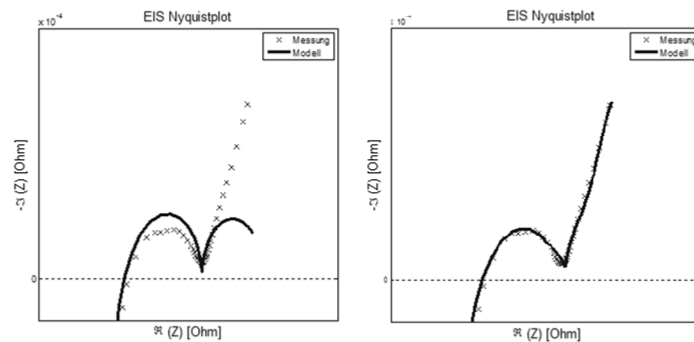


Abbildung 2.29.: Links: Batteriemodell mit zwei RC-Gliedern; Rechts: Batteriemodell mit vier RC-Gliedern (ent. aus [16])

Allerdings stellt sich die Parametrisierung der einzelnen RC-Glieder als sehr aufwendig dar. In [16] sowie in [28] werden Verfahren vorgestellt, die sich eigens mit der Parametrisierung der Batteriemodelle beschäftigt.

2.3.2. Optimiertes Batteriemodell

Da die Parametrisierung für die Annäherung an die Impedanzspektroskopie mittels RC-Gliedern sehr aufwendig ist, wird hier noch ein optimiertes Batteriemodell vorgestellt, bei dem auf die eine Bindung an reale Bauelemente verzichtet wird. Dabei handelt es sich um die, aus dem einfachen Batteriemodell errechnete Gleichung 2.29 und der in Abschnitt 2.1.2 vorgestellten Beschreibung der Warburg-Impedanz.

$$Z_{ges} = R_E + \frac{R_{ct}}{1+\omega^2 R_{ct}^2 C_{dl}^2} + \sigma\left(\omega\right)^{-1/2} - j\left(\frac{\omega R_{ct}^2 C_{dl}}{1+\omega^2 R_{ct}^2 C_{dl}^2} + \sigma\left(\omega\right)^{-1/2}\right) \tag{2.34}$$

Mit Hilfe dieses optimierten Batteriemodells lässt sich eine sehr gute Näherung an die reale Messung der Impedanzspektroskopie erreichen.

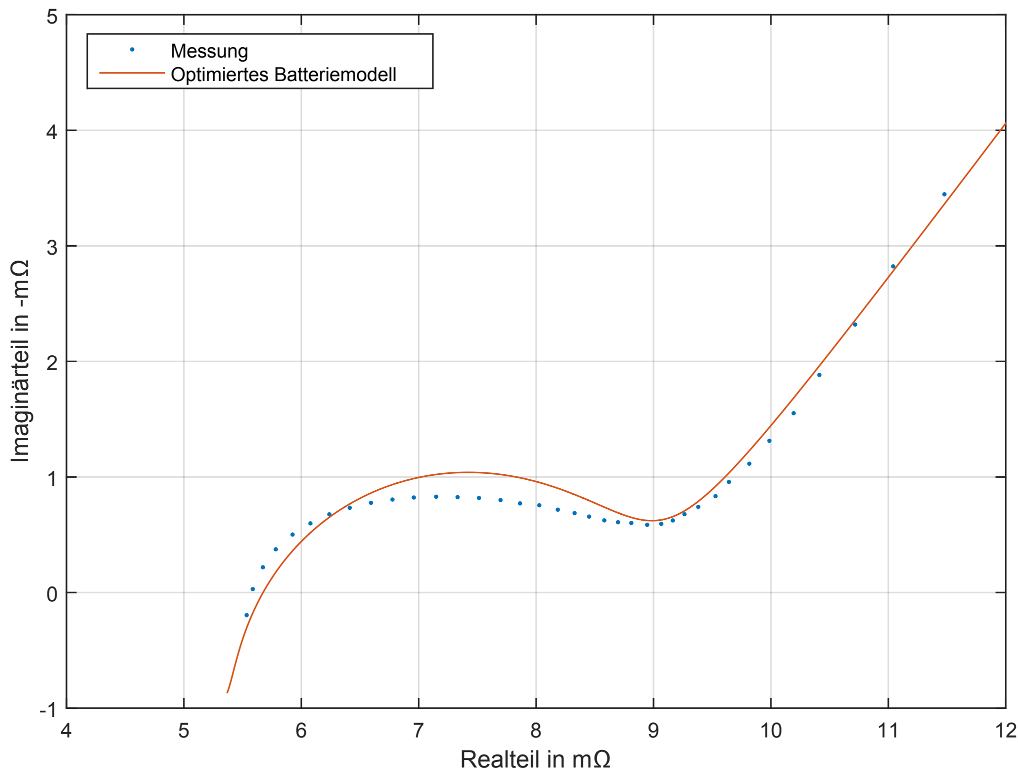


Abbildung 2.30.: Vergleich zwischen einem gemessenen Impedanzspektrum und dem vorgestellten optimiertes Batteriemodel

Anhand dieses optimierten Batteriemodells lassen sich nun Untersuchungen durchführen, wie sich Alterungseffekte der Batteriezelle auf das Impedanzspektrum auswirken.

2.3.3. Alterung am Batteriemodell

In diesem Abschnitt soll untersucht werden, wie sich das Impedanzspektrum verändert, wenn die Batteriezelle altert. Dies kann nun mit der ermittelten Gleichung 2.34 leicht untersucht werden. In [68] wird angegeben, dass sich bei der Alterung von Batteriezellen hauptsächlich die Leitfähigkeit des Aktivmaterials verschlechtert. Zudem nimmt die Speicherfähigkeit der Doppelschichtkapazität ab.
Dies bedeutet, dass sich im Batteriemodell die folgenden Bauteile verändern:

Tabelle 2.2.: Übersicht der Bauteiländerungen bei Alterung

Effekt durch Alterung	Auswirkung auf	Werteveränderung
Leitfähigkeit Aktivmaterial sinkt	R_E	Widerstand steigt
Übergang zw. Elektrolyten und Elektrode steigt	R_{ct}	Widerstand steigt
Doppelschichtkapazität sinkt	C_{dl}	Kapazität sinkt

Die obenstehende Tabelle gibt nur ein kleinen Teil der tatsächlichen Effekte der Batteriealterung wieder. In [26] werden eine großen Anzahl von verschiedenen Effekten vorgestellt, die zu einer Alterung der Batteriezelle führen. Die hier vorgestellten Effekte haben allerdings den größten Einfluss und sind für eine Untersuchung des Impedanzspektrums bei der Batteriealterung ausreichend.
Untersuchungen aus [60] haben ergeben, dass sich für den Widerstand R_E eine Änderung von ca. 10 % und für R_{ct} eine Änderung von ca. 12 % ergeben. Die Doppelschichtkapazität C_{dl} verändert mit einem Kapazitätsverlust von ca. 11 %. Diese Werte wurden nach 3.000 Zyklen mit einer Stromstärke von 5C, einer $LiFePO_4$ Zelle entnommen. Angaben, bei welcher Temperatur die Zyklen gefahren wurden, wurden dabei nicht gemacht.

Diese Ergebnisse können nun mit Gleichung 2.34 zu den Nominalwerten verglichen werden. Abbildung 2.31 zeigt die Änderungen des Impedanzspektrums, welches mit Hilfe des optimierten Batteriemodells ermittelt wurde.

Zu sehen ist, dass sich mit den veränderten Werten die komplette Impedanzkurve nach rechts verschiebt. Dies resultiert aus den angestiegenen Widerständen R_E und R_{ct}. Am Durchtrittspunkt ist die 10 %ige Widerstandsänderung von R_E gut zu erkennen. Die Veränderung von R_{ct} und C_{dl} wirken sich auf den Halbkreis aus. Ein direktes Ablesen der Bauteilwerte ist hier allerdings nicht ohne Weiteres möglich, da hier bereits der Einfluss der Warburg-Impedanz vorliegt.

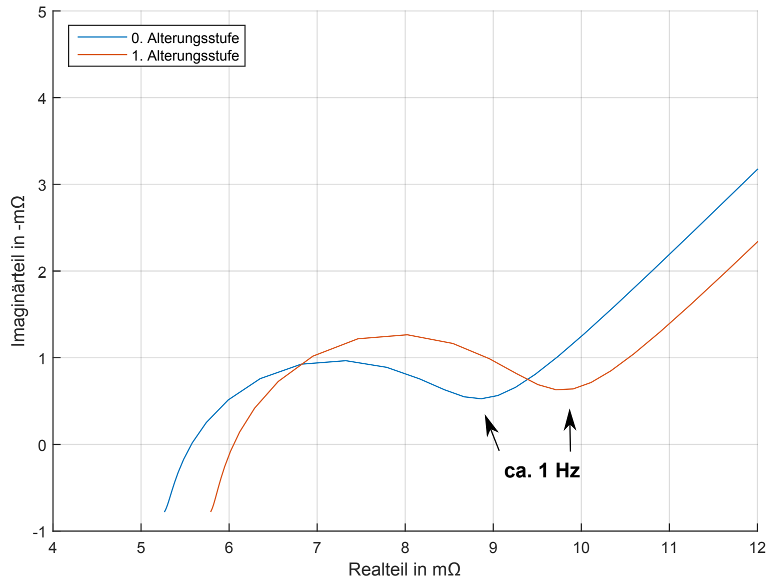


Abbildung 2.31.: Alterung am Batteriemodell

Ein möglicher Indikator für eine fortgeschrittene Alterung der Batteriezelle kann der Realteil des Wendepunkts zwischen RC-Halbkreis und der Geraden der Warburg-Impedanz sein. An diesem Punkt unterscheiden sich die beiden Alterungsstufen der Batteriezelle am meisten. In der Literatur wird für diesen Punkt meist eine Anregefrequenz von ca. 1 Hz angegeben. Diese Ergebnisse müssen später in der Konzeption des Messsystems beachtet werden, um den Alterungszustand mit Hilfe der Batteriesensoren zu bestimmen.

2.4. Reale Batteriealterung

Um das zuvor erstellte Batteriealterungsmodell zu verifizieren und die Auswirkung der Alterung an realen Batteriezellen zu ermitteln, wurde eine $LiFePO_4$-Batteriezelle künstlich gealtert und in regelmäßigen Abständen vermessen. Dabei konnte das theoretische Batteriealterungsmodell bestätigt werden. Die Abbildung 2.32 zeigt diese künstliche Alterung. Es ist jeweils die initiale Messung und die letzte Messung nach sieben Wochen künstlicher Alterung zu sehen. Sehr gut erkennbar ist, dass wie bei dem Batteriealterungsmodell, die komplette Impedanzkurve nach rechts verschoben wird. Ein besonders großer Unterschied im Realteil ist bei ca. 1 Hz zu sehen. Es lässt sich also auch dieser Punkt als guter Alterungsindikator bestätigen.

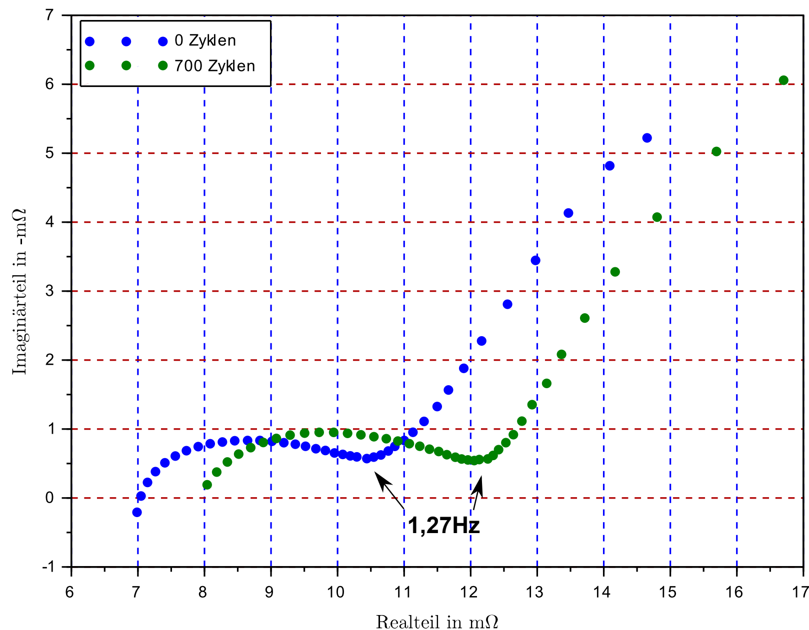


Abbildung 2.32.: Künstliche Alterung einer $LiFePO_4$ Zelle bei zwei verschiedenen Alterungszuständen

Es zeigt sich, dass sich mit dem zuvor entwickelten theoretischen Batteriemodell sehr gut das reale Verhalten des Impedanzspektrums beschreiben lässt. Dieses Batteriemodell kann nun genutzt werden, um die Spezifikationen einer möglichen Neukonzeptionierung des Messsystems vorzunehmen.

Ausführlich wird die durchgeführte künstliche Batteriealterung im Anhang D beschrieben.

3. Analyse und Neukonzeption des Messsystems

In diesem Kapitel wird eine Analyse des bestehenden Hard- und Softwaresystems durchgeführt. Dabei muss untersucht werden, inwieweit die aktuell vorhandene Hardware für die Untersuchung der funksynchronisierten Impedanzspektroskopie geeignet ist. Dazu müssen die Mindestanforderungen der Impedanzspektroskopie erarbeitet und mit dem vorhandenen Hard- und Softwaresystem verglichen werden. Daraus sollen dann Lösungskonzepte und mögliche Hard- und Softwareänderungen am bestehenden Messsystem entstehen.

3.1. Aktueller Stand des Messsystems

Der aktuelle Stand des Messsystems besteht aus den Zellsensoren der Klasse 3 und einem Batteriesteuergerät, welches in Abbildung Abbildung 3.1 zu sehen ist.

Abbildung 3.1.: Batteriesteuergerät mit CC1101-Transceiver Platine und Zellsensor der Klasse 3 mit hochauflösender AD-Wandler Erweiterungsplatine

Deren Hardware wird im Folgenden kurz vorgestellt um zu prüfen, ob diese die messtechnischen Anforderungen an die funksynchronisierte Impedanzspektroskopie erfüllen kann. Für weitere Einzelheiten der Hardware sei an dieser Stelle auf die bisherigen Arbeiten der Arbeitsgruppe verwiesen, die sich mit der Entwicklung der Zellsensoren und des Batteriesteuergeräts beschäftigt haben [48],[62],[14].

3.1.1. Zellsensor

Innerhalb der Arbeitsgruppe wurden schon mehrere Zellsensoren unterschiedlicher Klassen entwickelt. Alle haben dabei die Aufgabe, den Zellzustand zu ermitteln. Dabei wurden verschiedene Verfahren der Messungen angewendet. Auf die Klassen 1 und 2 sei an dieser Stelle nur hingewiesen, diese werden in dieser Arbeit nicht weiter diskutiert. Der in dieser Arbeit verwendete Zellsensor ist aus der Klasse 3, welche hier kurz vorgestellt wird.
Die Hauptmerkmale eines Zellsensors der Klasse 3 sind zum einen eine bidirektionale drahtlose Kommunikation zwischen Zellsensor und Batteriesteuergerät. Diese Kommunikation wird mittels eines aktiven Transceivers im 434 MHz ISM-Band realisiert. Zum anderen besitzt dieser über einen sehr energiesparenden Schlafmodus, welcher durch ein Funksignal des Batteriesteuergeräts unterbrochen werden kann.

Die messtechnischen Aufgaben des Zellsensors sind recht einfach und übersichtlich. Seine Hauptaufgaben sind die Messung der Spannung sowie der Temperatur der Zelle und der Balancierung der Zellspannung. Dabei handelt es sich um eine passive Balancierung, welche durch das Hinzuschalten von Widerständen realisiert wird. Dadurch soll die Zellspannung an die Spannung anderer Zellen innerhalb eines Batterieblocks angepasst werden, um diese vor einer Über- bzw. Tiefenentladung zu schützen. In der Arbeit [62] wurde für die Sensoren der Klasse 3 eine neue Messmethode entwickelt, mit der zeitlich hoch synchronisierte Spannungsmessungen möglich wurden. Mit der sog. Burstmessung sind Messungen von bis zu mehreren kSPS möglich. Diese zeitlich hoch synchronisierte Messmethode bildete die Grundlage für die Messung der elektrochemischen Impedanzspektroskopie mittels der Zellsensoren. Implementiert und getestet wurde die Impedanzmessung mit den Zellsensoren der Klasse 3 erstmalig in der Arbeit [48]. Darin wurde nachgewiesen, dass es möglich ist, mit der entwickelten Burstmessung Phasenverschiebungen zwischen Spannung und Strom zu messen und daraus die komplexen Impedanzen zu berechnen. Allerdings konnten noch keine Messungen im Impedanzbereich einer Batteriezelle durchgeführt werden.
An dieser Stelle greift nun diese Arbeit ein, um die Vermessung der Impedanzspektroskopie an den Zellen mit den Sensoren der Klasse 3 zu realisieren. Bisher beschäftigten sich drei verschiedene Arbeiten mit der Entwicklung bzw. Weiterentwicklung des Zellsensors der Klasse 3. Im Folgenden wird der Zellsensor der letzten Entwicklungsstufe analysiert.

Hardwareanalyse

In der letzten Arbeit [48] entstand erstmalig ein Zellsensor der Klasse 3, der für die hochgenaue Messung der Zellspannung einen externen AD-Wandler nutzt. Dabei handelt es sich um einen 24-Bit Delta-Sigma Wandler (Texas Instruments ADS1291) der auf einem Erweiterungsmodul (Abbildung 3.2) angebracht wurde.

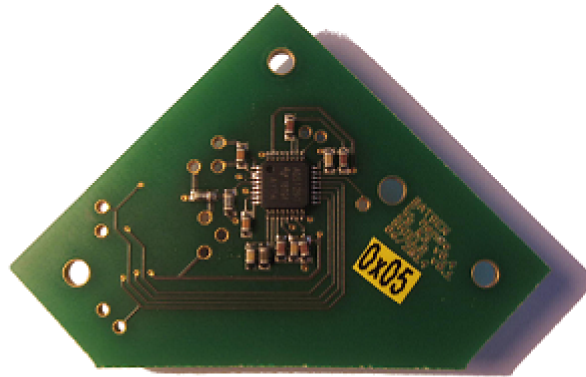


Abbildung 3.2.: AD-Wandler Erweiterungsmodul (ent. aus [48])

Der Zellsensor selbst wurde durch ein Redesign erneuert. Dabei kam ein neuer Mikrocontroller (Texas Instruments CC430F5137) zum Einsatz, der bereits einen aktiven Transceiver integriert hat. Somit konnte Platz auf dem Zellsensor eingespart werden, da weniger Bauteile eingesetzt werden mussten. Ein weiterer Vorteil bei der Verwendung des neuen Mikrocontrollers liegt bei seiner größeren Speicherkapazität. Diese hat sich im Vergleich zum vorhergehenden Controller von 2 kByte auf 4 kByte verdoppelt. Der Controller lässt sich bis zu einer max. Taktfrequenz von 20 MHz betreiben. Er kann somit um 4 MHz schneller getaktet werden als der vorher verwendete Mikrocontroller (Texas Instruments MSP430F235). Die übrige Hardware wurde bei diesem Redesign nicht verändert, wird im Folgenden aber kurz erklärt. Die Spannungsversorgung des Zellsensors wird über einen Step-Up/Down-Konverter (Texas Instruments TPS61201) auf 3,3 V gebracht. Dadurch wird die gesamte Elektronik des Sensors versorgt. Sämtliche Komponenten wurden somit auf 3,3 V ausgelegt. Des Weiteren besitzt der Zellsensor eine passive Wake-Up Schaltung. Mittels dieser Schaltung ist es möglich, den Zellsensor durch ein Funksignal aus einem energiesparenden Ruhezustand in den normalen Betriebsmodus zu bringen. Dies hat den Vorteil, dass der Zellsensor, wenn dieser keine Messungen durchführen soll, in einen sehr energiesparenden Ruhezustand versetzt werden kann. Nähere Details dazu sind in der Arbeit [14] zu finden. Da für die Wake-Up Schaltung sowie für den aktiven Transceiver dieselbe Antenne genutzt wird, kommt ein Antennenumschalter (Analog Devices ADG918) zum Einsatz, der den Antennenpfad umschalten kann. Die Temperaturüberwachung der Zelle erfolgt über einen, auf der Sensorplatine aufgebrachten Temperatursensor (TMP102). Dieser besitzt eine Warnfunktion, die automatisch bei der Überschreitung einer einstellbaren Temperatur ein Signal

an den Mikrocontroller schickt, der darauf reagieren kann. Die passive Ladungsbalancierung wird durch schaltbare Strompfade realisiert. Dabei können durch MOSFET-Schalter zwei Strompfade mit jeweils 20Ω Widerständen zu- bzw. abgeschaltet werden, über die überschüssige Ladung der Zelle abgeführt wird. Das Blockschaltbild in Abbildung 3.3 zeigt die eingesetzten Komponenten und deren Verbindungen auf dem Zellsensor der Klasse 3.

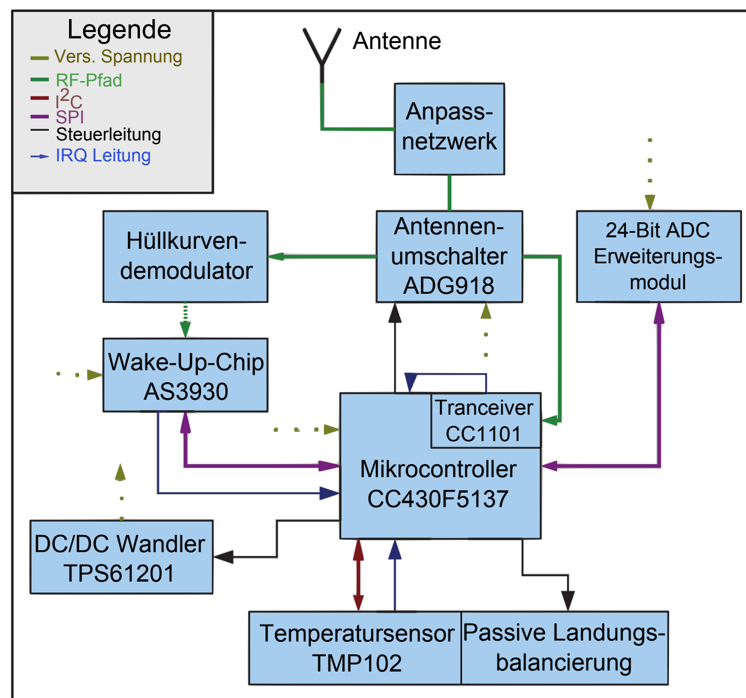


Abbildung 3.3.: Blockschaltbild des Zellsensors v0.4 und deren Verbindungen der Sensorklasse 3 (nach [48])

- Mikrocontroller Texas Instruments CC430F5137 mit integriertem Transceiver
- 24 Bit AD-Wandler Erweiterungsmodul Texas Instruments ADS1291
- 3,3 V Spannungsversorgung Texas Instruments TPS61201 Step Up/Down-Konverter
- Wake-Up Receiver AMS AS3930
- Antennenumschalter Analog Devices ADG918
- Passive Ladungsbalancierung mittels zwei schaltbarer Strompfade
- Temperaturmessung mit Warnfunktion Texas Instruments TMP102

Softwareanalyse

In der Softwareanalyse des Zellsensors werden die bisher möglichen Funktionen des Zellsensors betrachtet. Dieses sind die Spannungsaufnahme der Zellspannung, die Temperaturüberwachung der Zelle sowie die Balancierung der Zellladung. Diese Funktionen sind noch aus der Vorgängerarbeit unverändert geblieben, weshalb für nähere Details der Softwarefunktionen auf die Arbeit [62] bzw. [14] verwiesen wird. Nähere Beachtung findet hier die Möglichkeit der funksynchronisierten Messung der Zellspannung über den Sensor und der Speicherung der aufgenommenen Spannungswerten.

Die Spannungsmessungen für die funksynchronisierten Messungen finden mittels des auf dem Erweiterungsmodul befindlichen 24 Bit AD-Wandlers (Texas Instruments ADS1291) statt. In der Arbeit [48] wurde festgestellt, dass durch dieses Erweiterungsmodul eine maximale Messrate von bis 350 Hz realisiert werden kann, ohne die geforderte Messgenauigkeit von 10 μV zu unterschreiten. Bei dieser Messrate erreicht der AD-Wandler noch ein ENOB[1] von 18 Bit. Für die Minimierung des Speicherbedarfs wurde entschieden, die gemessenen Speicherwerte nicht in voller Auflösung abzuspeichern, sondern lediglich in einer Auflösung von 16 Bit. Dabei konnte noch ein zu messender Spannungsbereich von 2,8 V bis 3,9 V abgedeckt werden. Durch diese Reduzierung des Speicherbedarfs konnte eine Aufnahme von 1.900 Messwerten auf dem Zellsensor realisiert werden.

Zusammenfassung der Zellsensoranalyse

Der Zellsensor ist aktuell in der Lage, funksynchronisierte Messungen bis zu einer maximalen Messrate von 350 Hz durchzuführen. Dabei kann er im Spannungsbereich von 2,8 V bis 3,9 V eine maximale Anzahl von 1.900 Messwerten mit einer Auflösung von 16 Bit aufnehmen. Es konnte in der Arbeit [48] auch nachgewiesen werden, dass es möglich ist, über die funksynchronisierte Messmethode ein Impedanzspektrum zu messen. Messungen konnten dabei aber lediglich an einer Testimpedanz durchgeführt werden, die außerhalb des zu erwartenden Impedanzbereichs einer realen Batteriezelle lag. Ob es mit der funksynchronisierten Messmethode auch möglich ist, das Impedanzspektrum einer Batteriezelle zu vermessen, muss in dieser Arbeit untersucht werden.

[1] effektive Anzahl von Bits (engl. effective number of bits)

3.1.2. Batteriesteuergerät

Als Batteriesteuergerät für Zellsensoren der Klasse 3 wurde ursprünglich ein Evaluations-Board (Olimex MSP430-169STK) mit einem energiesparenden Controller des Typs MSP430F169 eingesetzt (Abbildung 3.4). Dieses verfügt über ein RS232 Interface, eine externe SPI-Schnittstelle sowie ein 16x2 LCD Display. Über die externe SPI-Schnittstelle wurde ein Transceiver-Board angesteuert, welches auf der Basis eines Texas Instruments CC1101 Transceiver die Kommunikationsschnittstelle mit den Zellsensoren herstellt. Im Projektverlauf hat sich allerdings herausgestellt, dass dieses Evaluations-Board für die Aufgaben des Batteriesteuergeräts nicht ausreichend ist.

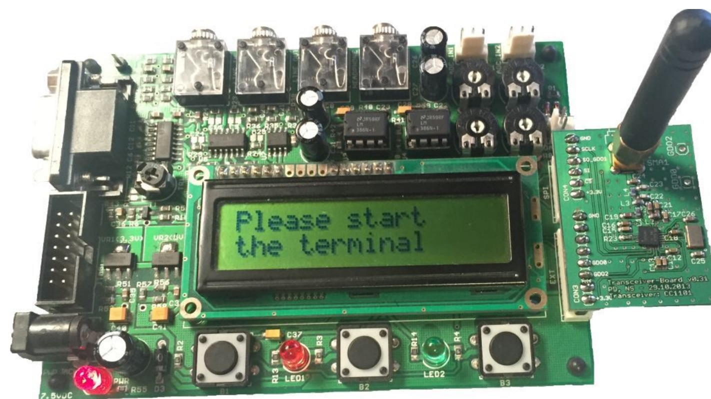


Abbildung 3.4.: Batteriesteuergerät für Zellsensoren der Klasse 3 auf Basis eines Evaluations-Board (Olimex MSP430-169STK)

So ist die Rechenleistung des mit 8 MHz getakteten Mikrocontrollers zu langsam um die Steuerung von mehreren Zellsensoren sowie der Signalverarbeitung der Daten zu übernehmen. Hauptnachteil ist aber die fehlende Möglichkeit zur Implementierung einer Strommessung, welche für die Messung der Impedanzspektroskopie dringend erforderlich ist. Grund hierfür ist, dass das Board keine weiteren Ein- bzw. Ausgänge mehr zu Verfügung stellt, an denen eine Strommessung realisiert werden könnte. Daher wurde entschieden, ein leistungsstärkeres Batteriesteuergerät zu entwickeln, welches über mehr Ein- bzw. Ausgänge verfügt. Da diese Neuentwicklung ebenfalls Bestandteil dieser Arbeit ist, wird auf eine nähere Hard- und Softwareanalyse des bisherigen Steuergeräts an dieser Stelle verzichtet. Für nähere Details des bisherigen Batteriesteuergerätes sei auf die Arbeiten [62] sowie [14] verwiesen.

3.2. Analyse der messtechnischen Anforderungen

In diesem Abschnitt sollen die messtechnischen Mindestanforderungen für die EIS-Messung ermittelt werden. Diese ergeben sich vor allem aus den gewünschten Anforderungen der Messgenauigkeit, des zu messenden Frequenz- oder auch des Impedanzbereichs. Um einen Überblick, der wichtigsten Parametern zu bekommen, werden zunächst kommerziell erhältliche EIS-Labormessgeräte auf ihr Leistungsspektrum hin untersucht. Dies soll einen ersten Aufschluss darüber geben, welche Parameter für die Messung der EIS wichtig sind und in welchem Bereich diese liegen.

3.2.1. Analyse kommerzieller EIS-Labormessgeräte

Für die Analyse kommerziell erhältlicher EIS-Labormessgeräte werden die beiden Geräte von FuelCon und Digatron auf ihre Parameter hin untersucht. Beide sind eigenständige Labormessgeräte mit eigener Anrege- und Messelektronik. In Tabelle 3.1 werden die wichtigsten Parameter, die für eine EIS-Messung nötig sind, aufgelistet.

Tabelle 3.1.: Übersicht kommerzieller EIS-Meter

	FuelCon	Digatron
Messfrequenzbereich	0,2 mHz - 100 kHz	1 mHz - 10 kHz
Impedanzbereich	5 $\mu\Omega$ - 15 Ω	300 $\mu\Omega$ - 3 Ω
Phasengenauigkeit	1 °	1 °
Impedanzgenauigkeit	± 1 %	± 1 %

Anhand dieser Tabelle kann ein erster Bereich definiert werden indem sich die Parameter für die EIS-Messung befinden sollen.

3.2.2. Messfrequenzbereich

Kommerzielle Geräte verfügen über einen sehr breiten Frequenzbereich. Typischerweise liegt der Frequenzbereich bei der Impedanzspektroskopie von Batteriezellen im Bereich von einigen kHz bis hin zu wenigen μHz [42]. Um eine Aussage über den SoH der Batteriezelle geben zu können, ist dieser komplette Frequenzbereich allerdings nicht nötig. Eine der wichtigsten Frequenzen für die Alterungsaussage ist der Bereich um 1 Hz. Dieser kann ein Anhaltspunkt für den SoH der Zelle sein. In der Abbildung 3.5 ist die Zellalterung aus Kapitel 2.4 nochmals dargestellt. Es sind die Impedanzspektren mit 0 Zyklen und nach 700

Zyklen bzw. nach sieben Wochen künstlicher Alterung zu sehen. Es sind auch exemplarisch drei verschiedene Signalfrequenzen aufgezeigt. Zu erkennen ist, dass sich das gesamte Impedanzspektrum mit zunehmender Alterung nach rechts verschiebt. Charakteristisch ist der Wendepunkt vom fallenden kapazitiven, zum steigenden kapazitiven Anteil bei der Signalfrequenz bei ca. 1 Hz.

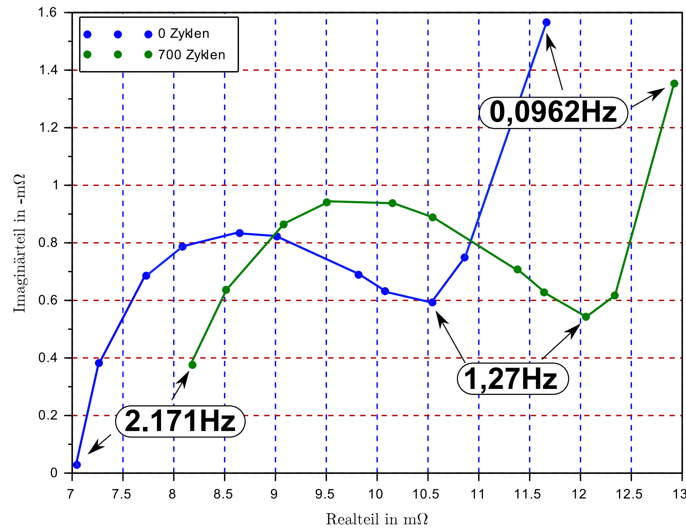


Abbildung 3.5.: Zellalterung - Einzelne Frequenzen

Geht man in den höheren Signal-Frequenzbereich, so erkennt man, dass bei hohen Frequenzen die kapazitiven Anteile weniger werden und die Impedanz reell wird. In diesem Frequenzbereich haben die elektrochemischen Eigenschaften der Batteriezelle wenig Einfluss. Beeinflusst wird dieser Bereich eher durch die eingesetzten Materialien und die Kontaktwiderstände. In den meisten Datenblättern von Batteriezellen wird deren Wechselstromwiderstand bei einer aufgeprägten Signalfrequenz von 1 kHz gemessen. Dadurch erhält man den Widerstand der Zelle, ohne den Einfluss der chemischen Vorgänge [68]. In [61] wird angegeben dass bei der Impedanzspektroskopie typischerweise bis 2 kHz gemessen wird.

Geht man in den Bereich unter 1 Hz, lässt sich eine Aussage über die Diffusionstransporte innerhalb der Zelle machen. Dieser Bereich erstreckt sich typischerweise bis zu einigen mHz. Durch den großen Zeitaufwand sind Messungen unterhalb von 10 mHz aber recht selten [61]. Ausreichend ist bereits eine Vermessung bis 100 mHz. Es kann nun festgehalten werden, dass für eine komplette Impedanzspektroskopie der Frequenzbereich von 100 mHz bis mindestens 2 kHz interessant ist.

3.2.3. Impedanzbereich

Die bisher gemessenen Impedanzen an der A123 $LiFePO_4$ Zelle hatten alle einen Realanteil im Bereich von ca. 7 mΩ bis ca. 17 mΩ. Der Imaginärteil liegt bei ca. 6 mΩ. Für das richtige Hardwarekonzept ist besonders der untere Innenwiderstandsbereich der Impedanzspektroskopie wichtig. Ein kleinerer Innenwiderstand bedeutet eine kleine Spannungsamplitude, die gemessen werden muss. Dementsprechend muss die Hardware auf die zu messende Impedanz ausgelegt werden. Der obere Wert des Realanteils ist für die Konzeption wenig von Bedeutung, da die zu messende Amplitude mit steigendem Widerstand auch ansteigt und dadurch einfacher zu messen ist.
Allerdings hängt der Innenwiderstand der Zelle stark von der Größe und Bauform ab. Um einen Überblick über die gängigen Innenwiderstände zu bekommen, werden in Tabelle 3.2 einige $LiFePO_4$ Batteriezellen hinsichtlich ihres Innenwiderstands verglichen. Dabei ist zu erkennen, dass der Innenwiderstand, abhängig von ihrer Kapazität und deren Bauform, stark variiert.

Tabelle 3.2.: Übersicht Innenwiderstand

	Innenwiderstand bei 1 kHz	Nennkapazität	Bauform
LiNANO Headway 40152SE	5 mΩ	15 Ah	Zylinder
A123 ANR26650M1-B	6 mΩ [1]	2500 mAh	Zylinder
ECC-LFPP 45Ah	1,2 mΩ [20]	45 Ah	Zylinder
A123 AHP14M1-A	1,3 mΩ	14 Ah	Pouch-Zelle

Die Werte der Innenwiderstände der ECC-LFPP 45Ah Zelle[2] sowie der A123 ANR26650M1-B konnten bereits durch Messungen bestätigt werden. Der kleinste Innenwiderstand liegt hier bei etwas über 1 mΩ. Somit wird als Untergrenze des kleinsten noch zu messenden Innenwiderstands mit 1 mΩ festgelegt.

[2]siehe dazu Anhang E

3.2.4. Spannungsbereich

Ein Spannungsbereich, in dem die Impedanzspektroskopie durchgeführt werden soll, wird bei den kommerzielen Geräten nicht angegeben. Dennoch ist dieser Wert für die Dimensionierung der Hardware wichtig. Primäres Ziel ist es, mit den Zellsensoren $LiFePO_4$ Batteriezellen zu vermessen. Diese besitzen einen mittleren Spannungsbereich von 3,4 V. Die Entladeschlussspannung beträgt dabei max. 2,0 V und die Ladeschlussspannung liegt typischerweise bei 3,6 V.
Als unteren max. Spannungsbereich können die 2,0 V der Entladeschlussspannung angenommen werden. Als oberer min. Spannungsbereich, der in jedem Fall noch gemessen werden soll, werden die 3,4 V des mittleren Spannungsbereichs gewählt.

3.2.5. Phasengenauigkeit

Die Phasengenauigkeit wird bei den kommerziellen Labormessgeräten zu 1 ° angegeben. Dies bedeutet, es lässt sich bereits eine Phasenverschiebung zwischen Strom- und Spannung von 1 ° detektieren. Bei niedrigen Frequenzen ist eine Detektierung zeitlich eher unkritisch, da die Perioden entsprechen lang sind und einer Verschiebung um 1 ° dementsprechend länger dauert. Bei schnelleren Frequenzen sind die Zeiten für eine 1 ° Verschiebung bereits im μs Bereich. So ist bei einem 100 Hz Signal der zeitliche Unterschied bei einer Verschiebung um 1 ° lediglich 27,7μs. Bei schnelleren Signalen verkürzt sich die Zeit entsprechend. Um diese Verschiebung durch das Batteriesteuergerät sowie den Zellsensor messen zu können, ist eine hohe Synchronität der Messwerte zwischen Strom- und Spannungsmessung nötig.

Ziel soll für die Implementierung der Impedanzspektroskopie auch eine Phasengenauigkeit von 1 ° sein. Ob diese Genauigkeit aber durch das Messsystem, aufgrund der nötigen Synchronität, erreicht werden kann muss erst noch überprüft werden.

3.3. Synchronisation des Messsystems

Die Synchronisierung der Messwerte ist eines der grundlegenden Anforderungen an das gesamte Messsystem. Dabei ist nicht nur die einzelne Synchronität zwischen Batteriesteuergerät und Zellsensor wichtig, sondern auch die Synchronität unter den einzelnen Sensoren, da jeder Sensor seinen Messwert zur gleichen Zeit aufnehmen muss. Es muss also zwischen zwei verschiedenen Synchronisationen unterschieden werden. Im Folgenden sollen diese beiden Synchronitäten untersucht und im Falle eines Fehlers deren Auswirkungen auf das Messergebnis hin betrachtet werden.

3.3.1. Synchronisation der Messwerte

In diesem Abschnitt wird die Synchronität zwischen Batteriesteuergerät und Zellsensor betrachtet. Dabei soll untersucht werden, wie sich ein Fehler in der Synchronisation auf das Messergebnis auswirkt. Anhand dieser Untersuchungen können die genauen Anforderungen an das Messsystem ermittelt werden. In einer ersten theoretischen Annäherung an das Verhalten wird eine Berechnung mit einem Strom und einer dazu synchronen Spannung durchgeführt. Dabei wird ein Strom mit 2,0 A und die dazu synchrone Spannung mit 2,0 V angenommen. Beide haben eine Frequenz von 1 Hz. Im Falle einer synchronen Messung von Strom- und Spannung ergibt sich eine Impedanz, berechnet durch das ohmsche Gesetz, von $\underline{Z} = 1\,\Omega + j0\,\Omega$.

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{2\,V \cdot (sin\,(0°) + j \cdot cos\,(0\,°))}{2\,A \cdot (sin\,(0\,°) + j \cdot cos\,(0\,°))} = 1\,\Omega + j0\,\Omega \tag{3.1}$$

Bei einer nicht synchronen Aufnahme verändert sich das Ergebnis der Impedanz. Es wird nun von einem Synchronisationsfehler von 125 ms ausgegangen. Dies bedeutet bei einer Frequenz von 1 Hz, eine Verschiebung des Spannungssignals um 45 °, da der Strom 125 ms vor dem Spannungssignal aufgenommen wird. Den daraus resultierenden zeitlichen Strom- und Spannungsverlauf zeigt Abbildung 3.6.

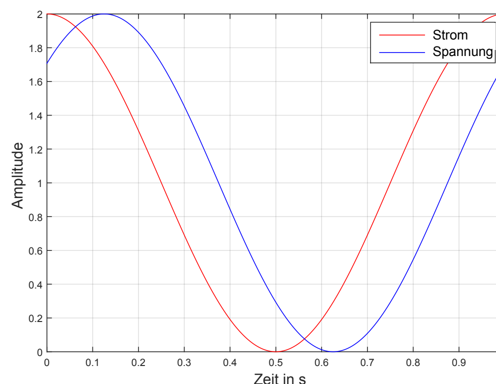


Abbildung 3.6.: Synchronisationsfehler zwischen Strom und Spannung um 125 ms

Durch die nicht synchrone Aufnahme von Strom und Spannung kommt es bei der Impedanz zu einer Veränderung, bedingt durch die vorhandene Phasenverschiebung von $45°$ der Spannung. Damit ergibt sich eine Impedanz von $\underline{Z} = 0.7071\,\Omega - j0.7071\,\Omega$

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{2V \cdot (sin\,(45°) + j \cdot cos\,(45°))}{2A \cdot (sin\,(0°) + j \cdot cos\,(0°))} = 0.7071\,\Omega - j0.7071\,\Omega \qquad (3.2)$$

Dieses Beispiel verdeutlicht, dass ein Synchronisationsfehler zwischen Strom und Spannung Einfluss auf die Impedanzmessung hat.
Nun ist zu untersuchen, in welchen Bereichen sich der mögliche Synchronisationsfehler im Messsystem bewegt und welche Auswirkungen dieser auf die gemessene Impedanz hat. Der Synchronisationsfehler setzt sich bei der Messung aus der Zeit zwischen der Stromaufnahme durch das Batteriesteuergerät und der Spannungsaufnahme durch den Zellsensor zusammen. In der Arbeit [62] wurde bereits festgestellt, dass sich zwischen Aussendung des Messimpulses durch das Batteriesteuergerät und dem Empfang durch den Zellsensor eine Laufzeit von 18,4 μs befindet. Sobald das Taktsignal am Zellsensor ankommt, löst dieser die AD-Wandler Messung für die Spannung aus. Dieser ist so eingestellt, dass von der AD-Wandler Auslösung bis zur Aufnahme 53,8 μs vergehen. Diese Zeit setzt sich zusammen aus 256 Takten (Sample-time) und 13 Takten (Convert-time). Durch die 5 MHz Taktung des AD-Wandlers ergeben sich die genannten 53,8 μs.

Zusammen mit der Laufzeit des Taktsignals von 18,4 μs ergibt sich ein Synchronisationsfehler von insgesamt 72.2 μs zwischen Strom- und Spannungsaufnahme.

$$18{,}4\,\mu\text{s}\,(\,\text{Laufzeit}\,) + 53{,}8\,\mu\text{s}\,(\,\text{AD-Wandler Messzeit}\,) = 72{,}2\,\mu\text{s} \qquad (3.3)$$

In erster Näherung ist also von einem Synchronisationsfehler zwischen Strom- und Spannungsaufnahme von 72,2 μs auszugehen. Dieser Synchronisationsfehler wird sich bei niedrigen Frequenzen wenig auf das Messergebnis auswirken. Bei einem 1 Hz Signal bedeutet der Fehler von 72,2 μs eine Verschiebung zwischen Spannung und Strom von 0,025848 °. Dabei entsteht einen Fehler von lediglich 0,05 % im Imaginärteil.

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{2\,V \cdot (sin\,(0,025848\,°) + j \cdot cos\,(0,025848\,°))}{2\,A \cdot (sin\,(0°) + j \cdot cos\,(0°))} = 1.0000 - j0.0005\Omega \qquad (3.4)$$

Bei schnelleren Signalen wirkt sich der Fehler, aufgrund der kürzeren Periode, deutlich stärker aus. Bei einem 2 kHz Signal beträgt die durch den Synchronisationsfehler verursachte Verschiebung bereits 51,984 °. Dabei entsteht neben dem Fehler im Imaginärteil auch ein Fehler im Realteil von 38,41 %. Im Imaginärteil beträgt die Abweichung zum synchronen Messwert sogar 78,78 %.

$$\underline{Z} = \frac{\underline{U}}{\underline{I}} = \frac{2\,V \cdot (sin\,(51,984\,°) + j \cdot cos\,(51,984\,°))}{2\,A \cdot (sin\,(0\,°) + j \cdot cos\,(0\,°))} = 0.6159 - j0.7878\,\Omega \qquad (3.5)$$

In Abbildung 3.7 ist zu erkennen, wie sich der Synchronisationsfehler bei einem 2 kHz Signal verhält.

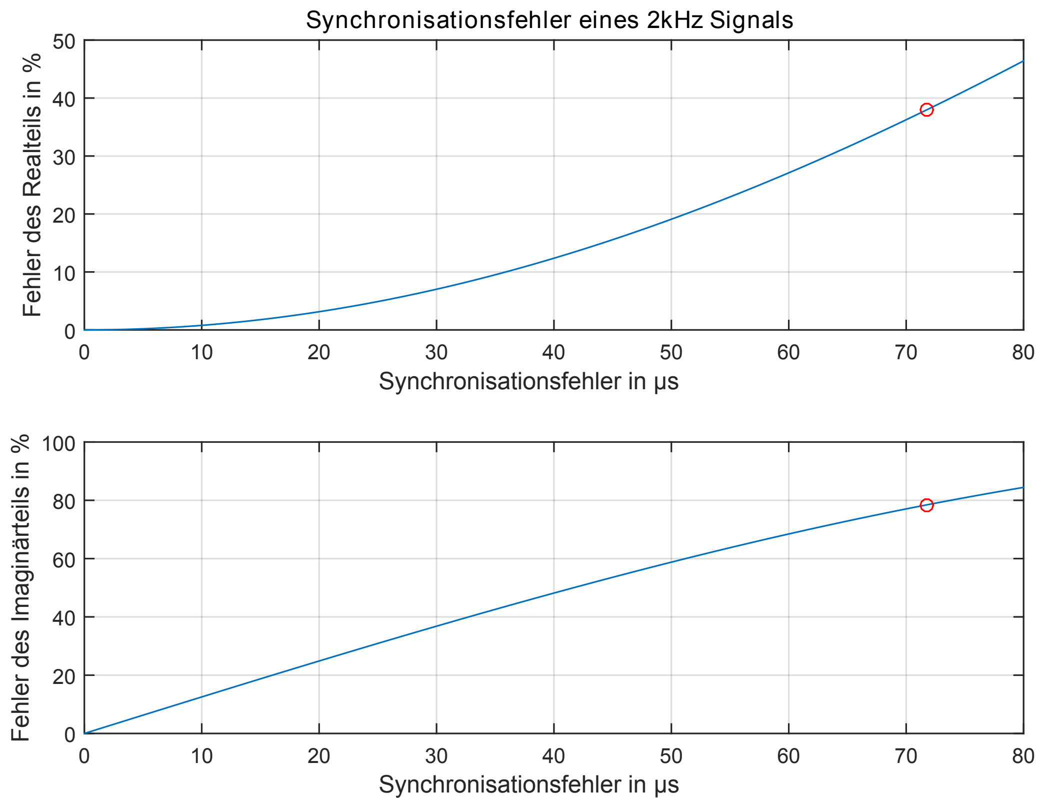


Abbildung 3.7.: Erste Näherung des Synchronisationsfehlers bei 2 kHz

Diese Rechnung zeigt, dass für die Realisierung der funksynchronisierten Impedanzspektroskopie die Synchronisierung der Strom- und Spannungsmessung sehr wichtig ist. Um eine Phasengenauigkeit von 1 ° zu erreichen, ist bei einem Signal von 2 kHz eine Mindestsynchronität von 1,388 μs zu erreichen.

$$\frac{\text{Periodendauer}}{360\,^\circ} = \frac{500\,\mu s}{360^\circ} = 1,388\,\mu s/^\circ \tag{3.6}$$

Abbildung 3.8 zeigt den Einfluss des Synchronisationsfehlers bei unterschiedlichen Synchronisationsfehlerzeiten. Daraus folgt, dass sich der Fehler bei 1,388 μs kaum mehr auf den Realteil der Impedanz auswirkt. Der Fehler des Imaginärteils liegt bei 2 kHz lediglich noch bei 1,741 %.

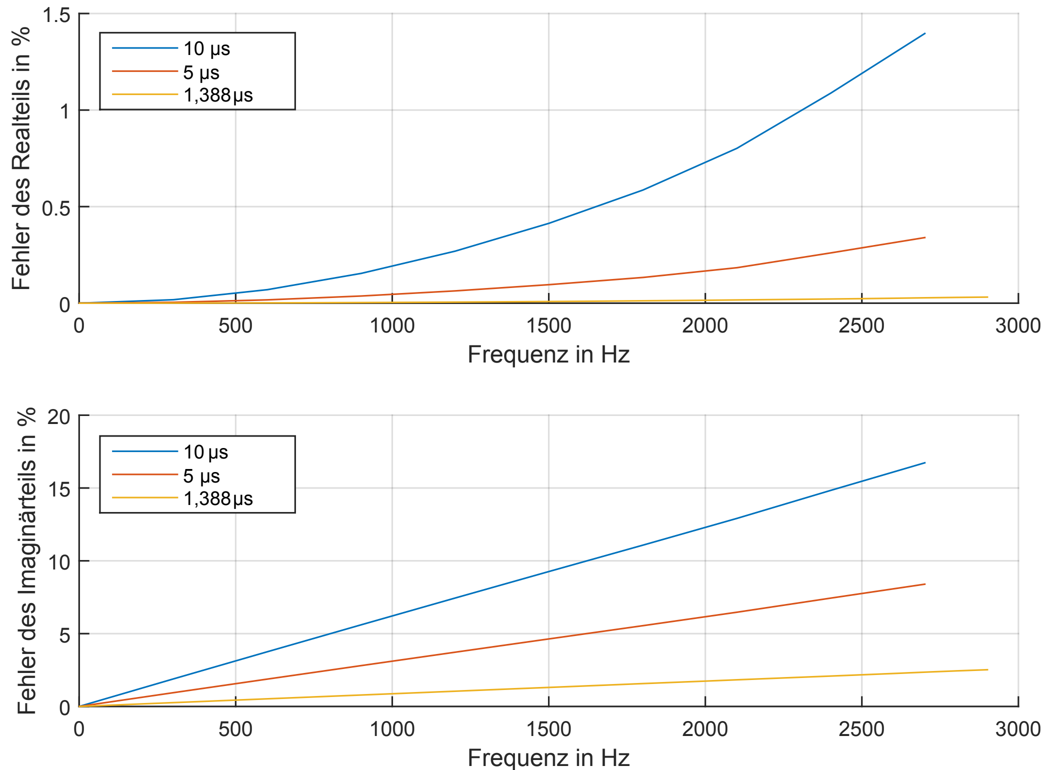


Abbildung 3.8.: Synchronisationsfehler in Abhängigkeit der Frequenz bei unterschiedlichen Synchronisationszeiten

Die Impedanzgenauigkeit für einen Synchronisationsfehler von 1,388 μs liegt für ein 2 kHz Signal knapp unter 1 %. Somit kann auch für das Kriterium der Impedanzgenauigkeit der Wert von unter 1 % angegeben werden. Vorausgesetzt der Synchronisationsfehler liegt unterhalb der ermittelten 1,388 μs.

3.3.2. Synchronisation zwischen den Sensoren

Die Synchronisation zwischen den einzelnen Zellsensoren ist wichtig, um die Messergebnisse untereinander vergleichen zu können. Ein Synchronisationsfehler wird hier im wesentlichen beeinflusst durch den Jitter, der bei der Demodulation des OOK-Signals an den einzelnen Zellsensoren entsteht. Diese Demodulation kann je nach Sensor unterschiedlich lange dauern, sodass die Zellsensoren zu unterschiedlichen Zeitpunkten messen. Dazu wurden mit den Zellsensoren aus der Arbeit [48] Messungen hinsichtlich des Jitterverhaltens durchgeführt. Es wurden zwei unterschiedliche Zellsensoren vermessen und miteinander verglichen [70]. Ausgewertet wurden dabei 400 Messimpulse, die vom Batteriesteuergerät als Broadcastnachricht an beide Zellsensoren gesendet wurden. Dabei entstand eine maximale Abweichung zwischen den beiden Sensoren von 777,0 ns beim Empfang des Messimpulses. Die durchschnittliche Abweichung lag bei 206.9 ns. Man liegt dabei zwar noch unterhalb der 1,388 μs, die im vorangegangenen Abschnitt ermittelt wurden, dennoch ist diese Zeit zu beachten, da dadurch die maximale erreichbare Genauigkeit limitiert wird.

Dynamische Laufzeitermittlung

Eine Möglichkeit, die unterschiedlichen Laufzeiten bei den Zellsensoren zu detektieren, ist die sog. Ping-Pong Messung [5]. Das Messprinzip ist dabei die Laufzeitmessung eines Triggerimpuls der ohne jeglichen Datenzusatz zwischen Batteriesteuergerät und Zellsensor hin und her gesendet wird (Abbildung 3.9). Daher der Name der Ping-Pong Messung.

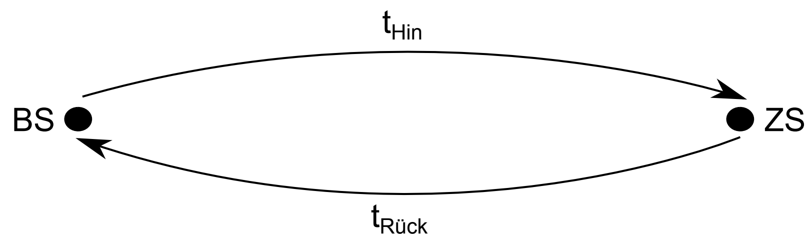


Abbildung 3.9.: Dynamische Laufzeitermittlung zwischen dem Batteriesteuergerät (BS) und dem Zellsensor (ZS)

Die Laufzeit des Triggerimpulses lässt sich durch durch die Zeiten des hin und her senden berechnen. Diese Messung berücksichtigt neben der eigentlichen Kanallaufzeit der Funkstrecke auch die Zeit, die für die Modulation bzw. Demodulation des Signals benötigt wird.

$$\text{Laufzeit} = \frac{(\mathrm{t_{Hin}} + \mathrm{t_{Rück}})}{2} \tag{3.7}$$

Durch die Messung an verschiedenen Zellsensoren lässt sich somit der Synchronisationsfehler Δ_{Synch} zwischen den Zellsensoren berechnen.

$$\Delta_{\text{Synch}} = |\text{Laufzeit Sensor 1} - \text{Laufzeit Sensor 2}| \tag{3.8}$$

Inwieweit durch diese Messmethode ein Unterschied zwischen mehreren Zellsensoren ermittelt werden kann, muss durch praktische Untersuchungen noch ermittelt werden, da die zu messenden Zeiten sich in der Größenordnung von wenigen Taktzyklen des Zellsensors befinden. Allerdings bietet dieses Verfahren eine gute und einfache Möglichkeit, die Laufzeit einzelner Sensoren im System zu messen. Dadurch können Laufzeitfehler schnell detektiert werden.

3.4. Größe der Messamplitude

Die Größe der Messamplitude ist ein wichtiges Thema für die Genauigkeit der Impedanzmessung, da die falsche Wahl der Messamplitude eine Verfälschung der gemessenen Impedanz zur Folge haben kann.
Für die Messung der Spannungsamplitude ist es vorteilhaft, dass die Spannungsamplitude des Wechselanteils so groß wie möglich ist. Dies erleichtert deren Messung enorm und verbessert deren Genauigkeit. In erster Näherung, ist also eine große Strom-Anregungsamplitude zu wählen, da ein größerer Anregestrom eine größere Spannungsamplitude zur Folge hat. Dies kann bei Batterien allerdings problematisch werden, da es sich dabei um ein nichtlineares System handelt. Werden Messungen über einen nichtlinearen Bereich getätigt, entstehen Verzerrungen, die sich im Frequenzspektrum bemerkbar machen. Diese Problematik wurde bereits in den Vorgängerarbeiten [48] [2] und in der bekannten Literatur zur Impedanzspektroskopie [42] erklärt und ausführlich behandelt.
Hieraus ergibt sich die Frage, wie groß die Messamplitude werden darf und welche Auswirkungen eine Messung im nichtlinearen Bereich der Batteriezelle auf das gemessene Impedanzspektrum hat.

3.4.1. Impedanzmessung an Batterien

Bei der Vermessung einer Batteriezelle ist ein besonderes Augenmerk auf die Anregungsamplitude zu richten, da die Batterieimpedanz stark stromabhängig ist. Diese Stromabhängigkeit wird durch den Widerstand R_{ct} verursacht [42], wie er bereits in Abschnitt 2.1.2 beschrieben wurde. Dabei wurde gezeigt, dass die Stromabhängigkeit des Widerstands R_{ct} und somit die Nichtlinearität des Systems durch die Butler-Volmer-Gleichung beschrieben werden kann. Nichtlinearitäten infolge von Temperatureinflüssen werden hier nicht betrachtet. Es wird von einer konstanten Temperatur ausgegangen.

Im Folgenden wird nun versucht, die Einflüsse an einem gemessenen Impedanzspektrum, die durch die Messungen an der Nichtlinearität entstehen, zu untersuchen. In Abbildung 3.10 ist die Nichtlinearität, beschrieben durch die Butler-Volmer-Gleichung, zu sehen. Es wurde dabei mit zwei unterschiedlichen Anregungsamplituden ohne Gleichstromanteil gemessen. Zum einen mit einer Anregungsamplitude von 0,5 A und zum anderen mit einer Anregungsamplitude von 5,0 A.

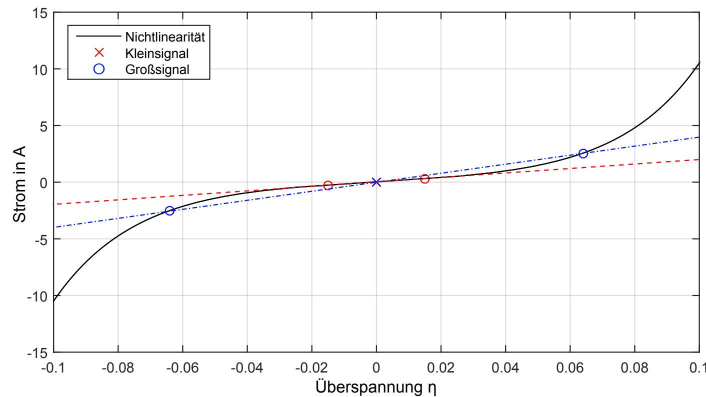


Abbildung 3.10.: Kleinsignalverhalten ohne Offset: $R_{\text{Kleinsignal}} = 59{,}6\,\text{m}\Omega$, $R_{\text{Großsignal}} = 25{,}6\,\text{m}\Omega$: Gleichstromanteil: 0,0 A, Kleinsignalamplitude: 0,5 A, Großsignalamplitude: 5,0 A

Die zu messende Impedanz ist jeweils die Tangente an der Nichtlinearität im Arbeitspunkt [42], in diesem Fall 63,1 mΩ. Es ergibt sich hier für die Anregung mit 0,5 A eine Impedanz von $R_{\text{Kleinsignal}} = 59{,}6\,\text{m}\Omega$. Mit der Anregungsamplitude von 5,0 A liegt die ermittelte Impedanz bei $R_{\text{Großsignal}} = 25{,}6\,\text{m}\Omega$. Die Impedanzen der beiden Anregungsamplituden weichen um mehr als den Faktor zwei voneinander ab. Es ist also deutlich zu sehen, dass ein Unterschied zwischen einer Kleinsignalamplitude und einer Großsignalamplitude besteht.

Eine Verbesserung ergibt sich bei der Hinzufügung eines Gleichstromanteils. Dadurch wird der Arbeitspunkt der Messung in einen quasi-linearen Bereich geschoben. Dadurch wirkt sich die Nichtlinearität nicht so stark auf die Messergebnisse aus. In Abbildung 3.11 ist ein Gleichstromanteil von 3 A auf die Messamplitude beaufschlagt. Die Impedanzen zeigen, dass hier die Abweichungen zwischen Klein- und Großsignalamplitude wesentlich kleiner sind als ohne Gleichstromanteil.

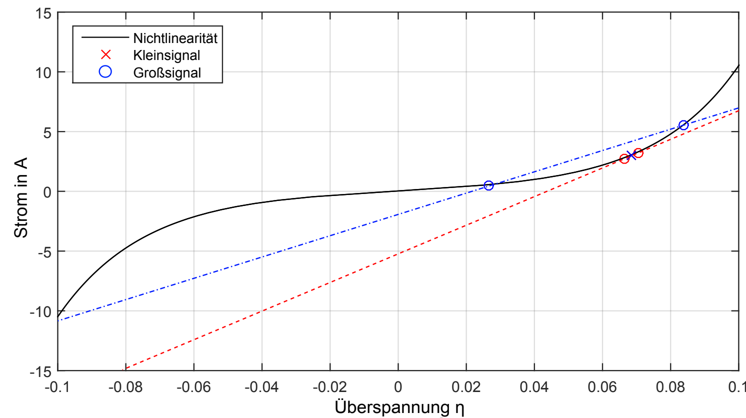


Abbildung 3.11.: Kleinsignalverhalten mit Offset: $R_{Kleinsignal} = 8{,}3\,m\Omega$, $R_{Großsignal} = 11{,}5\,m\Omega$: Gleichstromanteil: 3,0 A, Kleinsignalamplitude: 0,5 A, Großsignalamplitude: 5 A

Dies zeigt, dass sich die Anregung immer aus einem Gleichstromanteil und einer relativ kleinen Anregungsamplitude zusammen setzten sollte. Zu sehen ist dies auch bei Messungen mit dem FuelCon TrueData-EIS, dabei wird auf den Anregeamplitudenanteil immer einen Gleichstromanteil hinzufügen.
In Abbildung 3.12 ist die Auswirkung auf das Impedanzspektrum eines einfachen Randles-Modells zu sehen (vgl. Abschnitt 2.3.1). Dieses Beispiel zeigt, dass der Fehler der Nichtlinearität besonders bei niedrigen Frequenzen einen Einfluss auf das Impedanzspektrum hat. Grund hierfür ist, dass bei hohen Frequenzen die zum Widerstand R_{ct} parallel liegende Kapazität C_{dl} niederohmig wird und somit den Hauptteil des Anregestroms trägt. Erst bei niedrigen Anregefrequenzen kommutiert der Anregestrom zum nichtlinearen Widerstand R_{ct} [42].

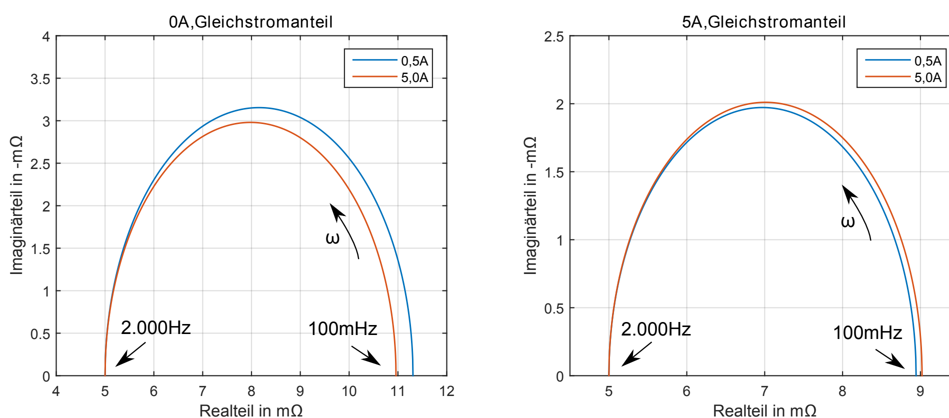


Abbildung 3.12.: Vergleich der abweichenden Impedanzspektren mit und ohne Gleichstromanteil

Über die genaue Größe der Anregungsamplitude gibt es in der Literatur keine konkreten Werte, da diese stark von der zu messenden Zelle abhängt. Allerdings werden Werte angegeben, die die maximale Größe der Messamplitude beschreiben. Diese sollen laut [4] den Wert von 10 mV nicht überschreiten, um in einem Quasi-Linearen-Bereich zu bleiben. In [2] wird angegeben, dass eine Messamplitude bis zu 5 mV genutzt werden kann, um eine Verzerrung durch die Nichtlinearität zu vermeiden. In [42] wird als Wert sogar 3 mV als guter Erfahrungswert angegeben.
Untersucht werden muss nun, wie groß die Auflösung des AD-Wandlers sein muss, um die Messamplitude noch in ausreichender Genauigkeit messen zu können. Es wurde festgelegt, dass das Messsignal noch mit mindestens 100 Stufen aufgelöst werden muss. Dadurch lässt sich die benötigte Auflösung des AD-Wandlers berechnen. Bei der Berechnung wird angenommen, das der AD-Wandler eine Referenzspannung von 2,5 V besitzt. Dadurch ergeben sich die folgenden Auflösungen für die unterschiedlichen Messamplituden.

Um eine 3 mV Messamplitude mit einer Auflösung von mindestens 100 Stufen messen zu können, ist ein AD-Wandler mit einer Auflösung von 16,34 Bit notwendig.

$$log_2\left(\frac{2,5\,V}{\left(\frac{3\,mV}{100}\right)}\right) = 16,34\,Bit \tag{3.9}$$

Bei der Einhaltung, der in [2] angegeben Messamplitude bis zu 5 mV, werden 15,61 Bit benötigt.

$$log_2\left(\frac{2,5\,V}{\left(\frac{5\,mV}{100}\right)}\right) = 15,61\,Bit \tag{3.10}$$

Nutzt man die Angabe mit einer Messamplitude von 10 mV, ergibt sich eine minimale Bitanzahl des AD-Wandlers von 14,61 Bit

$$log_2\left(\frac{2,5\,V}{\left(\frac{10\,mV}{100}\right)}\right) = 14,61\,Bit \tag{3.11}$$

Es lässt sich also festhalten, dass für die Impedanzmessung ein AD-Wandler mit mindestens 14,61 Bit vorhanden sein muss, damit eine Auflösung der Messamplitude von 10 mV in 100 Stufen erreicht werden kann. Vorzuziehen ist allerdings ein höherer Wert, um einerseits Störungen durch die Nichtlinearität der Batteriezelle zu vermeiden und andererseits eine höhere Genauigkeit zu erreichen. Für die AD-Wandler Auswahl sollte daher eine höchst mögliche Auflösung gewählt werden. Es wird daher ein AD-Wandler mit einer Mindestauflösung von 15,61 Bit bevorzugt.

3.5. Neukonzeption des Messsystems

Im folgenden Abschnitt sollen verschiedene Konzepte für eine Messung der Wechselspannung behandelt werden. Dabei besteht das Hauptproblem darin, dass eine sehr kleine Wechselspannung mit einem großen Gleichspannungsanteil gemessen werden soll. Im vorangegangenen Abschnitt 3.4 wurde ermittelt, dass die zu messende Amplitude des Wechselsignals einen Minimalwert von 3 mV erreichen kann. Diese 3 mV müssen dann auch noch ausreichend gut aufgelöst werden.
Grundsätzlich gibt es dabei zwei verschiedene Arten, die Messung durchzuführen. Zum einen kann die Spannung direkt gemessen werden. Dazu wird ein hochauflösender AD-Wandler benötigt der die kleine Wechselspannung mit ausreichender Auflösung erfasst. Diese Art der Wechselspannungsmessung wurde bereits im vorhergehenden Zellsensor v0.4 verwendet. Eine weitere Art der Messung ist es, nur den Wechselanteil der Spannung ohne Gleichspannungsanteil zu messen. Dies hat den Vorteil, dass das Wechselsignal verstärkt und mit einem niedriger auflösenden AD-Wandler vermessen werden kann. Der Abzug des Gleichspannungsanteils kann z.B. über eine geeignete analoge Vorverarbeitung des Signals realisiert werden. Im Folgenden werden beide Messkonzepte vorgestellt und hinsichtlich ihrer Realisierbarkeit überprüft. Dabei muss überprüft werden, ob weiterhin ein hochauflösender AD-Wandler verwendet werden soll oder ein Messkonzept mit analoger Vorverarbeitung erarbeitet wird.

3.5.1. Messkonzeption für eine hochauflösende Messung

Der Einsatz eines hochauflösenden AD-Wandlers wurde bereits in [48] erfolgreich getestet. Ein Problem dabei war, dass der gewählte AD-Wandler nur bis zu einer Frequenz von 350 Hz ausreichend genau messen konnte und somit nicht für eine komplette Impedanzspektroskopie an Batterien geeignet ist. Deshalb muss, falls diese Art der Wechselspannungsmessung angewandt wird, ein neuer AD-Wandler ausgewählt werden. Dazu werden nun einige geeignete Wandler und die momentan am Markt verfügbaren Techniken miteinander vorgestellt.

SAR-Wandler

Ein Sukzessives Approximation Register (SAR) Wandler ist ein AD-Wandler-Typ, der heutzutage häufig als integrierter AD-Wandler in Mikrocontrollern vorkommt. Dieser Wandler Typ zeichnet sich durch eine sehr schnelle Sample-Rate bei einer geringen Leistungsaufnahme aus. Typischerweise erreichen die SAR AD-Wandler eine Auflösung von 16-18 Bit. Die SAR-Wandler nutzen dabei das Wägeverfahren, um sich der zu messenden Spannung anzunähern. Das bedeutet, dass der digitale Datenstrom dem analogen Signal bitweise angenähert wird. Dabei wandelt ein N-Bit AD-Wandler mit SAR-Technik eine Eingangsspannung innerhalb von N-Zyklen in einen digitalen Datenstrom um. Ein Komparator vergleicht dabei das analoge Eingangssignal mit dem Wert des digitalen Datenstroms. Dabei wird verglichen, ob der Wert des digitalen Datenstroms größer oder kleiner ist als das analoge Eingangssignal. Dementsprechend wird der Datenstrom bitweise erhöht bzw. erniedrigt. Dieses Prinzip zeigt die Abbildung 3.13.

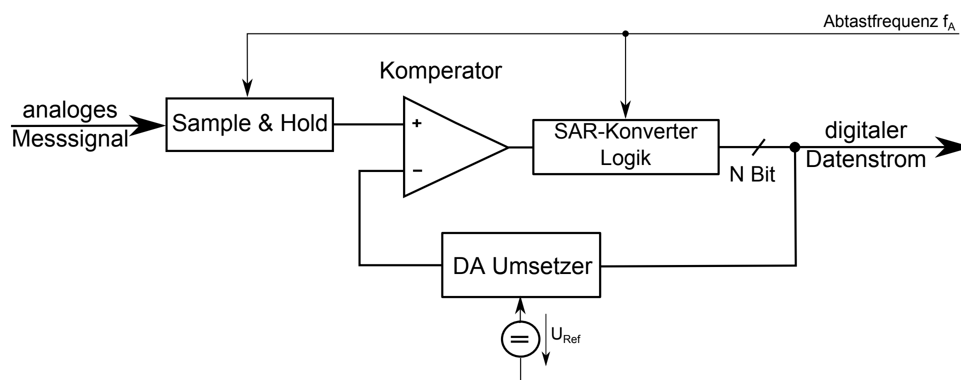


Abbildung 3.13.: Blockschaltbild eines SAR-AD-Wandlers (nach [67])

Typischerweise erreichen diese Art von AD-Wandlern nicht ganz so eine gute Auflösung wie Sigma-Delta Wandler, weshalb dieser Wandler-Typ hier auch näher betrachtet wird.

Delta-Sigma-Wandler

Ein Delta-Sigma-Umsetzer erreicht deutlich höhere Auflösungen als ein SAR-Wandler. Grund hierfür ist eine spezielle Technik, die eingesetzt wird, um den Signal-Rausch-Abstand (SNR) zwischen Messsignal und dem Quantisierungsrauschen zu erhöhen. Der Delta Sigma Wandler nutzt dabei eine Überabtastung des analogen Eingangssignals, um das Quantisierungsrauschen über einen weiten Frequenzbereich hin zu verteilen. So bleibt

beispielsweise bei einer Verdoppelung der Abtastfrequenz, die Leistung des Quantisierungsrauschens konstant, verteilt sich dabei allerdings auf dem doppelten Frequenzbereich. Bei einem Delta-Sigma-Wandler wird neben der Verteilung des Quantisierungsrauschens auch eine sog. Rauschformung durchgeführt. Dies geschieht durch die Integration im Delta-Sigma-Wandler, wobei die Leistung des Quantisierungsrauschens hin zu höheren Frequenzen verschoben wird. Abbildung 3.14 zeigt dabei eine einfache schematische Darstellung der Rauschformung.

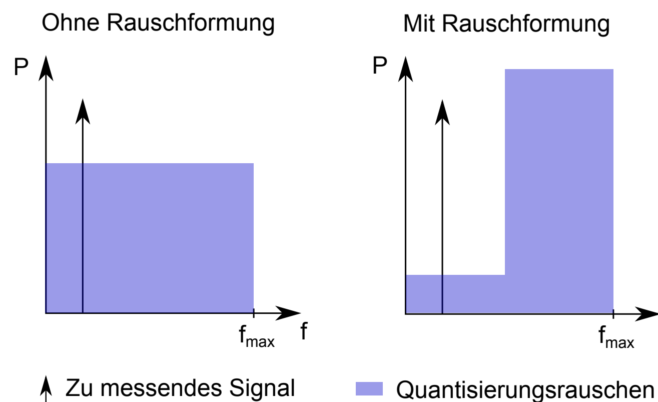


Abbildung 3.14.: Schematische Darstellung der Rauschformung (nach [67])

Durch diese Technik erreicht der Delta-Sigma-Wandler im unteren Frequenzbereich einen sehr guten SNR, weshalb er auch eine deutlich höhere Auflösung als ein SAR-Wandler erreicht. Abbildung 3.15 zeigt das Blockschaltbild eines Delta-Sigma-Wandler 1. Ordnung.

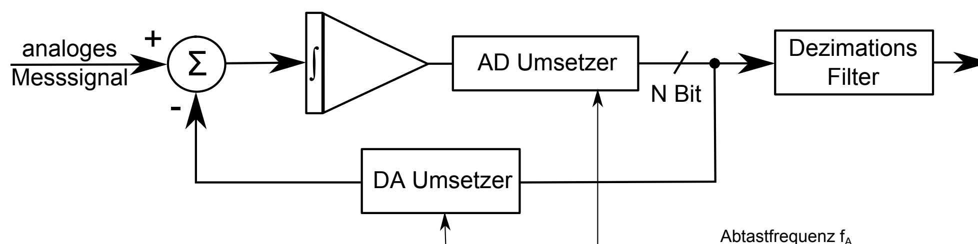


Abbildung 3.15.: Blockschaltbild eines Delta-Sigma-AD-Wandlers (nach [67])

Mit Delta-Sigma-Wandlern erhält man also eine wesentlich bessere Auflösung als mit dem SAR-Wandler. Allerdings erreichen diese Art der Wandler, bedingt durch die Rauschformung, eine geringere Sample-Rate.
Ein weiterer wichtiger Unterschied zwischen SAR-Wandlern und Delta-Sigma-Wandlern ist, dass der SAR-Wandler einen getakteten Eingang besitzt. Dadurch sind exakte Messungen bei einem Zeitpunkt möglich [67]. Der Delta-Sigma-Wandler besitzt keinen getakteten Eingang, sondern misst kontinuierlich und Integriert den Durchschnittswert über eine bestimmte Zeit. Dadurch sind mit ihm keine Messungen zu einem Exakten Zeitpunkt möglich.

AD-Wandler Auswahl

Für die Wahl eines geeigneten AD-Wandlers, wird eine Auswahl, der im Moment auf dem Markt gängigen AD-Wandler miteinander verglichen. Ein Wichtiges Kriterium ist dabei, dass die Spannungsaufnahmen des AD-Wandlers durch ein Signal unverzüglich ausgelöst werden kann. Dies ist notwendig, um die getaktete und synchrone Strom- und Spannungsaufnahme zu realisieren. Daher ist bei der Auswahl ein SAR-AD-Wandler vorzuziehen. Zwei sehr wichtige Kriterien sind die maximale effektive Auflösung und die maximale Datenrate des Wandlers. Der Wandler sollte eine minimale Auflösung von 15,61 Bit haben und eine Datenrate von mindesten 4 kSPS, um das Nyquistkriterium für ein Messsignal von 2 kHz zu erreichen.
Neben diesen Auswahlkriterien, ist auf die Versorgungsspannung des Wandlers zu achten. Der Zellsensor besitzt lediglich eine Spannungsversorgung von 3,3 V. Einige AD-Wandler benötigen aber eine Spannungsversorgung von 5,0 V. Um diese auf dem Zellsensor zu erreichen, ist ein zusätzlicher Hardwareaufwand erforderlich. Ein weiterer Hardwareaufwand wird auch verursacht, wenn eine externe Referenzspannung benötigt wird. Es ist also ein Wandler vorzuziehen, der über eine interne Referenzspannung verfügt. Zuletzt ist auch auf den Energiebedarf des Wandlers zu achten. Für erste Versuche kann auch ein Wandler eingesetzt werden, der einen höheren Energiebedarf hat. Wenn möglich ist aber ein Wandler mit niedrigem Energiebedarf zu wählen, um die Batteriezelle nicht unnötig zu belasten.

Tabelle 3.3.: Bewertungsmatrix AD-Wandler Auswahl

Eigenschaften	Sehr wichtig	Wichtig	Neutral
Wandlertyp	-	X	-
Auflösung	X	-	-
Datenrate	X	-	-
Versorgungsspannung	-	X	-
Referenzspannung	-	-	X
Energiebedarf	-	X	-

Die obenstehende Bewertungsmatrix gibt die Bewertung der einzelnen Eigenschaften und deren Gewichtung wieder. Anhand dieser Kriterien kann eine sinnvolle Auswahl eines geeigneten AD-Wandlers erfolgen.

Tabelle 3.4.: Übersicht verschiedener Digital-Analog-Wandler

Hersteller	**Hersteller-bezeichnung**	**Wandlertyp**	**Auflösung (max./ENOB)**	**Datenrate (max.)**	**Versorgungs-spannung**	**Referenz-spannung**	**Energie-bedarf**
AD	AD7767[9]	SAR	24 Bit/17,14 Bit	128 kSPS	2,5 V	extern	15,0 mW
AD	AD7690[11]	SAR	18 Bit/16,56 Bit	400 kSPS	5,0 V	extern	17,0 mW
AD	AD7641[8]	SAR	18 Bit/15,30 Bit	1.500 kSPS	2,5 V	2,048 V	75,0 mW
AD	AD7691[12]	SAR	18 Bit/15,82 Bit	250 kSPS	3,3 V	extern	1,35 mW bis 2,4 mW
AD	AD7176-2[10]	Delta-Sigma	24 Bit/20,09 Bit	250 kSPS	5,0 V	2,5 V	22,3 mW
AD	AD7989[13]	Delta-Sigma	18 Bit/12,53 Bit	100 kSPS	2,5 V	extern	700 μW
LT	LTC2376-20[65]	SAR	20 Bit/17,64 Bit	250 kSPS	2,5 V	extern	5,25 mW
TI	ADS1291[31]	Delta-Sigma	24 Bit/15,07 Bit	8 kSPS	3,3 V	2,42 V	350 μW pro Kanal
TI	ADS1271[30]	Delta-Sigma	24 Bit/16,07 Bit	105 kSPS	5,0 V	extern	92,0 mW
MI	MAX11200[37]	Delta-Sigma	24 Bit/19,00 Bit	120 SPS	3,3 V	2,5 V	300 μW

Hersteller Abkürzungen: TI = Texas Instruments, AD = Analog Devices, LT = Linear Technology, MI = Maxim Integrated

Die Auswertung ergab, dass nicht alle der gewählten AD-Wandler die erforderlichen effektiven 15,61 Bit erreichen. Da die effektive Bitanzahl allerdings stark von der Datenrate abhängt, können diese Werte nicht genau miteinander verglichen werden. Es wurden nun zwei Wandler ausgewählt, die alle Mindestanforderungen erfüllen. Diese erfüllen neben den Anforderungen an die Datenrate und effektive Auflösung auch die Anforderung an einen geringen Energiebedarf. Ausgewählt wurden die beiden Wandler von Analog Devices AD7691 und von Linear Technology LTC2376-20.

•AD7691
Der AD7691 von Analog Devices ist ein schneller SAR-Wandler mit einer nominalen Auflösung von 18 Bit. Er erreicht eine effektive Auflösung von 15,82 Bit, womit eine Amplitude von 4,32 mV noch mit ausreichender Genauigkeit gemessen werden kann. Die Versorgungsspannung liegt dabei bei 3,3 V. Er kann somit mit der Spannungsversorgung des Zellsensors betrieben werden. Der Energiebedarf liegt bei 1,35 mW. Der einzige Nachteil ist, dass eine externe Referenzspannungsquelle eingesetzt werden muss.

•LTC2376-20
Der LTC2376-20 von Linear Technology ist ein hochauflösender SAR-AD-Wandler mit 20 Bit nominaler Auflösung. Effektiv werden 17,64 Bit erreicht. Es wird dabei eine Sample-Rate von bis zu 250 kSPS erreicht. Dieser benötigt ebenfalls eine externe Referenzspannungsquelle. Ein weiterer Nachteil im Vergleich zum AD7691 ist der höhere Energiebedarf von 5,25 mW. Dennoch wird er wegen seiner sehr guten Auflösung und der hohen Geschwindigkeit zum Test ausgewählt.

Messbereichseinstellung

Die ausgewählten AD-Wandler haben beide einen Referenzspannungsbereich von 2,5 V. Um nun aber den geforderten Spannungsbereich von 2,0 V bis 3,4 V zu erreichen, muss die zu messende Spannung mindestens um 0,9 V abgesenkt werden um in den Referenzspannungsbereich zu kommen. Um dies zu erreichen, gibt es grundsätzlich drei unterschiedliche Möglichkeiten die im Folgenden kurz vorgestellt werden.

- Spannungsreduzierung durch Spannungsteiler
- Gleichspannungs-Kompensation durch Filterung
- Spannungsreduzierung durch Gleichspannungs-Abzug

Spannungsreduzierung durch Spannungsteiler

Spannungsreduzierung mittels eines Spannungsteilers ist zunächst ein sehr einfaches Prinzip, den Spannungsbereich von 2 V bis 3,4 V zu verkleinern. Zudem ist der Spannungsteiler leicht zu dimensionieren und hat einen sehr geringen Hardwareaufwand von lediglich zwei Widerständen. Durch den Einsatz eines 2:1 Spannungsteilers gelingt es die Spannung in den Bereich von 1 V bis 1,7 V zu verkleinern und somit in den Bereich des AD-Wandlers zu bringen. Ein Nachteil des Spannungsteilers ist es aber, dass dieser nicht nur den Gleichspannungsanteil verringert, sondern auch den Wechselanteil des Messsignals. Dadurch geht ein Bit Genauigkeit am Spannungsteiler verloren. So werden beim Einsatz des Spannungsteilers und einer Wechselspannungsamplitude von 3 mV nicht mehr 16,34 Bit, sondern 17,34 Bit benötigt.

$$log_2\left(\frac{2,5}{\left(\frac{3\,mV}{100}\right)/2}\right) = 17,34\,Bit \tag{3.12}$$

Gleichspannungs-Kompensation durch Filterung

Eine effektive und zugleich einfache Art, den Gleichanteil des Messsignals auszukoppeln, ist die Gleichspannungs-Kompensation durch Filterung. Dabei kommt ein RC-Hochpassfilter zum Einsatz, der nur den Wechselanteil des Messsignals durch lässt und die Gleichspannung blockiert.

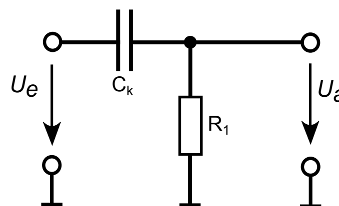


Abbildung 3.16.: Spannungs-Subtrahierschaltung

Dadurch, dass bei der Impedanzspektroskopie Messfrequenzen bis zu wenigen mHz gemessen werden sollen, muss die Grenzfrequenz des Filters weit unterhalb dieser Messfrequenzen liegen um die Messung nicht zu verfälschen. Die Grenzfrequenz f_G des Hochpass wird dabei maßgeblich durch den Wert des Kondensators C_k beeinflusst. Dieser muss ausreichend groß sein, damit die Grenzfrequenz f_G klein genug ist.

$$f_G = \frac{1}{2 \cdot \pi \cdot R \cdot C_k} \tag{3.13}$$

Abbildung 3.17 zeigt das Verhältnis zwischen Aus- und Eingangsspannung bei verschiedenen Frequenzen in Abhängigkeit der Kapazität des Kondensators. Zu sehen ist dabei, dass die Kapazität mit sinkender Frequenz stark ansteigt, um das Verhältnis zwischen der Ausgangsspannung U_A und der Eingangsspannung U_E auf den nötigen Wert von 1 zu halten, sodass die Messamplitude nicht verfälscht wird.

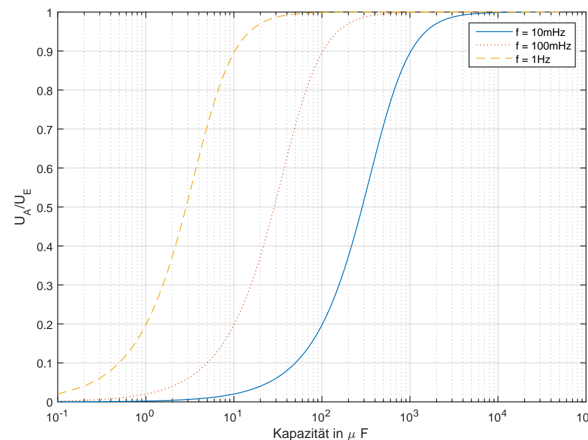


Abbildung 3.17.: Kapazität der DC-Filterung

Die Berechnungen zeigen, dass die Kapazität des Kondensators einen sehr großen Wert annimmt. Für eine Filterung bis 10 mHz sind dabei Kapazitäten nötig, bei denen die physische Bauteilgröße des Kondensators die Größe des Zellsensors weit überschreitet.

Spannungsreduzierung durch Gleichspannungs-Abzug

Eine weitere Möglichkeit den Gleichspannungsanteil zu minimieren, ist die Spannungsreduzierung durch Gleichspannungs-Abzug. Dabei soll ein gewisser Gleichspannungsanteil vom zu messenden Signal abgezogen werden, ohne den Wechselanteil zu beeinflussen. Möglich ist dies durch eine Subtrahierschaltung.
Mit Hilfe dieser Schaltung ist es möglich, den gewünschten Messbereich von 2,0 V bis 3,4 V in den Referenzspannungsbereich des AD-Wandlers zu bringen. Mittels der beiden Widerstände R_1 und R_2 (in Abbildung 3.18) lässt sich die abzuziehende Spannung einstellen. Diese werden so eingestellt, dass eine Spannung von min. 1,0 V abgezogen wird. Dadurch liegt der zu messende Spannungsbereich von 2,0 V - 3,4 V immer unterhalb der Referenzspannung des AD-Wandlers von 2,5 V und kann durch den Wandler erfasst werden.

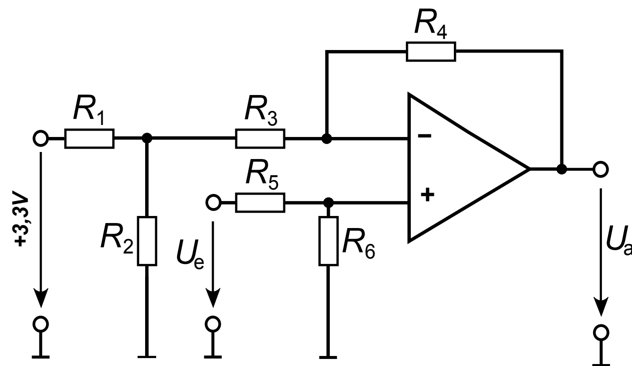


Abbildung 3.18.: Spannungs-Subtrahierschaltung

Ein Nachteil dieser Schaltung ist der mittlere Hardwareaufwand von einem Operationsverstärker und sechs Widerständen. Zusätzlich verschlechtert diese Schaltung die Energiebilanz des Zellsensors, da der Operationsverstärker versorgt werden muss und durch den Spannungsteiler ständig Strom fließt.

Auswahl der Messbereichseinstellung

Mit allen drei vorgestellten Methoden lässt sich ein Gleichspannungsanteil eines Signals reduzieren. Sie sind aber nicht alle geeignet für den Einsatz bei einer Impedanzspektroskopie. So lässt sich die Gleichspannungs-Kompensation durch Filterung mittels Kondensator allein dadurch nicht realisieren, dass die Bauteilgröße des Kondensators die Zellsensorgröße bzw. die Zellgröße um ein Weites überschreiten würde, um bis in den niedrigen Frequenzbereich hinein messen zu können. Die Methode mittels Spannungsteiler wurde bereits in [48] erfolgreich eingesetzt. Durch die dadurch entstehenden Auflösungsreduzierung, wird sich an dieser Stelle aber gegen diese Art der Messbereichseinstellung entschieden. Es wird die Methode der Spannungsreduzierung durch Gleichspannungs-Abzug für den Test der ausgewählten AD-Wandler gewählt.

Auswahl eines hochauflösenden AD-Wandlers

Beide auswählten AD-Wandler wurden mit der zuvor ausgewählten Schaltung zur Gleichspannungsreduzierung aufgebaut und getestet. Dabei konnten mit beiden AD-Wandlern Signale im geforderten Frequenzbereich und mit einer Amplitude von 3 mV aufgezeichnet werden. Beide AD-Wandler sind also geeignet für den Einsatz zur Erfassung der Spannung für die funksynchronisierte Impedanzspektroskopie. Für eine weitere Erprobung wurde sich für den Analog Devices AD7691 entschieden. Dieser hat zwar eine geringere Auflösung als der Linear Technology LTC2376-20, lieferte aber ausreichen gute Ergebnisse. Zudem zeigte er währen der Vortests eine wesentlich bessere und einfachere Ansteuerbarkeit.

3.5.2. Messkonzeption für eine Messung mit analoger Vorverarbeitung

Ein weiteres Messkonzept für die Messung der Impedanzspektroskopie ist das mit der analogen Vorverarbeitung des Messsignals. Dabei wird das Messsignal soweit analog vorverarbeitet, dass es mittels des internen 12-Bit SAR AD-Wandlers des eingesetzten Mikrocontrollers gemessen werden kann.
Dabei soll, wie bei der Messbereichsanpassung des hochauflösenden AD-Wandlers, zuerst der Gleichspannungsanteil des Messsignals weitestgehend kompensiert werden und das verbleibende Signal soweit verstärkt werden, dass es von einem 12-Bit AD-Wandler ausreichend genau gemessen werden kann. Ein Problem dabei kann der flexible Spannungsbereich von 2,0 V bis 3,4 V darstellen. Es können maximal 2,0 V Gleichspannung abgezogen werden, da sonst Messungen im unteren Spannungsbereich nicht mehr möglich sind und so die Forderungen hinsichtlich des Spannungsbereichs nicht mehr sichergestellt werden können. Werden allerdings nur die 2,0 V abgezogen und der aktuelle Gleichspannungsanteil beträgt 3,3 V, heißt das, dass auch der restliche Gleichspannungswert von 1,3 V mit verstärkt wird. Somit kann nicht einmal um den Faktor zwei verstärkt werden, ohne den Referenzspannungsbereich von 2,5 V des AD-Wandlers zu überschreiten. Eine Lösung dieses Problems ist die dynamisch einstellbare Subtrahierschaltung. Dies ist eine normale Subtrahierschaltung, wie sie im vorherigen Abschnitt bereits erläutert wurde, deren Spannungsabzug aber durch den Mikrocontroller einstellbar ist. Erreicht wird dies durch einen Rheostaten[3], der vom Mikrocontroller gesteuert wird und jeweils an den aktuellen Gleichspannungsanteil angepasst werden kann. Zusammen mit dem Widerstand R_1 bildet der Rheostat einen Spannungsteiler, der dann die abzuziehende Spannung U_{Abzug} vorgibt.

$$U_{\text{Abzug}} = 3,3\,V \cdot \frac{R_{\text{Rheo}}}{R_1 + R_{\text{Rheo}}} \tag{3.14}$$

In Abbildung 3.19 ist die Subtrahierschaltung mit dynamisch einstellbarem Spannungsbereich mittels Rheostat zu sehen.

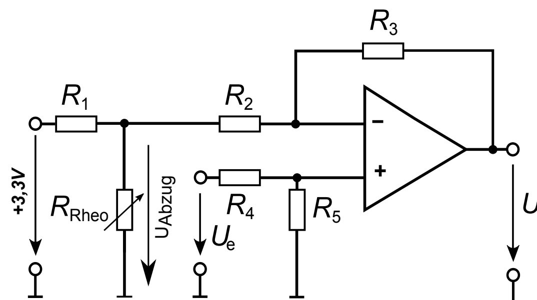


Abbildung 3.19.: Dynamisch einstellbare Subtrahierschaltung

[3]elektronisch einstellbarer Widerstand

Diese Schaltung unterscheidet sich zur der oben bereits vorgestellten Subtrahierschaltung lediglich durch den Rheostaten.
Nachdem das Messsignal einen geringeren Gleichspannungsanteil hat, kann dieses verstärkt werden. Diese Verstärkung erfolgt durch einen nicht invertierenden Verstärker am Ausgang der Subtrahierschaltung. Dessen Verstärkungsfaktor lässt sich mittels eines Spannungsteilers zwischen dem Ausgang des Verstärkers und dem negativen Eingang einstellen. Da es bei der Messung der Impedanzspektroskopie zu unterschiedlichen Verstärkungsfaktoren, bedingt durch unterschiedliche Impedanzbereiche und Gleichspannungen kommen wird, muss der Verstärkungsfaktor auch einstellbar sein. Dazu wird hier wieder ein Rheostat eingesetzt. So lässt sich die Spannung des Spannungsteilers und somit auch der Verstärkungsfaktor des Verstärkers durch den Mikrocontroller während des Betriebs anpassen. Die komplette analoge Vorverarbeitung mit der Subtrahierschaltung sowie des nicht invertierenden Verstärkers zeigt Abbildung 3.20.

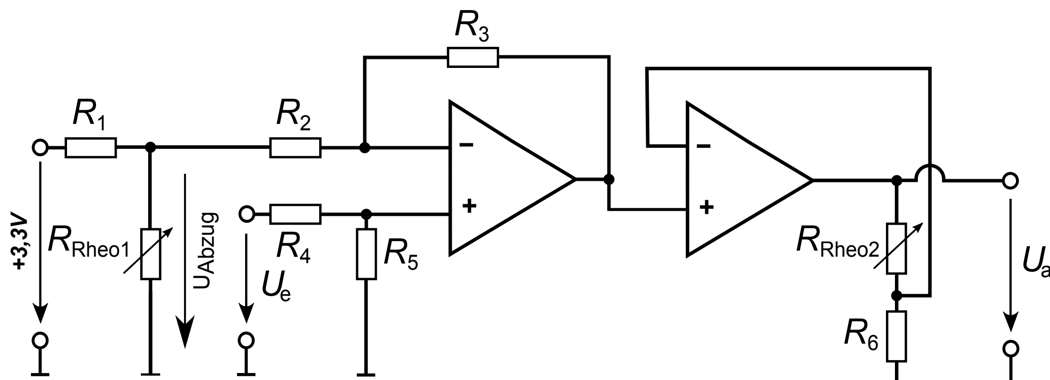


Abbildung 3.20.: Dynamisch einstellbare Subtrahierschaltung mit einstellbarer Verstärkung

Für den Grad der Verstärkung ist es wichtig, dass der Offsetanteil des Messsignals so gering wie möglich ist, um eine hohe Verstärkung des Wechselsignals zu erreichen. Um das Messsignal mit dem 12-Bit SAR AD-Wandler des CC430F5137 in der Aufteilung von 100 Spannungsstufen messen zu können, muss die Amplitude mindestens auf 61,04 mV verstärkt werden.

$$LSB = \frac{U_{\text{Ref}}}{2^{12}} \cdot 100 = \frac{2,5\,V}{2^{12}} \cdot 100 = 61,04\,mV \tag{3.15}$$

Bei einer Messamplitude von 3 mV bedeutet dies, dass das Signal um mindestens den Faktor 20,34 verstärkt werden muss, damit die 61,04 mV erreicht werden. Der restliche Offset darf dabei maximal 119,91 mV betragen, damit es zu keiner Verstärkung über die 2,5 V Referenzspannung des AD-Wandlers kommt. Für entsprechend größere Messamplituden wird der benötigte Verstärkungsfaktor dementsprechend kleiner und der maximale erlaubte Wert des Gleichspannungsanteils steigt an.

3.5.3. Vergleich der vorgestellten Konzepte

In Tabelle 3.5 wird ein kurzer Vergleich der beiden vorgestellten Messkonzepte gemacht. Dabei werden die Vor- und auch Nachteile der Konzepte gegenübergestellt.

Tabelle 3.5.: Vergleich der beiden vorgestellten Messkonzepte

Messkonzept	Vorteil	Nachteil
Hochauflösende Messung	• Signal kann einfach gemessen werden • Messung über den kompletten Frequenzbereich möglich	• Höchauflösende und schnelle AD-Wandler sind sehr teuer • Hohe Design-anforderungen
Analoge Vorverarbeitung	• Messung über den kompletten Frequenzbereich möglich • kostengünstig realisierbar	• mäßiger Hardwareaufwand • möglicher Fehlereinfluss durch die Vorverarbeitung

Zur Auswahl stehen die Lösungen mittels hochauflösendem AD-Wandler mit Gleichspannungskompensation und der Lösung mit einer analogen Vorverarbeitung und der Spannungsaufnahmen durch den internen 12-Bit AD-Wandler des eingesetzten Mikrocontrollers. Mit beiden Konzepten ist eine theoretische Messung der elektrochemischen Impedanzspektroskopie möglich.

Ein großer Vorteil des Konzepts der hochauflösenden Spannungsmessung ist es, dass das Messsignal einfach gemessen werden kann. Das heißt, es kann ohne zusätzliche Verstärkung erfasst werden. Dadurch können keine Fehler durch die Verstärkung oder durch Toleranzen von Bauteilen auftreten. Toleranzen und Abweichungen bei der Subtrahierschaltung haben keinen Einfluss auf die gemessene Impedanz, da nur der Wechselanteil für die Messung in die Berechnung mit einbezogen wird. Nachteilig bei dieser Lösung ist der nicht zu unterschätzende Designaufwand für eine korrekte Messung durch den hochauflösenden AD-Wandler. So können schon kleine Einkopplungen in die Leiterbahnen die Messwerte verfälschen und somit unbrauchbar machen. Des Weiteren sind die doch erheblichen Kosten der hochauflösenden AD-Wandler zu beachten, die in der Größenordnung der kompletten restlichen Elektronik des Zellsensors liegen können.

Die Lösung mittels der analogen Vorverarbeitung des Messsignals ist hingegen wesentlich kostengünstiger. Es lässt sich ebenfalls der komplette Frequenzbereich der für die Impedanzspektroskopie an Batteriezellen nötig ist abdecken. Der aufgeführte Nachteil dieses Konzepts ist der mäßige Hardwareaufwand, welcher im Vergleich zu den Vorteilen relativ gering einzuschätzen ist. Ein weiterer Nachteil wird in der Anfälligkeit gegenüber Störungen und in den Bauteiltoleranzen an der Verstärkerschaltung gesehen, welche das

Messsignal verfälschen können. Dieser Nachteil lässt sich aber durch die Verwendung von Widerständen mit einer geringen Toleranz weitgehend minimieren. Ein wesentlicher Vorteil dieser Lösung ist, dass der Gleichspannungsabzug und die Verstärkung variabel einstellbar sind. Dieses ist gerade in der Entwicklungsphase sehr vorteilhaft, da man sämtliche Bereiche testen kann.

In Hinblick auf eine spätere Anfertigung einer größeren Testserie oder sogar einer Massenproduktion mit hochintegrierter Elektronik in einem Chip, ist es kostentechnisch wesentlich günstiger das Konzept der analogen Vorverarbeitung weiter zu verfolgen. Der Hardwareaufwand hält sich dabei in einem überschaubaren Rahmen. Es kommen neben den herkömmlichen Kondensatoren und Widerständen lediglich zwei Operationsverstärker sowie zwei Rheostaten zum Einsatz. Vorstellbar ist auch, dass sich im weiteren Verlauf die beiden Rheostaten durch zwei Widerstände ersetzen lassen, wenn nicht der komplette Spannungsbereich von 2,0 V bis 3,4 V genutzt werden soll. Dies würde die Kosten der Bauteile und den Hardwareaufwand nochmals reduzieren, ist aber Anwendungsfall abhängig und muss in der Praxis erst verifiziert werden.

3.6. Berechnung des Impedanzspektrums durch verteilte Signalverarbeitung

Die Berechnung des Impedanzspektrums ist ein wichtiges Thema dieser Arbeit. Es muss anhand der Strom- und Spannungswerte die komplexe Impedanz der Batteriezelle ermittelt werden. In der Arbeit [48] wurden die aufgenommenen Strom- und Spannungsdaten als Rohdaten von den Zellsensoren an das Batteriesteuergerät gesendet und auf einem externen Computer weiterverarbeitet. Dabei werden pro zu messenden Frequenzpunkt bis zu 1.900 Messwerte aufgenommen. Um die Messdaten zu übertragen, werden die Daten in einzelnen Paketen mit jeweils 25 Messdaten an das Batteriesteuergerät gesendet. Dabei entsteht eine Paketgröße von 54 Byte, bestehend aus 50 Byte an Messwerten plus 4 Byte Header [62]. Bei einer Übetragungsgeschwindigkeit von 100 kbps ergibt das eine Übertagungszeit von 4,32 ms pro Paket. Für die komplette Übertragung der 1.900 Messwerte dauerte es dann 164,2 ms. Für ein Beispiel aus der Elektromobilität stellt man fest, dass die Übertragungszeiten bei einer großen Anzahl an Batteriezellen stark ansteigt. So sind z.B. beim Opel Ampera insgesamt 288 Batteriezellen verbaut [66]. Kommen die Zellsensoren darin zum Einsatz, so beträgt die Übertragungszeit für alle Zellen bereits 47,29 s. Dabei ist auch nur die reine Übertragungszeit berechnet, ohne weitere Verarbeitungszeiten. Diese Zeit stellt also die minimale Übertragungszeit dar. Kritisch ist neben der Übertragungszeit, die Datenmenge die versendet wird. Praktische Erfahrungen haben gezeigt, dass es während der Übertragung zu einzelnen kurzen Übertagungsabbrüchen, bedingt durch Paketverluste, kommen kann. Diese können zwar beliebig oft wiederholt werden, stellen aber eine erhöhte Kanalbelastung dar. Eine einfache Lösung für das Problem ist die Reduzierung der Datenmenge.
Realisiert werden kann dies durch eine verteilte Signalverarbeitung der komplexen Spannung direkt auf dem Zellsensor. Dadurch müssen nicht mehr die kompletten 1.900 Messwerte übertragen werden, sondern nur noch der Real- und Imaginärteil der komplexen Spannung. Dies führt zu einer drastischen Reduzierung der Datenmenge sowie auch der Übertragungszeit. Im Falles des Opel Amperas beträgt die Übertragungszeit für alle 288 Zellen lediglich noch 138,24 ms anstatt der 47,29 s.
Es ist also sinnvoll, eine Signalverarbeitung direkt auf dem Zellsensor durchzuführen, anstatt zentral auf dem Batteriesteuergerät oder einem externen Computer. Um nun eine geeignete Berechnungsart zu finden, die für die Implementierung auf einem Mikrocontroller geeignet ist, werden im Folgenden mehrere Berechnungsarten vorgestellt und auf ihre Effizienz und deren Rechenaufwand hin verglichen.

Frequenzanalyse

Wie bereits in den mathematischen Grundlagen zur Impedanzspektroskopie in Abschnitt 2.2.2 gezeigt, lässt sich die komplexe Impedanz aus der Phasenverschiebung zwischen Strom und Spannung berechnen. Dazu müssen allerdings die aufgenommenen Strom- und Spannungsdaten vom Zeitbereich in den Frequenzbereich transformiert werden.
Aus der Signalverarbeitung sind verschiedene Methoden bekannt, wie sich Signale aus dem Zeitbereich in den Frequenzbereich transformieren lassen. Eine klassische Methode in der Signalverarbeitung ist die kontinuierliche Fourier Transformation [50].

$$\underline{X}(\omega) = \mathcal{F}\{x(t)\} = \int_{-\infty}^{+\infty} x(t) \cdot e^{-j\omega t} \, \mathrm{d}t. \tag{3.16}$$

Diese Transformation erlaubt es, ein kontinuierliches Signal in ein kontinuierliches Spektrum zu überführen [53]. Wird diese Transformation mit den gemessenen Strom- und Spannungsdaten durchgeführt, erhält man durch Division der Frequenzspektren das Spektrum der gesuchten Impedanz $\underline{Z}(\omega)$ [4].

$$\underline{Z}(\omega) = \frac{\mathcal{F}\{u(t)\}}{\mathcal{F}\{i(t)\}} \tag{3.17}$$

Da es sich bei den gemessenen Signalen aber um diskrete Messwerte handelt, müssen andere Algorithmen angewandt werden, die zur Verarbeitung diskreter Signale geeignet sind. Im folgenden Abschnitt werden daher drei Verfahren vorgestellt, mit denen eine Frequenzanalyse mit diskreten Signalfolgen möglich ist.

3.6.1. Diskrete Fourier-Transformation - DFT

Mit der diskreten Fourier Transformation (DFT) ist es möglich, zeitdiskrete Signale in ein frequenzdiskretes Spektrum zu transformieren.

Für kontinuierliche Signale gilt die DTFT:

$$\underline{X}\left(e^{j\omega}\right) = \sum_{k=-\infty}^{\infty} x(k) \cdot e^{-j\omega k} \tag{3.18}$$

Das Frequenzspektrum ist dabei eine kontinuierliche Funktion von ω und ist periodisch mit 2π. Für ein diskretes Signal mit der endlichen Länge 0 ≤ k ≤ N-1 lässt sich die ω-Achse des Frequenzspektrums zwischen 0 ≤ω≤ 2π diskretisieren, indem die diskreten Frequenzspektren mit $\omega = \frac{2\pi n}{N}$ in den Grenzen 0 ≤ n ≤ N-1 berechnet werden.

$$\underline{X}\left[n\right] = \underline{X}\left(e^{j\omega}\right)|_{2\pi n/N} = \sum_{k=0}^{N-1} x\left[k\right] \cdot e^{-2j\pi nk/N} \tag{3.19}$$

Die DFT liefert ein komplexes Frequenzspektrum, welches in einen Real- und einen Imaginärteil aufgeteilt werden kann.

$$Re\left\{\underline{X}\left(\omega\right)\right\} = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cdot cos\left(\frac{2\pi n}{N}\right) \tag{3.20}$$

$$Im\left\{\underline{X}\left(\omega\right)\right\} = \frac{1}{N}\sum_{n=0}^{N-1} x_n \cdot sin\left(\frac{2\pi n}{N}\right) \tag{3.21}$$

Für die Berechnung der N DFT-Koeffizienten sind jeweils N komplexe Multiplikationen der Folgenelemente mit den komplexen Faktoren und N Additionen der Multiplikationsprodukte notwendig. Der Rechenaufwand der direkten Form der DFT steigt quadratisch mit der Transformationslänge an [73]. Das bedeutet, für eine N-Punkt DFT benötigt die direkte Form der DFT N^2 komplexe Multiplikationen.

3.6.2. Fast Fourier-Transformation - FFT

Die Fast Fourier-Transformation (FFT) gehört zu den Standardverfahren der digitalen Signalverarbeitung. Es ist ein von Cooley und Tukey 1965 entwickeltes Verfahren zur effektiveren Berechnung der DFT. Die Grundlage des Algorithmus besteht darin, die Berechnung der DFT der Länge N in zwei unterschiedlichen Berechnungen der Länge N/2 aufzuteilen. Dabei muss die Anzahl der aufgenommenen Abtastwerte einer Zweierpotenz entsprechen. Die Teilergebnisse dieser beiden Berechnungen werden dann wieder zu einem Ergebnis zusammengeführt. Der am häufigsten verwendete Algorithmus, der in der digitalen Signalverarbeitung verwendet wird, ist die Radix-2-FFT [73]. Der Vorteil der FFT besteht darin, dass das exponentielle Wachstum des Rechenaufwandes der normalen DFT durch ein lineares Wachstum der FFT ersetzt wird. Eine N-Punkt Radix-2-FFT benötigt dabei $N \cdot \log_2(N)$ komplexe Multiplikationen und die gleiche Anzahl an komplexen Additionen [44] für eine Berechnung der DFT. Die Aufteilung der Berechnung wird bei der Radix-2-FFT in eine gerade und eine ungerade Teilberechnung aufgeteilt [73].

$$X\left[k\right] = \sum_{n=0}^{N-1} x\left[n\right] \cdot W_N^{nk} = \sum_{n=0,2,...}^{N-2} x\left[n\right] \cdot W_N^{nk} + \sum_{n=1,3,...}^{N-1} x\left[n\right] \cdot W_N^{nk} \tag{3.22}$$

Mit der Substitution der Indizes

n = 2m für n gerade

n = 2m+1 für n ungerade

M = N/2

ergibt sich die Gleichung zu:

$$X[k] = \sum_{m=0}^{M-1} x[2m] W_N^{2mk} + \sum_{m=0}^{M-1} x[2m+1] W_N^{[2m+1]k} \tag{3.23}$$

Umgeformt ergeben sich zwei Transformationen der Länge N/2

$$X[k] = \sum_{m=0}^{M-1} x[2m] W_N^{mk} + W_N^k \cdot \sum_{m=0}^{M-1} x[2m+1] W_N^{[m]k} \tag{3.24}$$

Eine N-Punkt DFT wird also zerlegt in zwei N/2-Punkt DFTs. Dadurch entsteht eine Zeitersparnis im Gegensatz zu der konventionellen Berechnungsmethode der DFT.

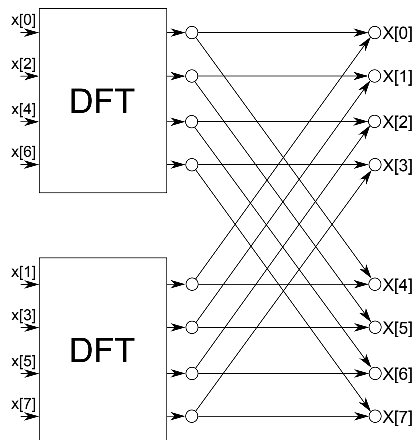


Abbildung 3.21.: Signalflussdiagramm einer Radix-2-FFT mit der Länge N = 8

Allerdings hat auch diese Methode der DFT-Berechnung Nachteile. So ist die Berechnung der DFT nicht während der Laufzeit möglich, da alle Abtastwerte für die Berechnung vorhanden sein müssen. Daraus ergibt sich der zweite Nachteil, dass ein relativ großer Speicher vorhanden sein muss, um alle Abtastwerte speichern zu können.

3.6.3. Goertzel-Algorithmus

Eine effektive Art zur Berechnung einzelner Spektrallinien ist der 1958 von Gerald Goertzel publizierte Goertzel-Algorithmus [21]. Dieser wird beispielsweise in der Fernmeldetechnik angewandt, um das DTMF Wahlverfahren zu realisieren [7] [3], da dort auch nur einzelne Spektrallinien für die Auswertung benötigt werden. Der rechnerische Vorteil des Algorithmus gegenüber der DFT liegt darin, dass der Goertzel-Algorithmus die Anzahl der Multiplikationen ungefähr um den Faktor zwei reduziert [40]. Dieser Vorteil resultiert aus der im Algorithmus verwendeten Periodizität. Durch die Reduzierung der Berechnungsschritte, im Vergleich zur konventionellen DFT, ist dieser Algorithmus sehr gut geeignet für die Implementierung in einem DSP bzw. Mikrocontroller.

Ein Nachteil des Goertzel-Algorithmus ist laut [42] seine Anfälligkeit gegenüber numerischer Ungenauigkeit und den daraus entstehenden Fehlern.

Um den Aufbau der Filterstruktur zu verstehen, wird hier die Herleitung nachvollzogen [43]. Zunächst wird die DFT Grundgleichung aufgestellt.

$$X[k] = \sum_{r=0}^{N-1} x[r] \cdot W_N^{nk} \tag{3.25}$$

W_N^{nk} beschreibt dabei den Twiddle-Faktor: $W_N^{nk} = e^{-j(2\pi kn/N)}$. Aufgrund der Periodizität des Twiddle-Faktors [40] [54] für $k \in \mathbb{N}$ gilt

$$W_N^{-Nk} = e^{j(2\pi kN/N)} = e^{j(2\pi k)} = 1 \tag{3.26}$$

Das heißt, dass W_N^{-Nk} auch an die Gleichung 3.25 multipliziert werden kann, ohne das Ergebnis der Gleichung zu verändern. Die Funktion lässt sich nun wie folgt beschreiben:

$$X[k] = W_N^{-Nk} \cdot \sum_{r=0}^{N-1} x[r]\, W_N^{nk}$$

$$X[k] = \sum_{r=0}^{N-1} x[r] \cdot W_N^{[N-nk][-k]} \tag{3.27}$$

Um die Ausgangsfolge des Filters zu bestimmen, wird die Gleichung 3.27 mit der Folge $u(n-r)$ multipliziert. Es ergibt sich daraus die folgende Ausgangsgleichung:

$$y_k[n] = \sum_{r=-\infty}^{+\infty} x[r]\, W_N^{[N-nk][-k]} u[n-r] \tag{3.28}$$

Durch die Beschränkung x[n] = 0 ≤ n ≤ N-1 kann die Gleichung als Faltung mit der Folge $W_N^{[N-nk][-k]} u[n-r]$ verstanden werden [54]. Deshalb kann $y_k[n]$ als die endliche Ausgangsfolge des Filters angesehen werden. Aus den beiden Gleichungen 3.27 und 3.28 und durch die Beschränkungen, dass für x[n] gilt 0 ≤ n ≤ N-1, lässt sich die folgende Gleichung aufstellen:

$$X[k] = y_k[n]\,|_{n=N} \tag{3.29}$$

Dies bedeutet, dass der Wert der Diskreten Fourier Transformation $X[k]$, der Ausgangswert $y[n]$ des Filters ist, wenn $n = N$ ist [23] [73] [54]. Die Gleichung 3.27 lässt sich auch einfach in eine rekursive Differenzengleichung überführen [23]:

$$y_k[m] = W_N^{-k} y[n-1] + x[n] \tag{3.30}$$

Mit den Koeffizienten $a_1 = -W_N^{-k}$ und $b_0 = 1$ lässt sich die Übertragungsfunktion des Goertzel-Filters 1. Ordnung aufstellen:

$$H[z] = \frac{1}{1 - W_N^{-k} z^{-1}} \tag{3.31}$$

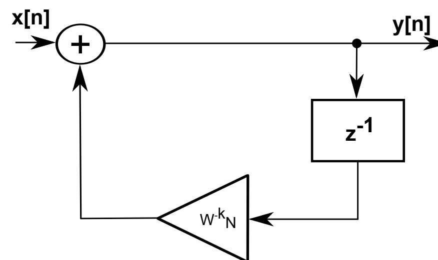


Abbildung 3.22.: Filterstruktur des Goertzel-Algorithmus 1. Ordnung

Das in der Abbildung 3.22 gezeigte Signalflussdiagramm zeigt noch nicht die effizienteste Umsetzung des Goertzel-Algorithmus. Für die Berechnung der N-ten Spektrallinie sind in dieser Form insgesamt 2N komplexe Multiplikationen und 2N komplexe Additionen notwendig. Dies entspricht 4N reellen Multiplikationen und 4N reellen Additionen [54]. Damit ist der Goertzel-Algorithmus 1. Ordnung sogar noch etwas schlechter als die direkte Berechnung durch die DFT. Ein Vorteil dieser Implementierung ist jedoch, dass die Berechnung des Filterkoeffizienten $a_1 = -W_N^{-k}$ hier nur einmal durchgeführt werden muss und für alle anderen Rechenschritte genutzt werden kann.
Eine effizientere Art des Goertzel-Algorithmus zeigen die folgenden Schritte. Dabei lässt sich die Anzahl der Multiplikationen um den Faktor 2 reduzieren. Nimmt man die Übertragungsfunktion der Gleichung 3.31 und erweitert diese mit dem komplex konjugierten

Nenner der Gleichung, lassen sich die Nennerkoeffizienten der Übertragungsfunktion reell machen [73].

$$H\left[z\right] = \frac{1}{1 - W_N^{-k} \cdot z^{-1}} = \frac{1}{1 - e^{(j2\pi k/N)} \cdot z^{-1}} \tag{3.32}$$

$$H\left[z\right] = \frac{1 - e^{(-j2\pi k/N)} \cdot z^{-1}}{\left(1 - e^{(j2\pi k/N)} \cdot z^{-1}\right) \cdot \left(1 - e^{(-j2\pi k/N)} \cdot z^{-1}\right)} \tag{3.33}$$

$$\frac{1 - e^{(-j2\pi k/N)} \cdot z^{-1}}{1 - 2cos\left(\frac{2\pi k}{N}\right) \cdot z^{-1} + z^{-2}} \tag{3.34}$$

mit $e^{(-j2\pi k/N)} = W_N^k$ folgt

$$= \frac{1 - W_N^k z^{-1}}{1 - 2cos\left(\frac{2\pi k}{N}\right) z^{-1} + z^{-2}}$$

Für die Realisierung der Filterstruktur wird die Direktform II gewählt. Der Vorteil dieser Form ist, dass beinahe die komplette Berechnung im rekursiven Teil des Filters stattfindet. Dabei kommt es im rekursiven Teil des Filters lediglich zu 4N reellen Additionen und 2N reellen Multiplikationen. Zusätzlich wird noch zum Zeitpunkt N=m die komplexe Multiplikation mit -$\mathrm{W}_\mathrm{N}^\mathrm{k}$ ausgeführt. Dies bedeutet nochmals zusätzliche vier reelle Multiplikationen und vier reelle Additionen. Damit ergibt sich der Gesamtaufwand für die Berechnung auf 2(N+2) reelle Multiplikationen und 4(N+1) reelle Additionen [54]. Das System kommt jetzt im Vergleich zu der Struktur 1. Ordnung mit ungefähr der Hälfte der Multiplikationen aus. Abbildung 3.23 zeigt die aus der Übertragungsfunktion resultierende rekursive Filterstruktur 2. Ordnung.

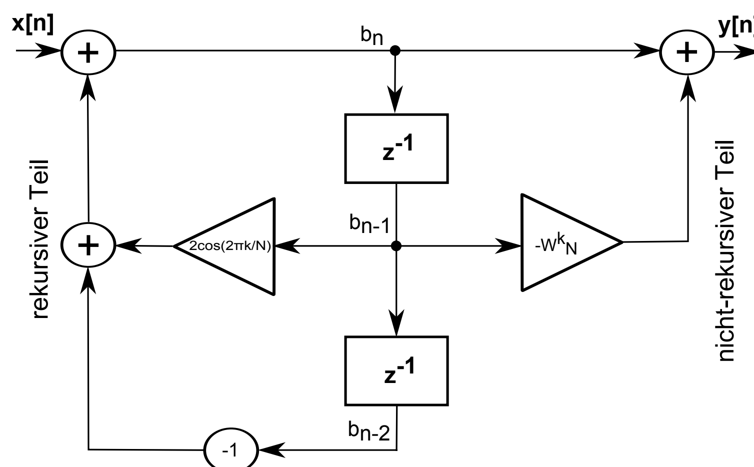


Abbildung 3.23.: Filterstruktur des Goertzel-Algorithmus (nach [64])

Die Ausgangsgleichung für y_k kann also wie folgt geschrieben werden:

$$y_k\left[m\right]|_{m=N} = X\left[k\right] = b_n - b_{n-1} \cdot W_N^k \tag{3.35}$$

Vergleich der Berechnungsarten

Hier sollen nun die vorgestellten Berechnungsverfahren hinsichtlich ihres Rechenaufwandes miteinander verglichen werden.

Tabelle 3.6.: Aufwandsvergleich der DFT-Berechnung

	DFT	Radix-2-FFT	Goertzel (pro Spektrallinie)
Multiplikationen	N^2 komplex	$N \cdot log_2 N$ komplex	$2(N+1)$ reell
Additionen	$N\,(N-1)$ komplex	$N \cdot log_2 N$ komplex	$4(N+1)$ reell

Es wird angenommen, dass innerhalb einer Recheneinheit pro zu berechnender komplexer Addition/Multiplikation zwei Taktzyklen und für eine reelle Addition/Multiplikation ein Taktzyklus benötigt werden. Dadurch lassen sich die einzelnen Berechnungsarten hinsichtlich ihrer Rechenlaufzeit miteinander vergleichen.

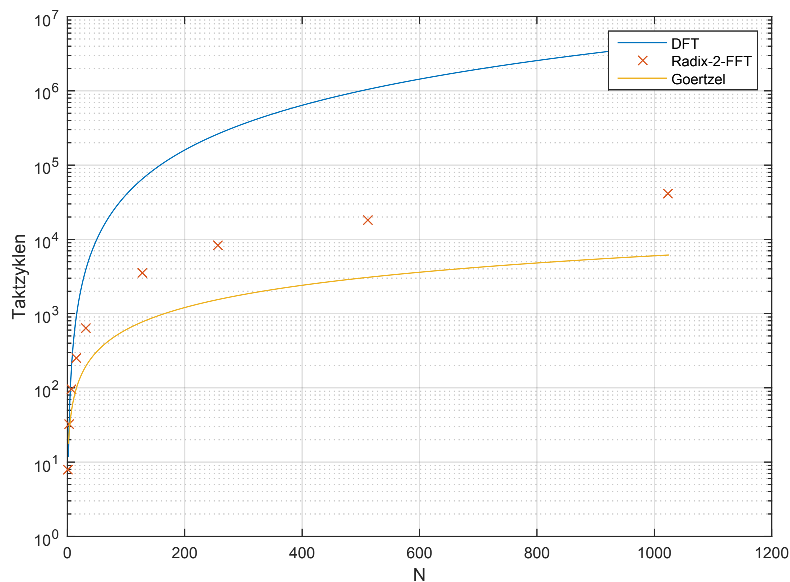


Abbildung 3.24.: Vergleich der Berechnungszeiten der Standard DFT, Radix-2-DFT und des Goertzel-Algorithmus

Die Abbildung 3.24 zeigt deutlich die unterschiedlichen Rechenlaufzeiten der einzelnen Berechungsarten (Radix-2-FFT ist nur möglich für $N = 2^p$ mit $p \in \mathbb{N}$). Dabei ist

allerdings hinzuzufügen, dass bei der Berechnung durch die konventionelle DFT und der Radix-2-FFT die gesamten Spektrallinien berechnet werden. Dagegen wird bei dem Goertzel-Algorithmus nur eine Spektrallinie berechnet. Da bei der Impedanzspektroskopie im einfachsten Fall lediglich eine Spektrallinie berechnet werden muss, stellt der Goertzel-Algorithmus hinsichtlich seines Rechenaufwands aber das beste Verfahren dar. Man muss bei der Auswahl des Algorithmus stark darauf achten, für welchen Anwendungsfall man diesen einsetzen möchte. Müssen beispielsweise alle N Spektrallinien einer Folge berechnet werden, so benötigt man bei der Verwendung des Goertzel-Algorithmus ungefähr 2N reelle Multiplikationen und $2N^2$ reelle Additionen. Dies ist zwar immer noch geringfügig effektiver als die konventionelle Berechnung der DFT Werte, allerdings ist der Berechnungsaufwand auch quadratisch von N abhängig. In diesem Fall ist dann die Berechnung mit der Radix-2-FFT vorzuziehen.
In [42] wird angegeben, dass sich der Goertzel-Algorithmus lohnt, wenn weniger als $\frac{5}{6} \cdot \log_2(N)$ Spektrallinien berechnet werden müssen. Werden mehr Spektrallinien gesucht, ist die Radix-2-FFT vorzuziehen.

3.7. Abschätzung und Zusammenfassung der Analyse

Im vergangenen Kapitel wurde ein Überblick über die Mindestanforderungen der funksynchronisierten Impedanzspektroskopie mit Zellsensoren geschaffen. Des Weiteren wurden verschiedene Konzepte für die Realisierung der Messmethode erarbeitet. Hier werden diese Konzepte nochmals zusammengetragen, um eine endgültige Neukonzeption des Messsystems zu erhalten. Ein Überblick über die erarbeiteten Mindestanforderungen für die Impedanzspektroskopie gibt die Tabelle 3.7.

Tabelle 3.7.: Mindestanforderung der Impedanzspektroskopie

	Mindestanforderung
Frequenzbereich	100 mHz - 2 kHz
Impedanzbereich	min. 1 mΩ
Phasengenauigkeit	1 °
Impedanzgenauigkeit	± 1 %
Spannungsbereich	2,0 V - 3,4 V

Aus diesen Anforderungen wird ersichtlich, dass ein Redesign des Zellsensors nötig ist. Zur Spannungsaufnahme der Messspannung wurden zwei Konzepte vorgestellt. Zum einen

die Spannungsaufnahme mit einem hochauflösenden AD-Wandler und zum anderen die Aufnahme durch das analog vorverarbeitete Spannungssignal mit dem, im Mikrocontroller integrierten AD-Wandler.
Es wurde entschieden, das Konzept mit der analogen Vorverarbeitung und der Spannungsaufnahme mittels internem AD-Wandler zu wählen. Des Weiteren soll aber die Messung durch einen hochauflösenden AD-Wandler als zusätzliche Alternative realisiert werden. Diese soll allerdings nur zum Vergleich dienen und als sekundäre Lösung angesehen werden. Realisiert wird die Messung mit dem hochauflösenden AD-Wandler als zusätzliche Aufsteckplatine.

Hinsichtlich der Software muss über die Komprimierung der Messdaten nachgedacht werden, um die vorhandenen Speicher-Ressourcen des Mikrocontrollers optimal ausnutzen zu können. Außerdem ist die Implementierung des Goertzel-Algorithmus auf den Zellsensoren wie auch dem Batteriesteuergerät zu realisieren.

Neben dem Redesign des Zellsensors soll auch das Batteriesteuergerät neu entwickelt werden. Es sollte ein deutlich leistungsstärkerer Mikrocontroller zum Einsatz kommen, mit dem auch die zeitkritischen Aufgaben der funksynchronisierten Impedanzspektroskopie verarbeitet werden können. Darüber hinaus muss das Batteriesteuergerät, neben der Möglichkeit mit den Zellsensoren zu kommunizieren, auch die Möglichkeit der Strommessung besitzen. Dies ist mit dem aktuell verwendeten Batteriesteuergerät nicht möglich.

4. Neuentwicklung und Redesign der bestehenden Hardware

In diesem Kapitel soll die Neuentwicklung und das Redesign der bestehenden Hardware vorgestellt werden. Dabei soll zum einen ein leistungsstärkeres Batteriesteuergerät mit einer drahtlosen Verbindungsschnittstelle zu den Zellsensoren und einer Möglichkeit der Strommessung entwickelt werden.
Zum anderen wird ein Redesign des Zellsensors durchgeführt. Darauf soll die in Abschnitt 3.5.2 vorgestellte analoge Vorverarbeitung realisiert und in das bestehende Hardwarekonzept eingefügt werden. Eine zusätzliche Möglichkeit zur Messung der Impedanzspektroskopie soll durch die Entwicklung einer Erweiterungsplatine für den Zellsensor möglich sein, welche die Messung mittels des hochauflösenden AD-Wandlers durchführen kann.

4.1. Entwicklung des Batteriesteuergeräts

Für eine schnelle Entwicklung eines Batteriesteuergeräts wurde entschieden, dieses auf der Basis eines Evaluations-Boards aufzubauen. Dies hat, neben der schnellen Verfügbarkeit, den Vorteil, dass diese meist kostengünstig erhältlich sind und sie eine Vielzahl von Anschlussmöglichkeiten besitzen. Die Mindestanforderungen an das Evaluations-Board sind dabei klar gestellt. Es muss die Möglichkeit besitzen, zwei Erweiterungsplatinen ansprechen zu können und über genügend Rechenleistung verfügen, um die zeitkritischen Anforderungen der Impedanzspektroskopie zu erfüllen, sowie die Signalverarbeitung der Messdaten durchführen zu können.
Auf der Suche nach einem geeigneten Evaluations-Board, fiel die Wahl auf ein leistungsstarkes ARM Cortex-M4 Board von Texas Instruments, welches im Folgenden näher vorgestellt wird.

4.1.1. Basismodul des Batteriesteuergeräts

Als Basismodul für das Batteriesteuergerät wurde das Tiva C Series TM4C1294 Connected LaunchPad von Texas Instruments (Abbildung 4.1) ausgewählt. Dieses erfüllt alle Anforderungen, die an das Batteriesteuergerät gestellt werden und bietet darüber hinaus weitere Anschlussmöglichkeiten, sodass weitere Entwicklungen und Erweiterungen auf dessen Basis möglich sind. Auf dem Board kommt ein leistungsstarker ARM Cortex-M4 zum Einsatz, der mit einer Taktgeschwindigkeit von bis zu 120 MHz betrieben werden kann. Zudem besitzt das Board die Möglichkeit, zwei Erweiterungsmodule mit jeweils 40 Ein- /Ausgänge anzuschließen. Des Weiteren werden sämtliche Ein- /Ausgängen des Mikrocontrollers heraus geführt, sodass diese mit externen Bauelementen bzw. Schaltungen verbunden werden können.

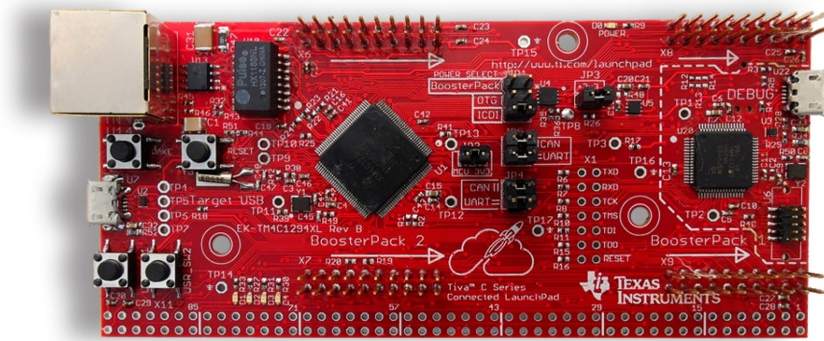


Abbildung 4.1.: Tiva C Series TM4C1294 Connected LaunchPad Evaluation Board (ent. aus[36])

Das Evaluation-Board bietet sich somit als ideale Entwicklungsplattform für das Batteriesteuergerät an.

- leistungsstarker 120 MHz 32 Bit ARM Cortex-M4 CPU mit Floating-Point Einheit
- 1 MB Flash, 256 kB SRAM, 6 kB EEPROM
- 8 x 32-bit Timer
- 12 Bit 2 MSPS AD-Wandler
- 2 x 40 Pin Erweiterungssteckplätze
- Spannungsversorgung von 3,3 V und 5,0 V

4.1.2. Transceiver-Erweiterungsmodul

Um eine Kommunikationsschnittstelle zwischen Batteriesteuergerät und den Zellsensoren herzustellen zu können, muss ein Transceivermodul entwickelt werden. In den Vorarbeiten [62] und [14] wird der Transceiver CC1101 von Texas Instruments verwendet. Aufgrund der bereits vorhandenen Erfahrungen mit diesem Baustein und der vorhandenen Software wurde entschieden, diesen auch wieder am neuen Batteriesteuergerät einzusetzen.

Durch den großzügig vorhandenen Platz von 50 mm x 28 mm auf dem Erweiterungsmodul, konnte beim Entwurf des PCB-Layouts auf das Referenzdesign des CC1101 von Texas Instruments zurückgegriffen werden [38]. Dadurch konnte das optimale Layout für das RF-Design gewählt werden. Dies besteht aus einem Anpassungsnetzwerk aus Induktivitäten und Kapazitäten, welche die Eingangsimpedanz des CC1101 an die Antennenimpedanz von 50 Ω anpasst. Die genauen Werte des Anpassungsnetzwerks können dem Schaltplan des Erweiterungsmoduls im Anhang K entnommen werden.
Als Taktquelle besitzt der CC1101 Transceiver einen externen 26 MHz Quarz mit den entsprechenden Lastkondensatoren. Der Transceiver kommuniziert mit dem Mikrocontroller über eine, auf die seitlichen Steckerleisten herausgeführten, SPI-Schnittelle. Neben dieser Schnittstelle werden die beiden frei konfigurierbaren Ein-/Ausgänge GDO0 und GDO2, welche für Interrupt-gesteuerte Meldungen wie das Empfangen eines Paketes genutzt werden, herausgeführt. Um diese Signale im Testbetrieb messen zu können, wurden zusätzliche Messpunkte auf der Platine vorgesehen. Die folgende Abbildung 4.2 zeigt das entworfene Layout des CC1101 Transceiver-Erweiterungsmoduls.

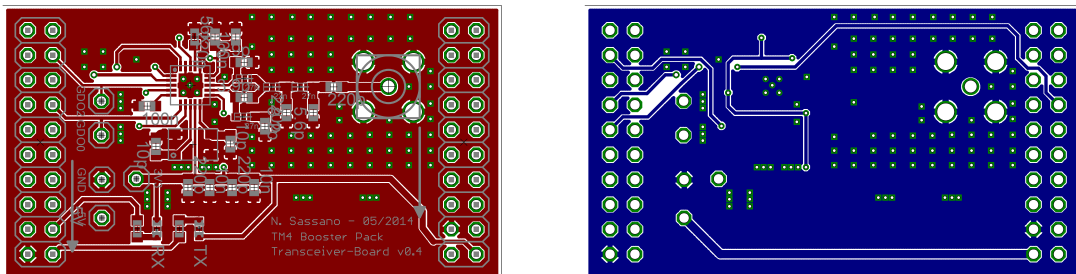


Abbildung 4.2.: Layout des CC1101 Transceiver-Erweiterungsmoduls
Links: Oberseite des Layouts. Rechts: Unterseite des Layouts

Darüber hinaus besitzt das Erweiterungsmodul zwei Sende- und Empfangs-LED die vom Mikrocontroller aus gesteuert werden und zur optischen Signalisierung vom Empfangen bzw. Senden von Datenpaketen dienen.

4.1.3. Strommess-Erweiterungsmodul

Die Realisierung der Strommessung erfolgt ebenfalls über ein Erweiterungsmodul, welches an das Batteriesteuergerät angeschlossen wird. Für die Strommessung soll ein Hall-Sensor eingesetzt werden, der in einem Chip integriert ist.

Gewählt wurde der Strommess-Sensor ACS716 von Allegro Microsystems. Dabei handelt es sich um einen Strommess-IC mit integriertem Hall-Sensor, der als Ausgabe eine lineare Spannung in Abhängigkeit vom durchflossenen Strom liefert. Dieser kann sowohl positive als auch negative Ströme messen. Die ausgegebene Spannung liegt dabei für positive Ströme im Bereich oberhalb und für negative Ströme unterhalb von 1,65 V. Diese Spannung kann vom AD-Wandler des Batteriesteuergeräts gemessen und in Stromwerte umgerechnet werden. Abbildung 4.3 zeigt das Layout des erstellten Erweiterungsmoduls. Gut zu erkennen sind dabei die beiden Stromanschlüsse des Erweiterungsmoduls. Diese befinden sich im unteren Teil der Platine. Die Stromkabel lassen sich durch eine Verschraubung (M6) mit dem Strommess-Erweiterungsmodul fest verbinden.

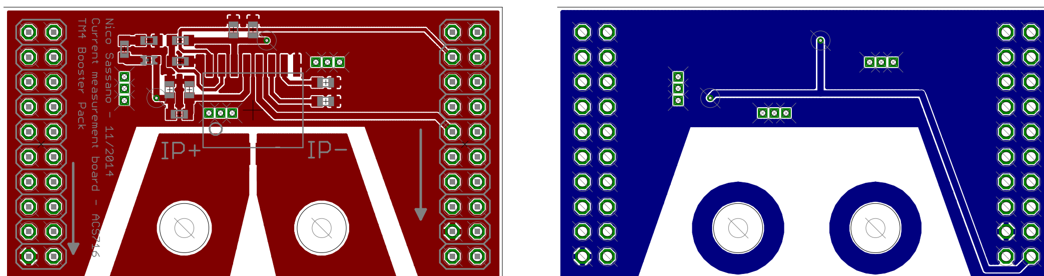


Abbildung 4.3.: Layout des ACS716 Strommess-Erweiterungsmoduls
Links: Oberseite des Layouts. Rechts: Unterseite des Layouts

Neben dem Strommess-IC besteht die restliche Beschaltung des Moduls aus Stützkondensatoren, einem Spannungsteiler für die Überstromeinstellung sowie einer Eingangsbandbreitenbegrenzung für das Strommessmodul. Über einen Enable-Eingang lässt sich die Messung des ACS712 ein- bzw. ausschalten. Dieser ist über die Steckerleiste zum Mikrocontroller geführt und lässt sich von diesem steuern. Ein genauer Schaltplan mit den Angaben der Dimensionierung findet sich im Anhang J.

Der Sensor ist in verschiedenen Strombereichen erhältlich, diese sind 6 A, 12 A sowie 25 A. Dabei unterscheiden sich die drei Varianten in der Strom-/Spannungsausgabe, die vom AD-Wandler des Mikrocontrollers gemessen wird. Dies ist bei der Berechnung der Stromwerte auf dem Batteriesteuergerät zu beachten.

4.1.4. Aufbau des Batteriesteuergeräts

Das Batteriesteuergerät besteht aus insgesamt drei Komponenten. Dem Transceiver-Modul, dem Strommess-Modul und dem Basismodul, welches die Steuerung der Erweiterungsmodule übernimmt. In Abbildung 4.4 wird der Aufbau des Batteriesteuergeräts und die einzelnen Verbindungen zu den Erweiterungsmodulen dargestellt.
Durch eine UART-Verbindung zwischen dem Batteriesteuergerät und einem externen PC, lassen sich Daten abrufen und Befehle an das Steuergerät senden.

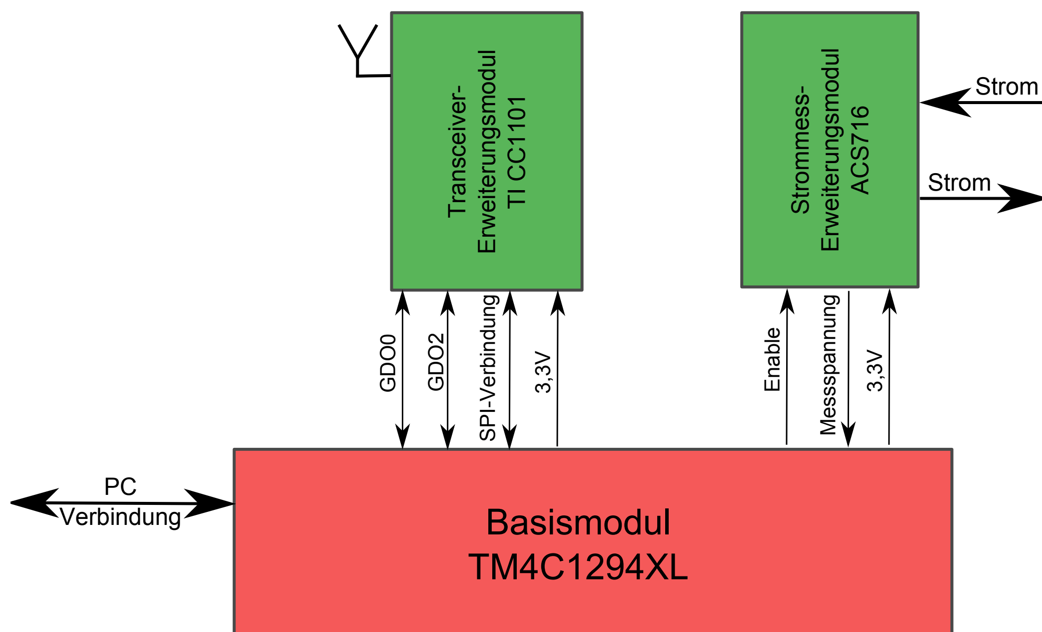


Abbildung 4.4.: Blockschaltbild des Batteriesteuergeräts

4.2. Redesign des Zellsensors

Im Laufe der Analyse hat sich gezeigt, dass ein Redesign des Zellsensors zwingend erforderlich ist, um die analoge Vorverarbeitung zu implementieren. Dabei soll die bisherige Hardware des Zellsensors beibehalten werden. Ebenfalls soll die Anschlussmöglichkeit für ein Erweiterungsmodul auf dem Zellsensor vorhanden bleiben.
Weiter soll der mechanische und elektrische Anschluss zur Verbindung des Zellsensors mit der Batteriezelle hinsichtlich der benötigten Fläche optimiert werden. Dadurch steht mehr Fläche für die Unterbringung der Bauteile zur Verfügung. Um eine weitere Optimierung der verfügbaren Fläche zu erhalten, wurde entschieden, sämtliche Standardbauelemente wie Widerstände, Induktivitäten sowie Kapazitäten in der Bauformgröße 0402 zu verwenden. Bisher wurden Standardbauelemente in der Größe 0603 verwendet.

4.2.1. Entwicklung der analoge Vorverarbeitung

Die analoge Vorverarbeitung wurde bereits in Kapitel 3.5.2 vorgestellt. Für die Realisierung der kompletten analogen Vorverarbeitung werden zwei Operationsverstärker, zwei Rheostaten sowie sechs Widerstände benötigt. Als Operationsverstärker wird ein OPA344 von Texas Instruments eingesetzt. Dabei handelt es sich um einen energiesparenden Operationsverstärker, der somit sehr gut in das Low-Power-Konzept des Zellsensors passt. Dieser Operationsverstärker ist zudem in einer Variante erhältlich, in der zwei Operationsverstärker integriert sind (OPA2344). Beide Operationsverstärker können somit platzsparend innerhalb eines Gehäuses auf dem Zellsensor untergebracht werden.
Als Rheostat wurde der AD5270 von Analog Devices gewählt. Dabei handelt es sich um einen elektronisch einstellbaren Widerstand, der sich über eine SPI-Schnittstelle in 1024 Stufen zwischen 0 Ω und seinem Nominalwert einstellen lässt. Der Rheostat AD5270 ist in drei verschiedenen nominalen Varianten 20 kΩ, 50 kΩ und 100 kΩ erhältlich.
Die Einstellung der analogen Vorverarbeitung bzw. der beiden Rheostaten ist abhängig vom Gleichspannungsanteil der Batteriespannung. Um die analoge Vorverarbeitung korrekt einstellen zu können, muss dem Mikrocontroller der Gleichspannungsanteil der Batteriespannung bekannt sein. Erfasst werden kann die Gleichspannung durch den internen 12 Bit AD-Wandler des Mikrocontrollers. Um den erforderlichen Spannungsbereich von 2,0 V bis 3,4 V durch den internen AD-Wandler des Mikrocontroller erfassen zu können, muss die Gleichspannung in den Referenzspannungsbereich (0,0 V bis 2,5 V) des Mikrocontrollers gebracht werden. Dies wird hier durch einen einfachen 2:1 Spannungsteiler realisiert, durch den dann eine Spannungserfassung im Bereich von 0,0 V bis 5,0 V mit dem internen 12 Bit AD-Wandler möglich ist. Der dadurch verursachte Auflösungsverlust durch den Spannungsteiler ist hier vernachlässigbar, da für diese Messung eine Genauigkeit im mV Bereich ausreichend ist. Die Abbildung 4.5 zeigt schematisch die Aufteilung der Zellspannung in zwei

verschiedene Pfade, die dann durch den internen 12 Bit AD-Wandler in verschiedenen Genauigkeiten gemessen werden kann.

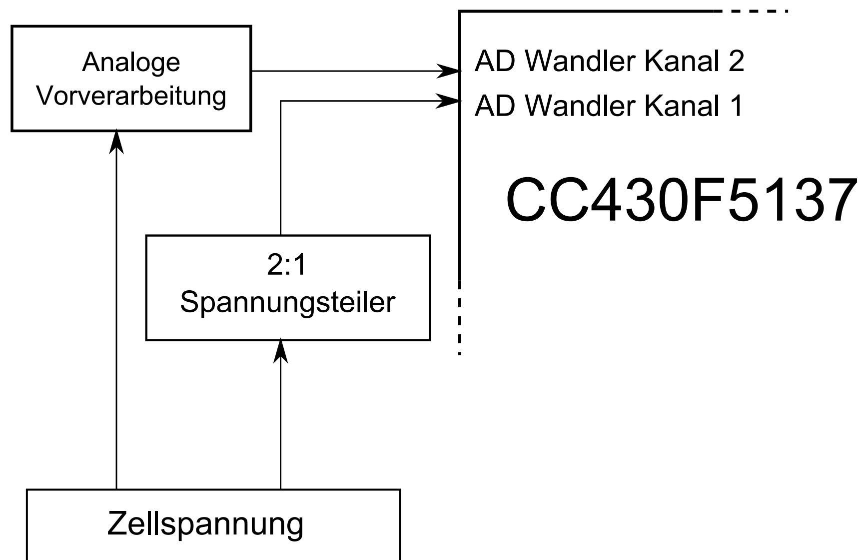


Abbildung 4.5.: Aufteilung der Zellspannung in zwei verschiedene Messpfade

Dimensionierung der analogen Vorverarbeitung

An dieser Stelle werden die Bauteile für die analoge Vorverarbeitung dimensioniert und ausgelegt. Es müssen die beiden Spannungsteiler für den Gleichspannungsabzug und die Verstärkung der analogen Vorverarbeitung dimensioniert und eine Wahl des Nominalwertes der Rheostaten vorgenommen werden.

Für die Dimensionierung der Spannungsteiler an der Subtrahierschaltung ist es wichtig zu wissen, wieviel Spannung maximal abgezogen werden soll. Da der Zellsensor nur über eine 3,3 V Spannungsversorgung verfügt, welche für den Spannungsabzug eingesetzt werden kann, kann auch maximal diese Spannung abgezogen werden. Da die Batteriespannung im unbelasteten Fall bis zu 3,4 V Spannung haben kann, ist also der maximal erreichbare Spannungsabzug durch die Subtrahierschaltung zu realisieren. In diesem Fall würde noch eine restliche Gleichspannung von 100 mV am Ausgang der Subtrahierschaltung existieren.
Es wurde sich dafür entschieden, den Spannungsteiler in einem Verhältnis von 1:100 aufzubauen. Realisiert wird dieser durch einen 1 kΩ Widerstand und dem Rheostaten mit 100 kΩ Nominalwert. Dadurch lässt sich ein hoher Spannungsabzug von bis zu 3,27 V erreichen. Der minimale Spannungsabzug ist dabei 0,0 V. Da ein Spannungsabzug von 0,0 V allerdings nicht nötig ist, wurde entschieden, einen zusätzlichen Widerstand von 1 kΩ nach dem Rheostaten einzufügen (siehe Abbildung 4.7 - R_2). Dadurch wird erreicht, dass auch bei der Einstellung des Rheostaten von 0 Ω, immer ein Spannungsabzug von 1,65 V vorhanden ist.

Dies hat den Vorteil, dass zum einen ein geringerer Strom fließt und zum anderen die Auflösung des Rheostaten erst bei 1,65 V beginnt und nicht bereits bei 0 V. Dadurch lässt sich der Spannungsabzug genauer einstellen. Abbildung 4.6 zeigt die einzelnen Auflösungsstufen des Rheostaten. Eine besonders gute Auflösung wird hier im oberen Spannungsbereich erreicht.

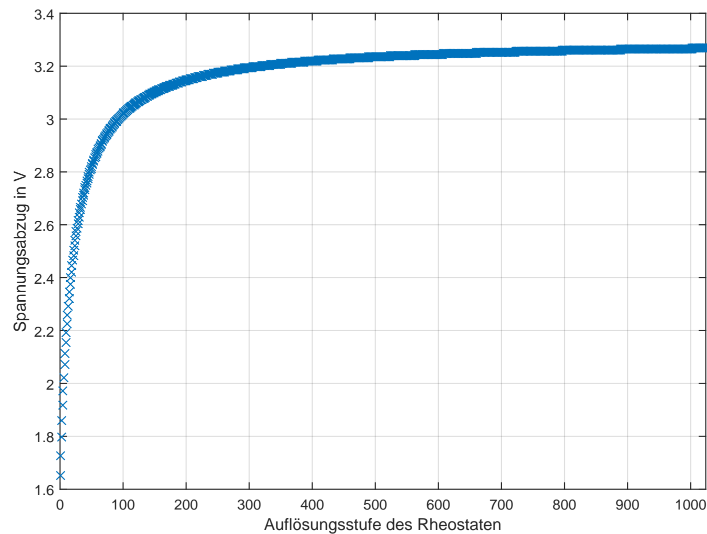


Abbildung 4.6.: Auflösungsstufen des Rheostaten

Die Abbildung 4.7 zeigt die Subtrahierschaltung mit dem zusätzlichen Widerstand hinter dem Rheostaten.

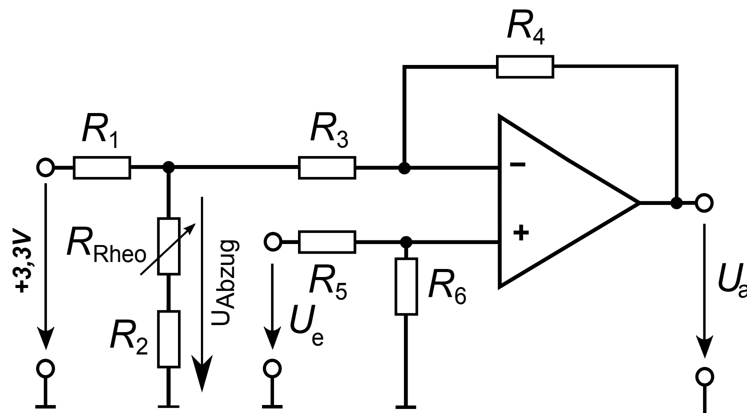


Abbildung 4.7.: Dynamisch einstellbare Subtrahierschaltung mit einem zusätzlichen Widerstand nach dem Rheostaten

Die einstellbaren Spannungen ergeben sich mit dem zusätzlichen Widerstand R_2 nach den folgenden Gleichungen:

$$U_{\text{max.}} = \frac{R_2 + R_{\text{Rheo}_{\text{max.}}}}{R_1 + R_2 + R_{\text{Rheo}_{\text{max.}}}} \cdot U_{\text{ref.}} = \frac{1\,k\Omega + 100\,k\Omega}{1\,k\Omega + 1\,k\Omega + 100\,k\Omega} \cdot 3,3\,V = 3,267\,V \quad (4.1)$$

$$U_{\text{min.}} = \frac{R_2 + R_{\text{Rheo}_{\text{min.}}}}{R_1 + R_2 + R_{\text{Rheo}_{\text{min.}}}} \cdot U_{\text{ref.}} = \frac{1\,k\Omega + 0\,\Omega}{1\,k\Omega + 1\,k\Omega + 0\,\Omega} \cdot 3,3\,V = 1,65\,V \quad (4.2)$$

Der Gleichspannungsabzug lässt sich somit in einem Bereich von $3,267\,V$ und $1,65\,V$ realisieren.

Für die Dimensionierung der Messverstärker lässt sich die Schaltung in Abbildung 4.8 annehmen.

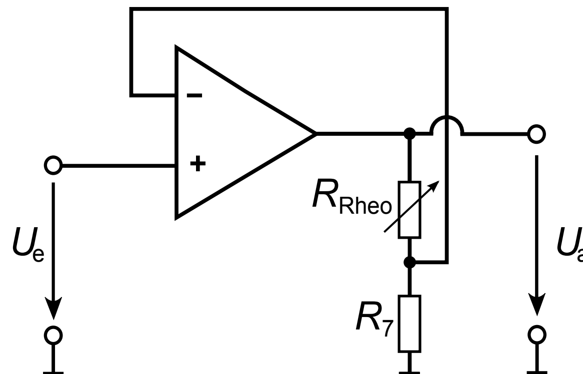


Abbildung 4.8.: Nichtinvertierender Messverstärker

Dabei kommen ein Operationsverstärker, ein Widerstand und ein Rheostat zum Einsatz. Diese Schaltung lässt sich einfach dimensionieren. Aus der Analyse ist bekannt, dass die Verstärkung mindestens den Verstärkungsfaktor 20,34 erreichen muss, um eine Messamplitude von 3 mV ausreichend verstärken zu können. Daher wurde entschieden, für die Verstärkerschaltung den Spannungsteiler so zu dimensionieren, dass ein maximaler Verstärkungsfaktor von V = 100 erreicht werden kann. Dazu wird wieder der Rheostat mit dem Nominalwert von $100\,\text{k}\Omega$ und ein Widerstand mit $1\,\text{k}\Omega$ gewählt. Dadurch lässt sich die Verstärkung linear zwischen den Verstärkungsfaktoren V = 1 und V = 100 in 1024 Stufen einstellen. Die Grenzen der Verstärkung berechnen sich dabei nach folgenden Gleichungen:

$$V_{\text{max.}} = 1 + \frac{R_{\text{Rheo}_{\text{max.}}}}{R_7} = 1 + \frac{100\,k\Omega}{1\,k\Omega} = 101 \quad (4.3)$$

$$V_{\text{min.}} = 1 + \frac{R_{\text{Rheo}_{\text{min.}}}}{R_7} = 1 + \frac{0\,\Omega}{1\,k\Omega} = 1 \quad (4.4)$$

Die Abbildung 4.9 zeigt die überarbeitete und dimensionierte analoge Vorverarbeitung mit zusätzlichem Widerstand hinter dem Rheostaten an der Subtrahierschaltung. Die komplette Schaltung besteht nun aus sieben Widerständen, zwei Rheostaten und zwei Operationsverstärkern und kann so auf den neuen Zellsensoren eingesetzt werden.

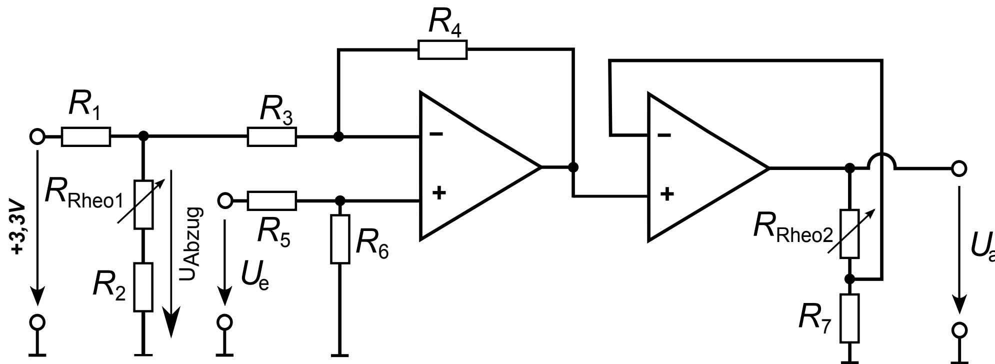


Abbildung 4.9.: Analoge Vorverarbeitung: R_1, R_2, R_7 = 1 kΩ und R_{Rheo1},R_{Rheo2},R_3-R_6 = 100 kΩ

4.2.2. Modifikationen des Schaltungsentwurfs

An dieser Stelle soll aufgezeigt werden, welche Änderungen sich in der Schaltung des Sensors, im Gegensatz zum vorhergehenden Zellsensor, ergeben haben. So wurden die Bauteilwerte der Anpassungsschaltung des RF-Moduls am eingesetzten Mikrocontroller abgeändert. Ausgetauscht wurden dabei zwei Induktivitäten. Diese Werte ergaben sich aus dem, von Texas Instruments vorgeschlagenen, Anpassungsnetzwerk für den Frequenzbereich bei 434 MHz [32]. Dabei wurden die Induktivitäten L303 und L304 ersetzt (Abbildung 4.10). Im vorhergehenden Sensor war L303 = 27 nH und L304 = 47 nH. Diese wurden nun zu L303 = 47 nH und L304 = 51 nH abgeändert.

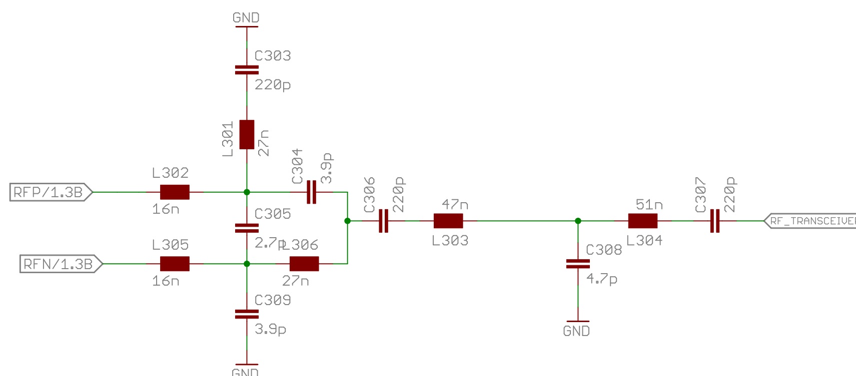


Abbildung 4.10.: Geändertes Anpassungsnetzwerk des CC430F5137

Eine weitere Modifikation der Schaltung ergab sich durch eine Softwareanpassung des Mikrocontrollers. In der Arbeit [48] wurde ein Problem beschrieben, dass die Signalisierung eines ankommenden Burstsignals nur über das Herausführen des demodulierten Signals und wieder Einspeisen in den Mikrocontroller als Interrupt ausgewertet werden kann. Dies konnte durch eine entsprechende Adressierung der Interrupt-Service-Routine (ISR) gelöst werden. Ein Vorteil ergibt sich dabei durch die zwei freiwerdenden Pins, die nun anderweitig genutzt werden können. Das Prinzip der Änderung ist in Abbildung 4.11 dargestellt.

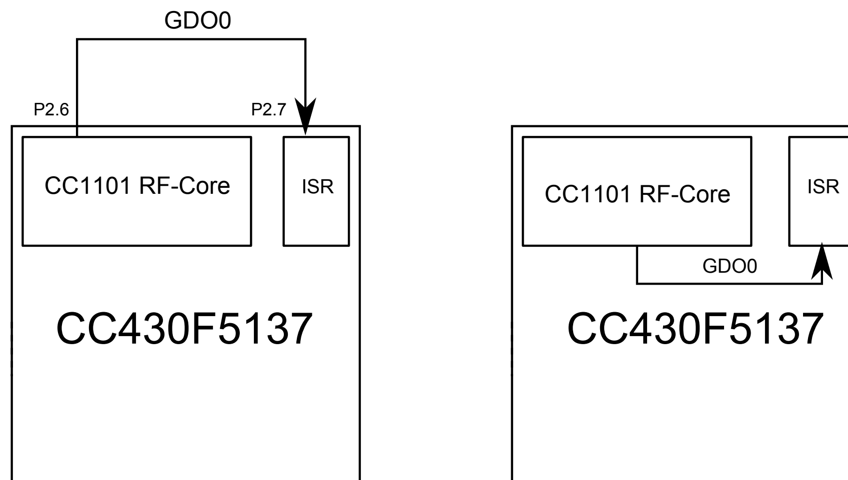


Abbildung 4.11.: Links: externe Interruptauslösung über Ein- und Ausgangeports am Mikrocontroller; Rechts: interne Interruptauslösung durch gezielte Adressierung der Interrupt Service Routine

Des Weiteren konnte ein Fehler an der JTAG-Schnittstelle behoben werden. Dieser Fehler wurde am vorherigen Sensor mittels eines Fädeldrahts korrigiert, um den Mikrocontroller programmieren zu können. Ursache dafür war eine falsch angeschlossene Signalleitung am Mikrocontroller. Die Signalleitung JTAG-Test wurde anstatt an Port 39, an Port 9 angeschlossen. Dies wurde im Redesign des Zellsensors berücksichtigt.

Darüber hinaus wurde ein Spannungsteiler entfernt, der für die Messung der 3,3 V Versorgungsspannung vorgesehen war. Diese Möglichkeit wurde in der Vergangenheit nicht genutzt und findet auch bei der Impedanzmessung keine Verwendung. Aus Platzgründen wurde deshalb entschieden, diesen zu entfernen.

Eine weitere Änderung ergibt sich durch die geplante Erweiterungsplatine, mit der die hochauflösende Messung realisiert werden soll. Dazu muss der Anschlussstecker abgeändert werden, der die Platine und den Zellsensor miteinander verbinden soll. Näheres dazu in Abschnitt 4.3.

Die Änderungen im Schaltungsentwurf umfassten die folgenden Punkte:

- Bauteilwerte des Anpassungsnetzwerts geändert
- GDO0 Interrupt-Signalisierung intern adressiert
- JTAG-Schnittstellenfehler behoben
- Entfernung der 3,3 V Spannungsmessung
- Änderung des Anschlusssteckers für die Erweiterungsplatine

Die restlichen Schaltungen konnten vom vorhergehenden Zellsensor übernommen werden. Die Tabelle 4.1 gibt einen Überblick über die eingesetzten IC-Bauelemente auf dem Zellsensor.

Tabelle 4.1.: Tabelle der eingesetzte IC-Bauelemente

	Bauteil	Bezeichnung	Hersteller
Steuerung	Mikrocontroller	CC430F5137	Texas Instrumens
analoge Vorverarbeitung	Operationsverstärker	OPA2344	Texas Instrumens
analoge Vorverarbeitung	Rheostat	AD5270-100	Analog Devices
Spannungsversorgung	BoostConverter	TPS61201	Texas Instrumens
RF-Design	Wake-Up Receiver	AS3930	AMS
RF-Design	Antennenumschalter	ADG918	Analog Devices
Temperaturmessung	Temperatursensor	TMP102	Texas Instrumens

Die nachfolgende Abbildung 4.12 gibt einen Überblick über die Verbindungen der einzelnen Funktionskomponenten.

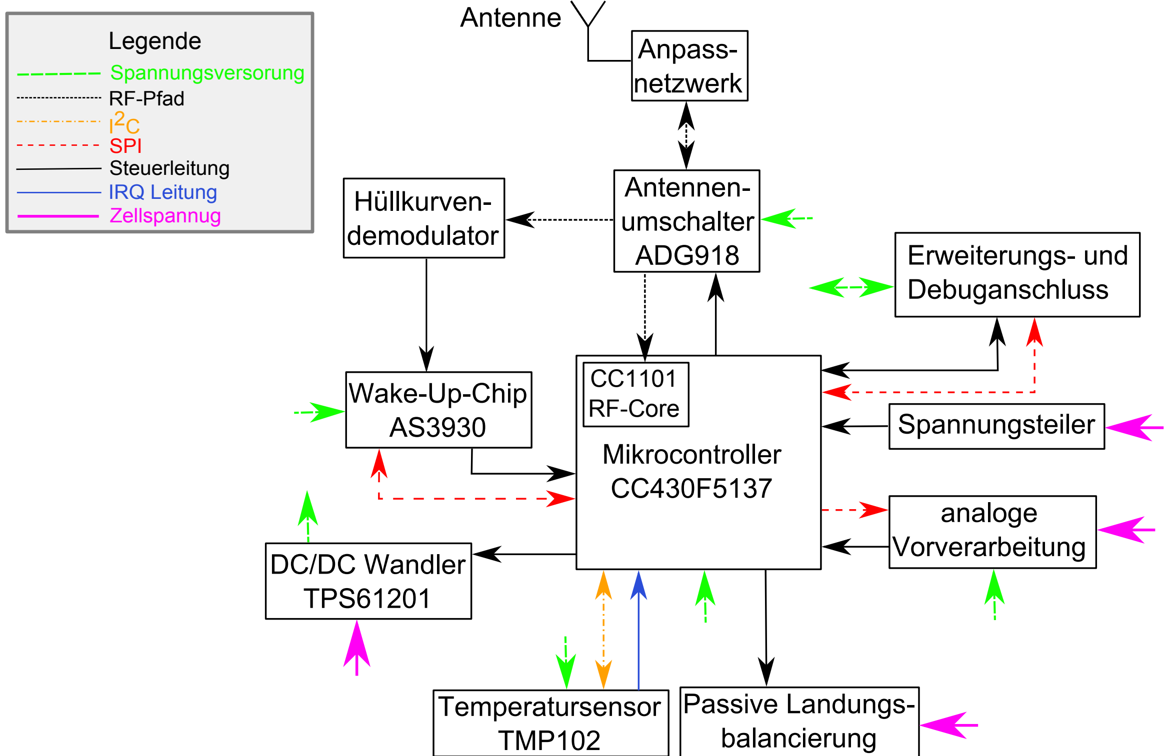


Abbildung 4.12.: Blockschaltbild des neu entwickelten Zellsensors

4.2.3. Mechanische- und elektrische Verbindung zwischen Zellsensor und Batteriezelle

Die mechanische- und elektrische Verbindungsfläche an den Platinen dient dazu, den Zellsensor auf dem Gehäuse einer ECC-LFPP 45Ah $LiFePO_4$-Zelle zu montieren und elektrisch zu Verbinden. Dieser Anschluss wurde bereits in der Arbeit [62] auf dem Zellsensor realisiert. Dimensioniert wurde die Verbindungsfläche für eine direkte Montage auf den Batteriezellen. Zwischenzeitlich wurde ein Befestigungsadapter entworfen, der die nötige Verbindungsfläche, die der Zellsensor aufweisen muss, wesentlich verkleinert hat. In Abbildung 4.13 ist die bisher vorhandene Verbindungsfläche mit der nun nötigen Kontaktfläche gegenübergestellt. Es zeigt sich, dass nun nicht mehr eine Kontaktfläche von Ø= 22 mm vorhanden sein muss. Durch den Befestigungsadapter reicht nun eine verkleinerte Verbindungsfläche von Ø= 11 mm aus.

Weitere Details über den Befestigungsadapter und die Zellmontage sind im Anhang G zu finden.

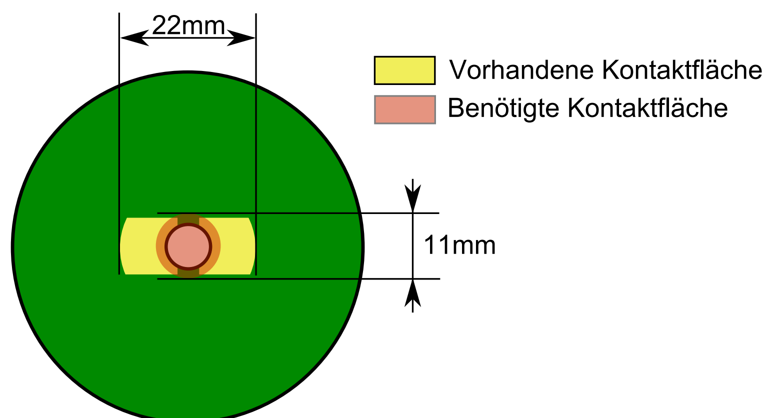


Abbildung 4.13.: Zu überarbeitende Kontaktfläche auf dem Zellsensor: Gegenüberstellung der vorhandenen Kontaktfläche zu der benötigten Kontaktfläche

4.2.4. Platinenlayout des Zellsensors

Durch das Einfügen der analogen Vorverarbeitung, der Änderung von Schaltungen sowie der Anpassung der Kontaktfläche ist es nötig, ein neues Layout des Zellsensor zu erarbeiten. Bei der Erstellung des Layouts wurde besonders auf das richtige Design der Hochfrequenzkomponenten geachtet. Aus den Erfahrungen der vergangenen Arbeiten war bekannt, dass das Design und die Platzierung der Bauteilkomponenten erheblichen Einfluss auf die Abstrahleigenschaften der Schleifenantenne haben kann.
Für das Layout wurden neben den Spezifikationen des PCB-Herstellers, eigene Designregeln festgelegt, die beim Platinenlayout einzuhalten sind.

• Es soll darauf geachtet werden, dass Bauelemente die zu einer Funktionsgruppe gehören, möglichst nahe beieinander liegen.

• Die Blockkondensatoren an den Versorgungsleitungen sollen so nah wie möglich an die entsprechenden IC gelegt werden. Beim Einsatz mehrerer Blockkondensatoren an einer Versorgungsleitung und einem IC ist darauf zu achten, dass die kleinere Kapazität sich näher am IC befindet.

• Die Leiterbahnbreite ist so zu dimensionieren, dass die Signal- und Versorgungsleitungen breit genug sind, um das Signal bzw. den Strom zu führen. Grundsätzlich ist die Leiterbahn so breit wie möglich zu dimensionieren, vorausgesetzt es werden damit keine anderen Komponenten bzw. die HF-Eigenschaften gestört.

• Der Pfad vom Anschluss der Batteriezelle an den Zellsensor über die analoge Vorverarbeitung bis zum AD-Wandler des Mikrocontrollers, soll so kurz wie möglich aufgebaut werden.

• Beim Layout der Signal- und Versorgungsleitungen ist eine parallele Leitungsführung zur Schleifenantenne, die eine Länge von ungefähr $\lambda/2$ aufweisen, zu vermeiden.

• Das Layout des HF-Anpassungspfads soll nach dem, vom Texas Instruments vorgeschlagenen Layout, aufgebaut werden [32].

Basierend auf diesen Designregeln kann ein Layout des Zellsensors erstellt werden. Um die Abstrahleigenschaften der Schleifenantenne zu bestimmen, wird eine FEM-Simulation des gesamten Platinenlayouts mit CST Microwave Studio durchgeführt. Dabei soll hauptsächlich der Einfluss von Leiterbahnen auf die Abstrahleigenschaften der Schleifenantenne untersucht und verbessert werden. Eine ausführliche Beschreibung der Simulation in CST Microwave Studio und der daraus resultierenden Ergebnisse wird im Anhang C gezeigt. Ergebnis dieser Simulation ist, dass einzelne Leiterbahnen verändert werden müssen. Besonders lange Versorgungsleitungen im Bereich von $\lambda/2$ der Sende- und Empfangsfrequenz von 434 MHz zeigen sich als problematisch hinsichtlich ihres Einflusses auf die Abstrahlei-

genschaften. Die Länge von $\lambda/2$ erstreckt sich hier im Bereich von ca. 16,29 cm und fällt somit in die Länge einiger Versorgungsleitungen auf dem Zellsensor.

$$\lambda/2 = \frac{1}{2} \cdot \frac{c_0}{f} \cdot \frac{1}{\sqrt{\epsilon_r}} = \frac{1}{2} \cdot \frac{300 \cdot 10^6\, m/s}{434 \cdot 10^6\, Hz} \cdot \frac{1}{\sqrt{4,5}} = 16,29\, cm \tag{4.5}$$

Durch die Simulation konnten einige Leiterbahnen identifiziert werden, die die Abstrahleigenschaften der Schleifenantenne beeinflussen. Diese wurden dann durch das Einbringen einer Induktivität für hochfrequente Signale unterbrochen.

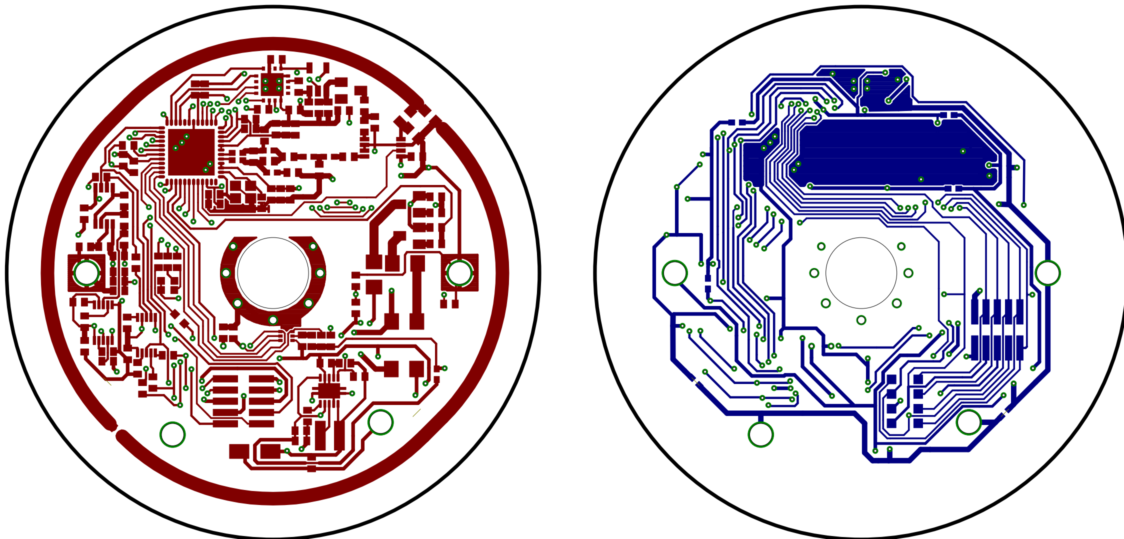


Abbildung 4.14.: Layout des neuen Zellsensors: Links: Layout der Oberseite. Rechts: Layout der Rückseite. Layout in höherer Auflösung in Anhang K.

In der Abbildung 4.14 ist zu sehen, dass die mechanische- und elektrische Kontaktfläche soweit überarbeitet wurde, dass nur noch der in Abschnitt 4.2.3 gezeigte Anschlussbereich an der Oberseite bestehen bleibt. Diese Fläche reicht aus, um den Sensor mit dem Befestigungsadapter zu montieren.

Das Design der äußeren Schleifenantenne bleibt unverändert. Wie auch die rechts und links, auf mittiger Höhe, liegenden zusätzlichen Spannungsversorgungs-Anschlüsse. Des Weiteren wurde die Position des Nadeladapter-Programmierpins beibehalten, welche sich auf der Unterseite des Zellsensors befinden. Gefertigt wird der Zellsensor als zweilagige Platine mit einer Kupferdicke von 35μm. Die Substrathöhe des Platinenmaterials, beträgt dabei 1,6 mm.

4.3. Hochauflösende AD-Wandler-Erweiterungsplatine

In die Schaltungsentwicklung ist auch, ist die Entwicklung der Erweiterungsplatine für die hochauflösende AD-Wandler Messung eingeflossen. Dabei kommt der in der Analyse ausgewählte AD-Wandler AD7691 von Analog Devices zum Einsatz. Das zu messende Signal wird dabei mit einem differentiellen Treiber (Linear Technology LTC6362) an den AD-Wandler herangeführt. Ebenfalls befindet sich die komplette analoge Vorverarbeitungsschaltung auf der Erweiterungsplatine. Es besteht also die Möglichkeit der direkten Messung mittels des hochauflösendem AD-Wandlers sowie der Aufbereitung des Signals durch die analoge Vorverarbeitung.
Es wurden neben der SPI-Schnittstelle, drei benötigte Chip-Select Signalleitungen, eine 3,3 V Spannungsversorgung und zwei frei programmierbare Ein- und Ausgänge vom Mikrocontroller des Zellsensors auf die Erweiterungsplatine geführt. Diese dienen als optionale Testpins, da auf dem Zellsensor selber aus Platzgründen keine Test- bzw. Messpins vorgesehen wurden. Dazu wurde der Anschlussstecker für die Erweiterungsplatine von einem 8-poligen auf einen 10-poligen Stecker abgeändert. Die Pinbelegung des Verbindungssteckers kann aus dem zugehörigen Schaltplan im Anhang J entnommen werden. Abbildung 4.15 zeigt das Layout der hochauflösenden AD-Wandler-Erweiterungsplatine.

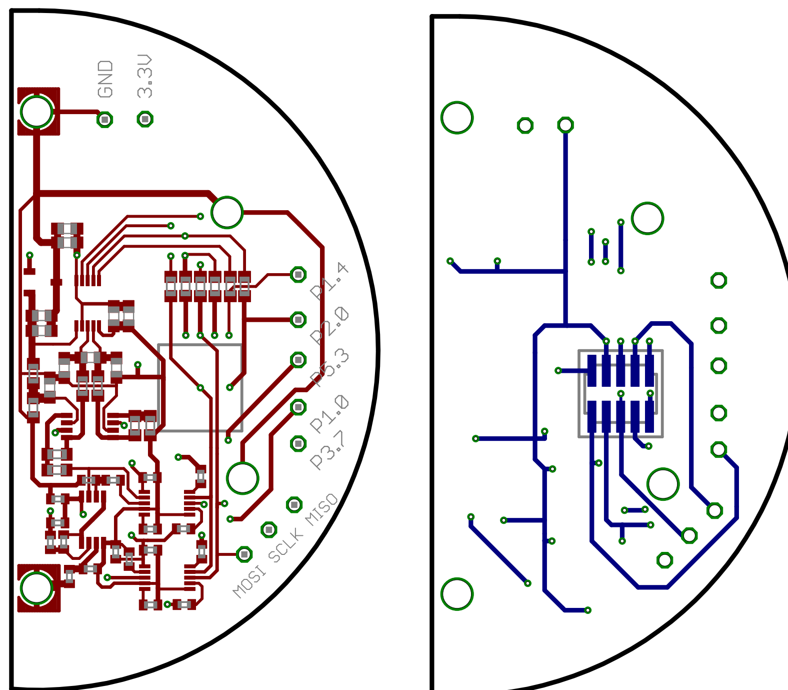


Abbildung 4.15.: Layout der hochauflösenden AD-Wandler-Erweiterungsplatine
Links: Layout der Oberseite. Rechts: Layout der Rückseite

5. Softwareentwurf des Messsystems

In diesem Kapitel werden die Softwarefunktionen des Messsystems vorgestellt. Es wird intensiv auf die Funktionen eingegangen, die für die Impedanzspektroskopie wichtig sind. Für die weiteren Funktionalitäten des Zellsensors und des Batteriesteuergeräts, die für die Messung der Impedanzspektroskopie nicht relevant sind, wird an dieser Stelle auf die Arbeiten [62] und [14] verwiesen. Darin werden die grundlegenden Softwarefunktionen für das Batteriesteuergerät und den Zellsensor erläutert, welche in Tabelle 5.1 aufgelistet sind. Diese Funktionen wurden für das Batteriesteuergerät und den Zellsensor übernommen.

Tabelle 5.1.: Funktionsübersicht Zellsensor v0.4

Zellsensorfunktionen	**Zellsensor v0.4**
Konfiguration des ZS durch die BS	✓
einzelne Spannungsmessung	✓
Temperaturmessung durch TMP102	✓
Temperaturmessung durch MSP430	✓
Ladungsbalancierung	✓
Adressierbares Wake-Up	✓
Burstmessung	✓

5.1. Zellsensor Software

Beim Softwareentwurf des Zellsensors geht es hauptsächlich um Funktionen, die für die Impedanzspektroskopie wichtig sind. Dabei muss zunächst die Steuerung der analogen Vorverarbeitung realisiert werden. Ebenso soll eine dynamische Laufzeitermittlung auf Seiten des Zellsensors sowie des Batteriesteuergeräts implementiert werden. Eine wichtige Implementierung stellt die des Goertzel-Algorithmus dar, welche in Kapitel 5.3 behandelt wird. Zunächst wird aber auf die Speicheroptimierung auf der Seite des Zellsensors eingegangen.

5.1.1. Speicheroptimierung der Messwerte

Die Anzahl der aufgenommenen Messwerte ist begrenzt durch die Speichergröße des Mikrocontrollers auf dem Zellsensor. Optimal ist eine Aufnahme von möglichst vielen Messwerten, um das zu messende Signal über einen langen Zeitraum hochauflösend abtasten zu können. Mit dem MSP430F235 des Zellsensors v0.2 war die Speicherung von ca. 800 Messwerten möglich [62]. Durch den Wechsel des Mikrocontrollers MSP430F235 auf den CC430F5137 in der Arbeit [48] steht nun der doppelte RAM Speicher von 4 kByte zur Verfügung. Dadurch gelang es, die maximale Aufnahmelänge auf 1.900 Messwerte zu erweitern. Durch eine Optimierung des Speichermanagements lässt sich die Anzahl dieser Werte aber nochmals steigern. Da der Mikrocontroller über eine 16 Bit Architektur verfügt, werden die Messwerte in einem 16 Bit Register abgelegt. Die aufgenommenen Messwerte des internen AD-Wandlers des CC430F5137 sind aber nur 12 Bit lang. So bleiben bei jeder Abspeicherung 4 Bit ungenutzt. Dies entspricht bei 1.900 abgespeicherten Messwerten, 7.600 ungenutzte Bit.

$$1.900 \cdot 4\,\text{Bit} = 7.600\,\text{Bit}$$

Nach einer Optimierung des Speichermanagements, können durch Nutzung dieser ungenutzten Bytes, bis zu 633 weitere Messwerte im Speicher abgelegt werden.

$$\frac{7.600\,\text{Bit}}{12\,\text{Bit}/\text{Messwert}} = 633,333\,\text{Messwerte}$$

Somit können bis zu 2.533 12 Bit Messwerte abgespeichert werden. Dies entspricht einer Speichererweiterung von rund 33 %. Diese Speicheroptimierung wird schematisch in Abbildung 5.1 gezeigt.

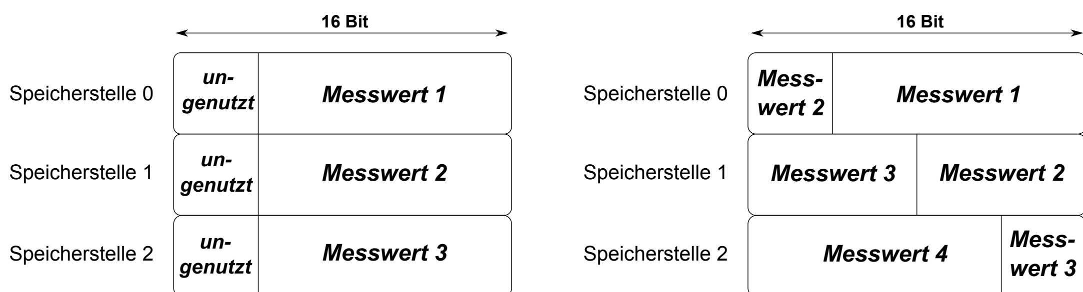


Abbildung 5.1.: Speichermanagement bei 12 Bit Messwerten, Links: Normales Speichermanagement. Rechts: Optimiertes Speichermanagement

Das Speichermanagement wird nun so konfiguriert, dass die Berechnung der richtigen Speicherstelle direkt nach der Wertaufnahme auf dem Zellsensor geschieht. Die Dekodierung erfolgt ebenfalls noch auf dem Zellsensor beim Abruf der Messwerte.

5.1.2. Dynamische Laufzeitermittlung

Die dynamische Laufzeitermittlung dient der Messung der Laufzeit eines Signals zwischen Batteriesteuergerät und Zellsensor. Das Konzept dieser Messung entstammt der Ping-Pong-Methode aus [5]. Dabei sendet das Batteriesteuergerät einen Impuls aus, der vom Zellsensor aufgenommen und wieder zurückgesendet wird. Auf Seiten des Batteriesteuergeräts wird die Zeit gemessen, wie lange der Impuls von der Aussendung bis zum Empfang benötigt. Abzüglich der Zeit, die der Zellsensor benötigt um den Impuls zurückzusenden, ergibt sich dadurch die Laufzeit zwischen Batteriesteuergerät und Zellsensor.

$$\text{Laufzeit} = \left(\frac{\text{Wartezeit BS}}{120\,\text{MHz}}\right) - \left(\frac{\text{Verarbeitungszeit ZS}}{16\,\text{MHz}}\right) \tag{5.1}$$

Das Senden und Empfangen des Impulses geschieht, wie auch bei der Burstmessung, über ein kurzzeitiges Abschalten des Trägersignals [62]. Die Messung der Laufzeit auf der Seite des Batteriesteuergeräts übernimmt ein Timer des Mikrocontrollers. Dieser wird mit der Taktrate des Batteriesteuergeräts von 120 MHz getaktet. Somit erhält dieser Timer eine zeitliche Auflösung von 8,33 ns. Auf der Seite des Zellsensors wird die Zeit zwischen empfangen und wieder aussenden des Taktsignals ebenfalls mittels eines Timers gemessen. Dieser wird auf dem Zellsensor mit einer Taktrate von 16 MHz getaktet. Daraus ergibt sich eine zeitliche Auflösung von 62,9 ns, die der Zellsensor messen kann.
Der Zustandsablauf der dynamischen Laufzeitermittlung in Abbildung 5.2 zeigt die Abhängigkeiten zwischen den einzelnen Zuständen. Diese Abhängikeiten sind jeweils mit den Pfeilen zwischen des Zustands des Batteriesteuergeräts und des Zellsensors aufgezeigt.

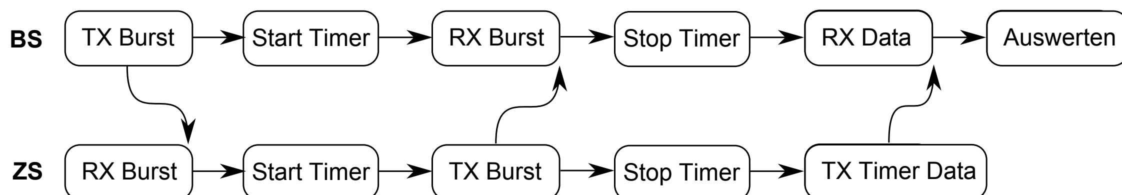


Abbildung 5.2.: Zustandsablauf und Abhängigkeiten der dynamischen Laufzeitermittlung

Da hier eine Messung über zwei unterschiedliche Zeitbasen gemacht wird, kann es zu Fehlmessungen kommen, wenn die Taktrate des Zellsensors nicht die erwarteten 16 MHz beträgt. Um dies zu verhindern, wird eine Taktkalibrierung implementiert, die den Takt des Zellsensors auf die gewünschten 16 MHz einstellen soll. Diese Kalibrierung wird ebenfalls über die Funkschnittstelle realisiert.

Taktkalibrierung

Ein Vorteil des auf dem Zellsensor eingesetzten Mikrocontrollers CC430F5137 ist, dass sich die DCO-Taktrate während des Betriebs einstellen lässt. Diese Funktion wird hier ausgenutzt, um eine Taktkalibrierung des internen DCO durchzuführen. Dazu sendet der Zellsensor ein Taktsignal mit definierter Pulslänge aus, das vom Batteriesteuergerät empfangen und ausgewertet wird. Anhand der empfangenen Pulslänge des Taktsignals, kann das Batteriesteuergerät Abweichungen des Zellsensortakts erfassen und diese korrigieren.
Das Sendemodul des Mikrocontrollers lässt sich direkt mit einem Timer des Mikrocontrollers verbinden (Abbildung 5.3). Da dieser Timer direkt durch den DCO getaktet wird, lässt sich ein Taktsignal senden, dessen Pulslänge sich in der Abhängigkeit des DCO-Takts verändert.

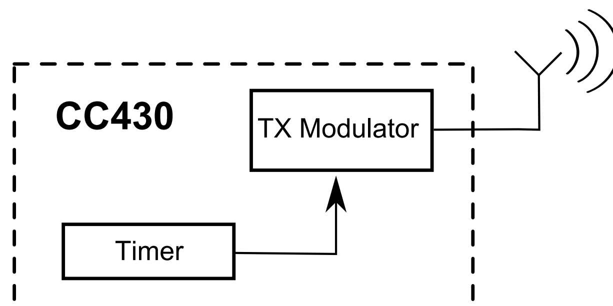


Abbildung 5.3.: Sendeprinzip der DCO-Kalibrierung (nach [52])

In Abbildung 5.4 ist das Prinzip dieser Messung dargestellt. Der Zellsensor sendet ein Signal mit einer Periodenlänge von 2 ms aus. Dies entspricht bei einem DCO-Takt von 16 MHz genau 32.000 DCO-Takten. Die Messung durch das Batteriesteuergerät erfolgt jeweils an den steigenden Flanken des Taktsignals.

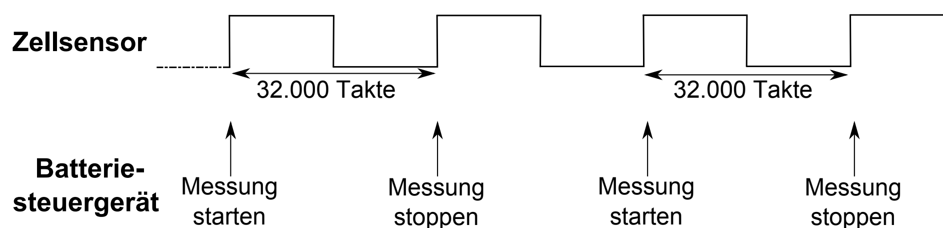


Abbildung 5.4.: Pulsmessung des Batteriesteuergerätes

Die Messung am Batteriesteuergerät ergibt im Idealfall 240.000 Takte. Diese entsprechen bei der 120 MHz Taktung genau 2 ms. Ergibt die Messung am Batteriesteuergerät weniger als 240.000 Takte ist dies ein Anzeichen dafür, dass der Takt des Zellsensors zu schnell ist. Werden mehr als 240.000 Takte gemessen, ist der Zellsensor DCO-Takt zu langsam.

Anhand dieser Messungen können Rückschlüsse auf die DCO-Geschwindigkeit des Zellsensors gezogen und von Seiten des Batteriesteuergeräts entsprechend reagiert werden. Eine Reaktion des Steuergeräts ist es, eine Nachricht an den entsprechenden Zellsensor zu schicken, der daraufhin seinen DCO-Takt verringert bzw. erhöht. Die Messung wird solange wiederholt, bis der Zellsensor DCO-Takt innerhalb eines vorgegebenen Toleranzbereichs liegt.

Abbildung 5.5 zeigt den schematischen Ablauf dieser Kalibrierung auf der Seite des Batteriesteuergeräts sowie an dem Zellsensor.

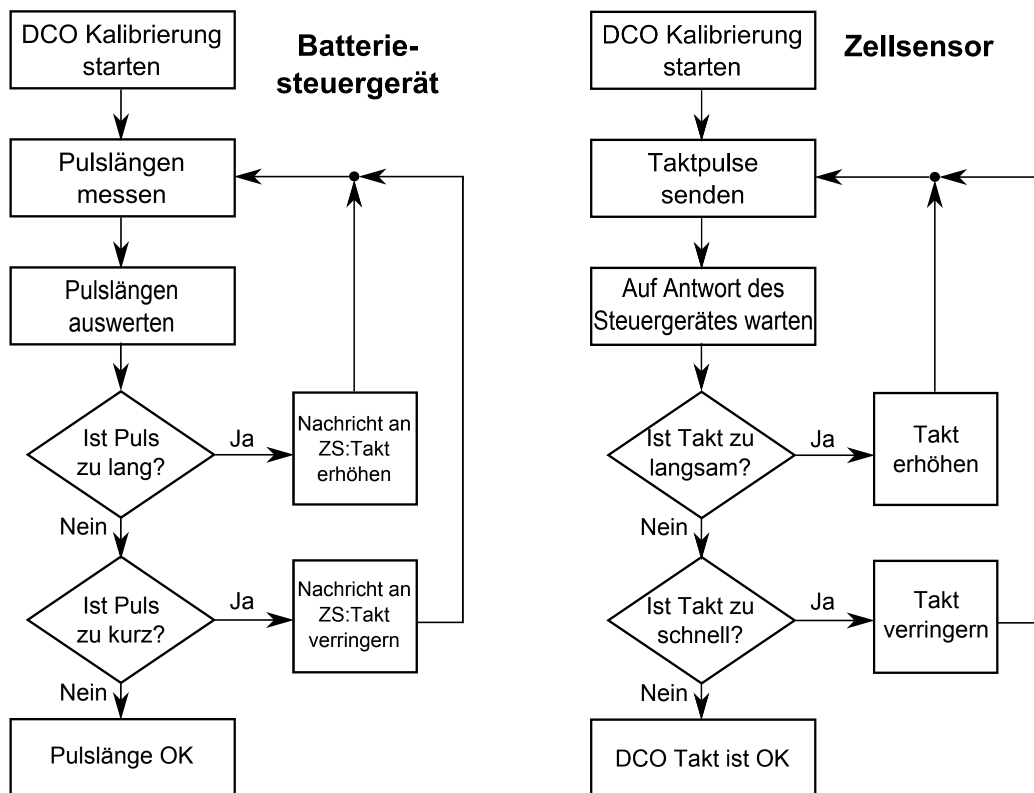


Abbildung 5.5.: Softwareablauf der DCO-Kalibrierung am Zellsensor

5.1.3. Steuerung der analogen Vorverarbeitung

Für die Realisierung der Impedanzspektroskopie ist die Steuerung der analogen Vorverarbeitung die wichtigste Softwareerweiterung auf Seiten des Zellsensors. Bei der Steuerung geht es hauptsächlich um die Einstellung der beiden Rheostaten, die zum einen den Gleichspannungsabzug an der Subtrahierschaltung und zum anderen den Grad der Verstärkung des Messsignals einstellen. Dabei gibt es zwei unterschiedliche Arten, wie die analogen Vorverarbeitung gesteuert werden kann. Diese sind die Einstellung unter Last und die Einstellung mittels der Messung des Gleichspannungsanteils.

Einstellung durch Messung des Gleichspannungsanteils

Für die Einstellung durch Messung des Gleichspannungsanteils wird der zusätzliche AD-Wandler Kanal verwendet, der bereits in Abbildung 4.5 auf Seite 96 gezeigt wurde. Dabei wird der Gleichspannungsanteil der Zellspannung über den Spannungsteiler gemessen. Aus dem gemessenen Gleichspannungsanteil lassen sich die Einstellungen für die Subtrahierschaltung sowie für die Spannungsverstärkung berechnen, wie es Abbildung 5.6 zeigt.

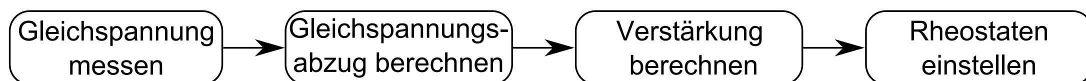


Abbildung 5.6.: Softwareschritte zur Einstellung mittels Gleichspannungsanteil

Ein Nachteil dieser Einstellungsmethode ist es, dass die Stärke der Anregung und somit die Höhe der Messamplitude nicht bekannt ist. Dadurch ist eine korrekte Berechnung der Verstärkung nicht möglich, sondern kann nur abgeschätzt werden. Ist die Messamplitude zu hoch, kann es vorkommen, dass die Messspannung soweit verstärkt wird, dass diese über den Referenzspannungsbereich des AD-Wandlers hinaus verstärkt wird und somit in die Begrenzung gerät.

Einstellung unter Last

Eine präzisere Art die analoge Vorverarbeitung einzustellen ist es, diese unter Last vorzunehmen. Dabei wird die Spannung über den Pfad der analogen Vorverarbeitung gemessen und ausgewertet. Bei dieser Einstellungsmethode wird die zu vermessende Batteriezelle mit einem Wechselstromanteil von 50 Hz angeregt. Durch diese Anregung und der daraus resultierenden Messamplitude kann der richtige Wert für den Gleichspannungsabzug sowie für die Verstärkung ermittelt werden. Bei der Einstellung der analogen Vorverarbeitung kann in zwei verschiedenen Stufen unterschieden werden.

1. Kompensierung der Gleichspannung
2. Verstärkung der Messspannung

Erste Stufe: Kompensierung der Gleichspannung

Die Kompensierung der Gleichspannung erfolgt dabei schrittweise. Der Rheostat wird langsam erhöht um einen höheren Spannungsabzug durch die Subtrahierschaltung zu erwirken. Dies geschieht solange, bis eine einstellbare untere Grenzspannung erreicht wird. Abbildung 5.7 zeigt das Ablaufschema dieser Einstellung.

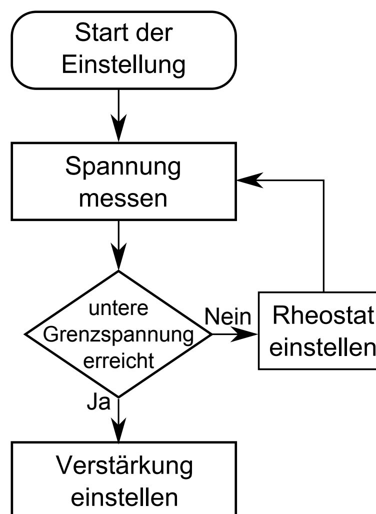


Abbildung 5.7.: Erste Stufe der analogen Vorverarbeitung: Gleichspannung abziehen

Bei dem Abzug der Gleichspannung kommt es nicht auf ein genaues Erreichen der vorgegebenen Grenzspannung an. Wichtig ist nur, dass ein größtmöglicher und stabiler Spannungsabzug erreicht wird. Der Wert des erfolgten Spannungsabzugs ist für die Berechnung des Impedanzspektrums sowie für die weitere Verstärkung nicht notwendig.

Zweite Stufe: Verstärkung der Messspannung

Nachdem der Gleichspannungsanteil auf die untere Grenzspannung reduziert wurde, kann die Messspannung verstärkt werden. Die Verstärkung geschieht hier, wie bei der ersten Stufe, durch eine schrittweise Erhöhung des Rheostaten an der Verstärkerschaltung. In Abbildung 5.8 ist auch das Ablaufschema der zweiten Stufe dargestellt. Es unterscheidet sich zu der ersten Stufe lediglich darin, dass hier eine obere Grenzspannung erreicht werden soll.

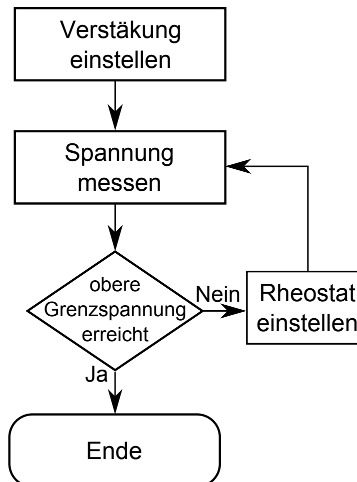


Abbildung 5.8.: Zweite Stufe der analogen Vorverarbeitung: Messwert verstärken

Bei der Verstärkung ist es nun wichtig, dass die richtige Frequenz von 50 Hz für die Anregung gewählt wird. Grund hierfür ist, dass das Messsignal 20 ms lang mit 1 kHz abgetastet wird. Dabei wird nach dem größten Amplitudenwert gesucht, nach dem entschieden wird ob die obere Grenzspannung erreicht worden ist. Wird eine langsamere Frequenz gewählt, kann es vorkommen, dass innerhalb dieser 20 ms nicht der höchste Amplitudenwert abgetastet wird, und somit die Verstärkung falsch eingestellt wird. Eine Fehleinstellung der Verstärkung lässt sich daran erkennen, dass ein Teil der Amplitude außerhalb des Referenzbereichs des AD-Wandlers liegt, und somit nicht erfasst werden kann. Abbildung 5.9 zeigt die Abtastung eines Messsignals bei 50 Hz sowie bei 25 Hz.

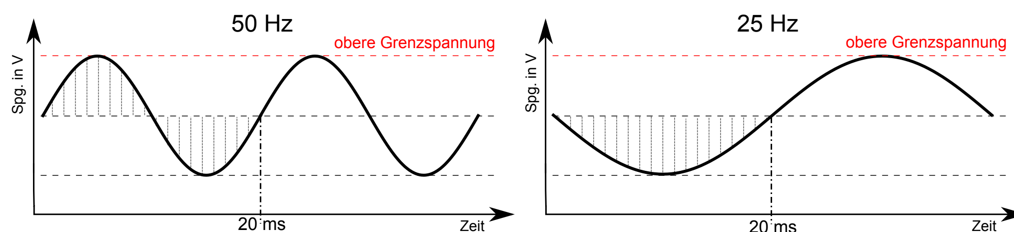


Abbildung 5.9.: Möglicher Fehler der Messwert-Verstärkung. Links: Abtastung eines 50 Hz Messsignals. Rechts: Worst-Case Abtastung eines 25 Hz Messsignals.

Eine längere Aufnahme des Messsignals ist zwar möglich, würde aber die gesamte Einstellzeit enorm verlängern. Die Abbildung 5.9 zeigt bei dem 25 Hz Signal den "Worst-Case"Fall, indem die Perioden nicht in ihrer vollen Amplitude vermessen werden. Dadurch wird angenommen, dass die obere Grenzspannung noch nicht erreicht ist und das Messsignal noch weiter verstärkt werden kann. Dieser Fehler kann durch die Wahl der richtigen Anregefrequenz vermieden werden.

Nachdem das Messsignal durch die analoge Vorverarbeitung entsprechend aufbereitet wurde, kann es durch den AD-Wandler des Mikrocontrollers gemessen werden. In Abbildung 5.10 sind die einzelnen Stufen der Bearbeitung nochmals dargestellt. In diesem Beispiel wird eine 3,3 V Batteriespannung mit einer Wechselamplitude von 4 mV beaufschlagt. Der Spannungsabzug der Subtrahierschaltung (1. Stufe) zieht die maximale Spannung von 3,276 V ab. Die 2. Stufe verstärkt die Messamplitude soweit, dass die obere Grenzspannung von 2,4 V erreicht wird.

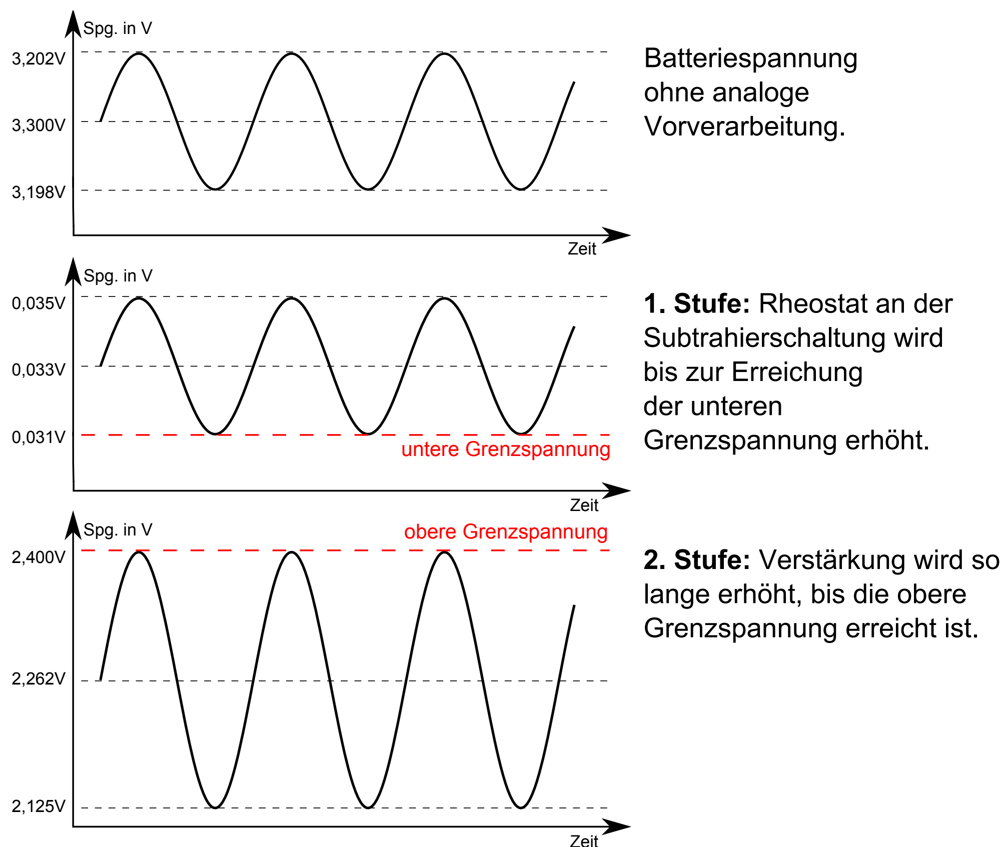


Abbildung 5.10.: Konzeptdarstellung der verschiedenen Stufen mit Grenzspannungen der Einstellung unter Last

Das ursprüngliche Messsignal von 4 mV erreicht nach der analogen Vorverarbeitung einen

Amplitudenwert von 275 mV und kann somit vom internen 12 Bit AD-Wandler erfasst werden. Die Verstärkung beträgt in diesem Fall V = 68,75.

5.2. Batteriesteuergerät Software

An dieser Stelle wird das grundlegende Softwarekonzept des Batteriesteuergeräts erläutert. Es soll ein Überblick über die Funktionen und die Arbeitsweise des Batteriesteuergeräts gegeben werden. Die Software musste aufgrund der Neuentwicklung komplett neu erarbeitet werden. Es wurde versucht, die einzelnen Funktionen des alten Steuergeräts aus den Arbeiten [14] und [62] zu übernehmen.
Die Steuerung soll wie beim Vorgängermodell über einen externen PC erfolgen. Dies kann durch die vorhandene UART-Schnittstelle realisiert werden. Die softwareseitige Abarbeitung von Aufgaben soll in verschiedene Zustände unterteilt werden. Dazu werden die einzelnen Funktionen modular aufgebaut. Dies bedeutet, dass Funktionsmodule weitestgehend unabhängig voneinander arbeiten können. So lässt sich eine bessere Wartung und Funktionsüberprüfung der Software realisieren.
Die Software wird in einem Automatenmodell realisiert. Die grundlegende Softwarestruktur des Batteriesteuergeräts ist in Abbildung 5.11 dargestellt. Es handelt sich um eine Endlosschleife, die durch Interrupts, wie der von der UART-Schnittstelle, unterbrochen werden kann. Einzelne Aufgaben werden in Unterautomaten abgearbeitet. Die Zuteilung in die einzelnen Unterautomaten, erfolgt durch die UART-Auswertung. Nach der Abarbeitung des Unterautomaten, geht die Software wieder in die Endlosschleife und wartet auf eine weitere Unterbrechung durch einen Interrupt.

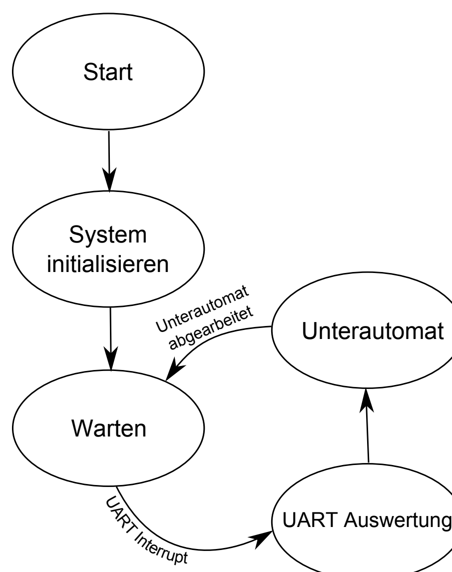


Abbildung 5.11.: Grundautomat des Batteriesteuergeräts

Da der Ablauf des Softwarekonzepts des Batteriesteuergeräts Interrupt gesteuert ist, müssen die Prioritäten an die verschiedenen Interrupts vergeben werden. Die Einteilung der einzelnen Prioritäten erfolgt, wie in Abbildung 5.12 zu sehen, nach der zulässigen Latenzzeit der vom Interrupt gesteuerten Funktion. So sind alle zeitkritischen Funktionen, wie Delay- und Timeout-Zeiten mit einer hohen Priorität zu versehen, damit diese rechtzeitig abgearbeitet werden können. Nicht zeitkritische Funktionen, wie die UART Kommunikation, bekommen eine niedrige Priorität.

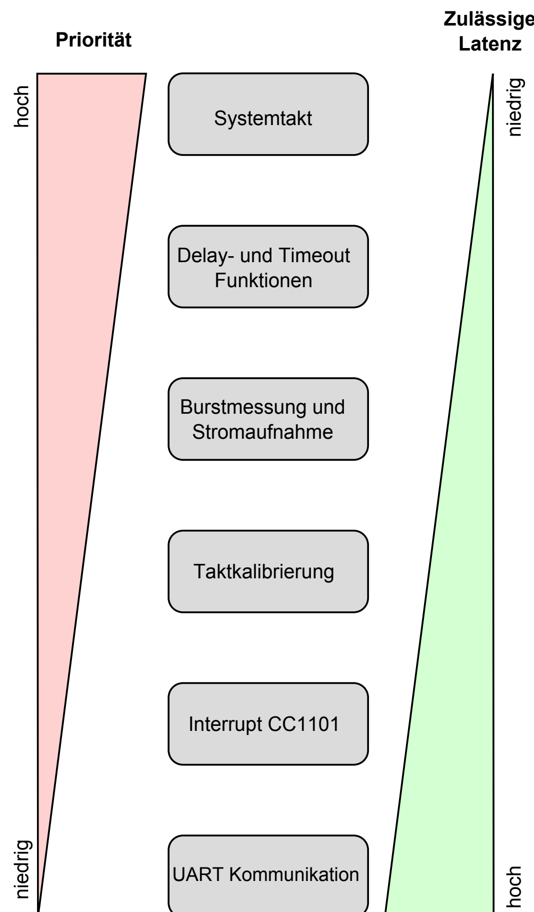


Abbildung 5.12.: Prioritätenverteilung am Batteriesteuergerät

5.2.1. Funktionsübersicht des Batteriesteuergeräts

In diesem Abschnitt wird auf die Funktionen des neu entwickelten Batteriesteuergeräts eingegangen. Weitestgehend sind diese Funktionen aus dem vorangegangenen Steuergerät übernommen worden, weshalb hier nur die wichtigsten Funktionen kurz erwähnt werden. Im Anhang F.1 befindet sich eine Befehlsliste, mit der die einzelnen Befehle ausgeführt

werden können.

Aufnahme von einzelnen Spannungswerten
Zur Messung einzelner Spannungen, lässt sich ein adressierter Befehl senden, der eine einzelne Spannungsmessung auf einem Zellsensor auslöst. Der Sensor schickt die Daten der aufgenommenen Spannung anschließend automatisch wieder an das Batteriesteuergerät zurück.

Burstmessung durchführen
Zum Start der Burstmessung kann ein Befehl gesendet werden, der als Broadcast-Message an alle Zellsensoren geht. Dabei ist die gewünschte Abtastfrequenz und die Anzahl der aufzunehmenden Werte anzugeben. Details dazu sind in [62] zu finden.

Abruf von Spannungswerten der Burstmessung
Das Abrufen von Spannungswerten der Burstmessung lässt sich durch einen Befehl ausführen. Dieser wird adressiert an einen Zellsensor gesendet. Der angesprochene Zellsensor antwortet daraufhin und sendet die aufgenommenen Spannungsdaten in einzelnen Paketen zurück an das Batteriesteuergerät. Details dazu sind in [62] zu finden.

Abruf der Goertzel DFT-Werten
Um die, durch den Goertzel-Algorithmus berechneten DFT-Werte, der bei der Burstmessung erfassten Spannungswerte zu erhalten, ist ein Befehl vom Batteriesteuergerät zum entsprechenden Zellsensor zu senden. Dabei muss die Anzahl der für die Berechnung verwendeten Perioden, die verwendete Burstfrequenz sowie die Signalfrequenz angegeben angegeben und an den Zellsensor übertragen werden. Also Rückgabewert erhält man dann die komplexen Werte der Spannung sowie des gemessenen Stroms.

Manuelle Einstellung der analogen Vorverarbeitung
Neben der bereits vorgestellten automatischen Einstellung der analogen Vorverarbeitung, wurde auch eine manuelle Einstellung implementiert. Dabei kann man, durch einen adressierten Befehl, dem Zellsensor die einzustellenden Werte für die Rheostaten der analogen Vorverarbeitung senden.

Dynamische Laufzeitmessung
Die bereits vorgestellte dynamische Laufzeitermittlung lässt sich durch einen einfachen adressierten Befehl an dem Batteriesteuergerät auslösen. Als Rückgabewert werden an die UART-Schnittstelle die gemessenen Takte des Batteriesteuergerät sowie des Zellsensors ausgegeben.

Konfigurationsdaten eines Zellsensors abrufen
Eine neue Funktion, die in dieser Arbeit implementiert wurde, ist das Abrufen von Konfigurationsdaten eines Zellsensors. Dazu lässt sich ein adressierter Befehl an den Zellsensor senden. Dieser schickt anschließend seine aktuellen Einstellung an das Steuergerät zurück, die daraufhin über die UART-Schnittstelle ausgegeben werden.

5.2.2. Taktaussendung und synchrone Strommessung

Die Taktaussendung und die Strommessung sind die wichtigsten Funktionen des Batteriesteuergeräts für die Impedanzspektroskopie. Diese Funktionen sind sehr zeitkritisch, weshalb diese komplett durch Timer-Interrupts gesteuert werden.
Die eigentliche Taktaussendung wurde in der Arbeit [62] entwickelt. Die Neuerung aus dieser Arbeit besteht in der synchronen Messung des Stroms. Dazu wird nach der Taktaussendung eine festgelegte Wartezeit abgearbeitet, bevor die Strommessung durchgeführt wird. Diese Wartezeit wurde in Abschnitt 3.3 zu 72,2 μs bestimmt. Abbildung 5.13 zeigt den Unterautomaten, der diese Funktion realisiert.

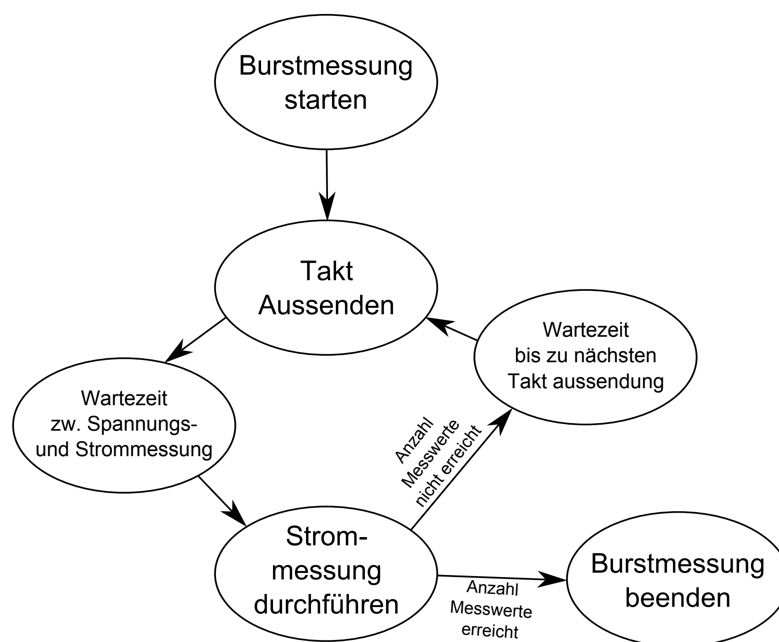


Abbildung 5.13.: Unterautomat der Burstmessung zur Impedanzspektroskopie

Da die Wartezeit zwischen dem Aussenden des Taktes und der Strommessung sehr zeitkritisch ist, wurde eine von Texas Instruments vorgegebene Delay-Funktion[1] verwendet, um die 72,2 μs Wartezeit zu erreichen. Diese Funktion benötigt für die Abarbeitung lediglich drei Taktzyklen [34]. Bei einer Taktung von 120 MHz lässt sich dadurch eine Genauigkeit der Wartezeit von bis zu 25 ns erreichen.

[1]TivaWare Funktion: SysCtlDelay()

5.3. Implementierung des Goertzel-Algorithmus

Durch den Einsatz des Goertzel-Algorithmus soll eine zeitintensive Übertragung der Messwerte verhindert werden. Dazu soll der in Kapitel 3.6.3 vorgestellte Algorithmus sowohl im Batteriesteuergerät wie auch auf den Zellsensoren implementiert werden.
Diese Implementierung des Goertzel-Algorithmus wird im Folgenden auf dem Zellsensor beschrieben. Die Implementierung auf dem Batteriesteuergerät ist dieselbe, weshalb darauf nicht weiter eingegangen wird.

5.3.1. Implementierung auf dem Zellsensor

Die Implementierung des Goertzel-Algorithmus erfordert eine effiziente Umsetzung der in Abschnitt 3.6.3 vorgestellten Filterstruktur. Dabei ist besonders auf die begrenzten Ressourcen des Mikrocontrollers zu achten. So müssen für die Berechnung der Real- und Imaginärteile unter anderem Sinus- und Kosinus-Werte berechnet werden. Um diese effizient berechnen zu können, wird ein MSP-Mathematik-Library von Texas Instruments verwendet [33]. Neben dieser Berechnungen müssen vor der eigentlichen Filterberechnung noch einige für die Berechnung wichtige Konstanten vorberechnet werden. Dazu müssen die Anregefrequenz, die Abtastfrequenz und die Anzahl der in die Berechnung einfließenden Perioden bekannt sein. Diese drei Werte werden dem Zellsensor durch das Steuergerät vor der Berechnung mitgeteilt. Berechnet werden muss zunächst die Konstante N.

$$\mathrm{N} = \frac{\text{Abtastfrequenz } \mathrm{F_s}}{\text{Signalfrequenz f}} \qquad \mathrm{N} \in \mathbb{N} \tag{5.2}$$

Diese Konstante gibt an, bis zu welcher Länge das Filter berechnet werden soll. N bildet sich als Quotient aus der Abtastfrequenz $\mathrm{F_s}$ und der Signalfrequenz f. Da $\mathrm{N} \in \mathbb{N}$ sein muss, wird ersichtlich, dass das Verhältnis zwischen Abtastfrequenz $\mathrm{F_s}$ und der Signalfrequenz f passend gewählt werden muss, damit das Filter korrekte Werte berechnen kann. Ist die Abtastfrequenz $\mathrm{F_s}$ nicht ein ganzzahliges Vielfaches der Signalfrequenz f, kommt es zu Fehlern in der Berechnung der Filterlänge N und somit zu Fehlern in der errechneten Impedanz.
Eine weitere Konstante die vorberechnet werden kann, ist die Konstante k. Die Konstante k bestimmt hierbei die zu detektierende Zielfrequenz [7].

$$\mathrm{k} = \left(\frac{\text{Anzahl Abtastpunke} \cdot \text{Signalfrequenz}}{\text{Abtastfrequnenz}} \right) \qquad k \in \mathbb{N} \tag{5.3}$$

Des Weiteren lässt sich ω berechnen. Dies ist für die Berechnung der Sinus- und Kosinus-Werte nötig:

$$\omega = \frac{2 \cdot \pi \cdot \mathrm{k}}{\text{Anzahl Abtastpunke}} \qquad \omega \in \mathbb{R} \tag{5.4}$$

$$\text{sinus} = \sin(\omega) \qquad \text{sinus} \in \mathbb{R} \tag{5.5}$$

$$\text{cosine} = \cos(\omega) \qquad \text{cosine} \in \mathbb{R} \tag{5.6}$$

Nun kann der nötige Filterkoeffizient koeff für den rekursiven Teil des Filters berechnet werden:

$$\text{koeff} = 2 \cdot \text{cosine} \qquad \text{koeff} \in \mathbb{R} \tag{5.7}$$

Mit diesen Vorberechnungen kann nun mit der eigentlichen Filterberechnung begonnen werden. Berechnet wird das Filter bis zu einer Länge von N+1. Da im Schritt N+1 kein Wert mehr vorhanden ist, wird der Wert Null in die Filterstruktur geschoben. Berechnet werden im rekursiven Teil des Filters (Abbildung 5.14) die Verzögerungsstufen b_n, b_{n-1} und b_{n-2} bis zu der Stelle N. Der nicht-rekursive Teil des Filters wird bis zur Stelle N nicht berechnet. Die einzelnen Verzögerungsstufen berechnen sich dabei wie folgt:

$$b_n = \text{koeff} \cdot b_{n-1} - b_{n-2} + x[n] \qquad b_n \in \mathbb{R} \tag{5.8}$$

$$b_{n-2} = b_{n-1} \qquad b_{n-2} \in \mathbb{R} \tag{5.9}$$

$$b_{n-1} = b_{n-2} \qquad b_{n-1} \in \mathbb{R} \tag{5.10}$$

Hier wird deutlich, dass bis zur Stelle N, lediglich N Multiplikationen sowie $2 \cdot N$ Additionen bzw. Subtraktionen benötigt werden. Der rekursive Teil des Filters ist in Abbildung 5.14 Rot markiert zu sehen. In diesem Teil werden die Berechnungen bis zum Wert N durchgeführt.

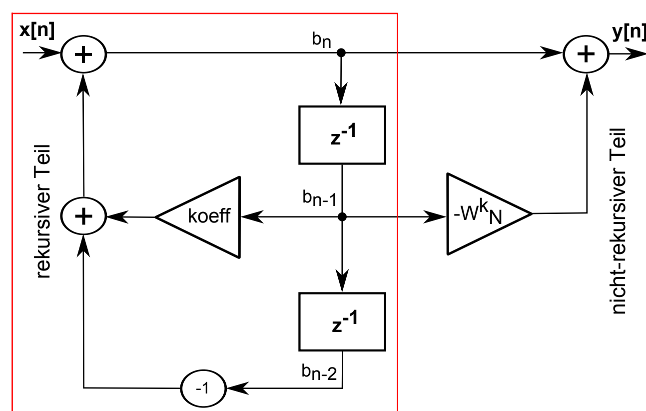


Abbildung 5.14.: Berechnung des rekursiven Filterteils

Lediglich im letzten Berechnungsschritt N+1 wird der nicht-rekursive Teil (Abbildung 5.15) des Filters berechnet.

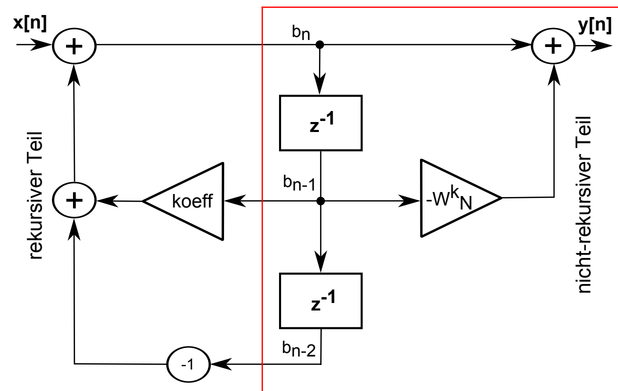


Abbildung 5.15.: Berechnung des nicht-rekursiven Filterteils

Da es hier zu einer komplexer Multiplikation kommt, muss für den letzten Berechnungsschritt die Struktur des Filters für die Implementierung auf dem Mikrocontroller optimiert werden. Diese optimierte Form zeigt Abbildung 5.16. Dabei werden neben der Filterstruktur auch die Koeffizienten für den rekursiven und den nicht rekursiven Teil des Filters geändert. Diese ergeben sich nun zu [64]:

$$\text{koeff1} = -\cos(\omega) \tag{5.11}$$

$$\text{koeff2} = \sin(\omega) \tag{5.12}$$

Die für den Berechnungsschritt N+1 optimierte Filterstruktur ist in Abbildung 5.16 zu sehen.

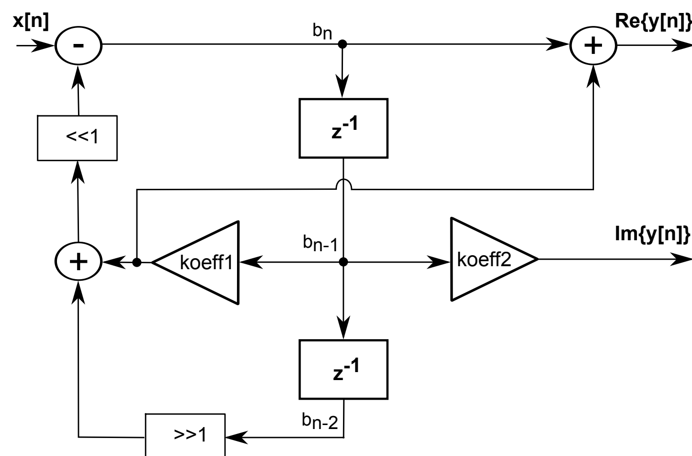


Abbildung 5.16.: Für die Mikrocontroller-Implementierung optimierte Filterstruktur des Berechnungsschritts N+1 (nach [64])

Die Differenzengleichung des für die Implementierung optimierten Filters ergibt sich nun zu:

$$b_n = x[n] - b_{n-2} + 2 \cdot \cos(\omega) \cdot b_{n-1} \tag{5.13}$$

Aus dieser Gleichung lässt sich der Real- und Imaginärteil wie folgt berechnen [64]:

$$\text{Re[N]} = bn_1 - bn_2 \cdot \cos(\omega); \tag{5.14}$$

$$\text{Im[N]} = bn_2 \cdot \sin(\omega); \tag{5.15}$$

Diese Rechnungen lässt sich nun auf dem Mikrocontroller implementieren, da sie nur noch aus vorberechneten Werten, reellen Additionen und reellen Multiplikationen bestehen.

5.3.2. Test der implementierten Filterstruktur

Um die entwickelte Filterstruktur zu testen, wird ein Strom- und Spannungssignal aufgenommen. Dabei handelt es sich um ein Sinussignal mit 250 Hz, welches mit 4 kHz abgetastet wird. Um eine Bewertung des implementierten Goertzel-Filters vornehmen zu können, werden die Messwerte einmal durch die von Matlab gegebene FFT- sowie der Matlab-Goertzel Funktion berechnet. Diese Werte können nun mit den Ergebnissen des implementierten Goertzel-Algorithmus verglichen werden. Die Ergebnisse zeigt Tabelle 5.2.

Tabelle 5.2.: Vergleich der entwickelten Goertzel-Filterstruktur

Berechnungsart	komplexe Spannung	komplexer Strom	komplexe Impedanz in mΩ
FFT Matlab	-5.25154 - 2.93072i	-717.783 - 479.133i	6.94656 - 5.53939i
Goertzel Matlab	-5.25154 - 2.93072i	-717.783 - 479.133i	6.94656 - 5.53939i
Goertzel Filterstruktur	-5.25137 - 2.93106i	-717.774 - 479.186i	6.94646 - 5.53885i

Im Vergleich der obenstehenden Werte ist zu erkennen, dass es Abweichungen in den Ergebnissen gibt. Allerdings handelt es sich dabei um minimale Abweichungen. Dies wird bei der Berechnung der komplexen Impedanz deutlich. So beträgt die Differenz im Realteil der Impedanz lediglich 100 nΩ. Im Imaginärteil liegt die Differenz zwischen den von Matlab berechneten Werten und den der Goertzel Filterstruktur bei 540 nΩ.
Da sich die Abweichungen zwischen den von Matlab berechneten Werten und der von der Filterstruktur berechneten Werten auf ein Minimum reduzieren, ist die Filterstruktur durchaus geeignet für den Einsatz auf dem Mikrocontroller.
Im Anhang H befindet sich ein nach dieser Filterstruktur entwickeltes PC-Testprogramm, mit dem sich die Goertzel-Filterstruktur mit verschiedenen Werten testen lässt.

6. Inbetriebnahme und Erprobung der Impedanzspektroskopie

In diesem Kapitel sollen die Inbetriebnahme und Erprobung der Impedanzspektroskopie durchgeführt werden. Dabei wird neben dem entwickelten Zellsensor auch das neu entwickelte Batteriesteuergerät getestet. Insbesondere soll dabei auf die Funktionen eingegangen werden, die für die funksynchronisierte Impedanzspektroskopie wichtig sind.
Die Erprobung der Impedanzspektroskopie soll mit Hilfe einer Testimpedanz durchgeführt werden. Verläuft diese erfolgreich, kann die Impedanzspektroskopie an Batteriezellen erfolgen. Dabei sollen Messreihen bei unterschiedlichen Alterungszuständen, unterschiedlichen Ladezuständen und verschiedenen Temperaturen durchgeführt werden.
Abbildung 6.1 zeigt einen der neuen Zellsensoren und das entwickelte Batteriesteuergerät mit aufgesteckter Stabantenne auf dem Transceiver-Erweiterungsmodul.

Abbildung 6.1.: Neu entwickeltes Messsystem mit Batteriesteuergerät und Zellsensor

6.1. Inbetriebnahme der entwickelten Hardware

Die Inbetriebnahme umfasst zunächst die Überprüfung der entwickelten Hard- und Software. Dabei sollen eventuell auftretende Fehler im Design oder der Programmierung gefunden und wenn möglich ausgebessert werden. Es werden sowohl die Hard- und Software des Batteriesteuergeräts, als auch der neu entwickelte Zellsensor getestet. Ein vollständiger Funktionstest des Batteriesteuergeräts ist allerdings nicht nötig, da dieses bereits in der Entwicklungsphase in der Arbeit [48] eingesetzt und erfolgreich getestet wurde. Dabei war allerdings noch keine Strommessung am Steuergerät implementiert.

Daher umfasst die Inbetriebnahme des Batteriesteuergeräts die folgenden Punkte:

- Kommunikation über das CC1101 Transceiver-Erweiterungsmodul
- Stromerfassung über das Strommess-Erweiterungsmodul

Die Inbetriebnahme des Zellsensors muss ausführlicher durchgeführt werden. Diese besteht aus den folgenden Punkten:

- Programmierung des Mikrocontrollers
- Berechnung des Anpassungsnetzwerks der Schleifenantenne
- Kontrolle der Spannungsversorgung
- Funktionstest der drahtlosen Kommunikation
- Abrufen einzelner Spannungswerte
- Steuerung der analogen Vorverarbeitung
- Funktionstest der dynamischen Laufzeitmessung
- Phasenrichtige Aufnahme der Spannung und abrufen der Daten

6.1.1. Inbetriebnahme des Batteriesteuergeräts

Abbildung 6.2 zeigt das aufgebaute Batteriesteuergerät mit den beiden aufgesteckten Erweiterungsmodulen. Ebenso sind die Stromkabel im linken Teil des Bildes zu sehen, die an das Strommess-Modul montiert sind.

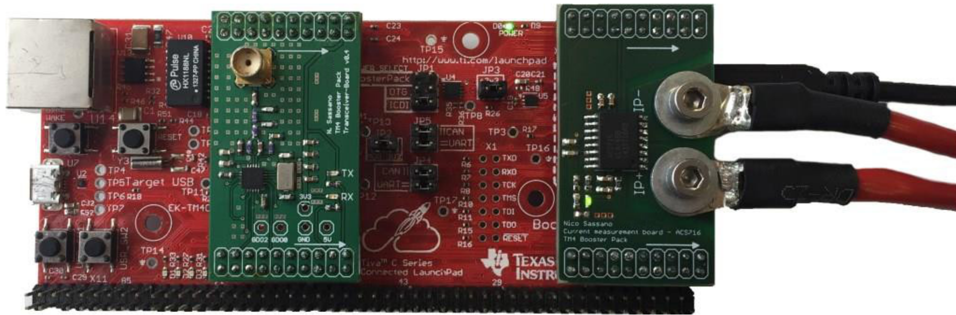


Abbildung 6.2.: Batteriesterngerät mit aufgesteckten Erweiterungsmodulen

Kommunikation über das CC1101 Transceiver-Erweiterungsmodul

Einen ersten Funktionstest während der Entwicklungsphase hatten das Batteriesteuergerät und das CC1101-Erweiterungsmodul bereits in der Arbeit [48]. Deshalb wird an dieser Stelle auf einen Funktionstest verzichtet. Einen allgemeinen Funktionstest erhält das Steuergerät ebenfalls durch die Inbetriebnahme des Zellsensors, da die meisten Hard- und Softwarefunktionen durch das Steuergerät sowie das CC1101 Transceiver-Erweiterungsmodul ausgelöst werden. Abbildung 6.3 zeigt das bestückte CC1101 Transceiver-Erweiterungsmodul und die darauf verbauten Komponenten.

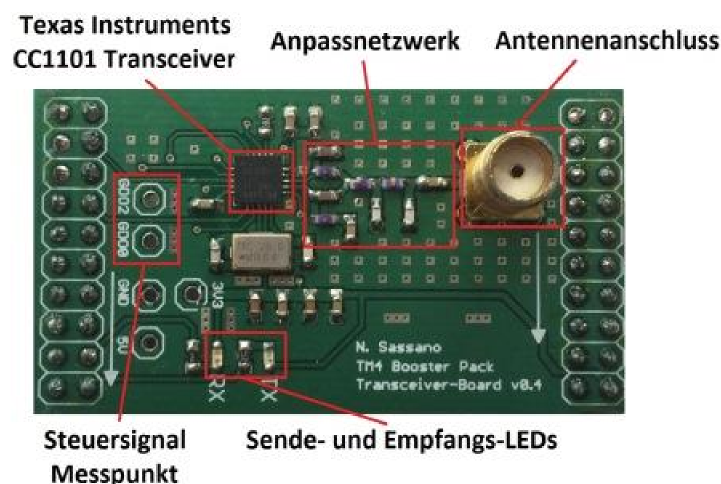


Abbildung 6.3.: Bestücktes CC1101 Transceiver-Erweiterungsmodul

Stromerfassung über das Strommess-Erweiterungsmodul

Abbildung 6.4 zeigt das bestückte Strommess-Erweiterungsmodul. Neben dem eigentlichen Strommess-Hall-Sensor (ACS716), sind die Stromanschlüsse im unteren Teil der Platine farblich hervorgehoben.

Abbildung 6.4.: Bestücktes Strommess-Erweiterungsmodul

Um die Funktionsweise der Stromerfassung über das Strommess-Erweiterungsmodul zu überprüfen, soll ein Strom (50 Hz und 2,0 A Amplitude) erfasst und die aus dem Modul erzeugte Spannung gemessen werden. Die aus dem Modul ausgegebene Spannung hängt vom durchflossenen Strom ab. Die Spannung bei einem Strom von 0 A liegt dabei bei 1,65 V. Je nachdem, wie die Richtung des Stroms erfasst wird, steigt bzw. sinkt die Spannung um 100 mV/A[1]. Es wird nun durch einen Signalgenerator ein 50 Hz Wechselsignal generiert, welches gemessen werden soll.
Um den nötigen Strom von 2,0 A zu erzeugen, wird ein Kepco Leistungsverstärker eingesetzt der den Strom über einen Leistungswiderstand treibt. Das Strommess-Erweiterungsmodul erfasst dabei den fließenden Strom und erzeugt dazu eine Spannung, die gemessen werden soll. Den prinzipiellen Messaufbau zeigt Abbildung 6.5.

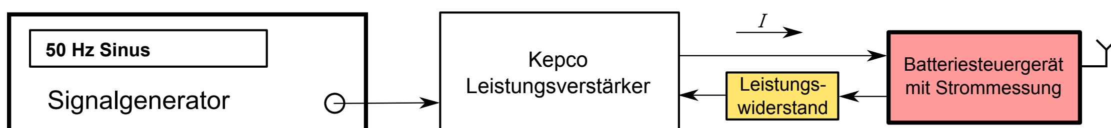


Abbildung 6.5.: Prinzipieller Messaufbau zum Test des Strommess-Erweiterungsmoduls

[1] ACS716 mit 6A max.: 100 mV/A, ACS716 mit 12A max.: 37 mV/A, ACS716 mit 25A max.: 18,5 mV/A

In Abbildung 6.6 ist die erzeugte Spannung am Strommess-Modul zu sehen. Zu erkennen ist, das durch den Signalgenerator erzeugte 50 Hz Sinussignal. Es lässt sich aus dieser Spannung der fließende Strom errechnen.
Dazu wird die Differenz der maximalen Spannung von ca. 2,26 V und der minimalen Spannung von 2,06 V bestimmt. Diese ergibt sich zu ca. 200 mV. Da es sich bei dem eingesetzten Strommess-IC um die Variante mit 6 A Maximalstrom handelt, kann der Strom zu 2 A bestimmt werden.

$$\Delta I = (\Delta \text{ Spannung in V}) / 100\,\text{mV/A} = ((2{,}26\,\text{V} - 2{,}06\,\text{V}) / 100\,\text{mV/A} = 2{,}0\,\text{A} \qquad (6.1)$$

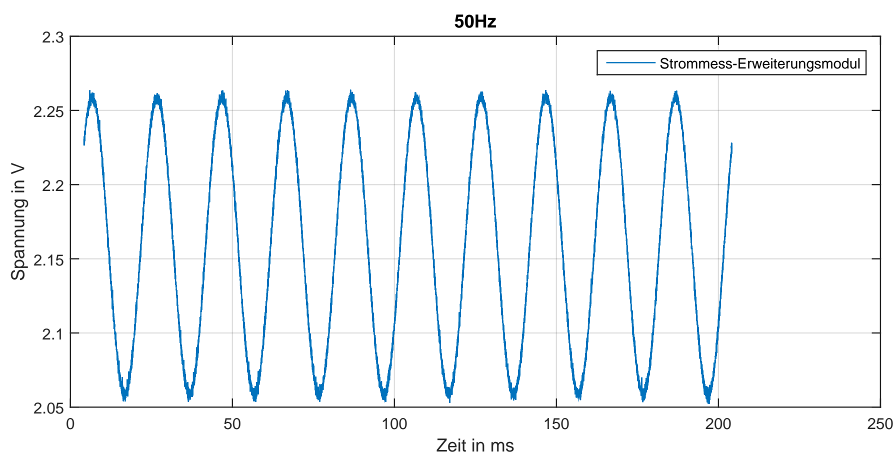


Abbildung 6.6.: Erzeugte Spannung durch das Strommess-Erweiterungsmodul

Dieser Test zeigt, dass das Strommess-Erweiterungsmodul wie erwartet funktioniert und somit für die Strommessung am Batteriesteuergerät eingesetzt werden kann.

6.1.2. Inbetriebnahme des Zellsensors

Die Abbildung 6.7 zeigt die Vorder- und Rückseite des bestückten Zellsensors. Einzelne Funktionsgruppen sowie einige aktive Bauelemente sind farblich hervorgehoben. Mittig ist der neu überarbeitete Anschluss zur elektrischen Verbindung zwischen dem Zellsensor und der Batteriezelle zu sehen. Dieser ist nun wesentlich kleiner dimensioniert als beim vorherigen Zellsensor.
Auf der Rückseite des Zellsensors befinden sich zwei JTAG-Interfaces. Es besteht die Möglichkeit der JTAG-Programmierung über eine Steckverbindung oder über einen Nadeladapter.

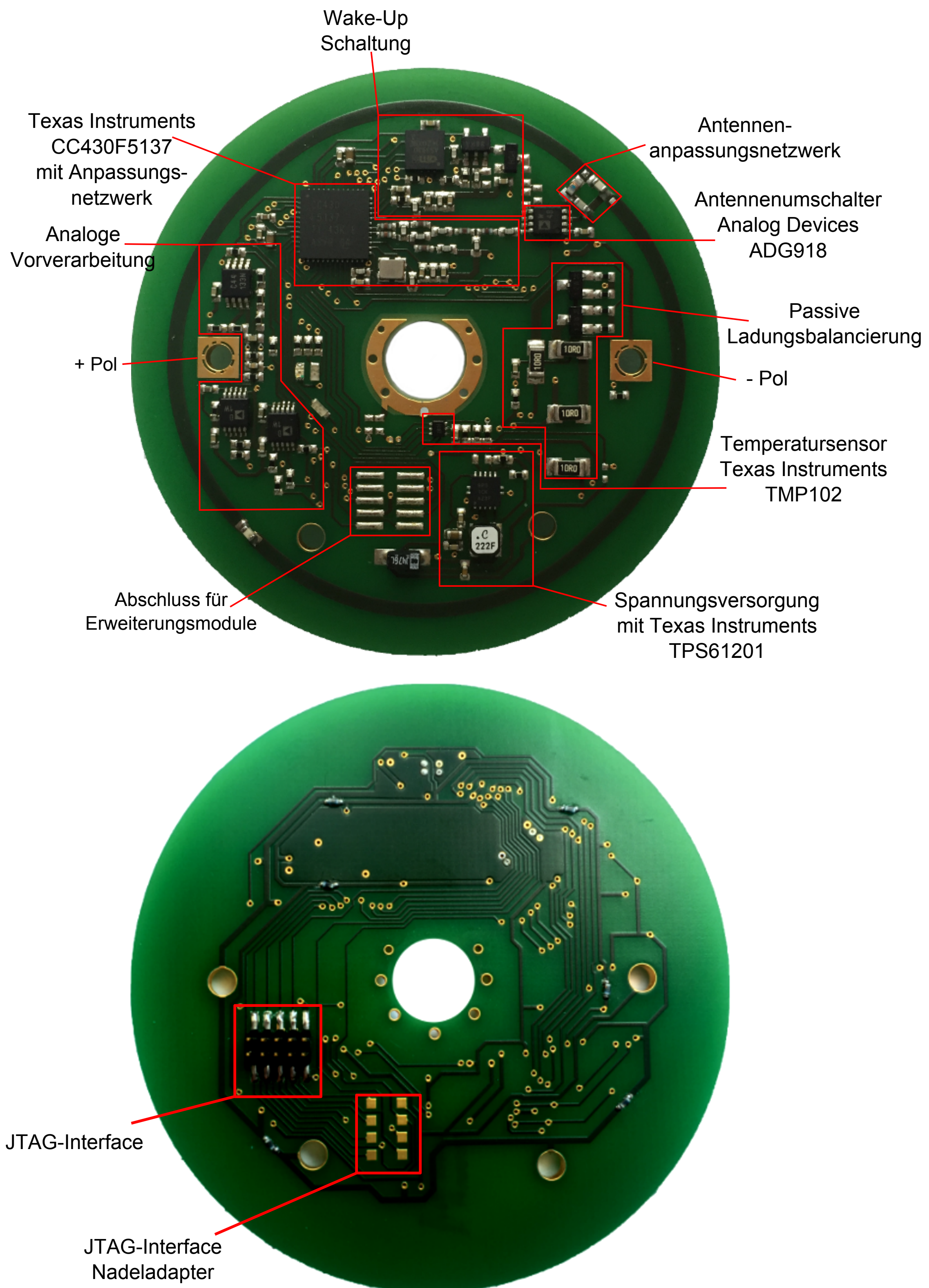


Abbildung 6.7.: Oben: Vorderseite des bestückten Zellsensors mit Beschriftung einzelner Funktionsteile und Bauteile.
Unten: Rückseite des Zellsensors mit den beiden JTAG-Interfaces

Impedanzanpassung der Schleifenantenne

Durch die, während der Entwicklung des Zellsensors durchgeführten Simulationen (siehe Anhang C), waren bereits S_{11} Streuparameter für die Schleifenantenne bekannt. Anhand dieser Streuparameter konnte ein Anpassungsnetzwerk berechnet werden. Ob dieses berechnete Netzwerk jedoch am bestückten Sensor die simulierten Ergebnisse liefert, ist noch zu testen. Abbildung 6.8 zeigt das, aus den simulierten Streuparametern berechnete, Anpassungsnetzwerk und deren Bauteilwerte.

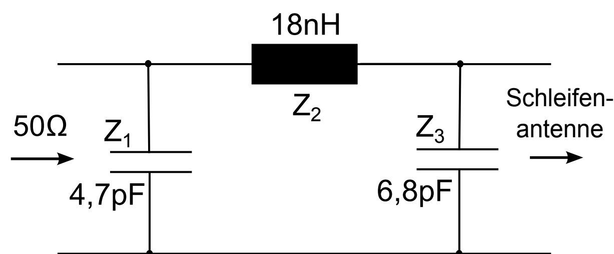


Abbildung 6.8.: Anpassungsnetzwerk der Schleifenantenne

Durch das, aus den Werten der Simulation berechnete Anpassungsnetzwerk, konnte die Resonanz der Schleifenantenne auf die gewünschten 434 MHz gebracht werden. Dabei nimmt der Eingangsreflexionsfaktor einen Wert von -17,92 dB an, was einer Leistungsreflexion von lediglich ca. 1,61% entspricht.

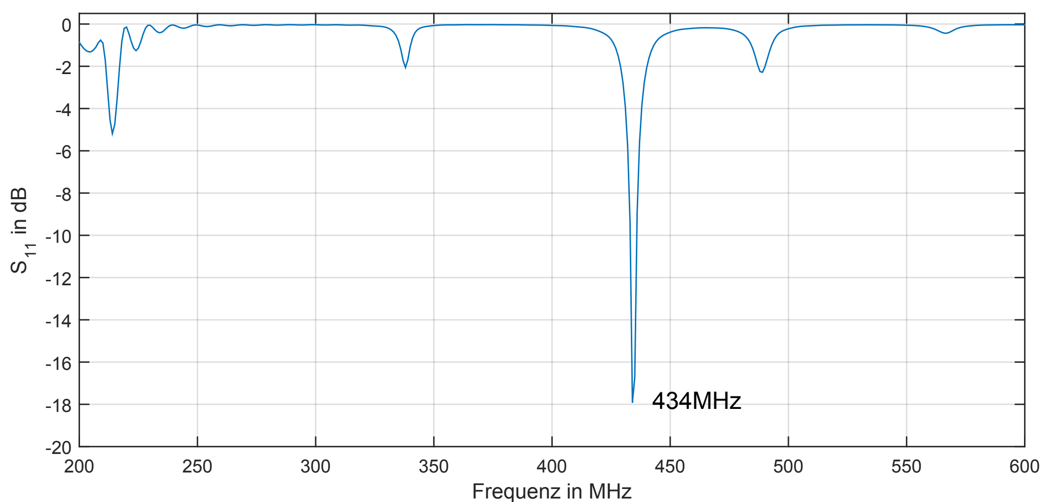


Abbildung 6.9.: Simulierte S_{11}-Parameter nach Anpassung der Schleifenantenne

Um die Abstrahleigenschaften mit dem berechneten Anpassungsnetzwerk zu testen, wurde ein Zellsensor im Abstand von 30 cm zu einem Spektrumanalysator aufgestellt. Dabei wurde die horizontale Abstrahlung des Zellsensors gemessen. Der Zellsensor strahlte dabei ein kontinuierliches Trägersignal mit einer Sendeleistung von 10 dBm ab. Abbildung 6.10 zeigt die durch den Spektrumanalysator empfangene Leistung. Zu sehen ist, dass sich der gesendete Träger deutlich vom Rauschen hervorhebt. Der Empfangspegel am Spektrumanalysator lag bei -38,66 dBm.

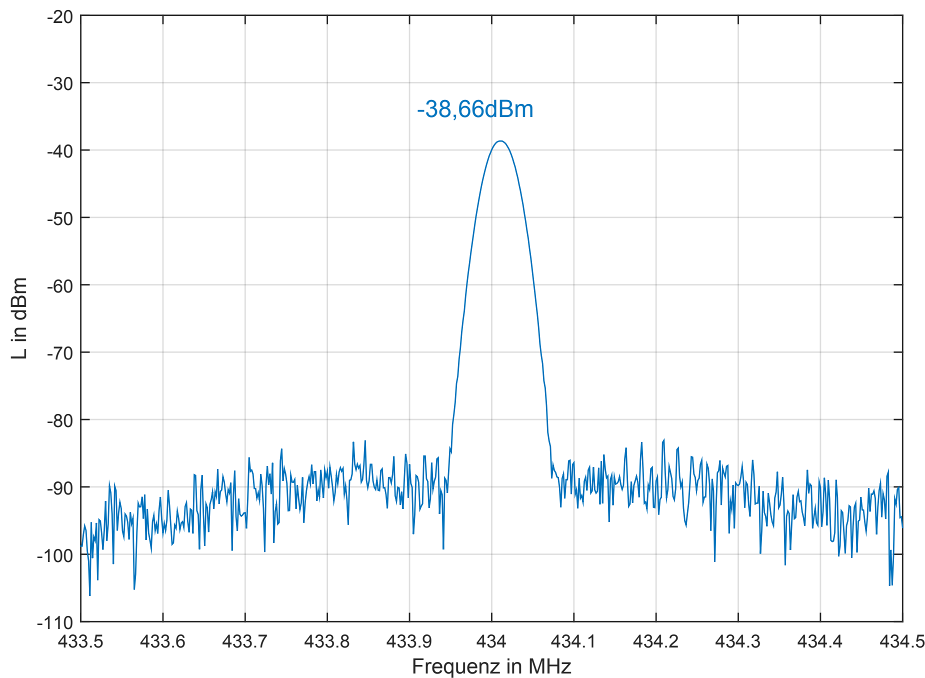


Abbildung 6.10.: Gemessene Abstrahlleistung der Schleifenantenne

Eine Datenkommunikation zwischen dem Zellsensor und dem Batteriesteuergerät ist damit ohne weiteres möglich. Erste Tests ergaben, dass damit eine paketorientierte Kommunikation zwischen Zellsensor und Steuergerät über eine Reichweite von >15 m möglich ist. Dies ist für die Erprobung sowie die Anwendung des Messsystems mehr als ausreichend. Daher wurde auf eine Vermessung der S_{11} Streuparameter mittels eines Netzwerk-Analysators an dieser Stelle verzichtet.

Spannungsversorgung

Ein bekanntes Problem der entwickelten Zellsensoren war der Einfluss des eingesetzten DC/DC-Wandlers TPS61201 auf die zu messende Zellspannung. Durch interne Schaltvorgänge wurden Störungen auf die Zellspannung übertragen. Diese Störungen hatten eine Amplitudenhöhe von ca. 20 bis 25 mV, abhängig von der Belastung durch die Elektronik des Zellsensors. Da sich die zu messenden Amplituden der Batteriespannung aber im Bereich von wenigen mV befinden, überlagerten die Störungen des DC/DC-Wandlers die zu messende Batteriespannung. Eine Lösung dieses Problems wurde in [62] entwickelt. Durch ein kurzzeitiges Abschalten des DC/DC-Wandlers, kurz vor der Messung, konnten die Störungen zum Messzeitpunkt verhindert werden.
Durch diese kurzzeitige Abschaltung der Spannungsversorgung kommt es aber auch zu einem kurzen Spannungseinbruch der 3,3 V Versorgungsspannung auf dem Zellsensor. Die Spannung kann dabei bis auf einen Wert von 2,7 V einbrechen. Es zeigte sich in der Arbeit [62], dass ein solcher Einbruch für die digitalen Bauelemente auf dem Zellsensor kein Problem darstellen.

Da nun aber auch eine analoge Baugruppe eingesetzt wird, ist der Einfluss dieser Abschaltung nochmals genauer zu untersuchen. Besonders anfällig für diesen Spannungseinbruch ist die Subtrahierschaltung der analogen Vorverarbeitung, da hier die abzuziehende Spannung durch einen Spannungsteiler eingestellt wird, der über die 3,3 V Versorgungsspannung versorgt wird. Schwankt nun diese 3,3 V Versorgungsspannung, schwankt auch die abzuziehende Spannung an der Subtrahierschaltung. Die nachfolgenden Berechnungen zeigen, dass sich die Spannungsschwankung der abzuziehenden Spannung bei einem maximalen Spannungsabzug im Bereich von 3,267 V und 2,673 V bewegt.

$$U_{3{,}3\,\mathrm{V}} = \frac{R_2 + R_{\mathrm{Rheo}_{\mathrm{max.}}}}{R_1 + R_2 + R_{\mathrm{Rheo}_{\mathrm{max.}}}} \cdot U_{\mathrm{ref.}} = \frac{1\,k\Omega + 100\,k\Omega}{1\,k\Omega + 1\,k\Omega + 100\,k\Omega} \cdot 3,3\,V = 3,267\,V \tag{6.2}$$

$$U_{2{,}7\,\mathrm{V}} = \frac{R_2 + R_{\mathrm{Rheo}_{\mathrm{max.}}}}{R_1 + R_2 + R_{\mathrm{Rheo}_{\mathrm{max.}}}} \cdot U_{\mathrm{ref.}} = \frac{1\,k\Omega + 100\,k\Omega}{1\,k\Omega + 1\,k\Omega + 100\,k\Omega} \cdot 2,7\,V = 2,673\,V \tag{6.3}$$

Daraus ergibt sich der Schwankungsbereich $\Delta_{\mathrm{Schwankung}}$ der Subtrahierschaltung.

$$\Delta_{\mathrm{Schwankung}} = U_{3{,}3\,\mathrm{V}} - U_{2{,}7\,\mathrm{V}} = 3,267\,V \;-\; 2,673\,V \;=\; 0,594\,V \tag{6.4}$$

Diese Schwankungen machen sich in der Ausgangsspannung der Subtrahierschaltung bemerkbar. Der Gleichspannungsbereich der Messspannung ist nun nicht mehr auf einer konstanten Spannung, sondern bewegt sich innerhalb des Schwankungsbereichs $\Delta_{\mathrm{Schwankung}}$.
Ist die Subtrahierschaltung nicht auf den maximalen Spannungsabzug eingestellt, ist die Auswirkung auf die Ausgangsspannung auch geringer. Bei einem minimalen Spannungsabzug liegt der Schwankungsbereich der Spannung aber immer noch bei ca. 200 mV.

Eine Messung der 3,3 V Versorgungsspannung und der Spannung am AD-Wandler Eingang bei einer Verstärkung V = 0 bestätigen diese Berechnung. Abbildung 6.11 zeigt den Einfluss durch die Spannungseinbrüche der 3,3 V Spannungsversorgung auf die analoge Vorverarbeitung.

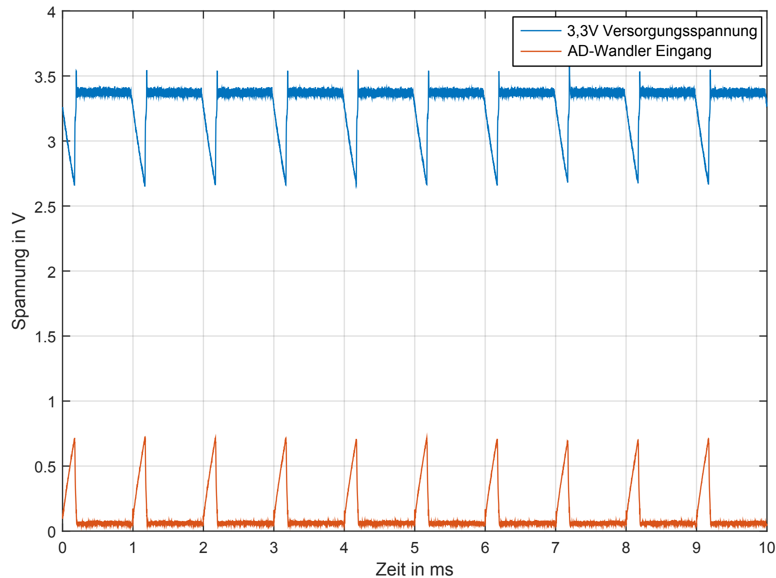


Abbildung 6.11.: Einfluss der Abschaltung des TPS61201 auf die analoge Vorverarbeitung

In der Abbildung 6.11 ist zu erkennen, dass sich die Einbrüche in der Spannungsversorgung nahezu gleich auf die Messspannung niederschlagen. Wird die Messspannung nun noch verstärkt, wird der Fehler noch vergrößert. Es kommt zu einem Fehler in der Messung bzw. die Messspannung geht über den Referenzspannungsbereich des AD-Wandlers hinaus.

Dieser Einfluss auf die Messspannung ist durchaus kritisch einzuschätzen. Aus der Konzeptionierung der analogen Vorverarbeitung in Kapitel 3.5.2 ist bekannt, dass der Gleichspannungsanteil der Messspannung so gering wie möglich sein sollte. Ein Schwanken des Gleichspannungsanteils im Bereich von mehreren 100 mV ist dabei nicht zulässig und verhindert die Möglichkeit, eine genaue Messung durchzuführen. Aus diesem Grund wurde beschlossen, alle weiteren Messungen, die mit der analogen Vorverarbeitung durchgeführt werden, mit einer externen Spannungsversorgung durchzuführen. Dadurch wird die zu messende Zellspannung nicht durch Störungen des DC/DC-Wandlers beeinflusst und es finden keine Spannungseinbrüche der 3,3 V Spannungsversorgung durch Abschaltung des DC/DC-Wandlers statt.
Ein möglicher Lösungsansatz zur Stabilisierung der Spannungsversorgung wird im Anhang B behandelt, findet aber keinen Einsatz mehr in den weiteren Messungen und in dieser Arbeit.

Steuerung der analogen Vorverarbeitung

In diesem Abschnitt soll die Steuerung der analogen Vorverarbeitung getestet werden. Implementiert wurde neben der automatischen Einstellung auch eine manuelle Einstellung der Vorverarbeitung. Manuelle Einstellungen lassen sich über das Batteriesteuergerät, mit den entsprechenden Einstellungswerten für die Rheostaten, übertragen. Einzelheiten zu den entsprechenden Befehlen finden sich im Anhang F.

Einen Funktionstest der automatischen Steuerung für die analogen des Vorverarbeitung zeigt die Messung in Abbildung 6.12. Dabei ist das Messsignal zu sehen, wie es durch die beiden Stufen der analogen Vorverarbeitung soweit bearbeitet wird, dass es durch den internen 12 Bit AD-Wandler des Mikrocontrollers gemessen werden kann. Diese beiden Stufen wurden bereits in Abschnitt 5.1.3 vorgestellt. Zu sehen ist zunächst die erste Stufe, in der der Gleichspannungsanteil abgezogen wird, bis die untere Grenzspannung erreicht wird. Ist diese erreicht, beginnt die 2. Stufe der analogen Vorverarbeitung. Dabei wird dass Messsignal bis zum Erreichen der oberen Grenzspannung verstärkt.

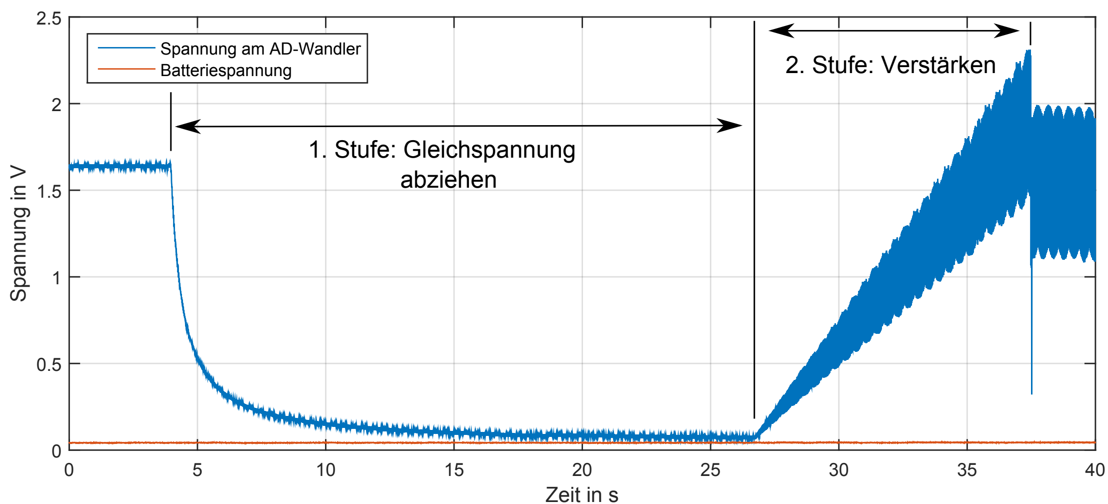


Abbildung 6.12.: Steuerung der analogen Vorverarbeitung

Der hintere Einbruch der verstärkten Messspannung, von ca. 2,30 V zu ca. 2,00 V, ist damit zu erklären, dass bei der Erreichung der oberen Grenzspannung, der interne DCO-Takt von 16 MHz auf 1 MHz umgeschalteten wird. Dadurch steigt die 3,3 V Versorgungsspannung leicht an, da die Strombelastung durch den Mikrocontroller hier ca. um den Faktor 10 sinkt [32]. Dies wirkt sich mit einem Absinken des Messsignals aus. Da die Burstmessung aber auch mit dem 16 MHz DCO-Takt durchgeführt wird, ist der erreichte Verstärkungsgrad richtig eingestellt. Bei dem in der Abbildung 6.12 zu sehenden Messsignal handelt es sich um ein 50 Hz Anregungssignal mit 1,2 A Wechselspannung. Dieses Signal konnte in

diesem Fall um den Faktor V = 46,12 verstärkt werden. Durch eine Rechnung lässt sich erkennen, ob die Grenzwerte eingehalten wurden. Als oberere Grenzspannung wurde 2,30 V eingestellt. Dies lässt sich auch in der Abbildung 6.12 erkennen. Die untere Grenzspannung wurde auf 30 mV gesetzt. Berechnen lässt sich dieser Grenzwert durch den Gleichanteil der Messspannung vor der DCO-Umschaltung mit ca 1,45 V. Mit dem Wert der Verstärkung lässt sich die untere Grenzspannung wie folgt berechnen:

$$V_{\text{Untere Grenzspannung}} = \frac{\text{Gleichspannung nach Verstärkung}}{\text{Verstärkung}} = \frac{1{,}45\,\text{V}}{46{,}12} = 31{,}52\,\text{mV} \tag{6.5}$$

Es ist zu sehen, dass die automatische Steuerung der analogen Vorverarbeitung wie erwartet funktioniert und die gesetzten Grenzwerte eingehalten werden.

Inbetriebnahme der dynamischen Laufzeitmessung

Die Inbetriebnahme der dynamischen Laufzeitmessung soll zeigen, ob das implementierte Verfahren genau genug ist, um eine verlässliche Aussage über das Laufzeitverhalten während der synchronisierten Messungen machen zu können. Dabei soll ein Vergleich zwischen den aufgenommenen Messdaten eines Oszilloskops und den mit der implementierten Laufzeitmessung aufgenommenen Messdaten durchgeführt werden.
Es wurde die Laufzeit von 1.500 Messungen aufgenommen und analysiert. Die dabei aufgenommenen Laufzeiten sind in Abbildung 6.13 zu sehen. Darin ist zu sehen, dass alle Messwerte ungefähr den gleichen zeitlichen Wert ergeben. Daran erkennt man, dass die implementierte Messung stabile Werte liefert. Dennoch sind Abweichungen an den Messwerten zu erkennen. Diese liegen, ausgehend vom Mittelwert bei bei 15,86 μs, im Bereich von $\pm$ 0,85 μs.

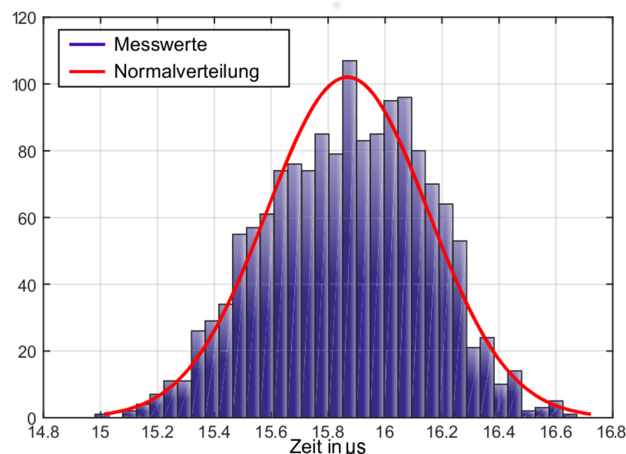


Abbildung 6.13.: Ermittelte Laufzeiten der dynamischen Laufzeitermittlung mit 1.500 Messwerten: Mittelwert = 15,86 μs, σ = 0,2832 μs

Ob das implementierte Messverfahren auch qualitativ präzise Messwerte liefert, muss im Vergleich zu einer Messung der Laufzeit, die mit einem Oszilloskop gemessen wird, überprüft werden. Für eine solche Messung wird am Batteriesteuergerät der Zeitpunkt des Aussendens und der Ankunft am Zellsensor mit einem Oszilloskop gemessen. Die Signalisierung der Aussendung bzw. Ankunft des Impulses wird dabei an einem Messpin ausgegeben. Die Ergebnisse dieser Messung sind in Abbildung 6.14 zu sehen. Es zeigt sich, dass die Messung durch das Oszilloskop ähnliche Messwerte liefert wie die implementierte Laufzeitmessung.

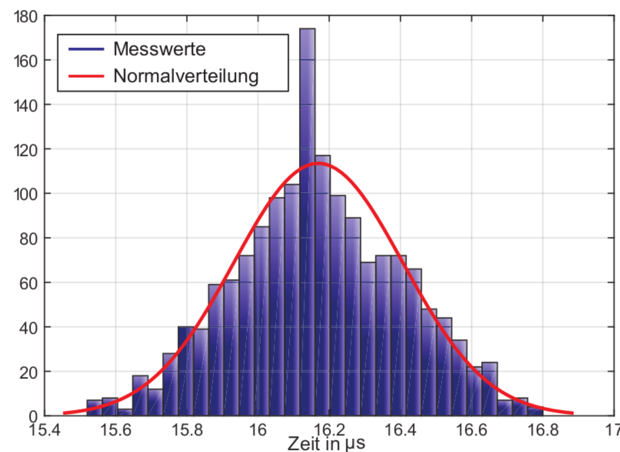


Abbildung 6.14.: Ermittelte Laufzeiten mittels Oszilloskop mit 1.500 Messwerten: Mittelwert = 16,16μs, σ=0,2274μs

Der Mittelwert der Oszilloskop-Messung liegt bei 16,16μs. Dies entspricht 0,3 μs mehr, als bei dem Mittelwert des implementierten Messverfahrens. Dies zeigt, dass die Ergebnisse der dynamischen Laufzeitmessung durchaus auch eine qualitative Aussage der Laufzeit zwischen Batteriesteuergerät und Zellsensor liefern kann. In den beiden Abbildungen 6.13 und 6.14 lässt sich auch erkennen, dass die aufgenommenen Messwerte Gleichverteilt sind und sich durch eine Normalverteilung annähern lassen. Dadurch lässt sich die prozentuale Verteilung und Abweichung der Messwerte zum Mittelwert berechnen. So ergibt die Standardabweichung der Normalverteilungen aus der implementierten Laufzeitmessung, dass 68,3% [56] aller Messwerte eine maximale Abweichung von 0,2832μs haben.
In der Tabelle 6.1 sind nochmals alle wichtigen Ergebnisse zur Laufzeitmessung zusammengefasst.

Tabelle 6.1.: Vergleich der Ergebnisse der Laufzeitmessung

	Mittelwert	**max. Abweichung**	**Standardabweichung** σ
Implementierte Laufzeitmesung	15,86 μs	$\pm$ 0,85 μs	0,2832 μs
Oszilloskop Laufzeitmesung	16,16 μs	$\pm$ 0,64 μs	0,2274 μs

Phasenrichtige Spannungsaufnahme des Messsystems

Bevor erste Messungen gestartet werden können, muss überprüft werden, ob eine phasenrichtige Aufnahme der Spannungswerte durch das Messsystem möglich ist. Dazu wird mittels eines Signalgenerators eine 10 Hz Wechselspannung mit 200 mV Amplitude an das Batteriesteuergeräts sowie an einen Zellsensor gegeben. Abbildung 6.15 zeigt den dazugehörigen Messaufbau. Erwartet wird dabei, dass die Messwerte des Batteriesteuergerät sowie des Zellsensors phasenrichtig aufgenommen werden.

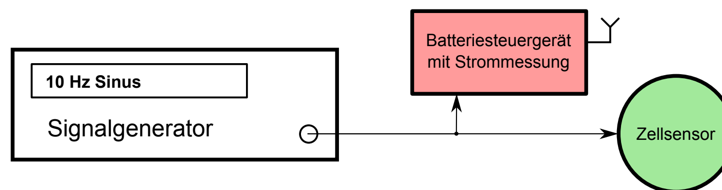


Abbildung 6.15.: Messaufbau für die phasenrichtige Strom- und Spannungsaufnahme des Messsystems

Die Beurteilung der phasenrichtigen Messung erfolgt dabei rein optisch. Wie in Abbildung 6.16 zu sehen ist, liegen die aufgenommenen Spannungen nahezu ideal übereinander. Eine Phasendifferenz zwischen den beiden Signalen ist nicht zu erkennen. Ebenso ist bei beiden Messungen die Amplitudenhöhe von 200 mV zu erkennen.

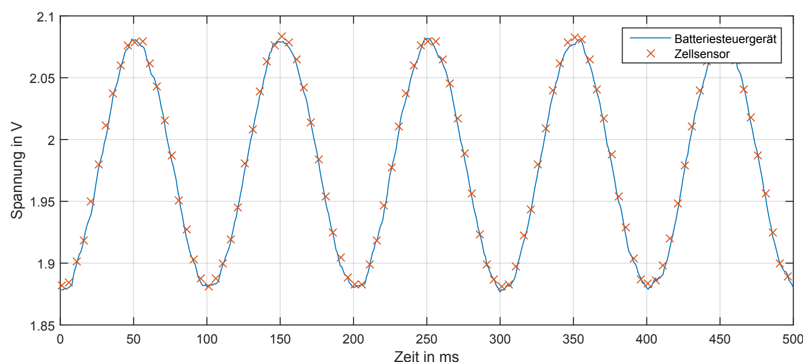


Abbildung 6.16.: Überprüfung der phasenrichtigen Spannungsaufnahme des Zellsensors und des Batteriesteuergeräts

Da nun alle für die Impedanzspektroskopie erforderlichen Hard- und Softwareteile getestet werden konnten, können ersten Impedanzmessungen durchgeführt werden.

6.2. Impedanzmessungen

In diesem Abschnitt sollen mit dem entwickelten Messsystem erste Impedanzmessungen durchführen werden. Insbesondere soll die Synchronität zwischen Strom- und Spannungsmessungen überprüft und angepasst werden. Dazu wird zunächst eine Testimpedanz vermessen. Verläuft diese Messung positiv, kann die Impedanzmessung an realen Batteriezellen getestet werden.

6.2.1. Vermessung einer Testimpedanz

Für die erste Messung wird eine Testimpedanz aufgebaut. Der Vorteil einer solchen Testimpedanz liegt darin, dass sich hier die Bauteilwerte nicht verändern. Bei einer Batteriezelle können sich von Messung zu Messung die Messwerte der Impedanz verändern. Diese Veränderungen werden unter anderem hervorgerufen durch die Alterung der Zelle, durch die Veränderung des Ladezustands sowie der Änderung der Zelltemperatur. Da bereits kleinste Änderungen dieser Werte das Impedanzspektrum beeinflussen, wird für die erste Inbetriebnahme eine Testimpedanz entworfen.
Anhand dieser Testimpedanz kann nun das Messverfahren erprobt und die Synchronität angepasst werden. Bei der Entwicklung dieser Testimpedanz wurde versucht, in den erwartenden Impedanzbereich von wenigen mΩ zu gelangen. Ebenfalls soll der Frequenzbereich, der bei der Impedanzspektroskopie durchfahren wird, annähernd erreicht werden. Abbildung 6.17 zeigt die dabei entstandene Testimpedanz.

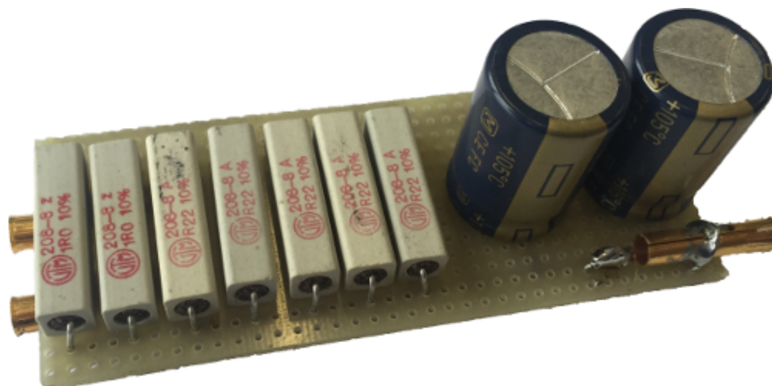


Abbildung 6.17.: Entwickelte Testimpedanz

Die Testimpedanz wird, wie das in Abschnitt 2.3.1 vorgestellte Batteriemodell, aufgebaut. Dieses Batteriemodell besteht aus einem Vorwiderstand und einem daran anschließenden RC-Glied. Das daraus entstehende Impedanzspektrum bildet in der Nyquist-Darstellung einen Halbkreis. Die Breite dieses Halbkreises wird bestimmt durch den ohmschen Anteil

im RC-Glied. Da dieser im mΩ Bereich liegen soll, werden fünf mal 220 mΩ Widerstände parallel geschaltet. Dadurch erreicht der Impedanz-Halbkreis eine Breite von 44 mΩ. Der Frequenzbereich sollte mindestens von 2 kHz bis in einen niedrigen Frequenzbereich von 1 Hz reichen. Dabei soll der Halbkreis möglichst komplett abgefahren werden. Durch die erarbeitete Formel 2.29 aus Abschnitt 2.3.1 wird ersichtlich, dass dazu die eingesetzte Kapazität möglichst groß gewählt werden muss.
Eingesetzt werden nun zwei parallel geschaltete 10.000 μF Kondensatoren. Dadurch wird erreicht, dass der Imaginärteil der Impedanz bei 2 kHz weitestgehend den Nulldurchgang erreicht. Dadurch kann der komplette Halbkreis abgefahren werden.
Da sich die Impedanzmessung nun im Bereich von wenigen mΩ bewegt, ist der Serienwiderstand (ESR[2]) der eingesetzten Kapazitäten zu beachten. Da diese auch Einfluss auf das erzeugte Impedanzspektrum haben. Abbildung 6.18 zeigt die entwickelte Schaltung der Testimpedanz.

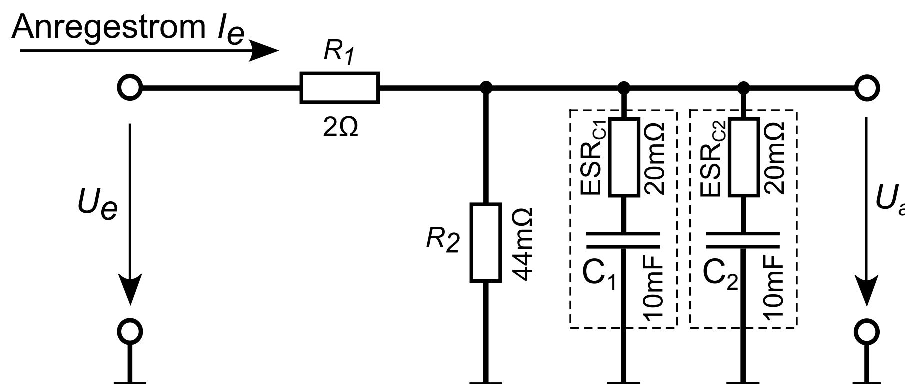


Abbildung 6.18.: Schaltplan der entwickelten Testimpedanz

Der Widerstand R_1 dient in dieser Schaltung lediglich als Strombegrenzung, hat aber keinen Einfluss auf die gemessene Impedanz, da die Ausgangsspannung U_a lediglich über den Widerstand R_2 und die beiden Kondensatoren C_1 und C_2 gemessen wird.

Angeregt wird die Testimpedanz bei dieser Messung mittels des FuelCon TrueData-EIS. Dies hat den Vorteil, dass der Anregestrom I_e genau eingestellt werden kann. Die Strommessung erfolgt über das Stommess-Erweiterungsmodul des Batteriesteuergeräts. Die Ausgangsspannung U_a der Testimpedanz wird von einem Zellsensor gemessen. Den prinzipiellen Messaufbau zur Vermessung der Testimpedanz zeigt Abbildung 6.19.

[2]Equivalent Series Resistance

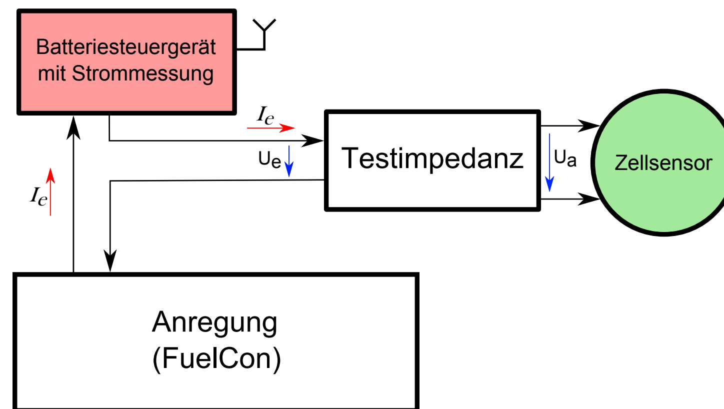


Abbildung 6.19.: Prinzipieller Messaufbau zur Vermessung der Testimpedanz

Die Messung erfolgt in 13 unterschiedlichen Frequenz-Messpunkten. Diese sind in Tabelle 6.2 mit den dazugehörigen Burstfrequenzen angegeben. Diese willkürlichen Frequenzwerte ergaben sich durch eine erste Testmessung mit dem FuelCon TrueData-EIS und wurden zum Vergleich der Messergebnisse übernommen.

Tabelle 6.2.: Frequenz-Messpunkte der Testimpedanz-Messung

Messpunkt	Frequenz	Burstfrequenz
1	1,0 Hz	500 Hz
2	5,24 Hz	500 Hz
3	9,10 Hz	1.000 Hz
4	20,84 Hz	1.000 Hz
5	36,20 Hz	1.000 Hz
6	62,87 Hz	1.000 Hz
7	82,87 Hz	1.000 Hz
8	143,95 Hz	2.000 Hz
9	250,04 Hz	8.000 Hz
10	434,32 Hz	8.000 Hz
11	754,42 Hz	8.000 Hz
12	1310,44 Hz	8.000 Hz
13	2276,25 Hz	8.000 Hz

Als Anregestrom wurde ein Wechselstrom von 1,2 A gewählt. Durch den niedrig gewählten Wechselanteil liegt die zu messende Wechselspannung auch in einem kleinen mV Bereich, wie sie bei einer Impedanzspektroskopie an einer realen Batteriezelle zu erwarten ist.
Dadurch, dass es sich bei dieser Messung um keine aktive Batteriezelle mit überlagertem Gleichspannungsanteil handelt, ist der Gleichspannungsanteil der zu messenden Spannung

sehr gering. Aus diesem Grund kann das Messsignal durch die analoge Vorverarbeitung sehr gut verstärkt werden. Allerdings muss für die eine Messung, mit geringem Gleichanteil < 1,65 V, die Subtrahierschaltung geändert werden. Dazu sind lediglich die beiden Widerstände R_1 und R_2 aus der Schaltung 4.9 auf Seite 99 zu entfernen. Dies bewirkt, dass wieder ein Gleichspannungsabzug von 0,0 V möglich ist.

Abbildung 6.20 zeigt das Ergebnis der ersten Impedanzmessung der Testimpedanz. Zu sehen ist zum einen das theoretische Impedanzspektrum, dass sich aus der Simulation der Schaltung ergibt. Zum anderen sind die Ergebnisse der Messung mit dem FuelCon TrueData-EIS (EIS Meter) und der Messung mit dem Zellsensor zu sehen. Beispielhaft sind drei verschiedene Frequenzpunkte im Spektrum eingezeichnet, um eine Orientierung der Frequenzpunkte zu bekommen.

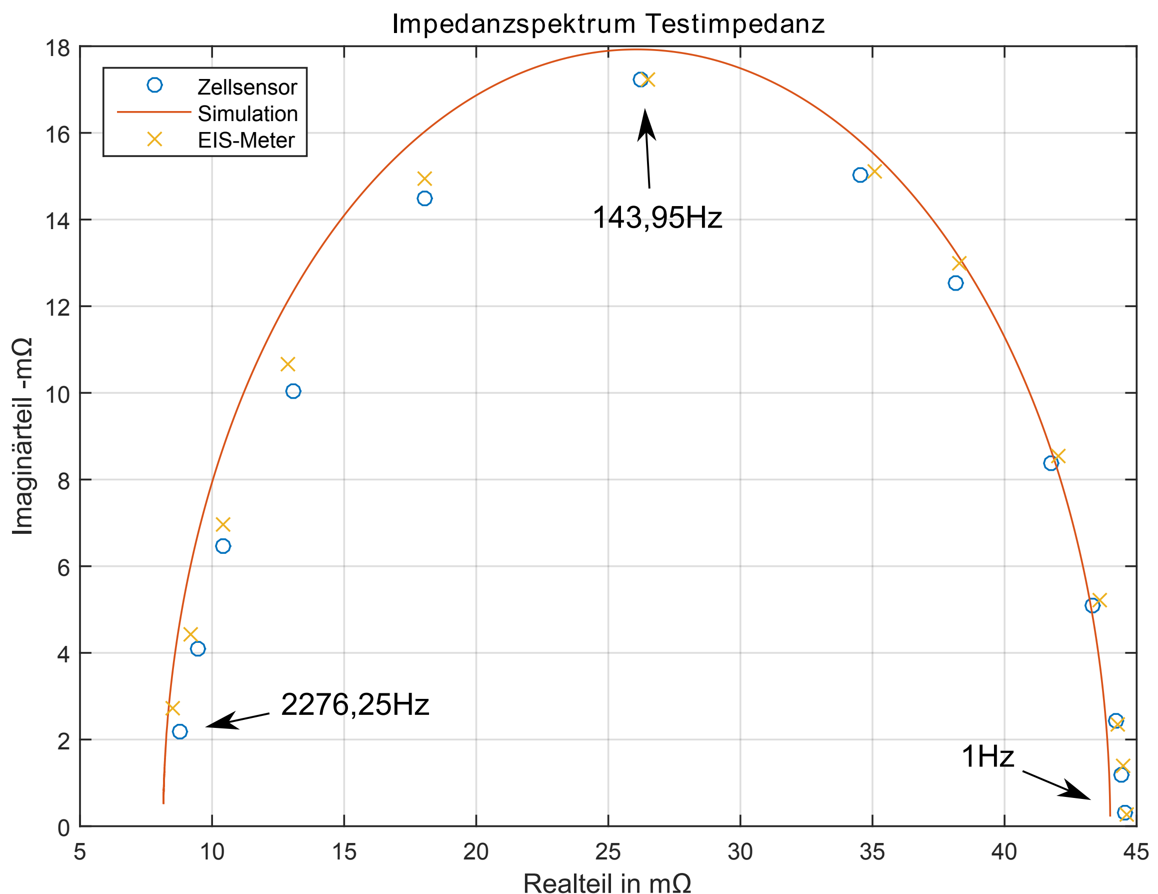


Abbildung 6.20.: Erste Impedanzmessung an der Testimpedanz mit einer Synchronisationszeit von 72,2μs

Zu erkennen ist, dass sich die beiden Messungen dem simulierten Spektrum annähern. Es gibt aber vor allem noch Unterschiede im Imaginärteil des Spektrums. Es wird nun ange-

nommen, dass die Messwerte des FuelCon TrueData-EIS die genaueren Messwerte liefert. Die Abweichung zwischen den simulierten Werten und denen des FuelCon TrueData-EIS werden in den Abweichungen der verwendeten Bauteile liegen. Vor allem die eingesetzten Kondensatoren können eine maximale Abweichung von bis zu $\pm 20\,\%$ [55] aufweisen. Simulationen ergaben, dass diese Abweichung durch einen leicht erhöhten ESR der Kondensatoren zustande kommen kann.

Nun ist die Abweichung zwischen den Werten des FuelCon TrueData-EIS und den Messwerten des in dieser Arbeit entwickelten Messsystems zu untersuchen. Es ist zu erkennen, dass der Realteil der Messwerte bei beiden Messungen nahezu übereinstimmten. Lediglich die Imaginärteile liegen etwas auseinander. Bei niedrigen Frequenzen ist die Abweichung geringer als bei hohen Frequenzen. Dieses Verhalten deutet auf eine fehlerhafte Synchronisationszeit zwischen Strom- und Spannungsmessung hin und geht aus den Berechnungen in Abschnitt 3.3 hervor. Durch diese Rechnungen wurde deutlich, dass eine zu lange Verzögerung zwischen Strom- und Spannungsaufnahme sich vor allem in einem zu großen Imaginärteil auswirkt. Im Umkehrschluß bedeutet dies, dass eine zu kurze Synchronisationszeit zwischen Strom- und Spannungsaufnahme zu einem geringeren Imaginärteil führt.

Durch diese Messung an der Testimpedanz lässt sich also feststellen, dass eine Erfassung des Impedanzspektrums möglich ist, das Messsystem selber aber noch nicht ausreichend genug synchronisiert ist. Die Messungen an der Testimpedanz wurden mit der in Abschnitt 3.3 berechneten Verzögerungszeit zwischen Taktaussendung und Stromaufnahme von 72,2 μs aufgenommen. Die Synchronität des Messsystems muss also weiter untersucht und die Synchronisationszeit korrigiert werden.

6.2.2. Untersuchung der Synchronität des Messsystems

Die vorherige Messung an der Testimpedanz hat gezeigt, dass die Zeit zwischen der Aussendung des Messimpulses und der Aufnahme der Strom- und Spannungswerte zu gering ist. Es wird nun nach möglichen Ursachen gesucht, die eine synchrone Aufnahme verhindern.

Durch die implementierte Laufzeitmessung ist bekannt, dass die Laufzeit des Messimpulses zwischen Batteriesteuergerät und Zellsensor nicht wie in der Rechnung in Abschnitt 3.3 18,4μs betragen, sondern lediglich 15,86μs. Mit dieser verkürzten Laufzeit ergibt sich eine theoretische Synchronisationszeit von 69,86μs. Die Zeit müsste sich also theoretisch verkürzen und nicht verlängern, wie es sich durch die Messung der Testimpedanz ergeben hat. Es muss also weitere Zeiten geben, die Einfluss auf die Synchronität des Messsystems haben.

Betrachtet wurde bisher nicht die Zeit, die der AD-Wandler des Batteriesteuergeräts benötigt, um die Spannung des Strommess-Erweiterungsmoduls aufzunehmen. Der AD-Wandler ist so eingestellt, dass dieser 3,2μs für eine Spannungsaufnahme benötigt. Diese Zeit verkürzt die nötige Wartezeit für eine synchrone Strom- und Spannungsaufnahme auf 66,66μs. Abbildung 6.21 gibt einen Überblick über die bisher ermittelten Zeiten, die für die Synchronität des Messsystems verantwortlich sind.

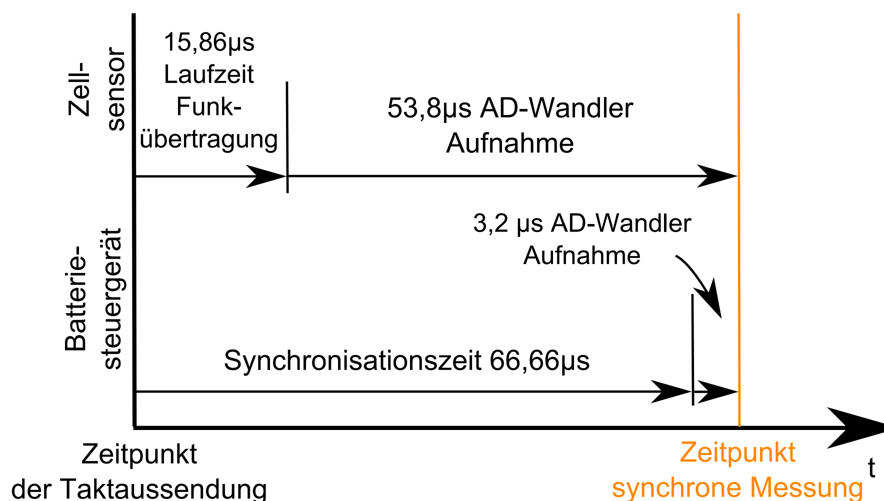


Abbildung 6.21.: Zeiten für die synchrone Strom- und Spannungsmessung

Bei einer Untersuchung der analogen Vorverarbeitung wurde festgestellt, dass diese einen Amplituden- und einen linearen Phasengang besitzt, und somit eine Verzögerung des Messsignals verursacht. Abbildung 6.22 zeigt den gemessenen Amplituden- und Phasengang der analogen Vorverarbeitung.

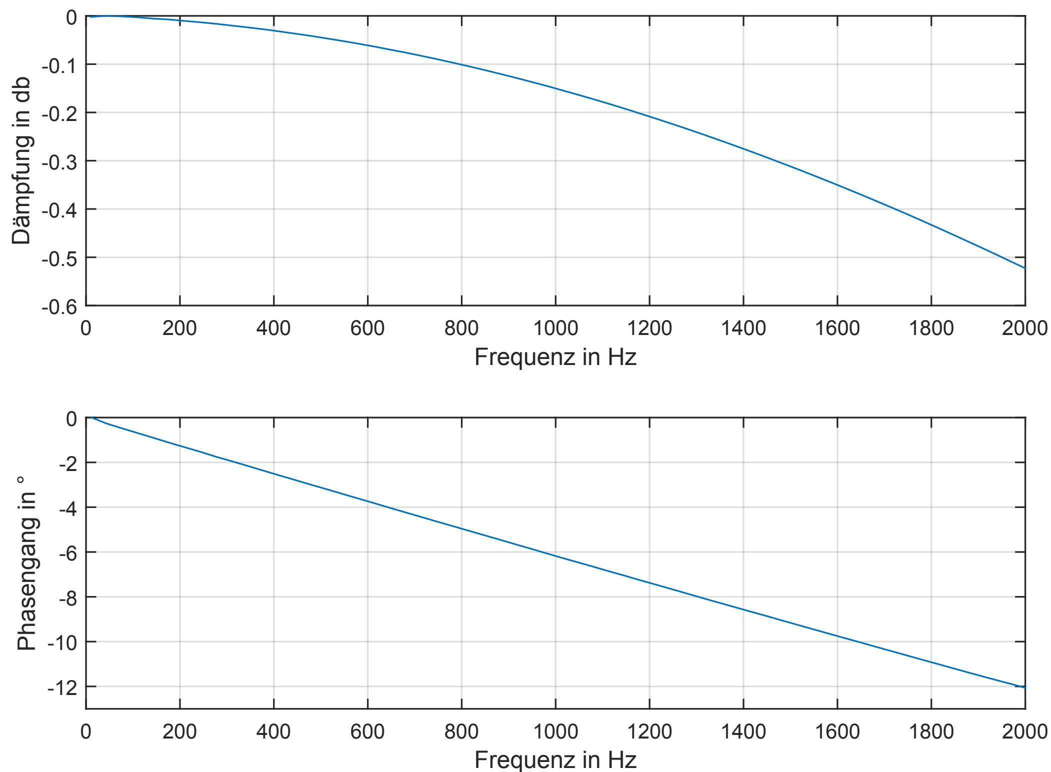


Abbildung 6.22.: Gemessener Amplituden- und Phasengang der analogen Vorverarbeitung

Durch den gemessenen Phasengang lässt sich eine zusätzlich benötigte Zeit, die für die Synchronisation notwendig ist, berechnen. Da es sich hierbei um einen linearen Phasengang handelt, kann ein beliebiger Punkt genommen werden, um die daraus resultierende Laufzeitverzögerung zu ermitteln. In diesem Fall wird der Punkt bei 2 kHz gewählt. Die durch den Phasengang verursachte Laufzeitverzögerung ergibt sich dadurch zu 16,66 μs.

$$t_{\text{Phasengang}} = -\frac{\text{Periodenzeit [s]}}{360\,^\circ} \cdot \text{Phasengang } [^\circ] = -\frac{\frac{1}{2\,\text{kHz}}}{360\,^\circ} \cdot \text{-}\,12\,^\circ = 16{,}66\,\mu\text{s} \tag{6.6}$$

Eine weitere Analyse des Messsystems ergab, dass neben der analogen Vorverstärkung auch das Strommess-Erweiterungsmodul eine Verzögerung zwischen den durchfließenden Strom und der daraus resultierenden Spannung aufzeigt. Der Hersteller des Strommess-IC ACS716 gibt für diesen Baustein eine typische Reaktionszeit von 4 μs [49] an, die in die Synchronisationszeit mit einbezogen werden muss.

Aus dieser Analyse des Messsystems geht hervor, dass eine unterschiedliche Phasenverschiebung zwischen Strom- und Spannung herrscht. Um dennoch synchrone Messwerte zu erhalten, muss die Zeitdifferenz zwischen Strom- und Spannung ermittelt und auf die Synchronisationszeit beaufschlagt werden. Diese Zeitdifferenz zwischen Messspannung

und Strom berechnet sich zu 12,66μs.

$$\Delta t = 16{,}66\mu s - 4{,}0\mu s = 12{,}66\mu s \tag{6.7}$$

Die Verschiebung zwischen Strom- und Spannung veranschaulicht Abbildung 6.23.

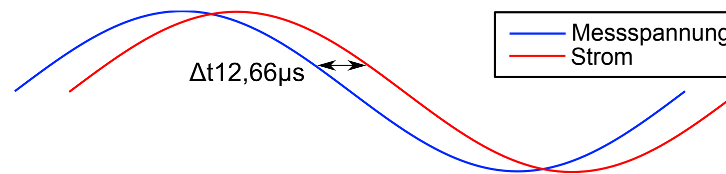


Abbildung 6.23.: Veranschaulichung der zeitlichen Verschiebung zwischen Messspannung und Strom

Es konnten durch die Analyse des Messsystems weitere Zeiten ermittelt werden, die bei der Berechnung der Synchronisationszeit miteinbezogen werden müssen. So benötigt der Zellsensor, nach dem Aussenden des Taktsignals insgesamt 69,86 μs bis die Spannungsaufnahme stattfindet. Das Batteriesteuergerät benötigt für die Messwerterfassung des Stromes insgesamt 3,2 μs. Die Verschiebung zwischen Strom- und Spannung beträgt 12,66 μs. Aus diesen Zeiten ergibt sich eine neue Synchronisationszeit von 79,12 μs.

$$\text{Synchronisationszeit} = (69{,}86\,\mu s + 12{,}66\,\mu s) - 3{,}2\,\mu s = 79{,}12\,\mu s \tag{6.8}$$

Für eine synchrone Messung ist der Strom erst 12,66 μs nach der Spannung zu messen. In Abbildung 6.24 werden die einzelnen Zeiten nochmals aufgeführt und entsprechend eingeordnet.

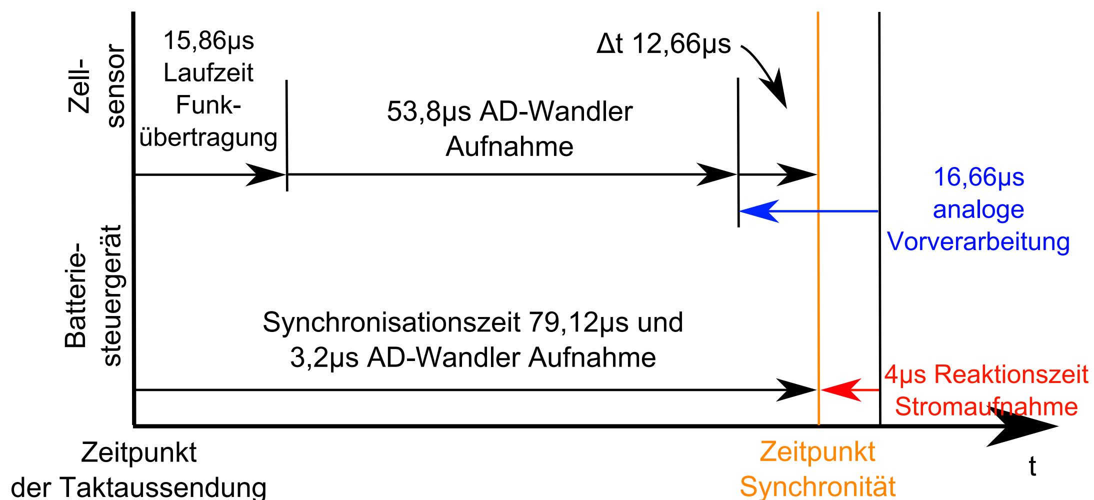


Abbildung 6.24.: Zusammensetzung der Zeiten für eine synchrone Strom- und Spannungsmessung

Mit dieser ermittelten Synchronisationszeit, kann nun eine erneute Messung an der Testimpedanz durchgeführt werden. Auch der in der Abbildung 6.22 zu sehende Amplitudengang kann nun bei der neuen Messung in die Berechnung der Impedanzen miteinbezogen werden.

Die erneute Messung der Testimpedanz mit angepasster Synchronisationszeit zeigt Abbildung 6.25.

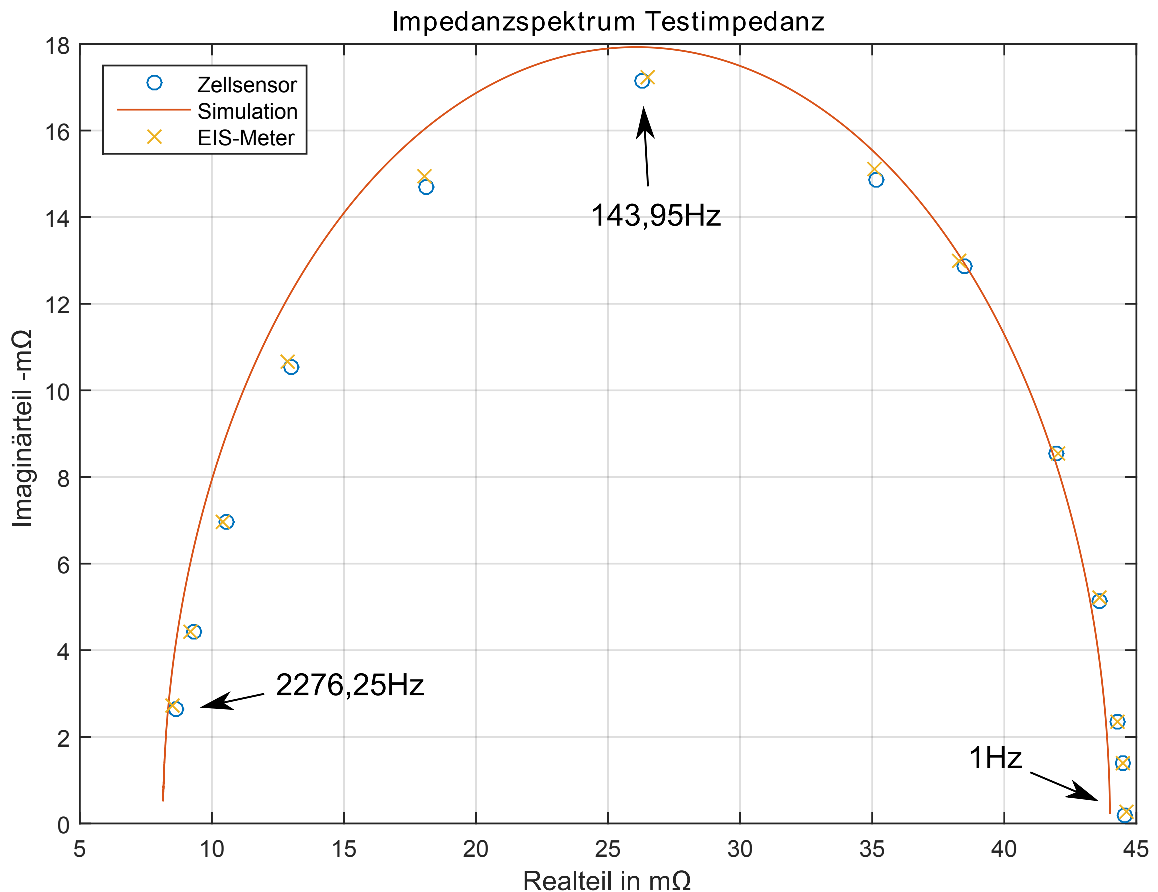


Abbildung 6.25.: Synchrones Impedanzspektrum der Testimpedanz mit einer Synchronisationszeit von 79,12μs

Es zeigt sich, dass die Impedanzmessung mit einer Wartezeit zwischen Taktaussendung und Strommessung von 79,12μs nun mit den Messwerten des FuelCon TrueData-EIS nahezu übereinstimmen. Der Imaginärteil der Messwerte ist nach der Erhöhung der Synchronisationszeit wie erwartet angestiegen.
Es kann also davon ausgegangen werden, dass das Messsystem nun synchrone Messungen zwischen Strom- und Spannungswerten liefert. Da die Messungen an der Testimpedanz erfolgreich sind, können nun erste Messungen an realen Batteriezellen durchgeführt werden.

6.3. Erprobung der Impedanzspektroskopie

Zur Erprobung der Impedanzspektroskopie an realen Batteriezellen wird wie bei der Messung der Testimpedanz zwischen den gemessenen Werten des FuelCon TrueData-EIS und den Messwerten des in dieser Arbeit realisierten Messsystems verglichen. Allerdings ist besonders im niedrigen Frequenzbereich mit Abweichungen zu rechnen, da dieser Bereich stark ladungsabhängig ist und von Messung zu Messung schwanken kann.
Vermessen werden $LiFePO_4$-Zellen von A123 (Typ: ANR26650M1-B). Diese besitzen einen typischen Innenwiderstand bei 1 kHz von 6mΩ [1]. Vermessen werden dabei die in der Tabelle 6.3 aufgeführten zwölf Messpunkte mit der jeweilig angegebenen Burstfrequenz. Angeregt wird die Batteriezelle dabei über das FuelCon TrueData-EIS mit einem Wechselstromanteil von 1,2 A.

Tabelle 6.3.: Frequenzpunkte der gemessenen Impedanzspektroskopie

Messpunkt	Signalfrequenz	Burstfrequenz
1	0,1 Hz	50,0 Hz
2	0,5 Hz	100,0 Hz
3	1,0 Hz	200,0 Hz
4	5,0 Hz	200,0 Hz
5	10,0 Hz	500,0 Hz
6	50,0 Hz	2.000,0 Hz
7	100,0 Hz	2.000,0 Hz
8	250,0 Hz	8.000,0 Hz
9	500,0 Hz	8.000,0 Hz
10	750,0 Hz	8.000,0 Hz
11	1.000,0 Hz	8.000,0 Hz
12	2.000,0 Hz	8.000,0 Hz

Die Abbildung 6.26 zeigt den bei den folgenden Messungen verwendeten Messaufbau für die Impedanzspektroskopie.

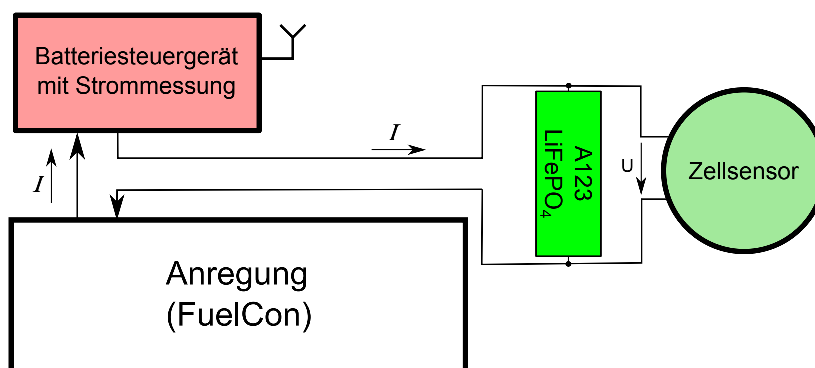


Abbildung 6.26.: Messaufbau für die Impedanzspektroskopie an einer $LiFePO_4$-Zelle

6.3.1. Impedanzmessung einer Batteriezelle

Das Ergebnis der Impedanzmessung einer realen Batteriezelle zeigt Abbildung 6.27.

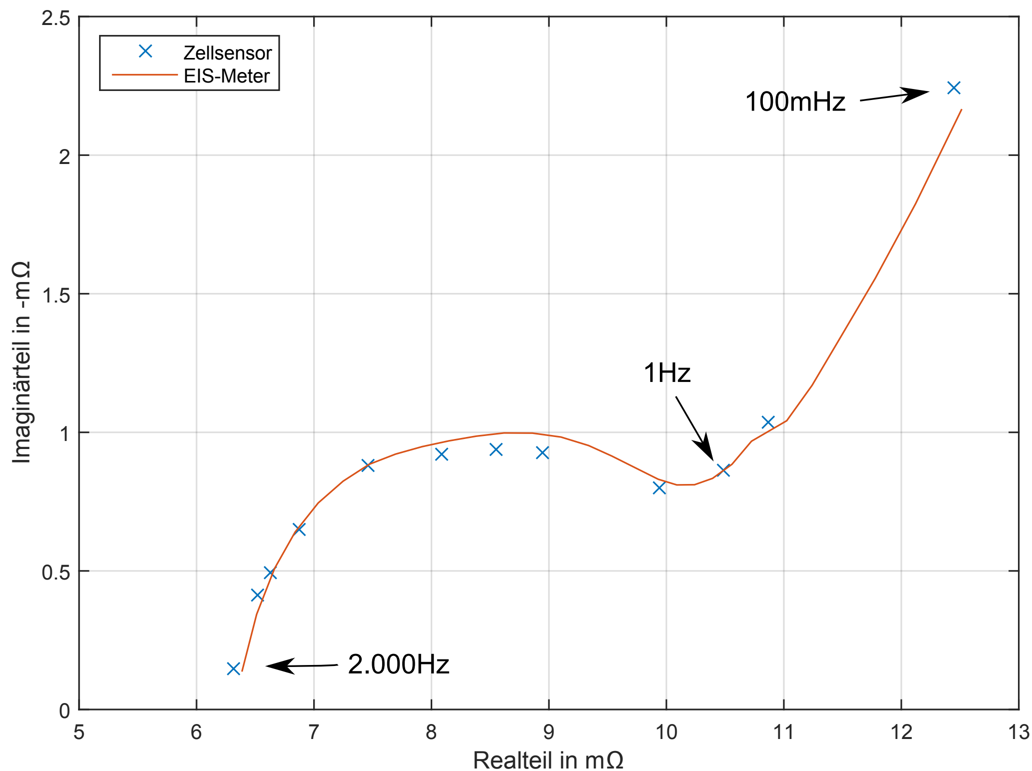


Abbildung 6.27.: Vergleich einer Impedanzmessung zwischen einem Labormessgerät und einer Messung mittels des Zellsensors. Dabei wurden Frequenzen zwischen 100 mHz und 2 kHz gewählt

Zu sehen ist, dass sich die Messwerte des FuelCon TrueData-EIS und die, durch das Messsystem aufgenommenen Messwerte sehr gut annähern. Es existieren lediglich geringe Abweichungen. Die größte Abweichung liegt bei dieser Messung im Messpunkt bei 100 mHz. Ob diese Abweichung durch eine Ladungsveränderung in der Batteriezelle oder durch einen Messfehler hervorgerufen wird, kann allerdings nicht festgestellt werden.
Das Auswertungsskript der ermittelten Messdaten dieser Messung ist in Anhang M zu finden.

Die Reproduzierbarkeit dieser Messung ist in Abbildung 6.28 zu sehen. Dabei handelt es sich um eine andere Batteriezelle des gleichen Typs als in der Messung zuvor. Es wurden wieder dieselben Frequenzpunkte aus der Tabelle 6.3 und dieselbe Anregung gewählt.

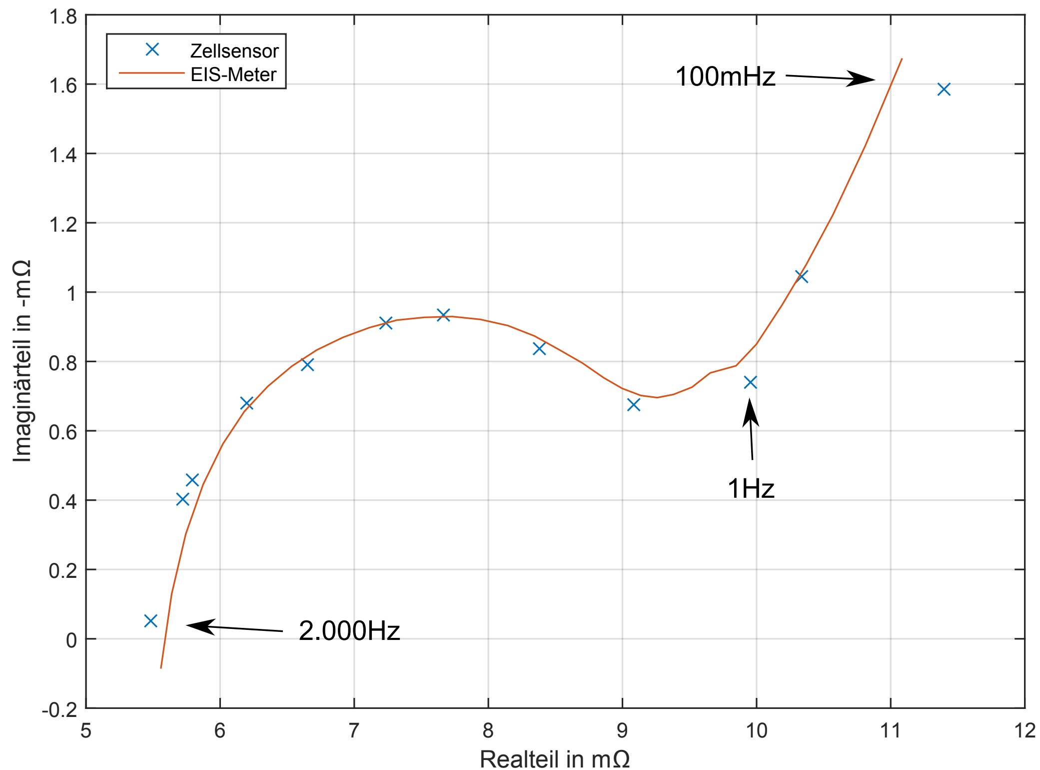


Abbildung 6.28.: Weiterer Vergleich einer Impedanzmessung zwischen einem Labormessgerät und einer Messung mittels des Zellsensors. Dabei wurden Frequenzen zwischen 100 mHz und 2 kHz gewählt

Auch diese Messung deckt sich wieder sehr gut mit der Messung des FuelCon TrueData-EIS. Eine Abweichung ist auch hier wieder im Frequenzpunkt 100 mHz zu sehen. Es bleibt unklar, ob es sich um eine Messungenauigkeit oder einen Effekt, bedingt durch einen unterschiedlichen Ladezustand der gemessenen Batteriezelle, handelt.
Eine weitere Ungenauigkeit gibt es in den höheren Frequenzpunkten oberhalb von 750 Hz. Dabei liegt der gemessene Imaginärteil der Messung etwas zu hoch. Dies ist ein Anzeichen, dass die Synchronisationszeit von 79,12 μs etwas zu groß ist.

Diese Messungen zeigen, dass das in dieser Arbeit entwickelte Messsystem in der Lage ist, eine funksynchronisierte elektrochemische Impedanzspektroskopie direkt an der Batteriezelle durchzuführen und dabei vergleichbare Messwerte wie mit einem Labor-Messgerät zu liefern.

6.3.2. Messungen an gealterten Batteriezellen

Um zu überprüfen, ob das entwickelte Messsystem auch in der Lage ist, eine Veränderung der Batteriezellen durch Alterung zu messen, wurde eine Batteriezelle in verschiedenen kalendarischen Alterungsstufen vermessen. Dazu ist eine $LiFePO_4$-Zelle (A123 Typ: ANR26650M1-B) über einen Zeitraum von ca. 5,5 Monaten bei Zimmertemperatur (ca. 20°C) gelagert und dreimal vermessen worden. Dabei zeigte sich eine Veränderungen des Impedanzspektrums. Die dabei gemessenen Spektren mit dem FuelCon TrueData-EIS zeigt Abbildung 6.29.

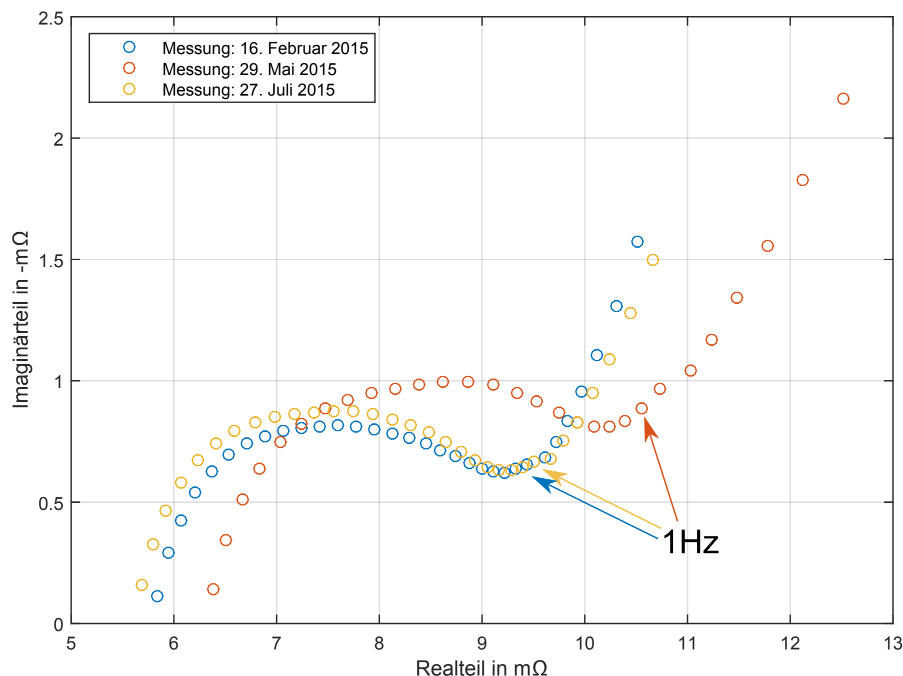


Abbildung 6.29.: Kalendarisch gemessene Alterung einer $LiFePO_4$-Zelle

Allerdings veränderte sich das Impedanzspektrum nicht wie bei der künstlichen Alterung (siehe Anhang D) mit einer Erhöhung des Realteils bei zunehmendem Alter. Vergleichspunkt ist hierbei der Messpunkt um 1 Hz. Die Initialmessung am 16. Februar 2015 und die Messung am 29. Mai 2015 verhalten sich dabei wie erwartet. Bei der Messung am 27. Juli 2015 ist der Realteil der Impedanzmessung allerdings wieder geringer. Das Impedanzspektrum blieb trotz mehrfacher Messungen gleich. Die Ursache, welches diesen Effekt auslöst, wurde nicht weiter untersucht, da diese vermutlich durch zellinterne chemische Reaktionen hervorgerufen wird. Wichtig hierbei ist, dass sich die beiden Impedanzspektren verändert haben und diese Veränderung gemessen werden kann.

Für die Alterungsmessung mit den Zellsensoren wurden die in der Tabelle 6.4 stehenden Messpunkte, gewählt. Es wurde auf den Messpunkt bei 100 mHz verzichtet, da dieser keine Aussagekraft mehr über die Alterung der Zelle liefert und um die benötigte Messzeit zu verkürzen.

Tabelle 6.4.: Frequenzpunkte der Alterungsmessung

Messpunkt	Signalfrequenz	Burstfrequenz
1	0,5 Hz	100,0 Hz
2	1,0 Hz	200,0 Hz
3	5,0 Hz	200,0 Hz
4	10,0 Hz	500,0 Hz
5	50,0 Hz	2.000,0 Hz
6	100,0 Hz	2.000,0 Hz
7	250,0 Hz	8.000,0 Hz
8	500,0 Hz	8.000,0 Hz
9	750,0 Hz	8.000,0 Hz
10	1.000,0 Hz	8.000,0 Hz
11	2.000,0 Hz	8.000,0 Hz

Bei der Initialmessung am 16. Februar 2015 waren die Zellsensoren noch in der Entwicklung, weshalb diese Messung nicht mit diesen Sensoren durchgeführt werden konnte. Die Messung von 29. Mai 2015 hingegen konnte bereits mit dem Messsystem erfasst werden. Ein Vergleich dieser Messungen mit den Messwerten des FuelCon TrueData-EIS zeigt Abbildung 6.30.

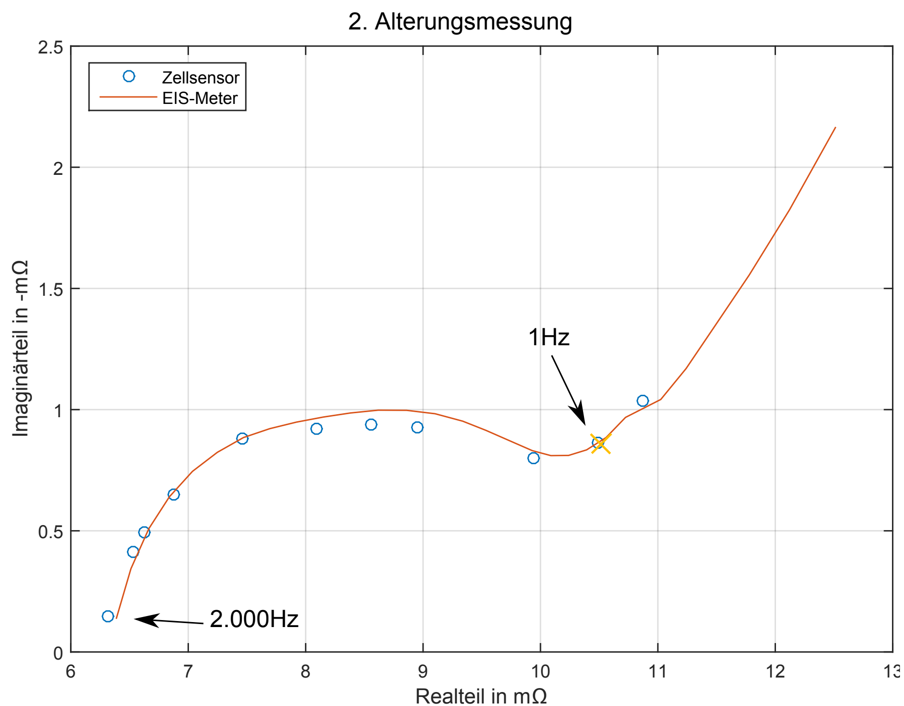


Abbildung 6.30.: 2. Alterungsmessung einer $LiFePO_4$-Zelle am 29. Mai 2015

Die Messung lässt erkennen, dass es nur minimale Abweichungen zwischen der Messung des FuelCon TrueData-EIS und der Messung der Zellsensoren gibt. Sehr gut stimmen hier die Frequenzpunkte beider Messungen um 1 Hz überein (gelbes Kreuz in Abbildung 6.30).

Die 3. Alterungsmessung konnte ebenfalls mit dem Messsystem durchgeführt werden. Auch hier stimmen die Messwerte wieder gut überein. Der Messpunkt um 1 Hz ist hier wieder mit einem gelben Kreuz in Abbildung 6.31 markiert. Auffällig bei dieser Messung ist, dass der Imaginärteil der Messpunkte etwas zu hoch liegt. Besonders der Messpunkt bei 2.000 Hz liegt ca. um 0,25mΩ im Real- und Imaginärteil vom Messwert des FuelCon TrueData-EIS verschoben.

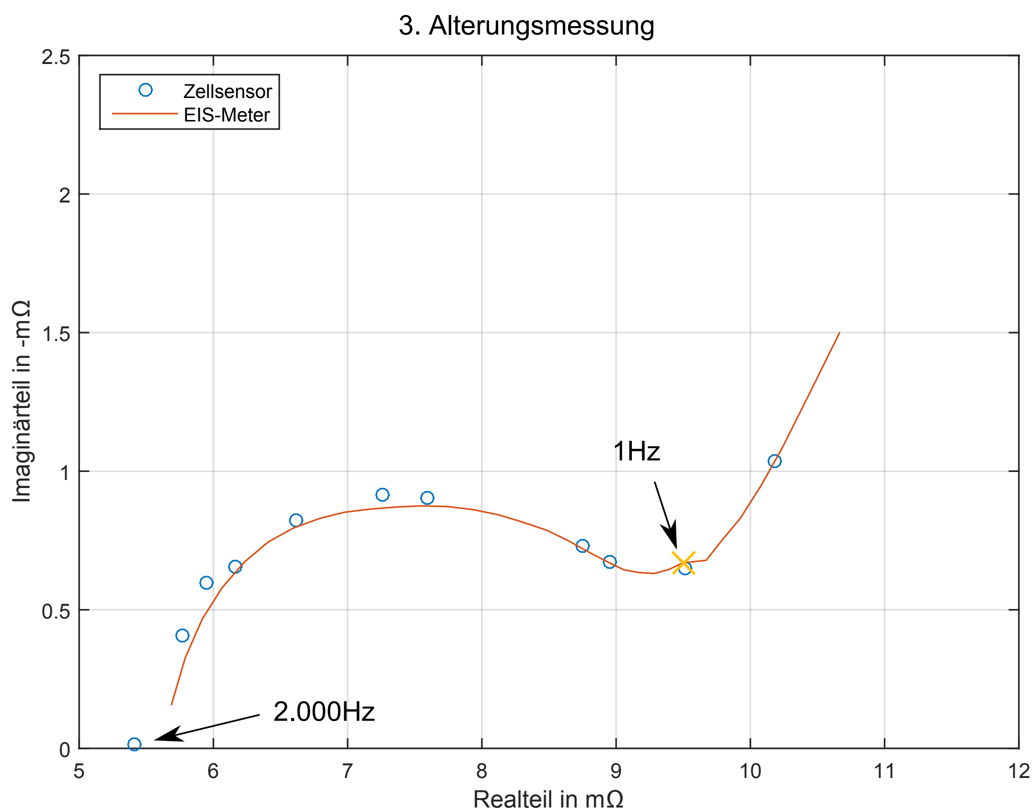


Abbildung 6.31.: 3. Alterungsmessung einer $LiFePO_4$-Zelle am 27. Juli 2015

In der Abbildung 6.32 sind die gemessenen Impedanzspektren nochmals gegenübergestellt. Der Verglich macht deutlich, dass die Alterung bzw. die Zustandsänderung der Zelle durch das Verfahren der funksynchronisierten Impedanzspektroskopie messbar ist. Es ergeben sich nur wenige Abweichungen zwischen den beiden Messungen. Dabei lässt sich aber nicht genau feststellen, ob es sich um Messfehler oder um eine Zustandsänderung, verursacht durch eine Ladungsänderung, handelt.

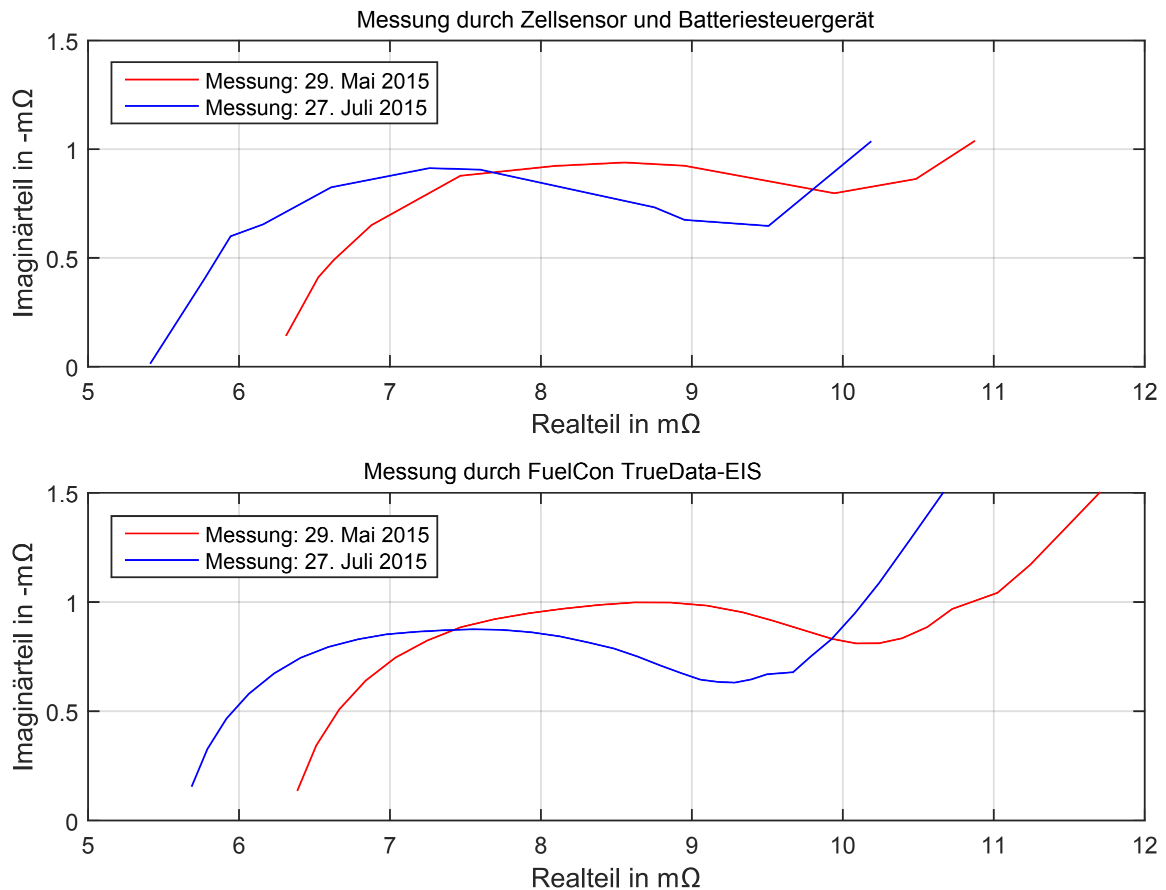


Abbildung 6.32.: Vergleich der gemessenen Impedanzspektren

6.3.3. Impedanzmessung bei verschiedenen Ladezuständen

Nun soll noch geprüft werden, ob das entwickelte Messsystem in der Lage ist, ein durch Ladungsveränderung unterschiedliches Impedanzspektrum zu messen. Dazu wurde die $LiFePO_4$-Zelle (A123 Typ: ANR26650M1-B) in unterschiedlichen Ladezuständen vermessen. Hierfür wurde zunächst das Impedanzspektrum in unterschiedlichen Ladezuständen durch das FuelCon TrueData-EIS gemessen, welche in Abbildung 6.35 zu sehen sind.

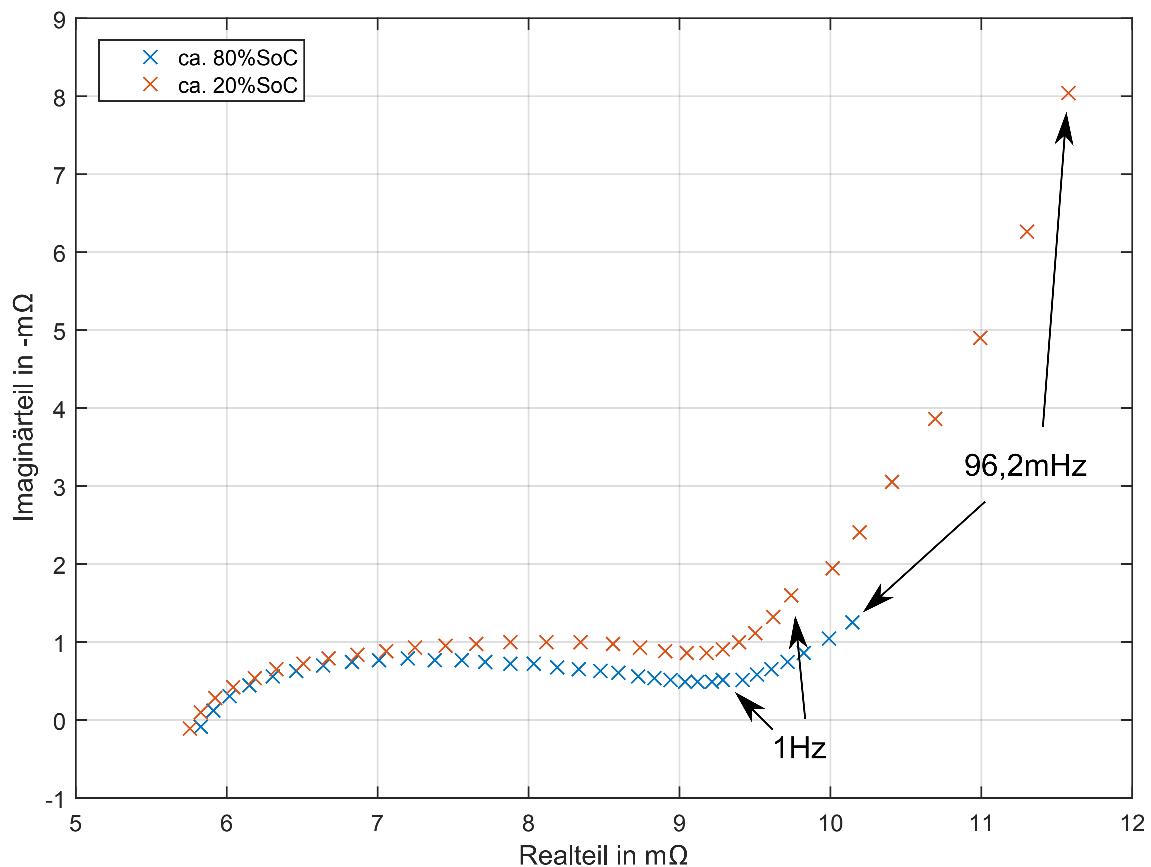


Abbildung 6.33.: Impedanzmessung bei verschiedenen Ladezuständen mit dem FuelCon TrueData-EIS

Die dargestellten Impedanzspektren zeigen die Batteriezelle bei ca. 80% und bei ca. 20% ihres Ladezustandes (mehr Ladezustände in Anhang E). Dabei ist, besonders bei den niedrigen Frequenz-Messpunkten < 1 Hz, eine Veränderung im Real- wie auch im Imaginärteil zu erkennen. Beide steigen, in Abhängigkeit der Anregefrequenz, stark an. Dieses Verhalten soll nun an einer anderen Batteriezelle gleichen Typs, mit den Zellsensoren und dem Batteriesteuergerät gemessen werden.

Diese Messungen mit dem eigenen Messsystem zeigt Abbildung 6.36. Dabei wurden wieder die Frequenz-Messpunkte zwischen 2 kHz und 100 mHz aus Tabelle 6.3 mit den dazugehörigen Burstfrequenzen gewählt. Für diese Messung wurde eine andere Batteriezelle gleichen Typs verwendet, weshalb kleine Abweichung im Realteil der Impedanz zu sehen sind. Es ist aber zu erkennen, dass das gemessene Impedanzspektrum dasselbe Verhalten zeigt, wie die Messung mit dem FuelCon TrueData-EIS. Auch hier steigt der Real- und Imaginärteil der Impedanz in Abhängigkeit des Ladezustandes wieder stark an.

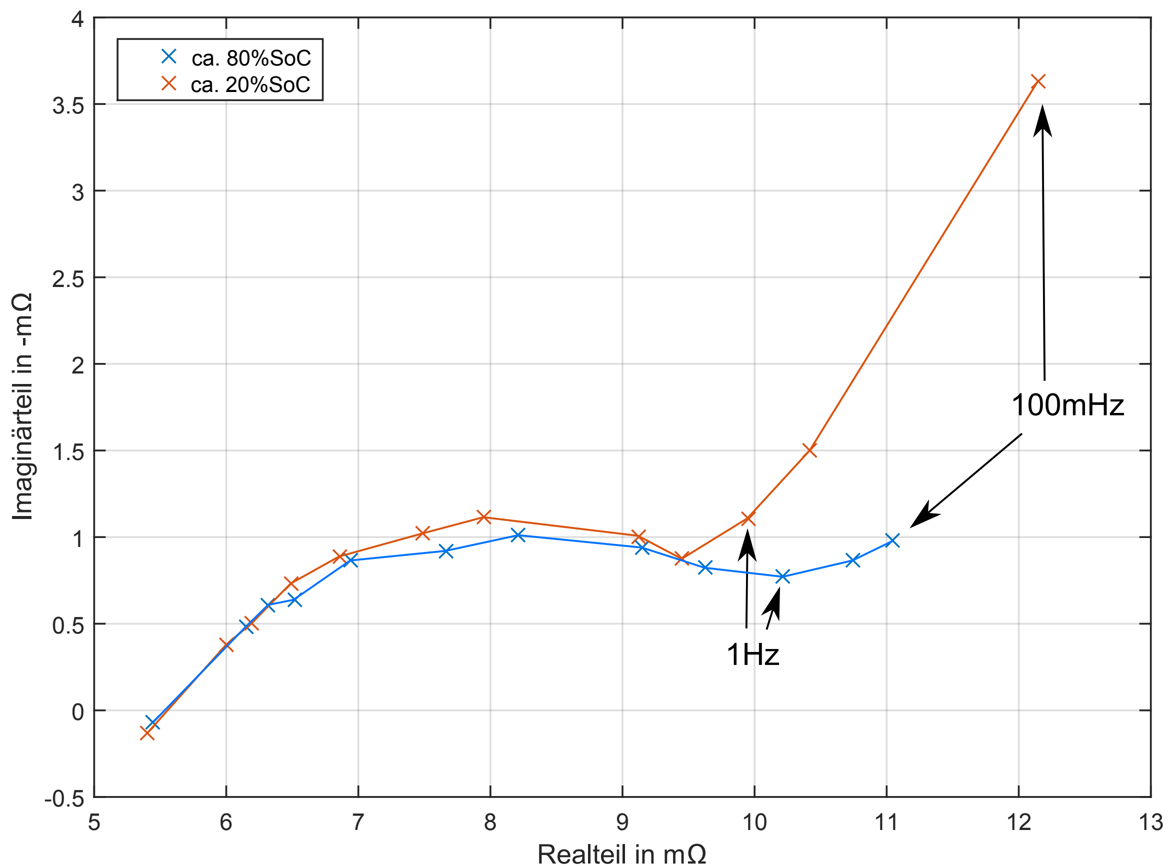


Abbildung 6.34.: Impedanzmessung bei verschiedenen Ladezuständen mit dem entwickelten Messsystem

Es zeigt sich, dass die entwickelte Hard- und Software in der Lage ist, Veränderungen der Zellchemie zu erfassen, die sich in Abhängigkeit des Ladezustands der Zelle verändern.

6.3.4. Impedanzmessung bei verschiedenen Temperaturen

Im Folgenden werden Messungen in unterschiedlichen Temperaturbereichen gezeigt. Verglichen werden hier wieder die Messungen mit dem FuelCon TrueData-EIS und den Messwerten aus dem, in dieser Arbeit entwickelten Messsystem. Es wurde darauf geachtet, dass bei den Messungen ungefähr der gleiche Ladezustand von ca. 80% SoC vorhanden war.

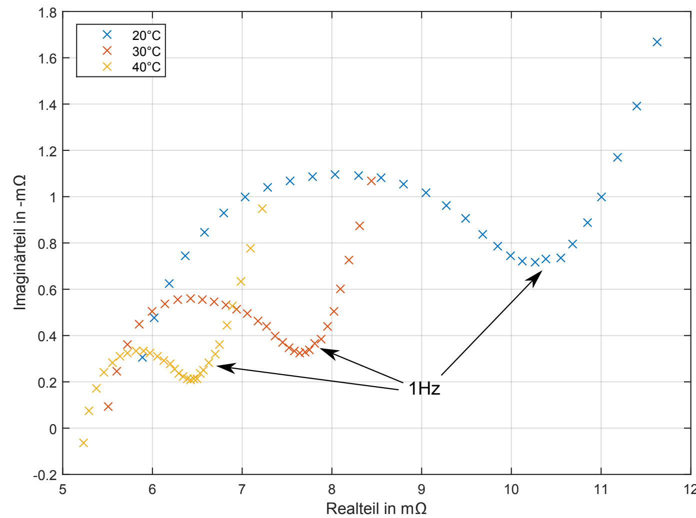


Abbildung 6.35.: Impedanzmessung bei verschiedenen Temperaturen: Gemessen mit FuelCon TrueData-EIS

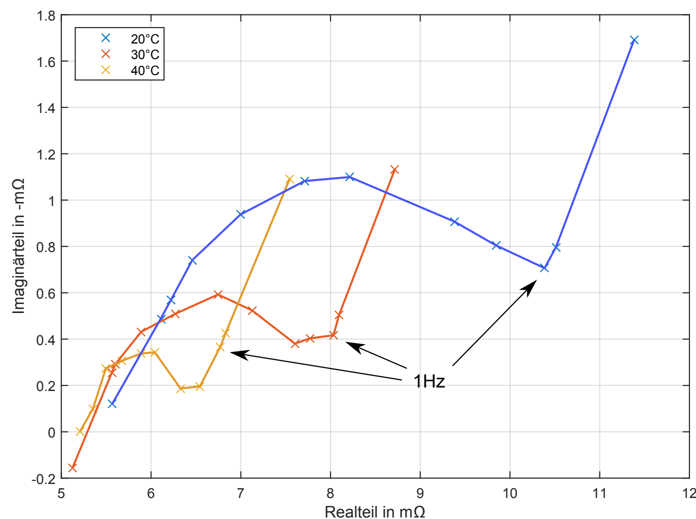


Abbildung 6.36.: Impedanzmessung bei verschiedenen Temperaturen: Gemessen mit dem entwickelten Messsystem

Die Messungen zeigen, dass auch Unterschiede der Zelltemperatur sehr gut erfasst werden können. Beide Messungen zeigen bei den unterschiedlichen Temperaturen das gleiche Verhalten. Besonders auffällig bei dieser Messung ist das "Wandern" des 1 Hz Frequenz-Messpunkts, weg vom Wendepunkt des Impedanzspektrums in Richtung des Diffusionsastes. Allerdings ist bei allen drei Messungen der Frequenz-Messpunkt vom 2.000 Hz leicht verschoben. Dies deutet wieder darauf hin, dass die gewählte Synchronisationszeit noch nicht optimal ist.

6.4. Auswertung und Bewertung der Messergebnisse

Die Ergebnisse der vorangegangenen Messungen bestätigen, dass es möglich ist, eine funksynchronisierte elektrochemische Impedanzspektroskopie mit den entwickelten Zellsensoren durchzuführen. Alle Messungen konnten mit dem FuelCon TrueData-EIS Labormessgerät bestätigt werden.

Dennoch kam es bei den Messungen besonders bei hohen Frequenz-Messpunkten zu leichten Abweichungen im Imaginärteil der Impedanz, welche meist über den Messwerten des FuelCon TrueData-EIS lagen. Dies deutet darauf hin, dass die eingestellte Synchronisationszeit von 79,12 μs zu hoch ist. Der vorhandene Synchronisationsfehler lässt sich mit den aufgenommenen Daten berechnen.
Dazu werden die Messdaten der Temperaturmessung bei 40 °C bei 2 kHz genommen. Dabei stimmt der Realteil nahezu überein. Der Imaginärteil hat eine Abweichung von 0,06 mΩ.

Tabelle 6.5.: Messdaten der Temperaturmessung bei 40 °C

	Messdaten bei 2 kHz
FuelCon TrueData-EIS	5,23 mΩ + j 0,06 mΩ
entwickeltes Messsystem	5,20 mΩ + j 0,00 mΩ

Die Messwerte haben eine Phasenverschiebung von 0,657 °. Mit diesem Wert lässt sich die zeitliche Abweichung Δ_t der beiden Messwerte zu 0,9125 μs bestimmen.

$$\Delta_t = \frac{\frac{1}{2\,\text{kHz}}}{360\,^\circ} \cdot 0{,}657\,^\circ = 0,9125\,\mu s \tag{6.9}$$

Die Abweichung befinden sich noch innerhalb der Toleranzzeit von 1,388 μs, welche in der Analyse im Abschnitt 3.3 ermittelt wurde. Da dieser Fehler in den meisten Messungen vorkommt, kann auch eine Anpassung der Synchronisationszeit vorgenommen werden. Diese Anpassung muss dann durch weitere Messungen überprüft werden.

Die Abweichung des Realteils bei hohen Frequenz-Messpunkten kann nicht von einem Synchronisationsfehler stammen, da sich ein Synchronisationsfehler kaum im Realteil des Messwertes bemerkbar macht. Dies lässt sich durch die Berechnung aus der Analyse im Abschnitt 3.3 bestätigen. Dieser Fehler entsteht durch einen Kalibrierungsfehler des AD-Wandlers.

Es zeigt sich mit diesen Messungen, dass Aussagen über den Zustand der Batteriezelle gemacht werden können. Zu sehen ist aber auch, dass die Messungen eine begrenzte Genauigkeit haben. So ergaben die Messungen, dass Vorgänge wie die Temperaturmessung, die eine große Änderung des Spektrums bewirken, sich sehr gut messen lassen. Ebenfalls konnten Ladungsänderungen an den Zellen gut erfasst werden. Dabei zeigte sich, dass der Betrag der Impedanz bei kleiner werdender Ladung und kleiner Anregefrequenz stark ansteigt. Dies könnte als Indikator für eine Ladungsdiagnostik der Batteriezelle dienen. Inwieweit sich diese Messungen mit den gleichen Ergebnissen wiederholen lassen, müssen aber weitere Tests zeigen.
Ebenso konnte die Alterung der Batteriezelle gemessen werden. Auch wenn diese nicht den theoretischen Vorstellungen entsprach, konnten die Messwerte des FuelCon TrueData-EIS durch das entwickelte Messsystem nachgewiesen werden. Dies zeigt, dass auch eine fortschreitende Alterung durch die funksynchronisierte Impedanzmessung messbar ist.

Die Anwendung der funksynchronisierten Impedanzspektroskopie mit dem jetzigen Stand der Hard- und Software ist möglich. Es muss aber darauf geachtet werden, dass es besonders im höheren Frequenzbereich zu Abweichungen im Real- und Imaginärteil kommen kann.

Mit diesem Stand können nun auch die eingangs erwähnten Forschungsfragen aus Abschnitt 1.2 beantwortet werden. Die Frage, ob die Messelektronik für die EIS-Messung soweit optimiert werden kann, dass es möglich ist, diese auf einem Zellsensor unterzubringen zu können, kann mit ja beantwortet werden. Durch die analoge Vorverarbeitung ist es gelungen, den internen 12 Bit AD-Wandler für die Messung der Wechselspannung an der Batteriezelle zu verwenden, ohne aufwendige Messelektronik einzusetzen.
Auch die Frage, ob es über die drahtlose Schnittstelle möglich ist, eine ausreichende Synchronisation der Strom- und Spannungswerte herzustellen, wurde untersucht und beantwortet. Innerhalb der festgelegten Toleranzen konnte erfolgreich eine Synchronisation der Strom- und Spannungswerte durchgeführt werden. Durch die ausreichende Synchronisierung gelang es dann auch vergleichbare Messungen zwischen dem FuelCon TrueData-EIS und dem in dieser Arbeit entwickelten Messsystem aufzunehmen.
Ebenfalls konnten Messungen bei unterschiedlichen Alterungs- und Ladezuständen sowie bei verschiedenen Temperaturen durchgeführt werden. Die Messungen folgten alle den erwarteten Impedanzkurven.

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung der Arbeit

In dieser Arbeit konnte erfolgreich ein Messsystem zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen entwickelt werden. Neben dieser Entwicklung wurden theoretische Grundlagen zur elektrochemischen Impedanzspektroskopie vorgestellt, welche einen Überblick über die in der Batteriezelle herrschenden chemischen Vorgänge geben, die für das Verständnis zur Impedanzspektroskopie notwendig sind. Besonders wurden Vorgänge beschrieben, die für eine Impedanzänderung der Zelle verantwortlich sind. Anhand dieser Impedanzänderungen wurde ein Batteriemodell erstellt, welches das theoretische Impedanzspektrum beschreibt.

Da es bereits einige Vorarbeiten zu dem Thema der funksynchronisierten Impedanzspektroskopie gab, wurde zunächst ein Überblick über die vorhandene Hard- und Software gegeben. Nach einer Analyse konnte festgestellt werden, dass die bisherigen Zellsensoren nicht in der Lage sind, eine vollständige funksynchronisierte Impedanzspektroskopie innerhalb des nötigen Frequenzbereichs durchzuführen. Aus diesem Grund wurde beschlossen, einen neuen Zellsensor zu entwickeln, mit dem der nötige Frequenzbereich für eine vollständige Impedanzspektroskopie erreicht werden kann.

Durch das, in den theoretischen Grundlagen vorgestellte Batteriemodell, konnten Kriterien für die Entwicklung eines neuen Messsystems erarbeitet werden. Daraufhin konnte eine Konzeptionierung der nötigen Hardware durchgeführt und berechnet werden. Es wurde beschlossen, dass bisherige Konzept der Messung mittels hochauflösendem AD-Wandler zu verlassen und ein Konzept mit analoger Vorverarbeitung des Messsignals zu verfolgen. Dadurch wurde es möglich, das Messsignal mit dem internen 12 Bit AD-Wandler des eingesetzten Mikrocontrollers CC430F5137 zu messen. Weiter wurde die Software des Zellsensors um wichtige Funktionen, die für die Optimierung der Datenübertragung und der Impedanzspektroskopie nötig sind, erweitert. So konnte erfolgreich eine Laufzeitmessung implementiert werden, mit deren Hilfe Abweichungen in der Synchronisation zwischen Batteriesteuergerät und den Zellsensoren detektiert werden können. Ebenso konnte eine verteilte Signalverarbeitung auf den Zellsensoren implementiert werden, wodurch das anfallende Datenvolumen innerhalb des Funkkanals erheblich reduziert werden konnte.

Neben der Neuentwicklung eines Batteriesteuergerätes mit drahtloser Kommunikationsschnittstelle und einer Möglichkeit zur Messung des Stroms, wurde der Zellsensor um die analoge Vorverarbeitung erweitert und ein neuer Zellsensor gefertigt. Dabei konnten die Funkeigenschaften des Sensors durch eine Simulation mit dem 3D FEM-Simulationsprogramm CST Microwave Studio untersucht und optimiert werden. Ergebnis dieser Simulation war eine wesentlich bessere Abstrahleigenschaft des Zellsensors.

Bei der Inbetriebnahme wurden alle neu entwickelten Funktionen überprüft. Dabei konnten keine Fehlfunktionen festgestellt werden. Durch die Vermessung einer Testimpedanz zeigte sich, dass sich die aus der Analyse ergebene theoretische Synchronisationszeit, die sich zwischen der Strom- und Spannungsmessung befindet, höher ist als ursprünglich angenommen. Eine Analyse des gesamtem Messsystems ergab, dass zwischen Strom- und Spannungserfassung eine Phasenverschiebung herrscht. Nachdem diese ausgeglichen war, konnten mit dem entwickelten System erste Messergebnisse erzielt werden, die mit denen eines Labormessgerätes vergleichbar sind.

Nach den erfolgreichen Messungen an einer Testimpedanz wurden erste Messreihen an einer $LiFePO_4$-Zelle durchgeführt. Dabei wurden Messungen an unterschiedlichen Alterungszuständen, verschiedenen Ladezuständen sowie bei unterschiedlichen Temperaturen durchgeführt. Die Messergebnisse der einzelnen Messreihen stimmten gut mit den Vergleichsmessungen des Labormessgeräts überein.

7.2. Ausblick

Die ersten Messungen mit der entwickelten funksynchronisierten Impedanzspektroskopie zeigten gute Ergebnisse. Der Zellsensor ist in Kombination mit dem Batteriesteuergerät nun auch in der Lage, eine elektrochemische Impedanzspektroskopie an $LiFePO_4$-Zellen durchzuführen. Dabei konnten alle Punkte der geforderten Aufgabenstellung (Anhang I) bearbeitet werden. Das entwickelte Messsystem kann nun weiter intensiv erprobt und getestet werden. Interessant wäre eine Erprobung über einen längeren Zeitraum an alternden Zellen, welche in dieser Arbeit wegen der benötigten Zeit für die Alterung nur bedingt durchgeführt werden konnte. Ebenso ist eine Erprobung an unterschiedlichen Batterietypen durchzuführen. Die in dieser Arbeit gezeigten Messungen wurden alle an einer A123 (Typ: ANR26650M1-B) $LiFePO_4$-Zelle durchgeführt.
Untersucht werden muss noch die Veränderung der Synchronisationszeit bei unterschiedlichen Verstärkungsstufen der analogen Vorverarbeitung. Die aufgenommene Messung in Abbildung 6.22 zeigt den Amplituden- und Phasengang bei einer Verstärkung von $V = 10{,}765$. Vermutet wird, dass es bei unterschiedlichen Verstärkungsstufen zu unterschiedlichen Phasengängen kommt. Dies verändert wiederum die nötige Synchronisationszeit, die zwischen Taktaussendung und Strommessung liegen muss. Werden unterschiedliche Phasengänge

festgestellt, so kann die nötige Synchronisationszeit als Wert im Batteriesteuergerät hinterlegt und entsprechend der Verstärkungsstufe gewählt werden.
Es bleiben aber noch mehr offene Punkte, die für weiterführende Arbeiten geeignet sind.

Spannungsversorgung
Die eigenständige Spannungsversorgung des Zellsensors stellt momentan das Hauptproblem dar um einen unabhängigen Betrieb von externen Spannungsquellen zu gewährleisten. In Anhang B wird ein möglicher Lösungsansatz diskutiert, womit die 3,3 V Versorgungsspannung während der Abschaltung des TPS61201 stabilisiert werden kann. Dieser Lösungsansatz muss aber aufgrund der dadurch reduzierten maximalen Abtastrate von 2 kHz nochmals überarbeitet werden, sodass auch höhere Abtastraten möglich werden.
Generell sollte über eine Überarbeitung der Zellspannungsversorgung nachgedacht werden. So würde eine Anhebung der 3,3 V Versorgungsspannung mehrere existierende Probleme lösen. Es könnte zum einen der Gleichspannungsabzug an der Subtrahierschaltung erhöht werden. Dadurch ließe sich das zu messende Spannungssignal einer voll geladenen $LiFePO_4$-Zelle deutlich mehr verstärken als das bisher möglich ist. Bisher können max. 3,26 V abgezogen werden, wodurch die Verstärkung begrenzt wird.
Zum anderen könnte durch eine Spannungsanhebung eine separate Versorgungsspannung über einen LDO für den Analogteil des Zellsensors aufgebaut werden. Um die Energiebilanz des Zellsensors nicht unnötig zu erhöhen, könnte dieser LDO vom Mikrocontroller gesteuert werden und nur bei Bedarf die Spannung für den Analogteil erzeugen.
Dazu könnte der TPS61200 von Texas Instruments genutzt werden. Dieser ist baugleich zum momentan eingesetzten TPS61201, der eine feste Ausgangsspannung von 3,3 V besitzt. Am TPS61200 lässt sich die Ausgangsspannung zwischen 1,8 V bis 5,5 V einstellen.

Kalibrierung und Untersuchung des Phasen- und Amplitudengangs
Für genauere Messergebnisse muss eine Spannungskalibrierung des AD-Wandlers auf dem Zellsensoren sowie auf dem Batteriesteuergerät durchgeführt werden. Zudem sollte der Amplitudengang der analogen Vorverarbeitung bei unterschiedlichen Verstärkungsstufen untersucht werden. Ebenso wie der Phasengang wird sich der Amplitudengang bei unterschiedlichen Verstärkungsstufen verändern.

Automatische Einstellung der analogen Vorverarbeitung
Softwaretechnisch sollte die Steuerung der automatischen Einstellung der analogen Vorverarbeitung optimiert werden. Momentan werden die zu erreichenden Grenzspannungen, durch die Einstellung der Rheostaten und anschließender Messung der Spannung schrittweise angenähert. Dies kann bei dem derzeit verwendeten Algorithmus, wenn alle Schrittstufen des Rheostaten durchlaufen werden müssen, bis zu 40,96 s dauern. Durch einen besseren Algorithmus der Annäherung könnte diese Zeit deutlich verkürzt werden.
So könnte z.B. das Wägeverfahren angewandt werden, um die Grenzspannungen schneller zu erreichen und die Einstellzeit zu verkürzen.

Synchronisation zwischen den Zellsensoren
Des Weiteren muss noch eine Betrachtung der Synchronität zwischen den Sensoren gemacht werden. Erste Messungen ergaben, dass der Unterschied der einzelnen Zellsensoren im Durchschnitt 206,9 ns beträgt [70]. Diese Messung wurde aber mit den Zellsensoren aus der Arbeit [48] durchgeführt und sollte mit den neuen Zellsensoren wiederholt werden. Da die selbe Hardware verwendet wurde, ist aber mit keiner größeren Abweichung zu den gemessenen Werten zu rechnen.

Erkennung einer Messung im nichtlinearen Bereich
Das Problem, dass mit der Messung einer Nichtlinearität einhergeht, wurde innerhalb dieser Arbeit bereits erarbeitet. Allerdings wurde nicht untersucht, wie sich solche fehlerhaften Messungen detektieren lassen. Eine Lösung für die Detektion einer solchen nichtlinearen Messung ist die Beobachtung der höheren Harmonischen der Anregefrequenz. Durch die Literatur ist bekannt, dass bei Messungen an Nichtlinearitäten neben der Grundschwingung der Anregefrequenz auch Harmonische Vielfache dieser Schwingung im Frequenzspektrum auftauchen. So lässt sich die Linearität der Impedanzmessung durch das Beobachten höherer Harmonischer der Anregungsfrequenz detektieren und bewerten [42]. Diese Möglichkeit könnte leicht in das Messsystem implementiert werden, da die nötige Signalverarbeitung bereits durch den Goertzel-Algorithmus auf dem Zellsensor und dem Batteriesteuergerät vorhanden ist.

Konzeption einer Anregung
Als weiteren Arbeitspunkt kann die Konzeption einer eigenen Anregung gesehen werden. Diese kann idealerweise durch das Batteriesteuergerät gesteuert werden. Innerhalb dieser Arbeit wurde als Anregequelle meist das FuelCon TrueData-EIS genutzt. Um das Messsystem unabhängig von diesem Labormessgerät betreiben zu können, ist aber eine eigene Anregung nötig.

Optimierung der verteilten Signalverarbeitung
Die verteilte Signalverarbeitung lässt sich dahingehend optimieren, dass die Berechnung des rekursiven Filterteils des Goertzel-Algorithmus zwischen den einzelnen Spannungsmessungen der Burstmessung auf dem Zellsensor stattfinden kann. Dadurch wäre nur noch ein minimaler Speicher für die Filter-Verzögerungsstufen notwendig, der bei jeder neuen Messung überschrieben werden können. Der bisherige verwendete Speicher für die 2.500 Spannungswerte würde dadurch nicht mehr benötigt werden, da die Messwerte direkt nach ihrer Aufnahme in der Filterstruktur verarbeitet werden könnten und weiter nicht mehr benötigt werden. Voraussetzung dafür ist eine schnelle Verarbeitung der Messdaten. So wäre die für die Verarbeitung zur Verfügung stehende Zeit abhängig von der verwendeten Burstrate, abzüglich der Zeit, die für die Spannungsaufnahme nötig ist.
Ob die Verarbeitung des Mikrocontrollers schnell genug ist, um auch bei höheren Burstfrequenzen schnell genug die Berechnung durchführen zu können, müssen Messungen in weiterführenden Arbeiten zeigen.

Tabellenverzeichnis

2.1. Übersicht der Kathodenmaterialien (nach [68]) 15
2.2. Übersicht der Bauteiländerungen bei Alterung 45

3.1. Übersicht kommerzieller EIS-Meter 54
3.2. Übersicht Innenwiderstand 56
3.3. Bewertungsmatrix AD-Wandler Auswahl 70
3.4. Übersicht verschiedener Digital-Analog-Wandler 71
3.5. Vergleich der beiden vorgestellten Messkonzepte 78
3.6. Aufwandsvergleich der DFT-Berechnung 87
3.7. Mindestanforderung der Impedanzspektroskopie 88

4.1. Tabelle der eingesetzte IC-Bauelemente 101

5.1. Funktionsübersicht Zellsensor v0.4 107
5.2. Vergleich der entwickelten Goertzel-Filterstruktur 123

6.1. Vergleich der Ergebnisse der Laufzeitmessung 136
6.2. Frequenz-Messpunkte der Testimpedanz-Messung 140
6.3. Frequenzpunkte der gemessenen Impedanzspektroskopie 147
6.4. Frequenzpunkte der Alterungsmessung 151
6.5. Messdaten der Temperaturmessung bei 40 °C 157

F.1. Befehlsübersicht für das Batteriesteuergerät 206

Abbildungsverzeichnis

1.1. Prinzip der Zellüberwachung und des Batteriemanagements (ent. aus [62]) . . . 9

2.1. Potenzialbereich verschiedener Aktivmaterialien (nach [72]) 13
2.2. Prinzipieller Aufbau einer $LiFePO_4$ Zelle (nach [72] [61]) 14
2.3. Überspannungen an einer Zelle im Ladefall (nach [68]) 17
2.4. Ohmsche Überspannung am Batteriemodell 18
2.5. Durchtrittsüberspannung am Batteriemodell 19
2.6. Im linken Bild ist die Butler-Volmer-Annäherung mit i_0 = 3 A, α = 0.5, T = 293 K und n = 2 zu sehen. Im rechten Bild ist der Durchtrittswiderstand in Abhängigkeit der Überspannung η_{ct} zu sehen (nach [42]). 20
2.7. Doppelschichtmodell mit Potenzialverlauf (nach [24]) 21
2.8. Ohmsche Überspannung: Doppelschichtkapazität 22
2.9. Im linken Bild ist der Betrag der Impedanz Z logarithmisch über der Frequenz dargestellt. Im rechten Bild ist der Imaginärteil in Abhängigkeit des Realteils aufgetragen 23
2.10. Vollständiges Batteriemodell der drei Überspannungen 23
2.11. Prinzip SEI-Wachstum (nach [61]) 24
2.12. SEI Wachstum am gealterten $LiFePo_4$ Material (ent. aus [26] [45]) 25
2.13. Aufbrechen der Graphit-Struktur (nach [26]) 26
2.14. Tatsächliche Kapazität einer Zelle (nach [68]) 26
2.15. Angeregtes System 28
2.16. Typische Phasenverschiebung zwischen Anregung und Antwort 29
2.17. Zweipol 30
2.18. Strom-Spannungs-Zeitsignale einer komplexen Impedanz 31
2.19. Strom-Spannungs-Zeigerdiagramm 32
2.20. Bode-Diagramm einer Impedanzspektroskopie einer $LiFePO_4$-Zelle 33
2.21. Nyquist-Diagramm einer Impedanzspektroskopie 34
2.22. Nyquist-Diagramm (nach [63]) 35
2.23. Verschiedene Batteriemodelle (nach [16]) 38
2.24. Einfaches Batteriemodell nach Randles [58] 39
2.25. Nyquist-Diagramm der Elektrodenimpedanz (nach [24]) 40
2.26. Erste Abschätzung der Parameter für das Batteriemodell aus einem beispielhafen Impedanzspektrum 41
2.27. Erweitertes Batteriemodell nach Randles [58] 42

2.28. Nyquist-Diagramm des erweiterten Randles Batteriemodells 43
2.29. Links: Batteriemodell mit zwei RC-Gliedern; Rechts: Batteriemodell mit vier RC-Gliedern (ent. aus [16]) 43
2.30. Vergleich zwischen einem gemessenen Impedanzspektrum und dem vorgestellten optimiertes Batteriemodel 44
2.31. Alterung am Batteriemodell 46
2.32. Künstliche Alterung einer $LiFePO_4$ Zelle bei zwei verschiedenen Alterungszuständen 47

3.1. Batteriesteuergerät mit CC1101-Transceiver Platine und Zellsensor der Klasse 3 mit hochauflösender AD-Wandler Erweiterungsplatine 48
3.2. AD-Wandler Erweiterungsmodul (ent. aus [48]) 50
3.3. Blockschaltbild des Zellsensors v0.4 und deren Verbindungen der Sensorklasse 3 (nach [48]) 51
3.4. Batteriesteuergerät für Zellsensoren der Klasse 3 auf Basis eines Evaluations-Board (Olimex MSP430-169STK) 53
3.5. Zellalterung - Einzelne Frequenzen 55
3.6. Synchronisationsfehler zwischen Strom und Spannung um 125 ms 58
3.7. Erste Näherung des Synchronisationsfehlers bei 2 kHz 60
3.8. Synchronisationsfehler in Abhängigkeit der Frequenz bei unterschiedlichen Synchronisationszeiten 61
3.9. Dynamische Laufzeitermittlung zwischen dem Batteriesteuergerät (BS) und dem Zellsensor (ZS) 62
3.10. Kleinsignalverhalten ohne Offset: $R_{Kleinsignal}$ = 59,6 mΩ, $R_{Großsignal}$ = 25,6 mΩ: Gleichstromanteil: 0,0 A, Kleinsignalamplitude: 0,5 A, Großsignalamplitude: 5,0 A 64
3.11. Kleinsignalverhalten mit Offset: $R_{Kleinsignal}$ = 8,3 mΩ, $R_{Großsignal}$ = 11,5 mΩ: Gleichstromanteil: 3,0 A, Kleinsignalamplitude: 0,5 A, Großsignalamplitude: 5 A 65
3.12. Vergleich der abweichenden Impedanzspektren mit und ohne Gleichstromanteil 65
3.13. Blockschaltbild eines SAR-AD-Wandlers (nach [67]) 68
3.14. Schematische Darstellung der Rauschformung (nach [67]) 69
3.15. Blockschaltbild eines Delta-Sigma-AD-Wandlers (nach [67]) 69
3.16. Spannungs-Subtrahierschaltung 73
3.17. Kapazität der DC-Filterung 74
3.18. Spannungs-Subtrahierschaltung 75
3.19. Dynamisch einstellbare Subtrahierschaltung 76
3.20. Dynamisch einstellbare Subtrahierschaltung mit einstellbarer Verstärkung . 77
3.21. Signalflussdiagramm einer Radix-2-FFT mit der Länge N = 8 83
3.22. Filterstruktur des Goertzel-Algorithmus 1. Ordnung 85
3.23. Filterstruktur des Goertzel-Algorithmus (nach [64]) 86

3.24. Vergleich der Berechnungszeiten der Standard DFT, Radix-2-DFT und des Goertzel-Algorithmus 87

4.1. Tiva C Series TM4C1294 Connected LaunchPad Evaluation Board (ent. aus[36]) 91
4.2. Layout des CC1101 Transceiver-Erweiterungsmoduls Links: Oberseite des Layouts. Rechts: Unterseite des Layouts 92
4.3. Layout des ACS716 Strommess-Erweiterungsmoduls Links: Oberseite des Layouts. Rechts: Unterseite des Layouts 93
4.4. Blockschaltbild des Batteriesteuergeräts 94
4.5. Aufteilung der Zellspannung in zwei verschiedene Messpfade 96
4.6. Auflösungsstufen des Rheostaten 97
4.7. Dynamisch einstellbare Subtrahierschaltung mit einem zusätzlichen Widerstand nach dem Rheostaten 97
4.8. Nichtinvertierender Messverstärker 98
4.9. Analoge Vorverarbeitung: R_1, R_2, R_7 = 1 kΩ und R_{Rheo1},R_{Rheo2},R_3-R_6 = 100 kΩ 99
4.10. Geändertes Anpassungsnetzwerk des CC430F5137 99
4.11. Links: externe Interruptauslösung über Ein- und Ausgangeports am Mikrocontroller; Rechts: interne Interruptauslösung durch gezielte Adressierung der Interrupt Service Routine 100
4.12. Blockschaltbild des neu entwickelten Zellsensors 102
4.13. Zu überarbeitende Kontaktfläche auf dem Zellsensor: Gegenüberstellung der vorhandenen Kontaktfläche zu der benötigten Kontaktfläche 103
4.14. Layout des neuen Zellsensors: Links: Layout der Oberseite. Rechts: Layout der Rückseite. Layout in höherer Auflösung in Anhang K. 105
4.15. Layout der hochauflösenden AD-Wandler-Erweiterungsplatine Links: Layout der Oberseite. Rechts: Layout der Rückseite 106

5.1. Speichermanagement bei 12 Bit Messwerten, Links: Normales Speichermanagement. Rechts: Optimiertes Speichermanagement 108
5.2. Zustandsablauf und Abhängigkeiten der dynamischen Laufzeitermittlung . 109
5.3. Sendeprinzip der DCO-Kalibrierung (nach [52]) 110
5.4. Pulsmessung des Batteriesteuergerätes 110
5.5. Softwareablauf der DCO-Kalibrierung am Zellsensor 111
5.6. Softwareschritte zur Einstellung mittels Gleichspannungsanteil 112
5.7. Erste Stufe der analogen Vorverarbeitung: Gleichspannung abziehen 113
5.8. Zweite Stufe der analogen Vorverarbeitung: Messwert verstärken 114
5.9. Möglicher Fehler der Messwert-Verstärkung. Links: Abtastung eines 50 Hz Messsignals. Rechts: Worst-Case Abtastung eines 25 Hz Messsignals. . . . 114
5.10. Konzeptdarstellung der verschiedenen Stufen mit Grenzspannungen der Einstellung unter Last 115

5.11. Grundautomat des Batteriesteuergeräts 116
5.12. Prioritätenverteilung am Batteriesteuergerät 117
5.13. Unterautomat der Burstmessung zur Impedanzspektroskopie 119
5.14. Berechnung des rekursiven Filterteils 121
5.15. Berechnung des nicht-rekursiven Filterteils 122
5.16. Für die Mikrocontroller-Implementierung optimierte Filterstruktur des Berechnungsschritts N+1 (nach [64]) 122

6.1. Neu entwickeltes Messsystem mit Batteriesteuergerät und Zellsensor 124
6.2. Batteriesterngerät mit aufgesteckten Erweiterungsmodulen 126
6.3. Bestücktes CC1101 Transceiver-Erweiterungsmodul 126
6.4. Bestücktes Strommess-Erweiterungsmodul 127
6.5. Prinzipieller Messaufbau zum Test des Strommess-Erweiterungsmoduls . . 127
6.6. Erzeugte Spannung durch das Strommess-Erweiterungsmodul 128
6.7. Oben: Vorderseite des bestückten Zellsensors mit Beschriftung einzelner Funktionsteile und Bauteile. Unten: Rückseite des Zellsensors mit den beiden JTAG-Interfaces 129
6.8. Anpassungsnetzwerk der Schleifenantenne 130
6.9. Simulierte S_{11}-Parameter nach Anpassung der Schleifenantenne 130
6.10. Gemessene Abstrahlleistung der Schleifenantenne 131
6.11. Einfluss der Abschaltung des TPS61201 auf die analoge Vorverarbeitung . . 133
6.12. Steuerung der analogen Vorverarbeitung 134
6.13. Ermittelte Laufzeiten der dynamischen Laufzeitermittlung mit 1.500 Messwerten: Mittelwert = 15,86 μs, σ = 0,2832 μs 135
6.14. Ermittelte Laufzeiten mittels Oszilloskop mit 1.500 Messwerten: Mittelwert = 16,16μs, σ=0,2274μs 136
6.15. Messaufbau für die phasenrichtige Strom- und Spannungsaufnahme des Messsystems 137
6.16. Überprüfung der phasenrichtigen Spannungsaufnahme des Zellsensors und des Batteriesteuergeräts 137
6.17. Entwickelte Testimpedanz 138
6.18. Schaltplan der entwickelten Testimpedanz 139
6.19. Prinzipieller Messaufbau zur Vermessung der Testimpedanz 140
6.20. Erste Impedanzmessung an der Testimpedanz mit einer Synchronisationszeit von 72,2μs 141
6.21. Zeiten für die synchrone Strom- und Spannungsmessung 143
6.22. Gemessener Amplituden- und Phasengang der analogen Vorverarbeitung . . 144
6.23. Veranschaulichung der zeitlichen Verschiebung zwischen Messspannung und Strom 145
6.24. Zusammensetzung der Zeiten für eine synchrone Strom- und Spannungsmessung 145

6.25. Synchrones Impedanzspektrum der Testimpedanz mit einer Synchronisationszeit von 79,12μs 146
6.26. Messaufbau für die Impedanzspektroskopie an einer $LiFePO_4$-Zelle 147
6.27. Vergleich einer Impedanzmessung zwischen einem Labormessgerät und einer Messung mittels des Zellsensors. Dabei wurden Frequenzen zwischen 100 mHz und 2 kHz gewählt 148
6.28. Weiterer Vergleich einer Impedanzmessung zwischen einem Labormessgerät und einer Messung mittels des Zellsensors. Dabei wurden Frequenzen zwischen 100 mHz und 2 kHz gewählt 149
6.29. Kalendarisch gemessene Alterung einer $LiFePO_4$-Zelle 150
6.30. 2. Alterungsmessung einer $LiFePO_4$-Zelle am 29. Mai 2015 151
6.31. 3. Alterungsmessung einer $LiFePO_4$-Zelle am 27. Juli 2015 152
6.32. Vergleich der gemessenen Impedanzspektren 153
6.33. Impedanzmessung bei verschiedenen Ladezuständen mit dem FuelCon TrueData-EIS 154
6.34. Impedanzmessung bei verschiedenen Ladezuständen mit dem entwickelten Messsystem 155
6.35. Impedanzmessung bei verschiedenen Temperaturen: Gemessen mit FuelCon TrueData-EIS 156
6.36. Impedanzmessung bei verschiedenen Temperaturen: Gemessen mit dem entwickelten Messsystem 156

A.1. Montierte AD-Wandler Erweiterungsplatine auf einem Zellsensor 178
A.2. Messung mit AD7691 AD-Wandler 179
A.3. Weitere Messung mit AD7691 AD-Wandler 180

B.1. Spannungsversorgungs-Erweiterungsmodul 181
B.2. Zellsensor mit montiertem Spannungsversorgungs-Erweiterungsmodul 182
B.3. Abschaltung des TPS61202 182

C.1. Platinen-Struktur in CST Microwave Studio 185
C.2. Platinen-Struktur mit Oberflächenströmen bei 434 MHz 186
C.3. 2D-Abstrahlcharakteristik bei 434 MHz 186
C.4. Platinen-Struktur mit Oberflächenströmen bei 509,5 MHz 187
C.5. 2D-Abstrahlcharakteristik bei 509,5 MHz 187
C.6. Platinen-Struktur mit eingebrachten Induktivitäten 188
C.7. Überarbeitete Platinen-Struktur mit Oberflächenströmen bei 434 MHz 189
C.8. Korrigiertes 2D-Abstrahlcharakteristik bei 434 MHz 189
C.9. 3D-Abstrahlcharakteristik bei 434 MHz mit eingesetzten Induktivitäten 190
C.10. Richtdiagramm der Schleifenantenne mit eingesetzten Induktivitäten in der Theta-Ebene 190
C.11. Streuparameter der Schleifenantenne / CST Microwave Studio 191

C.12. Errechnetes LC-Anpassungsnetzwerk zwischen Schleifenantenne und Antennenumschalter . . . 191
C.13. Optimierte Streuparameter der Schleifenantenne mit LC-Anpassungsnetzwerk/ AWR Design Environment . . . 192
C.14. Streuparametervergleich der simulierten und berechneten S_{11} Streuparameter 192
C.15. Aufgeschraubter Zellsensor auf $LiFePO_4$-Zelle . . . 193
C.16. Querschnitt des aufgeschraubten Zellsensors . . . 193
C.17. 2D-Abstrahlcharakteristik des aufgeschraubten Zellsensors . . . 194
C.18. 3D-Abstrahlcharakteristik des aufgeschraubten Zellsensors . . . 194
C.19. Zellsensor mit aufgetragener Epoxidharz-Schicht . . . 195
C.20. 2D-Abstrahlcharakteristik des Zellsensors mit Epoxidharz-Schicht . . . 195
C.21. Vergleich zwischen vergossenem und unvergossenem Zellsensor . . . 196

D.1. Aufbau zur künstlichen Batteriezellen-Alterung . . . 197
D.2. Künstliche Alterung einer $LiFePO_4$-Zelle . . . 198

E.1. Ladungsabhängige Impedanzmessungen mit FuelCon TrueData-EIS . . . 199
E.2. Ladungsabhängige Impedanzmessungen mit Zellsensoren und Batteriesteuergerät . . . 200
E.3. Impedanzmessungen an ECC-LFPP 45 Ah Zelle 1 . . . 201
E.4. Impedanzmessungen an ECC-LFPP 45 Ah Zelle 2 . . . 202

G.1. Befestigungsadapter . . . 207
G.2. Montierter und vergossener Zellsensor auf einer ECC-LFPP 45Ah $LiFePO_4$-Zelle . . . 207

H.1. PC lauffähiges Goertzel-Testprogramm . . . 210

Literaturverzeichnis

[1] A123. *Nanophosphate® High Power Lithium Ion Cell ANR26650M1-B*. A123 Systems, n.a. 2012. http://www.a123systems.com/lithium-ion-cells-26650-cylindrical-cell.htm; abgerufen am 17. Juni 2015.

[2] Alexander Angold. *Verfahren zur aufwandsreduzierten Elektrochemischen Impedanzspektroskopie für Starterbatterien*. HAW Hamburg, März 2014. Bachelorthesis.

[3] Kevin Banks. *The Goertzel Algorithm*. Embedded Systems Programming, September 2002. http://www.embedded.com/design/configurable-systems/4024443/The-Goertzel-Algorithm; abgerufen am 05. März 2015.

[4] Evgenij Barsoukov and J. Ross Macdonald, editors. *Impedance Spectroscopy: Theory, Experiment, and Applications*. Wiley-Interscience, 2 edition, 3 2005.

[5] Marcel Baunach. *Advanced Timestamping for pairwise Clock Drift Detection in Wireless Sensor - Actuator Networks*. Universitaet Potsdam, 13. GI/ITG KuVS Fachgespraech Sensornetze, Graz University of Technology / Austria / Institute for Technical Informatics, September 2014.

[6] Deutsche Bundesregierung. *Mobilität der Zukunft sauber und kostengünstig*. 2015. http://http://www.bundesregierung.de/Webs/Breg/DE/Themen/Energiewende/Mobilitaet/mobilitaet_zukunft/_node.html; abgerufen am 03. August 2015.

[7] Naim Dahnoun. *Goertzel Code*. Bristol University / Texas Instruments, 2004. http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter17.ppt; abgerufen am 05. März 2015.

[8] Analog Devices. *AD7641 Rev. 0*. Analog Devices, Inc, 2006. http://www.analog.com/static/imported-files/data_sheets/AD7641.pdf; abgerufen am 14. Februar 2015.

[9] Analog Devices. *AD7767 Rev. C*. Analog Devices, Inc, 2010. http://www.analog.com/static/imported-files/data_sheets/AD7767.pdf; abgerufen am 14. Februar 2015.

[10] Analog Devices. *AD7176-2 Rev. B*. Analog Devices, Inc, 2014. http://www.analog.com/static/imported-files/data_sheets/AD7176-2.pdf; abgerufen am 14. Februar 2015.

[11] Analog Devices. *AD7690 Rev. C*. Analog Devices, Inc, 2014. http://www.analog.com/static/imported-files/data_sheets/AD7690.pdf; abgerufen am 14. Februar 2015.

[12] Analog Devices. *AD7691 Rev. D*. Analog Devices, Inc, 2014. http://www.analog.com/static/imported-files/data_sheets/AD7691.pdf; abgerufen am 14. Februar 2015.

[13] Analog Devices. *AD7989 Rev. A*. Analog Devices, Inc, 2014. http://www.analog.com/static/imported-files/data_sheets/AD7989-1_7989-5.pdf; abgerufen am 14. Februar 2015.

[14] Phillip Durdaut. *Zellensensor für Fahrzeugbatterien mit Kommunikation und Wakeup-Funktion im ISM-Band bei 434MHz*. HAW Hamburg, Bachelorthesis, Feb. 2013.

[15] Nationale Plattform Elektromobilität. *Fortschrittsbericht 2014 Bilanz der Marktvorbereitung*. n.a. 2014. http://http://www.bmub.bund.de/fileadmin/Daten_BMU/Download_PDF/Verkehr/emob_fortschrifttsbericht_2014_bf.pdf; abgerufen am 03. August 2015.

[16] Jan Kießling und Xiaobo Liu-Henke Florian Quantmeyer. *Modellbildung und Identifikation der Energiespeicher für Elektrofahrzeuge*. Ostfalia Hochschule fuer angewandte Wissenschaften, n.a.

[17] Alex Friesen. *Influence of different aging mechanisms on the abuse behavior of commercial lithium-ion18650 type cells*. Kraftwerk Batterie 2015, Meet Battery Research Center, University of Münster, 2015.

[18] Heinrich Frohne, Karl-Heinz Löcherer, Hans Müller, Thomas Harriehausen, and Dieter Schwarzenau. *Moeller Grundlagen der Elektrotechnik*. Vieweg+Teubner Verlag, 21 edition, 2008.

[19] Ludwig J. Gauckler. *Electrochemical impedance spectroscopy*. ETH Zuerich, 2003. http://www.nonmet.mat.ethz.ch/education/courses/ceramic2/EIS.ppt; abgerufen am 24. September 2014.

[20] ECC Batteries GmbH. *Technische Daten ECC Batteries GmbH*. ECC Batteries GmbH, n.a. www.eccbatteries.com/files/kunden-_technische_daten_ecc_batteries_gmbh.pdf; abgerufen am 24. Juni 2015.

[21] Gerald Goertzel. *An Algorithm for the Evaluation of Finite Trigonometric Series*. Mathematical Association of America, The American Mathematical Monthly, Vol. 65, No. 1 (Jan., 1958), pp. 34-35, Januar 1958.

[22] José A. Gutierrez. *IEEE Std. 802.15.4. Enabling Pervasive Wireless Sensor Networks*. Berkeley University of California, Graduate Seminar on Sensor Actuator Networks, n.a. 2005. www.cs.berkeley.edu/~prabal/teaching/cs294-11-f05/slides/day21.pdf; abgerufen am 08. Juni 2015.

[23] Michael Haag. *Difference Equation*. OpenStax-CNX, Dezember 2013. http://cnx.org/contents/31c2d232-06c2-4aaa-a6cf-499378420b09@7/Difference_Equation; abgerufen am 26. März 2015.

[24] Carl H. Hamann and Wolf Vielstich. *Elektrochemie*. Wiley-VCH, 1 edition, Juni 1997.

[25] J. Vetter P. Novak M.R. Wagner C. Veitb K.-C. Möller J.O. Besenhard M. Winter M. Wohlfahrt-Mehrens C. Vogler A. Hammouched. *Ageing mechanisms in lithium-ion batteries*. Journal of Power Sources 147 (2005) 269?281, March 2005.

[26] Frieder Herb. *Alterungsmechanismen in Lithium-Ionen-Batterien und PEM-Brennstoffzellen und deren Einfluss auf die Eigenschaften von daraus bestehenden Hybrid-Systemen*. Universität Ulm, 5 2010.

[27] Arthur R. Von Hippel and Alexander S. Labounsky, editors. *Dielectric Materials and Applications*. Artech House Inc, 12 1995.

[28] S.M.M. Alavi C.R. Birkl D.A. Howey. *Time-domain fitting of battery electrochemical impedance models*. Journal of Power Sources 288 (2015) 345?352, August 2015.

[29] F. Huet. *A review of impedance measurements for determination of the state-of-charge or state-of-health of secondary batteries*. Journal of Power Sources 70 (1998) 59-69, Physique des Liquides et Electrochimie, Universitd Pierre et Marie Curie, Paris, 19. May 1997.

[30] Texas Instruments. *24-Bit, Wide Bandwidth Analog-to-Digital Converter, Rev. F*. Texas Instruments, Oktober 2007. http://www.ti.com/lit/ds/symlink/ads1271.pdf; abgerufen am 14. Februar 2015.

[31] Texas Instruments. *Low-Power,2-Channel,24-Bit Analog Front-End for Biopotential Measurements, Rev. B*. Texas Instruments, September 2012. http://www.ti.com/lit/ds/symlink/ads1291.pdf; abgerufen am 14. Februar 2015.

[32] Texas Instruments. *MSP430 SoC With RF Core, Rev. H*. Texas Instruments, September 2012. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=cc430f5137&fileType=pdf; abgerufen am 24. März 2015.

[33] Texas Instruments. *MSPMATHLIB: An Optimized MSP430 Library of Floating-Point Scalar Math Functions*. Texas Instruments, Mai 2013. http://www.ti.com/lit/ug/slau499/slau499.pdf; abgerufen am 21. Juli 2015.

[34] Texas Instruments. *TivaWare Peripheral Driver Library, Rev. 2.1.1.71*. Texas Instruments, Mai 2015. http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=spmu298&fileType=pdf; abgerufen am 21. Juli 2015.

[35] Texas Instruments. *TLV700 200-mA Low-IQ Low-Dropout Regulator for Portable Devices*. Texas Instruments, April 2015. http://www.ti.com/lit/ds/slvsa00e/slvsa00e.pdf; abgerufen am 04. August 2015.

[36] Texas Instuments. *ARM Cortex-M4F based MCU TM4C1294 Connected LaunchPad*. n.a. 2015. http://www.ti.com/tool/ek-TM4C1294xl; abgerufen am 03. August 2015.

[37] Maxim Integrated. *MAX11200/MAX11210, Rev. 2*. Maxim Integrated, Dezember 2012. http://datasheets.maximintegrated.com/en/ds/MAX11200-MAX11210.pdf; abgerufen am 26. Juni 2015.

[38] Suyash Jain. *Layout Review Techniques for Low Power RF Designs - SWRA367A*. Texas Instruments, August 2012. http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=swra367&fileType=pdf; abgerufen am 14. Juli 2015.

[39] Don H. Johnson. *Scanning Our Past - Origins of the Equivalent Circuit Concept: The Voltage-Source Equivalent*. IEEE, 2003.

[40] Douglas L. Jones. *Goertzel's Algorithm*. OpenStax-CNX, September 2006. http://cnx.org/contents/930e1c73-03a8-49ba-9c1d-c4e959944572@5/Goertzel%27s_Algorithm; abgerufen am 26. März 2015.

[41] Klaus W. Kark. *Antennen und Strahlungsfelder: Elektromagnetische Wellen auf Leitungen, im Freiraum und ihre Abstrahlung*. Springer Vieweg, 5., überarb. u. erw. aufl. 2014 edition, 4 2014.

[42] Martin Kiel. *Impedanzspektroskopie an Batterien unter besonderer Berücksichtigung von Batteriesensoren für den Feldeinsatz*. Shaker, 1., aufl. edition, 6 2013.

[43] Microstar Laboratories. *Quick Development of the Goertzel Filter*. Microstar Laboratories, 2009. http://www.mstarlabs.com/dsp/goertzel/goertzel.html; abgerufen am 26. März 2015.

[44] Douglas Lyon. *The Discrete Fourier Transform Part 2: Radix 2 FFT*. ETH Zuerich, July 2009. http://www.jot.fm/issues/issue_2009_07/column2.pdf; abgerufen am 29. März 2015.

[45] Robert Kostecki Marie Kerlau. *Interfacial Impedance Study of Li-Ion Composite Cathodes during Aging at Elevated Temperatures*. Journal of The Electrochemical Society, nr. 9, s. 1644 - 1648 edition, Vol. 153 (2006).

[46] Marian Mazurek. *Impedanzspektroskopie an Anodenkatalysatoren für Membranbrennstoffzellen*. Universität Darmstadt, 2006. http://tuprints.ulb.tu-darmstadt.de/694/; abgerufen am 08. April 2015.

[47] Eberhard Meissner and Gerolf Richter. *Battery Monitoring and Electrical Energy Management Precondition for future vehicle electric power systems*. Journal of Power Sources 116 (2003) 79?98, 2003.

[48] Eike Mense. *Hard- und Softwareentwicklung für einen drahtlosen Batterie-Zellen-Sensor zur elektrochemischen Impedanzspektroskopie*. HAW Hamburg, November 2014. Masterthesis.

[49] Allegro MicroSystems. *ACS716*. Allegro MicroSystems, 2013. http://www.allegromicro.com/~/media/Files/Datasheets/ACS716-Datasheet.ashx?la=en; abgerufen am 06. August 2015.

[50] Otto Mildenberger. *Übertragungstechnik: Grundlagen analog und digital (Studium Technik)*. Vieweg Verlagsgesellschaft, 1997 edition, 9 1997.

[51] Marta Baginska Benjamin J. Blaiszik Ryan J. Merriman Nancy R. Sottos Jeffrey S. Moore and Scott R. White. *Autonomic Shutdown of Lithium-Ion Batteries Using Thermoresponsive Microspheres*. Adv. Energy Mater., November 2011.

[52] Miguel Morales and Dung Dang. *CC430 RF Examples, Rev. C*. Texas Instruments, September 2013. http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=slaa465c&fileType=pdf; abgerufen am 13. März 2015.

[53] Jens-Rainer Ohm and Hans Dieter Lüke. *Signalübertragung*. Springer, 6., neubearb. u. erw. aufl. edition, 5 1995.

[54] Alan V. Oppenheim and Ronald W. Schafer. *Zeitdiskrete Signalverarbeitung*. Oldenbourg Wissenschaftsverlag, durchgesehene auflage edition, 12 1998.

[55] Panasonic. *Datasheet: Panasonic Aluminum Electrolytic Capacitor, Radial Lead Type FC A*. Panasonic.

[56] Lothar Papula. *Mathematische Formelsammlung für Ingenieure und Naturwissenschaftler*. Vieweg, 8. auflage edition, 2003.

[57] Andreas Jossen Peter Keil. *Aufbau und Parametrierung von Batteriemodellen*. Technische Universität München, Lehrstuhl für Elektrische Energiespeichertechnik Technische Universität München, Arcisstr. 21, 80333 München, n.a. https://mediatum.ub.tum.de/doc/1162416/1162416.pdf; abgerufen am 23. März 2015.

[58] J. E. B. Randles. *Kinetics of Rapid Electrode Reactions*. University of Birmingham, Department of Chemistry, University of Birmingham, 5. March 1947.

[59] Brian Redding. *Bluetooth Technology - Advanced*. Bluetooth World 2013, 2013.

[60] Michael Roscher. *Zustandserkennung von LiFePO4-Batterien für Hybrid- und Elektrofahrzeuge*. Shaker, 1., aufl. edition, 2 2011.

[61] Joachim Georg Roth. *Impedanzspektroskopie als Verfahren zur Alterungsanalyse von Hochleistungs-Lithium-Ionen-Zellen*. Dr. Hut, 6 2013.

[62] Nico Sassano. *Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung*. HAW Hamburg, August 2013. Bachelorthesis.

[63] Matthias Schöllmann. *Energiemanagement und Bordnetze - moderne Bordnetzarchitekturen und innovative Lösungen für Energiemanagementsysteme in Kraftfahrzeugen*. Expert-Verlag, Renningen, 1. aufl. edition, 2005.

[64] A. Tchegho. *Optimal High-Resolution Spectral Analyzer*. Technische Universität München, 2008. http://www.date-conference.com/proceedings1/papers/2008/date08/pdffiles/01.5_1.pdf; abgerufen am 22. Juli 2015.

[65] Linear Technology. *LTC2376-20 Rev. A*. Linear Technology, 2014. http://cds.linear.com/docs/en/datasheet/237620f.pdf; abgerufen am 14. Februar 2015.

[66] Dietmar Thate. *Energie aus der Kaffeepackung*. Opel AG, 7 2011. http://www.opel-blog.com/2011/07/01/energie-aus-der-kaffeepackung/; abgerufen am 02. Juli 2015.

[67] Ulrich Tietze, Christoph Schenk, and Eberhard Gamm. *Halbleiter-Schaltungstechnik*. Springer, 14., überarb. und erw. aufl. 2012 edition, 11 2012.

[68] Wolfgang Weydanz u. Andreas Jossen. *Moderne Akkumulatoren richtig einsetzen*. Reichardt Verlag, 1 edition, Januar 2006.

[69] Callaway E. Gorday P. Hester L. Gutierrez J.A. Naeve M. Heile B. Bahl V. *Home Networking with IEEE 802.15.4: A Developing Standard for Low-Rate Wireless Personal Area Networks*. Communications Magazine, IEEE (Volume:40 , Issue: 8), 8 2002.

[70] Karl-Ragmar Riemschneider Valentin Roscher, Nico Sassano. *Synchronisation using Wireless Trigger-Broadcast for Impedance Spectroscopy of Battery Cells*. IEEE Sensors Applications Symposium (SAS), Zadar, Croatia, 04 2015.

[71] Norbert Wagner. *Einsatz der Impedanzspektroskopie in der Brennstoffzellenforschung*. Oldenbourg Wissenschaftsverlag, 2011. http://elib.dlr.de/71382/1/tm0073.pdf; abgerufen am 28. Februar 2015.

[72] Henning Wallentowitz and Konrad Reif. *Handbuch Kraftfahrzeugelektronik: Grundlagen - Komponenten - Systeme - Anwendungen (ATZ/MTZ-Fachbuch)*. Vieweg+Teubner Verlag, 2006 edition, 9 2006.

[73] Martin Werner. *Digitale Signalverarbeitung mit MATLAB*. Vieweg+Teubner Verlag, 3., vollst. überarb. u. akt. aufl. 2006 edition, 4 2006.

Abkürzungsverzeichnis

AD Analog-Digital

BS Batteriesteuergerät

DFT diskreten Fourier Transformation

DoD Depth of Discharge

EIS elektrochemische Impedanzspektroskopie

ENOB Effective Number of Bits

EoL End of Life

ESR Equivalent Series Resistance

FFT Fast Fourier-Transformation

HAW Hochschule für Angewandte Wissenschaften

HF Hochfrequenz

IC Integrated Circuit

ISR Interrupt Service Routine

LiFePO Lithium-Eisenphosphat

OCV Open Circuit Voltage

OOK On-Off-Keying

PCB Printed Circuit Board

RF Radio frequency

SAR Sukzessives Approximation Register

SNR Signal-Rausch-Abstand

SoC State of Charge

SoH State of Health

ZS Zellsensor

A. Hochauflösende AD-Wandler Erweiterungsplatine

In diesem Abschnitt, wird die fertig montierte hochauflösende AD-Wandler Erweiterungsplatine vorgestellt und die mit diesem AD-Wandler erzielten Messungen aufgezeigt.

In Abbildung A.1 ist die montierte Erweiterungsplatine auf dem Zellsensor zu sehen. Diese Erweiterungsplatine lässt sich über eine einfache Steckverbindung auf der Unterseite mit den Zellsensor verbinden. Zur mechanischen Befestigung kann die Platine durch zwei Schrauben mit dem Zellsensor verbunden werden. Die einzelnen Komponenten der Schaltung sind hier farblich hervorgehoben. Zudem sind auf der Platine noch einige Ausgänge des Mikrocontrollers vorhanden, die als Debug Möglichkeiten genutzt werden können. Die genaue Belegung der Ausgänge ist aus dem Schaltplan in Anhang J zu entnehmen.

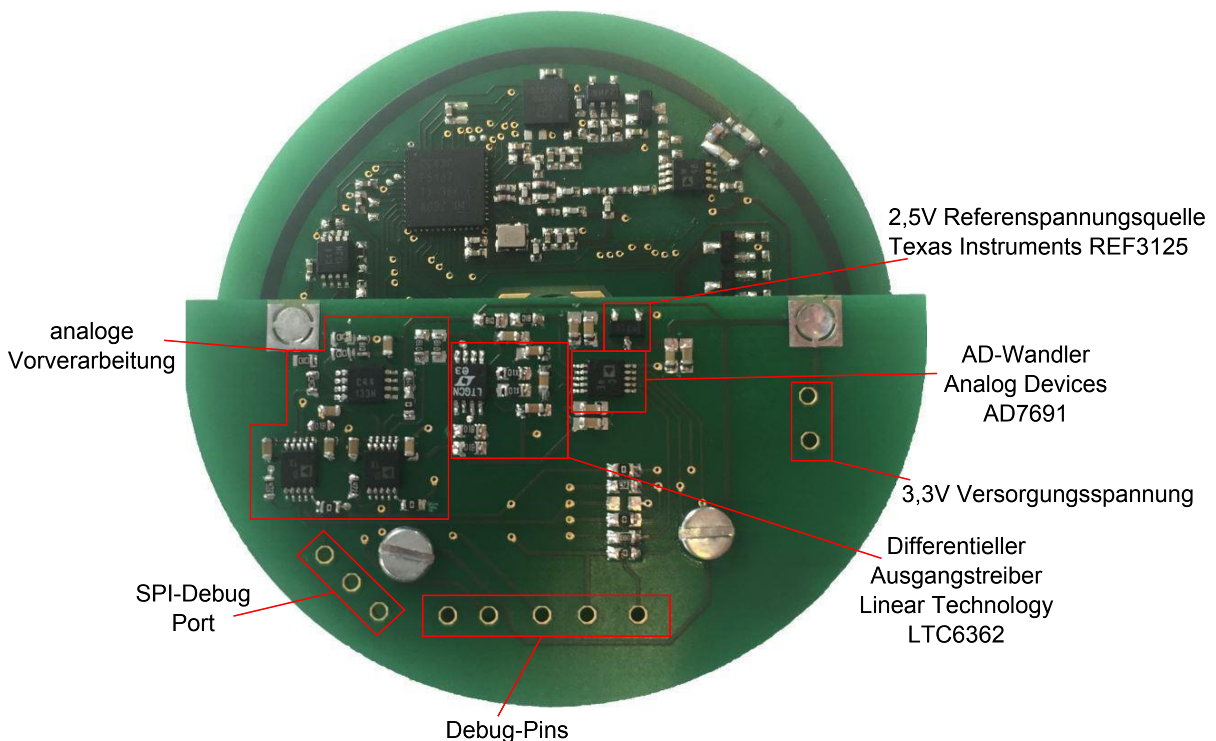


Abbildung A.1.: Montierte AD-Wandler Erweiterungsplatine auf einem Zellsensor

Bei der Inbetriebnahme der Erweiterungsplatine wurde festgestellt, dass ein Schaltungsfehler auf der Platine vorhanden ist. Dieser Fehler verhindert, dass der AD-Wandler

einen Interrupt an den Mikrocontroller ausgeben kann. Ursache hierfür war, dass der SPI-Dateneingang (SDI) am AD-Wandler mit dem SPI-Datenausgang (MOSI) des Mikrocontrollers verbunden war. Dieser muss aber mit der 3,3 V Versorgungsspannung verbunden sein, damit der Interrupt Betrieb möglich ist [12]. Dieser Fehler konnte aber leicht durch eine Lötbrücke behoben werden und wurde im Schaltplan und im Layout, die sich im Anhang J und K befinden, bereits ausgebessert.

Ausführliche Untersuchungen konnten mit dieser Erweiterungsplatine aus Zeitgründen nicht gemacht werden. Dennoch konnten in den Voruntersuchungen einige Messungen mit dieser Schaltung durchgeführt werden, dessen Ergebnisse hier kurz vorgestellt werden. Da die genaue Synchronisation zwischen den Strom- und Spannungswerten zu diesem Zeitpunkt noch in Bearbeitung war, wurde die Strommessung von einem zweiten Zellsensor durchgeführt. Dadurch konnte von einer synchronen Messung zwischen Storm- und Spannungswerten ausgegangen werden.

Im Folgenden werden zwei verschiedene Messungen gezeigt, die durch den hochauflösenden AD-Wandler gemessen werden konnten.

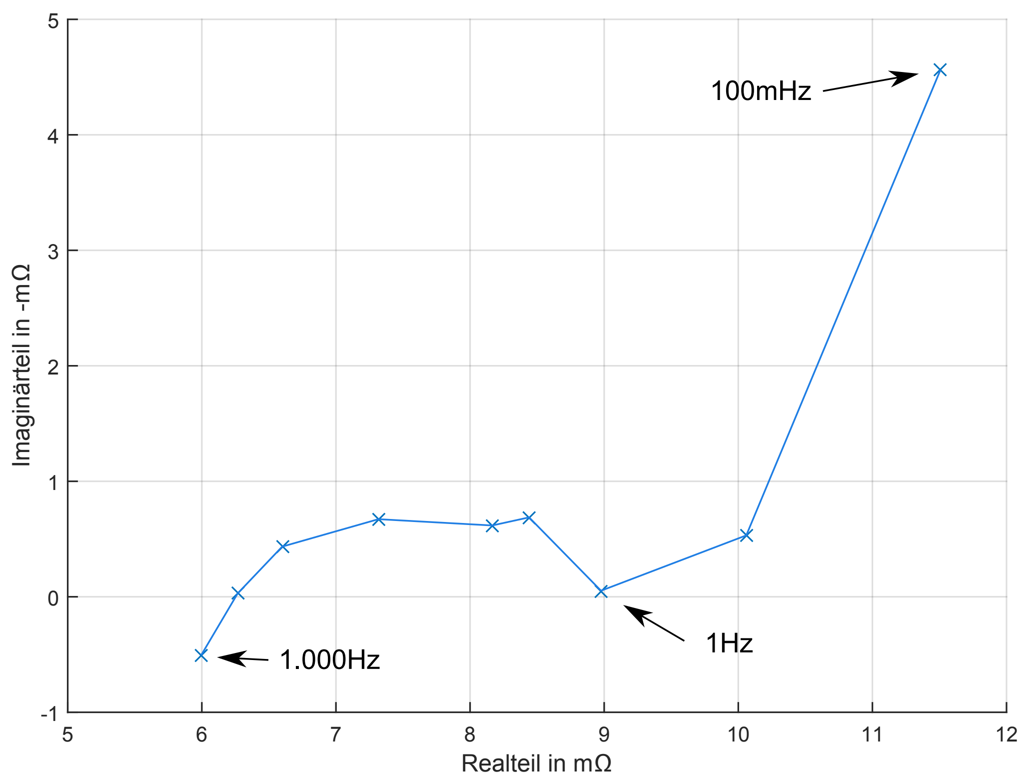


Abbildung A.2.: Messung mit AD7691 AD-Wandler

Die Abbildung A.2 zeigt die aufgenommen Impedanzkurve. Diese zeigt den typischen Verlauf, der erwartet wurde. Gemessen wurde dieses Impedanzspektrum bei normaler Zimmertemperatur von ca. 20°C. Der Ladezustand der Zelle wurde bei dieser Messung nicht beachtet. Nach der Auswertung der Messungen in Abschnitt 6.3.3 kann aber von einer entladenen Zelle ausgegangen werden.

Die folgende Messung in Abbildung A.3 zeigt ebenfalls eine Messung, die mit dem hochauflösenden AD-Wandler gemessen werden konnte.

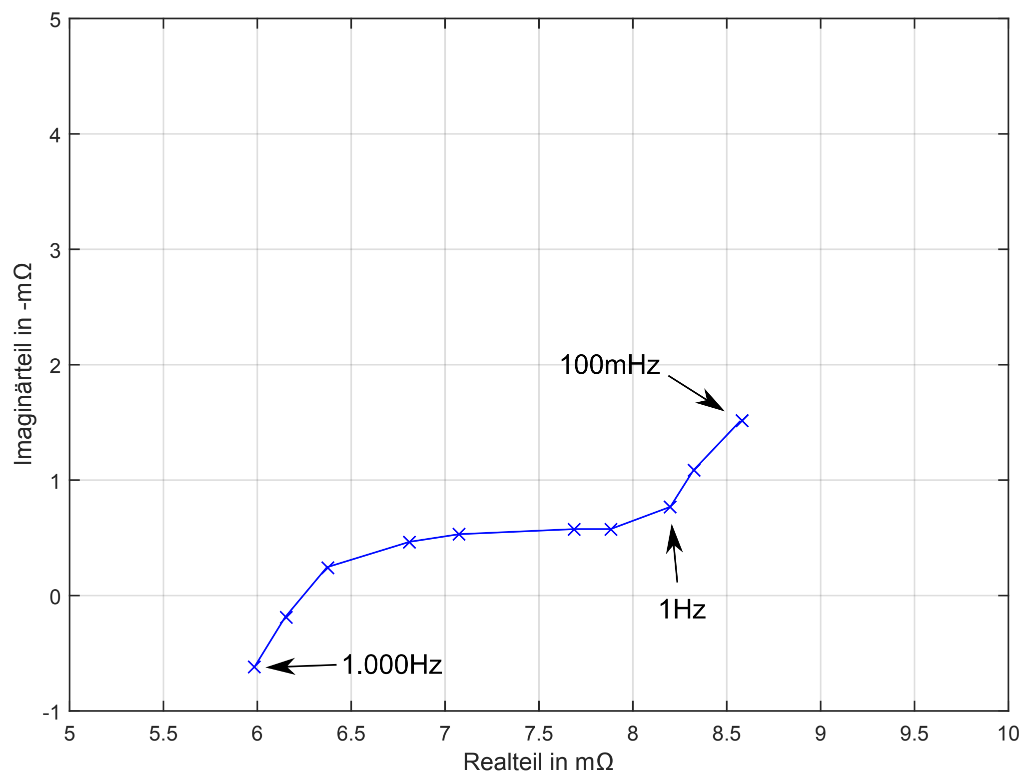


Abbildung A.3.: Weitere Messung mit AD7691 AD-Wandler

Zu beiden Messungen sind keine Vergleichswerte mit dem FuelCon TrueData-EIS vorhanden. Dennoch kann gesagt werden, dass die Messung mit dem hochauflösenden AD-Wandler ebenfalls sehr gute quantitative Messwerte liefert. Allerdings ist bei beiden Messungen zu beobachten, dass beide Impedanzkurven im Vergleich zu den Messungen mit dem internen AD-Wandler des Mikrocontrollers einen geringeren Imaginärteil aufweisen. Ob dies an einem Messfehler liegt oder durch auftretende Induktivitäten in der Schaltung oder des Messaufbaus liegt kann an dieser Stelle nicht gesagt werden. Dazu sind weitere Untersuchungen notwendig, die in dieser Arbeit nicht durchgeführt werden.

B. Spannungsversorgungs-Erweiterungsmodul

Ein möglicher Lösungsansatz zur Stabilisierung der Spannungsversorgung wird im Folgenden vorgestellt. Dabei sollen die Spannungseinbrüche, die in Abschnitt 6.1.2 gezeigt wurden, verhindert werden und somit ein ungestörter Betrieb der analogen Vorverarbeitung gewährleistet werden. Die Idee ist dabei ein höheres Spannungspotenzial auf den Zellsensor zu bringen, damit die Spannungseinbrüche bei einer Abschaltung nicht mehr in den Bereich der 3,3 V Spannungsversorgung fallen.

Dabei wird die Spannung zunächst durch einen Step-Up-Converter TPS61202 auf das Potenzial von 5,0 V gebracht. Da die Spannungsversorgung des Zellsensors aber niedriger ist, wird diese durch den LDO TLV70033DDCT von Texas Instruments wieder auf die benötigten 3,3 V gebracht.
Dieser besitzt einen geringen Dropout von 43 mV [35], wodurch ein Spannungsabfall von bis zu 1,657 V möglich ist, bevor die 3,3 V beeinflusst werden.

$$5{,}0\,\text{V} - 3{,}3\,\text{V} - 43\,\text{mV} = 1{,}657\,\text{V} \tag{B.1}$$

Diese Idee wurde auf einer Erweiterungsplatine realisiert, die einfach durch den schon vorhandenen Erweiterungsstecker mit dem Zellsensor verbunden wird. Lediglich der positive Zellanschluss muss mit der Zellsensorplatine und der Zelle verbunden werden. Abbildung B.1 zeigt das bestückte Erweiterungsmodul.

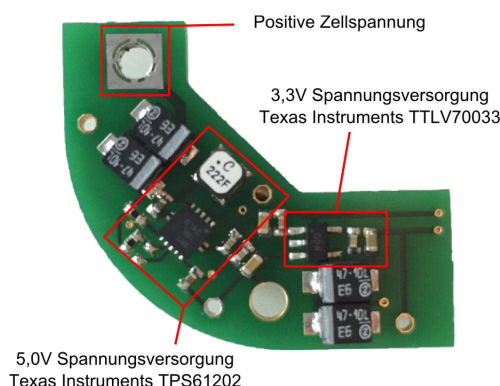


Abbildung B.1.: Spannungsversorgungs-Erweiterungsmodul

In Abbildung B.2 ist das montierte Erweiterungsmodul auf dem Zellsensor zu sehen.

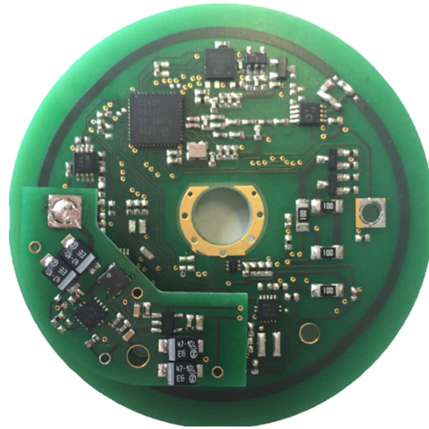


Abbildung B.2.: Zellsensor mit montiertem Spannungsversorgungs-Erweiterungsmodul

Erste Messungen mit den Erweiterungsmodul zeigten, dass durch diese Maßnahme die Beeinflussung der analogen Vorverarbeitung verhindert werden kann. Es konnten Messungen bis zu 6 kHz durchgeführt werden, bevor es wieder zu Störungen in der Zellversorgungsspannung kommt. Dies kommt daher, dass der TPS61202 ab einer bestimmtem Messfrequenz bzw. Abschaltrate nicht mehr schafft, die 5,0 V stabil zu halten, wodurch es zu einem langsamen Absinken der Spannung kommt.

Es wurde aber auch festgestellt, dass nach den Ab- und Anschalten des TPS61202 eine Mindestzeit von ca. 480μs vergehen muss, bevor eine neue Messung gestartet werden kann. Dies ist in Abbildung B.3 zu sehen.

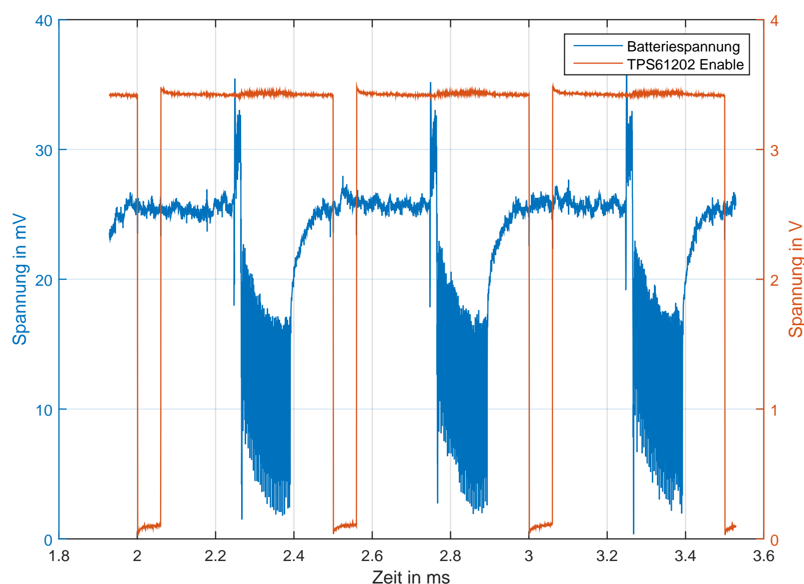


Abbildung B.3.: Abschaltung des TPS61202

In Abbildung B.3 sind die bekannten Störungen auf die Batteriespannung zu sehen. Neben den Störungen, sind auch die Bereiche der Abschaltung des TPS61202 zu sehen. Nach jeder Anschaltung (steigende Flanke in der Abbildung) des TPS61202 tauchen Störungen auf der Batteriespannung auf, die erst nach einer Zeit von ungefähr 480μs wieder verschwinden. Dies beschränkt die mögliche Messrate auf etwas über 2 kHz.

$$f_{max.} = \frac{1}{480\,\mu s} = 2083{,}3\,\text{Hz} \tag{B.2}$$

Ein schnellere Messrate ist zwar technisch möglich, aber es besteht aber die Gefahr, dass dadurch die Störungen mitgemessen werden.

Ob dieses Problem durch Schaltungsänderungen gelöst werden kann, muss in anderen Arbeiten untersucht werden. Ebenso kann untersucht werden, wie sich die Störungen auf die Messwerte auswirken.

C. Simulation der Schleifenantenne

Dieses Kapitel soll einen Einblick in die Simulation des Zellsensors hinsichtlich seiner Abstrahlcharakteristik geben. Simuliert wird dabei mit dem 3D-FEM Simulationsprogramm CST Microwave Studio. Dieses erlaubt, komplette Platinenstrukturen auf ihre Abstrahlcharakteristik hin zu simulieren und zu untersuchen. Ziel dieser Simulation soll es sein, das Layout zu optimieren und die Abstrahlung der Schleifenantenne zu verbessern. Ein besonderes Augenmerk soll dabei auf lange Leitungsführungen gelegt werden. Diese können bei bestimmten Längen als parasitäre Antenne wirken und die Abstrahlcharakteristik beeinflussen. Ein Problem der parasitären Antennen ist, dass sie in Resonanz geraten können und einen Teil der aufgenommenen Leistung phasenverschoben wieder abstrahlen. Dadurch kann eine Richtwirkung der Antenne entstehen, sodass die Abstrahlcharakteristik in eine Richtung gebündelt wird [41].
Gerade Leitungslängen im Bereich von λ/2 bis etwa λ/8 können dabei als Strahlungsdirektor oder Reflektor auftreten. Diese liegen bei 434 MHz im Bereich von wenigen Zentimetern. Also genau in dem Längenbereich, die viele Signalleitungen auf dem Zellsensor haben.

$$\lambda = \frac{c_0}{f} \cdot \frac{1}{\sqrt{\epsilon_r}} = \frac{300 \cdot 10^6 \, m/s}{434 \cdot 10^6 \, Hz} \cdot \frac{1}{\sqrt{4,5}} = 32,99 \, cm \tag{C.1}$$

Mithilfe dieser Simulation sollen diese Leitungsstücke identifiziert und verbessert werden, damit eine optimale Abstrahlung des Zellsensors gewährleistet wird.

Simulation des Sensorlayouts

Um das Layout des Zellsensors zu simulieren, wurde die gesamte Struktur in CST Microwave Studio geladen. Dabei wurden die obere und untere Kupferfläche mit einer Stärke von 35μm eingefügt. Die Durchkontaktierungen (VIA), welche die Oberseite der Platine mit der Unterseite elektrisch verbinden, wurden als massiver Kupferzylinder eingefügt, um die Struktur hinsichtlich ihrer Komplexität nicht weiter zu belasten. An der realen Platine sind diese Durchkontaktierungen zylindrisch aufgebaut. Abbildung C.1 zeigt die importierte Platinenstruktur.

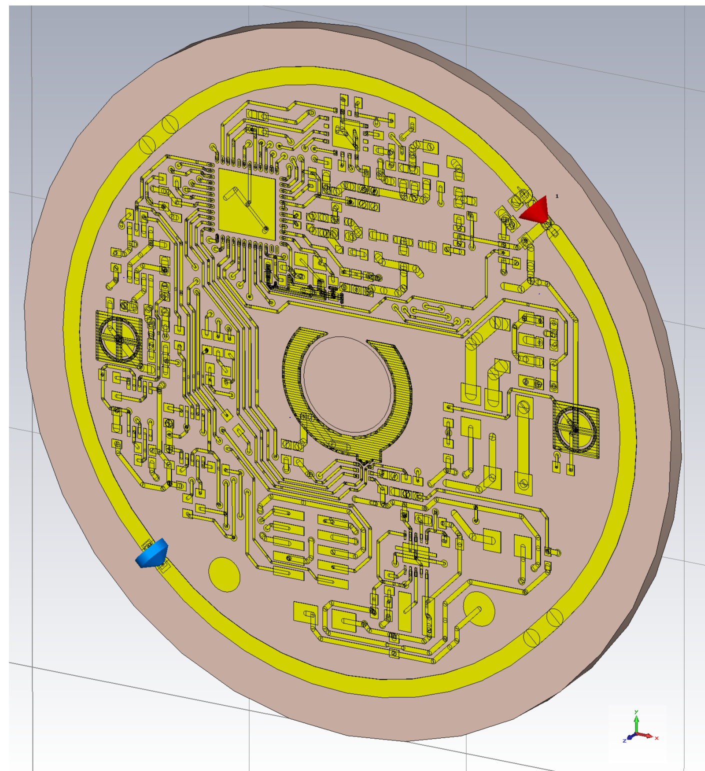


Abbildung C.1.: Platinen-Struktur in CST Microwave Studio

Als Platinenmaterial wurde das aus der Materialbibliothek bereitgestellte FR4 verwendet und mit einer Stärke von 1,6 mm definiert. Es werden somit alle Geometrien für die Simulation verwendet, wie sie der reale Zellsensor besitzt.

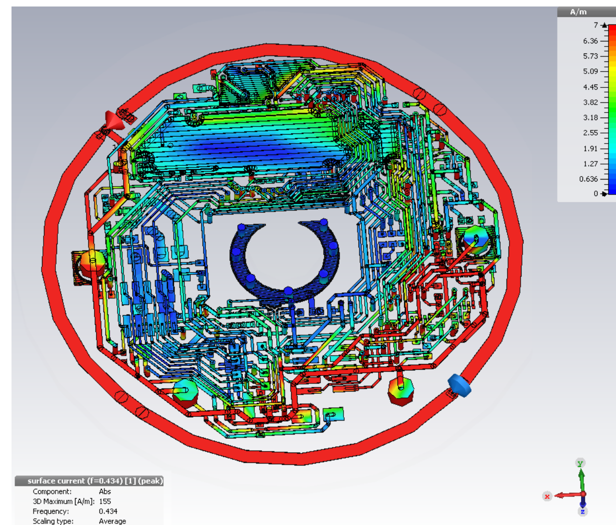


Abbildung C.2.: Platinen-Struktur mit Oberflächenströmen bei 434 MHz

In Abbildung C.2 ist der simulierte Zellsensor mit dessen Oberflächenströmen bei 434 MHz zu sehen. Zu erkennen ist, dass die äußere Schleifenantenne, wie gewünscht nach allen Seiten, eine hohe magnetische Feldstärke aufweist. Die inneren Leiterbahnen zeigen an manchen Stellen ebenfalls eine hohe magnetische Feldstärke. Dies zeigt, dass die inneren Strukturen ebenfalls angeregt werden.

Abbildung C.3 zeigt, dass es durch diese Resonanz der inneren Strukturen zu einer Strahlungskopplung kommt. Dadurch gibt es eine Bündelung der Abstrahlung in eine oder mehrere Richtungen. In diesem Fall erzeugen die Resonanzen der Leiterbahnen eine Bündelung in zwei verschiedene Richtungen.

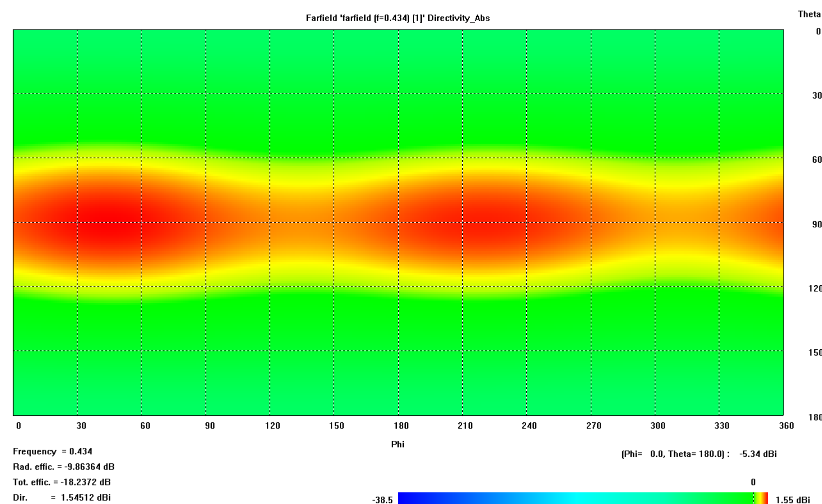


Abbildung C.3.: 2D-Abstrahlcharakteristik bei 434 MHz

Als Vergleich, wurde das gesamte Layout auch bei 509,5 MHz simuliert. Man erkennt in Abbildung C.4 eine deutlich größere Resonanz der Leiterbahnstrukturen als bei der Simulation mit 434 MHz, bedingt durch die geringere Wellenlänge λ bei 509,5 MHz. Dies führt nun zu einer stärkeren Bündelung der Abstrahlung.

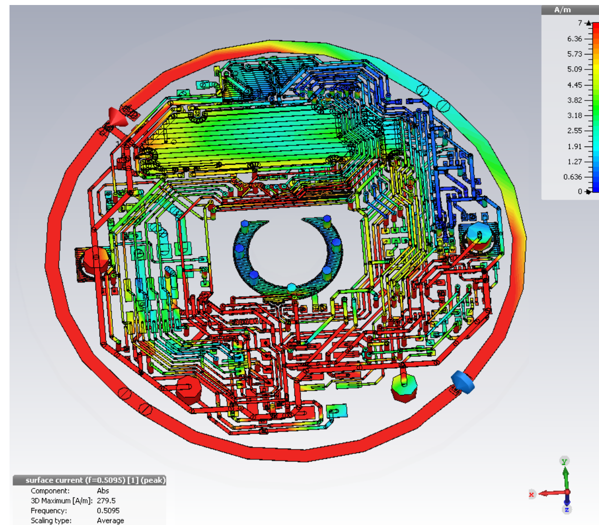


Abbildung C.4.: Platinen-Struktur mit Oberflächenströmen bei 509,5 MHz

Die 2D-Abstrahlcharakteristik bei 509,5 MHz in Abbildung C.5 zeigt den Einfluss dieser Bündelung. Die Abstrahlung der Schleifenantenne wird in eine Richtung gelenkt, wobei es zu einer starken Abschwächung im Bereich von Φ=250° kommt.

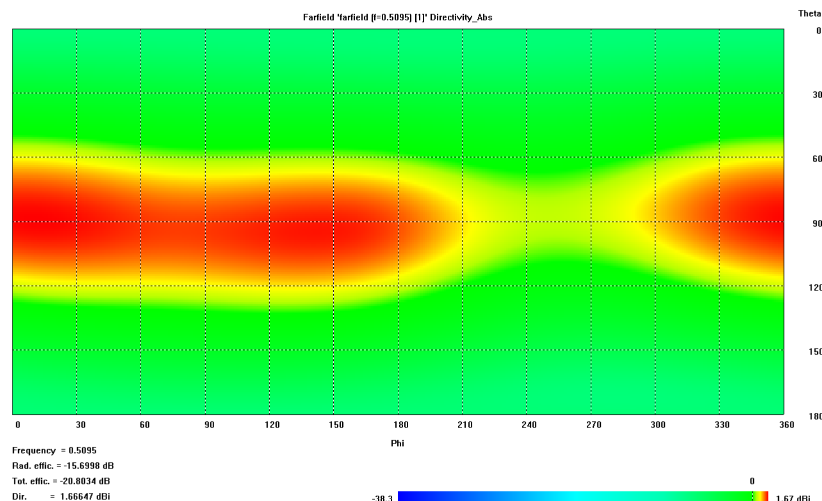


Abbildung C.5.: 2D-Abstrahlcharakteristik bei 509,5 MHz

Um die Ablenkung der Abstrahlung zu verhindern, werden an der Unterseite des Zellsensors jeweils an langen Leitungen, die in der Nähe von $\lambda/4$ liegen, Induktivitäten angebracht. Die Induktivitäten sollen als Blockade für hochfrequente Signale wirken und dadurch die Abstrahleigenschaft der Leitungsstücke verringern. Für die Simulation wurden dabei 27 nH Induktivitäten eingesetzt.

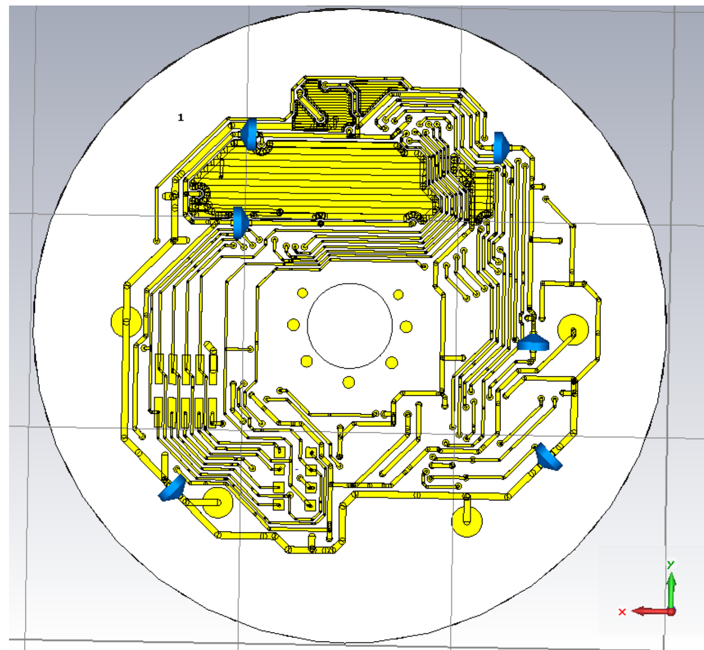


Abbildung C.6.: Platinen-Struktur mit eingebrachten Induktivitäten

Die Stellen, an denen die Induktivitäten eingebracht wurden, sind in Abbildung C.6 mit blauen Pfeilen markiert. Eine Induktivität wurde zusätzlich an der Oberseite angebracht, sodass insgesamt sieben Induktivitäten in das Platinenlayout eingebracht wurden.

Die eingebrachten 27 nH Induktivitäten zeigen eine deutliche Verbesserung hinsichtlich der resonanten Leiterbahnstrukturen. Die magnetische Feldstärke konzentriert sich hier zum größten Teil auf die äußere Schleifenantenne. Es sind nur noch kleine Leiterbahnstücke, die eine leichte magnetische Feldstärke aufweisen, vorhanden.

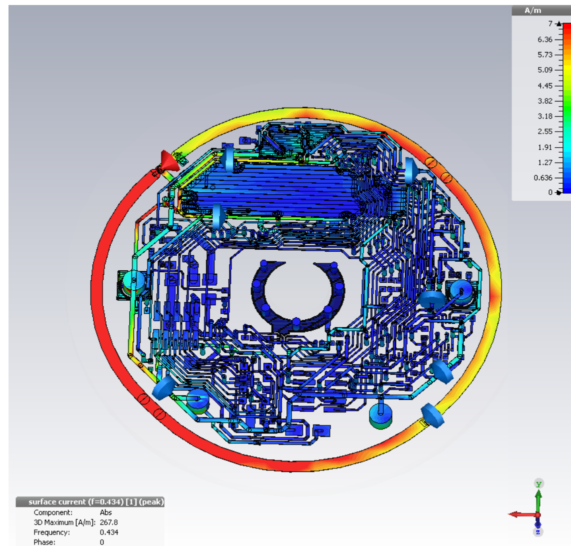


Abbildung C.7.: Überarbeitete Platinen-Struktur mit Oberflächenströmen bei 434 MHz

Das 2D-Abstrahlcharakteristik bei 434 MHz in Abbildung C.8 zeigt, dass die Antenne nun gleichmäßiger abstrahlt als ohne eingesetzte Induktivitäten.

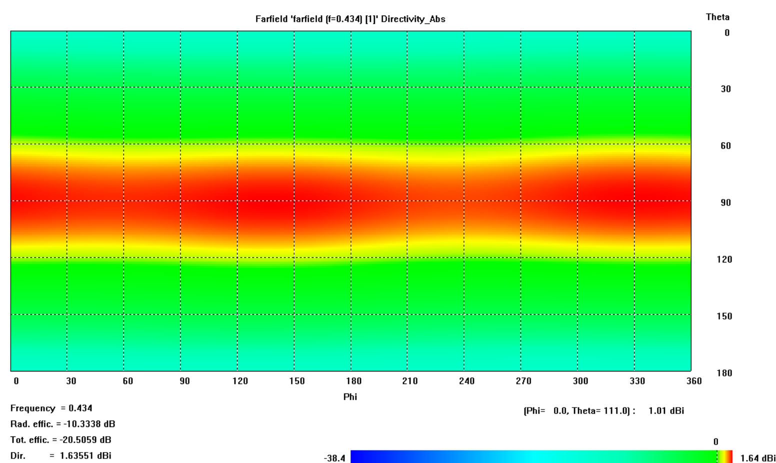


Abbildung C.8.: Korrigiertes 2D-Abstrahlcharakteristik bei 434 MHz

Die 3D-Abstrahlcharakteristik in Abbildung C.9 bei 434 MHz zeigt das erwartete typisch runde Strahlungsdiagramm eines Halbwellendipols.

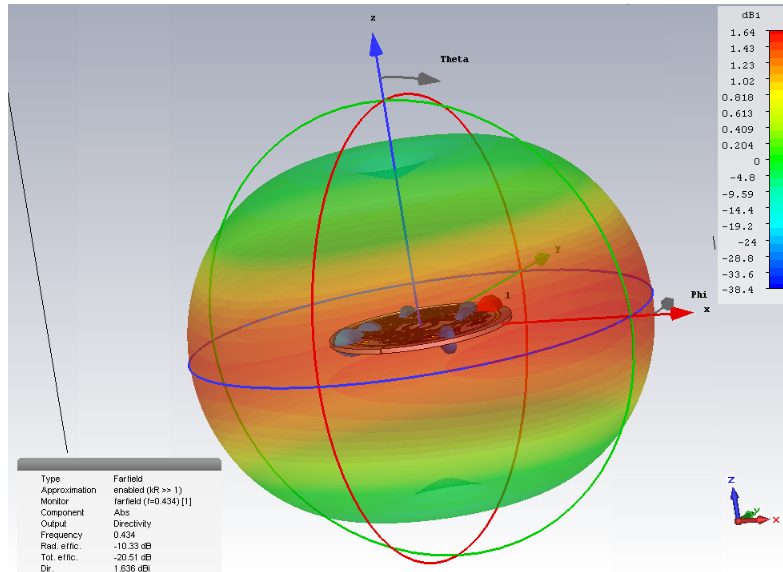


Abbildung C.9.: 3D-Abstrahlcharakteristik bei 434 MHz mit eingesetzten Induktivitäten

Ein Blick in die Theta-Ebene zeigt, dass der Zellsensor horizontal mit max. 1,55 dBi abstrahlt.

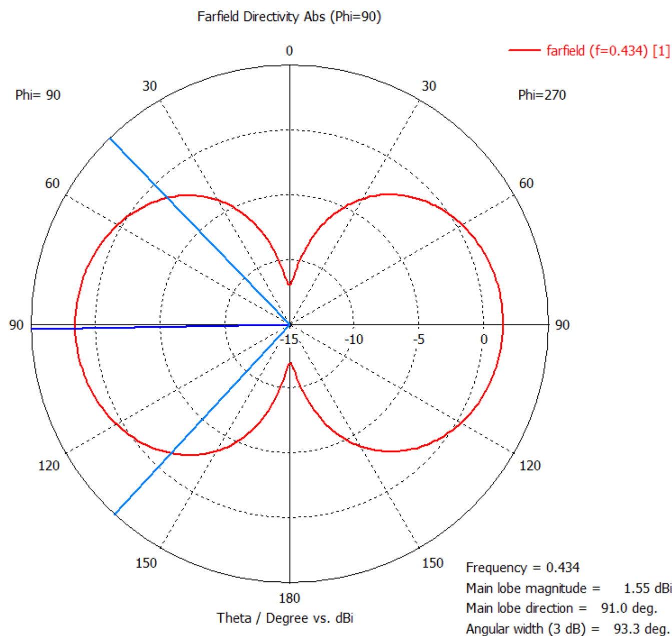


Abbildung C.10.: Richtdiagramm der Schleifenantenne mit eingesetzten Induktivitäten in der Theta-Ebene

Da es sich auf dem Zellsensor noch um eine nicht an 50 Ω angepasste Antenne handelt, muss der S-Parameter S_{11} noch bestimmt werden, um eine optimale Abstrahlwirkung zu erzielen. Der Eingangsreflexionsfaktor S_{11} in Abhängigkeit von der Frequenz ist in Abbildung C.11 dargestellt. Dabei kann man feststellen, dass die Antenne bei mehreren Frequenzen resonant wird. Eine besonders geringe Eingangsreflexion zeigt die Antenne bei einer Frequenz von 478,8 MHz. Bei den gewünschten 434 MHz hingegen, entsteht eine hohe Reflexion von -0,43886 dB. Dies entspricht einer reflektierten Leistung von ca. 90%, die von der Antenne nicht aufgenommen wird.

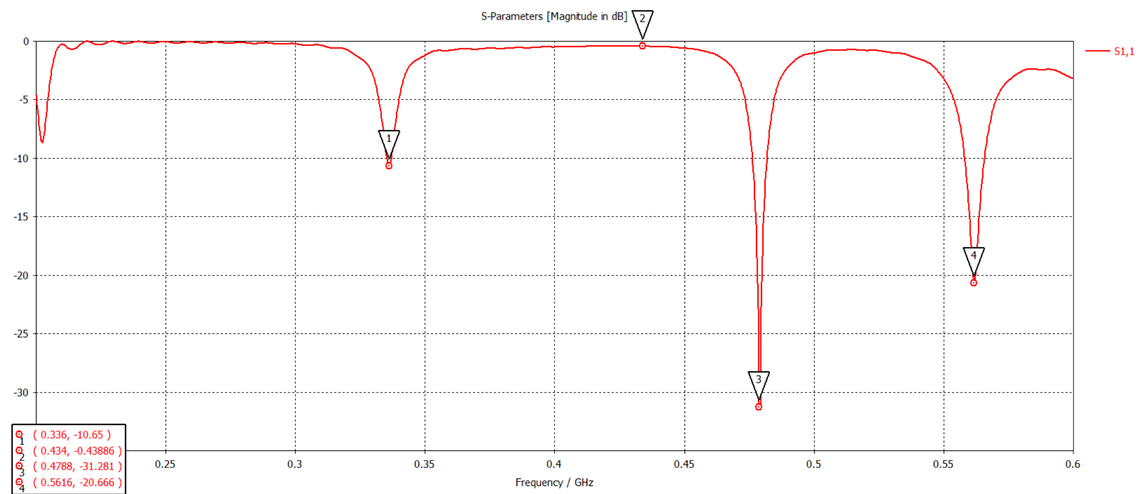


Abbildung C.11.: Streuparameter der Schleifenantenne / CST Microwave Studio

Um eine niedrigere Eingangsreflexion der Antenne bei 434 MHz zu erreichen, muss ein LC-Anpassungsnetzwerk zwischen Schleifenantenne und Antennenumschalter berechnet werden, um die Eingangsimpedanz der Antenne möglichst auf 50 Ω zu bringen und somit eine geringere Eingangsreflexion zu erhalten. Eine Berechnung in AWR Design Environment ergab das folgende LC-Anpassungsnetzwerk.

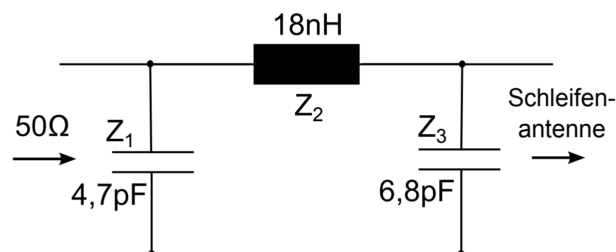


Abbildung C.12.: Errechnetes LC-Anpassungsnetzwerk zwischen Schleifenantenne und Antennenumschalter

Durch das berechnete LC-Anpassungsnetzwerk konnte die Eingangsreflexion bei 434 MHz stark reduziert werden. Diese konnte nun von -0,43886 dB auf -17,92 dB verbessert werden. Dabei reflektiert die Antenne lediglich noch 1,61% der ankommenden Leistung. Es wird also ca. 88,7% mehr Leistung von der Antenne aufgenommen als im unangepassten Fall.

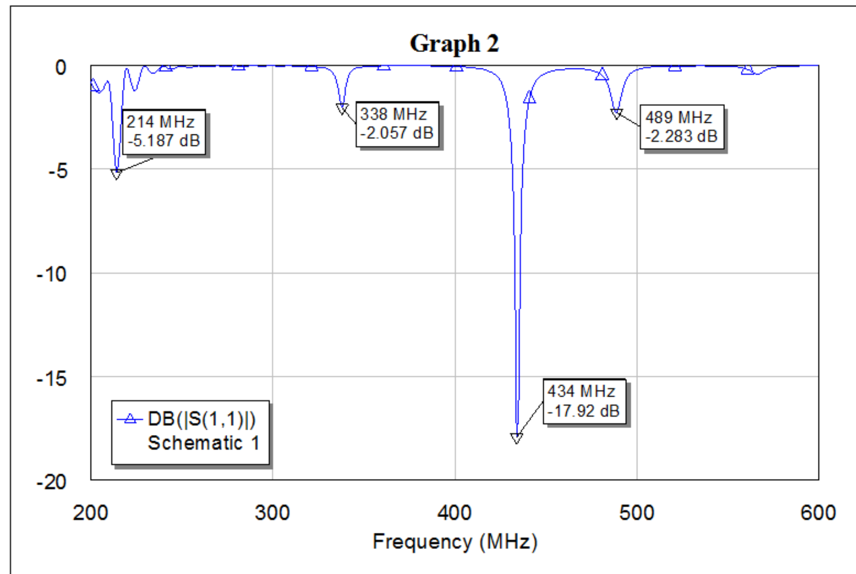


Abbildung C.13.: Optimierte Streuparameter der Schleifenantenne mit LC-Anpassungsnetzwerk/ AWR Design Environment

Ein Vergleich der berechneten Eingangsreflexion mit der Simulierten in Abbildung C.14 zeigt, dass die simulierten Werte gut mit den berechneten Werten übereinstimmen.

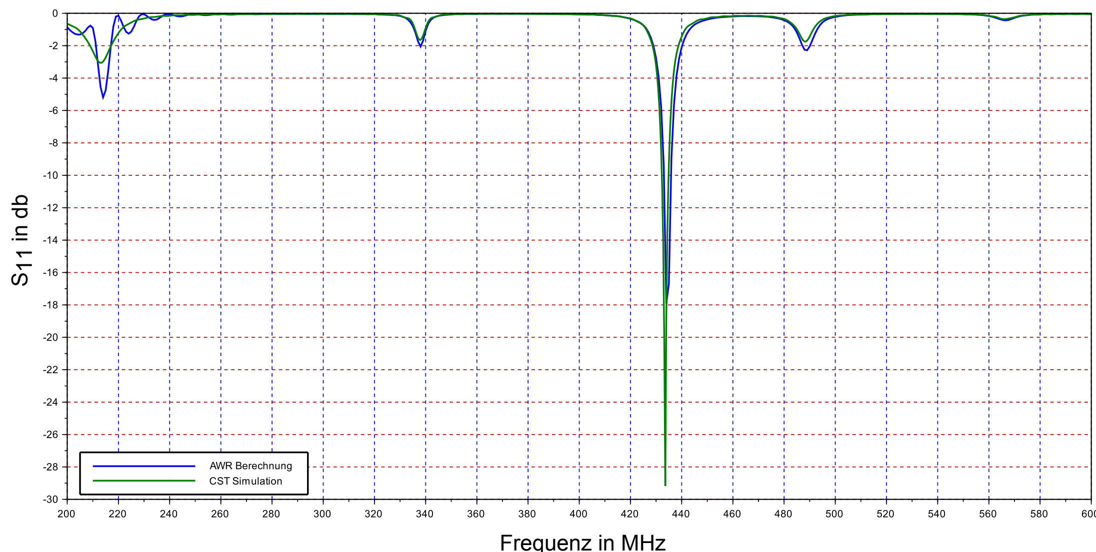


Abbildung C.14.: Streuparametervergleich der simulierten und berechneten S_{11} Streuparameter

Simulation des Zellsensors, montiert auf der Batteriezelle

Der Zellsensor soll auf einer Batteriezelle verbaut werden. Befestigt wird dieser mittels eines Befestigungsadapters aus Messing, der den Zellsensor fest mit der Zelle verbindet und als elektrischer Zellanschluss dient.
Da dieser vermutlich Einfluss auf die Abstrahlcharakteristik des Zellsensors haben wird, wird dieser nun im aufgeschraubten Zustand auf der $LiFePO_4$ Zelle simuliert.
Der Zellsensor wird dabei mit dem zuvor berechneten LC-Anpassungsnetzwerk und den Induktivitäten in langen Leitungsstücken simuliert.

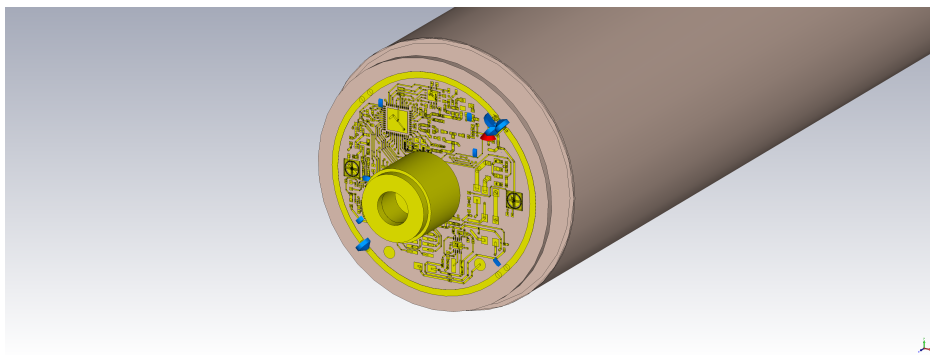


Abbildung C.15.: Aufgeschraubter Zellsensor auf $LiFePO_4$-Zelle

Im Querschnitt in Abbildung C.16 erkennt man, wie der Zellsensor auf die Zelle aufgebracht wird. Das Zellgehäuse besteht aus einer 1 mm dicken Aluminiumhülle. Der Schraubanschluss der Zelle wurde mit einem 17 mm dicken Zylinder mit einer 8 mm dicken Bohrung modelliert. Der Zellsensor klemmt dabei zwischen Zellanschluss und Adapterstück.

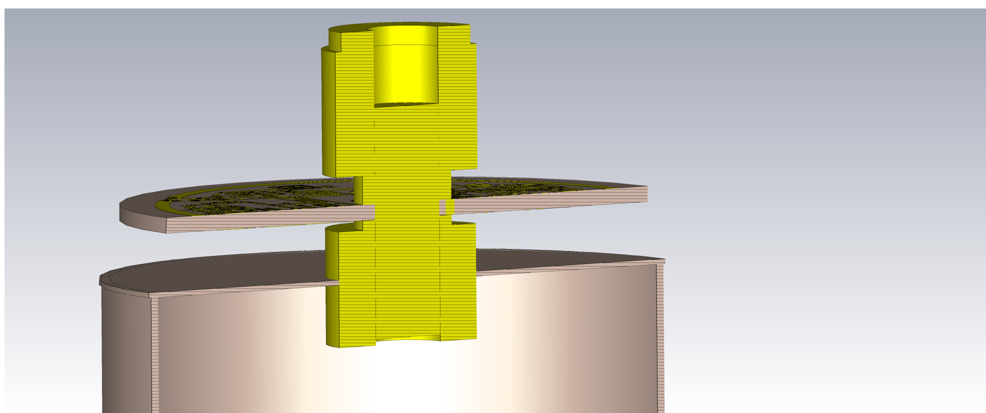


Abbildung C.16.: Querschnitt des aufgeschraubten Zellsensors

Die Simulationsergebnisse zeigen einen leichten Einfluss des Adapterstücks und des Aluminiumzellengehäuses auf die Abstrahlcharakteristik des Zellsensors. Die Abstrahlung liegt hier bei 1,56 dBi.

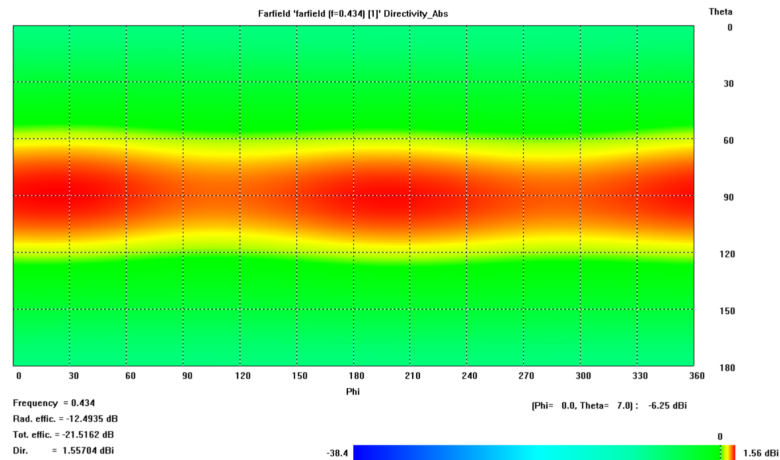


Abbildung C.17.: 2D-Abstrahlcharakteristik des aufgeschraubten Zellsensors

In Abbildung C.18 ist der montierte Zellsensor auf der Batteriezelle und dessen Abstrahlcharakteristik zu sehen.

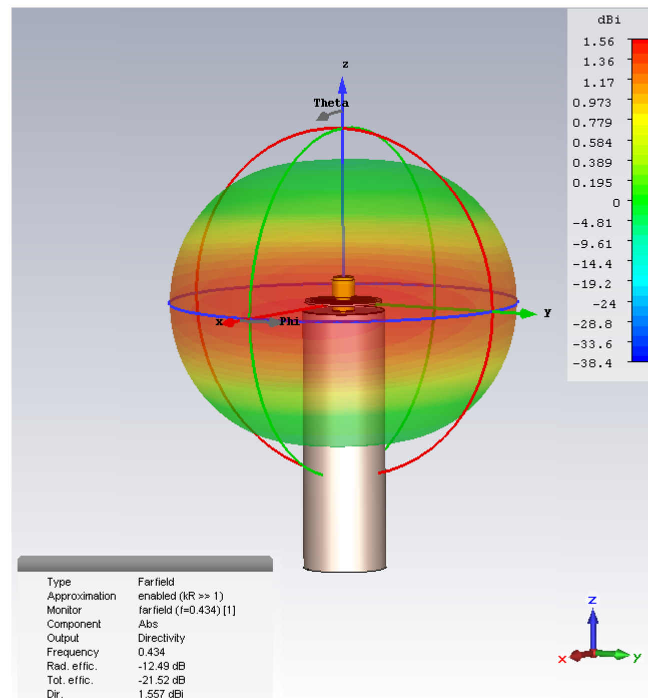


Abbildung C.18.: 3D-Abstrahlcharakteristik des aufgeschraubten Zellsensors

Simulation mit Epoxydharz

Abschließend wird noch der Einfluss von Epoxydharz auf den Zellsensor simuliert und untersucht. Grund hierfür ist, dass die Elektronik des Zellsensors durch die Aufbringung einer Epoxydharz-Schicht von äußeren mechanischen sowie auch chemischen Einflüssen geschützt werden soll. Dazu wird eine 2 mm dicke Epoxydharz-Schicht auf die Platinenstruktur aufgebracht. Dabei wurde das ε_r zu 3,6 und das $tan\delta_\varepsilon$ zu 0,04 gewählt um die Materialeigenschaften des Epoxydharzes zu simulieren [27]. Abbildung C.19 zeigt die aufgetragene Epoxidharz-Schicht auf dem Zellsensor.

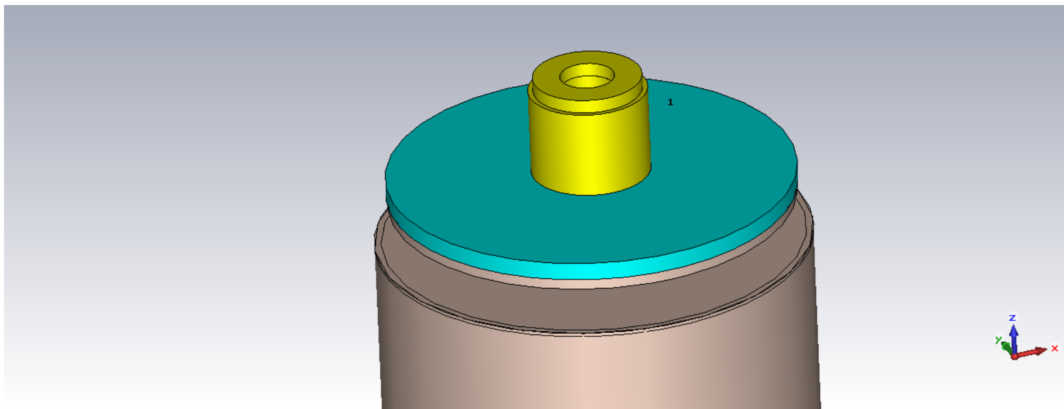


Abbildung C.19.: Zellsensor mit aufgetragener Epoxidharz-Schicht

Die daraus folgende 2D-Abstrahlcharakteristik zeigt die Abbildung C.20.

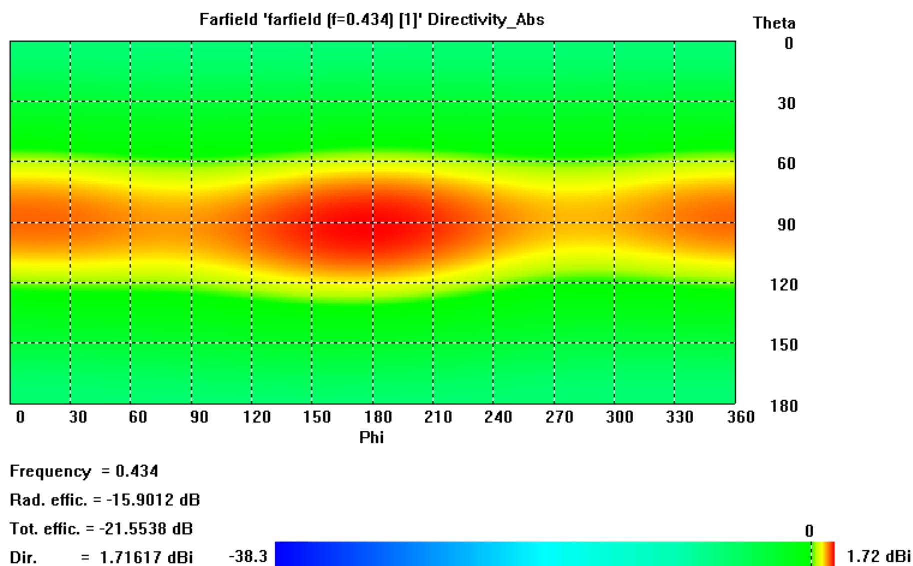


Abbildung C.20.: 2D-Abstrahlcharakteristik des Zellsensors mit Epoxidharz-Schicht

Das Ergebnis dieser Simulation zeigt, dass die aufgebrachte Epoxidharz-Schicht durchaus einen Einfluss auf die Abstrahlcharakteristik des Zellsensors hat. Dies bestätigen auch die Messungen, welche mit zwei realen Zellsensoren durchgeführt wurden und in Abbildung C.21 zu sehen ist. Dabei wurden ein unvergossener und ein vergossener Zellsensor im Abstand von 50 cm zu einem Spektrum-Analysator aufgestellt und vermessen. Beide sendeten dabei ein kontinuierliches Trägersignal ab, welches vom Spektrum-Analysator empfangen wurde. Dabei konnte ebenfalls ein Einfluss festgestellt werden, welcher sich in unterschiedlichen Empfangsleistungen darstellte. Eine Richtwirkung des Zellsensors konnte damit aber nicht nachgewiesen werden, dazu sind weitere Messungen in einer Antennen-Messhalle notwendig.

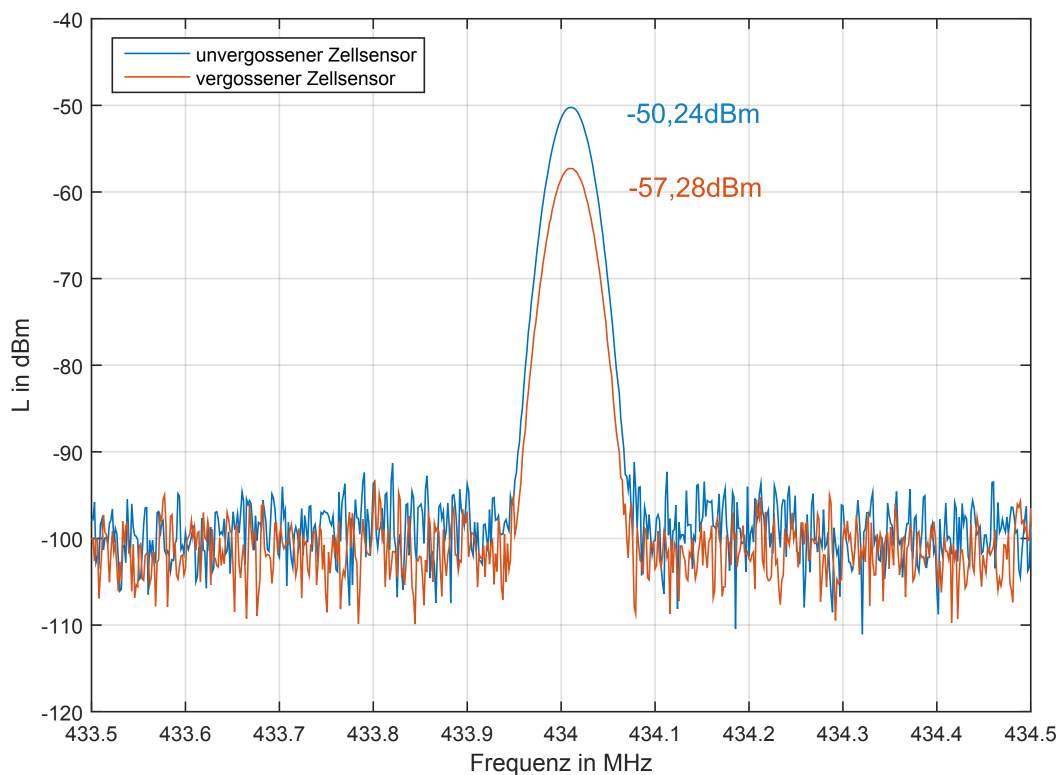


Abbildung C.21.: Vergleich zwischen vergossenem und unvergossenem Zellsensor

D. Alterung einer Batteriezelle

In diesem Kapitel wird die künstliche Alterung einer Batteriezelle vorgestellt, welche im Rahmen dieser Arbeit durchgeführt wurde.
Um zu untersuchen, wie sich das Impedanzspektrum einer $LiFePo_4$-Zelle über die Zeit verändert, wurde eine künstliche Alterung einer Zelle durchgeführt. Dabei ist aus der Literatur bekannt, dass eine Alterung durch Zugabe von Temperatur teilweise stark beschleunigt werden kann [26]. Es wird berichtet, das die SEI-Dicke mit zunehmender Temperatur stark anwächst [45]. Dies führt zu einem zunehmenden Kapazitätsverlust sowie einer Erhöhung des Innenwiderstands und somit zu einer Alterung der Zelle.
Eine solche künstliche Alterung wurde nun in einem mehrwöchigen Test durchgeführt. Das Impedanzspektrum der alternden Batteriezelle wurde dabei jede Woche durch das FuelCon TrueData-EIS vermessen. Durch das Lagern einer Batteriezelle bei einer Umgebungstemperatur von 55°C in einem Temperaturschrank wurde versucht, diese Alterung zu beschleunigen. Neben der Temperierung wurde die Batteriezelle dabei zyklisch aufgeladen und anschließen entladen. Dadurch erhielt die Zelle 100 volle Ladezyklen pro Woche. Dies sollte ebenfalls der schnelleren Alterung der Zelle dienen. Den technischen Aufbau zur künstlichen Alterung zeigt Abbildung D.1.

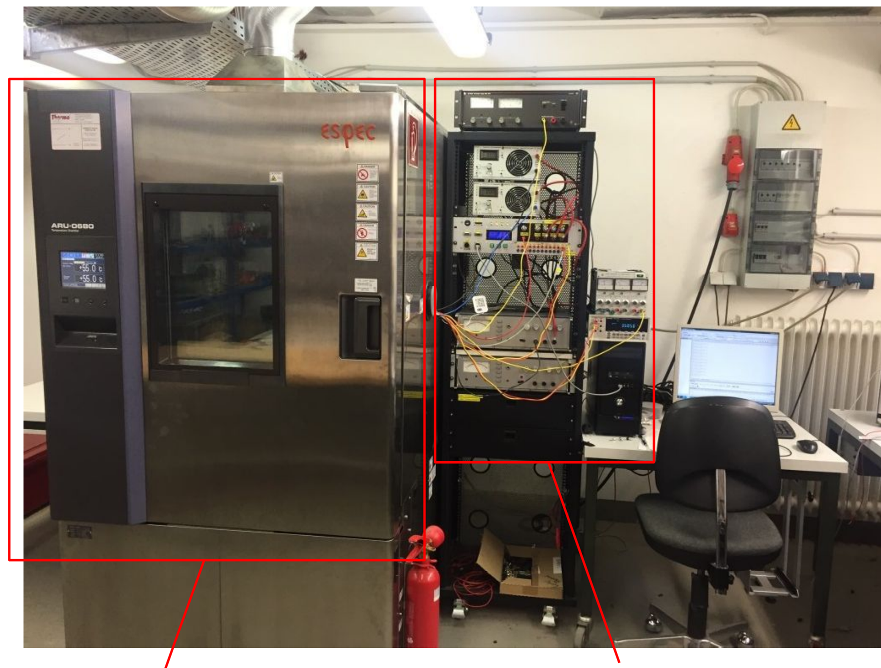


Abbildung D.1.: Aufbau zur künstlichen Batteriezellen-Alterung

Für die Vermessung der Zelle wurde diese einmal pro Woche aus dem Temperaturschrank entnommen damit diese auf Zimmertemperatur abkühlen konnte. Anschließen wurde die Zelle vermessen. Es wurde darauf geachtet, dass die Zelle bei der Vermessung den annähernd selben Ladungszustand hatte. Abbildung D.2 zeigt die Ergebnisse der sieben Wochen langen künstlichen Alterung.

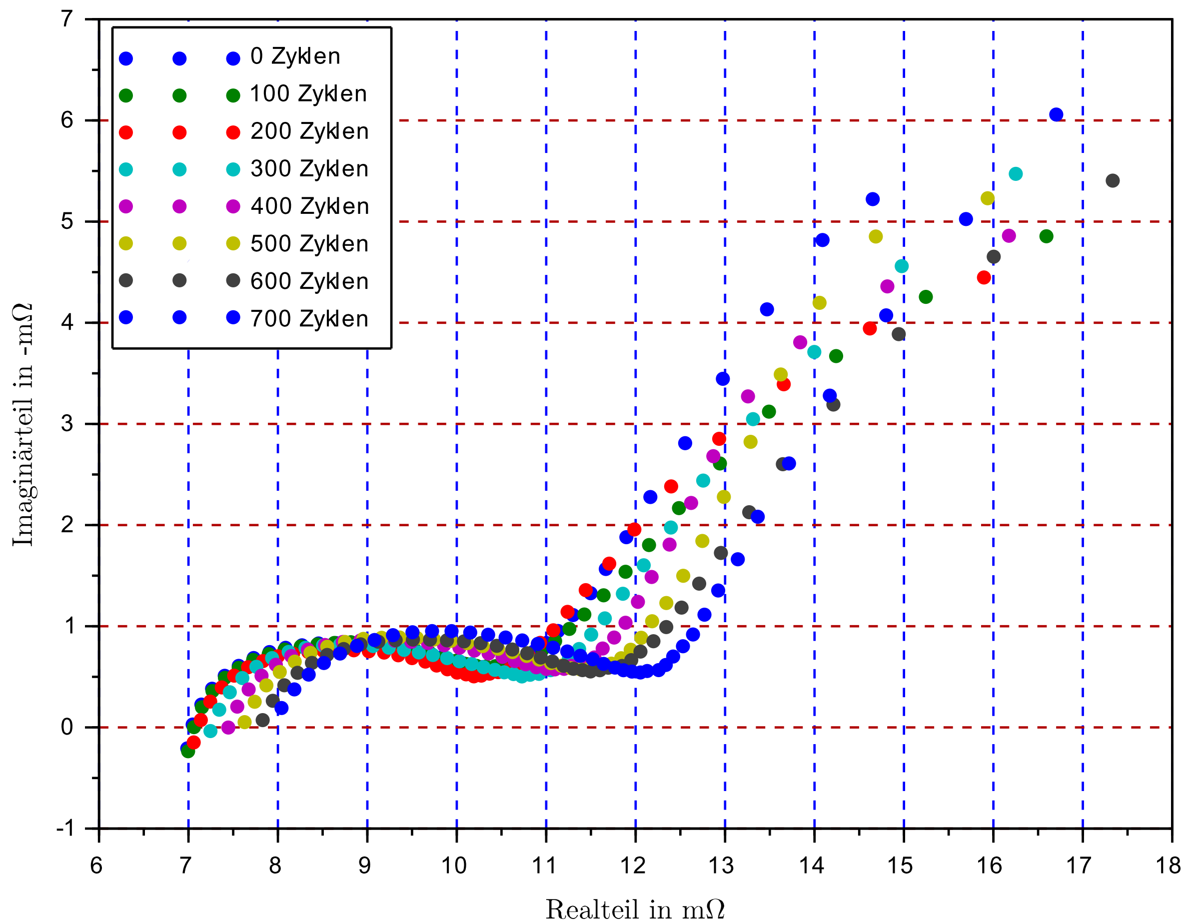


Abbildung D.2.: Künstliche Alterung einer $LiFePO_4$-Zelle

Dabei ist zu sehen, dass sich die Impedanzkurve mit zunehmender Alterung nach rechts bewegt. 100 Zyklen entsprechen dabei einer Woche der künstlichen Alterung.

E. Impedanzmessung

Ladungsabhängige Impedanzmessungen

Die Abbildung E.1 zeigt die vollständige Aufnahme der ladungsabhängigen Impedanzmessungen. Diese Messungen wurden mittels des FuelCon TrueData-EIS aufgenommen. Dabei wurden jeweils 40 Frequenzpunkte zwischen 3.000 Hz und 10 mHz vermessen.

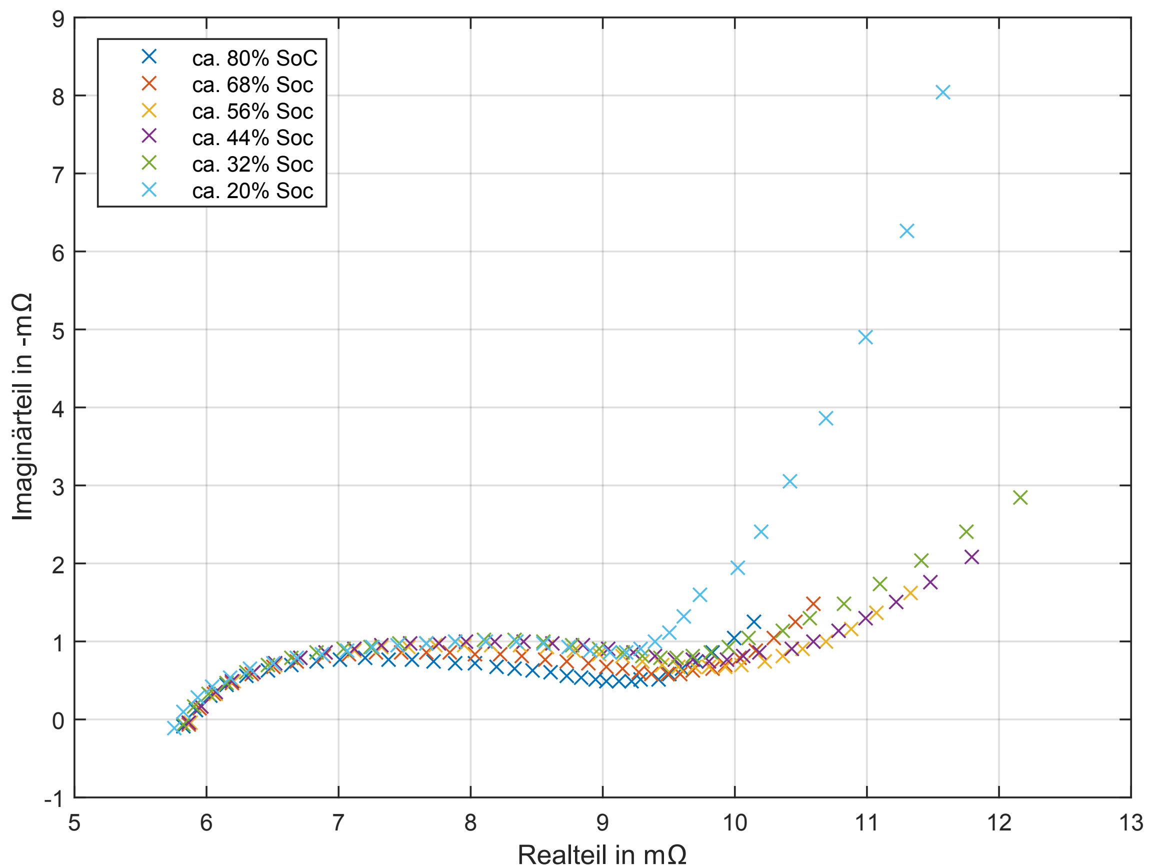


Abbildung E.1.: Ladungsabhängige Impedanzmessungen mit FuelCon TrueData-EIS

Abbildung E.2 zeigt die Vergleichsmessung zur ladungsabhängigen Impedanzmessung mit dem in dieser Arbeit entwickelten Messsystem.

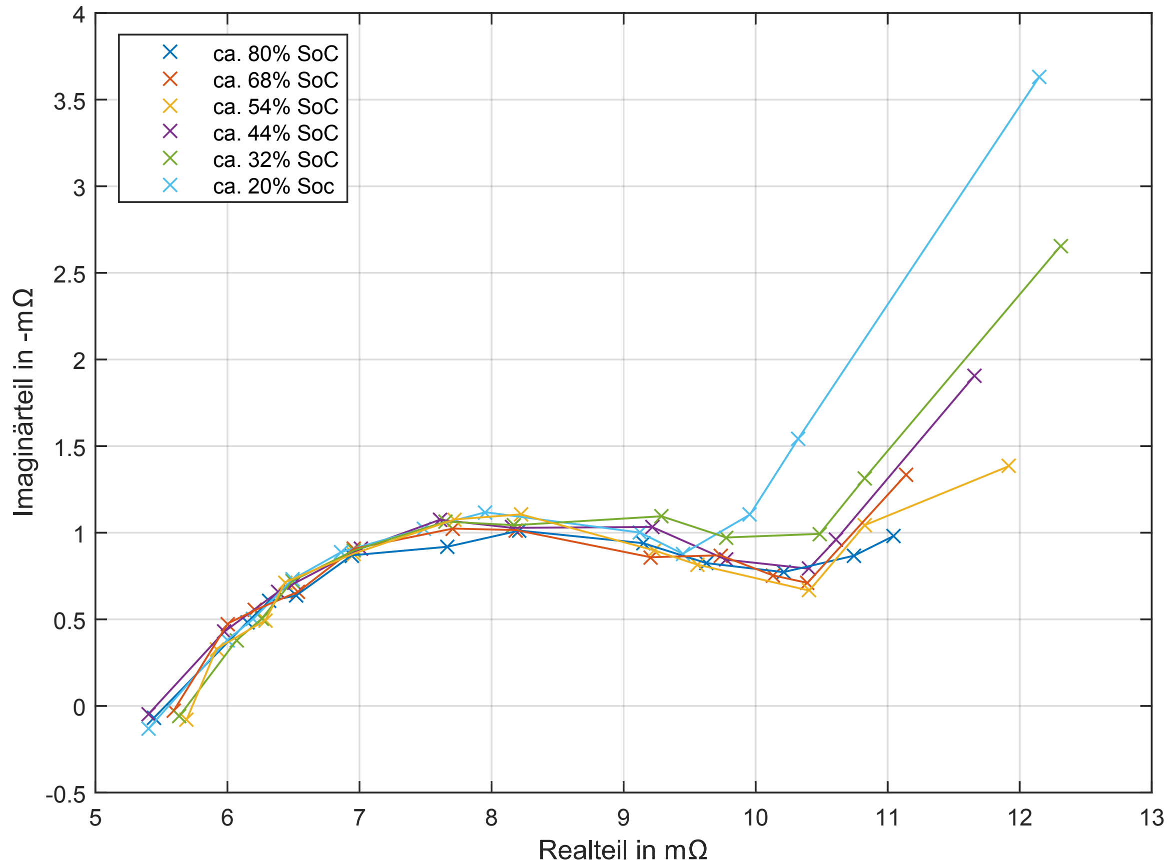


Abbildung E.2.: Ladungsabhängige Impedanzmessungen mit Zellsensoren und Batteriesteuergerät

Impedanzmessungen an ECC-LFPP 45 Ah Zellen
Aufgenommen wurden diese Messungen mit dem FuelCon TrueData-EIS mit 40 Messpunkten zwischen 3 kHz und 10 mHz.

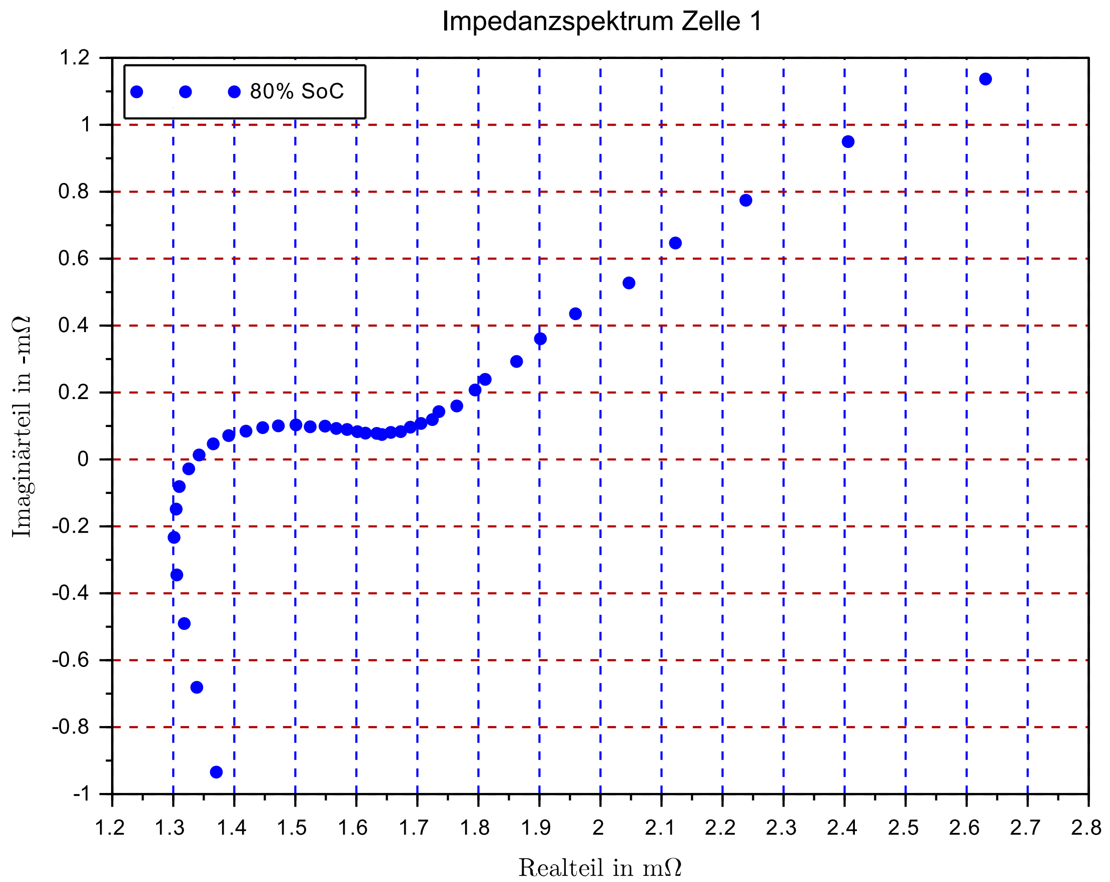


Abbildung E.3.: Impedanzmessungen an ECC-LFPP 45 Ah Zelle 1

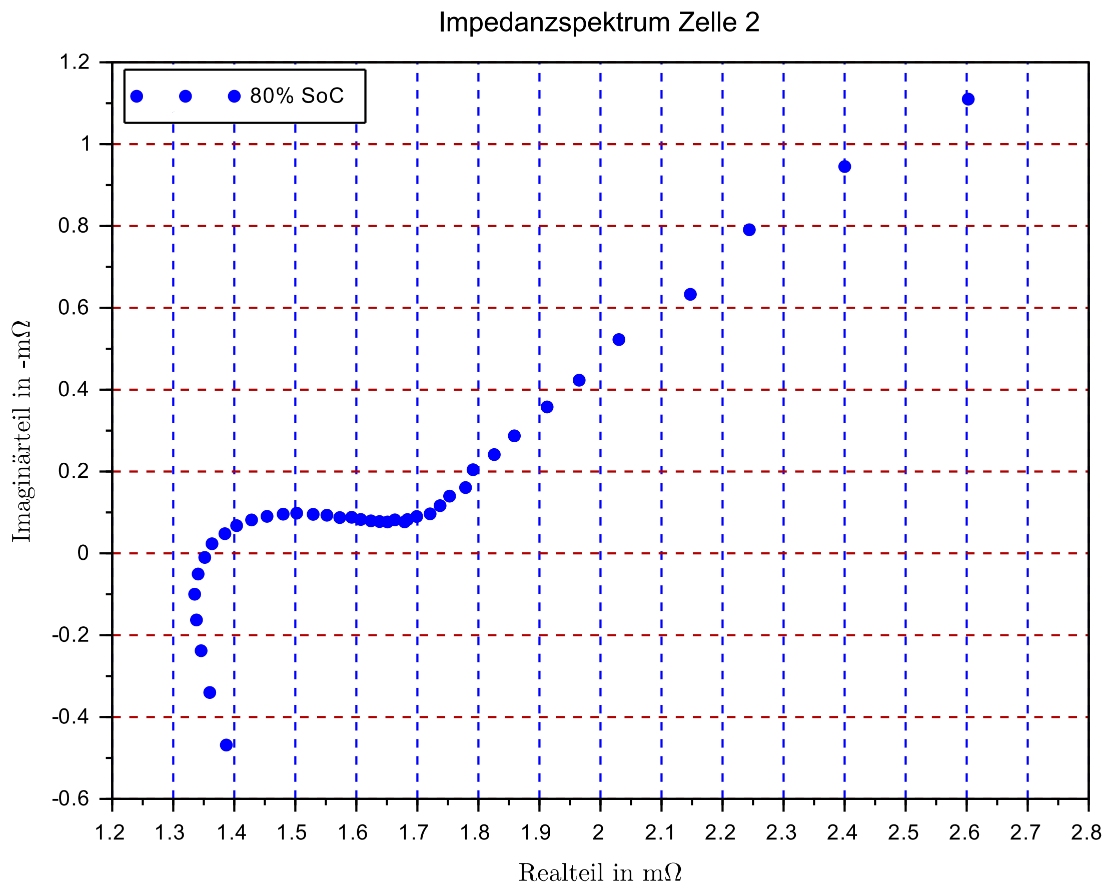


Abbildung E.4.: Impedanzmessungen an ECC-LFPP 45 Ah Zelle 2

F. Befehlsübersicht Batteriesteuergerät

Dieser Abschnitt soll zeigen, mit welchen Befehlen sich das Batteriesteuergerät und die Zellsensoren bedienen lassen.

Das Batteriesteuergerät lässt sich unter den folgenden Einstellungen mit einem Computer verbinden:

- 56.000 Baud
- 8 Data Bit
- 1 Stop Bit
- keine Paritätsbits
- Zeilenumbruch mit CR+LF

Zu beachten ist bei der Verbindung, dass das Batteriesteuergerät zunächst mit der Geschwindigkeit von 9.600 Baud verbunden werden muss, bevor auf 56.000 Baud umgestellt werden kann. Eine direkte Verbindung mit 56.000 Baud ist nicht möglich.

Im Folgenden sind die einzelnen Befehle und die dazugehörigen Parameter aufgeführt.

Einzelne Spannungsdaten abrufen
Um die einzelnen Spannungsdaten eines bestimmten Zellsensors abzurufen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

$$\text{get_voltage(HEX)} \tag{F.1}$$

HEX steht dabei für die Adresse des Zellsensors. So ist für den Spannungsabruf von Zellsensor Nr. 0x01, get_voltage(01) einzugeben. Dieser antwortet nach erfolgreicher Übertragung mit dem aufgenommenen Spannungswert.

Burstmessung starten
Um eine Burstmessung zu starten, ist der folgende Befehl einzugeben:

$$\text{burst(INT1,INT2)} \tag{F.2}$$

Dabei steht INT1 für die Abtastfrequenz. INT2 gibt die Anzahl der aufzunehmenden Samples an.

Burstmessung abrufen
Um die Rohdaten der Burstmessung abzurufen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

$$\text{get_burst(HEX)} \tag{F.3}$$

HEX steht dabei für die Adresse des Zellsensors. Als Rückgabewert werden zuerst die aufgenommenen Messdaten des entsprechenden Zellsensors ausgegeben. Der letzte Wert der Übertragung gibt den HEX-Wert des Verstärkungs-Rheostaten an, der bei dieser Messung verwendet wurde. Anschließend werden die Roh-Daten der Strommessung übertragen.

Aktuelle Konfiguration des Zellsensor oder des Batteriesteuergerät abrufen
Um die aktuelle Konfiguration des Zellsensor oder des Batteriesteuergeräts abzurufen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

$$\text{get_config(HEX)} \tag{F.4}$$

Rückgabewert ist die aktuelle Konfiguration des Zellsensors bzw. des Batteriesterngeräts. Für den Abruf der Batteriesteuergerät-Konfiguration wird als HEX-Wert 00 eingegeben. Ansonsten die Adresse des Zellsensors.

HEX Werte senden
Diese Funktion sendet vier verschiedene HEX-Werte als Broadcast-Message und wurde für Testzwecke implementiert.

$$\text{send_hex(HEX,HEX,HEX,HEX)} \tag{F.5}$$

Dynamische Laufzeitmessung durchführen
Um die dynamische Laufzeitmessung durchführen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

$$\text{cali_burst(HEX)} \tag{F.6}$$

HEX steht dabei für die Adresse des Zellsensors. Als Rückgabewert wird zunächst die gemessene Zeit des Batteriesteuergerät zurückgegeben, anschließend die Zeit des Zellsensors.

DCO-Takt des Zellsensor kalibrieren
Um den DCO-Takt des Zellsensors zu kalibrieren, ist der folgende Befehl an das Batteriesteuergerät zu senden:

$$\text{cali_clk(HEX)} \tag{F.7}$$

HEX steht dabei für die Adresse des Zellsensors. Die Kalibrierung wird solange durchgeführt, bis bis der DCO-Takt des Zellsensor innerhalb der vorgegeben Toleranz liegt.

Analoge Vorverarbeitung manuell einstellen
Um die analoge Vorverarbeitung manuell einzustellen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

set_ad5270_zs(HEX1,HEX2,HEX3) (F.8)

HEX steht dabei für die Adresse des Zellsensors. HEX2 gibt den einzustellenden Wert für den Rheostaten in der Subtrahierschaltung an. HEX3 gibt den einzustellenden Wert für den Rheostaten in der Verstärkerschaltung an.

Analoge Vorverarbeitung automatisch einstellen
Um die analoge Vorverarbeitung automatisch einzustellen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

set_preprocessing(HEX) (F.9)

HEX steht dabei für die Adresse des Zellsensors.

DFT-Werte der Burstmessung abrufen
Um die DFT-Werte der Burstmessung abrufen, ist der folgende Befehl an das Batteriesteuergerät zu senden:

set_goertzel(HEX,INT1,INT2,INT3) (F.10)

HEX steht dabei für die Adresse des Zellsensors. INT1 steht für die Anzahl, der für die Berechnung verwendeten Perioden. INT2 steht für die bei der Messung verwendeten Burstfrequenz. INT3 steht für die bei der Messung verwendeten Messfrequenz.

Tabelle F.1.: Befehlsübersicht für das Batteriesteuergerät

Befehl	**Übergabeparameter**	**Übergabetyp**	
help			help
get_voltage	ZS Adresse	HEX	get_voltage(HEX)
burst	Burstfrequenz und Anzahl Sample	Integer	burst(int,int)
get_burst	ZS Adresse	HEX	get_burst(HEX)
get_config	Adresse	HEX	get_config(HEX)
send_hex	Vier HEX Werte	HEX	send_hex(HEX,HEX,HEX,HEX)
set_brate	Übertragungsrate	Integer	set_brate(int)
cali_burst	ZS Adresse	HEX	cali_burst(HEX)
cali_clk	ZS Adresse	HEX	cali_clk(HEX)
set_ad5270_zs	ZS Adresse und Werte des Rheostaten	HEX	set_ad5270_zs(HEX,HEX,HEX)
get_goertzel	Senden der DFT Werte	HEX und Integer	get_goertzel(HEX,int,int,int)

G. Befestigungsadapter

Zur mechanischen und elektrischen Verbindung des Zellsensors mit einer ECC-LFPP 45Ah $LiFePO_4$-Zelle wurde ein Adapter entworfen. Konstruiert wurde der Befestigungsadapter von M. Eng. Christian Starken. Abbildung G.1 zeigt den Befestigungsadapter.

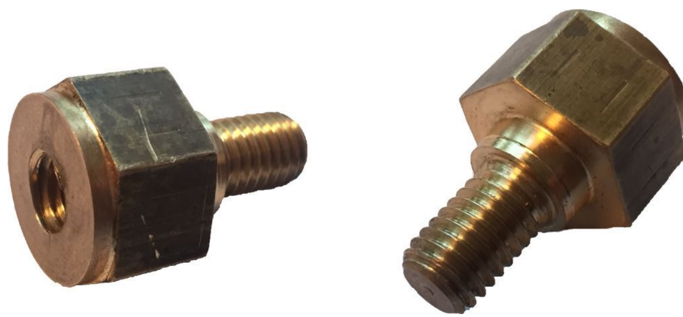
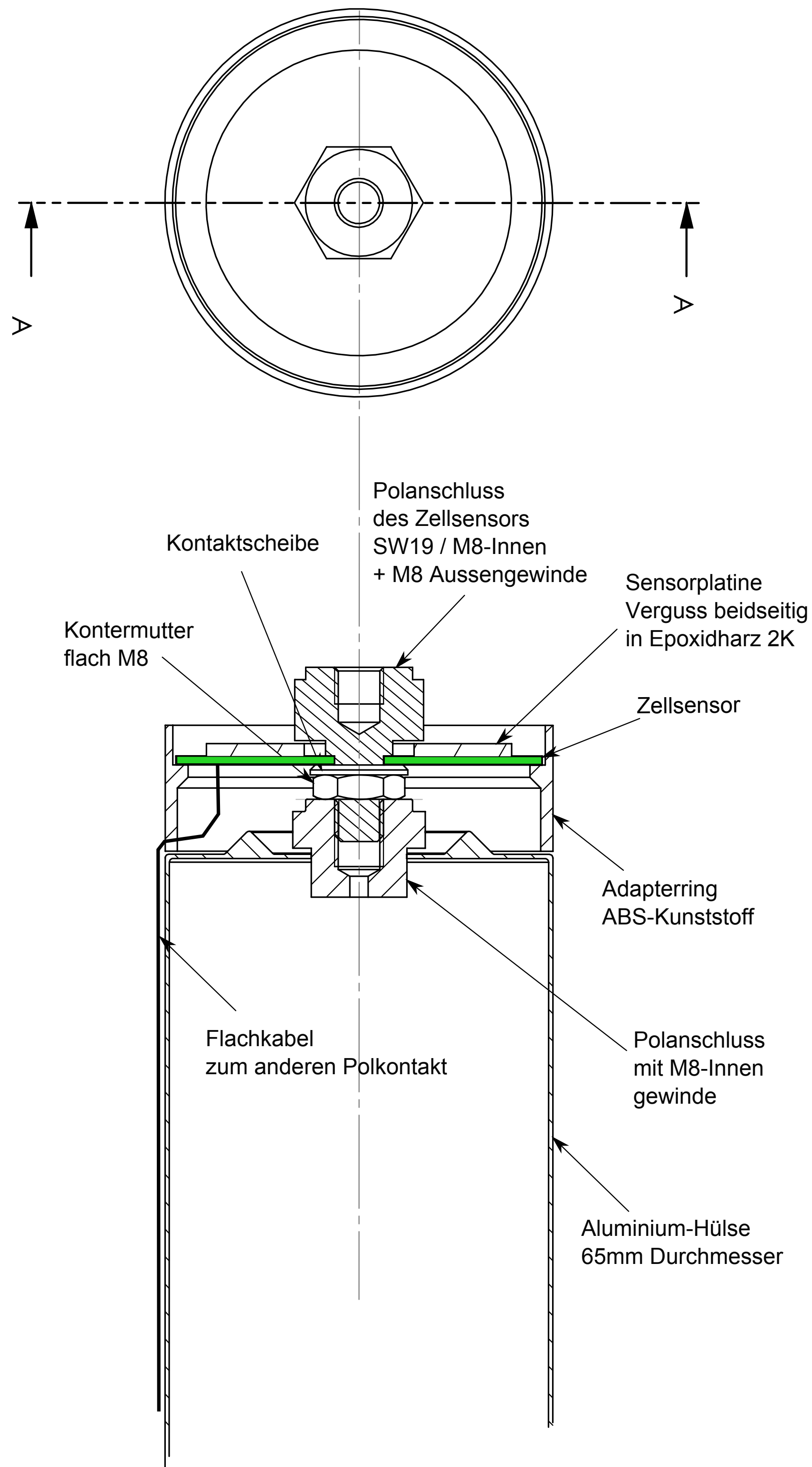
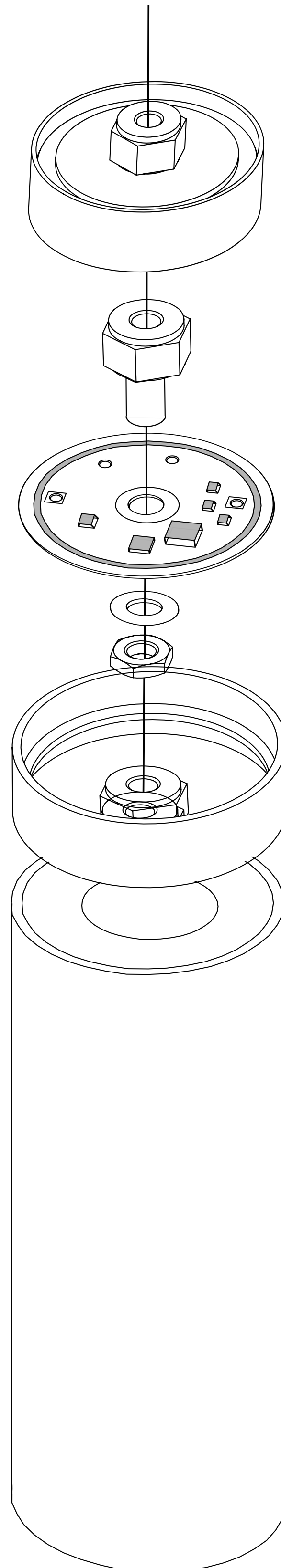


Abbildung G.1.: Befestigungsadapter

In der folgenden Abbildung G.2 ist der montierte und vergossene Zellsensor auf einer ECC-LFPP 45Ah $LiFePO_4$-Zelle zu sehen.

Abbildung G.2.: Montierter und vergossener Zellsensor auf einer ECC-LFPP 45Ah $LiFePO_4$-Zelle


A
A
Polanschluss
des Zellsensors
SW19 / M8-Innen
+ M8 Aussengewinde
Kontaktscheibe
Sensorplatine
Verguss beidseitig
in Epoxidharz 2K
Kontermutter
flach M8
Zellsensor
Adapterring
ABS-Kunststoff
Flachkabel
zum anderen Polkontakt
Polanschluss
mit M8-Innen
gewinde
Aluminium-Hülse
65mm Durchmesser



H. PC-lauffähiges Goertzel-Testprogramm

Zur ersten Verifizierung der entwickelten Goertzel-Filterstruktur, wurde zunächst ein PC-Lauffähiges Programm geschrieben. Ein Grund für das spezielle Programm ist, dass es auf dem Mikrocontroller sehr aufwendig ist, eine große Anzahl verschiedener Messdaten zu implementieren und diese zu testen, bedingt durch den begrenzten Speicher des Mikrocontrollers. Aufgebaut wurde die Filterstruktur wie in Abschnitt 5.3 beschrieben.

Das Testprogramm mit den dazugehörigen Messdaten befindet sich auch auf der beiliegenden CD. Der Quellcode des Goertzel Testprogramms findet sich im Anschluss der Abbildung H.1, welches die Konsolenausgabe des zeigt.

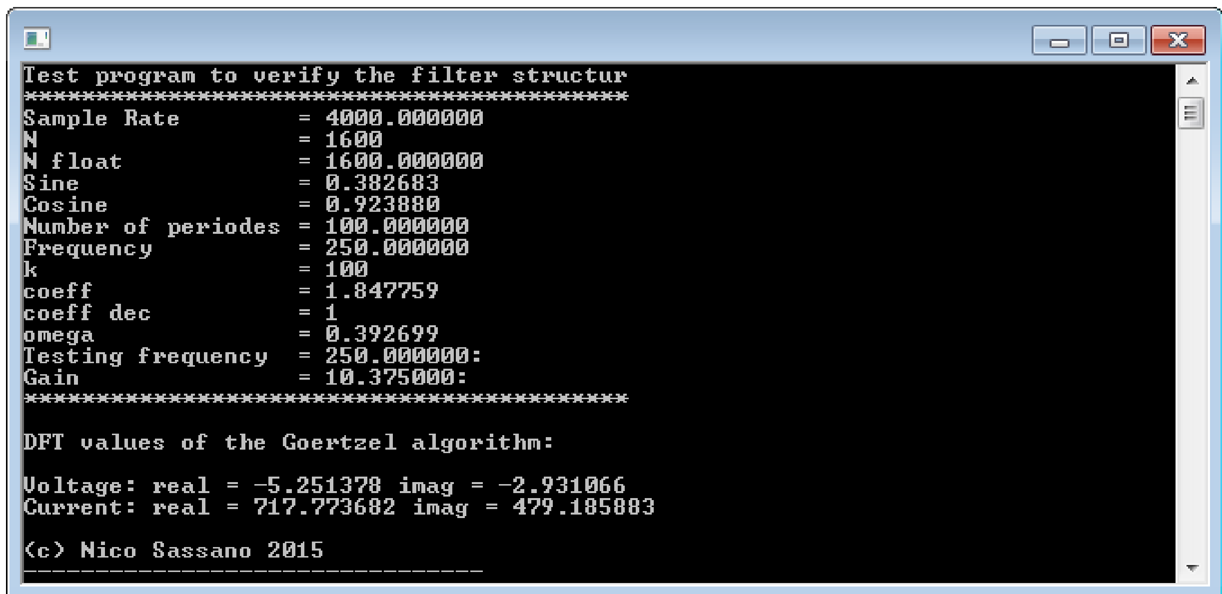


Abbildung H.1.: PC lauffähiges Goertzel-Testprogramm

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define Fs   4000.0
5  #define f    250.0
6  int u_int[1992] = { Daten auf beiliegender CD };
7  int i_int[2000] = { Daten auf beiliegender CD };
8
9  float u[2000];
10 float i[2000];
11
12 int main(void) {
13
14   int index;
15   int k;
16   int N_int;
17   int coeff_dec = 0;
18
19   float real_u;
20   float imag_u;
21   float real_i;
22   float imag_i;
23   float bn_0 = 0;
24   float bn_1 = 0;
25   float bn_2 = 0;
26   float N_float;
27   float omega;
28   float sine;
29   float cosine;
30   float coeff;
31   float steps;
32   float  m;
33   float  Np;
34
35  m = 100;               // Anzahl der Perioden pro Berechnung
36  Np = Fs/f;             // Np Anzahl Abtastwerte pro Periode
37   N_float = m * Np;     // Anzahl der Abtastpunke für die Berechnung
38   N_int = (int) N_float;
39
40   //0.00080586
41  for(index = 0;index < N_int+1;index++) {
42     i[index]  = ((3.3/4095)*i_int[index])*10;
43   }
44
45   float gain = 0.0;
46   gain = (1+((96.0*(100000.0/1024.0)+000)/ 1000));
47
48   for(index = 0;index < N_int+1;index++) {
49     u[index]  = ((2.50/4095)*u_int[index])/gain;
50   }
51
52   //*** Begin the Goertzel algorithm **//
53   k = (int) (0.5 + ((N_float * f) / Fs));
54   omega = (2.0 * M_PI * k) / N_float;
55   sine = sin(omega);
56   cosine = cos(omega);
57   coeff = 2.0 * cosine;
58   coeff_dec = (int) coeff;
59
60   for (index = 0; index < N_int + 1 ; index++) {
61     if(index == N_int) {
62         bn_0 = coeff * bn_1 - bn_2 +  0;
63     } else {
64         bn_0 = coeff * bn_1 - bn_2 +  u[index];
65     }
66     bn_2 = bn_1;
67     bn_1 = bn_0;
68   }
69
70   real_u = (bn_1 - bn_2 * cosine);
71   imag_u = (bn_2 * sine);
72
73   bn_0 = 0;
74   bn_1 = 0;
75   bn_2 = 0;
76
77   for (index = 0; index < N_int + 1 ; index++) {
78     if(index == N_int) {
79         bn_0 = coeff * bn_1 - bn_2 +  0;
80     } else {
81         bn_0 = coeff * bn_1 - bn_2 + (float) i[index];
82     }
83
84     bn_2 = bn_1;
85     bn_1 = bn_0;
```

```
86      }
87
88      real_i = (bn_1 - bn_2 * cosine);
89      imag_i = (bn_2 * sine);
90
91      //*** End the Goertzel algorithm **//
92      printf("Test program to verify the filter structur\n");
93      printf("*****************************************\n");
94      printf("Sample Rate         = %f\n", Fs);
95      printf("N                   = %d\n", N_int);
96      printf("N float             = %f\n", N_float);
97      printf("Sine                = %f\n", sine);
98      printf("Cosine              = %f\n", cosine);
99      printf("Number of periodes  = %f\n", m);
100     printf("Frequency           = %f\n", f);
101     printf("k                   = %d\n", k);
102     printf("coeff               = %f\n",coeff);
103     printf("coeff dec           = %d\n",coeff_dec);
104     printf("omega               = %f\n",omega);
105     printf("Testing frequency   = %f:\n", f);
106     printf("Gain                = %f:\n", gain);
107     printf("*****************************************\n");
108
109     printf("\nDFT values of the Goertzel algorithm:\n");
110     printf("\n");
111     printf("Voltage: real = %f imag = %f\n", real_u, imag_u);
112     printf("Current: real = %f imag = %f\n", real_i, imag_i);
113     printf("\n(c) Nico Sassano 2015");
114
115     return 0;
116 }
```

I. 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 15. Februar 2015

Masterthesis Nico Sassano

Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen

Motivation

In der Forschungsgruppe 'Batteriesensoren - BATSEN' werden an der HAW Hamburg Sensoren und Verfahren entwickelt, die Aussagen über den Ladezustand und die Alterung von Batterien ermöglichen sollen. Hierzu sollen die drahtlos kommunizierende Sensoren die Spannung und die Temperatur der Batteriezellen messen. Diese Informationen werden drahtlos an ein für die Gesamtbatterie zuständiges Steuergerät übertragen. Auswertealgorithmen kombinieren diese Sensordaten mit einer zentralen Strommessung und schätzen den Batteriezustand.
Aus der Literatur ist bekannt, dass der komplexe Innenwiderstand der Zellen eine differenzierte Aussage über das Verhalten der Batteriezellen zulässt. Hierfür wird das Verfahren der elektrochemischen Impedanzspektroskopie (EIS) eingesetzt. Es hat sich als Messverfahren bei der Entwicklung von neuen Batterie-Technologien etabliert. Die EIS erlaubt separierbare Rückschlüsse auf verschiedene Vorgänge innerhalb der Batterie. Bisher werden für die EIS spezialisierte Laborgeräte eingesetzt, welche für eine Betriebsüberwachung im Einsatzfall im Fahrzeug zu aufwändig sind.
Das Projekt BATSEN strebt an, diesen Aufwand deutlich zu senken. Hierfür sollen die drahtlosen Zellensensoren befähigt werden, die EIS-Messungen zu unterstützen. Ein Fernziel ist die Verwendung nicht nur im Laborgerät sondern als Teil des Batteriemanagementsystems in Elektrofahrzeugen.

Aufgabe

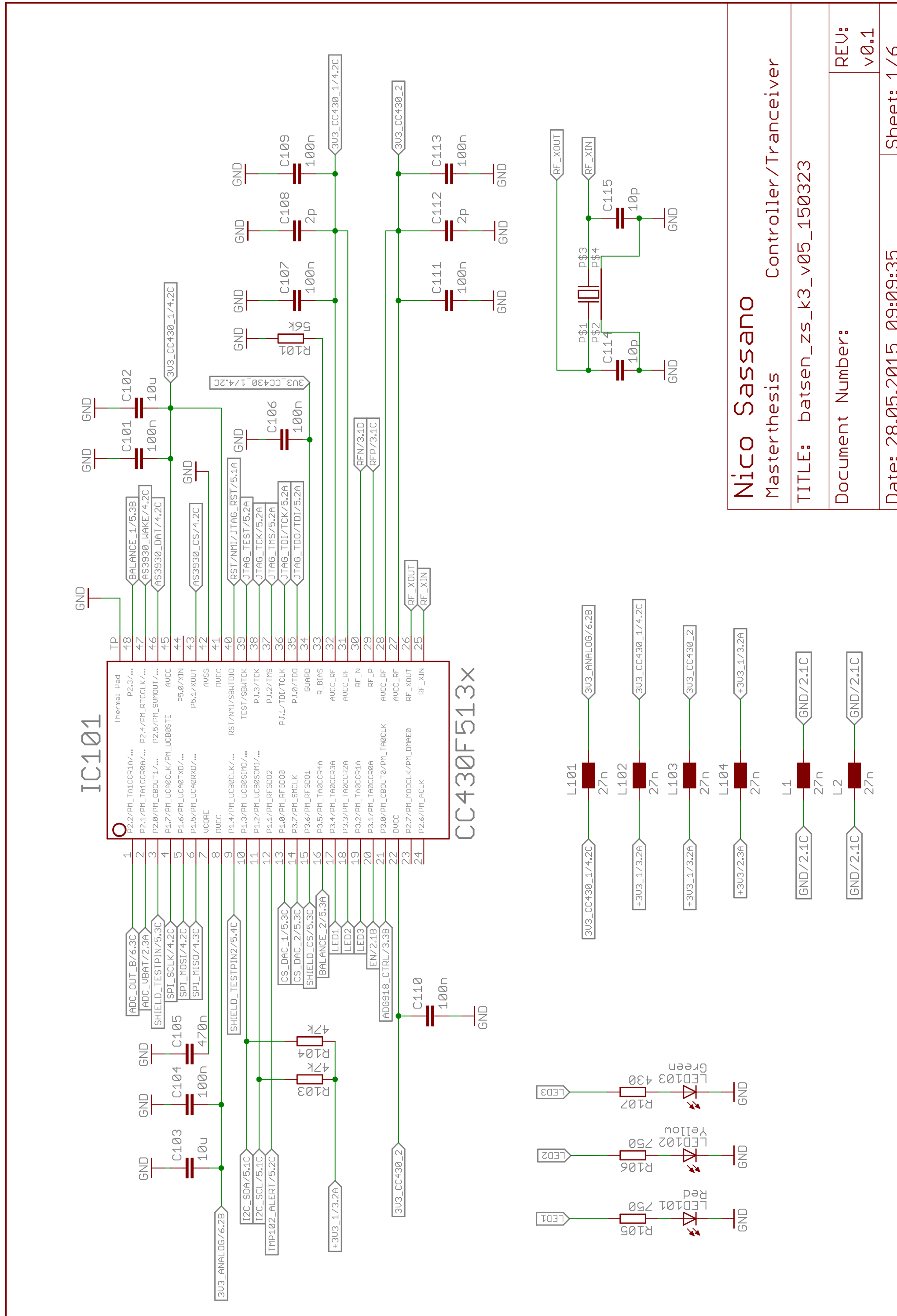
Herr Nico Sassano erhält die Aufgabe, einen bestehenden Zellensensor für die Durchführung der EIS an Batteriezellen zu erweitern.
Er kann dabei auf Vorarbeiten aus der Arbeitsgruppe BATSEN zurückgreifen, sowohl der Entwurf von Zellensensoren als auch Voruntersuchungen und Implementierungsteilschritte für elektrochemischen Impedanzspektroskopie sind bearbeitet worden. Obwohl dort viele Teilsaspekte behandelt wurden, konnte eine vollständige Impedanzspektroskopie noch nicht realisiert werden.
Wichtig ist, dabei vergleichbare Leistungsparameter wie die Laborgeräte zu erfüllen und die Anforderungen für die im Fahrzeug verwendeten großen LiFePo4-Zellen abzudecken. Laut der aktuellen Literatur werden für die EIS Wechselspannungen mit der Frequenz von einigen mHz bis hin von mehreren kHz benutzt. Zudem kann diese Wechselspannung eine Offsetspannung von mehr als 4V und eine sehr kleine Amplitude von wenigen mV besitzen. Der Zellensensor soll in der Lage sein, diese Wechselspannung ausreichend aufgelöst zu messen und zu verarbeiten. Hierzu ist der ADC und/oder die analoge Vorverarbeitung der Sensoren anzupassen. Neben der Entwicklung des Zellensensors, der diese Forderungen erfüllen soll, ist ein leistungsfähiges Batteriesteuergerät zu entwickeln. Dieses Steuergerät soll neben der drahtlosen Kommunikation mit den Zellensensoren auch in der Lage sein, den gesamten Strom, der durch die gesamte Batterie fließt, zu messen. Ein zu lösendes Kernproblem ist die phasenrichtige Messung von

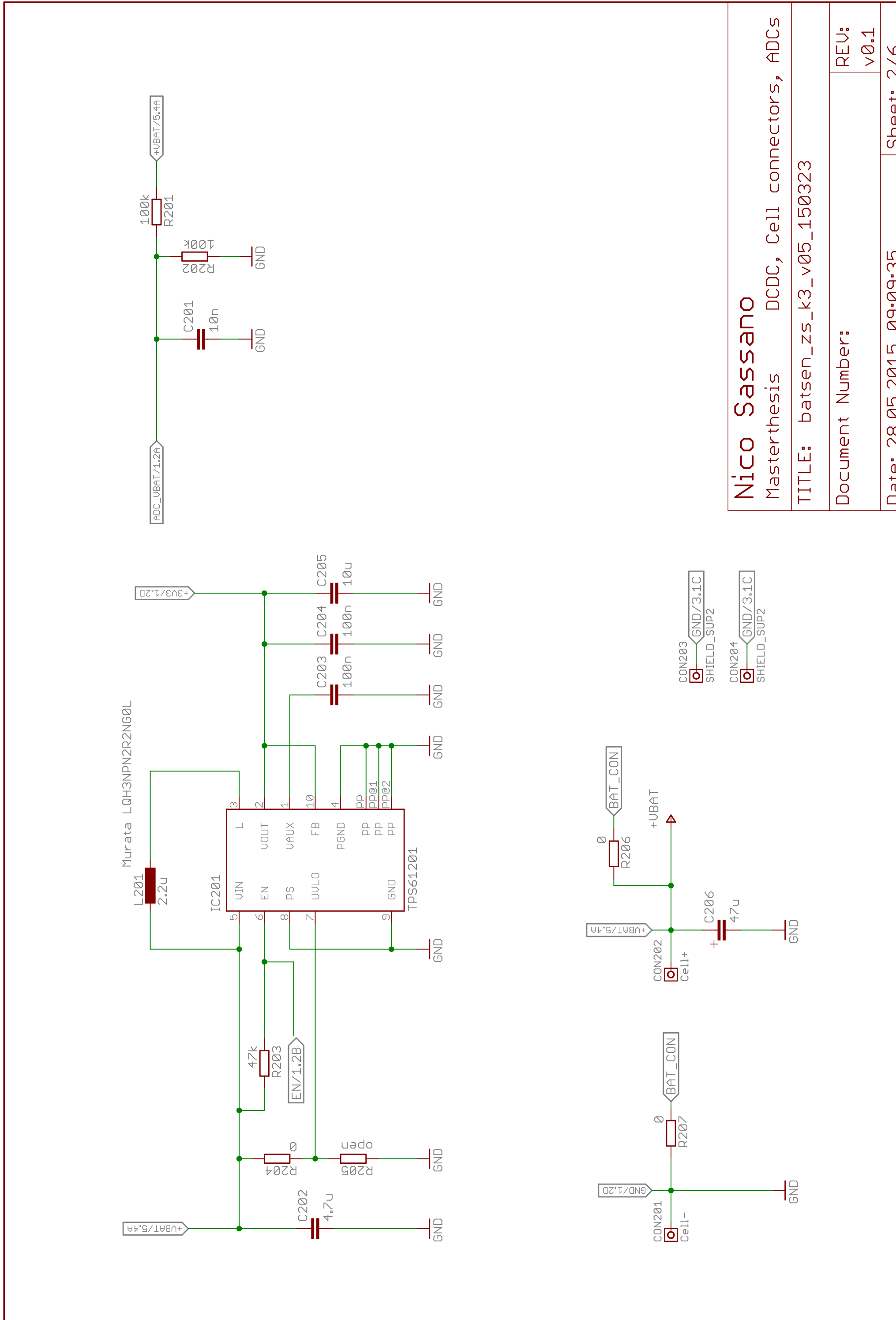
Strom und den Spannungen an allen Batteriezellen. Die dafür notwendige Synchronisierung erfordert ein spezialisiertes Funkprotokoll, das in der Software des Steuergeräetes und der Sensoren zu realisieren ist. Die Anregung ist zu konzipieren. Exemplarische Messungen an LiFePo4 Zellen sollen die Funktionsfähigkeit der implementierten Impedanzspektroskopie bestätigen. Eine Demonstration mit Anwendungsbezug soll möglich werden.
Die Aufgabenstellung umfasst die folgenden Punkte:

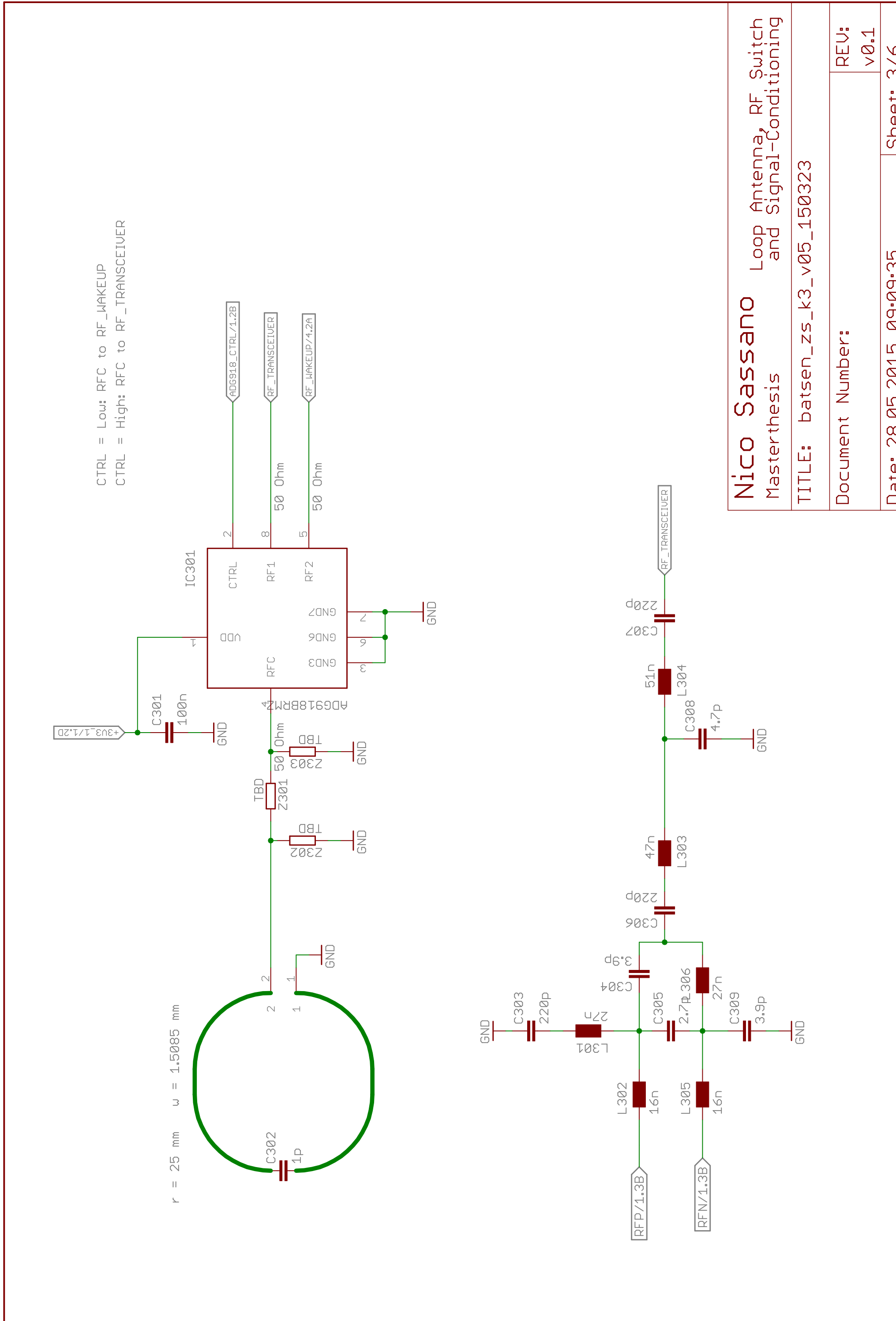
1) **Einführung und Grundlagen**
   - Prinzip der Zellen-Überwachung und des Batteriemanagements
   - Darstellung des aktuellen Standes der Vorarbeiten der Arbeitsgruppe
   - Einführung in Grundlagen der elektrochemischen Impedanzspektroskopie
   - Erfassen der auf den verschiedenen Sensorklassen verfügbaren Messauflösung, der Messraten und der speicherbedingt möglichen Blockgrößen
2) **Analyse und Konzeption**
   - Auswertung der vorhandenen Sensoren und Steuergeräte und des EIS-Laborgerätes
   - Analyse der messtechnischen Anforderungen für die EIS (Messauflösung, Synchronisation, Frequenzbereich) und einer geeeigneten Anregung
   - Lösungserarbeitung bzgl. ADC-Auflösung und/oder Offsetkompensation
   - Konzeption und Strukturierung eines Lösungsvorschlags
3) **Neuentwicklung und Redesign der Hardware**
   - Neuentwicklung eines Batteriesteuergerätes auf Basis eines ARM Cortex M4 Mikrocontrollers
   - Entwicklung einer Transceiverplatine und einer Strommessplatine für das Batteriesteuergerät
   - Erweiterung mit geeigneten ADC auf dem Zellensensor, ggf. mit einer analogen Vorverarbeitung
   - Redesign / Erweiterung der vorhandenen Zellensensoren
   - Entwurf, Aufbau und Inbetriebnahme der entwickelten Hardware
4) **Software-Entwurf u. Implementierung für die Impedanzspektroskopie**
   - Phasenrichtige Aufnahme von Strom und Zellspannungen im Batteriesteuergerät und in den Sensoren sowie der passend koordinierten Ansteuerung der Anregung
   - Softwareentwicklung für das Batteriesteuergerät und Einbindung der Transceiver- und der Strommessplatine sowie des Rechenverfahrens des EIS
   - Softwareerweiterung der Zellensensoren unter Berücksichtigung der begrenzten Ressourcen des Controller
4) **Erprobung und Funktionsnachweis**
   - Erstellen von EIS-Messreihen einer Batterie aus LiFePO4-Zellen über verschiedene SoC und Temperaturen
   - Genauigkeits-, Rausch- und Jitteruntersuchungen
5) **Auswertung und Bewertung der Ergebnisse**
   - Erstellen von EIS-Messreihen einer Batterie aus LiFePO4 -Zellen über verschiedene SoC und Temperaturen
   - Vergleich mit EIS-Messgeräten und der entwickelten Messmethode
   - Bewertung im Hinblick auf die Aussagekraft für Batteriekennwerte
   - Diskussion von Vor- und Nachteilen in Bezug auf die Anwendung

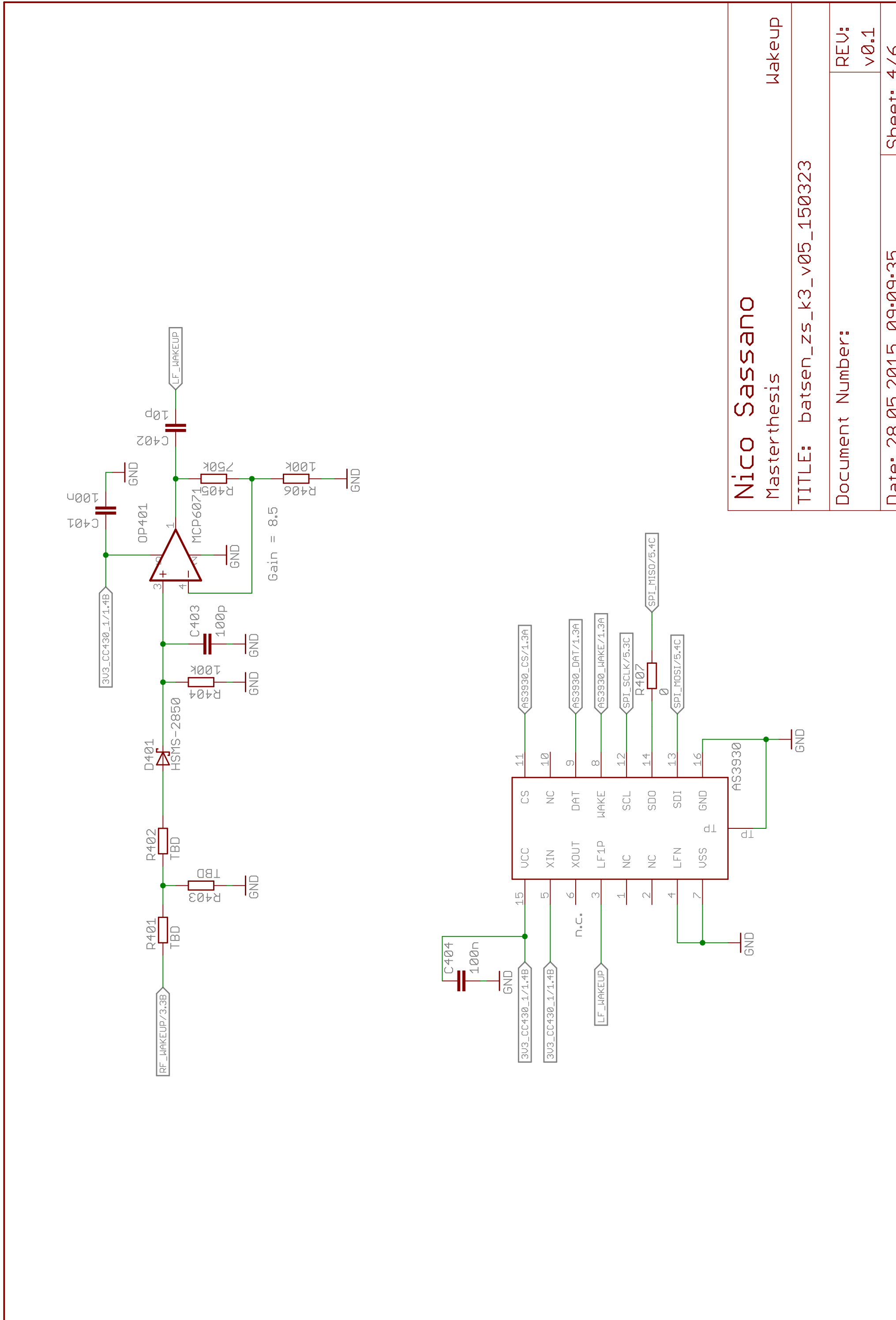
J. Schaltplan

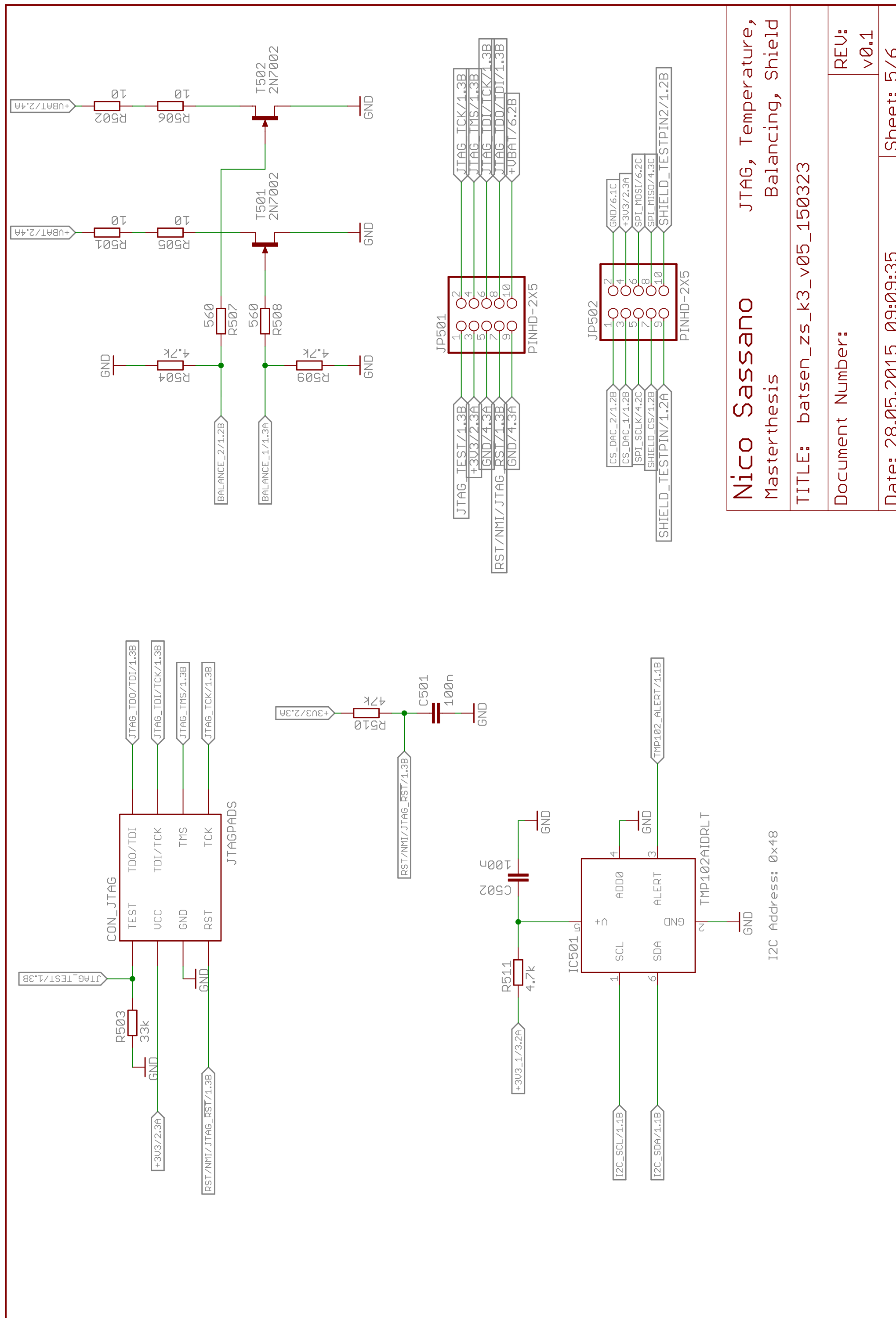
Schaltplan: Zellsensor

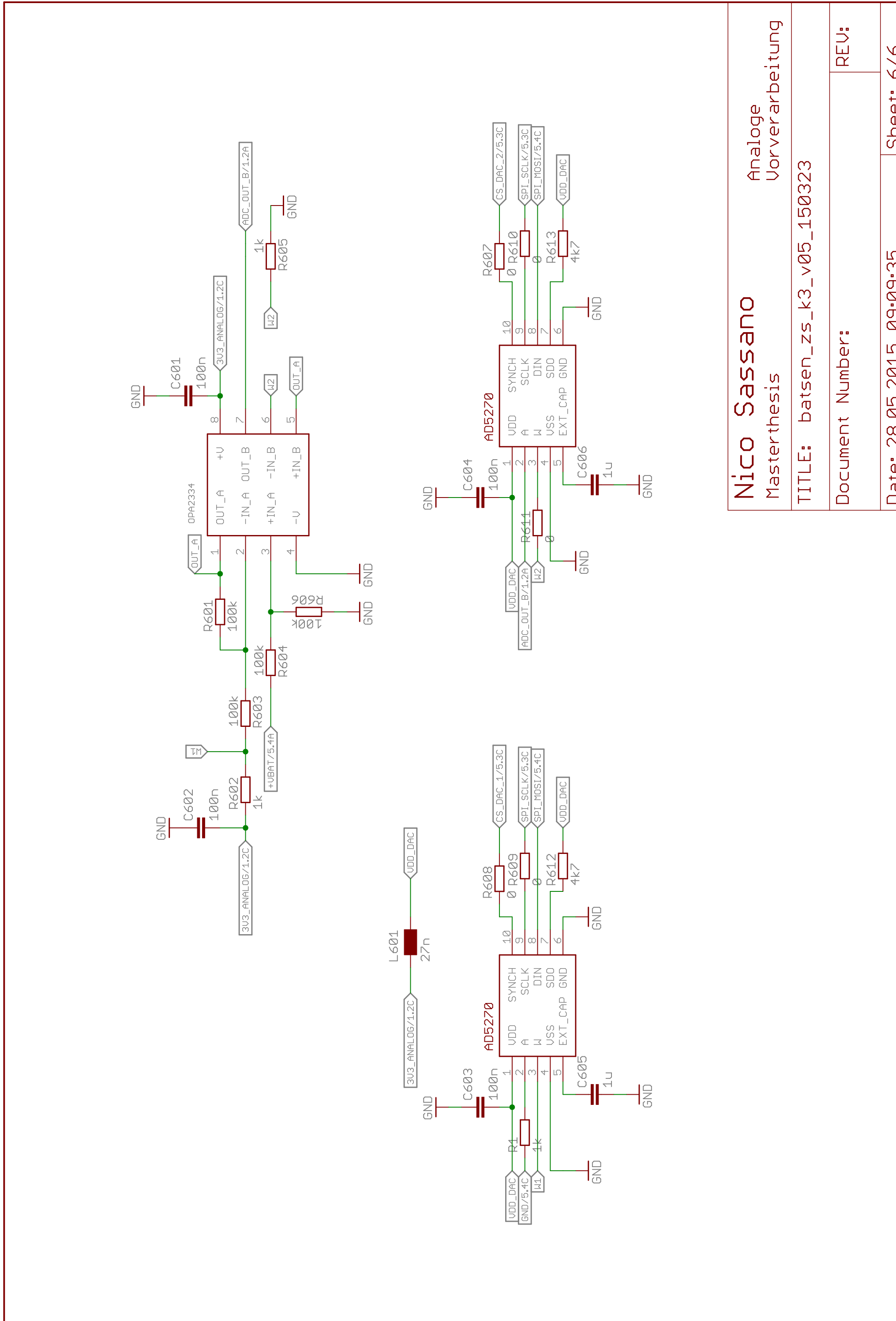



Nico Sassano
Masterthesis
DCDC, Cell connectors, ADCs
TITLE: batsen_zs_k3_v05_150323
Document Number:
REV: v0.1
Date: 28.05.2015 09:09:35
Sheet: 2/6
+VBAT/5.4A
R201
100k
R202
100k
C201
10n
GND
ADC_VBAT/1.2A
+3V3/1.2D
C205
10u
C204
100n
C203
100n
Murata LQH3NPN2R2NG0L
L201
2.2u
IC201
TPS61201
L
VOUT
VAUX
FB
PGND
PP
VIN
EN
PS
UVLO
GND
R203
47k
EN/1.2B
R204
0
R205
open
C202
4.7u
CON203
CON204
GND/3.1C
SHIELD_SUP2
BAT_CON
R206
R207
0
+VBAT
C206
47u
CON202
Cell+
CON201
Cell-
GND/1.2D

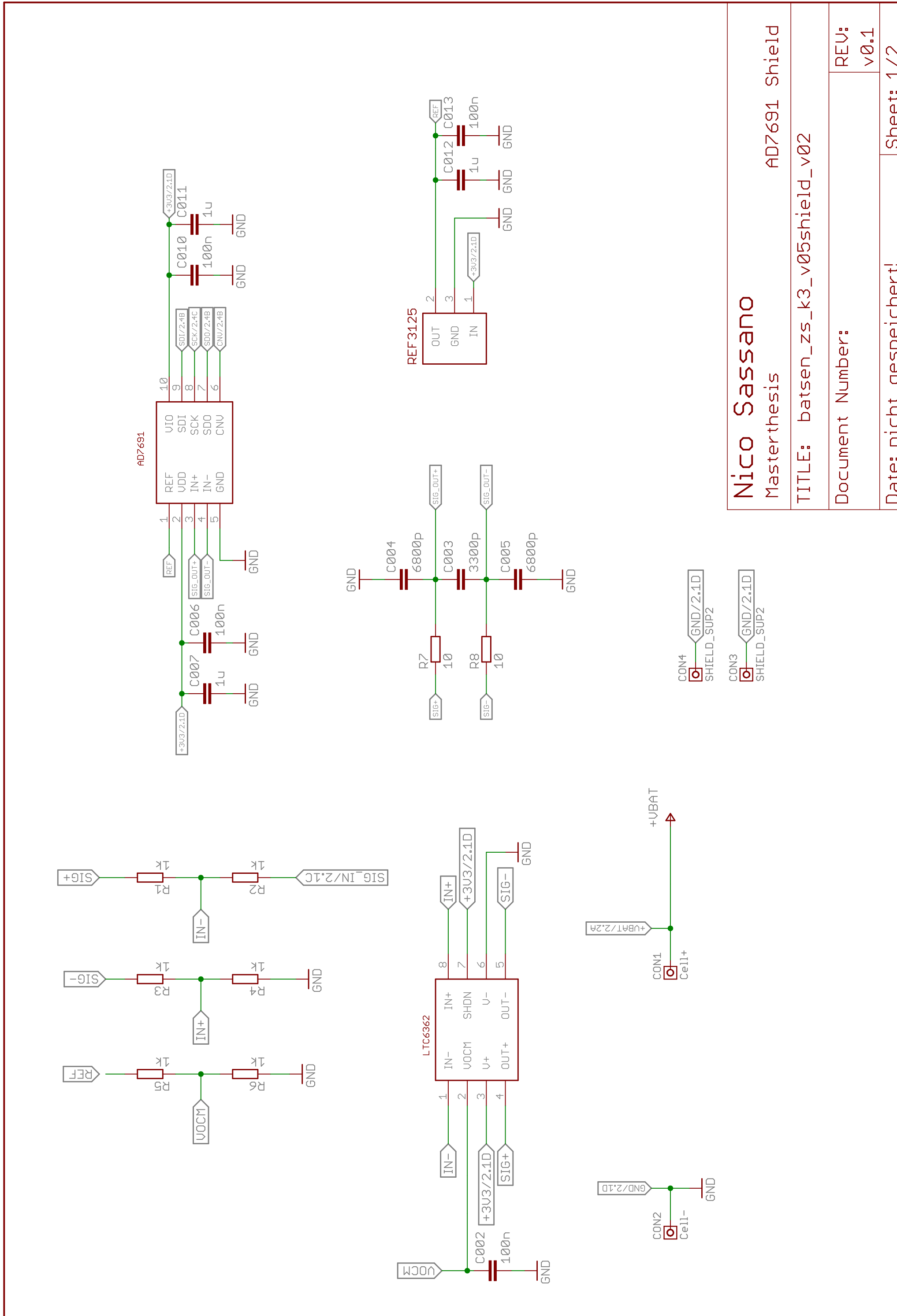

CTRL = Low: RFC to RF_WAKEUP
CTRL = High: RFC to RF_TRANSCEIVER
IC301
ADG918BRMZ
r = 25 mm   w = 1.5085 mm
Nico Sassano
Masterthesis
Loop Antenna, RF Switch
and Signal-Conditioning
TITLE: batsen_zs_k3_v05_150323
Document Number:
REV: v0.1
Date: 28.05.2015 09:09:35
Sheet: 3/6

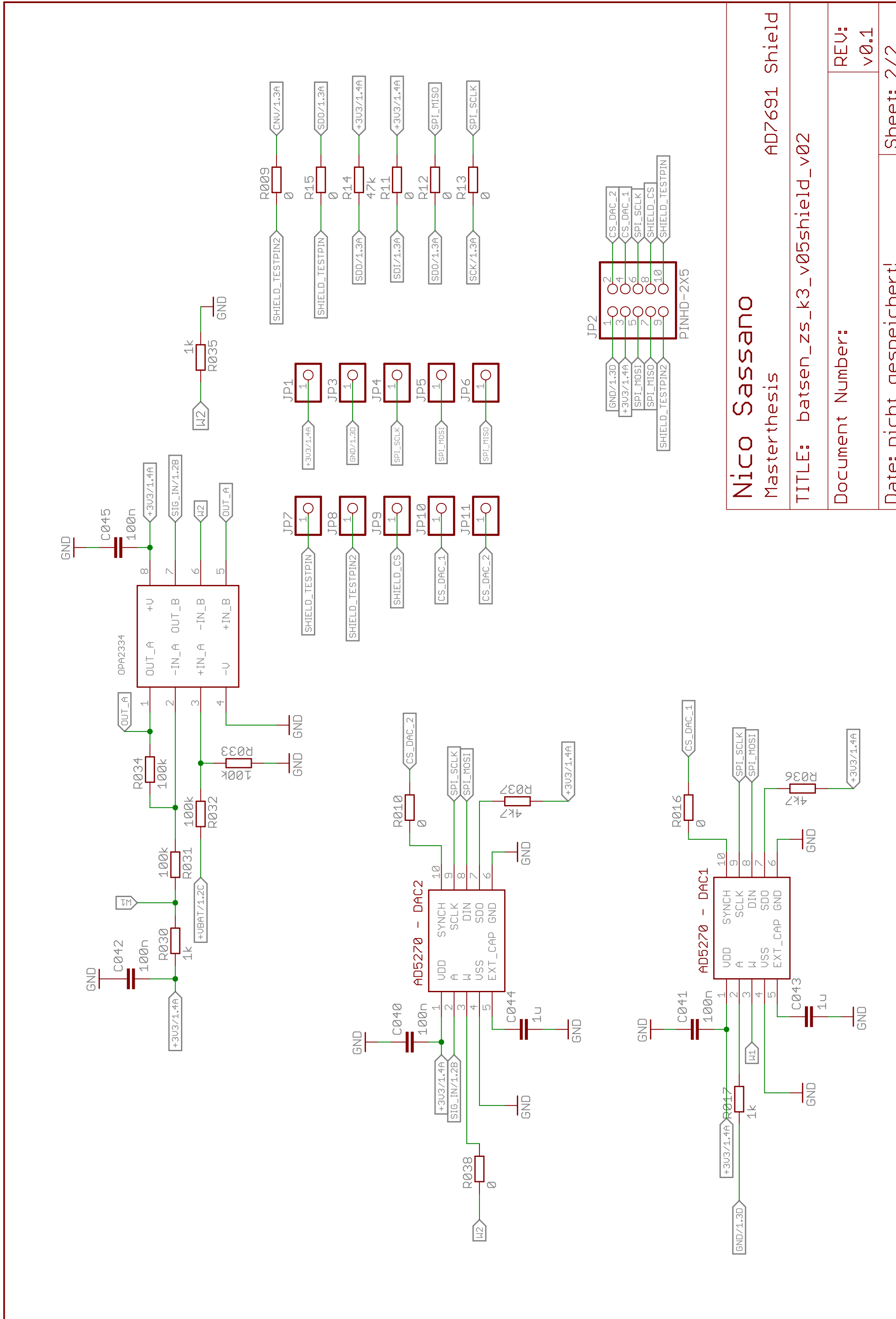

Nico Sassano
Masterthesis
Wakeup
TITLE: batsen_zs_k3_v05_150323
Document Number:
REV: v0.1
Date: 28.05.2015 09:09:35
Sheet: 4/6
Gain = 8.5
AS3930
MCP6071
HSMS-2850


Nico Sassano
Masterthesis
JTAG, Temperature, Balancing, Shield
TITLE: batsen_zs_k3_v05_150323
Document Number:
REV: v0.1
Date: 28.05.2015 09:09:35
Sheet: 5/6
JP501
PINHD-2X5
JP502
PINHD-2X5
T501 2N7002
T502 2N7002
CON_JTAG
JTAGPADS
IC501
TMP102AIDRLT
I2C Address: 0x48
GND

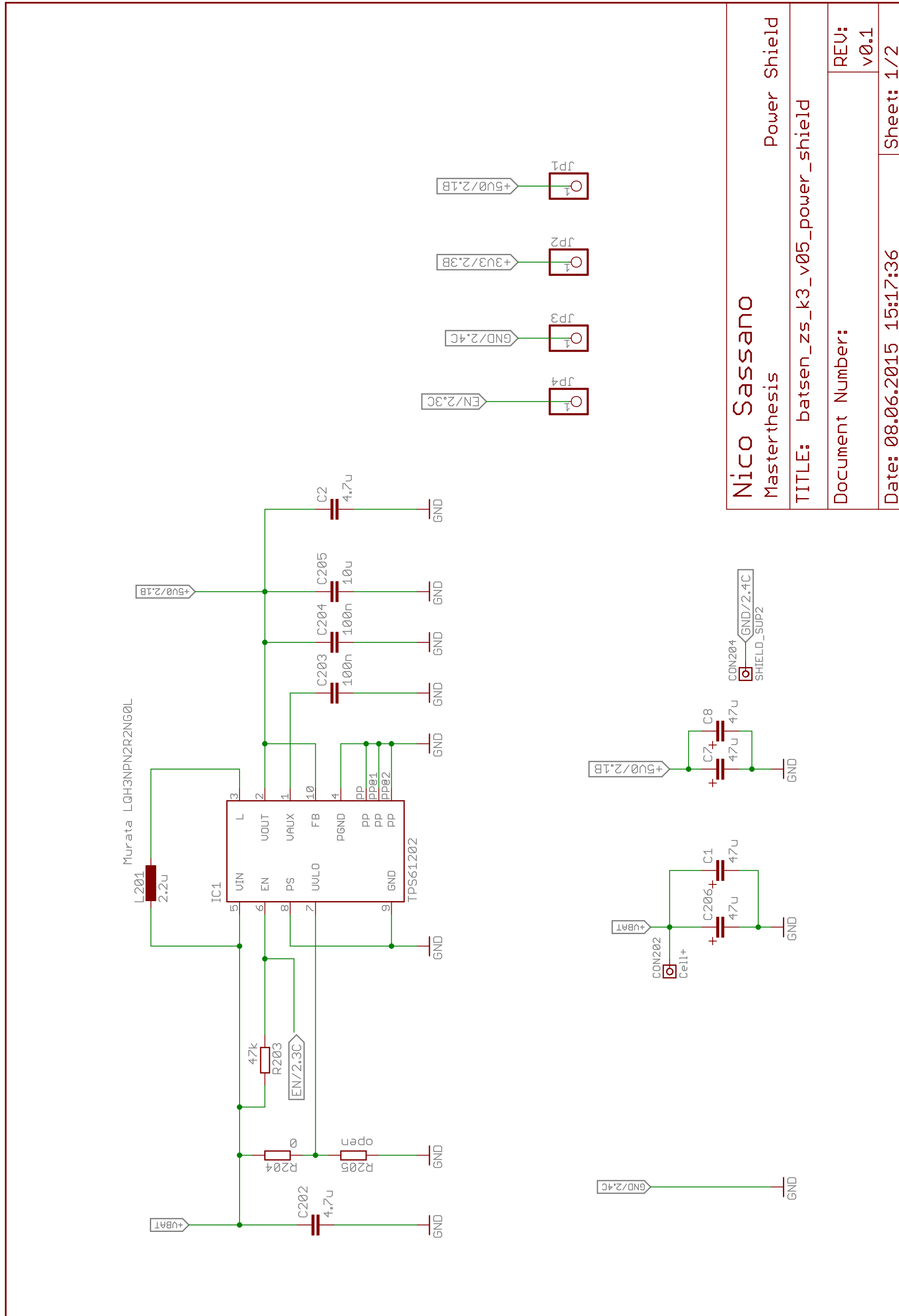

Nico Sassano
Masterthesis
Analoge
Vorverarbeitung
TITLE: batsen_zs_k3_v05_150323
Document Number:
REV:
Date: 28.05.2015 09:09:35
Sheet: 6/6
OPA2334
AD5270
AD5270
L601
27n

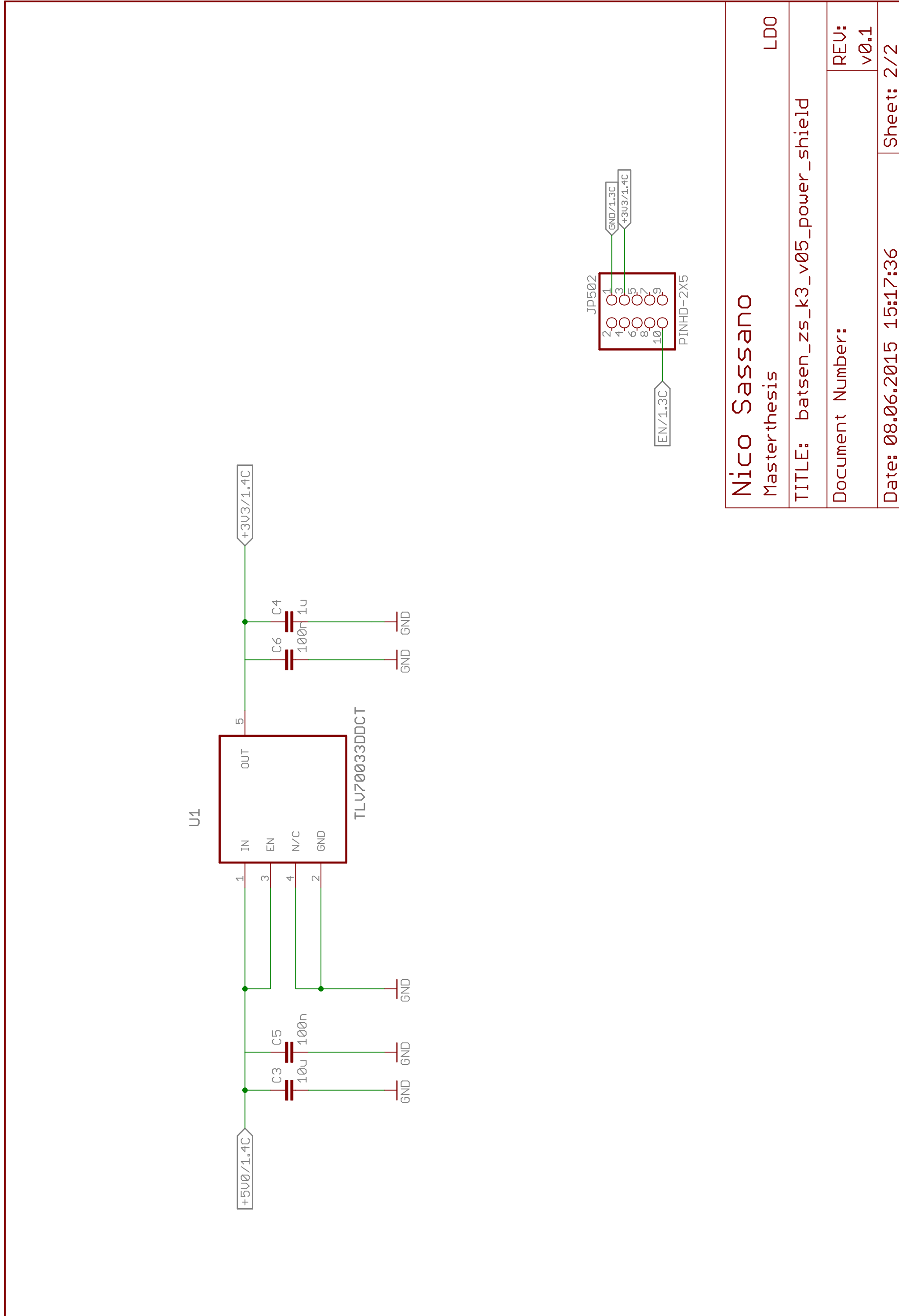
Schaltplan: Hochauflösende AD-Wandler Erweiterungsplatine



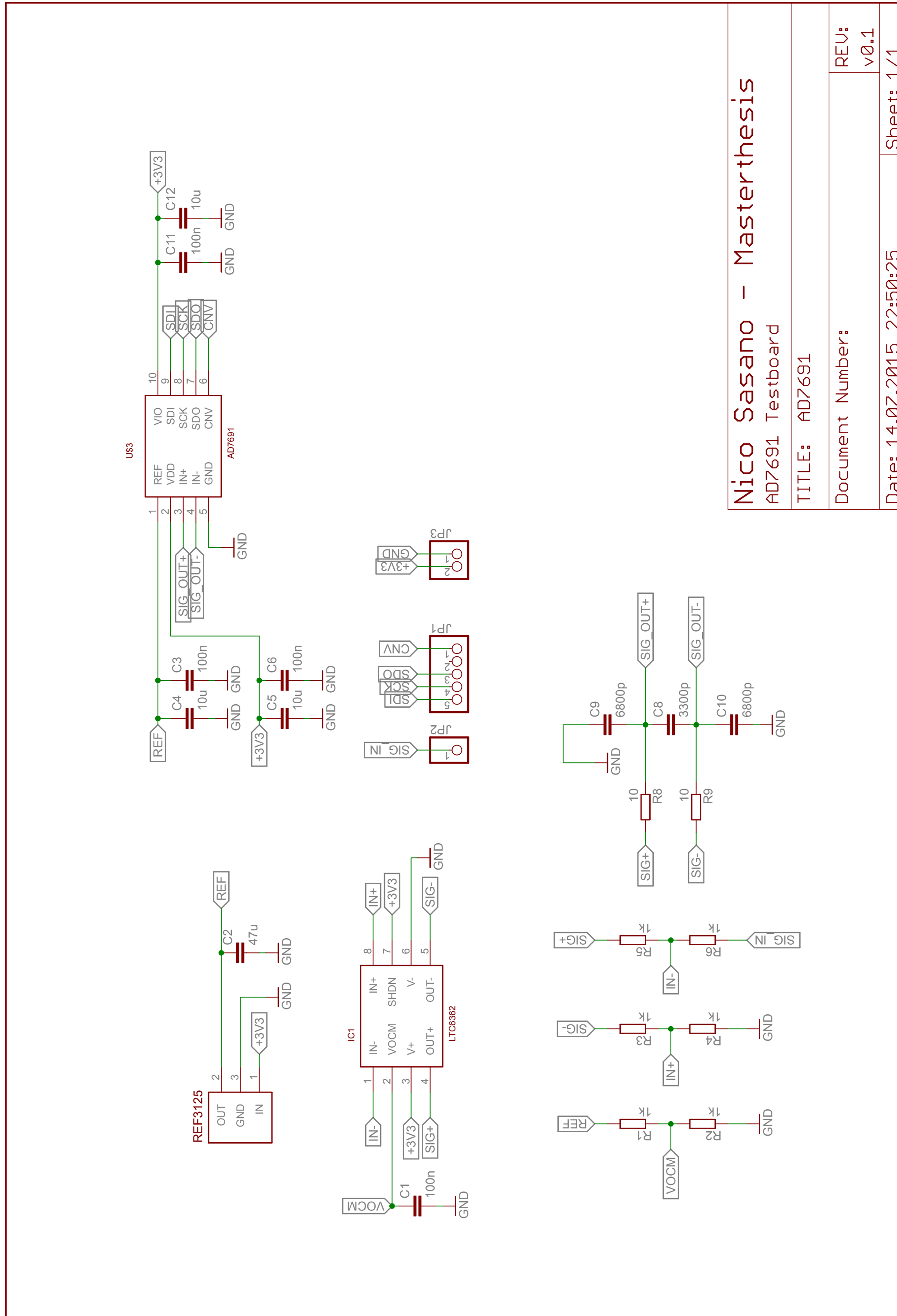

Nico Sassano
Masterthesis
AD7691 Shield
TITLE: batsen_zs_k3_v05shield_v02
Document Number:
REV: v0.1
Date: nicht gespeichert!
Sheet: 2/2
OPA2334
AD5270 - DAC2
AD5270 - DAC1
PINHD-2X5
JP2

Schaltplan: Spannungsversorgungs-Erweiterungsplatine

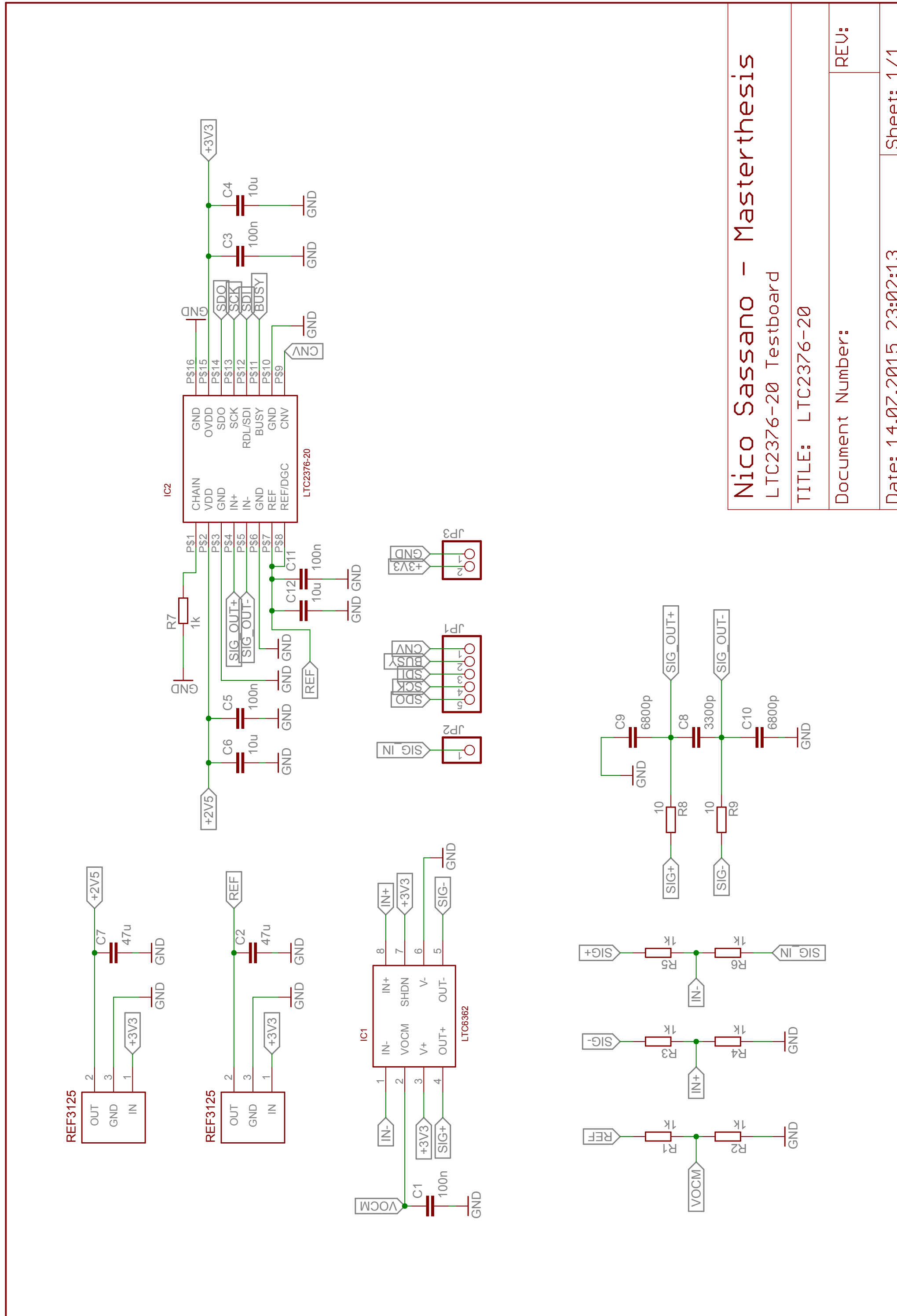



Nico Sassano
Masterthesis
LDO
TITLE: batsen_zs_k3_v05_power_shield
Document Number:
REV: v0.1
Date: 08.06.2015 15:17:36
Sheet: 2/2
JP502
PINHD-2X5
GND/1.3C
+3V3/1.4C
EN/1.3C
+3V3/1.4C
+5V0/1.4C
U1
TLV70033DDCT
IN
EN
N/C
GND
OUT
C4
1u
C6
100n
C5
100n
C3
10u
GND

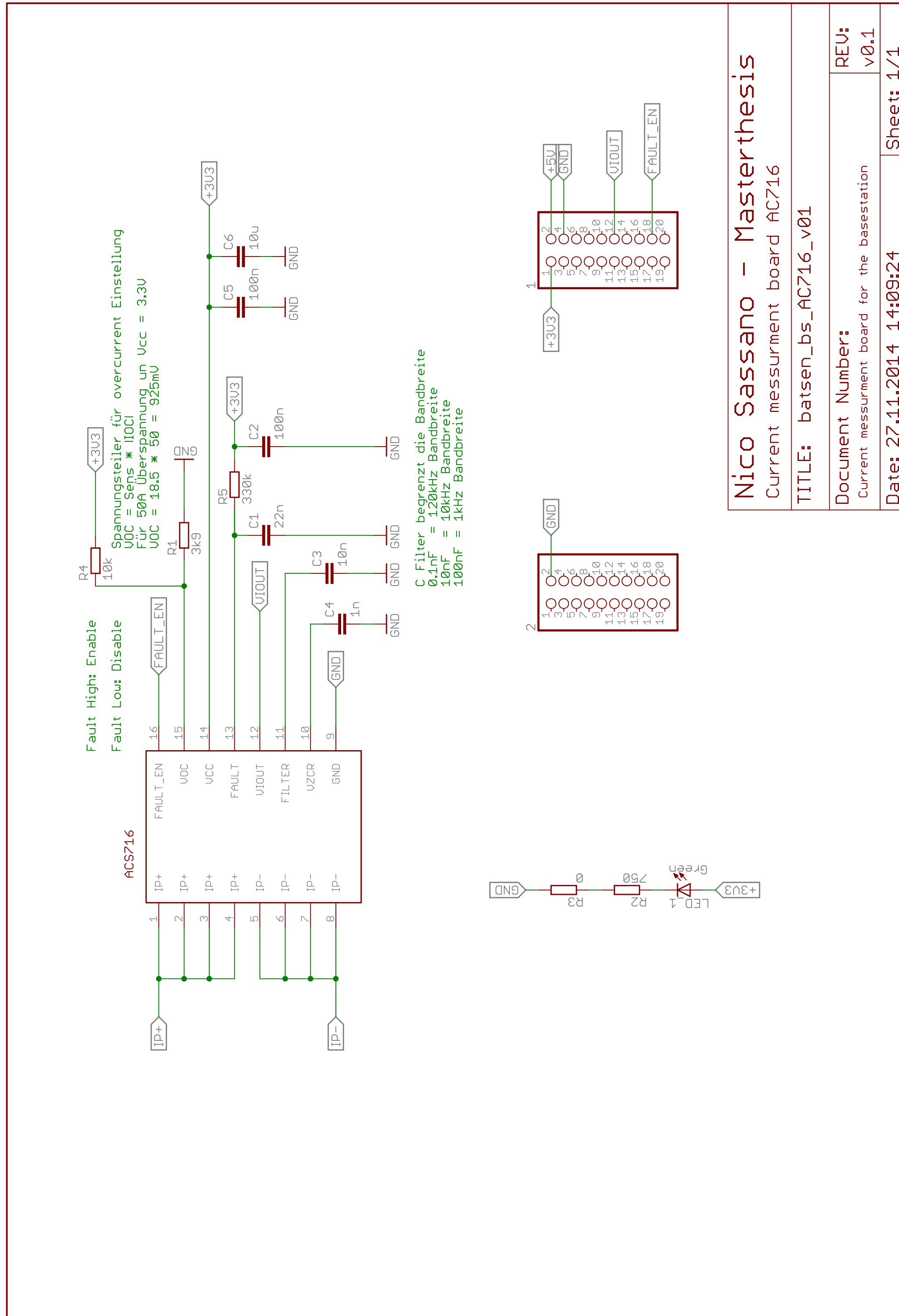
Schaltplan: AD7691 Testplatine



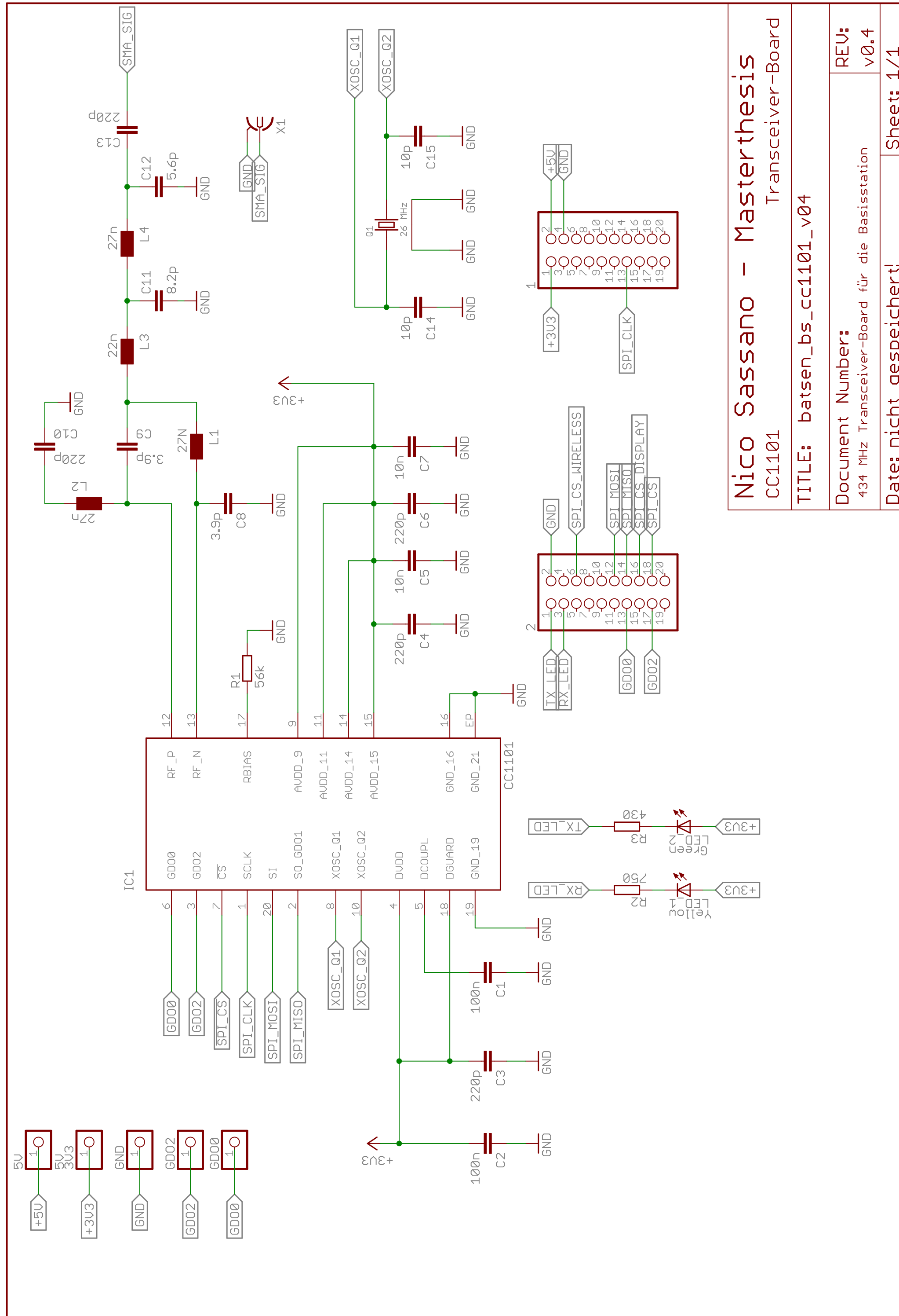
Schaltplan: LTC2376-20 Testplatine



Schaltplan: ACS716 Strommess-Erweiterungsplatine

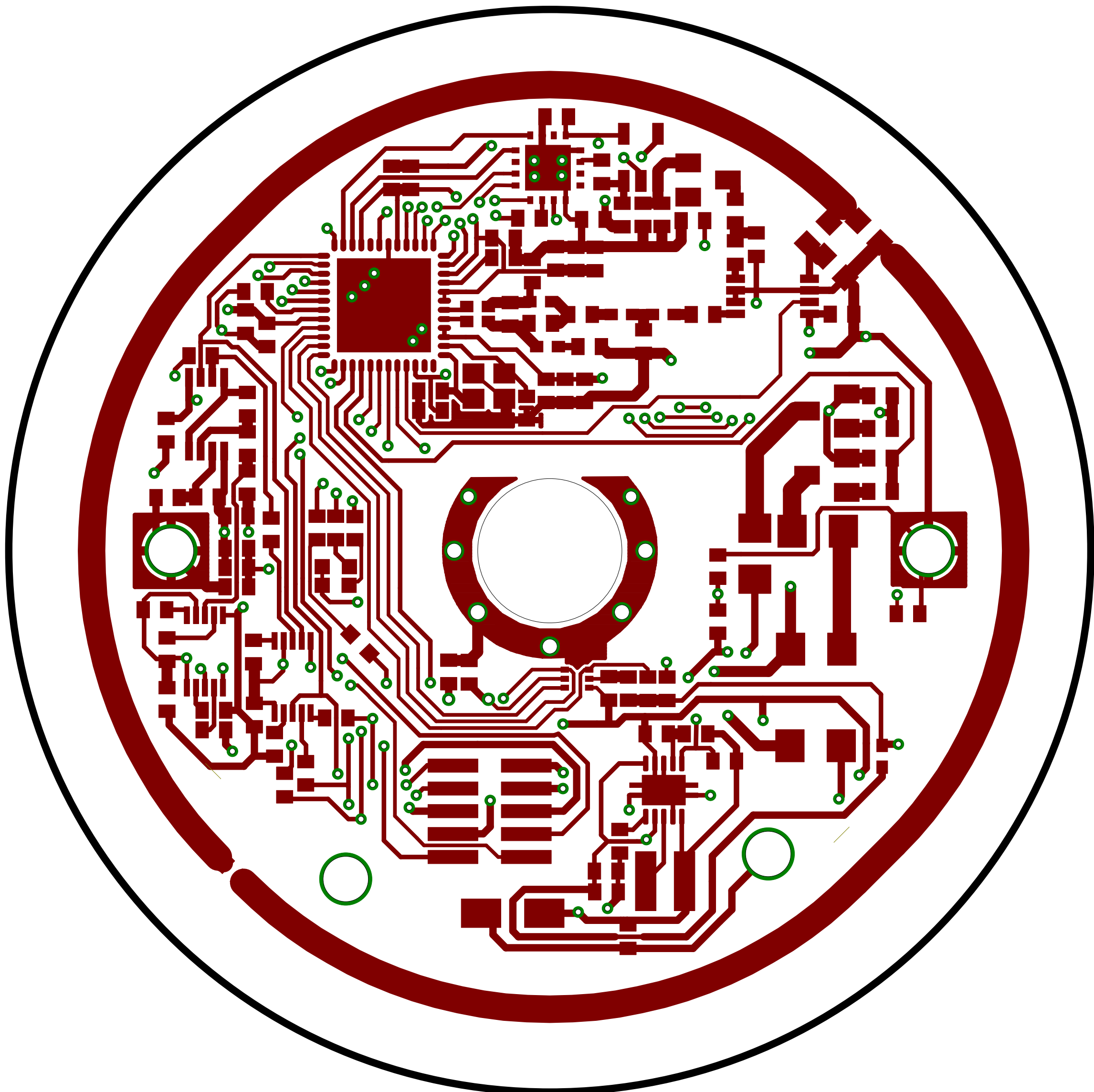


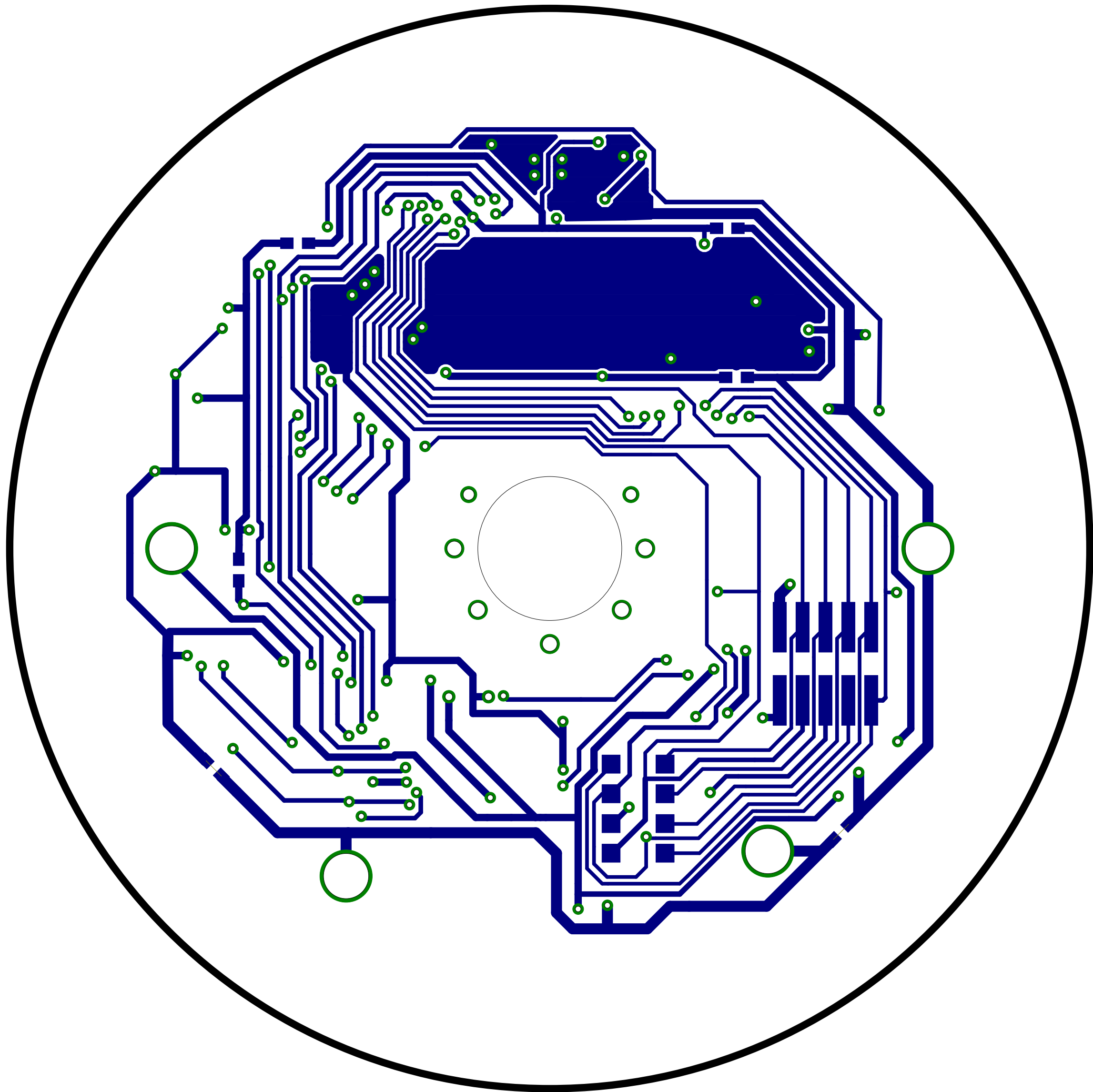
Schaltplan: CC1101 Transceiver-Erweiterungsplatine



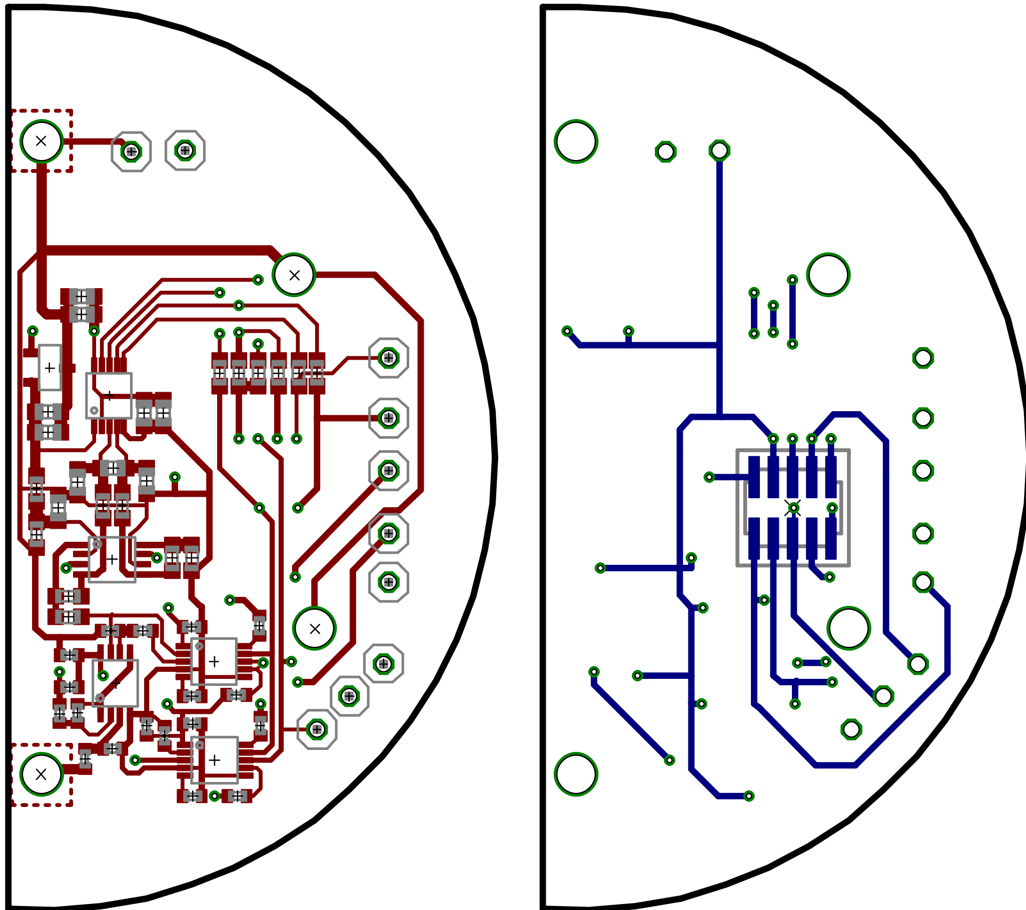
K. Platinenlayout

Layout: Zellsensor

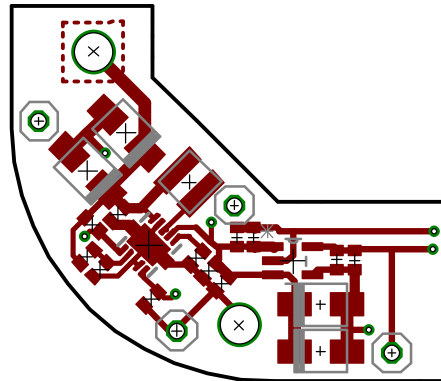


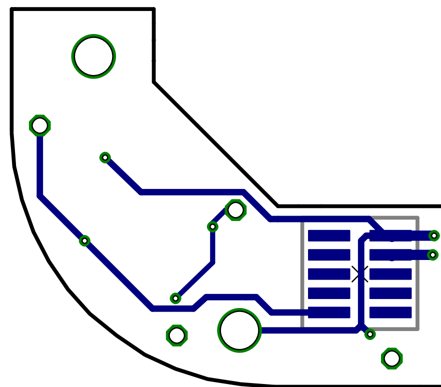


Layout: Hochauflösende AD-Wandler Erweiterungsplatine

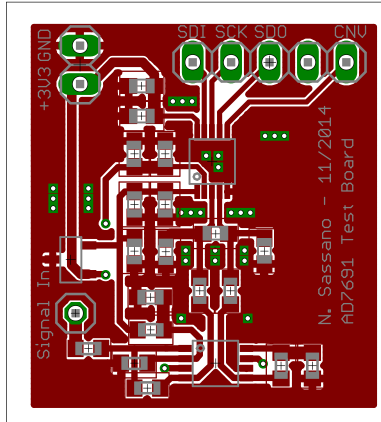


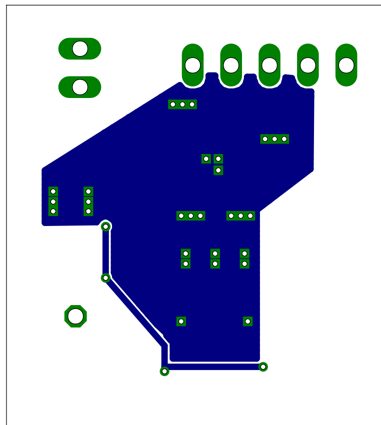
Layout: Spannungsversorgungs-Erweiterungsplatine



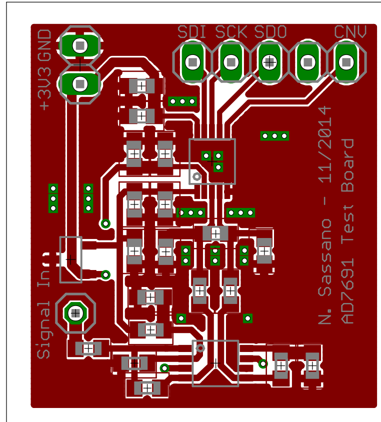


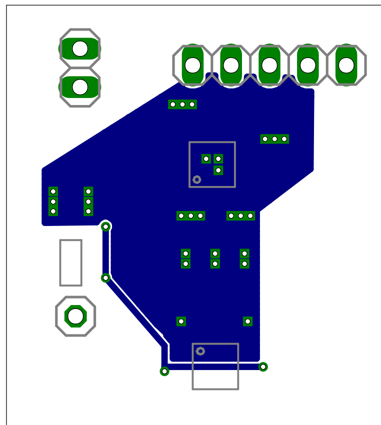
Layout: AD7691 Testplatine



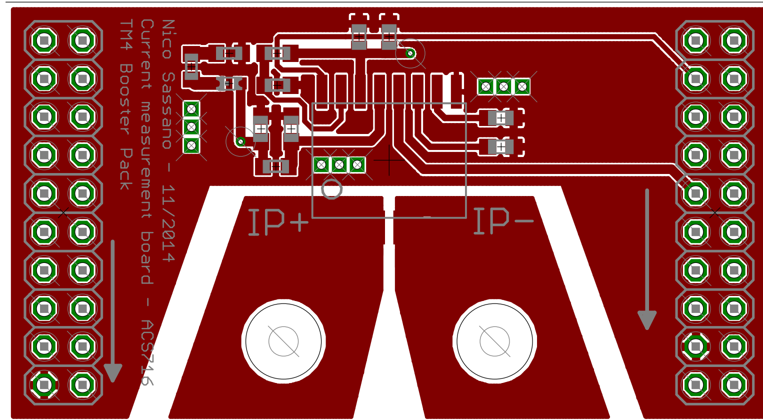


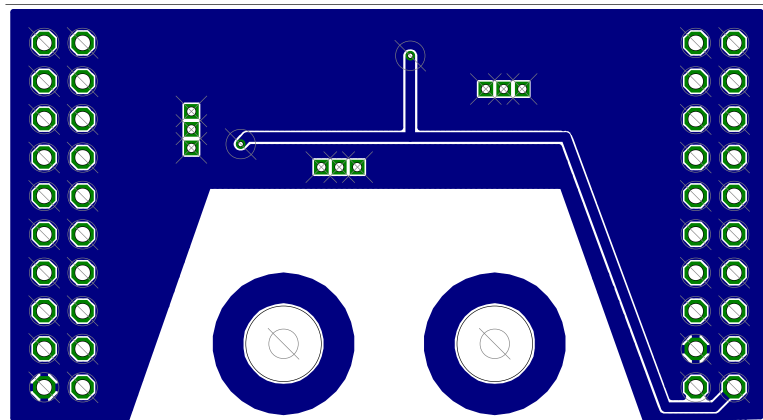
Layout: LTC2376-20 Testplatine



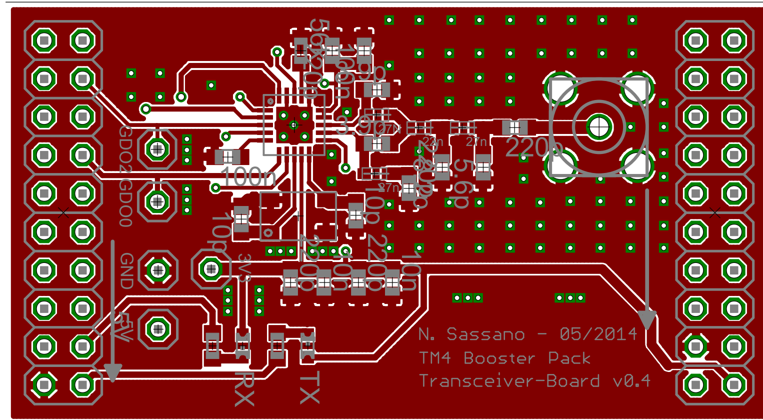


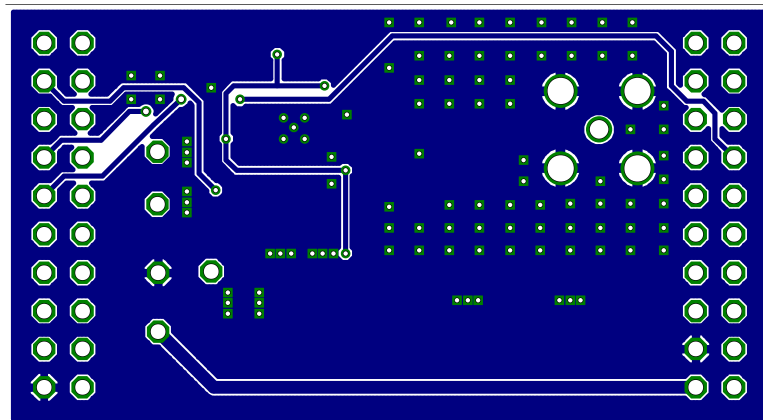
Layout: ACS716 Strommess-Erweiterungsplatine





Layout: CC1101 Transceiver-Erweiterungsplatine





L. Quellcode des Messsystem in C

Listings

Quellcode/GoertzelTestprogramm/goertzel_test.cpp 211

Quellcode/BS/AD5270.c 242
Quellcode/BS/AD7176.c 244
Quellcode/BS/AD7691.c 247
Quellcode/BS/adc.c 250
Quellcode/BS/CC1101.c 251
Quellcode/BS/convert.c 274
Quellcode/BS/gpio.c 279
Quellcode/BS/interrupts.c 281
Quellcode/BS/LTC2376.c 289
Quellcode/BS/main.c 293
Quellcode/BS/spi.c 312
Quellcode/BS/timer.c 313
Quellcode/BS/uart.c 316
Quellcode/BS/uart_check.c 317
Quellcode/BS/HeaderBS/AD5270.h 319
Quellcode/BS/HeaderBS/AD7176.h 319
Quellcode/BS/HeaderBS/AD7691.h 320
Quellcode/BS/HeaderBS/adc.h 320
Quellcode/BS/HeaderBS/CC1101.h 321
Quellcode/BS/HeaderBS/convert.h 325
Quellcode/BS/HeaderBS/gpio.h 325
Quellcode/BS/HeaderBS/interrupts.h 326
Quellcode/BS/HeaderBS/LTC2376.h 326
Quellcode/BS/HeaderBS/main.h 326
Quellcode/BS/HeaderBS/spi.h 327
Quellcode/BS/HeaderBS/convert.h 327
Quellcode/BS/HeaderBS/uart.h 327
Quellcode/BS/HeaderBS/uart_check.h 327
Quellcode/ZS/AD5270.c 328
Quellcode/ZS/AD7691.c 330
Quellcode/ZS/adc12.c 331
Quellcode/ZS/adg918.c 333

Quellcode/ZS/as3930.c 334
Quellcode/ZS/balancing.c 336
Quellcode/ZS/cc430.c 337
Quellcode/ZS/clk.c 344
Quellcode/ZS/delay.c 346
Quellcode/ZS/i2c.c 347
Quellcode/ZS/init.c 349
Quellcode/ZS/isr.c 351
Quellcode/ZS/led.c 364
Quellcode/ZS/main.c 365
Quellcode/ZS/spi.c 380
Quellcode/ZS/temp_sensor.c 381
Quellcode/ZS/timer.c 383
Quellcode/ZS/Header/AD5270.h 392
Quellcode/ZS/Header/adc12.h 392
Quellcode/ZS/Header/adg918.h 395
Quellcode/ZS/Header/as3930.h 395
Quellcode/ZS/Header/balancing.h 396
Quellcode/ZS/Header/cc430.h 396
Quellcode/ZS/Header/clk.h 398
Quellcode/ZS/Header/delay.h 398
Quellcode/ZS/Header/i2c.h 399
Quellcode/ZS/Header/init.h 399
Quellcode/ZS/Header/led.h 399
Quellcode/ZS/Header/main.h 400
Quellcode/ZS/Header/spi.h 404
Quellcode/ZS/Header/temp_sensor.h 405
Quellcode/ZS/Header/timer.h 406
Quellcode/ZS/Header/types.h 408

Quellcode/Matlab/EISMessung_Arbeit.m 409

```
/**********************************************************************************
**     File name       : AD5270.c
**     Hardware        : ARM Cortex M4 Basestation
**     Date            : 11/02/2015
**     Last Update     : 17/07/2015
**     Author          : Nico Sassano
**     Description     : A5270
**     State           : Final state
**********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/main.h"
#include "header/uart.h"
#include "header/spi.h"
#include "header/timer.h"
#include "header/interrupts.h"
#include "header/CC1101.h"
#include "header/gpio.h"
#include "header/convert.h"
#include "header/uart_check.h"
#include "header/adc.h"
#include "header/AD5270.h"

//*****************************************************
// This function is set the AD5270 DAC
// at the first OPA344 OP AMP
//*****************************************************
void write_AD5270_1(uint16_t res_value) {
    uint16_t value = 0;
    uint8_t upper_byte = 0;
    uint8_t lower_byte = 0;

    value = AD5270_WRITE_CMD | res_value;

    upper_byte = (value >> 8);
    lower_byte = (value >> 0);

    AD5270_CS1_OUTPUT;

    AD5270_CS1_ENABLE;

    SSIDataPutNonBlocking(SSI3_BASE, 0x1C);

    // Wait untill SSI3 is ready
    while(SSIBusy(SSI3_BASE));

    SSIDataPutNonBlocking(SSI3_BASE, 0x03);

    // Wait untill SSI3 is ready
    while(SSIBusy(SSI3_BASE));

    AD5270_CS1_DISABLE;

    delay_ms(1);
    AD5270_CS1_ENABLE;

    SSIDataPutNonBlocking(SSI3_BASE, upper_byte);

    // Wait untill SSI3 is ready
    while(SSIBusy(SSI3_BASE));

    SSIDataPutNonBlocking(SSI3_BASE, lower_byte);

    // Wait untill SSI3 is ready
    while(SSIBusy(SSI3_BASE));

    AD5270_CS1_DISABLE;
}

//*****************************************************
```

```
86  // This function is set the AD5270 DAC
87  // at the second OPA344 OP AMP
88  //****************************************************
89  void write_AD5270_2(uint16_t res_value) {
90      uint16_t value = 0;
91      uint8_t upper_byte = 0;
92      uint8_t lower_byte = 0;
93
94      value = AD5270_WRITE_CMD | res_value;
95
96      upper_byte = (value >> 8);
97      lower_byte = (value >> 0);
98
99      AD5270_CS2_OUTPUT;
100
101     AD5270_CS2_ENABLE;
102
103     SSIDataPutNonBlocking(SSI3_BASE, 0x1C);
104
105     // Wait untill SSI3 is ready
106     while(SSIBusy(SSI3_BASE));
107
108     SSIDataPutNonBlocking(SSI3_BASE, 0x03);
109
110     // Wait untill SSI3 is ready
111     while(SSIBusy(SSI3_BASE));
112
113     AD5270_CS2_DISABLE;
114
115     delay_ms(2);
116     AD5270_CS2_ENABLE;
117
118     SSIDataPutNonBlocking(SSI3_BASE, upper_byte);
119
120     // Wait untill SSI3 is ready
121     while(SSIBusy(SSI3_BASE));
122
123     SSIDataPutNonBlocking(SSI3_BASE, lower_byte);
124
125     // Wait untill SSI3 is ready
126     while(SSIBusy(SSI3_BASE));
127
128     AD5270_CS2_DISABLE;
129 }
130
131 //****************************************************
132 // This function set the output voltage
133 // after the first OPA334 OP AMP
134 //****************************************************
135 void ctr_voltage_AD5270(uint16_t final_voltage) {
136     uint16_t value_DAC = 0x000; // first value
137     uint16_t value_ADC = 0xFFF;
138     uint32_t pui32ADC0Value[1];
139
140     while(final_voltage < (value_ADC + 100)){
141         value_DAC = value_DAC + 10;
142         write_AD5270_1(value_DAC);
143         //************************************************************************
144         // !!!! ADC Wandlung ink. Abspeicherung dauert 3,2us !!!!!!!!
145         // ADC hat eine Referenzspannung von 3.3V
146         // Start the current messurment
147         ADCProcessorTrigger(ADC0_BASE, 3);              // Trigger the ADC conversion.
148         while(!ADCIntStatus(ADC0_BASE, 3, false));      // Wait for conversion to be completed
149         ADCIntClear(ADC0_BASE, 3);                      // Clear the ADC interrupt flag.
150         ADCSequenceDataGet(ADC0_BASE, 3, pui32ADC0Value);    // Read ADC Value.
151         value_ADC = pui32ADC0Value[0];
152         //************************************************************************
153     };
154 }
```

```c
/*********************************************************************************
**      File name       : AD5270.c
**      Hardware        : ARM Cortex M4 Basestation
**      Date            : 14/01/2015
**      Last Update     : 17/07/2015
**      Author          : Nico Sassano
**      Description     : AD7176
**      State           : Final state
*********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/main.h"
#include "header/uart.h"
#include "header/spi.h"
#include "header/timer.h"
#include "header/interrupts.h"
#include "header/CC1101.h"
#include "header/gpio.h"
#include "header/convert.h"
#include "header/uart_check.h"
#include "header/adc.h"
#include "header/AD7176.h"

extern uint32_t matlab_mode;

//*****************************************************************************
// AD7176 Test mode
// Data declaration for thesting AD7176
// This can be deleted after testing the ADC
//*****************************************************************************
extern volatile uint8_t AD7176_index;
extern volatile uint8_t AD7176_count;
extern volatile uint32_t AD7176_data[2]; // 24-bit data storage
extern volatile uint32_t AD7176_asynch_counter;
extern volatile uint32_t AD7176_values;
extern volatile uint32_t AD7176_frequency_time;
extern volatile uint32_t AD7176_value;
extern volatile uint32_t AD7176_value_storage[2000];

//*****************************************************
// This function is config the AD7176 ADC
//*****************************************************
void init_AD7176(void) {

    // BUSY Input Pin
    AD7176_BUSY_SET_INPUT;
    AD7176_IRQ_FALLING_EDGE;
    AD7176_IRQ_DISABLE;

    // CS LTC2376 Output Pin
    AD7176_CS_SET_OUTPUT;
    AD7176_CS_DISABLE;

}

//*****************************************************
// This function starts the sampling mode
//*****************************************************
void AD7176_start(uint8_t AD7176_frequency, uint8_t burst_values) {
    uint32_t index = 0;

    uint32_t value_high = 0;
    uint32_t value_low = 0;

    switch(burst_values) {
        case BURST_VALUES_50:
            AD7176_values = 50; break;

        case BURST_VALUES_100:
            AD7176_values = 100; break;
```

```
86
87          case BURST_VALUES_150:
88              AD7176_values = 150; break;
89
90          case BURST_VALUES_200:
91              AD7176_values = 200; break;
92
93          case BURST_VALUES_250:
94              AD7176_values = 250; break;
95
96          case BURST_VALUES_300:
97              AD7176_values = 300; break;
98
99          case BURST_VALUES_350:
100             AD7176_values = 350; break;
101
102         case BURST_VALUES_400:
103             AD7176_values = 400; break;
104
105         case BURST_VALUES_450:
106             AD7176_values = 450; break;
107
108         case BURST_VALUES_500:
109             AD7176_values = 500; break;
110
111         case BURST_VALUES_550:
112             AD7176_values = 550; break;
113
114         case BURST_VALUES_600:
115             AD7176_values = 600; break;
116
117         case BURST_VALUES_650:
118             AD7176_values = 650; break;
119
120         case BURST_VALUES_700:
121             AD7176_values = 700; break;
122
123         case BURST_VALUES_750:
124             AD7176_values = 750; break;
125
126         case BURST_VALUES_800:
127             AD7176_values = 800; break;
128
129         case BURST_VALUES_850:
130             AD7176_values = 850; break;
131
132         case BURST_VALUES_900:
133             AD7176_values = 900; break;
134
135         case BURST_VALUES_1000:
136             AD7176_values = 1000; break;
137
138         case BURST_VALUES_1500:
139             AD7176_values = 1500; break;
140
141         case BURST_VALUES_1900:
142             AD7176_values = 30000; break;
143     }
144
145     // To calculate the frequency_time
146     // (1/(frequency [Hz]*10^-6))
147     switch(AD7176_frequency) {
148         case BURST_FREQ_50HZ:
149             AD7176_frequency_time = 20000; break;
150
151         case BURST_FREQ_100HZ:
152             AD7176_frequency_time = 4000000;break;//10000; break;
153
154         case BURST_FREQ_150HZ:
155             AD7176_frequency_time = 6666; break;
156
157         case BURST_FREQ_200HZ:
158             AD7176_frequency_time = 5000; break;
159
160         case BURST_FREQ_250HZ:
161             AD7176_frequency_time = 4000; break;
162
163         case BURST_FREQ_300HZ:
164             AD7176_frequency_time = 3333; break;
165
166         case BURST_FREQ_350HZ:
167             AD7176_frequency_time = 2857; break;
168
169         case BURST_FREQ_400HZ:
170             AD7176_frequency_time = 2500; break;
```

```
171
172             case BURST_FREQ_450HZ:
173                 AD7176_frequency_time = 2222; break;
174
175             case BURST_FREQ_500HZ:
176                 AD7176_frequency_time = 2000; break;
177
178             case BURST_FREQ_550HZ:
179                 AD7176_frequency_time = 1818; break;
180
181             case BURST_FREQ_600HZ:
182                 AD7176_frequency_time = 1666; break;
183
184             case BURST_FREQ_650HZ:
185                 AD7176_frequency_time = 1538; break;
186
187             case BURST_FREQ_700HZ:
188                 AD7176_frequency_time = 1428; break;
189
190             case BURST_FREQ_750HZ:
191                 AD7176_frequency_time = 1333; break;
192
193             case BURST_FREQ_800HZ:
194                 AD7176_frequency_time = 1250; break;
195
196             case BURST_FREQ_850HZ:
197                 AD7176_frequency_time = 1176; break;
198
199             case BURST_FREQ_900HZ:
200                 AD7176_frequency_time = 1111; break;
201
202             case BURST_FREQ_950HZ:
203                 AD7176_frequency_time = 1052; break;
204
205             case BURST_FREQ_1000HZ:
206                 AD7176_frequency_time = 1000; break;
207
208             case BURST_FREQ_2000HZ:
209                 AD7176_frequency_time = 500; break;
210
211             case BURST_FREQ_4000HZ:
212                 AD7176_frequency_time = 250; break;
213
214             case BURST_FREQ_6000HZ:
215                 AD7176_frequency_time = 167; break;
216
217             case BURST_FREQ_8000HZ:
218                 AD7176_frequency_time = 125; break;
219
220             case BURST_FREQ_10000HZ:
221                 AD7176_frequency_time = 100; break;
222         }
223
224         AD7176_asynch_counter = 0;
225
226         // Config the timer for the ADC sampling
227         TIMER1config_AD7176(AD7176_frequency_time);
228
229         while(AD7176_asynch_counter != AD7176_values);
230         TIMER1_STOP;
231
232         value_low = AD7176_value_storage[0];
233         value_high = AD7176_value_storage[0];
234
235         for(index = 0 ; index < AD7176_asynch_counter ; index++) {
236
237             if(!matlab_mode) {   // If Matlab mode is off
238                 UARTprintf("%i\n",AD7176_value_storage[index]);
239             }else {              // If Matlab mode is on
240                 if(index == AD7176_asynch_counter-1) {
241                     UARTprintf("%i",AD7176_value_storage[index]);
242                 }else {
243                     UARTprintf("%i  ",AD7176_value_storage[index]);
244                 }
245             }
246
247             if(AD7176_value_storage[index] <= value_low)
248                 value_low = AD7176_value_storage[index];
249
250             if(AD7176_value_storage[index] >= value_high)
251                 value_high = AD7176_value_storage[index];
252
253         }
254     }
```

```
/*********************************************************************************
**      File name       : LTC2376.c
**      Hardware        : ARM Cortex M4 Basestation
**      Date            : 22/12/2015
**      Last Update     : 17/07/2015
**      Author          : Nico Sassano
**      Description     : LTC2376
**      State           : Final state
*********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/main.h"
#include "header/uart.h"
#include "header/spi.h"
#include "header/timer.h"
#include "header/interrupts.h"
#include "header/CC1101.h"
#include "header/gpio.h"
#include "header/convert.h"
#include "header/uart_check.h"
#include "header/adc.h"
#include "header/LTC2376.h"
#include "header/AD7691.h"

extern uint32_t matlab_mode;

//*****************************************************************************
// AD7691 Test mode
// Data declaration for thesting AD7691
// This can be deleted after testing the ADC
//*****************************************************************************
extern volatile uint8_t AD7691_index;
extern volatile uint8_t AD7691_count;
extern volatile uint32_t AD7691_data[2]; // 24-bit data storage
extern volatile uint32_t AD7691_asynch_counter;
extern volatile uint32_t AD7691_values;
extern volatile uint32_t AD7691_frequency_time;
extern volatile uint32_t AD7691_value;
extern volatile uint32_t AD7691_value_storage[2000];

//***************************************************
// This function is config the AD7691 ADC
//***************************************************
void init_ad7691(void) {

    // CNV Output Pin
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_6);
    AD7691_CNV_SET_LOW;

    // CS LTC2376 Output Pin
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1);
    AD7691_CS_DISABLE;

    AD7691_CS_DISABLE;
}

//***************************************************
// This function starts the sampling mode
//***************************************************
void ad7691_start(uint8_t ad7691_frequency, uint8_t burst_values) {
    uint32_t index = 0;

    uint32_t value_high = 0;
    uint32_t value_low = 0;

    switch(burst_values) {
        case BURST_VALUES_50:
            AD7691_values = 50; break;
```

```
86
87          case BURST_VALUES_100:
88              AD7691_values = 100; break;
89
90          case BURST_VALUES_150:
91              AD7691_values = 150; break;
92
93          case BURST_VALUES_200:
94              AD7691_values = 200; break;
95
96          case BURST_VALUES_250:
97              AD7691_values = 250; break;
98
99          case BURST_VALUES_300:
100             AD7691_values = 300; break;
101
102         case BURST_VALUES_350:
103             AD7691_values = 350; break;
104
105         case BURST_VALUES_400:
106             AD7691_values = 400; break;
107
108         case BURST_VALUES_450:
109             AD7691_values = 450; break;
110
111         case BURST_VALUES_500:
112             AD7691_values = 500; break;
113
114         case BURST_VALUES_550:
115             AD7691_values = 550; break;
116
117         case BURST_VALUES_600:
118             AD7691_values = 600; break;
119
120         case BURST_VALUES_650:
121             AD7691_values = 650; break;
122
123         case BURST_VALUES_700:
124             AD7691_values = 700; break;
125
126         case BURST_VALUES_750:
127             AD7691_values = 750; break;
128
129         case BURST_VALUES_800:
130             AD7691_values = 800; break;
131
132         case BURST_VALUES_850:
133             AD7691_values = 850; break;
134
135         case BURST_VALUES_900:
136             AD7691_values = 900; break;
137
138         case BURST_VALUES_1000:
139             AD7691_values = 1000; break;
140
141         case BURST_VALUES_1500:
142             AD7691_values = 1500; break;
143
144         case BURST_VALUES_1900:
145             AD7691_values = 30000; break;
146     }
147
148     // To calculate the frequency_time
149     // (1/(frequency [Hz]*10^-6))
150     switch(ad7691_frequency) {
151         case BURST_FREQ_50HZ:
152             AD7691_frequency_time = 20000; break;
153
154         case BURST_FREQ_100HZ:
155             AD7691_frequency_time = 10000; break;
156
157         case BURST_FREQ_150HZ:
158             AD7691_frequency_time = 6666; break;
159
160         case BURST_FREQ_200HZ:
161             AD7691_frequency_time = 5000; break;
162
163         case BURST_FREQ_250HZ:
164             AD7691_frequency_time = 4000; break;
165
166         case BURST_FREQ_300HZ:
167             AD7691_frequency_time = 3333; break;
168
169         case BURST_FREQ_350HZ:
170             AD7691_frequency_time = 2857; break;
```

```
171
172             case BURST_FREQ_400HZ:
173                 AD7691_frequency_time = 2500; break;
174
175             case BURST_FREQ_450HZ:
176                 AD7691_frequency_time = 2222; break;
177
178             case BURST_FREQ_500HZ:
179                 AD7691_frequency_time = 2000; break;
180
181             case BURST_FREQ_550HZ:
182                 AD7691_frequency_time = 1818; break;
183
184             case BURST_FREQ_600HZ:
185                 AD7691_frequency_time = 1666; break;
186
187             case BURST_FREQ_650HZ:
188                 AD7691_frequency_time = 1538; break;
189
190             case BURST_FREQ_700HZ:
191                 AD7691_frequency_time = 1428; break;
192
193             case BURST_FREQ_750HZ:
194                 AD7691_frequency_time = 1333; break;
195
196             case BURST_FREQ_800HZ:
197                 AD7691_frequency_time = 1250; break;
198
199             case BURST_FREQ_850HZ:
200                 AD7691_frequency_time = 1176; break;
201
202             case BURST_FREQ_900HZ:
203                 AD7691_frequency_time = 1111; break;
204
205             case BURST_FREQ_950HZ:
206                 AD7691_frequency_time = 1052; break;
207
208             case BURST_FREQ_1000HZ:
209                 AD7691_frequency_time = 1000; break;
210
211             case BURST_FREQ_2000HZ:
212                 AD7691_frequency_time = 500; break;
213
214             case BURST_FREQ_4000HZ:
215                 AD7691_frequency_time = 250; break;
216
217             case BURST_FREQ_6000HZ:
218                 AD7691_frequency_time = 167; break;
219
220             case BURST_FREQ_8000HZ:
221                 AD7691_frequency_time = 125; break;
222
223             case BURST_FREQ_10000HZ:
224                 AD7691_frequency_time = 100; break;
225         }
226
227         AD7691_asynch_counter = 0;
228         TIMER1config_AD7691(AD7691_frequency_time);
229
230         while(AD7691_asynch_counter != AD7691_values);
231         TIMER1_STOP;
232
233         value_low = AD7691_value_storage[0];
234         value_high = AD7691_value_storage[0];
235
236         for(index = 0 ; index < AD7691_asynch_counter ; index++) {
237
238             if(!matlab_mode) {   // If Matlab mode is off
239                 UARTprintf("%i\n",AD7691_value_storage[index]);
240             }else {              // If Matlab mode is on
241                 if(index == AD7691_asynch_counter-1) {
242                     UARTprintf("%i",AD7691_value_storage[index]);
243                 }else {
244                     UARTprintf("%i  ",AD7691_value_storage[index]);
245                 }
246             }
247
248             if(AD7691_value_storage[index] <= value_low)
249                 value_low = AD7691_value_storage[index];
250
251             if(AD7691_value_storage[index] >= value_high)
252                 value_high = AD7691_value_storage[index];
253         }
254     }
```

```c
/*******************************************************************************
**     File name       : adc.c
**     Hardware        : ARM Cortex M4 Basestation
**     Date            : 17/09/2014
**     Last Update     : 17/07/2015
**     Author          : Nico Sassano
**     Description     : ADC
**     State           : Final state
*******************************************************************************/
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

void init_adc() {
    // The ADC0 peripheral must be enabled for use.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    // Using PE3 -> This pin is AIN0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);

    // Enable sample sequence 3 with a processor signal trigger.  Sequence 3
    // will do a single sample when the processor sends a signal to start the
    // conversion.  Each ADC module has 4 programmable sequences, sequence 0
    // to sequence 3.  This example is arbitrarily using sequence 3.
    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);

    // Configure step 0 on sequence 3.  Sample channel 0 (ADC_CTL_CH0) in
    // single-ended mode (default) and configure the interrupt flag
    // (ADC_CTL_IE) to be set when the sample is done.  Tell the ADC logic
    // that this is the last conversion on sequence 3 (ADC_CTL_END).  Sequence
    // 3 has only one programmable step.  Sequence 1 and 2 have 4 steps, and
    // sequence 0 has 8 programmable steps.  Since we are only doing a single
    // conversion using sequence 3 we will only configure step 0.  For more
    // information on the ADC sequences and steps, reference the datasheet.
    ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 | ADC_CTL_IE | ADC_CTL_END);

    // Since sample sequence 3 is now configured, it must be enabled.
    ADCSequenceEnable(ADC0_BASE, 3);

    // Clear the interrupt status flag.  This is done to make sure the
    // interrupt flag is cleared before we sample.
    ADCIntClear(ADC0_BASE, 3);

}

uint32_t read_adc() {
    uint32_t pui32ADC0Value[1];

    // Trigger the ADC conversion.
    ADCProcessorTrigger(ADC0_BASE, 3);

    // Wait for conversion to be completed.
    while(!ADCIntStatus(ADC0_BASE, 3, false))

    // Clear the ADC interrupt flag.
    ADCIntClear(ADC0_BASE, 3);

    // Read ADC Value.
    ADCSequenceDataGet(ADC0_BASE, 3, pui32ADC0Value);

    return pui32ADC0Value[0];
}
```

```c
/**********************************************************************************
**     File name       : CC1101.c
**     Hardware        : ARM Cortex M4 Basestation
**     Date            : 20/08/2014
**     Last Update     : 17/07/2015
**     Author          : Nico Sassano
**     Description     : CC1101
**     State           : Final state
**********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/uart.h"
#include "header/interrupts.h"
#include "header/cc1101.h"
#include "header/spi.h"
#include "header/timer.h"

// Private function declaration
void cc1101_config_tx(uint8_t brate);
void cc1101_config_rx(uint8_t brate);
void cc1101_tx(unsigned long packetes_to_tx);
void cc1101_rx(uint8_t);

void cc1101_set_rx(unsigned long brate);

void cc1101_fill_tx_fifo(uint8_t *, uint8_t );

void wakeup(void);

void cc1101_enable_crc(void);

uint8_t cc1101_spi_read_register(uint8_t address);
void cc1101_spi_read_register_burst(uint8_t , uint8_t *, uint8_t );

extern volatile uint8_t tx_data_length;

/*********************************************
** Communication states
*********************************************/
extern volatile uint8_t command_recived_flag;

extern volatile uint8_t brate_start;
extern volatile uint8_t brate;
extern volatile uint16_t asynch_counter;
extern uint32_t rx_timeout;
extern uint32_t tx_timeout;

/*********************************************
** Interruptflags
*********************************************/
extern volatile uint8_t irq_mode;           // PORT1 IRQ Status
extern volatile uint8_t irq_send_done_flag; // CC1101 senden
extern volatile uint8_t irq_cali_flag;      // Flag for burat calibration


//*****************************************************************
//*** public functions ********************************************
//*****************************************************************

//****************************************************
// This function is nessesary to start up the CC1101
//****************************************************
void cc1101_power_up_reset(void) {

    // Enable the TX LED on the Tranceiver Board V0.4
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_1);

    // Enable the RX LED on the Tranceiver Board V0.4
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);
```

```
86        GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_4);
87
88        // Enable the Chip select Port: Pin Q1
89        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);
90        GPIOPinTypeGPIOOutput(GPIO_PORTQ_BASE, GPIO_PIN_1);
91
92        // Set GDO0
93        CC1101_GDO0_SET_INPUT;
94
95        // Set GDO2 and the interrupt
96        CC1101_GDO2_SET_INPUT;
97        GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_6,GPIO_RISING_EDGE);
98        GPIOIntDisable(GPIO_PORTK_BASE,GPIO_INT_PIN_6);
99        IntDisable(INT_GPIOK);
100
101       // Manual power-on reset
102       CS_ENABLE;
103       delay_ms(3);
104
105       CS_DISABLE;
106       delay_ms(3);
107
108       CS_ENABLE;
109       delay_ms(3);
110
111       CS_DISABLE;
112       delay_ms(3);
113
114       cc1101_reset();
115   }
116
117   //****************************************************
118   // Function to configure the CC1101 for TX packages
119   //****************************************************
120   void tx_packet(uint8_t packetes, uint8_t *txbuf) {
121
122       CC1101_GDO2_CLEAR_IRQ;       // Clear the sync word detected interrupt
123       CC1101_GDO2_IRQ_ENABLE;      // Enable the sync word detected interrupt
124
125       TX_DONE_UNSET;               // Unset the IRQ flags
126       IRQ_SET_TX;
127
128       cc1101_set_tx(brate);
129       cc1101_fill_tx_fifo(txbuf, packetes);
130       cc1101_tx(packetes);
131
132       TX_LED_ON;                   // Turn the TX LED on
133       timeout_start(tx_timeout);       // Start the timeout timer
134
135       while((!TX_DONE_IS_SET)&&(!TIMEOUT_IS_SET));   //Wait until data was send
136       //delay_ms(1);
137
138       TX_LED_OFF;                  // Turn the TX LED off
139
140       tx_data_length = 0;          // Reset the data length
141
142       cc1101_idle();               // Go to the idle mode
143   }
144
145   //****************************************************
146   // This function configure the CC1101 for the RX mode
147   //****************************************************
148   void rx_packet(uint8_t packages) {
149       CC1101_GDO2_CLEAR_IRQ;       // Clear the sync word detected interrupt
150       CC1101_GDO2_IRQ_DISABLE;     // Enable the sync word detected interrupt
151
152       cc1101_set_rx(brate);        // Set the CC1101 for the RX mode
153       cc1101_rx(packages);         // Transceiver in RX state
154
155       COMMAND_REVIVED_UNSET;       // Unset the IRQ flags
156       IRQ_SET_RX;
157
158       CC1101_GDO2_CLEAR_IRQ;       // Clear the sync word detected interrupt
159       CC1101_GDO2_IRQ_ENABLE;      // Enable the sync word detected interrupt
160
161       timeout_start(rx_timeout);       // Start the timeout timer
162       while((!COMMAND_REVIVED_IS_SET)&&(!TIMEOUT_IS_SET));     // Wait until data packages are received or timeout is
              arrived
163
164       if (COMMAND_REVIVED_IS_SET)
165           timeout_stop();
166   }
167
168   //****************************************************
169   // This function starts the asynchronous mode
```

```
170 //*****************************************************
171 void cc1101_tx_asynchronous_mode(void) {
172     uint32_t spiSendData[1];
173 
174     spiSendData[0] = CC1101_CS_STX;
175     SPI_send_data(1,spiSendData);
176 }
177 
178 //*****************************************************
179 // This function starts the asynchronous mode
180 //*****************************************************
181 void cc1101_rx_asynchronous_mode(void) {
182     uint32_t spiSendData[1];
183 
184     spiSendData[0] = CC1101_CS_SRX;
185     SPI_send_data(1,spiSendData);
186 }
187 
188 //*****************************************************
189 // This function read out the RX FIFO
190 //*****************************************************
191 void cc1101_read_rx_fifo(uint8_t *buffer, uint8_t length) {
192     // Disable the receiver
193     cc1101_idle();
194 
195     // Transfer the bytes via SPI from the RX fifo
196     cc1101_spi_read_register_burst(CC1101_ML_RXFIFO, buffer, length);
197 }
198 
199 //*****************************************************
200 // This function returns the number of the received bytes
201 //*****************************************************
202 uint8_t cc1101_get_rxbytes(void) {
203     return (cc1101_spi_read_register(CC1101_SR_RXBYTES) & 0x7F);
204 }
205 
206 //*****************************************************
207 // This function starts the burst measurement mode
208 //*****************************************************
209 void burst_start(uint8_t burst_frequency, uint8_t burst_values) {
210 
211     uint8_t packet[4];
212     uint32_t values = 0;
213     uint32_t frequency_time = 0;
214 
215     switch(burst_values) {
216         case BURST_VALUES_50:
217             values = 50;
218         break;
219 
220         case BURST_VALUES_100:
221             values = 100;
222         break;
223 
224         case BURST_VALUES_150:
225             values = 150;
226         break;
227 
228         case BURST_VALUES_200:
229             values = 200;
230         break;
231 
232         case BURST_VALUES_250:
233             values = 250;
234         break;
235 
236         case BURST_VALUES_300:
237             values = 300;
238         break;
239 
240         case BURST_VALUES_350:
241             values = 350;
242         break;
243 
244         case BURST_VALUES_400:
245             values = 400;
246         break;
247 
248         case BURST_VALUES_450:
249             values = 450;
250         break;
251 
252         case BURST_VALUES_500:
253             values = 500;
254         break;
```

```
255
256            case BURST_VALUES_550:
257                values = 550;
258            break;
259
260            case BURST_VALUES_600:
261                values = 600;
262            break;
263
264            case BURST_VALUES_650:
265                values = 650;
266            break;
267
268            case BURST_VALUES_700:
269                values = 700;
270            break;
271
272            case BURST_VALUES_750:
273                values = 750;
274            break;
275
276            case BURST_VALUES_800:
277                values = 800;
278            break;
279
280            case BURST_VALUES_850:
281                values = 850;
282            break;
283
284            case BURST_VALUES_900:
285                values = 900;
286            break;
287
288            case BURST_VALUES_1000:
289                values = 1000;
290            break;
291
292            case BURST_VALUES_1500:
293                values = 1500;
294            break;
295
296            case BURST_VALUES_1900:
297                values = 1900;
298            break;
299
300            case BURST_VALUES_2000:
301                values = 2000;
302            break;
303
304            case BURST_VALUES_2500:
305                values = 2500;
306            break;
307
308        }
309
310        // To calculate the frequency_time
311        // (1/(frequency [Hz]*10^-6))
312        switch(burst_frequency) {
313            case BURST_FREQ_50HZ:    // 50.00Hz
314                frequency_time = 2400000; break;
315
316            case BURST_FREQ_100HZ:   // 100Hz
317                frequency_time = 1199100; break;
318
319            case BURST_FREQ_150HZ:   // 150Hz
320                frequency_time = 799800; break;
321
322            case BURST_FREQ_200HZ:
323                frequency_time = 599900; break;
324
325            case BURST_FREQ_250HZ:
326                frequency_time = 480000; break;
327
328            case BURST_FREQ_300HZ:
329                frequency_time = 400000; break;
330
331            case BURST_FREQ_350HZ:
332                frequency_time = 342857; break;
333
334            case BURST_FREQ_400HZ:
335                frequency_time = 300000; break;
336
337            case BURST_FREQ_450HZ:
338                frequency_time = 266667; break;
339
```

```
340         case BURST_FREQ_500HZ:   //  500Hz
341             frequency_time = 239648; break;
342
343         case BURST_FREQ_550HZ:
344             frequency_time = 218182; break;
345
346         case BURST_FREQ_600HZ:   //  600Hz
347             frequency_time = 199648; break;
348
349         case BURST_FREQ_650HZ:
350             frequency_time = 184615; break;
351
352         case BURST_FREQ_700HZ:
353             frequency_time = 171429; break;
354
355         case BURST_FREQ_750HZ:
356             frequency_time = 160000; break;
357
358         case BURST_FREQ_800HZ:
359             frequency_time = 150000; break;
360
361         case BURST_FREQ_850HZ:
362             frequency_time = 141176; break;
363
364         case BURST_FREQ_900HZ:
365             frequency_time = 133333; break;
366
367         case BURST_FREQ_950HZ:
368             frequency_time = 126316; break;
369
370         case BURST_FREQ_1000HZ:  //  1.000Hz
371             frequency_time = 119610; break;
372
373         case BURST_FREQ_2000HZ:  //  2.000Hz
374             frequency_time = 59648; break;
375
376         case BURST_FREQ_4000HZ:  //  4.000Hz
377             frequency_time = 29648; break;
378
379         case BURST_FREQ_6000HZ:  //5.999Hz
380             frequency_time = 19648; break;
381
382         case BURST_FREQ_8000HZ:  //8.000Hz
383             frequency_time = 14648; break;
384
385         case BURST_FREQ_10000HZ:     //10kHz
386             frequency_time = 11624; break;
387
388         case BURST_FREQ_12000HZ:
389             frequency_time = 10000; break;
390
391         case BURST_FREQ_14000HZ:
392             frequency_time = 8571; break;
393
394         case BURST_FREQ_16000HZ:
395             frequency_time = 7500; break;
396
397         case BURST_FREQ_18000HZ:
398             frequency_time = 6667; break;
399
400         case BURST_FREQ_20000HZ:     //20kHz
401             frequency_time = 5628; break;
402
403         case BURST_FREQ_22000HZ:
404             frequency_time = 5455; break;
405
406         case BURST_FREQ_24000HZ:
407             frequency_time = 5000; break;
408
409         case BURST_FREQ_26000HZ:
410             frequency_time = 4615; break;
411
412         case BURST_FREQ_28000HZ:
413             frequency_time = 4286; break;
414
415         case BURST_FREQ_30000HZ:     //  30.01kHz
416             frequency_time = 3627; break;
417
418         case BURST_FREQ_32000HZ:
419             frequency_time = 3750; break;
420
421         case BURST_FREQ_34000HZ:
422             frequency_time = 3529; break;
423
424         case BURST_FREQ_36000HZ:
```

```
425                 frequency_time = 3333; break;
426
427             case BURST_FREQ_38000HZ:
428                 frequency_time = 3158; break;
429
430             case BURST_FREQ_40000HZ:      // 40.01kHz
431                 frequency_time = 2627; break;
432
433             case BURST_FREQ_45000HZ:
434                 frequency_time = 2667; break;
435
436             case BURST_FREQ_50000HZ:      //50.02kHz
437                 frequency_time = 2027; break;
438
439             case BURST_FREQ_55000HZ:
440                 frequency_time = 2182; break;
441
442             case BURST_FREQ_60000HZ:      // 60.03kHz
443                 frequency_time = 1627; break;
444
445             case BURST_FREQ_65000HZ:
446                 frequency_time = 1846; break;
447
448             case BURST_FREQ_70000HZ:      // 69.93kHz
449                 frequency_time = 1339; break;
450
451             case BURST_FREQ_75000HZ:
452                 frequency_time = 1600; break;
453
454             case BURST_FREQ_80000HZ:      //80.0kHz
455                 frequency_time = 1128; break;
456
457             case BURST_FREQ_85000HZ:
458                 frequency_time = 1412 - 354; break;
459
460             case BURST_FREQ_90000HZ:      //90.09kHz
461                 frequency_time = 960; break;
462
463             case BURST_FREQ_95000HZ:
464                 frequency_time = 1263; break;
465
466             case BURST_FREQ_100000HZ:
467                 frequency_time = 1200; break;
468
469         }
470
471         packet[0] = BROADCAST;
472         packet[1] = COMMAND_DOWNLINK_BURST_MODE;
473         packet[2] = burst_frequency;
474         packet[3] = burst_values;
475
476         tx_data_length = 0;                                      // Normal Data length
477         tx_packet(HEADER_LENGTH + tx_data_length, packet);       // Config CC1101 for data tranceive
478
479         // Start the burts messurment
480         CC1101_GDO2_IRQ_DISABLE;
481         CC1101_GDO2_CLEAR_IRQ;
482
483         CC1101_GDO0_SET_INPUT;
484         cc1101_reset();                  // Reset all registers to their default values and go to idle state
485         cc1101_config_burst_tx();        // Configure the transceiver for sending the wakeup signal
486         cc1101_tx_asynchronous_mode();   // Change to TX state
487
488         CC1101_GDO0_SET_OUTPUT;
489         CC1101_GDO0_SET_HIGH;
490
491         asynch_counter = 0;
492         delay_ms(100);
493         TIMER1config(frequency_time);
494
495         while(asynch_counter!=values);
496         ROM_TimerDisable(TIMER1_BASE, TIMER_A);
497         CC1101_GDO0_SET_INPUT;
498
499         packet[0] = BROADCAST;
500         packet[1] = COMMAND_DOWNLINK_BURST_CHECK;
501         packet[2] = 0x00;
502         packet[3] = 0x00;
503
504         tx_data_length = 0;                                      // Normal Data length
505         tx_packet(HEADER_LENGTH + tx_data_length, packet);       // Config CC1101 for data tranceive
506     }
507
508     //*******************************************************
509     //*** private functions *********************************
```

```
510 //*******************************************************
511
512 //******************************************************
513 // This function sets the CC1101 in the TX mode
514 //******************************************************
515 void cc1101_set_tx(unsigned long brate) {
516
517     cc1101_reset();             // Reset chip and go to idle state
518     cc1101_config_tx(brate);    // Configure the CC1101 for sending packets
519 }
520
521 //******************************************************
522 // This function sets the CC1101 in the RX mode
523 //******************************************************
524 void cc1101_set_rx(unsigned long brate) {
525
526     cc1101_reset();             // Reset chip and go to idle state
527     cc1101_config_rx(brate);    // Configure the CC1101 for receiving packets
528 }
529
530 //******************************************************
531 // This function resets the CC1101
532 //******************************************************
533 void cc1101_reset(void) {
534     uint32_t spiSendData[0];
535     uint32_t index = 0;
536
537     spiSendData[0] = 0x30;
538     SPI_send_data(1, spiSendData);
539
540 //  // TODO: Ändern in 5ms
541 //  delay_ms(1);    // Time to reset
542
543     for(index=0;index<=1200;index++)
544         asm("NOP");
545 }
546
547 //******************************************************
548 // This starts the TX Mode and TX the data
549 //******************************************************
550 void cc1101_tx(unsigned long packetes_to_tx) {
551     uint32_t spiSendData[2];
552
553     // DATA packet length
554     spiSendData[0] = CC1101_CR_PKTLEN;
555     spiSendData[1] = packetes_to_tx;
556     SPI_send_data(2, spiSendData);
557
558     // TX state
559     spiSendData[0] = CC1101_CS_STX;
560     SPI_send_data(1, spiSendData);
561
562     // Wait 5 ms for TX finished
563     delay_ms(5);
564 }
565
566 //******************************************************
567 // This function strobes the command to clear the TX FIFO
568 //******************************************************
569 void cc1101_clear_tx_fifo(void) {
570     uint32_t spiSendData[1];
571
572     spiSendData[0] = CC1101_CS_SFTX;
573     SPI_send_data(1, spiSendData);
574 }
575
576 //******************************************************
577 // This function strobes the command to fill the TX FIFO
578 //******************************************************
579 void cc1101_fill_tx_fifo(uint8_t * buffer, uint8_t length) {
580      uint32_t i;
581
582      uint32_t spiSendData[5];
583
584     // Clear TX FIFO
585     cc1101_clear_tx_fifo();
586
587     // Transfer the bytes via SPI to the TX fifo of the transceiver
588     spiSendData[0] = CC1101_ML_TXFIFO_BURST | CC1101_WRITE_BURST;
589     for (i = 1; i < length+1; i++) {
590         spiSendData[i] = buffer[i-1];
591     }
592
593     SPI_send_data(5, spiSendData);
594 }
```

```
595
596  //*****************************************************
597  // This function strobes the command to go in the idle mode
598  //*****************************************************
599  void cc1101_idle(void) {
600      uint32_t spiSendData[1];
601
602      spiSendData[0] = CC1101_CS_SIDLE;
603      SPI_send_data(1, spiSendData);
604  }
605
606  //*****************************************************
607  // This function sets the CRC Mode in the CC1101
608  //*****************************************************
609  void cc1101_enable_crc(void) {
610      // Read the current state of the Register, that includes the CRC Bit
611      uint8_t current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
612
613      uint32_t spiSendData[3];
614
615      spiSendData[0] = CC1101_CR_PKTCTRL0;
616      spiSendData[1] = current | 0x04;
617      SPI_send_data(2, spiSendData);
618  }
619
620  //*****************************************************
621  // This function readout a register in the CC1101 and
622  // return this
623  //*****************************************************
624  uint8_t cc1101_spi_read_register(uint8_t address) {
625      uint32_t x[7]={0};
626
627      CS_ENABLE;
628      SSIDataPutNonBlocking(SSI3_BASE, (address | CC1101_READ_SINGLE));
629
630      // Clear the RX FIFO
631      while(SSIDataGetNonBlocking(SSI3_BASE, &x[0]));
632
633      // Wait untill SSI3 is ready
634      while(SSIBusy(SSI3_BASE));
635
636      SSIDataPutNonBlocking(SSI3_BASE, 0x00);
637      SSIDataGetNonBlocking(SSI3_BASE,&x[0]);
638
639      x[0] &= 0x00FF;
640
641      // Wait untill SSI3 is ready
642      while(SSIBusy(SSI3_BASE));
643
644      CS_DISABLE;
645
646      return x[0];
647  }
648
649  //*****************************************************
650  // This function readout a burst register in the CC1101 and
651  // return this
652  //*****************************************************
653  void cc1101_spi_read_register_burst(uint8_t address, uint8_t *buffer, uint8_t count) {
654      uint8_t i = 0;
655      uint32_t x;
656
657      CS_ENABLE;
658      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 1);
659
660      SSIDataPutNonBlocking(SSI3_BASE, (address | CC1101_READ_BURST));
661
662      // Clear the RX FIFO
663      while(SSIDataGetNonBlocking(SSI3_BASE, &x));
664
665      // Wait untill SSI3 is ready
666      while(SSIBusy(SSI3_BASE));
667
668      // Dummy read
669      SSIDataPutNonBlocking(SSI3_BASE, 0x00);
670      SSIDataGetNonBlocking(SSI3_BASE,&x);
671
672      // Wait untill SSI3 is ready
673      while(SSIBusy(SSI3_BASE));
674
675      for (i = 0; i < count; i++) {
676
677          SSIDataPutNonBlocking(SSI3_BASE, 0x00);
678          SSIDataGetNonBlocking(SSI3_BASE,&x);
679
```

```
680             buffer[i] = x & 0x00FF;
681 
682             // Wait untill SSI3 is ready
683             while(SSIBusy(SSI3_BASE));
684         }
685         GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0);
686 
687         CS_DISABLE;
688 
689     }
690 
691     void cc1101_config_tx(uint8_t brate) {
692         uint32_t spiSendData[4];
693 
694         // Rising edge on GDO0 when packet received and CRC check OK
695         // (Deasserted when first byte is read from RX FIFO)
696         spiSendData[0] = CC1101_CR_IOCFG0;
697         spiSendData[1] = 0x07;
698         SPI_send_data(2, spiSendData);
699 
700         // Rising edge on GDO2 when sync word has been received
701         spiSendData[0] = CC1101_CR_IOCFG2;
702         spiSendData[1] = 0x06;
703         SPI_send_data(2, spiSendData);
704 
705         // The 4 SYNC bytes: 0x12 0x09 (repeated once)
706         spiSendData[0] = CC1101_CR_SYNC1;
707         spiSendData[1] = 0x81;
708         SPI_send_data(2, spiSendData);
709 
710         spiSendData[0] = CC1101_CR_SYNC0;
711         spiSendData[1] = 0x81;
712         SPI_send_data(2, spiSendData);
713 
714         // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
715         spiSendData[0] = CC1101_CR_PKTCTRL1;
716         spiSendData[1] = 0x0A;
717         SPI_send_data(2, spiSendData);
718 
719         // No whitening, FIFO mode, CRC disable, Fixed packet length
720         spiSendData[0] = CC1101_CR_PKTCTRL0;
721         spiSendData[1] = 0x04;
722         SPI_send_data(2, spiSendData);//0x04 für CRC!
723 
724         // Device Address
725         spiSendData[0] = CC1101_CR_ADDR;
726         spiSendData[1] = 0x00;
727         SPI_send_data(2, spiSendData);
728 
729         // Channel 0
730         spiSendData[0] = CC1101_CR_CHANNR;
731         spiSendData[1] = 0x00;
732         SPI_send_data(2, spiSendData);
733 
734         // IF frequency: 152.34375 kHz
735         spiSendData[0] = CC1101_CR_FSCTRL1;
736         spiSendData[1] = 0x06;
737         SPI_send_data(2, spiSendData);
738 
739         // Carrier frequency: 433.999969 MHz
740         spiSendData[0] = CC1101_CR_FREQ2;
741         spiSendData[1] = 0x10;
742         SPI_send_data(2, spiSendData);
743 
744         spiSendData[0] = CC1101_CR_FREQ1;
745         spiSendData[1] = 0xB1;
746         SPI_send_data(2, spiSendData);
747 
748         spiSendData[0] = CC1101_CR_FREQ0;
749         spiSendData[1] = 0x3B;
750         SPI_send_data(2, spiSendData);
751 
752         switch(brate) {
753             /*****************************************************************************************
754              ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
755              ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
756              *****************************************************************************************/
757             case 40: {
758                 spiSendData[0] = CC1101_CR_MDMCFG4;
759                 spiSendData[1] = 0x8A;
760                 SPI_send_data(2, spiSendData);
761 
762                 spiSendData[0] = CC1101_CR_MDMCFG3;
763                 spiSendData[1] = 0x93;
764                 SPI_send_data(2, spiSendData);
```

```
765
766                spiSendData[0] = CC1101_CR_MDMCFG2;
767                spiSendData[1] = 0x33;
768                SPI_send_data(2, spiSendData);
769
770                spiSendData[0] = CC1101_CR_MDMCFG1;
771                spiSendData[1] = 0x22;
772                SPI_send_data(2, spiSendData);
773
774                spiSendData[0] = CC1101_CR_MDMCFG0;
775                spiSendData[1] = 0x7A;
776                SPI_send_data(2, spiSendData);
777            }break;
778
779            /*******************************************************************************************
780             ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
781             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
782             *******************************************************************************************/
783            case 60: {
784                spiSendData[0] = CC1101_CR_MDMCFG4;
785                spiSendData[1] = 0x8B;
786                SPI_send_data(2, spiSendData);
787
788                spiSendData[0] = CC1101_CR_MDMCFG3;
789                spiSendData[1] = 0x2E;
790                SPI_send_data(2, spiSendData);
791
792                spiSendData[0] = CC1101_CR_MDMCFG2;
793                spiSendData[1] = 0x33;
794                SPI_send_data(2, spiSendData);
795
796                spiSendData[0] = CC1101_CR_MDMCFG1;
797                spiSendData[1] = 0x22;
798                SPI_send_data(2, spiSendData);
799
800                spiSendData[0] = CC1101_CR_MDMCFG0;
801                spiSendData[1] = 0x7A;
802                SPI_send_data(2, spiSendData);
803            }break;
804
805            /*******************************************************************************************
806             ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
807             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
808             *******************************************************************************************/
809            case 80: {
810                spiSendData[0] = CC1101_CR_MDMCFG4;
811                spiSendData[1] = 0x8B;
812                SPI_send_data(2, spiSendData);
813
814                spiSendData[0] = CC1101_CR_MDMCFG3;
815                spiSendData[1] = 0x93;
816                SPI_send_data(2, spiSendData);
817
818                spiSendData[0] = CC1101_CR_MDMCFG2;
819                spiSendData[1] = 0x33;
820                SPI_send_data(2, spiSendData);
821
822                spiSendData[0] = CC1101_CR_MDMCFG1;
823                spiSendData[1] = 0x22;
824                SPI_send_data(2, spiSendData);
825
826                spiSendData[0] = CC1101_CR_MDMCFG0;
827                spiSendData[1] = 0x7A;
828                SPI_send_data(2, spiSendData);
829            }break;
830
831            /*******************************************************************************************
832             ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
833             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
834             *******************************************************************************************/
835            case 100: {
836                spiSendData[0] = CC1101_CR_MDMCFG4;
837                spiSendData[1] = 0x8B;
838                SPI_send_data(2, spiSendData);
839
840                spiSendData[0] = CC1101_CR_MDMCFG3;
841                spiSendData[1] = 0xF8;
842                SPI_send_data(2, spiSendData);
843
844                spiSendData[0] = CC1101_CR_MDMCFG2;
845                spiSendData[1] = 0x33;
846                SPI_send_data(2, spiSendData);
847
848                spiSendData[0] = CC1101_CR_MDMCFG1;
849                spiSendData[1] = 0x22;
```

```
850                 SPI_send_data(2, spiSendData);
851
852                 spiSendData[0] = CC1101_CR_MDMCFG0;
853                 spiSendData[1] = 0x7A;
854                 SPI_send_data(2, spiSendData);
855             }break;
856
857             /*****************************************************************************************
858              ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
859              ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
860              *****************************************************************************************/
861             case 120: {
862                 spiSendData[0] = CC1101_CR_MDMCFG4;
863                 spiSendData[1] = 0x8C;
864                 SPI_send_data(2, spiSendData);
865
866                 spiSendData[0] = CC1101_CR_MDMCFG3;
867                 spiSendData[1] = 0x2E;
868                 SPI_send_data(2, spiSendData);
869
870                 spiSendData[0] = CC1101_CR_MDMCFG2;
871                 spiSendData[1] = 0x33;
872                 SPI_send_data(2, spiSendData);
873
874                 spiSendData[0] = CC1101_CR_MDMCFG1;
875                 spiSendData[1] = 0x22;
876                 SPI_send_data(2, spiSendData);
877
878                 spiSendData[0] = CC1101_CR_MDMCFG0;
879                 spiSendData[1] = 0x7A;
880                 SPI_send_data(2, spiSendData);
881             }break;
882
883             /*****************************************************************************************
884              ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
885              ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
886              *****************************************************************************************/
887             case 140: {
888                 spiSendData[0] = CC1101_CR_MDMCFG4;
889                 spiSendData[1] = 0x8C;
890                 SPI_send_data(2, spiSendData);
891
892                 spiSendData[0] = CC1101_CR_MDMCFG3;
893                 spiSendData[1] = 0x61;
894                 SPI_send_data(2, spiSendData);
895
896                 spiSendData[0] = CC1101_CR_MDMCFG2;
897                 spiSendData[1] = 0x33;
898                 SPI_send_data(2, spiSendData);
899
900                 spiSendData[0] = CC1101_CR_MDMCFG1;
901                 spiSendData[1] = 0x22;
902                 SPI_send_data(2, spiSendData);
903
904                 spiSendData[0] = CC1101_CR_MDMCFG0;
905                 spiSendData[1] = 0x7A;
906                 SPI_send_data(2, spiSendData);
907             }break;
908
909             /*****************************************************************************************
910              ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
911              ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
912              *****************************************************************************************/
913             case 160: {
914                 spiSendData[0] = CC1101_CR_MDMCFG4;
915                 spiSendData[1] = 0x8C;
916                 SPI_send_data(2, spiSendData);
917
918                 spiSendData[0] = CC1101_CR_MDMCFG3;
919                 spiSendData[1] = 0x93;
920                 SPI_send_data(2, spiSendData);
921
922                 spiSendData[0] = CC1101_CR_MDMCFG2;
923                 spiSendData[1] = 0x33;
924                 SPI_send_data(2, spiSendData);
925
926                 spiSendData[0] = CC1101_CR_MDMCFG1;
927                 spiSendData[1] = 0x22;
928                 SPI_send_data(2, spiSendData);
929
930                 spiSendData[0] = CC1101_CR_MDMCFG0;
931                 spiSendData[1] = 0x7A;
932                 SPI_send_data(2, spiSendData);
933             }break;
934
```

```
935         /*******************************************************************************************
936          ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
937          ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
938          ******************************************************************************************/
939         case 180: {
940             spiSendData[0] = CC1101_CR_MDMCFG4;
941             spiSendData[1] = 0x8C;
942             SPI_send_data(2, spiSendData);
943 
944             spiSendData[0] = CC1101_CR_MDMCFG3;
945             spiSendData[1] = 0xC6;
946             SPI_send_data(2, spiSendData);
947 
948             spiSendData[0] = CC1101_CR_MDMCFG2;
949             spiSendData[1] = 0x33;
950             SPI_send_data(2, spiSendData);
951 
952             spiSendData[0] = CC1101_CR_MDMCFG1;
953             spiSendData[1] = 0x22;
954             SPI_send_data(2, spiSendData);
955 
956             spiSendData[0] = CC1101_CR_MDMCFG0;
957             spiSendData[1] = 0x7A;
958             SPI_send_data(2, spiSendData);
959         }break;
960 
961     }
962 
963     // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
964     spiSendData[0] = CC1101_CR_MCSM0;
965     spiSendData[1] = 0x10;
966     SPI_send_data(2, spiSendData);
967 
968     // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
969     spiSendData[0] = CC1101_CR_FOCCFG;
970     spiSendData[1] = 0x16;
971     SPI_send_data(2, spiSendData);
972 
973     spiSendData[0] = 0x7E;
974     spiSendData[1] = 0x12;
975     spiSendData[2] = 0xc0;
976     spiSendData[3] = 0x02;
977     SPI_send_data(4, spiSendData);
978 
979     // 10 dBm output power
980     spiSendData[0] = CC1101_CR_FREND0;
981     spiSendData[1] = 0x11;
982     SPI_send_data(2, spiSendData);
983 }
984 
985 void cc1101_config_rx(uint8_t brate) {
986     uint32_t spiSendData[3];
987 
988     // Rising edge on GDO0 when packet received and CRC check OK
989     // (Deasserted when first byte is read from RX FIFO)
990     spiSendData[0] = CC1101_CR_IOCFG0;
991     spiSendData[1] = 0x07;
992     SPI_send_data(2, spiSendData);
993 
994     // Rising edge on GDO2 when sync word has been received
995     spiSendData[0] = CC1101_CR_IOCFG2;
996     spiSendData[1] = 0x06;
997     SPI_send_data(2, spiSendData);
998 
999     // RX FIFO and TX FIFO Thresholds
1000    spiSendData[0] = CC1101_CR_FIFOTHR;
1001    spiSendData[1] = 0x47;
1002    SPI_send_data(2, spiSendData);
1003 
1004    // The 4 SYNC bytes: 0x12 0x09 (repeated once)
1005    spiSendData[0] = CC1101_CR_SYNC1;
1006    spiSendData[1] = 0x81;
1007    SPI_send_data(2, spiSendData);
1008 
1009    spiSendData[0] = CC1101_CR_SYNC0;
1010    spiSendData[1] = 0x81;
1011    SPI_send_data(2, spiSendData);
1012 
1013    // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
1014    spiSendData[0] = CC1101_CR_PKTCTRL1;
1015    spiSendData[1] = 0x04;
1016    SPI_send_data(2, spiSendData);
1017 
1018    // No whitening, FIFO mode, CRC disable, Fixed packet length
1019    spiSendData[0] = CC1101_CR_PKTCTRL0;
```

```
1020        spiSendData[1] = 0x00;
1021        SPI_send_data(2, spiSendData);
1022
1023        // Device Address
1024        spiSendData[0] = CC1101_CR_ADDR;
1025        spiSendData[1] = 0x00;
1026        SPI_send_data(2, spiSendData);
1027
1028        // Channel 0
1029        spiSendData[0] = CC1101_CR_CHANNR;
1030        spiSendData[1] = 0x00;
1031        SPI_send_data(2, spiSendData);
1032
1033        // IF frequency: 152.34375 kHz
1034        spiSendData[0] = CC1101_CR_FSCTRL1;
1035        spiSendData[1] = 0x06;
1036        SPI_send_data(2, spiSendData);
1037
1038        // Carrier frequency: 433.999969 MHz
1039        spiSendData[0] = CC1101_CR_FREQ2;
1040        spiSendData[1] = 0x10;
1041        SPI_send_data(2, spiSendData);
1042
1043        spiSendData[0] = CC1101_CR_FREQ1;
1044        spiSendData[1] = 0xB1;
1045        SPI_send_data(2, spiSendData);
1046
1047        spiSendData[0] = CC1101_CR_FREQ0;
1048        spiSendData[1] = 0x3B;
1049        SPI_send_data(2, spiSendData);
1050
1051        switch(brate) {
1052            /*******************************************************************************************
1053            ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
1054            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1055            *******************************************************************************************/
1056            case 40: {
1057                spiSendData[0] = CC1101_CR_MDMCFG4;
1058                spiSendData[1] = 0x8A;
1059                SPI_send_data(2, spiSendData);
1060
1061                spiSendData[0] = CC1101_CR_MDMCFG3;
1062                spiSendData[1] = 0x93;
1063                SPI_send_data(2, spiSendData);
1064
1065                spiSendData[0] = CC1101_CR_MDMCFG2;
1066                spiSendData[1] = 0x33;
1067                SPI_send_data(2, spiSendData);
1068
1069                spiSendData[0] = CC1101_CR_MDMCFG1;
1070                spiSendData[1] = 0x22;
1071                SPI_send_data(2, spiSendData);
1072
1073                spiSendData[0] = CC1101_CR_MDMCFG0;
1074                spiSendData[1] = 0x7A;
1075                SPI_send_data(2, spiSendData);
1076            }break;
1077
1078            /*******************************************************************************************
1079            ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
1080            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1081            ********************************************************************************************/
1082            case 60: {
1083                spiSendData[0] = CC1101_CR_MDMCFG4;
1084                spiSendData[1] = 0x8B;
1085                SPI_send_data(2, spiSendData);
1086
1087                spiSendData[0] = CC1101_CR_MDMCFG3;
1088                spiSendData[1] = 0x2E;
1089                SPI_send_data(2, spiSendData);
1090
1091                spiSendData[0] = CC1101_CR_MDMCFG2;
1092                spiSendData[1] = 0x33;
1093                SPI_send_data(2, spiSendData);
1094
1095                spiSendData[0] = CC1101_CR_MDMCFG1;
1096                spiSendData[1] = 0x22;
1097                SPI_send_data(2, spiSendData);
1098
1099                spiSendData[0] = CC1101_CR_MDMCFG0;
1100                spiSendData[1] = 0x7A;
1101                SPI_send_data(2, spiSendData);
1102            }break;
1103
1104            /*******************************************************************************************
```

```
1105            ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
1106            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1107            ***************************************************************************************/
1108            case 80: {
1109                spiSendData[0] = CC1101_CR_MDMCFG4;
1110                spiSendData[1] = 0x8B;
1111                SPI_send_data(2, spiSendData);
1112
1113                spiSendData[0] = CC1101_CR_MDMCFG3;
1114                spiSendData[1] = 0x93;
1115                SPI_send_data(2, spiSendData);
1116
1117                spiSendData[0] = CC1101_CR_MDMCFG2;
1118                spiSendData[1] = 0x33;
1119                SPI_send_data(2, spiSendData);
1120
1121                spiSendData[0] = CC1101_CR_MDMCFG1;
1122                spiSendData[1] = 0x22;
1123                SPI_send_data(2, spiSendData);
1124
1125                spiSendData[0] = CC1101_CR_MDMCFG0;
1126                spiSendData[1] = 0x7A;
1127                SPI_send_data(2, spiSendData);
1128            }break;
1129
1130            /***************************************************************************************
1131            ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
1132            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1133            ***************************************************************************************/
1134            case 100: {
1135                spiSendData[0] = CC1101_CR_MDMCFG4;
1136                spiSendData[1] = 0x8B;
1137                SPI_send_data(2, spiSendData);
1138
1139                spiSendData[0] = CC1101_CR_MDMCFG3;
1140                spiSendData[1] = 0xF8;
1141                SPI_send_data(2, spiSendData);
1142
1143                spiSendData[0] = CC1101_CR_MDMCFG2;
1144                spiSendData[1] = 0x33;
1145                SPI_send_data(2, spiSendData);
1146
1147                spiSendData[0] = CC1101_CR_MDMCFG1;
1148                spiSendData[1] = 0x22;
1149                SPI_send_data(2, spiSendData);
1150
1151                spiSendData[0] = CC1101_CR_MDMCFG0;
1152                spiSendData[1] = 0x7A;
1153                SPI_send_data(2, spiSendData);
1154            }break;
1155
1156            /***************************************************************************************
1157            ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
1158            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1159            ***************************************************************************************/
1160            case 120: {
1161                spiSendData[0] = CC1101_CR_MDMCFG4;
1162                spiSendData[1] = 0x8C;
1163                SPI_send_data(2, spiSendData);
1164
1165                spiSendData[0] = CC1101_CR_MDMCFG3;
1166                spiSendData[1] = 0x2E;
1167                SPI_send_data(2, spiSendData);
1168
1169                spiSendData[0] = CC1101_CR_MDMCFG2;
1170                spiSendData[1] = 0x33;
1171                SPI_send_data(2, spiSendData);
1172
1173                spiSendData[0] = CC1101_CR_MDMCFG1;
1174                spiSendData[1] = 0x22;
1175                SPI_send_data(2, spiSendData);
1176
1177                spiSendData[0] = CC1101_CR_MDMCFG0;
1178                spiSendData[1] = 0x7A;
1179                SPI_send_data(2, spiSendData);
1180            }break;
1181
1182            /***************************************************************************************
1183            ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
1184            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1185            ***************************************************************************************/
1186            case 140: {
1187                spiSendData[0] = CC1101_CR_MDMCFG4;
1188                spiSendData[1] = 0x8C;
1189                SPI_send_data(2, spiSendData);
```

```
1190
1191                spiSendData[0] = CC1101_CR_MDMCFG3;
1192                spiSendData[1] = 0x61;
1193                SPI_send_data(2, spiSendData);
1194
1195                spiSendData[0] = CC1101_CR_MDMCFG2;
1196                spiSendData[1] = 0x33;
1197                SPI_send_data(2, spiSendData);
1198
1199                spiSendData[0] = CC1101_CR_MDMCFG1;
1200                spiSendData[1] = 0x22;
1201                SPI_send_data(2, spiSendData);
1202
1203                spiSendData[0] = CC1101_CR_MDMCFG0;
1204                spiSendData[1] = 0x7A;
1205                SPI_send_data(2, spiSendData);
1206            }break;
1207
1208            /*******************************************************************************************
1209            ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
1210            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1211            *******************************************************************************************/
1212            case 160: {
1213                spiSendData[0] = CC1101_CR_MDMCFG4;
1214                spiSendData[1] = 0x8C;
1215                SPI_send_data(2, spiSendData);
1216
1217                spiSendData[0] = CC1101_CR_MDMCFG3;
1218                spiSendData[1] = 0x93;
1219                SPI_send_data(2, spiSendData);
1220
1221                spiSendData[0] = CC1101_CR_MDMCFG2;
1222                spiSendData[1] = 0x33;
1223                SPI_send_data(2, spiSendData);
1224
1225                spiSendData[0] = CC1101_CR_MDMCFG1;
1226                spiSendData[1] = 0x22;
1227                SPI_send_data(2, spiSendData);
1228
1229                spiSendData[0] = CC1101_CR_MDMCFG0;
1230                spiSendData[1] = 0x7A;
1231                SPI_send_data(2, spiSendData);
1232            }break;
1233
1234            /*******************************************************************************************
1235            ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
1236            ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
1237            *******************************************************************************************/
1238            case 180: {
1239                spiSendData[0] = CC1101_CR_MDMCFG4;
1240                spiSendData[1] = 0x8C;
1241                SPI_send_data(2, spiSendData);
1242
1243                spiSendData[0] = CC1101_CR_MDMCFG3;
1244                spiSendData[1] = 0xC6;
1245                SPI_send_data(2, spiSendData);
1246
1247                spiSendData[0] = CC1101_CR_MDMCFG2;
1248                spiSendData[1] = 0x33;
1249                SPI_send_data(2, spiSendData);
1250
1251                spiSendData[0] = CC1101_CR_MDMCFG1;
1252                spiSendData[1] = 0x22;
1253                SPI_send_data(2, spiSendData);
1254
1255                spiSendData[0] = CC1101_CR_MDMCFG0;
1256                spiSendData[1] = 0x7A;
1257                SPI_send_data(2, spiSendData);
1258            }break;
1259
1260        }
1261
1262        // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
1263        spiSendData[0] = CC1101_CR_MCSM0;
1264        spiSendData[1] = 0x10;
1265        SPI_send_data(2, spiSendData);
1266
1267        // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
1268        spiSendData[0] = CC1101_CR_FOCCFG;
1269        spiSendData[1] = 0x16;
1270        SPI_send_data(2, spiSendData);
1271
1272        // 10 dBm output power
1273        spiSendData[0] = 0x7E;
1274        spiSendData[1] = 0x12;
```

```
1275     spiSendData[2] = 0xc0;
1276     SPI_send_data(3, spiSendData);
1277
1278     spiSendData[0] = CC1101_CR_FREND0;
1279     spiSendData[1] = 0x11;
1280     SPI_send_data(2, spiSendData);
1281 }
1282
1283 void cc1101_config_burst_tx(void) {
1284     uint32_t spiSendData[3];
1285
1286     // GDO2 Output Pin Configuration 2
1287     // 0x2E -> High impedance (3-state)
1288     spiSendData[0] = CC1101_CR_IOCFG2;
1289     spiSendData[1] = 0x2E;
1290     SPI_send_data(2, spiSendData);
1291
1292     // GDO0 Output Pin Configuration 0
1293     // 0x2E -> High impedance (3-state)
1294     spiSendData[0] = CC1101_CR_IOCFG0;
1295     spiSendData[1] = 0x2E;
1296     SPI_send_data(2, spiSendData);
1297
1298     // Asynchronous serial mode, Infinite packet length mode
1299     spiSendData[0] = CC1101_CR_PKTCTRL0;
1300     spiSendData[1] = 0x32;
1301     SPI_send_data(2, spiSendData);
1302
1303     // Channel Number
1304     spiSendData[0] = CC1101_CR_CHANNR;
1305     spiSendData[1] = 0x00;
1306     SPI_send_data(2, spiSendData);
1307
1308     // Carrier frequency
1309     spiSendData[0] = CC1101_CR_FSCTRL0;
1310     spiSendData[1] = 0x06;
1311     SPI_send_data(2, spiSendData);
1312
1313     // Frequency Control Word, High Byte
1314     spiSendData[0] = CC1101_CR_FREQ2;
1315     spiSendData[1] = 0x10;
1316     SPI_send_data(2, spiSendData);
1317
1318     // Frequency Control Word, Middle Byte
1319     spiSendData[0] = CC1101_CR_FREQ1;
1320     spiSendData[1] = 0xB1;
1321     SPI_send_data(2, spiSendData);
1322
1323     //Frequency Control Word, LOW Byte
1324     spiSendData[0] = CC1101_CR_FREQ0;
1325     spiSendData[1] = 0x3B;
1326     SPI_send_data(2, spiSendData);
1327
1328     /**********************************************/
1329     // Modem Configuration 4
1330     // 250kBaud              -> 0xxD
1331     // RX Filter 58.035714   -> 0xFx
1332     spiSendData[0] = CC1101_CR_MDMCFG4;
1333     spiSendData[1] = 0xFD;
1334     SPI_send_data(2, spiSendData);
1335
1336     // Modem Configuration 3
1337     // 250kBaud -> 0x3B
1338     spiSendData[0] = CC1101_CR_MDMCFG3;
1339     spiSendData[1] = 0x3B;
1340     SPI_send_data(2, spiSendData);
1341
1342     // Modem Configuration 2
1343     // OOK -> 0x30
1344     spiSendData[0] = CC1101_CR_MDMCFG2;
1345     spiSendData[1] = 0x30;
1346     SPI_send_data(2, spiSendData);
1347
1348     // Modem Configuration 1
1349     // Channel spacing 199.951172kHz -> 0x22
1350     spiSendData[0] = CC1101_CR_MDMCFG1;
1351     spiSendData[1] = 0x00;
1352     SPI_send_data(2, spiSendData);
1353
1354     /**********************************************/
1355     // Modem Deviation Setting
1356     spiSendData[0] = CC1101_CR_DEVIATN;
1357     spiSendData[1] = 0x15;
1358     SPI_send_data(2, spiSendData);
1359
```

```
1360     // Main Radio Control State Machine Configuration 0
1361     spiSendData[0] = CC1101_CR_MCSM0;
1362     spiSendData[1] = 0x18;
1363     SPI_send_data(2, spiSendData);
1364
1365     // Frequency Offset Compensation Configuration
1366     spiSendData[0] = CC1101_CR_FOCCFG;
1367     spiSendData[1] = 0x16;
1368     SPI_send_data(2, spiSendData);
1369
1370     // 10 dBm output power
1371     spiSendData[0] = 0x7E;
1372     spiSendData[1] = 0x12;
1373     spiSendData[2] = 0xc0;
1374     SPI_send_data(3, spiSendData);
1375
1376     spiSendData[0] = CC1101_CR_FREND0;
1377     spiSendData[1] = 0x11;
1378     SPI_send_data(2, spiSendData);
1379
1380 }
1381
1382 void cc1101_config_cali_burst(void) {
1383     uint32_t spiSendData[3];
1384
1385     // GDO2 Output Pin Configuration 2
1386     // 0x2E -> High impedance (3-state)
1387     spiSendData[0] = CC1101_CR_IOCFG2;
1388     spiSendData[1] = 0x2E;
1389     SPI_send_data(2, spiSendData);
1390
1391     // GDO0 Output Pin Configuration 0
1392     // 0x2E -> High impedance (3-state)
1393     spiSendData[0] = CC1101_CR_IOCFG0;
1394     spiSendData[1] = 0x2E;
1395     SPI_send_data(2, spiSendData);
1396
1397     // Asynchronous serial mode, Infinite packet length mode
1398     spiSendData[0] = CC1101_CR_PKTCTRL0;
1399     spiSendData[1] = 0x32;
1400     SPI_send_data(2, spiSendData);
1401
1402     // Channel Number
1403     spiSendData[0] = CC1101_CR_CHANNR;
1404     spiSendData[1] = 0x00;
1405     SPI_send_data(2, spiSendData);
1406
1407     // Carrier frequency
1408     spiSendData[0] = CC1101_CR_FSCTRL0;
1409     spiSendData[1] = 0x06;
1410     SPI_send_data(2, spiSendData);
1411
1412     // Frequency Control Word, High Byte
1413     spiSendData[0] = CC1101_CR_FREQ2;
1414     spiSendData[1] = 0x10;
1415     SPI_send_data(2, spiSendData);
1416
1417     // Frequency Control Word, Middle Byte
1418     spiSendData[0] = CC1101_CR_FREQ1;
1419     spiSendData[1] = 0xB1;
1420     SPI_send_data(2, spiSendData);
1421
1422     //Frequency Control Word, LOW Byte
1423     spiSendData[0] = CC1101_CR_FREQ0;
1424     spiSendData[1] = 0x3B;
1425     SPI_send_data(2, spiSendData);
1426
1427     /***********************************************/
1428     // Modem Configuration 4
1429     // 250kBaud          -> 0xxD
1430     // RX Filter 58.035714 -> 0xFx
1431     spiSendData[0] = CC1101_CR_MDMCFG4;
1432     spiSendData[1] = 0xFD;
1433     SPI_send_data(2, spiSendData);
1434
1435     // Modem Configuration 3
1436     // 250kBaud -> 0x3B
1437     spiSendData[0] = CC1101_CR_MDMCFG3;
1438     spiSendData[1] = 0x3B;
1439     SPI_send_data(2, spiSendData);
1440
1441     // Modem Configuration 2
1442     // OOK -> 0x30
1443     spiSendData[0] = CC1101_CR_MDMCFG2;
1444     spiSendData[1] = 0x30;
```

```
1445     SPI_send_data(2, spiSendData);
1446
1447     // Modem Configuration 1
1448     // Channel spacing 199.951172kHz -> 0x22
1449     spiSendData[0] = CC1101_CR_MDMCFG1;
1450     spiSendData[1] = 0x00;
1451     SPI_send_data(2, spiSendData);
1452
1453     /************************************************/
1454     // Modem Deviation Setting
1455     spiSendData[0] = CC1101_CR_DEVIATN;
1456     spiSendData[1] = 0x15;
1457     SPI_send_data(2, spiSendData);
1458
1459     // Main Radio Control State Machine Configuration 0
1460     spiSendData[0] = CC1101_CR_MCSM0;
1461     spiSendData[1] = 0x18;
1462     SPI_send_data(2, spiSendData);
1463
1464     // Frequency Offset Compensation Configuration
1465     spiSendData[0] = CC1101_CR_FOCCFG;
1466     spiSendData[1] = 0x16;
1467     SPI_send_data(2, spiSendData);
1468
1469     // 10 dBm output power
1470     spiSendData[0] = 0x7E;
1471     spiSendData[1] = 0x12;
1472     spiSendData[2] = 0xc0;
1473     SPI_send_data(3, spiSendData);
1474
1475     spiSendData[0] = CC1101_CR_FREND0;
1476     spiSendData[1] = 0x11;
1477     SPI_send_data(2, spiSendData);
1478
1479 }
1480
1481 void cc1101_config_burst_rx(void) {
1482     uint32_t spiSendData[3];
1483
1484     // GDO2 Output Pin Configuration 2
1485     spiSendData[0] = CC1101_CR_IOCFG2;
1486     spiSendData[1] = 0x0D;
1487     SPI_send_data(2, spiSendData);
1488
1489     // GDO2 Output Pin Configuration 1
1490     // 0x2E -> High impedance (3-state)
1491     spiSendData[0] = CC1101_CR_IOCFG1;
1492     spiSendData[1] = 0x2E;
1493     SPI_send_data(2, spiSendData);
1494
1495     // GDO1 Output Pin Configuration 0
1496     spiSendData[0] = CC1101_CR_IOCFG0;
1497     spiSendData[1] = 0x36;
1498     SPI_send_data(2, spiSendData);
1499
1500     // RX FIFO and TX FIFO Thresholds
1501     spiSendData[0] = CC1101_CR_FIFOTHR;
1502     spiSendData[1] = 0x47;
1503     SPI_send_data(2, spiSendData);
1504
1505     // Asynchronous serial mode, Infinite packet length mode
1506     spiSendData[0] = CC1101_CR_PKTCTRL0;
1507     spiSendData[1] = 0x32;
1508     SPI_send_data(2, spiSendData);
1509
1510     // Carrier frequency
1511     spiSendData[0] = CC1101_CR_FSCTRL1;
1512     spiSendData[1] = 0x06;
1513     SPI_send_data(2, spiSendData);
1514
1515     // Frequency Control Word, High Byte
1516     spiSendData[0] = CC1101_CR_FREQ2;
1517     spiSendData[1] = 0x10;
1518     SPI_send_data(2, spiSendData);
1519
1520     // Frequency Control Word, Middle Byte
1521     spiSendData[0] = CC1101_CR_FREQ1;
1522     spiSendData[1] = 0xB1;
1523     SPI_send_data(2, spiSendData);
1524
1525     //Frequency Control Word, LOW Byte
1526     spiSendData[0] = CC1101_CR_FREQ0;
1527     spiSendData[1] = 0x3B;
1528     SPI_send_data(2, spiSendData);
1529
```

```
1530        /*************************************************/
1531        // Modem Configuration 4
1532        // 250kBaud              -> 0xxD
1533        // RX Filter 58.035714   -> 0xFx
1534        spiSendData[0] = CC1101_CR_MDMCFG4;
1535        spiSendData[1] = 0x5D;
1536        SPI_send_data(2, spiSendData);
1537
1538        // Modem Configuration 3
1539        // 250kBaud -> 0x3B
1540        spiSendData[0] = CC1101_CR_MDMCFG3;
1541        spiSendData[1] = 0x3B;
1542        SPI_send_data(2, spiSendData);
1543
1544        // Modem Configuration 2
1545        // OOK -> 0x30
1546        spiSendData[0] = CC1101_CR_MDMCFG2;
1547        spiSendData[1] = 0x30;
1548        SPI_send_data(2, spiSendData);
1549
1550        // Modem Configuration 1
1551        // Channel spacing 199.951172kHz -> 0x22
1552        spiSendData[0] = CC1101_CR_MDMCFG1;
1553        spiSendData[1] = 0x22;
1554        SPI_send_data(2, spiSendData);
1555
1556        // Modem Configuration 0
1557        spiSendData[0] = CC1101_CR_MDMCFG0;
1558        spiSendData[1] = 0xF8;
1559        SPI_send_data(2, spiSendData);
1560
1561        /*************************************************/
1562        // Modem Deviation Setting
1563        spiSendData[0] = CC1101_CR_DEVIATN;
1564        spiSendData[1] = 0x15;
1565        SPI_send_data(2, spiSendData);
1566
1567        // Main Radio Control State Machine Configuration 2
1568        spiSendData[0] = CC1101_CR_MCSM2;
1569        spiSendData[1] = 0x07;
1570        SPI_send_data(2, spiSendData);
1571
1572        // Main Radio Control State Machine Configuration 1
1573        spiSendData[0] = CC1101_CR_MCSM1;
1574        spiSendData[1] = 0x30;
1575        SPI_send_data(2, spiSendData);
1576
1577        // Main Radio Control State Machine Configuration 0
1578        spiSendData[0] = CC1101_CR_MCSM0;
1579        spiSendData[1] = 0x18;
1580        SPI_send_data(2, spiSendData);
1581
1582        // Frequency Offset Compensation Configuration
1583        spiSendData[0] = CC1101_CR_FOCCFG;
1584        spiSendData[1] = 0x16;
1585        SPI_send_data(2, spiSendData);
1586
1587        // Bit Synchronization Configuration
1588        spiSendData[0] = CC1101_CR_BSCFG;
1589        spiSendData[1] = 0x6C;
1590        SPI_send_data(2, spiSendData);
1591
1592        // AGCCTRL2
1593        spiSendData[0] = CC1101_CR_AGCCTRL2;
1594        spiSendData[1] = 0x03;
1595        SPI_send_data(2, spiSendData);
1596
1597        // AGCCTRL1
1598        spiSendData[0] = CC1101_CR_AGCCTRL1;
1599        spiSendData[1] = 0x40;
1600        SPI_send_data(2, spiSendData);
1601
1602        // AGCCTRL0
1603        spiSendData[0] = CC1101_CR_AGCCTRL0;
1604        spiSendData[1] = 0x91;
1605        SPI_send_data(2, spiSendData);
1606
1607         // Wake On Radio Contro
1608        spiSendData[0] = CC1101_CR_WORCTRL;
1609        spiSendData[1] = 0x8F;
1610        SPI_send_data(2, spiSendData);
1611
1612        // Front End RX Configuration 1
1613        spiSendData[0] = CC1101_CR_FREND1;
1614        spiSendData[1] = 0x56;
```

```
1615        SPI_send_data(2, spiSendData);
1616
1617        // Front End RX Configuration 0
1618        spiSendData[0] = CC1101_CR_FREND0;
1619        spiSendData[1] = 0x11;
1620        SPI_send_data(2, spiSendData);
1621
1622        // 10 dBm output power
1623        spiSendData[0] = 0x7E;
1624        spiSendData[1] = 0x12;
1625        spiSendData[2] = 0xc0;
1626        SPI_send_data(3, spiSendData);
1627    }
1628
1629    void cc1101_config_burst_cali(void) {
1630        uint32_t spiSendData[3];
1631
1632        // GDO2 Output Pin Configuration 2
1633        spiSendData[0] = CC1101_CR_IOCFG2;
1634        spiSendData[1] = 0x0D;
1635        SPI_send_data(2, spiSendData);
1636
1637        // GDO0 Output Pin Configuration 0
1638        spiSendData[0] = CC1101_CR_IOCFG0;
1639        spiSendData[1] = 0x2E;
1640        SPI_send_data(2, spiSendData);
1641
1642        // RX FIFO and TX FIFO Thresholds
1643        // RX Attenuation 18dB -> 0x77
1644        // RX Attenuation 12dB -> 0x67
1645        // RX Attenuation  6dB -> 0x57
1646        // RX Attenuation  0dB -> 0x47
1647        spiSendData[0] = CC1101_CR_FIFOTHR;
1648        spiSendData[1] = 0x47;
1649        SPI_send_data(2, spiSendData);
1650
1651        // Asynchronous serial mode, Infinite packet length mode
1652        spiSendData[0] = CC1101_CR_PKTCTRL0;
1653        spiSendData[1] = 0x32;
1654        SPI_send_data(2, spiSendData);
1655
1656        // Frequency Synthesizer Control 1
1657        spiSendData[0] = CC1101_CR_FSCTRL1;
1658        spiSendData[1] = 0x06;
1659        SPI_send_data(2, spiSendData);
1660
1661        // Frequency Control Word, High Byte
1662        spiSendData[0] = CC1101_CR_FREQ2;
1663        spiSendData[1] = 0x10;
1664        SPI_send_data(2, spiSendData);
1665
1666        // Frequency Control Word, Middle Byte
1667        spiSendData[0] = CC1101_CR_FREQ1;
1668        spiSendData[1] = 0xB1;
1669        SPI_send_data(2, spiSendData);
1670
1671        //Frequency Control Word, LOW Byte
1672        spiSendData[0] = CC1101_CR_FREQ0;
1673        spiSendData[1] = 0x3B;
1674        SPI_send_data(2, spiSendData);
1675
1676        /***********************************************/
1677        // Modem Configuration 4
1678        spiSendData[0] = CC1101_CR_MDMCFG4;
1679        spiSendData[1] = 0x5D;
1680        SPI_send_data(2, spiSendData);
1681
1682        // Modem Configuration 3
1683        spiSendData[0] = CC1101_CR_MDMCFG3;
1684        spiSendData[1] = 0x3B;
1685        SPI_send_data(2, spiSendData);
1686
1687        // Modem Configuration 2
1688        // OOK -> 0x30
1689        spiSendData[0] = CC1101_CR_MDMCFG2;
1690        spiSendData[1] = 0x30;
1691        SPI_send_data(2, spiSendData);
1692
1693        // Modem Configuration 0
1694        // Channel spacing 199.951172kHz -> 0xF8
1695        spiSendData[0] = CC1101_CR_MDMCFG0;
1696        spiSendData[1] = 0xF8;
1697        SPI_send_data(2, spiSendData);
1698
1699        /***********************************************/
```

```
1700
1701        // Modem Deviation Setting
1702        spiSendData[0] = CC1101_CR_DEVIATN;
1703        spiSendData[1] = 0x15;
1704        SPI_send_data(2, spiSendData);
1705
1706        // Main Radio Control State Machine Configuration 1
1707        spiSendData[0] = CC1101_CR_MCSM1;
1708        spiSendData[1] = 0x07;
1709        SPI_send_data(2, spiSendData);
1710
1711        // Main Radio Control State Machine Configuration 0
1712        spiSendData[0] = CC1101_CR_MCSM0;
1713        spiSendData[1] = 0x18;
1714        SPI_send_data(2, spiSendData);
1715
1716        // Frequency Offset Compensation Configuration
1717        spiSendData[0] = CC1101_CR_FOCCFG;
1718        spiSendData[1] = 0x16;
1719        SPI_send_data(2, spiSendData);
1720
1721        // Bit Synchronization Configuration
1722        spiSendData[0] = CC1101_CR_BSCFG;
1723        spiSendData[1] = 0x6C;
1724        SPI_send_data(2, spiSendData);
1725
1726        // Wake On Radio Control
1727        spiSendData[0] = CC1101_CR_WORCTRL;
1728        spiSendData[1] = 0x8F;
1729        SPI_send_data(2, spiSendData);
1730
1731        // Front End RX Configuration 1
1732        spiSendData[0] = CC1101_CR_FREND1;
1733        spiSendData[1] = 0x56;
1734        SPI_send_data(2, spiSendData);
1735
1736        // Front End RX Configuration 0
1737        spiSendData[0] = CC1101_CR_FREND0;
1738        spiSendData[1] = 0x11;
1739        SPI_send_data(2, spiSendData);
1740
1741        // 10 dBm output power
1742        spiSendData[0] = 0x7E;
1743        spiSendData[1] = 0x00;
1744        spiSendData[2] = 0x60;
1745        SPI_send_data(3, spiSendData);
1746    }
1747
1748    void cc1101_config_cali_clk(void) {
1749        uint32_t spiSendData[3];
1750
1751        // GDO2 Output Pin Configuration 2
1752        spiSendData[0] = CC1101_CR_IOCFG2;
1753        spiSendData[1] = 0x0D;
1754        SPI_send_data(2, spiSendData);
1755
1756        // GDO0 Output Pin Configuration 0
1757        spiSendData[0] = CC1101_CR_IOCFG0;
1758        spiSendData[1] = 0x2E;
1759        SPI_send_data(2, spiSendData);
1760
1761        // RX FIFO and TX FIFO Thresholds
1762        // RX Attenuation 18dB -> 0x77
1763        // RX Attenuation 12dB -> 0x67
1764        // RX Attenuation  6dB -> 0x57
1765        // RX Attenuation  0dB -> 0x47
1766        spiSendData[0] = CC1101_CR_FIFOTHR;
1767        spiSendData[1] = 0x47;
1768        SPI_send_data(2, spiSendData);
1769
1770        // Asynchronous serial mode, Infinite packet length mode
1771        spiSendData[0] = CC1101_CR_PKTCTRL0;
1772        spiSendData[1] = 0x32;
1773        SPI_send_data(2, spiSendData);
1774
1775        // Frequency Synthesizer Control 1
1776        spiSendData[0] = CC1101_CR_FSCTRL1;
1777        spiSendData[1] = 0x06;
1778        SPI_send_data(2, spiSendData);
1779
1780        // Frequency Control Word, High Byte
1781        spiSendData[0] = CC1101_CR_FREQ2;
1782        spiSendData[1] = 0x10;
1783        SPI_send_data(2, spiSendData);
1784
```

```
1785        // Frequency Control Word, Middle Byte
1786        spiSendData[0] = CC1101_CR_FREQ1;
1787        spiSendData[1] = 0xB1;
1788        SPI_send_data(2, spiSendData);
1789
1790        //Frequency Control Word, LOW Byte
1791        spiSendData[0] = CC1101_CR_FREQ0;
1792        spiSendData[1] = 0x3B;
1793        SPI_send_data(2, spiSendData);
1794
1795        /************************************************/
1796        // Modem Configuration 4
1797        spiSendData[0] = CC1101_CR_MDMCFG4;
1798        spiSendData[1] = 0x5D;
1799        SPI_send_data(2, spiSendData);
1800
1801        // Modem Configuration 3
1802        spiSendData[0] = CC1101_CR_MDMCFG3;
1803        spiSendData[1] = 0x3B;
1804        SPI_send_data(2, spiSendData);
1805
1806        // Modem Configuration 2
1807        // OOK --> 0x30
1808        spiSendData[0] = CC1101_CR_MDMCFG2;
1809        spiSendData[1] = 0x30;
1810        SPI_send_data(2, spiSendData);
1811
1812        /************************************************/
1813
1814        // Modem Deviation Setting
1815        spiSendData[0] = CC1101_CR_DEVIATN;
1816        spiSendData[1] = 0x15;
1817        SPI_send_data(2, spiSendData);
1818
1819        // Main Radio Control State Machine Configuration 1
1820        spiSendData[0] = CC1101_CR_MCSM1;
1821        spiSendData[1] = 0x0B;
1822        SPI_send_data(2, spiSendData);
1823
1824        // Main Radio Control State Machine Configuration 0
1825        spiSendData[0] = CC1101_CR_MCSM0;
1826        spiSendData[1] = 0x18;
1827        SPI_send_data(2, spiSendData);
1828
1829        // Frequency Offset Compensation Configuration
1830        spiSendData[0] = CC1101_CR_FOCCFG;
1831        spiSendData[1] = 0x16;
1832        SPI_send_data(2, spiSendData);
1833
1834        // Wake On Radio Control
1835        spiSendData[0] = CC1101_CR_WORCTRL;
1836        spiSendData[1] = 0x8F;
1837        SPI_send_data(2, spiSendData);
1838
1839        // Front End RX Configuration 0
1840        spiSendData[0] = CC1101_CR_FREND0;
1841        spiSendData[1] = 0x11;
1842        SPI_send_data(2, spiSendData);
1843
1844        // 10 dBm output power
1845        spiSendData[0] = 0x7E;
1846        spiSendData[1] = 0x12;
1847        spiSendData[2] = 0xc0;
1848        SPI_send_data(3, spiSendData);
1849    }
1850
1851    void wakeup(void) {
1852            cc1101_power_up_reset();
1853            cc1101_reset();
1854            cc1101_config_burst_tx();
1855            cc1101_tx_asynchronous_mode();
1856            delay_ms(1000);
1857    }
1858
1859    void cc1101_clear_rx_fifo(void) {
1860        uint32_t spiSendData[1];
1861
1862        // DATA packet length
1863        spiSendData[0] = CC1101_CS_SFRX;
1864        SPI_send_data(1, spiSendData);
1865    }
1866
1867    void cc1101_rx(uint8_t packages_to_rx) {
1868        uint32_t spiSendData[2];
1869
```

```
1870        // DATA packet length
1871        spiSendData[0] = CC1101_CR_PKTLEN;
1872        spiSendData[1] = packages_to_rx;
1873        SPI_send_data(2, spiSendData);
1874
1875        //Enable CRC check when receiving data
1876        cc1101_enable_crc();
1877
1878        // Clear RX FIFO
1879        cc1101_clear_rx_fifo();
1880
1881        // RX state
1882        spiSendData[0] = CC1101_CS_SRX;
1883        SPI_send_data(1, spiSendData);
1884
1885    }
```

```
1  /*****************************************************************************
2  **     File name       : convert.c
3  **     Hardware        : ARM Cortex M4 Basestation
4  **     Date            : 14/09/2014
5  **     Last Update     : 27/07/2015
6  **     Author          : Nico Sassano
7  **     Description     : Convert function
8  **     State           : Final state
9  *****************************************************************************/
10 #include <stdint.h>
11 #include <stdbool.h>
12 #include "inc/hw_ints.h"
13 #include "inc/hw_memmap.h"
14 #include "inc/hw_types.h"
15 #include "driverlib/debug.h"
16 #include "driverlib/fpu.h"
17 #include "driverlib/ssi.h"
18 #include "driverlib/gpio.h"
19 #include "driverlib/interrupt.h"
20 #include "driverlib/pin_map.h"
21 #include "driverlib/rom.h"
22 #include "driverlib/rom_map.h"
23 #include "driverlib/sysctl.h"
24 #include "driverlib/timer.h"
25 #include "driverlib/uart.h"
26 #include "utils/uartstdio.h"
27 #include "header/main.h"
28 #include "header/uart.h"
29 #include "header/spi.h"
30 #include "header/timer.h"
31 #include "header/interrupts.h"
32 #include "header/CC1101.h"
33 #include "header/gpio.h"
34 #include "header/convert.h"
35
36 uint8_t char_to_int(char character) {
37     uint8_t value = 0x00;
38
39     if(character == 0x30) {
40         value = 0;
41     } else if(character == 0x31) {
42         value = 1;
43     } else if(character == 0x32) {
44         value = 2;
45     } else if(character == 0x33) {
46         value = 3;
47     } else if(character == 0x34) {
48         value = 4;
49     } else if(character == 0x35) {
50         value = 5;
51     } else if(character == 0x36) {
52         value = 6;
53     } else if(character == 0x37) {
54         value = 7;
55     } else if(character == 0x38) {
56         value = 8;
57     } else if(character == 0x39) {
58         value = 9;
59     }
60
61     return value;
62 }
63
64 uint8_t char_to_int2(char character, char character2) {
65     uint8_t value = 0x00;
66
67     if(character == 0x30) {
68         value = 0;
69     } else if(character == 0x31) {
70         value = 10;
71     } else if(character == 0x32) {
72         value = 20;
73     } else if(character == 0x33) {
74         value = 30;
75     } else if(character == 0x34) {
76         value = 40;
77     } else if(character == 0x35) {
78         value = 50;
79     } else if(character == 0x36) {
80         value = 60;
81     } else if(character == 0x37) {
82         value = 70;
83     } else if(character == 0x38) {
84         value = 80;
85     } else if(character == 0x39) {
```

```
86            value = 90;
87        }
88
89        if(character2 == 0x30) {
90            value = value + 0;
91        } else if(character2 == 0x31) {
92            value = value + 1;
93        } else if(character2 == 0x32) {
94            value = value + 2;
95        } else if(character2 == 0x33) {
96            value = value + 3;
97        } else if(character2 == 0x34) {
98            value = value + 4;
99        } else if(character2 == 0x35) {
100           value = value + 5;
101       } else if(character2 == 0x36) {
102           value = value + 6;
103       } else if(character2 == 0x37) {
104           value = value + 7;
105       } else if(character2 == 0x38) {
106           value = value + 8;
107       } else if(character2 == 0x39) {
108           value = value + 9;
109       }
110
111       return value;
112   }
113
114   uint8_t char_to_int3(char character, char character2, char character3) {
115       uint8_t value = 0x00;
116
117       if(character == 0x30) {
118           value = 000;
119       } else if(character == 0x31) {
120           value = 100;
121       } else if(character == 0x32) {
122           value = 200;
123       }
124
125       if(character2 == 0x30) {
126           value = value + 00;
127       } else if(character2 == 0x31) {
128           value = value + 10;
129       } else if(character2 == 0x32) {
130           value = value + 20;
131       } else if(character2 == 0x33) {
132           value = value + 30;
133       } else if(character2 == 0x34) {
134           value = value + 40;
135       } else if(character2 == 0x35) {
136           value = value + 50;
137       } else if(character2 == 0x36) {
138           value = value + 60;
139       } else if(character2 == 0x37) {
140           value = value + 70;
141       } else if(character2 == 0x38) {
142           value = value + 80;
143       } else if(character2 == 0x39) {
144           value = value + 90;
145       }
146
147       if(character3 == 0x30) {
148           value = value + 0;
149       } else if(character3 == 0x31) {
150           value = value + 1;
151       } else if(character3 == 0x32) {
152           value = value + 2;
153       } else if(character3 == 0x33) {
154           value = value + 3;
155       } else if(character3 == 0x34) {
156           value = value + 4;
157       } else if(character3 == 0x35) {
158           value = value + 5;
159       } else if(character3 == 0x36) {
160           value = value + 6;
161       } else if(character3 == 0x37) {
162           value = value + 7;
163       } else if(character3 == 0x38) {
164           value = value + 8;
165       } else if(character3 == 0x39) {
166           value = value + 9;
167       }
168       return value;
169   }
170
```

```
171 uint8_t char_to_hex(char character1, char character2) {
172     uint8_t hex = 0x00;
173 
174     if(character1 == 0x30) {
175         hex = 0x00 << 4;
176     } else if(character1 == 0x31) {
177         hex = 0x01 << 4;
178     } else if(character1 == 0x32) {
179         hex = 0x02 << 4;
180     } else if(character1 == 0x33) {
181         hex = 0x03 << 4;
182     } else if(character1 == 0x34) {
183         hex = 0x04 << 4;
184     } else if(character1 == 0x35) {
185         hex = 0x05 << 4;
186     } else if(character1 == 0x36) {
187         hex = 0x06 << 4;
188     } else if(character1 == 0x37) {
189         hex = 0x07 << 4;
190     } else if(character1 == 0x38) {
191         hex = 0x08 << 4;
192     } else if(character1 == 0x39) {
193         hex = 0x09 << 4;
194     } else if(character1 == 0x61) {
195         hex = 0x0a << 4;
196     } else if(character1 == 0x62) {
197         hex = 0x0b << 4;
198     } else if(character1 == 0x63) {
199         hex = 0x0c << 4;
200     } else if(character1 == 0x64) {
201         hex = 0x0d << 4;
202     } else if(character1 == 0x65) {
203         hex = 0x0e << 4;
204     } else if(character1 == 0x66) {
205         hex = 0x0f << 4;
206     } else if(character1 == 0x41) {
207         hex = 0x0A << 4;
208     } else if(character1 == 0x42) {
209         hex = 0x0B << 4;
210     } else if(character1 == 0x43) {
211         hex = 0x0C << 4;
212     } else if(character1 == 0x44) {
213         hex = 0x0D << 4;
214     } else if(character1 == 0x45) {
215         hex = 0x0E << 4;
216     } else if(character1 == 0x46) {
217         hex = 0x0F << 4;
218     }
219 
220     if(character2 == 0x30) {
221         hex |=0x00 << 0;
222     } else if(character2 == 0x31) {
223         hex |=0x01 << 0;
224     } else if(character2 == 0x32) {
225         hex |=0x02 << 0;
226     } else if(character2 == 0x33) {
227         hex |=0x03 << 0;
228     } else if(character2 == 0x34) {
229         hex |=0x04 << 0;
230     } else if(character2 == 0x35) {
231         hex |=0x05 << 0;
232     } else if(character2 == 0x36) {
233         hex |=0x06 << 0;
234     } else if(character2 == 0x37) {
235         hex |=0x07 << 0;
236     } else if(character2 == 0x38) {
237         hex |=0x08 << 0;
238     } else if(character2 == 0x39) {
239         hex |=0x09 << 0;
240     } else if(character2 == 0x61) {
241         hex |=0x0a << 0;
242     } else if(character2 == 0x62) {
243         hex |=0x0b << 0;
244     } else if(character2 == 0x63) {
245         hex |=0x0c << 0;
246     } else if(character2 == 0x64) {
247         hex |=0x0d << 0;
248     } else if(character2 == 0x65) {
249         hex |=0x0e << 0;
250     } else if(character2 == 0x66) {
251         hex |=0x0f << 0;
252     } else if(character2 == 0x41) {
253         hex |=0x0A << 0;
254     } else if(character2 == 0x42) {
255         hex |=0x0B << 0;
```

```
256         } else if(character2 == 0x43) {
257             hex |=0x0C << 0;
258         } else if(character2 == 0x44) {
259             hex |=0x0D << 0;
260         } else if(character2 == 0x45) {
261             hex |=0x0E << 0;
262         } else if(character2 == 0x46) {
263             hex |=0x0F << 0;
264         }
265
266         return hex;
267     }
268
269     uint16_t char_to_hex3(char character1, char character2, char character3) {
270         uint16_t hex = 0x0000;
271
272         if(character1 == 0x30) {
273             hex = 0x00 << 8;
274         } else if(character1 == 0x31) {
275             hex = 0x01 << 8;
276         } else if(character1 == 0x32) {
277             hex = 0x02 << 8;
278         } else if(character1 == 0x33) {
279             hex = 0x03 << 8;
280         } else if(character1 == 0x34) {
281             hex = 0x04 << 8;
282         } else if(character1 == 0x35) {
283             hex = 0x05 << 8;
284         } else if(character1 == 0x36) {
285             hex = 0x06 << 8;
286         } else if(character1 == 0x37) {
287             hex = 0x07 << 8;
288         } else if(character1 == 0x38) {
289             hex = 0x08 << 8;
290         } else if(character1 == 0x39) {
291             hex = 0x09 << 8;
292         } else if(character1 == 0x61) {
293             hex = 0x0a << 8;
294         } else if(character1 == 0x62) {
295             hex = 0x0b << 8;
296         } else if(character1 == 0x63) {
297             hex = 0x0c << 8;
298         } else if(character1 == 0x64) {
299             hex = 0x0d << 8;
300         } else if(character1 == 0x65) {
301             hex = 0x0e << 8;
302         } else if(character1 == 0x66) {
303             hex = 0x0f << 8;
304         } else if(character1 == 0x41) {
305             hex = 0x0A << 8;
306         } else if(character1 == 0x42) {
307             hex = 0x0B << 8;
308         } else if(character1 == 0x43) {
309             hex = 0x0C << 8;
310         } else if(character1 == 0x44) {
311             hex = 0x0D << 8;
312         } else if(character1 == 0x45) {
313             hex = 0x0E << 8;
314         } else if(character1 == 0x46) {
315             hex = 0x0F << 8;
316         }
317
318         if(character2 == 0x30) {
319             hex |=0x00 << 4;
320         } else if(character2 == 0x31) {
321             hex |=0x01 << 4;
322         } else if(character2 == 0x32) {
323             hex |=0x02 << 4;
324         } else if(character2 == 0x33) {
325             hex |=0x03 << 4;
326         } else if(character2 == 0x34) {
327             hex |=0x04 << 4;
328         } else if(character2 == 0x35) {
329             hex |=0x05 << 4;
330         } else if(character2 == 0x36) {
331             hex |=0x06 << 4;
332         } else if(character2 == 0x37) {
333             hex |=0x07 << 4;
334         } else if(character2 == 0x38) {
335             hex |=0x08 << 4;
336         } else if(character2 == 0x39) {
337             hex |=0x09 << 4;
338         } else if(character2 == 0x61) {
339             hex |=0x0a << 4;
340         } else if(character2 == 0x62) {
```

```
341             hex |=0x0b << 4;
342         } else if(character2 == 0x63) {
343             hex |=0x0c << 4;
344         } else if(character2 == 0x64) {
345             hex |=0x0d << 4;
346         } else if(character2 == 0x65) {
347             hex |=0x0e << 4;
348         } else if(character2 == 0x66) {
349             hex |=0x0f << 4;
350         } else if(character2 == 0x41) {
351             hex |=0x0A << 4;
352         } else if(character2 == 0x42) {
353             hex |=0x0B << 4;
354         } else if(character2 == 0x43) {
355             hex |=0x0C << 4;
356         } else if(character2 == 0x44) {
357             hex |=0x0D << 4;
358         } else if(character2 == 0x45) {
359             hex |=0x0E << 4;
360         } else if(character2 == 0x46) {
361             hex |=0x0F << 4;
362         }
363 
364         if(character3 == 0x30) {
365             hex |=0x00 << 0;
366         } else if(character3 == 0x31) {
367             hex |=0x01 << 0;
368         } else if(character3 == 0x32) {
369             hex |=0x02 << 0;
370         } else if(character3 == 0x33) {
371             hex |=0x03 << 0;
372         } else if(character3 == 0x34) {
373             hex |=0x04 << 0;
374         } else if(character3 == 0x35) {
375             hex |=0x05 << 0;
376         } else if(character3 == 0x36) {
377             hex |=0x06 << 0;
378         } else if(character3 == 0x37) {
379             hex |=0x07 << 0;
380         } else if(character3 == 0x38) {
381             hex |=0x08 << 0;
382         } else if(character3 == 0x39) {
383             hex |=0x09 << 0;
384         } else if(character3 == 0x61) {
385             hex |=0x0a << 0;
386         } else if(character3 == 0x62) {
387             hex |=0x0b << 0;
388         } else if(character3 == 0x63) {
389             hex |=0x0c << 0;
390         } else if(character3 == 0x64) {
391             hex |=0x0d << 0;
392         } else if(character3 == 0x65) {
393             hex |=0x0e << 0;
394         } else if(character3 == 0x66) {
395             hex |=0x0f << 0;
396         } else if(character3 == 0x41) {
397             hex |=0x0A << 0;
398         } else if(character3 == 0x42) {
399             hex |=0x0B << 0;
400         } else if(character3 == 0x43) {
401             hex |=0x0C << 0;
402         } else if(character3 == 0x44) {
403             hex |=0x0D << 0;
404         } else if(character3 == 0x45) {
405             hex |=0x0E << 0;
406         } else if(character3 == 0x46) {
407             hex |=0x0F << 0;
408         }
409 
410         return hex;
411     }
```

```
/*******************************************************************************
**      File name       : gpio.c
**      Hardware        : ARM Cortex M4 Basestation
**      Date            : 08/08/2014
**      Last Update     : 27/07/2015
**      Author          : Nico Sassano
**      Description     : GPIO Functions
**      State           : Final state
*******************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/uart.h"
#include "header/spi.h"
#include "header/timer.h"
#include "header/interrupts.h"
#include "header/CC1101.h"
#include "header/gpio.h"


void GPIOinit(){
    // Enable the GPIO port that is used for the on-board LEDs.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);

    // Current Sensor
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_4);
    ROM_GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_4, 0xFF);

    // Testpin
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTQ_BASE, GPIO_PIN_4);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_2);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_4);

    // Enable the GPIO pins for the LEDs.
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_1);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_4);
}

void LED_test() {
    LED0_ON;
    delay_ms(100);
    LED0_OFF;
    LED1_ON;
    delay_ms(100);
    LED1_OFF;
    LED2_ON;
    delay_ms(100);
    LED2_OFF;
    LED3_ON;
    delay_ms(100);
    LED3_OFF;
    RX_LED_ON;
    delay_ms(100);
    RX_LED_OFF;
    TX_LED_ON;
    delay_ms(100);
    TX_LED_OFF;

    RX_LED_ON;
    delay_ms(100);
    RX_LED_OFF;
    LED3_ON;
    delay_ms(100);
    LED3_OFF;
```

```
86          LED2_ON;
87          delay_ms(100);
88          LED2_OFF;
89          LED1_ON;
90          delay_ms(100);
91          LED1_OFF;
92          LED0_ON;
93          delay_ms(100);
94          LED0_OFF;
95
96          delay_ms(100);
97      }
```

```c
/*********************************************************************************
**     File name      : interrupts.c
**     Hardware       : ARM Cortex M4 Basestation
**     Date           : 16/05/2014
**     Last Update    : 27/07/2015
**     Author         : Nico Sassano
**     Description    : Interrupt function
**     State          : Final state
*********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/ssi.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/timer.h"
#include "utils/uartstdio.h"
#include "header/main.h"
#include "header/interrupts.h"
#include "header/cc1101.h"
#include "header/gpio.h"
#include "header/timer.h"
#include "header/adc.h"
#include "header/LTC2376.h"
#include "header/AD7691.h"
#include "header/AD7176.h"

//*****************************************************************************
// System settings and clock rate
//*****************************************************************************
extern uint32_t g_ui32SysClock;
extern uint32_t delay_flag;
uint32_t g_ui32Flags;

//*****************************************************************************
// Communication states
//*****************************************************************************
extern uint8_t adress_this_station;                      // Adress of this Station

//*****************************************************************************
// Variables for the CC1101 communication
//*****************************************************************************
extern volatile uint8_t rx_data_length;                  // Number of expect packetes in the RX Mode
extern volatile uint8_t tx_data_length;                  // Number of packetes to send
extern volatile uint8_t rx_command;                      // Received command
extern volatile uint8_t rx_data[64];                     // RX Data buffer
extern volatile uint8_t command_recived_flag;            // Flag for a command received

//*****************************************************************************
// Communication UART
//*****************************************************************************
extern volatile uint8_t rx_uart;                         // RX command
extern volatile uint8_t uart_rx_flag;                    // Flag for the UART
volatile unsigned char rx_char[31];                      // UART data buffer

//*****************************************************************************
// Interrupt and communiction flags
//*****************************************************************************
extern volatile uint8_t irq_mode;                        // Flag for the GDO2 mode
extern volatile uint8_t irq_send_done_flag;              // Flag for send done
extern volatile uint8_t irq_cali_flag;                   // Flag for burat calibration
extern volatile uint8_t irq_timer0_mode;                 // Flag for the timer mode
extern volatile uint8_t irq_timer1_mode;                 // Flag for the timer1 mode
extern volatile uint8_t irq_delay_flag;                  // Flag for the delay timer
extern volatile uint8_t irq_timeout;                     // Flag for the time out timer

//*****************************************************************************
// Burst Mode
//*****************************************************************************
extern volatile uint8_t burst_freq;                      // Burst Frequency
extern volatile uint16_t asynch_counter;                 // Counts the TX trigger
extern volatile uint32_t burst_config;                   // Frequency time for the burst timer
extern volatile uint32_t burst_trigger_time;             // Time (in us) for the burst trigger
extern volatile uint32_t burst_current[2000];            // Save the current date

//*****************************************************************************
```

```
86  // Clock calibration mode
87  //****************************************************************************
88  extern volatile uint32_t clk_cali_counter;              // Counter for the clock calibration
89  extern volatile uint32_t adc_value[50];
90  uint32_t value = 0;                                      // Temp. value storage
91
92  //****************************************************************************
93  // LTC2376-20 Test mode
94  // Data declaration for thesting LTC2376
95  // This can be deleted after testing the ADC
96  //****************************************************************************
97  extern volatile uint8_t LTC2376_index;
98  extern volatile uint8_t LTC2376_count;
99  extern volatile uint32_t LTC2376_data[3];                // 24-bit data storage
100 extern volatile uint32_t LTC2376_asynch_counter;
101 extern volatile uint32_t LTC2376_values;
102 extern volatile uint32_t LTC2376_frequency_time;
103 extern volatile uint32_t LTC2376_value;
104 extern volatile uint32_t LTC2376_value_storage[40000];
105
106 //****************************************************************************
107 // AD7691 Test mode
108 // Data declaration for thesting AD7691
109 // This can be deleted after testing the ADC
110 //****************************************************************************
111 extern volatile uint8_t AD7691_index;
112 extern volatile uint8_t AD7691_count;
113 extern volatile uint32_t AD7691_data[2];                 // 20-bit data storage
114 extern volatile uint32_t AD7691_asynch_counter;
115 extern volatile uint32_t AD7691_values;
116 extern volatile uint32_t AD7691_frequency_time;
117 extern volatile uint32_t AD7691_value;
118 extern volatile uint32_t AD7691_value_storage[2000];
119
120 //****************************************************************************
121 // AD7176 Test mode
122 // Data declaration for thesting AD7176
123 // This can be deleted after testing the ADC
124 //****************************************************************************
125 extern volatile uint8_t AD7176_index;
126 extern volatile uint8_t AD7176_count;
127 extern volatile uint32_t AD7176_data[3];                 // 3*8bit data storage
128 extern volatile uint32_t AD7176_asynch_counter;
129 extern volatile uint32_t AD7176_values;
130 extern volatile uint32_t AD7176_frequency_time;
131 extern volatile uint32_t AD7176_value;
132 extern volatile uint32_t AD7176_value_storage[2000];
133
134 //****************************************************************************
135 // The PortK interrupt handler, used for the communication with the CC1101
136 //****************************************************************************
137 void PortKIntHandler(void) {
138     uint32_t status = 0xFFFF;
139     uint32_t ADC_buffer;
140
141     status = GPIOIntStatus(GPIO_PORTK_BASE, true);
142     GPIOIntClear(GPIO_PORTK_BASE, status);
143
144     switch(status) {
145         /*****************************************************/
146         case GPIO_INT_PIN_0:    break;   // Interrupt on PortK0
147         /*****************************************************/
148         case GPIO_INT_PIN_1:    break;   // Interrupt on PortK1
149         /*****************************************************/
150         case GPIO_INT_PIN_2:    break;   // Interrupt on PortK2
151         /*****************************************************/
152         case GPIO_INT_PIN_3:    break;   // Interrupt on PortK3
153         /*****************************************************/
154         case GPIO_INT_PIN_4:    break;   // Interrupt on PortK4
155         /*****************************************************/
156         case GPIO_INT_PIN_5:    break;   // Interrupt on PortK5
157         /*****************************************************/
158         // Interrupt on GDO2 from CC1101 board
159         case GPIO_INT_PIN_6:             // Interrupt on PortK6
160             // Clear the GDO2 interrupt
161             CC1101_GDO2_CLEAR_IRQ;
162             CC1101_GDO2_IRQ_DISABLE;
163
164             //*** BEGIN DCO Calibration ********************************************************
165             if(IRQ_IS_GDO2_CLOCK_CALI_START) {
166                 TIMER2_START;                       // Start timer 2
167                 IRQ_SET_GDO2_CLOCK_CALI_END;        // Switch to detect the end of the periode
168                 CC1101_GDO2_IRQ_ENABLE;
169
170             } else if(IRQ_IS_GDO2_CLOCK_CALI_END) {
```

```
171                     adc_value[clk_cali_counter] = (g_ui32SysClock − ROM_TimerValueGet(TIMER2_BASE, TIMER_A));
172                     if((adc_value[clk_cali_counter]>200000) && (adc_value[clk_cali_counter]<290000)){           // noisfilter
173                         clk_cali_counter++;
174                     }
175 
176                     ROM_TimerDisable(TIMER2_BASE, TIMER_A);
177                     TIMER2config();                    // Timer to messure the triggerspace
178                     IRQ_SET_GDO2_CLOCK_CALI_START;
179                     CC1101_GDO2_IRQ_ENABLE;
180                 //*** END DCO Calibration *******************************************************
181 
182                 }else if(IRQ_IS_CALI_BURST) {
183                     GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0x00);      // Testpin
184                     value = ROM_TimerValueGet(TIMER2_BASE, TIMER_A);
185                     adc_value[0] = (g_ui32SysClock − value);
186                     ROM_TimerDisable(TIMER2_BASE, TIMER_A);
187                     RX_CALI_UNSET;
188 
189                 }else if(IRQ_IS_TX) {
190                     TX_DONE_SET;
191                     timeout_stop(); //Stop the timeout mode
192 
193                 } else if(IRQ_IS_RX) {
194                     RX_LED_ON;   // Turn the RX LED on
195 
196                     timeout_stop(); //Stop the timeout mode
197 
198                     rx_command = COMMAND_DOWNLINK_UNKWOWN;
199 
200                     COMMAND_REVIVED_SET;
201 
202                     delay_ms(10);
203                     //if (CC1101_GDO0_IS_HIGH) {
204 
205                         uint8_t rxbuf[64];
206 
207                         cc1101_read_rx_fifo(rxbuf, HEADER_LENGTH + rx_data_length);
208                         //rx_data_length = 0; // Datenlänge wieder zurücksetzen
209 
210                         rx_command = rxbuf[1];   // Received Command
211 
212                         // Data packet for this station?
213                         if (rxbuf[0] == adress_this_station) {
214                             uint8_t index;
215                             for (index = 0; index < (HEADER_LENGTH + rx_data_length); index++) {
216                                 rx_data[index] = rxbuf[index];
217                             }
218                         }
219                     //}
220                     RX_LED_OFF; // Turn the RX LED off
221                 }
222             break;
223             /*****************************************************/
224             case GPIO_INT_PIN_7:            // Interrupt on PortK0
225 
226                 if(TIMER1_MODE_IS_LTC2376) {
227                     // Clear the LTC2376 interrupt
228                     LTC2376_CLEAR_IRQ;
229                     LTC2376_IRQ_DISABLE;      // Disable the BUSY interrupt
230 
231                     // start to receive the data from the ADC
232                     LTC2376_CS_ENABLE;
233 
234                     LTC2376_CNV_SET_LOW;
235 
236                     for (LTC2376_index = 0; LTC2376_index < LTC2376_count ; LTC2376_index++) {
237 
238                         SSIDataPutNonBlocking(SSI3_BASE, 0x00);
239                         SSIDataGetNonBlocking(SSI3_BASE,&ADC_buffer);
240 
241                         LTC2376_data[LTC2376_index] = ADC_buffer & 0xFF;
242 
243                         // Wait untill SSI3 is ready
244                         while(SSIBusy(SSI3_BASE));
245                     }
246 
247                     LTC2376_CS_DISABLE;
248 
249                     LTC2376_value = 0;
250 
251                     LTC2376_value = (LTC2376_data[0]) << 12;
252                     LTC2376_value |= (LTC2376_data[1]& 0xFF) << 4 ;
253                     LTC2376_value |= (LTC2376_data[2]& 0xFF) >> 4 ;
254 
255                     LTC2376_value_storage[LTC2376_asynch_counter] = LTC2376_value;
```

```
256                 LTC2376_asynch_counter++;    // Increment the sampling counter
257             }else if(TIMER1_MODE_IS_AD7691) {
258                 AD7691_CLEAR_IRQ;
259                 AD7691_IRQ_DISABLE;
260
261                 delay_us(10);
262
263                 for (AD7691_index = 0; AD7691_index < AD7691_count ; AD7691_index++) {
264
265                     SSIDataPutNonBlocking(SSI3_BASE, 0x000);
266                     SSIDataGetNonBlocking(SSI3_BASE,&ADC_buffer);
267
268                     AD7691_data[AD7691_index] = ADC_buffer & 0x0FFF;
269                 }
270
271                 AD7691_CS_DISABLE;
272
273                 AD7691_value = 0;
274
275                 AD7691_value = (AD7691_data[0] & 0x1FF) << 9;
276                 AD7691_value |= (AD7691_data[1] & 0x3FE) >> 1;
277
278                 AD7691_value_storage[AD7691_asynch_counter] = 0xFFFF;
279                 AD7691_value_storage[AD7691_asynch_counter] = AD7691_value;
280                 AD7691_asynch_counter++;    // Increment the sampling counter
281
282
283             }else if(TIMER1_MODE_IS_AD7176) {
284                 AD7176_CLEAR_IRQ;
285                 AD7176_IRQ_DISABLE;
286
287                 // Readout the data from the ADC
288                 for (AD7176_index = 0; AD7176_index < AD7176_count ; AD7176_index++) {
289
290                     SSIDataPutNonBlocking(SSI3_BASE, 0x00);
291                     SSIDataGetNonBlocking(SSI3_BASE,&ADC_buffer);
292
293                     AD7176_data[AD7176_index] = ADC_buffer & 0xFF;
294
295                     // Wait untill SSI3 is ready
296                     while(SSIBusy(SSI3_BASE));
297                 }
298
299                 AD7176_CS_DISABLE;
300
301                 AD7176_value  = (AD7176_data[0] & 0xFF) << 16;
302                 AD7176_value |= (AD7176_data[1] & 0xFF) << 8 ;
303                 AD7176_value |= (AD7176_data[2] & 0xFF) << 0 ;
304
305                 AD7176_value_storage[AD7176_asynch_counter] = AD7176_value;
306                 AD7176_asynch_counter++;    // Increment the sampling counter
307
308             }
309         break;
310         /****************************************************/
311         default: break;                  // Default case
312     }
313 }
314
315 //**************************************************************************
316 // The UART interrupt handler for incoming characters
317 //**************************************************************************
318 void UARTIntHandler(void) {
319     uint32_t ui32Status;
320     uint8_t index = 0;
321     uint8_t index2 = 0;
322
323     // Get the interrrupt status.
324     ui32Status = ROM_UARTIntStatus(UART0_BASE, true);
325
326     // Clear the asserted interrupts.
327     ROM_UARTIntClear(UART0_BASE, ui32Status);
328
329     // Read characters untill the terminator CR
330     do{
331         rx_char[index] = UARTgetc(); //UARTCharGet(UART0_BASE);
332         index++;
333     }while(rx_char[index-1] != 0x0D);
334
335     //*****************************************************************************************
336     // Detection the command "help"
337     //*****************************************************************************************
338     if ((rx_char[0] == 0x68) && (rx_char[1] == 0x65) && (rx_char[2] == 0x6c) && (rx_char[3] == 0x70)) {
339         rx_uart = HELP;
340         uart_rx_flag = 1;
```

```
341
342     //*********************************************************************************************
343     // Detection the command "get_voltage"
344     //*********************************************************************************************
345     }  else if ((rx_char[0] == 0x67) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
346                 (rx_char[3] == 0x5F) && (rx_char[4] == 0x76) && (rx_char[5] == 0x6F) &&
347                 (rx_char[6] == 0x6C) && (rx_char[7] == 0x74) && (rx_char[8] == 0x61) &&
348                 (rx_char[9] == 0x67) && (rx_char[10] == 0x65)) {
349
350         rx_uart = GET_VOLTAGE;
351         uart_rx_flag = 1;
352     //*********************************************************************************************
353     // Detection the command "get_goertzel"
354     //*********************************************************************************************
355     }  else if ((rx_char[0] == 0x67) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
356                 (rx_char[3] == 0x5F) && (rx_char[4] == 0x67) && (rx_char[5] == 0x6F) &&
357                 (rx_char[6] == 0x65) && (rx_char[7] == 0x72) && (rx_char[8] == 0x74) &&
358                 (rx_char[9] == 0x7A) && (rx_char[10] == 0x65)&& (rx_char[11] == 0x6C)) {
359
360         rx_uart = GET_GORTZEL;
361         uart_rx_flag = 1;
362
363     //*********************************************************************************************
364     // Detection the command "send_hex()"
365     //*********************************************************************************************
366     }  else if ((rx_char[0] == 0x73) && (rx_char[1] == 0x65) && (rx_char[2] == 0x6E) &&
367                 (rx_char[3] == 0x64) && (rx_char[4] == 0x5F) && (rx_char[5] == 0x68) &&
368                 (rx_char[6] == 0x65) && (rx_char[7] == 0x78)) {
369
370             rx_uart = SEND_HEX;
371             uart_rx_flag = 1;
372     //*********************************************************************************************
373     // Detection the command "set_proprocessing"
374     //*********************************************************************************************
375     }   else if ((rx_char[0] == 0x73) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
376                  (rx_char[3] == 0x5F) && (rx_char[4] == 0x70) && (rx_char[5] == 0x72) &&
377                  (rx_char[6] == 0x65) && (rx_char[7] == 0x70) && (rx_char[8] == 0x72) &&
378                  (rx_char[9] == 0x6F) && (rx_char[10] == 0x63) && (rx_char[11] == 0x65) &&
379                  (rx_char[12] == 0x73) && (rx_char[13] == 0x73) && (rx_char[14] == 0x69) &&
380                  (rx_char[15] == 0x6E) && (rx_char[16] == 0x67)) {
381
382         rx_uart = SET_PROPROCESSING;
383         uart_rx_flag = 1;
384     //*********************************************************************************************
385     // Detection the command "burst"
386     //*********************************************************************************************
387     }   else if ((rx_char[0] == 0x62) && (rx_char[1] == 0x75) && (rx_char[2] == 0x72) &&
388                  (rx_char[3] == 0x73) && (rx_char[4] == 0x74)) {
389
390         rx_uart = BURST;
391         uart_rx_flag = 1;
392     //*********************************************************************************************
393     // Detection the command "cali_burst"
394     //*********************************************************************************************
395     }   else if ((rx_char[0] == 0x63) && (rx_char[1] == 0x61) && (rx_char[2] == 0x6c) &&
396                  (rx_char[3] == 0x69) && (rx_char[4] == 0x5F) && (rx_char[5] == 0x62) &&
397                  (rx_char[6] == 0x75) && (rx_char[7] == 0x72) && (rx_char[8] == 0x73) &&
398                  (rx_char[9] == 0x74)) {
399
400             rx_uart = CALI_BURST;
401             uart_rx_flag = 1;
402
403     //*********************************************************************************************
404     // Detection the command "cali_clk"
405     //*********************************************************************************************
406     }   else if ((rx_char[0] == 0x63) && (rx_char[1] == 0x61) && (rx_char[2] == 0x6c) &&
407                  (rx_char[3] == 0x69) && (rx_char[4] == 0x5F) && (rx_char[5] == 0x63) &&
408                  (rx_char[6] == 0x6c) && (rx_char[7] == 0x6B)) {
409
410             rx_uart = CALI_CLK;
411             uart_rx_flag = 1;
412
413     //*********************************************************************************************
414     // Detection the command "get_burst"
415     //*********************************************************************************************
416     }   else if ((rx_char[0] == 0x67) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
417                  (rx_char[3] == 0x5F) && (rx_char[4] == 0x62) && (rx_char[5] == 0x75) &&
418                  (rx_char[6] == 0x72) && (rx_char[7] == 0x73) && (rx_char[8] == 0x74)) {
419
420         rx_uart = GET_BURST;
421         uart_rx_flag = 1;
422
423     //*********************************************************************************************
424     // Detection the command "get_config"
425     //*********************************************************************************************
```

```
426     }    else if ((rx_char[0] == 0x67) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
427              (rx_char[3] == 0x5F) && (rx_char[4] == 0x63) && (rx_char[5] == 0x6F) &&
428              (rx_char[6] == 0x6E) && (rx_char[7] == 0x66) && (rx_char[8] == 0x69) &&
429              (rx_char[9] == 0x67)) {
430
431         if((rx_char[11] == 0x30) && (rx_char[12] == 0x30)) {
432             rx_uart = GET_CONFIG_BS;
433         } else {
434             rx_uart = GET_CONFIG_ZS;
435         }
436
437         uart_rx_flag = 1;
438
439      //****************************************************************************************************
440      // Detection the command "set_brate"
441      //****************************************************************************************************
442      }  else if ((rx_char[0] == 0x73) && (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
443              (rx_char[3] == 0x5F) && (rx_char[4] == 0x62) && (rx_char[5] == 0x72) &&
444              (rx_char[6] == 0x61) && (rx_char[7] == 0x74) && (rx_char[8] == 0x65)) {
445
446          rx_uart = SET_BRATE;
447          uart_rx_flag = 1;
448     //****************************************************************************************************
449     // Detection the command "matlabmode"
450     //****************************************************************************************************
451     }    else if ((rx_char[0] == 0x6D) && (rx_char[1] == 0x61) && (rx_char[2] == 0x74) &&
452          (rx_char[3] == 0x6C) && (rx_char[4] == 0x61) && (rx_char[5] == 0x62) &&
453          (rx_char[6] == 0x6D) && (rx_char[7] == 0x6F) && (rx_char[8] == 0x64) &&
454          (rx_char[9] == 0x65)) {
455
456         rx_uart = MATLABMODE;
457         uart_rx_flag = 1;
458
459     //****************************************************************************************************
460     // Detection the command "LTC2376MODE"
461     //****************************************************************************************************
462     }    else if ((rx_char[0] == 0x6C) && (rx_char[1] == 0x74) && (rx_char[2] == 0x63) &&
463              (rx_char[3] == 0x32) && (rx_char[4] == 0x33) && (rx_char[5] == 0x37) &&
464              (rx_char[6] == 0x36) && (rx_char[7] == 0x6D) && (rx_char[8] == 0x6F) &&
465              (rx_char[9] == 0x64) && (rx_char[10] == 0x65)) {
466
467         rx_uart = LTC2376MODE;
468         uart_rx_flag = 1;
469
470     //****************************************************************************************************
471     // Detection the command "AD7691MODE"
472     //****************************************************************************************************
473     }    else if ((rx_char[0] == 0x61) && (rx_char[1] == 0x64) && (rx_char[2] == 0x37) &&
474              (rx_char[3] == 0x36) && (rx_char[4] == 0x39) && (rx_char[5] == 0x31) &&
475              (rx_char[6] == 0x6D) && (rx_char[7] == 0x6F) && (rx_char[8] == 0x64) &&
476              (rx_char[9] == 0x65)) {
477
478         rx_uart = AD7691MODE;
479         uart_rx_flag = 1;
480
481     //****************************************************************************************************
482     // Detection the command "SET_AD5270_BS"
483     //****************************************************************************************************
484     }    else if ((rx_char[0] == 0x73) &&  (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
485              (rx_char[3] == 0x5F) &&  (rx_char[4] == 0x61) && (rx_char[5] == 0x64) &&
486              (rx_char[6] == 0x35) &&  (rx_char[7] == 0x32) && (rx_char[8] == 0x37) &&
487              (rx_char[9] == 0x30) && (rx_char[10] == 0x5F) && (rx_char[11] == 0x62) &&
488              (rx_char[12] == 0x73)) {
489
490             rx_uart = SET_AD5270_BS;
491             uart_rx_flag = 1;
492
493     //****************************************************************************************************
494     // Detection the command "SET_AD5270_ZS"
495     //****************************************************************************************************
496     }    else if ((rx_char[0] == 0x73) &&  (rx_char[1] == 0x65) && (rx_char[2] == 0x74) &&
497          (rx_char[3] == 0x5F) &&  (rx_char[4] == 0x61) && (rx_char[5] == 0x64) &&
498          (rx_char[6] == 0x35) &&  (rx_char[7] == 0x32) && (rx_char[8] == 0x37) &&
499          (rx_char[9] == 0x30) && (rx_char[10] == 0x5F) && (rx_char[11] == 0x7A) &&
500          (rx_char[12] == 0x73)) {
501
502         rx_uart = SET_AD5270_ZS;
503         uart_rx_flag = 1;
504
505     //****************************************************************************************************
506     // Detection the command "why"
507     //****************************************************************************************************
508     }else if ((rx_char[0] == 0x77) && (rx_char[1] == 0x68) && (rx_char[2] == 0x79)) {
509         // why
510         rx_uart = WHY;
```

```
511             uart_rx_flag = 1;
512
513         } else {
514             UARTprintf("\n****************************************\n");
515             UARTprintf("Unknown command!\nYour input was: ");
516             for(index2 = 0; index2 <= index-1;index2++)
517                 UARTprintf("%c",rx_char[index2]);
518
519             UARTprintf("\n****************************************\n");
520         }
521     }
522
523     //******************************************************************************
524     // The interrupt handler for the delay and the timeout function.
525     //******************************************************************************
526     void Timer0IntHandler(void) {
527
528         // Clear the timer interrupt.
529         ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
530         ROM_TimerDisable(TIMER0_BASE, TIMER_A);
531
532         if(TIMER_MODE_IS_TIMEOUT) { // Check if ithis the delay timer or the timeout timer
533             timeout_stop(); //Stop the timeout mode
534             TIMEOUT_SET;
535         } else if(TIMER_MODE_IS_DELAY){
536             timeout_stop(); //Stop the delay mode
537             irq_delay_flag = 1;
538         }
539     }
540
541     //******************************************************************************
542     // The interrupt handler for for the burst and for the LTC2376 mode
543     //******************************************************************************
544     void Timer1IntHandler(void) {
545        uint32_t pui32ADC0Value[1];
546
547         //*** Variables for the AD7176 test mode ***
548         uint8_t length = 3;
549         uint8_t data[3];
550         uint16_t index = 0;
551         //******************************************
552
553         // Clear the timer interrupt.
554         ROM_TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
555
556         if(TIMER1_MODE_IS_BURST) {  // Check if burst falg is set
557             if(CC1101_GDO0_IS_HIGH) {
558                 CC1101_GDO0_SET_LOW;
559
560                 ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, burst_trigger_time); //Config the clocktime -> g_ui32SysClock / config
561
562             } else {
563                 GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0x00);     // Testpin
564                 CC1101_GDO0_SET_HIGH; // send the burst trigger
565
566                 ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, burst_config - burst_trigger_time);
567
568                 //SysCtlDelay(3302);          // 82,55us
569                 //SysCtlDelay(3140);          // 78,5us
570                 //SysCtlDelay(3180);          // 79,5us
571                 SysCtlDelay(3165);            // 79,125us
572                 //SysCtlDelay(3100);          // 77,5us
573                 //SysCtlDelay(3060);          // 76,5us
574                 //SysCtlDelay(3020);          // 75,5us
575                 //SysCtlDelay(2940);          // 73,5us
576                 //SysCtlDelay(2900);              // 72,5us
577                 //SysCtlDelay(2780);          // 69,5us
578
579                 GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0xFF);     // Testpin
580
581                 //****************************************************************************
582                 // !!!! ADC Wandlung ink. abspeicherung dauert 3,2us !!!!!!!!
583                 // ADC hat eine Referenzspannung von 3.3V
584                 // Start the current messurment
585                 ADCProcessorTrigger(ADC0_BASE, 3);             // Trigger the ADC conversion.
586                 while(!ADCIntStatus(ADC0_BASE, 3, false));  // Wait for conversion to be completed
587                 ADCIntClear(ADC0_BASE, 3);                   // Clear the ADC interrupt flag.
588                 ADCSequenceDataGet(ADC0_BASE, 3, pui32ADC0Value);    // Read ADC Value.
589                 GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0xFF);     // Testpin
590                 burst_current[asynch_counter] = pui32ADC0Value[0];
591                 //****************************************************************************
592
593                 asynch_counter++;
594             }
595         }else if(TIMER1_MODE_IS_LTC2376) {  // Check if burst falg is set
```

```
596            LTC2376_CNV_SET_HIGH;              // start a new convertion
597            LTC2376_IRQ_ENABLE;                // Enable the BUSY interrupt
598
599        }else if(TIMER1_MODE_IS_AD7691) {    // Check if burst falg is set
600            AD7691_CNV_SET_HIGH;           // start a new convertion
601            delay_us(5);
602            AD7691_CNV_SET_LOW;
603            AD7691_IRQ_ENABLE;             // Enable the BUSY interrupt
604
605 //         //*************************************************************************
606 //         // !!!! ADC Wandlung ink. abspeicherung dauert 3,2us !!!!!!!!
607 //         // Start the current messurment
608 //         ADCProcessorTrigger(ADC0_BASE, 3);            // Trigger the ADC conversion.
609 //         while(!ADCIntStatus(ADC0_BASE, 3, false));   // Wait for conversion to be completed
610 //         ADCIntClear(ADC0_BASE, 3);                    // Clear the ADC interrupt flag.
611 //         ADCSequenceDataGet(ADC0_BASE, 3, pui32ADC0Value);    // Read ADC Value.
612 //         burst_current[AD7691_asynch_counter] = pui32ADC0Value[0];
613 //         //*************************************************************************
614
615        }else if(TIMER1_MODE_IS_AD7176) {    // Check if burst falg is set
616            AD7176_CS_ENABLE;
617
618            // Config the ADC for single conversion mode -> 0x8010
619            length = 3;
620            data[0] = ADCMODE;
621            data[1] = 0x80;
622            data[2] = 0x10;
623
624            for(index = 0; index < length; index++) {
625                SSIDataPut(SSI3_BASE, data[index]);
626
627                // Wait until SSI3 is done transferring all the data in the transmit FIFO.
628                while(SSIBusy(SSI3_BASE));
629            }
630
631            AD7176_IRQ_ENABLE;
632        }
633    }
634
635    void Timer2IntHandler(void) {
636
637    }
```

```c
/********************************************************************************
**     File name       : LTC2376.c
**     Hardware        : ARM Cortex M4 Basestation
**     Date            : 22/12/2014
**     Last Update     : 27/07/2015
**     Author          : Nico Sassano
**     Description     : LTC2376 function
**     State           : Final state
********************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/main.h"
#include "header/uart.h"
#include "header/spi.h"
#include "header/timer.h"
#include "header/interrupts.h"
#include "header/CC1101.h"
#include "header/gpio.h"
#include "header/convert.h"
#include "header/uart_check.h"
#include "header/adc.h"
#include "header/LTC2376.h"

extern uint32_t matlab_mode;

//**************************************************************************
// LTC2376-20 Test mode
// Data declaration for thesting LTC2376
// This can be deleted after testing the ADC
//**************************************************************************
extern volatile uint8_t LTC2376_index;
extern volatile uint8_t LTC2376_count;
extern volatile uint32_t LTC2376_data[3]; // 24-bit data storage
extern volatile uint32_t LTC2376_asynch_counter;
extern volatile uint32_t LTC2376_values;
extern volatile uint32_t LTC2376_frequency_time;
extern volatile uint32_t LTC2376_value;
extern volatile uint32_t LTC2376_value_storage[40000];

//*****************************************************
// This function is config the LTC2376 ADC
//*****************************************************
void init_ltc2376(void) {

    // BUSY Input Pin
    LTC2376_BUSY_SET_INPUT;

    // CNV Output Pin
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_6);
    LTC2376_CNV_SET_LOW;

    // CS LTC2376 Output Pin
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1);
    LTC2376_CS_DISABLE;

    // Set BUSY as interrupt
    LTC2376_IRQ_FALLING_EDGE;
    LTC2376_IRQ_DISABLE;

    LTC2376_CS_DISABLE;

}

//*****************************************************
// This function starts the sampling mode
//*****************************************************
void ltc2376_start(uint8_t ltc2376_frequency, uint8_t burst_values) {
    uint32_t index = 0;
```

```
86
87      uint32_t value_high = 0;
88      uint32_t value_low = 0;
89
90      switch(burst_values) {
91          case BURST_VALUES_50:
92              LTC2376_values = 50; break;
93
94          case BURST_VALUES_100:
95              LTC2376_values = 100; break;
96
97          case BURST_VALUES_150:
98              LTC2376_values = 150; break;
99
100         case BURST_VALUES_200:
101             LTC2376_values = 200; break;
102
103         case BURST_VALUES_250:
104             LTC2376_values = 250; break;
105
106         case BURST_VALUES_300:
107             LTC2376_values = 300; break;
108
109         case BURST_VALUES_350:
110             LTC2376_values = 350; break;
111
112         case BURST_VALUES_400:
113             LTC2376_values = 400; break;
114
115         case BURST_VALUES_450:
116             LTC2376_values = 450; break;
117
118         case BURST_VALUES_500:
119             LTC2376_values = 500; break;
120
121         case BURST_VALUES_550:
122             LTC2376_values = 550; break;
123
124         case BURST_VALUES_600:
125             LTC2376_values = 600; break;
126
127         case BURST_VALUES_650:
128             LTC2376_values = 650; break;
129
130         case BURST_VALUES_700:
131             LTC2376_values = 700; break;
132
133         case BURST_VALUES_750:
134             LTC2376_values = 750; break;
135
136         case BURST_VALUES_800:
137             LTC2376_values = 800; break;
138
139         case BURST_VALUES_850:
140             LTC2376_values = 850; break;
141
142         case BURST_VALUES_900:
143             LTC2376_values = 900; break;
144
145         case BURST_VALUES_1000:
146             LTC2376_values = 1000; break;
147
148         case BURST_VALUES_1500:
149             LTC2376_values = 1500; break;
150
151         case BURST_VALUES_1900:
152             LTC2376_values = 30000; break;
153     }
154
155     // To calculate the frequency_time
156     // (1/(frequency [Hz]*10^-6))
157     switch(ltc2376_frequency) {
158         case BURST_FREQ_50HZ:
159             LTC2376_frequency_time = 20000; break;
160
161         case BURST_FREQ_100HZ:
162             LTC2376_frequency_time = 10000; break;
163
164         case BURST_FREQ_150HZ:
165             LTC2376_frequency_time = 6666; break;
166
167         case BURST_FREQ_200HZ:
168             LTC2376_frequency_time = 5000; break;
169
170         case BURST_FREQ_250HZ:
```

```
171                 LTC2376_frequency_time = 4000; break;
172
173             case BURST_FREQ_300HZ:
174                 LTC2376_frequency_time = 3333; break;
175
176             case BURST_FREQ_350HZ:
177                 LTC2376_frequency_time = 2857; break;
178
179             case BURST_FREQ_400HZ:
180                 LTC2376_frequency_time = 2500; break;
181
182             case BURST_FREQ_450HZ:
183                 LTC2376_frequency_time = 2222; break;
184
185             case BURST_FREQ_500HZ:
186                 LTC2376_frequency_time = 2000; break;
187
188             case BURST_FREQ_550HZ:
189                 LTC2376_frequency_time = 1818; break;
190
191             case BURST_FREQ_600HZ:
192                 LTC2376_frequency_time = 1666; break;
193
194             case BURST_FREQ_650HZ:
195                 LTC2376_frequency_time = 1538; break;
196
197             case BURST_FREQ_700HZ:
198                 LTC2376_frequency_time = 1428; break;
199
200             case BURST_FREQ_750HZ:
201                 LTC2376_frequency_time = 1333; break;
202
203             case BURST_FREQ_800HZ:
204                 LTC2376_frequency_time = 1250; break;
205
206             case BURST_FREQ_850HZ:
207                 LTC2376_frequency_time = 1176; break;
208
209             case BURST_FREQ_900HZ:
210                 LTC2376_frequency_time = 1111; break;
211
212             case BURST_FREQ_950HZ:
213                 LTC2376_frequency_time = 1052; break;
214
215             case BURST_FREQ_1000HZ:
216                 LTC2376_frequency_time = 1000; break;
217
218             case BURST_FREQ_2000HZ:
219                 LTC2376_frequency_time = 500; break;
220
221             case BURST_FREQ_4000HZ:
222                 LTC2376_frequency_time = 250; break;
223
224             case BURST_FREQ_6000HZ:
225                 LTC2376_frequency_time = 167; break;
226
227             case BURST_FREQ_8000HZ:
228                 LTC2376_frequency_time = 100; //125; break;
229
230             case BURST_FREQ_10000HZ:
231                 LTC2376_frequency_time = 100; break;
232
233         }
234
235         // Start the LTC2376 messurment
236         LTC2376_IRQ_DISABLE;
237         LTC2376_CLEAR_IRQ;
238
239         LTC2376_BUSY_SET_INPUT;
240         LTC2376_IRQ_FALLING_EDGE;
241         LTC2376_IRQ_DISABLE;
242
243         LTC2376_asynch_counter = 0;
244
245         TIMER1config_LTC2376(LTC2376_frequency_time);
246
247         while(LTC2376_asynch_counter != LTC2376_values);
248         TIMER1_STOP;
249
250         value_low = LTC2376_value_storage[0];
251         value_high = LTC2376_value_storage[0];
252
253         for(index = 0 ; index < LTC2376_asynch_counter ; index++) {
254             if(!matlab_mode) {   // If Matlab mode is off
255                 UARTprintf("%i\n",LTC2376_value_storage[index]);
```

```
256             }else {                  // If Matlab mode is on
257                 if(index == LTC2376_asynch_counter-1) {
258                     UARTprintf("%i",LTC2376_value_storage[index]);
259                 }else {
260                     UARTprintf("%i  ",LTC2376_value_storage[index]);
261                 }
262             }
263 
264             if(LTC2376_value_storage[index] <= value_low)
265                 value_low = LTC2376_value_storage[index];
266 
267             if(LTC2376_value_storage[index] >= value_high)
268                 value_high = LTC2376_value_storage[index];
269         }
270 }
```

```
1  /******************************************************************************
2  **     File name       :  main.c
3  **     Hardware        :  ARM Cortex M4 Basestation
4  **     Date            :  16/05/2014
5  **     Last Update     :  17/07/2015
6  **     Author          :  Nico Sassano
7  **     Description     :  Main routine
8  **     State           :  Final state
9  ******************************************************************************/
10
11 #include <stdint.h>
12 #include <stdbool.h>
13 #include <math.h>
14 #include "inc/hw_ints.h"
15 #include "inc/hw_memmap.h"
16 #include "inc/hw_types.h"
17 #include "driverlib/debug.h"
18 #include "driverlib/fpu.h"
19 #include "driverlib/ssi.h"
20 #include "driverlib/gpio.h"
21 #include "driverlib/interrupt.h"
22 #include "driverlib/pin_map.h"
23 #include "driverlib/rom.h"
24 #include "driverlib/rom_map.h"
25 #include "driverlib/sysctl.h"
26 #include "driverlib/timer.h"
27 #include "driverlib/uart.h"
28 #include "utils/uartstdio.h"
29 #include "header/main.h"
30 #include "header/uart.h"
31 #include "header/spi.h"
32 #include "header/timer.h"
33 #include "header/interrupts.h"
34 #include "header/CC1101.h"
35 #include "header/gpio.h"
36 #include "header/convert.h"
37 #include "header/uart_check.h"
38 #include "header/adc.h"
39 #include "header/LTC2376.h"
40 #include "header/AD7691.h"
41 #include "header/AD7176.h"
42 #include "header/AD5270.h"
43
44 //****************************************************************************
45 // System settings and clock rate
46 //****************************************************************************
47 uint32_t g_sw_version                  = 150717;
48 uint32_t g_ui32SysClock;
49 uint32_t g_ui32Flags;
50 uint32_t spiBitRate                    = 100000;
51 uint32_t spiBit                        = 8;
52 uint32_t spiConfig                     = SSI_FRF_MOTO_MODE_0;
53 uint32_t rx_timeout                    = 2000;
54 uint32_t tx_timeout                    = 2000;
55 uint32_t ui32Baud                      = 56000;
56 uint32_t ui32PortNum                   = 0;
57 uint32_t ui32SrcClock                  = 0;
58 uint32_t matlab_mode                   = 1;
59
60 //****************************************************************************
61 // Communication states
62 //****************************************************************************
63 uint8_t brate                          = 100;
64 uint8_t packet[10]                     = {0x00};
65 uint8_t adress_this_station            = 0xFF;
66
67 //****************************************************************************
68 // Variables for the CC1101 communication
69 //****************************************************************************
70 volatile uint8_t rx_data_length        = 0;            // Number of expect packetes in the RX Mode
71 volatile uint8_t tx_data_length        = 0;            // Number of packetes to send
72 volatile uint8_t frame_counter         = 0;            // Counter for the frames
73 volatile uint8_t rx_command            = COMMAND_WAIT; // Received command
74 volatile uint8_t rx_data[64]           = {0x00};       // RX Data buffer
75 volatile uint8_t command_recived_flag  = 0;            // Flag for a command received
76
77 //****************************************************************************
78 // Communication UART
79 //****************************************************************************
80 volatile uint8_t rx_uart               = 0;            // RX command
81 volatile uint8_t uart_rx_flag          = 0;            // Flag for the UART
82 volatile unsigned char rx_char[31];                    // UART data buffer
83
84 //****************************************************************************
85 // Interrupt and communiction flags
```

```
86  //*****************************************************************************
87  volatile uint8_t irq_mode               = TX_MODE;       // Flag for the GDO2 mode
88  volatile uint8_t irq_send_done_flag     = 0;             // Flag for send done
89  volatile uint8_t irq_cali_flag          = 0;             // Flag for burat calibration
90
91  volatile uint8_t irq_timer0_mode        = DELAY_MODE;    // Flag for the timer mode
92  volatile uint8_t irq_timer1_mode        = BURST_MODE;    // Flag for the timer1 mode
93  volatile uint8_t irq_delay_flag         = 0;             // Flag for the delay timer
94  volatile uint8_t irq_timeout            = 0;             // Flag for the time out timer
95
96  //*****************************************************************************
97  // Burst Mode
98  //*****************************************************************************
99  volatile uint8_t burst_freq             = BURST_FREQ_1000HZ; // Burst Frequenz
100 uint8_t burst_values                    = 0;             // Anzahl der Burst Werte
101 uint16_t seq_number                     = 0;             // Zählt die Seq.nummer
102 volatile uint16_t asynch_counter        = 0;             // Counts the TX trigger
103 volatile uint32_t burst_config          = 0;             // Frequency time for the burst timer
104
105 volatile uint32_t burst_data[2000];                      // Save the burst data
106 volatile uint32_t burst_current[2000];                   // Save the current date
107 volatile uint32_t burst_trigger_time    = 500;           // Time (in us) for the burst trigger
108
109
110 //*****************************************************************************
111 // Clock calibration mode
112 //*****************************************************************************
113 volatile uint32_t clk_cali_counter      = 0;             // Counter for the clock calibration
114 volatile uint32_t adc_value[40];
115 volatile uint32_t clk_numb_counts       = 40;            // number of measurments for the calibration
116 volatile uint32_t clk_tolerance         = 200;           // tolerance of the clk calibration
117
118 //*****************************************************************************
119 // LTC2376-20 Test mode
120 // Data declaration for thesting LTC2376
121 // This can be deleted after testing the ADC
122 //*****************************************************************************
123 volatile uint8_t LTC2376_index = 0;
124 volatile uint8_t LTC2376_count = 3;
125 volatile uint32_t LTC2376_data[3];                       // 20-bit data storage
126 volatile uint32_t LTC2376_asynch_counter = 0;
127 volatile uint32_t LTC2376_values = 0;
128 volatile uint32_t LTC2376_frequency_time = 0;
129 volatile uint32_t LTC2376_value;
130 volatile uint32_t LTC2376_value_storage[40000];
131
132 //*****************************************************************************
133 // AD7691 Test mode
134 // Data declaration for thesting AD7691
135 // This can be deleted after testing the ADC
136 //*****************************************************************************
137 volatile uint8_t AD7691_index = 0;
138 volatile uint8_t AD7691_count = 2;
139 volatile uint32_t AD7691_data[2];                        // 20-bit data storage
140 volatile uint32_t AD7691_asynch_counter = 0;
141 volatile uint32_t AD7691_values = 0;
142 volatile uint32_t AD7691_frequency_time = 0;
143 volatile uint32_t AD7691_value;
144 volatile uint32_t AD7691_value_storage[2000];
145
146 //*****************************************************************************
147 // AD7176 Test mode
148 // Data declaration for thesting AD7176
149 // This can be deleted after testing the ADC
150 //*****************************************************************************
151 volatile uint8_t AD7176_index = 0;
152 volatile uint8_t AD7176_count = 3;                       // 3*8Bit readout
153 volatile uint32_t AD7176_data[3];                        // 3*8bit data storage
154 volatile uint32_t AD7176_asynch_counter = 0;
155 volatile uint32_t AD7176_values = 0;
156 volatile uint32_t AD7176_frequency_time = 0;
157 volatile uint32_t AD7176_value;
158 volatile uint32_t AD7176_value_storage[2000];
159
160 //*****************************************************************************
161 // Goertzel calcuation
162 //*****************************************************************************
163 float Fs_goertzel               = 0.0;  // Burstfrequncy
164 float f_goertzel                = 0.0;  // Signal frequency
165 float m                         = 0.0;
166 float bn_0 = 0;
167 float bn_1 = 0;
168 float bn_2 = 0;
169 float N_float;
170 float omega;
```

```
171 float sine;
172 float cosine;
173 float coeff;
174
175 float Np;
176 float f_voltage = 0.0;
177
178 typedef union {
179     float float_data;
180     uint8_t hex_data[4];
181 } FB;
182
183 FB real_char;
184 FB imag_char;
185
186 //*****************************************************************************
187 // The error routine that is called if the driver library encounters an error.
188 //*****************************************************************************
189 #ifdef DEBUG
190 void
191 __error__(char *pcFilename, uint32_t ui32Line)
192 {
193 }
194 #endif
195
196
197 int main(void) {
198     ROM_IntDisable(INT_WATCHDOG);   // Disable the watchdog-timer
199
200     // Set the clocking to run directly from the crystal at 120MHz.
201     g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
202             SYSCTL_OSC_MAIN |
203             SYSCTL_USE_PLL |
204             SYSCTL_CFG_VCO_480), 120000000);
205
206     ui32SrcClock = g_ui32SysClock;
207
208     // Initialize the UART and write status.
209     UARTinit(ui32PortNum,ui32Baud,ui32SrcClock);
210
211     // Initialize the SPI
212     SPIinit(spiBitRate, spiBit, spiConfig);
213
214     // Initialize the timer.
215     TIMER0init();   // For the timer
216     TIMER1init();   // For the burt measurement
217     TIMER2init();
218
219     GPIOinit();
220
221     init_ltc2376();
222     //init_ad7691();
223
224     init_adc();
225
226     cc1101_power_up_reset();
227
228     LED_test();
229
230     // Interrupt Priority
231     IntPrioritySet(INT_SYSCTL, 0x00); // Systick 1st Priority
232     IntPrioritySet(INT_ADC0SS0, 0x10);
233     IntPrioritySet(INT_TIMER0A, 0x20);
234     IntPrioritySet(INT_TIMER1A, 0x40);
235     IntPrioritySet(INT_TIMER2A, 0x60);
236     IntPrioritySet(INT_GPIOK, 0x80);
237     IntPrioritySet(INT_UART0, 0xA0);
238
239     // Enable processor interrupts.
240     ROM_IntMasterEnable();
241
242     while(1) {
243
244         while(!uart_rx_flag);
245
246         switch(rx_uart) {
247             //*******************************************************
248             // "help" Open the help menue
249             //*******************************************************
250             case HELP: {
251                 UARTprintf("\n
                        ***************************************************************************************************\
                        n");
252
253                 UARTprintf("These are the possible commands:\n");
```

```
254                     UARTprintf("--------------------------------------------\n");
255                     UARTprintf("\n");
256                     UARTprintf("get_voltage(NUMBER OF SENSOR)              -> Receive the voltage of a sensor\n");
257                     UARTprintf("for get the voltage of sensor 0x02, please type -> get_voltage(02)\n");
258                     UARTprintf("\n");
259                     UARTprintf("\n");
260                     UARTprintf("burst(BURST FREQUENCY, NUMBER OF VALUES) -> Starts the burst mode\n");
261                     UARTprintf("Possible frequencys and number of values are the following:\n");
262                     UARTprintf("burst frequency   number of values\n");
263                     UARTprintf("_________________________________\n");
264                     UARTprintf("4.000 Hz               1.900\n");
265                     UARTprintf("2.000 Hz               1.500\n");
266                     UARTprintf("1.000 Hz               1.000\n");
267                     UARTprintf("  950 Hz                 900\n");
268                     UARTprintf("  900 Hz                 850\n");
269                     UARTprintf("  850 Hz                 800\n");
270                     UARTprintf("  800 Hz                 750\n");
271                     UARTprintf("  750 Hz                 700\n");
272                     UARTprintf("  700 Hz                 650\n");
273                     UARTprintf("  650 Hz                 600\n");
274                     UARTprintf("  600 Hz                 550\n");
275                     UARTprintf("  550 Hz                 500\n");
276                     UARTprintf("  500 Hz                 450\n");
277                     UARTprintf("  450 Hz                 400\n");
278                     UARTprintf("  400 Hz                 350\n");
279                     UARTprintf("  350 Hz                 300\n");
280                     UARTprintf("  300 Hz                 250\n");
281                     UARTprintf("  250 Hz                 200\n");
282                     UARTprintf("  200 Hz                 150\n");
283                     UARTprintf("  150 Hz                 100\n");
284                     UARTprintf("  100 Hz                  50\n");
285                     UARTprintf("   50 Hz\n");
286                     UARTprintf("\n");
287                     UARTprintf("\n");
288                     UARTprintf("send_hex(HEX1 HEX2 HEX3 HEX4)              -> Sends four hex values to a sensor and expet a four
                            hex answer form the sensor\n");
289                     UARTprintf("get_config                                 -> Returns the actually config of the basestation\n")
                            ;
290
291                     UARTprintf("\n
                            ****************************************************************************************************\
                            n");
292
293                     uart_rx_flag = 0;    // Reset the UART flag
294                 } break;
295                 //************************************************************************
296                 case WHY: {
297                     UARTprintf("Why not?!?\n");
298                     uart_rx_flag = 0;    // Reset the UART flag
299                 } break;
300
301                 //************************************************************************
302                 // get_voltage(Sensor Address);
303                 // returns the voltage of the cell
304                 //************************************************************************
305                 case GET_VOLTAGE: {
306                     uint16_t volt_msb = 0;
307                     uint16_t volt_lsb = 0;
308
309                     UARTprintf("\n");
310                     uart_rx_flag = 0;    // Reset the UART flag
311
312                     if(check_input()) { // Check if the type command was correct
313
314                         // rx_char[12] and rx_char[13] are the adress of the sensor
315                         packet[0] = char_to_hex(rx_char[12],rx_char[13]);
316                         packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE;
317                         packet[2] = 0x00;
318                         packet[3] = 0x00;
319
320                         tx_data_length = 0;                                   // Normal Data length
321                         tx_packet(HEADER_LENGTH + tx_data_length, packet);     // Config CC1101 for data tranceive
322
323                         delay_ms(1000);      // wait till the next command is send
324
325                         // rx_char[12] and rx_char[13] are the adress of the sensor
326                         packet[0] = char_to_hex(rx_char[12],rx_char[13]);
327                         packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE;
328                         packet[2] = 0x00;
329                         packet[3] = 0x00;
330
331                         tx_data_length = 0;                                   // Normal Data length
332                         tx_packet(HEADER_LENGTH + tx_data_length, packet);     // Config CC1101 for data tranceive
333
334                         rx_data_length = 2;                                   // Normal Data length
```

```
335                     rx_packet(HEADER_LENGTH + rx_data_length);              // Config CC1101 for data receive
336
337                     // Decode the volage information form the sensor
338                     volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
339                     volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
340
341                     // Check if the timeout flag is set
342                     if (TIMEOUT_IS_SET) {
343                         if(!matlab_mode) {
344                             UARTprintf("No answer from sensor 0x%x!\n",packet[0]);
345                         }
346                     } else {
347                         if(!matlab_mode) {
348                             UARTprintf("%i\n",(volt_msb | volt_lsb));
349                         }else {
350                             UARTprintf("%i\n",(volt_msb | volt_lsb));
351                         }
352                     }
353                 }
354
355             } break;
356
357             //**********************************************************************
358             // get_goertzel();
359             // returns the real and imaginary part of the messurement data
360             //**********************************************************************
361             case GET_GORTZEL: {
362
363                 uart_rx_flag = 0;   // Reset the UART flag
364
365                 uint16_t num_samples = asynch_counter;
366                 uint16_t N = 0;     // Anzahl der Abtastpunke für die Berechnung
367                 uint16_t Np = 0;    // Np Anzahl Abtastwerte pro Periode
368
369                 uint16_t print_freq = (char_to_int2(rx_char[25],rx_char[26])*100 + char_to_int2(rx_char[27],rx_char[28]))
                        ;
370                 uint16_t print_freq2 = (char_to_int(rx_char[30]));
371
372                 m = (float) char_to_int2(rx_char[16],rx_char[17]);
373                 Fs_goertzel = (float) (char_to_int3(rx_char[19],rx_char[20],rx_char[21])*100 + char_to_int2(rx_char[22],
                        rx_char[23]));
374                 f_goertzel = (float) (char_to_int2(rx_char[25],rx_char[26])*100 + char_to_int2(rx_char[27],rx_char[28]) +
                         0.1*char_to_int(rx_char[30]));
375
376                 Np = Fs_goertzel/f_goertzel;
377                 N = m * Np;
378
379                 UARTprintf("\n");
380
381                 if(N > num_samples) {
382                     UARTprintf("****************************** ERROR
                            *****************************************************\n");
383                     UARTprintf("It is not possible to calculate these numbers of periodes, please check your input values
                            !!\n");
384                     UARTprintf("To calculate these number of periodes, you need %i samples. But you only have %i samples
                            .\n",N,num_samples);
385                     UARTprintf("****************************** ERROR
                            *****************************************************\n");
386                 }else {
387                     UARTprintf("Start to calculate the DFT values with %i samples\n",N);
388                     UARTprintf("The possible samples are %i\n",num_samples);
389                     UARTprintf("Number of periodes: %i\n",(int)m);
390                     UARTprintf("Burst frequency:    %i Hz\n",(int)Fs_goertzel);
391                     UARTprintf("Signal frequency:   %i.%i Hz\n",print_freq,print_freq2);
392
393                     // Send the number of periodes
394                     packet[0] = char_to_hex(rx_char[13],rx_char[14]);
395                     packet[1] = COMMAND_GOERTZEL_NUMB_PERI;
396                     packet[2] = char_to_int2(rx_char[16],rx_char[17]);
397                     packet[3] = 0x00;
398
399                     tx_data_length = 0;                                     // Normal Data length
400                     tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data tranceive
401
402                     delay_ms(100);
403
404                     // Send the Goertzel frequency
405                     packet[0] = char_to_hex(rx_char[13],rx_char[14]);
406                     packet[1] = COMMAND_GOERTZEL_FREQ;
407                     packet[2] = char_to_int3(rx_char[19],rx_char[20],rx_char[21]);
408                     packet[3] = char_to_int2(rx_char[22],rx_char[23]);
409
410                     tx_data_length = 0;                                     // Normal Data length
411                     tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data tranceive
412
```

```
413                 delay_ms(100);
414
415                 // Send the stimuli frequency
416                 packet[0] = char_to_hex(rx_char[13],rx_char[14]);
417                 packet[1] = COMMAND_GOERTZEL_STIMULI_FREQ;
418                 packet[2] = char_to_int2(rx_char[25],rx_char[26]);
419                 packet[3] = char_to_int2(rx_char[27],rx_char[28]);
420
421                 tx_data_length = 0;                          // Normal Data length
422                 tx_packet(HEADER_LENGTH + tx_data_length, packet);   // Config CC1101 for data tranceive
423
424                 delay_ms(100);
425
426                 // Send the stimuli frequency
427                 packet[0] = char_to_hex(rx_char[13],rx_char[14]);
428                 packet[1] = COMMAND_GOERTZEL_STIMULI_FREQ2;
429                 packet[2] = char_to_int(rx_char[30]);
430                 packet[3] = 0x00;
431
432                 tx_data_length = 0;                          // Normal Data length
433                 tx_packet(HEADER_LENGTH + tx_data_length, packet);   // Config CC1101 for data tranceive
434
435                 delay_ms(100);
436
437                 packet[0] = char_to_hex(rx_char[13],rx_char[14]);
438                 packet[1] = COMMAND_GOERTZEL;
439                 packet[2] = 0x00;
440                 packet[3] = 0x00;
441
442                 tx_data_length = 0;
443                 // Normal Data length
444                 tx_packet(HEADER_LENGTH + tx_data_length, packet);   // Config CC1101 for data tranceive
445
446                 rx_timeout  = 10000;                            // expand the delay time
447                 rx_data_length = 8;                          // Normal Data length
448                 rx_packet(HEADER_LENGTH + rx_data_length);        // Config CC1101 for data receive
449                 rx_timeout  = 2000;                          // resert the delay time
450
451                 // Decode the volage information form the sensor
452                 real_char.hex_data[3] = (rx_data[HEADER_LENGTH + 0] & 0xFF);
453                 real_char.hex_data[2] = (rx_data[HEADER_LENGTH + 1] & 0xFF);
454                 real_char.hex_data[1] = (rx_data[HEADER_LENGTH + 2] & 0xFF);
455                 real_char.hex_data[0] = (rx_data[HEADER_LENGTH + 3] & 0xFF);
456
457                 imag_char.hex_data[3] = (rx_data[HEADER_LENGTH + 4] & 0xFF);
458                 imag_char.hex_data[2] = (rx_data[HEADER_LENGTH + 5] & 0xFF);
459                 imag_char.hex_data[1] = (rx_data[HEADER_LENGTH + 6] & 0xFF);
460                 imag_char.hex_data[0] = (rx_data[HEADER_LENGTH + 7] & 0xFF);
461
462                 rx_data_length = 0;        // Normal Data length
463
464                 // Check if the timeout flag is set
465                 if (TIMEOUT_IS_SET) {
466                     UARTprintf("No answer from sensor 0x%x!\n",packet[0]);
467                 } else {
468                     // Printing the data
469                     if(real_char.hex_data[3] < 0x10) {
470                         UARTprintf("%i",0);
471                     }
472                     UARTprintf("%x",real_char.hex_data[3]);
473
474                     if(real_char.hex_data[2] < 0x10) {
475                         UARTprintf("%i",0);
476                     }
477                     UARTprintf("%x",real_char.hex_data[2]);
478
479                     if(real_char.hex_data[1] < 0x10) {
480                         UARTprintf("%i",0);
481                     }
482                     UARTprintf("%x",real_char.hex_data[1]);
483
484                     if(real_char.hex_data[0] < 0x10) {
485                         UARTprintf("%i",0);
486                     }
487                     UARTprintf("%x\n",real_char.hex_data[0]);
488
489                     if(imag_char.hex_data[3] < 0x10) {
490                         UARTprintf("%i",0);
491                     }
492                     UARTprintf("%x",imag_char.hex_data[3]);
493
494                     if(imag_char.hex_data[2] < 0x10) {
495                         UARTprintf("%i",0);
496                     }
497                     UARTprintf("%x",imag_char.hex_data[2]);
```

```
498
499                 if(imag_char.hex_data[1] < 0x10) {
500                     UARTprintf("%i",0);
501                 }
502                 UARTprintf("%x",imag_char.hex_data[1]);
503
504                 if(imag_char.hex_data[0] < 0x10) {
505                     UARTprintf("%i",0);
506                 }
507                 UARTprintf("%x\n",imag_char.hex_data[0]);
508             }
509
510             // Calculate the Goertzel with the current data
511
512             int index;
513             int k;
514             int N_int;
515
516             real_char.float_data = 0.0;
517             imag_char.float_data = 0.0;
518
519             bn_0 = 0;    // Reset the coefficients
520             bn_1 = 0;    // Reset the coefficients
521             bn_2 = 0;    // Reset the coefficients
522
523             Np = Fs_goertzel/f_goertzel;     // Np Anzahl Abtastwerte pro Periode
524             N_float = m * Np;                // Anzahl der Abtastpunke für die Berechnung
525             N_int = (int) N_float;
526
527
528             // Precalculate the coefficients and the sine/cosine
529             k = (int) (0.5 + ((N_float * f_goertzel) / Fs_goertzel));
530             omega = (2.0 * 3.141592654 * k) / N_float;
531             sine = sinf(omega);
532             cosine = cosf(omega);
533             coeff = 2.0 * cosine;
534
535             //*** BEGIN the Goertzel algorithm ***//
536             for (index = 0; index < N_int + 1; index++) {
537
538                 if(index == N_int) {     // the last sample have to be zero
539                     bn_0 = coeff * bn_1 - bn_2 +  0;
540                 } else {
541                     f_voltage = (((0.00080586)*burst_current[index])*(1/0.1));
542
543                     bn_0 = coeff * bn_1 - bn_2 + f_voltage;
544                 }
545
546                 bn_2 = bn_1;
547                 bn_1 = bn_0;
548             }
549
550             real_char.float_data = (bn_1 - bn_2 * cosine);
551             imag_char.float_data = (bn_2 * sine);
552
553             //*** END the Goertzel algorithm ***//
554
555             // Printing the data
556             if(real_char.hex_data[3] < 0x10) {
557                 UARTprintf("%i",0);
558             }
559             UARTprintf("%x",real_char.hex_data[3]);
560
561             if(real_char.hex_data[2] < 0x10) {
562                 UARTprintf("%i",0);
563             }
564             UARTprintf("%x",real_char.hex_data[2]);
565
566             if(real_char.hex_data[1] < 0x10) {
567                 UARTprintf("%i",0);
568             }
569             UARTprintf("%x",real_char.hex_data[1]);
570
571             if(real_char.hex_data[0] < 0x10) {
572                 UARTprintf("%i",0);
573             }
574             UARTprintf("%x\n",real_char.hex_data[0]);
575
576             if(imag_char.hex_data[3] < 0x10) {
577                 UARTprintf("%i",0);
578             }
579             UARTprintf("%x",imag_char.hex_data[3]);
580
581             if(imag_char.hex_data[2] < 0x10) {
582                 UARTprintf("%i",0);
```

```
583                         }
584                         UARTprintf("%x",imag_char.hex_data[2]);
585
586                         if(imag_char.hex_data[1] < 0x10) {
587                             UARTprintf("%i",0);
588                         }
589                         UARTprintf("%x",imag_char.hex_data[1]);
590
591                         if(imag_char.hex_data[0] < 0x10) {
592                             UARTprintf("%i",0);
593                         }
594                         UARTprintf("%x\n",imag_char.hex_data[0]);
595                     }
596
597
598             } break;
599
600             //***********************************************************************
601             // get_config(ADDRESS OF THE SENSOR);
602             // returns the actual config of the cellsensor
603             //***********************************************************************
604             case GET_CONFIG_ZS: {
605                 uart_rx_flag = 0;   // Reset the UART flag
606
607                 // rx_char[7] and rx_char[8] are the adress of the sensor
608                 packet[0] = char_to_hex(rx_char[11],rx_char[12]);
609                 packet[1] = COMMAND_SW;
610                 packet[2] = 0x00;
611                 packet[3] = 0x00;
612
613                 tx_data_length = 0;                                  // Normal Data length
614                 tx_packet(HEADER_LENGTH + tx_data_length, packet);   // Config CC1101 for data tranceive
615
616                 rx_data_length = 18;                                 // Normal Data length
617                 rx_packet(HEADER_LENGTH + rx_data_length);           // Config CC1101 for data receive
618
619                 // Check if the timeout flag is set
620                 if (TIMEOUT_IS_SET) {
621                     UARTprintf("No answer from sensor 0x%x!\n",packet[0]);
622                 } else {
623                     UARTprintf("**** Software of sensor 0x%x *****\n",packet[0]);
624                     UARTprintf("Software version         : V%i\n", rx_data[4]);
625                     UARTprintf("\n");
626                     if(rx_data[5] == 0x00) {
627                         UARTprintf("Selected ADC             : CC430 ADC\n");
628                     } else if(rx_data[5] == 0x01) {
629                         UARTprintf("Selected ADC             : AD7691\n");
630                     }
631                     UARTprintf("\n");
632                     UARTprintf("CC1101 data rate         : %i kbps\n", rx_data[6]);
633                     UARTprintf("Burst frame lenght       : %i byte per frame\n", rx_data[7]);
634                     UARTprintf("\n");
635
636                     UARTprintf("\n**** Balancing configuartion ****\n");
637                     UARTprintf("Target balancing value   : %i \n", (rx_data[8] << 8) + rx_data[9] );
638                     UARTprintf("Upper balancing limit    : %i \n", (rx_data[10] << 8) + rx_data[11]);
639                     UARTprintf("Lower balancing limit    : %i \n", (rx_data[12] << 8) + rx_data[13]);
640                     UARTprintf("Upper alarm limit        : %i \n", (rx_data[14] << 8) + rx_data[15]);
641                     UARTprintf("Lower alarm limit        : %i \n", (rx_data[16] << 8) + rx_data[17] );
642
643                     UARTprintf("\n**** Preprocessing information ****\n");
644                     UARTprintf("Rheostat for the offset  : 0x%x \n", (rx_data[18] << 8) + (rx_data[19] << 0));
645                     UARTprintf("Rheostat for the gain    : 0x%x \n", (rx_data[20] << 8) + (rx_data[21] << 0));
646
647                 }
648
649             } break;
650
651             //************************************************************
652             // "send_hex" sends four hex values with the cc1101
653             // Example: send_hex(00 04 00 00) -> sends the hex values "0x00 0x04 0x00 0x00"
654             //************************************************************
655             case SEND_HEX: {
656                 UARTprintf("Send HEX\n");
657                 uart_rx_flag = 0;   // Reset the UART flag
658
659                 if(check_input()) { // Check if the type command was correct
660                     packet[0] = char_to_hex( rx_char[9],rx_char[10]);
661                     packet[1] = char_to_hex(rx_char[12],rx_char[13]);
662                     packet[2] = char_to_hex(rx_char[15],rx_char[16]);
663                     packet[3] = char_to_hex(rx_char[18],rx_char[19]);
664
665                     tx_data_length = 0;         // Normal Data length
666                     tx_packet(4, packet);       // Config CC1101 for data tranceive
667
```

```
668                 rx_data_length = 2;                                         // Normal Data length
669                 rx_packet(HEADER_LENGTH + rx_data_length);                  // Config CC1101 for data receive
670 
671                 // Check if the timeout flag is set
672                 if (TIMEOUT_IS_SET) {
673                     if(!matlab_mode) {
674                         UARTprintf("Timeout pending!\n");
675                     }
676                 } else {
677                     uint8_t index;
678                     for (index = 0; index < (HEADER_LENGTH + rx_data_length); index++) {
679                         UARTprintf("%x\n",rx_data[index]);
680                     }
681                     UARTprintf("\n");
682                 }
683             }
684 
685         } break;
686 
687         //***********************************************************
688         // get_config(ADDRESS OF THE BASESTATION);
689         // returns the actual config of the basestation
690         //***********************************************************
691         case GET_CONFIG_BS: {
692             uart_rx_flag = 0;   // Reset the UART flag
693 
694             if(check_input()) { // Check if the type command was correct
695                 UARTprintf("**** Software base station *****\n");
696                 UARTprintf("Software version         : V%i\n",g_sw_version);
697 
698                 UARTprintf("\n**** System configuraton *****\n");
699                 UARTprintf("Clock speed              : %i MHz\n",ui32SrcClock/1000000);
700                 UARTprintf("Wait to RX data          : %i ms\n",rx_timeout);
701                 UARTprintf("Wait to TX data          : %i ms\n",tx_timeout);
702 
703                 UARTprintf("\n**** SPI configuraton ********\n");
704                 UARTprintf("SPI speed                : %i kHz\n",spiBitRate/1000);
705                 UARTprintf("SPI number of bits       : %i Bit\n",spiBit);
706                 if(spiConfig == SSI_FRF_MOTO_MODE_0) {
707                     UARTprintf("SPI config               : Mode 0\n",spiBit);
708                 } else if(spiConfig == SSI_FRF_MOTO_MODE_1) {
709                     UARTprintf("SPI config               : Mode 1\n",spiBit);
710                 } else if(spiConfig == SSI_FRF_MOTO_MODE_2) {
711                     UARTprintf("SPI config               : Mode 2\n",spiBit);
712                 } else if(spiConfig == SSI_FRF_MOTO_MODE_3) {
713                     UARTprintf("SPI config               : Mode 3\n",spiBit);
714                 } else if(spiConfig == SSI_FRF_TI) {
715                     UARTprintf("SPI config               : TI format\n",spiBit);
716                 } else if(spiConfig == SSI_FRF_NMW) {
717                     UARTprintf("SPI config               : National MicroWire frame format\n",spiBit);
718                 }
719 
720                 UARTprintf("\n**** CC1101 configuraton *****\n");
721                 UARTprintf("CC1101 data rate         : %i kbps\n",brate);
722                 UARTprintf("Adress this station      : 0x%x\n", adress_this_station);
723 
724                 UARTprintf("\n**** UART configuraton *******\n");
725                 UARTprintf("UART port number         : %i\n",ui32PortNum);
726                 UARTprintf("UART data rate           : %i Baud\n",ui32Baud);
727                 UARTprintf("UART clock               : %i MHz\n",ui32SrcClock/1000000);
728                 UARTprintf("RX Terminator            : CR\n");
729                 UARTprintf("TX Terminator            : CR + LF\n");
730 
731                 UARTprintf("\n**** Matlab mode *************\n");
732                 if(!matlab_mode) {
733                     UARTprintf("Matlab mode is           : OFF\n");
734                 } else {
735                     UARTprintf("Matlab mode is           : ON\n");
736                 }
737 
738                 UARTprintf("\n**** DCO calibration *************\n");
739                 UARTprintf("Number of measurements   : %i\n",clk_numb_counts);
740                 UARTprintf("Tolerance of the DCO     : %i Ticks\n",clk_tolerance);
741 
742 
743             }
744 
745         } break;
746 
747         //***********************************************************
748         // "set_brate" mode.
749         //***********************************************************
750         case SET_BRATE: {
751             UARTprintf("Set Boud rate!\n");
752             uart_rx_flag = 0;   // Reset the UART flag
```

```
753                         check_input();
754
755                         if(rx_char[12] != 0x30)
756                             brate = (10*char_to_int(rx_char[10])) + (char_to_int(rx_char[12]));
757                         else {
758                             brate = (100*char_to_int(rx_char[10])) + (10*char_to_int(rx_char[11])) + (char_to_int(rx_char[12]));
759                         }
760
761                     } break;
762
763                     //*************************************************************
764                     // "matlabmode" mode.
765                     //*************************************************************
766                     case MATLABMODE: {
767                         uart_rx_flag = 0;   // Reset the UART flag
768                         //check_input();
769
770                         if((rx_char[11] == 0x6F) && (rx_char[12] == 0x6E)) {
771                             matlab_mode = 1;
772                             UARTprintf("Matlab mode is on\n");
773                         }else if((rx_char[11] == 0x6F) && (rx_char[12] == 0x66) && (rx_char[13] == 0x66)) {
774                             matlab_mode = 0;
775                             UARTprintf("Matlab mode is off\n");
776                         }
777
778                     } break;
779
780                     //*************************************************************
781                     // set_preprocessing();
782                     // set the analog preprocessing
783                     //*************************************************************
784                     case SET_PROPROCESSING: {
785
786                     uart_rx_flag = 0;   // Reset the UART flag
787
788                     UARTprintf("Set preprocessing on sensor 0x%x\n", char_to_hex(rx_char[18],rx_char[19]));
789
790                     // rx_char[12] and rx_char[13] are the adress of the sensor
791                     packet[0] = char_to_hex(rx_char[18],rx_char[19]);
792                     packet[1] = COMMAND_SET_PREPROCESSING;
793                     packet[2] = 0x00;
794                     packet[3] = 0x00;
795
796                     tx_data_length = 0;                                         // Normal Data length
797                     tx_packet(HEADER_LENGTH + tx_data_length, packet);          // Config CC1101 for data tranceive
798
799                     } break;
800                     //*************************************************************
801                     // Burstmode. Detect fhe frequency and the number of values and start the burst mode
802                     //*************************************************************
803                     case BURST: {
804                         uint8_t frequency_true = 0;
805                         uint8_t value = 0;
806                         uint8_t value_true = 0;
807                         uint8_t position = 0;
808
809                         uart_rx_flag = 0;   // Reset the UART flag
810
811                         if((rx_char[6] == 0x31) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) && (
                                rx_char[10] == 0x30) && (rx_char[11] == 0x30)) {
812                             burst_freq = BURST_FREQ_100000HZ;
813                             position = 12;
814                             frequency_true = 1;
815                         } else if((rx_char[6] == 0x39) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                                (rx_char[10] == 0x30)) {
816                             burst_freq = BURST_FREQ_95000HZ;
817                             position = 11;
818                             frequency_true = 1;
819                         } else if((rx_char[6] == 0x39) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                                (rx_char[10] == 0x30)) {
820                             burst_freq = BURST_FREQ_90000HZ;
821                             position = 11;
822                             frequency_true = 1;
823                         } else if((rx_char[6] == 0x38) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                                (rx_char[10] == 0x30)) {
824                             burst_freq = BURST_FREQ_85000HZ;
825                             position = 11;
826                             frequency_true = 1;
827                         } else if((rx_char[6] == 0x38) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                                (rx_char[10] == 0x30)) {
828                             burst_freq = BURST_FREQ_80000HZ;
829                             position = 11;
830                             frequency_true = 1;
831                         } else if((rx_char[6] == 0x37) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                                (rx_char[10] == 0x30)) {
```

```
832                 burst_freq = BURST_FREQ_75000HZ;
833                 position = 11;
834                 frequency_true = 1;
835             } else if((rx_char[6] == 0x37) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
836                 burst_freq = BURST_FREQ_70000HZ;
837                 position = 11;
838                 frequency_true = 1;
839             } else if((rx_char[6] == 0x36) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
840                 burst_freq = BURST_FREQ_65000HZ;
841                 position = 11;
842                 frequency_true = 1;
843             } else if((rx_char[6] == 0x36) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
844                 burst_freq = BURST_FREQ_60000HZ;
845                 position = 11;
846                 frequency_true = 1;
847             } else if((rx_char[6] == 0x35) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
848                 burst_freq = BURST_FREQ_55000HZ;
849                 position = 11;
850                 frequency_true = 1;
851             } else if((rx_char[6] == 0x35) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
852                 burst_freq = BURST_FREQ_50000HZ;
853                 position = 11;
854                 frequency_true = 1;
855             } else if((rx_char[6] == 0x34) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
856                 burst_freq = BURST_FREQ_45000HZ;
857                 position = 11;
858                 frequency_true = 1;
859             } else if((rx_char[6] == 0x34) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
860                 burst_freq = BURST_FREQ_40000HZ;
861                 position = 11;
862                 frequency_true = 1;
863             } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x38) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
864                 burst_freq = BURST_FREQ_38000HZ;
865                 position = 11;
866                 frequency_true = 1;
867             } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x36) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
868                 burst_freq = BURST_FREQ_36000HZ;
869                 position = 11;
870                 frequency_true = 1;
871             } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x34) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
872                 burst_freq = BURST_FREQ_34000HZ;
873                 position = 11;
874                 frequency_true = 1;
875             } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x32) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
876                 burst_freq = BURST_FREQ_32000HZ;
877                 position = 11;
878                 frequency_true = 1;
879             } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
880                 burst_freq = BURST_FREQ_30000HZ;
881                 position = 11;
882                 frequency_true = 1;
883             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x38) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
884                 burst_freq = BURST_FREQ_28000HZ;
885                 position = 11;
886                 frequency_true = 1;
887             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x36) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
888                 burst_freq = BURST_FREQ_26000HZ;
889                 position = 11;
890                 frequency_true = 1;
891             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x34) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
892                 burst_freq = BURST_FREQ_24000HZ;
893                 position = 11;
894                 frequency_true = 1;
895             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x32) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
896                 burst_freq = BURST_FREQ_22000HZ;
897                 position = 11;
898                 frequency_true = 1;
899             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
```

```
900                 burst_freq = BURST_FREQ_20000HZ;
901                 position = 11;
902                 frequency_true = 1;
903             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x38) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
904                 burst_freq = BURST_FREQ_18000HZ;
905                 position = 11;
906                 frequency_true = 1;
907             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x36) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
908                 burst_freq = BURST_FREQ_16000HZ;
909                 position = 11;
910                 frequency_true = 1;
911             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x34) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
912                 burst_freq = BURST_FREQ_14000HZ;
913                 position = 11;
914                 frequency_true = 1;
915             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x32) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
916                 burst_freq = BURST_FREQ_12000HZ;
917                 position = 11;
918                 frequency_true = 1;
919             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30) &&
                    (rx_char[10] == 0x30)) {
920                 burst_freq = BURST_FREQ_10000HZ;
921                 position = 11;
922                 frequency_true = 1;
923             } else if((rx_char[6] == 0x38) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30)) {
924                 burst_freq = BURST_FREQ_8000HZ;
925                 position = 10;
926                 frequency_true = 1;
927             } else if((rx_char[6] == 0x36) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30)) {
928                 burst_freq = BURST_FREQ_6000HZ;
929                 position = 10;
930                 frequency_true = 1;
931             } else if((rx_char[6] == 0x34) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30)) {
932                 burst_freq = BURST_FREQ_4000HZ;
933                 position = 10;
934                 frequency_true = 1;
935             } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30)) {
936                 burst_freq = BURST_FREQ_2000HZ;
937                 position = 10;
938                 frequency_true = 1;
939             } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30) && (rx_char[9] == 0x30)) {
940                 burst_freq = BURST_FREQ_1000HZ;
941                 position = 10;
942                 frequency_true = 1;
943             } else if((rx_char[6] == 0x39) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
944                 burst_freq = BURST_FREQ_950HZ;
945                 position = 9;
946                 frequency_true = 1;
947             } else if((rx_char[6] == 0x39) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
948                 burst_freq = BURST_FREQ_900HZ;
949                 position = 9;
950                 frequency_true = 1;
951             } else if((rx_char[6] == 0x38) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
952                 burst_freq = BURST_FREQ_850HZ;
953                 position = 9;
954                 frequency_true = 1;
955             } else if((rx_char[6] == 0x38) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
956                 burst_freq = BURST_FREQ_800HZ;
957                 position = 9;
958                 frequency_true = 1;
959             } else if((rx_char[6] == 0x37) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
960                 burst_freq = BURST_FREQ_750HZ;
961                 position = 9;
962                 frequency_true = 1;
963             } else if((rx_char[6] == 0x37) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
964                 burst_freq = BURST_FREQ_700HZ;
965                 position = 9;
966                 frequency_true = 1;
967             } else if((rx_char[6] == 0x36) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
968                 burst_freq = BURST_FREQ_650HZ;
969                 position = 9;
970                 frequency_true = 1;
971             } else if((rx_char[6] == 0x36) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
972                 burst_freq = BURST_FREQ_600HZ;
973                 position = 9;
974                 frequency_true = 1;
975             } else if((rx_char[6] == 0x35) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
976                 burst_freq = BURST_FREQ_550HZ;
977                 position = 9;
978                 frequency_true = 1;
979             } else if((rx_char[6] == 0x35) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
```

```
980                     burst_freq = BURST_FREQ_500HZ;
981                     position = 9;
982                     frequency_true = 1;
983                 } else if((rx_char[6] == 0x34) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
984                     burst_freq = BURST_FREQ_450HZ;
985                     position = 9;
986                     frequency_true = 1;
987                 } else if((rx_char[6] == 0x34) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
988                     burst_freq = BURST_FREQ_400HZ;
989                     position = 9;
990                     frequency_true = 1;
991                 } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
992                     burst_freq = BURST_FREQ_350HZ;
993                     position = 9;
994                     frequency_true = 1;
995                 } else if((rx_char[6] == 0x33) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
996                     burst_freq = BURST_FREQ_300HZ;
997                     position = 9;
998                     frequency_true = 1;
999                 } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
1000                    burst_freq = BURST_FREQ_250HZ;
1001                    position = 9;
1002                    frequency_true = 1;
1003                } else if((rx_char[6] == 0x32) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
1004                    burst_freq = BURST_FREQ_200HZ;
1005                    position = 9;
1006                    frequency_true = 1;
1007                } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x35) && (rx_char[8] == 0x30)) {
1008                    burst_freq = BURST_FREQ_150HZ;
1009                    position = 9;
1010                    frequency_true = 1;
1011                } else if((rx_char[6] == 0x31) && (rx_char[7] == 0x30) && (rx_char[8] == 0x30)) {
1012                    burst_freq = BURST_FREQ_100HZ;
1013                    position = 9;
1014                    frequency_true = 1;
1015                } else if((rx_char[6] == 0x35) && (rx_char[7] == 0x30)) {
1016                    burst_freq = BURST_FREQ_50HZ;
1017                    position = 8;
1018                    frequency_true = 1;
1019                }
1020
1021                if((rx_char[position+1] == 0x32) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30) && (
                        rx_char[position+4] == 0x30)) {
1022                    value = BURST_VALUES_2500;
1023                    value_true = 1;
1024                } else if((rx_char[position+1] == 0x32) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        && (rx_char[position+4] == 0x30)) {
1025                    value = BURST_VALUES_2000;
1026                    value_true = 1;
1027                } else if((rx_char[position+1] == 0x31) && (rx_char[position+2] == 0x39) && (rx_char[position+3] == 0x30)
                        && (rx_char[position+4] == 0x30)) {
1028                    value = BURST_VALUES_1900;
1029                    value_true = 1;
1030                } else if((rx_char[position+1] == 0x31) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        && (rx_char[position+4] == 0x30)) {
1031                    value = BURST_VALUES_1500;
1032                    value_true = 1;
1033                } else if((rx_char[position+1] == 0x31) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        && (rx_char[position+4] == 0x30)) {
1034                    value = BURST_VALUES_1000;
1035                    value_true = 1;
1036                } else if((rx_char[position+1] == 0x39) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1037                    value = BURST_VALUES_900;
1038                    value_true = 1;
1039                } else if((rx_char[position+1] == 0x38) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1040                    value = BURST_VALUES_850;
1041                    value_true = 1;
1042                } else if((rx_char[position+1] == 0x38) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1043                    value = BURST_VALUES_800;
1044                    value_true = 1;
1045                } else if((rx_char[position+1] == 0x37) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1046                    value = BURST_VALUES_750;
1047                    value_true = 1;
1048                } else if((rx_char[position+1] == 0x37) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1049                    value = BURST_VALUES_700;
1050                    value_true = 1;
1051                } else if((rx_char[position+1] == 0x36) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1052                    value = BURST_VALUES_650;
1053                    value_true = 1;
```

```
1054                } else if((rx_char[position+1] == 0x36) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1055                    value = BURST_VALUES_600;
1056                    value_true = 1;
1057                } else if((rx_char[position+1] == 0x35) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1058                    value = BURST_VALUES_550;
1059                    value_true = 1;
1060                } else if((rx_char[position+1] == 0x35) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1061                    value = BURST_VALUES_500;
1062                    value_true = 1;
1063                } else if((rx_char[position+1] == 0x34) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1064                    value = BURST_VALUES_450;
1065                    value_true = 1;
1066                } else if((rx_char[position+1] == 0x34) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1067                    value = BURST_VALUES_400;
1068                    value_true = 1;
1069                } else if((rx_char[position+1] == 0x33) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1070                    value = BURST_VALUES_350;
1071                    value_true = 1;
1072                } else if((rx_char[position+1] == 0x33) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1073                    value = BURST_VALUES_300;
1074                    value_true = 1;
1075                } else if((rx_char[position+1] == 0x32) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1076                    value = BURST_VALUES_250;
1077                    value_true = 1;
1078                } else if((rx_char[position+1] == 0x32) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1079                    value = BURST_VALUES_200;
1080                    value_true = 1;
1081                } else if((rx_char[position+1] == 0x31) && (rx_char[position+2] == 0x35) && (rx_char[position+3] == 0x30)
                        ) {
1082                    value = BURST_VALUES_150;
1083                    value_true = 1;
1084                } else if((rx_char[position+1] == 0x31) && (rx_char[position+2] == 0x30) && (rx_char[position+3] == 0x30)
                        ) {
1085                    value = BURST_VALUES_100;
1086                    value_true = 1;
1087                } else if((rx_char[position+1] == 0x35) && (rx_char[position+2] == 0x30)) {
1088                    value = BURST_VALUES_50;
1089                    value_true = 1;
1090                }
1091
1092                if ((frequency_true) && (value_true)) {
1093                    UARTprintf("Burst has start\n");
1094                    burst_start(burst_freq, value);
1095                    UARTprintf("Burst has end!\n");
1096                } else if (frequency_true == 0) {
1097                    UARTprintf("\n***** ERROR *****\n");
1098                    UARTprintf("Frequency is not valid!\nValid frequencys are:\n");
1099                    UARTprintf("10000Hz 8000Hz\n 6000Hz 4000Hz\n 2000Hz 1000Hz\n  950Hz 900Hz\n  850Hz 800Hz\n  750Hz 700
                            Hz\n  650Hz 600Hz\n  550Hz 500Hz\n  450Hz 400Hz\n  350Hz 300Hz\n  250Hz 200Hz\n  150Hz 100Hz\n
                              50Hz\n");
1100                    UARTprintf("\nFor example: A burst measurement with a frequency of\n1000Hz an 700 values type the
                            follow command in the command line:\n");
1101                    UARTprintf("burst(1000,700)\n");
1102                    UARTprintf("\n***** ERROR *****\n");
1103                } else if (value_true == 0) {
1104                    UARTprintf("\n***** ERROR *****\n");
1105                    UARTprintf("Number of values are not valid!\nValid values are:\n");
1106                    UARTprintf("1.000\n900\n850\n800\n750\n700\n650\n600\n550\n500\n450\n400\n350\n300\n250\n200\n150\
                            n100\n50\n");
1107                    UARTprintf("\nFor example: A burst measurement with a frequency of\n1000Hz an 700 values type the
                            follow command in the command line:\n");
1108                    UARTprintf("burst(1000,700)\n");
1109                    UARTprintf("\n***** ERROR *****\n");
1110                };
1111
1112            } break;
1113
1114            //**********************************************************
1115            // get_burst(ADDRESS OF THE SENSOR)
1116            // returns the burst data of the cell sensor
1117            //**********************************************************
1118            case GET_BURST: {
1119                uint8_t frame_number = 0;
1120                uint16_t volt_msb = 0;
1121                uint16_t volt_lsb = 0;
1122                uint16_t index = 0;
```

```
1123                uint16_t gain = 0;
1124
1125                uart_rx_flag = 0;   // Reset the UART flag
1126
1127
1128                // rx_char[12] and rx_char[13] are the adress of the sensor
1129                packet[0] = char_to_hex(rx_char[10],rx_char[11]);
1130                packet[1] = COMMAND_DOWNLINK_BURST_DATA_RQ;
1131                packet[2] = 0x00;
1132                packet[3] = 0x00;
1133
1134                tx_data_length = 0;                              // Normal Data length
1135                tx_packet(HEADER_LENGTH + tx_data_length, packet);    // Config CC1101 for data tranceive
1136
1137                rx_data_length = 0;                              // Normal Data length
1138                rx_packet(HEADER_LENGTH + rx_data_length);       // Config CC1101 for data receive
1139
1140                // Check if the timeout flag is set
1141                if (TIMEOUT_IS_SET) {
1142                    UARTprintf("No answer from sensor 0x%x!\n",packet[0]);
1143                } else {
1144
1145                    frame_counter   = rx_data[2];
1146                    rx_data_length  = rx_data[3];
1147                    seq_number = 0;             // Sequenznummer zurücksetzen
1148
1149                    if(frame_counter == 0)
1150                        UARTprintf("The sensor has answer, but there are no burst data available on sensor 0x%x!\n",
                                packet[0]);
1151
1152                    for(frame_number = 0;frame_number<frame_counter;frame_number++) {
1153                        delay_ms(100);  // wait befor the next package is send
1154
1155                        // rx_char[12] and rx_char[13] are the adress of the sensor
1156                        packet[0] = char_to_hex(rx_char[10],rx_char[11]);
1157                        packet[1] = COMMAND_DOWNLINK_BURST_DATA_RX;
1158                        packet[2] = frame_number;
1159                        packet[3] = 0x00;
1160
1161                        tx_data_length = 0;
1162                        tx_packet(HEADER_LENGTH + tx_data_length, packet);
1163
1164                        // The rx_data_length ist set from the sensor after the burst data request
1165                        rx_packet(HEADER_LENGTH + rx_data_length);
1166
1167                        // Check if the timeout flag is set
1168                        if (TIMEOUT_IS_SET) {
1169                            UARTprintf("Connection lost to sensor 0x%x!\n",packet[0]);
1170                        } else {
1171                            index = 0;
1172                            // Decode the volage information form the sensor
1173                            for(seq_number = 0; seq_number <= rx_data_length; seq_number++) {
1174
1175                                gain = (rx_data[2] << 8) & 0x0F00;
1176                                gain |= (rx_data[3] << 0) & 0xFF;
1177
1178                                if((rx_data[2] >> 4) == 0x00) { // 12Bit ADC
1179                                    // BEGIN For 12 bit ADC
                                            **************************************************************************
1180                                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0xFF)) << 8) & 0xFF00;
1181                                    seq_number++;
1182                                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0xFF)) << 0) & 0x00FF;
1183
1184                                    burst_data[index] = volt_msb | volt_lsb;
1185                                    // END For 12 bit ADC
                                            ****************************************************************************
1186                                } else {
1187                                    // For testing AD7691 BEGIN
                                            **************************************************************************
1188                                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0xFF)) << 8) & 0xFF00;
1189                                    seq_number++;
1190                                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0xFF)) << 0) & 0x00FF;
1191
1192                                    burst_data[index] = (volt_msb | volt_lsb);  // for AD7691 testing on the cell sensor
                                            21.01.15 NS
1193                                    burst_data[index] = burst_data[index] << 2; // for AD7691 testing on the cell sensor
                                            21.01.15 NS
1194                                    // For testing AD7691 END
                                            ****************************************************************************
1195                                }
1196                                index++;
1197                            }
1198                            // Print the burst data
1199                            for (index = 0; index < (rx_data_length/2); index++)
1200                                //UARTprintf("Frame: %i    Nr: %i    Value: %i\n",frame_number, index, burst_data[index]);
```

```
1201                    UARTprintf("%i  ",burst_data[index]);
1202                }
1203            }
1204            UARTprintf("Gain: 0x%x \n",gain);
1205
1206            UARTprintf("Printing the current data\n");
1207
1208            for (index = 0; index < asynch_counter; index++)
1209                UARTprintf("%i  ",burst_current[index]);
1210
1211            UARTprintf("\nSending burst data has end\n\n",packet[0]);
1212
1213        }
1214
1215    } break;
1216
1217    //************************************************************
1218    // cali_burst(ADDRESS OF THE CELL SENSOR)
1219    // calibrate mode to detect the delay time between the base station and the cell sensor
1220    //************************************************************
1221    case CALI_BURST: {
1222        uint32_t spiSendData[2];
1223        uint16_t time_zs = 0;
1224
1225        uart_rx_flag = 0;    // Reset the UART flag
1226
1227        GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0x00);     // Testpin
1228        //B4_ON;
1229        packet[0] = BROADCAST;
1230        packet[1] = COMMAND_CALI_BURST;
1231        packet[2] = 0;
1232        packet[3] = 0;
1233
1234        tx_data_length = 0;                                  // Normal Data length
1235        tx_packet(HEADER_LENGTH + tx_data_length, packet);   // Config CC1101 for data tranceive
1236        //B4_OFF;
1237
1238        //*** Grundeinstellungen ****************************************************
1239        // GDO0 ist Output und GDO2 ist Input
1240        CC1101_GDO2_IRQ_DISABLE;
1241        CC1101_GDO2_CLEAR_IRQ;
1242        CC1101_GDO2_IRQ_FALLING_EDGE;        // IRQ is trigger on a falling egde
1243
1244        CC1101_GDO0_SET_OUTPUT;          // GDO0 is an output
1245        CC1101_GDO0_SET_HIGH;            // Set GDO0 on high
1246
1247        IRQ_SET_CALI_BURST;              // Auf den ersten IRQ warten
1248        RX_CALI_SET;                     // RX Flag setzen
1249        //******************************************************************
1250        // CC1101 Konfigurieren
1251        cc1101_reset();                  // Reset all registers to their default values and go to idle state
1252        cc1101_config_burst_cali();      // Configure the transceiver for sending the wakeup signal
1253        TIMER2config();                  // Timer to messure the triggerspace
1254        //*** Puls senden ***************
1255
1256        spiSendData[0] = CC1101_CS_STX;
1257        SPI_send_data(1,spiSendData);
1258
1259        B4_ON;
1260        delay_ms(50);                    // Timer till the sensor is ready (6ms)
1261        //*** AB HIER GEHT ES LOS ********
1262        CC1101_GDO0_SET_LOW;
1263        GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_4, 0xFF);     // Testpin
1264        TIMER2_START;                    // Start timer 2
1265
1266        /*** Puls empfangen *****/
1267        spiSendData[0] = CC1101_CS_SIDLE;
1268        SPI_send_data(1, spiSendData);
1269
1270        spiSendData[0] = CC1101_CS_SRX;
1271        SPI_send_data(1, spiSendData);
1272        delay_us(1800); // 1
1273
1274
1275        CC1101_GDO2_CLEAR_IRQ;           // GDO2 Clear
1276        CC1101_GDO2_IRQ_ENABLE;          // GDO2 Enable
1277        while(RX_CALI_IS_SET);           // Warten bis Impuls empfangen wurde
1278
1279        cc1101_reset();                  // Reset all registers to their default values and go to idle state
1280        rx_data_length = 2;                                  // Normal Data length
1281        rx_packet(HEADER_LENGTH + rx_data_length);           // Config CC1101 for data receive
1282
1283        if (TIMEOUT_IS_SET) {
1284            UARTprintf("Connection lost to sensor 0x%x!\n",packet[0]);
1285        } else {
```

```
1286
1287                time_zs = 0xFFFF;
1288                time_zs = (rx_data[4] << 8) & 0xFF00;
1289                time_zs |= (rx_data[5] << 0) & 0x00FF;
1290
1291                UARTprintf("Laufzeit: %i\n",adc_value[0]);
1292                UARTprintf("Laufzeit: %i\n",time_zs);
1293
1294                packet[0] = BROADCAST;
1295                packet[1] = COMMAND_DOWNLINK_BURST_CHECK;
1296                packet[2] = 0x00;
1297                packet[3] = 0x00;
1298            };
1299
1300            tx_data_length = 0;                                 // Normal Data length
1301            tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data tranceive
1302
1303
1304
1305        } break;
1306
1307        //************************************************************
1308        // cali_clk(ADDRESS OF THE SENSOR)
1309        // calibrate mode to detect the delay time between the base station and the cell sensor
1310        //************************************************************
1311        case CALI_CLK: {
1312
1313            uart_rx_flag = 0;    // Reset the UART flag
1314
1315            uint32_t frequency = 0;
1316            uint32_t index = 0;
1317
1318            packet[0] = BROADCAST;
1319            packet[1] = COMMAND_CALI_CLK;
1320            packet[2] = 0;
1321            packet[3] = 0;
1322
1323            tx_data_length = 0;                                 // Normal Data length
1324            tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data tranceive
1325
1326            do{     // Schleife wird solange ausgeführt bis Takt ok ist
1327                //*** Grundeinstellungen ****************************************************
1328                // GDO0 ist Output und GDO2 ist Input
1329                CC1101_GDO2_IRQ_DISABLE;
1330                CC1101_GDO2_CLEAR_IRQ;
1331                CC1101_GDO2_IRQ_FALLING_EDGE;    // IRQ is trigger on a falling egde
1332
1333                CC1101_GDO0_SET_OUTPUT;          // GDO0 is an output
1334                CC1101_GDO0_SET_HIGH;            // Set GDO0 on high
1335
1336                //***Takt synchronisierung **************************************************
1337
1338                TIMER2config();                  // Timer to messure the triggerspace
1339                cc1101_reset();                  // Reset all registers to their default values and go to idle state
1340                cc1101_config_cali_clk();    // Configure the transceiver for sending the wakeup signal
1341                cc1101_rx_asynchronous_mode();   // Ab hier wird empfangen
1342                IRQ_SET_GDO2_CLOCK_CALI_START;   // IRQ setzen
1343                CC1101_GDO2_CLEAR_IRQ;           // GDO2 clear
1344                delay_ms(40);                    // Wait untill sensor is ready and TX signal
1345                timeout_start(tx_timeout);       // Start the timeout timer
1346                CC1101_GDO2_IRQ_ENABLE;          // GDO2 enable
1347
1348                while((clk_cali_counter<=clk_numb_counts+5)&&(!TIMEOUT_IS_SET));      // collect values
1349
1350                timeout_stop(); //Stop the timeout mode
1351                clk_cali_counter = 0;
1352                cc1101_reset();
1353                TIMER2_STOP;
1354
1355                CC1101_GDO0_SET_INPUT;           // GDO0
1356
1357                frequency = 0;
1358                UARTprintf("-------------------------\n");
1359                for(index = 1; index!=clk_numb_counts+1;index++) {   // Alle adc Werte zusammenrechnen
1360                    UARTprintf("%i\n",adc_value[index]);
1361                    frequency = frequency + adc_value[index];
1362                }
1363                UARTprintf("-------------------------\n");
1364
1365                frequency = frequency/clk_numb_counts;       // Durchschnitt errechnen
1366
1367                UARTprintf("%i\n",frequency);
1368                UARTprintf("-------------------------\n");
1369
1370                // Ist der durchschnitt der 20 Werte innerhalb der toleranz?
```

```
1371                    // Gemessen werden 2ms. Dies enspricht einem Wert von 240.000
1372                    // 16.000/((240.000/120*10^6)/2) = 16MHz
1373                    // Toleranz wird nun auf +- 1MHz festgelegt
1374                    // das heißt +- 1509, also 241.509 bzw. 238.491
1375
1376                    packet[0] = BROADCAST;
1377                    packet[1] = COMMAND_CLK_OK;
1378
1379                    if(frequency >= 240200) // Frequency is to low
1380                        packet[1] = COMMAND_CLK_UP;
1381                    else if(frequency <= 239800) // Frequency is to high
1382                        packet[1] = COMMAND_CLK_DOWN;
1383
1384                    packet[2] = 0;
1385                    packet[3] = 0;
1386
1387                    delay_ms(200);
1388
1389                    tx_data_length = 0;                                     // Normal Data length
1390                    tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data transceive
1391
1392
1393                }while(packet[1] != COMMAND_CLK_OK);
1394
1395
1396            } break;
1397            //**********************************************************
1398            // AD5270 test mode for the basestation.
1399            //**********************************************************
1400            case SET_AD5270_BS: {
1401                uint16_t res_value = 0;
1402                uint16_t res_value1 = 0;
1403                uart_rx_flag = 0;   // Reset the UART flag
1404
1405                // Initialize the SPI
1406                SPIinit(spiBitRate, spiBit, SSI_FRF_MOTO_MODE_1);
1407
1408                // Convert the input char values into hex values
1409                res_value = char_to_hex3(rx_char[14],rx_char[15],rx_char[16]);
1410                res_value1 = char_to_hex3(rx_char[18],rx_char[19],rx_char[20]);
1411
1412                write_AD5270_1(res_value);
1413                write_AD5270_2(res_value1);
1414
1415                UARTprintf("Setting the AD5270 on the basestation: 0x%x and 0x%x\n",res_value,res_value1);
1416
1417                // Initialize the SPI
1418                SPIinit(spiBitRate, spiBit, SSI_FRF_MOTO_MODE_0);
1419
1420            } break;
1421
1422            //**********************************************************
1423            // AD5270 test mode for the cellsensor.
1424            //**********************************************************
1425            case SET_AD5270_ZS: {
1426                uint16_t res_value = 0;
1427                uint16_t res_value1 = 0;
1428                uint8_t upper_byte = 0;
1429                uint8_t lower_byte = 0;
1430                uint8_t zs_adress = 0;
1431
1432                uart_rx_flag = 0;   // Reset the UART flag
1433
1434                // Convert the input char values into hex values
1435                zs_adress = char_to_hex(rx_char[14],rx_char[15]);
1436                res_value = char_to_hex3(rx_char[17],rx_char[18],rx_char[19]);
1437                res_value1 = char_to_hex3(rx_char[21],rx_char[22],rx_char[23]);
1438
1439                upper_byte = (res_value >> 8);
1440                lower_byte = (res_value >> 0);
1441
1442                packet[0] = zs_adress;
1443                packet[1] = COMMAND_AD5270_1;
1444                packet[2] = upper_byte;
1445                packet[3] = lower_byte;
1446
1447                tx_data_length = 0;                                     // Normal Data length
1448                tx_packet(HEADER_LENGTH + tx_data_length, packet);      // Config CC1101 for data tranceive
1449
1450                delay_ms(500);
1451
1452                upper_byte = (res_value1 >> 8);
1453                lower_byte = (res_value1 >> 0);
1454
1455                packet[0] = zs_adress;
```

```
1456                    packet[1] = COMMAND_AD5270_2;
1457                    packet[2] = upper_byte;
1458                    packet[3] = lower_byte;
1459
1460                    tx_data_length = 0;                               // Normal Data length
1461                    tx_packet(HEADER_LENGTH + tx_data_length, packet);       // Config CC1101 for data tranceive
1462
1463                    UARTprintf("Setting the AD5270 on the cellsensor 0x%x: 0x%x and 0x%x\n",zs_adress,res_value,res_value1);
1464
1465                } break;
1466
1467                //*********************************************************
1468                // Error.
1469                //*********************************************************
1470                default: {
1471                    UARTprintf("ERROR!\n");
1472                    uart_rx_flag = 0;    // Reset the UART flag
1473                } break;
1474            }
1475        }
1476    }
```

```
/*******************************************************************************
**     File name       : spi.c
**     Hardware        : ARM Cortex M4 Basestation
**     Date            : 06/06/2014
**     Last Update     : 27/07/2015
**     Author          : Nico Sassano
**     Description     : SPI function
**     State           : Final state
*******************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/fpu.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "header/uart.h"
#include "header/interrupts.h"
#include "header/CC1101.h"


void SPIinit(uint32_t spiBitRate, uint8_t spiBit, uint32_t spiConfig) {

    // The SSI3 peripheral must be enabled for use.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI3);

    // The GPIOQ peripheral must be enabled for use.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);

    // Configure the pin muxing.
    GPIOPinConfigure(GPIO_PQ0_SSI3CLK);
    //GPIOPinConfigure(GPIO_PQ1_SSI3FSS);
    GPIOPinConfigure(GPIO_PQ2_SSI3XDAT0);
    GPIOPinConfigure(GPIO_PQ3_SSI3XDAT1);


    // Configure the GPIO settings for the SSI pins.
    //       PQ2 - SSI3Tx      MOSI
    //       PQ3 - SSI3Rx      MISO
    //       PQ1 - SSI3Fss     Chip select
    //       PQ0 - SSI3CLK     CLK
    GPIOPinTypeSSI(GPIO_PORTQ_BASE, GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3);

    // 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(SSI3_BASE, SysCtlClockGet(), spiConfig, SSI_MODE_MASTER, spiBitRate, spiBit);

    // Enable the SSI3 module.
    SSIEnable(SSI3_BASE);
}

void SPI_send_data(int length, uint32_t data[10]) {
    int index = 0;

    CS_ENABLE;

    for(index = 0; index < length; index++) {
        SSIDataPut(SSI3_BASE, data[index]);

        // Wait until SSI3 is done transferring all the data in the transmit FIFO.
        while(SSIBusy(SSI3_BASE));
    }
    CS_DISABLE;
}
```

```
/*******************************************************************************
**     File name      : timer.c
**     Hardware       : ARM Cortex M4 Basestation
**     Date           : 16/05/2014
**     Last Update    : 27/07/2015
**     Author         : Nico Sassano
**     Description    : Timer function
**     State          : Final state
*******************************************************************************/
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/timer.h"
#include "utils/uartstdio.h"
#include "header/timer.h"
#include "header/cc1101.h"

//*****************************************************************************
// System clock rate in Hz.
//*****************************************************************************
extern uint32_t g_ui32SysClock;
extern volatile uint8_t irq_delay_flag;

extern volatile uint8_t irq_timer0_mode;
extern volatile uint8_t irq_timer1_mode;     // Flag for the timer1 mode
extern volatile uint8_t irq_timeout;         // Flag for the time out timer

extern volatile uint32_t burst_config;

void TIMER0init(){
    // Enable the peripherals used by this example.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

    UARTprintf("TIMER0 initialization done...\n");
}

void TIMER1init(){
    // Enable the peripherals used by this example.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

    UARTprintf("TIMER1 initialization done...\n");
}

void TIMER2init(){
    // Enable the peripherals used by this example.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);

    UARTprintf("TIMER2 initialization done...\n");
}
//*****************************************************************************
// Function to config the Timer0
// Timer0 is the counter for the delay function
// Timer is in ms
//*****************************************************************************
void TIMER0config_ms(uint32_t config){
    // Configure the two 32-bit one shot timers.
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_ONE_SHOT);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, (g_ui32SysClock/1000)*config );

    // Setup the interrupts for the timer timeouts.
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Enable the timer 0.
    TIMER0_START;
}

//*****************************************************************************
// Function to config the Timer0
// Timer0 is the counter for the delay function
// Timer is in us
//*****************************************************************************
void TIMER0config_us(uint32_t config){
    // Configure the two 32-bit one shot timers.
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_ONE_SHOT);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, (g_ui32SysClock/1000000)*config );
```

```
86
87      // Setup the interrupts for the timer timeouts.
88      ROM_IntEnable(INT_TIMER0A);
89      ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
90
91      // Enable the timer 0.
92      TIMER0_START;
93  }
94
95  //*****************************************************************************
96  // Function to config the Timer1
97  // Timer1 is for the burst function
98  // Time is in us
99  //*****************************************************************************
100 void TIMER1config(uint32_t config) {
101
102     burst_config = config;
103     TIMER1_MODE_SET_BURST;  // Set flag to burst mode
104
105     // Configure the two 32-bit periodic timers.
106     ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
107     ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, (g_ui32SysClock/1000000)*10000);
108
109     // Setup the interrupt
110     ROM_IntEnable(INT_TIMER1A);
111     ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
112
113     // Enable the timer 1.
114     TIMER1_START;
115 }
116
117 //*****************************************************************************
118 // Function to config the Timer1
119 // Timer1 is for the LTC2376 test sampling mode
120 // Time is in us
121 //*****************************************************************************
122 void TIMER1config_LTC2376(uint32_t config) {
123
124     TIMER1_MODE_SET_LTC2376;    // Set flag to LTC2376 sampling mode
125
126     // Configure the two 32-bit periodic timers.
127     ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
128     ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, (g_ui32SysClock/1000000)*config);
129
130     // Setup the interrupt
131     ROM_IntEnable(INT_TIMER1A);
132     ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
133
134     // Enable the timer 1.
135     TIMER1_START;
136 }
137
138 //*****************************************************************************
139 // Function to config the Timer1
140 // Timer1 is for the AD7691 test sampling mode
141 // Time is in us
142 //*****************************************************************************
143 void TIMER1config_AD7691(uint32_t config) {
144
145     TIMER1_MODE_SET_AD7691; // Set flag to AD7691 sampling mode
146
147     // Configure the two 32-bit periodic timers.
148     ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
149     ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, (g_ui32SysClock/1000000)*config);
150
151     // Setup the interrupt
152     ROM_IntEnable(INT_TIMER1A);
153     ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
154
155     // Enable the timer 1.
156     TIMER1_START;
157 }
158
159 //*****************************************************************************
160 // Function to config the Timer1
161 // Timer1 is for the AD7176 test sampling mode
162 // Time is in us
163 //*****************************************************************************
164 void TIMER1config_AD7176(uint32_t config) {
165
166     TIMER1_MODE_SET_AD7176; // Set flag to AD7176 sampling mode
167
168     // Configure the two 32-bit periodic timers.
169     ROM_TimerConfigure(TIMER1_BASE, TIMER_CFG_PERIODIC);
170     ROM_TimerLoadSet(TIMER1_BASE, TIMER_A, (g_ui32SysClock/1000000)*config);
```

```
171
172     // Setup the interrupt
173     ROM_IntEnable(INT_TIMER1A);
174     ROM_TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
175
176     // Enable the timer 1.
177     TIMER1_START;
178 }
179
180 //*****************************************************************************
181 // Function to config the Timer2
182 // Timer to messure the triggerspace of the DCO calibration
183 //*****************************************************************************
184 void TIMER2config() {
185
186     // Configure the two 32-bit periodic timers.
187     ROM_TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);
188     ROM_TimerLoadSet(TIMER2_BASE, TIMER_A, g_ui32SysClock);
189
190     //ROM_TimerLoadSet(TIMER2_BASE, TIMER_A, g_ui32SysClock);
191 }
192
193 // Delay function in ms
194 void delay_ms(uint32_t ms){
195     TIMER_MODE_SET_DELAY;
196     irq_delay_flag = 0;
197     TIMER0config_ms(ms);
198
199     while(!irq_delay_flag);
200 }
201
202 // Delay function in us
203 void delay_us(uint32_t us){
204     TIMER_MODE_SET_DELAY;
205     irq_delay_flag = 0;
206     TIMER0config_us(us);
207
208     while(!irq_delay_flag);
209 }
210
211 // Timeout function
212 void timeout_start(uint32_t ms){
213     TIMER_MODE_SET_TIMEOUT;
214     TIMEOUT_UNSET;
215     TIMER0config_ms(ms);
216
217     //while(!TIMEOUT_IS_SET);
218 }
219
220 void timeout_stop() {
221     TIMER_MODE_SET_DELAY;
222
223     // Disable the timers.
224     ROM_TimerDisable(TIMER0_BASE, TIMER_A);
225
226 }
```

```
1  /***********************************************************************************
2  **      File name       :  uart.c
3  **      Hardware        :  ARM Cortex M4 Basestation
4  **      Date            :  14/09/2014
5  **      Last Update     :  27/07/2015
6  **      Author          :  Nico Sassano
7  **      Description     :  UART function
8  **      State           :  Final state
9  **********************************************************************************/
10 #include <stdint.h>
11 #include <stdbool.h>
12 #include "inc/hw_ints.h"
13 #include "inc/hw_memmap.h"
14 #include "driverlib/debug.h"
15 #include "driverlib/gpio.h"
16 #include "driverlib/interrupt.h"
17 #include "driverlib/pin_map.h"
18 #include "driverlib/rom.h"
19 #include "driverlib/rom_map.h"
20 #include "driverlib/sysctl.h"
21 #include "driverlib/uart.h"
22 #include "utils/uartstdio.h"
23 #include "header/uart.h"
24
25 //*****************************************************************************
26 // Declared extern variable
27 //*****************************************************************************
28 extern uint32_t g_ui32SysClock;
29
30 //*****************************************************************************
31 // Send a string to the UART.
32 //*****************************************************************************
33 void
34 UARTSend(const uint8_t *pui8Buffer, uint32_t ui32Count)
35 {
36     GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 1);
37
38     // Loop while there are more characters to send
39     while(ui32Count--)
40     {
41         // Write the next character to the UART
42         ROM_UARTCharPutNonBlocking(UART0_BASE, *pui8Buffer++);
43
44         // Delay for 1 millisecond, necessary because UART Baud rate is to slow
45         SysCtlDelay(g_ui32SysClock / (1000 * 3));
46
47     }
48     GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0);
49 }
50
51 void UARTinit(uint32_t port_num, uint32_t data_rate, uint32_t clock_rate){
52
53     // Enable the GPIO Peripheral used by the UART
54     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
55
56     // Enable UART0
57     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
58
59     // Configure GPIO Pins for UART mode
60     ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
61     ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
62     ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
63
64     // Enable the UART interrupt
65     ROM_IntEnable(INT_UART0);
66     ROM_UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);
67
68     UARTFIFODisable(UART0_BASE);
69
70     // Initialize the UART
71     UARTStdioConfig(port_num, data_rate, clock_rate);
72
73     UARTprintf("System initialization done...\n");
74     UARTprintf("UART initialization done...\n");
75
76 }
```

```
1   /**********************************************************************************
2   **     File name        : uart_check.c
3   **     Hardware         : ARM Cortex M4 Basestation
4   **     Date             : 15/09/2014
5   **     Last Update      : 27/07/2015
6   **     Author           : Nico Sassano
7   **     Description      : UART check function
8   **     State            : Final state
9   **********************************************************************************/
10  #include <stdint.h>
11  #include <stdbool.h>
12  #include "inc/hw_ints.h"
13  #include "inc/hw_memmap.h"
14  #include "inc/hw_types.h"
15  #include "driverlib/debug.h"
16  #include "driverlib/fpu.h"
17  #include "driverlib/ssi.h"
18  #include "driverlib/gpio.h"
19  #include "driverlib/interrupt.h"
20  #include "driverlib/pin_map.h"
21  #include "driverlib/rom.h"
22  #include "driverlib/rom_map.h"
23  #include "driverlib/sysctl.h"
24  #include "driverlib/timer.h"
25  #include "driverlib/uart.h"
26  #include "utils/uartstdio.h"
27  #include "header/main.h"
28  #include "header/uart.h"
29  #include "header/spi.h"
30  #include "header/timer.h"
31  #include "header/interrupts.h"
32  #include "header/CC1101.h"
33  #include "header/gpio.h"
34  #include "header/convert.h"
35  #include "header/uart_check.h"
36
37  extern volatile uint8_t rx_uart;
38  extern volatile uint8_t uart_rx_flag;
39  extern volatile unsigned char rx_char[30];
40
41  uint8_t check_input() {
42      uint8_t check = 0;
43      switch(rx_uart) {
44                  //*****************************************************************
45                  // "help" Open the help menue
46                  //*****************************************************************
47                  case HELP: {
48                      if ((rx_char[0] == 0x68) && (rx_char[1] == 0x65) && (rx_char[2] == 0x6c) && (rx_char[3] == 0x70))
49                          check = 1;
50
51                  } break;
52                  //*****************************************************************
53                  case WHY: {
54
55                  } break;
56                  //*****************************************************************
57                  case GET_VOLTAGE: {
58                      if ((rx_char[0]  == 0x67) && (rx_char[1]  == 0x65) && (rx_char[2]  == 0x74) &&
59                          (rx_char[3]  == 0x5F) && (rx_char[4]  == 0x76) && (rx_char[5]  == 0x6F) &&
60                          (rx_char[6]  == 0x6C) && (rx_char[7]  == 0x74) && (rx_char[8]  == 0x61) &&
61                          (rx_char[9]  == 0x67) && (rx_char[10] == 0x65) && (rx_char[11] == 0x28) &&
62                          (rx_char[14] == 0x29)) {
63                          check = 1;
64                      } else {
65                          UARTprintf("Unknow command!\n");
66                          UARTprintf("For getting an voltage of an sensor, please type for example: get_voltage(02) for the
                                voltage of sensor 0x02\n");
67                      }
68
69                  } break;
70                  //******************************************************************
71                  // "send_hex" sends four hex values with the cc1101
72                  // Example: send_hex(00 04 00 00) -> sends the hex values "0x00 0x04 0x00 0x00"
73                  //******************************************************************
74                  case SEND_HEX: {
75                      if ((rx_char[0]  == 0x73) && (rx_char[1]  == 0x65) && (rx_char[2]  == 0x6E) &&
76                          (rx_char[3]  == 0x64) && (rx_char[4]  == 0x5F) && (rx_char[5]  == 0x68) &&
77                          (rx_char[6]  == 0x65) && (rx_char[7]  == 0x78) && (rx_char[8]  == 0x28) &&
78                          (rx_char[11] == 0x20) && (rx_char[14] == 0x20) && (rx_char[17] == 0x20) &&
79                          (rx_char[20] == 0x29)) {
80                          check = 1;
81                      } else {
82                          UARTprintf("Unknow command!\n");
83                          UARTprintf("For sending hex values to a sensor, please type the follow command to send 0xAB 0x02
                              0x32 0x45\n");
```

```
84                              UARTprintf("send_hex(AB 02 32 45)\n");
85                          }
86
87                      } break;
88                      //*******************************************************************
89                      // "get_conig" mode. Returns the actual config of the basestation
90                      //*******************************************************************
91                      case GET_CONFIG_BS: {
92                          if ((rx_char[0] == 0x67) && (rx_char[1] == 0x65)  && (rx_char[2] == 0x74) &&
93                              (rx_char[3] == 0x5F) && (rx_char[4] == 0x63)  && (rx_char[5] == 0x6F) &&
94                              (rx_char[6] == 0x6E) && (rx_char[7] == 0x66)  && (rx_char[8] == 0x69) &&
95                              (rx_char[9] == 0x67)) {
96                              check = 1;
97                          } else {
98                              UARTprintf("Unknow command!\n");
99                              UARTprintf("For getting the configuration of the basestation, please type:\n");
100                             UARTprintf("get_config(00)\n");
101                         }
102                     } break;
103                     //*******************************************************************
104                     // "set_brate" mode.
105                     //*******************************************************************
106                     case SET_BRATE: {
107
108                     } break;
109                     //*******************************************************************
110                     // Burstmode. Detect fhe frequency and the number of values and start the burst mode
111                     //*******************************************************************
112                     case BURST: {
113
114
115                     } break;
116
117                     default: {
118                         UARTprintf("ERROR!\n");
119                         uart_rx_flag = 0;
120                     } break;
121
122                 }
123     return check;
124 }
```

```
1  /************************************************************************************
2  **      File name       : AD5270.h
3  **      Hardware        : ARM Cortex M4 Basestation
4  **      Date            : 11/02/2015
5  **      Last Update     : 17/07/2015
6  **      Author          : Nico Sassano
7  **      Description     : AD5270
8  **      State           : Final state
9  ************************************************************************************/
10 #ifndef AD5270_H_
11 #define AD5270_H_
12
13 #define AD5270_NOP_CMD              0x0000
14 #define AD5270_WRITE_CMD            0x0400
15 #define AD5270_READ_CMD             0x0800
16 #define AD5270_STORE_CMD            0x0C00
17 #define AD5270_SW_RST_CMD           0x1000
18 #define AD5270_SW_SHD_ON_CMD        0x2401
19
20 // Definition for the SYNC Output
21 #define AD5270_CS1_OUTPUT           ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);\
22                                     ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1)
23
24 #define AD5270_CS2_OUTPUT           ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);\
25                                     ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_2)
26
27 #define AD5270_CS1_ENABLE           GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0x00)
28 #define AD5270_CS1_DISABLE          GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0xFF)
29
30 #define AD5270_CS2_ENABLE           GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_2, 0x00)
31 #define AD5270_CS2_DISABLE          GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_2, 0xFF)
32
33
34 // Declarate the prototype functions
35 void write_AD5270_1(uint16_t);
36 void write_AD5270_2(uint16_t);
37 void ctr_voltage_AD5270(uint16_t);
38
39 #endif /* AD5270_H_ */
```

```
1  /************************************************************************************
2  **      File name       : AD7176.h
3  **      Hardware        : ARM Cortex M4 Basestation
4  **      Date            : 14/01/2015
5  **      Last Update     : 02/08/2015
6  **      Author          : Nico Sassano
7  **      Description     : AD7176
8  **      State           : Final state
9  ************************************************************************************/
10 #ifndef AD7176_H_
11 #define AD7176_H_
12
13 #define COMMS                   0x00
14 #define STATUS                  0x00
15 #define ADCMODE                 0x01
16 #define IFMODE                  0x02
17 #define REGCHECK                0x03
18 #define DATA                    0x04
19 #define GPIOCON                 0x06
20 #define ID                      0x07
21 #define CHMAP0                  0x10
22 #define CHMAP1                  0x11
23 #define CHMAP2                  0x12
24 #define CHMAP3                  0x13
25 #define SETUPCON0               0x20
26 #define SETUPCON1               0x21
27 #define SETUPCON2               0x22
28 #define SETUPCON3               0x23
29 #define FILTCON0                0x28
30 #define FILTCON1                0x29
31 #define FILTCON2                0x2A
32 #define FILTCON3                0x2B
33 #define OFFSET0                 0x30
34 #define OFFSET1                 0x31
35 #define OFFSET2                 0x32
36 #define OFFSET3                 0x33
37 #define GAIN0                   0x38
38 #define GAIN1                   0x39
39 #define GAIN2                   0x3A
40 #define GAIN3                   0x3B
41
42 // Definitions for the ouputs
43 #define AD7176_CS_SET_OUTPUT        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);\
44                                     ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1)
45
```

```
46  #define AD7176_CS_ENABLE          GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0x00)
47  #define AD7176_CS_DISABLE         GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0xFF)
48
49  // Definitions for the inputs
50  #define AD7176_BUSY_SET_INPUT     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
51                                    ROM_GPIOPinTypeGPIOInput(GPIO_PORTK_BASE, GPIO_PIN_7)
52
53  #define AD7176_BUSY_SET_OUTPUT    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
54                                    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_7)
55
56
57  // Definitions for the AD7176 IRQ setup
58  #define AD7176_IRQ_ENABLE         GPIOIntEnable(GPIO_PORTK_BASE,GPIO_INT_PIN_7);\
59                                    IntEnable(INT_GPIOK)
60
61  #define AD7176_IRQ_DISABLE        GPIOIntDisable(GPIO_PORTK_BASE, GPIO_PIN_7);\
62                                    IntDisable(INT_GPIOK)
63
64  #define AD7176_IRQ_FALLING_EDGE GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_FALLING_EDGE);
65  #define AD7176_IRQ_RISING_EDGE  GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_RISING_EDGE);
66
67  #define AD7176_CLEAR_IRQ          GPIOIntClear(GPIO_PORTK_BASE,GPIO_PIN_7)
68
69  void init_AD7176(void);
70  void AD7176_start(uint8_t, uint8_t);
71
72  #endif /* AD7176_H_ */
```

```
1   /**********************************************************************************
2   **      File name       : AD7691.h
3   **      Hardware        : ARM Cortex M4 Basestation
4   **      Date            : 22/12/2015
5   **      Last Update     : 02/08/2015
6   **      Author          : Nico Sassano
7   **      Description     : AD7691
8   **      State           : Final state
9   **********************************************************************************/
10  #ifndef AD7691_H_
11  #define AD7691_H_
12
13
14  // Definitions for the ouputs
15  #define AD7691_CS_ENABLE          GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0x00)
16  #define AD7691_CS_DISABLE         GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0xFF)
17
18  #define AD7691_CNV_SET_HIGH GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_6, 0xFF)
19  #define AD7691_CNV_SET_LOW        GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_6, 0x00)
20
21
22  // Definitions for the inputs
23  #define AD7691_BUSY_SET_INPUT     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
24                                    ROM_GPIOPinTypeGPIOInput(GPIO_PORTK_BASE, GPIO_PIN_7)
25
26  #define AD7691_BUSY_SET_OUTPUT    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
27                                    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_7)
28
29
30  // Definitions for the LTC2376 IRQ setup
31  #define AD7691_IRQ_ENABLE         GPIOIntEnable(GPIO_PORTK_BASE,GPIO_INT_PIN_7);\
32                                    IntEnable(INT_GPIOK)
33
34  #define AD7691_IRQ_DISABLE        GPIOIntDisable(GPIO_PORTK_BASE, GPIO_PIN_7);\
35                                    IntDisable(INT_GPIOK)
36
37  #define AD7691_IRQ_FALLING_EDGE GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_FALLING_EDGE);
38  #define AD7691_IRQ_RISING_EDGE GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_RISING_EDGE);
39
40  #define AD7691_CLEAR_IRQ      GPIOIntClear(GPIO_PORTK_BASE,GPIO_PIN_7)
41
42  // Declarate the prototype functions
43  void init_ad7691(void);
44  void ad7691_start(uint8_t, uint8_t);
45
46  #endif /* TLC2376_H_ */
```

```
1   /**********************************************************************************
2   **      File name       : adc.h
3   **      Hardware        : ARM Cortex M4 Basestation
4   **      Date            : 14/09/2015
5   **      Last Update     : 02/08/2015
6   **      Author          : Nico Sassano
7   **      Description     : ADC12
8   **      State           : Final state
9   **********************************************************************************/
10  #ifndef ADC_H_
```

```
11  #define ADC_H_
12
13  void init_adc(void);
14  uint32_t read_adc(void);
15
16  #endif /* ADC_H_ */
```

```
1   /********************************************************************************
2   **    File name      : CC1101.h
3   **    Hardware       : ARM Cortex M4 Basestation
4   **    Date           : 14/09/2015
5   **    Last Update    : 28/10/2015
6   **    Author         : Nico Sassano
7   **    Description    : CC1101
8   **    State          : Final state
9   ********************************************************************************/
10  #ifndef CC1101_H_
11  #define CC1101_H_
12
13
14  void cc1101_power_up_reset(void);
15  void tx_packet(uint8_t , uint8_t *);
16  void rx_packet(uint8_t);
17  void cc1101_tx_asynchronous_mode(void);
18  void cc1101_rx_asynchronous_mode(void);
19  uint8_t cc1101_get_rxbytes(void);
20  void cc1101_read_rx_fifo(uint8_t *, uint8_t);
21  void cc1101_reset(void);
22  void cc1101_config_burst_tx(void);
23  void cc1101_config_burst_rx(void);
24  void cc1101_config_burst_cali(void);
25  void cc1101_config_cali_clk(void);
26  void cc1101_config_cali_burst(void);
27  void burst_start(uint8_t, uint8_t);
28  void cc1101_idle(void);
29  void cc1101_set_tx(unsigned long brate);
30
31  extern volatile int GDO2_IRQ_MODE;
32  extern volatile char packed[6];
33  extern int state;
34  extern volatile int IRQ_TX_FLAG;
35  extern volatile int IRQ_RX_FLAG;
36  extern volatile uint8_t irq_timeout;        // Flag for the time out timer
37
38  // CRC check done
39  #define CC1101_GDO0_DIR_IN              (CC1101_GDO0_PxDIR &= ~CC1101_GDO0_PIN)
40  #define CC1101_GDO0_IN                  GPIOPinRead(GPIO_PORTH_BASE, GPIO_PIN_0)
41
42  // Sync word detected
43  #define CC1101_GDO2_IRQ_ENABLE          GPIOIntEnable(GPIO_PORTK_BASE,GPIO_INT_PIN_6);\
44                                          IntEnable(INT_GPIOK)
45
46  #define CC1101_GDO2_IRQ_DISABLE         GPIOIntDisable(GPIO_PORTK_BASE, GPIO_PIN_6);\
47                                          IntDisable(INT_GPIOK)
48
49  #define CC1101_GDO2_IRQ_FALLING_EDGE    GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_6, GPIO_FALLING_EDGE);
50  #define CC1101_GDO2_IRQ_RISING_EDGE     GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_6, GPIO_RISING_EDGE);
51
52  #define CC1101_GDO2_CLEAR_IRQ           GPIOIntClear(GPIO_PORTK_BASE,GPIO_PIN_6)
53
54  #define HEADER_LENGTH                                   0x04
55  #define ADDRESS_THIS_SENSOR                             0x00
56
57  /*****************************************************
58  ** IRQ Macros
59  *****************************************************/
60  #define IRQ_SET_RX                      (irq_mode = RX_MODE)
61  #define IRQ_IS_RX                       (irq_mode == RX_MODE)
62  #define IRQ_SET_TX                      (irq_mode = TX_MODE)
63  #define IRQ_IS_TX                       (irq_mode == TX_MODE)
64  #define IRQ_SET_BURST                   (irq_mode = BURST_MODE)
65  #define IRQ_IS_BURST                    (irq_mode == BURST_MODE)
66  #define IRQ_SET_CALI_BURST              (irq_mode = CALI_BURST_MODE)
67  #define IRQ_IS_CALI_BURST               (irq_mode == CALI_BURST_MODE)
68  #define IRQ_SET_GDO2_READY              (irq_mode = GDO2_READY)
69  #define IRQ_IS_GDO2_READY               (irq_mode == GDO2_READY)
70  #define IRQ_SET_GDO2_CLOCK_CALI_START   (irq_mode = CLOCK_CALI_START)
71  #define IRQ_IS_GDO2_CLOCK_CALI_START    (irq_mode == CLOCK_CALI_START)
72  #define IRQ_SET_GDO2_CLOCK_CALI_END     (irq_mode = CLOCK_CALI_END)
73  #define IRQ_IS_GDO2_CLOCK_CALI_END      (irq_mode == CLOCK_CALI_END)
74
75  /*****************************************************
76  ** Interrupt Modes
77  *****************************************************/
78  #define TX_MODE                                         0x00
```

```
79   #define RX_MODE                                                 0x01
80   #define BURST_MODE                                              0x02
81   #define CALI_BURST_MODE                                         0x03
82   #define GDO2_READY                                              0x04
83   #define CLOCK_CALI_START                                        0x05
84   #define CLOCK_CALI_END                                          0x06
85
86   /*** Downlink commands ***********************************/
87   #define COMMAND_WAKEUP                                          0x00
88   #define COMMAND_WAKEUP_DONE                                     0x01
89   #define COMMAND_DOWNLINK_IS_AWAKE                               0x02
90   #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE                         0x03
91   #define COMMAND_DOWNLINK_SEND_VOLTAGE                           0x04
92   #define COMMAND_DOWNLINK_SLEEP                                  0x05
93   #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE                     0x06
94   #define COMMAND_DOWNLINK_SEND_TEMPERATURE                       0x07
95   #define COMMAND_DOWNLINK_BALANCING_ON                           0x08
96   #define COMMAND_DOWNLINK_BALANCING_OFF                          0x09
97   #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE               0x0A
98   #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE             0x0B
99   #define COMMAND_DOWNLINK_BURST_MODE                             0x0C
100  #define COMMAND_DOWNLINK_BURST_DATA_RQ                          0x0D
101  #define COMMAND_DOWNLINK_BURST_DATA_RX                          0x0E
102  #define COMMAND_DOWNLINK_BURST_CHECK                            0x0F
103  #define COMMAND_DOWNLINK_CONFIG_SET                             0x10
104  #define COMMAND_DOWNLINK_CONFIG                                 0x11
105  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102              0x12
106  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102                0x13
107  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102_CALI         0x14
108  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI           0x15
109  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430              0x16
110  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430                0x17
111  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430_CALI         0x18
112  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI           0x19
113  #define COMMAND_DOWNLINK_CALIBRATION_TMP102                     0x1A
114  #define COMMAND_DOWNLINK_CALIBRATION_MSP430                     0x1B
115  #define COMMAND_DOWNLINK_CALIBRATION_ADC                        0x1C
116  #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE_ALL           0x1D
117  #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE_ALL         0x1E
118  #define COMMAND_CALI_BURST                                      0x1F
119  #define COMMAND_CLK_OK                                          0x21
120  #define COMMAND_CLK_UP                                          0x20
121  #define COMMAND_CLK_DOWN                                        0x22
122  #define COMMAND_CALI_CLK                                        0x23
123  #define COMMAND_SW                                              0x24
124  #define COMMAND_GOERTZEL                                        0x25
125  #define COMMAND_AD5270_1                                        0x26
126  #define COMMAND_AD5270_2                                        0x27
127  #define COMMAND_SET_PREPROCESSING                               0x28
128  #define COMMAND_GOERTZEL_FREQ                                   0x29
129  #define COMMAND_GOERTZEL_STIMULI_FREQ                           0x2A
130  #define COMMAND_GOERTZEL_STIMULI_FREQ2                          0x2B
131  #define COMMAND_GOERTZEL_NUMB_PERI                              0x2C
132
133  #define COMMAND_BACK_FROM_BURST                                 0xFD
134  #define COMMAND_WAIT                                            0xFE
135  #define COMMAND_DOWNLINK_UNKWOWN                                0xFF
136
137  // Frequency offset added to the base frequency (in units of 1.59 kHz - 1.65 kHz)
138  #define FREQUENCY_OFFSET                    6
139
140  /* Command Strobes (Table 42 in datasheet) */
141  #define cc1101_config_packet                     0x30
142  #define CC1101_CS_SFSTXON               0x31
143  #define CC1101_CS_SXOFF                 0x32
144  #define CC1101_CS_SCAL                  0x33
145  #define CC1101_CS_SRX                   0x34
146  #define CC1101_CS_STX                   0x35
147  #define CC1101_CS_SIDLE                 0x36
148  #define CC1101_CS_SAFC                  0x37
149  #define CC1101_CS_SWOR                  0x38
150  #define CC1101_CS_SPWD                  0x39
151  #define CC1101_CS_SFRX                  0x3A
152  #define CC1101_CS_SFTX                  0x3B
153  #define CC1101_CS_SWORRST               0x3C
154  #define CC1101_CS_SNOP                  0x3D
155
156  /* Configuration Registers */
157  #define CC1101_CR_IOCFG2                0x00     /* GDO2 output pin configuration */
158  #define CC1101_CR_IOCFG1                0x01     /* GDO1 output pin configuration */
159  #define CC1101_CR_IOCFG0                0x02     /* GDO0 output pin configuration */
160  #define CC1101_CR_FIFOTHR               0x03     /* RX FIFO and TX FIFO thresholds */
161  #define CC1101_CR_SYNC1                 0x04     /* Sync word, high byte */
162  #define CC1101_CR_SYNC0                 0x05     /* Sync word, low byte */
163  #define CC1101_CR_PKTLEN                0x06     /* Packet length */
```

```
164 #define CC1101_CR_PKTCTRL1           0x07     /* Packet automation control */
165 #define CC1101_CR_PKTCTRL0           0x08     /* Packet automation control */
166 #define CC1101_CR_ADDR               0x09     /* Device address */
167 #define CC1101_CR_CHANNR             0x0A     /* Channel number */
168 #define CC1101_CR_FSCTRL1            0x0B     /* Frequency synthesizer control */
169 #define CC1101_CR_FSCTRL0            0x0C     /* Frequency synthesizer control */
170 #define CC1101_CR_FREQ2              0x0D     /* Frequency control word, high byte */
171 #define CC1101_CR_FREQ1              0x0E     /* Frequency control word, middle byte */
172 #define CC1101_CR_FREQ0              0x0F     /* Frequency control word, low byte */
173 #define CC1101_CR_MDMCFG4            0x10     /* Modem configuration */
174 #define CC1101_CR_MDMCFG3            0x11     /* Modem configuration */
175 #define CC1101_CR_MDMCFG2            0x12     /* Modem configuration */
176 #define CC1101_CR_MDMCFG1            0x13     /* Modem configuration */
177 #define CC1101_CR_MDMCFG0            0x14     /* Modem configuration */
178 #define CC1101_CR_DEVIATN            0x15     /* Modem deviation setting */
179 #define CC1101_CR_MCSM2              0x16     /* Main Radio Cntrl State Machine config */
180 #define CC1101_CR_MCSM1              0x17     /* Main Radio Cntrl State Machine config */
181 #define CC1101_CR_MCSM0              0x18     /* Main Radio Cntrl State Machine config */
182 #define CC1101_CR_FOCCFG             0x19     /* Frequency Offset Compensation config */
183 #define CC1101_CR_BSCFG              0x1A     /* Bit Synchronization configuration */
184 #define CC1101_CR_AGCCTRL2           0x1B     /* AGC control */
185 #define CC1101_CR_AGCCTRL1           0x1C     /* AGC control */
186 #define CC1101_CR_AGCCTRL0           0x1D     /* AGC control */
187 #define CC1101_CR_WOREVT1            0x1E     /* High byte Event 0 timeout */
188 #define CC1101_CR_WOREVT0            0x1F     /* Low byte Event 0 timeout */
189 #define CC1101_CR_WORCTRL            0x20     /* Wake On Radio control */
190 #define CC1101_CR_FREND1             0x21     /* Front end RX configuration */
191 #define CC1101_CR_FREND0             0x22     /* Front end TX configuration */
192 #define CC1101_CR_FSCAL3             0x23     /* Frequency synthesizer calibration */
193 #define CC1101_CR_FSCAL2             0x24     /* Frequency synthesizer calibration */
194 #define CC1101_CR_FSCAL1             0x25     /* Frequency synthesizer calibration */
195 #define CC1101_CR_FSCAL0             0x26     /* Frequency synthesizer calibration */
196 #define CC1101_CR_RCCTRL1            0x27     /* RC oscillator configuration */
197 #define CC1101_CR_RCCTRL0            0x28     /* RC oscillator configuration */
198 #define CC1101_CR_FSTEST             0x29     /* Frequency synthesizer cal control */
199 #define CC1101_CR_PTEST              0x2A     /* Production test */
200 #define CC1101_CR_AGCTEST            0x2B     /* AGC test */
201 #define CC1101_CR_TEST2              0x2C     /* Various test settings */
202 #define CC1101_CR_TEST1              0x2D     /* Various test settings */
203 #define CC1101_CR_TEST0              0x2E     /* Various test settings */
204
205 /* Status Registers */
206
207 #define CC1101_SR_PARTNUM            0x30     /* Part number */
208 #define CC1101_SR_VERSION            0x31     /* Current version number */
209 #define CC1101_SR_FREQEST            0x32     /* Frequency offset estimate */
210 #define CC1101_SR_LQI                0x33     /* Demodulator estimate for link quality */
211 #define CC1101_SR_RSSI               0x34     /* Received signal strength indication */
212 #define CC1101_SR_MARCSTATE          0x35     /* Control state machine state */
213 #define CC1101_SR_WORTIME1           0x36     /* High byte of WOR timer */
214 #define CC1101_SR_WORTIME0           0x37     /* Low byte of WOR timer */
215 #define CC1101_SR_PKTSTATUS          0x38     /* Current GDOx status and packet status */
216 #define CC1101_SR_VCO_VC_DAC         0x39     /* Current setting from PLL cal module */
217 #define CC1101_SR_TXBYTES            0x3A     /* Underflow and # of bytes in TXFIFO */
218 #define CC1101_SR_RXBYTES            0x3B     /* Overflow and # of bytes in RXFIFO */
219 #define CC1101_SR_RCCTRL1_STATUS     0x3C     /* Last RC oscilator calibration results */
220 #define CC1101_SR_RCCTRL0_STATUS     0x3D     /* Last RC oscilator calibration results */
221
222 /* Single / Burst access */
223
224 #define CC1101_WRITE_BURST           0x40
225 #define CC1101_READ_SINGLE           0x80
226 #define CC1101_READ_BURST            0xC0
227
228 /* Memory locations */
229
230 #define CC1101_ML_PATABLE            0x3E
231 #define CC1101_ML_TXFIFO             0x3F
232 #define CC1101_ML_TXFIFO_BURST       0x7F
233 #define CC1101_ML_RXFIFO             0x3F
234
235 #define BROADCAST                    0x00
236
237 // Definitions for the CC1101 Tranceiver Board V0.4
238 #define TX_LED_ON          GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_1, 0x00)
239 #define TX_LED_OFF         GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_1, 0xFF)
240
241 #define RX_LED_ON          GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, 0x00)
242 #define RX_LED_OFF         GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_4, 0xFF)
243
244 #define CS_ENABLE          GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_1, 0x00)
245 #define CS_DISABLE         GPIOPinWrite(GPIO_PORTQ_BASE, GPIO_PIN_1, 0xFF)
246
247 #define CC1101_GDO0_SET_INPUT     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);\
248                                   ROM_GPIOPinTypeGPIOInput(GPIO_PORTH_BASE, GPIO_PIN_0)
```

```
249
250 #define CC1101_GDO0_SET_OUTPUT   ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);\
251                                  ROM_GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_0)
252
253 #define CC1101_GDO2_SET_INPUT    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
254                                  ROM_GPIOPinTypeGPIOInput(GPIO_PORTK_BASE, GPIO_PIN_6)
255
256 #define CC1101_GDO2_SET_OUTPUT   ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
257                                  ROM_GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_6)
258
259 #define CC1101_GDO0_IS_HIGH      GPIOPinRead(GPIO_PORTH_BASE, GPIO_PIN_0) == 1
260 #define CC1101_GDO0_IS_LOW       GPIOPinRead(GPIO_PORTH_BASE, GPIO_PIN_0) == 0
261
262 #define CC1101_GDO0_SET_HIGH     GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_0, 0xFF)
263 #define CC1101_GDO0_SET_LOW      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_0, 0x00)
264
265 #define COMMAND_REVIVED_SET          command_recived_flag = 1
266 #define COMMAND_REVIVED_UNSET        command_recived_flag = 0
267 #define COMMAND_REVIVED_IS_SET       command_recived_flag == 1
268 #define COMMAND_REVIVED_IS_UNSET     command_recived_flag == 0
269
270 #define TX_DONE_SET                  irq_send_done_flag = 1
271 #define TX_DONE_UNSET                irq_send_done_flag = 0
272 #define TX_DONE_IS_SET               irq_send_done_flag == 1
273 #define TX_DONE_IS_UNSET             irq_send_done_flag == 0
274
275 #define RX_CALI_SET                  irq_cali_flag = 1
276 #define RX_CALI_UNSET                irq_cali_flag = 0
277 #define RX_CALI_IS_SET               irq_cali_flag == 1
278 #define RX_CALI_IS_UNSET             irq_cali_flag == 0
279
280 /*********************************************************
281 ** Frequenzen für die Burstmessung
282 *********************************************************/
283 #define BURST_FREQ_100000HZ                        0x34
284 #define BURST_FREQ_95000HZ                         0x33
285 #define BURST_FREQ_90000HZ                         0x32
286 #define BURST_FREQ_85000HZ                         0x31
287 #define BURST_FREQ_80000HZ                         0x30
288 #define BURST_FREQ_75000HZ                         0x2F
289 #define BURST_FREQ_70000HZ                         0x2E
290 #define BURST_FREQ_65000HZ                         0x2D
291 #define BURST_FREQ_60000HZ                         0x2C
292 #define BURST_FREQ_55000HZ                         0x2B
293 #define BURST_FREQ_50000HZ                         0x2A
294 #define BURST_FREQ_45000HZ                         0x29
295 #define BURST_FREQ_40000HZ                         0x28
296 #define BURST_FREQ_38000HZ                         0x27
297 #define BURST_FREQ_36000HZ                         0x26
298 #define BURST_FREQ_34000HZ                         0x25
299 #define BURST_FREQ_32000HZ                         0x24
300 #define BURST_FREQ_30000HZ                         0x23
301 #define BURST_FREQ_28000HZ                         0x22
302 #define BURST_FREQ_26000HZ                         0x21
303 #define BURST_FREQ_24000HZ                         0x20
304 #define BURST_FREQ_22000HZ                         0x1F
305 #define BURST_FREQ_20000HZ                         0x1E
306 #define BURST_FREQ_18000HZ                         0x1C
307 #define BURST_FREQ_16000HZ                         0x1B
308 #define BURST_FREQ_14000HZ                         0x1A
309 #define BURST_FREQ_12000HZ                         0x19
310 #define BURST_FREQ_10000HZ                         0x18
311 #define BURST_FREQ_8000HZ                          0x17
312 #define BURST_FREQ_6000HZ                          0x16
313 #define BURST_FREQ_4000HZ                          0x15
314 #define BURST_FREQ_2000HZ                          0x14
315 #define BURST_FREQ_1000HZ                          0x13
316 #define BURST_FREQ_950HZ                           0x12
317 #define BURST_FREQ_900HZ                           0x11
318 #define BURST_FREQ_850HZ                           0x10
319 #define BURST_FREQ_800HZ                           0x0F
320 #define BURST_FREQ_750HZ                           0x0E
321 #define BURST_FREQ_700HZ                           0x0D
322 #define BURST_FREQ_650HZ                           0x0C
323 #define BURST_FREQ_600HZ                           0x0B
324 #define BURST_FREQ_550HZ                           0x0A
325 #define BURST_FREQ_500HZ                           0x09
326 #define BURST_FREQ_450HZ                           0x08
327 #define BURST_FREQ_400HZ                           0x07
328 #define BURST_FREQ_350HZ                           0x06
329 #define BURST_FREQ_300HZ                           0x05
330 #define BURST_FREQ_250HZ                           0x04
331 #define BURST_FREQ_200HZ                           0x03
332 #define BURST_FREQ_150HZ                           0x02
333 #define BURST_FREQ_100HZ                           0x01
```

```
334 #define BURST_FREQ_50HZ                        0x00
335
336 #define BURST_VALUES_50                        0x01
337 #define BURST_VALUES_100                       0x02
338 #define BURST_VALUES_150                       0x03
339 #define BURST_VALUES_200                       0x04
340 #define BURST_VALUES_250                       0x05
341 #define BURST_VALUES_300                       0x06
342 #define BURST_VALUES_350                       0x07
343 #define BURST_VALUES_400                       0x08
344 #define BURST_VALUES_450                       0x09
345 #define BURST_VALUES_500                       0x0A
346 #define BURST_VALUES_550                       0x0B
347 #define BURST_VALUES_600                       0x0C
348 #define BURST_VALUES_650                       0x0D
349 #define BURST_VALUES_700                       0x0E
350 #define BURST_VALUES_750                       0x0F
351 #define BURST_VALUES_800                       0x10
352 #define BURST_VALUES_850                       0x11
353 #define BURST_VALUES_900                       0x12
354 #define BURST_VALUES_1000                      0x13
355 #define BURST_VALUES_1500                      0x14
356 #define BURST_VALUES_1900                      0x15
357 #define BURST_VALUES_2000                      0x16
358 #define BURST_VALUES_2500                      0x17
359
360 #endif /* CC1101_H_ */
```

```
 1 /*******************************************************************************
 2 **     File name       : convert.h
 3 **     Hardware        : ARM Cortex M4 Basestation
 4 **     Date            : 14/09/2015
 5 **     Last Update     : 02/08/2015
 6 **     Author          : Nico Sassano
 7 **     Description     : convert
 8 **     State           : Final state
 9 *******************************************************************************/
10 #ifndef CONVERT_H_
11 #define CONVERT_H_
12
13 uint8_t char_to_int(char);
14 uint8_t char_to_int2(char, char);
15 uint8_t char_to_int3(char, char, char);
16 uint8_t char_to_hex(char, char);
17 uint16_t char_to_hex3(char, char, char);
18
19 #endif /* CONVERT_H_ */
```

```
 1 /*******************************************************************************
 2 **     File name       : gpio.h
 3 **     Hardware        : ARM Cortex M4 Basestation
 4 **     Date            : 08/08/2014
 5 **     Last Update     : 02/08/2015
 6 **     Author          : Nico Sassano
 7 **     Description     : convert
 8 **     State           : Final state
 9 *******************************************************************************/
10 #ifndef GPIO_H_
11 #define GPIO_H_
12
13 void GPIOinit(void);
14 void LED_test(void);
15
16 #define B4_ON        GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF)
17 #define B4_OFF       GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00)
18
19 #define LED0_ON      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0xFF)
20 #define LED0_OFF     GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0x00)
21
22 #define LED1_ON      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, 0xFF)
23 #define LED1_OFF     GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, 0x00)
24
25 #define LED2_ON      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_4, 0xFF)
26 #define LED2_OFF     GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_4, 0x00)
27
28 #define LED3_ON      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0xFF)
29 #define LED3_OFF     GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x00)
30
31 #define LED_ALL_ON   GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0xFF);\
32                      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, 0xFF);\
33                      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_4, 0xFF);\
34                      GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0xFF)
35
36 #define LED_ALL_OFF  GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0x00);\
37                      GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, 0x00);\
```

```
38                         GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_4, 0x00);\
39                         GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x00)
40
41 #endif /* GPIO_H_ */
```

```
1  /*********************************************************************************
2  **     File name      : interrupts.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 16/05/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : interrupts
8  **     State          : Final state
9  *********************************************************************************/
10 #ifndef INTERRUPTS_H_
11 #define INTERRUPTS_H_
12
13 void GDO2IntHandler(void);
14 void UARTIntHandler(void);
15 void Timer0IntHandler(void);
16 void Timer1IntHandler(void);
17
18 #endif /* INTERRUPTS_H_ */
```

```
1  /*********************************************************************************
2  **     File name      : interrupts.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 22/12/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : interrupts
8  **     State          : Final state
9  *********************************************************************************/
10 #ifndef LTC2376_H_
11 #define LTC2376_H_
12
13 // Definitions for the ouputs
14 #define LTC2376_CS_ENABLE         GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0x00)
15 #define LTC2376_CS_DISABLE        GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0xFF)
16
17 #define LTC2376_CNV_SET_HIGH      GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_6, 0xFF)
18 #define LTC2376_CNV_SET_LOW       GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_6, 0x00)
19
20 // Definitions for the inputs
21 #define LTC2376_BUSY_SET_INPUT    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
22                                   ROM_GPIOPinTypeGPIOInput(GPIO_PORTK_BASE, GPIO_PIN_7)
23
24 #define LTC2376_BUSY_SET_OUTPUT   ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);\
25                                   ROM_GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_7)
26
27 // Definitions for the LTC2376 IRQ setup
28 #define LTC2376_IRQ_ENABLE        GPIOIntEnable(GPIO_PORTK_BASE,GPIO_INT_PIN_7);\
29                                   IntEnable(INT_GPIOK)
30
31 #define LTC2376_IRQ_DISABLE       GPIOIntDisable(GPIO_PORTK_BASE, GPIO_PIN_7);\
32                                   IntDisable(INT_GPIOK)
33
34 #define LTC2376_IRQ_FALLING_EDGE  GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_FALLING_EDGE);
35 #define LTC2376_IRQ_RISING_EDGE   GPIOIntTypeSet(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_RISING_EDGE);
36
37 #define LTC2376_CLEAR_IRQ         GPIOIntClear(GPIO_PORTK_BASE,GPIO_PIN_7)
38
39 // Declarate the prototype functions
40 void init_ltc2376(void);
41 void ltc2376_start(uint8_t, uint8_t);
42
43 #endif /* LTC2376_H_ */
```

```
1  /*********************************************************************************
2  **     File name      : main.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 14/05/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : main
8  **     State          : Final state
9  *********************************************************************************/
10 #ifndef MAIN_H_
11 #define MAIN_H_
12
13 #define HELP                      0x00
14 #define GET_VOLTAGE               0x01
15 #define BURST                     0x02
16 #define GET_BURST                 0x03
```

```
17 #define GET_CONFIG_BS        0x04
18 #define SEND_HEX             0x05
19 #define SET_BRATE            0x06
20 #define CALI_BURST           0x07
21 #define CALI_CLK             0x08
22 #define MATLABMODE           0x09
23 #define LTC2376MODE          0x0A
24 #define AD7691MODE           0x0B
25 #define AD7176MODE           0x0C
26 #define SET_AD5270_BS        0x0D
27 #define SET_AD5270_ZS        0x0E
28 #define GET_CONFIG_ZS        0x0F
29 #define SET_PROPROCESSING    0x10
30 #define GET_GORTZEL          0x11
31 #define WHY                  0xEE
32
33 #endif /* MAIN_H_ */
```

```
1  /*************************************************************************************
2  **     File name      : spi.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 06/06/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : spi
8  **     State          : Final state
9  *************************************************************************************/
10 #ifndef SPI_H_
11 #define SPI_H_
12
13 void SPIinit(uint32_t, uint8_t, uint32_t);
14 void SPI_send_data(int length, uint32_t *);
15
16 #endif /* SPI_H_ */
```

```
1  /*************************************************************************************
2  **     File name      : convert.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 14/09/2015
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : convert
8  **     State          : Final state
9  *************************************************************************************/
10 #ifndef CONVERT_H_
11 #define CONVERT_H_
12
13 uint8_t char_to_int(char);
14 uint8_t char_to_int2(char, char);
15 uint8_t char_to_int3(char, char, char);
16 uint8_t char_to_hex(char, char);
17 uint16_t char_to_hex3(char, char, char);
18
19 #endif /* CONVERT_H_ */
```

```
1  /*************************************************************************************
2  **     File name      : uart.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 16/05/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : uart
8  **     State          : Final state
9  *************************************************************************************/
10 #ifndef UART_H_
11 #define UART_H_
12
13
14 void UARTSend(const uint8_t *pui8Buffer, uint32_t ui32Count);
15 void UARTinit(uint32_t,uint32_t,uint32_t);
16
17 #endif /* UART_H_ */
```

```
1  /*************************************************************************************
2  **     File name      : uart_check.h
3  **     Hardware       : ARM Cortex M4 Basestation
4  **     Date           : 15/09/2014
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : uart_check
8  **     State          : Final state
9  *************************************************************************************/
10 #ifndef UART_CHECK_H_
```

```
11 #define UART_CHECK_H_
12
13 uint8_t check_input(void);
14
15 #endif /* UART_CHECK_H_ */
```

```
1 /*******************************************************************************
2 **     File name      : AD5270.c
3 **     Hardware       : BATSEN ZS Klasse 3 v0.5
4 **     Date           : 18/04/2015
5 **     Last Update    : 02/08/2015
6 **     Author         : Nico Sassano
7 **     Description    : AD5270
8 **     State          : Final state
9 *******************************************************************************/
10 #include "header\main.h"
11 //****************************************************
12 // This function is set the AD5270 DAC
13 // at the first OPA344 OP AMP
14 //****************************************************
15 void write_AD5270_1(uint16_t res_value) {
16     uint16_t value = 0;
17     uint8_t upper_byte = 0;
18     uint8_t lower_byte = 0;
19
20     value = AD5270_WRITE_CMD | res_value;
21
22     upper_byte = (value >> 8);
23     lower_byte = (value >> 0);
24
25      AD5270_1_CS_PxDIR |= AD5270_1_CS_PIN;
26
27     // Enable update of the wiper position and the 50-TP memory contents through the digital interface
28      AD5270_1_CS_ENABLE;
29
30     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
31     UCA0TXBUF = 0x1C;               // Send configuration mode and register
32     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
33     UCA0TXBUF = 0x03;               // Send configuration mode and register
34     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
35
36     AD5270_1_CS_DISABLE;
37
38     delay_ms(1);
39
40     // Update the wiper position
41     AD5270_1_CS_ENABLE;
42
43     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
44     UCA0TXBUF = upper_byte;         // Send configuration mode and register
45     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
46     UCA0TXBUF = lower_byte;         // Send configuration mode and register
47     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
48
49     AD5270_1_CS_DISABLE;
50 }
51
52 //****************************************************
53 // This function is set the AD5270 DAC
54 // at the second OPA344 OP AMP
55 //****************************************************
56 void write_AD5270_2(uint16_t res_value) {
57     uint16_t value = 0;
58     uint8_t upper_byte = 0;
59     uint8_t lower_byte = 0;
60
61     value = AD5270_WRITE_CMD | res_value;
62
63     upper_byte = (value >> 8);
64     lower_byte = (value >> 0);
65
66      AD5270_2_CS_PxDIR |= AD5270_2_CS_PIN;
67
68     // Enable update of the wiper position and the 50-TP memory contents through the digital interface
69      AD5270_2_CS_ENABLE;
70
71     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
72     UCA0TXBUF = 0x1C;               // Send configuration mode and register
73     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
74     UCA0TXBUF = 0x03;               // Send configuration mode and register
75     while (UCA0STAT & UCBUSY);      // Wait for TX to finish
76
77     AD5270_2_CS_DISABLE;
78
79     delay_ms(1);
```

```
80
81      // Update the wiper position
82      AD5270_2_CS_ENABLE;
83
84      while (UCA0STAT & UCBUSY);       // Wait for TX to finish
85      UCA0TXBUF = upper_byte;          // Send configuration mode and register
86      while (UCA0STAT & UCBUSY);       // Wait for TX to finish
87      UCA0TXBUF = lower_byte;          // Send configuration mode and register
88      while (UCA0STAT & UCBUSY);       // Wait for TX to finish
89
90      AD5270_2_CS_DISABLE;
91  }
```

```
/***********************************************************************************
**      File name       : AD7691.c
**      Hardware        : BATSEN ZS Klasse 3 v0.5
**      Date            : 22/12/2014
**      Last Update     : 03/05/2015
**      Author          : Nico Sassano
**      Description     : Functions for the ADC AD7691
**      State           : Final state
***********************************************************************************/
#include "header\main.h"
extern volatile uint32_t AD7691_value_storage[1];
//*****************************************************
// This function is config the AD7691 ADC
//*****************************************************
void init_ad7691(void) {

    // init_spi();
    // BUSY Input Pin
    AD7691_BUSY_PxDIR &= ~AD7691_BUSY_PIN;

    // CNV Output Pin
    AD7691_CNV_PxDIR|= AD7691_CNV_PIN;
    AD7691_CNV_SET_LOW;

    // Set BUSY as interrupt
    AD7691_SET_IRQ_FALLING_EDGE;
    AD7691_BUSY_IRQ_DISABLE;
}
```

```c
/************************************************************************************
**     Description:      adc12.c
**     Hardware:         BATSEN ZS 3 v0.5
**     Date:             06/03/2013
**     Last Update:      26/02/2015
**     Author:           Nico Sassano
**     State            : Final state
************************************************************************************/
#include "header\main.h"
uint16_t adc12_get_volt_sample(uint8_t clk_set) {
    uint16_t adc_value = 0;

    TPS61201_DISABLE;                 // TPS Enable off

    if(clk_set == 16) {
        ADC12CTL0 |= (ADC12ENC | ADC12SC);  // Start conversion with sampling
        while (!(ADC12IFG & BIT0));         // Wait while conversion is active
        P1OUT &= ~BIT4; // OFF
        TPS61201_ENABLE;                    // TPS Enable on

    } else if(clk_set == 1){
        ADC12CTL0 |= (ADC12ENC | ADC12SC);  // Start conversion with sampling
        while (!(ADC12IFG & BIT0));         // Wait while conversion is active
        TPS61201_ENABLE;                    // TPS Enable on
    }

    TPS61201_ENABLE;                        // TPS Enable on
    adc_value = (ADC12MEM0 & 0x0FFF);       // Readout the ADC memory

    //adc12_disable();     // ADC ausschalten

    return adc_value;
}

uint16_t adc12_get_temp_sample(uint8_t clk_set) {
    uint16_t temp_value = 0;

    adc12_temp_init(clk_set);

    ADC12CTL0 |= (ADC12ENC | ADC12SC);             // Start conversion with sampling
    while ((ADC12IFG & 0x0001) != 0x0001) ; // Wait while conversion is active
    temp_value = (ADC12MEM0 & 0x0FFF);

    //adc12_disable();     // Disable ADC again for saving energy
    adc12_volt_init(clk_set,1);

    return temp_value;
}

void adc12_volt_init(uint8_t clk_set, uint8_t channel) {
    ADC12CTL0 = 0;  // reset
    ADC12CTL1 = 0;  // reset

    /*************************************************************************
    ** Important notice.
    ** The hold time of the ADC have to be bigger than 23.733 us.
    *************************************************************************/
    switch(clk_set) {
        case 1: {
            // Enable 2.5V shared reference, disable temperature sensor to save power
            REFCTL0 |= REFMSTR;          // REF Master Control
            REFCTL0 |= REFVSEL_2;        // REF Reference Voltage Level Select 2.5V
            REFCTL0 |= REFON;            // REF Reference On
            REFCTL0 |= REFTCOFF;         // REF Temp.Sensor off

            ADC12CTL0 |= ADC12REF2_5V;   // Set internal REF 2.5V
            ADC12CTL0 |= REFON;          // REF Reference On
            ADC12CTL0 |= ADC12ON;        // ADC12 On
            ADC12CTL0 |= ADC12SHT02;     // 64 ADC12CLK cycles -> 64us (64 us > 23.733 us)

            ADC12CTL1 = ADC12CSTARTADD_0;    // Save conversion result to ADC12MEM0
            ADC12CTL1 |= ADC12CONSEQ_0;      // Single-channel and single-conversion
            ADC12CTL1 |= ADC12SHP;           // ADC12 Sample/Hold Pulse Mode

            ADC12MCTL0 = ADC12SREF_1;        // ADC12 Select Reference 1

            if(channel == 1) {
                ADC12_SET_INPUT_CH1;             // Select ADC channel 1
            }else if(channel == 2) {
                ADC12_SET_INPUT_CH2;             // Select ADC channel 2
            }

            __delay_cycles(75);          // 75 us delay @ ~1MHz

            ADC12CTL0 |= ADC12ENC;       // ADC12 Enable Conversion
```

```
86
87          }break;
88
89          case 16: {
90              // Enable 2.5V shared reference, disable temperature sensor to save power
91              REFCTL0 |= REFMSTR;                // REF Master Control
92              REFCTL0 |= REFVSEL_2;              // REF Reference Voltage Level Select 2.5V
93              REFCTL0 |= REFON;                  // REF Reference On
94              REFCTL0 |= REFTCOFF;               // REF Temp.Sensor off
95
96              ADC12CTL0 = ADC12REF2_5V;          // Set internal REF 2.5V
97              ADC12CTL0 |= REFON;                // REF Reference On
98              ADC12CTL0 |= ADC12ON;              // ADC12 On
99              ADC12CTL0 |= ADC12SHT03;           // 256 ADC12CLK cycles
100                                                // 256 * (1/5MHz) = 51,2us (51,2 us > 23.733 us)
101
102             ADC12CTL1 = ADC12CSTARTADD_0;      // Save conversion result to ADC12MEM0
103             ADC12CTL1 |= ADC12CONSEQ_0;        // Single-channel and single-conversion
104             ADC12CTL1 |= ADC12SHP;             // ADC12 Sample/Hold Pulse Mode
105
106             ADC12MCTL0 = ADC12SREF_1;          // ADC12 Select Reference 1
107
108             if(channel == 1) {
109                 ADC12_SET_INPUT_CH1;               // Select ADC channel 1
110             }else if(channel == 2) {
111                 ADC12_SET_INPUT_CH2;               // Select ADC channel 2
112             }
113
114             __delay_cycles(75);           // 75 us delay @ ~16MHz
115
116             ADC12CTL0 |= ADC12ENC;             // ADC12 Enable Conversion
117         }break;
118
119         default: break;
120     }
121 }
122
123 void adc12_temp_init(uint8_t clk_set) {
124     ADC12CTL0 = 0;  // reset
125     ADC12CTL1 = 0;  // reset
126
127     /****************************************************************************
128     ** Sampel and Hold muss bei jeder CLK änderung neu eingestellt werden.
129     ** Die Hold-Zeit muss größer 37.56 us sein
130     ****************************************************************************/
131
132     ADC12CTL0 = ADC12SHT0_4 + REFON + ADC12REF2_5V + ADC12ON;     // Internal ref = 1.5V
133     ADC12CTL1 |= ADC12SHP;                            // enable sample timer
134     ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_10;              // ADC i/p ch A10 = temp sense i/p
135
136     ADC12CTL0 |= ADC12ENC;
137 }
138
139 void adc12_disable(void) {
140     ADC12CTL0 &= ~ADC12ENC;        // Disable conversion
141     ADC12CTL0 &= ~REFON;      // Disable reference voltage
142     ADC12CTL0 &= ~ADC12ON;    // Disable ADC12
143 }
```

```
1  /********************************************************************************
2  **     Description:     ADG918.c
3  **     Hardware:        BATSEN ZS 3 v0.5
4  **     Date:            06/03/2013
5  **     Last Update:     26/02/2015
6  **     Author         : Nico Sassano
7  **     State          : Final state
8  ********************************************************************************/
                                                                                          */
9  #include "header\main.h"
10 void adg918_init (void) {
11 ADG918_CTRL_PxDIR |= ADG918_CTRL_PIN ;
12 }
13
14 void adg918_wakeup (void) {
15 ADG918_CTRL_PxOUT &= ~ADG918_CTRL_PIN ;
16 }
17
18 void adg918_transceiver (void) {
19 ADG918_CTRL_PxOUT |= ADG918_CTRL_PIN ;
20  }
```

```
1   /*********************************************************************************
2   **     Description    : AS3930.c
3   **     Hardware       : BATSEN ZS 3 v0.5
4   **     Date           : 10/02/2012
5   **     Last Update    : 02/08/2015
6   **     Author         : Phillip Durdaut, Nico Sassano
7   **     State          : Final state
8   *********************************************************************************/
9   #include "header\main.h"
10
11  /* Configuration Modes (bits 15-14) */
12  #define AS3930_WRITE                 0x00
13  #define AS3930_READ                  0x01
14  #define AS3930_COMMAND               0x03
15
16  /* Configuration Registers (bits 13-8) */
17  #define AS3930_R00                   0x00
18  #define AS3930_R01                   0x01
19  #define AS3930_R02                   0x02
20  #define AS3930_R03                   0x03
21  #define AS3930_R04                   0x04
22  #define AS3930_R05                   0x05
23  #define AS3930_R06                   0x06
24  #define AS3930_R07                   0x07
25  #define AS3930_R08                   0x08
26  #define AS3930_R09                   0x09
27  #define AS3930_R10                   0x0a
28  #define AS3930_R11                   0x0b
29  #define AS3930_R12                   0x0c
30  #define AS3930_R13                   0x0d
31
32  /* Commands (bits 7-0) */
33
34  #define AS3930_CLEAR_WAKE            0x00
35  #define AS3930_RESET_RSSI            0x01
36  #define AS3930_TRIM_OSC              0x02
37  #define AS3930_CLEAR_FALSE           0x03
38  #define AS3930_PRESET_DEFAULT        0x04
39
40  void as3930_spi_setup(void);
41  void as3930_spi_write_register(uint8_t address, uint8_t value);
42  char as3930_spi_read_register(uint8_t address);
43  void as3930_spi_command(uint8_t command);
44
45  void as3930_init(void) {
46      as3930_spi_setup();
47      //spi_setup(AS3930);
48
49      AS3930_WAKE_DIR_IN;
50      AS3930_WAKE_IRQ_RISING_EDGE;
51      AS3930_WAKE_IRQ_DISABLE;
52      AS3930_WAKE_CLEAR_IRQ;
53  }
54
55  void as3930_config_no_pattern(void) {
56      as3930_spi_write_register(AS3930_R01, BIT5);
57  }
58
59  void as3930_preset_default(void) {
60      as3930_spi_command(AS3930_PRESET_DEFAULT);
61  }
62
63  void as3930_clear_wakeup(void) {
64      as3930_spi_command(AS3930_CLEAR_WAKE);
65  }
66
67  uint8_t as3930_get_rssi(void) {
68      return (spi_read_register(AS3930_R10,AS3930_READ << 6) & 0x1F);
69  }
70
71  void as3930_spi_setup(void) {
72      AS3930_CS_PxDIR |= AS3930_CS_PIN;    // CS is output
73      AS3930_CS_PxOUT &= ~AS3930_CS_PIN;   // Chip disable
74
75      UCA0CTL1 |= UCSWRST;                 // Hold state machine in reset
76
77      UCA0CTL0 &= ~UCCKPH;                      // Data is changed on rising edge
78      UCA0CTL0 &= ~UCCKPL;                      // Inactive clock is low
79      UCA0CTL0 |= (UCMST | UCMSB | UCSYNC);    // MSB first, Master mode, Synchronous mode
80      UCA0CTL0 &= ~(UCMODE1 | UCMODE0);        // 3-pin SPI
81      UCA0MCTL = 0;                             // No modulation
82
83      // SMCLK / 2
84      UCA0CTL1 |= (UCSSEL1 | UCSSEL0);
85      UCA0BR0 = 2;
```

```
86      UCA0BR1 = 0;
87
88      // SPI functionality for pins
89      AS3930_SPI_PxSEL |= (AS3930_SPI_MOSI_PIN | AS3930_SPI_MISO_PIN | AS3930_SPI_CLK_PIN);
90      AS3930_SPI_PxDIR |= (AS3930_SPI_MOSI_PIN | AS3930_SPI_CLK_PIN); // MOSI and CLK are outputs
91      AS3930_SPI_PxDIR &= ~AS3930_SPI_MISO_PIN; // MISO is input
92
93      UCA0CTL1 &= ~UCSWRST;    // Initialize USART state machine
94  }
95
96  void as3930_spi_write_register(uint8_t address, uint8_t value) {
97      AS3930_CS_PxOUT |= AS3930_CS_PIN;              // Chip enable
98      while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
99      UCA0TXBUF = address | (AS3930_WRITE << 6);    // Send configuration mode and register
100     while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
101     UCA0TXBUF = value;                            // Send value
102     while (UCA0STAT & UCBUSY);                    // Wait for TX complete
103     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;            // Chip disable
104 }
105
106 char as3930_spi_read_register(uint8_t address) {
107     uint8_t value;
108
109     AS3930_CS_PxOUT |= AS3930_CS_PIN;              // Chip enable
110     while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
111     UCA0TXBUF = address | (AS3930_READ << 6);     // Send configuration mode and register
112     while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
113     UCA0TXBUF = 0;                                // Dummy write so we can read data
114     while (UCA0STAT & UCBUSY);                    // Wait for TX complete
115     value = UCA0RXBUF;                            // Read data
116     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;            // Chip disable
117
118     return value;
119 }
120
121 void as3930_spi_command(uint8_t command) {
122     AS3930_CS_PxOUT |= AS3930_CS_PIN;                  // Chip enable
123     while (UCA0STAT & UCBUSY);                        // Wait for TX to finish
124     UCA0TXBUF = command | (AS3930_COMMAND << 6);      // Send configuration mode and register
125     while (UCA0STAT & UCBUSY);                        // Wait for TX to finish
126     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;                // Chip disable
127 }
```

```
1   /*******************************************************************************
2   **      Description     : balancing.c
3   **      Hardware        : BATSEN ZS 3 v0.5
4   **      Date            : 10/02/2012
5   **      Last Update     : 02/08/2015
6   **      Author          : Phillip Durdaut, Nico Sassano
7   **      State           : Final state
8   *******************************************************************************/
9   #include "header\main.h"
10  void balancing_init (void) { // Ports initialisieren
11      BALANCE_PORT_1_PxDIR |= BALANCE_PORT_1_PIN ;
12      BALANCE_PORT_2_PxDIR |= BALANCE_PORT_2_PIN ;
13   }
14
15  void balancing_on (void) { //Balancierung an
16      BALANCE_PORT_1_PxOUT |= BALANCE_PORT_1_PIN ;
17      BALANCE_PORT_2_PxOUT |= BALANCE_PORT_2_PIN ;
18   }
19
20  void balancing_port_1_on (void) { // Pfad 1 anschalten
21      BALANCE_PORT_1_PxOUT |= BALANCE_PORT_1_PIN ;
22   }
23
24  void balancing_port_2_on (void) { // Pfad 2 anschalten
25      BALANCE_PORT_2_PxOUT |= BALANCE_PORT_2_PIN ;
26   }
27
28  void balancing_off (void) { // Balancierung ausschalten
29      BALANCE_PORT_1_PxOUT &=~BALANCE_PORT_1_PIN ;
30      BALANCE_PORT_2_PxOUT &=~BALANCE_PORT_2_PIN ;
31   }
32
33  void balancing_port_1_off (void) { // Pfad 1 ausschalten
34      BALANCE_PORT_1_PxOUT &=~BALANCE_PORT_1_PIN ;
35   }
36
37  void balancing_port_2_off (void) { // Pfad 2 ausschalten
38      BALANCE_PORT_2_PxOUT &=~BALANCE_PORT_2_PIN ;
39   }
```

```
1   /*********************************************************************************
2   **     Description    : CC430.c
3   **     Hardware       : BATSEN ZS 3 v0.5
4   **     Date           : 10/12/2014
5   **     Last Update    : 02/08/2015
6   **     Author         : Nico Sassano
7   **     State          : Final state
8   *********************************************************************************/
9   #include "header\main.h"
10
11  #define PATABLE_LENGTH 8
12  #define  PATABLE_VAL          (0xC6)            // 0 dBm output
13  uint8_t cc430_patable[PATABLE_LENGTH] = { 0x00, 0xC6, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};  // 10dBm output
14
15  extern volatile uint8_t brate;
16  extern volatile uint8_t irq_mode;
17  extern volatile uint8_t irq_timera;
18  extern volatile uint8_t irq_send_done_flag;
19  extern volatile uint8_t command_recived;
20
21  void cc430_tx(uint8_t packetes, uint8_t *txbuf) {
22      /***********************************
23       * Sendezeit optimierung 26.03.13 NS
24       ***********************************/
25      cc430_set_tx(brate);
26      IRQ_SET_TX;
27      CC430_END_OF_PKT_CLEAR_IRQ;         // Clear the sync word detected interrupt
28      CC430_END_OF_PKT_IRQ_ENABLE;        // Enable the end of package interrupt
29
30      if(IRQ_TIMERA_IS_BALANCING) {
31          TIMER_A_STOP;
32          TIMER_A_0_CM_IRQ_DISABLE;
33      }
34
35      led_on(LED_TX);
36      cc430_fill_tx_fifo(txbuf, packetes);
37      cc430_tx_start(packetes);
38
39      while(!irq_send_done_flag);
40
41      if(IRQ_TIMERA_IS_BALANCING) {
42          TIMER_A_0_CM_IRQ_ENABLE;
43          TIMER_A_START_UP_MODE;
44      }
45
46      led_off(LED_TX);
47
48      irq_send_done_flag = 0;
49
50      cc430_idle();
51
52  }
53
54  /*****************************************************************************************
55  **
56  *****************************************************************************************/
57  void cc430_rx(uint8_t packages) {
58      cc430_set_rx(brate);
59      IRQ_SET_RX;
60      CC430_END_OF_PKT_CLEAR_IRQ;         // Clear the sync word detected interrupt
61      CC430_END_OF_PKT_IRQ_ENABLE;        // Enable the sync word detected interrupt
62      CC430_END_OF_PKT_CLEAR_IRQ;
63      cc430_rx_start(packages);                   // Transceiver in RX state
64      command_recived = 0;
65  }
66
67  /*****************************************************************************************
68  **
69  *****************************************************************************************/
70  void tx_carrier(void) {
71      led_on(LED_TX);
72      adg918_transceiver();
73      cc430_reset();          // Reset chip and go to idle state
74      SetVCore(2);
75
76
77      cc430_config_no_packet();
78      // Internal timer output (10.5 kHz) to Radio TX
79        // NOTE: SMARTF_CC430 IOCFG0 should = 0x2E. When IOCFG0 = 0x2D,
80        // asynchronous data into the radio is taken from GDO0 and not the timer.
81        TA1CCR0 = 50;
82        TA1CCR1 = 50;
83
84        TA1CCTL0 = OUTMOD_4;
85        TA1CCTL1 = OUTMOD_4;
```

```
86          TA1CTL = TASSEL__SMCLK + MC_1 + TACLR;
87
88          //Transmit the TX waveform asynchronously
89        cc430_tx_carrier();
90
91        while(1);
92    }
93
94    void cc430_set_tx(uint8_t brate) {
95        cc430_init_tx();
96        cc430_reset();                  // Reset chip and go to idle state
97        cc430_config_packet(brate);     // Configure the transceiver for sending packets
98    }
99
100   void cc430_set_rx(uint8_t brate) {
101       cc430_init_rx();
102       cc430_reset();                  // Reset chip and go to idle state
103       cc430_config_packet(brate);     // Configure the transceiver for sending packets
104   }
105
106   void cc430_init_rx(void) {
107       CC430_END_OF_PKT_IRQ_MODE;
108       CC430_END_OF_PKT_IRQ_DISABLE;
109       CC430_END_OF_PKT_CLEAR_IRQ;
110   }
111
112   void cc430_init_burst(void) {
113
114       CC430_GDO2_IRQ_FALLING_EDGE;
115       CC430_GDO2_IRQ_DISABLE;
116       CC430_GDO2_IRQ_CLEAR;
117   }
118
119   void cc430_init_tx(void) {
120       CC430_END_OF_PKT_IRQ_MODE;
121       CC430_END_OF_PKT_IRQ_DISABLE;
122       CC430_END_OF_PKT_CLEAR_IRQ;
123   }
124
125   void cc430_reset(void) {
126       Strobe (RF_SRES);               // Reset the Radio Core
127       delay_ms(5);                    // Important refresh time
128   }
129
130   /*******************************************************************************
131   ** Konfiguration für die Burst Kalibrierung
132   *******************************************************************************/
133   void cc430_config_cali(void) {
134       // GDO2 Output Pin Configuration 2
135       // 0x0D -> Serial Data Output. Used for asynchronous serial mode
136       WriteSingleReg(IOCFG2, 0x2D);
137       // GDO0 Output Pin Configuration 0
138       WriteSingleReg(IOCFG0, 0x2E);
139       // RX FIFO and TX FIFO Thresholds
140       WriteSingleReg(FIFOTHR, 0x47);
141       // Packet Automation Control 0
142       WriteSingleReg(PKTCTRL0, 0x35);
143       // Frequency Synthesizer Control 1
144       WriteSingleReg(FSCTRL1, 0x06);
145       // Frequency Control Word, High Byte
146       WriteSingleReg(FREQ2, 0x10);
147       // Frequency Control Word, Middle Byte
148       WriteSingleReg(FREQ1, 0xB1);
149       //Frequency Control Word, LOW Byte
150       WriteSingleReg(FREQ0, 0x3B);
151       /************************************************/
152       // Modem Configuration 4
153       WriteSingleReg(MDMCFG4, 0x5D);
154       // Modem Configuration 3
155       WriteSingleReg(MDMCFG3, 0x3B);
156       // Modem Configuration 2
157       WriteSingleReg(MDMCFG2, 0x30);
158       /************************************************/
159       // Modem Deviation Setting
160       WriteSingleReg(DEVIATN, 0x15);
161       // Main Radio Control State Machine Configuration 1
162       WriteSingleReg(MCSM1, 0x30);
163       // Main Radio Control State Machine Configuration 0
164       WriteSingleReg(MCSM0, 0x10);
165       // Frequency Offset Compensation Configuration
166       WriteSingleReg(FOCCFG, 0x16);
167       // Wake On Radio Control
168       WriteSingleReg(WORCTRL, 0xFB);
169       // Front End RX Configuration 0
170       WriteSingleReg(FREND0, 0x11);
```

```
171        // 10 dBm output power
172        WriteBurstPATable(cc430_patable, PATABLE_LENGTH);   // output power
173        //WriteSinglePATable(PATABLE_VAL);
174    }
175
176    void cc430_config_no_packet(void) {
177        WriteSingleReg(IOCFG2,0x0B);      // GDO2 Output Configuration
178        WriteSingleReg(IOCFG0,0x2D);      // GDO0 Output Configuration
179        WriteSingleReg(FIFOTHR,0x47);     // RX FIFO and TX FIFO Thresholds
180        WriteSingleReg(PKTCTRL0,0x32);    // Packet Automation Control
181        WriteSingleReg(FSCTRL1,0x06);     // Frequency Synthesizer Control
182        WriteSingleReg(FREQ2,0x10);       // Frequency Control Word, High Byte
183        WriteSingleReg(FREQ1,0xB1);       // Frequency Control Word, Middle Byte
184        WriteSingleReg(FREQ0,0x3B);       // Frequency Control Word, Low Byte
185        WriteSingleReg(MDMCFG4,0xCA);     // Modem Configuration
186        WriteSingleReg(MDMCFG3,0x93);     // Modem Configuration
187        WriteSingleReg(MDMCFG2,0x30);     // Modem Configuration
188        WriteSingleReg(DEVIATN,0x34);     // Modem Deviation Setting
189        WriteSingleReg(MCSM0,0x10);       // Main Radio Control State Machine Configuration
190        WriteSingleReg(FOCCFG,0x16);      // Frequency Offset Compensation Configuration
191        WriteSingleReg(WORCTRL,0xFB);     // Wake On Radio Control
192        WriteSingleReg(FREND0,0x11);      // Front End TX Configuration
193        WriteSingleReg(FSCAL3,0xE9);      // Frequency Synthesizer Calibration
194        WriteSingleReg(FSCAL2,0x2A);      // Frequency Synthesizer Calibration
195        WriteSingleReg(FSCAL1,0x00);      // Frequency Synthesizer Calibration
196        WriteSingleReg(FSCAL0,0x1F);      // Frequency Synthesizer Calibration
197        WriteSinglePATable(PATABLE_VAL);
198    }
199
200    void cc430_config_no_packet_tx(void) {
201
202        WriteSingleReg(IOCFG2, 0x02E);        // 0x0D -> Serial Data Output. Used for asynchronous serial mode
203        WriteSingleReg(IOCFG1, 0x2E);     // 0x2E -> High impedance (3-state)
204        WriteSingleReg(IOCFG0, 0x0D);      // 0x36 -> CLK_XOSC/8
205       WriteSingleReg(PKTCTRL0, 0x32);   // Packet Automation Control 0
206       WriteSingleReg(CHANNR,0x00);
207       WriteSingleReg(FSCTRL0, 0x06);    // Frequency Synthesizer Control 1
208       WriteSingleReg(FREQ2, 0x10);      // Frequency Control Word, High Byte
209       WriteSingleReg(FREQ1, 0xB1);      // Frequency Control Word, Middle Byte
210       WriteSingleReg(FREQ0, 0x3B);      // Frequency Control Word, LOW Byte
211
212       /***********************************************/
213       // Modem Configuration 4
214       // 250kBaud              ->  0xxD
215       // RX Filter 58.035714   ->  0xFx
216       WriteSingleReg(MDMCFG4, 0xFD);    // org
217       // Modem Configuration 3
218       // 250kBaud -> 0x3B
219       WriteSingleReg(MDMCFG3, 0x3B);    // org
220
221       // Modem Configuration 2
222       // OOK -> 0x30
223       WriteSingleReg(MDMCFG2, 0x30);
224       // Modem Configuration 1
225       // Channel spacing 199.951172kHz -> 0x22
226       WriteSingleReg(MDMCFG1, 0x22);
227       // Modem Configuration 0
228
229       /***********************************************/
230       WriteSingleReg(DEVIATN, 0x15);    // Modem Deviation Setting
231       WriteSingleReg(MCSM0, 0x18);      // Main Radio Control State Machine Configuration 0
232       WriteSingleReg(FOCCFG, 0x16);     // Frequency Offset Compensation Configuration
233       WriteSingleReg(FREND0, 0x11);     // Front End RX Configuration 0
234
235       WriteBurstPATable(cc430_patable, PATABLE_LENGTH);   // output power
236    }
237
238    void cc430_config_no_packet_rx(void) {
239
240       WriteSingleReg(IOCFG2, 0x0D);     // 0x0D -> Serial Data Output. Used for asynchronous serial mode
241       WriteSingleReg(IOCFG1, 0x2E);     // 0x2E -> High impedance (3-state)
242       WriteSingleReg(IOCFG0, 0x36);     // 0x36 -> CLK_XOSC/8
243       WriteSingleReg(FIFOTHR, 0x47);    // RX FIFO and TX FIFO Thresholds
244       WriteSingleReg(PKTCTRL0, 0x32);   // Packet Automation Control 0
245       WriteSingleReg(FSCTRL1, 0x06);    // Frequency Synthesizer Control 1
246       WriteSingleReg(FREQ2, 0x10);      // Frequency Control Word, High Byte
247       WriteSingleReg(FREQ1, 0xB1);      // Frequency Control Word, Middle Byte
248       WriteSingleReg(FREQ0, 0x3B);      // Frequency Control Word, LOW Byte
249
250       /***********************************************/
251       // Modem Configuration 4
252       // 250kBaud              ->  0xxD
253       // RX Filter 58.035714   ->  0xFx
254       WriteSingleReg(MDMCFG4, 0x5D);    // org
255       // Modem Configuration 3
```

```
256     // 250kBaud -> 0x3B
257     WriteSingleReg(MDMCFG3, 0x3B);    // org
258
259     // Modem Configuration 2
260     // OOK -> 0x30
261     WriteSingleReg(MDMCFG2, 0x30);
262     // Modem Configuration 1
263     // Channel spacing 199.951172kHz -> 0x22
264     WriteSingleReg(MDMCFG1, 0x22);
265     // Modem Configuration 0
266     // Channel spacing 199.951172kHz -> 0xF8
267     WriteSingleReg(MDMCFG0, 0xF8);
268     /***********************************************/
269     WriteSingleReg(DEVIATN, 0x15);    // Modem Deviation Setting
270     WriteSingleReg(MCSM2, 0x07);      // Main Radio Control State Machine Configuration 2
271     WriteSingleReg(MCSM1, 0x30);      // Main Radio Control State Machine Configuration 1
272     WriteSingleReg(MCSM0, 0x18);      // Main Radio Control State Machine Configuration 0
273     WriteSingleReg(FOCCFG, 0x16);     // Frequency Offset Compensation Configuration
274     WriteSingleReg(BSCFG, 0x6C);      // Bit Synchronization Configuration
275     WriteSingleReg(AGCCTRL2, 0x03);   // AGC Control 2
276     WriteSingleReg(AGCCTRL1, 0x40);   // AGC Control 1
277     WriteSingleReg(AGCCTRL0, 0x91);   // AGC Control 0
278     WriteSingleReg(WORCTRL, 0x8F);    // Wake On Radio Control
279     WriteSingleReg(FREND1, 0x56);     // Front End RX Configuration 1
280     WriteSingleReg(FREND0, 0x11);     // Front End RX Configuration 0
281     WriteBurstPATable(cc430_patable, PATABLE_LENGTH);  // output power
282 }
283
284 void cc430_config_cali_burst(void) {
285
286     WriteSingleReg(IOCFG2, 0x0D);     // 0x0D -> Serial Data Output. Used for asynchronous serial mode
287     WriteSingleReg(IOCFG1, 0x2E);     // 0x2E -> High impedance (3-state)
288     WriteSingleReg(IOCFG0, 0x2D);     // 0x36 -> CLK_XOSC/8
289     WriteSingleReg(FIFOTHR, 0x47);    // RX FIFO and TX FIFO Thresholds
290     WriteSingleReg(PKTCTRL0, 0x32);   // Packet Automation Control 0
291     WriteSingleReg(FSCTRL1, 0x06);    // Frequency Synthesizer Control 1
292     WriteSingleReg(FREQ2, 0x10);      // Frequency Control Word, High Byte
293     WriteSingleReg(FREQ1, 0xB1);      // Frequency Control Word, Middle Byte
294     WriteSingleReg(FREQ0, 0x3B);      // Frequency Control Word, LOW Byte
295
296     /***********************************************/
297     // Modem Configuration 4
298     // 250kBaud            ->  0xxD
299     // RX Filter 58.035714  ->  0xFx
300     WriteSingleReg(MDMCFG4, 0x5D);    // org
301     // Modem Configuration 3
302     // 250kBaud -> 0x3B
303     WriteSingleReg(MDMCFG3, 0x3B);    // org
304     // Modem Configuration 2
305     // OOK -> 0x30
306     WriteSingleReg(MDMCFG2, 0x30);
307     // Modem Configuration 1
308     // Channel spacing 199.951172kHz -> 0x22
309     WriteSingleReg(MDMCFG1, 0x22);
310     // Modem Configuration 0
311     // Channel spacing 199.951172kHz -> 0xF8
312     WriteSingleReg(MDMCFG0, 0xF8);
313     /***********************************************/
314     WriteSingleReg(DEVIATN, 0x15);    // Modem Deviation Setting
315     WriteSingleReg(MCSM2, 0x07);      // Main Radio Control State Machine Configuration 2
316     WriteSingleReg(MCSM1, 0x30);      // Main Radio Control State Machine Configuration 1
317     WriteSingleReg(MCSM0, 0x18);      // Main Radio Control State Machine Configuration 0
318     WriteSingleReg(FOCCFG, 0x16);     // Frequency Offset Compensation Configuration
319     WriteSingleReg(BSCFG, 0x6C);      // Bit Synchronization Configuration
320     WriteSingleReg(AGCCTRL2, 0x03);   // AGC Control 2
321     WriteSingleReg(AGCCTRL1, 0x40);   // AGC Control 1
322     WriteSingleReg(AGCCTRL0, 0x91);   // AGC Control 0
323     WriteSingleReg(WORCTRL, 0x8F);    // Wake On Radio Control
324     WriteSingleReg(FREND1, 0x56);     // Front End RX Configuration 1
325     WriteSingleReg(FREND0, 0x11);     // Front End RX Configuration 0
326
327     //WriteBurstPATable(cc430_patable, PATABLE_LENGTH);  // output power
328     WriteSinglePATable(PATABLE_VAL);
329 }
330
331 void cc430_config_packet(uint8_t brate) {
332       WriteSingleReg(IOCFG0,0x06);   //GDO0 Output Configuration
333       WriteSingleReg(IOCFG2,0x29);   //GDO0 Output Configuration
334       WriteSingleReg(FIFOTHR,0x47);  //RX FIFO and TX FIFO Thresholds
335       WriteSingleReg(SYNC1,0x81);  //  Sync-Word Hi-byte
336       WriteSingleReg(SYNC0,0x81);  //  Sync-Word Lo-byte
337       WriteSingleReg(PKTCTRL0,0x04);  // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast.
338       WriteSingleReg(PKTCTRL1,0x0A);  //     // No whitening, FIFO mode, CRC enabled, Fixed packet length
339       WriteSingleReg(ADDR,ADDRESS_THIS_SENSOR);  //     // Adress of this Sensor
340       WriteSingleReg(FSCTRL1,0x06);  // IF frequency: 152.34375 kHz
```

```
341         WriteSingleReg(FREQ2,0x10);    // Carrier frequency: 433.999969 MHz
342         WriteSingleReg(FREQ1,0xB1);    //
343         WriteSingleReg(FREQ0,0x3B);    //
344 
345         WriteSingleReg(AGCCTRL2, 0x05);
346         WriteSingleReg(AGCCTRL1, 0x00);
347         WriteSingleReg(AGCCTRL0, 0x91);
348         switch(brate) {
349                 /*************************************************************************************
350                 ** Übertragungstest
351                 ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
352                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
353                 *************************************************************************************/
354                 case 40: {
355                     WriteSingleReg(MDMCFG4, 0x8A);
356                     WriteSingleReg(MDMCFG3, 0x93);
357                     WriteSingleReg(MDMCFG2, 0x33);
358                     WriteSingleReg(MDMCFG1, 0x22);
359                     WriteSingleReg(MDMCFG0, 0x7A);
360                 }break;
361 
362                 /*************************************************************************************
363                 ** Übertragungstest
364                 ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
365                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
366                 *************************************************************************************/
367                 case 60: {
368                     WriteSingleReg(MDMCFG4, 0x8B);
369                     WriteSingleReg(MDMCFG3, 0x2E);
370                     WriteSingleReg(MDMCFG2, 0x33);
371                     WriteSingleReg(MDMCFG1, 0x22);
372                     WriteSingleReg(MDMCFG0, 0x7A);
373                 }break;
374 
375                 /*************************************************************************************
376                 ** Übertragungstest
377                 ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
378                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
379                 *************************************************************************************/
380                 case 80: {
381                     WriteSingleReg(MDMCFG4, 0x8B);
382                     WriteSingleReg(MDMCFG3, 0x93);
383                     WriteSingleReg(MDMCFG2, 0x33);
384                     WriteSingleReg(MDMCFG1, 0x22);
385                     WriteSingleReg(MDMCFG0, 0x7A);
386                 }break;
387 
388                 /*************************************************************************************
389                 ** Übertragungstest
390                 ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
391                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
392                 *************************************************************************************/
393                 case 100: {
394                     WriteSingleReg(MDMCFG4, 0x8B);
395                     WriteSingleReg(MDMCFG3, 0xF8);
396                     WriteSingleReg(MDMCFG2, 0x33);
397                     WriteSingleReg(MDMCFG1, 0x22);
398                     WriteSingleReg(MDMCFG0, 0x7A);
399                 }break;
400 
401                 /*************************************************************************************
402                 ** Übertragungstest
403                 ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
404                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
405                 *************************************************************************************/
406                 case 120: {
407                     WriteSingleReg(MDMCFG4, 0x8C);
408                     WriteSingleReg(MDMCFG3, 0x2E);
409                     WriteSingleReg(MDMCFG2, 0x33);
410                     WriteSingleReg(MDMCFG1, 0x22);
411                     WriteSingleReg(MDMCFG0, 0x7A);
412                 }break;
413 
414                 /*************************************************************************************
415                 ** Übertragungstest
416                 ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
417                 ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
418                 *************************************************************************************/
419                 case 140: {
420                     WriteSingleReg(MDMCFG4, 0x8C);
421                     WriteSingleReg(MDMCFG3, 0x61);
422                     WriteSingleReg(MDMCFG2, 0x33);
423                     WriteSingleReg(MDMCFG1, 0x22);
424                     WriteSingleReg(MDMCFG0, 0x7A);
```

```
425             }break;
426
427             /*********************************************************************************
428             ** Übertragungstest
429             ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
430             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
431             *********************************************************************************/
432             case 160: {
433                 WriteSingleReg(MDMCFG4, 0x8C);
434                 WriteSingleReg(MDMCFG3, 0x93);
435                 WriteSingleReg(MDMCFG2, 0x33);
436                 WriteSingleReg(MDMCFG1, 0x22);
437                 WriteSingleReg(MDMCFG0, 0x7A);
438             }break;
439
440             /*********************************************************************************
441             ** Übertragungstest
442             ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
443             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
444             *********************************************************************************/
445             case 180: {
446                 WriteSingleReg(MDMCFG4, 0x8C);
447                 WriteSingleReg(MDMCFG3, 0xC6);
448                 WriteSingleReg(MDMCFG2, 0x33);
449                 WriteSingleReg(MDMCFG1, 0x22);
450                 WriteSingleReg(MDMCFG0, 0x7A);
451             }break;
452
453         }
454     WriteSingleReg(DEVIATN,0x15); //Modem Deviation Setting
455     WriteSingleReg(MCSM0,0x10);   // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
456     WriteSingleReg(FOCCFG,0x16);  // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm ->
          SmartRF Studio
457
458     WriteBurstPATable(cc430_patable, PATABLE_LENGTH);
459
460     WriteSingleReg(FREND0,0x11);  //Front End TX Configuration
461     WriteSingleReg(FSCAL3,0xEA);  //Frequency Synthesizer Calibration
462     WriteSingleReg(FSCAL2,0x2A);  //Frequency Synthesizer Calibration
463     WriteSingleReg(FSCAL1,0x00);  //Frequency Synthesizer Calibration
464     WriteSingleReg(FSCAL0,0x1F);  //Frequency Synthesizer Calibration
465 }
466
467 void cc430_fill_tx_fifo(uint8_t * buffer, uint8_t length) {
468     // Clear TX FIFO
469     cc430_clear_tx_fifo();
470
471     // Transfer the bytes to the TX fifo of the transceiver
472     WriteBurstReg(RF_TXFIFOWR, buffer, length);
473 }
474
475 void cc430_read_rx_fifo(uint8_t * buffer, uint8_t length) {
476     // Disable the receiver
477     cc430_idle();
478     // Transfer the bytes via SPI from the RX fifo
479     ReadBurstReg(RF_RXFIFORD, buffer, length);
480 }
481 void cc430_enable_crc(void) {
482     uint8_t current_state;
483     current_state = ReadSingleReg(PKTCTRL1);
484     WriteSingleReg(PKTCTRL1,    current_state | CC430_CRC_EN );
485 }
486
487 void cc430_disable_crc(void) {
488     uint8_t current_state;
489     current_state = ReadSingleReg(PKTCTRL1);
490     WriteSingleReg(PKTCTRL1,    current_state & ~CC430_CRC_EN );
491 }
492 void cc430_clear_tx_fifo(void) {
493     Strobe(RF_SIDLE);            // Needs to be in Idle state to Flush Fifo
494     Strobe(RF_SFTX);           // Strobe SFTX -> Flush TX-Fifo
495 }
496
497 void cc430_clear_rx_fifo(void) {
498     Strobe(RF_SIDLE);            // Needs to be in Idle state to Flush Fifo
499     Strobe(RF_SFRX);           // Strobe SFTX -> Flush RX-Fifo
500 }
501
502 uint8_t  cc430_get_rxbytes(void) {
503     return (ReadSingleReg(RXBYTES));
504 }
505
506 void cc430_sleep(void) {
507     Strobe(RF_SPWD);
508 }
```

```
509
510 void cc430_idle(void) {
511     Strobe(RF_SIDLE);
512 }
513
514 void cc430_tx_carrier(void) {
515     Strobe(RF_SIDLE);
516     Strobe(RF_STX);
517 }
518 void cc430_tx_start(uint8_t packages_to_tx) {
519      // DATA packet length
520     WriteSingleReg(PKTLEN, packages_to_tx);
521
522     // Enable CRC calculation when transmitting data
523     cc430_enable_crc();
524
525     // TX state
526     Strobe(RF_STX);
527
528     // Wait 10 ms for TX finished
529     //delay_ms(8);
530 }
531
532 void cc430_rx_start(uint8_t packages_to_rx) {
533     // DATA packet length
534     WriteSingleReg(PKTLEN, packages_to_rx);
535
536     // Enable CRC check when receiving data
537     cc430_enable_crc();
538
539     // Clear RX FIFO
540     cc430_clear_rx_fifo();
541
542     // RX state
543     Strobe(RF_SRX);
544 }
545
546 void cc430_burst_rx(void) {
547     //Strobe(RF_SIDLE);
548     Strobe(RF_SRX);
549 }
550
551 void cc430_burst_tx(void) {
552     // TX state
553     Strobe(RF_STX);
554 }
555
556 uint8_t cc430_get_partnum(void) {
557     return ReadSingleReg(PARTNUM);
558 }
559
560 uint8_t cc430_get_version(void) {
561     return ReadSingleReg(VERSION);
562 }
563
564 uint8_t cc430_get_marcstate(void) {
565     return ReadSingleReg(MARCSTATE);
566 }
```

```
1   /*******************************************************************************
2   **     Description    : clk.c
3   **     Hardware       : BATSEN ZS 3 v0.5
4   **     Date           : 24/04/2013
5   **     Last Update    : 02/08/2015
6   **     Author         : Nico Sassano
7   **     State          : Final state
8   *******************************************************************************/
9   #include "header\main.h"
10
11  extern uint8_t clk_set;
12  extern uint32_t dco_value;
13
14  void clk_init_startup(void) {    // 06.05.13 NS
15
16      clk_set = 1;     //CLK ist auf 1MHz gesetzt
17      // Set external XT off
18      UCSCTL6   |= XT2OFF;
19
20      __bis_SR_register(SCG0);                  // Disable the FLL control loop
21      UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
22      UCSCTL1 = DCORSEL_3;                      // Select DCO range 16MHz operation
23      UCSCTL2 = FLLD_1 + DCO_MP_1MHZ;                   // Set DCO Multiplier for 8MHz
24                                                // (N + 1) * FLLRef = Fdco
25                                                // (249 + 1) * 32768 = 8MHz
26                                                // Set FLL Div = fDCOCLK/2
27      __bic_SR_register(SCG0);                  // Enable the FLL control loop
28      // Worst-case settling time for the DCO when the DCO range bits have been
29      // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
30      // UG for optimization.
31      // 32 x 32 x 1 MHz / 32768 Hz = 31250 = MCLK cycles for DCO to settle
32      __delay_cycles(31250);
33
34  }
35
36  void clk_init_MHz(uint8_t mhz) {
37
38      switch(mhz) {
39          case 1: {
40              clk_set = 1;     //CLK ist auf 1MHz gesetzt
41              __bis_SR_register(SCG0);                  // Disable the FLL control loop
42              UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
43              UCSCTL1 = DCORSEL_3;                      // Select DCO range 16MHz operation
44              UCSCTL2 = FLLD_1 + DCO_MP_1MHZ;                   // Set DCO Multiplier for 8MHz
45                                                        // (N + 1) * FLLRef = Fdco
46                                                        // (249 + 1) * 32768 = 8MHz
47                                                        // Set FLL Div = fDCOCLK/2
48              __bic_SR_register(SCG0);                  // Enable the FLL control loop
49              // Worst-case settling time for the DCO when the DCO range bits have been
50              // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
51              // UG for optimization.
52              // 32 x 32 x 1 MHz / 32768 Hz = 31250 = MCLK cycles for DCO to settle
53              __delay_cycles(31250);
54
55          }break;
56
57          case 8: {
58
59              clk_set = 8;     //CLK ist auf 8MHz gesetzt
60
61              __bis_SR_register(SCG0);                  // Disable the FLL control loop
62              UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
63              UCSCTL1 = DCORSEL_5;                      // Select DCO range 16MHz operation
64              UCSCTL2 = FLLD_1 + DCO_MP_8MHZ;                   // Set DCO Multiplier for 8MHz
65                                                        // (N + 1) * FLLRef = Fdco
66                                                        // (249 + 1) * 32768 = 8MHz
67                                                        // Set FLL Div = fDCOCLK/2
68              __bic_SR_register(SCG0);                  // Enable the FLL control loop
69
70              // Worst-case settling time for the DCO when the DCO range bits have been
71              // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
72              // UG for optimization.
73              // 32 x 32 x 8 MHz / 32,768 Hz = 250000 = MCLK cycles for DCO to settle
74              __delay_cycles(250000);
75          }break;
76
77          case 12: {
78
79              clk_set = 12;     //CLK ist auf 12MHz gesetzt
80
81              __bis_SR_register(SCG0);                  // Disable the FLL control loop
82              UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
83              UCSCTL1 = DCORSEL_6;                      // Select DCO range 24MHz operation
84              UCSCTL2 = FLLD_1 + DCO_MP_12MHZ;                  // Set DCO Multiplier for 12MHz
85                                                        // (N + 1) * FLLRef = Fdco
```

```
86                                                      // (374 + 1) * 32768 = 12MHz
87                                                      // Set FLL Div = fDCOCLK/2
88              __bic_SR_register(SCG0);                // Enable the FLL control loop
89
90              // Worst-case settling time for the DCO when the DCO range bits have been
91              // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
92              // UG for optimization.
93              // 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
94              __delay_cycles(375000);
95          }break;
96
97          case 16: {
98
99              clk_set = 16;     //CLK ist auf 16MHz gesetzt
100
101
102             __bis_SR_register(SCG0);                // Disable the FLL control loop
103             UCSCTL0 = 0x0000;                       // Set lowest possible DCOx, MODx
104             UCSCTL1 = DCORSEL_7;                    // Select DCO range 24MHz operation
105             UCSCTL2 = FLLD_1 + dco_value;           // Set DCO Multiplier for 12MHz
106                                                     // (N + 1) * FLLRef = Fdco
107                                                     // (487 + 1) * 32768 = 16MHz
108                                                     // Set FLL Div = fDCOCLK/2
109             __bic_SR_register(SCG0);                // Enable the FLL control loop
110
111             // Worst-case settling time for the DCO when the DCO range bits have been
112             // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
113             // UG for optimization.
114             // 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
115             __delay_cycles(500000);
116         }break;
117     }
118 }
```

```
1  /*******************************************************************************
2  **    Description   : delay.c
3  **    Hardware      : BATSEN ZS 3 v0.5
4  **    Date          : 09/04/2013
5  **    Last Update   : 02/08/2015
6  **    Author        : Nico Sassano
7  **    State         : Final state
8  *******************************************************************************/
9  #include "header\main.h"
10 extern uint8_t clk_set;
11 void delay(u16_t i);
12
13 void delay_10us_16MHz(u16_t delay_10us_16MHz) {
14     while (delay_10us_16MHz > 0) {
15         delay(DELAY_CYCLES_PER_10US_16MHZ);
16         delay_10us_16MHz--;
17     }
18 }
19
20 void delay(u16_t i) {
21     while (i > 0) {
22         _NOP();
23         i--;
24     }
25 }
```

```
1   /*********************************************************************************
2   **     Description     : i2c.c
3   **     Hardware        : BATSEN ZS 3 v0.5
4   **     Date            : 05/03/2013
5   **     Last Update     : 02/08/2015
6   **     Author          : Nico Sassano
7   **     State           : Final state
8   *********************************************************************************/
9   #include "header\main.h"
10  #define UCB0BR_BIT_CLK 100.0      // UCB0 Bit Clock [kHz]
11                                    // UCB0 Baud Rate Control 0 Setting
12  void i2c_init(void) {
13      I2C_SDA_PxSEL |= I2C_SDA_PIN;    //Set I2C Pins to I2C Communication
14      I2C_SCL_PxSEL |= I2C_SCL_PIN;
15
16
17      UCB0CTL1 |= UCSWRST;          // Reset the USCI logic
18
19
20      /** control register 0 *******************************************************
21      * Set UCB0CTL  -> Master mode | I2C Mode | Synchronous mode
22      *              -> Own address is a 7-bit address, Address slave
23      *                 with 7-bit address
24      ******************************************************************************/
25      UCB0CTL0 = (UCMST | UCMODE1 | UCMODE0 | UCSYNC);
26
27      /** control register 1 *******************************************************
28      * Set UCB0CTL1 -> clock source SMCLK | Software reset enable
29      *              -> Receiver | Acknowledge normally | No STOP generated |
30      *                 Do not generate START condition
31      ******************************************************************************/
32      UCB0CTL1 = (UCSSEL1 | UCSWRST);
33
34
35      /** bit rate control register 0/1 ********************************************
36      * -> Datasheet S. 476 <-
37      * Baud rate = 100kbps
38      * -> f_brclk = SMCLK = 0.5MHz
39      * -> UCBRX = 5
40      * f_bitclk = f_brclk / UCBRX = 100kHz,
41      ******************************************************************************/
42      //UCB0BR0 = ((uint16_t) SMCLK / UCB0BR_BIT_CLK); // low byte, UCB0RX = (UCB0R0 + UCB0R1 x 128)
43      UCB0BR0 = 0x0A;
44      UCB0BR1 = 0x00; // high byte
45
46      /** Own Address Register *****************************************************
47      * I2C own 7-bit address = 0x7B (123)
48      ******************************************************************************/
49      UCB0I2COA |= 0x007B;
50
51      /** Interrupt Enable Register ************************************************
52      * Interrupt disabled for Not-acknowledge, Start/Stop condition and
53      * Arbitration lost
54      ******************************************************************************/
55      UCB0I2CIE = 0x00;
56  }
57
58  void i2c_write(uint8_t slave_address, uint8_t data_length, uint8_t *data) {
59      uint8_t i;
60
61      UCB0CTL1 |= UCTR;                               // set transmitter-mode
62      UCB0I2CSA = slave_address;                      // set slave address
63      UCB0CTL1 &= ~UCSWRST;                           // unset SWRESET
64      UCB0CTL1 |= UCTXSTT;                            // send START-CON
65      UCB0TXBUF = data[0];                            // then write data
66
67      for(i = 1; i < data_length; i++) {
68          while(1) { // wait until ...
69              /*if((UCB0STAT & UCNACKIFG) == UCNACKIFG)      // NACK from slave
70              {
71                  UCB0CTL1 |= UCTXSTP;                    // then send STOP-CON
72                  UCB0STAT &= ~UCNACKIFG;                 // reset flag
73                  UCB0CTL1 |= UCSWRST;                    // set SWRESET
74                  break;
75              }*/
76
77              if((IFG2 & UCB0TXIFG) == UCB0TXIFG)         // data / start-con was send
78              {
79                  UCB0TXBUF = data[i];                    // then write data
80                  break;
81              }
82          }
83      }
84
85      while((IFG2 & UCB0TXIFG) != UCB0TXIFG);             // wait until data/start-con was send
```

```
86
87      UCB0CTL1 |= UCTXSTP;                                    // send STOP-COND
88      while((UCB0CTL1 & UCTXSTP) == UCTXSTP);                 // wait until STOP-con was send
89
90      UCB0CTL1 |= UCSWRST;                                    // set SWRESET
91  }
92
93  void i2c_read(uint8_t slave_address, uint8_t data_length, uint8_t *data) {
94      uint8_t i;
95
96      UCB0CTL1 &= ~UCTR;              // reset transmitter-mode
97      UCB0I2CSA = slave_address;      // set slave address of sensor
98      UCB0CTL1 &= ~UCSWRST;           // unset SWRESET
99      UCB0CTL1 |= UCTXSTT;            // send START-CON
100
101     if(data_length > 1)
102     {
103         for(i = 0; i < data_length; i++)
104         {
105             while(1)
106             {
107                 /*****************************************************************
108                 * IFG2 = Interrupt Flag Register 2
109                 *****************************************************************/
110                 if((IFG2 & UCB0RXIFG) == UCB0RXIFG) // data / start-con was send
111                 {
112                     data[i] = UCB0RXBUF;            // readout data
113                     if(i == data_length-2)          // if next byte will be the last one
114                         UCB0CTL1 |= UCTXSTP;        // send STOP-CON after next receive
115
116                     break;
117                 }
118             }
119         }
120     }
121     else
122     {
123         while((UCB0CTL1 & UCTXSTT) == UCTXSTT); // wait for acknowledge of slave,
124         UCB0CTL1 |= UCTXSTP; // then send STOP-COND immediately
125
126         data[0] = UCB0RXBUF; // readout data
127     }
128
129     while((UCB0CTL1 & UCTXSTP) == UCTXSTP); // wait until STOP-con was send
130     UCB0CTL1 |= UCSWRST; // set SWRESET
131 }
```

```
1   /*********************************************************************************
2   **     Description    : init.c
3   **     Hardware       : BATSEN ZS 3 v0.5
4   **     Date           : 09/04/2013
5   **     Last Update    : 02/08/2015
6   **     Author         : Nico Sassano
7   **     State          : Final state
8   *********************************************************************************/
9   #include "header\main.h"
10
11  extern volatile uint8_t brate_start;
12  extern volatile uint8_t brate;
13  extern volatile uint16_t upper_alarm_temp;
14  extern volatile uint16_t lower_alarm_temp;
15  extern volatile uint16_t sample_burst_buf[BURST_VALUES];
16  extern volatile uint8_t irq_alert;          // TMP102 Alarm
17
18  void init(void) {
19
20      TPS61201_PxDIR |= TPS61201_PIN;
21      TPS61201_ENABLE;
22
23      _EINT();     //global interrupts enable
24
25      // LEDs initialisieren
26      led_init();
27      i2c_init();
28      brate = brate_start;
29
30      // RF-switch
31      adg918_init();
32      adg918_wakeup();     // Connect the loop antenna with the wakeup circuit
33      adg918_transceiver();     // Connect the loop antenna with the wakeup circuit
34
35      // Balancing unit
36      balancing_init();
37      balancing_off();     // Balancing unit off
38
39      /**********************************
40      ** Timer konfigurieren
41      **********************************/
42      cc430_reset();
43      cc430_sleep();              // Power down state
44
45      // Configure packet received interrupt
46      CC430_END_OF_PKT_CLEAR_IRQ;
47      CC430_END_OF_PKT_IRQ_DISABLE;
48
49      // LF wakeup receiver
50      as3930_init();
51      as3930_preset_default();     // Reset
52      as3930_config_no_pattern(); // Wakeup upon LF carrier detection
53      as3930_clear_wakeup();       // Clear wakeup
54
55      // Alles LED leuchten 3 Sekunden beim start
56      led_on(LED_ALL);
57      delay_ms(300);
58      led_off(LED_ALL);
59  }
60
61  void init_for_sleep(void) {
62
63      // RF-switch
64      adg918_init();
65      adg918_wakeup();     // Connect the loop antenna with the wakeup circuit
66
67      balancing_off();     // Balancing unit off
68
69      // CC1101 transceiver
70      //cc430_init();
71      cc430_reset();
72      cc430_sleep();              // Power down state
73
74      // Configure packet received interrupt
75      CC430_END_OF_PKT_CLEAR_IRQ;
76      CC430_END_OF_PKT_IRQ_DISABLE;
77
78      // LF wakeup receiver
79      as3930_init();
80      as3930_preset_default();     // Reset
81      as3930_config_no_pattern(); // Wakeup upon LF carrier detection
82      as3930_clear_wakeup();       // Clear wakeup
83
84      // Configure Wake interrupt
85      AS3930_WAKE_CLEAR_IRQ;
```

```
86        AS3930_WAKE_IRQ_ENABLE;
87
88        // Global interrupt enable
89
90        _EINT();
91    }
```

```
1  /*******************************************************************************
2  **     Description    : isr.c
3  **     Hardware       : BATSEN ZS 3 v0.5
4  **     Date           : 09/04/2013
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     State          : Final state
8  *******************************************************************************/
9  #include "header\main.h"
10
11 extern volatile uint8_t wakeup_state;
12 extern volatile state_t state;
13 /*********************************************
14 ** CLK
15 *********************************************/
16 extern volatile uint8_t clk_set;
17
18 /*********************************************
19 ** RX
20 *********************************************/
21 extern volatile uint8_t rx_command;
22 extern volatile uint8_t rx_data_length;
23
24 /*********************************************
25 ** Balancing
26 *********************************************/
27 extern volatile uint16_t balancing_volt;  // Zielwert der Balancierung
28 extern volatile uint16_t balancing_time;
29 extern volatile uint8_t balanc_state;
30 extern volatile uint8_t temp_state;
31 extern volatile uint16_t upper_balanc_temp;
32 extern volatile uint16_t lower_balanc_temp;
33 extern volatile uint16_t upper_alarm_temp;
34 extern volatile uint16_t lower_alarm_temp;
35
36 /*********************************************
37 ** Data buffer
38 *********************************************/
39 uint16_t config_value;
40 extern volatile uint16_t sample_burst_buf[BURST_VALUES];
41 extern volatile uint16_t sample_buf_volt;                // The latest cell voltage sample
42 //extern volatile uint16_t sample_buf_temp;
43
44 /*********************************************
45 ** Kalibrierung
46 *********************************************/
47 extern volatile uint8_t cali_pos_offset_tmp102;
48 extern volatile uint8_t cali_neg_offset_tmp102;
49 extern volatile uint8_t cali_pos_offset_msp430;
50 extern volatile uint8_t cali_neg_offset_msp430;
51 extern volatile uint8_t cali_pos_offset_adc;
52 extern volatile uint8_t cali_neg_offset_adc;
53 extern volatile uint32_t clk_cali_counter;               // Counter for the clock calibration
54
55 /*********************************************
56 ** Burst Mode
57 *********************************************/
58 extern volatile uint8_t burst_freq;            // Burst Frequenz
59 extern volatile uint16_t burst_values;          // Anzahl der erwarteten  Burst Werte
60 extern volatile uint16_t burst_counter;        // Counter der Burst-Werte
61 extern volatile uint16_t burst_memory;          // Counter der Burst-Speichers
62 extern volatile uint8_t frame_number;
63 extern volatile uint8_t burst_frame_lenght;   // 50 -> 25 Werten
64 extern volatile uint8_t burst_error;
65 extern volatile uint8_t burst_error_flag;
66
67 /*********************************************
68 ** ADC Select
69 *********************************************/
70 extern volatile uint8_t  adc_mode;
71
72 ///*********************************************
73 //** Goertzel
74 //*********************************************/
75 extern volatile float Fs_goertzel;  // Burstfrequncy
76 extern volatile float f_goertzel;   // Signal frequency
77 extern volatile float m;
78
79 /*********************************************
80 ** AD5270
81 *********************************************/
82 extern volatile uint16_t ad5270_1_value;
83 extern volatile uint16_t ad5270_2_value;
84
85 /*********************************************
```

```
86  ** Interruptflags
87  ***********************************************/
88  extern volatile uint8_t irq_mode;               // PORT1 IRQ Status
89  extern volatile uint8_t irq_send_done_flag;     // CC1101 senden
90  extern volatile uint8_t irq_alert;              // TMP102 Alarm
91  extern volatile uint8_t irq_timera;             // Status TimerA
92  extern volatile uint8_t irq_timerb;             // Status TimerB
93
94
95  /**********************************************
96  ** Communication states
97  **********************************************/
98  extern volatile uint8_t command_recived;
99  extern volatile uint8_t wakeup_state;
100 extern volatile uint8_t brate_start;
101 extern volatile uint8_t brate;
102
103 extern volatile uint16_t delay_counter;
104
105 extern volatile uint8_t gdo0_state;
106 extern volatile uint8_t burst_cali_flag;
107
108 volatile uint32_t CC430_GDO0_OUT_PMAP = 0;
109
110 //*************************************************************************
111 // AD7691 Test mode
112 // Data declaration for thesting AD7691
113 // This can be deleted after testing the ADC
114 //*************************************************************************
115 //extern volatile uint8_t AD7691_index;
116 //extern volatile uint8_t AD7691_count;
117 //extern volatile uint32_t AD7691_data[2]; // 24-bit data storage
118 //extern volatile uint32_t AD7691_asynch_counter;
119 //extern volatile   uint32_t AD7691_values;
120 //extern volatile   uint32_t AD7691_frequency_time;
121 //extern volatile uint32_t AD7691_value;
122 extern volatile uint32_t AD7691_value_storage[1];
123
124
125 uint16_t index = 0;
126
127 #pragma vector=WDT_VECTOR
128 __interrupt void wdttimer(void)
129 {
130
131 }
132
133 /************************************************************************************************
134 ** CC1101 Radio Core Interrupts
135 ** See CC430F5137 CC430 Family User's Guide (Rev. E) -> slau259e.pdf table 25-5 on page 675
136 ************************************************************************************************/
137 #pragma vector=CC1101_VECTOR
138 __interrupt void CC1101_ISR(void) {
139   switch(__even_in_range(RF1AIV,32)) {   // Prioritizing Radio Core Interrupt
140
141
142     case RF1AIV_NONE: break;
143     /*********************************************************
144     * Case 2: RFIFG0 Based on GDO0 signal
145     * programmable using IOCFG0 (0x02) register of radio core.
146     *********************************************************/
147     case RF1AIV_RFIFG0: break;
148
149     /*********************************************************
150     * Case 4: RFIFG1 Based on GDO1 signal
151     * programmable using IOCFG1 (0x01) register of radio core.
152     *********************************************************/
153     case RF1AIV_RFIFG1: break;
154
155     /*********************************************************
156     * Case 6: RFIFG2 Based on GDO2 signal
157     * programmable using IOCFG2 (0x00) register of radio core.
158     *********************************************************/
159     case RF1AIV_RFIFG2:
160
161             /**************************************************
162              ** IRQ Burstmode
163              **************************************************/
164             // Interrupt disablen
165             CC430_GDO2_IRQ_DISABLE;
166             CC430_GDO2_IRQ_CLEAR;
167
168             TIMER_B_STOP;    // Timer stoppen
169             P1OUT &= ~BIT4; // OFF
170             if(IRQ_IS_CALI_BURST) {
```

```
                AD7691_CNV_SET_HIGH;
                TA0CTL |= MC_1;
                BURST_CALI_FLAG_SET;
            } else if(IRQ_IS_BURST) {

                if(ADC12_INT_IS_SELECT) {
                    // For 12 Bit ADC BEGIN ***************************************************************************

                    uint16_t adc_voltage = adc12_get_volt_sample(clk_set);
                    //sample_burst_buf[burst_counter] = adc_voltage;

                    uint8_t mod = burst_counter % 4;

                    // Hier werden die ADC-Werte Codiert, bzw. die Bits geschoben
                    if(mod == 0){
                        sample_burst_buf[burst_memory] = ((adc_voltage & 0x0FFF) << 4) & 0xFFF0;

                    }else if(mod == 1){
                        sample_burst_buf[burst_memory] |= ((adc_voltage & 0x0F00) >> 8) & 0x000F;
                        burst_memory++;

                        sample_burst_buf[burst_memory] = ((adc_voltage & 0x00FF) << 8) & 0xFF00;

                    }else if(mod == 2){
                        sample_burst_buf[burst_memory] |= ((adc_voltage & 0x0FF0) >> 4) & 0x00FF;
                        burst_memory++;

                        sample_burst_buf[burst_memory] = ((adc_voltage & 0x000F) << 12)& 0xF000;

                    }else if(mod == 3){
                        sample_burst_buf[burst_memory] |= ((adc_voltage & 0x0FFF) << 0) & 0x0FFF;
                        burst_memory++;
                    }

                    P1OUT |= BIT4; // On
                    burst_counter++;    // Burstwert hochzählen
                    // For 12 Bit ADC END *****************************************************************************

                }else if(AD7691_EXT_IS_SELECT) {
                    //*** For testing AD7691 BEGIN
                        ***************************************************************************

                    AD7691_CNV_SET_HIGH;         // start a new convertion

                    AD7691_SET_IRQ_FALLING_EDGE;
                    AD7691_BUSY_CLEAR_IRQ;
                    AD7691_BUSY_IRQ_ENABLE;

                    AD7691_CNV_SET_LOW;
                    // Now waiting for a AD7691 Interrupt
                    //*** For testing AD7691 END
                        *****************************************************************************
                }

                if(burst_freq >= BURST_FREQ_2000HZ) {
                    CC430_GDO2_IRQ_CLEAR;
                    if(burst_counter < burst_values)
                        CC430_GDO2_IRQ_ENABLE;
                    //Flag zurücksetzen
                    TIMER_B_FLAG_RESET;
                }

                    // Timer einstellen
                    timer_b_init_burst(burst_freq);

                    // Timer starten
                    TIMER_B_START_UP_MODE;
                }


            break;
        /************************************************************
        * Case 8: RFIFG3
        * Positive edge: RX FIFO filled or above the RX FIFO threshold.
        * Negative edge: RX FIFO drained below RX FIFO threshold.
        * This is equal to GDOx_CFG=0 on the CC1101 module
        ************************************************************/
        case RF1AIV_RFIFG3: break;

        /************************************************************
        * Case 10: RFIFG4
        * Positive edge: RX FIFO filled or above the RX FIFO threshold or end of packet is reached.
        * Negative edge: RX FIFO empty.
        * This is equal to GDOx_CFG=1 on the CC1101 module
        ************************************************************/
```

```
254         case RF1AIV_RFIFG4: break;
255
256         /**********************************************************
257         * Case 12: RFIFG5
258         * Positive edge: TX FIFO filled or above the TX FIFO threshold.
259         * Negative edge: TX FIFO below TX FIFO threshold.
260         * This is equal to GDOx_CFG=2 on the CC1101 module
261         **********************************************************/
262         case RF1AIV_RFIFG5: break;
263
264         /**********************************************************
265         * Case 14: RFIFG6
266         * Positive edge: TX FIFO full.
267         * Negative edge: TX FIFO below TX FIFO threshold.
268         * This is equal to GDOx_CFG=3 on the CC1101 module
269         **********************************************************/
270         case RF1AIV_RFIFG6: break;
271
272         /**********************************************************
273         * Case 16: RFIFG7
274         * Positive edge: RX FIFO overflowed.
275         * Negative edge: RX FIFO flushed.
276         * This is equal toGDOx_CFG=4 on the CC1101 module
277         **********************************************************/
278         case RF1AIV_RFIFG7: break;
279
280         /**********************************************************
281         * Case 18: RFIFG8
282         * Positive edge: TX FIFO underflowed.
283         * Negative edge: TX FIFO flushed.
284         * This is equal to GDOx_CFG=5 on the CC1101 module
285         **********************************************************/
286         case RF1AIV_RFIFG8: break;
287
288         /***************************************************************************************
289         * Case 20: RFIFG9
290         * Positive edge: Sync word sent or received.
291         * Negative edge: End of packet or in RX when optional address check
292         * fails or RX FIFO overflows or in TX when TX FIFO underflows.
293         * This is equal to GDOx_CFG=6 on the CC1101 module
294         ***************************************************************************************/
295         case RF1AIV_RFIFG9: // End of Packet IRQ Recieved
296
297             if(IRQ_IS_RX) {
298             /****************************************************
299             ** IRQ Packet empfangen
300             ****************************************************/
301                 //TIMER_A_0_CM_IRQ_DISABLE;
302                 _DINT();                    // Globale Interrups ausschalten
303
304                 volatile uint8_t bytes_in_fifo = 0;
305                 CC430_END_OF_PKT_IRQ_DISABLE;
306                // CC430_END_OF_PKT_CLEAR_IRQ;     // Seems to Auto-Clear on Read from RX-Fifo TODO : verifizieren
307
308                 rx_command = COMMAND_DOWNLINK_UNKWOWN;
309                 command_recived = 1;
310                 bytes_in_fifo = cc430_get_rxbytes();
311                 if(bytes_in_fifo >= 2)
312                     bytes_in_fifo -=2;        //Non-Zero Values from get_rxbytes are to great by 2 TODO Ursache ermitteln
313
314                 uint8_t rxbuf[64];
315                 // Check whether received data is valid
316                 if ((bytes_in_fifo == HEADER_LENGTH + rx_data_length)) {
317
318                     cc430_read_rx_fifo(rxbuf, HEADER_LENGTH + rx_data_length);
319                     rx_data_length = 0; // Datenlänge wieder zurücksetzen
320
321                     if (rxbuf[0] == BROADCAST || rxbuf[0] == ADDRESS_THIS_SENSOR) { // TODO Adresse müsste autmotisch geprüft
                            werden.
322
323                         led_on(LED_RX);           // Empfangs LED an
324
325                         rx_command = rxbuf[1];   // Empfangenes Kommando
326
327                         switch(rx_command) {
328                             /**********************************************
329                              ** Dekodierung des WAKEUP Kommandos
330                             ** Paketzusammenstellung:
331                             ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
332                             **********************************************/
333                             case COMMAND_WAKEUP: {
334                                 state = S_WAKEUP_RX;
335                                 wakeup_state = 1;
336                             }break;
337
```

```
338                 /***********************************************
339                 ** Dekodierung des WAKEUP_DONE Kommandos
340                 ** Paketzusammenstellung:
341                 ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
342                 ************************************************/
343                 case COMMAND_WAKEUP_DONE: {
344                     state = S_WAKEUP_DONE;
345                     if(!wakeup_state)
346                         rx_command = COMMAND_DOWNLINK_SLEEP;
347 
348                 }break;
349 
350                 /***********************************************
351                 ** Dekodierung des AD5270_1 Kommandos
352                 ** Paketzusammenstellung:
353                 ** ( Adr. ZS | Kommando | value 1 | value 2 )
354                 ************************************************/
355                 case COMMAND_AD5270_1: {
356                     ad5270_1_value = rxbuf[2];
357                     ad5270_2_value = rxbuf[3];
358                 }break;
359 
360                 /***********************************************
361                 ** Dekodierung des AD5270_2 Kommandos
362                 ** Paketzusammenstellung:
363                 ** ( Adr. ZS | Kommando | value 1 | value 2 )
364                 ************************************************/
365                 case COMMAND_AD5270_2: {
366                     ad5270_1_value = rxbuf[2];
367                     ad5270_2_value = rxbuf[3];
368                 }break;
369 
370                 /***********************************************
371                 ** Dekodierung des CONFIG_SET Kommandos
372                 ** Paketzusammenstellung:
373                 ** ( Adr. ZS | Kommando | Anzahl | 0x00 )
374                 ************************************************/
375                 case COMMAND_DOWNLINK_CONFIG_SET: {
376                     rx_data_length = rxbuf[2];  // Länge der nächsten Datensendung
377 
378                 }break;
379 
380                 /***********************************************
381                 ** Dekodierung des CONFIG Kommandos
382                 ** Paketzusammenstellung:
383                 **      0         1              2       3
384                 ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
385                 **     0          1       2          3       4          5       6          7       8          9
                                10
386                 ** ( Brate | oberer Bal. Wert | unterer Bal. Wert | oberer Alarmwert | unterer Alarmwert | Burst
                        Framelänge | ADC Auswahl)
387                 ************************************************/
388                 case COMMAND_DOWNLINK_CONFIG: {
389                     rx_data_length  = 0; // Datenlänge zurücksetzen
390                     uint16_t msb    = 0;
391                     uint16_t lsb    = 0;
392 
393                     // Übertragungsrate
394                     brate               = rxbuf[HEADER_LENGTH + 0];
395 
396                     // oberer Balancierungswert
397                     msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 1] & 0x0F)) << 8) & 0xFF00;
398                     lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 2] & 0xFF)) << 0) & 0x00FF;
399                     upper_balanc_temp   = msb | lsb;
400                     msb = 0;
401                     lsb = 0;
402 
403                     // unterer Balancierungswert
404                     msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 3] & 0x0F)) << 8) & 0xFF00;
405                     lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 4] & 0xFF)) << 0) & 0x00FF;
406                     lower_balanc_temp   = msb | lsb;
407                     msb = 0;
408                     lsb = 0;
409 
410                     // oberer Alarmwert
411                     msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 5] & 0x0F)) << 8) & 0xFF00;
412                     lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 6] & 0xFF)) << 0) & 0x00FF;
413                     upper_alarm_temp    = msb | lsb;
414                     msb = 0;
415                     lsb = 0;
416 
417                     // unterer Alarmwert
418                     msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 7] & 0x0F)) << 8) & 0xFF00;
419                     lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 8] & 0xFF)) << 0) & 0x00FF;
420                     lower_alarm_temp    = msb | lsb;
```

```
421
422                 // Burst Framelänge
423                 burst_frame_lenght  = rxbuf[HEADER_LENGTH + 9];
424
425
426             }break;
427
428             /*************************************************
429             ** Dekodierung des Balancierungsheader
430             ** Paketzusammenstellung:
431             ** ( Adr. ZS | Kommando | VOLT_MSB | VOLT_LSB )
432             *************************************************/
433             case COMMAND_DOWNLINK_BALANCING_ON: {
434                 uint16_t volt_msb = 0;
435                 uint16_t volt_lsb = 0;
436
437                 volt_msb = (((uint16_t)(rxbuf[3] & 0x0F)) << 8) & 0xFF00;
438                 volt_lsb = (((uint16_t)(rxbuf[4] & 0xFF)) << 0) & 0x00FF;
439
440                 balancing_volt = volt_msb | volt_lsb;
441                 balancing_volt = 2620;  //TEST
442
443             }break;
444
445             /*************************************************
446             ** Dekodierung des Burstheader
447             ** Paketzusammenstellung:
448             ** ( Adr. ZS | Kommando | Burst Freq | Anzahl Werte )
449             *************************************************/
450             case COMMAND_DOWNLINK_BURST_MODE: {
451                 burst_freq          = rxbuf[2];
452                 uint8_t burst_value = rxbuf[3];
453
454                switch(burst_value) {
455                    case BURST_VALUES_50:
456                        burst_values = 50;
457                    break;
458
459                    case BURST_VALUES_100:
460                        burst_values = 100;
461                    break;
462
463                    case BURST_VALUES_150:
464                        burst_values = 150;
465                    break;
466
467                    case BURST_VALUES_200:
468                        burst_values = 200;
469                    break;
470
471                    case BURST_VALUES_250:
472                        burst_values = 250;
473                    break;
474
475                    case BURST_VALUES_300:
476                        burst_values = 300;
477                    break;
478
479                    case BURST_VALUES_350:
480                        burst_values = 350;
481                    break;
482
483                    case BURST_VALUES_400:
484                        burst_values = 400;
485                    break;
486
487                    case BURST_VALUES_450:
488                        burst_values = 450;
489                    break;
490
491                    case BURST_VALUES_500:
492                        burst_values = 500;
493                    break;
494
495                    case BURST_VALUES_550:
496                        burst_values = 550;
497                    break;
498
499                    case BURST_VALUES_600:
500                        burst_values = 600;
501                    break;
502
503                    case BURST_VALUES_650:
504                        burst_values = 650;
505                    break;
```

```
506
507                 case BURST_VALUES_700:
508                     burst_values = 700;
509                 break;
510
511                 case BURST_VALUES_750:
512                     burst_values = 750;
513                 break;
514
515                 case BURST_VALUES_800:
516                     burst_values = 800;
517                 break;
518
519                 case BURST_VALUES_850:
520                     burst_values = 850;
521                 break;
522
523                 case BURST_VALUES_900:
524                     burst_values = 900;
525                 break;
526
527                 case BURST_VALUES_1000:
528                     burst_values = 1000;
529                 break;
530
531                 case BURST_VALUES_1500:
532                     burst_values = 1500;
533                 break;
534
535                 case BURST_VALUES_1900:
536                     burst_values = 1900;
537                 break;
538
539                 case BURST_VALUES_2000:
540                     burst_values = 2000;
541                 break;
542
543                 case BURST_VALUES_2500:
544                     burst_values = 2500;
545                 break;
546
547             }
548
549         }
550
551         /************************************************
552         ** Dekodierung des Burstheader
553         ** Paketzusammenstellung:
554         ** ( Adr. ZS | Kommando | Frame Nummer | 0x00 )
555         ************************************************/
556         case COMMAND_DOWNLINK_BURST_DATA_RX: {
557             frame_number      = rxbuf[2];
558
559         }break;
560
561         /************************************************
562         ** Dekodierung des Goertzelheader:
563         ** Paketzusammenstellung:
564         ** ( Adr. ZS | Kommando | Numb of pa. | 0x00 )
565         ************************************************/
566         case COMMAND_GOERTZEL_NUMB_PERI: {
567             uint16_t value = 0;
568
569             // Dekodierung der Signalfrequenz
570             value = (rxbuf[2]);
571             m = (float) value;
572
573         }break;
574
575         /************************************************
576         ** Dekodierung des Goertzelheader:
577         ** Paketzusammenstellung:
578         ** ( Adr. ZS | Kommando | Siganlfreq. | Siganlfreq )
579         ************************************************/
580         case COMMAND_GOERTZEL_STIMULI_FREQ: {
581             uint16_t value = 0;
582
583             // Dekodierung der Signalfrequenz
584             value = 100*(rxbuf[2]);
585             value = value + (rxbuf[3]);
586             f_goertzel = (float) value;
587
588         }break;
589
590         /************************************************
```

```
591                         ** Dekodierung des Goertzelheader:
592                         ** Paketzusammenstellung:
593                         ** ( Adr. ZS | Kommando | Siganlfreq. | 0x00 )
594                         ***********************************************/
595                         case COMMAND_GOERTZEL_STIMULI_FREQ2: {
596                             uint16_t value = 0;
597 
598                             // Dekodierung der Signalfrequenz
599                             value = (rxbuf[2]);
600                             f_goertzel = f_goertzel + 0.1* (float) value;
601 
602                         }break;
603 
604                         /***********************************************
605                         ** Dekodierung des Goertzelheader:
606                         ** Paketzusammenstellung:
607                         ** ( Adr. ZS | Kommando | Siganlfreq. | Siganlfreq )
608                         ***********************************************/
609                         case COMMAND_GOERTZEL_FREQ: {
610                             uint16_t value = 0;
611 
612                             // Dekodierung der Signalfrequenz
613                             value = 100*(rxbuf[2]);
614                             value = value + (rxbuf[3]);
615                             Fs_goertzel = (float) value;
616 
617                         }break;
618 
619                         /***********************************************
620                         ** Dekodierung des Kalibrierungsheader: TMP102
621                         ** Paketzusammenstellung:
622                         ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
623                         ***********************************************/
624                         case COMMAND_DOWNLINK_CALIBRATION_TMP102: {
625                             cali_pos_offset_tmp102 = rxbuf[2];
626                             cali_neg_offset_tmp102 = rxbuf[3];
627                         }break;
628 
629                         /***********************************************
630                         ** Dekodierung des Kalibrierungsheader: MSP430
631                         ** Paketzusammenstellung:
632                         ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
633                         ***********************************************/
634                         case COMMAND_DOWNLINK_CALIBRATION_MSP430: {
635                             cali_pos_offset_msp430 = rxbuf[2];
636                             cali_neg_offset_msp430 = rxbuf[3];
637                         }break;
638 
639                         /***********************************************
640                         ** Dekodierung des Kalibrierungsheader: ADC
641                         ** Paketzusammenstellung:
642                         ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
643                         ***********************************************/
644                         case COMMAND_DOWNLINK_CALIBRATION_ADC: {
645                             cali_pos_offset_adc = rxbuf[2];
646                             cali_neg_offset_adc = rxbuf[3];
647                         }break;
648 
649 
650                         default: break;
651                     }
652 
653                 }
654                 led_off(LED_RX);
655             } else {
656                 cc430_read_rx_fifo(rxbuf, bytes_in_fifo); //TODO CHeck. Dummy read to reset Rx Flag
657             }
658 
659 
660             //TIMER_A_0_CM_IRQ_ENABLE;
661             _EINT();                // Globale Interrups einschalten
662 
663         } else if(IRQ_IS_TX) {
664             CC430_END_OF_PKT_IRQ_DISABLE;
665             //CC430_END_OF_PKT_CLEAR_IRQ;   Seems to Auto-Clear  TODO : Herausfinden, bei was Auto-Clear
666             irq_send_done_flag = 1;
667         }
668         break;
669     /*******************************************************
670     * Case 22: RFIFG10
671     * Positive edge: Packet received with CRC OK.
672     * Negative edge: First byte read from RX FIFO.
673     * This is equal to GDOx_CFG=7 on the CC1101 module
674     *******************************************************/
675     case RF1AIV_RFIFG10: break;
```

```
676
677     /*********************************************************
678     * Case 24: RFIFG11
679     * Positive edge: Preamble quality reached (PQI) is above programmed PQT value.
680     * Negative edge: (LPW)
681     * This is equal to GDOx_CFG=8 on the CC1101 module
682     *********************************************************/
683     case RF1AIV_RFIFG11: break;
684
685     /*********************************************************
686     * Case 26: RFIFG12
687     * Positive edge: Clear channel assessment when RSSI level is below threshold (dependent on the current RFIFG12
            CCA_MODE setting).
688     * Negative edge: RSSI level is above threshold.
689     * This is equal to GDOx_CFG=9 on the CC1101 module
690     *********************************************************/
691     case RF1AIV_RFIFG12: break;
692
693     /*********************************************************
694     * Case 28: RFIFG13
695     * Positive edge: Carrier sense. RSSI level is above threshold.
696     * Negative edge: RSSI level is below threshold.
697     * This is equal to GDOx_CFG=14 on the CC1101 module
698     *********************************************************/
699     case RF1AIV_RFIFG13: break;
700
701     /*********************************************************
702     * Case 30: RFIFG14
703     * Positive edge: WOR event 0
704     * Negative edge: WOR event 0 + 1 ACLK.
705     * This is equal to GDOx_CFG=36 on the CC1101 module
706     *********************************************************/
707     case RF1AIV_RFIFG14: break;
708
709     /*********************************************************
710     * Case 32: RFIFG15
711     * Positive edge: WOR event 1
712     * Negative edge: RF oscillator stable or next WOR event0 triggered.
713     * This is equal to Equal to GDOx_CFG=37 on the CC1101 module
714     *********************************************************/
715     case RF1AIV_RFIFG15: break;
716   }
717 }
718
719
720 /*********************************************************************
721 ** IRQ Port1
722 *********************************************************************/
723 #pragma vector=PORT1_VECTOR
724 __interrupt void PORT1_ISR(void) {
725     switch(__even_in_range(P1IV,16)) {
726         case P1IV_NONE: break;                // No Interrupt pending
727         /****************************************************/
728         case P1IV_P1IFG0: break;              // P1IV P1IFG.0
729         /****************************************************/
730         case P1IV_P1IFG1:                     // P1IV P1IFG.1
731             // Temp IRQ Empfangeln
732
733             if(IRQ_ALERT_IS_LOW) {            // Anfrage Alarm auslösen
734
735                 TMP102_ALERT_IRQ_DISABLE;     // Interrupt stoppen
736                 TMP102_ALERT_CLEAR_IRQ;
737
738                 //led_off(LED_AWAKE);
739                 IRQ_ALERT_SET_HIGH;           // Alarm setzen
740
741                 TMP102_SET_IRQ_RISING_EDGE;  // IRQ auf steigende Flnke setzen
742                 TMP102_ALERT_CLEAR_IRQ;
743                 TMP102_ALERT_IRQ_ENABLE;
744
745             }else if(IRQ_ALERT_IS_HIGH) {    // Anfrage Ararm zurücksetzen
746
747                 TMP102_ALERT_IRQ_DISABLE;     // Interrupt stoppen
748                 TMP102_ALERT_CLEAR_IRQ;
749
750                 //led_on(LED_AWAKE);
751                 IRQ_ALERT_SET_LOW;            // Alarm löschen
752                 TMP102_SET_IRQ_FALLING_EDGE; // IRQ auf fallende Flanke setzen
753
754                 TMP102_ALERT_CLEAR_IRQ;
755                 TMP102_ALERT_IRQ_ENABLE;
756             }break;
757         /****************************************************/
758         case P1IV_P1IFG2: break;              // P1IV P1IFG.2
759         /****************************************************/
```

```
760          case P1IV_P1IFG3: break;              // P1IV P1IFG.3
761          /*****************************************************/
762          case P1IV_P1IFG4: break;              // P1IV P1IFG.4
763          /*****************************************************/
764          case P1IV_P1IFG5: break;              // P1IV P1IFG.5
765          /*****************************************************/
766          case P1IV_P1IFG6: break;              // P1IV P1IFG.6
767          /*****************************************************/
768          case P1IV_P1IFG7: break;              // P1IV P1IFG.7
769      }
770  }
771
772  /*************************************************************************
773  ** IRQ Port2
774  *************************************************************************/
775  #pragma vector=PORT2_VECTOR
776  __interrupt void PORT2_ISR(void) {
777      switch(__even_in_range(P2IV,16)) {
778          case P2IV_NONE:          break; // No Interrupt
779          /*****************************************************/
780          case P2IV_P2IFG0:     // P2IV P2IFG.0
781              /*****************************************************
782               ** IRQ AD7691 testing
783               ** Readout the sampling date from the AD7691
784               ** The ADC values are stored in an 16 Bit array!!
785               *****************************************************/
786              AD7691_BUSY_IRQ_DISABLE;
787              AD7691_BUSY_CLEAR_IRQ;
788
789              sample_burst_buf[burst_counter] = 0;
790
791              UCA0TXBUF = 0x00;                 // Send address and register
792              while (UCA0STAT & UCBUSY);        // Wait for TX to finish
793              sample_burst_buf[burst_counter] |= (UCA0RXBUF & 0xFF);// Read data from buffer
794              sample_burst_buf[burst_counter] = sample_burst_buf[burst_counter] << 8;// Read data from buffer
795              UCA0TXBUF = 0x00;                 // Write dummy Byte
796              while (UCA0STAT & UCBUSY);        // Wait for TX complete
797              sample_burst_buf[burst_counter] |= (UCA0RXBUF & 0xFF) << 0;// Read data from buffer
798
799              UCA0TXBUF = 0x00;                 // Write dummy Byte
800              while (UCA0STAT & UCBUSY);        // Wait for TX complete
801
802              burst_counter++;
803
804              break;
805          /*****************************************************/
806          case P2IV_P2IFG1:    break; // P2IV P2IFG.1
807          /*****************************************************/
808          case P2IV_P2IFG2:    break; // P2IV P2IFG.2
809          /*****************************************************/
810          case P2IV_P2IFG3:    break;      // P2IV P2IFG.3
811          /*****************************************************/
812          case P2IV_P2IFG4:    break; // P2IV P2IFG.4
813          /*****************************************************/
814          case P2IV_P2IFG5:            // P2IV P2IFG.5
815              /*****************************************************
816              ** IRQ Wakeup empfangen
817              *****************************************************/
818              // Exit Low Power Mode 4
819              EXIT_LPM4;
820
821              // Disable and clear the wake interrupt as the sensor is awake now
822              AS3930_WAKE_IRQ_DISABLE;
823              AS3930_WAKE_CLEAR_IRQ;
824
825              //TMP102_SET_IRQ_FALLING_EDGE;
826              //TMP102_ALERT_IRQ_ENABLE;
827              //TMP102_ALERT_CLEAR_IRQ;
828
829              // Turn on the red LED
830              led_on(LED_AWAKE);
831              //TIMER_B_START;
832
833              // Change to RX state
834              adg918_transceiver();             // Connect the loop antenne with the transceiver
835              adc12_volt_init(clk_set,1);         // Initialize ADC
836
837              state = S_WAKEUP;
838
839              rx_data_length = 0;               // Standart Datenlänge empangen
840              cc430_set_rx(brate_start);
841              cc430_rx(HEADER_LENGTH + rx_data_length);
842
843              break;
844          /*****************************************************/
```

```
845         case P2IV_P2IFG6:    break; // P2IV P2IFG.6
846         /**************************************************/
847         case P2IV_P2IFG7:    break; // P2IV P2IFG.7
848       }
849 }
850
851 /**************************************************************************
852 ** IRQ Timer0 A0
853 ** TImer for the balancing function
854 **************************************************************************/
855 #pragma vector=TIMER0_A0_VECTOR
856 __interrupt void TIMER0_A0_ISR(void) {
857
858     if(IRQ_TIMERA_IS_DELAY) {
859         __bic_SR_register_on_exit(LPM4_bits);
860     }
861
862     if (IRQ_TIMERA_IS_BALANCING) {
863
864         // Keine nested Interrupt
865         CC430_END_OF_PKT_IRQ_DISABLE;
866         CC430_GDO2_IRQ_DISABLE;
867
868         balancing_time++;
869
870         // 10min Balancieren -> 10min * 60sek *2 = 1200
871         if(balancing_time < 1200) {
872
873             uint16_t actual_volt = 0x0000;
874             uint16_t actual_temp = 0x0000;
875
876             // Aktuelle Werte holen
877             actual_volt = adc12_get_volt_sample(clk_set);
878             actual_temp = temp_sensor_get_temp();
879
880             sample_buf_volt = actual_volt;
881
882             // Temperaturkontrolle
883             if (TEMP_IS_NORMAL) {                // Temperatur ist Normal
884                 if (actual_temp > upper_balanc_temp)    // Ist aktuelle Temp. zu hoch?
885                     TEMP_SET_HIGH;
886             } else if(TEMP_IS_HIGH) {
887                 if (actual_temp < lower_balanc_temp)    // Ist aktuelle Temp. ok?
888                     TEMP_SET_NORMAL;
889             }
890
891             // Spannungskontrolle
892             if (actual_volt > balancing_volt) {
893                 if (TEMP_IS_NORMAL) {
894                     balancing_on();
895                 } else {
896                     balancing_off();
897                 }
898             } else { // Stop Balancing
899                 TIMER_A_STOP;
900                 balancing_off();
901                 balanc_state = OFF;
902                 IRQ_TIMERA_UNSET_BALANCING;
903                 TIMER_A_0_CM_IRQ_DISABLE;
904                 TIMER_A_1_CM_IRQ_DISABLE;
905             }
906         } else {
907             TIMER_A_STOP;
908             balancing_off();
909             balanc_state = OFF;
910             IRQ_TIMERA_UNSET_BALANCING;
911             TIMER_A_0_CM_IRQ_DISABLE;
912             TIMER_A_1_CM_IRQ_DISABLE;
913         }
914
915         if(balanc_state == OFF) {
916             TIMER_A_0_CM_IRQ_DISABLE;
917             TIMER_A_1_CM_IRQ_DISABLE;
918         }
919
920         CC430_END_OF_PKT_IRQ_ENABLE;
921         CC430_GDO2_IRQ_ENABLE;
922     }
923 }
924
925 /**************************************************************************
926 ** IRQ Timer0 A1
927 ** Actually unused
928 **************************************************************************/
929 #pragma vector=TIMER0_A1_VECTOR
```

```
930   __interrupt void TIMER0_A1_ISR(void) {
931     switch(__even_in_range(TA0IV,14)) {
932       case TA0IV_NONE: break;       // No Interrupt pending
933       /****************************************************/
934       case TA0IV_TACCR1: break;     // TA0CCR1_CCIFG
935       /****************************************************/
936       case TA0IV_TACCR2: break;     // TA0CCR2_CCIFG
937       /****************************************************/
938       case TA0IV_TACCR3: break;     // TA0CCR3_CCIFG
939       /****************************************************/
940       case TA0IV_TACCR4: break;     // TA0CCR4_CCIFG
941       /****************************************************/
942       case TA0IV_5: break;          // Reserved
943       /****************************************************/
944       case TA0IV_6: break;          // Reserved
945       /****************************************************/
946       case TA0IV_TAIFG: break;      // TA0IFG Timer Overflow
947     }
948   }
949
950   /*********************************************************************
951   ** IRQ Timer1 A0
952   ** This timer close the time slice for the burst measurement
953   *********************************************************************/
954   #pragma vector=TIMER1_A0_VECTOR
955   __interrupt void TIMER1_A0_ISR(void) {
956
957       //TA1CCR0 CCIFG
958       if(IRQ_TIMERB_IS_BURST) {
959           if(burst_counter >= burst_values) { //Sind alles Werte durchgekommen
960
961               CC430_GDO2_IRQ_DISABLE;
962               CC430_GDO2_IRQ_CLEAR;
963
964               TIMER_B_STOP;
965               TIMER_B_0_CM_IRQ_DISABLE;
966               TIMER_B_1_CM_IRQ_DISABLE;
967
968               burst_error = 0;     // Burst error zurücksetzen
969
970               //Flag zurücksetzen
971               TIMER_B_FLAG_RESET;
972
973               IRQ_TIMERB_UNSET_BURST;
974               rx_command = COMMAND_BACK_FROM_BURST;
975               command_recived = 1;
976
977           } else { //Fehler wird dedektiert
978               BURST_ERROR_FLAG_SET;          // Fehlerflag setzen
979
980               // bei hohen Frequenzen gibt es keine Fensterung mehr, Interrupt bleibt freigeschalten
981               if(burst_freq >= BURST_FREQ_2000HZ) {
982                   CC430_GDO2_IRQ_CLEAR;
983                   CC430_GDO2_IRQ_ENABLE;
984
985                   //Flag zurücksetzen
986                   TIMER_B_FLAG_RESET;
987               }else{
988                   // Close the time slice
989                   CC430_GDO2_IRQ_DISABLE;
990                   CC430_GDO2_IRQ_CLEAR;
991               }
992
993               // Timer einstellen
994               timer_b_init_burst(burst_freq);
995
996               // Timer starten
997               TIMER_B_START_UP_MODE;
998
999
1000              //sample_burst_buf[burst_counter] = 0xFFFF;   // Fehlerwert
1001              burst_counter++;                             // Burstcounter hochzählen
1002              burst_error++;                               // Burstfehler hochzählen
1003          }
1004
1005      /********************************
1006      * Counter for the delay function
1007      ********************************/
1008      } else if(IRQ_TIMERB_IS_DELAY) {
1009          delay_counter++;
1010      } else if(IRQ_TIMERB_IS_CLOCK_CALI) {   // Interrupt for the clock calibration
1011          clk_cali_counter++;
1012      }
1013  }
1014
```

```
1015
1016  /*************************************************************************
1017  ** IRQ Timer1 A1
1018  ** TACCR1 open the time slice for the burst measurement
1019  *************************************************************************/
1020  #pragma vector=TIMER1_A1_VECTOR
1021  __interrupt void TIMER1_A1_ISR(void) {
1022    switch(__even_in_range(TA1IV,14)) {
1023
1024      case TA1IV_NONE:     break;  //  No Interrupt pending
1025      /*****************************************************/
1026      case TA1IV_TACCR1:           // TA1CCR1_CCIFG
1027          // 528 Zeit für ISR abarbeitung bis Timer startet
1028          // test = TBR + 528;
1029          if(IRQ_TIMERB_IS_BURST) {
1030              // Interrupt enabeln
1031              CC430_GDO2_IRQ_CLEAR;
1032  //          AD7691_CS_DISABLE;  // AD7691 testing
1033              if(burst_counter < burst_values)
1034                  CC430_GDO2_IRQ_ENABLE;
1035
1036              //Flag zurücksetzen
1037              TIMER_B_FLAG_RESET;
1038          } break;
1039      /*****************************************************/
1040      case TA1IV_TACCR2:  break;  // TA1CCR2_CCIFG
1041      /*****************************************************/
1042      case TA1IV_3:       break;  // Reserved
1043      /*****************************************************/
1044      case TA1IV_4:       break;  // Reserved
1045      /*****************************************************/
1046      case TA1IV_5:       break;  // Reserved
1047      /*****************************************************/
1048      case TA1IV_6:       break;  // Reserved
1049      /*****************************************************/
1050      case TA1IV_TAIFG:   break;  // TA1IFG Timer Overflow
1051    }
1052  }
```

```
1   /*******************************************************************************
2   **      Description     : led.c
3   **      Hardware        : BATSEN ZS 3 v0.5
4   **      Date            : 09/12/2014
5   **      Last Update     : 02/08/2015
6   **      Author          : Nico Sassano
7   **      State           : Final state
8   *******************************************************************************/
9   #include "header\main.h"
10
11  void led_init (void) {
12          LED_RED_PxDIR           |= LED_RED_PIN;
13          LED_YELLOW_PxDIR        |= LED_YELLOW_PIN;
14          LED_GREEN_PxDIR         |= LED_GREEN_PIN;
15          led_off(LED_ALL);
16      }
17
18  void led_on (LED_t led) {
19      #ifdef ENABLE_LEDS
20      switch( led ) {
21          case LED_AWAKE:         LED_RED_PxOUT           |= LED_RED_PIN;
22                                  break;
23
24          case LED_TX:            LED_YELLOW_PxOUT        |= LED_YELLOW_PIN;
25                                  break;
26
27          case LED_RX:            LED_GREEN_PxOUT         |= LED_GREEN_PIN;
28                                  break;
29
30          case LED_ALL:           led_on(LED_AWAKE);
31                                  led_on(LED_TX);
32                                  led_on(LED_RX);
33                                  break;
34
35          default :               break;
36      }
37  # endif /* ENABLE_LEDS */
38   }
39
40  void led_off (LED_t led) {
41
42      switch (led) {
43          case LED_AWAKE:         LED_RED_PxOUT           &= ~LED_RED_PIN;
44                                  break;
45
46          case LED_TX:            LED_YELLOW_PxOUT        &= ~LED_YELLOW_PIN;
47                                  break;
48
49          case LED_RX:            LED_GREEN_PxOUT         &= ~LED_GREEN_PIN;
50                                  break;
51
52          case LED_ALL:           led_off(LED_AWAKE);
53                                  led_off(LED_TX);
54                                  led_off(LED_RX);
55                                  break;
56
57          default :               break;
58      }
59  }
60
61  void led_toggle (LED_t led) {
62      #ifdef ENABLE_LEDS
63
64      switch(led) {
65          case LED_AWAKE:         LED_RED_PxOUT           ^= LED_RED_PIN;
66                                  break;
67
68          case LED_TX:            LED_YELLOW_PxOUT        ^= LED_YELLOW_PIN;
69                                  break;
70          case LED_RX:            LED_GREEN_PxOUT         ^= LED_GREEN_PIN;
71                                  break;
72
73          case LED_ALL:           led_toggle(LED_AWAKE);
74                                  led_toggle(LED_TX);
75                                  led_toggle(LED_RX);
76                                  break;
77
78          default :               break;
79      }
80  # endif /* ENABLE_LEDS */
81   }
```

```
1  /*********************************************************************************
2  **     Description    : main.c
3  **     Hardware       : BATSEN ZS 3 v0.5
4  **     Date           : 06/03/2013
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     State          : Final state
8  *********************************************************************************/
9  #include "header\main.h"
10
11  volatile uint8_t state = 0;
12
13  //*****************************************************************************
14  // System settings and clock rate
15  //*****************************************************************************
16  uint8_t volatile clk_set                  = 1;
17  uint16_t sw_version                       = 320;
18  uint32_t dco_value                        = DCO_MP_16MHZ;
19
20  /*********************************************
21  ** RX
22  *********************************************/
23  volatile uint8_t rx_command               = COMMAND_WAIT;
24  volatile uint8_t rx_data_length           = 0;
25  volatile uint8_t tx_data_length           = 0;
26
27  /*********************************************
28  ** TX
29  *********************************************/
30  uint8_t packet[64]                        = {0};  // TX packet
31
32  /*********************************************
33  ** Balancing
34  *********************************************/
35  volatile uint16_t balancing_volt          = 0x500;  // Zielwert der Balancierung
36  volatile uint16_t balancing_time          = 0x000;
37  volatile uint8_t balanc_state             = OFF;
38  volatile uint8_t temp_state               = TEMP_NORMAL;
39  volatile uint16_t upper_balanc_temp       = 0x250;
40  volatile uint16_t lower_balanc_temp       = 0x200;
41  volatile uint16_t upper_alarm_temp        = 0x1A00;
42  volatile uint16_t lower_alarm_temp        = 0x1900;
43
44  /*********************************************
45  ** Data buffer
46  *********************************************/
47  uint16_t config_value;
48  volatile uint16_t sample_burst_buf[BURST_VALUES];
49  volatile uint16_t sample_buf_volt;            // The latest cell voltage sample
50  volatile uint16_t sample_buf_temp;
51  volatile uint16_t sample_buf_temp_msp;
52
53  /*********************************************
54  ** Burst Mode
55  *********************************************/
56  volatile uint8_t burst_freq               = 0;    // Burst Frequenz
57  volatile uint16_t burst_values            = 0;    // Anzahl der erwarteten Burst Werte
58  volatile uint16_t burst_counter           = 0;    // Counter der Burst-Werte
59  volatile uint16_t burst_memory            = 0;    // Counter der Burst-Speichers
60  volatile uint8_t burst_frame_counter      = 0;
61  volatile uint8_t burst_frame_lenght       = 48;   // 50 -> 25 Werten
62  volatile uint8_t frame_number             = 0;
63  volatile uint8_t burst_error              = 0;
64  volatile uint8_t burst_error_flag         = 0;
65
66  /*********************************************
67  ** Goertzel
68  *********************************************/
69  volatile float Fs_goertzel                = 0.0;  // Burstfrequncy
70  volatile float f_goertzel                 = 0.0;  // Signal frequency
71  volatile float m                          = 0.0;
72  //volatile float burst_lenght                 = 0.0;
73  float real;
74  float imag;
75  float bn_0 = 0;
76  float bn_1 = 0;
77  float bn_2 = 0;
78  float N_float;
79  float omega;
80  float sine;
81  float cosine;
82  float coeff;
83
84  float Np;
85
```

```
86   float f_gain = 0.0;
87   float f_voltage = 0.0;
88
89   uint16_t goertzel_index = 0;
90
91
92   typedef union {
93       float float_data;
94       unsigned char hex_data[4];
95   } FB;
96
97   FB real_char;
98   FB imag_char;
99
100  /**********************************************
101  ** AD5270
102  ***********************************************/
103  volatile uint16_t ad5270_1_value;
104  volatile uint16_t ad5270_2_value;
105
106  /**********************************************
107  ** Kalibrierung
108  ***********************************************/
109  volatile uint8_t cali_pos_offset_tmp102 = 0;
110  volatile uint8_t cali_neg_offset_tmp102 = 0;
111  volatile uint8_t cali_pos_offset_msp430 = 0;
112  volatile uint8_t cali_neg_offset_msp430 = 0;
113  volatile uint8_t cali_pos_offset_adc    = 0;
114  volatile uint8_t cali_neg_offset_adc    = 0;
115  volatile uint32_t clk_cali_counter      = 0;    // Counter for the clock calibration
116
117  /**********************************************
118  ** ADC Select
119  ***********************************************/
120  volatile uint8_t  adc_mode = ADC12_INT_SELECT;
121
122  /**********************************************
123  ** Interruptflags
124  ***********************************************/
125  volatile uint8_t irq_mode              = TX_MODE;  // PORT1 IRQ Status
126  volatile uint8_t irq_send_done_flag    = 0;        // cc430 senden
127  volatile uint8_t irq_alert             = 0;        // TMP102 Alarm
128  volatile uint8_t irq_timera            = 0;        // Status TimerA
129  volatile uint8_t irq_timerb            = 0;        // Status TimerB
130  volatile uint8_t cali_flag             = 0;        // Burst Cali flag
131  volatile uint8_t irq_cali_flag         = 0;        // Flag for burat calibration
132
133  /**********************************************
134  ** Communication states
135  **********************************************/
136  volatile uint8_t command_recived       = 0;
137  volatile uint8_t wakeup_state          = 0x00;
138
139  volatile uint8_t brate_start           = 100;
140  volatile uint8_t brate                 = 100;
141
142  volatile uint16_t delay_counter        = 0;
143
144  volatile uint8_t gdo0_state = 0x00;
145  volatile uint8_t burst_cali_flag       = 0;
146
147  //****************************************************************************
148  // Preprocessing
149  //****************************************************************************
150  volatile uint16_t gain = 0x0000;
151  volatile uint16_t offset = 0x03FF;
152
153  void main(void) {
154      WDTCTL = WDTPW+WDTHOLD;                    // Stop watchdog timer
155  //    WDTCTL = WDT_MDLY_32;              // WDT as interval timer (period 0,5 ms)
156  //    SFRIE1|=WDTIE;                           //enable the interrupt
157
158
159      //*********************************************************
160      //P5SEL |= BIT5;
161      //P5DIR |= BIT5;
162
163      P1DIR |= BIT4;               // Testport
164      P1OUT &= ~BIT4; // Off
165
166      /*******************************************************
167       ** Initialisierung
168       *******************************************************/
169      init();
170
```

```
171        // Disable and clear the wake interrupt as the sensor is awake now
172        AS3930_WAKE_IRQ_DISABLE;
173        AS3930_WAKE_CLEAR_IRQ;
174
175        // Configure packet received interrupt
176        CC430_END_OF_PKT_CLEAR_IRQ;
177        CC430_END_OF_PKT_IRQ_ENABLE;
178
179        // Turn on the red LED
180        led_on(LED_AWAKE);
181
182        // Change to RX state
183        adg918_transceiver();          // Connect the loop antenne with the transceiver
184        adc12_volt_init(clk_set,1);           // Initialize ADC
185
186        state = S_WAKEUP;
187
188        rx_data_length = 0;            // Standart Datenlänge empangen
189        cc430_set_rx(brate_start);
190        cc430_rx(HEADER_LENGTH + rx_data_length);
191
192        write_AD5270_1(0x3FF);
193        write_AD5270_2(0x000);
194        SFRIE1|=WDTIE;                        //enable the interrupt
195
196        //tx_carrier();
197
198        /*********************************************************
199        ** Endless loop
200        *********************************************************/
201        while(1) {
202            if(command_recived > 0) {    // Kommando wurde Empfangen
203                // Determine what the base station wants this sensor to do now
204                switch(rx_command) {
205                    /*********************************************************
206                    ** Unbekanntes Kommando empfangen
207                    *********************************************************/
208                    case COMMAND_DOWNLINK_UNKWOWN: {
209
210                        cc430_rx(HEADER_LENGTH + rx_data_length);
211                    } break;
212
213                    /*********************************************************
214                    ** Kommando zum WAKEUP empfangen
215                    *********************************************************/
216                    case COMMAND_WAKEUP: {
217                        state = S_WAKEUP_RX;
218
219                        cc430_rx(HEADER_LENGTH + rx_data_length);
220                    } break;
221                    /*********************************************************
222                    ** Kommando WAKUP DONE empfangen
223                    *********************************************************/
224                    case COMMAND_WAKEUP_DONE: {
225                        state = S_WAKEUP_DONE;
226
227                        cc430_rx(HEADER_LENGTH + rx_data_length);
228                    } break;
229                    /*********************************************************
230                    ** Set the first AD5270 1
231                    *********************************************************/
232                    case COMMAND_AD5270_1: {
233                        uint16_t value = 0;
234
235                        value = (ad5270_1_value << 8);
236                        value|= (ad5270_2_value << 0);
237
238                        write_AD5270_1(value);
239
240                        offset = value;
241
242                        cc430_rx(HEADER_LENGTH + rx_data_length);
243
244                    } break;
245
246                    /*********************************************************
247                    ** Set the second AD5270 2
248                    *********************************************************/
249                    case COMMAND_AD5270_2: {
250                        uint16_t value = 0;
251
252                        value = (ad5270_1_value << 8);
253                        value|= (ad5270_2_value << 0);
254
255                        write_AD5270_2(value);
```

```
256
257                         gain = value;
258
259                         cc430_rx(HEADER_LENGTH + rx_data_length);
260
261                     } break;
262
263                     /*********************************************************
264                     ** Senden des AWAKE Test
265                     *********************************************************/
266                     case COMMAND_DOWNLINK_IS_AWAKE: {
267
268 //                      uint8_t packet[HEADER_LENGTH] = {0};
269
270                         packet[0] = ADDRESS_BASE_STATION;
271                         packet[1] = ADDRESS_THIS_SENSOR;
272                         packet[2] = 0x01;
273                         packet[3] = 0x00;
274
275
276                         cc430_tx(HEADER_LENGTH, packet);
277
278
279                         cc430_rx(HEADER_LENGTH + rx_data_length);
280
281                     } break;
282
283                     /*********************************************************
284                     ** Einstellung zum Empfang von Konfigurationen
285                     *********************************************************/
286                     case COMMAND_DOWNLINK_CONFIG_SET: {
287
288                         cc430_rx(HEADER_LENGTH + rx_data_length);
289                     } break;
290
291                     /*********************************************************
292                     ** Konfiguration empfangen, danach neu Konfigurieren
293                     *********************************************************/
294                     case COMMAND_DOWNLINK_CONFIG: {
295
296                         /********************************
297                         ** Temperatursensor konfigurieren
298                         ********************************/
299                         temp_sensor_set_alert(lower_alarm_temp, upper_alarm_temp);
300
301                         cc430_rx(HEADER_LENGTH + rx_data_length);
302                     } break;
303
304                     /*********************************************************
305                     ** Kalibrierung des TMP102
306                     *********************************************************/
307                     case COMMAND_DOWNLINK_CALIBRATION_TMP102: {
308
309                         cc430_rx(HEADER_LENGTH + rx_data_length);
310                     } break;
311
312                     /*********************************************************
313                     ** Kalibrierung des MSP430
314                     *********************************************************/
315                     case COMMAND_DOWNLINK_CALIBRATION_MSP430: {
316
317                         cc430_rx(HEADER_LENGTH + rx_data_length);
318                     } break;
319
320                     /*********************************************************
321                     ** Aufnahme der einfachen Spannungs- und Temp.-Messung
322                     *********************************************************/
323                     case COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE: {
324
325                         // IRQ TimerA anhalten
326                         // Da Interrupt stört NS 27.06.13
327                         if(IRQ_TIMERA_IS_BALANCING) {
328                             TIMER_A_STOP;
329                             TIMER_A_0_CM_IRQ_DISABLE;
330                         }
331
332                         if (balanc_state == OFF)    //TEST
333                             sample_buf_volt = adc12_get_volt_sample(clk_set);
334
335                         sample_buf_temp = temp_sensor_get_temp();
336                         sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
337                         sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
338
339
340                         //  IRQ TimerA vortsetzen
```

```
341                             if(IRQ_TIMERA_IS_BALANCING) {
342                                 TIMER_A_0_CM_IRQ_ENABLE;
343                                 TIMER_A_START_UP_MODE;
344                             }
345
346                             cc430_rx(HEADER_LENGTH + rx_data_length);
347                         } break;
348
349                         /**********************************************************
350                         ** Senden der einfachen Spannungs- und Temp.-Messung
351                         **********************************************************/
352                         case COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE: {
353
354 //                          uint8_t packet[8];
355
356                             packet[0] = ADDRESS_BASE_STATION;
357                             packet[1] = ADDRESS_THIS_SENSOR;
358                             packet[2] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE;
359                             packet[3] = 0x00;
360
361                             packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
362                             packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
363
364                             packet[6] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
365                             packet[7] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
366
367
368                             cc430_tx(8, packet);
369
370
371                             cc430_rx(HEADER_LENGTH + rx_data_length);
372                         } break;
373
374                         /**********************************************************
375                         ** Senden der Konfigurationsinformationen
376                         **********************************************************/
377                         case COMMAND_SW: {
378
379                             packet[0] = ADDRESS_BASE_STATION;
380                             packet[1] = ADDRESS_THIS_SENSOR;
381                             packet[2] = COMMAND_SW;
382                             packet[3] = 0x00;
383
384                             packet[4] = sw_version;
385                             packet[5] = adc_mode;
386                             packet[6] = brate;
387
388                             packet[7] = burst_frame_lenght;
389
390                             packet[8] = (balancing_volt >> 8) & 0xFF;
391                             packet[9] = (balancing_volt >> 0) & 0xFF;
392
393                             packet[10] = (upper_balanc_temp >> 8) & 0xFF;
394                             packet[11] = (upper_balanc_temp >> 0) & 0xFF;
395
396                             packet[12] = (lower_balanc_temp >> 8) & 0xFF;
397                             packet[13] = (lower_balanc_temp >> 0) & 0xFF;
398
399                             packet[14] = (upper_alarm_temp >> 8) & 0xFF;
400                             packet[15] = (upper_alarm_temp >> 0) & 0xFF;
401
402                             packet[16] = (lower_alarm_temp >> 8) & 0xFF;
403                             packet[17] = (lower_alarm_temp >> 0) & 0xFF;
404
405                             packet[18] = (offset >> 8) & 0x0F;
406                             packet[19] = (offset >> 0) & 0xFF;
407
408                             packet[20] = (gain >> 8) & 0x0F;
409                             packet[21] = (gain >> 0) & 0xFF;
410
411                             cc430_tx(22, packet);
412
413                             cc430_rx(HEADER_LENGTH + rx_data_length);
414                         } break;
415
416
417                         /**********************************************************
418                         ** Aufnahme der einfachen Spannungs- und Temp.-Messung
419                         **********************************************************/
420                         case COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE_ALL: {
421
422                             // IRQ TimerA anhalten
423                             // Da Interrupt stört NS 27.06.13
424                             if(IRQ_TIMERA_IS_BALANCING) {
425                                 TIMER_A_STOP;
```

```
426                     TIMER_A_0_CM_IRQ_DISABLE;
427                 }
428
429                 sample_buf_volt = adc12_get_volt_sample(clk_set);
430
431                 sample_buf_temp = temp_sensor_get_temp();
432                 sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
433                 sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
434
435                 sample_buf_temp_msp = adc12_get_temp_sample(clk_set);
436
437
438                 //  IRQ TimerA vortsetzen
439                 if(IRQ_TIMERA_IS_BALANCING) {
440                     TIMER_A_0_CM_IRQ_ENABLE;
441                     TIMER_A_START_UP_MODE;
442                 }
443
444                 cc430_rx(HEADER_LENGTH + rx_data_length);
445             } break;
446
447             /*************************************************************
448             ** Senden der einfachen Spannungs- und Temp.-Messung
449             *************************************************************/
450             case COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE_ALL: {
451
452                 packet[0] = ADDRESS_BASE_STATION;
453                 packet[1] = ADDRESS_THIS_SENSOR;
454                 packet[2] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE;
455                 packet[3] = 0x00;
456
457                 packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
458                 packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
459
460                 packet[6] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
461                 packet[7] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
462
463                 packet[8] = (uint8_t)((sample_buf_temp_msp >> 8) & 0x0F);
464                 packet[9] = (uint8_t)((sample_buf_temp_msp >> 0) & 0xFF);
465
466                 cc430_tx(10, packet);
467
468
469                 cc430_rx(HEADER_LENGTH + rx_data_length);
470             } break;
471
472             /*************************************************************
473             ** Aufnahme der einfachen Temperaturmessung
474             ** mit TMP102
475             *************************************************************/
476             case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102: {
477
478                 if(IRQ_TIMERA_IS_BALANCING) {
479                     TIMER_A_STOP;
480                     TIMER_A_0_CM_IRQ_DISABLE;
481                 }
482
483                 sample_buf_temp = temp_sensor_get_temp();
484                 sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
485                 sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
486
487                 //  IRQ TimerA vortsetzen
488                 if(IRQ_TIMERA_IS_BALANCING) {
489                     TIMER_A_0_CM_IRQ_ENABLE;
490                     TIMER_A_START_UP_MODE;
491                 }
492
493                 cc430_rx(HEADER_LENGTH + rx_data_length);
494             } break;
495
496             /*************************************************************
497             ** Senden der einfachen Temperaturmessung
498             ** mit TMP102
499             *************************************************************/
500             case COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102: {
501
502                 packet[0] = ADDRESS_BASE_STATION;
503                 packet[1] = ADDRESS_THIS_SENSOR;
504                 packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102;
505                 packet[3] = 0x00;
506
507                 packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
508                 packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
509
510                 cc430_tx(6, packet);
```

```
511
512                 cc430_rx(HEADER_LENGTH + rx_data_length);
513             } break;
514
515             /***********************************************************
516             ** Aufnahme der einfachen Temperaturmessung
517             ** mit TMP102 zur Kalibrierung
518             ***********************************************************/
519             case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102_CALI: {
520
521                 // IRQ TimerA anhalten
522                 // Da Interrupt stört NS 27.06.13
523                 if(IRQ_TIMERA_IS_BALANCING) {
524                     TIMER_A_STOP;
525                     TIMER_A_0_CM_IRQ_DISABLE;
526                 }
527
528                 sample_buf_temp = temp_sensor_get_temp();
529
530                 //  IRQ TimerA vortsetzen
531                 if(IRQ_TIMERA_IS_BALANCING) {
532                     TIMER_A_0_CM_IRQ_ENABLE;
533                     TIMER_A_START_UP_MODE;
534                 }
535
536                 delay_ms(1);
537
538                 cc430_rx(HEADER_LENGTH + rx_data_length);
539             } break;
540
541             /***********************************************************
542             ** Senden der einfachen Temperaturmessung
543             ** mit TMP102 zur Kalibrierung
544             ***********************************************************/
545             case COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI: {
546
547                 packet[0] = ADDRESS_BASE_STATION;
548                 packet[1] = ADDRESS_THIS_SENSOR;
549                 packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI;
550                 packet[3] = 0x00;
551
552                 packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
553                 packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
554
555                 cc430_tx(6, packet);
556
557                 cc430_rx(HEADER_LENGTH + rx_data_length);
558             } break;
559
560             /***********************************************************
561             ** Aufnahme der einfachen Temperaturmessung
562             ** mit MSP430 zur Kalibrierung
563             ***********************************************************/
564             case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430_CALI: {
565
566                 sample_buf_temp_msp = adc12_get_temp_sample(clk_set);
567                 delay_ms(1);
568
569                 cc430_rx(HEADER_LENGTH + rx_data_length);
570             } break;
571
572             /***********************************************************
573             ** Senden der einfachen Temperaturmessung
574             ** mit MSP430 zur Kalibrierung
575             ***********************************************************/
576             case COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI: {
577
578                 packet[0] = ADDRESS_BASE_STATION;
579                 packet[1] = ADDRESS_THIS_SENSOR;
580                 packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI;
581                 packet[3] = 0x00;
582
583                 packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
584                 packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
585
586                 cc430_tx(6, packet);
587
588                 cc430_rx(HEADER_LENGTH + rx_data_length);
589             } break;
590
591             /***********************************************************
592             ** Aufnahme der einfachen Temperaturmessung
593             ** mit MSP430
594             ***********************************************************/
595             case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430: {
```

```
596
597                 sample_buf_temp = adc12_get_temp_sample(clk_set);
598                 sample_buf_temp = sample_buf_temp + cali_pos_offset_msp430;
599                 sample_buf_temp = sample_buf_temp - cali_neg_offset_msp430;
600                 delay_ms(1);
601
602                 cc430_rx(HEADER_LENGTH + rx_data_length);
603             } break;
604
605             /***********************************************************
606             ** Senden der einfachen Temperaturmessung
607             ** mit MSP430
608             ***********************************************************/
609             case COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430: {
610
611                 packet[0] = ADDRESS_BASE_STATION;
612                 packet[1] = ADDRESS_THIS_SENSOR;
613                 packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430;
614                 packet[3] = 0x00;
615
616                 packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
617                 packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
618
619                 cc430_tx(6, packet);
620
621                 cc430_rx(HEADER_LENGTH + rx_data_length);
622             } break;
623
624             /***********************************************************
625             ** Aufnahme der einfachen Spannungsmessung
626             ***********************************************************/
627             case COMMAND_DOWNLINK_SAMPLE_VOLTAGE: {
628
629                 sample_buf_volt = adc12_get_volt_sample(clk_set);
630                 delay_ms(10);
631
632
633                 cc430_rx(HEADER_LENGTH + rx_data_length);
634             } break;
635
636             /***********************************************************
637             ** Senden der einfachen Spannungsmessung
638             ***********************************************************/
639             case COMMAND_DOWNLINK_SEND_VOLTAGE: {
640
641                 packet[0] = ADDRESS_BASE_STATION;
642                 packet[1] = ADDRESS_THIS_SENSOR;
643                 packet[2] = 0x00; //COMMAND_DOWNLINK_SEND_VOLTAGE;
644                 packet[3] = 0x00;
645
646                 packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
647                 packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
648
649                 cc430_tx(6, packet);
650
651                 cc430_rx(HEADER_LENGTH + rx_data_length);
652             } break;
653
654             /***********************************************************
655             ** Kommando zum Balancing empfangen
656             ***********************************************************/
657             case COMMAND_DOWNLINK_BALANCING_ON: {
658                 timer_a_init_balanc(); // TimerA wird initialisiert
659                 TIMER_A_START_UP_MODE;
660                 balanc_state = ON;
661
662                 cc430_rx(HEADER_LENGTH + rx_data_length);
663             } break;
664
665             /***********************************************************
666             ** Kommando zum Balancing ausschalten empfangen
667             ***********************************************************/
668             case COMMAND_DOWNLINK_BALANCING_OFF: {
669                 TIMER_A_STOP;
670                 balancing_off();
671                 balanc_state = OFF;
672
673
674                 cc430_rx(HEADER_LENGTH + rx_data_length);
675             } break;
676
677             /***********************************************************
678             ** Command to set the analog preprocessing
679             ***********************************************************/
680             case COMMAND_SET_PREPROCESSING: {
```

```
681
682             uint16_t voltage = 0;
683             uint16_t voltage_temp = 0;
684             uint16_t value = 0;
685             uint16_t data;
686             uint8_t index = 0;
687
688             clk_init_MHz(16);               // DCO auf 16 MHz setzen
689             clk_set = 16;                   // CLK ist auf 16MHz gesetzt
690
691             // Interrupt sperren
692             CC430_GDO2_IRQ_DISABLE;
693             CC430_GDO2_IRQ_CLEAR;
694
695             cc430_init_burst();              // Auf Empfang konfigurieren
696             cc430_reset();                  // cc430 reset
697             cc430_config_no_packet_rx();    // Auf asynchronen Empfang stellen
698             cc430_burst_rx();               // Ab hier wird empfangen
699             IRQ_SET_BURST;                  // IRQ auf Burst Mode stellen
700
701             adc12_volt_init(clk_set,2);       // Initialize ADC auf 16 MHz
702             delay_ms(100);
703
704             // Verstärkung auf Null setzen
705             write_AD5270_2(0x000);
706
707             // Minimale Spannung Abziehen
708             write_AD5270_1(0x000);
709
710             delay_ms(10);
711             voltage = adc12_get_volt_sample(clk_set);
712
713             // Spannung auf Zielwert einstellen oder maximal abziehen
714             while((voltage >= 30) && (value < 0x3FF)) {
715
716                 led_on(LED_TX);
717                 value++;
718                 write_AD5270_1(value);
719
720                 for(index=0;index < 20;index++) {
721
722                     delay_ms(1);
723                     data = adc12_get_volt_sample(clk_set);
724
725                     voltage = voltage + data;
726                 }
727                 voltage = voltage/20;
728
729                 led_off(LED_TX);
730             }
731
732             offset = value;
733
734             value = 0;
735             voltage = 0;
736
737             // Verstärkung einstellen
738             while((voltage <= 4000) && (value < 0x3FF)) {
739                 led_on(LED_RX);
740                 value++;
741                 write_AD5270_2(value);
742
743                 for(index=0;index < 20;index++) {
744                     delay_ms(1);
745                     voltage_temp = adc12_get_volt_sample(clk_set);
746
747                     // Nach dem größten Wert suchen
748                     if(voltage < voltage_temp)
749                         voltage = voltage_temp;
750                 }
751                 delay_ms(1);
752                 led_off(LED_RX);
753             }
754
755             //gain = 1 + ((97*value)/1000);
756             gain = value;
757
758             clk_init_MHz(1);            // DCO auf 1 MHz setzen
759             clk_set = 1;                // CLK ist auf 1 MHz gesetzt
760
761             cc430_rx(HEADER_LENGTH + rx_data_length);
762
763         } break;
764
765         /**********************************************************
```

```
766         ** Kommando zur Burstmessung
767         ***********************************************************/
768         case COMMAND_DOWNLINK_BURST_MODE: {
769
770             clk_init_MHz(16);                // DCO auf 16 MHz setzen
771             clk_set = 16;                    // CLK ist auf 16MHz gesetzt
772
773             adc12_volt_init(clk_set,2);        // Initialize ADC auf 16 MHz
774             burst_counter = 0;               // Zurücksetzen
775             burst_memory = 0;
776             //init_ad7691();                       // AD7691 testing
777             led_off(LED_ALL);
778
779             // Interrupt sperren
780             CC430_GDO2_IRQ_DISABLE;
781             CC430_GDO2_IRQ_CLEAR;
782
783             // IRQ TimerA anhalten
784             // Da Interrupt stört NS 27.06.13
785             if(IRQ_TIMERA_IS_BALANCING) {
786                 TIMER_A_STOP;
787                 TIMER_A_0_CM_IRQ_DISABLE;
788             }
789
790             cc430_init_burst();               // Auf Empfang konfigurieren
791             cc430_reset();                   // cc430 reset
792             cc430_config_no_packet_rx();     // Auf asynchronen Empfang stellen
793             cc430_burst_rx();                // Ab hier wird empfangen
794             IRQ_SET_BURST;                    // IRQ auf Burst Mode stellen
795             delay_ms(32);                     // Wartezeit
796
797             // Interrupt freischalten
798             CC430_GDO2_IRQ_CLEAR;
799             CC430_GDO2_IRQ_ENABLE;
800             // Ab hier wird Interruptgesteuert empfangen
801
802             command_recived = 0;
803         } break;
804
805         /***********************************************************
806         ** Kommando wenn vorher BURST MODE war, um CLK unzustellen
807         ***********************************************************/
808         case COMMAND_BACK_FROM_BURST: {
809
810             _DINT();                 // Globale Interrups ausschalten
811             clk_init_MHz(1);
812             _EINT();                 // Globale Interrups einschalten
813
814             // Turn on the red LED
815             led_on(LED_AWAKE);
816
817             // IRQ TimerA fortsetzen
818             // Da Interrupt stört NS 27.06.13
819             if(IRQ_TIMERA_IS_BALANCING) {
820                 TIMER_A_START_UP_MODE;
821                 TIMER_A_0_CM_IRQ_ENABLE;
822             }
823
824             cc430_rx(HEADER_LENGTH + rx_data_length);
825         } break;
826
827         /***********************************************************
828         ** Leersendung
829         ***********************************************************/
830         case COMMAND_DOWNLINK_BURST_CHECK: {
831
832             cc430_rx(HEADER_LENGTH + rx_data_length);
833         } break;
834
835
836         /***********************************************************
837         ** Anfrage der Burst Daten
838         ***********************************************************/
839         case COMMAND_DOWNLINK_BURST_DATA_RQ: {
840             //Berechnung der Anzahl der Frames
841             uint16_t index = burst_counter;
842             //burst_counter = 0;            // Zurücksetzen
843             burst_frame_counter = 0;     // Zurücksetzen
844
845             while(index >= (burst_frame_lenght/2)) {
846                 burst_frame_counter++;
847                 index = index - (burst_frame_lenght/2);
848             }
849             //if(index != 0)  // Berechnung wenn weniger als (burst_frame_lenght/2) übrig bleibt
850             //     burst_frame_counter++;
```

```
851
852                 packet[0] = ADDRESS_BASE_STATION;
853                 packet[1] = ADDRESS_THIS_SENSOR;
854                 packet[2] = burst_frame_counter;  // Anzahl der Frames
855                 packet[3] = burst_frame_lenght;   // Länge der Frames ohne Header
856
857                 cc430_tx(4,packet);
858
859                 cc430_rx(HEADER_LENGTH + rx_data_length);
860             } break;
861
862             /**********************************************************
863             ** Receive the Goertzel Frequnecy
864             **********************************************************/
865             case COMMAND_GOERTZEL_FREQ: {
866                 cc430_rx(HEADER_LENGTH + rx_data_length);
867             } break;
868
869             /**********************************************************
870             ** Receive the Goertzel stimuli Frequnecy
871             **********************************************************/
872             case COMMAND_GOERTZEL_STIMULI_FREQ: {
873                 cc430_rx(HEADER_LENGTH + rx_data_length);
874             } break;
875
876             /**********************************************************
877             ** Receive the Goertzel stimuli Frequnecy
878             **********************************************************/
879             case COMMAND_GOERTZEL_STIMULI_FREQ2: {
880                 cc430_rx(HEADER_LENGTH + rx_data_length);
881             } break;
882
883             /**********************************************************
884             ** Receive the Goertzel number of periodes
885             **********************************************************/
886             case COMMAND_GOERTZEL_NUMB_PERI: {
887                 cc430_rx(HEADER_LENGTH + rx_data_length);
888             } break;
889
890             /**********************************************************
891             ** Calculate the Goertzel
892             **********************************************************/
893             case COMMAND_GOERTZEL: {
894
895                 int k;
896                 int N_int;
897                 int mod = 0;
898
899                 uint16_t data = 0;
900                 uint16_t goertzel_memory_counter = 0;
901
902                 real_char.float_data = 0.0;
903                 imag_char.float_data = 0.0;
904
905                 f_gain = 1 + (gain*(100000/1024))/1000;
906
907                 bn_0 = 0;   // Reset the coefficients
908                 bn_1 = 0;   // Reset the coefficients
909                 bn_2 = 0;   // Reset the coefficients
910
911                 //*************************************************************
912                 Np = Fs_goertzel/f_goertzel;            // Np Anzahl Abtastwerte pro Periode
913                 N_float = m * Np;                       // Anzahl der Abtastpunke für die Berechnung
914                 N_int = (int) N_float;
915
916
917                 k = (int) (0.5 + ((N_float * f_goertzel) / Fs_goertzel));
918                 omega = (2.0 * 3.141592654 * k) / N_float;
919                 sine = sinf(omega);
920                 cosine = cosf(omega);
921                 coeff = 2.0 * cosine;
922
923                 //*** BEGIN the Goertzel algorithm ***//
924                 for (goertzel_index = 0; goertzel_index < N_int+1; goertzel_index++) {
925
926                     if(goertzel_index == N_int) {   // The last value in the calculation have to be zero
927                         bn_0 = coeff * bn_1 - bn_2 +  0;
928                     } else {
929                         // Dekodieren der Daten
930                         mod = goertzel_index % 4;
931
932                         if(mod == 0){
933                             data = (sample_burst_buf[goertzel_memory_counter] >> 4) & 0x0FFF;
934
935                         }else if(mod == 1){
```

```
936                             data = (sample_burst_buf[goertzel_memory_counter] << 8) & 0x0F00;
937                             goertzel_memory_counter++;
938 
939                             data |= (sample_burst_buf[goertzel_memory_counter] >> 8) & 0x00FF;
940 
941                         }else if(mod == 2){
942                             data = (sample_burst_buf[goertzel_memory_counter] << 4) & 0x0FF0;
943                             goertzel_memory_counter++;
944 
945                             data |= (sample_burst_buf[goertzel_memory_counter] >> 12) & 0x000F;
946 
947                         }else if(mod == 3){
948                             data = (sample_burst_buf[goertzel_memory_counter] >> 0 ) & 0x0FFF;
949                             goertzel_memory_counter++;
950                         }
951 
952                         //f_voltage = (((2.5/4095)*data)/f_gain);
953                         f_voltage = (((0.000568986)*data)/f_gain);
954 
955                         bn_0 = coeff * bn_1 - bn_2 + f_voltage;
956                     }
957 
958 
959                     bn_2 = bn_1;
960                     bn_1 = bn_0;
961                 }
962 
963                 real = (bn_1 - bn_2 * cosine);
964                 imag = (bn_2 * sine);
965 
966                 real_char.float_data =  real;
967                 imag_char.float_data =  imag;
968 
969                 //*** End the Goertzel algorithm ***//
970 
971                 packet[0] = ADDRESS_BASE_STATION;
972                 packet[1] = ADDRESS_THIS_SENSOR;
973                 packet[2] = 0x00;
974                 packet[3] = 0x00;
975 
976                 packet[4] = real_char.hex_data[3];
977                 packet[5] = real_char.hex_data[2];
978                 packet[6] = real_char.hex_data[1];
979                 packet[7] = real_char.hex_data[0];
980 
981                 packet[8] = imag_char.hex_data[3];
982                 packet[9] = imag_char.hex_data[2];
983                 packet[10] = imag_char.hex_data[1];
984                 packet[11] = imag_char.hex_data[0];
985 
986                 delay_ms(50);
987 
988                 tx_data_length = 8;
989                 cc430_tx((HEADER_LENGTH + tx_data_length),packet);
990 
991                 rx_data_length = 0;
992                 cc430_rx(HEADER_LENGTH + rx_data_length);
993             } break;
994 
995             /***********************************************************
996             ** Senden der Burst Daten an das Batteriesteuergerät
997             ***********************************************************/
998             case COMMAND_DOWNLINK_BURST_DATA_RX: {
999                 uint8_t gain_value = 0x00;
1000                uint16_t seq_number      = 0;
1001                uint8_t sample_counter   = 0;
1002                uint8_t packages_counter = 0;
1003
1004                packet[0] = ADDRESS_BASE_STATION;
1005                packet[1] = ADDRESS_THIS_SENSOR;
1006
1007                // Coding the gain value and the selected ADC
1008                gain_value = (adc_mode << 7);
1009                gain_value |= (gain >> 8);
1010
1011                packet[2] = gain_value;             // Welcher ADC wurde verwendet
1012                packet[3] = gain & 0xFF;
1013
1014
1015                if(adc_mode == ADC12_INT_SELECT) {  // Wenn der 12 Bit ADC verwendet wurde, sind die Daten Codiert
1016                    for(packages_counter=0;packages_counter < burst_frame_lenght;packages_counter++) {
1017
1018                        packet[HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                sample_counter] >> 12) & 0x0F);
1019                        packages_counter++;
```

```
1020
1021                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 4) & 0xFF);
1022                         packages_counter++;
1023
1024                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] << 0) & 0x0F);
1025                         packages_counter++;
1026                         sample_counter++;
1027
1028                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 8) & 0xFF);
1029                         packages_counter++;
1030
1031                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 4) & 0x0F);
1032                         packages_counter++;
1033
1034                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] << 4) & 0xF0);
1035                         sample_counter++;
1036                         packet [HEADER_LENGTH + packages_counter] |= (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 12) & 0x0F);
1037                         packages_counter++;
1038
1039                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 8) & 0x0F);
1040                         packages_counter++;
1041
1042                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(18))+
                                   sample_counter] >> 0) & 0xFF);
1043                         sample_counter++;
1044                     }
1045                 } else if (AD7691_EXT_SELECT) {
1046                     for(packages_counter=0;packages_counter < burst_frame_lenght;packages_counter++) {
1047                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(
                                   burst_frame_lenght/2))+sample_counter] >> 8) & 0xFF);
1048                         seq_number++;        // Sequenznummer hochzählen
1049                         packages_counter++;
1050                         packet [HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(
                                   burst_frame_lenght/2))+sample_counter] >> 0) & 0xFF);
1051                         seq_number++;        // Sequenznummer hochzählen
1052                         sample_counter++;
1053                     }
1054                 }
1055
1056                 cc430_tx((HEADER_LENGTH + burst_frame_lenght),packet);
1057
1058                 cc430_rx(HEADER_LENGTH + rx_data_length);
1059             } break;
1060
1061             /*************************************************************
1062             ** Kommando zur Takt Kalibrierung
1063             *************************************************************/
1064             case COMMAND_CALI_CLK: {
1065
1066
1067                 //uint32_t dco_value = 480;
1068
1069                 AD7691_CNV_SET_LOW;          // Testpin
1070
1071                 do{
1072                     if(rx_command == COMMAND_CLK_UP)
1073                         dco_value++;
1074                     else if(rx_command == COMMAND_CLK_DOWN)
1075                         dco_value--;
1076                     else if(rx_command == COMMAND_CALI_BURST)
1077                         dco_value = dco_value;
1078
1079
1080                     //*** BEGIN DCO auf neuen Wert setzen ****************************************
1081                     __bis_SR_register(SCG0);               // Disable the FLL control loop
1082                     UCSCTL0 = 0x0000;                      // Set lowest possible DCOx, MODx
1083                     UCSCTL1 = DCORSEL_7;                   // Select DCO range 24MHz operation
1084                     UCSCTL2 = FLLD_1 + dco_value;          // Set DCO Multiplier for 12MHz
1085                                                            // (N + 1) * FLLRef = Fdco
1086                                                            // (487 + 1) * 32768 = 16MHz
1087                                                            // Set FLL Div = fDCOCLK/2
1088                     __bic_SR_register(SCG0);               // Enable the FLL control loop
1089
1090                     __delay_cycles(500000);
1091                     //*** END DCO auf neuen Wert setzen ****************************************
1092
1093
1094                     //*** Grundeinstellungen *********************************************
```

```
1095                // GDO0 ist Output und GDO2 ist Input
1096                CC430_GDO2_DIR_IN;
1097                CC430_GDO2_IRQ_DISABLE;
1098                CC430_GDO2_IRQ_CLEAR;
1099                CC430_GDO2_IRQ_FALLING_EDGE;
1100
1101                CC430_GDO0_SET_HIGH;
1102                //*********************************************************************
1103
1104                //*** Takt synchronisierung *****************************************
1105                cc430_reset();                  // Reset all registers to their default values and go to idle
                        state
1106
1107                PMMCTL0_H = 0xA5;               // Get write-access to port mapping regs
1108                PMMCTL0_L |= PMMHPMRE_L;
1109                PMMCTL0_H = 0x00;               // Lock Port mapping
1110
1111                cc430_config_cali();            // Configure the transceiver for sending the wakeup signal
1112
1113                timer_b_init_clock_cali();
1114
1115                //************************************************************************
1116                // Map the timer to Port P2.2 (thats only for debuging)
1117                // Init P2.2 to output TX signal from the radio
1118                P2SEL |= BIT3;
1119                P2DIR |= BIT3;
1120                PMAPPWD = 0x02D52;                          // Get write-access to port mapping regs
1121                P2MAP3 = PM_TA1CCR0A;                       // Map TA output to TX to P2.2
1122                PMAPPWD = 0x00;                             // Lock Port mapping
1123                //************************************************************************
1124
1125                cc430_burst_tx();
1126                AD7691_CNV_SET_HIGH;            // Testpin
1127                while(clk_cali_counter<300);
1128                AD7691_CNV_SET_LOW;         // Testpin
1129                clk_cali_counter = 0;
1130                TIMER_A_STOP;
1131
1132                //*** BEGIN DCO wieder auf 1MHz setzen *****************************************
1133                clk_set = 1;    //CLK ist auf 1MHz gesetzt
1134                __bis_SR_register(SCG0);                    // Disable the FLL control loop
1135                UCSCTL0 = 0x0000;                           // Set lowest possible DCOx, MODx
1136                UCSCTL1 = DCORSEL_3;                        // Select DCO range 16MHz operation
1137                UCSCTL2 = FLLD_1 + DCO_MP_1MHZ;             // Set DCO Multiplier for 8MHz
1138                                                            // (N + 1) * FLLRef = Fdco
1139                                                            // (249 + 1) * 32768 = 8MHz
1140                                                            // Set FLL Div = fDCOCLK/2
1141                __bic_SR_register(SCG0);                    // Enable the FLL control loop
1142
1143                __delay_cycles(31250);
1144                //*** END DCO wieder auf 1MHz setzen *****************************************
1145
1146                cc430_rx(HEADER_LENGTH + rx_data_length);   // Get in RX mode
1147
1148                command_recived = 0;                        // Reset the RX flag
1149                while(!command_recived);                    // Wait until a command is received
1150
1151                command_recived = 0;                        // Reset the RX flag
1152            }while(rx_command != COMMAND_CLK_OK);           // Check if the CLK is ok
1153            //**********************************************************************
1154
1155             cc430_rx(HEADER_LENGTH + rx_data_length);
1156            command_recived = 0;
1157        } break;
1158
1159        /************************************************************
1160        ** Kommando zur Laufzeitermittlung
1161        ************************************************************/
1162        case COMMAND_CALI_BURST: {
1163
1164            clk_init_MHz(16);               // DCO auf 16 MHz setzen
1165            clk_set = 16;                   // CLK ist auf 16MHz gesetzt
1166            uint16_t time_adc = 0;
1167
1168            AD7691_CNV_SET_LOW;
1169            timer_a_test();
1170            //*** Grundeinstellungen *********************************************
1171            // GDO0 ist Output und GDO2 ist Input
1172            CC430_GDO2_DIR_IN;
1173            CC430_GDO2_IRQ_DISABLE;
1174            CC430_GDO2_IRQ_CLEAR;
1175            CC430_GDO2_IRQ_FALLING_EDGE;        // IRQ is trigger on a rising egde
1176
1177            CC430_GDO0_ACCESS;              // Get access to the GDO0 PIN
1178            CC430_GDO0_SET_HIGH;            // Set GDO0 pin high
```

```
1179                    //*** Grundeinstellungen *********************************************
1180
1181                    //cc430_init_burst();            // Auf Empfang konfigurieren
1182                    cc430_reset();                   // cc430 reset
1183                    cc430_config_cali_burst();
1184                    cc430_burst_rx();                // Ab hier wird empfangen
1185                    IRQ_SET_CALI_BURST;              // IRQ auf Burst Mode stellen
1186                    BURST_CALI_FLAG_UNSET;
1187                    delay_ms(1);
1188                    // Interrupt freischalten
1189                    CC430_GDO2_IRQ_CLEAR;
1190                    CC430_GDO2_IRQ_ENABLE;
1191
1192                    // Ab hier wird Interruptgesteuert empfangen
1193                    while(BURST_CALI_FLAG_IS_UNSET);
1194
1195                    //*********************************************************************
1196  //                cc430_reset();          // Reset chip and go to idle state
1197  //                cc430_config_cali_burst();
1198                    CC430_GDO0_SET_HIGH;             // Set GDO0 pin high
1199                    Strobe(RF_SIDLE);
1200                    Strobe(RF_STX);
1201                    delay_ms(2);
1202                    AD7691_CNV_SET_LOW;          // Testpin
1203                    Strobe(RF_SIDLE);
1204
1205                    time_adc = TA0R;
1206                    TIMER_B_STOP;
1207
1208                    clk_init_MHz(1);                 // DCO auf 16 MHz setzen
1209                    clk_set = 1;                    // CLK ist auf 16MHz gesetzt
1210
1211                    delay_ms(50);
1212
1213                    packet[0] = ADDRESS_BASE_STATION;
1214                    packet[1] = ADDRESS_THIS_SENSOR;
1215                    packet[2] = (time_adc >> 8) & 0xFF;
1216                    packet[3] = (time_adc >> 0) & 0xFF;
1217
1218                    packet[4] = (time_adc >> 8) & 0xFF;
1219                    packet[5] = (time_adc >> 0) & 0xFF;
1220
1221                    cc430_tx(6, packet);
1222
1223                    cc430_rx(HEADER_LENGTH + rx_data_length);
1224                    command_recived = 0;
1225
1226                } break;
1227
1228                /**********************************************************
1229                ** Kommando SLEEP empfangen
1230                **********************************************************/
1231                case COMMAND_DOWNLINK_SLEEP: {
1232
1233                    led_on(LED_ALL);
1234                    delay_ms(1000);
1235                    led_off(LED_ALL);
1236
1237                    init_for_sleep();
1238
1239                    ENTER_LPM4;
1240
1241                } break;
1242
1243                /**********************************************************
1244                **
1245                **********************************************************/
1246                case COMMAND_WAIT: {
1247
1248                    cc430_rx(HEADER_LENGTH + rx_data_length);
1249                } break;
1250            }
1251
1252        }
1253    }
1254  }
```

```
1   /*********************************************************************************
2   **    Description    : spi.c
3   **    Hardware       : BATSEN ZS 3 v0.5
4   **    Date           : 22/11/2014
5   **    Last Update    : 02/08/2015
6   **    Author         : Nico Sassano
7   **    State          : Final state
8   *********************************************************************************/
9   #include "header\main.h"
10
11  void init_spi(void) {
12      UCA0CTL1 |= UCSWRST;                          // **Put state machine in reset**
13
14      UCA0CTL0 |= UCMST;                          // Sync. Mode: Master Select
15      UCA0CTL0 |= UCSYNC;                         // Sync-Mode  0:UART-Mode / 1:SPI-Mode
16      UCA0CTL0 &= ~UCCKPL;                        // Sync. Mode: Clock Polarity
17      UCA0CTL0 &= ~UCCKPH;                        // Sync. Mode: Clock Polarity
18      UCA0CTL0 |= UCMSB;                          // Async. Mode: MSB first  0:LSB / 1:MSB
19
20      UCA0CTL1 |= (UCSSEL1 | UCSSEL0);
21      UCA0BR0 = 8;                                // /2
22      UCA0BR1 = 0;                                //
23
24      UCA0MCTL = 0;                               // No modulation
25
26      // Confic the SPI functionality
27      SPI_PxSEL |= SPI_CLK_PIN;
28      SPI_PxSEL |= SPI_MOSI_PIN;
29
30      SPI_PxSEL |= SPI_MISO_PIN;
31      SPI_PxDIR |= SPI_CLK_PIN;                   // Define CLK as outputs
32      SPI_PxDIR |= SPI_MOSI_PIN;                  // Define MOSI as outputs
33      SPI_PxDIR &= ~SPI_MISO_PIN;                 // MISO is input
34
35      UCA0CTL1 &= ~UCSWRST;                       // **Initialize USCI state machine**
36
37  }
38
39  void spi_write_register(uint8_t address, uint8_t reg, uint8_t value) {
40
41      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
42      UCA0TXBUF = address | reg;          // Send address and register
43      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
44      UCA0TXBUF = 0x00;                   // Write dummy Byte
45      while (UCA0STAT & UCBUSY);          // Wait for TX complete
46      UCA0TXBUF = value;                  // Send value
47      while (UCA0STAT & UCBUSY);          // Wait for TX complete
48
49  }
50
51  char spi_read_register(uint8_t address, uint8_t reg) {
52      uint8_t value;
53
54      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
55      UCA0TXBUF = address | reg;          // Send address and register
56      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
57      UCA0TXBUF = 0x00;                   // Write dummy Byte
58      while (UCA0STAT & UCBUSY);          // Wait for TX complete
59      UCA0TXBUF = 0x00;                   // Write dummy Byte
60      while (UCA0STAT & UCBUSY);          // Wait for TX complete
61      value = UCA0RXBUF;                  // Read data from buffer
62
63      return value;                       // Return the value
64  }
65
66  void spi_send_command(uint8_t command, uint8_t reg) {
67
68      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
69      UCA0TXBUF = command | reg;               // Send configuration mode and register
70      while (UCA0STAT & UCBUSY);          // Wait for TX to finish
71
72  }
```

```c
/************************************************************************************
**     Description    : temp_sensor.c
**     Hardware       : BATSEN ZS 3 v0.5
**     Date           : 07/03/2013
**     Last Update    : 02/08/2015
**     Author         : Nico Sassano
**     State          : Final state
************************************************************************************/
#include "header\main.h"

extern uint16_t config_value;

void temp_sensor_init(void) {
    i2c_init();
    temp_sensor_get_contr_reg();
}

void temp_sensor_get_contr_reg() {
    uint8_t data[2];

    /**************************************************************
    * Pointer wird auf das Config Register ausgerichtet
    **************************************************************/
    data[0] = TMP102_REG_CONF; // Value for Pointer-Register

    i2c_write(ADDR_TMP102, 1, data);      // Set Pointer-Register

    i2c_read(ADDR_TMP102, 2, data);       // Read Control-Register

    config_value = (uint16_t) data[0] << 4;
    config_value |= data[1] >> 4;
}

void temp_sensor_config_reg() {
    uint8_t data[3];

    /*************************************************************
    * Hier wird der Temperatursensor konfiguriert
    *************************************************************/
    TMP102_EM_OFF;
    TMP102_CON_RATE_4;
    TMP102_SD_OFF;
    TMP102_COMPERATOR_MODE;
    TMP102_POL_INV;
    TMP102_FAULTS_6;

    data[0] = TMP102_REG_CONF;                      // Value for Pointer-Register
    data[1] = (uint8_t) (config_value >> 8);        // Byte 1 at first
    data[2] = (uint8_t) (config_value & 0x00FF);    // Byte 2 at last

    i2c_write(ADDR_TMP102, 3, data);                // Write Control-Register
}

/*****************************************************************
** Setzen der Alarmtemperatur
** 1 Digit -> 0,0625°C
** 0x1900  -> 25°C
*****************************************************************/
void temp_sensor_set_alert(uint16_t temp_low, uint16_t temp_high) {
    uint8_t data[3];

    /**************************************************************
    * Low Daten setzen
    **************************************************************/
    data[0] = TMP102_REG_LOW;                       // Value for Pointer-Register
    data[1] = (uint8_t) (temp_low >> 8);            // Byte 1 at first
    data[2] = (uint8_t) (temp_low & 0x00FF);        // Byte 2 at last

    i2c_write(ADDR_TMP102, 3, data);                // Write Control-Register

    /**************************************************************
    * High Daten setzen
    **************************************************************/
    data[0] = TMP102_REG_HIGH;                      // Value for Pointer-Register
    data[1] = (uint8_t) (temp_high >> 8);           // Byte 1 at first
    data[2] = (uint8_t) (temp_high & 0x00FF);       // Byte 2 at last

    i2c_write(ADDR_TMP102, 3, data);                // Write Control-Register
}

uint16_t temp_sensor_get_temp(void) {

    uint8_t data[2];
    uint16_t temp_value = 0x0000;

```

```
86      data[0] = TMP102_REG_TEMP;                          // Value for Pointer-Register
87      i2c_write(ADDR_TMP102, 1, data);                    // Set Pointer-Register
88      temp_value = 0x0000;
89      i2c_read(ADDR_TMP102, 2, data);
90
91      temp_value = (uint16_t) data[0] << 4;
92      temp_value |= data[1] >> 4;
93
94      return temp_value;
95  }
```

```
1  /*******************************************************************************
2  **     Description    : timer.c
3  **     Hardware       : BATSEN ZS 3 v0.5
4  **     Date           : 13/03/2013
5  **     Last Update    : 02/08/2015
6  **     Author         : Nico Sassano
7  **     State          : Final state
8  *******************************************************************************/
9  #include "header\main.h"
10
11 extern volatile uint8_t irq_timera;           // Status TimerA
12 extern volatile uint8_t irq_timerb;           // Status TimerB
13 extern volatile uint8_t burst_error_flag;
14 extern volatile uint8_t adc_select;
15 extern volatile uint8_t clk_set;
16 extern volatile uint16_t delay_counter;
17
18 /**********************************************************
19 ** Timer A Initialisierung für die Balancierung
20 ** 1MHz; DIV=8; TACCR0 = 62500 -> 500ms
21 **********************************************************/
22 void timer_a_init_balanc(void) {
23      IRQ_TIMERA_SET_BALANCING;
24      TIMER_A_STOP;
25      TIMER_A_RESET;
26      TIMER_A_SOURCE_SMCLK;
27      TIMER_A_SOURCE_DIV_8;
28
29      TACCR0 = 62500;
30      TIMER_A_0_CM_IRQ_ENABLE;
31 }
32
33 /**********************************************************
34 ** Timer A TEST TIMER
35 ** 1MHz; DIV=8; TACCR0 = 62500 -> 500ms
36 **********************************************************/
37 void timer_a_test(void) {
38      TIMER_A_STOP;
39      TIMER_A_RESET;
40      TIMER_A_SOURCE_SMCLK;
41      TIMER_A_SOURCE_DIV_1;
42
43      TACCR0 = 62500;
44      TIMER_A_0_CM_IRQ_ENABLE;
45 }
46
47 /**********************************************************
48 ** Timer B Initialisierung Trigger Abstandsmessung für die
49 ** Burstkalibrierung
50 ** 16MHz; DIV=1; TACCR0 = 65536 -> 4,096ms
51 **********************************************************/
52 void timer_b_init_cali(void) {
53      TIMER_B_STOP;
54      TIMER_B_RESET;
55
56      TIMER_B_SOURCE_SMCLK;
57      TIMER_B_SOURCE_DIV_1;
58      //TAR = 0;
59      //TACCR0 = 0x00;
60      //TIMER_A_0_CM_IRQ_ENABLE;
61
62 }
63
64 /**********************************************************
65 ** Timer A Initialisierung für die Taktkalibrierung
66 ** 16MHz; DIV=8; TACCR0 = 16000 -> 1ms
67 **********************************************************/
68 void timer_b_init_clock_cali(void) {
69      IRQ_TIMERB_SET_CLOCK_CALI;
70      TA1CCR0 = 16000;
71      TA1CCR2 = 16000;
72
73      TA1CCTL0 = OUTMOD_4;     /* PWM output mode: 4 - toggle */
74      TA1CCTL1 = OUTMOD_4;     /* PWM output mode: 4 - toggle */
75
76      TA1CTL = TASSEL__SMCLK + MC_1 + TACLR;
77
78      TIMER_B_0_CM_IRQ_ENABLE;
79 }
80
81 /**********************************************************
82 ** Timer B
83 ** Initialisierung für die Burstmessung
84 **********************************************************/
85 void timer_b_init_burst(uint8_t freq) {
```

```
86          IRQ_TIMERB_SET_BURST;
87
88          TIMER_B_STOP;
89          TIMER_B_RESET;
90          TIMER_B_SOURCE_SMCLK;
91
92          switch(freq) {
93
94              /************************
95              ** 16MHz; TimerDiv 1
96              ** Periode   -> 100us
97              ** 1472 (ISR)-> 92us
98              ************************/
99              case BURST_FREQ_10000HZ: {
100                 TIMER_B_SOURCE_DIV_1;
101
102                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
103                     TBCCR0 = 2000 - 1472;    // 125us - 92us (ISR)
104                 } else {
105                     TBCCR0 = 1600 - 184;     // 100 us - 11,5us (ISR)
106                     BURST_ERROR_FLAG_UNSET;
107                 }
108
109                 // Interrupts für TimerB freischalten
110                 TIMER_B_0_CM_IRQ_ENABLE;
111                 TIMER_B_1_CM_IRQ_DISABLE;
112             }break;
113
114             /************************
115             ** 16MHz; TimerDiv 1
116             ** Periode      -> 125us
117             ** 1472 (ISR)   -> 92us
118             ************************/
119             case BURST_FREQ_8000HZ: {
120                 TIMER_B_SOURCE_DIV_1;
121
122                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
123                     TBCCR0 = 2400 - 1472;    // 150us - 92us (ISR)
124                 } else {
125                     TBCCR0 = 2000 - 184;     // 125 us - 11,5us (ISR)
126                     BURST_ERROR_FLAG_UNSET;
127                 }
128
129                 // Interrupts für TimerB freischalten
130                 TIMER_B_0_CM_IRQ_ENABLE;
131                 TIMER_B_1_CM_IRQ_DISABLE;
132             }break;
133
134             /************************
135             ** 16MHz; TimerDiv 1
136             ** Periode   -> 166us
137             ** 1472 (ISR)-> 92us
138             ************************/
139             case BURST_FREQ_6000HZ: {
140                 TIMER_B_SOURCE_DIV_1;
141
142                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
143                     TBCCR0 = 3200 - 1472;     // 200 us - 92us (ISR)
144                 } else {
145                     TBCCR0 = 2656 - 184;      // 166 us - 11,5us (ISR)
146                     BURST_ERROR_FLAG_UNSET;
147                 }
148
149                 // Interrupts für TimerB freischalten
150                 TIMER_B_0_CM_IRQ_ENABLE;
151                 TIMER_B_1_CM_IRQ_DISABLE;
152             }break;
153
154             /************************
155             ** 16MHz; TimerDiv 1
156             ** Periode   -> 250us
157             ** 1472 (ISR)-> 92us
158             ************************/
159             case BURST_FREQ_4000HZ: {
160                 TIMER_B_SOURCE_DIV_1;
161
162                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
163                     TBCCR0 = 4400 - 1472;    // 275us - 92us (ISR)
164                 } else {
165                     TBCCR0 = 4000 - 184;     // 250 us - 11,5us (ISR)
166                     BURST_ERROR_FLAG_UNSET;
167                 }
168
169                 // Interrupts für TimerB freischalten
170                 TIMER_B_0_CM_IRQ_ENABLE;
```

```
171                 TIMER_B_1_CM_IRQ_DISABLE;
172             }break;
173
174             /************************
175             ** 16MHz; TimerDiv 1
176             ** Periode  -> 500us
177             ** 1472 (ISR)-> 92us
178             ************************/
179             case BURST_FREQ_2000HZ: {
180                 TIMER_B_SOURCE_DIV_1;
181
182                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
183                     TBCCR0 = 9600 - 1472;    // 600us - 92us (ISR)
184                 } else {
185                     TBCCR0 = 8000 - 184;     // 500 us - 11,5us (ISR)
186                     BURST_ERROR_FLAG_UNSET;
187                 }
188
189                 // Interrupts für TimerB freischalten
190                 TIMER_B_0_CM_IRQ_ENABLE;
191                 TIMER_B_1_CM_IRQ_DISABLE;
192             }break;
193
194             /************************
195             ** 16MHz; TimerDiv 1
196             ** Periode  -> 1ms
197             ** 12800    -> 800us
198             ** 19200    -> 1.2ms
199             ** 1472 (ISR)-> 92us
200             ************************/
201             case BURST_FREQ_1000HZ: {
202                 TIMER_B_SOURCE_DIV_1;
203
204                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
205                     TBCCR1 = 12800 - 1472;   // 800us  - 92us (ISR)
206                     TBCCR0 = 19200 - 1472;   // 1200us - 92us (ISR)
207                 } else {
208                     TBCCR1 = 8600 ;          // 600 us
209                     TBCCR0 = 16000;          // 1000 us
210                     BURST_ERROR_FLAG_UNSET;
211                 }
212
213                 // Interrupts für TimerB freischalten
214                 TIMER_B_0_CM_IRQ_ENABLE;
215                 TIMER_B_1_CM_IRQ_ENABLE;
216
217             }break;
218
219             /************************
220             ** 16MHz; TimerDiv 1
221             ** Periode  -> 1.052ms
222             ** 13632    -> 0.852 ms
223             ** 20032    -> 1.252 ms
224             ** 1472 (ISR)-> 92us
225             ************************/
226             case BURST_FREQ_950HZ: {
227                 TIMER_B_SOURCE_DIV_1;
228                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
229                     TBCCR1 = 13632 - 1472;  // 0.852 ms - 33 us
230                     TBCCR0 = 20032 - 1472;   // 1.252 ms - 33 us
231                 } else {
232                     TBCCR1 = 10432;          // 652 us
233                     TBCCR0 = 16832;          // 1052 us
234                     BURST_ERROR_FLAG_UNSET;
235                 }
236
237                 // Interrupts für TimerB freischalten
238                 TIMER_B_0_CM_IRQ_ENABLE;
239                 TIMER_B_1_CM_IRQ_ENABLE;
240             }break;
241
242             /************************
243             ** 16MHz; TimerDiv 1
244             ** Periode  -> 1.10 ms
245             ** 14400    -> 0.90 ms
246             ** 20800    -> 1.30 ms
247             ** 1472 (ISR)-> 92us
248             ************************/
249             case BURST_FREQ_900HZ: {
250                 TIMER_B_SOURCE_DIV_1;
251                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
252                     TBCCR1 = 14400 - 1472;
253                     TBCCR0 = 20800 - 1472;
254                 } else {
255                     TBCCR1 = 11200;          // 700 us
```

```
256             TBCCR0 = 17600;            // 1.1 ms
257             BURST_ERROR_FLAG_UNSET;
258         }
259
260         // Interrupts für TimerB freischalten
261         TIMER_B_0_CM_IRQ_ENABLE;
262         TIMER_B_1_CM_IRQ_ENABLE;
263     }break;
264
265     /***********************
266     ** 16MHz; TimerDiv 1
267     ** Periode   -> 1.176 ms
268     ** 15616     -> 0.976 ms
269     ** 22016     -> 1.376 ms
270     ** 1472 (ISR)-> 92us
271     ***********************/
272     case BURST_FREQ_850HZ: {
273         TIMER_B_SOURCE_DIV_1;
274         if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
275             TBCCR1 = 15616 - 1472;
276             TBCCR0 = 22016 - 1472;
277         } else {
278             TBCCR1 = 12416;            // 0.776 ms
279             TBCCR0 = 18816;            // 1.176 ms
280             BURST_ERROR_FLAG_UNSET;
281         }
282
283         // Interrupts für TimerB freischalten
284         TIMER_B_0_CM_IRQ_ENABLE;
285         TIMER_B_1_CM_IRQ_ENABLE;
286     }break;
287
288     /***********************
289     ** 16MHz; TimerDiv 1
290     ** Periode   -> 1.25 ms
291     ** 16800     -> 1.05 ms
292     ** 23200     -> 1.45 ms
293     ** 1472 (ISR)-> 92us
294     ***********************/
295     case BURST_FREQ_800HZ: {
296         TIMER_B_SOURCE_DIV_1;
297         if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
298             TBCCR1 = 16800 - 1472;
299             TBCCR0 = 23200 - 1472;
300         } else {
301             TBCCR1 = 13600;            // 850us
302             TBCCR0 = 20000;            // 1.25 ms
303             BURST_ERROR_FLAG_UNSET;
304         }
305
306         // Interrupts für TimerB freischalten
307         TIMER_B_0_CM_IRQ_ENABLE;
308         TIMER_B_1_CM_IRQ_ENABLE;
309     }break;
310
311     /***********************
312     ** 16MHz; TimerDiv 1
313     ** Periode   -> 1.33 ms
314     ** 18080     -> 1.13 ms
315     ** 24480     -> 1.53 ms
316     ** 1472 (ISR)-> 92us
317     ***********************/
318     case BURST_FREQ_750HZ: {
319         TIMER_B_SOURCE_DIV_1;
320         if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
321             TBCCR1 = 18080 - 1472;
322             TBCCR0 = 24480 - 1472;
323         } else {
324             TBCCR1 = 14880;            // 0.930 ms
325             TBCCR0 = 21280;            // 1.330 ms
326             BURST_ERROR_FLAG_UNSET;
327         }
328
329         // Interrupts für TimerB freischalten
330         TIMER_B_0_CM_IRQ_ENABLE;
331         TIMER_B_1_CM_IRQ_ENABLE;
332     }break;
333
334     /***********************
335     ** 16MHz; TimerDiv 1
336     ** Periode   -> 1.49 ms
337     ** 20640     -> 1.29 ms
338     ** 27040     -> 1.69 ms
339     ** 1472 (ISR)-> 92us
340     ***********************/
```

```
341         case BURST_FREQ_700HZ: {
342             TIMER_B_SOURCE_DIV_1;
343             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
344                 TBCCR1 = 20640 - 1472;
345                 TBCCR0 = 27040 - 1472;
346             } else {
347                 TBCCR1 = 17440;          // 1.09 ms
348                 TBCCR0 = 23840;          // 1.49 ms
349                 BURST_ERROR_FLAG_UNSET;
350             }
351
352             // Interrupts für TimerB freischalten
353             TIMER_B_0_CM_IRQ_ENABLE;
354             TIMER_B_1_CM_IRQ_ENABLE;
355         }break;
356
357         /************************
358         ** 16MHz; TimerDiv 1
359         ** Periode  -> 1.54 ms
360         ** 21440    -> 1.34 ms
361         ** 27840    -> 1.74 ms
362         ** 1472 (ISR)-> 92us
363         ************************/
364         case BURST_FREQ_650HZ: {
365             TIMER_B_SOURCE_DIV_1;
366
367             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
368                 TBCCR1 = 21440 - 1472;
369                 TBCCR0 = 27840 - 1472;
370             } else {
371                 TBCCR1 = 18240;          // 1.14 ms
372                 TBCCR0 = 24640;          // 1.54 ms
373                 BURST_ERROR_FLAG_UNSET;
374             }
375
376             // Interrupts für TimerB freischalten
377             TIMER_B_0_CM_IRQ_ENABLE;
378             TIMER_B_1_CM_IRQ_ENABLE;
379         }break;
380
381         /************************
382         ** 16MHz; TimerDiv 1
383         ** Periode  -> 1.60 ms
384         ** 22400    -> 1.40 ms
385         ** 28800    -> 1.80 ms
386         ** 1472 (ISR)-> 92us
387         ************************/
388         case BURST_FREQ_600HZ: {
389             TIMER_B_SOURCE_DIV_1;
390
391             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
392                 TBCCR1 = 22400 - 1472;
393                 TBCCR0 = 28800 - 1472;
394             } else {
395                 TBCCR1 = 19200;          // 1.2 ms
396                 TBCCR0 = 25600;          // 1.6 ms
397                 BURST_ERROR_FLAG_UNSET;
398             }
399
400             // Interrupts für TimerB freischalten
401             TIMER_B_0_CM_IRQ_ENABLE;
402             TIMER_B_1_CM_IRQ_ENABLE;
403         }break;
404
405         /************************
406         ** 16MHz; TimerDiv 1
407         ** Periode  -> 1.80 ms
408         ** 25600    -> 1.60 ms
409         ** 32000    -> 2.00 ms
410         ** 1472 (ISR)-> 92us
411         ************************/
412         case BURST_FREQ_550HZ: {
413             TIMER_B_SOURCE_DIV_1;
414
415             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
416                 TBCCR1 = 25600 - 1472;
417                 TBCCR0 = 32000 - 1472;
418             } else {
419                 TBCCR1 = 22400;          // 1.4 ms
420                 TBCCR0 = 28800;          // 1.8 ms
421                 BURST_ERROR_FLAG_UNSET;
422             }
423
424             // Interrupts für TimerB freischalten
425             TIMER_B_0_CM_IRQ_ENABLE;
```

```
426                 TIMER_B_1_CM_IRQ_ENABLE;
427             }break;
428
429             /************************
430             ** 16MHz; TimerDiv 1
431             ** Periode   -> 2.00 ms
432             ** 28800     -> 1.80 ms
433             ** 35200     -> 2.20 ms
434             ** 1472 (ISR)-> 92us
435             ************************/
436             case BURST_FREQ_500HZ: {
437                 TIMER_B_SOURCE_DIV_1;
438
439                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
440                     TBCCR1 = 28800 - 1472;
441                     TBCCR0 = 35200 - 1472;
442                 } else {
443                     TBCCR1 = 25600;            // 1.6 ms
444                     TBCCR0 = 32000;            // 2.0 ms
445                     BURST_ERROR_FLAG_UNSET;
446                 }
447
448                 // Interrupts für TimerB freischalten
449                 TIMER_B_0_CM_IRQ_ENABLE;
450                 TIMER_B_1_CM_IRQ_ENABLE;
451             }break;
452
453             /************************
454             ** 16MHz; TimerDiv 1
455             ** Periode   -> 2.22 ms
456             ** 32320     -> 2.02 ms
457             ** 38720     -> 2.42 ms
458             ** 1472 (ISR)-> 92us
459             ************************/
460             case BURST_FREQ_450HZ: {
461                 TIMER_B_SOURCE_DIV_1;
462
463                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
464                     TBCCR1 = 32320 - 1472;
465                     TBCCR0 = 38720 - 1472;
466                 } else {
467                     TBCCR1 = 29120;            // 1.82 ms
468                     TBCCR0 = 35520;            // 2.22 ms
469                     BURST_ERROR_FLAG_UNSET;
470                 }
471
472                 // Interrupts für TimerB freischalten
473                 TIMER_B_0_CM_IRQ_ENABLE;
474                 TIMER_B_1_CM_IRQ_ENABLE;
475             }break;
476
477             /************************
478             ** 16MHz; TimerDiv 1
479             ** Periode   -> 2.50 ms
480             ** 36800     -> 2.30 ms
481             ** 43200     -> 2.00 ms
482             ** 1472 (ISR)-> 92us
483             ************************/
484             case BURST_FREQ_400HZ: {
485                 TIMER_B_SOURCE_DIV_1;
486
487                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
488                     TBCCR1 = 36800 - 1472;
489                     TBCCR0 = 43200 - 1472;
490                 } else {
491                     TBCCR1 = 33600;            // 2.1 ms
492                     TBCCR0 = 40000;            // 2.5 ms
493                     BURST_ERROR_FLAG_UNSET;
494                 }
495
496                 // Interrupts für TimerB freischalten
497                 TIMER_B_0_CM_IRQ_ENABLE;
498                 TIMER_B_1_CM_IRQ_ENABLE;
499             }break;
500
501             /************************
502             ** 16MHz; TimerDiv 1
503             ** Periode   -> 2.85 ms
504             ** 42400     -> 2.65 ms
505             ** 48800     -> 3.05 ms
506             ** 1472 (ISR)-> 92us
507             ************************/
508             case BURST_FREQ_350HZ: {
509                 TIMER_B_SOURCE_DIV_1;
510
```

```
511                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
512                     TBCCR1 = 42400 - 1472;
513                     TBCCR0 = 48800 - 1472;
514                 } else {
515                     TBCCR1 = 39200;             // 2.45 ms
516                     TBCCR0 = 45600;             // 2.85 ms
517                     BURST_ERROR_FLAG_UNSET;
518                 }
519 
520                 // Interrupts für TimerB freischalten
521                 TIMER_B_0_CM_IRQ_ENABLE;
522                 TIMER_B_1_CM_IRQ_ENABLE;
523             }break;
524 
525             /************************
526             ** 16MHz; TimerDiv 1
527             ** Periode  -> 3.33 ms
528             ** 50080    -> 3.13 ms
529             ** 56480    -> 3.53 ms
530             ** 1472 (ISR)-> 92us
531             ************************/
532             case BURST_FREQ_300HZ: {
533                 TIMER_B_SOURCE_DIV_1;
534 
535                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
536                     TBCCR1 = 50080 - 1472;
537                     TBCCR0 = 56480 - 1472;
538                 } else {
539                     TBCCR1 = 46880;             // 2.93 ms
540                     TBCCR0 = 53280;             // 3.33 ms
541                     BURST_ERROR_FLAG_UNSET;
542                 }
543 
544                 // Interrupts für TimerB freischalten
545                 TIMER_B_0_CM_IRQ_ENABLE;
546                 TIMER_B_1_CM_IRQ_ENABLE;
547             }break;
548 
549             /************************
550             ** 16MHz; TimerDiv 2
551             ** Periode  -> 4.00 ms
552             ** 30400    -> 3.80 ms
553             ** 33600    -> 4.20 ms
554             ** 264 (ISR)-> 92us
555             ************************/
556             case BURST_FREQ_250HZ: {
557                 TIMER_B_SOURCE_DIV_2;
558 
559                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
560                     TBCCR1 = 30400 - 264;
561                     TBCCR0 = 33600 - 264;
562                 } else {
563                     TBCCR1 = 28800;             // 3.6 ms
564                     TBCCR0 = 32800;             // 4.1 ms
565                     BURST_ERROR_FLAG_UNSET;
566                 }
567 
568                 // Interrupts für TimerB freischalten
569                 TIMER_B_0_CM_IRQ_ENABLE;
570                 TIMER_B_1_CM_IRQ_ENABLE;
571             }break;
572 
573             /************************
574             ** 16MHz; TimerDiv 2
575             ** Periode  -> 5.00 ms
576             ** 38400    -> 4.80 ms
577             ** 41600    -> 5.20 ms
578             ** 264 (ISR)-> 92us
579             ************************/
580             case BURST_FREQ_200HZ: {
581                 TIMER_B_SOURCE_DIV_2;
582 
583                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
584                     TBCCR1 = 38400 - 264;
585                     TBCCR0 = 41600 - 264;
586                 } else {
587                     TBCCR1 = 36800;             // 4.6 ms
588                     TBCCR0 = 40000;             // 5.0 ms
589                     BURST_ERROR_FLAG_UNSET;
590                 }
591 
592                 // Interrupts für TimerB freischalten
593                 TIMER_B_0_CM_IRQ_ENABLE;
594                 TIMER_B_1_CM_IRQ_ENABLE;
595             }break;
```

```
596
597         /************************
598         ** 16MHz; TimerDiv 2
599         ** Periode  -> 6.66 ms
600         ** 51680    -> 6.46 ms
601         ** 54880    -> 6.86 ms
602         ** 264 (ISR)-> 92us
603         ************************/
604         case BURST_FREQ_150HZ: {
605             TIMER_B_SOURCE_DIV_2;
606
607             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
608                 TBCCR1 = 51680 - 264;
609                 TBCCR0 = 54880 - 264;
610
611             } else {
612                 TBCCR1 = 50080;         // 6.26 ms
613                 TBCCR0 = 53280;         // 6.66 ms
614                 BURST_ERROR_FLAG_UNSET;
615             }
616
617             // Interrupts für TimerB freischalten
618             TIMER_B_0_CM_IRQ_ENABLE;
619             TIMER_B_1_CM_IRQ_ENABLE;
620         }break;
621
622         /************************
623         ** 16MHz; TimerDiv 4
624         ** Periode  -> 10.0 ms
625         ** 39200    -> 9.80 ms
626         ** 40800    -> 10.2 ms
627         ** 132 (ISR)-> 92us             //TODO NACH MEINER RECHNUNG 368 fuer 92us wohin ist der rest?
628         ** 6534 (ISR AD24) -> 4.553ms
629         ************************/
630         case BURST_FREQ_100HZ: {
631             TIMER_B_SOURCE_DIV_4;
632
633             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert
634                 TBCCR1 = 39200 - 132;
635                 TBCCR0 = 40800 - 132;
636             } else {
637
638                 TBCCR1 = 38400;         //  9.6 ms
639                 TBCCR0 = 40025;         // 10.0 ms
640
641                 BURST_ERROR_FLAG_UNSET;
642             }
643
644             // Interrupts für TimerB freischalten
645             TIMER_B_0_CM_IRQ_ENABLE;
646             TIMER_B_1_CM_IRQ_ENABLE;
647         }break;
648
649         /************************
650         ** 16MHz; TimerDiv 8
651         ** Periode  -> 20.0 ms
652         ** 39600    -> 19.8 ms
653         ** 40400    -> 20.2 ms
654         ** 66 (ISR) -> 92us
655         ************************/
656         case BURST_FREQ_50HZ: {
657             TIMER_B_SOURCE_DIV_8;
658
659             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
660                 TBCCR1 = 39600 - 66;
661                 TBCCR0 = 40400 - 66;
662             } else {
663                 TBCCR1 = 39200;         // 19.6 ms
664                 TBCCR0 = 40000;         // 20.0 ms
665                 BURST_ERROR_FLAG_UNSET;
666             }
667
668             // Interrupts für TimerB freischalten
669             TIMER_B_0_CM_IRQ_ENABLE;
670             TIMER_B_1_CM_IRQ_ENABLE;
671         }break;
672     }
673 }
674
675 /**************************************************************
676 ** Timer B
677 ** Timer for the delay function
678 **************************************************************/
679 void delay_ms(uint16_t ms) {
680     delay_counter = 0;
```

```
681        IRQ_TIMERB_SET_DELAY;
682        TIMER_B_STOP;
683        TIMER_B_RESET;
684        TIMER_B_SOURCE_SMCLK;
685        TIMER_B_SOURCE_DIV_1;
686
687        switch(clk_set) {
688            case 1:
689                TBCCR0 = 1000;
690                break;
691
692            case 12:
693                TBCCR0 = 12000;
694
695            case 16:
696                TBCCR0 = 16000;
697
698            default: break;
699        }
700
701        TIMER_B_0_CM_IRQ_ENABLE;
702        TIMER_B_START_UP_MODE;
703
704        // Wait until the delay time is over
705        while(delay_counter<ms);
706
707        TIMER_B_STOP;
708        IRQ_TIMERB_UNSET_DELAY;
709    }
```

```
1   /*************************************************************************************
2   **     File name      : AD5270.h
3   **     Hardware       : BATSEN ZS Klasse 3 v0.5
4   **     Date           : 18/04/2015
5   **     Last Update    : 03/05/2015
6   **     Author         : Nico Sassano
7   **     Description    : Header file for the Rheostat AD5270
8   *************************************************************************************/
9   #ifndef AD5270_H_
10  #define AD5270_H_
11
12  #define AD5270_NOP_CMD              0x0000
13  #define AD5270_WRITE_CMD            0x0400
14  #define AD5270_READ_CMD             0x0800
15  #define AD5270_STORE_CMD            0x0C00
16  #define AD5270_SW_RST_CMD           0x1000
17  #define AD5270_SW_SHD_ON_CMD        0x2401
18
19  #define AD5270_1_CS_PxDIR           P1DIR
20  #define AD5270_1_CS_PxOUT           P1OUT
21  #define AD5270_1_CS_PIN             BIT0
22
23  #define AD5270_2_CS_PxDIR           P3DIR
24  #define AD5270_2_CS_PxOUT           P3OUT
25  #define AD5270_2_CS_PIN             BIT7
26
27  #define AD5270_SPI_PxSEL            P1SEL
28  #define AD5270_SPI_PxDIR            P1DIR
29  #define AD5270_SPI_PxIN             P1IN
30  #define AD5270_SPI_MOSI_PIN         BIT6
31  #define AD5270_SPI_MISO_PIN         BIT5
32  #define AD5270_SPI_CLK_PIN          BIT7
33
34  #define AD5270_1_CS_ENABLE          AD5270_1_CS_PxOUT &= ~AD5270_1_CS_PIN
35  #define AD5270_2_CS_ENABLE          AD5270_2_CS_PxOUT &= ~AD5270_2_CS_PIN
36
37  #define AD5270_1_CS_DISABLE         AD5270_1_CS_PxOUT |= AD5270_1_CS_PIN
38  #define AD5270_2_CS_DISABLE         AD5270_2_CS_PxOUT |= AD5270_2_CS_PIN
39
40  // Declarate the prototype functions
41  void write_AD5270_1(uint16_t);
42  void write_AD5270_2(uint16_t);
43
44  #endif /* AD5270_H_ */
```

```
1   /*************************************************************************************
2   **     File name      : adc12.h
3   **     Hardware       : BATSEN ZS Klasse 3 v0.5
4   **     Date           : 06/03/2013
5   **     Last Update    : 03/08/2015
6   **     Author         : Nico Sassano
7   **     Description    : Header for the ADC12
8   *************************************************************************************/
9   #ifndef ADC12_H_
10  #define ADC12_H_
11
12  #include "main.h"
13
14  #define ADC12_VBAT_PxSEL                P2SEL
15  #define ADC12_VBAT_PxDIR                P2DIR
16  #define ADC12_VBAT_PIN                  BIT2
17
18  #define ADC12_DIR_IN                    (ADC12_VBAT_PxDIR &= ~ADC12_VBAT_PIN)
19  #define ADC12_SET_PIN                   (ADC12_VBAT_PxSEL |= ADC12_VBAT_PIN)
20
21  /*****************************************************************
22  ** Sample-and-hold time for registers ADC12MEM8 to ADC12MEM15
23  *****************************************************************/
24  #define ADC12_SHT1_CLK_4                (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT01 | ADC12SHT00))
25  #define ADC12_SHT1_CLK_8                (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT01)); \
26                                          (ADC12CTL0 |= (ADC12SHT00))
27  #define ADC12_SHT1_CLK_16               (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT00)); \
28                                          (ADC12CTL0 |= (ADC12SHT01))
29  #define ADC12_SHT1_CLK_32               (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02)); \
30                                          (ADC12CTL0 |= (ADC12SHT01 | ADC12SHT00))
31  #define ADC12_SHT1_CLK_64               (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT01 | ADC12SHT00)); \
32                                          (ADC12CTL0 |= (ADC12SHT02))
33  #define ADC12_SHT1_CLK_96               (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT01)); \
34                                          (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT00))
35  #define ADC12_SHT1_CLK_128              (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT00)); \
36                                          (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT01))
37  #define ADC12_SHT1_CLK_192              (ADC12CTL0 &=~(ADC12SHT03)); \
38                                          (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT01 | ADC12SHT00))
39  #define ADC12_SHT1_CLK_256              (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT01 | ADC12SHT00)); \
40                                          (ADC12CTL0 |= (ADC12SHT03))
```

```
41  #define ADC12_SHT1_CLK_384        (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT01)); \
42                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT00))
43  #define ADC12_SHT1_CLK_512        (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT00)); \
44                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT01))
45  #define ADC12_SHT1_CLK_768        (ADC12CTL0 &=~(ADC12SHT02)); \
46                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT01 |ADC12SHT00))
47  #define ADC12_SHT1_CLK_1024       (ADC12CTL0 &=~(ADC12SHT01 | ADC12SHT00)); \
48                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT02))
49
50  /****************************************************************
51  ** Sample-and-hold time for registers ADC12MEM0 to ADC12MEM7
52  ****************************************************************/
53  #define ADC12_SHT0_CLK_4          (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT01 | ADC12SHT00))
54  #define ADC12_SHT0_CLK_8          (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT01)); \
55                                    (ADC12CTL0 |= (ADC12SHT00))
56  #define ADC12_SHT0_CLK_16         (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02 | ADC12SHT00)); \
57                                    (ADC12CTL0 |= (ADC12SHT01))
58  #define ADC12_SHT0_CLK_32         (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT02)); \
59                                    (ADC12CTL0 |= (ADC12SHT01 | ADC12SHT00))
60  #define ADC12_SHT0_CLK_64         (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT01 | ADC12SHT00)); \
61                                    (ADC12CTL0 |= (ADC12SHT02))
62  #define ADC12_SHT0_CLK_96         (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT01)); \
63                                    (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT00))
64  #define ADC12_SHT0_CLK_128        (ADC12CTL0 &=~(ADC12SHT03 | ADC12SHT00)); \
65                                    (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT01))
66  #define ADC12_SHT0_CLK_192        (ADC12CTL0 &=~(ADC12SHT03)); \
67                                    (ADC12CTL0 |= (ADC12SHT02 | ADC12SHT01 | ADC12SHT00))
68  #define ADC12_SHT0_CLK_256        (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT01 | ADC12SHT00)); \
69                                    (ADC12CTL0 |= (ADC12SHT03))
70  #define ADC12_SHT0_CLK_384        (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT01)); \
71                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT00))
72  #define ADC12_SHT0_CLK_512        (ADC12CTL0 &=~(ADC12SHT02 | ADC12SHT00)); \
73                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT01))
74  #define ADC12_SHT0_CLK_768        (ADC12CTL0 &=~(ADC12SHT02)); \
75                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT01 |ADC12SHT00))
76  #define ADC12_SHT0_CLK_1024       (ADC12CTL0 &=~(ADC12SHT01 | ADC12SHT00)); \
77                                    (ADC12CTL0 |= (ADC12SHT03 | ADC12SHT02))
78
79  /****************************************************************************
80  ** Reference generator voltage
81  ****************************************************************************/
82  #define ADC12_REF_SET_1_5V        (ADC12CTL0 &=~REF2_5V)
83  #define ADC12_REF_SET_2_5V        (ADC12CTL0 |= REF2_5V)
84
85  /****************************************************************************
86  ** Reference generator on/off
87  ****************************************************************************/
88  #define ADC12_SET_REF_OFF         (ADC12CTL0 &=~REFON)
89  #define ADC12_SET_REF_ON          (ADC12CTL0 |= REFON)
90
91  /****************************************************************************
92  ** ADC12 on/off
93  ****************************************************************************/
94  #define ADC12_SET_OFF             (ADC12CTL0 &=~ADC12ON)
95  #define ADC12_SET_ON              (ADC12CTL0 |= ADC12ON)
96
97  /****************************************************************************
98  ** ADC12 clock divider
99  ****************************************************************************/
100 #define ADC12DIV_SET_1            (ADC12CTL1 &=~(ADC12DIV2 | ADC12DIV1 | ADC12DIV0))
101 #define ADC12DIV_SET_2            (ADC12CTL1 &=~(ADC12DIV2 | ADC12DIV1 )); \
102                                   (ADC12CTL1 |= (ADC12DIV0))
103 #define ADC12DIV_SET_3            (ADC12CTL1 &=~(ADC12DIV2 | ADC12DIV0 )); \
104                                   (ADC12CTL1 |= (ADC12DIV1))
105 #define ADC12DIV_SET_4            (ADC12CTL1 &=~(ADC12DIV2)); \
106                                   (ADC12CTL1 |= (ADC12DIV1 | ADC12DIV0))
107 #define ADC12DIV_SET_5            (ADC12CTL1 &=~(ADC12DIV1 | ADC12DIV0 )); \
108                                   (ADC12CTL1 |= (ADC12DIV2))
109 #define ADC12DIV_SET_6            (ADC12CTL1 &=~(ADC12DIV1)); \
110                                   (ADC12CTL1 |= (ADC12DIV2 | ADC12DIV0))
111 #define ADC12DIV_SET_7            (ADC12CTL1 &=~(ADC12DIV0)); \
112                                   (ADC12CTL1 |= (ADC12DIV2 | ADC12DIV1))
113 #define ADC12DIV_SET_8            (ADC12CTL1 |= (ADC12DIV2 | ADC12DIV1 | ADC12DIV0))
114
115 /****************************************************************************
116 ** ADC12 clock source select
117 ****************************************************************************/
118 #define ADC12SSEL_SET_ADC12OSC    (ADC12CTL1 &=~(ADC12SSEL1 | ADC12SSEL0))
119 #define ADC12SSEL_SET_ACLK        (ADC12CTL1 &=~(ADC12SSEL1)); \
120                                   (ADC12CTL1 |= (ADC12SSEL0))
121 #define ADC12SSEL_SET_MCLK        (ADC12CTL1 &=~(ADC12SSEL0)); \
122                                   (ADC12CTL1 |= (ADC12SSEL1))
123 #define ADC12SSEL_SET_SCLK        (ADC12CTL1 |= (ADC12SSEL1 | ADC12SSEL0))
124
125 /****************************************************************************
```

```
126 ** ADC12 Conversion start address
127 ****************************************************************************/
128 #define ADC12MEM_SET_0            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD2 | CSTARTADD1 | CSTARTADD0))
129 #define ADC12MEM_SET_1            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD2 | CSTARTADD1)); \
130                                   (ADC12CTL1 |=  (CSTARTADD0))
131 #define ADC12MEM_SET_2            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD2 | CSTARTADD0)); \
132                                   (ADC12CTL1 |=  (CSTARTADD1))
133 #define ADC12MEM_SET_3            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD2)); \
134                                   (ADC12CTL1 |=  (CSTARTADD1 | CSTARTADD0))
135 #define ADC12MEM_SET_4            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD1 | CSTARTADD0)); \
136                                   (ADC12CTL1 |=  (CSTARTADD2))
137 #define ADC12MEM_SET_5            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD1)); \
138                                   (ADC12CTL1 |=  (CSTARTADD2 | CSTARTADD0))
139 #define ADC12MEM_SET_6            (ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD0)); \
140                                   (ADC12CTL1 |=  (CSTARTADD2 | CSTARTADD1))
141 #define ADC12MEM_SET_7            (ADC12CTL1 &= ~(CSTARTADD3)); \
142                                   (ADC12CTL1 |=  (CSTARTADD2 | CSTARTADD1 | CSTARTADD0))
143 #define ADC12MEM_SET_8            (ADC12CTL1 &= ~(CSTARTADD2 | CSTARTADD1 | CSTARTADD0)); \
144                                   (ADC12CTL1 |=  (CSTARTADD3))
145 #define ADC12MEM_SET_9            (ADC12CTL1 &= ~(CSTARTADD2 | CSTARTADD1)); \
146                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD0))
147 #define ADC12MEM_SET_10           (ADC12CTL1 &= ~(CSTARTADD2 | CSTARTADD0)); \
148                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD1))
149 #define ADC12MEM_SET_11           (ADC12CTL1 &= ~(CSTARTADD2)); \
150                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD1 | CSTARTADD0))
151
152 #define ADC12MEM_SET_12           (ADC12CTL1 &= ~(CSTARTADD1 | CSTARTADD0)); \
153                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD2))
154 #define ADC12MEM_SET_13           (ADC12CTL1 &= ~(CSTARTADD1); \
155                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD2 | CSTARTADD0)))
156 #define ADC12MEM_SET_14           (ADC12CTL1 &= ~(CSTARTADD0); \
157                                   (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD2 | CSTARTADD1)))
158 #define ADC12MEM_SET_15           (ADC12CTL1 |=  (CSTARTADD3 | CSTARTADD2 | CSTARTADD1 | CSTARTADD0))
159
160
161 /****************************************************************************
162 ** Conversion sequence mode select
163 ** CONSEQ0 −> Single−channel, single−conversion
164 ** CONSEQ1 −> Sequence−of−channels
165 ** CONSEQ2 −> Repeat−single−channel
166 ** CONSEQ3 −> Repeat−sequence−of−channels
167 ****************************************************************************/
168 #define ADC12_SET_CONSEQ0         (ADC12CTL1 &= ~(CONSEQ1 | CONSEQ0))
169 #define ADC12_SET_CONSEQ1         (ADC12CTL1 &= ~(CONSEQ1); \
170                                   (ADC12CTL1 |=  (CONSEQ0)
171 #define ADC12_SET_CONSEQ2         (ADC12CTL1 &= ~(CONSEQ0); \
172                                   (ADC12CTL1 |=  (CONSEQ1)
173 #define ADC12_SET_CONSEQ3         (ADC12CTL1 |=  (CONSEQ1 | CONSEQ0))
174
175 /****************************************************************************
176 ** Sample−and−hold pulse−mode select
177 ** SHP0 −> SAMPCON signal is sourced from the sample−input signal
178 ** SHP1 −> SAMPCON signal is sourced from the sampling timer
179 ****************************************************************************/
180 #define ADC12_SET_SHP0            (ADC12CTL1  &= ~(SHP))
181 #define ADC12_SET_SHP1            (ADC12CTL1  |=  (SHP))
182
183 /****************************************************************************
184 ** Select reference
185 ** REF0 −> VR+ = AVCC and VR− = AVSS
186 ** REF1 −> VR+ = VREF+ and VR− = AVSS
187 ** REF2 −> VR+ = VeREF+ and VR− = AVSS
188 ** REF3 −> VR+ = VeREF+ and VR− = AVSS
189 ** REF4 −> VR+ = AVCC and VR− = VREF−/ VeREF−
190 ** REF5 −> VR+ = VREF+ and VR− = VREF−/ VeREF−
191 ** REF6 −> VR+ = VeREF+ and VR− = VREF−/ VeREF−
192 ** REF7 −> VR+ = VeREF+ and VR− = VREF−/ VeREF
193 ****************************************************************************/
194 #define ADC12_SET_REF0            (ADC12MCTL0 |= SREF_0)
195 #define ADC12_SET_REF1            (ADC12MCTL0 |= SREF_1)
196 #define ADC12_SET_REF2            (ADC12MCTL0 |= SREF_2)
197 #define ADC12_SET_REF3            (ADC12MCTL0 |= SREF_3)
198 #define ADC12_SET_REF4            (ADC12MCTL0 |= SREF_4)
199 #define ADC12_SET_REF5            (ADC12MCTL0 |= SREF_5)
200 #define ADC12_SET_REF6            (ADC12MCTL0 |= SREF_6)
201 #define ADC12_SET_REF7            (ADC12MCTL0 |= SREF_7)
202
203 /****************************************************************************
204 ** Input channel select
205 **
206 ****************************************************************************/
207 #define ADC12_SET_INPUT_CH0       (ADC12MCTL0 |= ADC12INCH_0)
208 #define ADC12_SET_INPUT_CH1       (ADC12MCTL0 |= ADC12INCH_1)
209 #define ADC12_SET_INPUT_CH2       (ADC12MCTL0 |= ADC12INCH_2)
210 #define ADC12_SET_INPUT_CH3       (ADC12MCTL0 |= ADC12INCH_3)
```

```
211  #define ADC12_SET_INPUT_CH4          (ADC12MCTL0 |= ADC12INCH_4)
212  #define ADC12_SET_INPUT_CH5          (ADC12MCTL0 |= ADC12INCH_5)
213  #define ADC12_SET_INPUT_CH6          (ADC12MCTL0 |= ADC12INCH_6)
214  #define ADC12_SET_INPUT_CH7          (ADC12MCTL0 |= ADC12INCH_7)
215  #define ADC12_SET_TEMPERATUR         (ADC12MCTL0 |= ADC12INCH_10)
216
217  /*------------------------------------------------------------------------
218    Public functions
219  ------------------------------------------------------------------------*/
220
221  uint16_t adc12_get_volt_sample(uint8_t);
222  uint16_t adc12_get_temp_sample(uint8_t);
223  void adc12_init(void);
224  void adc12_volt_init(uint8_t, uint8_t);
225  void adc12_temp_init(uint8_t);
226  void adc12_disable(void);
227
228  #endif /* ADC12_H_ */
```

```
 1  /*****************************************************************************
 2  **     File name      : adc12.h
 3  **     Hardware       : BATSEN ZS Klasse 3 v0.5
 4  **     Date           : 11/22/2012
 5  **     Last Update    : 03/08/2015
 6  **     Author         : Phillip Durdaut and Nico Sassano
 7  **     Description    : Header for the ADG918
 8  *****************************************************************************/
 9  #ifndef ADG918_H_
10  #define ADG918_H_
11  #include "main.h"
12
13  #define ADG918_CTRL_PxDIR ( P3DIR )
14  #define ADG918_CTRL_PxOUT (P3OUT)
15  #define ADG918_CTRL_PIN ( BIT0 )
16
17  void adg918_init (void) ;
18  void adg918_wakeup (void);
19  void adg918_transceiver (void) ;
20  #endif /* ADG918_H_ */
```

```
 1  /*****************************************************************************
 2  **     File name      : AS3930.h
 3  **     Hardware       : BATSEN ZS Klasse 3 v0.5
 4  **     Date           : 10/09/2012
 5  **     Last Update    : 09/06/2015
 6  **     Author         : Nico Sassano
 7  **     Description    : Header for the AS3930
 8  *****************************************************************************/
 9  #ifndef AS3930_H_
10  #define AS3930_H_
11
12  #include "main.h"
13
14  #define AS3930_CS_PxDIR              P5DIR
15  #define AS3930_CS_PxOUT              P5OUT
16  #define AS3930_CS_PIN                BIT1
17
18  #define AS3930_SPI_PxSEL             P1SEL
19  #define AS3930_SPI_PxDIR             P1DIR
20  #define AS3930_SPI_PxIN              P1IN
21  #define AS3930_SPI_MOSI_PIN          BIT6
22  #define AS3930_SPI_MISO_PIN          BIT5
23  #define AS3930_SPI_CLK_PIN           BIT7
24  //
25  #define AS3930_WAKE_PxDIR            P2DIR
26  #define AS3930_WAKE_PxIES            P2IES
27  #define AS3930_WAKE_PxIE             P2IE
28  #define AS3930_WAKE_PxIFG            P2IFG
29  #define AS3930_WAKE_PIN              BIT5
30
31  #define AS3930_WAKE_DIR_IN               (AS3930_WAKE_PxDIR &= ~AS3930_WAKE_PIN)
32  #define AS3930_WAKE_IRQ_RISING_EDGE      (AS3930_WAKE_PxIES &= ~AS3930_WAKE_PIN)
33  #define AS3930_WAKE_IRQ_FALLING_EDGE     (AS3930_WAKE_PxIES |= AS3930_WAKE_PIN)
34  #define AS3930_WAKE_IRQ_ENABLE           (AS3930_WAKE_PxIE |= AS3930_WAKE_PIN)
35  #define AS3930_WAKE_IRQ_DISABLE          (AS3930_WAKE_PxIE &= ~AS3930_WAKE_PIN)
36  #define AS3930_WAKE_IRQ_PENDING          ((AS3930_WAKE_PxIFG & AS3930_WAKE_PIN) == AS3930_WAKE_PIN)
37  #define AS3930_WAKE_CLEAR_IRQ            (AS3930_WAKE_PxIFG &= ~(AS3930_WAKE_PIN))
38
39
40  void as3930_init(void);
41  void as3930_config_no_pattern(void);
42  void as3930_preset_default(void);
43  void as3930_clear_wakeup(void);
44  uint8_t as3930_get_rssi(void);
45
```

```
46  #endif /* AS3930_H_ */
```

```
1   /*****************************************************************************
2   **      File name       : balancing.h
3   **      Hardware        : BATSEN ZS Klasse 3 v0.5
4   **      Date            : 10/09/2013
5   **      Last Update     : 03/08/2015
6   **      Author          : Nico Sassano
7   **      Description     : Header for the passive balancing
8   *****************************************************************************/
9   #ifndef BALANCING_H_
10  #define BALANCING_H_
11  #include "main.h"
12
13  #define BALANCE_PORT_1_PxDIR      (P2DIR)
14  #define BALANCE_PORT_1_PxOUT      (P2OUT)
15  #define BALANCE_PORT_1_PIN        (BIT3)
16  #define BALANCE_PORT_2_PxDIR      (P3DIR)
17  #define BALANCE_PORT_2_PxOUT      (P3OUT)
18  #define BALANCE_PORT_2_PIN        (BIT5)
19
20  void balancing_init(void);
21  void balancing_on (void);
22  void balancing_off (void);
23  #endif  /* BALANCING_H_ */
```

```
1   /*****************************************************************************
2   **      File name       : cc430.h
3   **      Hardware        : BATSEN ZS Klasse 3 v0.5
4   **      Date            : 24/09/2014
5   **      Last Update     : 09/06/2015
6   **      Author          : Nico Sassano
7   **      Description     : Header for the CC430
8   *****************************************************************************/
9   #ifndef cc430_H_
10  #define cc430_H_
11
12  #include "main.h"
13
14  /*******************************************
15  ** Defines for enabling the interrupts for a
16  ** associated signal on RFIFGx
17  *******************************************/
18  #define RFIE0    0x0001
19  #define RFIE1    0x0002
20  #define RFIE2    0x0004
21  #define RFIE3    0x0008
22  #define RFIE4    0x0010
23  #define RFIE5    0x0020
24  #define RFIE6    0x0040
25  #define RFIE7    0x0080
26  #define RFIE8    0x0100
27  #define RFIE9    0x0200
28  #define RFIE10   0x0400
29  #define RFIE11   0x0800
30  #define RFIE12   0x1000
31  #define RFIE13   0x2000
32  #define RFIE14   0x4000
33  #define RFIE15   0x8000
34
35  /*******************************************
36  ** Defines for the interrupt edge select
37  ** Interrupt edge select bit RFIESx allows to trigger an interrupt on the positive edge (RFIES = 0)
38  ** or on the negative (RFIES = 1) edge of the associated signal.
39  *******************************************/
40  #define RFIES0   0x0001
41  #define RFIES1   0x0002
42  #define RFIES2   0x0004
43  #define RFIES3   0x0008
44  #define RFIES4   0x0010
45  #define RFIES5   0x0020
46  #define RFIES6   0x0040
47  #define RFIES7   0x0080
48  #define RFIES8   0x0100
49  #define RFIES9   0x0200
50  #define RFIES10  0x0400
51  #define RFIES11  0x0800
52  #define RFIES12  0x1000
53  #define RFIES13  0x2000
54  #define RFIES14  0x4000
55  #define RFIES15  0x8000
56
57  /*******************************************
58  ** Defines for detecting the interrupt pending
59  *******************************************/
```

```
60  #define RFIFG0  0x0001
61  #define RFIFG1  0x0002
62  #define RFIFG2  0x0004
63  #define RFIFG3  0x0008
64  #define RFIFG4  0x0010
65  #define RFIFG5  0x0020
66  #define RFIFG6  0x0040
67  #define RFIFG7  0x0080
68  #define RFIFG8  0x0100
69  #define RFIFG9  0x0200
70  #define RFIFG10 0x0400
71  #define RFIFG11 0x0800
72  #define RFIFG12 0x1000
73  #define RFIFG13 0x2000
74  #define RFIFG14 0x4000
75  #define RFIFG15 0x8000
76  /*** Debug ports ***********************************************************************
77  * Here are the definitions for the GDOx debug ports
78  ***************************************************************************************/
79  #define CC430_GDO0_PxDIR              P1DIR
80  #define CC430_GDO0_PxOUT              P1OUT
81  #define CC430_GDO0_PIN                BIT6
82  #define CC430_GDO0_POS                6
83
84  #define CC430_GDO2_PxDIR              P1DIR
85  #define CC430_GDO2_PxOUT              P1OUT
86  #define CC430_GDO2_PIN                BIT7
87  #define CC430_GDO2_POS                7
88
89  /*** Write in the GDO0 register ****/
90  #define CC430_GDO0_ACCESS                        P1SEL |= BIT0
91  #define CC430_GDO0_SET_HIGH                      (WriteSingleReg(IOCFG0, 0x2D)); \
92                                                   (gdo0_state = 0x01)
93
94  #define CC430_GDO0_SET_LOW                       (WriteSingleReg(IOCFG0, 0x2E)); \
95                                                   (gdo0_state = 0x00)
96
97
98  #define IS_GDO0_STATE_LOW                        gdo0_state == 0x00
99  #define IS_GDO0_STATE_HIGH                       gdo0_state == 0x01
100 /*** IRQ RFIES0 ***********************************************************************
101 * RFIES0 based on the GDO0 on the CC1101 module
102 * This is the output on the asynchron mode
103 ***************************************************************************************/
104 #define CC430_GDO0_IRQ_RISING_EDGE               (RF1AIES &= ~RFIES0)     // Positive edge
105 #define CC430_GDO0_IRQ_FALLING_EDGE              (RF1AIES |= RFIES0)      // Negative edge
106 #define CC430_GDO0_IRQ_ENABLE                    (RF1AIE |= RFIES0)
107 #define CC430_GDO0_IRQ_DISABLE                   (RF1AIE &= ~RFIES0)
108 #define CC430_GDO0_IRQ_PENDING                   ((RF1AIFG & RFIES0) == RFIES0)
109 #define CC430_GDO0_IRQ_CLEAR                     (RF1AIFG &= ~(RFIES0))
110 /*** IRQ RFIES2 ***********************************************************************
111 * RFIES2 based on the GDO2 on the CC1101 module
112 * This is the input on the asynchrone mode
113 ***************************************************************************************/
114 #define CC430_GDO2_DIR_IN                        (CC430_GDO2_PxDIR &= ~CC430_GDO2_PIN)
115 #define CC430_GDO2_IN                            ((CC430_GDO2_PxIN & CC430_GDO2_PIN) >> CC430_GDO2_POS)
116 #define CC430_GDO2_DIR_OUT                       (CC430_GDO2_PxDIR |= CC430_GDO2_PIN)
117 #define CC430_GDO2_IRQ_RISING_EDGE               (RF1AIES &= ~RFIES2)     // Positive edge
118 #define CC430_GDO2_IRQ_FALLING_EDGE              (RF1AIES |= RFIES2)      // Negative edge
119 #define CC430_GDO2_IRQ_ENABLE                    (RF1AIE |= RFIE2)
120 #define CC430_GDO2_IRQ_DISABLE                   (RF1AIE &= ~RFIE2)
121 #define CC430_GDO2_IRQ_PENDING                   ((RF1AIFG & RFIFG2) == RFIFG2)
122 #define CC430_GDO2_IRQ_CLEAR                     (RF1AIFG &= ~(RFIFG2))
123 /*** IRQ RFIFG9 ***********************************************************************
124 * Positive edge: Sync word sent or received.
125 * Sync word detected
126 ***************************************************************************************/
127 #define CC430_SYNC_DETECT_IRQ_MODE               (RF1AIES &= ~RFIES9)     // Positive edge
128 #define CC430_SYNC_DETECT_IRQ_ENABLE             (RF1AIE |= RFIE9)
129 #define CC430_SYNC_DETECT_IRQ_DISABLE            (RF1AIE &= ~RFIE9)
130 #define CC430_SYNC_DETECT_IRQ_PENDING            ((RF1AIFG & RFIFG9) == RFIFG9)
131 #define CC430_SYNC_DETECT_CLEAR_IRQ              (RF1AIFG &= ~(RFIFG9))
132 /*** IRQ RFIFG9 ***********************************************************************
133 * Negative edge: End of packet or in RX when optional address check
134 * End of Packet
135 ***************************************************************************************/
136 #define CC430_END_OF_PKT_IRQ_MODE                (RF1AIES |= RFIES9)      // Negative edge
137 #define CC430_END_OF_PKT_IRQ_ENABLE              (RF1AIE |= RFIE9)
138 #define CC430_END_OF_PKT_IRQ_DISABLE             (RF1AIE &= ~RFIE9)
139 #define CC430_END_OF_PKT_IRQ_PENDING             ((RF1AIFG & RFIFG9) == RFIFG9)
140 #define CC430_END_OF_PKT_CLEAR_IRQ               (RF1AIFG &= ~(RFIFG9))
141 // ************************************************************************************
142 #define CC430_CRC_OK_IRQ_EDGE                    (RF1AIES &= ~RFIES10)
143 #define CC430_CRC_OK_IRQ_ENABLE                  (RF1AIE |= RFIE10)
144 #define CC430_CRC_OK_IRQ_DISABLE                 (RF1AIE &= ~RFIE10)
```

```
145 #define CC430_CRC_OK_IRQ_PENDING            ((RF1AIFG & RFIFG10) == RFIFG10)
146 #define CC430_CRC_OK_CLEAR_IRQ              (RF1AIFG &= ~(RFIFG10))
147
148 #define GDx_TriState 0x2E
149
150 #define CC430_CRC_EN 0x04
151
152 void cc430_tx(uint8_t, uint8_t *);
153 void cc430_rx(uint8_t);
154 void tx_carrier(void);
155 void cc430_set_tx(uint8_t);
156 void cc430_set_rx(uint8_t);
157 void cc430_init_rx(void);
158 void cc430_init_tx(void);
159 void cc430_init_burst(void);
160 void cc430_reset(void);
161 void cc430_config_no_packet(void);
162 void cc430_config_no_packet_rx(void);
163 void cc430_config_no_packet_tx(void);
164 void cc430_config_cali_burst(void);
165 void cc430_config_packet(uint8_t brate);
166 void cc430_fill_tx_fifo(uint8_t * buffer, uint8_t length);
167 void cc430_read_rx_fifo(uint8_t * buffer, uint8_t length);
168 void cc430_enable_crc(void);
169 void cc430_disable_crc(void);
170 void cc430_clear_tx_fifo(void);
171 void cc430_clear_rx_fifo(void);
172 uint8_t cc430_get_rxbytes(void);
173 void cc430_sleep(void);
174 void cc430_idle(void);
175 void cc430_tx_carrier(void);
176 void cc430_tx_start(uint8_t);
177 void cc430_rx_start(uint8_t);
178 void cc430_burst_rx(void);
179 void cc430_burst_tx(void);
180 uint8_t cc430_get_partnum(void);
181 uint8_t cc430_get_version(void);
182 uint8_t cc430_get_marcstate(void);
183
184 void cc430_config_cali(void);
185
186 #endif /* cc430_H_ */
```

```
1  /******************************************************************************
2  **     File name      : clk.h
3  **     Hardware       : BATSEN ZS Klasse 3 v0.5
4  **     Date           : 24/04/2013
5  **     Last Update    : 03/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : Header for clk
8  ******************************************************************************/
9  #include "main.h"
10
11 #ifndef CLK_H_
12 #define CLK_H_
13
14 #define DCO_MP_1MHZ 29
15 #define DCO_MP_8MHZ 242
16 #define DCO_MP_12MHZ 365
17 #define DCO_MP_16MHZ 487
18
19 void clk_init_startup(void);
20 void clk_init_MHz(uint8_t mhz);
21
22 #endif /* CLK_H_ */
```

```
1  /******************************************************************************
2  **     File name      : delay.h
3  **     Hardware       : BATSEN ZS Klasse 3 v0.5
4  **     Date           : 09/04/2013
5  **     Last Update    : 03/08/2015
6  **     Author         : Nico Sassano
7  **     Description    : Header for delay
8  ******************************************************************************/
9  #ifndef DELAY_H_
10 #define DELAY_H_
11
12 #include "main.h"
13
14 #define DELAY_CYCLES_PER_MS              259
15 #define DELAY_CYCLES_PER_100US_1MHZ      22
16 #define DELAY_CYCLES_PER_10US_16MHZ      36
17
18 void delay_10us_16MHz(uint16_t);
19
```

```
20  #endif /* DELAY_H_ */
```

```
1   /**********************************************************************************
2   **     File name       : i2c.h
3   **     Hardware        : BATSEN ZS Klasse 3 v0.5
4   **     Date            : 09/04/2013
5   **     Last Update     : 03/08/2015
6   **     Author          : Nico Sassano
7   **     Description     : Header for I2C Interface
8   *********************************************************************************/
9   #ifndef I2C_H_
10  #define I2C_H_
11
12  #include "main.h"
13
14  /**********************************************************************************
15  *   Defines
16  *********************************************************************************/
17   #define I2C_SDA_PxSEL   (P1SEL)
18  #define I2C_SDA_PIN     (BIT3)
19  #define I2C_SCL_PxSEL   (P1SEL)
20  #define I2C_SCL_PIN     (BIT2)
21
22  #define UCB0TXIFG    UCTXIFG
23  #define UCB0RXIFG    UCRXIFG
24  #define UCB0I2CIE    UCB0IE
25  #define IFG2         UCB0IFG
26
27  /**********************************************************************************
28  *   Prototyping declaration
29  *********************************************************************************/
30  unsigned char *PTxData;                        // Pointer to TX data
31  unsigned char TXByteCtr;
32
33  void i2c_init(void);
34  void i2c_write(uint8_t , uint8_t , uint8_t *);
35  void i2c_read(uint8_t , uint8_t , uint8_t *);
36
37  #endif
```

```
1   /**********************************************************************************
2   **     File name       : init.h
3   **     Hardware        : BATSEN ZS Klasse 3 v0.5
4   **     Date            : 05/03/2013
5   **     Last Update     : 03/08/2015
6   **     Author          : Nico Sassano
7   **     Description     : Header for init
8   *********************************************************************************/
9   #ifndef INIT_H_
10  #define INIT_H_
11
12  void init(void);
13  void init_for_sleep(void);
14
15  #endif /* INIT_H_ */
```

```
1   /**********************************************************************************
2   **     File name       : led.h
3   **     Hardware        : BATSEN ZS Klasse 3 v0.5
4   **     Date            : 05/03/2013
5   **     Last Update     : 03/08/2015
6   **     Author          : Phillip Durdaut and Nico Sassano
7   **     Description     : Header for LED
8   *********************************************************************************/
9   #ifndef LED_H_
10  #define LED_H_
11
12  #include "main.h"
13
14  #define LED_RED_PxDIR          (P3DIR)
15  #define LED_RED_PxOUT          (P3OUT)
16  #define LED_RED_PIN            (BIT4)
17  #define LED_YELLOW_PxDIR       (P3DIR)
18  #define LED_YELLOW_PxOUT       (P3OUT)
19  #define LED_YELLOW_PIN         (BIT3)
20  #define LED_GREEN_PxDIR        (P3DIR)
21  #define LED_GREEN_PxOUT        (P3OUT)
22  #define LED_GREEN_PIN          (BIT2)
23
24  typedef enum {
25      LED_AWAKE = 0,
26      LED_TX,
27      LED_RX,
28      LED_ALL
```

```
29  }LED_t;
30
31  void led_init(void) ;
32  void led_on(LED_t led) ;
33  void led_off(LED_t led) ;
34  void led_toggle(LED_t led) ;
35
36  #endif /* LED_H_ */
```

```
1   /*********************************************************************************
2   **     File name       : main.h
3   **     Hardware        : BATSEN ZS Klasse 3 v0.5
4   **     Date            : 06/03/2013
5   **     Last Update     : 03/05/2015
6   **     Author          : Nico Sassano
7   **     Description     : Header file for the main routine
8   *********************************************************************************/
9   #ifndef MAIN_H_
10  #define MAIN_H_
11
12  #include <cc430f5137.h>
13  #include <signal.h>
14  #include <stdio.h>
15  #include <math.h>
16
17  #include "types.h"
18  #include "adc12.h"
19  #include "adg918.h"
20  #include "as3930.h"
21  #include "balancing.h"
22  #include "cc430.h"
23  #include "delay.h"
24  #include "led.h"
25  #include "i2c.h"
26  #include "temp_sensor.h"
27  #include "timer.h"
28  #include "clk.h"
29  #include "init.h"
30  #include "RF1A.h"
31  #include "hal_pmm.h"
32  #include "spi.h"
33  #include "AD7691.h"
34  #include "AD5270.h"
35
36  /*--------------------------------------------------------------------------------
37     Fix error in msp430x23x.h
38  --------------------------------------------------------------------------------*/
39
40  //#define __MSP430_HAS_ADC12__
41  //#include <msp430/adc12.h>
42
43  /*--------------------------------------------------------------------------------
44     Defines
45  --------------------------------------------------------------------------------*/
46  #define ENABLE_LEDS
47
48  /*****************************************************
49  * Port defines
50  *****************************************************/
51  //Erweiterungsplatine
52  #define TPS61201_PxDIR   (P2DIR)
53  #define TPS61201_PxOUT   (P2OUT)
54  #define TPS61201_PIN     (BIT0)
55
56  //#define TPS61201_PxDIR      (P3DIR)
57  //#define TPS61201_PxOUT      (P3OUT)
58  //#define TPS61201_PIN    (BIT1)
59
60  #define TPS61201_INIT TPS61201_PxDIR |= TPS61201_PIN ;
61  #define TPS61201_ENABLE TPS61201_PxOUT |= TPS61201_PIN
62  #define TPS61201_DISABLE TPS61201_PxOUT &=~TPS61201_PIN
63
64
65
66  // Testport deklaration für v0.5
67  #define TEST_PORT_PxDIR                                   (P2DIR)
68  #define TEST_PORT_PxOUT                                   (P2OUT)
69  #define TEST_PORT_PxSEL                                   (P2SEL)
70  #define TEST_PORT_PIN                                     (BIT0)
71
72  #define TEST_PORT_SET_OUTPUT         TEST_PORT_PxDIR |= TEST_PORT_PxOUT;
73
74  //#define TEST_PIN_ON            TEST_PORT_PxOUT |= TEST_PORT_PIN
75  //#define TEST_PIN_OFF           TEST_PORT_PxOUT &=~TEST_PORT_PIN
76
```

```
77
78  #define ON  0x01
79  #define OFF 0x00
80
81  //#define ENABLE_LEDS
82
83  #define DOWNLINK_DATA_BYTES            4//6        // Incl. address byte
84  #define UPLINK_DATA_BYTES              7//11       // Incl. address byte
85
86  // Frequency offset added to the base frequency (in units of 1.59 kHz - 1.65 kHz)
87  #define FREQUENCY_OFFSET               0
88
89  // Address definitions
90  #define BROADCAST                                          0x00
91  #define ADDRESS_BASE_STATION                               0xFF
92  #define ADDRESS_THIS_SENSOR                                0x02
93
94  /** Downlink commands ***********************************/
95  #define COMMAND_WAKEUP                                     0x00
96  #define COMMAND_WAKEUP_DONE                                0x01
97  #define COMMAND_DOWNLINK_IS_AWAKE                          0x02
98  #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE                    0x03
99  #define COMMAND_DOWNLINK_SEND_VOLTAGE                      0x04
100 #define COMMAND_DOWNLINK_SLEEP                             0x05
101 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE                0x06
102 #define COMMAND_DOWNLINK_SEND_TEMPERATURE                  0x07
103 #define COMMAND_DOWNLINK_BALANCING_ON                      0x08
104 #define COMMAND_DOWNLINK_BALANCING_OFF                     0x09
105 #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE          0x0A
106 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE        0x0B
107 #define COMMAND_DOWNLINK_BURST_MODE                        0x0C
108 #define COMMAND_DOWNLINK_BURST_DATA_RQ                     0x0D
109 #define COMMAND_DOWNLINK_BURST_DATA_RX                     0x0E
110 #define COMMAND_DOWNLINK_BURST_CHECK                       0x0F
111 #define COMMAND_DOWNLINK_CONFIG_SET                        0x10
112 #define COMMAND_DOWNLINK_CONFIG                            0x11
113 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102         0x12
114 #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102           0x13
115 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102_CALI    0x14
116 #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI      0x15
117 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430         0x16
118 #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430           0x17
119 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430_CALI    0x18
120 #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI      0x19
121 #define COMMAND_DOWNLINK_CALIBRATION_TMP102                0x1A
122 #define COMMAND_DOWNLINK_CALIBRATION_MSP430                0x1B
123 #define COMMAND_DOWNLINK_CALIBRATION_ADC                   0x1C
124 #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE_ALL      0x1D
125 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE_ALL    0x1E
126 #define COMMAND_CALI_BURST                                 0x1F
127 #define COMMAND_CLK_OK                                     0x21
128 #define COMMAND_CLK_UP                                     0x20
129 #define COMMAND_CLK_DOWN                                   0x22
130 #define COMMAND_CALI_CLK                                   0x23
131 #define COMMAND_SW                                         0x24
132 #define COMMAND_GOERTZEL                                   0x25
133 #define COMMAND_AD5270_1                                   0x26
134 #define COMMAND_AD5270_2                                   0x27
135 #define COMMAND_SET_PREPROCESSING                          0x28
136 #define COMMAND_GOERTZEL_FREQ                              0x29
137 #define COMMAND_GOERTZEL_STIMULI_FREQ                      0x2A
138 #define COMMAND_GOERTZEL_STIMULI_FREQ2                     0x2B
139 #define COMMAND_GOERTZEL_NUMB_PERI                         0x2C
140
141
142 #define COMMAND_BACK_FROM_BURST                            0xFD
143 #define COMMAND_WAIT                                       0xFE
144 /*** TEST DOWNLINKS ***************************************/
145 #define COMMAND_DOWNLINK_ERROR_TEST                        0x99
146
147 #define COMMAND_DOWNLINK_UNKWOWN                           0xFF
148
149 #define HEADER_LENGTH                                      0x04
150
151 /*******************************************************
152 ** Frequenzen für die Burstmessung
153 *******************************************************/
154 #define BURST_FREQ_100000HZ                                0x34
155 #define BURST_FREQ_95000HZ                                 0x33
156 #define BURST_FREQ_90000HZ                                 0x32
157 #define BURST_FREQ_85000HZ                                 0x31
158 #define BURST_FREQ_80000HZ                                 0x30
159 #define BURST_FREQ_75000HZ                                 0x2F
160 #define BURST_FREQ_70000HZ                                 0x2E
161 #define BURST_FREQ_65000HZ                                 0x2D
```

```
162 #define BURST_FREQ_60000HZ                                   0x2C
163 #define BURST_FREQ_55000HZ                                   0x2B
164 #define BURST_FREQ_50000HZ                                   0x2A
165 #define BURST_FREQ_45000HZ                                   0x29
166 #define BURST_FREQ_40000HZ                                   0x28
167 #define BURST_FREQ_38000HZ                                   0x27
168 #define BURST_FREQ_36000HZ                                   0x26
169 #define BURST_FREQ_34000HZ                                   0x25
170 #define BURST_FREQ_32000HZ                                   0x24
171 #define BURST_FREQ_30000HZ                                   0x23
172 #define BURST_FREQ_28000HZ                                   0x22
173 #define BURST_FREQ_26000HZ                                   0x21
174 #define BURST_FREQ_24000HZ                                   0x20
175 #define BURST_FREQ_22000HZ                                   0x1F
176 #define BURST_FREQ_20000HZ                                   0x1E
177 #define BURST_FREQ_18000HZ                                   0x1C
178 #define BURST_FREQ_16000HZ                                   0x1B
179 #define BURST_FREQ_14000HZ                                   0x1A
180 #define BURST_FREQ_12000HZ                                   0x19
181 #define BURST_FREQ_10000HZ                                   0x18
182 #define BURST_FREQ_8000HZ                                    0x17
183 #define BURST_FREQ_6000HZ                                    0x16
184 #define BURST_FREQ_4000HZ                                    0x15
185 #define BURST_FREQ_2000HZ                                    0x14
186 #define BURST_FREQ_1000HZ                                    0x13
187 #define BURST_FREQ_950HZ                                     0x12
188 #define BURST_FREQ_900HZ                                     0x11
189 #define BURST_FREQ_850HZ                                     0x10
190 #define BURST_FREQ_800HZ                                     0x0F
191 #define BURST_FREQ_750HZ                                     0x0E
192 #define BURST_FREQ_700HZ                                     0x0D
193 #define BURST_FREQ_650HZ                                     0x0C
194 #define BURST_FREQ_600HZ                                     0x0B
195 #define BURST_FREQ_550HZ                                     0x0A
196 #define BURST_FREQ_500HZ                                     0x09
197 #define BURST_FREQ_450HZ                                     0x08
198 #define BURST_FREQ_400HZ                                     0x07
199 #define BURST_FREQ_350HZ                                     0x06
200 #define BURST_FREQ_300HZ                                     0x05
201 #define BURST_FREQ_250HZ                                     0x04
202 #define BURST_FREQ_200HZ                                     0x03
203 #define BURST_FREQ_150HZ                                     0x02
204 #define BURST_FREQ_100HZ                                     0x01
205 #define BURST_FREQ_50HZ                                      0x00
206
207 #define BURST_VALUES                                         1800
208 #define BURST_VALUES_50                                      0x01
209 #define BURST_VALUES_100                                     0x02
210 #define BURST_VALUES_150                                     0x03
211 #define BURST_VALUES_200                                     0x04
212 #define BURST_VALUES_250                                     0x05
213 #define BURST_VALUES_300                                     0x06
214 #define BURST_VALUES_350                                     0x07
215 #define BURST_VALUES_400                                     0x08
216 #define BURST_VALUES_450                                     0x09
217 #define BURST_VALUES_500                                     0x0A
218 #define BURST_VALUES_550                                     0x0B
219 #define BURST_VALUES_600                                     0x0C
220 #define BURST_VALUES_650                                     0x0D
221 #define BURST_VALUES_700                                     0x0E
222 #define BURST_VALUES_750                                     0x0F
223 #define BURST_VALUES_800                                     0x10
224 #define BURST_VALUES_850                                     0x11
225 #define BURST_VALUES_900                                     0x12
226 #define BURST_VALUES_1000                                    0x13
227 #define BURST_VALUES_1500                                    0x14
228 #define BURST_VALUES_1900                                    0x15
229 #define BURST_VALUES_2000                                    0x16
230 #define BURST_VALUES_2500                                    0x17
231
232 #define BURST_SIGNAL_FREQ_01                                 0x01
233 #define BURST_SIGNAL_FREQ_05                                 0x02
234 #define BURST_SIGNAL_FREQ_1                                  0x03
235 #define BURST_SIGNAL_FREQ_5                                  0x04
236 #define BURST_SIGNAL_FREQ_10                                 0x05
237 #define BURST_SIGNAL_FREQ_20                                 0x06
238 #define BURST_SIGNAL_FREQ_30                                 0x07
239 #define BURST_SIGNAL_FREQ_40                                 0x08
240 #define BURST_SIGNAL_FREQ_50                                 0x09
241 #define BURST_SIGNAL_FREQ_60                                 0x0A
242 #define BURST_SIGNAL_FREQ_70                                 0x0B
243 #define BURST_SIGNAL_FREQ_80                                 0x0C
244 #define BURST_SIGNAL_FREQ_90                                 0x0D
245 #define BURST_SIGNAL_FREQ_100                                0x0E
246 #define BURST_SIGNAL_FREQ_200                                0x0F
```

```
247 #define BURST_SIGNAL_FREQ_300                    0x10
248 #define BURST_SIGNAL_FREQ_400                    0x11
249 #define BURST_SIGNAL_FREQ_500                    0x12
250 #define BURST_SIGNAL_FREQ_600                    0x13
251 #define BURST_SIGNAL_FREQ_700                    0x14
252 #define BURST_SIGNAL_FREQ_800                    0x15
253 #define BURST_SIGNAL_FREQ_900                    0x16
254 #define BURST_SIGNAL_FREQ_1000                   0x17
255
256
257
258 /*******************************************************
259 ** Interrupt Modes
260 *******************************************************/
261 #define TX_MODE                              0x00
262 #define RX_MODE                              0x01
263 #define BURST_MODE                           0x02
264 #define BURST_CALI_MODE                      0x03
265
266 #define ALERT_HIGH                           0x00
267 #define ALERT_LOW                            0x01
268
269 #define TEMP_NORMAL                          0x00
270 #define TEMP_HIGH                            0x01
271
272 /******************************************************
273 ** TIMER Macros
274 ******************************************************/
275 #define TIMERA_BALANCING                     0x01
276 #define TIMERA_DELAY                         0x02
277
278 #define TIMERB_BURST                         0x01
279 #define TIMERB_BURST_END                     0x02
280 #define TIMERB_DELAY                         0x03
281 #define TIMERB_CLOCK_CALI                    0x04
282
283 // Clock frequencies
284 #define DCOCLK                    1000000  // DCO Clock frequency
285 #define MCLK                      1000000  // Main System Clock frequency
286 #define SMCLK                     1000000  // Sub System Clock frequency
287
288 /******************************************************
289 ** IRQ Macros
290 ******************************************************/
291 #define IRQ_SET_RX                  (irq_mode = RX_MODE)
292 #define IRQ_IS_RX                   (irq_mode == RX_MODE)
293 #define IRQ_SET_TX                  (irq_mode = TX_MODE)
294 #define IRQ_IS_TX                   (irq_mode == TX_MODE)
295 #define IRQ_SET_BURST               (irq_mode = BURST_MODE)
296 #define IRQ_IS_BURST                (irq_mode == BURST_MODE)
297 #define IRQ_SET_CALI_BURST          (irq_mode = BURST_CALI_MODE)
298 #define IRQ_IS_CALI_BURST           (irq_mode == BURST_CALI_MODE)
299
300 /*******************************************************
301 ** IRQ_ALERT_SET_HIGH    bei zu hoher Temperatur
302 ** IRQ_ALERT_IS_LOW      bei normaler Temperatur
303 *******************************************************/
304 #define IRQ_ALERT_SET_HIGH          (irq_alert = ALERT_HIGH)
305 #define IRQ_ALERT_IS_HIGH           (irq_alert == ALERT_HIGH)
306 #define IRQ_ALERT_SET_LOW           (irq_alert = ALERT_LOW)
307 #define IRQ_ALERT_IS_LOW            (irq_alert == ALERT_LOW)
308
309 #define TEMP_IS_NORMAL              (temp_state == TEMP_NORMAL)
310 #define TEMP_SET_NORMAL             (temp_state = TEMP_NORMAL)
311 #define TEMP_IS_HIGH                (temp_state == TEMP_HIGH)
312 #define TEMP_SET_HIGH               (temp_state = TEMP_HIGH)
313
314 /*******************************************************
315 ** IRQ TIMER
316 *******************************************************/
317 #define IRQ_TIMERA_SET_BALANCING     (irq_timera = TIMERA_BALANCING)
318 #define IRQ_TIMERA_UNSET_BALANCING   (irq_timera = 0x00)
319 #define IRQ_TIMERA_IS_BALANCING      (irq_timera == TIMERA_BALANCING)
320 #define IRQ_TIMERA_IS_DELAY          (irq_timera == TIMERA_DELAY)
321
322 #define IRQ_TIMERB_SET_CLOCK_CALI     (irq_timerb = TIMERB_CLOCK_CALI)
323 #define IRQ_TIMERB_UNSET_CLOCK_CALI   (irq_timerb = 0x00)
324 #define IRQ_TIMERB_IS_CLOCK_CALI      (irq_timerb == TIMERB_CLOCK_CALI)
325
326 #define IRQ_TIMERB_SET_BURST         (irq_timerb = TIMERB_BURST)
327 #define IRQ_TIMERB_UNSET_BURST       (irq_timerb = 0x00)
328 #define IRQ_TIMERB_IS_BURST          (irq_timerb == TIMERB_BURST)
329
330 #define IRQ_TIMERB_SET_DELAY         (irq_timerb = TIMERB_DELAY)
331 #define IRQ_TIMERB_UNSET_DELAY       (irq_timerb = 0x00)
```

```
332 #define IRQ_TIMERB_IS_DELAY          (irq_timerb == TIMERB_DELAY)
333
334 /*******************************************************
335 ** Burst Flag Macros
336 *******************************************************/
337 #define BURST_ERROR_FLAG_SET         (burst_error_flag = 0x01)
338 #define BURST_ERROR_FLAG_UNSET       (burst_error_flag = 0x00)
339 #define BURST_ERROR_FLAG_IS_SET      (burst_error_flag == 0x01)
340 #define BURST_ERROR_FLAG_IS_UNSET    (burst_error_flag == 0x00)
341
342 #define BURST_CALI_FLAG_SET         (burst_cali_flag = 0x01)
343 #define BURST_CALI_FLAG_UNSET       (burst_cali_flag = 0x00)
344 #define BURST_CALI_FLAG_IS_SET      (burst_cali_flag == 0x01)
345 #define BURST_CALI_FLAG_IS_UNSET    (burst_cali_flag == 0x00)
346
347 /*******************************************************
348  ** ADC AUSWAHL
349  *******************************************************/
350 #define ADC12_INT_SELECT             0x00
351 #define AD7691_EXT_SELECT            0x01
352
353 #define ADC12_INT_IS_SELECT          adc_mode == ADC12_INT_SELECT
354 #define AD7691_EXT_IS_SELECT         adc_mode == AD7691_EXT_SELECT
355
356 /*----------------------------------------------------------------------------
357    Types
358 ----------------------------------------------------------------------------*/
359
360 typedef enum {  S_INIT,
361                 S_SLEEP,
362                 S_RX,
363                 S_WAKEUP,
364                 S_WAKEUP_RX,
365                 S_WAKEUP_DONE,
366                 S_TX_AWAKE,
367                 S_SAMPLE,
368                 S_TX_SAMPLE,
369                 S_BALANC_FIRST_START,
370                 S_BALANC_ON,
371                 S_BALANC_OFF
372               } state_t ;
373
374
375 //uint16_t temp_high = 0x01A0; //0x01A0 -> 26°C
376 //uint16_t temp_low = 0x0190; // 0x0190 -> 25°C
377
378 /*----------------------------------------------------------------------------
379    Macros
380 ----------------------------------------------------------------------------*/
381
382 #define ENTER_LPM4                        __bis_SR_register(LPM4_bits + GIE);
383 #define EXIT_LPM4                         __bic_SR_register_on_exit(LPM4_bits);
384
385
386
387 #endif /* MAIN_H_ */
```

```
 1 /*****************************************************************************
 2 **     File name      : spi.h
 3 **     Hardware       : BATSEN ZS Klasse 3 v0.5
 4 **     Date           : 21/12/2014
 5 **     Last Update    : 03/08/2015
 6 **     Author         : Nico Sassano
 7 **     Description    : Header file for the spi routine
 8 *****************************************************************************/
 9 #ifndef SPI_H_
10 #define SPI_H_
11
12 #include "main.h"
13
14 #define LTC2376                      0x00
15 #define AS3930                       0x01
16
17 // Defines for the LTC2376 ADC
18 #define SPI_LTC2376_CS_PxDIR         P5DIR
19 #define SPI_LTC2376_CS_PxOUT         P5OUT
20 #define SPI_LTC2376_CS_PIN           BIT0
21
22 #define SPI_PxSEL                    P1SEL
23 #define SPI_PxDIR                    P1DIR
24 #define SPI_PxIN                     P1IN
25 #define SPI_MOSI_PIN                 BIT6
26 #define SPI_MISO_PIN                 BIT5
27 #define SPI_CLK_PIN                  BIT7
28
```

```
29  #define SPI_LTC2376_CS_ENABLE        SPI_LTC2376_CS_PxOUT &= ~SPI_LTC2376_CS_PIN
30  #define SPI_LTC2376_CS_DISABLE       SPI_LTC2376_CS_PxOUT |= SPI_LTC2376_CS_PIN
31
32  // Defines for the AS3930
33  #define SPI_AS3930_CS_PxDIR          P5DIR
34  #define SPI_AS3930_CS_PxOUT          P5OUT
35  #define SPI_AS3930_CS_PIN            BIT0
36
37  #define SPI_PxSEL                    P1SEL
38  #define SPI_PxDIR                    P1DIR
39  #define SPI_PxOUT                    P1OUT
40  #define SPI_PxIN                     P1IN
41  #define SPI_MOSI_PIN                 BIT6
42  #define SPI_MISO_PIN                 BIT5
43  #define SPI_CLK_PIN                  BIT7
44
45  #define SPI_AS3930_CS_ENABLE         SPI_AS3930_CS_PxOUT &= ~SPI_AS3930_CS_PIN
46  #define SPI_AS3930_CS_DISABLE        SPI_AS3930_CS_PxOUT |= SPI_AS3930_CS_PIN
47
48  void init_spi(void);
49  void spi_write_register(uint8_t, uint8_t, uint8_t);
50  char spi_read_register(uint8_t, uint8_t);
51  void spi_send_command(uint8_t, uint8_t);
52
53  #endif /* SPI_H_ */
```

```
1   /*********************************************************************************
2   **    File name      : temp_sensor.h
3   **    Hardware       : BATSEN ZS Klasse 3 v0.5
4   **    Date           : 07/03/2013
5   **    Last Update    : 03/08/2015
6   **    Author         : Nico Sassano
7   **    Description    : Header file for the Temperatur Sensor routine
8   *********************************************************************************/
9   #ifndef TEMP_SENSOR_H_
10  #define TEMP_SENSOR_H_
11
12  #include "main.h"
13
14  /**** Adress of the Temp. Sensor ***************************************************/
15  #define ADDR_TMP102      0x48
16
17  #define TMP102_ALERT_PxREN          P1REN
18  #define TMP102_ALERT_PxOUT          P1OUT
19  #define TMP102_ALERT_PxDIR          P1DIR
20  #define TMP102_ALERT_PxIES          P1IES
21  #define TMP102_ALERT_PxIE           P1IE
22  #define TMP102_ALERT_PxIFG          P1IFG
23  #define TMP102_ALERT_PIN            BIT1
24
25  #define TMP102_ALERT_REN_EN                  (TMP102_ALERT_PxREN |= TMP102_ALERT_PIN)
26  #define TMP102_ALERT_SET_PULLUP              (TMP102_ALERT_PxOUT |= TMP102_ALERT_PIN)
27  #define TMP102_ALERT_SET_PULLDOWN            (TMP102_ALERT_PxOUT &= ~TMP102_ALERT_PIN)
28  #define TMP102_ALERT_DIR_IN                  (TMP102_ALERT_PxDIR &= ~TMP102_ALERT_PIN)
29  #define TMP102_ALERT_IRQ_RISING_EDGE         (TMP102_ALERT_PxIES &= ~TMP102_ALERT_PIN)
30  #define TMP102_ALERT_IRQ_FALLING_EDGE        (TMP102_ALERT_PxIES |= TMP102_ALERT_PIN)
31  #define TMP102_ALERT_IRQ_ENABLE              (TMP102_ALERT_PxIE  |= TMP102_ALERT_PIN)
32  #define TMP102_ALERT_IRQ_DISABLE             (TMP102_ALERT_PxIE  &= ~TMP102_ALERT_PIN)
33  #define TMP102_ALERT_IRQ_PENDING             ((TMP102_ALERT_PxIFG & TMP102_ALERT_PIN) == TMP102_ALERT_PIN)
34  #define TMP102_ALERT_CLEAR_IRQ               (TMP102_ALERT_PxIFG &= ~(TMP102_ALERT_PIN))
35
36  #define TMP102_SET_IRQ_RISING_EDGE           (TMP102_ALERT_DIR_IN); \
37                                               (TMP102_ALERT_REN_EN); \
38                                               (TMP102_ALERT_SET_PULLUP); \
39                                               (TMP102_ALERT_IRQ_RISING_EDGE); \
40
41  #define TMP102_SET_IRQ_FALLING_EDGE          (TMP102_ALERT_DIR_IN); \
42                                               (TMP102_ALERT_REN_EN); \
43                                               (TMP102_ALERT_SET_PULLUP); \
44                                               (TMP102_ALERT_IRQ_FALLING_EDGE); \
45
46
47  /**** Register adresses ************************************************************/
48  #define TMP102_REG_TEMP          0x00     // Temperatur
49  #define TMP102_REG_CONF          0x01     // Configuration
50  #define TMP102_REG_HIGH          0x02     // Temperatur Low
51  #define TMP102_REG_LOW           0x03     // Temperatur High
52
53  #define TMP102_REG_OS            0x8000
54  #define TMP102_REG_R1            0x4000
55  #define TMP102_REG_R0            0x2000
56  #define TMP102_REG_F1            0x1000
57  #define TMP102_REG_F0            0x0800
58  #define TMP102_REG_POL           0x0400
59  #define TMP102_REG_TM            0x0200
```

```
60  #define TMP102_REG_SD            0x0100
61
62  #define TMP102_REG_CR1           0x0080
63  #define TMP102_REG_CR0           0x0040
64  #define TMP102_REG_AL            0x0020
65  #define TMP102_REG_EM            0x0010
66
67  #define TMP102_EM_OFF            (config_value &=~TMP102_REG_EM)
68  #define TMP102_EM_ON             (config_value |= TMP102_REG_EM)
69  #define TMP102_SD_OFF            (config_value &=~TMP102_REG_SD)
70  #define TMP102_SD_ON             (config_value |= TMP102_REG_SD)
71  #define TMP102_COMPERATOR_MODE   (config_value &=~TMP102_REG_TM)
72  #define TMP102_INTERRUPT_MODE    (config_value |= TMP102_REG_TM)
73  #define TMP102_POL_INV           (config_value &=~TMP102_REG_POL)
74  #define TMP102_POL_NORM          (config_value |= TMP102_REG_POL)
75
76  #define TMP102_FAULTS_1          (config_value &= ~(TMP102_REG_F1)); \
77                                   (config_value &= ~(TMP102_REG_F0));
78  #define TMP102_FAULTS_2          (config_value &= ~(TMP102_REG_F1)); \
79                                   (config_value |=  (TMP102_REG_F0));
80  #define TMP102_FAULTS_4          (config_value |=  (TMP102_REG_F1)); \
81                                   (config_value &= ~(TMP102_REG_F0));
82  #define TMP102_FAULTS_6          (config_value |=  (TMP102_REG_F1)); \
83                                   (config_value |=  (TMP102_REG_F0));
84
85  #define TMP102_CON_RATE_1        (config_value &= ~(TMP102_REG_CR1)); \
86                                   (config_value &= ~(TMP102_REG_CR0));
87  #define TMP102_CON_RATE_2        (config_value &= ~(TMP102_REG_CR1)); \
88                                   (config_value |=  (TMP102_REG_CR0));
89  #define TMP102_CON_RATE_4        (config_value |=  (TMP102_REG_CR1)); \
90                                   (config_value &= ~(TMP102_REG_CR0));
91  #define TMP102_CON_RATE_6        (config_value |=  (TMP102_REG_CR1)); \
92                                   (config_value |=  (TMP102_REG_CR0));
93
94  /**** Prototyp declaration ******************************************************/
95  void temp_sensor_init(void);
96  void temp_sensor_get_contr_reg(void);
97  uint16_t temp_sensor_get_temp(void);
98  void temp_sensor_config_reg(void);
99  void temp_sensor_set_alert(uint16_t, uint16_t);
100
101 #endif /* TEMP_SENSOR_H_ */
```

```
1   /*****************************************************************************
2   **    File name      : imer.h
3   **    Hardware       : BATSEN ZS Klasse 3 v0.5
4   **    Date           : 13/03/2013
5   **    Last Update    : 03/08/2015
6   **    Author         : Nico Sassano
7   **    Description    : Header file for the timer routine
8   *****************************************************************************/
9   #ifndef TIMER_H_
10  #define TIMER_H_
11
12  #include "main.h"
13
14  /**************************************************************
15  **  Bending Defines for CC430 use EM 31.07.2014
16  **************************************************************/
17  #define TACCR0 TA0CCR0
18  #define TACCR1 TA0CCR1
19  #define TACCTL0 TA0CCTL0
20  #define TACCTL1 TA0CCTL1
21  #define TACTL TA0CTL
22  #define TAR TA0R
23  #define TAEX0 TA0EX0
24
25  #define TBCCR0 TA1CCR0
26  #define TBCCR1 TA1CCR1
27  #define TBCCTL0 TA1CCTL0
28  #define TBCCTL1 TA1CCTL1
29  #define TBCLR TACLR
30  #define TBCTL TA1CTL
31  #define TBSSEL_2 TASSEL_2
32
33  /**************************************************************
34  **  Defines Timer A
35  **************************************************************/
36  #define TIMER_A_START_UP_MODE       (TACTL |= MC0)
37  #define TIMER_A_START_CM_MODE       (TACTL |= MC1)
38  #define TIMER_A_START_UD_MODE       (TACTL |=(MC1 | MC0))
39
40  #define TIMER_A_STOP                (TACTL &= ~(MC0 | MC1))
41
42  #define TIMER_A_RESET               (TACTL = TACLR)
```

```
43  #define TIMER_A_FLAG_RESET0           (TACCTL0 &= ~(CCIFG|COV))
44
45  #define TIMER_A_SOURCE_TACLK          (TACTL = TASSEL_0)
46  #define TIMER_A_SOURCE_ACLK           (TACTL = TASSEL_1)
47  #define TIMER_A_SOURCE_SMCLK          (TACTL = TASSEL_2)
48
49  #define TIMER_A_SOURCE_DIV_8          (TACTL |= (ID1 | ID0))
50  #define TIMER_A_SOURCE_DIV_4          (TACTL |= ID1); \
51                                        (TACTL &= ~ID0);
52  #define TIMER_A_SOURCE_DIV_2          (TACTL &= ~ID1); \
53                                        (TACTL |= ID0);
54  #define TIMER_A_SOURCE_DIV_1          (TACTL &= ~ID1); \
55                                        (TACTL &= ~ID0);
56
57  #define TIMER_A_0_OUTMOD_RESET_SET    (TACCTL0 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
58  #define TIMER_A_1_OUTMOD_RESET_SET    (TACCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
59  #define TIMER_A_2_OUTMOD_RESET_SET    (TACCTL2 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
60
61  #define TIMER_A_0_OUTMOD_TOGGLE_SET (TACCTL0 |= (OUTMOD2 | OUTMOD1)); \
62                                        (TACCTL0 &= ~OUTMOD0);
63  #define TIMER_A_1_OUTMOD_TOGGLE_SET (TACCTL1 |= (OUTMOD2 | OUTMOD1)); \
64                                        (TACCTL1 &= ~OUTMOD0);
65  #define TIMER_A_2_OUTMOD_TOGGLE_SET (TACCTL2 |= (OUTMOD2 | OUTMOD1)); \
66                                        (TACCTL2 &= ~OUTMOD0);
67
68  #define TIMER_A_0_OUTMOD_RESET        (TACCTL0 |= (OUTMOD2 | OUTMOD0)); \
69                                        (TACCTL0 &= ~OUTMOD1);
70  #define TIMER_A_1_OUTMOD_RESET        (TACCTL1 |= (OUTMOD2 | OUTMOD0)); \
71                                        (TACCTL1 &= ~OUTMOD1);
72  #define TIMER_A_2_OUTMOD_RESET        (TACCTL2 |= (OUTMOD2 | OUTMOD0)); \
73                                        (TACCTL2 &= ~OUTMOD1);
74
75  #define TIMER_A_0_CM_IRQ_ENABLE       (TACCTL0 |= CCIE);
76  #define TIMER_A_1_CM_IRQ_ENABLE       (TACCTL1 |= CCIE);
77  #define TIMER_A_2_CM_IRQ_ENABLE       (TACCTL2 |= CCIE);
78
79  #define TIMER_A_0_CM_IRQ_DISABLE      (TACCTL0 &= ~CCIE);
80  #define TIMER_A_1_CM_IRQ_DISABLE      (TACCTL1 &= ~CCIE);
81  #define TIMER_A_2_CM_IRQ_DISABLE      (TACCTL2 &= ~CCIE);
82
83
84  /**************************************************************
85  **  Defines Timer B
86  **************************************************************/
87  #define TIMER_B_START_UP_MODE         (TBCTL |= MC0)
88  #define TIMER_B_START_CM_MODE         (TBCTL |= MC1)
89  #define TIMER_B_START_UD_MODE         (TBCTL |=(MC1 | MC0))
90
91  #define TIMER_B_STOP                  (TBCTL &= ~(MC0 | MC1))
92
93  #define TIMER_B_RESET                 (TBCTL = TBCLR)
94  #define TIMER_B_FLAG_RESET            (TBCCTL1 &= ~(CCIFG|COV))
95
96  #define TIMER_B_SOURCE_TACLK          (TBCTL = TBSSEL_0)
97  #define TIMER_B_SOURCE_ACLK           (TBCTL = TBSSEL_1)
98  #define TIMER_B_SOURCE_SMCLK          (TBCTL = TBSSEL_2)
99
100 #define TIMER_B_SOURCE_DIV_8          (TBCTL |= (ID1 | ID0))
101 #define TIMER_B_SOURCE_DIV_4          (TBCTL |= ID1); \
102                                       (TBCTL &= ~ID0);
103 #define TIMER_B_SOURCE_DIV_2          (TBCTL &= ~ID1); \
104                                       (TBCTL |= ID0);
105 #define TIMER_B_SOURCE_DIV_1          (TBCTL &= ~ID1); \
106                                       (TBCTL &= ~ID0);
107
108 #define TIMER_B_CM_RISING_EDGE        (TBCCTL0 |= CM1)
109
110 #define TIMER_B_0_OUTMOD_RESET_SET    (TBCCTL0 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
111 #define TIMER_B_1_OUTMOD_RESET_SET    (TBCCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
112 #define TIMER_B_2_OUTMOD_RESET_SET    (TBCCTL2 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
113
114 #define TIMER_B_0_OUTMOD_TOGGLE_SET (TBCCTL0 |= (OUTMOD2 | OUTMOD1)); \
115                                       (TBCCTL0 &= ~OUTMOD0);
116 #define TIMER_B_1_OUTMOD_TOGGLE_SET (TBCCTL1 |= (OUTMOD2 | OUTMOD1)); \
117                                       (TBCCTL1 &= ~OUTMOD0);
118 #define TIMER_B_2_OUTMOD_TOGGLE_SET (TBCCTL2 |= (OUTMOD2 | OUTMOD1)); \
119                                       (TBCCTL2 &= ~OUTMOD0);
120
121 #define TIMER_B_0_OUTMOD_RESET        (TBCCTL0 |= (OUTMOD2 | OUTMOD0)); \
122                                       (TBCCTL0 &= ~OUTMOD1);
123 #define TIMER_B_1_OUTMOD_RESET        (TBCCTL1 |= (OUTMOD2 | OUTMOD0)); \
124                                       (TBCCTL1 &= ~OUTMOD1);
125 #define TIMER_B_2_OUTMOD_RESET        (TBCCTL2 |= (OUTMOD2 | OUTMOD0)); \
126                                       (TBCCTL2 &= ~OUTMOD1);
127
```

```
128 #define TIMER_B_0_CM_IRQ_ENABLE      (TBCCTL0  |=  CCIE);
129 #define TIMER_B_1_CM_IRQ_ENABLE      (TBCCTL1  |=  CCIE);
130 #define TIMER_B_2_CM_IRQ_ENABLE      (TBCCTL2  |=  CCIE);
131
132 #define TIMER_B_0_CM_IRQ_DISABLE     (TBCCTL0  &=  ~CCIE);
133 #define TIMER_B_1_CM_IRQ_DISABLE     (TBCCTL1  &=  ~CCIE);
134 #define TIMER_B_2_CM_IRQ_DISABLE     (TBCCTL2  &=  ~CCIE);
135
136
137 /**** Prototyp declaration ****************************************************/
138 void timer_a_init(void);
139 void timer_a_init_balanc(void);
140 void timer_a_test(void);
141 void timer_b_init_cali(void);
142 void timer_b_init_clock_cali(void);
143 void timer_a_delay_ms(uint16_t);
144 void timer_b_init_burst(uint8_t);
145 void timer_b_init_burst_end(uint8_t);
146 void delay_ms(uint16_t);
147 void timer_a_delay(uint16_t);
148
149 #endif
```

```
 1 /*-----------------------------------------------------------------------------
 2      Description:    MSP430 (mspgcc) data types.
 3      Date:           12/02/2012
 4      Last Update:    12/02/2012
 5      Author:         Phillip Durdaut
 6 -----------------------------------------------------------------------------*/
 7
 8 #ifndef TYPES_H_
 9 #define TYPES_H_
10
11 /*-----------------------------------------------------------------------------
12   Types
13 -----------------------------------------------------------------------------*/
14
15 typedef unsigned char        uint8_t;
16 typedef signed char          sint8_t;
17 typedef unsigned int         uint16_t;
18 typedef signed int           sint16_t;
19 typedef unsigned long        uint32_t;
20 typedef signed long          sint32_t;
21 typedef unsigned long long   uint64_t;
22 typedef signed long long     sint64_t;
23
24 typedef unsigned char        u8_t;
25 typedef signed char          s8_t;
26 typedef unsigned int         u16_t;
27 typedef signed int           s16_t;
28 typedef unsigned long        u32_t;
29 typedef signed long          s32_t;
30 typedef unsigned long long   u64_t;
31 typedef signed long long     s64_t;
32 #endif /* TYPES_H_ */
```

M. Quellcode Matlab Auswertung

```
1   %% Erste EIS Messung mit finalem Sensor
2   % Delay Zeit auf BS waren  SysCtlDelay(3302);
3   % Test mit Sensor 0x02
4   % Stromversorgung über DCDC
5   % Anregung: FuelCon
6   % Zelle: 7
7   %
8   % Rheostat Gleichspannungsabzug : 0x3ff
9   % Rheostat Verstärkun           : 0x02f
10
11  clear all
12  close all;
13
14  % Messdaten FuelCon vom 29.05.15
15  daten = [
16  2102.5085   -6.38827     0.139448
17  1473.514    -6.511894    0.343586
18  1032.6919   -6.664385    0.508807
19  723.7479    -6.839265    0.640194
20  507.2287    -7.035753    0.745141
21  355.4842    -7.247742    0.823737
22  249.1362    -7.469934    0.884654
23  174.6037    -7.694266    0.921424
24  122.3686    -7.921372    0.948425
25  85.7603     -8.149984    0.969278
26  60.1039     -8.380949    0.986115
27  42.123      -8.618113    0.99769
28  29.5213     -8.859452    0.997143
29  20.6896     -9.102606    0.982995
30  14.5        -9.337853    0.95234
31  10.1621     -9.535304    0.914475
32  7.122       -9.752221    0.868444
33  4.9914      -9.930971    0.830639
34  3.4981      -10.088704   0.810213
35  2.4516      -10.239494   0.810948
36  1.7182      -10.39311    0.833895
37  1.2042      -10.558719   0.884799
38  0.8439      -10.724551   0.967921
39  0.5914      -11.024745   1.0419
40  0.4145      -11.240341   1.169454
41  0.2905      -11.485869   1.345238
42  0.2036      -11.778277   1.555528
43  0.1427      -12.121537   1.82476
44  0.1         -12.511893   2.163643
45  ];
46
47  %%
48  mess = 0;
49  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
50  %% 50Hz Burst, 0.1Hz Signal, 2000Samples
51  mess = mess + 1;
52  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*0.1));
53
54  % Messwerte Rohdaten
55  v = [Daten auf der beiliegenden CD];
56  c = [Daten auf der beiliegenden CD];
57
58  % Berechnung der Stromwerte
59  c = (((c.*(0.00080586))-1.65)*(10));
60
61  %Berechnung der Spannungswerte
62  gain = (1+((47*(100000/1024))/ 1000));
63  v = ((v.*(adc_value))/gain);
64
65  Fs = 50;     % Burstfreqenz
66  f = 0.1;     % Anregefreqenz
67  m = 1;       % Anzahl der Perioden pro Berechnung
68  Np = Fs/f;   % Np Anzahl Abtastwerte pro Periode
69  N = m * Np;  % Anzahl der Abtastpunke für die Berechnung
70
```

```
71  c = c - min(c(1:N));
72  v = v - min(v(1:N));
73
74  freq_indices = round(f/Fs*N) + 1;
75  I = goertzel(c(1:N),freq_indices);
76  U =  goertzel(v(1:N),freq_indices);
77  Z_Z_2905(mess) = U/I;
78  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79  %% 100Hz Burst, 0.5Hz Signal, 2000Samples
80  mess = mess + 1;
81  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*0.5));
82
83  % Messwerte Rohdaten
84  v = [Daten auf der beiliegenden CD];
85  c = [Daten auf der beiliegenden CD];
86
87  % Berechnung der Stromwerte
88  c = (((c.*(0.00080586))-1.65)*(10));
89
90  %Berechnung der Spannungswerte
91  gain = (1+((47*(100000/1024))/ 1000));
92  v = ((v.*(adc_value))/gain);
93
94  Fs = 100;   % Burstfreqenz
95  f = 0.5;    % Anregefreqenz
96  m = 5;      % Anzahl der Perioden pro Berechnung
97  Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
98  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
99
100 c = c - min(c(1:N));
101 v = v - min(v(1:N));
102
103 freq_indices = round(f/Fs*N) + 1;
104 I = goertzel(c(1:N),freq_indices);
105 U =  goertzel(v(1:N),freq_indices);
106 Z_Z_2905(mess) = U/I;
107 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
108 %% 200Hz Burst, 1Hz Signal
109 mess = mess + 1;
110 adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*1));
111
112 % Messwerte Rohdaten
113 v = [Daten auf der beiliegenden CD];
114 c = [Daten auf der beiliegenden CD];
115
116 % Berechnung der Stromwerte
117 c = (((c.*(0.00080586))-1.65)*(10));
118
119 %Berechnung der Spannungswerte
120 gain = (1+((47*(100000/1024))/ 1000));
121 v = ((v.*(adc_value))/gain);
122
123 Fs = 200;   % Burstfreqenz
124 f = 1.03;   % Anregefreqenz
125 m = 9;      % Anzahl der Perioden pro Berechnung
126 Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
127 N = m * Np; % Anzahl der Abtastpunke für die Berechnung
128
129 c = c - min(c(1:N));
130 v = v - min(v(1:N));
131
132 freq_indices = round(f/Fs*N) + 1;
133 I = goertzel(c(1:N),freq_indices);
134 U =  goertzel(v(1:N),freq_indices);
135 Z_Z_2905(mess) = U/I;
136 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
137 %% 200Hz Burst, 5Hz Signal
138 mess = mess + 1;
139 adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*5));
140
141 % Messwerte Rohdaten
142 v = [Daten auf der beiliegenden CD];
143 c = [Daten auf der beiliegenden CD];
144
145 % Berechnung der Stromwerte
146 c = (((c.*(0.00080586))-1.65)*(10));
147
148 %Berechnung der Spannungswerte
149 gain = (1+((47*(100000/1024))/ 1000));
150 v = ((v.*(adc_value))/gain);
151
152 Fs = 200;   % Burstfreqenz
153 f = 5;      % Anregefreqenz
154 m = 20;     % Anzahl der Perioden pro Berechnung
155 Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
```

```
156  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
157
158  c = c - min(c(1:N));
159  v = v - min(v(1:N));
160
161  freq_indices = round(f/Fs*N) + 1;
162  I = goertzel(c(1:N),freq_indices);
163  U =  goertzel(v(1:N),freq_indices);
164  Z_Z_2905(mess) = U/I;
165  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
166  %% 1000Hz Burst, 10Hz Signal
167  mess = mess + 1;
168  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*10));
169
170  % Messwerte Rohdaten
171  v = [Daten auf der beiliegenden CD];
172  c = [Daten auf der beiliegenden CD];
173
174  % Berechnung der Stromwerte
175  c = (((c.*(0.00080586))-1.65)*(10));
176
177  %Berechnung der Spannungswerte
178  gain = (1+((50*(100000/1024))/ 1000));
179  v = ((v.*(adc_value))/gain);
180
181  Fs = 1000;  % Burstfreqenz
182  f = 10;     % Anregefreqenz
183  m = 10;     % Anzahl der Perioden pro Berechnung
184  Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
185  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
186
187  c = c - min(c(1:N));
188  v = v - min(v(1:N));
189
190  freq_indices = round(f/Fs*N) + 1;
191  I = goertzel(c(1:N),freq_indices);
192  U =  goertzel(v(1:N),freq_indices);
193  Z_Z_2905(mess) = U/I;
194  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
195  %% 2000Hz Burst, 50Hz Signal
196  mess = mess + 1;
197  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*50));
198
199  % Messwerte Rohdaten
200  v = [Daten auf der beiliegenden CD];
201  c = [Daten auf der beiliegenden CD];
202
203  % Berechnung der Stromwerte
204  c = (((c.*(0.00080586))-1.65)*(10));
205
206  %Berechnung der Spannungswerte
207  gain = (1+((47*(100000/1024))/ 1000));
208  v = ((v.*(adc_value))/gain);
209
210  Fs = 2000;  % Burstfreqenz
211  f = 50;     % Anregefreqenz
212  m = 25;     % Anzahl der Perioden pro Berechnung
213  Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
214  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
215
216  c = c - min(c(1:N));
217  v = v - min(v(1:N));
218
219  freq_indices = round(f/Fs*N) + 1;
220  I = goertzel(c(1:N),freq_indices);
221  U =  goertzel(v(1:N),freq_indices);
222  Z_Z_2905(mess) = U/I;
223  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
224  %% 2000Hz Burst, 100Hz Signal
225  mess = mess + 1;
226  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*100));
227
228  % Messwerte Rohdaten
229  v = [Daten auf der beiliegenden CD];
230  c = [Daten auf der beiliegenden CD];
231
232  % Berechnung der Stromwerte
233  c = (((c.*(0.00080586))-1.65)*(10));
234
235  %Berechnung der Spannungswerte
236  gain = (1+((47*(100000/1024))/ 1000));
237  v = ((v.*(adc_value))/gain);
238
239  Fs = 2000;  % Burstfreqenz
240  f = 100;    % Anregefreqenz
```

```
241  m = 98;      % Anzahl der Perioden pro Berechnung
242  Np = Fs/f;   % Np Anzahl Abtastwerte pro Periode
243  N = m * Np;  % Anzahl der Abtastpunke für die Berechnung
244
245  c = c - min(c(1:N));
246  v = v - min(v(1:N));
247
248  freq_indices = round(f/Fs*N) + 1;
249  I = goertzel(c(1:N),freq_indices);
250  U =  goertzel(v(1:N),freq_indices);
251  Z_Z_2905(mess) = U/I;
252  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
253  %% 8000Hz Burst, 250Hz Signal
254  mess = mess + 1;
255  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*250));
256
257  % Messwerte Rohdaten
258  v = [Daten auf der beiliegenden CD];
259  c = [Daten auf der beiliegenden CD];
260
261  % Berechnung der Stromwerte
262  c = (((c.*(0.00080586))-1.65)*(10));
263
264  %Berechnung der Spannungswerte
265  gain = (1+((47*(100000/1024))/ 1000));
266  v = ((v.*(adc_value))/gain);
267
268  Fs = 8000;  % Burstfreqenz
269  f = 250;    % Anregefreqenz
270  m = 16;     % Anzahl der Perioden pro Berechnung
271  Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
272  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
273
274  c = c - min(c(1:N));
275  v = v - min(v(1:N));
276
277  freq_indices = round(f/Fs*N) + 1;
278  I = goertzel(c(1:N),freq_indices);
279  U =  goertzel(v(1:N),freq_indices);
280  Z_Z_2905(mess) = U/I;
281  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
282  %% 8000Hz Burst, 500Hz Signal
283  mess = mess + 1;
284  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*500));
285
286  % Messwerte Rohdaten
287  v = [Daten auf der beiliegenden CD];
288  c = [Daten auf der beiliegenden CD];
289
290  % Berechnung der Stromwerte
291  c = (((c.*(0.00080586))-1.65)*(10));
292
293  %Berechnung der Spannungswerte
294  gain = (1+((47*(100000/1024))/ 1000));
295  v = ((v.*(adc_value))/gain);
296
297  Fs = 8000;  % Burstfreqenz
298  f = 500;    % Anregefreqenz
299  m = 90;     % Anzahl der Perioden pro Berechnung
300  Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
301  N = m * Np; % Anzahl der Abtastpunke für die Berechnung
302
303  c = c - min(c(1:N));
304  v = v - min(v(1:N));
305
306  freq_indices = round(f/Fs*N) + 1;
307  I = goertzel(c(1:N),freq_indices);
308  U =  goertzel(v(1:N),freq_indices);
309  Z_Z_2905(mess) = U/I;
310  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
311  %% 8000Hz Burst, 750Hz Signal
312  mess = mess + 1;
313  adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*750));
314
315  % Messwerte Rohdaten
316  v = [Daten auf der beiliegenden CD];
317  c = [Daten auf der beiliegenden CD];
318
319  % Berechnung der Stromwerte
320  c = (((c.*(0.00080586))-1.65)*(10));
321
322  %Berechnung der Spannungswerte
323  gain = (1+((47*(100000/1024))/ 1000));
324  v = ((v.*(adc_value))/gain);
325
```

```
326 Fs = 8000;  % Burstfreqenz
327 f = 750;    % Anregefreqenz
328 m = 50;     % Anzahl der Perioden pro Berechnung
329 Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
330 N = m * Np; % Anzahl der Abtastpunke für die Berechnung
331
332 c = c - min(c(1:N));
333 v = v - min(v(1:N));
334
335 freq_indices = round(f/Fs*N) + 1;
336 I = goertzel(c(1:N),freq_indices);
337 U =  goertzel(v(1:N),freq_indices);
338 Z_Z_2905(mess) = U/I;
339 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
340 %% 8000Hz Burst, 1000Hz Signal
341 mess = mess + 1;
342 adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*1000));
343
344 % Messwerte Rohdaten
345 v = [Daten auf der beiliegenden CD];
346 c = [Daten auf der beiliegenden CD];
347
348 % Berechnung der Stromwerte
349 c = (((c.*(0.00080586))-1.65)*(10));
350
351 %Berechnung der Spannungswerte
352 gain = (1+((47*(100000/1024))/ 1000));
353 v = ((v.*(adc_value))/gain);
354
355 Fs = 8000;  % Burstfreqenz
356 f = 1000;   % Anregefreqenz
357 m = 50;     % Anzahl der Perioden pro Berechnung
358 Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
359 N = m * Np; % Anzahl der Abtastpunke für die Berechnung
360
361 c = c - min(c(1:N));
362 v = v - min(v(1:N));
363
364 freq_indices = round(f/Fs*N) + 1;
365 I = goertzel(c(1:N),freq_indices);
366 U =  goertzel(v(1:N),freq_indices);
367 Z_Z_2905(mess) = U/I;
368 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
369 %% 8000Hz Burst, 2000Hz Signal
370 mess = mess + 1;
371 adc_value = (2.42/4095) -(((2.42/4095)/100)*((8.0/2000)*2000));
372
373 % Messwerte Rohdaten
374 v = [Daten auf der beiliegenden CD];
375 c = [Daten auf der beiliegenden CD];
376
377 % Berechnung der Stromwerte
378 c = (((c.*(0.00080586))-1.65)*(10));
379
380 %Berechnung der Spannungswerte
381 gain = (1+((47*(100000/1024))/ 1000));
382 v = ((v.*(adc_value))/gain);
383
384 Fs = 8000;  % Burstfreqenz
385 f = 2000;   % Anregefreqenz
386 m = 5;      % Anzahl der Perioden pro Berechnung
387 Np = Fs/f;  % Np Anzahl Abtastwerte pro Periode
388 N = m * Np; % Anzahl der Abtastpunke für die Berechnung
389
390 c = c - min(c(1:N));
391 v = v - min(v(1:N));
392
393 freq_indices = round(f/Fs*N) + 1;
394 I = goertzel(c(1:N),freq_indices);
395 U =  goertzel(v(1:N),freq_indices);
396 Z_Z_2905(mess) = U/I;
397 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
398 %% Plot der Daten
399 h = figure(1);
400 plot((real(Z_Z_2905)*1000),(-imag(Z_Z_2905)*1000),'x',-daten(:,2),daten(:,3));
401 xlabel('Realteil in m\Omega');
402 ylabel('Imaginärteil in -m\Omega');
403 grid on;
404 hold on;
405 xlim([5 13]);
```

N. Verzeichnisstruktur des Datenträgers

Die gedruckten Versionen dieser Masterarbeit ist ein Datenträger beigefügt, der bei Herrn Prof. Dr.-Ing. Riemschneider an der HAW Hamburg eingesehen werden kann.
Das Hauptverzeichnis des Datenträgers ist wie folgt aufgeteilt:

• **Auswertung der EIS Daten**: Dieser Ordner beinhaltet das Auswerteskript der EIS-Messung mit den dazugehörigen Messwerten.

• **Auswertung Testimpedanz**: Dieser Ordner beinhaltet das Auswerteskript für die Vermessung der Testimpedanz mit den dazugehörigen Messwerten.

• **Goertzel Testprogramm**: Dieser Ordner beinhaltet den Quellcode des PC Goetzel-Testprogramm mit den für die Auswertung benutzten Messwerten.

• **Platinendaten**: Dieser Ordner beinhaltet sämtliche Platinen und Schaltungsunterlagen, die im Laufe dieser Arbeit angefertigt wurden

• **Quellcode Batteriesteuergerät**: Dieser Ordner beinhaltet den Quellcode des Batteriesteuergeräts.

• **Quellcode Zellsensor**: Dieser Ordner beinhaltet den Quellcode des Zellsensors.

• **Thesis Nico Sassano**: Dieser Ordner beinhaltet die vorliegende Arbeit in elektronischer Form.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit 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, 14. August 2015
Ort, Datum

Unterschrift