Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Nico Sassano

Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Nico Sassano

Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung

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

Betreuender Prüfer : Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr.-Ing. Ralf Wendel

Abgegeben am 16. August 2013

Nico Sassano

Thema der Bachelorthesis

Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung

Stichworte

Batteriezellensensor, Hardwareentwicklung, Softwareentwicklung, drahtlose Kommunikation, Mikrocontroller, MSP430F235, CC1101, funksynchronisierte Messung

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Untersuchung und der Entwicklung eines neuen Messverfahrens, mit dem es möglich sein soll, funksynchronisierte Messungen an einer Batteriezelle durchzuführen. Diese Messungen sollen dabei durch einen Takt, ausgehend von einem Batteriesteuersystem, ausgelöst werden und anschließend über eine Funkschnittstelle an diese übertragen werden.

Nico Sassano

Title of the paper

Development and construction of the hard- and software for a wireless communicating battery cell sensor with radio-synchronised measurement

Keywords

Battery Cell Sensor, hard- and software development, Wireless communi-cating, Microcontroller, MSP430F235, CC1101, radio-synchronised

Abstract

This thesis discusses the development and construction of a new measure-ment method. Its purpose is to enable radio-synchronised measurements of battery cell sensors. A battery management system triggers a tact that starts the measurement and subsequently transfers the measurements through a radio interface back to the management system.

Inhaltsverzeichnis

1. Einführung

1.1. Das Projekt BATSEN

Heutzutage nehmen Batterien und Akkumulatoren in vielen Bereichen unseres Lebens eine wichtige Rolle ein, sei es in mobilen Geräten oder in modernen Fahrzeugen. Die Anforderungen an Batterien bzw. Akkumulatoren steigen ständig an. Gewünscht ist, dass diese immer kleiner, leichter und leistungsfähiger werden. Besonders in modernen PKW nimmt die Anzahl der Verbraucher stetig zu. Zunächst hatte die Batterie im KFZ lediglich die Aufgabe das Fahrzeug zu starten. Daher kommt auch der noch heute gängige Name “Starterbatterie”. Die zunehmende Anzahl an modernen Informations- und Unterhaltungssystemen belastet die Batterie und verringert die Entladedauer. Unter ungünstigen Bedingungen, wie z.B. niedrige Temperaturen, kann dies zum Ausfall des Fahrzeugs führen. Die aktuelle Pannenstatistik des ADAC zeigt, dass die häufigste Pannenursache bei PKWs auf den Ausfall der Bordbatterie zurückzuführen ist, 2012 lag dieser bei 31,7% [1].
Diese Statistik zeigt deutlich das ein dringender Handlungsbedarf an der Entwicklung an den sogenannten Batteriemanagementsystemen (BMS) besteht. Einige Hersteller haben bereits solche Systeme auf dem Markt gebracht. Diese sind meist in der Lage, die aktuelle Spannung und die Temperatur der Batterie zu messen. Diese sind allerdings meist Drahtgebunden und können meist nur die Batterie als Gesamtes überwachen. Wichtig ist aber auch das Verhalten einzelner Zellen zu kennen, da eine Batterie zumeist aus mehreren Zellen besteht, die in Reihe geschaltet sind. Ist eine Zelle dieser Batterie schwach oder defekt, lässt sich dies nicht immer durch eine Gesamtüberwachung feststellen.
Das Forschungsprojekt “Drahtlose Zellensensoren für Fahrzeugbatterien”, kurz BATSEN der Hochschule für Angewandte Wissenschaften Hamburg, beschäftigt sich mit der Entwicklung eines solchen drahtlosen Batteriemanagementsystems zur Überwachung der einzelnen Zellen. Gefördert vom Bundesministerium für Bildung und Forschung sowie Partner aus der Industrie, entwickelt das Forschungsteam drahtlose Batteriezellensensoren. Diese Zellensensoren haben die Aufgabe, die Spannung sowie die Temperatur der einzelnen Batteriezelle zu messen und die Daten an ein gemeinsames Batteriesteuergerät zu übermitteln. Das Batteriesteuergerät übernimmt dabei auch die Messung des Stromes.
Abbildung 1.1 zeigt dabei den prinzipiellen Aufbau einer Batterie mit integrierten Zellensensoren.

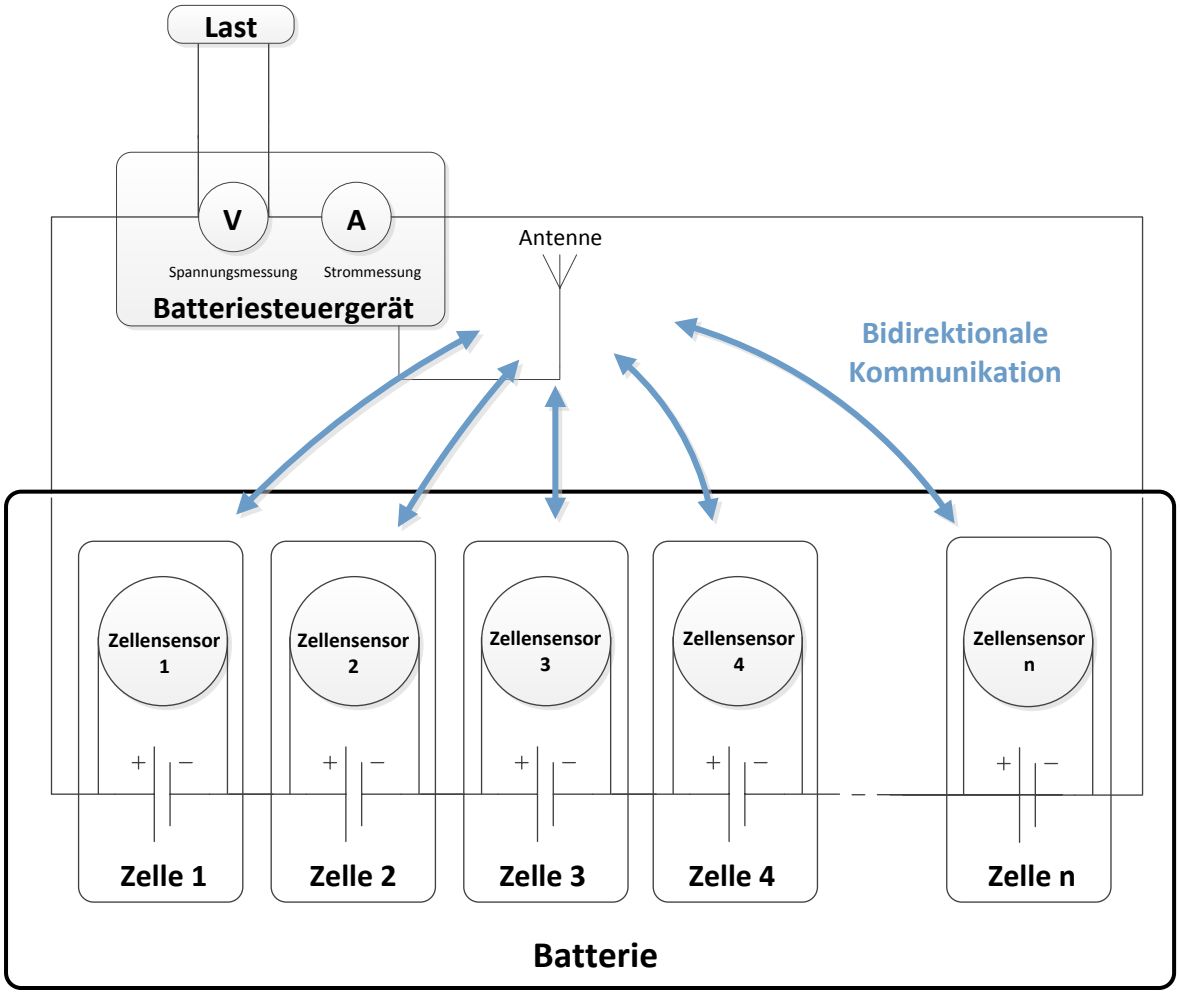


Abbildung 1.1.: Prinzipschaltbild mit überwachten Zellensensoren nach [17]

Aus den durch die Zellensensoren ermittelten Daten und dem durch das Batteriesteuergerät gemessenen Strom lassen sich Aussagen über den Zustand der Batterie machen. Zum einen lässt sich der Ladezustand, State of Charge (SOC), zum anderen der Gesundheitszustand, State of Health (SOH) der Batterie ermitteln. Ziel ist es, dadurch eine Warnung vor drohendem Batterieausfall zu generieren, die optimale wirtschaftliche Ausnutzung der Batterie und die Betriebssicherheit sicherzustellen [17]. Ziel des Projekts BATSEN ist es, einen kostengünstigen und in die Batterie integrierbaren Zellensensor zu entwickeln.

1.2. Lösungsvarianten der Zellensensoren

Bisher wurden verschiedene Lösungsvarianten im Projekt BATSEN entwickelt. Diese Sensortypen wurden innerhalb des Forschungsprojekts in drei unterschiedliche Klassen eingeteilt. Im Wesentlichen unterscheiden sich diese Sensorklassen in der Möglichkeit, dass diese mit Empfängern ausgestattet sind. Durch einen integrierten Empfänger ist es dem Zellensensor möglich, Befehle von dem Batteriesteuergerät zu erhalten. Dieser Empfang von Daten bzw. von Befehlen wird im Weiteren als Downlink bezeichnet.
Das Senden von Daten vom Zellensensor an das Batteriesteuergerät wird als Uplink bezeichnet.
Die nachfolgende Tabelle soll einen kurzen Überblick über die drei Sensorklassen geben.

Tabelle 1.1.: Übersicht der Sensorklassen nach [15]

Sensorklasse	**Klasse 1**	**Klasse 2**	**Klasse 3**
Mögliche Übertragungrichtungen	nur Uplink kein Downlink vorhanden	Uplink mit reduzierterm Downlink (Broadcast Wakeup)	Uplink und Downlink
Empfänger im Sensor	kein Empfänger vorhanden	passiver Empfänger	aktiver Empfänger
Messbetrieb und Kommunikation	Autonom	Teilautonom	Zentral gesteuert
Erforderlicher Hardwareaufwand	niedrig	aufwendig	sehr aufwendig

Im Projekt BATSEN wurden bereits in mehreren Bachelor-, Diplom-, sowie Masterarbeiten alle drei Sensortypen entwickelt und erprobt. Zuletzt wurde ein Zellensensor der Klasse 3 entwickelt und erfolgreich getestet [4].

Zellensensor Klasse 1

Die Zellensensoren der Klasse 1 wurden zuletzt in der Diplomarbeit von Stephan Plaschke [21] und in der Bachelorthesis von Sergej Ilgin [6] entwickelt und erprobt.
Die Zellensensoren der Klasse 1 sind in der Lage, die Zellspannung und die Temperatur der Zellen zu messen. Die Übertragung zum Batteriesteuergerät erfolgt dabei über einen auf dem Zellensensor vorhandenen UHF-Transmitter, welcher im 433MHz Band arbeitet.
Ein Problem dieser Sensoren ist, dass diese über keinen Downlinkkanal verfügen und somit nicht von dem Batteriesteuergerät Befehle zur Spannungs- oder zur Temperaturmessung empfangen können. Die Zellensensoren entscheiden selbst, wann Spannung- und Temperaturwerte aufgenommen werden sollen und senden diese anschließend zurück zum Batteriesteuergerät. Durch die Asynchronität der Sendungen der verschiedenen Zellensensoren kann es zu Überlagerungen der einzelnen Sendungen kommen, welche die Daten für das Batteriesteuergerät unbrauchbar macht. Dieses Problem wurde in der Diplomarbeit von Simon Püttjer [19] durch die Anpassung des Sendezeitpunkts eine pseudozufällige Sendeverzögerung gelöst. Dennoch bleibt der wesentliche Nachteil dieser Klasse, dass es nicht möglich ist, bestimmte Kommandos an die Zellensensoren zu senden. Zur Lösung dieses Problems wurde ein Zellensensor der Klasse 2 entwickelt und erprobt.

Zellensensor Klasse 2

Der in der Masterarbeit von Niels Jegenhorst [14] angefertigte Zellensensor ist ein Sensor der Klasse 2. Dieser enthält eine RFID-Schaltung, welche auf der Frequenz von 13,56 MHz empfängt. Diese Schaltung ist für das Empfangen eines Wake-Up-Befehls und dem Empfang von Messzeitpunkten vorhanden. Im Weiteren ist dieser Sensor mit einem UHF-Sender ausgestattet, der auf 433 MHz sendet. Diese Frequenz wird für den Uplink von Messwerten zum Batteriesteuergerät genutzt.
Durch den neuen Broadcast Downlink-Kanal von dem Batteriesteuergerät zum Zellensensor ist es nun erstmalig möglich, den Zellensensor aktiv zu steuern. Dadurch wurde es möglich, die Zellensensoren in einen Schlafzustand zu setzen und diese wieder aufzuwecken, wenn diese wieder messen sollen. Diese neue Wake-Up-Funktion des Empfängers ermöglicht einen nahezu passiven Betrieb des deaktivierten Zellensensors, da dieser im Schlafzustand kaum Strom benötigt.
Dennoch hat diese Sensorklasse den Nachteil eines hohen Platzbedarfs. Da dieser für zwei verschiedene Frequenzen ausgelegt ist, werden auch zwei unterschiedliche Antennen benötigt. Ein weiterer Nachteil ist die hohe Sendeleistung von 750 mW des Batteriesteuergeräts. Um den Zellensensor klein zu halten, wurde nun versucht, das Senden und Empfangen von Sendungen auf ein Frequenzband zu reduzieren.

Zellensensor Klasse 3

Ein Sensor der Klasse 3 wurde während der Bachelorthesis von Phillip Durdaut [4] entwickelt. Dieser Zellensensor verfügt über einen Transceiver mit dem der Up- und Downlink zum Batteriesteuergerät im 434 MHz ISM Band möglich ist. Es wurde also ein vollwertiger, bidirektionaler Kommunikationspfad zwischen dem Batteriesteuergerät und den Zellensensoren realisiert.
Der Up- und Downlinkkanal läuft hier erstmalig auf derselben Frequenz. Dadurch ist auf dem Zellensensor nur noch eine Antenne für das Senden und Empfangen der Daten nötig und der Platzbedarf minimiert sich enorm. Durch den aktiven Transceiver ist es dem Zellensensor jetzt möglich, Daten auf derselben Frequenz zu empfangen und auch zu senden. Dies bringt den Vorteil, dass nun die einzelnen Zellensensoren durch das Batteriesteuergerät angesprochen und diese dann auch entsprechend antworten können. Ein weiterer Schwerpunkt der Arbeit war die Energieeffizienz des Systems zu optimieren.

1.3. Ziel dieser Arbeit und Motivation

Ziel dieser Arbeit ist es, den in der Bachelorarbeit "*Zellensensor für Fahrzeugbatterien mit Kommunikation und Wake-Up-Funktion im ISM-Band bei 434 MHz*"[4] entwickelten Zellensensor zu überarbeiten und dessen Funktionen zu erweitern. Insbesondere sollen Grundlagen für eine neue Messmethode untersucht werden. Dabei soll es sich um eine funksynchronisierte Messung handeln, die in der Lage ist genaue zeitlich übereinstimmende Messungen über mehrere Sensoren hinweg durchzuführen. Gesteuert soll dies durch ein zentrales Steuersystem, welches dabei den Messtakt vorgeben soll. Durch eine zukünftige parallel dazu stattfindende Strommessung am Batteriesteuergerät soll durch diese neue Messmethode die Grundlagen für eine zukünftige Impedanzspektroskopie der Batteriezellen gelegt werden. Dies ist ein Verfahren, welches heute schon ein bekanntes Messverfahren in der Batterietechnik ist. Neben dieser Grundlagenbildung sollen mit dieser Messung eventuelle Spannungseinbrüche bei Hochstromereignissen detektiert werden. Dies ist mit den bisherigen Sensoren der Klasse 1 und 2 nur bedingt, bzw. nicht möglich. Um diese dynamischen Ereignisse präzise detektieren zu können, ist aber eine synchrone und hochgenaue Taktung zwischen dem Zellensensor und des Batteriesteuergerätes nötig. Die zu messenden Hochstromereignisse sind dabei zwar kurz, dennoch fallen durch die hohe Abtastung des Spannungsverlaufs sehr viel Daten an die verarbeitet werden, müssen. Dazu ist eine effektive Datenübertragung an das Batteriesteuergerät notwendig.

In den Voruntersuchungen soll eine Möglichkeit zur Realisierung dieser Messmethode erarbeitet und durch praktische Untersuchungen beurteilt werden. Dabei soll neben der Funktion der Messung, auch die Präzision des Verfahrens untersucht werden. Durch die Realisierung dieser Messfunktion kann eine Grundlage für weitere Arbeiten im Forschungsprojekt BATSEN entstehen.

Neben der Entstehung der neuen Messfunktion sollen die bisherigen Funktionen des Zellensensors erweitert werden. Dabei soll der Temperatursensor und dessen Funktionen voll in die Software des Zellensensors eingefügt werden. Des Weiteren soll eine Steuerung einer passiven Ladungsbalancierung realisiert werden. Diese soll die Zellenspannung bis zu einem ihr, von dem Steuergerät angegebenen Spannungswert balancieren. Dabei sollen durch den Zellensensor selbst, die Parameter Temperatur, Zeit und Spannung gesteuert werden.

2. Analyse des bestehenden Systems und dessen Anforderungen

2.1. Analyse des bestehenden Systems

2.1.1. Der Zellensensor

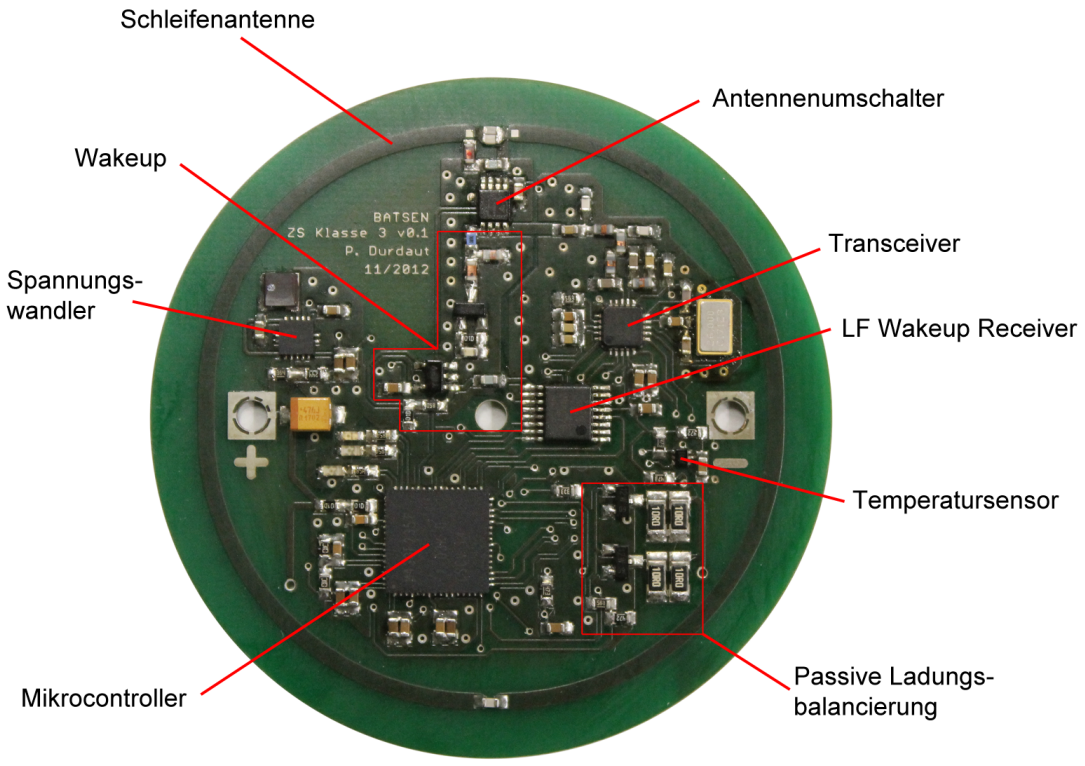


Abbildung 2.1.: Zellensensor V0.1 / entnommen aus [4]

Der Zellensensor V0.1 (Abbildung 2.1) soll nun Analysiert werden. Die bestehende Hardware soll in einem späteren Redesign des Zellensensors komplett erhalten bleiben, da diese

bereits erprobt und erfolgreich getestet wurden. Dadurch ist eine Recherche nach neuen Bauelementen nicht nötig. Im Folgenden wird die Hardware, die dann auch auf dem neuen Zellensensor V0.2 zum Einsatz kommt, kurz vorgestellt.

Hardwareanalyse des Zellensensors

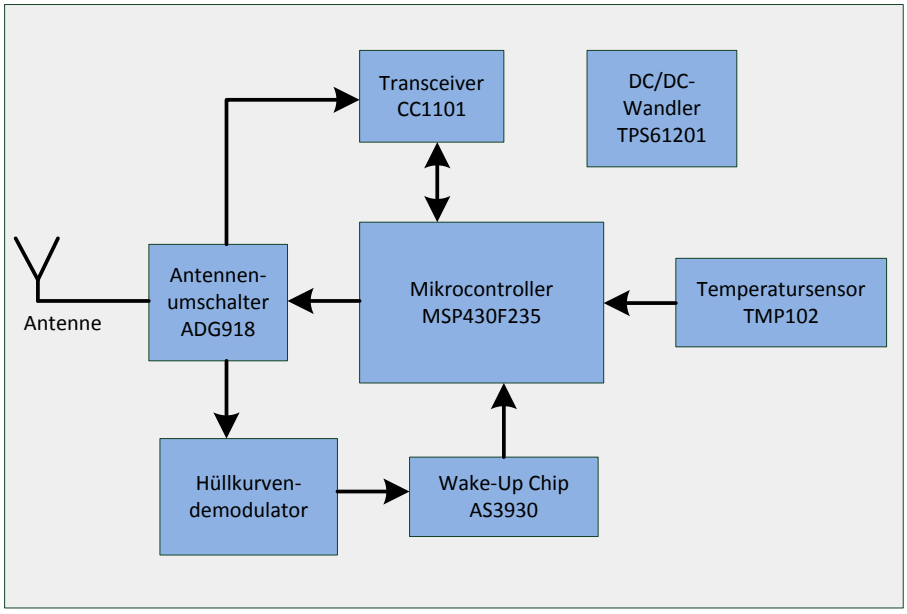


Abbildung 2.2.: Blockschaltbild des bestehenden Systems

Die Tabelle 2.1 gibt einen kompakten Überblick über die auf dem Zellensensor verwendete Hardware.

Tabelle 2.1.: Übersicht Hardwarekomponenten des Zellensensors V0.1

Bauteil	Bezeichnung	Hersteller
Mikrocontroller	MSP430F235	Texas Instruments
UHF Transceiver	CC1101	Texas Instruments
Spannungswandler	TPS61201	Texas Instruments
LF Wake-Up Receiver	AS3930	AMS
Antennenumschalter	ADG918BRM	Analog Devices
Temperatursensor	TMP102	Texas Instruments

Der Mikrocontroller

Zur Steuerung des Zellensensors kommt der 16-Bit-Mikrocontroller MSP430F235 zum Einsatz. Dieser verfügt über einen 16 kByte großen Flash-Speicher und 2 kByte RAM. Die MSP430 Mikrocontrollerserie der Firma Texas Instruments ist besonders geeignet für den Einsatz in batteriebetriebenen Geräten, da diese Serie über verschiedene Energiesparzustände verfügt. So lässt sich die Stromaufnahme im tiefsten Schlafmodus (LPM4) auf unter $1\mu A$ bringen. Ein weiterer Vorteil dieser Controller ist, dass sich der interne Digital Controlled Oscillator (DCO) während des Betriebs in mehreren Abstufungen zwischen 1 MHz und 16 MHz umschalten lässt. So lässt sich die Taktrate bei zeitkritischen Messungen erhöhen und im normalen Betrieb wieder senken.
Der Controller verfügt über mehrere Kommunikationsschnittstellen. So ist eine I2C Schnittstelle sowie eine SPI Schnittstelle vorhanden, die beide genutzt werden. Die SPI Schnittstelle dient zu Kommunikation und zum Datenaustausch mit dem UHF-Transceiver sowie zur Konfiguration des Wake-Up-Receiver. Die I2C Schnittstelle wird zur Kommunikation mit dem auf dem Zellensensor vorhandenen Temperatursensor genutzt. Weiter besitzt der Mikrocontroller zwei interne 16-Bit Timer, die in der aktuellen Konfiguration aber nicht genutzt. Ein 12-Bit A/D Umsetzer wird zur Messung der Zellenspannung verwendet. Dieser A/D Umsetzer verwendet eine interne Referenzspannung mit 2,5 V.
Dadurch, dass der interne DCO zur Taktgenerierung genutzt wird, ist kein zusätzlicher externer Quarz notwendig. Ein Vorteil der Platz- und Kostenersparnis einbringt. Ein Nachteil ist, dass der interne DCO eine gewisse Ungenauigkeit und eine starke Abhängigkeit zur Temperatur aufweist [12]. Ob dies zu Problemen bei der neuen Messmethode führt, muss im Weiteren untersucht werden.

MSP430F235 Übersicht

- 16-Bit-Mikrocontroller
- 16 kByte Flash-Speicher / 2 kByte RAM
- verschiedene Energiesparzustände
- umschaltbarer DCO (1MHz - 16MHz)
- I2C und SPI Schnittstelle
- 2 16-Bit Timer
- 12-Bit A/D Umsetzer
- kein externer Quarz notwendig

UHF-Transceiver

Als UHF-Transceiver wurde der CC1101 von Texas Instruments gewählt. Dieser wird zum Senden und Empfangen von Daten dem Batteriesteuergerät genutzt. Mit diesem Transceiver ist es möglich, eine Datenrate von bis zu 250 kbps bei 434MHz zu erreichen. Der Transceiver übernimmt die automatische Versendung der Datenpakete. Das heißt, es müssen lediglich die zu sendenden Daten an den CC1101 übergeben werden. Diese werden dann in den internen 64 Byte FIFO-Ringspeicher des Transceiver geschrieben. Die weitere Paketverwaltung der Daten übernimmt dann der CC1101.
An den CC1101 ist ein externer 26MHz Quarz angeschlossen. Dieser ist notwendig, um einen genauen Referenztakt zu bekommen.

CC1101 Übersicht

- Sende- Empfangsfrequenz 434 MHz
- Übertragungsraten bis 250 kbps
- 64 Byte FIFO-Ringspeicher
- externer 26 MHz Quarz

Der Spannungswandler

Der Zellensensor nimmt seine Versorgungsspannung direkt von der zu überwachenden Zelle. Da diese Zellenspannung aber nicht stabil 3.3 V beträgt, was für die Versorgung der aktiven Komponenten nötig ist, muss ein Spannungswandler eingesetzt werden. Dazu wurde der Spannungswandler TPS61201 von Texas Instruments eingesetzt. Dieser wurde bereits in anderen Arbeiten im Forschungsprojekt BATSEN eingesetzt.
Da der Zellensensor an verschiedenen Akkumulatortechnologien, mit unterschiedlichen Zellspannungen eingesetzt werden soll, ist dieser Spannungswandler gut geeignet. Er ermöglicht eine stabile Ausgangsspannung von 3.3 V bei Eingangsspannungen von 0.3 V bis 5.5 V. Abhängig von dieser Eingangsspannung liefert der TPS61201 einen Ausgangsstrom von 300 mA bis 500 mA, welcher für den Betrieb des Zellensensors ausreicht.
Über einen externen Spannungsteiler lässt sich einstellen, bis zu welcher Eingangsspannung U_{VLO} der TPS61201 noch arbeiten soll. Diese untere Schwellwertspannung U_{VLO} wurde auf dem Zellensensor V0.1 auf einen Wert von 0.6 V eingestellt. Fällt die Zellspannung unter diesen Wert, schaltet sich die Spannungsversorgung ab und somit auch der Zellensensor.

TPS61201 Übersicht

- Ausgangsspannung 3,3 V
- Eingangsspannungen von 0.3 V bis 5.5 V
- Ausgangsstrom von 300 mA bis 500 mA
- Einstellbare Abschaltspannung

Der LF Wake-Up Receiver

Der verwendete LF Wake-Up Receiver *AS3930 - LF Wake-Up Receiver* kommt von der Firma AMS. Dieser Receiver dient zur Erkennung des Wake-Up Signals. Der Chip arbeitet im Langwellenfrequenzbereich von 110 kHz bis 150 kHz.
Als Wake-Up Frequenz wurde in [4] die Frequenz von 125 kHz festgelegt, sodass der Baustein diese erkennt. Dieses Signal muss mindestens 550 μs am AS3930 anliegen, damit es als gültiges Wake-Up Signal erkannt wird. Bei der erfolgreichen Detektion dieser Frequenz stellt der AS3930 ein digitales Ausgangssignal zur Verfügung, welches vom Mikrocontroller als Interrupt registriert werden kann. Somit kann der Mikrocontroller aus seinem Schlafzustand aufgeweckt werden und dann die weitere Steuerung auf dem Zellensensor übernehmen. Konfigurieren lässt sich der AS3930 über die SPI Schnittstelle.

AS3930 Übersicht

- Arbeitet im Langwellenfrequenzbereich von 110 kHz bis 150 kHz
- Wake-Up Frequenz wurde auf 125 kHz festgelegt
- gibt ein digitales Ausgangssignal aus
- Konfiguration über SPI

Antennenumschalter

Da der Zellensensor über eine Antenne verfügt, es aber zwei verschiedene Empfängerpfade mit unterschiedlichen Impedanzen gibt, ist der Einsatz eines Antennenumschalters notwendig. Verwendet wird auf dem Zellensensor der Antennenumschalter *ADG918* von der Firma Analog Devices. Der Antennenumschalter besitzt einen Eingang, der mit einem der beiden Ausgängen verbunden werden kann. Die Auswahl, welcher der beiden Ausgänge angesteuert werden soll, kann über einen Steuereingang durch den Mikrocontroller ausgewählt werden.

Die Eingangsimpedanz des ADG918 beträgt 50 Ω. Dies bedeutet, dass ein Impedanzanpassungsnetzwerk zwischen Antenne und Antennenumschalter vorhanden sein muss um die Antennenimpedanz ebenfalls auf 50 Ω zu bringen.
Der Strombedarf laut Datenblatt beträgt lediglich 1 µA [3]. Dieser darf aber auch nicht zu groß sein, da der Antennenumschalter auch im Schlafzustand aktiv sein muss.

ADG918 Übersicht

- Eingangsimpedanz 50 Ω
- Ausgangspfad lässt sich umschalten
- geringer Strombedarf

Der Temperatursensor

Als Temperatursensor wird der TMP102 von Texas Instruments eingesetzt. Dieser ist bereits auf der Sensorplatine vorhanden, allerdings durch die Software noch nicht angebunden. Der Temperatursensor ist besonders gut für den Einsatz auf dem Zellensensor geeignet, da er eine geringe Stromaufnahme von 15 µA und eine kleine Bauform hat [10]. Da der TMP102 direkt auf der Sensorplatine V0.1 verbaut ist, kann dieser sehr gut für die Überwachung der Zelltemperatur eingesetzt werden. Dieser besitzt intern einen 12-Bit Delta-Sigma-A/D Umsetzer zur Erfassung der Temperatur. Diesen Temperaturwert gibt der TMP102 über eine integrierte I2C Schnittstelle an den Mikrocontroller weiter.
Des Weiteren besitzt der Temperatursensor einen digitalen Ausgang, der bei einer einzustellenden Schwelltemperatur ein Signal ausgibt. Dies kann durch den Mikrocontroller erfasst werden. Der Temperatursensor besitzt eine Temperaturauflösung von 0,0625 °C und eine Genauigkeit von 0,5 °C im Temperaturbereich von -25 °C bis +85 °C.

TMP102 Übersicht

- I2C Anbindung
- 12-Bit Delta-Sigma-A/D Umsetzer zur Erfassung der Temperatur
- 0,0625 °C Temperaturauflösung
- 0,5 °C Genauigkeit
- -25 °C bis +85 °C Temperaturbereich
- 15 µA Stromaufnahme

Softwareanalyse des Zellensensors

Die Abbildung 2.3 zeigt das Zustandsdiagramm eines Zellensensors nach [4]. Die Software ist so Programmiert, dass nachdem die Spannungsversorgung angeschlossen wird, der Zellensensor alle nötigen Komponenten initialisiert, die er für den Betrieb benötigt. Nach der Initialisierung geht der Zellensensor in den sogenannten “Sleep” Modus. Dies ist ein besonders effizienter Energiesparmodus des Mikrocontroller MSP430. In diesem Modus hat der Mikrocontroller einen Stromverbrauch von unter 1 μA [13]. Dieser Modus wird vom Hersteller des Mikrocontroller auch “Low-Power-Mode4” (LPM4) genannt. In diesem Betriebsmodus lässt sich der Controller nur durch einen externen Interrupt wecken. Dies geschieht auf dem Zellensensor, wenn dieser das Wake-Up-Signal erhält. Dabei gibt der Wake-Up Receiver AS3930 diesen Interrupt an den Mikrocontroller aus und weckt ihn somit auf. Nach diesem Aufwachen, stellt sich die Software auf den Empfang von Befehlen des Batteriesteuergeräts ein. Dieser warten nun, bis ein Befehl empfangen wurde. Das Empfangen eines Befehls übernimmt der Tranceiver CC1101. Dieser löst bei erfolgreichem Empfang eines Befehls am Mikrocontroller einen Interrupt aus, der die Daten anschließend über die SPI Verbindung von Tranceiver abruft. Wurde der Befehl im Mikrocontroller verarbeitet, geht dieser wieder in den Empfangsmodus und ist bereit zum Empfang und Verarbeiten neuer Befehle.

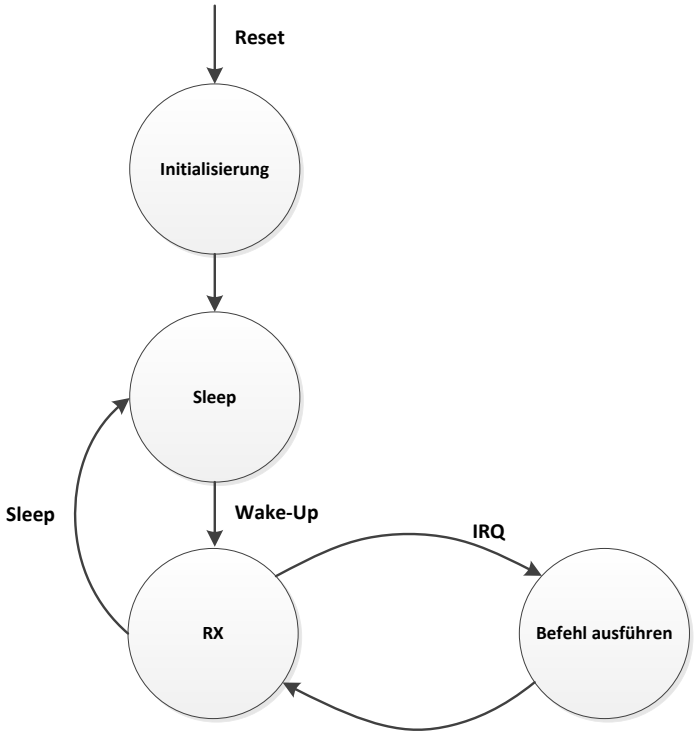


Abbildung 2.3.: Softwarekonzept nach [4]

2.1.2. Das Batteriesteuergerät

Nun soll das Batteriesteuergerät Analysiert werden. Dies wurde ebenfalls aus [4] übernommen.

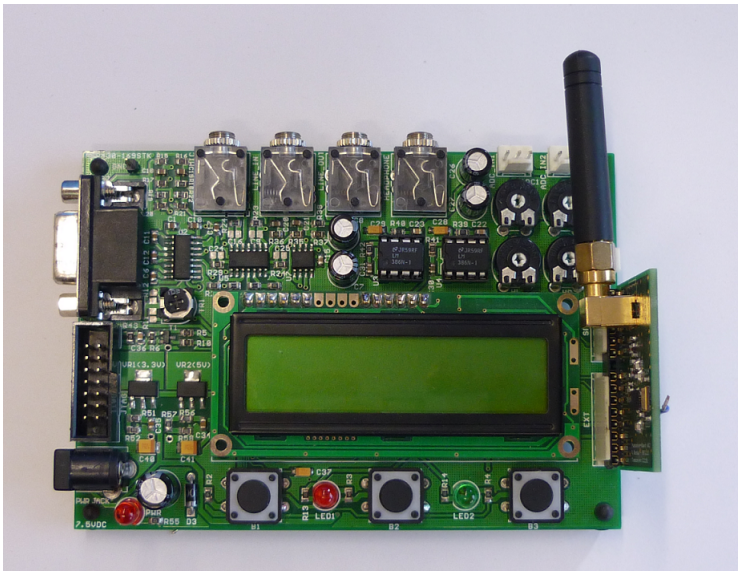


Abbildung 2.4.: Batteristeuergerät

Hardwareanalyse des Batteriesteuergeräts

Das Batteriesteuergerät besteht aus dem Entwicklungsboard MSP430-169STK des Herstellers OLIMEX Ltd. Als Mikrocontroller wird auf dem Board ein MSP430F169 der Firma Texas Instruments eingesetzt. Dieser Mikrocontroller besitzt einen 60kB + 256B Flash Memory und 2kB RAM [11]. Getaktet wird dieser mit einem 8MHz Quarz. Das Entwicklungsboard besitzt einige Anzeige-, Eingabe- sowie Ausgabemöglichkeiten. Darunter befindet sich ein 16 x 2 LC-Display, welches zur Anzeige von Meldungen dient. Die Steuerung des Batteriesteuergeräts erfolgt über drei auf dem Entwicklungsboard befindlichen Druckknöpfen. Damit lässt sich das das Wake-Up auslösen und die einzelne Spannungsmessung starten.
Die Ausgabe der einzelnen Spannungswerte erfolgt über die RS232-Schnittstelle, über die Daten an den PC gesendet werden können. Als weiterer Kommunikationsport wird die SPI Schnittstelle genutzt. Diese wird mit dem auf dem Tranceiverboard genutzten CC1101 verbunden.

Das Transceiverboard

An das Batteriesteuergerät ist ein Tranceiverboard mittels eines Steckverbinder verbunden. Über diese Steckverbindung erhält der Tranceiver CC1101 seine nötigen Betriebspannung von 3,3 V und Kontakt via SPI zum auf dem Entwicklungsboard sitzenden Mikrocontroller MSP430F169. Über diese Schnittstelle erhält der Tranceiver seine nötige Konfiguration sowie die zu sendenden Daten.

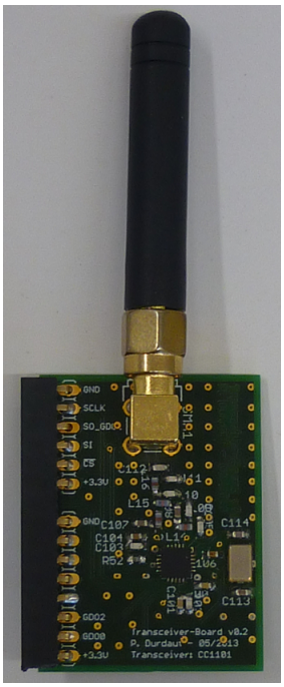


Abbildung 2.5.: Transceiverboard nach [4]

Softwareanalyse des Batteriesteuergeräts

Die Batteriesteuergerät Software ist in der Lage, einzelne Spannungswerte von einer einstellbaren Anzahl von Sensoren abzurufen.
Dazu schickt das Batteriesteuergerät verschiedene Kommandos an einen bestimmten oder auch an alle Zellensensoren. Es ist also möglich, einzelne Sensoren anzusprechen oder auch alle gemeinsam, im sogenannten Broadcast Befehl. Den Ablauf der Batteriesteuergerät Software zeigt Abbildung 2.6, welche aus [4] übernommen wurde.
Im Zustand “S_ INIT” wird die komplette Hardware initialisiert. Neben der Konfiguration des Timers für das Wake-Up-Signal, der Ein- und Ausgänge des Batteriestergeräts wird auch die der Transceiver CC1101 für das Senden der Daten konfiguriert. Sobald nun der Taster 1 auf dem Batteriesteuergerät gedrückt wird, wird das Wake-Up-Signal ausgesendet. Dies

geschieht im Zustand “S_ BROADCAST_ WAKEUP”. Das Wake-Up-Signal wird für die Dauer von 10 ms gesendet. Ist diese Zeit abgelaufen, wird kontrolliert, ob alle Zellensensoren aufgewacht und bereit sind. Die Zustände hierfür sind “S_ TX_ IS_ AWAKE” und “S_ RX_ IS_ AWAKE”. Nach dieser Kontrolle wird im Sekundentakt ein Samplebefehl als Broadcast Kommando ausgesendet. Der Zellensensor antwortet daraufhin mit einer Sendung des aufgenommenen Spannungswerts. Die Ausgabe dieses Werts erfolgt über die RS-232 Schnittstelle an den PC.

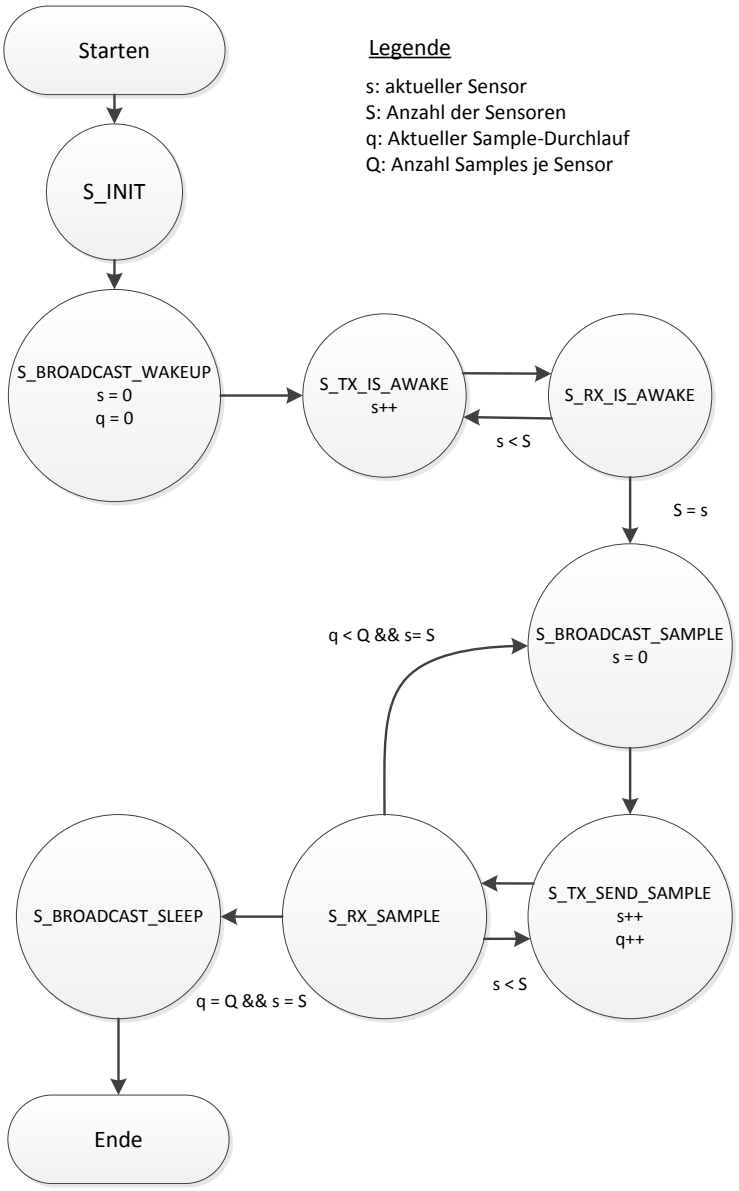


Abbildung 2.6.: Zustandsdiagramm für die Software des Batteriesteuergeräts nach [4], S.100

2.2. Anforderungsanalyse der Hardware

2.2.1. Mechanische Anforderungen

Der neue Zellensensor V0.2 soll wie der Sensor V0.1 wieder an einer Lithium-Eisenphosphat-Batteriezelle, kurz LiFePo4 eingesetzt werden. Eine solche Batteriezelle kann eine elektrische Ladung von bis zu 48Ah bei 3,3 V haben [5]. Die hier verwendete Zelle kommt von der Firma ECC Repenning GmbH aus Geesthacht. Diese Batteriezellen wurden bereits in einer vergangenen Arbeit als Starterbatterie in einem Fahrzeug eingesetzt und erfolgreich getestet [16]. Der bisherige Zellensensor war dafür vorgesehen, direkt im Gehäuse der Batteriezelle verbaut zu werden. Mit dem neuen Zellensensor V0.2 soll dies auch weiterhin möglich sein. Es soll aber auch die Möglichkeit bestehen, ihn direkt auf den Gehäusen der Batteriezelle zu montieren.
Das Gehäuse der LiFePo4 Zelle besitzt an der positiven- sowie an der negativen Seite jeweils ein M8 x 1,25 Innengewinde zum sicheren Anschluss der Batteriezellen [5].

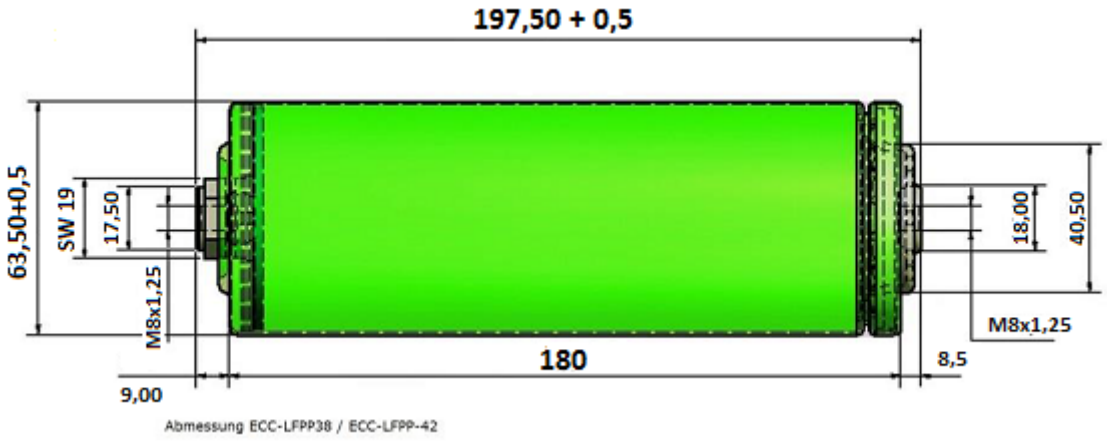


Abbildung 2.7.: Abmessungen einer Rundzelle der Firma ECC Repenning GmbH Entnommen aus [5], S.1

Um die Zellensensorplatine auf der Rundzelle montieren zu können, muss also ein entsprechendes Befestigungsloch beim Redesign der Platine vorgesehen werden.

2.2.2. Ladungsbalancierung

Die Ladungsbalancierung ist ein Verfahren um einzelne Zellen auf das gleiche Ladungsniveau zu bringen. Batteriezellen besitzen fertigungsbedingt Schwankungen in ihrer Kapazität,

ihrer Selbstenladungsrate und ihrer Impedanz [8]. Dadurch das üblicherweise einzelne Zellen seriell bzw. in Reihe zu einer größeren Batterie zusammengeschaltet werden, führt dies zu verschiedenen Problemen.
Durch den unterschiedlichen Innenwiderstand der Zellen werden diese dann unterschiedlich stark belastet. Dies kann dazu führen, dass bei einer Entladung einzelne Zellen tiefenentladen werden, während andere Zellen noch Ladung haben.
Beim Aufladen der Zellen ist dieser Effekt genau umgekehrt. Schwächere Zellen werden nicht komplett aufgeladen, bzw. stärkere Zellen überladen. Um dies zu verhindern, müssen die Zellen auf einen gleichen Ladungsstand gebracht werden. Obwohl es technisch möglich ist, Ladung von stärkeren auf schwächere Zellen zu übertragen, ist dieses Verfahren aufgrund fehlender Verdrahtung nicht praktikabel. Es wird daher eine einfache Variante der passiven Ladungsbalancierung verwendet, bei der die stärkeren Zellen auf das Niveau der schwächeren entladen werden. Eine einfache Methode um den Zellen Ladung zu entziehen ist, parallel an die Zelle eine Last zu schalten. Auf den Zellensensoren V0.1 wurde dies mit zwei parallelgeschalteten 20Ω[1] Widerständen realisiert. Durch die Verteilung des Gesamtwiderstands auf vier verschiedene Teilwiderstände, wird eine größere Verlustleistung als bei nur einem Widerstand erreicht.
Diese sogenannten Balancierungswiderstände wurden in der V0.1 dicht nebeneinander platziert. Erste Versuche mit der Balancierung ergaben, dass sich während der Balancierung die Widerstände stark erhitzen. Deshalb sollen diese auf der neuen Zellensensorplatine räumlich getrennt werden um die Hitze besser zu verteilen. Zudem sollen die beiden parallelgeschalteten Balancierungspfade einzeln geschaltet werden können. Dies ermöglicht zwei unterschiedlich starke Balancierungsarten. Das Ein- und Ausschalten der Nebenstrompfade übernimmt der Mikrocontroller.

2.2.3. Platzierung des Temperatursensors

Der eingesetzte Temperatursensor TMP102 wurde bisher durch den Mikrocontroller des Zellensensors noch nicht angesteuert. Dieser kann, bei einem Einsatz des Zellensensors in der Batteriezelle, dessen Temperatur überwachen.
Wichtig ist aber auch eine Überwachung der Temperatur der Ladungsbalancierung. Wie bereits erwähnt, kann es während der Balancierung zu einer starken thermischen Belastung durch die Balancierwiderstände kommen. Aus diesem Grund soll der Temperatursensor in der Nähe der Balancierungswiderstände platziert werden.

[1] bestehend aus zwei in Reihe geschalteten 10Ω Widerständen

2.2.4. Abschaltung des DC/DC-Wandlers

Ein bekanntes Problem des DC/DC-Wandlers ist die Beeinflussung der Eingangsspannung durch interne Schaltvorgänge. Diese verursachen Störungen auf der Eingangsseite des DC/DC-Wandlers. Da diese Spannung aber gemessen werden soll, ist es wichtig, die Störungen zu minimieren oder gar zu beseitigen. Da das Problem bereits bekannt ist, und es im Projekt BATSEN bereits Erfahrungen mit dem eingesetzten DC/DC-Wandler TPS61201 von Texas Instruments gibt, existiert die Lösung durch kurzzeitiges Abschalten der Wandler. Durch das Abschalten des Wandlers wird die zu messende Eingangsspannung nicht mehr durch die Schaltvorgänge beeinflusst und es kann die Batteriespannung gemessen werden. Der eingesetzte TPS61201 besitzt dazu einen Eingang, der zum An- und Abschalten des Wandler dient. Dieser wurde bei dem Zellensensor V0.1 nicht mit dem Mikrocontroller verbunden. In der neuen Version des Zellensensors soll die Möglichkeit bestehen, den Wandler mittels dem Mikrocontroller zu steuern. Dies ist beim Redesign der Zellensensorplatine zu beachten.

2.3. Anforderungsanalyse der Software

Die neue Softwareversion des Batteriesteuergeräts, sowie des neuen Zellensensors V0.2 soll verschiedene Funktionen unterstützen.

Unterstützte Funktionen des Zellensensors V0.2

- Adressierbares Wake-Up
- Einzelne Spannungsmessung
- Temperaturmessung TMP102
- Temperaturmessung MSP430
- Ladungsbalancierung
- Funksynchronisiertes Messverfahren

2.3.1. Das Protokoll

Bei der bidirektionalen Kommunikation zwischen dem Batteriesteuergerät und den Zellensensoren bindet der Transceiver CC1101 die zu sendenden Daten in ein Übertragungsframe ein. Zusätzlich zu den Daten, die er vom Mikrocontroller erhält, fügt der CC1101 eine vier Byte lange Präambel, vier Sync Byte und zwei Byte CRC Prüfsumme hinzu. Diese werden auf der Empfangsseite automatisch wieder entfernt und bei erfolgreichem Empfang die Daten an den Mikrocontroller weitergegeben. Bei der Betrachtung des verwendeten Protokolls fällt auf, dass die Länge der zu sendenden Daten fest ist. Nicht genutzte Datenbyte werden also trotzdem übertragen. Dies kann teilweise zu langen Übertragungszeiten führen.
Im Hinblick auf die in dieser Arbeit neu eingeführte funksynchronisierte Messung, muss dieses Protokoll überarbeitet werden, da bei diesem Messverfahren große Datenmengen anfallen. Werden diese Datenmengen über das bisherige Protokoll gesendet, kommt es zu sehr langen Sendezeiten.
Bei einer angenommenen Messung, wobei 1.000 Messwerte mit je 2 Byte an das Batteriesteuergerät übertragen werden sollen, ergibt sich die Gesamtzahl von 200 zu sendenden Paketen, wenn pro Paket nur 10 Datenbyte übertragen werden können.

$$AnzahlderPakete = \frac{1.000 \cdot 2Byte}{10Byte/Paket} = 200Pakete$$

Die Abbildungen 2.9 und 2.8 zeigen den genutzten Frameaufbau einer Sendung. Eine Uplink-Sendung hat hierbei eine effektive Datenlänge von 11 Byte, wobei 1 Byte das Adressbyte darstellt. Es können also 10 Byte an Daten gesendet werden.

Abbildung 2.8.: Frameaufbau Uplink

Der Downlink hingegen besitzt eine Datenlänge von 6 Byte. Dabei ist 1 Byte wieder ein Adressbyte. Es können somit maximal 5 Datenbyte im Downlink an den Zellensensor übertragen werden.

Abbildung 2.9.: Frameaufbau Downlink

Der Transceiver CC1101 enthält einen FIFO-Ringbuffer in der Größe von 64 Byte [7], in den die gesendeten Daten geschrieben werden können. Diese werden dann in ein gemeinsames Paket aufbereitet und gesendet. Durch bestimmte Ansteuerung des Transceiver lassen sich auch noch mehr Bytes pro Paket versenden. Um die Ansteuerung des CC1101 einfach zu halten, wird auf diese Möglichkeit verzichtet. Es wurde daher entschieden, die maximale Länge der Daten auf unter 64 Byte pro Paket zu beschränken. Welche Datenlänge sich für die Versendung der, bei der funksynchronisierten Messung anfallenden Daten eignet, muss in der späteren Arbeit genauer untersucht werden.

2.3.2. Die Software des Batteriesteuergeräts

Die Software des Batteriesteuergeräts muss an die neuen Funktionen angepasst werden. Bisher wurde lediglich eine Funktion durch das Batteriesteuergerät ausgeführt, das Aussenden des Wake-Up-Signals. Anschließend wurde im Sekundentakt eine Spannungsmessung abgerufen.
Ausgelöst wurde diese Messung durch einen Tastendruck auf einen der drei vorhanden Tasten. Durch die neuen umfangreichen Funktionen muss eine neue Möglichkeit gefunden werden, wie sich das Batteriesteuergerät steuern lässt.
Möglich wäre ein Aufbau einer Menüstruktur, die durch die drei vorhandenen Taster gesteuert wird. Als Anzeige steht das 16 x 2 LC Display zur Verfügung. Angesichts der geplanten

Anzahl der Funktionen scheint diese Art der Steuerung als sehr umständlich. Eine andere Möglichkeit ist die Steuerung des Batteriesteuergeräts über die vorhandene RS232 Schnittstelle. Dadurch lässt sich eine leichte Steuerung ermöglichen. In der bisherigen Software des Batteriesteuergeräts wurde die RS232 Schnittstelle lediglich zum Senden der aufgenommenen Spannungswerte von dem Batteriesteuergerät zum PC genutzt.

2.3.3. Die Zellensensorsoftware

Im Folgenden werden die wichtigsten Softwareänderungen aufgelistet, die zur Implementierung der neuen Sensorfunktionen nötig werden, oder in der Vorarbeit von [4] offen geblieben sind.

Einbindung des Temperatursensors

Um den auf dem Zellensensor vorhandenen Temperatursensor TMP102 nutzen zu können, ist es nötig, diesen in der Software einzubinden. Er wird über I2C angesprochen und konfiguriert. Wie bereits im Abschnitt 2.2.3 auf Seite 25 angesprochen, soll der Temperatursensor zur Überwachung der Zelltemperatur und der Temperatur der Balancierungswiderständen dienen. Für die Überwachung der Zelltemperatur soll die “Alarmfunktion” des Sensors genutzt werden. Bei dieser Alarmfunktion lassen sich zwei Temperaturschwellen im Sensor einstellen, bei dem ein Ausgangspin wahlweise gesetzt oder gelöscht wird [10]. Dieser Pin kann dann vom Mikrocontroller als Interrupt ausgewertet werden, sodass für die Überwachung keine zusätzliche Rechenzeit geopfert werden muss. So lässt sich die Zelltemperatur auch im Schlafzustand (LPM4)[1] des Zellensensors überwachen und bei einer Überhitzung der Zelle können entsprechende Gegenmaßnahmen ergriffen werden.
Bei der Überwachung der Balancierungswiderstände soll hingegen die aktuelle Temperatur abgerufen und durch den Mikrocontroller ausgewertet werden.

Einbindung der Balancierung

Im Zellensensor V0.1 konnte der Balancierungspfad lediglich an- und ausgeschaltet werden. Gesucht ist nun eine intelligente Steuerung, die die Balancierung regelt. Gewünscht ist, dass das Batteriesteuergerät mit der Balancierung wenig belastet wird. Dieses soll lediglich den Befehl zur Balancierung und den zu erreichenden Spannungswert der Zelle an den Zellensensor aussenden. Die restliche Steuerung soll dem Zellensensor überlassen werden.

[1] LPM4 ist der tiefste Schlafzustand des MSP430F235

Dieser soll die Zellspannung bis zum vorgegebenen Spannungswert balancieren. Dabei sollen aber bestimmte Temperaturen nicht überschritten werden. Wird diese Grenze dennoch überschritten, soll die Balancierung bis zum Erreichen einer unteren Temperaturgrenze pausieren.

Adressierbares Wake-Up

In der Zellensensorsoftware soll ein adressierbares Wake-Up realisiert werden. Der Zellensensor soll dabei erkennen, ob er aufgeweckt werden soll, oder ob dieser wieder in den Schlafzustand (LPM4) zurückkehren kann.
Da das Wake-Up-Signal von dem Batteriesteuergerät nur an alle Zellensensoren geschickt werden kann und dies nicht adressierbar ist, muss ein Weg gefunden werden, der es möglich macht, dass nur einzelne Sensoren aufgeweckt werden und andere weiter im Schlafmodus bleiben.

Umschalten der DCO-Frequenz

Der interne Digital Controlled Oscillator (DCO) des Mikrocontrollers MSP430F235 ist in der Lage, seinen Takt während des Betriebs umzustellen [13]. Angedacht ist eine Umstellung des DCO-Takts von der normalen Taktfrequenz von 1 MHz zu einer Frequenz von 16 MHz. Da dies bisher noch nicht getestet wurde, muss dies in den Voruntersuchungen hinsichtlich der Einschwingzeit des DCO und der Einsetzbarkeit getestet werden.

3. Analyse und Konzeption der Burstmessung

3.1. Die Idee der Burstmessung

Der Name “Burstmessung” ist innerhalb des Forschungsprojets BATSEN entstanden. Es soll eine funksynchrone Messmethode sein, die eine hohe Anzahl von Spannungswerten innerhalb kurzer Zeit, von mehreren Zellensensoren liefern kann. Es soll also ein “Burst” an Messwerten aufgenommen werden. Somit entstand der Name der Burstmessung, wie es im weiteren auch genannt wird.

3.2. Anforderungsanalyse der Burstmessung

Die Burstmessung soll eine neue Messmethode sein, mit der es möglich ist, zeitlich hochgenaue Spannungsmessungen durchzuführen. Diese Messungen sollen durch einen Takt, ausgehend vom Batteriesteuergerät, ausgelöst werden. Dabei soll zwischen der Aussendung des Taktes und der Aufnahme der Spannung auf dem Zellensensor eine möglichst hohe synchronität herrschen. Durch eine gleichzeitige Stromaufnahme am Batteriesteuergerät, lassen sich die aufgenommenen Spannungswerte mit den Stromwerten zeitlich vergleichen. Ein Fernziel dadurch ist die Nutzung der synchron aufgenommenen Spannungs- und Stromwerte für eine Impedanzspektroskopie der einzelnen Batteriezellen. Ein Verfahren, dass heutzutage bereits häufig in der Batterietechnik angewandt wird, um aussagen über die Gesundheit der Batterien zu machen. Für die angestrebte Impedanzspektroskopie werden in der Literatur [22] Frequenzen bis zu 1 kHz als aussagekräftig angegeben. Um dies zu erreichen, ist eine minimale Messfrequenz von 2 kHz nötig, da das Abtasttheorem besagt “*Ein mit der Grenzfrequenz f_g bandbegrenztes Signal $f(t)$ wird vollständig durch einzelne Signalwerte beschrieben, die im Abstand $\Delta t = 1/(2 \cdot f_g)$ entnommen werden*”[1]. Somit ist eine Anforderung der Burstmessung eine minimale Abtatsfrequenz von 2 kHz. Neben der

[1] Entnommen aus [18], S.59

Grundlagenbildung für die Impedanzspektroskopie, soll die Burstmessung auch für die Erfassung dynamischer Vorgänge genutzt werden können. So ist ein möglicher Einsatzbereich dieser Messmethode, die Erfassung der Spannung während den sogenannten Hochstromereignissen. Diese Hochstromereignisse sind meist kurzzeitige und hohe Belastungen für die Batterie. Dies kann z.B. bei einem Motorstart eines PKW sein. Dabei fließen kurzzeitig hohe Ströme, die einen Spannungseinbruch an den Batteriezellen verursachen. Diese Spannungseinbrüche an den einzelnen Zellen soll durch die Burstmessung aufgezeichnet werden können.

Abbildung 3.1 zeigt den Startvorgang eines Mercedes Benz Vito L, 4 Zylinder Diesel, 95kW, 2.0 l. Diese Aufnahmen wurden in der Diplomarbeit von Simon Püttjer [19] aufgenommen. Diese Aufnahmen wurde mithilfe zweier Oszilloskope die direkt an den einzelnen Zellen angeschlossen waren aufgenommen.

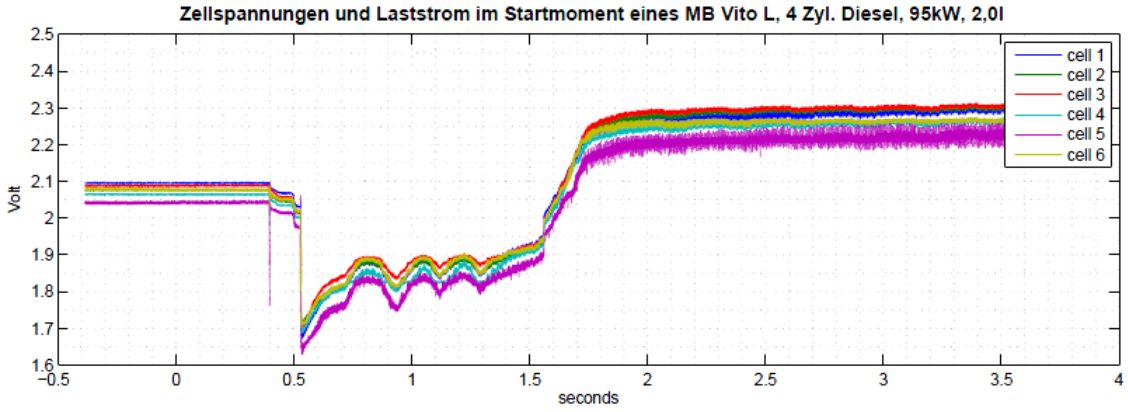


Abbildung 3.1.: Zellspannungen in einer Starterbatterie Entnommen aus [19], S. 46

Zu sehen sind hier die einzelnen Zellspannungen während des Motorstarts. Um diese Vorgänge durch die Burstmessung erfassen zu können, ist eine schnelle Abtastung der Spannungen erforderlich. Dies fordert eine schnelle Messaufnahme und Verarbeitung der Daten auf der Seite des Zellensensors. Der interessante Bereich des Spannungsverlaufs in Abbildung 3.1 erstreckt sich über ca. 1,5 s. Durch die Erfassung dieses Bereichs und durch die gewünschte hohe Abtastung fallen dabei eine Vielzahl von Daten an. Diese Daten müssen wieder an das Batteriesteuergerät gesendet werden. Dazu ist eine effektive Datenübertragung notwendig.

Ein Ziel der Burstmessung soll es also sein, solche dynamischen Vorgänge, wie den Spannungseinbruch beim Start eines Motors zeitlich möglichst genau erfassen zu können. Dies soll im folgen Untersucht werden. Diese Untersuchung soll später dann auch Grundlage für die Impedanzspektroskopie sein, welches allerdings nicht Teil dieser Arbeit ist.

3.3. Konzeption der Burstmessung

Im Rahmen dieser Arbeit werden verschiedene Konzepte zur Realisierung der Burstmessung und zur Erfüllung der gegebenen Anforderungen untersucht. Zu Untersuchen sind dabei verschiedene Übertragungsarten die zur Synchronisation der Zellensensoren durch das Batteriesteuergerät eingesetzt werden können. Eine Anforderung an die Übertragung zwischen dem Zellensensor und Batteriesteuergerät soll dabei eine möglichst geringe Verzögerung des Taktsignals sein. Wichtig ist ebenfalls, das diese Verzögerung zeitlich stabil ist.

3.3.1. Synchronisation durch Paketsendung

Eine einfache Lösung der Synchronisierung ist es, den Messzeitpunkt durch das Batteriesteuergerät paketorientiert zu senden. Dazu schickt das Batteriesteuergerät den Zellensensoren einen Broadcast Befehl, der den Messvorgang startet. Ähnlich wie bei der einzelnen Spannungsmessung, die bereits mit dem Zellensensor möglich ist. Ein wesentlicher Vorteil dieser Methode ist es, dass diese leicht zu realisieren ist, da diese Art der Kommunikation zwischen Batteriesteuergerät und Zellensensor bereits erprobt und erfolgreich eingesetzt wurde. Ein Nachteil ist, dass durch die Aufbereitung der Pakete im Transceiver Latenzzeiten entstehen, die nicht beeinflussbar bzw. kontrollierbar sind. Weiter benötigt die Aufbereitung des Pakets bereits mehrere Millisekunden. Dadurch ist die Messfrequenz von 2 kHz nicht erreichbar. Da dies allerdings eine der Anforderungen an die Burstmessung ist, wird dieses Konzept nicht weiter verfolgt.

3.3.2. Synchronisation durch direkte Übertragung

Eine weitere Möglichkeit zur Synchronisierung der Zellensensoren ist es, ein Signal mittels dem sogenannten “asynchronen Modus” zu versenden. Die Bezeichnung als asynchronen Modus ist die Herstellerbezeichnung von Texas Instruments und wird im weiteren auch so verwendet. Der asynchronen Modus bedeutet eine direkte Aufmodulierung eines anliegenden Signals. Wie bei der Übertragung des Wake-Up-Signals wird hier ein 434 MHz-Trägersignal übertragen, welches durch den Mikrocontroller an- und abgeschaltet werden kann. Abbildung 3.2 soll dieses Prinzip verdeutlichen. Zu sehen ist ein 25 Hz Taktsignal, welches Übertragen wird. Dieses Taktsignal wird vom Mikrocontroller gesteuert und an den Transceiver übergeben. Dieser sendet daraufhin das Trägersignal bei einem anliegenden High-Pegel. Bei einem anliegenden Low-Pegel des Taktsignals, wird die Trägersendung unterbrochen. Diese Sendeart wird als On-Off-Keying (OOK) bezeichnet. Welche auch bei der paketorientierten Übertragung genutzt wird. Dabei übernimmt aber der Transceiver selbst die Steuerung dieses Signals.

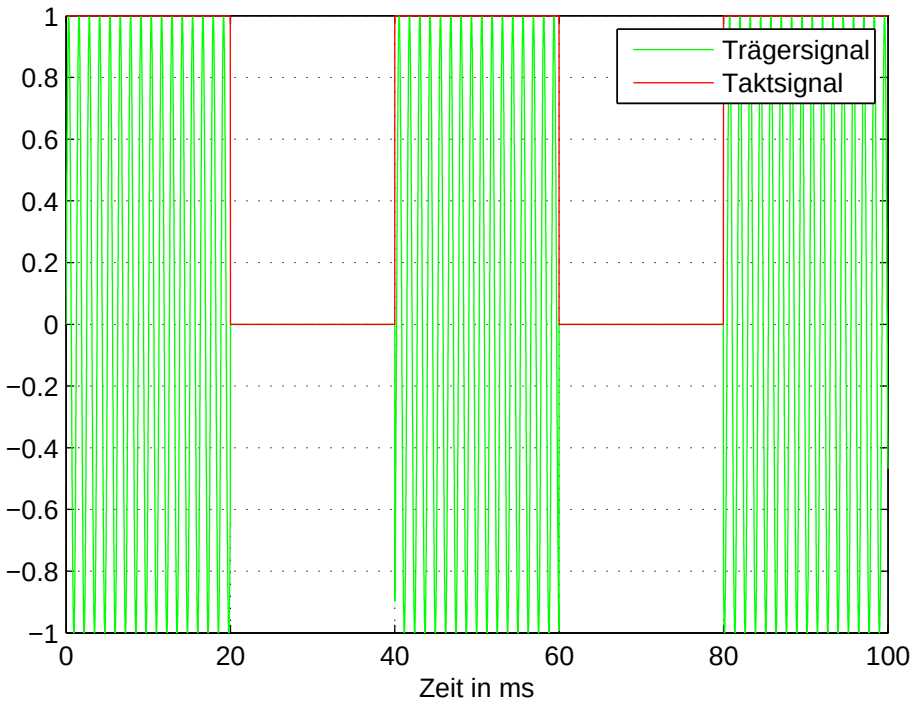


Abbildung 3.2.: Übertragung eines 25 Hz Taktsignals

Da der asynchrone Modus bereits für das Senden des Wake-Up-Signals genutzt wurde, muss die Möglichkeit des Sendens nicht weiter untersucht werden.
Dennoch muss überprüft werden, ob auch das Empfangen dieses Signals möglich ist, da das Wake-Up-Signal auf dem Zellensensor nicht vom Transceiver, sondern von dem Wake-Up-Chip AS3930 von der Firma AMS ausgewertet wird [4]. Laut Datenblatt des Transceiver CC1101 lässt sich das demodulierte Signal über einen der drei GPIO Pins ausgeben. Dieses muss dann vom Mikrocontroller ausgewertet werden. Wichtig für die Synchronität der Messung ist, dass die Latenzzeit zwischen Batteriesteuergerät und Zellensensor nicht zu groß ist. Eine mögliche Latenzzeit kann, verursacht durch die Modulation auf dem Batteriesteuergerät und der Demodulation auf dem Zellensensor, auftreten.

3.3.3. Geschwindigkeit

Im Normalbetrieb des Zellensensors läuft der Mikrocontroller MSP430F235 mit einer Taktgeschwindigkeit von 1 MHz. Da die mögliche Geschwindigkeit der Burstmessung abhängig von der Taktrate des Zellensensors ist, ist das Ziel, die Taktrate während der Burstmessung zu erhöhen. Der MSP430F235 bietet dazu die Möglichkeit, den DCO-Takt während

des Betriebs umzustellen. Einstellbar ist dabei ein maximaler Takt von 16MHz. Weitere Zwischenwerte sind ebenfalls einstellbar, werden aber nicht weiter untersucht, da für die Burstmessung ein maximaler Takt erwünscht ist.

3.3.4. Energiebedarf

Das Energieverhalten während der Burstmessung wird in dieser Arbeit nicht weiter zu betrachtet, da hier vordergründig die generelle Realisierbarkeit der Burstmessung untersucht wird. Es wird außerdem vermutet, dass der Energieverbrauch der Burstmessung insgesamt unkritisch ist, da es sich um eine Kurzzeitmessung handelt. Ferner ist eine geplante Anwendung der Burstmessung das erfassen von Hochstromereignissen, wie z.b. beim Motorstart in einem Verbrennungsmotor eines PKW. Der maximale Energieverbrauch des Sensors ist hierbei etliche Größenordnungen kleiner als der Energiebedarf des Startermotors und kann somit vernachlässigt werden.

3.3.5. Die Übertragung der Daten

Es ist sinnvoll die Übertragung der Daten an das Batteriesteuergerät erst nach Beendigung der Burstmessung durchzuführen. Grund hierfür ist die benötigte Sendezeit des Zellensensors. Betrachtet man nur die reine Sendezeit, die der Transceiver für das Senden der Daten benötig, stößt man sehr schnell an eine maximale Burstgrenze. Bei 40 kBaud Übertragungsrate und einer minimalen Datenmenge von 13 Byte, bestehend aus 4 Byte Preamble, 4 Byte Sync, 1 Byte Zellensensoradresse, 2 Byte Messdaten und 2 Byte CRC-Prüfsumme ergibt sich eine rechnerische Übertragungszeit von 2,6 ms.

$$t_{Send} = \frac{13Byte \cdot 8Bit}{40kBaud/s} = 2,6ms$$

Selbst bei der maximal möglichen Übertragungsrate von 250 kBaud/s ergibt sich immer noch eine reine Sendezeit von 416μs.

$$t_{Send} = \frac{13Byte \cdot 8Bit}{250kBaud/s} = 416\mu s$$

Da jeder Sensor einen Timeslot dieser Länge zur Übertragung der Daten benötigt addiert sich diese Zeit mit jedem weiteren Sensor. Bei vier Zellensensoren, die ihre Daten senden müssen, ist man bei 250 kBaud bereits bei einer reinen Sendezeit von 1,66 ms. Damit lässt sich lediglich eine Burstfrequenz von maximal 600 Hz erreichen. Allerdings sind darin die Verarbeitungszeiten des Mikrocontrollers noch nicht mit einberechnet, welche die maximale

Burstrate nochmals um ein Vielfaches verringern würde. Deshalb wurde ein Konzept mit sofortiger Sendung der Daten nicht weiter verfolgt.
Da die Daten also auf dem Zellensensor zwischengespeichert werden sollen, benötigt der Mikrocontroller genügend Speicher. Der eingesetzte Controller besitzt einen internen RAM-Speicher von 2 kByte [13]. Da der verwendete A/D-Umsetzer eine Genauigkeit von 12 Bit besitzt, werden für eine Messwertspeicherung 2 Byte benötigt. Rechnerisch können somit maximal 1.000 Messwerte im internen RAM gespeichert werden. Dies reicht vorerst aus, um die Messmethode zu erproben.

4. Realisierung und Redesign

4.1. Neugestaltung des Protokolls

Der bisherige Frame ist aus verschiedenen Kommunikations- und Nutzdatenbyte aufgebaut.

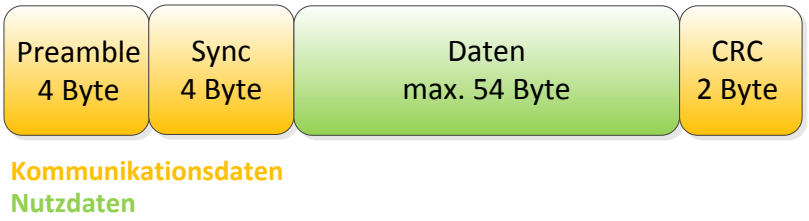


Abbildung 4.1.: Kommunikations- und Datenbyte

Mit dem hier eingesetzten Transceiver CC1101 von Texas Instruments ist es möglich, eine maximale Framelänge von 64 Bytes zu senden [7]. Die Kommunikationsbytes werden vom Transceiver beim Senden automatisch hinzugefügt und beim Empfangen wieder entfernt, sodass nur die gesendeten Nutzdaten beim Empfänger an den Mikrocontroller weitergegeben werden.
Die Kommunikationsbytes bestehen am Anfang der Sendung aus 4 Preamble- und 4 Sync-Byte. Diese dienen im Transceiver zur Erkennung des Beginns der Nutzdaten. Nach den Nutzdaten folgt noch eine 2 Byte lange CRC-Prüfsumme, die zur Detektion von Sendefehlern dient. Durch das Hinzukommen neuer Funktionen des Zellensensors, ist eine Neugestaltung des Protokolls zwischen dem Batteriesteuergerät und den Zellensensoren nötig. Wie bereits in der Anforderungsanalyse der Software erwähnt, soll die Nutzdatenlänge dynamisch an die jeweilige Funktion angepasst werden können. Die Länge der Nutzdaten soll dabei nicht größer als 64 Byte sein, um eine einfache Kommunikation zu ermöglichen.

Eine Vorüberlegung ergibt, dass bei der Sendung vom Zellensensor zum Batteriesteuergerät (Uplink) immer die Adresse des Steuergeräts und die Adresse des sendenden Zellensensors mitgesendet werden.

Bei einer Sendung von dem Batteriesteuergerät an den Zellensensor (Downlink) wird immer die Adresse des jeweiligen Zellensensors, und das auszuführende Kommando gesendet. Dies bedeutet, dass bei einer Up- und Downlink Sendung immer mindestens 2 Byte an Nutzdaten gesendet werden. Entschieden wurde einen Header mit der Größe von 4 Byte zu implementieren, wie sie im Abbildung 4.3 und Abbildung 4.4 zu sehen sind. In den zusätzlichen 2 Byte können noch Steuerbefehle oder Zusatzinformationen mitgesendet werden. Die Länge der zu sendenden Daten wurde auf maximal 50 Byte beschränkt, um die Übertragung der Burstdaten zu vereinfachen.

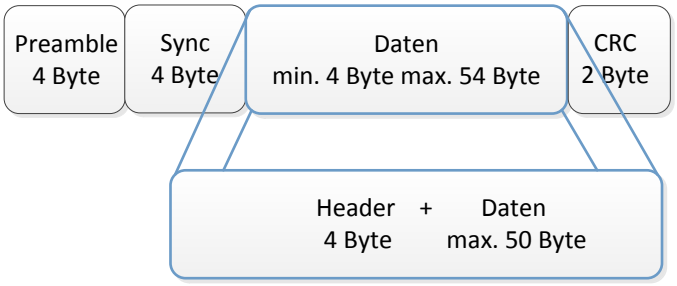


Abbildung 4.2.: Neuer Frameaufbau mit Header

Somit ist es nun möglich, 50 Byte an Nutzdaten mit einer Sendung zu übertragen. Berechnet man das Beispiel aus der Anforderungsanalyse der Software (siehe S.27) erneut, so stellt man fest, dass nun bei einer Übertragung von 1.000 Messwerten, mit jeweils 2 Byte Länge, eine Gesamtanzahl von 40 Paketen ausreicht, um alle Daten an das Batteriesteuergerät zu übertragen.

$$\text{Anzahl der Pakete} = \frac{1.000 \cdot 2Byte}{50Byte/Paket} = 40 \text{ Pakete}$$

Messungen der Sendezeit am CC1101 die mit einer Sendegeschwindigkeit von 100 kBaud aufgenommen wurden ergaben, dass nicht die Größe der Nutzdaten in einem Paket, sondern die Anzahl der zu sendenden Pakete einen wesentlichen Einfluss auf die Gesamtübertragungszeit der Daten hat. Hierzu wurde ein Paket mit 6 Byte Daten versendet. Dieses benötigte 3,44 ms zum Aufbereiten und zum Versenden der Daten. Zum Vergleich wurde ein Paket mit insgesamt 54 Nutzdatenbyte versendet. Vermutet wurde, dass das Versenden nun ungefähr 9 mal länger dauert, da die neunfache Menge an Daten zu versenden ist. Dies würde eine Zeit von ca. 30,96 ms bedeuten. Tatsächlich benötigte der Transceiver lediglich 9,92 ms für das Aufbereiten und Versenden der Daten. Dies zeigt, dass es zeitlich gesehen effektiver ist, mehr Daten in einem Paket zu versenden.

Ein Nachteil daran ist allerdings, dass das komplette Paket bei einem Übertragungsfehler verworfen werden muss, und somit mehr Daten verloren gehen.

Die beiden Abbildungen 4.3 und 4.4 zeigen jeweils den Aufbau des neuen Header einer Uplink- bzw. Downlink-Sendung.

Abbildung 4.3.: Header einer Uplink-Sendung

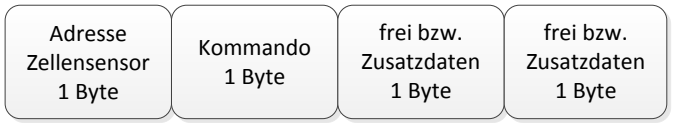


Abbildung 4.4.: Header einer Downlink-Sendung

4.2. Erweiterung der Sensorfunktionen

Der in der Bachelorarbeit von Phillip Durdaut [4] entwickelte Zellensensor V0.1 der Klasse 3 soll nun durch verschiedene Funktionen erweitert werden. Die von Phillip Durdaut entwickelte Software ist in der Lage, einzelne Spannungswerte zu messen und diese an das Batteriesteuergerät zu übertragen. Die Tabelle 4.1 gibt einen Überblick zwischen dem in [4] entwickelten Zellensensor V0.1 und dem nun hier entstehenden Zellensensor V0.2.

Tabelle 4.1.: Übersicht der Zellensensorfunktionen

Zellensensorfunktionen	Zellensensor V0.1	Zellensensor V0.2
Konfiguration des ZS durch die BS	-	✓
einzelne Spannungsmessung	✓	✓
Temperaturmessung durch TMP102	-	✓
Temperaturmessung durch MSP430	-	✓
Ladungsbalancierung	-	✓
Adressierbares Wake-Up	-	✓
Burstmessung	-	✓

Da der Aufbau des Sendeframe geändert wurde, sind auch die bereits vorhandenen Funktionen in der Software anzupassen. Dies wird im folgenden ebenfalls erläutert.

4.2.1. Konfiguration des Zellensensors durch das Batteriesteuergerät

Eine neue Funktion ist, den Zellensensor durch das Batteriesteuergerät zu konfigurieren. Zweck dieser Konfiguration ist es, dem Zellensensor aktuelle Grenzen und Werte zu übermitteln. So können die oberen und unteren Temperaturgrenzen für die Balancierungssteuerung eingestellt werden. Auch die Übertragungsrate der Zellensensoren lassen sich dadurch einstellen. Falls der Zellensensor nicht durch das Batteriesteuergerät konfiguriert wird, nimmt der Zellensensor seine Standartwerte, die in der Software der Zellensensoren hinterlegt sind.

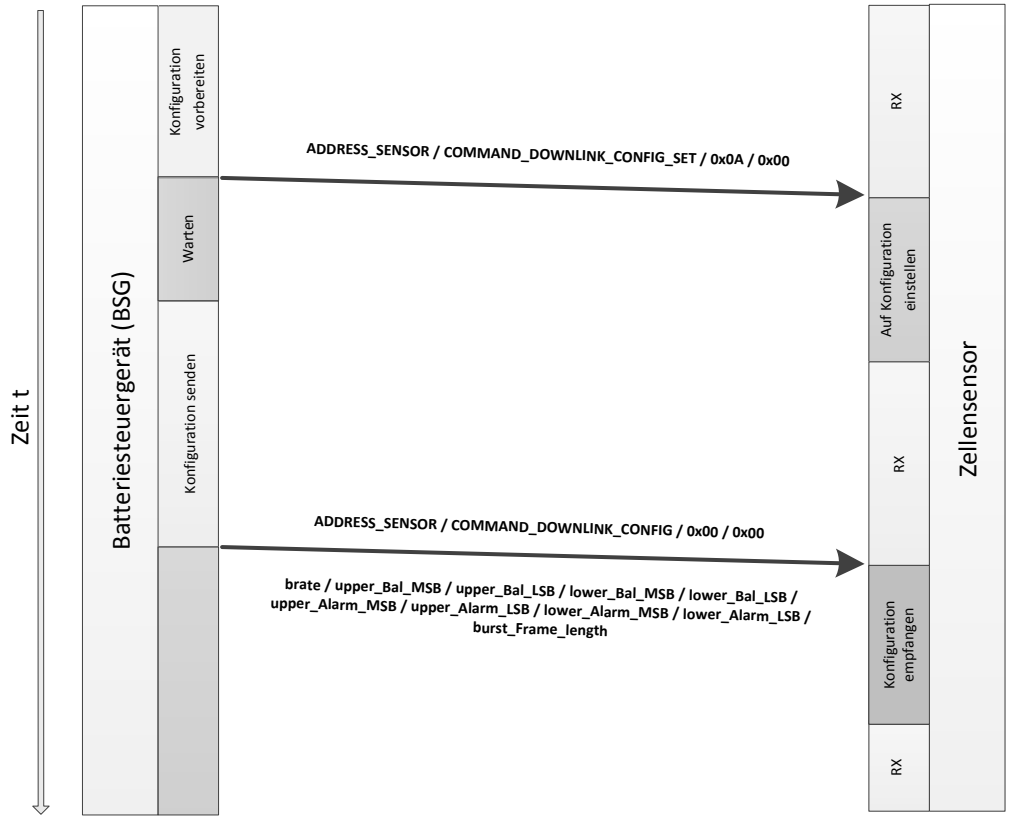


Abbildung 4.5.: Zeitlicher Ablauf der Konfiguration

In Abbildung 4.5 ist der Ablauf dieser Konfiguration zu sehen. In der ersten Sendung schickt das Batteriesteuergerät an den Zellensensor dem Befehl zur Vorbereitung auf den Empfang der Konfiguration. Das dritte Byte dieser Sendung (0x0A)[1] gibt dem Zellensensor an, dass die nächste Sendung 10 weitere Byte enthält. Damit stellt sich der Zellensensor auf

[1] Dezimal=10

den Empfang von insgesamt 14 Datenbyte ein. Bestehend aus dem 4 Byte großen Header und weiteren 10 Byte, die die Konfigurationsdaten enthalten. Die in der Abbildung 4.5 zu sehenden Abkürzungen, bei der Sendung der Konfiguration, werden in folgender Tabelle kurz erläutert.

Tabelle 4.2.: Übersicht der in der Konfiguration gesendeten Daten

Abkürzung	Bedeutung
brate	Übertragungsrate
upper Bal	obere Balancierungsgrenze
lower Bal	unter Balancierungsgrenze
upper Alarm	obere Alarmgrenze
lower Alarm	untere Alarmgrenze
burst Frame lenght	Datenlänge einer Burstsendung

4.2.2. Einzelne Spannungsmessung

Die Spannungsmessung ist eine Funktion, die bereits in der Softwareversion des Zellensensors V0.1 vorhanden ist. Das Batteriesteuergerät sendet dazu einen Befehl zur Spannungsmessung als Broadcast an alle Zellensensoren. Diese nehmen, über ihren im Mikrocontroller integrierten 12-Bit A/D Umsetzer, die aktuell anliegende Batteriespannung auf und speichern diese in jeweils 2 Byte ab. Diese beiden Byte werden als "*Voltage_ MSB*" und "*Voltage_ LSB*" bezeichnet und werden so lange gespeichert, bis das Batteriesteuergerät eine neue Spannungsaufnahme durch einen Broadcastbefehl auslöst. Den zeitlichen Ablauf sowie die nötige Paketzusammenstellung veranschaulicht Abbildung 4.6. Zu sehen ist dabei, dass der Zellensensor bei der letzten Sendung sechs Byte sendet. Bestehend aus dem 4 Byte langen Header, plus den beiden Byte, die die aufgenommenen Spannungswerte enthalten.

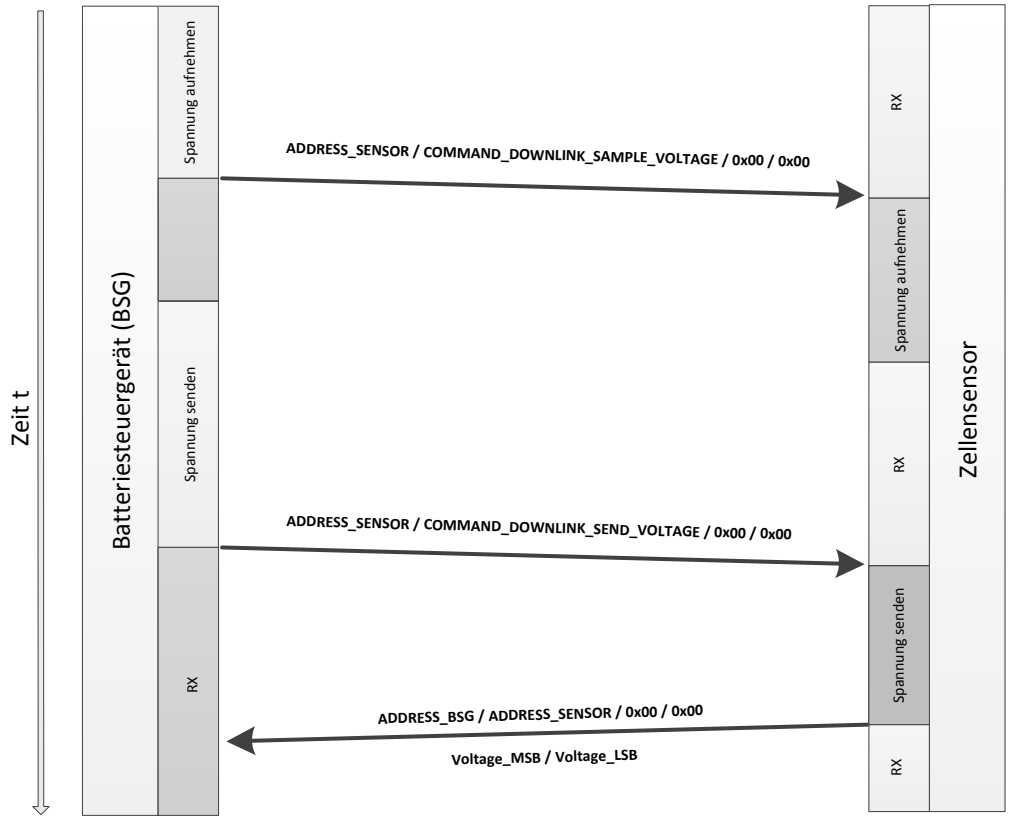


Abbildung 4.6.: Zeitlicher Ablauf einer Spannungsmessung

4.2.3. Temperaturmessung durch TMP102

Die Temperaturmessung über den Temperatursensor TMP102 [10] soll eine neue Funktion des Zellensensors V0.2 sein. Zur Auslösung dieser Messung gibt das Batteriesteuergerät einen Broadcastbefehl an die Zellensensoren mit dem Befehl der Temperaturmessung aus. Die Zellensensoren messen dann die aktuelle Temperatur mit Hilfe des Temperatursensors TMP102. Dieser gibt die Temperatur in einer 12-Bit Auflösung wieder, welche sich über eine I2C Schnittstelle durch den auf dem Zellensensor befindlichen Mikrocontroller abrufen lässt. Nachdem das Batteriesteuergerät den weiteren Befehl zur Temperatursendung an einen einzelnen Zellensensor ausgegeben hat, antwortet dieser unmittelbar danach mit einer Sendung, bestehend aus dem 4 Byte großen Header und den 2 Byte, die die aufgenommene Temperatur enthalten. Die Kommunikation läuft als genau wie bei der Spannungsmessung ab. Ein zeitlicher Ablaufplan der Temperaturmessung über den TMP102 befindet sich im Anhang auf Seite 133.

4.2.4. Alarmfunktion

Der Temperatursensor TMP102 verfügt ebenso über eine Alarmfunktion. Dabei gibt der Temperatursensor beim erreichen eines bestimmten Werts, ein digitales Signal aus, welches vom Mikrocontroller ausgewertet wird. Dazu lassen sich zwei verschiedene Temperaturen angeben, bei dem der Temperatursensor einen “Alarm” ausgeben soll. Diese Alarmgrenzen werden mittels der Konfiguration durch das Batteriesteuergerät festgelegt (siehe dazu Abschnitt 4.2.1). Sie können also jederzeit geändert und somit auf verschiedene Situationen angepasst werden. Eine weiter Verwendung dieser Alarmfunktion ist in der jetzigen Software nicht vorgesehen. Dadurch, dass die Funktion vollkommen in der Initialisierung und der Konfiguration eingebaut ist, kann diese jedoch jederzeit verwendet werden.

4.2.5. Temperaturmessung durch MSP430

Eine weitere Möglichkeit zur Temperaturmessung bietet der Mikrocontroller selbst an. Dieser besitzt intern einen Temperatursensor mit 12-Bit Auflösung. Diese Messung bietet sich besonders während der Balancierung an, da der Temperatursensor TMP102 durch die Wärmeentwicklung an den Balancierungswiderständen nicht mehr die aktuelle Zelltemperatur messen kann. Die Ansteuerung durch das Batteriesteuergerät läuft hierbei genauso wie bei dem Temperatursensor TMP102 ab. Es unterscheidet sich lediglich in den gesendeten Befehlen. Ein zeitlicher Ablauf dieser Messung befindet sich im Anhang auf Seite 134.

4.2.6. Spannungs- und Temperaturmessung

Eine weitere Funktion der Software des Zellensensors V0.2, ist eine kombinierte Spannungs- und Temperaturmessung. Diese besteht aus einer Messung der Batteriespannung durch den 12-Bit A/D Wandler des Mikrocontrollers und einer Temperaturmessung durch den TMP102. Dazu besitzt das Batteriesteuergerät einen eigenen Befehl. Dieser wird als Broadcastbefehl an die Zellensensoren gesendet und startet daraufhin die Spannungs- und Temperaturmessung. Danach kann das Batteriesteuergerät die Daten von jedem einzelnen Sensor nacheinander abrufen. Erhält ein Zellensensor den Befehl zum Senden der Daten, schickt dieser ein 8 Byte großes Uplink Paket. Dieses Paket besteht wieder aus dem 4 Byte großen Header und den beiden, jeweils 2 Byte großen Daten der Spannungs- und Temperaturmessung. Abbildung 4.7 zeigt den zeitlichen Ablauf dieser Abfrage.

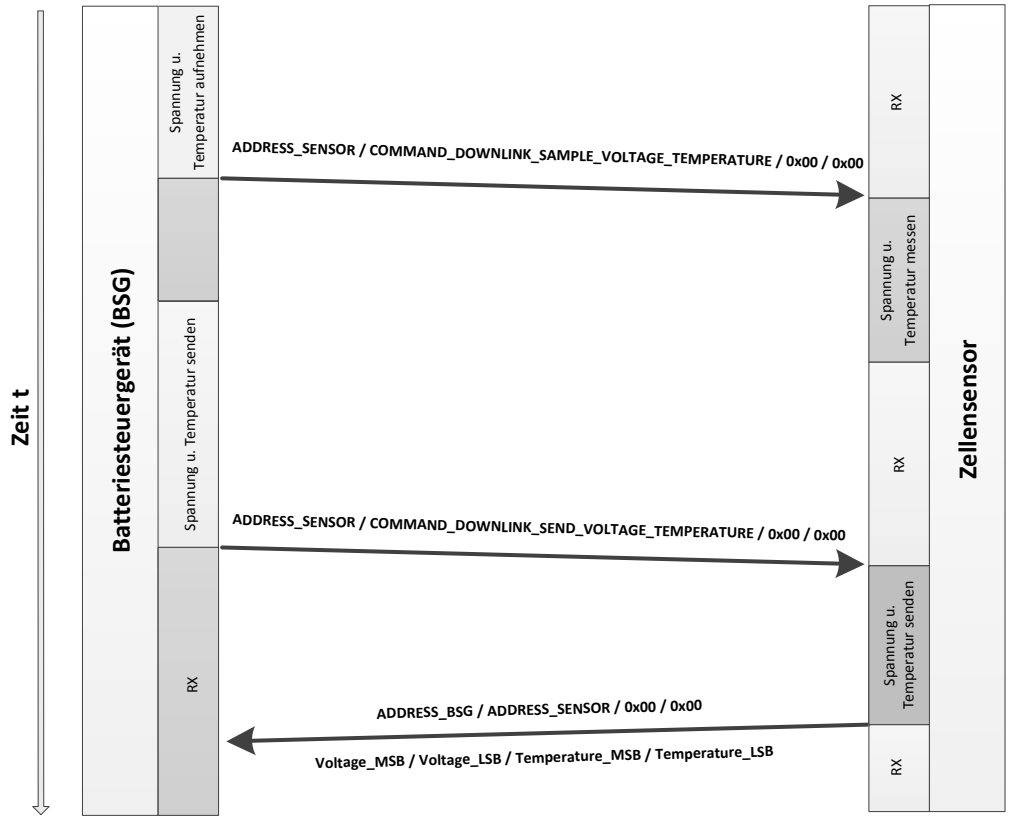


Abbildung 4.7.: Zeitlicher Ablauf einer Spannungs- und Temperaturmessung

4.2.7. Ladungsbalancierung

Durch die Möglichkeit der passiven Ladungsbalancierung kommt eine wichtige Funktion für den Betrieb an Lithium-Ionen-Zellen hinzu. Implementiert wurde hier eine eigenständige Ladungsbalancierungssteuerung auf dem Zellensensor. Das heißt, der Zellensenor erhält lediglich den Befehl zur Balancierung durch das Batteriesteuergerät und übernimmt die restliche Steuerung selbst. Das Batteriesteuergerät wird somit nicht weiter durch die Balancierung auf dem Zellensensor belastet. Neben dem Befehl zur Balancierung enthält das Downlink Paket noch einen Spannungswert, der durch die Balancierung erreicht werden soll.

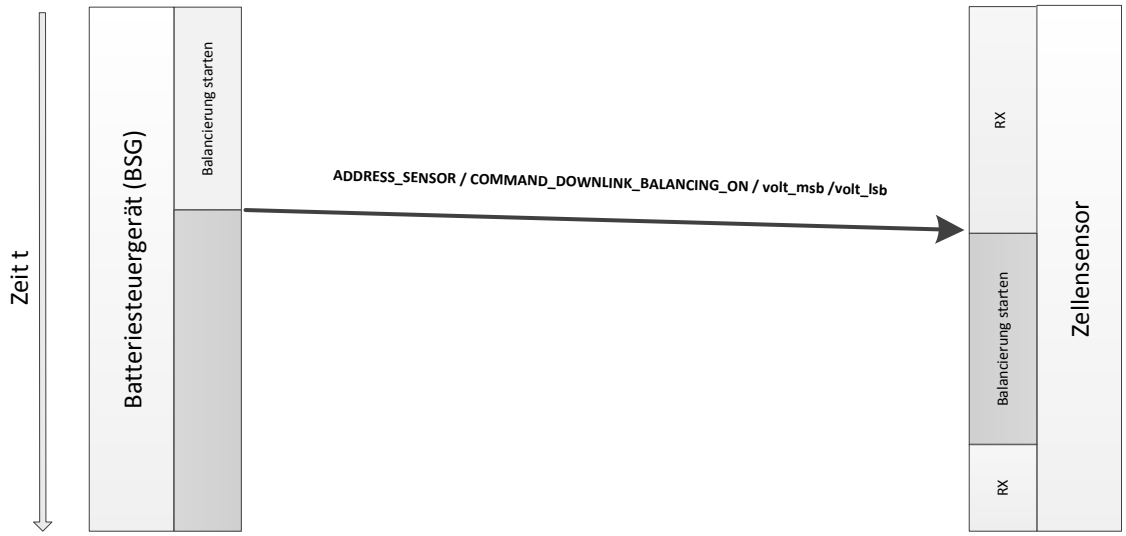


Abbildung 4.8.: Header Downlink Balancierung anschalten

Ein wichtiger Punkt der Steuerung ist die Temperaturüberwachung der Balancierungswiderstände. Bei Voruntersuchungen konnten an diesen, Temperaturen von weit über 70 °C festgestellt werden. Durch den Temperatursensor lässt sich ein Schutz vor Überhitzung dieser Widerstände bzw. der thermischen Belastung der Zelle realisieren. Platziert wurde der Sensor in unmittelbarer Nähe zweier Balancierungswiderstände. Die Steuerung wurde so konzipiert, dass nur unterhalb einer bestimmten Temperatur balanciert werden darf. Wird diese obere Temperaturgrenze erreicht, wird mit der Ladungsbalancierung pausiert, bis die Temperatur unter eine untere Temperaturgrenze gefallen ist. Ist diese unterschritten, wird die Balancierung wieder fortgesetzt. Sind die Temperaturbedingungen erfüllt, wird die aktuelle Batteriespannung gemessen und mit der zu erreichenden Spannung der Zelle verglichen. Ist die Spannung, die durch das Batteriesteuergerät vorgegeben ist, noch nicht erreicht, wird 500 ms später wieder mit der Temperaturmessung begonnen und der Ablauf wiederholt sich. Ist die Zielspannung erreicht, wird die Balancierung beendet. Abbildung 4.9 zeigt den Ablauf dieser Steuerung.

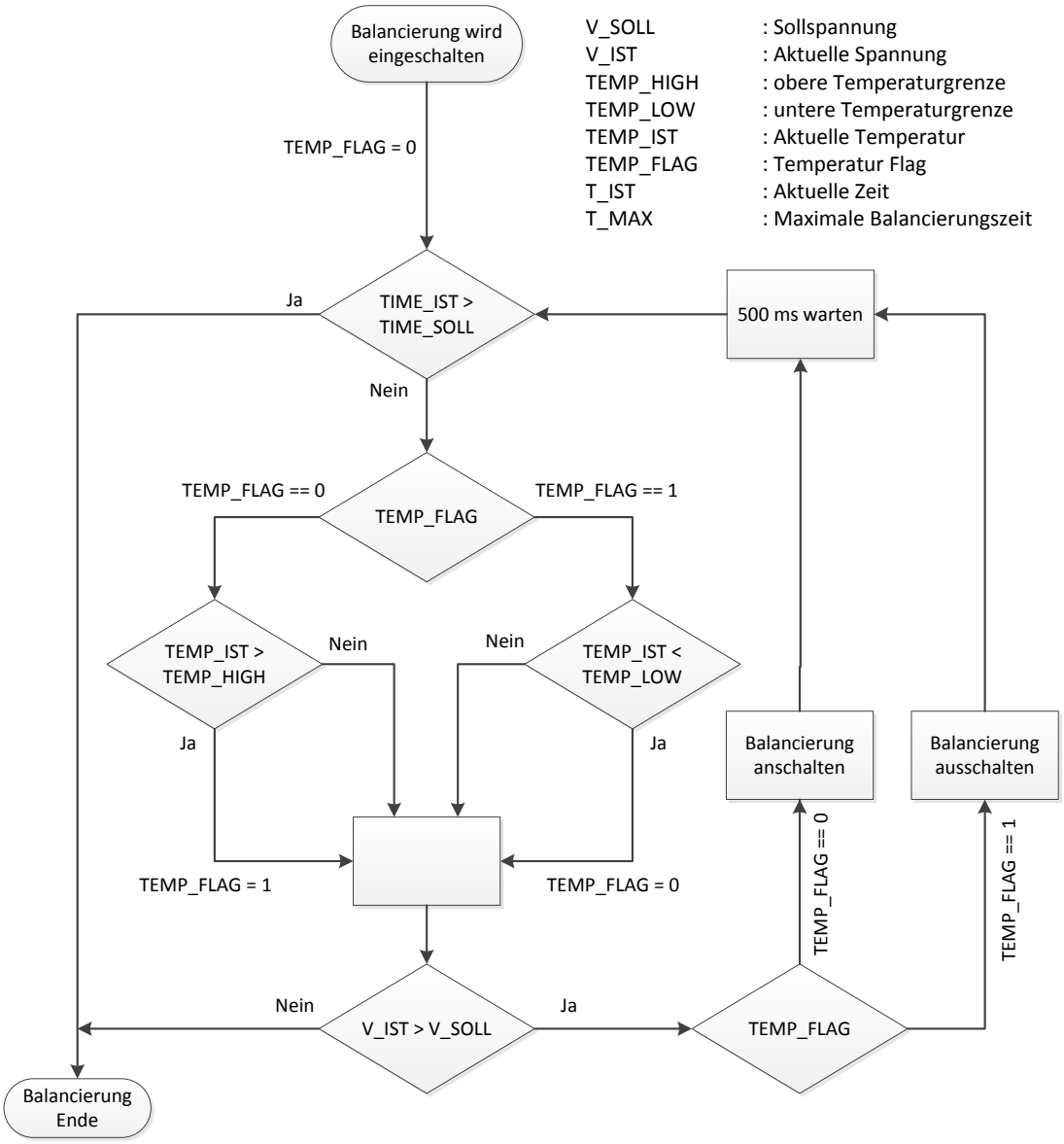


Abbildung 4.9.: Ablauf der Ladungsbalancierung mit Temperatur- und Zeitkontrol-le

Ein theoretischer Temperaturverlauf der Steuerung zeigt die nachfolgende Abbildung 4.10.

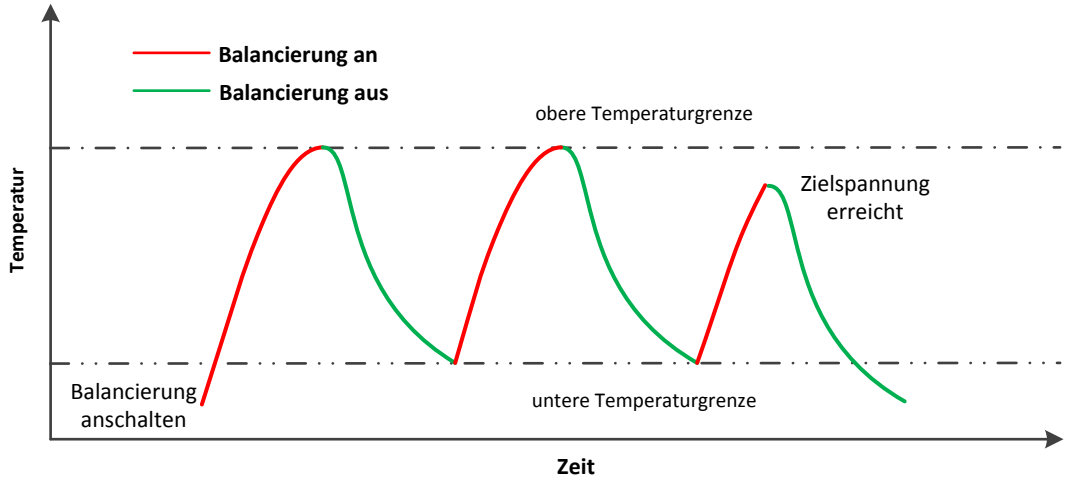


Abbildung 4.10.: Theoretischer Temperaturverlauf der Balancierungssteuerung

Darüber hinaus ist für die Steuerung der Ladungsbalancierung eine maximale Balancierungszeit vorgegeben. Diese wird in jedem Durchlauf der Steuerung kontrolliert. Dies hat den Zweck, die Balancierung während einer Ladephase der Zelle nach einer bestimmten Zeit zu unterbrechen. Wird die Zelle während eines Balancierungsvorgangs geladen, kann der vorgegebene Spannungswert nicht erreicht werden, da die Zelle ständig nachgeladen wird. Um dies zu verhindern, ist in der Steuerung eine maximale Zeit vorgegeben. Diese kann jederzeit in der Software angepasst werden.

Wird während der normalen Balancierungszeit der vorgegebene Spannungswert nicht erreicht, muss das Batteriesteuergerät einen erneuten Balancierungsbefehl an den Zellensensor senden.

4.2.8. Adressierbares Wake-Up

In der bisherigen Zellensensorsoftware ließen sich die Zellensensoren nur gemeinsam durch ein, von dem Batteriesteuergerät ausgesandtes, Wake-Up-Signal aufwecken. Will man nun aber nur einen bestimmten Zellensenor aufwecken, ist dies bisher nicht möglich. Gesucht wird nun nach einer Möglichkeit einzelne Zellensensoren aufzuwecken und die anderen Sensoren im Schlafmodus (LPM4) zu lassen. Der Vorteil dieser Methode ist, dass ein einzelner Sensor aufgeweckt werden kann, welcher eine Spannungs- und/oder Temperaturmessung an einer einzelnen Zelle durchführt, ohne das die anderen Sensoren dazu wach sein müssen. Das Batteriesteuergerät schickt dazu zunächst das Wake-Up-Signal an alle Zellensensoren. Da dieses Signal nur an alle Zellensensoren gesendet werden kann, wachen zunächst

alle auf. Nun schickt das Batteriesteuergerät ein Kommando an den gewünschten Sensor der wach bleiben soll. Ist an die Zellensensoren, die nun wach bleiben sollen das Kommando geschickt worden, folgt ein weiterer Befehl zum Abschluss des Wake-Up-Vorgangs an alle Zellensensoren. Hat nun ein Zellensensor kein Kommando zum wach bleiben erhalten, geht dieser wieder in den Schlafzustand über. Abbildung 4.11 zeigt dabei den schematischen Ablauf des adressierbaren Wake-Up.

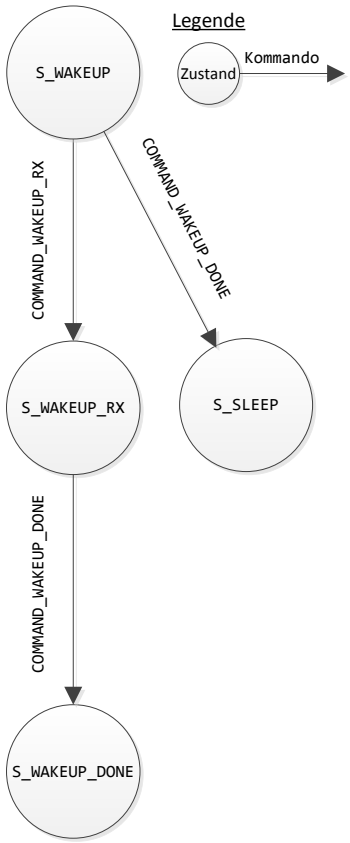


Abbildung 4.11.: Ablauf des adressierbaren Wake-Up

4.3. Erweiterung der Funktion des Batteriesteuergeräts

Eine Erweiterung der Funktion des Batteriesteuergeräts war hinsichtlich der Eingabemöglichkeit notwendig. Bisher konnte lediglich der Startzeitpunkt zum Aussenden des Wake-Up-Signals bestimmt werden. Dies erfolgte durch einen Tastendruck am Batteriesteuergerät. Da diese Möglichkeit der Eingabe für die umfassenden Funktionen nicht ausreicht, ist die vorhandene RS232 Schnittstelle als Eingabemöglichkeit genutzt worden.

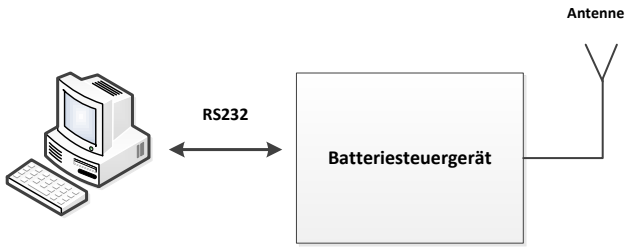


Abbildung 4.12.: Verbindung zwischen Batteriesteuergerät und PC

Dabei wird die Auswahl der verschiedenen Funktionen mit Hilfe einer einzugebenden Zahlenkombination realisiert. Eine Liste mit den entsprechenden Zahlenkombinationen befindet sich im Anhang auf Seite 136. Alternativ lässt sich diese Liste auch im Terminal, durch die Eingabe von “help” anzeigen. Dadurch lässt sich Folgende Ausgabe generieren.

```
Help menue:
0  -> Wake-up and config
1  -> Sending wake-up only
2  -> Checking wake-up
3  -> Set sensor for config
4  -> Sending the config
5  -> Sample voltage
6  -> Sending voltage
7  -> Sample voltage and temperature of the TMP102
8  -> Send voltage and temperature of the TMP102
9  -> Sample temperature of the TMP102
10 -> Send temperature of the TMP102
11 -> Sample temperature of the MSP430
12 -> Send temperature of the MSP430
13 -> Start burst measurement
14 -> Get config data for burst measurement
15 -> Start burst data transmission
16 -> Balancing on
17 -> Balancing off
```

Abbildung 4.13.: Help Menue

Das Batteriesteuergerät initialisiert nach dem Start zunächst alle für den Betrieb wichtigen Komponenten. Danach wird auf die Eingabe durch den Benutzer über die RS232 Schnittstel-

le gewartet. Die Eingabe wird als beendet interpretiert, wenn der Terminator “CR[2]” durch das Batteriesteuergerät empfangen wurde. Danach beginnt die Auswertung und Ausführung des Befehls. Nach der Ausführung ist das Batteriesteuergerät wieder bereit, eine neue Eingabe durch den Benutzer zu empfangen. Abbildung 4.14 zeigt den stark vereinfachten Ablauf der Befehlseingabe über die RS232 Schittstelle im Batteriesteuergerät.

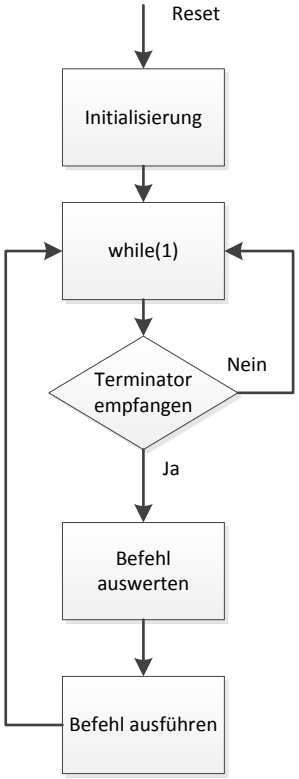


Abbildung 4.14.: Ablauf der Steuerung über RS232

Die Einstellungen für das Senden und Empfangen über die RS232 Schnittstelle lauten

Einstellungen RS-232

- 28.800 Baud
- 8 Datenbit
- 1 Stopbit
- Keine Parität
- Terminator CR

[2]Carriage Return

4.4. Burstmessung

4.4.1. Taktübertragung

Für die Funktion der Burstmessung ist eine korrekte Taktübertagung von dem Batteriesteuergerät zu den Zellensensoren notwendig. Bisher wurde diese Art der Übertragung auch schon beim Senden des Wake-Up-Signals genutzt. Bei dieser Funktion wurde das empfangene Signal aber von dem Hüllkurvendemodulator des Wake-Up-Pfades ausgewertet. Bei der Burstmessung soll das Taktsignal aber vom Transceiver CC1101 demoduliert werden. Dazu ist die Qualität und die Demodulationszeit des empfangenen Signals im Transceiver zu untersuchen. Zum Test der Übertragung wird durch das Batteriesteuergerät ein dauerhaftes 100Hz Taktsignal an die Zellensensoren gesendet. Dieses Taktsignal hat ein Tastverhältnis von 50%. In Abbildung 4.15 ist der gesendete Takt des Batteriesteuergeräts (BS) und der auf dem Zellensensor (ZS) empfangene Takt zu sehen. Das Signal “Takt der BS” ist das Taktsignal, welches von dem Batteriesteuergerät an den Eingang des Transceiver gegeben wird. Auf dem Zellensensor wurde das Signal hinter dem Transceiver CC1101 abgegriffen, zu sehen als “Takt am ZS”. Gut zu erkennen ist dabei das 100 Hz Signal des Batteriesteuergeräts.

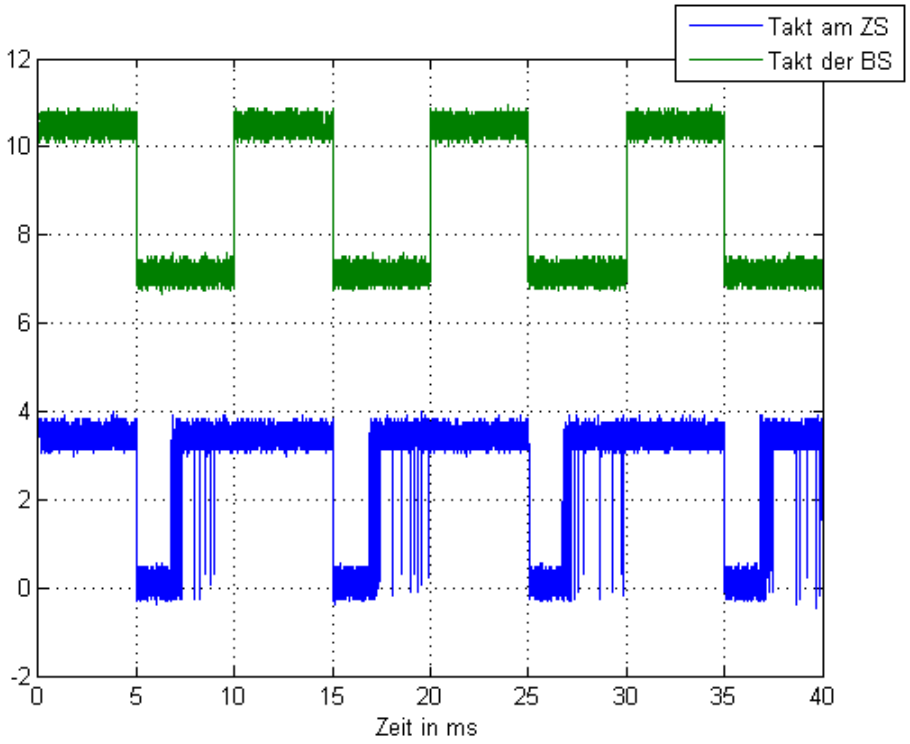


Abbildung 4.15.: Gesendetes Taktsignal / 100Hz

Auf der Seite des Zellensensors lässt sich dieses Signal mit der selben Periode wieder erkennen. Allerdings ist das Tastverhältnis von 50% nicht mehr gegeben und es sind starke Störungen im Signal zu erkennen. Dadurch, dass dieses Signal direkt als Interrupt vom Mikrocontroller ausgewertet werden soll, ist dieses Verhalten störend. Durch eine Analyse des demodulierten Taktsignals lässt sich feststellen, dass diese Störungen erst nach einer gewissen Zeit auftreten, immer nach ca. 1,81 ms. Zudem treten nur Störungen auf, wenn der Träger des CC1101 abgeschaltet ist. Dies lässt darauf schließen, dass diese Störungen von Transceiver selbst verursacht werden. Durch das Datenblatt des CC1101 [7] wurde festgestellt, dass diese Störungen von einem internen “Auto Gain Controller” verursacht werden. Dieser versucht im Falle eines fehlenden Trägersignals, den Transceiver nachzuregeln. Diese Nachregelung verursacht die Störungen am Ausgang.

Das Taktsignal wir nun von dem Batteriesteuergerät in einem anderen Tastverhältnis gesendet, sodass der Low-Pegel des Signals maximal 500μs andauert. Dies verhindert das der Transceiver versucht das Signal nachzuregeln und verhindert damit die Störungen.

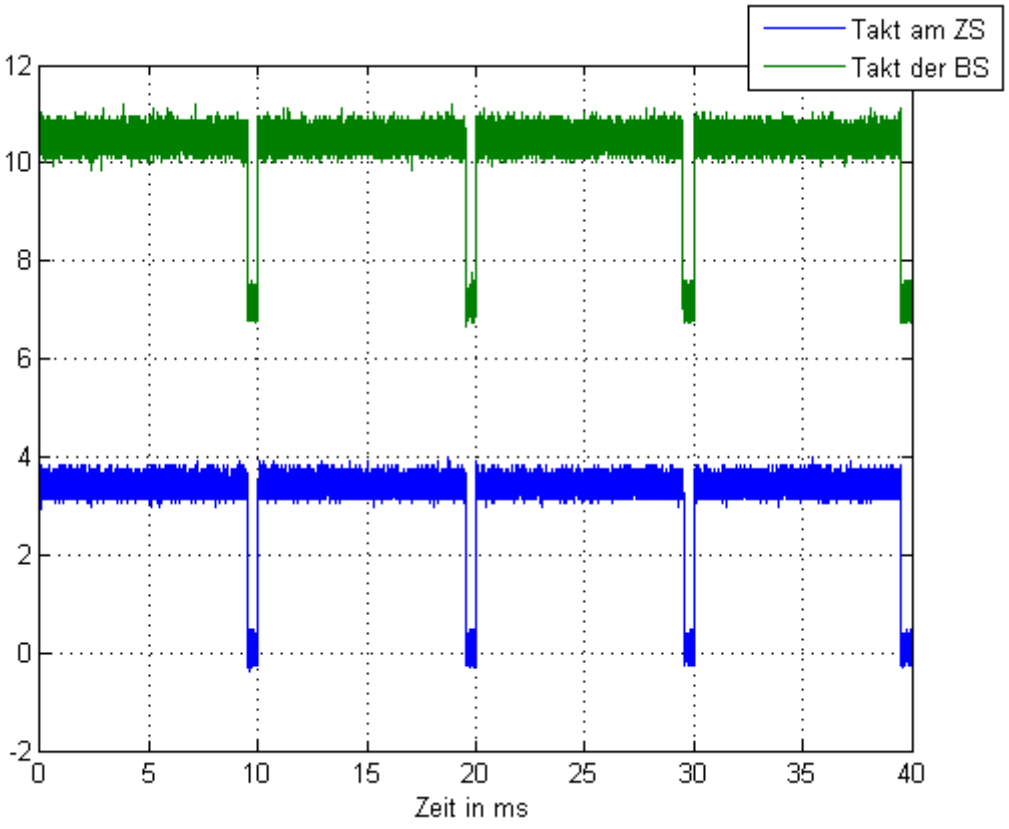


Abbildung 4.16.: Fensterung des Bursttakts

Fensterung des Bursttakts

In den Voruntersuchungen hat sich gezeigt, dass die Qualität der Taktübertragung bei ausreichendem Empfang der Zellensensoren sehr gut ist. Dennoch kann es vorkommen, dass einzelne Fehler in der Übertragung auftreten. Diese Fehler machen sich als kurze Pegeleinbrüche am Ausgang des Transceiver bemerkbar.

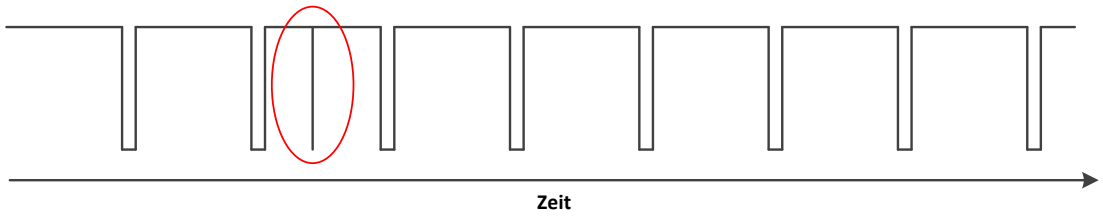


Abbildung 4.17.: Takt mit einzelnem Fehler

Da bereits kurze Pegeleinbrüche, am Mikrocontroller zu einer Fehlinterpretation des Taktsignals und somit zu fehlerhaften Messungen führen, muss dieses Problem behoben werden. Ursache dieser Störungen sind Fehler in der Übertragung des Trägersignals. Es wurde entschieden, diese Störungen durch eine Fensterung des Taktsignals zu lösen. Da die Frequenz des übertragenen Taktes dem Zellensensor bekannt ist, betrachtet der Mikrocontoller nur die Pegeleinbrüche unmittelbar vor und nach dem ihm vermutenden Zeitpunkt. In Abbildung 4.18 ist das Taktsignal mit einem Pegeleinbruch zu sehen. Die grün dargestellten Bereiche sind die Zeiten, in dem ein Pegelwechsel vom Mikrocontroller erwartet wird.

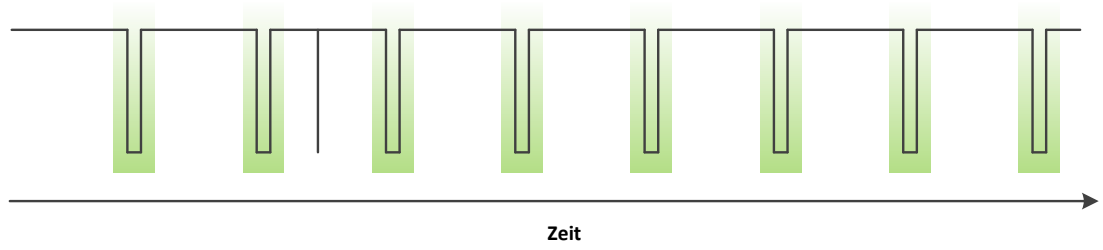


Abbildung 4.18.: Fensterung der Bursttakte

Erreicht wird diese Fensterung durch eine Steuerung mit Hilfe eines Timers. Da die Taktfrequenz bekannt ist, kann der Timer entsprechend auf die erwartete Periode eingestellt werden. Der Timer wird durch den internen DCO des Mikrocontrollers getaktet. Da dieser eine Abweichung von bis zu 6% [12] haben kann, ist es nötig, einen Toleranzbereich für die Fensterung zu bestimmen. Als Toleranz wurde zunächst ein Bereich von $\pm$ 200μs ausgewählt. Für kleinere Burstfrequenzen (<300Hz) kann dieser Bereich unter Umständen zu

klein sein, da die Periodenzeit dieser Frequenzen größer ist und sich dadurch die DCO-Abweichungen bemerkbar machen können. Die Mindestbreite des Toleranzbereichs errechnet sich wie folgt.

$$\frac{\text{Periodenzeit}}{100\,\%} \cdot \text{max. Abweichung des DCO [\%]} = \text{min. Toleranzbereich [s]} \tag{4.1}$$

Bei einer Frequenz von 300Hz ist der Toleranzbereich gleich der möglichen Abweichung des Timer.

$$\frac{3,33\ ms}{100\%} \cdot 6\,\% = 200\,\mu s$$

Dies kann zur Folge haben, dass das gesendete Taktsignal des Batteriesteuergeräts nicht mehr in den Bereich der Fensterung fällt. Falls dieses Problem auftreten wird, müsste lediglich die Fensterbreite des Zellensensors bei der entsprechenden Frequenz vergrößert werden. Da mit einer so hohen DCO-Abweichung unter Laborbedingungen aber nicht zu rechnen ist, wurde für die Untersuchung des Messprinzips für alle Frequenzen dieselbe Fensterbreite gewählt.

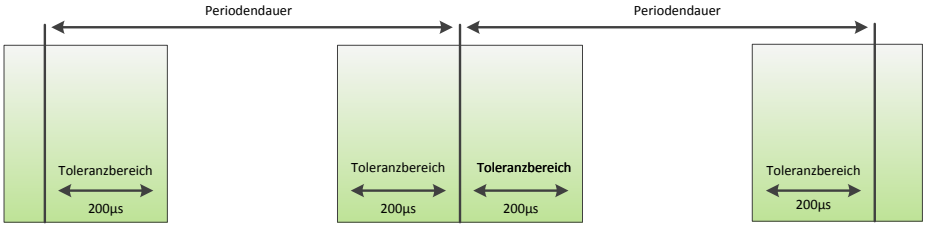


Abbildung 4.19.: Toleranzbereich der Fensterung

4.4.2. Softwarerealisierung des Batteriesteuergeräts

Das Batteriesteuergerät übernimmt bei der Burstmessung die zentrale Steuerung aller Zellensensoren. Es gibt den Messtakt und die Anzahl der zu messenden Werte vor. Um den Zellensensoren mitzuteilen, dass nun eine Burstmessung folgt, wird zunächst der Befehl zur Burstmessung als Broadcast an alle Zellensensoren gesendet. Im Header dieser Sendung stecken zudem noch die Informationen über die Burstfrequenz sowie die Anzahl der aufzunehmenden Werte. Abbildung 4.20 zeigt den Aufbau des Header zur Burstmessung.

Abbildung 4.20.: Header des Sendebefehls der Burstmessung

Diese zusätzlichen Informationen sind für die Konfiguration des Zellensensors notwendig, da sich dieser auf die entsprechende Burstfrequenz einrichten muss. Nachdem diese Sendung ausgesandt wurde, kann mit der Umstellung des CC1101 Sendemodus begonnen werden. Im normalen Betrieb sendet diese im paketorientierten Modus. Bei der Burstmessung wird hingegen mit dem nicht paketorientierten Modus gesendet. Im Datenblatt von Texas Instruments wird diese Sendeart auch “asynchroner Modus” genannt. In dieser Sendeart moduliert der Transceiver lediglich das Signal, welches er an den Eingangspin bekommt, auf die Trägerfrequenz auf und versendet dies. Dabei fügt er dem Signal keine weiteren Informationen hinzu. Nach der Umstellung muss noch eine Wartezeit eingefügt werden, damit sichergestellt ist, dass sich die Zellensensoren auch auf die Burstmessung umgestellt haben. Nach dieser Zeit kann der Timer, der den Messtakt ausgibt, gestartet werden. Dabei werden die ausgesendeten Takte durch das Batteriesteuergerät gezählt. Ist die geforderte Anzahl an Takten ausgesendet, wird für 100 ms ein Dauerträger gesendet. Abbildung 4.21 zeigt den schematischen Ablauf in dem Batteriesteuergerät während der Burstmessung.

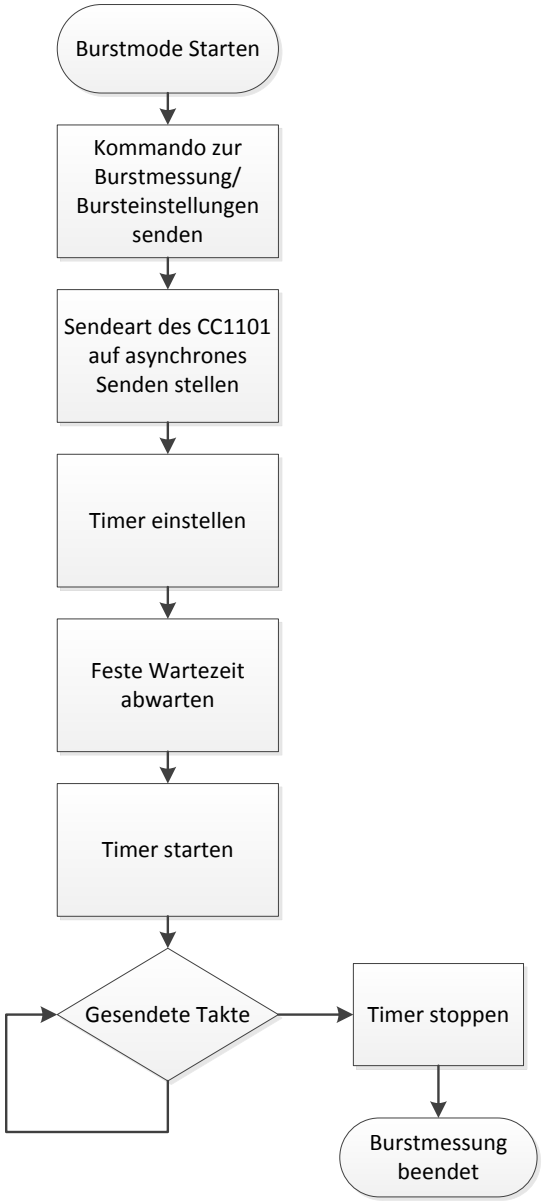


Abbildung 4.21.: UML der Burstmessung / Batteriesteuergerät

4.4.3. Softwarerealisierung des Zellensensors

Die Zellensensorsteuerung läuft während der Burstmessung rein interruptgesteuert. Zuvor müssen aber noch einige Einstellungen durch die Software gemacht werden, um den Sensor auf die Messung vorzubereiten. Sobald der Zellensensor den Befehl zur Burstmessung erhalten hat, ist es notwendig, zunächst den DCO-Takt des Zellensenors von 1 MHz auf

16 MHz zu stellen. Dieser Befehl zur Burstmessung wird noch packetorientiert gesendet. Durch die Umstellung des DCO-Takts, ist es notwendig die Haltezeit des A/D-Wandler neu einzustellen. Bedingt durch die Eingangsimpedanz am A/D-Wandler und des externen anliegenden Vorwiderstands, beträgt die Aufnahmezeit des Wandler mindestens 37.56 μs. Diese Zeit berechnet sich nach der folgenden Formel[13].

$$t_{sample} > (R_S + R_I) \cdot In\left(2^{13}\right) \cdot C_I + 800\,ns \tag{4.2}$$

$$37.56\,\mu s > (100\,k + 2\,k) \cdot In\left(2^{13}\right) \cdot 40\,pF + 800\,ns$$

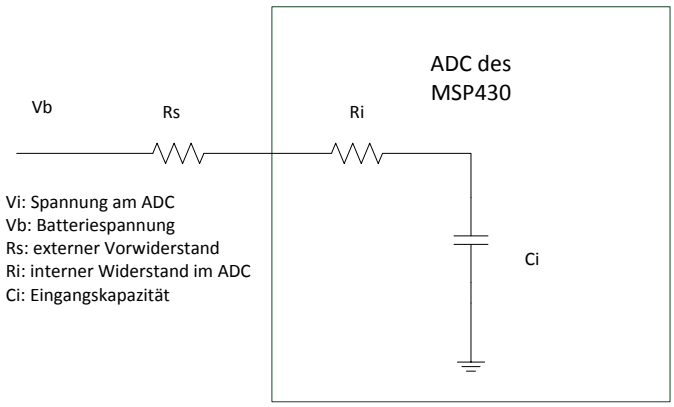


Abbildung 4.22.: Beschaltung des A/D-Wandler: modifiziert entnommen aus [13]

Es wird daher die nächst höhere einstellbare Aufnahmezeit von 64 μs festgelegt.

Der Transceiver muss nun auch seinen Empfangsmodus auf den sogenannten “asynchronen Modus” umstellen. Da die Umstellung des Empfangsmodus im CC1101 etwas Zeit benötigt, ist noch eine Wartezeit von 2 ms nötig. Danach kann der Interrupt am Eingangsport des Mikrocontrollers für den Empfang des Taktsignals freigegeben werden.

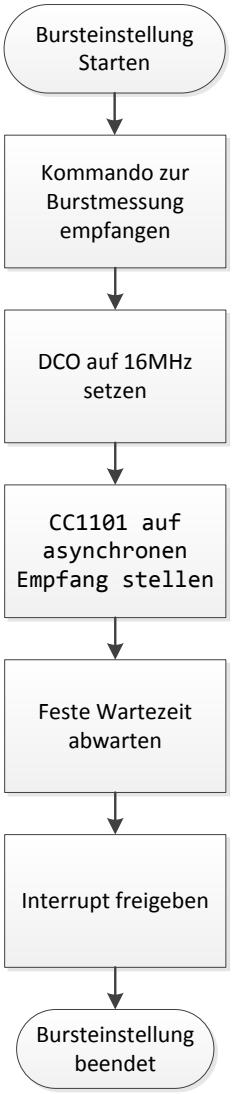


Abbildung 4.23.: UML der Burstmessungskonfiguration / Zellensensor

Es wird nun auf den Empfang des ersten Taktsignals gewartet. Sobald dieses empfangen wurde, beginnt die Steuerung mit den Timern. Dies ist in Abbildung 4.24 zu sehen. Die Timer steuern nun das Freigeben und die Sperrung des Interrupteingangs Port 1. Durch diese Steuerung wird die Fensterung realisiert. In der Abbildung 4.24 bezeichnet die Zeit TW1 (Time Window 1) die Zeit, die von der Sperrung des Port 1 nach einer Spannungsaufnahme des A/D-Wandler, bis zur seiner wieder Freigabe vergehen muss. Während dieser Zeit auftretende Interrupts werden nicht berücksichtigt, da davon ausgegangen wird, dass diese von Störungen ausgelöst werden. Die Zeit TW2, ist die Zeit in der ein Interrupt auftreten darf. Wird in dieser Zeit jedoch keiner ausgelöst, wird der Port spätestens nach Ablauf der Zeit

TW2 wieder gesperrt. Die Zeit TW2 setzt sich zusammen aus dem doppelten Toleranzbereich der Fensterung, hier also 2 x 200 μs. Wird innerhalb dieser Zeit, ein Interrupt ausgelöst, wird dies als Taktsignal gewertet und eine ADC Messung ausgelöst. Dadurch wird der Timer neu gestartet und die Zeit von TW1 beginnt von vorne. Dies wird solange wiederholt, bis die erforderliche Anzahl der Taktperioden verstrichen ist.

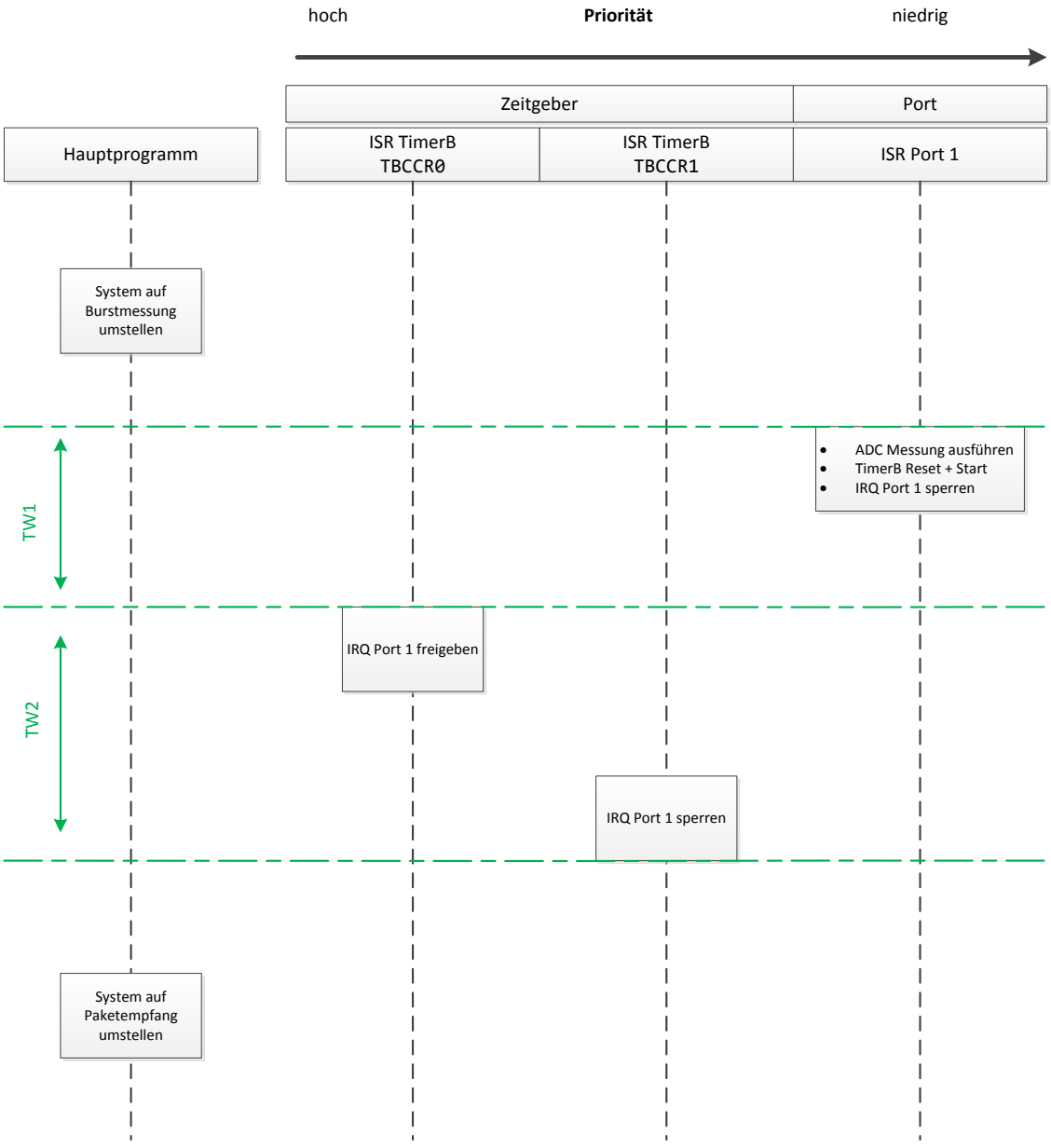


Abbildung 4.24.: IRQ der Burstmessung / Zellensensor

Erkennung von Burstfehlern

Die Erkennung von Burstfehlern wird ebenfalls durch den steuerenden Timer gelöst. Wird in der Zeit TW2, wie in Abbildung 4.24 dargestellt, kein Interrupt ausgelöst, wird das als Fehler registriert. Da innerhalb des Zeitfensters von TW2 ein Taktsignal die ADC Messung auslösen soll, ist ein Fehler in der Taktübertragung vorhanden. Als aufgenommener Spannungswert wird dann der Heximalwert 0xFFFF anstelle des ADC Werts abgespeichert. Sind noch nicht alle Messwerte aufgenommen, wird der Interrupt an Port 1 wieder gesperrt und das Zeitfenster von TW1 beginnt wieder von vorne. Zu beachten ist dabei, dass die Zeit TW1 nun um die Toleranzbreite der Fensterung kürzer sein muss, da genau diese Zeit bei der Fehlmessung länger gewartet wurde.

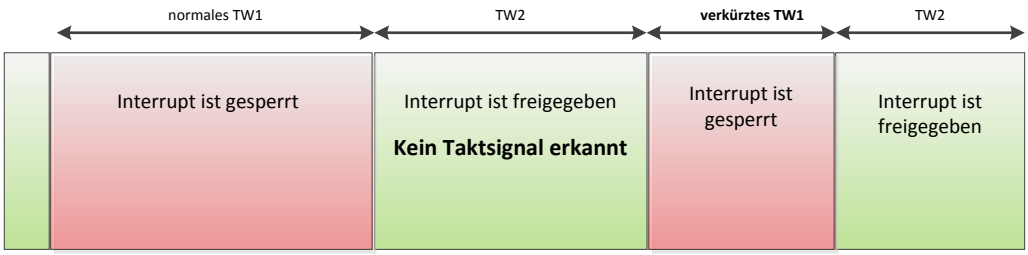


Abbildung 4.25.: Erkennung von Burstfehlern

Erfassung von DCO-Fehlern

Da aus Platz-, und vor allem Kostengründen kein Quarz als Taktgeber verbaut wurde, musste hier auf den ungenauen internen DCO zurückgegriffen werden. Da der DCO des Zellensensors bei 16 MHz Taktrate bis zu 6 % abweichen kann [12] ist es sinnvoll, eine Möglichkeit zu haben, die Abweichung zu erkennen und gegebenenfalls zu korrigieren. Besonders bei zeitkritischen Vorgängen, wie hier der Burstmessung, ist man auf einen genauen Takt des Mikrocontrollers angewiesen. Deswegen muss es eine Möglichkeit geben, einen Fehler des internen DCO zu erfassen. Um eine Abweichung zu registrieren ist es nötig, einen Referenztakt zu haben. Die Burstmessung kann sich dabei als Referenz anbieten. Da durch das Batteriesteuergerät bekannt ist, um welche Frequenz es sich bei der Burstmessung handelt, kann man das ankommende Taktsignal als Referenz für die Fehlererkennung nutzen. Um nun die DCO-Abweichung zu bestimmen, ist die Zeitdifferenz zwischen zwei Pegelflanken im Taktsignal mit Hilfe eines Timers zu messen. Um dazu nicht einen eigenen Timer zu verwenden, lässt sich die Zeit aus dem Timer für die Fensterung entnehmen. Dazu wird der aktuelle Timerwert vor dem Start des Timer ausgelesen. Dieser Wert entspricht der Anzahl der Takte[3], die die DCO innerhalb des letzten Periodendurchlaufs erreicht hat. Verglichen mit

[3] Bei einem Vorteiler des Timers von 1

dem Soll-Taktwert, lässt sich somit eine Abweichung der DCO errechnen. Eine Auswertung der Abweichung wurde innerhalb dieser Arbeit nicht implementiert, da dies nur ein anzuwendender Nebeneffekt der Burstmessung ist.

Detektierung des Ende der Burstmessung

Der Zellensensor soll die Beendigung der Burstmessung selbständig erkennen. Dazu zählt der Zellensensor die Anzahl der Messungen und die Anzahl der nicht erhaltenen Taktsignale mit. In dem Kommando, welches das Batteriesteuergerät an den Zellensensor gesendet hat, befindet sich auch die Anzahl der aufzunehmenden Messungen. Stimmt die Anzahl mit den mitgezählten Takten überein, beendet der Zellensensor die Messung. So ist sichergestellt, dass der Zellensensor wieder in den paketorientierten Empfangsmodus übergeht, auch wenn er kein Taktsignal mehr erhält. Die maximale Empfangszeit berechnet sich dann wie folgt.

$$\text{Anzahl Messungen} \cdot \text{Periodendauer der Burstfrequenz [s]} = \text{max. Empfangszeit [s]} \quad (4.3)$$

Bei einer 1 kHz Burstmessung und 500 aufzunehmenden Messwerte ergibt sich eine Gesamtaufnahmezeit von

$$500 \cdot 1\ ms = 500\ ms$$

4.4.4. Zwischenspeicherung und Übertragung

Durch die Burstmessung fallen eine große Anzahl von Spannungsdaten an, die verarbeitet werden müssen. Da die aufgenommenen Daten nicht sofort an das Batteriesteuergerät gesendet werden können, müssen diese auf dem Sensor zwischengespeichert werden. Diese werden direkt im RAM des Mikrocontrollers abgespeichert. Die Größe des Speichers begrenzt dabei die Anzahl der aufzunehmenden Spannungswerte. Der interne RAM des eingesetzten Mikrocontroller besitzt eine Größe von 2 kByte. Theoretisch ist damit eine Zwischenspeicherung von 1.024, jeweils 2 Byte großen Spannungswerten möglich. Dies wird aber im normalen Betrieb nicht erreicht, da das laufende Programm ebenfalls einen gewissen Teil des Speichers beansprucht. Um alle Daten an das Batteriesteuergerät senden zu können, muss dies in mehreren Paketen geschehen. Der Zellensensor berechnet anhand der Menge der aufgenommenen Werte die Anzahl der zu sendenden Pakete. Da ein aufgenommener Wert in 2 Byte abgespeichert wird, muss die Anzahl mit 2 multipliziert werden.

$$\frac{\text{Anzahl der Werte} \cdot 2\ \text{Byte}}{\text{Byte pro Paket}} = \text{Anzahl der Pakete} \tag{4.4}$$

Bei einer aufgenommenen Anzahl von 1.000 Spannungswerten und der maximalen Anzahl von 50 Datenbyte pro Paket, ergeben sich 40 zu sendende Pakete. Diese Anzahl wird dem Batteriesteuergerät mitgeteilt. Dies zeigt Abbildung 4.26 des zeitlichen Ablaufs der Datenanfrage, dabei wird dem Batteriesteuergerät die Anzahl der zu erwartenden Pakete “BURST_ FRAME_ COUNTER” und die Paketlänge “BURST_ FRAME_ LENGTH” mitgeteilt.

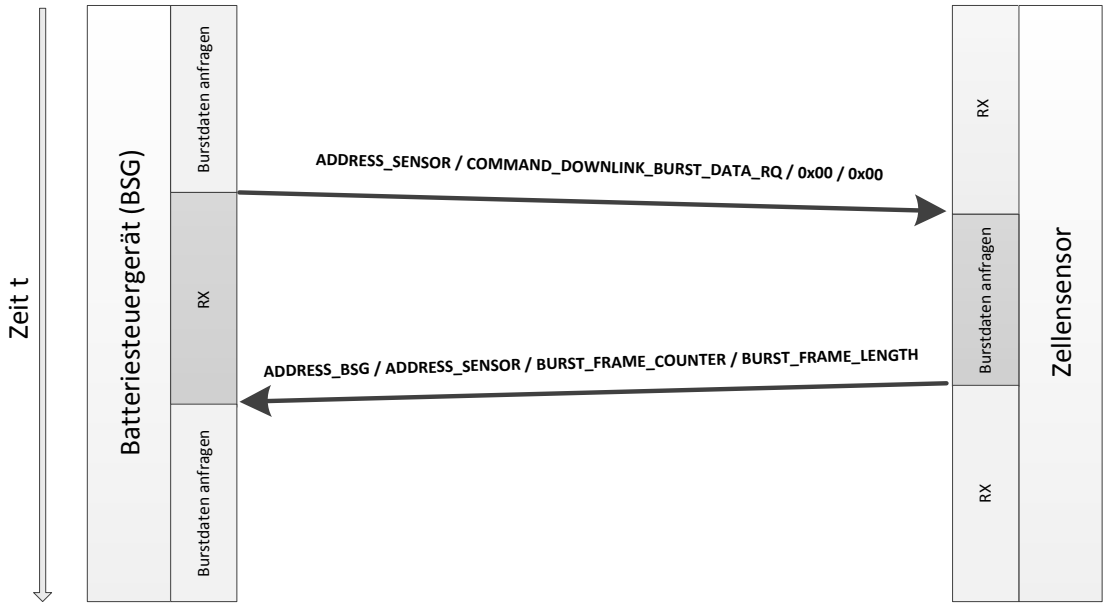


Abbildung 4.26.: Anfrage Burstmessung

Das Batteriesteuergerät stellt sich nun auf die entsprechende Paketlänge ein. Anschließend kann die Übertragung der Daten beginnen. Abbildung 4.28 zeigt den zeitlichen Ablauf des Sendens der Burstdaten vom Zellensenor an das Batteriesteuergerät. Das Steuergerät schickt dazu einen Befehl, der die Adresse des Zellensensors, das Kommando zum Senden der Daten und die gewünschte Framenummer enthält. Durch die Angabe der Framenummer kann auch nur ein bestimmter Teil der Burstdaten auf dem Zellensensor abgerufen werden. Dies zeigt Abbildung 4.27.

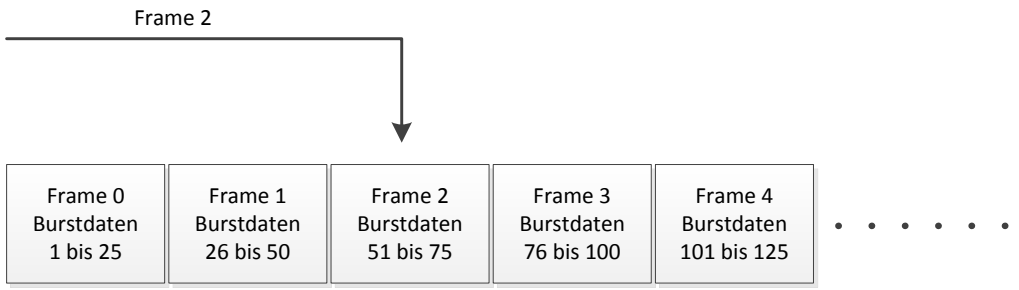


Abbildung 4.27.: Abruf eines Frames

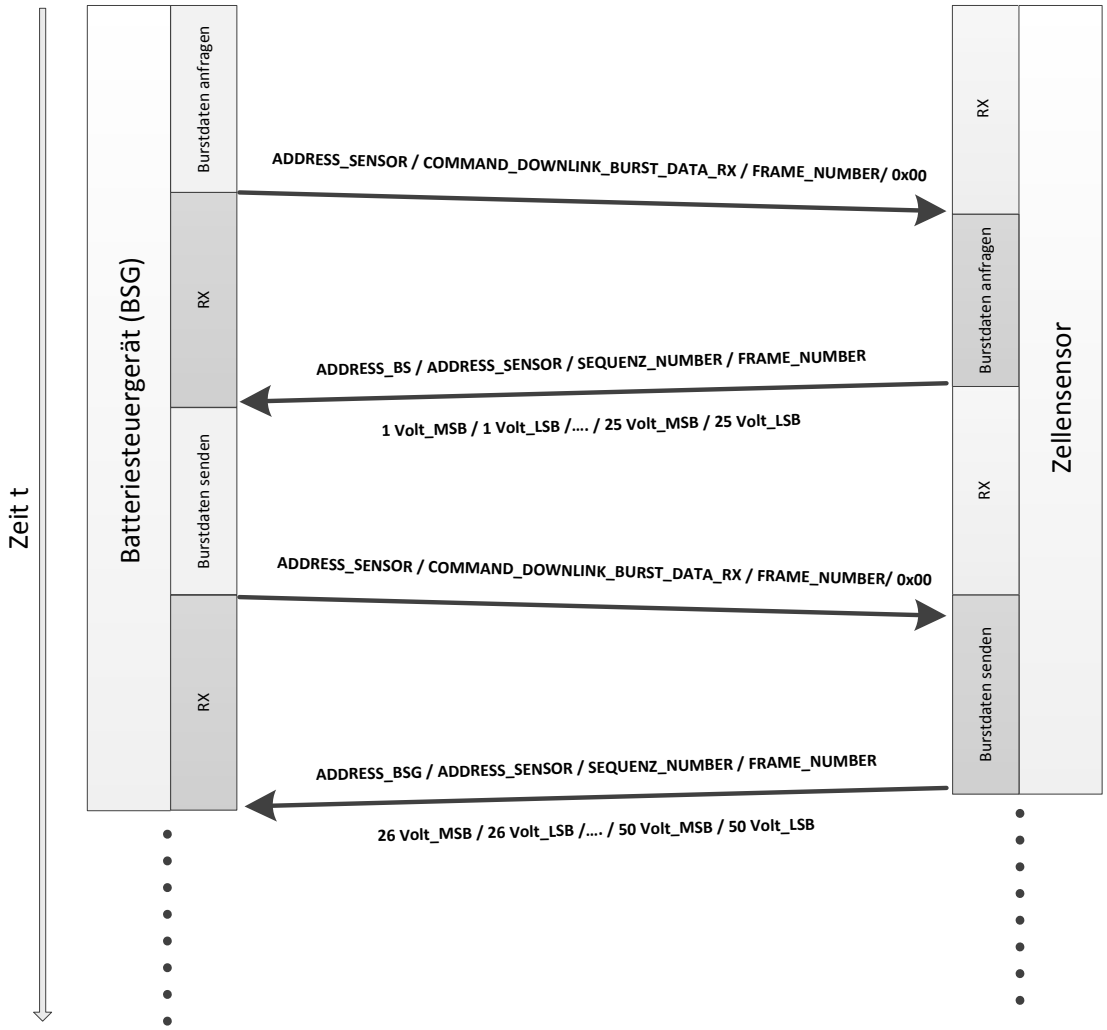


Abbildung 4.28.: Senden der Burstdaten

Die Spannungsdaten sind solange verfügbar, bis eine weitere Burstmessung durchgeführt wird und die Daten dadurch überschrieben werden.

4.5. Redesign der Zellensensorplatine

Im Redesign soll die Zellensensorplatine V0.1 nach [4] überarbeitet werden. Dabei sollen an der Schaltungstechnik keine großen Änderungen vorgenommen werden. Lediglich sollen offen gebliebene Punkte eingearbeitet werden. Eine Änderung betrifft die Abschaltung des DC/DC-Wandlers. Es soll möglich sein, diesen während einer Spannungsmessung abschalten zu können. Dazu benötigt der Wandler eine Verbindung zum steuernden Mikrocontroller, welcher den DC/DC-Wandler dann an- und abschalten kann. Dies wird in Abschnitt 4.5.1 genauer erläutert.
Des Weiteren soll der Balancierungspfad in zwei Teile aufgeteilt und räumlich voneinander getrennt werden um die thermische Belastung für die Platine zu minimieren.
Überarbeitet werden soll auch das Design der Platine um eine Montage auf einer einer 48Ah LiFePo4[5] Zelle zu ermöglichen. Die Zellensensorplatine V0.1 war dafür konzipiert um in das Zellengehäuse eingebaut zu werden. Die neue Platine des Zellensensors V0.2 soll für den Einbau in die Zelle, sowie für die Montage auf den Zellen geeignet sein. Dies hat den Vorteil, dass es dadurch möglich wird, Zellen auch nachträglich mit dem neuen Zellensensor auszustatten.

4.5.1. Der DC/DC-Wandler TPS61201

Zur Spannungsversorgung des Zellensensors wird der DC/DC-Wandler TPS61201 von der Firma Texas Instruments verwendet. Dieser wurde bereits in mehreren vergangenen Arbeiten im Projekt BATSEN verwendet.
Dieser ist für die Spannungsversorgung der aktiven Komponenten des Zellensensors mit einer stabilisierten Betriebsspannung von 3,3 V zuständig. Der DC/DC-Wandler ist in der Lage, diese Betriebsspannung noch bei einer Eingangsspannung von 0,3V zu erreichen [9]. Durch eine externe Beschaltung des DC/DC-Wandlers ist eine untere Schwellwertspannung einstellbar, bei der der DC/DC-Wandler noch aktiv ist. In der V0.1 des Zellensensors wurde diese untere Schwellwertspannung bei 0,6V festgelegt. Um zu testen, ob diese untere Grenze noch weiter gesenkt werden kann, bei der der DC/DC-Wandler noch eine zuverlässigen Betriebspannung liefert, wurde der Widerstand R1 durch einen 0 Ω ausgetauscht. Der Widerstand R2 bleibt unbestückt. Dies ermöglicht auch die spätere Verwendung der fest einstellbaren Schwellwertspannung.
Des weiteren besitzt der TPS61201 einen ENABLE-Eingang. Durch diesen Eingang kann der TPS61201 extern aktiviert bzw. deaktiviert werden. In der V0.1 wurde dieser Eingang dauerhaft auf die Eingangsspannung gelegt. Dies bedeutet, dass der DC/DC-Wandler dauerhaft aktiviert ist, solange eine Eingangsspannung anliegt. Im Redesign der Zellensensorplatine wird dieser ENABLE-Eingang nun mit dem Mikrocontroller verbunden. Dadurch ist es möglich den DC/DC-Wandler durch den Mikrocontroller zu steuern. Abbildung 4.29 zeigt

die abgeänderte Beschaltung des DC/DC-Wandlers TPS61201. Am Anschluss 6 des ICs ist dabei die ENABLE-Leitung zu sehen. Zwischen Anschluss 5 und 7 befindet sich der Spannungsteiler, der zur Einstellung der unteren Schwellwertspannung dient.

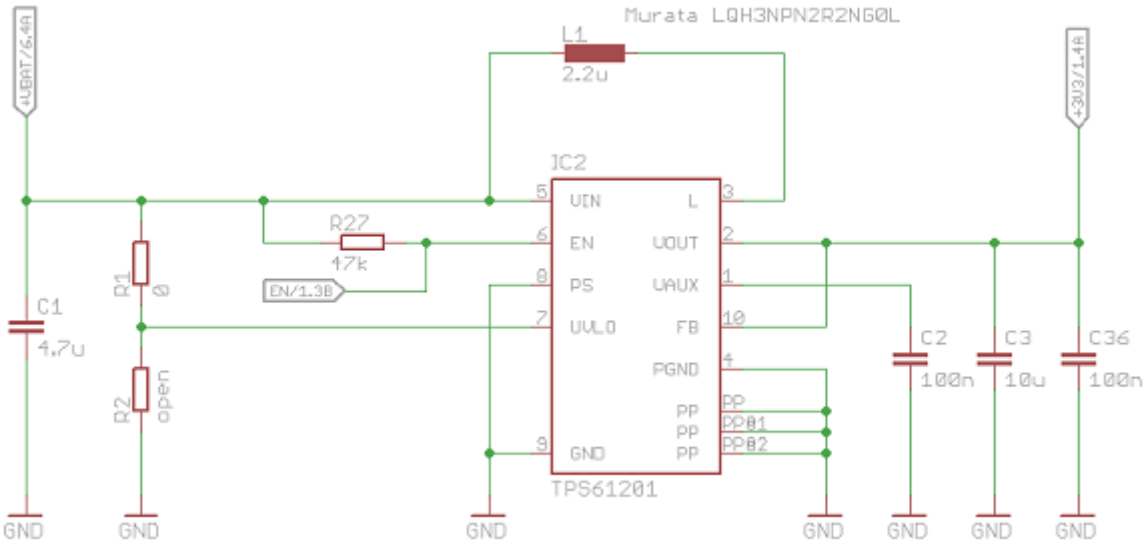


Abbildung 4.29.: Beschaltung des TPS61201

4.5.2. Die Balancierung

Die Balancierungsschaltung besteht auf dem Zellensensor V0.1 aus einem Balancierungspfad, der zwei parallelgeschaltete Pfade, mit jeweils zwei in Reihe geschaltete 10 Ω Widerstände besitzt. Somit werden insgesamt vier Widerständen eingesetzt, die räumlich dicht platziert sind. Dies kann bei der Balancierung zu einer starken thermischen Belastung für die Platine und der Zelle führen. Um diese Belastung zu reduzieren, werden die Balancierungswiderstände räumlich voneinander getrennt. Des Weiteren sollen sie in zwei unterschiedliche Balancierungspfade unterteilt werden, die vom Mikrocontroller einzeln angesprochen werden können. Auf der Zellensensorplatine V0.1 konnte der Balancierungspfad nur als gesamtes geschaltet werden. Eine Unterteilung der Nebenstrompfade ermöglicht es, mit zwei unterschiedlich starken Lasten die Ladungsbalancierung durchzuführen. Die Beschaltung der passiven Ladungsbalancierung zeigt Abbildung 4.30.

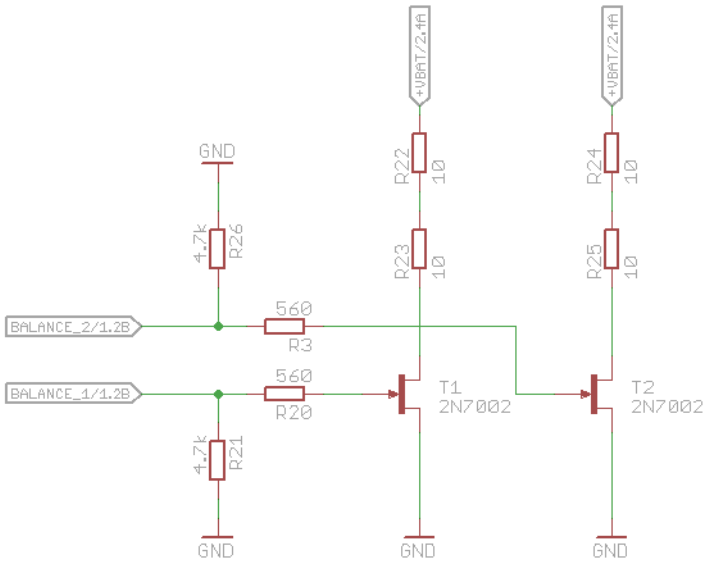


Abbildung 4.30.: Schaltung der passiven Ladungsbalancierung

4.5.3. Programmierinterface

Eine zusätzliche Neuerung, soll ein JTAG-Programmieranschluss über eine Steckverbindung sein. Bisher wurde der Mikrocontroller MSP430F235 über einen Nadeladapter programmiert. Diese Art der Programmierung ist vorteilhaft, da auf der Platine lediglich Kontaktflächen für das aufsetzten der Nadeln vorhanden sein muss. Dadurch lässt sich Platz auf der Platine und vor allem Kosten für eine fest aufgelötete Steckverbindung sparen. Allerdings hat sich gezeigt, dass durch häufiges Wechseln der Zellensensoren auf den Nadeladaptern die Kupferflächen für die Nadeln abnutzen und dadurch Kontaktprobleme entstehen können. Da bei der Entwicklung der Zellensensoren ein häufiges Programmieren der Hardware notwendig ist, wurde entschieden einen Abschluss für eine Steckverbindung vorzusehen. Dieser Steckverbinder soll auf der Rückseite der Platine montierbar sein und sich bei Bedarf auch wieder lösen lassen. Des Weiteren kann über diese Verbindung der Zellensensor mit Spannung versorgt werden. Dies dient aber lediglich Testzwecken.

4.5.4. Montage der Zellensensoren

Es soll möglich sein die Zellensensoren auf das Gehäuse einer 48Ah LiFePo4 Zelle der Firma ECC Repenning GmbH[5] zu montieren. Diese Zellen besitzen an der Oberseite ein M8 Gewinde zum Anschluss und zur Befestigung der Zelle. Dieses Gewinde kann genutzt werden um den Zellensensor an der Zelle zu befestigen. Abbildung 4.31 zeigt die Abmessungen des Zellendeckels.

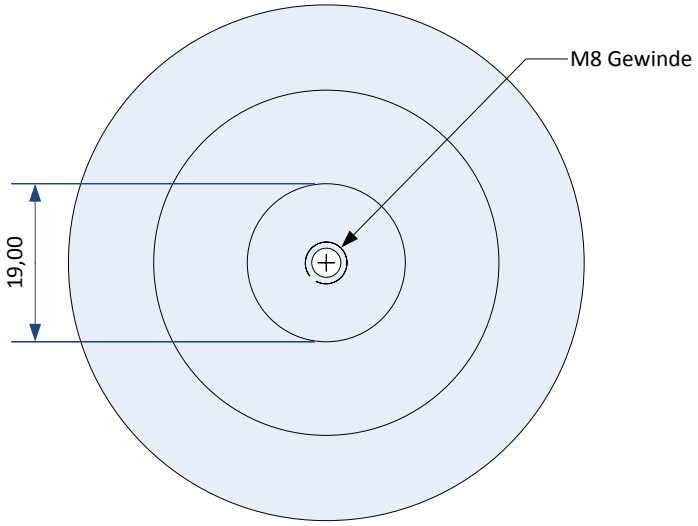


Abbildung 4.31.: Draufsicht des Zellendeckels

Der 19mm breite Kontakt in Abbildung 4.32 ist der positive Anschluss der Zelle. Dieser dient dann gleichzeitig zur Spannungsversorgung des Zellensensors. Es muss dann lediglich ein Kontakt zum negativen Anschluss der Zelle hergestellt werden. Als Kontaktfläche soll hierbei eine Kupferfläche um das Montageloch auf der Ober- und auch der Unterseite der Platine dienen. Dabei wird durch die Verschraubung der Platine an den Zellenpol der Kontakt geschlossen.
Es soll die Möglichkeit bestehen, den Sensor an der positiven Seite, als auch an der negativen Seite der Zelle anzubringen. Diese Möglichkeit erfordert eine flexible Einstellung der Polarität der Kupferflächen. Diese wird über zwei Lötbrücken realisiert, die je nach Montage der Sensoren an die Zelle geschlossen, bzw. geöffnet werden können.

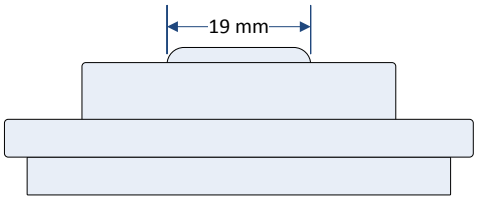


Abbildung 4.32.: Seitenansicht des Zellendeckels

Um die Montage realisieren zu können, ist ein Montageloch von 8,1 mm Durchmesser in der Platine eingeplant. Dies zeigt Abbildung 4.33.

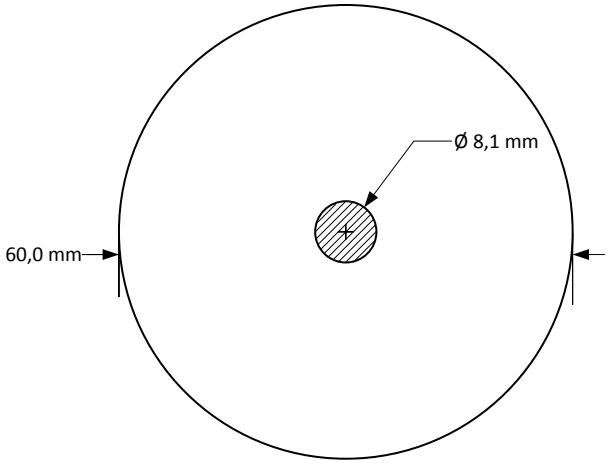


Abbildung 4.33.: Abmessungen der neuen Sensorplatine

5. Praktische Untersuchungen und Erprobung

5.1. Aufbau der Sensorplatine

In diesem Kapitel soll der neue Zellensensor V0.2 zum ersten mal erprobt werden. Des weiteren wird die allgemeine Funktion des Sensors getestet werden. Besonders die Funktion der Burstmessung soll dabei untersucht, und mit den in den Anforderung gestellten Anforderungen verglichen werden.

5.1.1. Bestückter Zellensensor

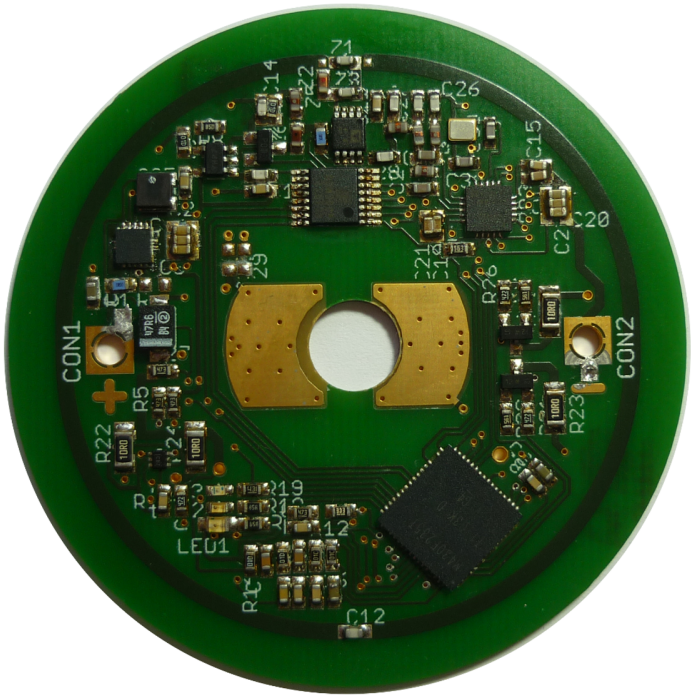


Abbildung 5.1.: Bestückte Zellensensorplatine / Vorderseite

Die Abbildung 5.1 zeigt den bestückten Zellensensor V0.2. Mittig ist die 8,1 mm große Bohrung für die Montage auf der LiFePo4 Rundzelle zu sehen. Daneben sind die zwei großen Kontaktflächen zur Verbindung mit dem jeweiligen Potenzial an der Montagefläche. Links und rechts Außen, mit CON1 und CON2 bezeichnet, befinden sich ebenfalls Anschlüsse für die Zellenspannung.

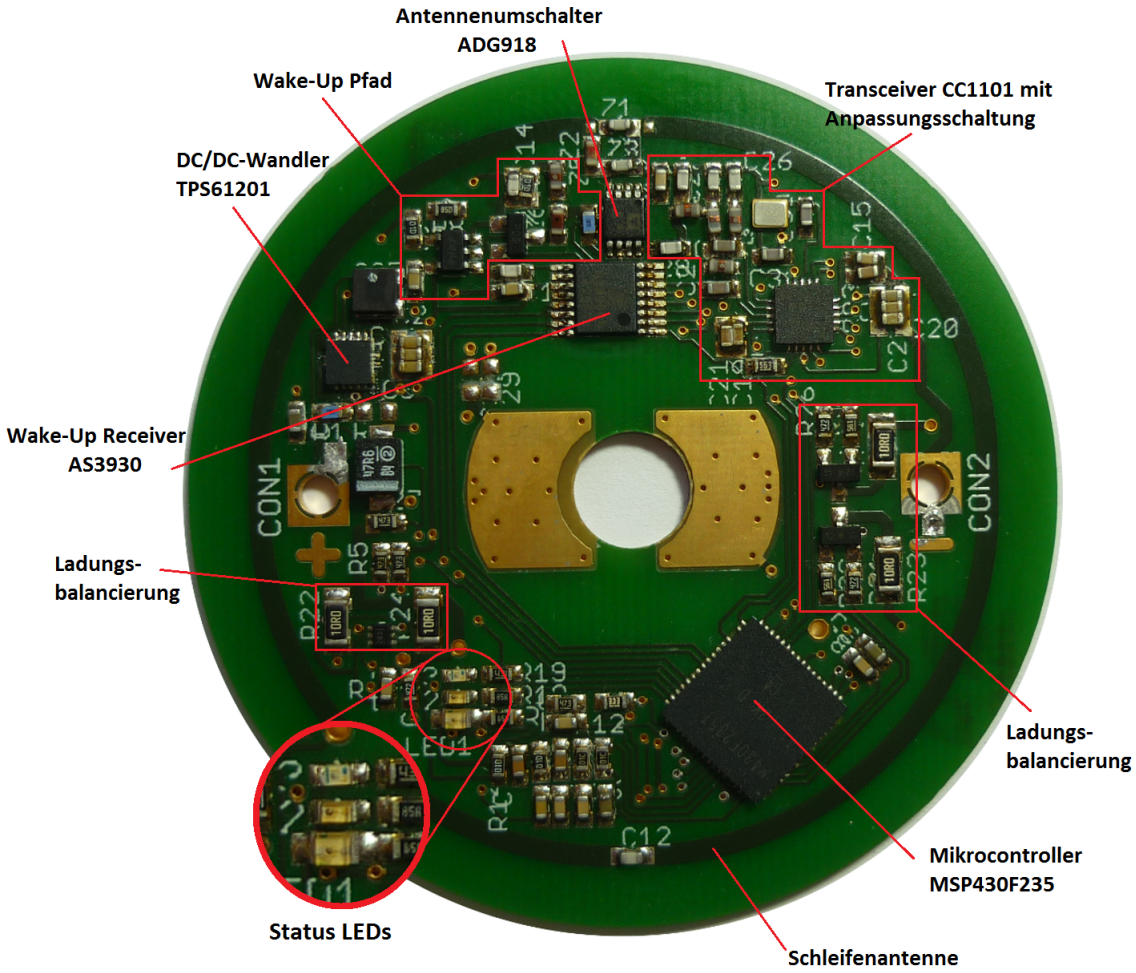


Abbildung 5.2.: Bestückte Zellensensorplatine / Vorderseite 2

Abbildung 5.2 zeigt den Zellensensor mit der Platzierung der einzelnen Bauelemente. Im unteren vergrößerten Ausschnitt sind drei Status LEDs zu sehen. Die untere rote LED leuchtet, wenn der Zellensensor sich im aufgeweckten Zustand befindet. Die mittlere gelbe LED zeigt an, wenn Daten gesendet werden. Die obere grüne LED leuchtet, wenn Daten empfangen werden.

In der oberen Vergrößerung der Abbildung 5.3 sind die Lötbrücken für die Einstellung des jeweiligen Potenzials auf der montierten Kontaktfläche der LiFePo4 Zelle zu erkennen. Die linke Lötbrücke ist bei einer Montage auf der Masseseite der Zelle zu schließen. Die rechte Lötbrücke ist für die Montage auf der positiven Seite der Zelle zu schließen.

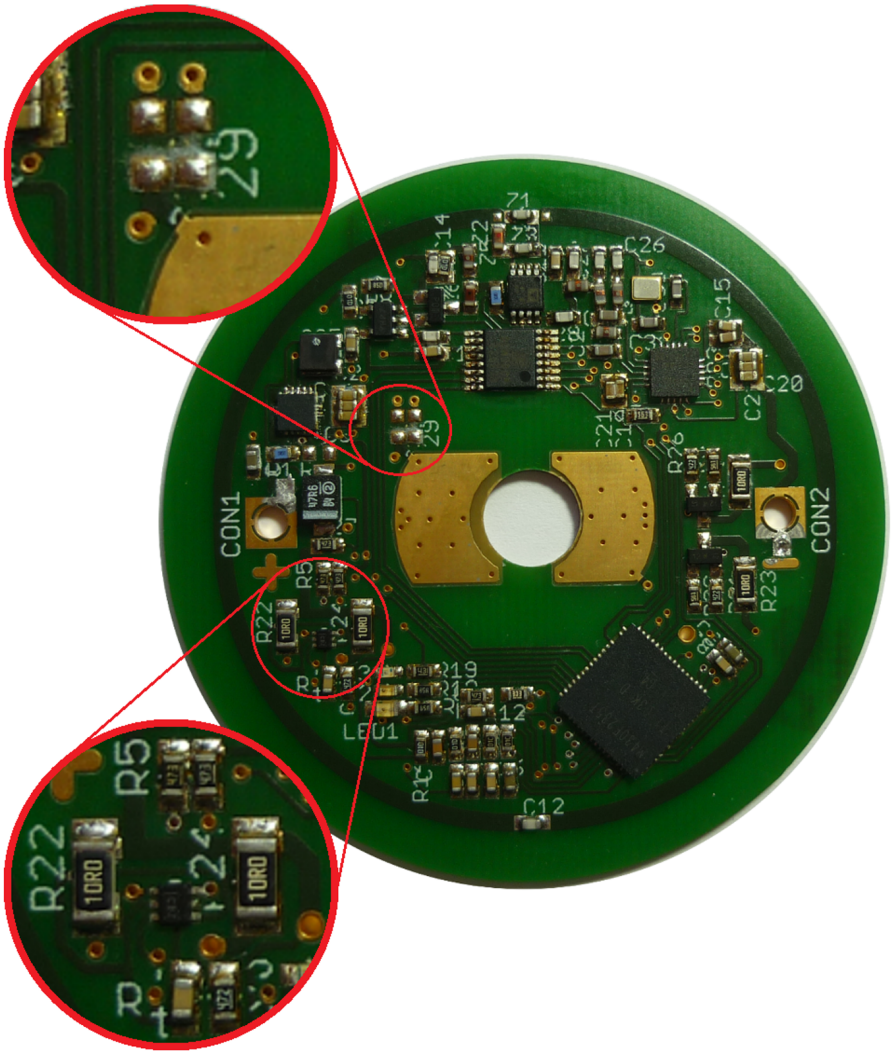


Abbildung 5.3.: Bestückte Zellensensorplatine / Vorderseite 3

Die untere Vergrößerung zeigt den externen Temperatursensor TMP102 inmitten zwei Balancierungswiderständen.

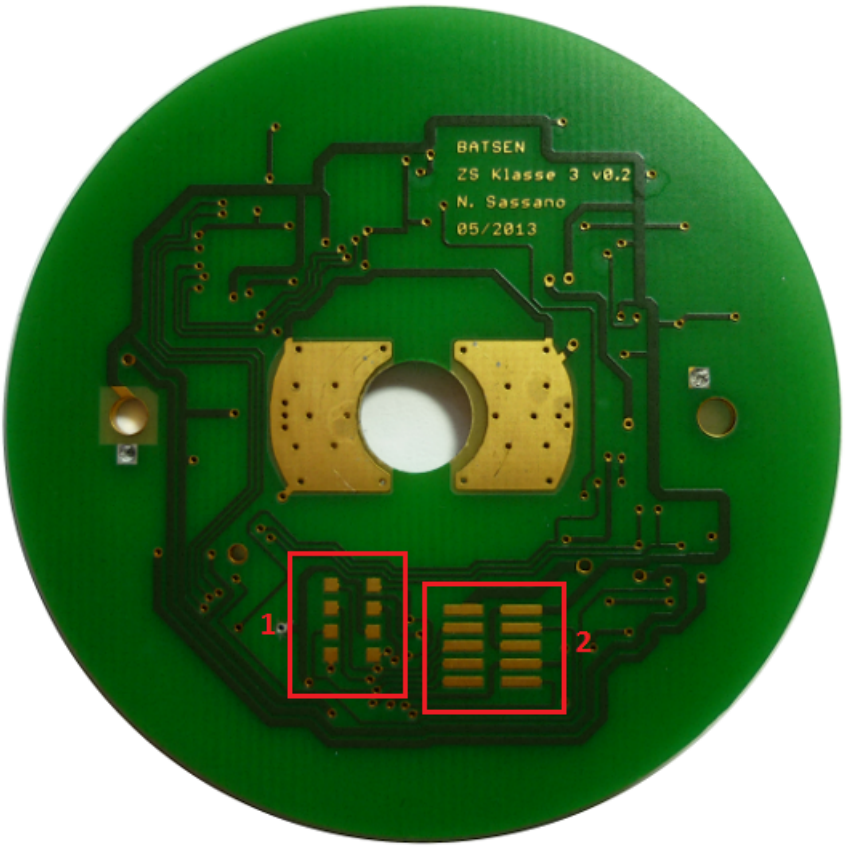


Abbildung 5.4.: Bestückte Zellensensorplatine / Rückseite

Abbildung 5.4 zeigt die Zellensensorplatine von der Rückseite. Die mit 1 markierten Anschlüsse sind die Kontaktpins für das Programmieren über den Nadeladapter. Die mit 2 markierten Anschlüsse sind die Kontakte für das Programmieren über den Steckverbinder.

5.1.2. Montage auf den Zellen

Abbildung 5.5 zeigt einen montierten Zellensensor auf der positiven Seite einer LiFePo4 Zelle der Firma EEC Repenning GmbH [5]. Der Zellensensor muss dabei noch mit Hilfe eines Kabels mit der negativen Seite verbunden werden. Eine Montage auf der negativen Seite der Zelle ist ebenfalls möglich. Bei der Montage ist darauf zu achten, dass die richtigen Lötbrücken geschlossen/geöffnet sind. Siehe dazu Seite 72.

Abbildung 5.5.: Montierter Zellensensor

5.1.3. Messplatine

Zur Anpassung der Antenne sowie des Wake-Up-Pfads wurde eine Messplatine angefertigt. Zwar wurde die Schleifenantenne sowie der Wake-Up-Pfad ohne Veränderungen übernommen, dennoch wird vermutet, dass sich durch die geometrische Veränderung der Bauteile die Resonanz des jeweiligen Pfades verstimmt. Dadurch ist eine neue Anpassung des Impedanznetzwerks notwendig. Um diese Resonanz messen zu können, wurde eine Messplatine, mit denselben Bauteilanordnungen wie der Zellensensor V0.2 gefertigt. Diese Messplatine unterscheidet sich lediglich durch eine Auftrennung des Antennenpfads zwischen dem Impedanzanpassnetzwerk der Schleifenantenne und dem Antennenumschalter ADG918BRM [3]. Anstelle des Antennenumschalters wurde eine SMA-Buchse[1] mit dem Impedanzanpassnetzwerk verbunden. Somit lässt sich die Resonanzanpassung mit Hilfe eines Netzwerkanalysators bestimmen.
Ebenfalls wurde eine SMA-Buchse an den Eingang des Wake-Up-Pfades verbunden. Ein Schaltplan mit den Veränderungen zum normalen Zellensensor befindet sich im Anhang ab Seite 131.

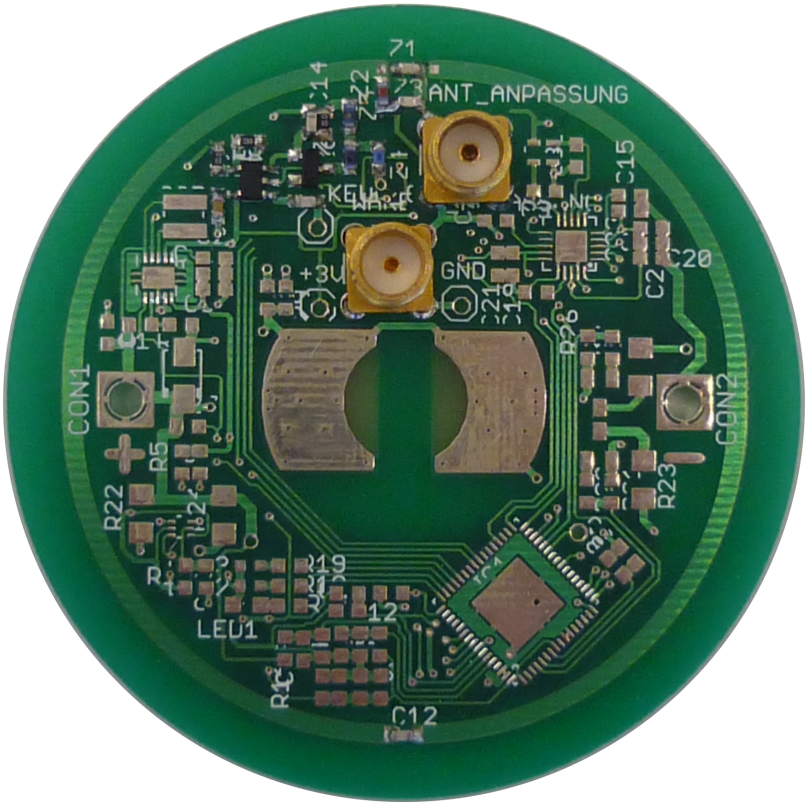


Abbildung 5.6.: Bestückte Messplatine

[1] Sub-Miniature-A Buchse

5.2. Inbetriebnahme des Zellensensors

Bei der Inbetriebnahme der neuen Zellensensoren musste festgestellt werden, dass sich die Impedanzanpassung der Schleifenantenne verschoben haben muss. Im Vergleich zu den Zellensensoren V0.1 war ein wesentlich schlechterer Empfang erkennbar. Es konnten lediglich Strecken von wenigen Zentimetern überbrückt werden. Mit den Zellensensoren V0.1 wurden hingegen mehrere Meter überbrückt.

5.2.1. Impedanzanpassung der Schleifenantenne

Mit Hilfe der angefertigten Messplatine war es möglich, den Eingangsreflexionsfaktor der Schleifenantenne zu messen. Mit Hilfe der Hochfrequenzsimulationssoftware Microwave Office der Firma AWR wurde versucht die optimalen Bauteilwerte für die Impedanzanpassung zu finden. Die beiden nächsten Abbildungen zeigen die Ergebnisse dieser Simulation.

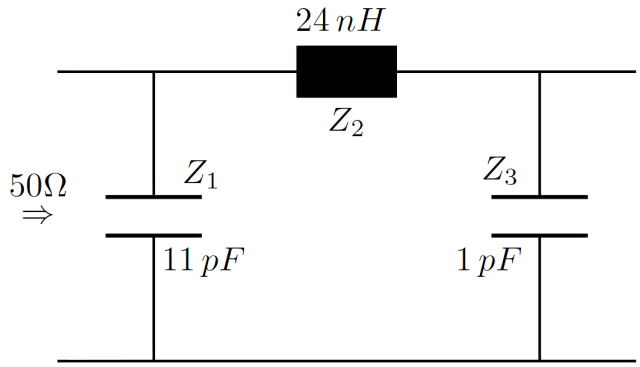


Abbildung 5.7.: Impedanzanpassung

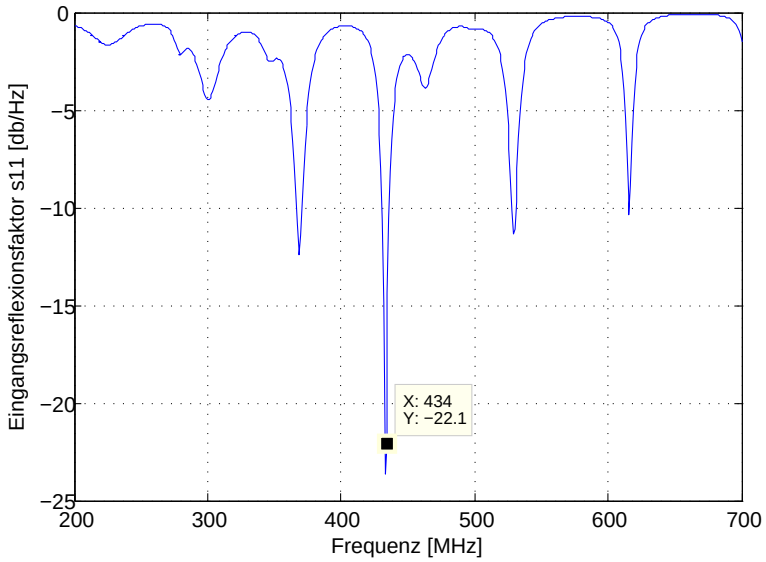


Abbildung 5.8.: Angepasster Antennenpfad in AWR

Die Simulation ergab einen niedrigen Eingangsreflexionsfaktor von -22,1 dB bei der Frequenz von 434 MHz.
Bei der Erprobung dieser Werte zeigte sich aber, dass diese Werte nicht zu der gewünschten Anpassung führte. Bei manchen Sensoren war keinerlei Empfang mit der Schleifenantenne vorhanden. Aus diesem Grund wurde entschieden die Bauteilwerte aus [4] zu übernehmen und diese Werte leicht zu variieren. Dadurch wurde versucht eine Verbesserung des Eingangsreflexionsfaktor zu erreichen.
Die besten Ergebnisse erzielt man mit einer Lötbrücke am Kondensator C12. Siehe dazu Abbildung 5.10.

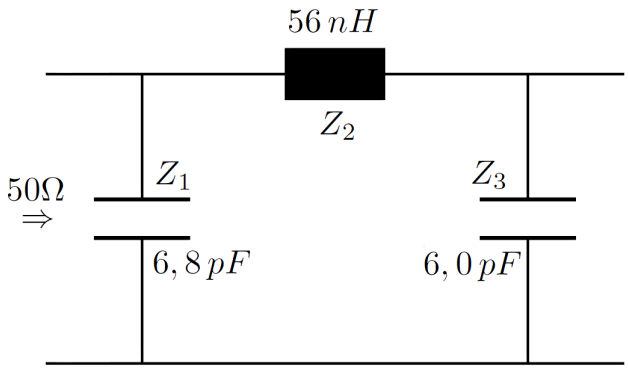


Abbildung 5.9.: Impedanzanpassung 2

Die Bauteile in der oberen Vergrößerung zeigen die Impedanzen Z_1, Z_2 und Z_3.

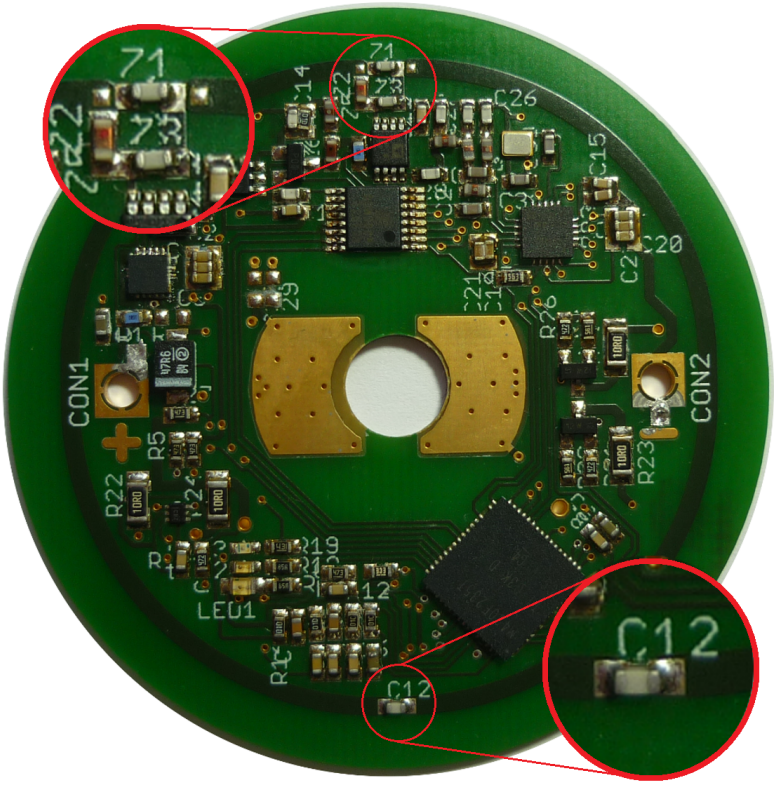


Abbildung 5.10.: Impedanzanpassung 3

In der unteren Vergrößerung ist der Kondensator C_{12} zu sehen. Dieser hat einen Wert von 1 pF und wurde durch eine Lötbrücke ersetzt.

Leider konnte mit diesen Werten der Eingangsreflexionsfaktor nicht mehr gemessen werden, da der Zugang zu den nötigen Laboren zu dieser Zeit wegen Bauarbeiten nicht mehr möglich war. Da diese Werte aber gute Ergebnisse lieferten, wurden die Zellensensoren mit diesen Betrieben.

Anpassung des Wake-Up-Pfads

Die Werte für das Anpassungsnetzwerk des Wake-Up-Pfades wurden aus [4] übernommen. Diese lieferten gute Ergebnisse und konnten somit ohne weitere Veränderungen übernommen und eingesetzt werden.

5.2.2. Allgemeiner Funktionstest

Nach dem Aufbau der Zellensensorplatine und deren Programmierung erfolgte ein allgemeiner Funktionstest des Sensors. Dabei wurden die folgenden Funktionen getestet.

Allgemeiner Funktionstest

- Wecken durch Wake-Up-Signal
- Konfiguration durch das Batteriesteuergerät
- Abruf einer einzelnen Spannung
- Abruf eines Spannungs- und Temperaturwerts (TMP102)
- Abruf der Temperatur (TMP102)
- Abruf der Temperatur (MSP430)
- Burstmessung
- Burstdaten abrufen

Alle diese Funktionen konnten dabei erfolgreich getestet werden.

5.2.3. Kalibrierung des Temperatursensors TMP102

Zur ersten Überprüfung der Genauigkeit des Temperatursensors TMP102 wurde der Zellensensor in einem Temperatur-Klimaschrank betrieben. Dabei sollten verschieden Temperaturbereiche zwischen -20 °C und +60 °C abgefahren werden. Dazu wurde ein 12 Stunden langer Test gemacht, indem die Temperatur alle 2 Stunden um 20 °C ansteigen soll. Hierfür wurde am Temperatur-Klimaschrank ein Programm erstellt, welches die Temperatur eine Stunde halten soll und danach innerhalb einer Stunde die Temperatur um 20 °C erhöht. Die Haltezeit von einer Stunde soll zum Einpendeln der Temperatur des Klimaschranks und der Bauteile auf dem Zellensensor dienen. Die Temperatur des Zellensensors wird dabei jede Stunde abgerufen. Bei der Auswertung der Temperaturdaten entstand folgende Messkurve.

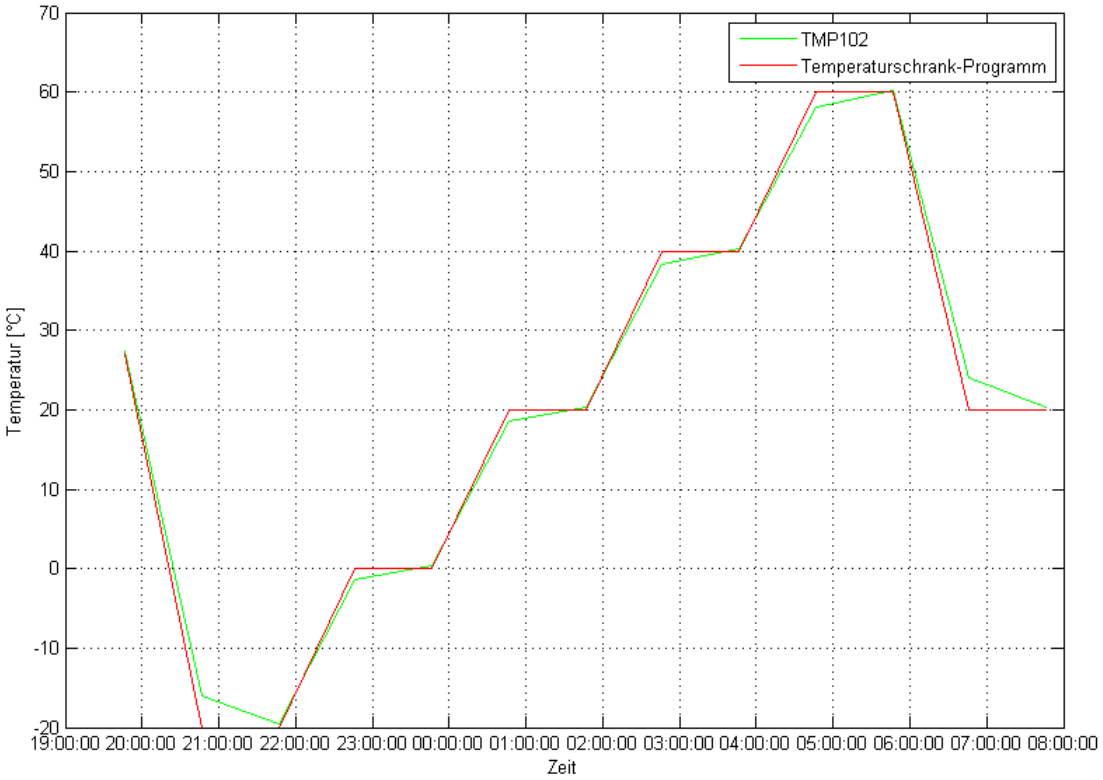


Abbildung 5.11.: Temperaturverlauf des Temperatursensors

Bei der Auswertung der Messkurve fiel auf, dass der Temperatursensor bereits eine hohe Genauigkeit hat. Der rot dargestellte Temperaturverlauf zeigt lediglich den Sollwert der Temperatur des Klimaschranks an. Nach der Einpendelzeit von einer Stunde stimmen die Temperaturen zwischen dem Sollwert des Klimaschranks und den gemessenen Werten des Temperatursensors nahezu überein. Daher ist eine Kalibrierung des Temperatursensors TMP102 nicht nötig.

5.2.4. Balancierung

Zu testen war noch die Steuerung der passiven Balancierungsschaltung. Diese soll, durch den Temperatursensor TMP102 gesteuert, die Balancierung beim erreichen einer oberen Temperaturgrenze stoppen. Danach soll sie so lange mit dem weiteren Balancieren warten, bis die Temperatur unter eine vorgegebene Grenze gefallen ist. Dies soll solang fortgeführt werden, bis die durch das Batteriesteuergerät vorgegebene Spannung erreicht worden ist, oder eine maximale Balancierungszeit erreicht wurde. Da im normalen Balancierbetrieb der Zellensensor keine Daten an das Batteriesteuergerät schickt, musste die Software für die

Messung angepasst werden. Für den Testbetrieb schickt der Zellensensor alle 500 ms einen Temperaturwert des TMP102 an das Steuergerät. Aus diesen Daten konnte dann ein Temperaturverlauf der Steuerung generiert werden.

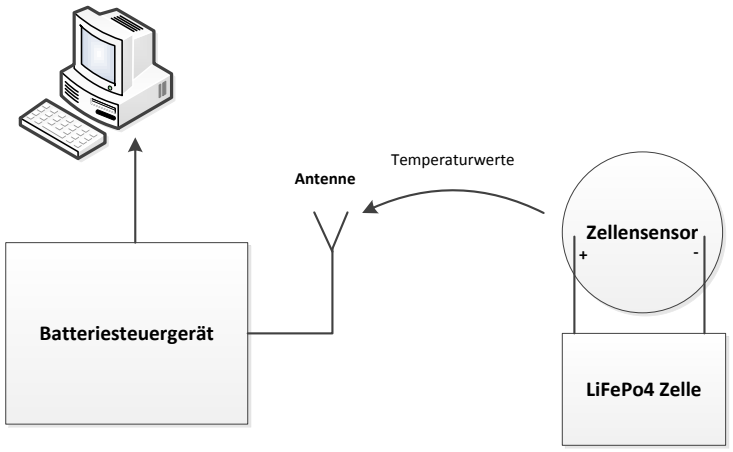


Abbildung 5.12.: Messaufbau zur Balancierung

Abbildung 5.13 zeigt die dabei aufgenommene Temperaturkurve der Steuerung. Die obere Temperaturgrenze wurde auf 50 °C und die untere Temperaturgrenze auf 34 °C eingestellt. Zu sehen ist, wie die Temperatur zu Beginn stark ansteigt, bis sie die obere Temperaturgrenze erreicht hat. Danach wird die Balancierung ausgeschaltet, bis die untere Temperaturgrenze erreicht ist. Es zeigt sich eine typische Temperaturkurve, wie sie theoretisch erwartet wurde. Nachdem die Balancierung das dritte Mal angeschaltet wurde, kurz vor der oberen Temperaturgrenze, erreicht die Balancierungsspannung ihren vorgegebenen Wert. Daraufhin schaltet die Steuerung die Balancierung ab. Zu sehen ist, wie die Temperatur unterhalb der unteren Temperaturgrenze sinkt und nicht mehr ansteigt. Dies bedeutet, dass die Balancierung beendet ist. Dadurch zeigt sich, dass die Balancierungssteuerung wie den Anforderungen entsprechend funktioniert.

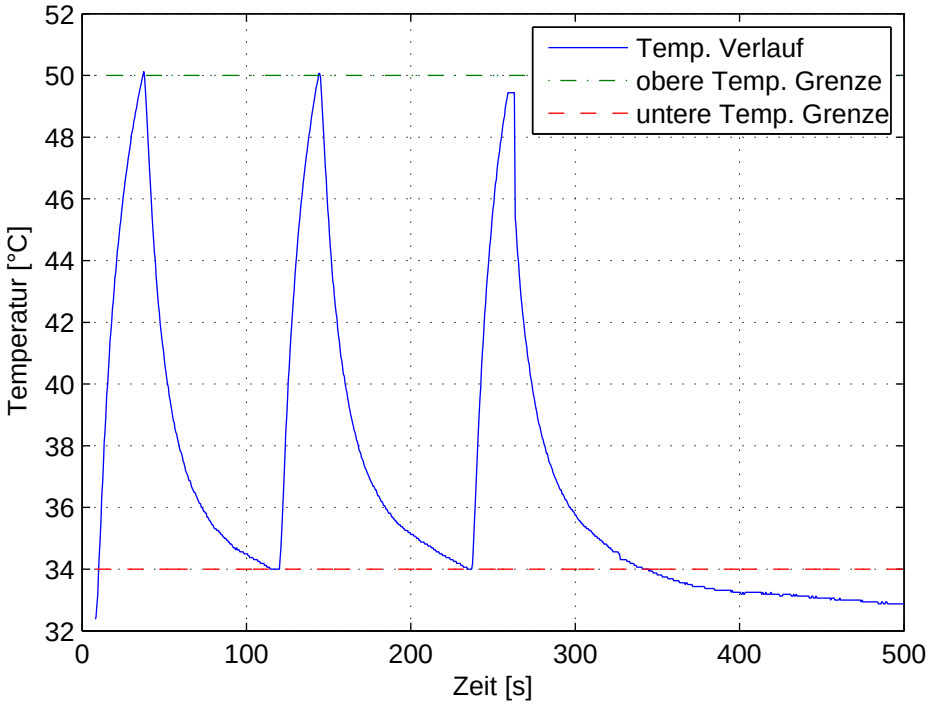


Abbildung 5.13.: Temperaturverlauf der Balancierungssteuerung

5.3. Einfluss des DC/DC-Wandlers auf die Zellenspannung

Nun soll noch der Einfluss des DC/DC-Wandlers auf die Batteriespannung untersucht werden. Bekannt ist, dass dieser durch interne Schaltvorgänge Störungen an der Eingangsspannung verursacht. Wie stark diese Störungen auf die zu messende Betriebsspannung sind und ob diese eine Beeinflussung auf den Betrieb des Zellensensors haben, muss im Weiteren untersucht werden.

5.3.1. Einfluss auf die Betriebsspannung

Zunächst wurde die Zellenspannung bei normalem Betrieb aufgenommen. Dies bedeutet, dass sich der Zellensensor im Empfangsmodus mit einer Taktfrequenz der DCO von 1 MHz befand. Als Vergleich wurde die anliegende Zellenspannung bei einem eingestellten DCO-Takt von 16 Mhz aufgenommen, da die DCO während der Burtsmessung ebenfalls bei 16 MHz getaktet wird. Dabei zeigte sich, dass der DC/DC-Wandler mit steigender Last an seinem Ausgang stärkere Störungen an der Zellenspannung verursacht. Eine höhere Last ist deswegen vorhanden, da der Mikrocontroller, bei steigender DCO Frequenz, mehr Strom benötigt. Dieser steig durch die Umstellung der DCO um ca. das 10-Fache an [12].
Die Beeinflussung der Zellenspannung ist in Abbildung 5.14 deutlich zu erkennen. Der obere Spannungsverlauf zeigt dabei die Eingangsspannung bei einem DCO-Takt von 1 MHz, die untere bei einem DCO-Takt von 16 MHz.

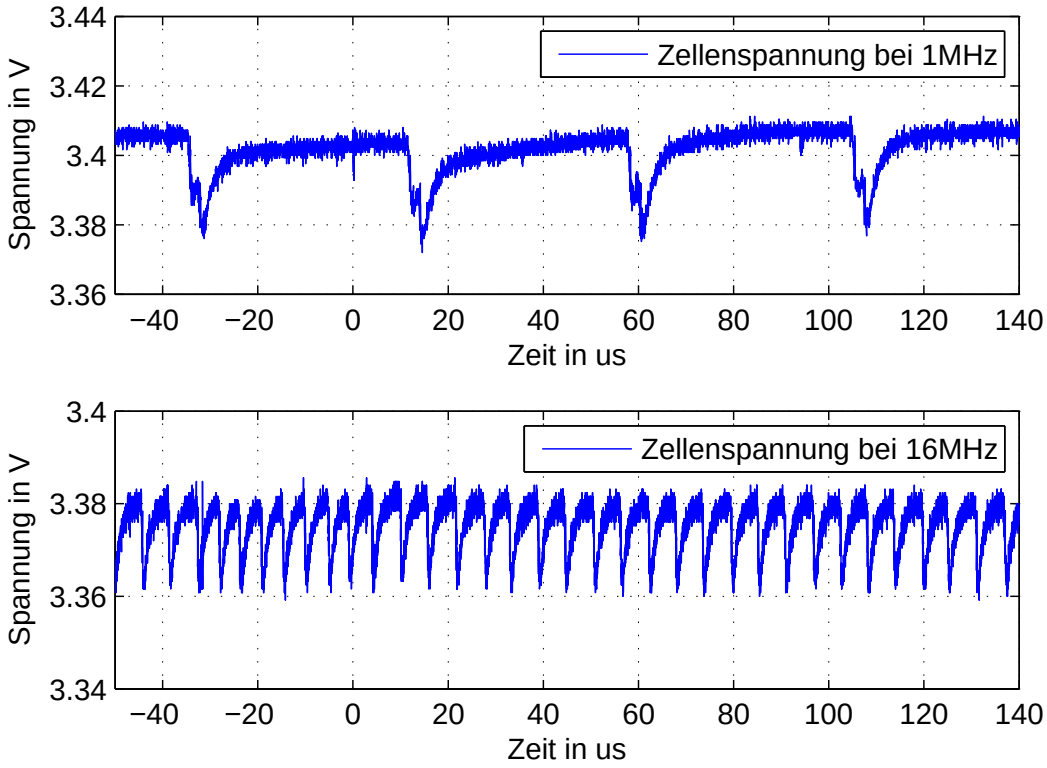


Abbildung 5.14.: Verhalten des DC/DC-Wandlers bei Last

Dieser Test macht deutlich, dass für eine genaue Spannungsmessung der Zellenspannung diese Störungen minimiert werden müssen. Besonders bei der Burstmessung können diese Störungen erheblichen Einfluss auf die Messergebnisse haben.

Abschaltung während einer Spannungsmessung

Nun wurde Untersucht, wie sich der DC/DC-Wandler bei der An- und Abschaltung durch den Mikrocontroller verhält. Dazu wurde eine einzelne Spannungsmessung bei 1MHz DCO-Takt durchgeführt. Kurz vor der ADC-Messung wurde nun der DC/DC-Wandler abgeschaltet. Wie in Abschnitt 4.4.3 auf Seite 56 berechnet, benötigt der ADC eine Mindestzeit von 64 μs für die Aufnahme des Spannungswerts. Für diese Zeit soll der DC/DC-Wandler also abgeschaltet werden. Der Wandler wurde während der Messung insgesamt 100 μs ausgeschaltet. Die Spannungsverläufe der Messung sind in Abbildung 5.15 zu sehen. Der obere Spannungsverlauf zeigt dabei den Moment der Abschaltung des DC/DC-Wandlers (LOW Phase). Der untere Spannungsverlauf zeigt die Zellenspannung.

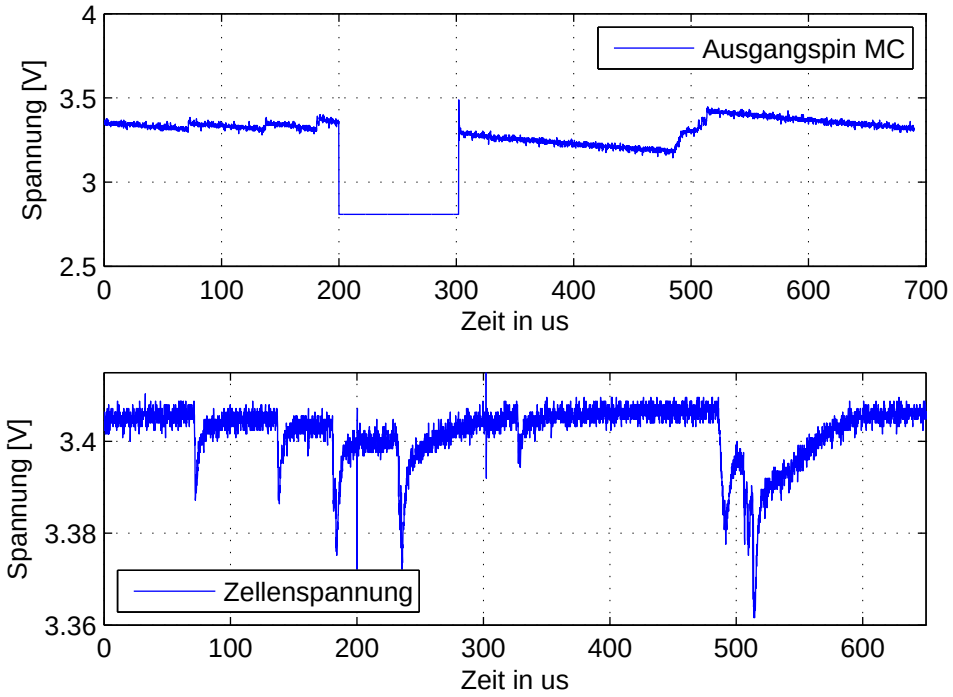


Abbildung 5.15.: Abschaltung des DC/DC-Wandlers während einer einzelnen ADC-Messung

Es ist zu erkennen, dass während der Abschaltung die Störungen des DC/DC-Wandlers immer noch vorhanden sind. Erst ca. 200 μs nach der Abschaltung sind keine Störungen mehr auf der Zellenspannung zu erkennen. Dies bedeutet, dass eine Abschaltung für die Zeit der ADC-Messung keine Auswirkungen hinsichtlich eines genaueren Messergebnisses hat. Ebenfalls lässt sich feststellen, dass der DC/DC-Wandler eine Zeit von ca. 180 μs benötigt, um nach dem Anschalten durch den Mikrocontroller wieder aktiv zu werden.

Abschaltung während der Burstmessung

Im Anschluss wird die selbe Messung nochmals bei einer 1 kHz Burstmessung durchgeführt. Die gesamte Abschaltdauer ist hier etwas kürzer, verursacht durch den schnelleren DCO-Takt. Dadurch werden einige ADC-Einstellungen schneller abgearbeitet. Die Abschaltdauer beträgt hierbei ca. 75 μs. Auch hier ist im oberen Teil der Abbildung 5.16 wieder der Zeitpunkt der Abschaltung (LOW Phase) zu sehen. Bei diesem Spannungsverlauf handelt es sich um einen Ausgangspin des Mikrocontrollers. Dieser wird bei der Abschaltung des DC/DC-Wandlers auf Low gezogen und bei erneutem Anschalten des Wandlers auf High. Gut zu erkenne ist, wie der Ausgangsport nach dem erneuten Anschalten langsam abfällt.

Daran lässt sich die sinkende Betriebsspannung auf dem Zellensensor erkennen. Diese steigt erst wieder an, nachdem der DC/DC-Wandler wieder in Betrieb ist. Dieses Verhalten war bereits in Abbildung 5.15 zu sehen. Dies war aber nicht so stark wie in dem Test mit 16 MHz DCO-Takt. Grund hierfür ist der größere Stromverbrauch des Mikrocontrollers während der Burstmessung.

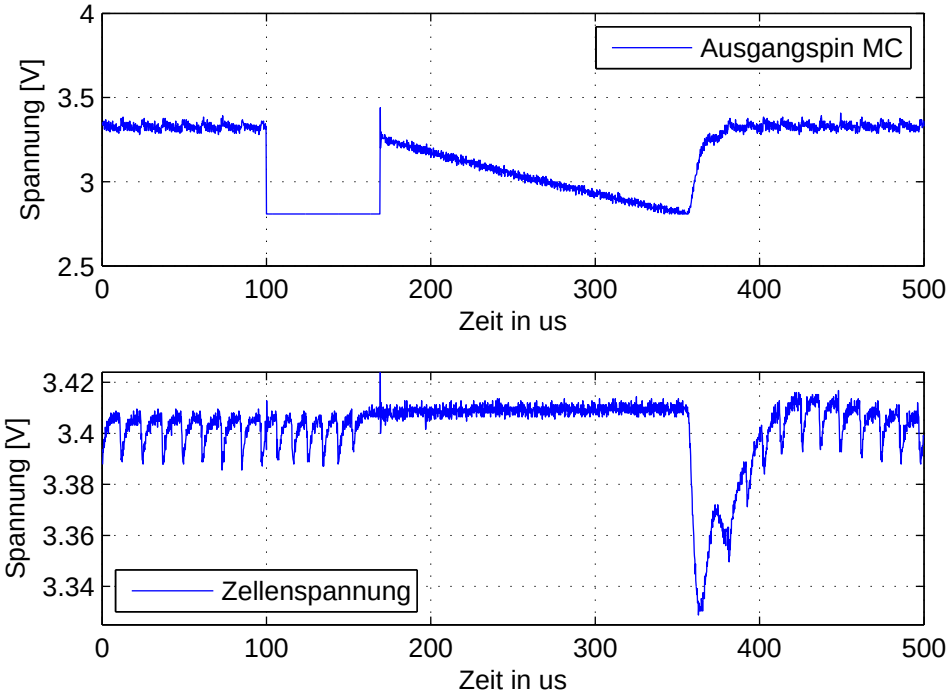


Abbildung 5.16.: Abschaltung des DC/DC-Wandlers während der Burstmessung

Zu erkennen ist bei dieser Messung, dass auch hier wieder die Abschaltung des DC/DC-Wandlers während der ADC-Messung nicht den gewünschten Erfolg bringt. Hier ist die Zellenspannung ca. 100 μs nach der Abschaltung erst frei von Störungen. Der DC/DC-Wandler reagiert auch hier wieder erst ca. 180 μs nach der Wiedereinschaltung und versorgt den Zellensensor mit der nötigen Betriebsspannung.

Abschaltung vor einer Spannungsmessung

Es wird nun vermutet, dass eine Abschaltung des DC/DC-Wandlers vor der eigentlichen ADC-Messung die nötige störungsfreie Zellenspannung liefern könnte. Um dies zu überprüfen wurde wieder eine einzelne Spannungsmessung durchgeführt. Dieses Mal wurde der

DC/DC-Wandler aber ca. 150 μs vor der ADC-Messung für ca. 80 μs ausgeschaltet. Der obere Spannungsverlauf in Abbildung 5.17 zeigt dabei den Moment der ADC-Messung (LOW Phase). Der untere Spannungsverlauf zeigt die Zellenspannung.

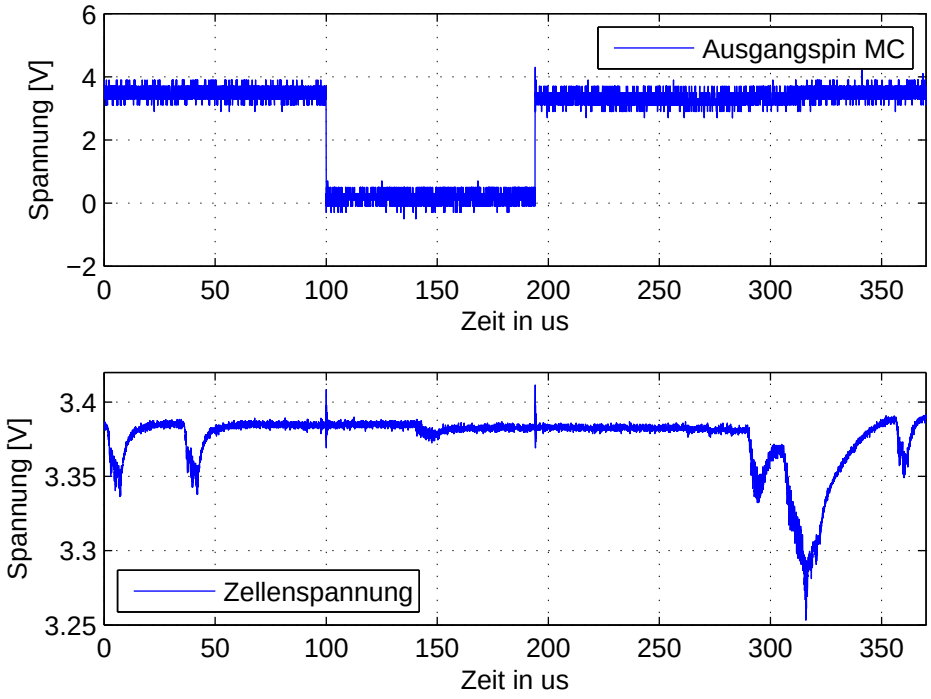


Abbildung 5.17.: Abschaltung des DC/DC-Wandlers vor einer einzelnen ADC-Messung

Zu erkennen ist, dass durch die Verschiebung der Abschaltung vor die ADC-Messung, die gewünschte Störungsfreiheit der zu messenden Spannung erreicht wird. Auch hier sind wieder ca. 180 μs für die Reaktivierung des DC/DC-Wandlers nötig.

Abschaltung vor einer Burstmessung

Dieses Prinzip, der Verschiebung des Abschaltzeitpunktes, wird nun auch bei der Burstmessung untersucht. In Abbildung 5.18 ist im oberen Spannungsverlauf der Abschaltzeitpunkt des DC/DC-Wandlers mit anschießender ADC-Messung zu sehen. Vom Zeitpunkt bei 25 μs bis ca. 80 μs wird der DC/DC-Wandler ausgeschaltet. Anschießend folgt die ADC-Messung der Zellenspannung bis zum Zeitpunkt 155 μs.

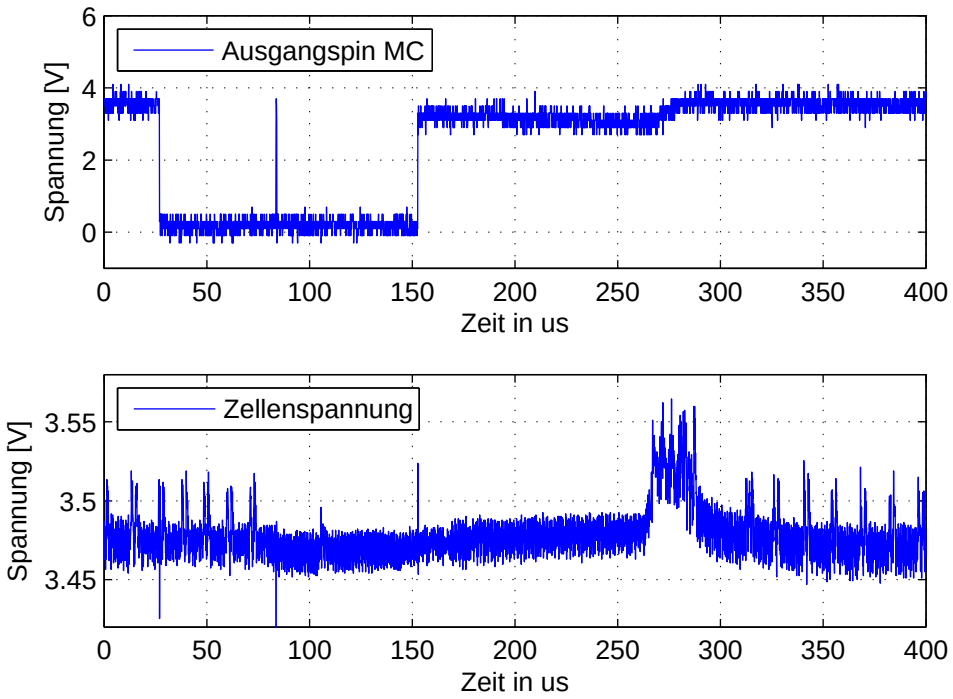


Abbildung 5.18.: Abschaltung des DC/DC-Wandlers vor einer Burstmessung

Während der ADC-Messung ist auch hier die Zellenspannung weitestgehend frei von Störungen. Allerdings ist auch hier, im oberen Spannungsverlauf (Ausgangspin MC), wieder eine abfallende Betriebsspannung zu sehen. Diese fällt bis zu dem Zeitpunkt, bei dem der DC/DC-Wandler wieder aktiv wird. Dies sind auch hier wieder ca. 180 μs nach dem Wiedereinschalten des DC/DC-Wandlers.
Diese Messungen zeigen, dass der Einfluss des DC/DC-Wandlers auf die Zellenspannung zu beachten ist, da dieser, abhängig von seiner Last, starke Störungen verursacht. Weiter war zu beobachten, dass der Wandler nicht sofort auf die An- und Abschaltung durch den Mikrocontroller reagiert. Dadurch müssen bestimmte Wartezeiten eingehalten werden. Diese müssen in der Software des Zellensensors noch berücksichtigt werden.

5.4. Untersuchung der Burstmessung

In diesem Abschnitt soll die Funktion der Burstmessung untersucht werden. Dabei ist zum einen eine hohe Messrate erwünscht, zum anderen soll dabei aber auch noch eine hohe Genauigkeit erzielt werden. Besonders wichtig ist eine feste Latenzzeit bei der Taktübertragung zwischen Batteriesteuergerät und den Zellensensoren. Durch eine schwankende Latenz bei dieser Übertragung wird es sehr schwierig eine genaue Messung zu erzielen, hinsichtlich der geplanten parallel stattfindenden Strommessung durch das Batteriesteuergerät. Da diese Schwankungen dazu führen, dass die aufgenommenen Stromwerte nicht mehr mit den durch die Zellensensoren aufgenommenen Spannungswerten verglichen werden können.
Ist die Latenzzeit hingegen konstant, kann diese einfach herausgerechnet werden.

5.4.1. Ermittlung der Messgrenzen

Wichtig ist die Bestimmung der möglichen Messgrenzen des Systems. Das heißt, bis zu welcher Messfrequenz noch Spannungswerte aufgenommen werden können und welche Komponenten diese beeinflussen. Von den bisherigen Messungen sind schon zwei Zeiten bekannt, die die Messgeschwindigkeit der Burstmessung beeinflussen. Diese sind zum einen die Wartezeit der An- und Abschaltung des DC/DC-Wandlers und zum anderen die Zeit, die für die ADC-Messung benötigt wird. Hinzu kommen noch Zeiten, die für den Aufruf der ISR[2] und der Abarbeitung von Funktionen innerhalb der ISR nötig sind. Da die Taktung des Mikrocontrollers aber 16MHz beträgt und diese Zeiten daher um ein vielfaches kleiner sind, werden diese zunächst nicht beachtet.
Für die theoretische Ermittlung der Messgrenze werden die Zeiten aus der Messung in Abschnitt 5.3.1 auf Seite 87 herangezogen. Diese waren für die An- und Abschaltung des DC/DC-Wandlers 55 μs und für die ADC-Messung 75 μs.

[2] Interrupt Service Routine

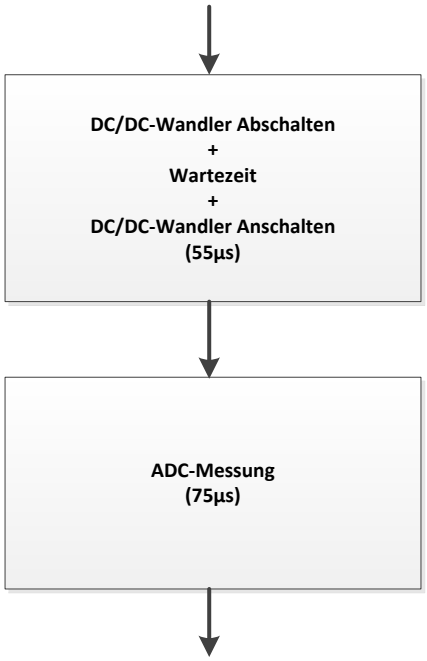


Abbildung 5.19.: Zeiten der theoretischen Messgrenze

Mit diesen Zeiten lässt sich eine erste Abschätzung der Messgrenzen berechnen.

$$f_{max} = \frac{1}{55\,\mu s + 75\,\mu s} = 7692,31\,Hz$$

Theoretisch lässt sich also die geforderte Messfrequenz von 2 kHz der Burstmessung leicht erreichen. Dies muss nun praktisch untersucht werden.

Testdurchführung

Um die theoretisch errechnete Messgrenze zu überprüfen, wurde ein Test mit der Messfrequenz von 7,5 kHz durchgeführt.

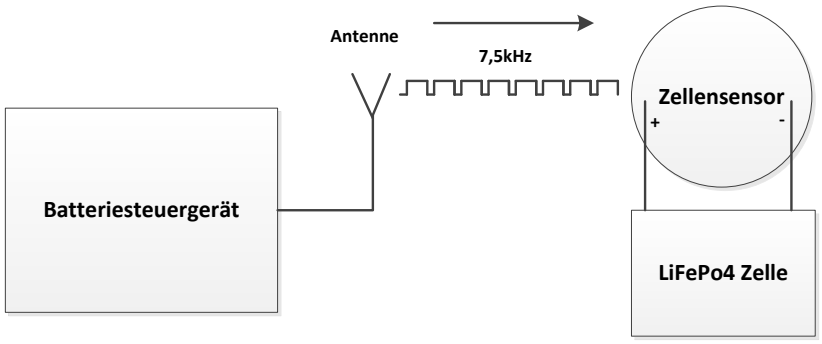


Abbildung 5.20.: Testaufbau zur Messgrenzenbestimmung

Bei der Durchführung der Messung zeigte sich kurz nach dem Anschalten der Burstmessung, dass sich der Zellensensor neu startete. Dies war erkennbar an den Status LEDs auf dem Zellensensor.
Vermutet wird, dass dadurch, dass der DC/DC-Wandler bei jeder Messung abgeschaltet wird, sich die Betriebsspannung nicht mehr erholen kann und somit nach ein paar ADC-Messungen unterhalb der geforderten Betriebsspannung des Mikrocontrollers fällt.
Um diese Vermutung zu bestätigen, wurde der Test wiederholt. Dabei wurden die gesendeten Messtakte des Batteriesteuergeräts und ein Ausgang des Mikrocontrollers mittels eines Oszilloskops aufgenommen. Der Ausgang am Mikrocontroller wurde dabei immer auf High-Potenzial geschaltet, wenn der DC/DC-Wandler durch den Mikrocontroller angeschaltet war.

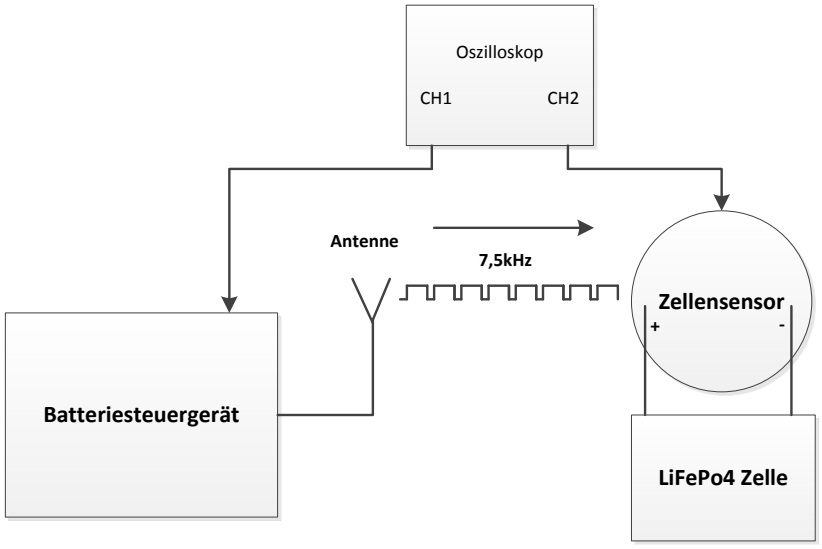


Abbildung 5.21.: Testaufbau zur Messgrenzenbestimmung / 2

Abbildung 5.22 zeigt das Ergebnis dieser Messung. Im oberen Spannungsverlauf ist zu erkennen, dass die Betriebsspannung nach der sechsten Messung zusammenbricht und der Zellensensor neu gestartet wird. Zu beachten ist in dieser Abbildung, dass der Spannungsverlauf darstellungsbedingt einen Offset von 4 V hat.

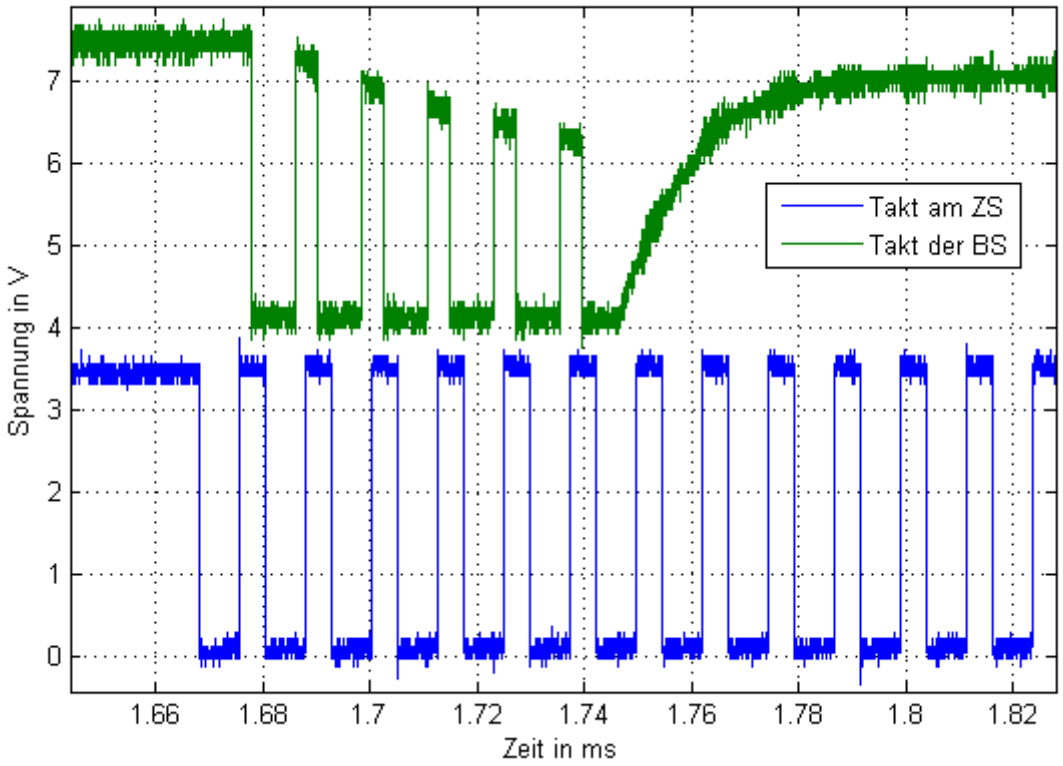


Abbildung 5.22.: Abschalten des DC/DC-Wandlers bei 7,5kHz

Durch diese Messung wird ersichtlich, dass der DC/DC-Wandler mehr Zeit zur Aktivierung und zur Stabilisierung der Betriebsspannung benötigt. In den Messungen im Abschnitt 5.3 war zu erkennen, dass der DC/DC-Wandler nach einer Zeit von ca. 180 μs wieder aktiv wurde. Diese Zeit wird nun in die Berechnung mit einbezogen und eine neue theoretische Messzeit ermittelt.

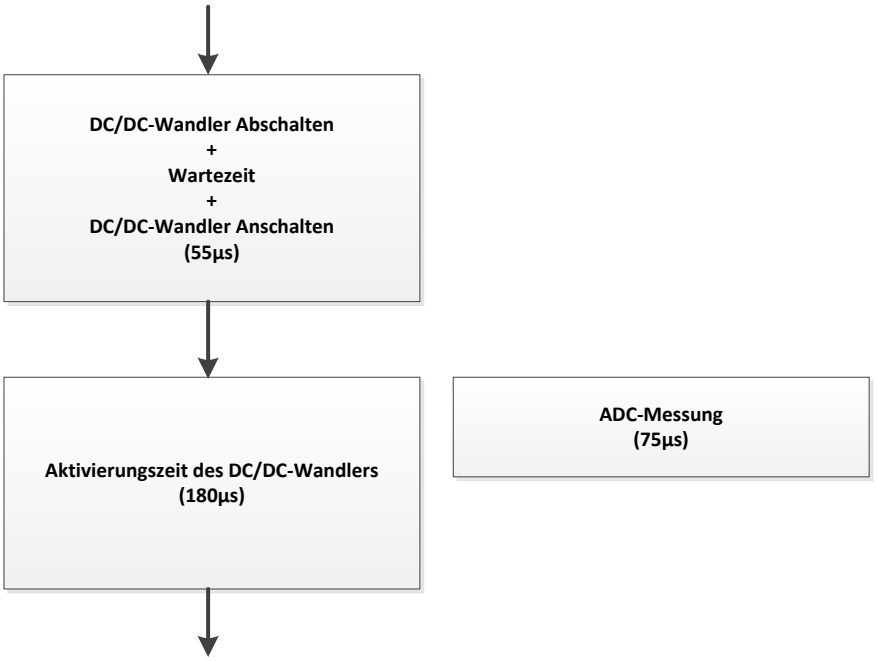


Abbildung 5.23.: Zeiten der theoretischen Messgrenze / 2

Die Zeit der ADC-Messung wird nun nicht mehr in die Rechnung mit einbezogen, da diese Messung innerhalb der Aktivierungszeit des DC/DC-Wandlers liegt. Es lässt sich somit eine neue maximale Burstfrequenz berechnen.

$$f_{max} = \frac{1}{55\,\mu s + 180\,\mu s} = 4255,32 Hz$$

Um diese Frequenz zu überprüfen wurde erneut ein Test durchgeführt. Dabei wurde mit einer Burstfrequenz von 4 kHz gemessen. Durch das Oszilloskop wurde das von dem Batteriesteuergerät sendende 4 kHz Taktsignal und die Zeit, in der der DC/DC-Wandler durch den Mikrocontroller aktiviert ist, aufgezeichnet.

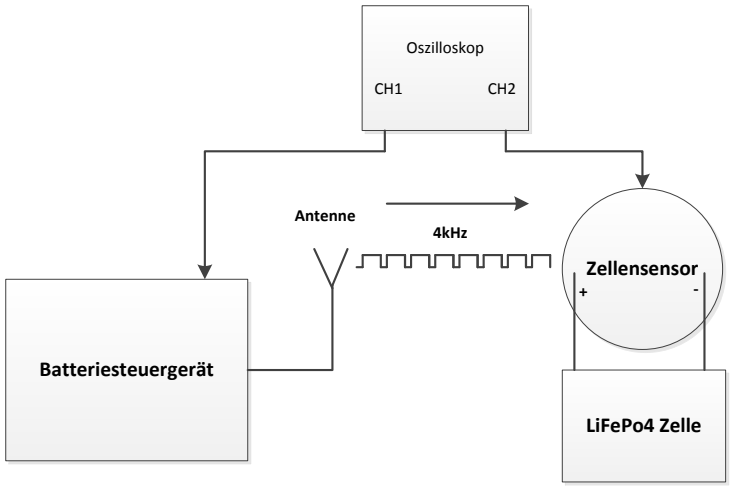


Abbildung 5.24.: Testaufbau zur Messgrenzenbestimmung / 4kHz

In Abbildung 5.25 (oberer Spannungsverlauf) ist sehr gut zu erkennen, wie die Betriebsspannung, während der DC/DC-Wandler bereits aktiviert ist, noch abfällt. Kurz bevor der Mikrocontroller den DC/DC-Wandler deaktiviert, steigt die Betriebsspannung wieder an.

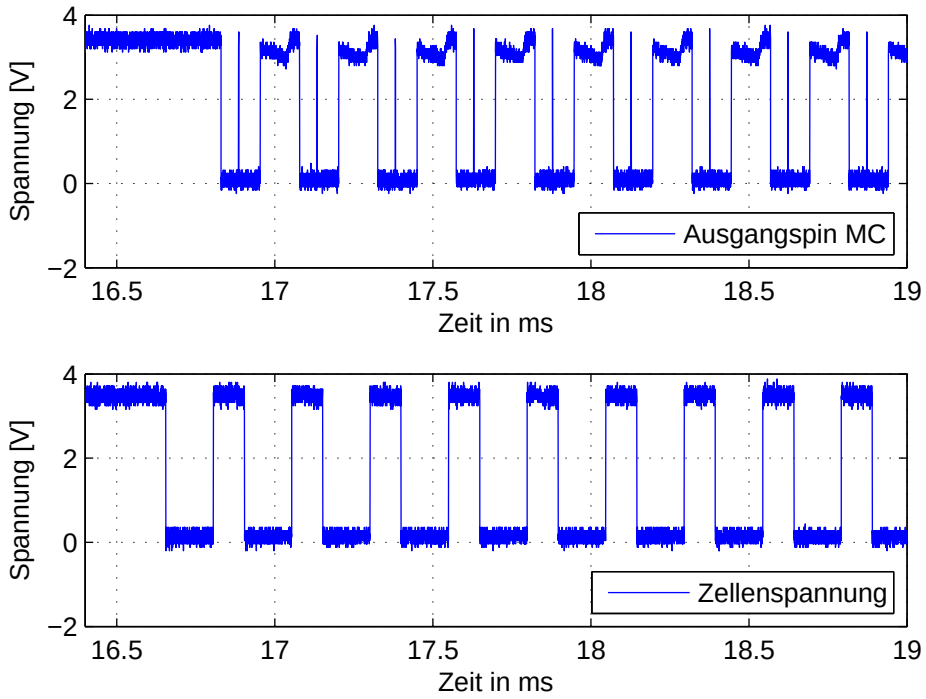


Abbildung 5.25.: Abschalten des DC/DC-Wandlers bei 4kHz

Somit kann festgelegt werden, dass die maximal mögliche Burstfrequenz mit der aktuellen Hardware bei 4kHz liegt. Beeinträchtigt wird diese Frequenz vor allem durch die Abschaltung des DC/DC-Wandlers. Diese Abschaltung ist allerdings für einen genauen Messbetrieb notwendig und kann daher nicht weggelassen werden.

5.4.2. Synchronisationsübereinstimmung

In diesem Abschnitt soll untersucht werden, wie synchron der Takt des Batteriesteuergeräts an den Zellensensoren ausgewertet wird. Dazu wurde zunächst die Latenzzeit zwischen dem Sender und Empfänger gemessen. Diese Zeit setzt sich aus drei Komponenten zusammen. Den Zeiten, die der Transceiver auf der Sendeseite für die Aufmodulierung, sowie zum Demodulieren auf der Empfängerseite des Signals benötigt. Zudem kommt noch die Laufzeit der Übertragung hinzu. Da die Laufzeit des Signals aber um ein Vielfaches kürzer ist, als die benötigte Zeit für die Auf- und Demodulation, kann diese vernachlässigt werden.

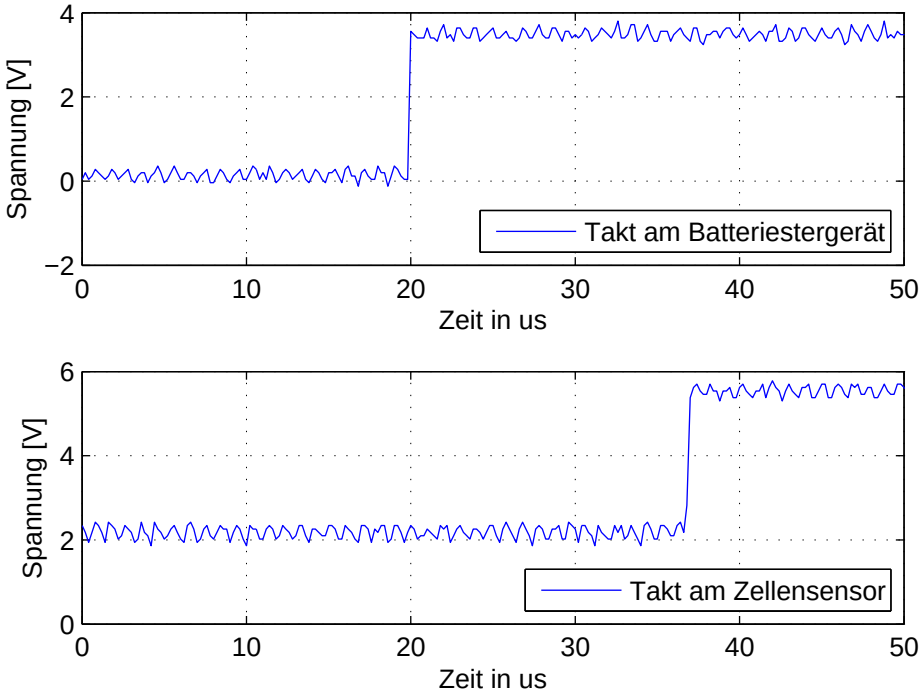


Abbildung 5.26.: Latenzzeit zwischen Batteriesteuergerät und Zellensensor

Aus der Messung in Abbildung 5.26 geht hervor, dass zwischen der Erzeugung des Taktsignals auf der Senderseite und dem Empfang des Taktes am Mikrocontroller auf der Empfängerseite, eine Latenzzeit von 18,4 μs vorhanden ist. Diese Latenzzeit ist in allen Messungen fest. Das heißt, es gab keinerlei Schwankungen während dieser Zeit. Damit kann angenommen werden, dass für das Auf- und Demodulieren des Taktsignals immer die selbe Zeit benötigt wird.

Erfassung eines Sinussignals

Zum ersten Test der Synchronität des Systems soll nun ein 100 Hz Sinussignal durch den Zellensensor gemessen werden. Dieses wird vom Zellensensor mit einer Burstfrequenz von 4 kHz abgetastet. Das Signal wird dabei von einem Signalgenerator erzeugt und mittels eines Oszilloskops aufgenommen. Mit dem Oszilloskop wird ebenfalls das Taktsignal des Batteriesteuergeräts aufgenommen, um die Zeitpunkte der Abtastung durch den Zellensensor zu erfassen.

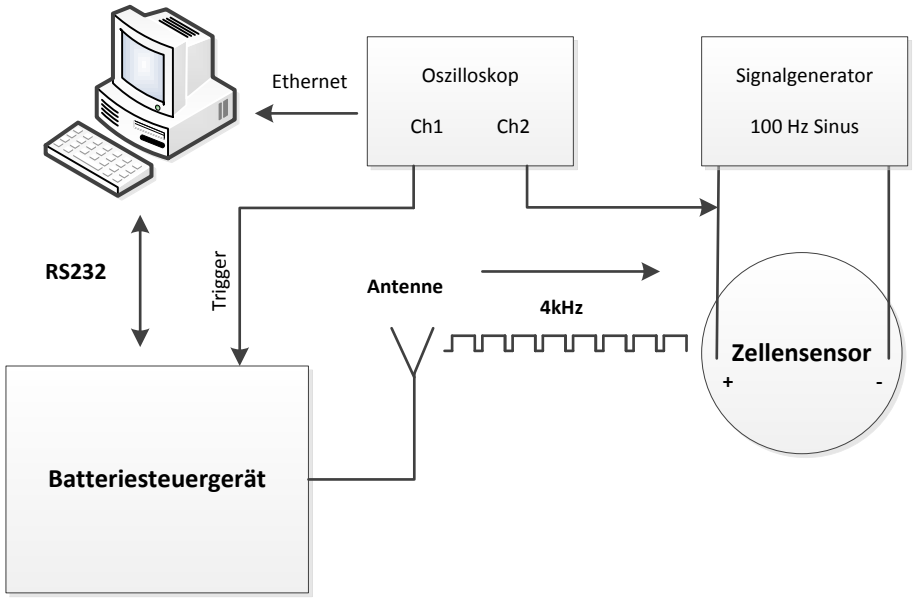


Abbildung 5.27.: Messaufbau zur Erfassung der Synchronisation

Die Daten des Oszilloskops und des Zellensensors wurden mittels Matlab ausgewertet. Dabei müssen die, durch den Zellensensor aufgenommenen, Spannungswerte mit denen des Oszilloskops zeitlich zugeordnet werden. Dies geschieht mit Hilfe des aufgenommenen Taktsignals (Trigger) des Batteriesteuergeräts. Die Abbildung 5.28 zeigt dabei die ausgewerteten Daten.

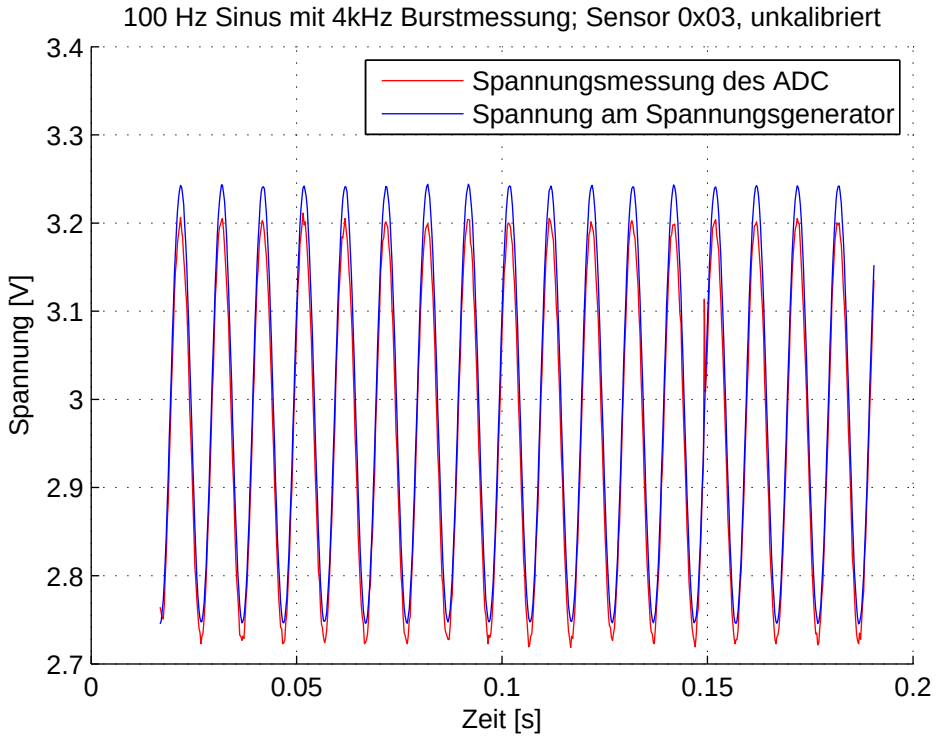
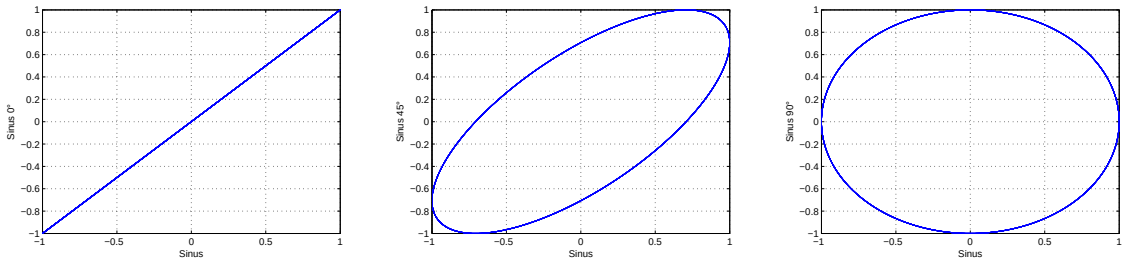


Abbildung 5.28.: Erfasster 100Hz Sinus

Es ist zu erkennen, dass die durch den Zellensensor gemessenen Spannungen zeitlich mit denen des Signalgenerators zusammenpassen. Die grundsätzliche Funktion der in dieser Arbeit entwickelten Burstmessung ist damit bewiesen. Zeitlich sind keine Abweichungen der Spannungswerte zu erkennen. Lediglich die Spannungshöhe weicht mit denen, durch das Oszilloskop aufgenommenen Spannungen ab. Dabei handelt es sich aber um einen Offset-fehler. Dieser lässt sich durch eine Kalibrierung der Spannung beheben. Leider konnte aus zeitlichen Gründen keine Kalibrierung der Spannungen mehr durchgeführt werden.

Phasenverschiebung bei verschiedenen Signalfrequenzen

Im ersten Test handelte es sich bei dem zu messenden Signal um ein im Vergleich zur Burstfrequenz langsames Signal. Deshalb ist zu überprüfen, wie sich die Messung bei höheren Signalfrequenzen verhält. Interessant ist dabei die Phasenverschiebung zwischen dem Referenzsignal des Signalgenerators und dem gemessenen Signal des Zellensensors. Eine gute Darstellungsart zum Messen von Phasenverschiebungen sind dabei die sogenannten Lissajousfiguren. Diese zeigen die Phasenverschiebung zweier Signale mittels Öffnung einer Ellipse.



(a) Phasenverschiebung 0° (b) Phasenverschiebung 45° (c) Phasenverschiebung 90°

Abbildung 5.29.: Lissajousfiguren

Dargestellt sind in Abbildung 5.29 drei verschieden Fälle der Phasenverschiebung. Der Referenzsinus ist dabei auf der X-Achse des Diagramms abgebildet. Der um eine bestimmte Phase verschobene Sinus ist auf der Y-Achse abgetragen. Die Abbildung (a) zeigt eine Phasenverschiebung von 0°. Dabei wird lediglich ein Strich angezeigt. Die Ellipse ist noch vollständig geschlossen. Bei einer Phasenverschiebung von 45°, wie in (b), öffnet sich die Ellipse. Haben die beiden Signale eine Phasenverschiebung von 90° ist die Ellipse weit geöffnet und es entsteht ein vollständiger Kreis. Dies bedeutet, je weiter die Ellipse geöffnet ist, desto größer ist die vorhandene Phasenverschiebung der beiden Signale. Diese Darstellungsform wird nun angewandt, um eine Phasenverschiebung in den gemessenen Signalen zu erkennen.

Um den Test durchzuführen, wurde wieder der Testaufbau der Abbildung 5.27 auf Seite 97 verwendet. Als Burstfrequenz wurden 2 kHz gewählt und damit verschiedene Signalfrequenzen gemessen.

In der ersten Lissajousfigur (Abbildung 5.30) wurde ein 100 Hz Sinussignal mittels des Signalgenerators erzeugt und durch den Zellensensor gemessen. Zu erkennen ist eine leichte Phasenverschiebung zwischen dem Referenzsignal und dem gemessenen Sinus. Dies ist an der leicht geöffneten Ellipse zu erkennen.

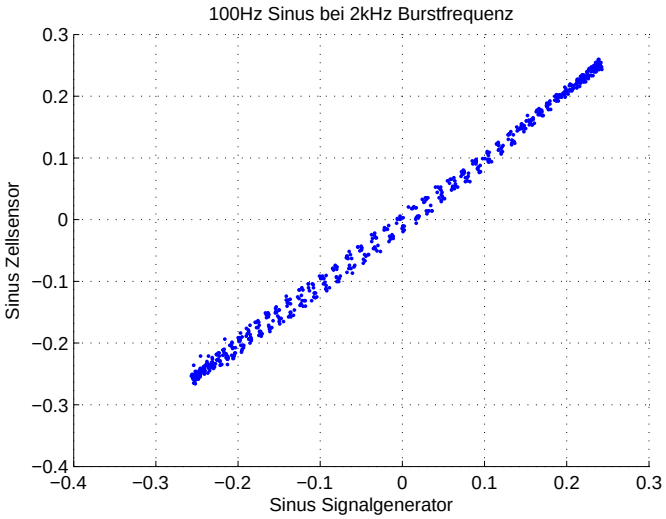


Abbildung 5.30.: Lissajous Figur: 100Hz

Die zweite Messung wird in Abbildung 5.31 dargestellt. Dabei wurde die Frequenz des Sinussignals auf 200Hz erhöht. Die Burstfrequenz blieb auf 2 kHz. Dabei ist eine Weitung der Ellipse zu erkennen. Dies bedeutet, dass eine größere Phasenverschiebung zwischen dem Referenzsinus und dem gemessenen Sinus vorliegt.

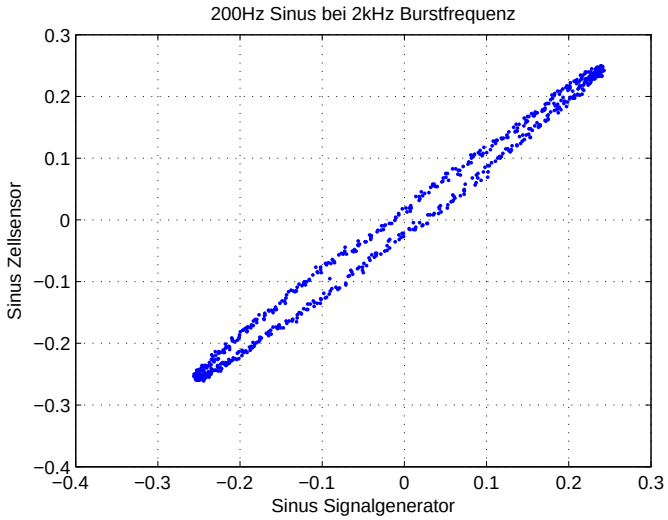


Abbildung 5.31.: Lissajous Figur: 200Hz

Die nächste zu messende Frequenz des Sinus war 500 Hz. Dabei ist in Abbildung 5.32 eine deutlich größere Öffnung der Ellipse zu erkennen, welche die Folge einer steigenden Phasenverschiebung ist.

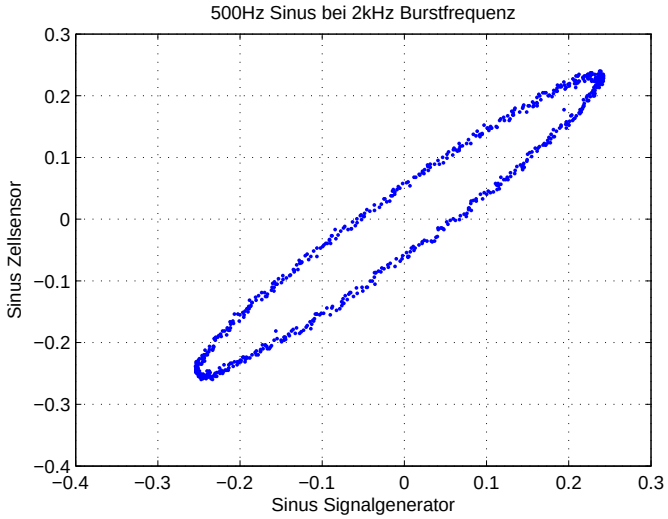


Abbildung 5.32.: Lissajous Figur: 500Hz

Die letzte Messung wurde mittels eines Referenzsinus mit der Frequenz von 1.000 kHz durchgeführt. Dabei handelt es sich um das schnellste Signal, welches bei einer Burstfre-

quenz von 2.000 kHz noch zu messen ist[3]. Bei der Betrachtung der entstandenen Lissajous Figur, erkennt man das sich die Phasenverschiebung vergrößert hat.

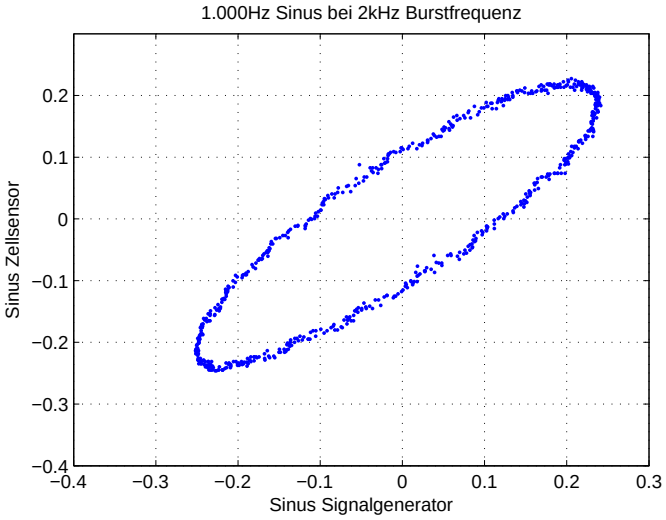


Abbildung 5.33.: Lissajous Figur: 1.000Hz

Durch diese Messung ist festgestellt worden, dass die Phasenverschiebung bei steigender Frequenz anwächst. Ein vermuteter Grund dieser frequenzabhängigen Phasenverschiebung ist zum einen die Latenzzeit der Taktübertragung zwischen Batteriesteuergerät und dem Zellensensor und zum anderen, die Zeit, die der AD-Wandler für die Messung des Spannungswerts benötigt. Die Latenzzeit wurde bei der vorangegangenen Messungen mit einer Zeit von 18,4 μs bestimmt. Für die Zeit, die der AD-Wandler für eine Aufnahme eines Spannungswerts benötigen soll, wurden 64 μs eingestellt (siehe dazu 57). Mit diesen Zeiten als Grundlage, lässt sich nun die Phasenverschiebung der gemessenen Signale in einer ersten Näherung berechnen. Zeiten für das Auslösen und Abarbeiten des Interrupts am Mikrocontroller werden zunächst vernachlässigt. Somit ergibt sich eine Gesamtverzögerung der Messung von 82,4 μs.

$$\text{Latenzzeit der Taktübertragung + AD-Wander Messzeit} = \text{Gesamtverzögerung} \tag{5.1}$$

$$18,4\,\mu s + 64,0\,\mu s = 82,4\,\mu s$$

Damit kann nun die mögliche Abweichung berechnet werden.

$$\frac{\text{Abweichung[s]} \cdot 360\,^{\circ}}{\text{Periodenzeit[s]}} = \text{Abweichung}\,[^{\circ}] \tag{5.2}$$

[3] Abtasttheorem auf Seite 31

Für den 100 Hz Sinus ergeben sich 2,96 ° Abweichung

$$\frac{82,4\,\mu s \cdot 360\,^\circ}{10\,ms} = 2,96\,^\circ$$

Für den 200 Hz Sinus ergeben sich 5,93 ° Abweichung

$$\frac{82,4\,\mu s \cdot 360\,^\circ}{5\,ms} = 5,93\,^\circ$$

Für den 500 Hz Sinus ergeben sich 14,83 ° Abweichung

$$\frac{82,4\,\mu s \cdot 360\,^\circ}{2\,ms} = 14,83\,^\circ$$

Für den 1.000 Hz Sinus ergeben sich 29,66 ° Abweichung

$$\frac{82,4\,\mu s \cdot 360\,^\circ}{1\,ms} = 29,66\,^\circ$$

Korrigiert man nun die durch den Zellensensor gemessenen Werte mit der Zeit von 82,4 μs, erhält man die folgenden Figuren.

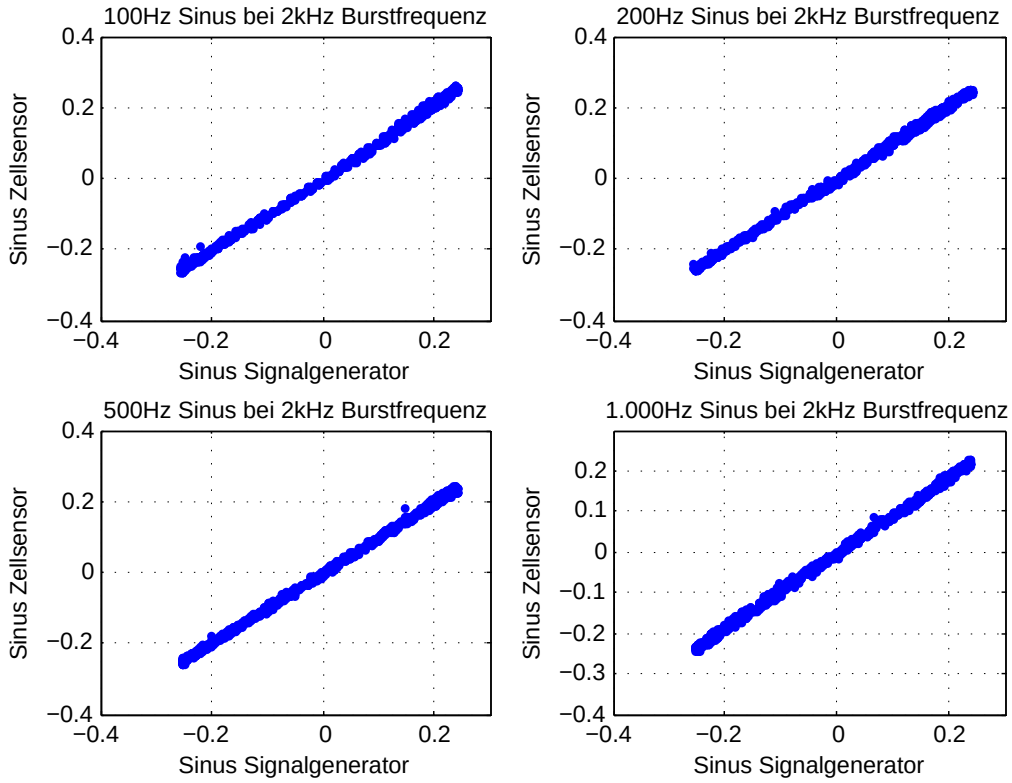


Abbildung 5.34.: Korrigierte Lissajous Figuren

Es zeigt sich, dass die Phasenverschiebung bei den unterschiedlichen Frequenzen allein durch die Latenzzeit des Taktsignals zwischen Batteriesteuergerät und dem Zellensensor und der Aufnahmezeit durch den ADC des Mikrocontrollers zustande kommt. Daraus lässt sich schließen, dass die entwickelte Messmethode selbst bei hohen Frequenzen korrekte Werte bezüglich der Phase liefert.

5.4.3. Erprobung mit aufgezeichneten Signalen

Nun soll versucht werden, den Spannungsverlauf eines Hochstromereignisses mithilfe der Burstmessung aufzunehmen. Dazu wurde der Spannungsverlauf, der beim Start eines PKW vorkommt, in einen, aus einer vergangenen Bachelorarbeit [20] entwickelten, Zellspannungsgenerator geladen. Dieser soll nun den zu messenden Spannungsverlauf wiedergeben. Dabei wurde mit einer Burstfrequenz von 1 kHz gemessen und es sollten 700 Messwerte aufgenommen werden. Dies entspricht einer Aufnahmezeit von insgesamt 700 ms. Den aufgenommenen Verlauf zeigt die folgende Abbildung. Der blau dargestellte Spannungsverlauf zeigt die vom Zellspannungsgenerator ausgegebene Spannung. Der rote Spannungsverlauf ist die durch den Zellensensor aufgenommene Spannung. Vergleicht man diese miteinander, sind sie nahezu identisch.

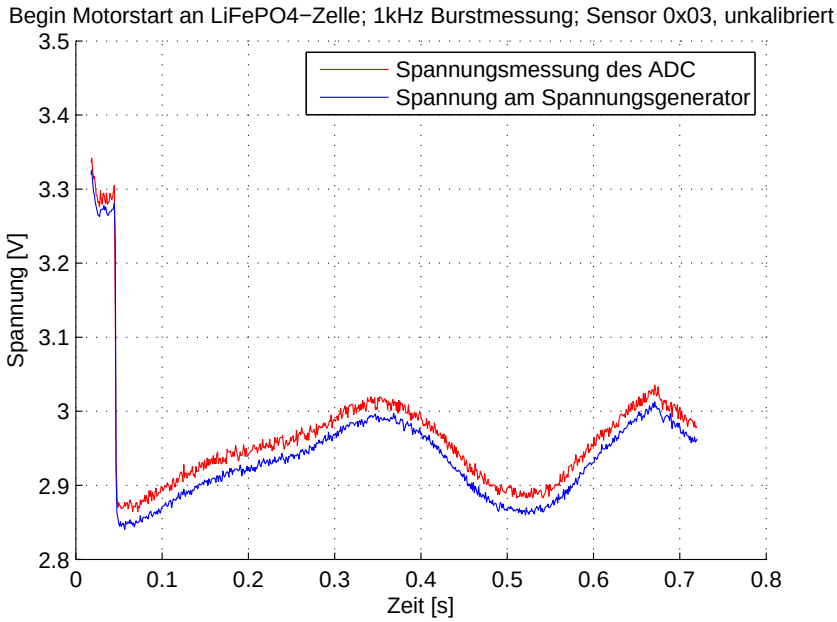


Abbildung 5.35.: Erfassung eines Motorstarts an einer LiFePo4 Zelle

Allerdings ist auch hier wieder ein erkennbarer Offset vorhanden. Da diese Zellensensoren aber noch nicht kalibriert sind, lässt sich dieser Fehler durch eine Kalibrierung der Sensoren beheben. Somit ist auch die Anforderung, Hochstromereignisse zu erfassen, erfüllt worden. Abbildung 5.36 zeigt hierzu nochmal den Vergleich zwischen der in dieser Arbeit untersuchten Möglichkeit zur Erfassung von Hochstromereignissen der Sensorklasse 3 und die Möglichkeit der Klasse 1 Sensoren. Dabei ist ein deutlicher Unterschied zwischen dem aufgezeichneten Spannungsverlauf des Klasse 1 Sensors und den, durch die Burstmessung aufgenommenen, Spannungswerte ersichtlich. Dies zeigt einen deutlichen Fortschritt in der Entwicklung der Zellensensoren.

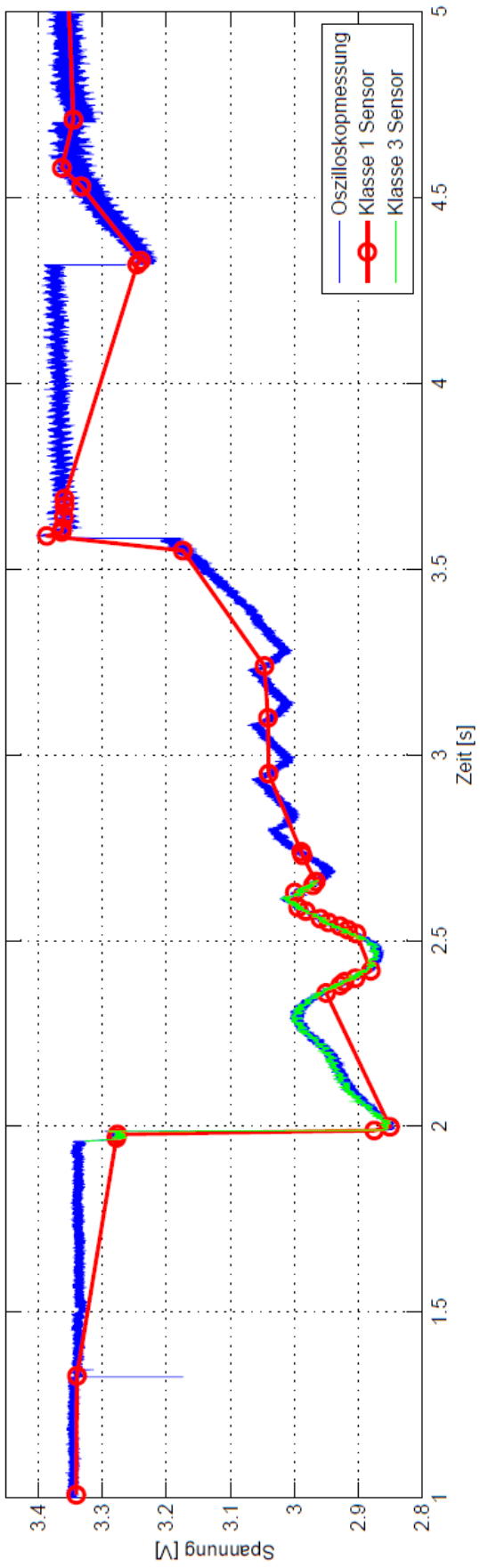


Abbildung 5.36.: Erfassung eines Motorstarts an einer LiFePo4 Zelle / Vergleich

6. Zusammenfassung und Bewertung

In der vorliegenden Arbeit ist ein erfolgreiches Redesign eines Zellensensors durchgeführt worden. Bei diesem Sensor handelt es sich um einen Zellensensor der Klasse 3, welcher im Projekt BATSEN definiert wurde.

Es wurde ein modifizierter Platinenentwurf des Zellensensors entwickelt. Dieser sollte neben der Möglichkeit zur Integration innerhalb einer LiFePo4 Zelle, auch die Möglichkeit besitzen, außerhalb einer solchen Zelle befestigt werden zu können. Neben hardwaretechnischen Änderungen wurde die Software des Batteriesteuergeräts und des Zellensensors überarbeitet.

Dabei wurde der Funktionsumfang des Zellensensors um mehrere Funktionen erweitert. Wie in der Aufgabenstellung (siehe Anhang S. 120) gefordert, wurde dazu der bereits auf dem Zellensensor vorhandene Temperatursensor in die Software mit integriert. Die Software kann nun neben dem Abruf der Temperatur des Sensors, auch die vom Temperatursensor unterstütze Alarmfunktion nutzen. Ebenso wurde gemäß Aufgabenstellung eine Balancierungssteuerung implementiert. Diese balanciert die Zellenspannung nicht nur auf die von dem Batteriesteuergerät vorgegebene Spannung, sondern kontrolliert zudem noch die benötigte Zeit der Balancierung und die Temperatur an den Balancierungswiderständen.

Eine weitere Funktion die hinzugefügt wurde, ist das adressierbare Wake-Up. Damit können einzelne Sensoren geweckt und angesprochen werden, während sich andere Sensoren weiter in einem Stromsparmodus befinden. Kernstück dieser Arbeit war die Untersuchung und Erprobung einer funksynchronisierten Messung. Diese wurde innerhalb dieser Arbeit als Burstmessung bezeichnet. Durch diese Burstmessung ist es nun mögliche, Spannungsverläufe hochauflösend darzustellen. Durch eine Erweiterung des Batteriesteuergeräts, welches allerdings nicht Teil dieser Arbeit war, wurde die Grundlagen für die spätere Impedanzspektroskopie geschaffen.

6.1. Bewertung der Erweiterungen und des Redesign des Sensors

Die geforderten Erweiterungen des bestehenden Systems konnten alle erfüllt werden. Schwerpunkt der Erweiterungen waren die Einbindung des Temperatursensors in die Software des Systems und der Entwurf und die Implementierung einer Balancierungssteuerung.

Der Temperatursensor TMP102 zeigte sich in der Anwendung im Temperatur-Klimaschrank als sehr genau. Geplant war es, eine Kalibrierung der Temperatur an diesem Sensor durchzuführen. Im Test zeigte sich allerdings, dass diese nicht nötig war, da der Temperatursensor bereits ohne größere Abweichungen misst. Ob und wie groß die tatsächlichen Abweichungen sind, konnte mit den zur Verfügung stehenden Messgeräten nicht überprüft werden, da die Abweichungen des Sensors in einem sehr kleinen Bereich liegen. Bei dem Sensor handelt es sich um einen IC, von dem die Temperaturdaten über die I2C Schnittstelle abgerufen werden. Die Temperatur ist dabei nicht von externen Bauteilen abhängig. Dadurch ist davon auszugehen, dass der Hersteller “Texas Instruments” diesen bereits kalibriert ausliefert.

Die implementierte Steuerung der passiven Ladungsbalancierung zeigt in den Testläufen gute Ergebnisse. Allerdings konnten aus Zeitgründen keine Langzeittests durchgeführt werden. Ebenfalls fehlen Erfahrungswerte, welche Temperaturgrenzen bei dieser Steuerung sinnvoll sind um die thermische Belastung der Platine und der Zelle gering zu halten. Bei den hier durchgeführten Tests wurden die Temperaturgrenzen der Steuerung frei gewählt. Diese Grenzen lassen sich allerdings leicht in der Software ändern.

Das Redesign der Zellensensorplatine war ebenfalls erfolgreich. So ist es nun möglich, die Sensorplatine wahlweise in einer LiFePo4 Rundzelle zu integrieren, oder diese direkt an den Anschlusspolen zu montieren. Leider zeigte sich in der Inbetriebnahme der Platine, dass sich durch die geometrisch enge Platzierung der Bauteile die Resonanzfrequenz der Schleifenantenne verschoben hat. Durch eine Variation der Bauteile des Anpassnetzwerks konnte dies allerdings wieder teilweise korrigiert werden. Dennoch ist dies für einen weiteren Betrieb der Platine zu untersuchen.

Das Hinzufügen eines zusätzlichen Programmierinterface zeigte sich während der Entwicklung des Zellensensors als sehr praktisch. Da dadurch eine schnelle Programmierung der einzelnen Zellen möglich war. Für das einmalige Programmieren einer Kleinserie bietet sich aber weiterhin die Programmierung über den Nadeladapter an, da dadurch der Aufwand für die Montage des Steckers eingespart werden kann.

6.2. Bewertung der synchronisierten Burstmessung

Die hier erarbeitete Burstmessung erwies sich in den durchgeführten Tests als sehr präzise. Die Anforderung an eine Mindest-Burtsfrequenz von 2 kHz konnten weit übertroffen werden. Dennoch zeigte die Burstmessung bei Frequenzen über 4 kHz Grenzen. Ein wesentlicher Einflussfaktor auf die zu erreichende Burstfrequenz ist der eingesetzte DC/DC-Wandler. Bedingt durch die nötige Abschaltung und die darauf folgende Aktivierungszeit, trägt dieser einen Großteil zur benötigten Gesamtmesszeit bei.

Auch die Anzahl der aufzunehmenden Messwerte stellt hinsichtlich der Aufnahmedauer eine Grenze dieser Messmethode. Durch den begrenzten Speicher des Mikrocontrollers MSP430F235 von 2 kByte, kann nur eine bestimmte Anzahl von Messwerten aufgenommen werden. In der Praxis zeigte sich, dass diese Grenze bei ca. 800 Werten lag. Dies reicht allerdings aus, um die Messmetohode zu erproben.

6.3. Ausblick

Mit der Einführung der synchronisierten Burstmessung hat der Zellensensor eine wichtige neue Funktion erhalten. Durch die Funktionserweiterung ist es nun möglich, Spannungsverläufe hochauflösend darzustellen. Dies war mit den vergangenen Sensoren der Klasse 1 und Klasse 2 nur bedingt bis gar nicht möglich. Dadurch ist diese Arbeit ein wertvoller Beitrag für das Forschungsprojekt BATSEN. Diese neue Messmethode stellt darüber hinaus eine wichtige Grundlage für weiterführende Arbeiten und Untersuchungen dar. So kann durch die Realisierung der parallelen Strommessung durch das Batteriesteuergerät eine drahtlose Impedanzspektroskopie der Batteriezellen durchgeführt werden.

Ein offen gebliebener Punkt ist die Kalibrierung der Spannung sowie die Kalibrierung des Temperatursensors innerhalb des MSP430F235. Aus zeitlichen Gründen konnte dies im Zuge der vorliegenden Arbeit nicht mehr durchgeführt werden.

In weiteren Arbeitsschritten sollten der hier Entwickelte Zellensensor weiter bearbeitet werden. So muss die Anpassung der Schleifenantenne am Zellensensor nochmals überarbeitet werden. Durch das dichte Zusammenrücken der Bauteile hat sich vermutlich die Resonanz der Antenne verschoben. Dadurch muss das Impedanzanpassnetzwerk neu ausgelegt werden, um einen besseren Eingangsreflexionsfaktor zu erhalten. Diese Anpassung konnte wegen Umbaumaßnahmen in den nötigen Laboren nicht mehr realisiert werden. Falls die Anpassung des Impedanzanpassnetzwerks keine Besserung der Empfangsprobleme bringt, könnten die Bauteile bei einem eventuellen weiteren Redesign wieder mit einem größeren Abstand platziert werden. Dies ist möglich, da es noch einige Platzreserven um das Montageloch der Platine gibt. Dieser Platz war vorgesehen, um die Zellensensoren gegebenenfalls

an größeren Zellenpolen befestigen zu können. Dabei wäre eine Montage auf einen Zellenpol mit bis zu 20mm Durchmesser möglich.

Um weiteren Platz zu schaffen, kann der vorhandene Wake-Up Receiver AS3930 der Firma AMS ausgetauscht werden. Bisher wird der Receiver in einer 16-Pin-TSSOP Bauform verwendet. Dieser hat eine Platzbedarf von 6,4 x 5,0 mm. Laut Hersteller ist dieser jetzt auch in der kleineren QFN-Bauform erhältlich. Dieser Chip hat dann nur noch einen Platzbedarf von 4 x 4 mm [2].

Weiter werden bei der Übertragung der Spannungs- und Temperaturdaten an das Batteriesteuergerät 4 Bit nicht genutzt, die aber mitgesendet werden müssen. Da die ADC- und die Temperaturmessung jeweils 12 Bit liefern, müssen diese in jeweils 2 Byte gesendet werden. So besteht zum Beispiel die Möglichkeit, einen Zeitstempel mit zu senden oder die Daten so zu komprimieren, dass 4 Messwerte mit je 12 Bit in 6 Byte gesendet werden, statt wie bisher in 8 Byte. Dies würde die Sendedauer um 25 % verringern.

Ebenfalls ist nun die Verwendung der im Temperatursensor TMP102 und in der Software vorhandenen Alarmfunktion möglich. Dieser könnte das Batteriesteuergerät über eine Temperaturüberschreitung der Zelle Informieren. Allerdings ist es dazu notwendig, dass der Zellensensor eigenständige Botschaften an das Batteriesteuergerät senden kann. Bisher ist dies nicht möglich, da der Zellensensor nur auf Befehl eine Antwort sendet. Dies könnte aber rein durch eine Softwareänderung gelöst werden. Ein weiterer interessanter Punkt könnte die nähere Betrachtung der DCO-Fehlerkorrektur sein. In Abschnitt 4.4.3 wurde kurz eine Möglichkeit gezeigt, wie derzeit die Fehlerkorrektur in der Software erfasst werden kann.

Eine weitere Möglichkeit zur DCO-Fehlerkorrektur bietet der Transceiver CC1101. Dieser ist mit einem 26 MHz Quarz ausgestattet. Durch eine Konfiguration lässt sich dieser Takt über einen Ausgangspin mit verschiedenen Vorteilern ausgeben [7]. Dieses Signal kann dann ebenfalls über einen Timer ausgewertet und mit dem internen DCO-Takt verglichen werden. Mit dieser Methode lässt sich ein sehr genaues Referenztaktsignal ausgeben. Nachteilig ist, dass während dieser Zeit, in der der Transceiver als Taktquelle dient, dieser nicht im Sende- oder Empfangsbetrieb sein darf. Anzudenken wäre, dass das Batteriesteuergerät ein Kommando an die Zellensensoren zur Synchronisierung der DCO durch den Transceiver sendet. Dadurch weiß das Batteriesteuergerät, das die Sensoren eine gewisse Zeit keine Daten Empfangen oder auch senden können.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] ADAC. *ADAC Pannenstatistik 2013*. Abgerufen am: 14.07.2013. www.adac.de/_mmm/pdf/27841_169836.pdf.

[2] AMS. *AS3930 Single Channel Low Frequency Wakeup Receiver*. Abgerufen am: 31.07.2013. www.ams.com/eng/content/download/23692/414425/file/AS3930_Datasheet_v1_00.pdf.

[3] Analog Devices. *Wideband 4 GHz, 43 dB Isolation at 1 GHz*. Abgerufen am: 30.07.2013. www.analog.com/static/imported-files/data_sheets/ADG918_919.pdf.

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

[5] eec Repenning GmbH. *ECC-LFPP 45 ECC-LFPP 48*. Abgerufen am: 01.08.2013. www.eccbatteries.com/files/datenblatt_lfpp_45-lfpp_48_06-2013.pdf.

[6] Sergej Ilgin. *Drahtlose Sensoren für Batteriemodule - Konzeption, Kalibrierung, Hard- und Softwareentwicklung*. HAW Hamburg, Bachelorthesis, Aug. 2011. .

[7] Texas Instruments. *CC1101 - Low-Power Sub-1 GHz RF Transceiver*. Abgerufen am: 20.03.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=cc1101&fileType=pdf.

[8] Texas Instruments. *Cell balancing buys extra run time and battery life*. Abgerufen am: 22.07.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=slyt322&fileType=pdf.

[9] Texas Instruments. *LOW INPUT VOLTAGE SYNCHRONOUS BOOST CONVERTER WITH 1.3-A SWITCHES*. Abgerufen am: 13.07.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=tps61201&fileType=pdf.

[10] Texas Instruments. *Low Power Digital Temperature Sensor With SMBus/Two-Wire Serial Interface in SOT563*. Abgerufen am: 30.07.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=tmp102&fileType=pdf.

[11] Texas Instruments. *MSP430F15x, MSP430F16x, MSP430F161x MIXED SIGNAL MICROCONTROLLER*. Abgerufen am: 31.07.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=msp430f169&fileType=pdf.

[12] Texas Instruments. *MSP430F23x, MSP430F24x(1), MSP430F2410 Mixed Signal Microcontroller (Rev. I)*. Abgerufen am: 31.07.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=msp430f235&fileType=pdf.

[13] Texas Instruments. *MSP430x2xx Family User's Guide*. Abgerufen am: 20.03.2013. http://www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=slau144i&fileType=pdf.

[14] Niels Jegenhorst. *Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation*. Fachhochschule Westküste/HAW Hamburg, Masterthesis, Okt. 2011. .

[15] Karl-Ragmar Riemschneider Matthias Schneider Jürgen Vollmer, Günter Müller. *FUELLING THE CLIMATE 2012 - Klimaschutz und Elektromobilität*. Handelskammer Hamburg 16. Mai 2012, Abgerufen am: 09.06.2013. http://e-mobility-nsr.eu/fileadmin/user_upload/downloads/Fuelling_the_Climate_2012/05_Vollmer_Mueller_Riemschneider_Schneider_2012.pdf.

[16] Rico Loschwitz. *Überwachung-, Zellenbalancierungs- und Leistungselektronik für eine Starterbatterie in Lithium-Eisen-Phosphat-Technologie*. HAW Hamburg, Bachelorthesis, Feb. 2013. .

[17] Karl-Ragmar Riemschneider Matthias Schneider. *Drahtlose Sensoren in den Zellen von Fahrzeug-Batterien*. 21. Internationale Wissenschaftliche Konferenz Mittweida, Abgerufen am: 11.07.2013. www.staff.hs-mittweida.de/%7Edelport/dlslides.php?id=318.

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

[19] Simon Puettjer. *Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellsensoren*. HAW Hamburg, Diplomarbeit, Juni 2011. .

[20] Tobias Steinmann. *Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator*. HAW Hamburg, Masterthesis, Aug. 2012. .

[21] StephanPlaschke. *Experimentalsystem für drahtlose Batteriesensorik*. HAW Hamburg, Diplomarbeit, Juli 2008. .

[22] Wolfgang Weydanz and Andreas Jossen. *Moderne Akkumulatoren richtig einsetzen*. Reichardt Verlag, 1., aufl. edition, 1 2006.

A. Abkürzungsverzeichnis

Abkürzungsverzeichnis

ADC / AD	Analog / Digital Wandler
DC/DC-Wandler	Gleichspannungswandler
DCO	Digitally Controlled Oscillator
FIFO	Fist In First Out
I2C	Inter-Integrated Circuit
ISM	Industrial, Scientific and Medical
LiFePo4	Lithium-Eisenphosphat
LPM4	Low Power Mode 4
OOK	On Off Keying
RAM	Random Access Memory
RFID	Radio-Frequency Identification
SMA	Sub-Miniature-A
SOC	State Of Charge
SOH	State Of Health
SPI	Serial Peripheral Interface
UHF	Ultra-High-Frequency

B. Aufgabenstellung

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. Mai 2013

Bachelorthesis Nico Sassano

Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung

Motivation

Zunehmend werden moderne Lithium-Technologien auch in großen Batterien eingesetzt. Für den optimierten Betrieb und die Erfassung des Lade- und Alterungszustandes ist ein messtechnischer Zugang zu den Zellen erforderlich. Im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Forschungsvorhabens BATSEN (drahtlose Zellensensoren für Fahrzeugbatterien) werden dafür Lösungen untersucht. In dieser Arbeit sollen Grundlagen untersucht werden, ob es auch mit drahtlosen Systemen gelingt, genaue zeitliche Übereinstimmung der Messung der Sensoren und der zentralen Systeme zu erreichen. Die synchronisierte Messung könnte zukünftig die Möglichkeit der Impedanzspektroskopie eröffnen, die ein bewährtes Labormessverfahren in Batterietechnik ist.

Aufgabe

Herr Nico Sassano erhält die Aufgabe, Verbesserungen und Erweiterung für die Zellen-Sensoren zu entwickeln. Als neuer Beitrag zum Projekt soll eine Betriebsfunktion entstehen, die vom zentralen Batteriemanagementsystem zum Sensor Synchronisationstakte als Broadcast im sog. Downlink überträgt. Mit diesen Takten sollen die Spannungsmessungen der Sensoren präzise zeitgleich erfolgen und zwischengespeichert werden. Anschließend sollen die Messwerte als Datenblöcke oder einzeln nacheinander vom Sensor zur Managementeinheit im sog. Uplink übertragen werden. Dort sollen die Spannungswerten mit ebenfalls parallel und synchron erfassten Stromwerten zusammengeführt werden.
Weiterhin ist der Sensor konstruktiv zu modifizieren, damit er am Zellenpol großer Lithium-Rundzellen montiert werden kann. Dazu ist ein neuer Platinenentwurf notwendig.
Die Balancierungsfunktion soll erprobt und die thermische Belastung besser verteilt werden. Die Software des Sensorcontrollers ist entsprechend zu erweitern.
Mit den Sensoren sollten exemplarische Kalibrierungen erfolgen, damit sowohl für die zeitliche Erfassung als auch für die Werteerfassung ausreichende Genauigkeit vorliegt und die Funktionserprobung uneingeschränkt möglich wird.
Für die Abschlussarbeit sind die folgenden Arbeitspakete geplant:

1. Einführung und Analyse der Rahmenbedingungen
   - Einarbeitung in die Projektzielstellung
   - Darstellung der bereits im Projekt erarbeiteten Lösungsvarianten der Zellensensoren
   - Darstellung der bereits vorliegenden Softwarekonzepte

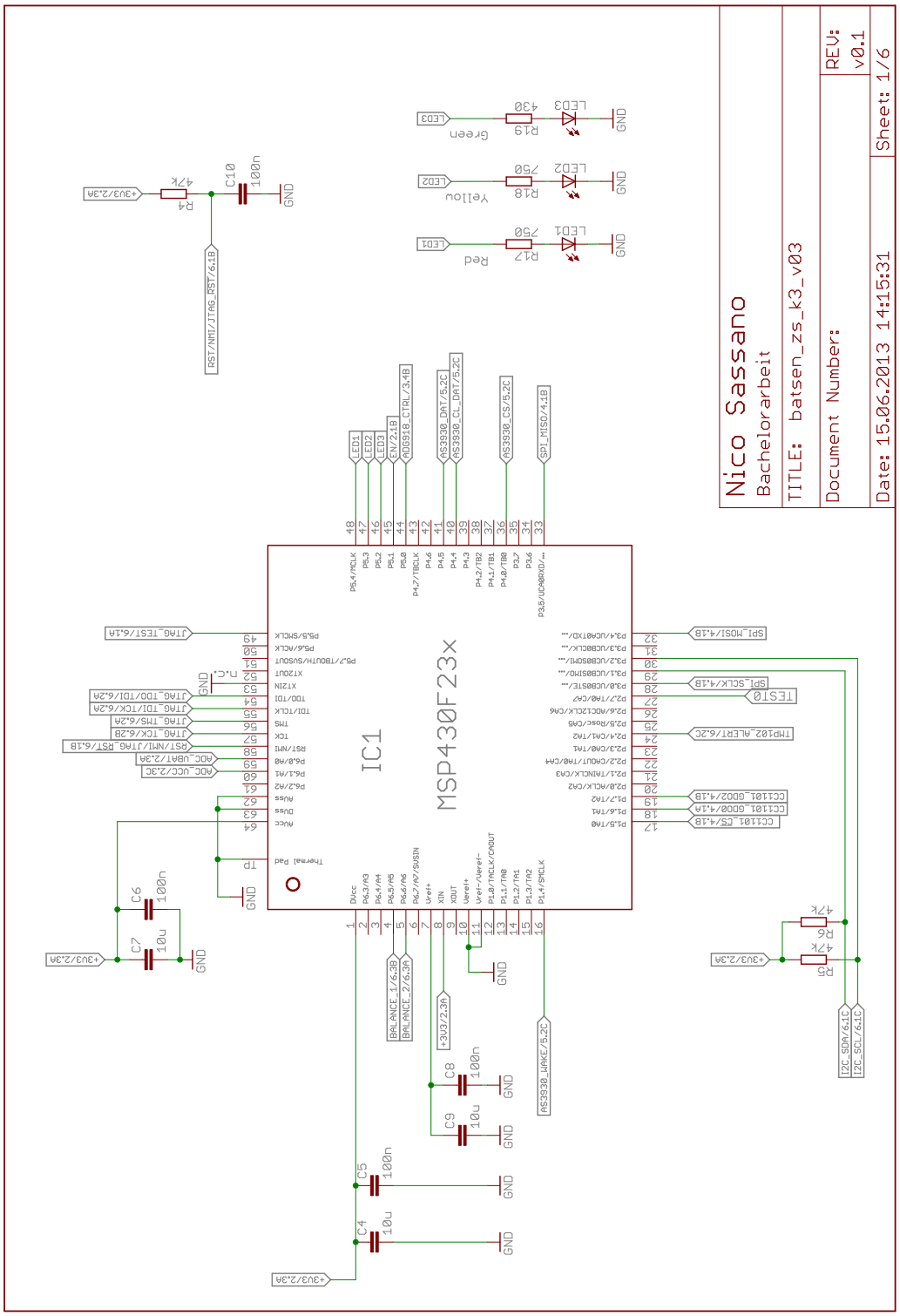
2. Erweiterung der Sensorfunktion und Redesign des Sensors
   - Entwurf und Implementation der Balancierungssteuerung in Software
   - Erarbeiten, Softwareunterstützung und Test der Kalibrierung der Temperatursensoren
   - Einbindung der Alarmfunktion des Temperatursensors
   - Funktionserweiterung vom Broadcast zum adressierbaren WakeUp in Software
   - Modifizierter Platinenentwurf, Aufbau und Inbetriebnahme
3. Neue Sensorfunktion: synchronisierte Burstmessung
   - Analyse und Konzeption für die Burstmessung unter Nutzung geeigneter Betriebsarten des Transceivers (direkte Ansteuerung durch den Controller oder Paketübertragung)
   - Untersuchung der Möglichkeiten für Erfassung der Sensor-DCO-Fehler und deren Korrektur in der Software
   - Entwurf und Implemenation der Zwischenspeicherung und Übertragung
   - Lösungsansätze zur Fehlererkennung bzw. -behandlung in der Burstmessung
4. Praktische Untersuchungen und Erprobung der eweiterten Zellensensoren
   - Erprobung und Ermittlung der Synchronisationsübereinstimmung
   - Ermittlung der Grenzen für die Messgeschwindigkeit und Blockgrößen, auch im Zusammenhang mit der anschließenden Übertragungsdauer
   - Inbetriebnahme und Labortest
   - Erprobung und Demonstration mit synthetischen oder aufgezeichneten Signalen
5. Auswertung und Bewertung
   - Bewertung der Erweiterungen des Sensors, wie Balancierung, Kalibrierung und Alarmfunktion
   - Beurteilung der synchronisierten Burstmessung als neues Messverfahren
   - Diskussion von Vor- und Nachteilen, gelöste und offene Punkte, Ausblick

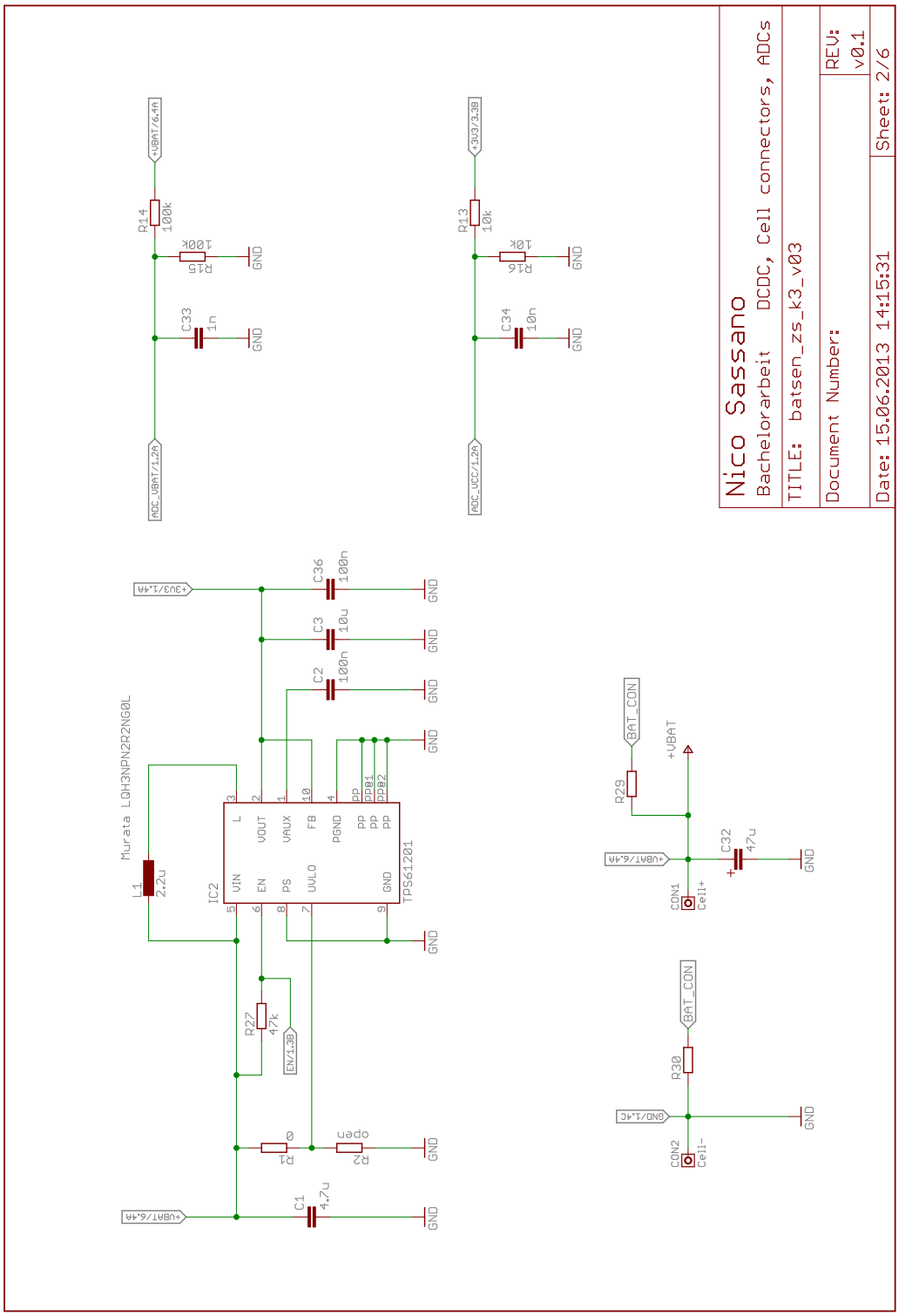
Dokumentation

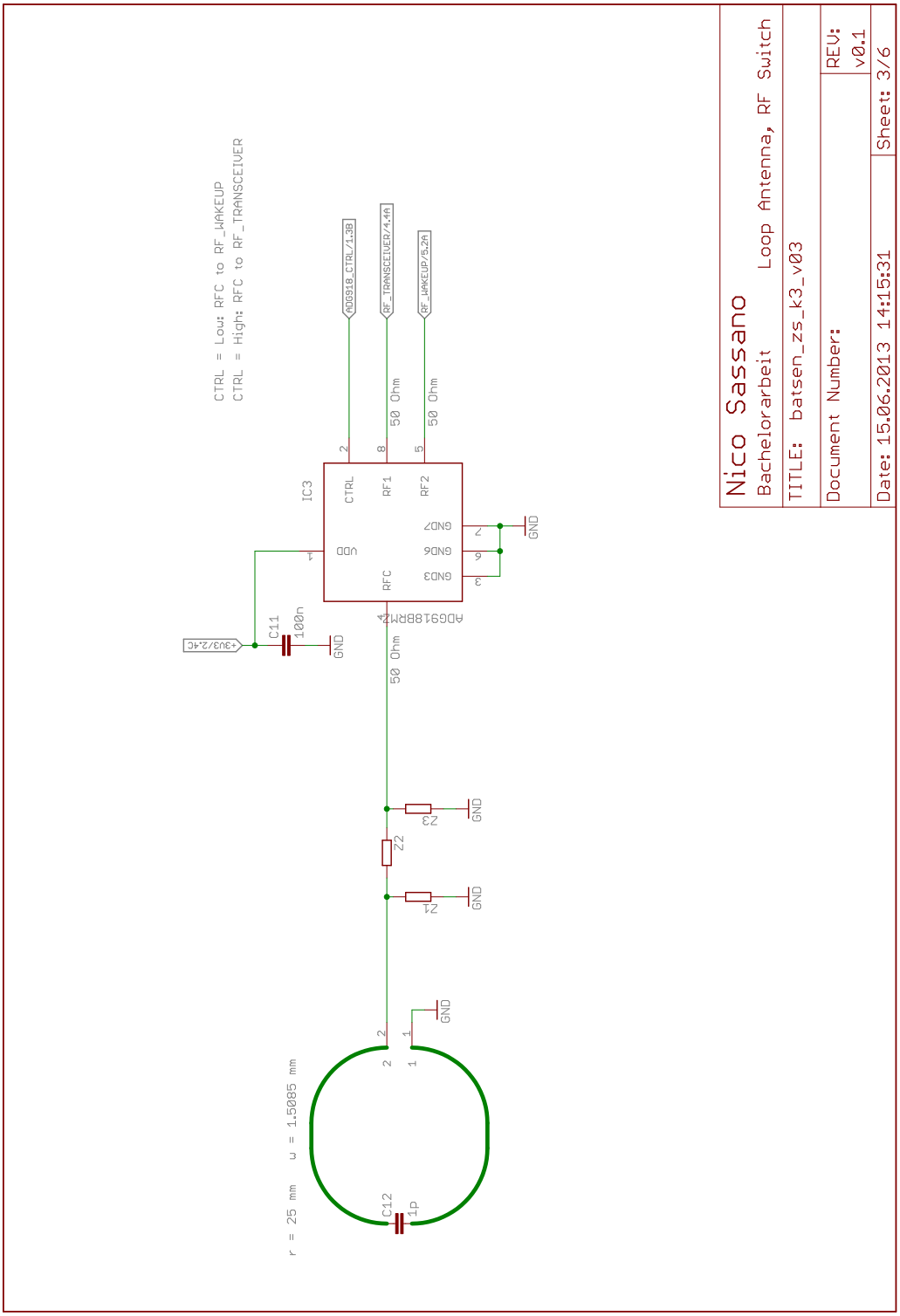
Die Fachliteratur, die Vorarbeiten und die kommerziellen Unterlagen sind zielgerichtet zu recherchieren. Die gesetzten Rahmenbedingungen, gewählte Lösung und die Funktionsweise sind gut nachvollziehbar zu dokumentieren. Die Messergebnisse sind in exemplarischem Umfang zu erfassen und auszuwerten. Die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

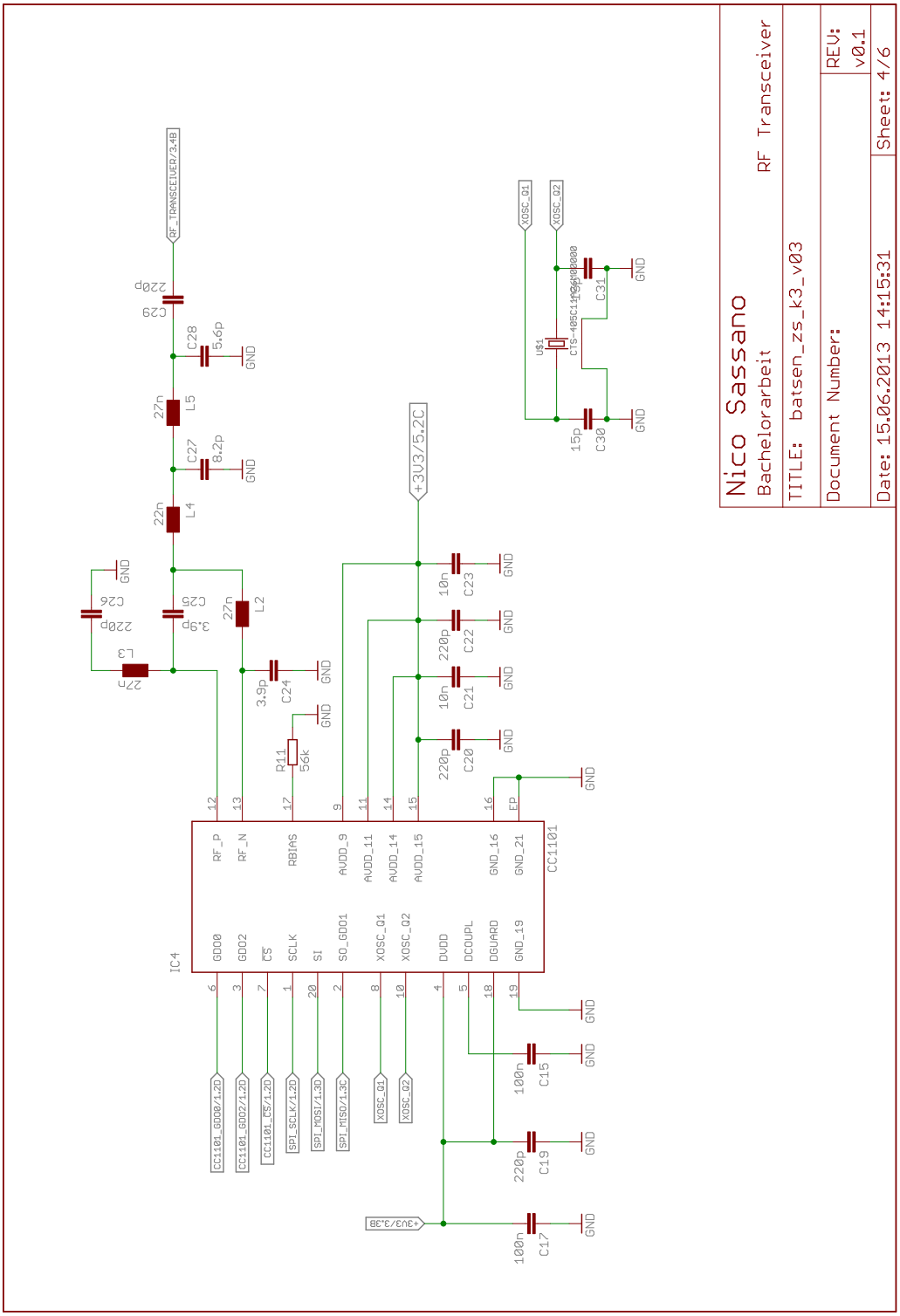
C. Schaltplan

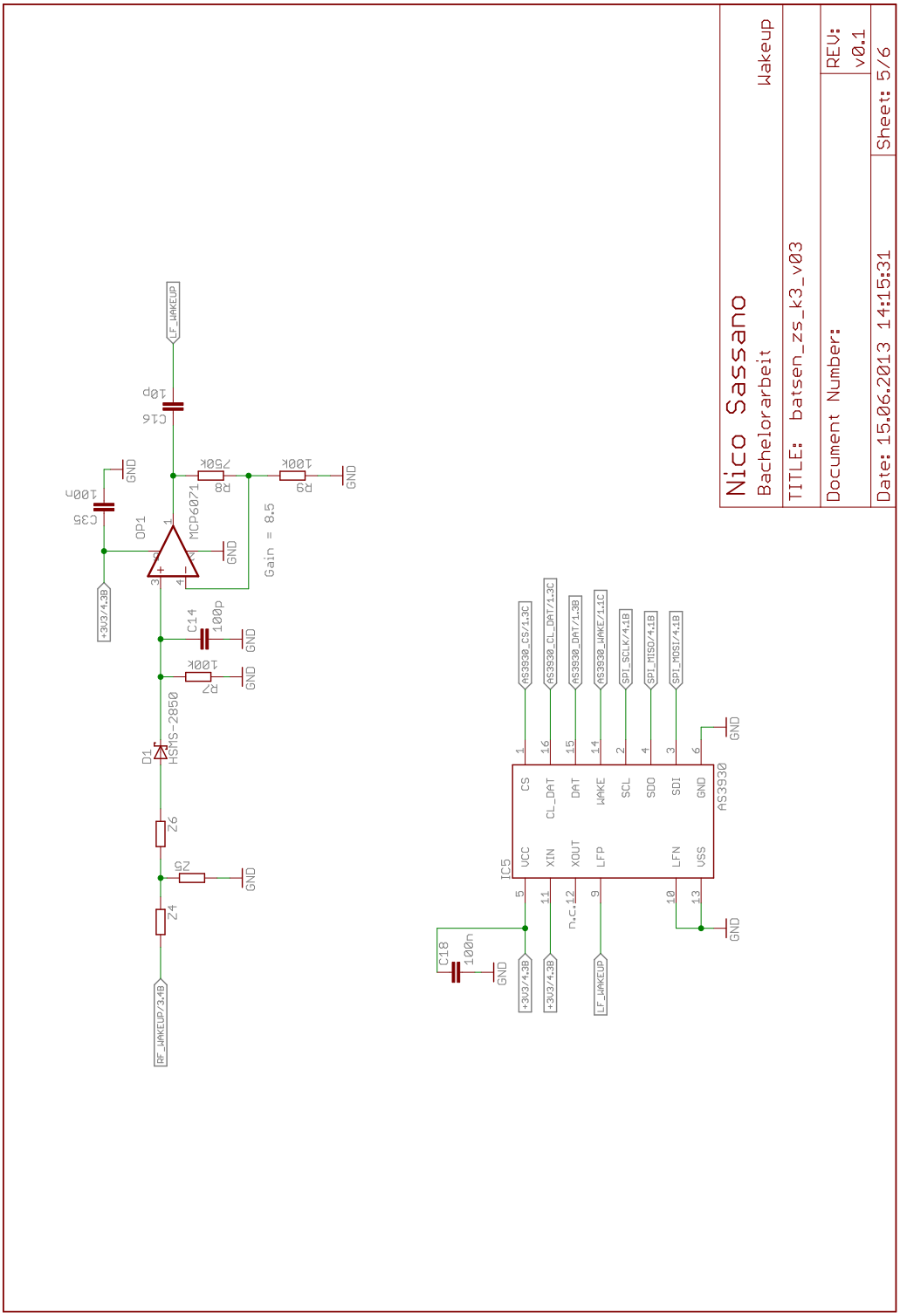
C.1. Zellensensor

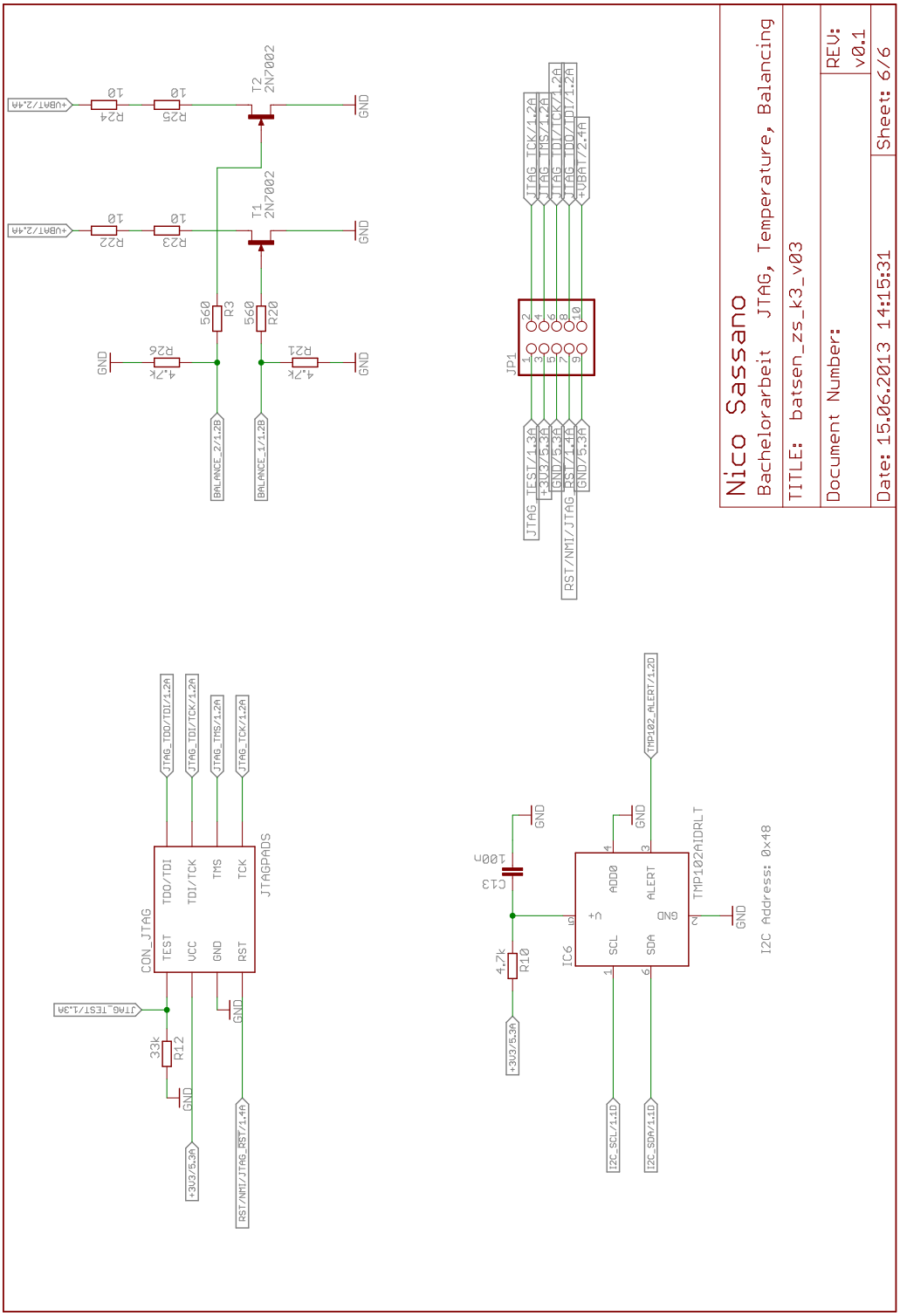

Nico Sassano
Bachelorarbeit
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 1/6
MSP430F23x
IC1
LED1
LED2
LED3
Red
Yellow
Green
R17 750
R18 750
R19 430
R4 47K
C10 100n
RST/NMI/JTAG_RST/6,1B
ADS918_CTRL/3,4B
AS3930_DAT/5,2C
AS3930_CL_DAT/5,2C
AS3930_CS/5,2C
SPI_MISO/4,1B
SPI_MOSI/4,1B
SPI_SCLK/4,1B
TEST0
TMP102_ALERT/6,2C
CC1101_CS/4,1B
CC1101_GDO0/4,1B
CC1101_GDO2/4,1B
JTAG_TEST/6,1A
JTAG_TDO/TDI/6,2A
JTAG_TDI/TCK/6,2A
JTAG_TMS/6,2A
JTAG_TCK/6,2B
RST/NMI/JTAG_RST/6,1B
ADC_UBAT/2,3A
ADC_VCC/2,3C
BALANCE_1/6,3B
BALANCE_2/6,3A
+3U3/2,3A
AS3930_WAKE/5,2C
I2C_SDA/6,1C
I2C_SCL/6,1C
R5 47K
R6 47K
C4 10u
C5 100n
C6 100n
C7 10u
C8 100n
C9 10u
GND


Nico Sassano
Bachelorarbeit DCDC, Cell connectors, ADCs
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 2/6
+VBAT/6.4A
R14 100k
R15 100k
GND
C33 1n
ADC_VBAT/1.2A
+3V3/3.3B
R13 10k
R16 10k
C34 10n
ADC_VCC/1.2A
+3V3/1.4A
C36 100n
C3 10u
C2 100n
Murata LQH3NPN2R2NG0L
L1 2.2u
IC2
TPS61201
VIN
EN
PS
UVLO
L
VOUT
VAUX
FB
PGND
PP
GND
R27 47k
EN/1.3B
R1 0
R2 open
C1 4.7u
BAT_CON
R29
C32 47u
CON1 Cell+
R30
GND/1.4C
CON2 Cell-

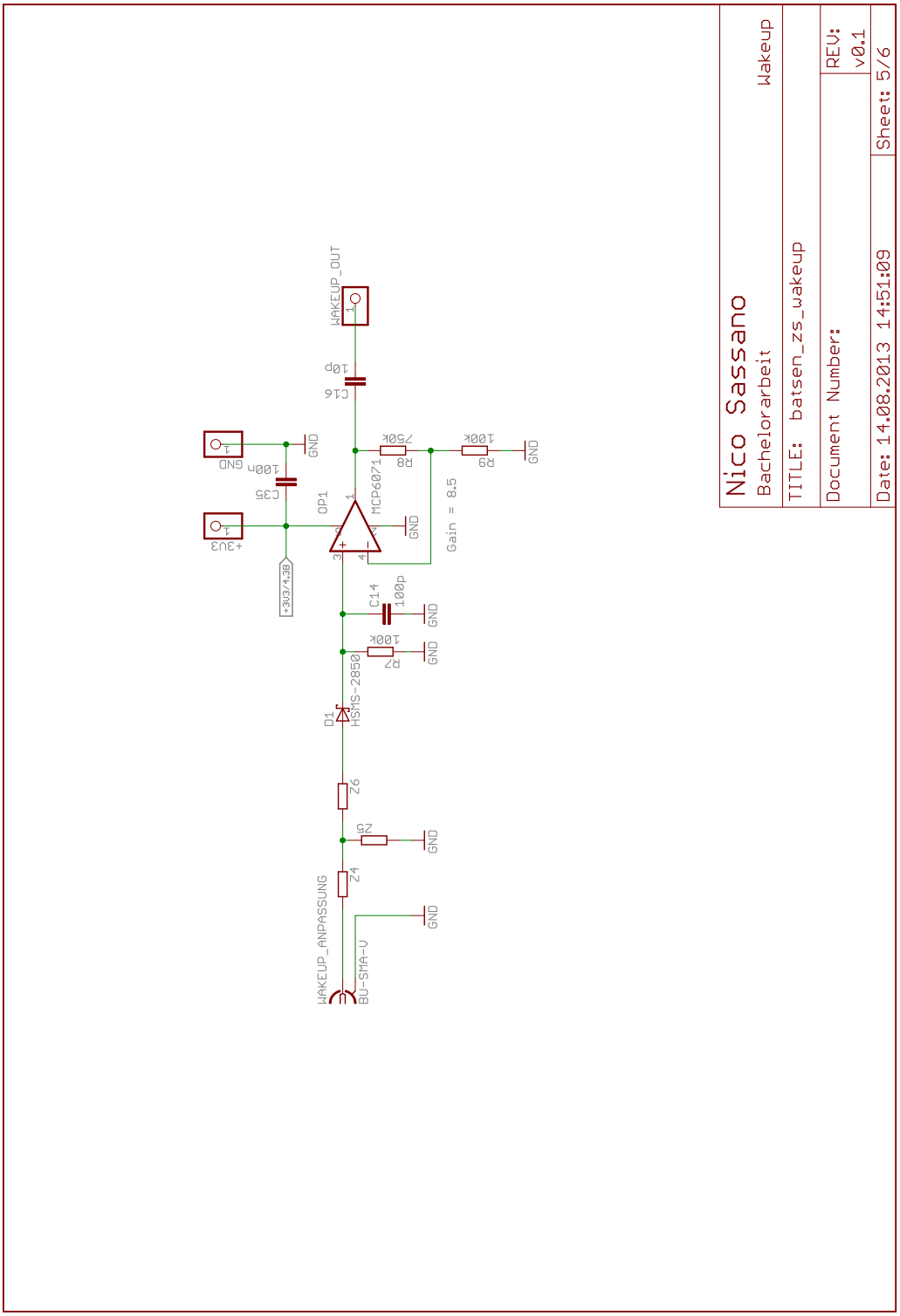

CTRL = Low: RFC to RF_WAKEUP
CTRL = High: RFC to RF_TRANSCEIVER
ADG918_CTRL/1.3B
RF_TRANSCEIVER/4.4A
RF_WAKEUP/5.2A
50 Ohm
50 Ohm
IC3
CTRL
RF1
RF2
VDD
GND7
GND6
GND3
RFC
ADG918BRMZ
+3V3/2.4C
C11
100n
GND
50 Ohm
Z3
Z2
Z1
GND
GND
GND
r = 25 mm w = 1.5085 mm
C12
1p
Nico Sassano
Bachelorarbeit
Loop Antenna, RF Switch
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 3/6

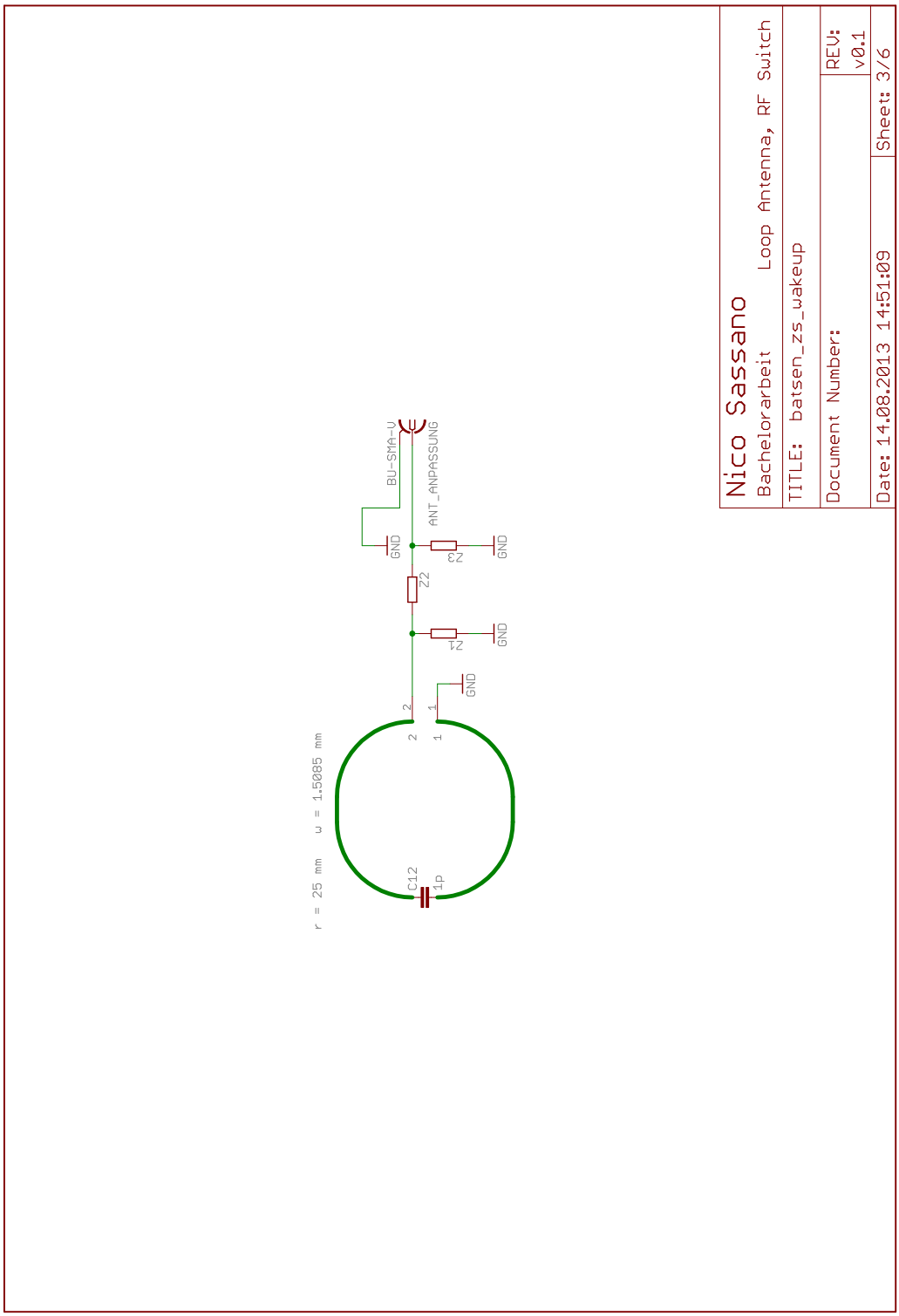

Nico Sassano
Bachelorarbeit
RF Transceiver
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 4/6
CC1101
IC4
RF_TRANSCEIVER/3.4B
+3V3/5.2C
+3V3/3.3B
GND
XOSC_Q1
XOSC_Q2
L2
L3
L4
L5
C15
C17
C19
C20
C21
C22
C23
C24
C25
C26
C27
C28
C29
C30
C31
R11
56k
15p
100n
220p
10n
3.9p
8.2p
5.6p
27n
22n
CC1101_GDO0/1.2D
CC1101_GDO2/1.2D
CC1101_CS/1.2D
SPI_SCLK/1.2D
SPI_MOSI/1.3D
SPI_MISO/1.3C


Nico Sassano
Bachelorarbeit
Wakeup
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 5/6
LF_WAKEUP
C16 10p
R8 750k
R9 100k
C15 100n
OP1 MCP6071
Gain = 8.5
+3V3/4.3B
C14 100p
R7 100k
D1 HSMS-2850
Z6
R6
Z4
RF_WAKEUP/3.4B
IC5 AS3930
CS CL_DAT DAT WAKE SCL SDO SDI GND
VCC XIN XOUT LFP LFN VSS
n.c.
AS3930_CS/1.3C
AS3930_CL_DAT/1.3C
AS3930_DAT/1.3B
AS3930_WAKE/1.1C
SPI_SCLK/4.1B
SPI_MISO/4.1B
SPI_MOSI/4.1B
C18 100n


Nico Sassano
Bachelorarbeit JTAG, Temperature, Balancing
TITLE: batsen_zs_k3_v03
Document Number:
REV: v0.1
Date: 15.06.2013 14:15:31
Sheet: 6/6
T2 2N7002
T1 2N7002
R24 10
R25 10
R22 10
R23 10
R3 560
R20 560
R26 4.7K
R21 4.7K
BALANCE_2/1.2B
BALANCE_1/1.2B
JP1
JTAG_TCK/1.2A
JTAG_TMS/1.2A
JTAG_TDI/TCK/1.2A
JTAG_TDO/TDI/1.2A
+VBAT/2.4A
JTAG_TEST/1.3A
+3V3/5.3A
GND/5.3A
RST/NMI/JTAG_RST/1.4A
CON_JTAG
JTAGPADS
TDO/TDI
TDI/TCK
TMS
TCK
TEST
VCC
GND
RST
R12 33k
IC6
TMP102AIDRLT
ADD0
ALERT
SCL
SDA
C13 100n
R10 4.7k
TMP102_ALERT/1.2D
I2C_SCL/1.1D
I2C_SDA/1.1D
I2C Address: 0x48

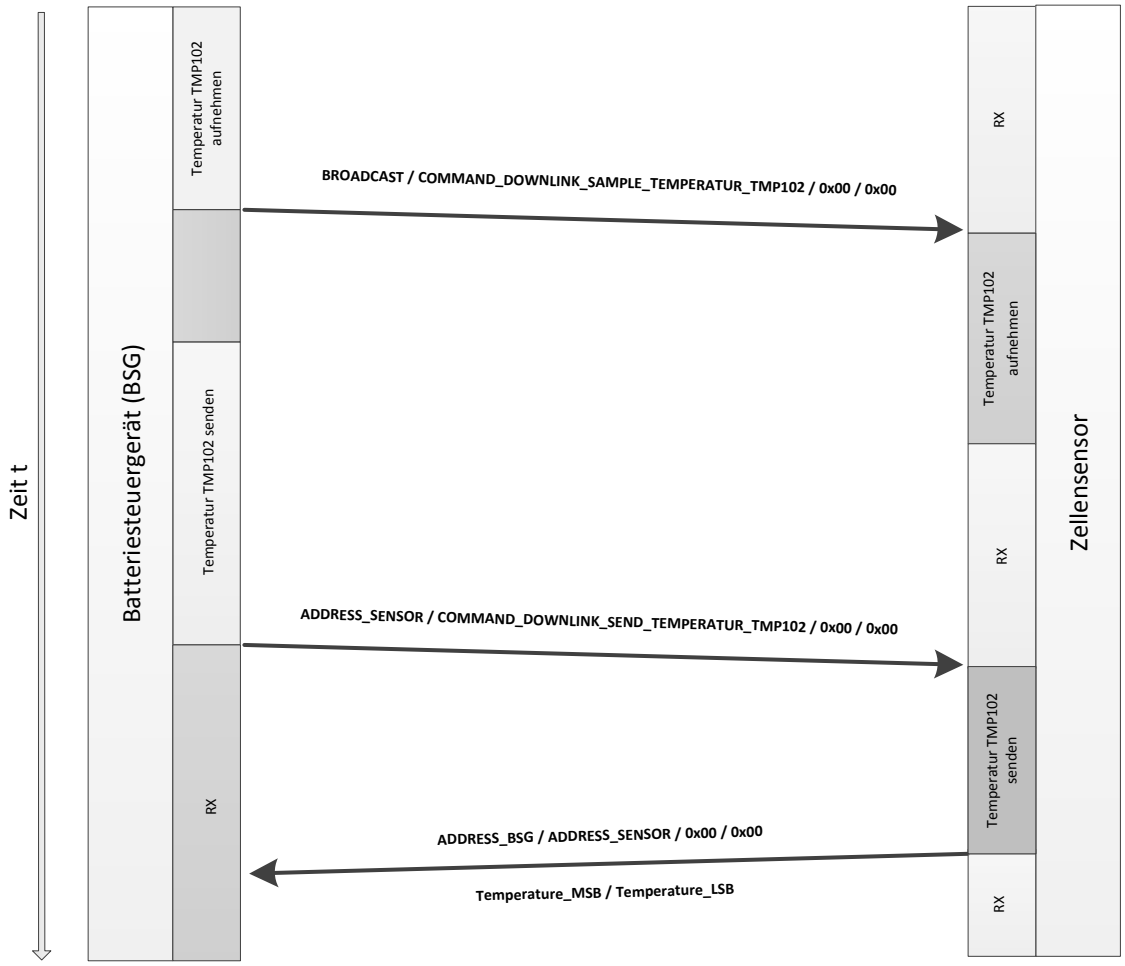
C.2. Messplatine



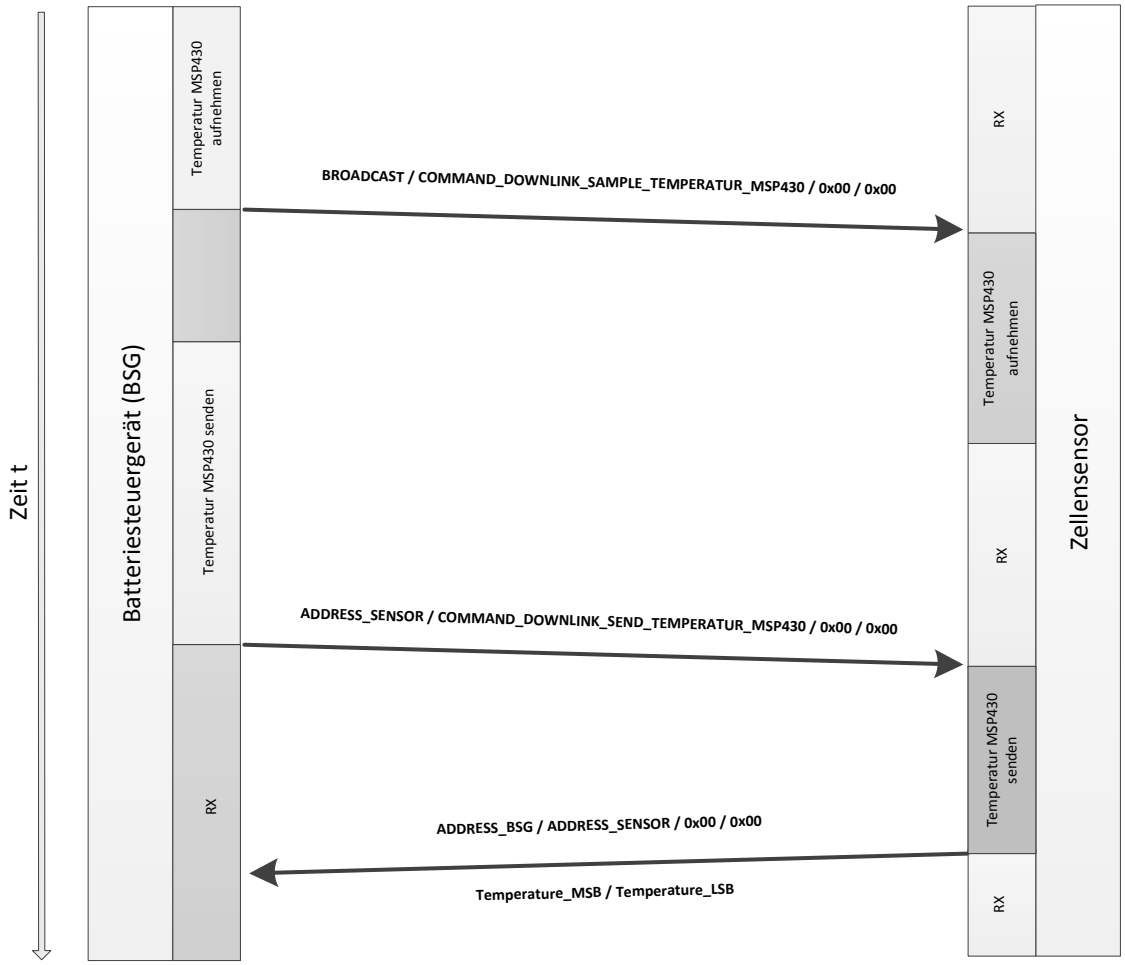

Nico Sassano
Bachelorarbeit
Loop Antenna, RF Switch
TITLE: batsen_zs_wakeup
Document Number:
REV: v0.1
Date: 14.08.2013 14:51:09
Sheet: 3/6
BU-SMA-V
ANT_ANPASSUNG
GND
C2
Z2
N2
GND
GND
GND
2
1
2
1
C12
1p
r = 25 mm w = 1.5085 mm

D. Zeitliche Abläufe der Funktionen

D.1. Ablauf für die Temperaturmessung TMP102



D.2. Ablauf für die Temperaturmessung MSP430



E. Befehlsliste

Befehlsliste

Befehlsnummer	Aktion
0	Alle Sensoren aufwecken und konfigurieren
1	Wake-Up-Signal senden
2	Überprüfen ob Sensoren wach sind
3	Konfiguration vorbereiten
4	Konfiguration senden
5	Spannungswert aufnehmen
6	Spannungswert senden
7	Spannungs- und Temperaturwert aufnehmen
8	Spannungs- und Temperaturwert senden
9	Temperatur (TMP102) aufnehmen
10	Temperatur (TMP102) senden
11	Temperatur (MSP430) aufnehmen
12	Temperatur (MSP430) senden
13	Burtsmessung starten
14	Senden der Burstdaten vorbereiten
15	Senden der Burstdaten
16	Balancierung starten
17	Balancierung stoppen

F. Quellcode

Listings

F.1. Zellensensor

F.1.1. adc12.c

```
1  /*******************************************************************************
2  **    Description:    adc12.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *******************************************************************************/
8
9  #include "main.h"
10
11 void adc12_temp_init(uint8_t);
12 void adc12_disable(void);
13
14 // ADC für Spannungsmessung
15 uint16_t adc12_get_volt_sample(uint8_t clk_set) {
16     uint16_t adc_value = 0;
17
18     TPS61201_DISABLE;               // TPS Enable off
19
20     if(clk_set == 16) { // DCO Takt bei 16 MHz
21
22         delay_10us_16MHz(6);
23
24         TPS61201_ENABLE;     // TPS Enable on
25
26         ADC12CTL0 |= (ENC | ADC12SC);               // Start conversion with sampling
27         while ((ADC12IFG & 0x0001) != 0x0001) ;     // Wait while conversion is active
28
29     } else if(clk_set == 1) {     // DCO Takt bei 1 MHz
30
31         ADC12CTL0 |= (ENC | ADC12SC);               // Start conversion with sampling
32         while ((ADC12IFG & 0x0001) != 0x0001) ;     // Wait while conversion is active
33         TPS61201_ENABLE;     // TPS Enable on
34     }
35
36     adc_value = (ADC12MEM0 & 0x0FFF);
37
38     return adc_value;
39 }
40
41 //ADC für Temperaturmessung
42 uint16_t adc12_get_temp_sample(uint8_t clk_set) {
```

```
43      uint16_t temp_value = 0;
44
45      adc12_temp_init(clk_set);
46
47      ADC12CTL0 |= (ENC | ADC12SC);           // Start conversion with sampling
48      while ((ADC12IFG & 0x0001) != 0x0001) ; // Wait while conversion is active
49      temp_value = (ADC12MEM0 & 0x0FFF);
50
51      adc12_volt_init(clk_set);
52
53      return temp_value;
54  }
55
56  // ADC für Spannungsmessung initialisiern
57  void adc12_volt_init(uint8_t clk_set) {
58      ADC12CTL0 = 0;  // reset
59      ADC12CTL1 = 0;  // reset
60
61      ADC12_VBAT_PxDIR &= ~ADC12_VBAT_PIN;     // ADC pin is input
62      ADC12_VBAT_PxSEL |= ADC12_VBAT_PIN;      // ADC functionality for pin
63
64      /*************************************************************************
65      ** Sampel and Hold muss bei jeder CLK Änderung neu eingestellt werden.
66      ** Die Hold-Zeit muss größer 37.56 us sein
67      *************************************************************************/
68      switch(clk_set) {
69          case 1: {
70              // ADC12CLK = SMCLK / 1
71              ADC12CTL1 &= ~(ADC12DIV2 | ADC12DIV1 | ADC12DIV0);
72              ADC12CTL1 |= (ADC12SSEL1 | ADC12SSEL1);
73
74              ADC12_SHT0_CLK_64;       //(64 * (1/1MHz)) = 64us (64 us > 37.56 us)
75          }break;
76
77          case 8: {
78              // ADC12CLK = SMCLK / 1
79              ADC12CTL1 &= ~(ADC12DIV2 | ADC12DIV1 | ADC12DIV0);
80              ADC12CTL1 |= (ADC12SSEL1 | ADC12SSEL1);
81
82              ADC12_SHT0_CLK_512;      //(512 * (1/8MHz)) = 64us (64 us > 37.56 us)
83          }break;
84
85          case 16: {
86              // ADC12CLK = SMCLK / 4
87              ADC12CTL1 &= ~(ADC12DIV2 | ADC12DIV1 | ADC12DIV0);
88              ADC12CTL1 |=  (ADC12DIV1 | ADC12DIV0);
89              ADC12CTL1 |=  (ADC12SSEL1 | ADC12SSEL1);
90
91              ADC12_SHT0_CLK_256;     //(256 * (4/16MHz)) = 64 us (64 us > 37.56 us)
92
93          }break;
94
95          default: break;
96      }
97
98      ADC12CTL0 |= REF2_5V;    // Interner 2.5V Referenzgenerator nutzen
99      ADC12CTL0 |= REFON;      // Enable reference voltage
100     ADC12CTL0 |= ADC12ON;    // Enable ADC12
101
102     // Save conversion result to ADC12MEM0
103     ADC12CTL1 &= ~(CSTARTADD3 | CSTARTADD2 | CSTARTADD1 | CSTARTADD0);
104
105     ADC12CTL1 &= ~(CONSEQ1 | CONSEQ0);  // Single-channel, single-conversion
106     ADC12CTL1 |= SHP;        // SAMPCON sourced from sampling timer
107
108     // Input channel A0, VR+ = VREF+, VR- = AVss
109     ADC12MCTL0 = SREF_1;
110     ADC12_SET_INPUT_CH0;
111 }
112
113 // ADC für Temperaturmessung initialisiern
114 void adc12_temp_init(uint8_t clk_set) {
115     ADC12CTL0 = 0;  // reset
116     ADC12CTL1 = 0;  // reset
117     ADC12CTL0 = SHT0_4 + REFON + REF2_5V + ADC12ON;     // Internal ref = 1.5V
118     ADC12CTL1 |= SHP;                           // enable sample timer
119     ADC12MCTL0 = SREF_1 + INCH_10;              // ADC i/p ch A10 = temp sense i/p
120
121     ADC12CTL0 |= ENC;
122 }
123
124 // ADC ausschalten
125 void adc12_disable(void) {
126     ADC12CTL0 &= ~ENC;       // Disable conversion
127     ADC12CTL0 &= ~REFON;     // Disable reference voltage
```

```
128         ADC12CTL0 &= ~ADC12ON;     // Disable ADC12
129     }
```

F.1.2. adc12.h

```
1   /*******************************************************************************
2   **     Description:     adc12.h
3   **     Hardware:        BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
4   **     Date:            06/03/2013
5   **     Author:          Nico Sassano
6   **                      modified taken from Phillip Durdaut
7   *******************************************************************************/
8
9   #ifndef ADC12_H_
10  #define ADC12_H_
11
12  #include "main.h"
13
14  #define ADC12_VBAT_PxSEL            P6SEL
15  #define ADC12_VBAT_PxDIR            P6DIR
16  #define ADC12_VBAT_PIN              BIT0
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 &=~(SHT03 | SHT02 | SHT01 | SHT00))
25  #define ADC12_SHT1_CLK_8            (ADC12CTL0 &=~(SHT03 | SHT02 | SHT01)); \
26                                      (ADC12CTL0 |= (SHT00))
27  #define ADC12_SHT1_CLK_16           (ADC12CTL0 &=~(SHT03 | SHT02 | SHT00)); \
28                                      (ADC12CTL0 |= (SHT01))
29  #define ADC12_SHT1_CLK_32           (ADC12CTL0 &=~(SHT03 | SHT02)); \
30                                      (ADC12CTL0 |= (SHT01 | SHT00))
31  #define ADC12_SHT1_CLK_64           (ADC12CTL0 &=~(SHT03 | SHT01 | SHT00)); \
32                                      (ADC12CTL0 |= (SHT02))
33  #define ADC12_SHT1_CLK_96           (ADC12CTL0 &=~(SHT03 | SHT01)); \
34                                      (ADC12CTL0 |= (SHT02 | SHT00))
35  #define ADC12_SHT1_CLK_128          (ADC12CTL0 &=~(SHT03 | SHT00)); \
36                                      (ADC12CTL0 |= (SHT02 | SHT01))
37  #define ADC12_SHT1_CLK_192          (ADC12CTL0 &=~(SHT03)); \
38                                      (ADC12CTL0 |= (SHT02 | SHT01 | SHT00))
39  #define ADC12_SHT1_CLK_256          (ADC12CTL0 &=~(SHT02 | SHT01 | SHT00)); \
40                                      (ADC12CTL0 |= (SHT03))
41  #define ADC12_SHT1_CLK_384          (ADC12CTL0 &=~(SHT02 | SHT01)); \
42                                      (ADC12CTL0 |= (SHT03 | SHT00))
43  #define ADC12_SHT1_CLK_512          (ADC12CTL0 &=~(SHT02 | SHT00)); \
44                                      (ADC12CTL0 |= (SHT03 | SHT01))
45  #define ADC12_SHT1_CLK_768          (ADC12CTL0 &=~(SHT02)); \
46                                      (ADC12CTL0 |= (SHT03 | SHT01 |SHT00))
47  #define ADC12_SHT1_CLK_1024         (ADC12CTL0 &=~(SHT01 | SHT00)); \
48                                      (ADC12CTL0 |= (SHT03 | SHT02))
49
50  /*****************************************************************
51  ** Sample–and–hold time for registers ADC12MEM0 to ADC12MEM7
52  *****************************************************************/
53  #define ADC12_SHT0_CLK_4            (ADC12CTL0 &=~(SHT03 | SHT02 | SHT01 | SHT00))
54  #define ADC12_SHT0_CLK_8            (ADC12CTL0 &=~(SHT03 | SHT02 | SHT01)); \
55                                      (ADC12CTL0 |= (SHT00))
56  #define ADC12_SHT0_CLK_16           (ADC12CTL0 &=~(SHT03 | SHT02 | SHT00)); \
57                                      (ADC12CTL0 |= (SHT01))
58  #define ADC12_SHT0_CLK_32           (ADC12CTL0 &=~(SHT03 | SHT02)); \
59                                      (ADC12CTL0 |= (SHT01 | SHT00))
60  #define ADC12_SHT0_CLK_64           (ADC12CTL0 &=~(SHT03 | SHT01 | SHT00)); \
61                                      (ADC12CTL0 |= (SHT02))
62  #define ADC12_SHT0_CLK_96           (ADC12CTL0 &=~(SHT03 | SHT01)); \
63                                      (ADC12CTL0 |= (SHT02 | SHT00))
64  #define ADC12_SHT0_CLK_128          (ADC12CTL0 &=~(SHT03 | SHT00)); \
65                                      (ADC12CTL0 |= (SHT02 | SHT01))
66  #define ADC12_SHT0_CLK_192          (ADC12CTL0 &=~(SHT03)); \
67                                      (ADC12CTL0 |= (SHT02 | SHT01 | SHT00))
68  #define ADC12_SHT0_CLK_256          (ADC12CTL0 &=~(SHT02 | SHT01 | SHT00)); \
69                                      (ADC12CTL0 |= (SHT03))
70  #define ADC12_SHT0_CLK_384          (ADC12CTL0 &=~(SHT02 | SHT01)); \
71                                      (ADC12CTL0 |= (SHT03 | SHT00))
72  #define ADC12_SHT0_CLK_512          (ADC12CTL0 &=~(SHT02 | SHT00)); \
73                                      (ADC12CTL0 |= (SHT03 | SHT01))
74  #define ADC12_SHT0_CLK_768          (ADC12CTL0 &=~(SHT02)); \
75                                      (ADC12CTL0 |= (SHT03 | SHT01 |SHT00))
76  #define ADC12_SHT0_CLK_1024         (ADC12CTL0 &=~(SHT01 | SHT00)); \
77                                      (ADC12CTL0 |= (SHT03 | SHT02))
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 #define ADC12_SET_INPUT_CH0         (ADC12MCTL0 |= INCH_0)
207 #define ADC12_SET_INPUT_CH1         (ADC12MCTL0 |= INCH_1)
208 #define ADC12_SET_INPUT_CH2         (ADC12MCTL0 |= INCH_2)
209 #define ADC12_SET_INPUT_CH3         (ADC12MCTL0 |= INCH_3)
210 #define ADC12_SET_INPUT_CH4         (ADC12MCTL0 |= INCH_4)
211 #define ADC12_SET_INPUT_CH5         (ADC12MCTL0 |= INCH_5)
212 #define ADC12_SET_INPUT_CH6         (ADC12MCTL0 |= INCH_6)
213 #define ADC12_SET_INPUT_CH7         (ADC12MCTL0 |= INCH_7)
214 #define ADC12_SET_TEMPERATUR        (ADC12MCTL0 |= INCH_10)
215
216
217 uint16_t adc12_get_volt_sample(uint8_t);
218 uint16_t adc12_get_temp_sample(uint8_t);
219 void adc12_init(void);
220 void adc12_volt_init(uint8_t);
221
222 #endif /* ADC12_H_ */
```

F.1.3. adg918.c

```
1  /*-------------------------------------------------------------------------
2      Description:    ADG918 RF-switch driver.
3      Date:           11/22/2012
4      Last Update:    11/22/2012
5      Author:         Phillip Durdaut
6  -------------------------------------------------------------------------*/
7
8  #include "main.h"
9
10 /*-------------------------------------------------------------------------
11   Public functions
12 -------------------------------------------------------------------------*/
13
14 void adg918_init(void)
15 {
16     ADG918_CTRL_PxDIR |= ADG918_CTRL_PIN;
17 }
18
19 void adg918_wakeup(void)
20 {
21     ADG918_CTRL_PxOUT &= ~ADG918_CTRL_PIN;
22 }
23
24 void adg918_transceiver(void)
25 {
26     ADG918_CTRL_PxOUT |= ADG918_CTRL_PIN;
27 }
```

F.1.4. adg918.h

```
1  /*-------------------------------------------------------------------
2      Description:    ADG918 RF-switch driver.
3      Date:           11/22/2012
4      Last Update:    11/22/2012
5      Author:         Phillip Durdaut
6  -------------------------------------------------------------------*/
7
8  #ifndef ADG918_H_
9  #define ADG918_H_
10
11 #include "main.h"
12
13 /*-------------------------------------------------------------------
14   Defines
15 -------------------------------------------------------------------*/
16
17 #define ADG918_CTRL_PxDIR      (P5DIR)
18 #define ADG918_CTRL_PxOUT      (P5OUT)
19 #define ADG918_CTRL_PIN        (BIT0)
20
21 /*-------------------------------------------------------------------
22   Public functions
23 -------------------------------------------------------------------*/
24
25 void adg918_init(void);
26 void adg918_wakeup(void);
27 void adg918_transceiver(void);
28
29 #endif /* ADG918_H_ */
```

F.1.5. as3930.c

```
/*-----------------------------------------------------------------------
    Description:    Driver for austriamicrosystems AS3930 Single Channel
                    Low Frequency Wakeup Receiver.
    Date:           10/02/2012
    Last Update:    10/03/2012
    Author:         Phillip Durdaut
-----------------------------------------------------------------------*/

#include "main.h"

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

/* Configuration Modes (bits 15-14) */

#define AS3930_WRITE                0x00
#define AS3930_READ                 0x01
#define AS3930_COMMAND              0x03

/* Configuration Registers (bits 13-8) */

#define AS3930_R00                  0x00
#define AS3930_R01                  0x01
#define AS3930_R02                  0x02
#define AS3930_R03                  0x03
#define AS3930_R04                  0x04
#define AS3930_R05                  0x05
#define AS3930_R06                  0x06
#define AS3930_R07                  0x07
#define AS3930_R08                  0x08
#define AS3930_R09                  0x09
#define AS3930_R10                  0x0a
#define AS3930_R11                  0x0b
#define AS3930_R12                  0x0c
#define AS3930_R13                  0x0d

/* Commands (bits 7-0) */

#define AS3930_CLEAR_WAKE           0x00
#define AS3930_RESET_RSSI           0x01
#define AS3930_TRIM_OSC             0x02
#define AS3930_CLEAR_FALSE          0x03
#define AS3930_PRESET_DEFAULT       0x04

/*-----------------------------------------------------------------------
  Prototypes of the private functions
-----------------------------------------------------------------------*/

void as3930_spi_setup(void);
void as3930_spi_write_register(u8_t address, u8_t value);
char as3930_spi_read_register(u8_t address);
void as3930_spi_command(u8_t command);

/*-----------------------------------------------------------------------
  Public functions
-----------------------------------------------------------------------*/

void as3930_init(void)
{
    as3930_spi_setup();

    AS3930_WAKE_DIR_IN;
    AS3930_WAKE_IRQ_RISING_EDGE;
    AS3930_WAKE_IRQ_DISABLE;
    AS3930_WAKE_CLEAR_IRQ;
}

void as3930_config_no_pattern(void)
{
    as3930_spi_write_register(AS3930_R01, BIT5);  // Data correlation disable, Crystal oscillator disable
    //as3930_spi_write_register(AS3930_R07, BIT5);  // Automatic time-out after 50 ms
}

void as3930_preset_default(void)
{
    as3930_spi_command(AS3930_PRESET_DEFAULT);
}

void as3930_clear_wakeup(void) {
    as3930_spi_command(AS3930_CLEAR_WAKE);
}

```

```
84  u8_t as3930_get_rssi(void)
85  {
86      return (as3930_spi_read_register(AS3930_R10) & 0x1F);
87  }
88
89  /*---------------------------------------------------------------------------
90    Private functions
91  ---------------------------------------------------------------------------*/
92
93  void as3930_spi_setup(void)
94  {
95      AS3930_CS_PxDIR |= AS3930_CS_PIN;    // CS is output
96      AS3930_CS_PxOUT &= ~AS3930_CS_PIN;   // Chip disable
97
98      UCA0CTL1 |= UCSWRST;                 // Hold state machine in reset
99
100     UCA0CTL0 &= ~UCCKPH;                     // Data is changed on rising edge
101     UCA0CTL0 &= ~UCCKPL;                     // Inactive clock is low
102     UCA0CTL0 |= (UCMST | UCMSB | UCSYNC);    // MSB first, Master mode, Synchronous mode
103     UCA0CTL0 &= ~(UCMODE1 | UCMODE0);        // 3-pin SPI
104     UCA0MCTL = 0;                            // No modulation
105
106     // SMCLK / 2
107     UCA0CTL1 |= (UCSSEL1 | UCSSEL0);
108     UCA0BR0 = 2;
109     UCA0BR1 = 0;
110
111     // SPI functionality for pins
112     AS3930_SPI_PxSEL |= (AS3930_SPI_MOSI_PIN | AS3930_SPI_MISO_PIN | AS3930_SPI_CLK_PIN);
113     AS3930_SPI_PxDIR |= (AS3930_SPI_MOSI_PIN | AS3930_SPI_CLK_PIN); // MOSI and CLK are outputs
114     AS3930_SPI_PxDIR &= ~AS3930_SPI_MISO_PIN; // MISO is input
115
116     UCA0CTL1 &= ~UCSWRST;    // Initialize USART state machine
117 }
118
119 void as3930_spi_write_register(u8_t address, u8_t value)
120 {
121     AS3930_CS_PxOUT |= AS3930_CS_PIN;           // Chip enable
122     while (UCA0STAT & UCBUSY);                  // Wait for TX to finish
123     UCA0TXBUF = address | (AS3930_WRITE << 6);  // Send configuration mode and register
124     while (UCA0STAT & UCBUSY);                  // Wait for TX to finish
125     UCA0TXBUF = value;                          // Send value
126     while (UCA0STAT & UCBUSY);                  // Wait for TX complete
127     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;          // Chip disable
128 }
129
130 char as3930_spi_read_register(u8_t address)
131 {
132     u8_t value;
133
134     AS3930_CS_PxOUT |= AS3930_CS_PIN;           // Chip enable
135     while (UCA0STAT & UCBUSY);                  // Wait for TX to finish
136     UCA0TXBUF = address | (AS3930_READ << 6);   // Send configuration mode and register
137     while (UCA0STAT & UCBUSY);                  // Wait for TX to finish
138     UCA0TXBUF = 0;                              // Dummy write so we can read data
139     while (UCA0STAT & UCBUSY);                  // Wait for TX complete
140     value = UCA0RXBUF;                          // Read data
141     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;          // Chip disable
142
143     return value;
144 }
145
146 void as3930_spi_command(u8_t command)
147 {
148     AS3930_CS_PxOUT |= AS3930_CS_PIN;               // Chip enable
149     while (UCA0STAT & UCBUSY);                      // Wait for TX to finish
150     UCA0TXBUF = command | (AS3930_COMMAND << 6);     // Send configuration mode and register
151     while (UCA0STAT & UCBUSY);                      // Wait for TX to finish
152     AS3930_CS_PxOUT &= ~AS3930_CS_PIN;              // Chip disable
153 }
```

F.1.6. as3930.h

```
1  /*-------------------------------------------------------------------
2      Description:    Driver for austriamicrosystems AS3930 Single Channel
3                      Low Frequency Wakeup Receiver.
4      Date:           10/09/2012
5      Last Update:    10/09/2012
6      Author:         Phillip Durdaut
7  -------------------------------------------------------------------*/
8
9  #ifndef AS3930_H_
10 #define AS3930_H_
11
12 #include "main.h"
13
14 /*-------------------------------------------------------------------
15   Defines -> Need to be changed depending on hardware
16   -------------------------------------------------------------------*/
17
18 #define AS3930_CS_PxDIR              P4DIR
19 #define AS3930_CS_PxOUT              P4OUT
20 #define AS3930_CS_PIN                BIT0
21
22 #define AS3930_SPI_PxSEL             P3SEL
23 #define AS3930_SPI_PxDIR             P3DIR
24 #define AS3930_SPI_PxIN              P3IN
25 #define AS3930_SPI_MOSI_PIN          BIT4
26 #define AS3930_SPI_MISO_PIN          BIT5
27 #define AS3930_SPI_CLK_PIN           BIT0
28
29 #define AS3930_WAKE_PxDIR            P1DIR
30 #define AS3930_WAKE_PxIES            P1IES
31 #define AS3930_WAKE_PxIE             P1IE
32 #define AS3930_WAKE_PxIFG            P1IFG
33 #define AS3930_WAKE_PIN              BIT4
34
35 /*-------------------------------------------------------------------
36   Macros
37 -------------------------------------------------------------------*/
38
39 #define AS3930_WAKE_DIR_IN           (AS3930_WAKE_PxDIR &= ~AS3930_WAKE_PIN)
40 #define AS3930_WAKE_IRQ_RISING_EDGE  (AS3930_WAKE_PxIES &= ~AS3930_WAKE_PIN)
41 #define AS3930_WAKE_IRQ_FALLING_EDGE (AS3930_WAKE_PxIES |= AS3930_WAKE_PIN)
42 #define AS3930_WAKE_IRQ_ENABLE       (AS3930_WAKE_PxIE |= AS3930_WAKE_PIN)
43 #define AS3930_WAKE_IRQ_DISABLE      (AS3930_WAKE_PxIE &= ~AS3930_WAKE_PIN)
44 #define AS3930_WAKE_IRQ_PENDING      ((AS3930_WAKE_PxIFG & AS3930_WAKE_PIN) == AS3930_WAKE_PIN)
45 #define AS3930_WAKE_CLEAR_IRQ        (AS3930_WAKE_PxIFG &= ~(AS3930_WAKE_PIN))
46
47 /*-------------------------------------------------------------------
48   Public functions
49 -------------------------------------------------------------------*/
50
51 void as3930_init(void);
52 void as3930_config_no_pattern(void);
53 void as3930_preset_default(void);
54 void as3930_clear_wakeup(void);
55 u8_t as3930_get_rssi(void);
56
57 #endif /* AS3930_H_ */
```

F.1.7. balancing.c

```
1  /*****************************************************************************
2  **    Description:    balancing.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *****************************************************************************/
8
9  #include "main.h"
10
11 void balancing_init(void) { // Ports initialisieren
12     BALANCE_PORT_1_PxDIR |= BALANCE_PORT_1_PIN;
13     BALANCE_PORT_2_PxDIR |= BALANCE_PORT_2_PIN;
14 }
15
16 void balancing_on(void) {   // Balancierung an
17     BALANCE_PORT_1_PxOUT |= BALANCE_PORT_1_PIN;
18     BALANCE_PORT_2_PxOUT |= BALANCE_PORT_2_PIN;
19 }
20
21 void balancing_port_1_on(void) {    // Pfad 1 anschalten
22     BALANCE_PORT_1_PxOUT |= BALANCE_PORT_1_PIN;
23 }
24
25 void balancing_port_2_on(void) {    // Pfad 2 anschalten
26     BALANCE_PORT_2_PxOUT |= BALANCE_PORT_2_PIN;
27 }
28
29 void balancing_off(void) {  // Balancierung ausschalten
30     BALANCE_PORT_1_PxOUT &=~BALANCE_PORT_1_PIN;
31     BALANCE_PORT_2_PxOUT &=~BALANCE_PORT_2_PIN;
32 }
33
34 void balancing_port_1_off(void) {   // Pfad 1 ausschalten
35     BALANCE_PORT_1_PxOUT &=~BALANCE_PORT_1_PIN;
36 }
37
38 void balancing_port_2_off(void) {   // Pfad 2 ausschalten
39     BALANCE_PORT_2_PxOUT &=~BALANCE_PORT_2_PIN;
40 }
```

F.1.8. balancing.h

```
1  /*******************************************************************************
2  **    Description:    balancing.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *******************************************************************************/
8
9  #ifndef BALANCING_H_
10 #define BALANCING_H_
11
12 #include "main.h"
13
14 /*******************************************************************************
15 **  Defines
16 *******************************************************************************/
17
18 #define BALANCE_PORT_1_PxDIR      (P6DIR)
19 #define BALANCE_PORT_1_PxOUT      (P6OUT)
20 #define BALANCE_PORT_1_PIN        (BIT5)
21
22 #define BALANCE_PORT_2_PxDIR      (P6DIR)
23 #define BALANCE_PORT_2_PxOUT      (P6OUT)
24 #define BALANCE_PORT_2_PIN        (BIT6)
25
26 void balancing_init(void);
27 void balancing_on(void);
28 void balancing_off(void);
29
30 #endif /* BALANCING_H_ */
```

F.1.9. cc1101.c

```
1   /*****************************************************************************
2   **    Description:    cc1101.c
3   **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **    Date:           06/03/2013
5   **    Author:         Nico Sassano
6   **                    modified taken from Phillip Durdaut
7   *****************************************************************************/
8
9   #include "main.h"
10
11  /*****************************************************************************
12  **  Variables
13   *****************************************************************************/
14
15  /* Output power table (433 MHz): -30, 10 (dBm) */
16  //uint8_t cc1101_patable[] = { 0x12, 0xc0 };
17  uint8_t cc1101_patable[] = { 0x12, 0xc0 };
18  uint8_t cc1101_patable_a[] = { 0xc0 };
19  uint8_t cc1101_patablelen = 2;
20
21  /*****************************************************************************
22  **  Defines
23  *****************************************************************************/
24
25  /* Command Strobes (Table 42 in datasheet) */
26  #define CC1101_CS_SRES              0x30
27  #define CC1101_CS_SFSTXON           0x31
28  #define CC1101_CS_SXOFF             0x32
29  #define CC1101_CS_SCAL              0x33
30  #define CC1101_CS_SRX               0x34
31  #define CC1101_CS_STX               0x35
32  #define CC1101_CS_SIDLE             0x36
33  #define CC1101_CS_SAFC              0x37
34  #define CC1101_CS_SWOR              0x38
35  #define CC1101_CS_SPWD              0x39
36  #define CC1101_CS_SFRX              0x3A
37  #define CC1101_CS_SFTX              0x3B
38  #define CC1101_CS_SWORRST           0x3C
39  #define CC1101_CS_SNOP              0x3D
40
41  /* Configuration Registers */
42  #define CC1101_CR_IOCFG2            0x00    /* GDO2 output pin configuration */
43  #define CC1101_CR_IOCFG1            0x01    /* GDO1 output pin configuration */
44  #define CC1101_CR_IOCFG0            0x02    /* GDO0 output pin configuration */
45  #define CC1101_CR_FIFOTHR           0x03    /* RX FIFO and TX FIFO thresholds */
46  #define CC1101_CR_SYNC1             0x04    /* Sync word, high byte */
47  #define CC1101_CR_SYNC0             0x05    /* Sync word, low byte */
48  #define CC1101_CR_PKTLEN            0x06    /* Packet length */
49  #define CC1101_CR_PKTCTRL1          0x07    /* Packet automation control */
50  #define CC1101_CR_PKTCTRL0          0x08    /* Packet automation control */
51  #define CC1101_CR_ADDR              0x09    /* Device address */
52  #define CC1101_CR_CHANNR            0x0A    /* Channel number */
53  #define CC1101_CR_FSCTRL1           0x0B    /* Frequency synthesizer control */
54  #define CC1101_CR_FSCTRL0           0x0C    /* Frequency synthesizer control */
55  #define CC1101_CR_FREQ2             0x0D    /* Frequency control word, high byte */
56  #define CC1101_CR_FREQ1             0x0E    /* Frequency control word, middle byte */
57  #define CC1101_CR_FREQ0             0x0F    /* Frequency control word, low byte */
58  #define CC1101_CR_MDMCFG4           0x10    /* Modem configuration */
59  #define CC1101_CR_MDMCFG3           0x11    /* Modem configuration */
60  #define CC1101_CR_MDMCFG2           0x12    /* Modem configuration */
61  #define CC1101_CR_MDMCFG1           0x13    /* Modem configuration */
62  #define CC1101_CR_MDMCFG0           0x14    /* Modem configuration */
63  #define CC1101_CR_DEVIATN           0x15    /* Modem deviation setting */
64  #define CC1101_CR_MCSM2             0x16    /* Main Radio Cntrl State Machine config */
65  #define CC1101_CR_MCSM1             0x17    /* Main Radio Cntrl State Machine config */
66  #define CC1101_CR_MCSM0             0x18    /* Main Radio Cntrl State Machine config */
67  #define CC1101_CR_FOCCFG            0x19    /* Frequency Offset Compensation config */
68  #define CC1101_CR_BSCFG             0x1A    /* Bit Synchronization configuration */
69  #define CC1101_CR_AGCCTRL2          0x1B    /* AGC control */
70  #define CC1101_CR_AGCCTRL1          0x1C    /* AGC control */
71  #define CC1101_CR_AGCCTRL0          0x1D    /* AGC control */
72  #define CC1101_CR_WOREVT1           0x1E    /* High byte Event 0 timeout */
73  #define CC1101_CR_WOREVT0           0x1F    /* Low byte Event 0 timeout */
74  #define CC1101_CR_WORCTRL           0x20    /* Wake On Radio control */
75  #define CC1101_CR_FREND1            0x21    /* Front end RX configuration */
76  #define CC1101_CR_FREND0            0x22    /* Front end TX configuration */
77  #define CC1101_CR_FSCAL3            0x23    /* Frequency synthesizer calibration */
78  #define CC1101_CR_FSCAL2            0x24    /* Frequency synthesizer calibration */
79  #define CC1101_CR_FSCAL1            0x25    /* Frequency synthesizer calibration */
80  #define CC1101_CR_FSCAL0            0x26    /* Frequency synthesizer calibration */
81  #define CC1101_CR_RCCTRL1           0x27    /* RC oscillator configuration */
82  #define CC1101_CR_RCCTRL0           0x28    /* RC oscillator configuration */
83  #define CC1101_CR_FSTEST            0x29    /* Frequency synthesizer cal control */
```

```
84  #define CC1101_CR_PTEST             0x2A    /* Production test */
85  #define CC1101_CR_AGCTEST           0x2B    /* AGC test */
86  #define CC1101_CR_TEST2             0x2C    /* Various test settings */
87  #define CC1101_CR_TEST1             0x2D    /* Various test settings */
88  #define CC1101_CR_TEST0             0x2E    /* Various test settings */
89
90  /* Status Registers */
91  #define CC1101_SR_PARTNUM           0x30    /* Part number */
92  #define CC1101_SR_VERSION           0x31    /* Current version number */
93  #define CC1101_SR_FREQEST           0x32    /* Frequency offset estimate */
94  #define CC1101_SR_LQI               0x33    /* Demodulator estimate for link quality */
95  #define CC1101_SR_RSSI              0x34    /* Received signal strength indication */
96  #define CC1101_SR_MARCSTATE         0x35    /* Control state machine state */
97  #define CC1101_SR_WORTIME1          0x36    /* High byte of WOR timer */
98  #define CC1101_SR_WORTIME0          0x37    /* Low byte of WOR timer */
99  #define CC1101_SR_PKTSTATUS         0x38    /* Current GDOx status and packet status */
100 #define CC1101_SR_VCO_VC_DAC        0x39    /* Current setting from PLL cal module */
101 #define CC1101_SR_TXBYTES           0x3A    /* Underflow and # of bytes in TXFIFO */
102 #define CC1101_SR_RXBYTES           0x3B    /* Overflow and # of bytes in RXFIFO */
103 #define CC1101_SR_RCCTRL1_STATUS    0x3C    /* Last RC oscilator calibration results */
104 #define CC1101_SR_RCCTRL0_STATUS    0x3D    /* Last RC oscilator calibration results */
105
106 /* Single / Burst access */
107 #define CC1101_WRITE_BURST          0x40
108 #define CC1101_READ_SINGLE          0x80
109 #define CC1101_READ_BURST           0xC0
110
111 /* Memory locations */
112 #define CC1101_ML_PATABLE           0x3E
113 #define CC1101_ML_TXFIFO            0x3F
114 #define CC1101_ML_RXFIFO            0x3F
115
116 /*****************************************************************************
117 **  Prototypes of the private functions
118 *****************************************************************************/
119
120 void cc1101_spi_setup(void);
121 void cc1101_spi_write_register(uint8_t address, uint8_t value);
122 void cc1101_spi_write_register_burst(uint8_t address, uint8_t * buffer, uint8_t count);
123 char cc1101_spi_read_register(uint8_t address);
124 void cc1101_spi_read_register_burst(uint8_t address, uint8_t *buffer, uint8_t count);
125 uint8_t cc1101_spi_read_status(uint8_t address);
126 void cc1101_spi_command_strobe(uint8_t strobe);
127
128 /*****************************************************************************
129 **  Public functions
130 *****************************************************************************/
131
132 void cc1101_init(void) {    // Initialisierung des CC1101
133     cc1101_spi_setup();
134
135     CC1101_GDO2_DIR_IN;
136     CC1101_GDO2_IRQ_RISING_EDGE;
137     CC1101_GDO2_IRQ_DISABLE;
138     CC1101_GDO2_CLEAR_IRQ;
139 }
140
141 void cc1101_set_tx(uint8_t brate) { // CC1101 auf senden stellen
142     cc1101_init_tx();
143     cc1101_reset();                 // Reset chip and go to idle state
144     cc1101_config_packet(brate);    // Configure the transceiver for sending packets
145 }
146
147 void cc1101_set_rx(uint8_t brate) { // CC1101 auf empfangen stellen
148     cc1101_init_rx();
149     cc1101_reset();                 // Reset chip and go to idle state
150     cc1101_config_packet(brate);    // Configure the transceiver for sending packets
151 }
152
153 void cc1101_init_rx(void) {         // CC1101 auf senden initialisieren
154     cc1101_spi_setup();
155
156     CC1101_GDO2_DIR_IN;
157     CC1101_GDO2_IRQ_RISING_EDGE;
158     CC1101_GDO2_IRQ_DISABLE;
159     CC1101_GDO2_CLEAR_IRQ;
160
161 }
162
163 void cc1101_init_tx(void) {         // CC1101 auf empfangen initialisieren
164     cc1101_spi_setup();
165
166     CC1101_GDO2_DIR_IN;
167     CC1101_GDO2_IRQ_FALLING_EDGE;
168     CC1101_GDO2_IRQ_DISABLE;
```

```
169         CC1101_GDO2_CLEAR_IRQ;
170     }
171
172     void cc1101_power_up_reset(void) {
173         // Manual power-on reset
174         CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;         // Chip enable
175         delay_100us_1MHz(1);
176         CC1101_CSn_PxOUT |= CC1101_CSn_PIN;          // Chip disable
177         delay_100us_1MHz(1);
178         CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;         // Chip enable
179         delay_100us_1MHz(5);
180         CC1101_CSn_PxOUT |= CC1101_CSn_PIN;          // Chip disable
181         delay_100us_1MHz(5);
182
183         cc1101_reset();
184     }
185
186     void cc1101_reset(void) {
187         CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;         // Chip enable
188
189         while (UCA0STAT & UCBUSY);                   // Wait for TX to finish
190         UCA0TXBUF = CC1101_CS_SRES;                  // Send reset strobe
191         while (UCA0STAT & UCBUSY);                   // Wait for TX to finish
192
193         CC1101_CSn_PxOUT |= CC1101_CSn_PIN;          // Chip disable
194
195         delay_ms(1);
196     }
197
198     /*****************************************************************************
199     ** CC1101 für den asynchronen Empfang einstellen
200     *****************************************************************************/
201     void cc1101_config_no_packet(void) {
202         // GDO0 at high impedance
203         cc1101_spi_write_register(CC1101_CR_IOCFG0, 0x2E);
204
205         // GDO2 at high impedance
206         cc1101_spi_write_register(CC1101_CR_IOCFG2, 0x2E);
207
208         // Asynchronous serial mode, Infinite packet length mode
209         cc1101_spi_write_register(CC1101_CR_PKTCTRL0, 0x32);
210
211         // Channel 0
212         cc1101_spi_write_register(CC1101_CR_CHANNR, 0x00);
213
214         // Carrier frequency
215         cc1101_spi_write_register(CC1101_CR_FSCTRL0, FREQUENCY_OFFSET);
216
217         // Carrier frequency: 433.999969 MHz
218         cc1101_spi_write_register(CC1101_CR_FREQ2, 0x10);
219         cc1101_spi_write_register(CC1101_CR_FREQ1, 0xB1);
220         cc1101_spi_write_register(CC1101_CR_FREQ0, 0x3B);
221
222         //cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0xFD); //11111101 //NS
223         cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0xFD);
224
225         // 250 kBaud, OOK
226         cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x3B); //Data Rate
227         cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x30); //ASK/OOK
228
229         cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x00); //???
230
231         cc1101_spi_write_register(CC1101_CR_DEVIATN, 0x15); //Modem deviation setting
232         cc1101_spi_write_register(CC1101_CR_MCSM0, 0x18);   //Main Radio Control State Machine configuration
233
234         // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
235         cc1101_spi_write_register(CC1101_CR_FOCCFG, 0x16);
236
237         // 10 dBm output power
238         cc1101_spi_write_register_burst(CC1101_ML_PATABLE, cc1101_patable, cc1101_patablelen);
239         cc1101_spi_write_register(CC1101_CR_FREND0, 0x11);
240     }
241
242     /*****************************************************************************
243     ** CC1101 für den asynchronen Empfang einstellen
244     *****************************************************************************/
245     void cc1101_config_no_packet_rx(void) {
246         // GDO2 Output Pin Configuration 2
247         // 0x0D -> Serial Data Output. Used for asynchronous serial mode
248         cc1101_spi_write_register(CC1101_CR_IOCFG2, 0x0D);
249
250         // GDO1 Output Pin Configuration 1
251         // 0x2E -> High impedance (3-state)
252         cc1101_spi_write_register(CC1101_CR_IOCFG1, 0x2E);
253
```

```
254         // GDO0 Output Pin Configuration 0
255         // 0x36 -> CLK_XOSC/8
256         cc1101_spi_write_register(CC1101_CR_IOCFG0, 0x36);
257
258         // RX FIFO and TX FIFO Thresholds
259         cc1101_spi_write_register(CC1101_CR_FIFOTHR, 0x47);
260
261         // Packet Automation Control 0
262         cc1101_spi_write_register(CC1101_CR_PKTCTRL0, 0x32);
263
264         // Frequency Synthesizer Control 1
265         cc1101_spi_write_register(CC1101_CR_FSCTRL1, 0x06);
266
267         // Frequency Control Word, High Byte
268         cc1101_spi_write_register(CC1101_CR_FREQ2, 0x10);
269
270         // Frequency Control Word, Middle Byte
271         cc1101_spi_write_register(CC1101_CR_FREQ1, 0xB1);
272
273         //Frequency Control Word, LOW Byte
274         cc1101_spi_write_register(CC1101_CR_FREQ0, 0x3B);
275
276         /*************************************************/
277         // Modem Configuration 4
278         // 250kBaud              ->  0xxD
279         // RX Filter 58.035714   ->  0xFx
280         cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x5D);
281
282         // Modem Configuration 3
283         // 250kBaud -> 0x3B
284         cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x3B);
285
286         // Modem Configuration 2
287         // OOK -> 0x30
288         cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x30);
289
290         // Modem Configuration 1
291         // Channel spacing 199.951172kHz -> 0x22
292         cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
293
294         // Modem Configuration 0
295         // Channel spacing 199.951172kHz -> 0xF8
296         cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0xF8);
297         /*************************************************/
298
299         // Modem Deviation Setting
300         cc1101_spi_write_register(CC1101_CR_DEVIATN, 0x15);
301
302         // Main Radio Control State Machine Configuration 2
303         cc1101_spi_write_register(CC1101_CR_MCSM2, 0x07);
304
305         // Main Radio Control State Machine Configuration 1
306         cc1101_spi_write_register(CC1101_CR_MCSM1, 0x30);
307
308         // Main Radio Control State Machine Configuration 0
309         cc1101_spi_write_register(CC1101_CR_MCSM0, 0x18);
310
311         // Frequency Offset Compensation Configuration
312         cc1101_spi_write_register(CC1101_CR_FOCCFG, 0x16);
313
314         // Bit Synchronization Configuration
315         cc1101_spi_write_register(CC1101_CR_BSCFG, 0x6C);
316
317         // AGC Control 2
318         cc1101_spi_write_register(CC1101_CR_AGCCTRL2, 0x03);
319
320         // AGC Control 1
321         cc1101_spi_write_register(CC1101_CR_AGCCTRL1, 0x40);
322
323         // AGC Control 0
324         cc1101_spi_write_register(CC1101_CR_AGCCTRL0, 0x91);
325
326         // Wake On Radio Control
327         cc1101_spi_write_register(CC1101_CR_WORCTRL, 0x8F);
328
329         // Front End RX Configuration 1
330         cc1101_spi_write_register(CC1101_CR_FREND1, 0x56);
331
332         // Front End RX Configuration 0
333         cc1101_spi_write_register(CC1101_CR_FREND0, 0x11);
334
335         // 10 dBm output power
336         cc1101_spi_write_register_burst(CC1101_ML_PATABLE, cc1101_patable, 2);
337     }
338
```

```
339 /*****************************************************************************
340 ** CC1101 für das paketorientierete Senden/Empfangen einstellen
341 *****************************************************************************/
342 void cc1101_config_packet(uint8_t brate) {
343     // Rising edge on GDO0 when packet received and CRC check OK
344     // (Deasserted when first byte is read from RX FIFO)
345     cc1101_spi_write_register(CC1101_CR_IOCFG0, 0x07);
346 
347     // Rising edge on GDO2 when sync word has been received
348     cc1101_spi_write_register(CC1101_CR_IOCFG2, 0x06);
349 
350     // 4 SYNC bytes: 0x12 0x09 (repeated once)
351     cc1101_spi_write_register(CC1101_CR_SYNC1, 0x81);
352     cc1101_spi_write_register(CC1101_CR_SYNC0, 0x81);
353 
354     // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
355     cc1101_spi_write_register(CC1101_CR_PKTCTRL1, 0x0A);
356 
357     // No whitening, FIFO mode, CRC disable, Fixed packet length
358     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, 0x00);
359 
360     // Device Address
361     cc1101_spi_write_register(CC1101_CR_ADDR, ADDRESS_THIS_SENSOR);
362 
363     // Channel 0
364     cc1101_spi_write_register(CC1101_CR_CHANNR, 0x00);
365 
366     // IF frequency: 152.34375 kHz
367     cc1101_spi_write_register(CC1101_CR_FSCTRL1, 0x06);
368 
369     // Carrier frequency
370     cc1101_spi_write_register(CC1101_CR_FSCTRL0, FREQUENCY_OFFSET);
371 
372     // Carrier frequency: 433.999969 MHz
373     cc1101_spi_write_register(CC1101_CR_FREQ2, 0x10);
374     cc1101_spi_write_register(CC1101_CR_FREQ1, 0xB1);
375     cc1101_spi_write_register(CC1101_CR_FREQ0, 0x3B);
376 
377     switch(brate) {
378         /*************************************************************************************
379         ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
380         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
381         *************************************************************************************/
382         case 40: {
383             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8A);
384             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
385             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
386             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
387             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
388         }break;
389 
390         /*************************************************************************************
391         ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
392         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
393         *************************************************************************************/
394         case 60: {
395             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
396             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x2E);
397             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
398             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
399             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
400         }break;
401 
402         /*************************************************************************************
403         ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
404         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
405         *************************************************************************************/
406         case 80: {
407             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
408             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
409             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
410             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
411             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
412         }break;
413 
414         /*************************************************************************************
415         ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
416         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
417         *************************************************************************************/
418         case 100: {
419             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
420             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0xF8);
421             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
422             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
423             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
```

```
424             }break;
425
426             /*****************************************************************************
427             ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
428             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
429             *****************************************************************************/
430             case 120: {
431                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
432                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x2E);
433                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
434                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
435                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
436             }break;
437
438             /*****************************************************************************
439             ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
440             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
441             *****************************************************************************/
442             case 140: {
443                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
444                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x61);
445                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
446                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
447                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
448             }break;
449
450             /*****************************************************************************
451             ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
452             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
453             *****************************************************************************/
454             case 160: {
455                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
456                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
457                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
458                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
459                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
460             }break;
461
462             /*****************************************************************************
463             ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
464             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
465             *****************************************************************************/
466             case 180: {
467                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
468                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0xC6);
469                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
470                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
471                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
472             }break;
473
474         }
475 //      // AGC Control 2
476 //      cc1101_spi_write_register(CC1101_CR_AGCCTRL2, 0x03);
477 //
478 //      // AGC Control 1
479 //      cc1101_spi_write_register(CC1101_CR_AGCCTRL1, 0x40);
480 //
481 //      // AGC Control 0
482 //      cc1101_spi_write_register(CC1101_CR_AGCCTRL0, 0x91);
483
484         // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
485         cc1101_spi_write_register(CC1101_CR_MCSM0, 0x10);
486
487         // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
488         cc1101_spi_write_register(CC1101_CR_FOCCFG, 0x16);
489
490         //cc1101_spi_write_register(CC1101_CR_FREND1, 0x00);
491         cc1101_spi_write_register_burst(CC1101_ML_PATABLE, cc1101_patable, cc1101_patablelen);
492         cc1101_spi_write_register(CC1101_CR_FREND0, 0x11);      // 10 dBm output power
493     }
494
495     void cc1101_fill_tx_fifo(uint8_t * buffer, uint8_t length) {
496         // Clear TX FIFO
497         cc1101_clear_tx_fifo();
498
499         // Transfer the bytes via SPI to the TX fifo of the transceiver
500         cc1101_spi_write_register_burst(CC1101_ML_TXFIFO, buffer, length);
501     }
502
503     void cc1101_read_rx_fifo(uint8_t * buffer, uint8_t length) {
504         // Disable the receiver
505         cc1101_idle();
506
507         // Transfer the bytes via SPI from the RX fifo
508         cc1101_spi_read_register_burst(CC1101_ML_RXFIFO, buffer, length);
```

```
509 }
510
511 void cc1101_enable_crc(void) {
512     uint8_t current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
513     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, current | BIT2);
514 }
515
516 void cc1101_disable_crc(void) {
517     uint8_t current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
518     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, current & ~BIT2);
519 }
520
521 void cc1101_clear_tx_fifo(void) {
522     cc1101_spi_command_strobe(CC1101_CS_SFTX);
523 }
524
525 void cc1101_clear_rx_fifo(void) {
526     cc1101_spi_command_strobe(CC1101_CS_SFRX);
527 }
528
529 uint8_t cc1101_get_rxbytes(void) {
530     return (cc1101_spi_read_status(CC1101_SR_RXBYTES) & 0x7F);
531 }
532
533 void cc1101_sleep(void) {
534     cc1101_spi_command_strobe(CC1101_CS_SPWD);
535 }
536
537 void cc1101_idle(void) {
538     cc1101_spi_command_strobe(CC1101_CS_SIDLE);
539 }
540
541 void cc1101_tx_carrier(void) {
542     cc1101_spi_command_strobe(CC1101_CS_STX);
543 }
544
545 void cc1101_tx(uint8_t packages_to_tx) {
546     // DATA packet length
547     cc1101_spi_write_register(CC1101_CR_PKTLEN, packages_to_tx);
548
549     // Enable CRC calculation when transmitting data
550     cc1101_enable_crc();
551
552     // TX state
553     cc1101_spi_command_strobe(CC1101_CS_STX);
554
555     // Wait 10 ms for TX finished
556     //delay_ms(8);
557 }
558
559 void cc1101_rx(uint8_t packages_to_rx) {
560     // DATA packet length
561     cc1101_spi_write_register(CC1101_CR_PKTLEN, packages_to_rx);
562
563     // Enable CRC check when receiving data
564     cc1101_enable_crc();
565
566     // Clear RX FIFO
567     cc1101_clear_rx_fifo();
568
569     // RX state
570     cc1101_spi_command_strobe(CC1101_CS_SRX);
571 }
572
573 void cc1101_burst_rx(void) {
574     // RX state
575     cc1101_spi_command_strobe(CC1101_CS_SRX);
576 }
577
578
579 uint8_t cc1101_get_partnum(void) {
580     return cc1101_spi_read_status(CC1101_SR_PARTNUM);
581 }
582
583 uint8_t cc1101_get_version(void) {
584     return cc1101_spi_read_status(CC1101_SR_VERSION);
585 }
586
587 uint8_t cc1101_get_marcstate(void) {
588     return (cc1101_spi_read_status(CC1101_SR_MARCSTATE) & 0x1F);
589 }
590
591 /***************************************************************************
592 **  Private functions
593 ***************************************************************************/
```

```
594
595 void cc1101_spi_setup(void) {
596     CC1101_CSn_PxDIR |= CC1101_CSn_PIN;      // nCS is output
597     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;      // Unselect CC1101 chip
598
599     UCA0CTL1 |= UCSWRST;                     // Hold state machine in reset
600
601     UCA0CTL0 |= UCCKPH;                      // Data is sampled on rising edge
602     UCA0CTL0 &= ~UCCKPL;                     // Inactive clock is low
603     UCA0CTL0 |= (UCMST | UCMSB | UCSYNC);    // MSB first, Master mode, Synchronous mode
604     UCA0CTL0 &= ~(UCMODE1 | UCMODE0);        // 3-pin SPI
605     UCA0MCTL = 0;                            // No modulation
606
607     // SMCLK / 2
608     UCA0CTL1 |= (UCSSEL1 | UCSSEL0);
609     UCA0BR0 = 5;
610     UCA0BR1 = 0;
611
612     // SPI functionality for pins
613     CC1101_SPI_PxSEL |= (CC1101_SPI_SI_PIN | CC1101_SPI_SO_GDO1_PIN | CC1101_SPI_SCLK_PIN);
614     CC1101_SPI_PxDIR |= (CC1101_SPI_SI_PIN | CC1101_SPI_SCLK_PIN); // MOSI and CLK are outputs
615     CC1101_SPI_PxDIR &= ~CC1101_SPI_SO_GDO1_PIN; // MISO is input
616
617     UCA0CTL1 &= ~UCSWRST;    // Initialize USART state machine
618 }
619
620 void cc1101_spi_write_register(uint8_t address, uint8_t value) {
621     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;      // Chip enable
622     while (UCA0STAT & UCBUSY);                // Wait for TX to finish
623     UCA0TXBUF = address;                      // Send address
624     while (UCA0STAT & UCBUSY);                // Wait for TX to finish
625     UCA0TXBUF = value;                        // Send value
626     while(UCA0STAT & UCBUSY);                 // Wait for TX complete
627     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;       // Chip disable
628 }
629
630 void cc1101_spi_write_register_burst(uint8_t address, uint8_t * buffer, uint8_t count) {
631     uint8_t i;
632
633     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;      // Chip enable
634     while (UCA0STAT & UCBUSY);                // Wait for TX to finish
635     UCA0TXBUF = address | CC1101_WRITE_BURST; // Send address
636
637     for (i = 0; i < count; i++) {
638         while (UCA0STAT & UCBUSY);            // Wait for TX to finish
639         UCA0TXBUF = buffer[i];                // Send data
640     }
641
642     while(UCA0STAT & UCBUSY);                 // Wait for TX complete
643     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;       // Chip disable
644 }
645
646 char cc1101_spi_read_register(uint8_t address) {
647     uint8_t x;
648
649     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;          // Chip enable
650     while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
651     UCA0TXBUF = (address | CC1101_READ_SINGLE);   // Send address
652     while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
653     UCA0TXBUF = 0;                                // Dummy write so we can read data
654     while(UCA0STAT & UCBUSY);                     // Wait for TX complete
655     x = UCA0RXBUF;                                // Read data
656     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;           // Chip disable
657
658     return x;
659 }
660
661 void cc1101_spi_read_register_burst(uint8_t address, uint8_t * buffer, uint8_t count) {
662     uint16_t i;
663
664     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;         // Chip enable
665     while (UCA0STAT & UCBUSY);                   // Wait for TX to finish
666     UCA0TXBUF = (address | CC1101_READ_BURST);   // Send address
667     while (UCA0STAT & UCBUSY);                   // Wait for TX to finish
668
669     for (i = 0; i < count; i++) {
670         UCA0TXBUF = 0;                           // Initiate next data RX, meanwhile
671         while (UCA0STAT & UCBUSY);               // Wait for TX to finish
672         buffer[i] = UCA0RXBUF;                   // Store data from last data RX
673     }
674
675     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;       // Chip disable
676 }
677
678 uint8_t cc1101_spi_read_status(uint8_t address) {
```

```
679         uint8_t status;
680
681         CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;          // Chip enable
682         while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
683         UCA0TXBUF = (address | CC1101_READ_BURST);     // Send address
684         while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
685         UCA0TXBUF = 0;                                // Dummy write so we can read data
686         while (UCA0STAT & UCBUSY);                    // Wait for TX complete
687         status = UCA0RXBUF;                           // Read data
688         CC1101_CSn_PxOUT |= CC1101_CSn_PIN;           // Chip disable
689
690         return status;
691     }
692
693     void cc1101_spi_command_strobe(uint8_t strobe) {
694         CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;          // Chip enable
695         while (UCA0STAT & UCBUSY);                    // Wait for TX to finish
696         UCA0TXBUF = strobe;                           // Send strobe
697         while (UCA0STAT & UCBUSY);                    // Wait for TX complete
698         CC1101_CSn_PxOUT |= CC1101_CSn_PIN;           // Chip disable
699     }
```

F.1.10. cc1101.h

```
/*******************************************************************************
**     Description:    cc1101.h
**     Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
**     Date:           06/03/2013
**     Author:         Nico Sassano
**                     modified taken from Phillip Durdaut
*******************************************************************************/

#ifndef CC1101_H_
#define CC1101_H_

#include "main.h"

#define CC1101_CSn_PxDIR            P1DIR
#define CC1101_CSn_PxOUT            P1OUT
#define CC1101_CSn_PIN              BIT5

#define CC1101_SPI_PxSEL            P3SEL
#define CC1101_SPI_PxDIR            P3DIR
#define CC1101_SPI_PxIN             P3IN
#define CC1101_SPI_SI_PIN           BIT4
#define CC1101_SPI_SO_GDO1_PIN      BIT5
#define CC1101_SPI_SCLK_PIN         BIT0

#define CC1101_GDO0_PxDIR           P1DIR
#define CC1101_GDO0_PxIN            P1IN
#define CC1101_GDO0_PxIES           P1IES
#define CC1101_GDO0_PxIE            P1IE
#define CC1101_GDO0_PxIFG           P1IFG
#define CC1101_GDO0_PIN             BIT6
#define CC1101_GDO0_POS             6

#define CC1101_GDO2_PxDIR           P1DIR
#define CC1101_GDO2_PxIN            P1IN
#define CC1101_GDO2_PxIES           P1IES
#define CC1101_GDO2_PxIE            P1IE
#define CC1101_GDO2_PxIFG           P1IFG
#define CC1101_GDO2_PIN             BIT7
#define CC1101_GDO2_POS             7

/*-------------------------------------------------------------------------------
  Macros
-------------------------------------------------------------------------------*/

// CRC check done
#define CC1101_GDO0_DIR_IN              (CC1101_GDO0_PxDIR &= ~CC1101_GDO0_PIN)
#define CC1101_GDO0_IN                  ((CC1101_GDO0_PxIN & CC1101_GDO0_PIN) >> CC1101_GDO0_POS)

// Sync word detected
#define CC1101_GDO2_DIR_IN              (CC1101_GDO2_PxDIR &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IN                  ((CC1101_GDO2_PxIN & CC1101_GDO2_PIN) >> CC1101_GDO2_POS)
#define CC1101_GDO2_IRQ_RISING_EDGE     (CC1101_GDO2_PxIES &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_FALLING_EDGE    (CC1101_GDO2_PxIES |= CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_ENABLE          (CC1101_GDO2_PxIE |= CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_DISABLE         (CC1101_GDO2_PxIE &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_PENDING         ((CC1101_GDO2_PxIFG & CC1101_GDO2_PIN) == CC1101_GDO2_PIN)
#define CC1101_GDO2_CLEAR_IRQ           (CC1101_GDO2_PxIFG &= ~(CC1101_GDO2_PIN))


// CRC ok //NS
#define CC1101_GDO0_DIR_IN              (CC1101_GDO0_PxDIR &= ~CC1101_GDO0_PIN)
#define CC1101_GDO0_IN                  ((CC1101_GDO0_PxIN & CC1101_GDO0_PIN) >> CC1101_GDO0_POS)
#define CC1101_GDO0_IRQ_RISING_EDGE     (CC1101_GDO0_PxIES &= ~CC1101_GDO0_PIN)
#define CC1101_GDO0_IRQ_FALLING_EDGE    (CC1101_GDO0_PxIES |= CC1101_GDO0_PIN)
#define CC1101_GDO0_IRQ_ENABLE          (CC1101_GDO0_PxIE |= CC1101_GDO0_PIN)
#define CC1101_GDO0_IRQ_DISABLE         (CC1101_GDO0_PxIE &= ~CC1101_GDO0_PIN)
#define CC1101_GDO0_IRQ_PENDING         ((CC1101_GDO0_PxIFG & CC1101_GDO0_PIN) == CC1101_GDO0_PIN)
#define CC1101_GDO0_CLEAR_IRQ           (CC1101_GDO0_PxIFG &= ~(CC1101_GDO0_PIN))

/*-------------------------------------------------------------------------------
  Public functions
-------------------------------------------------------------------------------*/
void cc1101_set_tx(uint8_t);        //27.03.13 NS
void cc1101_set_rx(uint8_t);        //27.03.13 NS
void cc1101_init_rx(void);       //27.03.13 NS
void cc1101_init_tx(void);       //27.03.13 NS
void cc1101_init(void);
void cc1101_power_up_reset(void);
void cc1101_reset(void);
void cc1101_config_no_packet(void);
void cc1101_config_no_packet_rx(void);
void cc1101_config_packet(uint8_t);
void cc1101_fill_tx_fifo(uint8_t * buffer, uint8_t length);
```

```
84  void cc1101_read_rx_fifo(uint8_t * buffer, uint8_t length);
85  void cc1101_enable_crc(void);
86  void cc1101_disable_crc(void);
87  void cc1101_clear_tx_fifo(void);
88  void cc1101_clear_rx_fifo(void);
89  uint8_t cc1101_get_rxbytes(void);
90  void cc1101_sleep(void);
91  void cc1101_idle(void);
92  void cc1101_tx_carrier(void);
93  void cc1101_tx(uint8_t);
94  void cc1101_rx(uint8_t);
95  void cc1101_burst_rx(void);
96  uint8_t cc1101_get_partnum(void);
97  uint8_t cc1101_get_version(void);
98  uint8_t cc1101_get_marcstate(void);
99
100 #endif /* CC1101_H_ */
```

F.1.11. clk.c

```
1  /*******************************************************************************
2  **    Description:    clk.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           24/04/2013
5  **    Author:         Nico Sassano
6  *******************************************************************************/
7
8  #include "main.h"
9
10 extern uint8_t clk_set;
11
12 void clk_init_startup(void) {   // 06.05.13 NS
13
14     clk_set = 1;     //CLK ist auf 1MHz gesetzt
15
16     DCOCTL = CALDCO_1MHZ;
17
18     // Set external XT off
19     BCSCTL1  = XT2OFF;
20
21     // Set for 1Mhz
22     BCSCTL1 |= CALBC1_1MHZ;
23
24     // MCLK = DCOCLK, div MCLK = 0
25     BCSCTL2  = DIVM_0;
26
27     // SMCLK = DCOCLK, div SMCLK = 0
28     BCSCTL2 |= DIVS_0;
29
30     // No external DCO resistor
31     BCSCTL2 &= ~BIT0;
32
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
42             // Calibrate DCO with calibration data for 1 MHz
43             DCOCTL = CALDCO_1MHZ;
44
45             // Turn off crystal oscillator XT2
46             BCSCTL1 = XT2OFF;
47
48             // Load BCSCTL1 calibration Data for 1 MHz
49             BCSCTL1 |= CALBC1_1MHZ;
50         }break;
51
52         case 8: {
53             clk_set = 8;     //CLK ist auf 8MHz gesetzt
54
55             DCOCTL = CALDCO_8MHZ;
56
57             // Set external XT off
58             BCSCTL1  = XT2OFF;
59
60             // Set for 8Mhz
61             BCSCTL1 |= CALBC1_8MHZ;
62
63             // MCLK = DCOCLK, div MCLK = 0
64             BCSCTL2  = DIVM_0;
65
66             // SMCLK = DCOCLK, div SMCLK = 0
67             BCSCTL2 |= DIVS_0;
68
69             // No external DCO resistor
70             BCSCTL2 &= ~BIT0;
71         }break;
72
73         case 12: {
74             clk_set = 12;     //CLK ist auf 12MHz gesetzt
75
76             DCOCTL = CALDCO_12MHZ;
77
78             // Set external XT off
79             BCSCTL1  = XT2OFF;
80
81             // Set for 12Mhz
82             BCSCTL1 |= CALBC1_12MHZ;
83
```

```
84                  // MCLK = DCOCLK, div MCLK = 0
85                  BCSCTL2  = DIVM_0;
86
87                  // SMCLK = DCOCLK, div SMCLK = 0
88                  BCSCTL2 |= DIVS_0;
89
90                  // No external DCO resistor
91                  BCSCTL2 &= ~BIT0;
92              }break;
93
94              case 16: {
95                  clk_set = 16;     //CLK ist auf 16MHz gesetzt
96
97                  DCOCTL = CALDCO_16MHZ;
98
99                  // Turn off crystal oscillator XT2
100                 BCSCTL1 = XT2OFF;
101
102                 // Load BCSCTL1 calibration Data for 1 MHz
103                 BCSCTL1 |= CALBC1_16MHZ;
104             }break;
105         }
106 }
```

F.1.12. clk.h

```
1  /*******************************************************************************
2  **     Description:     clk.h
3  **     Hardware:        BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **     Date:            24/04/2013
5  **     Author:          Nico Sassano
6  *******************************************************************************/
7
8  #ifndef CLK_H_
9  #define CLK_H_
10
11 #include "main.h"
12
13 void clk_init_startup(void);
14 void clk_init_MHz(uint8_t mhz);
15
16 #endif /* CLK_H_ */
```

F.1.13. delay.c

```
1   /*******************************************************************************
2   **    Description:    delay.c
3   **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **    Date:           06/03/2013
5   **    Author:         Nico Sassano
6   **                    modified taken from Phillip Durdaut
7   *******************************************************************************/
8
9   #include "main.h"
10
11  extern uint8_t clk_set;
12
13  /*-------------------------------------------------------------------------*
14    Prototypes of the private functions
15  -------------------------------------------------------------------------*/
16
17  void delay(u16_t i);
18
19  /*-------------------------------------------------------------------------*
20    Public functions
21  -------------------------------------------------------------------------*/
22
23  void delay_ms(u16_t delay_ms)
24  {
25      while (delay_ms > 0) {
26          delay(DELAY_CYCLES_PER_MS);
27          delay_ms--;
28      }
29  }
30
31  void delay_100us_1MHz(u16_t delay_100us_1MHz)
32  {
33      while (delay_100us_1MHz > 0) {
34          delay(DELAY_CYCLES_PER_100US_1MHZ);
35          delay_100us_1MHz--;
36      }
37  }
38
39  void delay_10us_1MHz(u16_t delay_10us_1MHz)
40  {
41      while (delay_10us_1MHz > 0) {
42          _NOP();
43          delay_10us_1MHz--;
44      }
45  }
46
47  void delay_10us_16MHz(u16_t delay_10us_16MHz)
48  {
49      while (delay_10us_16MHz > 0) {
50              delay(DELAY_CYCLES_PER_10US_16MHZ);
51              delay_10us_16MHz--;
52      }
53  }
54
55
56  void new_delay_ms(uint16_t time_ms) {
57      uint16_t index = 0;
58
59      switch(clk_set) {
60          case 1: {
61              while (time_ms > 0) {
62                  index = 70;
63                  while (index > 0) {
64                      _NOP();
65                      index--;
66                  }
67                  time_ms--;
68              }
69
70          }break;
71
72          case 8: {
73              while (time_ms > 0) {
74                  index = 560;
75                  while (index > 0) {
76                      _NOP();
77                      index--;
78                  }
79                  time_ms--;
80              }
81          }break;
82
```

```
83          case 12: {
84              while (time_ms > 0) {
85                  index = 840;
86                  while (index > 0) {
87                      _NOP();
88                      index--;
89                  }
90                  time_ms--;
91              }
92          }break;
93
94          case 16: {
95              while (time_ms > 0) {
96                  index = 1120;
97                  while (index > 0) {
98                      _NOP();
99                      index--;
100                 }
101                 time_ms--;
102             }
103         }break;
104     }
105 }
106
107 /*-------------------------------------------------------------------
108   Private functions
109 -------------------------------------------------------------------*/
110
111 void delay(u16_t i)
112 {
113     while (i > 0) {
114         _NOP();
115         i--;
116     }
117 }
```

F.1.14. delay.h

```
1  /*******************************************************************************
2  **    Description:    delay.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *******************************************************************************/
8
9  #ifndef DELAY_H_
10 #define DELAY_H_
11
12 #include "main.h"
13
14 /*-------------------------------------------------------------------------------
15   Defines
16 -------------------------------------------------------------------------------*/
17
18 #define DELAY_CYCLES_PER_MS             70
19 #define DELAY_CYCLES_PER_100US_1MHZ     1
20 #define DELAY_CYCLES_PER_10US_1MHZ      1
21 #define DELAY_CYCLES_PER_10US_16MHZ     7   //113
22
23 /*-------------------------------------------------------------------------------
24   Public functions
25 -------------------------------------------------------------------------------*/
26
27 void delay_ms(u16_t delay_ms);
28 void delay_100us_1MHz(u16_t delay_100us);
29 void delay_10us_1MHz(uint16_t);
30 void delay_10us_16MHz(uint16_t);
31 void new_delay_ms(uint16_t);
32
33 #endif /* DELAY_H_ */
```

F.1.15. i2c.c

```
/*****************************************************************************
**     Description:    i2c.c
**     Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
**     Date:           06/03/2013
**     Author:         Nico Sassano
**                     modified taken from Niels Jegenhorst
*****************************************************************************/

#include "main.h"


#define UCB0BR_BIT_CLK 100.0     // UCB0 Bit Clock [kHz]
                                 // UCB0 Baud Rate Control 0 Setting

void i2c_init(void)
{

    P3SEL = 0x06;                // Set Pin 3.1 and 3.2 for I2C communication

    UCB0CTL1 |= UCSWRST;         // Reset the USCI logic

    /** control register 0 ******************************************************
    * Set UCB0CTL  -> Master mode | I2C Mode | Synchronous mode
    *              -> Own address is a 7-bit address, Address slave
    *                 with 7-bit address
    *****************************************************************************/
    UCB0CTL0 = (UCMST | UCMODE1 | UCMODE0 | UCSYNC);

    /** control register 1 ******************************************************
    * Set UCB0CTL1 -> clock source SMCLK | Software reset enable
    *              -> Receiver | Acknowledge normally | No STOP generated |
    *                 Do not generate START condition
    *****************************************************************************/
    UCB0CTL1 = (UCSSEL1 | UCSWRST);

    /** bit rate control register 0/1 *******************************************
    * -> Datasheet S. 476 <-
    *
    * Baud rate = 100kbps
    * -> f_brclk = SMCLK = 0.5MHz
    * -> UCBRX = 5
    * f_bitclk = f_brclk / UCBRX = 100kHz,
    *****************************************************************************/
    UCB0BR0 = ((uint16_t) SMCLK / UCB0BR_BIT_CLK); // low byte, UCB0RX = (UCB0R0 + UCB0R1 x 128)
    UCB0BR1 = 0x00; // high byte

    /** Own Address Register ****************************************************
    * I2C own 7-bit address = 0x7B (123)
    *****************************************************************************/
    UCB0I2COA |= 0x007B;

    /** Interrupt Enable Register ***********************************************
    * Interrupt disabled for Not-acknowledge, Start/Stop condition and
    * Arbitration lost
    *****************************************************************************/
    UCB0I2CIE = 0x00;

}

void i2c_write(uint8_t slave_address, uint8_t data_length, uint8_t *data)
{
    uint8_t i;

    UCB0CTL1 |= UCTR;                                   // set transmitter-mode
    UCB0I2CSA = slave_address;                          // set slave address
    UCB0CTL1 &= ~UCSWRST;                               // unset SWRESET
    UCB0CTL1 |= UCTXSTT;                                // send START-CON
    UCB0TXBUF = data[0];                                // then write data

    for(i = 1; i < data_length; i++) {
        while(1) { // wait until ...
            /*if((UCB0STAT & UCNACKIFG) == UCNACKIFG)     // NACK from slave
            {
                UCB0CTL1 |= UCTXSTP;                    // then send STOP-CON
                UCB0STAT &= ~UCNACKIFG;                 // reset flag
                UCB0CTL1 |= UCSWRST;                    // set SWRESET
                break;
            }*/

            if((IFG2 & UCB0TXIFG) == UCB0TXIFG)         // data / start-con was send
            {
                UCB0TXBUF = data[i];                    // then write data
                break;
```

```
84                  }
85              }
86          }
87
88          while((IFG2 & UCB0TXIFG) != UCB0TXIFG);             // wait until data/start-con was send
89
90          UCB0CTL1 |= UCTXSTP;                                // send STOP-COND
91          while((UCB0CTL1 & UCTXSTP) == UCTXSTP);             // wait until STOP-con was send
92
93          UCB0CTL1 |= UCSWRST;                                // set SWRESET
94      }
95
96      void i2c_read(uint8_t slave_address, uint8_t data_length, uint8_t *data)
97      {
98          uint8_t i;
99
100         UCB0CTL1 &= ~UCTR;          // reset transmitter-mode
101         UCB0I2CSA = slave_address;  // set slave address of sensor
102         UCB0CTL1 &= ~UCSWRST;       // unset SWRESET
103         UCB0CTL1 |= UCTXSTT;        // send START-CON
104
105         if(data_length > 1)
106         {
107             for(i = 0; i < data_length; i++)
108             {
109                 while(1)
110                 {
111                     /*****************************************************************
112                     * IFG2 = Interrupt Flag Register 2
113                     *****************************************************************/
114                     if((IFG2 & UCB0RXIFG) == UCB0RXIFG) // data / start-con was send
115                     {
116                         data[i] = UCB0RXBUF;            // readout data
117                         if(i == data_length-2)          // if next byte will be the last one
118                             UCB0CTL1 |= UCTXSTP;        // send STOP-CON after next receive
119
120                         break;
121                     }
122                 }
123             }
124         }
125         else
126         {
127             while((UCB0CTL1 & UCTXSTT) == UCTXSTT); // wait for acknowledge of slave,
128             UCB0CTL1 |= UCTXSTP; // then send STOP-COND immediately
129
130             data[0] = UCB0RXBUF; // readout data
131         }
132
133         while((UCB0CTL1 & UCTXSTP) == UCTXSTP); // wait until STOP-con was send
134         UCB0CTL1 |= UCSWRST; // set SWRESET
135     }
```

F.1.16. i2c.h

```
1  /*******************************************************************************
2  **     Description:     i2c.h
3  **     Hardware:        BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **     Date:            06/03/2013
5  **     Author:          Nico Sassano
6  **                      modified taken from Niels Jegenhorst
7  *******************************************************************************/
8  #ifndef I2C_H_
9  #define I2C_H_
10
11 #include "main.h"
12
13 /*******************************************************************************
14 *  Defines
15 *******************************************************************************/
16
17 #define LED_1_PxDIR      (P5DIR)
18 #define LED_1_PxOUT      (P5OUT)
19 #define LED_1_PIN        (BIT4)
20
21 #define LED_2_PxDIR      (P5DIR)
22 #define LED_2_PxOUT      (P5OUT)
23 #define LED_2_PIN        (BIT3)
24
25 #define LED_3_PxDIR      (P5DIR)
26 #define LED_3_PxOUT      (P5OUT)
27 #define LED_3_PIN        (BIT2)
28
29 /*******************************************************************************
30 *  Prototyping declaration
31 *******************************************************************************/
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 /* I2C_H_ */
```

F.1.17. init.c

```
/*******************************************************************************
**    Description:    init.c
**    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
**    Date:           09/04/2013
**    Last Update:    18/07/2013
**    Author:         Nico Sassano
*******************************************************************************/

#include "main.h"

extern volatile uint8_t brate_start;
extern volatile uint8_t brate;

extern volatile uint16_t upper_alarm_temp;
extern volatile uint16_t lower_alarm_temp;

extern volatile uint8_t irq_alert;        // TMP102 Alarm

void init(void) {

    TPS61201_ENABLE;

    // LEDs initialisieren
    led_init();
    i2c_init();
    // Test Port als Ausgang konfigurieren
    TEST_PORT_PxDIR |= TEST_PORT_PIN;

    // Alles LED leuchten 3 Sekunden beim start
    led_on(LED_ALL);
    delay_ms(3000);
    led_off(LED_ALL);

    brate = brate_start;

    // RF-switch
    adg918_init();
    adg918_wakeup();    // Connect the loop antenna with the wakeup circuit

    // Balancing unit
    balancing_init();
    balancing_off();    // Balancing unit off

    /***********************************
    ** Temperatursensor konfigurieren
    ***********************************/
    temp_sensor_init();
    temp_sensor_config_reg();
    temp_sensor_set_alert(lower_alarm_temp, upper_alarm_temp);
    IRQ_ALERT_SET_LOW;      // normale Temperaur setzen
    temp_sensor_get_contr_reg();

    /***********************************
    ** Timer konfigurieren
    ***********************************/
    timer_a_init();
    timer_b_init();

    // CC1101 transceiver
    cc1101_init();
    cc1101_power_up_reset();

    cc1101_sleep();             // Power down state

    // Configure packet received interrupt
    CC1101_GDO2_CLEAR_IRQ;
    CC1101_GDO2_IRQ_DISABLE;

    // LF wakeup receiver
    as3930_init();
    as3930_preset_default();    // Reset
    as3930_config_no_pattern(); // Wakeup upon LF carrier detection
    as3930_clear_wakeup();      // Clear wakeup

    // Configure Wake interrupt
    AS3930_WAKE_CLEAR_IRQ;
    AS3930_WAKE_IRQ_ENABLE;

    // Global interrupt enable
    _EINT();
    // Enter Low Power Mode 4 (LPM4) with all clocks disabled
    ENTER_LPM4;

```

```
84  }
85
86  void init_for_sleep(void) {
87
88      // RF-switch
89      adg918_init();
90      adg918_wakeup();     // Connect the loop antenna with the wakeup circuit
91
92      balancing_off();     // Balancing unit off
93
94      // CC1101 transceiver
95      cc1101_init();
96      cc1101_power_up_reset();
97      cc1101_sleep();              // Power down state
98
99      // Configure packet received interrupt
100     CC1101_GDO2_CLEAR_IRQ;
101     CC1101_GDO2_IRQ_DISABLE;
102
103     // LF wakeup receiver
104     as3930_init();
105     as3930_preset_default();     // Reset
106     as3930_config_no_pattern(); // Wakeup upon LF carrier detection
107     as3930_clear_wakeup();       // Clear wakeup
108
109     // Configure Wake interrupt
110     AS3930_WAKE_CLEAR_IRQ;
111     AS3930_WAKE_IRQ_ENABLE;
112
113     // Global interrupt enable
114     _EINT();
115     // Enter Low Power Mode 4 (LPM4) with all clocks disabled
116     ENTER_LPM4;
117
118 }
```

F.1.18. init.h

```
1  /*******************************************************************************
2  **    Description:    init.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           09/04/2013
5  **    Author:         Nico Sassano
6  *******************************************************************************/
7
8  #ifndef INIT_H_
9  #define INIT_H_
10
11 void init(void);
12 void init_for_sleep(void);
13
14 #endif /* INIT_H_ */
```

F.1.19. isr.c

```
1  /*******************************************************************************
2  **    Description:    isr.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           09/04/2013
5  **    Last Update:    18/07/2013
6  **    Author:         Nico Sassano
7  *******************************************************************************/
8
9  #include "main.h"
10
11
12 extern volatile uint8_t wakeup_state;
13
14 extern volatile uint8_t state;
15 /*********************************************
16 ** CLK
17 *********************************************/
18 extern volatile uint8_t clk_set;
19
20 /*********************************************
21 ** RX
22 *********************************************/
23 extern volatile uint8_t rx_command;
24 extern volatile uint8_t rx_data_length;
25
26 /*********************************************
27 ** Balancing
28 *********************************************/
29 extern volatile uint16_t balancing_volt;    // Zielwert der Balancierung
30 extern volatile uint16_t balancing_time;
31 extern volatile uint8_t balanc_state;
32 extern volatile uint8_t temp_state;
33 extern volatile uint16_t upper_balanc_temp;
34 extern volatile uint16_t lower_balanc_temp;
35 extern volatile uint16_t upper_alarm_temp;
36 extern volatile uint16_t lower_alarm_temp;
37
38 /*********************************************
39 ** Data buffer
40 *********************************************/
41 uint16_t config_value;
42 extern volatile uint16_t sample_burst_buf[500];
43 extern volatile uint16_t sample_buf_volt;   // The latest cell voltage sample
44
45 /*********************************************
46 ** Kalibrierung
47 *********************************************/
48 extern volatile uint8_t cali_pos_offset_tmp102;
49 extern volatile uint8_t cali_neg_offset_tmp102;
50 extern volatile uint8_t cali_pos_offset_msp430;
51 extern volatile uint8_t cali_neg_offset_msp430;
52 extern volatile uint8_t cali_pos_offset_adc;
53 extern volatile uint8_t cali_neg_offset_adc;
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 uint8_t frame_number;
62 extern volatile uint8_t burst_frame_lenght; // 50 -> 25 Werten
63 extern volatile uint8_t burst_error;
64 extern volatile uint8_t burst_error_flag;
65
66 /*********************************************
67 ** Interruptflags
68 *********************************************/
69 extern volatile uint8_t irq_mode;           // PORT1 IRQ Status
70 extern volatile uint8_t irq_send_done_flag; // CC1101 senden
71 extern volatile uint8_t irq_alert;          // TMP102 Alarm
72 extern volatile uint8_t irq_timera;         // Status TimerA
73 extern volatile uint8_t irq_timerb;         // Status TimerB
74
75
76 /*********************************************
77 ** Communication states
78 *********************************************/
79 extern volatile uint8_t command_recived;
80 extern volatile uint8_t wakeup_state;
81 extern volatile uint8_t brate_start;
82 extern volatile uint8_t brate;
83
```

```
84  /*********************************************
85  ** DCO Error
86  *********************************************/
87  volatile uint16_t dco_error1 = 0;
88  volatile uint16_t dco_error2 = 0;
89
90  /*****************************************************************************************************
91  **
92  *****************************************************************************************************/
93  interrupt (PORT1_VECTOR) isr_port1(void) {
94
95      /*****************************************************
96      ** IRQ Burstmode
97      *****************************************************/
98      if ((CC1101_GDO2_IRQ_PENDING) && (IRQ_IS_BURST)) {
99
100         // Interrupt disablen
101         CC1101_GDO2_IRQ_DISABLE;
102         CC1101_GDO2_CLEAR_IRQ;
103
104         TIMER_B_STOP;
105
106         sample_burst_buf[burst_counter] = adc12_get_volt_sample(clk_set);
107         burst_counter++;
108
109         if(burst_freq == BURST_FREQ_2000HZ || burst_freq == BURST_FREQ_4000HZ || burst_freq == BURST_FREQ_6000HZ
110                 || burst_freq == BURST_FREQ_8000HZ || burst_freq == BURST_FREQ_10000HZ || burst_freq == BURST_FREQ_7500HZ) {
111             CC1101_GDO2_CLEAR_IRQ;
112             CC1101_GDO2_IRQ_ENABLE;
113             //Flag zurücksetzen
114             TIMER_B_FLAG_RESET;
115         }
116
117         // Timer einstellen
118         timer_b_init_burst(burst_freq);
119
120         // Werte für DCO Fehlerekennung abrufen
121         dco_error2 = dco_error1;
122         dco_error1 = TBR;
123
124         // Timer starten
125         TIMER_B_START_UP_MODE;
126
127
128     } else if (AS3930_WAKE_IRQ_PENDING) {       // Wake interrupt occured
129     /*****************************************************
130     ** IRQ Wakeup empfangen
131     *****************************************************/
132         // Exit Low Power Mode 4
133         EXIT_LPM4;
134
135         // Disable and clear the wake interrupt as the sensor is awake now
136         AS3930_WAKE_IRQ_DISABLE;
137         AS3930_WAKE_CLEAR_IRQ;
138
139         // Turn on the red LED
140         led_on(LED_AWAKE);
141         //TIMER_B_START;
142
143         // Change to RX state
144         adg918_transceiver();           // Connect the loop antenne with the transceiver
145         adc12_volt_init(clk_set);       // Initialize ADC
146
147         state = S_WAKEUP;
148
149         rx_data_length = 0;             // Standart Datenlänge empangen
150         cc1101_set_rx(brate_start);
151         rx(HEADER_LENGTH + rx_data_length);
152
153     } else if ((CC1101_GDO2_IRQ_PENDING) && (IRQ_IS_RX)) {  // Sync word detected
154     /*****************************************************
155     ** IRQ Packet empfangen
156     *****************************************************/
157         _DINT();                // Globale Interrups ausschalten
158
159         CC1101_GDO2_IRQ_DISABLE;
160         CC1101_GDO2_CLEAR_IRQ;
161
162         rx_command = COMMAND_DOWNLINK_UNKWOWN;
163         command_recived = 1;
164         // Wait until packet completely received
165         // while(CC1101_GDO2_IN);
166         delay_ms(10);
167
168         // Check whether received data is valid
```

```
169         if ((CC1101_GDO0_IN == 1) && (cc1101_get_rxbytes() == HEADER_LENGTH + rx_data_length)) {
170
171             uint8_t rxbuf[HEADER_LENGTH + rx_data_length];
172
173             cc1101_read_rx_fifo(rxbuf, HEADER_LENGTH + rx_data_length);
174             rx_data_length = 0; // Datenlänge wieder zurücksetzen
175
176             if (rxbuf[0] == BROADCAST || rxbuf[0] == ADDRESS_THIS_SENSOR) {
177
178                 led_on(LED_RX);          // Empfangs LED an
179
180                 rx_command = rxbuf[1];   // Empfangenes Kommando
181
182                 switch(rx_command) {
183                     /***********************************************
184                      ** Dekodierung des WAKEUP Kommandos
185                     ** Paketzusammenstellung:
186                     ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
187                     ***********************************************/
188                     case COMMAND_WAKEUP: {
189                         state = S_WAKEUP_RX;
190                         wakeup_state = 1;
191                     }break;
192
193                     /***********************************************
194                     ** Dekodierung des WAKEUP_DONE Kommandos
195                     ** Paketzusammenstellung:
196                     ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
197                     ***********************************************/
198                     case COMMAND_WAKEUP_DONE: {
199                         state = S_WAKEUP_DONE;
200                         if(!wakeup_state)
201                             rx_command = COMMAND_DOWNLINK_SLEEP;
202
203                     }break;
204
205                     /***********************************************
206                     ** Dekodierung des CONFIG_SET Kommandos
207                     ** Paketzusammenstellung:
208                     ** ( Adr. ZS | Kommando | Anzahl | 0x00 )
209                     ***********************************************/
210                     case COMMAND_DOWNLINK_CONFIG_SET: {
211                         rx_data_length = rxbuf[2];  // Länge der nächsten Datensendung
212
213                     }break;
214
215                     /***********************************************
216                     ** Dekodierung des CONFIG Kommandos
217                     ** Paketzusammenstellung:
218                     ** ( Adr. ZS | Kommando | 0x00 | 0x00 )
219                     **      1          2                3                4              5
220                     ** ( Brate | oberer Bal. Wert | unterer Bal. Wert | oberer Alarmwert | unterer Alarmwert | Burst Framelänge)
221                     ***********************************************/
222                     case COMMAND_DOWNLINK_CONFIG: {
223                         rx_data_length  = 0; // Datenlänge zurücksetzen
224                         uint16_t msb    = 0;
225                         uint16_t lsb    = 0;
226
227                         // Übertragungsrate
228                         brate               = rxbuf[HEADER_LENGTH + 0];
229
230                         // oberer Balancierungswert
231                         msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 1] & 0x0F)) << 8) & 0xFF00;
232                         lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 2] & 0xFF)) << 0) & 0x00FF;
233                         upper_balanc_temp   = msb | lsb;
234                         msb = 0;
235                         lsb = 0;
236
237                         // unterer Balancierungswert
238                         msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 3] & 0x0F)) << 8) & 0xFF00;
239                         lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 4] & 0xFF)) << 0) & 0x00FF;
240                         lower_balanc_temp   = msb | lsb;
241                         msb = 0;
242                         lsb = 0;
243
244                         // oberer Alarmwert
245                         msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 5] & 0x0F)) << 8) & 0xFF00;
246                         lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 6] & 0xFF)) << 0) & 0x00FF;
247                         upper_alarm_temp    = msb | lsb;
248                         msb = 0;
249                         lsb = 0;
250
251                         // unterer Alarmwert
252                         msb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 7] & 0x0F)) << 8) & 0xFF00;
253                         lsb                 = (((uint16_t)(rxbuf[HEADER_LENGTH + 8] & 0xFF)) << 0) & 0x00FF;
```

```
254                         lower_alarm_temp     = msb | lsb;
255
256                         // Burst Framelänge
257                         burst_frame_lenght   = rxbuf[HEADER_LENGTH + 9];
258
259                     }break;
260
261                     /***********************************************
262                     ** Dekodierung des Balancierungsheader
263                     ** Paketzusammenstellung:
264                     ** ( Adr. ZS | Kommando | VOLT_MSB | VOLT_LSB )
265                     ***********************************************/
266                     case COMMAND_DOWNLINK_BALANCING_ON: {
267                         uint16_t volt_msb = 0;
268                         uint16_t volt_lsb = 0;
269
270                         volt_msb = (((uint16_t)(rxbuf[3] & 0x0F)) << 8) & 0xFF00;
271                         volt_lsb = (((uint16_t)(rxbuf[4] & 0xFF)) << 0) & 0x00FF;
272
273                         balancing_volt = volt_msb | volt_lsb;
274                         balancing_volt = 2620;  //TEST
275
276                     }break;
277
278                     /***********************************************
279                     ** Dekodierung des Burstheader
280                     ** Paketzusammenstellung:
281                     ** ( Adr. ZS | Kommando | Burst Freq | Anzahl Werte )
282                     ***********************************************/
283                     case COMMAND_DOWNLINK_BURST_MODE: {
284                         burst_freq          = rxbuf[2];
285                         uint8_t burst_value = rxbuf[3];
286
287                         // Wie viele Messwerte sollen aufgenommen werden?
288                        switch(burst_value) {
289                             case BURST_VALUES_50:
290                                 burst_values = 50;
291                             break;
292
293                             case BURST_VALUES_100:
294                                 burst_values = 100;
295                             break;
296
297                             case BURST_VALUES_150:
298                                 burst_values = 150;
299                             break;
300
301                             case BURST_VALUES_200:
302                                 burst_values = 200;
303                             break;
304
305                             case BURST_VALUES_250:
306                                 burst_values = 250;
307                             break;
308
309                             case BURST_VALUES_300:
310                                 burst_values = 300;
311                             break;
312
313                             case BURST_VALUES_350:
314                                 burst_values = 350;
315                             break;
316
317                             case BURST_VALUES_400:
318                                 burst_values = 400;
319                             break;
320
321                             case BURST_VALUES_450:
322                                 burst_values = 450;
323                             break;
324
325                             case BURST_VALUES_500:
326                                 burst_values = 500;
327                             break;
328
329                             case BURST_VALUES_550:
330                                 burst_values = 550;
331                             break;
332
333                             case BURST_VALUES_600:
334                                 burst_values = 600;
335                             break;
336
337                             case BURST_VALUES_650:
338                                 burst_values = 650;
```

```
339                             break;
340
341                             case BURST_VALUES_700:
342                                 burst_values = 700;
343                             break;
344
345                             case BURST_VALUES_750:
346                                 burst_values = 750;
347                             break;
348
349                             case BURST_VALUES_800:
350                                 burst_values = 800;
351                             break;
352
353                             case BURST_VALUES_850:
354                                 burst_values = 850;
355                             break;
356
357                             case BURST_VALUES_900:
358                                 burst_values = 900;
359                             break;
360                         }
361
362                     }break;
363
364                     /***********************************************
365                     ** Dekodierung des Burstheader
366                     ** Paketzusammenstellung:
367                     ** ( Adr. ZS | Kommando | Frame Nummer | 0x00 )
368                     ***********************************************/
369                     case COMMAND_DOWNLINK_BURST_DATA_RX: {
370                         frame_number       = rxbuf[2];
371
372                     }break;
373
374                     /***********************************************
375                     ** Dekodierung des Kalibrierungsheader: TMP102
376                     ** Paketzusammenstellung:
377                     ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
378                     ***********************************************/
379                     case COMMAND_DOWNLINK_CALIBRATION_TMP102: {
380                         cali_pos_offset_tmp102 = rxbuf[2];
381                         cali_neg_offset_tmp102 = rxbuf[3];
382                     }break;
383
384                     /***********************************************
385                     ** Dekodierung des Kalibrierungsheader: MSP430
386                     ** Paketzusammenstellung:
387                     ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
388                     ***********************************************/
389                     case COMMAND_DOWNLINK_CALIBRATION_MSP430: {
390                         cali_pos_offset_msp430 = rxbuf[2];
391                         cali_neg_offset_msp430 = rxbuf[3];
392                     }break;
393
394                     /***********************************************
395                     ** Dekodierung des Kalibrierungsheader: ADC
396                     ** Paketzusammenstellung:
397                     ** ( Adr. ZS | Kommando | Pos. Offset | Neg. Offset )
398                     ***********************************************/
399                     case COMMAND_DOWNLINK_CALIBRATION_ADC: {
400                         cali_pos_offset_adc = rxbuf[2];
401                         cali_neg_offset_adc = rxbuf[3];
402                     }break;
403
404                     default: break;
405                 }
406             }
407             led_off(LED_RX);
408         }
409         _EINT();                // Globale Interrups einschalten
410
411     } else if ((CC1101_GDO2_IRQ_PENDING) && (IRQ_IS_TX)) {
412         CC1101_GDO2_IRQ_DISABLE;
413         CC1101_GDO2_CLEAR_IRQ;
414         irq_send_done_flag = 1;
415     }
416 }
417
418 /***********************************************************************
419 ** IRQ Temperatursensor TMP102
420 ** Alarmausgang
421 ***********************************************************************/
422 interrupt (PORT2_VECTOR) isr_port2(void) {
423
```

```
424         if(TMP102_ALERT_IRQ_PENDING) {
425
426             if(IRQ_ALERT_IS_LOW) {          // Anfrage Alarm auslösen
427
428                 TMP102_ALERT_IRQ_DISABLE;   // Interrupt stoppen
429                 TMP102_ALERT_CLEAR_IRQ;
430
431                 //led_off(LED_AWAKE);
432                 IRQ_ALERT_SET_HIGH;         // Alarm setzen
433
434                 TMP102_SET_IRQ_RISING_EDGE; // IRQ auf steigende FInke setzen
435                 TMP102_ALERT_CLEAR_IRQ;
436                 TMP102_ALERT_IRQ_ENABLE;
437
438             }else if(IRQ_ALERT_IS_HIGH) {   // Anfrage Ararm zurücksetzen
439
440                 TMP102_ALERT_IRQ_DISABLE;   // Interrupt stoppen
441                 TMP102_ALERT_CLEAR_IRQ;
442
443                 //led_on(LED_AWAKE);
444                 IRQ_ALERT_SET_LOW;          // Alarm löschen
445                 TMP102_SET_IRQ_FALLING_EDGE;// IRQ auf fallende Flanke setzen
446
447                 TMP102_ALERT_CLEAR_IRQ;
448                 TMP102_ALERT_IRQ_ENABLE;
449             }
450         }
451     }
452
453     /*************************************************************************
454     ** ISR TIMERA0: Balancierung wird alle 500ms ausgelöst
455     *************************************************************************/
456     interrupt ( TIMERA0_VECTOR ) isr_timer_A0 (void) {
457
458         if (IRQ_TIMERA_IS_BALANCING) {
459
460             // Interrupt GDO2 sperren
461             CC1101_GDO2_IRQ_DISABLE;
462
463             // BAlancierzeit hochzählen
464             balancing_time++;
465
466             // Testweise 10min Balancieren -> 10min * 60sek *2 = 1200
467             if(balancing_time < 1200) {
468
469                 uint16_t actual_volt = 0x0000;
470                 uint16_t actual_temp = 0x0000;
471
472                 // Aktuelle Werte holen
473                 actual_volt = adc12_get_volt_sample(clk_set);
474                 actual_temp = temp_sensor_get_temp();
475
476                 sample_buf_volt = actual_volt; // Spannungswert übernehmen
477
478                 // Temperaturkontrolle
479                 if (TEMP_IS_NORMAL) {                       // Temperatur ist Normal
480                     if (actual_temp > upper_balanc_temp)    // Ist aktuelle Temp. zu hoch?
481                         TEMP_SET_HIGH;
482                 } else if(TEMP_IS_HIGH) {                   // Temperatur ist zu hoch
483                     if (actual_temp < lower_balanc_temp)    // Ist aktuelle Temp. ok?
484                         TEMP_SET_NORMAL;
485                 }
486
487                 // Spannungskontrolle
488                 if (actual_volt > balancing_volt) {
489                     if (TEMP_IS_NORMAL) {
490                         balancing_on();
491                     } else {
492                         balancing_off();
493                     }
494                 } else { // Stop Balancing
495                     TIMER_A_STOP;
496                     balancing_off();
497                     balanc_state = OFF;
498                     IRQ_TIMERA_UNSET_BALANCING;
499                     TIMER_A_0_CM_IRQ_DISABLE;
500                     TIMER_A_1_CM_IRQ_DISABLE;
501                 }
502             } else {
503                 TIMER_A_STOP;
504                 balancing_off();
505                 balanc_state = OFF;
506                 IRQ_TIMERA_UNSET_BALANCING;
507                 TIMER_A_0_CM_IRQ_DISABLE;
508                 TIMER_A_1_CM_IRQ_DISABLE;
```

```
509             }
510
511             if(balanc_state == OFF) {
512                 TIMER_A_0_CM_IRQ_DISABLE;
513                 TIMER_A_1_CM_IRQ_DISABLE;
514             }
515
516             // Interrupt GDO2 freigeben
517             CC1101_GDO2_IRQ_ENABLE;
518         }
519     }
520
521     /**************************************************************************
522     ** ISR TIMERB: Zeitfenster für die Burstmessung
523     **************************************************************************/
524     interrupt ( TIMERB1_VECTOR ) isr_timer_B1 (void) {
525
526         if(IRQ_TIMERB_IS_BURST) {
527             // Interrupt enabeln
528             CC1101_GDO2_CLEAR_IRQ;
529             CC1101_GDO2_IRQ_ENABLE;
530
531             //Flag zurücksetzen
532             TIMER_B_FLAG_RESET;
533         }
534     }
535
536     /**************************************************************************
537     ** ISR TIMERB: Zeitfenster für die Burstmessung
538     **************************************************************************/
539     interrupt ( TIMERB0_VECTOR ) isr_timer_B0 (void) {
540
541         if(IRQ_TIMERB_IS_BURST) {
542             if(burst_counter >= burst_values) { //Sind alles Werte durchgekommen
543                 CC1101_GDO2_IRQ_DISABLE;
544                 CC1101_GDO2_CLEAR_IRQ;
545
546                 TIMER_B_STOP;
547                 TIMER_B_0_CM_IRQ_DISABLE;
548                 TIMER_B_1_CM_IRQ_DISABLE;
549
550                 burst_error = 0;     // Burst error zurücksetzen
551
552                 //Flag zurücksetzen
553                 TIMER_B_FLAG_RESET;
554
555                 IRQ_TIMERB_UNSET_BURST;
556                 rx_command = COMMAND_BACK_FROM_BURST;
557                 command_recived = 1;
558
559             } else { //Fehler dedektiert
560
561                 BURST_ERROR_FLAG_SET;        // Fehlerflag setzen
562
563                 // Interrupt disablen
564                 CC1101_GDO2_IRQ_DISABLE;
565                 CC1101_GDO2_CLEAR_IRQ;
566
567                 // Hier keine Fensterung, da Frequenz zu schnell
568                 if(burst_freq == BURST_FREQ_2000HZ || burst_freq == BURST_FREQ_4000HZ ||burst_freq == BURST_FREQ_6000HZ
569                         || burst_freq == BURST_FREQ_8000HZ || burst_freq == BURST_FREQ_10000HZ || burst_freq == BURST_FREQ_7500HZ) {
570                     CC1101_GDO2_CLEAR_IRQ;
571                     CC1101_GDO2_IRQ_ENABLE;
572
573                     //Flag zurücksetzen
574                     TIMER_B_FLAG_RESET;
575                 }
576
577                 // Timer einstellen
578                 timer_b_init_burst(burst_freq);
579
580                 // Timer starten
581                 TIMER_B_START_UP_MODE;
582
583                 // Fehlerwert speichern
584                 sample_burst_buf[burst_counter] = 0xFFFF;    // Fehlerwert
585                 burst_counter++;
586                 burst_error++;
587             }
588         }
589     }
```

F.1.20. led.c

```
1   /*-------------------------------------------------------------------
2       Description:    LED driver.
3       Date:           11/22/2012
4       Last Update:    11/22/2012
5       Author:         Phillip Durdaut
6   -------------------------------------------------------------------*/
7
8   #include "main.h"
9
10  /*-------------------------------------------------------------------
11    Public functions
12  -------------------------------------------------------------------*/
13
14  void led_init(void)
15  {
16      LED_1_PxDIR |= LED_1_PIN;
17      LED_2_PxDIR |= LED_2_PIN;
18      LED_3_PxDIR |= LED_3_PIN;
19
20      led_off(LED_ALL);
21  }
22
23  void led_on(LED_t led)
24  {
25  #ifdef ENABLE_LEDS
26      switch(led) {
27          case LED_AWAKE: LED_1_PxOUT |= LED_1_PIN; break;
28          case LED_TX:    LED_2_PxOUT |= LED_2_PIN; break;
29          case LED_RX:    LED_3_PxOUT |= LED_3_PIN; break;
30          case LED_ALL:   led_on(LED_AWAKE);
31                          led_on(LED_TX);
32                          led_on(LED_RX);
33                          break;
34          default: break;
35      }
36  #endif /* ENABLE_LEDS */
37  }
38
39  void led_off(LED_t led)
40  {
41      switch(led) {
42          case LED_AWAKE: LED_1_PxOUT &= ~LED_1_PIN; break;
43          case LED_TX:    LED_2_PxOUT &= ~LED_2_PIN; break;
44          case LED_RX:    LED_3_PxOUT &= ~LED_3_PIN; break;
45          case LED_ALL:   led_off(LED_AWAKE);
46                          led_off(LED_TX);
47                          led_off(LED_RX);
48                          break;
49          default: break;
50      }
51  }
52
53  void led_toggle(LED_t led)
54  {
55  #ifdef ENABLE_LEDS
56      switch(led) {
57          case LED_AWAKE: LED_1_PxOUT ^= LED_1_PIN; break;
58          case LED_TX:    LED_2_PxOUT ^= LED_2_PIN; break;
59          case LED_RX:    LED_3_PxOUT ^= LED_3_PIN; break;
60          case LED_ALL:   led_toggle(LED_AWAKE);
61                          led_toggle(LED_TX);
62                          led_toggle(LED_RX);
63                          break;
64          default: break;
65      }
66  #endif /* ENABLE_LEDS */
67  }
```

F.1.21. led.h

```
1  /*-------------------------------------------------------------------
2      Description:    LED driver.
3      Date:           11/22/2012
4      Last Update:    11/22/2012
5      Author:         Phillip Durdaut
6  -------------------------------------------------------------------*/
7
8  #ifndef LED_H_
9  #define LED_H_
10
11 #include "main.h"
12
13 /*-------------------------------------------------------------------
14   Defines
15 -------------------------------------------------------------------*/
16
17 #define LED_1_PxDIR      (P5DIR)
18 #define LED_1_PxOUT      (P5OUT)
19 #define LED_1_PIN        (BIT4)
20
21 #define LED_2_PxDIR      (P5DIR)
22 #define LED_2_PxOUT      (P5OUT)
23 #define LED_2_PIN        (BIT3)
24
25 #define LED_3_PxDIR      (P5DIR)
26 #define LED_3_PxOUT      (P5OUT)
27 #define LED_3_PIN        (BIT2)
28
29 /*-------------------------------------------------------------------
30   Types
31 -------------------------------------------------------------------*/
32
33 typedef enum
34 {
35   LED_AWAKE = 0,
36   LED_TX,
37   LED_RX,
38   LED_ALL
39 } LED_t;
40
41 /*-------------------------------------------------------------------
42   Public functions
43 -------------------------------------------------------------------*/
44
45 void led_init(void);
46 void led_on(LED_t led);
47 void led_off(LED_t led);
48 void led_toggle(LED_t led);
49
50 #endif /* LED_H_ */
```

F.1.22. main.c

```
1  /*****************************************************************************
2  **    Description:    main.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *****************************************************************************/
8
9  #include "main.h"
10
11 volatile uint8_t state = 0;
12
13 /*********************************************
14 ** CLK
15 *********************************************/
16 uint8_t clk_set = 1;
17
18 /*********************************************
19 ** RX
20 *********************************************/
21 volatile uint8_t rx_command = COMMAND_WAIT;
22 volatile uint8_t rx_data_length         = 0;
23 volatile uint8_t tx_data_length         = 0;
24
25 /*********************************************
26 ** Balancing
27 *********************************************/
28 volatile uint16_t balancing_volt        = 0x500;  // Zielwert der Balancierung
29 volatile uint16_t balancing_time        = 0x000;
30 volatile uint8_t balanc_state           = OFF;
31 volatile uint8_t temp_state             = TEMP_NORMAL;
32 volatile uint16_t upper_balanc_temp     = 0x250;
33 volatile uint16_t lower_balanc_temp     = 0x200;
34 volatile uint16_t upper_alarm_temp      = 0x1A00;
35 volatile uint16_t lower_alarm_temp      = 0x1900;
36
37 /*********************************************
38 ** Data buffer
39 *********************************************/
40 uint16_t config_value;
41 volatile uint16_t sample_burst_buf[900];
42 volatile uint16_t sample_buf_volt;          // The latest cell voltage sample
43 volatile uint16_t sample_buf_temp;
44 volatile uint16_t sample_buf_temp_msp;
45
46 /*********************************************
47 ** Burst Mode
48 *********************************************/
49 volatile uint8_t burst_freq             = 0;    // Burst Frequenz
50 volatile uint16_t burst_values          = 0;    // Anzahl der erwarteten Burst Werte
51 volatile uint16_t burst_counter         = 0;    // Counter der Burst-Werte
52 volatile uint8_t burst_frame_counter    = 0;
53 volatile uint8_t burst_frame_lenght     = 0;   // 50 -> 25 Werten
54 volatile uint8_t frame_number           = 0;
55 volatile uint8_t burst_error            = 0;
56 volatile uint8_t burst_error_flag       = 0;
57
58 /*********************************************
59 ** Kalibrierung
60 *********************************************/
61 volatile uint8_t cali_pos_offset_tmp102 = 0;
62 volatile uint8_t cali_neg_offset_tmp102 = 0;
63 volatile uint8_t cali_pos_offset_msp430 = 0;
64 volatile uint8_t cali_neg_offset_msp430 = 0;
65 volatile uint8_t cali_pos_offset_adc    = 0;
66 volatile uint8_t cali_neg_offset_adc    = 0;
67
68
69 /*********************************************
70 ** Interruptflags
71 *********************************************/
72 volatile uint8_t irq_mode               = TX_MODE;  // PORT1 IRQ Status
73 volatile uint8_t irq_send_done_flag     = 0;        // CC1101 senden
74 volatile uint8_t irq_alert              = 0;        // TMP102 Alarm
75 volatile uint8_t irq_timera             = 0;        // Status TimerA
76 volatile uint8_t irq_timerb             = 0;        // Status TimerB
77
78 /*********************************************
79 ** Communication states
80 *********************************************/
81 volatile uint8_t command_recived        = 0;
82 volatile uint8_t wakeup_state           = 0x00;
83
```

```
84  volatile uint8_t brate_start            = 40;
85  volatile uint8_t brate                  = 40;
86
87  void tx_carrier(void);
88  void tx_packet(uint8_t, uint8_t *);
89
90
91
92  int main(void) {
93      clk_init_startup();
94
95      P5SEL |= BIT5;
96      P5DIR |= BIT5;
97
98      /*********************************************************
99       ** Initialisierung
100      *********************************************************/
101     init();
102
103     /*********************************************************
104     ** Endless loop
105     **
106     *********************************************************/
107     while (1) {
108
109         if(command_recived) {// Kommando wurde Empfangen
110             // Determine what the base station wants this sensor to do now
111             switch(rx_command) {
112                 /*************************************************************
113                 ** Unbekanntes Kommando empfangen
114                 *************************************************************/
115                 case COMMAND_DOWNLINK_UNKWOWN: {
116
117                     rx(HEADER_LENGTH + rx_data_length);
118                 } break;
119
120                 /*************************************************************
121                 ** Kommando zum WAKEUP empfangen
122                 *************************************************************/
123                 case COMMAND_WAKEUP: {
124                     state = S_WAKEUP_RX;
125
126                     rx(HEADER_LENGTH + rx_data_length);
127                 } break;
128                 /*************************************************************
129                 ** Kommando WAKUP DONE empfangen
130                 *************************************************************/
131                 case COMMAND_WAKEUP_DONE: {
132                     state = S_WAKEUP_DONE;
133
134                     rx(HEADER_LENGTH + rx_data_length);
135                 } break;
136
137                 /*************************************************************
138                 ** Senden des AWAKE Test
139                 *************************************************************/
140                 case COMMAND_DOWNLINK_IS_AWAKE: {
141
142                     uint8_t packet[HEADER_LENGTH] = {0};
143
144                     packet[0] = ADDRESS_BASE_STATION;
145                     packet[1] = ADDRESS_THIS_SENSOR;
146                     packet[2] = 0x01;
147                     packet[3] = 0x00;
148
149                     tx_packet(HEADER_LENGTH, packet);
150
151                     rx(HEADER_LENGTH + rx_data_length);
152
153                 } break;
154
155                 /*************************************************************
156                 ** Einstellung zum Empfang von Konfigurationen
157                 *************************************************************/
158                 case COMMAND_DOWNLINK_CONFIG_SET: {
159
160                     rx(HEADER_LENGTH + rx_data_length);
161                 } break;
162
163                 /*************************************************************
164                 ** Konfiguration empfangen, danach neu Konfigurieren
165                 *************************************************************/
166                 case COMMAND_DOWNLINK_CONFIG: {
167
168                     /*********************************
```

```
169                     ** Temperatursensor konfigurieren
170                     ************************************/
171                     temp_sensor_set_alert(lower_alarm_temp, upper_alarm_temp);
172
173                     rx(HEADER_LENGTH + rx_data_length);
174                 } break;
175
176                 /************************************************************
177                 ** Kalibrierung des TMP102
178                 ************************************************************/
179                 case COMMAND_DOWNLINK_CALIBRATION_TMP102: {
180
181                     rx(HEADER_LENGTH + rx_data_length);
182                 } break;
183
184                 /************************************************************
185                 ** Kalibrierung des MSP430
186                 ************************************************************/
187                 case COMMAND_DOWNLINK_CALIBRATION_MSP430: {
188
189                     rx(HEADER_LENGTH + rx_data_length);
190                 } break;
191
192                 /************************************************************
193                 ** Aufnahme der einfachen Spannungs- und Temp.-Messung
194                 ************************************************************/
195                 case COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE: {
196
197                     // IRQ TimerA anhalten
198                     // Da Interrupt stört NS 27.06.13
199                     if(IRQ_TIMERA_IS_BALANCING) {
200                         TIMER_A_STOP;
201                         TIMER_A_0_CM_IRQ_DISABLE;
202                     }
203
204                     if (balanc_state == OFF)     //TEST
205                         sample_buf_volt = adc12_get_volt_sample(clk_set);
206
207                     sample_buf_temp = temp_sensor_get_temp();
208                     sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
209                     sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
210
211
212                     //  IRQ TimerA vortsetzen
213                     if(IRQ_TIMERA_IS_BALANCING) {
214                         TIMER_A_0_CM_IRQ_ENABLE;
215                         TIMER_A_START_UP_MODE;
216                     }
217
218                     rx(HEADER_LENGTH + rx_data_length);
219                 } break;
220
221                 /************************************************************
222                 ** Senden der einfachen Spannungs- und Temp.-Messung
223                 ************************************************************/
224                 case COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE: {
225
226                     uint8_t packet[8];
227
228                     packet[0] = ADDRESS_BASE_STATION;
229                     packet[1] = ADDRESS_THIS_SENSOR;
230                     packet[2] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE;
231                     packet[3] = 0x00;
232
233                     packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
234                     packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
235
236                     packet[6] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
237                     packet[7] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
238
239                     tx_packet(8, packet);
240
241                     rx(HEADER_LENGTH + rx_data_length);
242                 } break;
243
244                 /************************************************************
245                 ** Aufnahme der einfachen Spannungs- und Temp.-Messung
246                 ************************************************************/
247                 case COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE_ALL: {
248
249                     // IRQ TimerA anhalten
250                     // Da Interrupt stört
251                     if(IRQ_TIMERA_IS_BALANCING) {
252                         TIMER_A_STOP;
253                         TIMER_A_0_CM_IRQ_DISABLE;
```

```
254                     }
255
256                     sample_buf_volt = adc12_get_volt_sample(clk_set);
257
258                     sample_buf_temp = temp_sensor_get_temp();
259                     sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
260                     sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
261
262                     sample_buf_temp_msp = adc12_get_temp_sample(clk_set);
263
264
265                     //  IRQ TimerA vortsetzen
266                     if(IRQ_TIMERA_IS_BALANCING) {
267                         TIMER_A_0_CM_IRQ_ENABLE;
268                         TIMER_A_START_UP_MODE;
269                     }
270
271                     rx(HEADER_LENGTH + rx_data_length);
272                 } break;
273
274                 /*************************************************************
275                 ** Senden der einfachen Spannungs- und Temp.-Messung
276                 *************************************************************/
277                 case COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE_ALL: {
278
279                     uint8_t packet[8];
280
281                     packet[0] = ADDRESS_BASE_STATION;
282                     packet[1] = ADDRESS_THIS_SENSOR;
283                     packet[2] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE;
284                     packet[3] = 0x00;
285
286                     packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
287                     packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
288
289                     packet[6] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
290                     packet[7] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
291
292                     packet[8] = (uint8_t)((sample_buf_temp_msp >> 8) & 0x0F);
293                     packet[9] = (uint8_t)((sample_buf_temp_msp >> 0) & 0xFF);
294
295                     tx_packet(10, packet);
296
297
298                     rx(HEADER_LENGTH + rx_data_length);
299                 } break;
300
301                 /*************************************************************
302                 ** Aufnahme der einfachen Temperaturmessung
303                 ** mit TMP102
304                 *************************************************************/
305                 case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102: {
306
307                     // IRQ TimerA anhalten
308                     // Da Interrupt stört NS
309                     if(IRQ_TIMERA_IS_BALANCING) {
310                         TIMER_A_STOP;
311                         TIMER_A_0_CM_IRQ_DISABLE;
312                     }
313
314                     sample_buf_temp = temp_sensor_get_temp();
315                     sample_buf_temp = sample_buf_temp + cali_pos_offset_tmp102;
316                     sample_buf_temp = sample_buf_temp - cali_neg_offset_tmp102;
317
318                     //  IRQ TimerA vortsetzen
319                     if(IRQ_TIMERA_IS_BALANCING) {
320                         TIMER_A_0_CM_IRQ_ENABLE;
321                         TIMER_A_START_UP_MODE;
322                     }
323
324                     rx(HEADER_LENGTH + rx_data_length);
325                 } break;
326
327                 /*************************************************************
328                 ** Senden der einfachen Temperaturmessung
329                 ** mit TMP102
330                 *************************************************************/
331                 case COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102: {
332
333                     uint8_t packet[6];
334
335                     packet[0] = ADDRESS_BASE_STATION;
336                     packet[1] = ADDRESS_THIS_SENSOR;
337                     packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102;
338                     packet[3] = 0x00;
```

```
339
340                     packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
341                     packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
342
343                     tx_packet(6, packet);
344
345                     rx(HEADER_LENGTH + rx_data_length);
346                 } break;
347
348                 /*************************************************************
349                 ** Aufnahme der einfachen Temperaturmessung
350                 ** mit TMP102 zur Kalibrierung
351                 *************************************************************/
352                 case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102_CALI: {
353
354                     // IRQ TimerA anhalten
355                     // Da Interrupt stört NS 27.06.13
356                     if(IRQ_TIMERA_IS_BALANCING) {
357                         TIMER_A_STOP;
358                         TIMER_A_0_CM_IRQ_DISABLE;
359                     }
360
361                     sample_buf_temp = temp_sensor_get_temp();
362
363                     //  IRQ TimerA vortsetzen
364                     if(IRQ_TIMERA_IS_BALANCING) {
365                         TIMER_A_0_CM_IRQ_ENABLE;
366                         TIMER_A_START_UP_MODE;
367                     }
368
369                     delay_ms(1);
370
371                     rx(HEADER_LENGTH + rx_data_length);
372                 } break;
373
374                 /*************************************************************
375                 ** Senden der einfachen Temperaturmessung
376                 ** mit TMP102 zur Kalibrierung
377                 *************************************************************/
378                 case COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI: {
379
380                     uint8_t packet[6];
381
382                     packet[0] = ADDRESS_BASE_STATION;
383                     packet[1] = ADDRESS_THIS_SENSOR;
384                     packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI;
385                     packet[3] = 0x00;
386
387                     packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
388                     packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
389
390                     tx_packet(6, packet);
391
392                     rx(HEADER_LENGTH + rx_data_length);
393                 } break;
394
395                 /*************************************************************
396                 ** Aufnahme der einfachen Temperaturmessung
397                 ** mit MSP430 zur Kalibrierung
398                 *************************************************************/
399                 case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430_CALI: {
400
401                     sample_buf_temp_msp = adc12_get_temp_sample(clk_set);
402                     delay_ms(1);
403
404                     rx(HEADER_LENGTH + rx_data_length);
405                 } break;
406
407                 /*************************************************************
408                 ** Senden der einfachen Temperaturmessung
409                 ** mit MSP430 zur Kalibrierung
410                 *************************************************************/
411                 case COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI: {
412
413                     uint8_t packet[6];
414
415                     packet[0] = ADDRESS_BASE_STATION;
416                     packet[1] = ADDRESS_THIS_SENSOR;
417                     packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI;
418                     packet[3] = 0x00;
419
420                     packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
421                     packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
422
423                     tx_packet(6, packet);
```

```
424
425                     rx(HEADER_LENGTH + rx_data_length);
426                 } break;
427
428                 /*************************************************************
429                 ** Aufnahme der einfachen Temperaturmessung
430                 ** mit MSP430
431                 *************************************************************/
432                 case COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430: {
433
434                     sample_buf_temp = adc12_get_temp_sample(clk_set);
435                     sample_buf_temp = sample_buf_temp + cali_pos_offset_msp430;
436                     sample_buf_temp = sample_buf_temp - cali_neg_offset_msp430;
437                     delay_ms(1);
438
439                     rx(HEADER_LENGTH + rx_data_length);
440                 } break;
441
442                 /*************************************************************
443                 ** Senden der einfachen Temperaturmessung
444                 ** mit MSP430
445                 *************************************************************/
446                 case COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430: {
447
448                     uint8_t packet[6];
449
450                     packet[0] = ADDRESS_BASE_STATION;
451                     packet[1] = ADDRESS_THIS_SENSOR;
452                     packet[2] = COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430;
453                     packet[3] = 0x00;
454
455                     packet[4] = (uint8_t)((sample_buf_temp >> 8) & 0x0F);
456                     packet[5] = (uint8_t)((sample_buf_temp >> 0) & 0xFF);
457
458                     tx_packet(6, packet);
459
460                     rx(HEADER_LENGTH + rx_data_length);
461                 } break;
462
463                 /*************************************************************
464                 ** Aufnahme der einfachen Spannungsmessung
465                 *************************************************************/
466                 case COMMAND_DOWNLINK_SAMPLE_VOLTAGE: {
467
468                     sample_buf_volt = adc12_get_volt_sample(clk_set);
469                     delay_ms(1);
470
471
472                     rx(HEADER_LENGTH + rx_data_length);
473                 } break;
474
475                 /*************************************************************
476                 ** Senden der einfachen Spannungsmessung
477                 *************************************************************/
478                 case COMMAND_DOWNLINK_SEND_VOLTAGE: {
479
480
481                     uint8_t packet[6];
482
483                     packet[0] = ADDRESS_BASE_STATION;
484                     packet[1] = ADDRESS_THIS_SENSOR;
485                     packet[2] = 0x00;//COMMAND_DOWNLINK_SEND_VOLTAGE;
486                     packet[3] = 0x00;
487
488                     packet[4] = (uint8_t)((sample_buf_volt >> 8) & 0x0F);
489                     packet[5] = (uint8_t)((sample_buf_volt >> 0) & 0xFF);
490
491                     tx_packet(6, packet);
492
493                     rx(HEADER_LENGTH + rx_data_length);
494                 } break;
495
496                 /*************************************************************
497                 ** Kommando zum Balancing empfangen
498                 *************************************************************/
499                 case COMMAND_DOWNLINK_BALANCING_ON: {
500                     timer_a_init_balanc();  // TimerA wird initialisiert
501                     TIMER_A_START_UP_MODE;
502                     balanc_state = ON;
503
504                     rx(HEADER_LENGTH + rx_data_length);
505                 } break;
506
507                 /*************************************************************
508                 ** Kommando zum Balancing ausschalten empfangen
```

```
509                 ************************************************************/
510                 case COMMAND_DOWNLINK_BALANCING_OFF: {
511                     TIMER_A_STOP;
512                     balancing_off();
513                     balanc_state = OFF;
514
515
516                     rx(HEADER_LENGTH + rx_data_length);
517                 } break;
518
519                 /************************************************************
520                 ** Befehl zur Burstmessung empfangen
521                 ************************************************************/
522                 case COMMAND_DOWNLINK_BURST_MODE: {
523
524                     // DCO auf 16 MHz setzen
525                     // Reihenfolge der Registerzuweisung beachten!
526                     DCOCTL = 0;
527                     BCSCTL1 = CALBC1_16MHZ;
528                     DCOCTL = CALDCO_16MHZ;
529                     clk_set = 16;                   //CLK ist auf 16MHz gesetzt
530
531                     adc12_volt_init(clk_set);   // Initialize ADC
532                     burst_counter = 0;          // Zurücksetzen
533
534                     // Interrupt sperren
535                     CC1101_GDO2_IRQ_DISABLE;
536                     CC1101_GDO2_CLEAR_IRQ;
537
538
539                     // IRQ TimerA anhalten
540                     if(IRQ_TIMERA_IS_BALANCING) {
541                         TIMER_A_STOP;
542                         TIMER_A_0_CM_IRQ_DISABLE;
543                     }
544
545                     cc1101_init_rx();       // GDO2 konfigurieren
546                     cc1101_reset();         // Reset chip and go to idle state
547                     cc1101_config_no_packet_rx();
548                     cc1101_burst_rx();      // Ab hier wird empfangen
549                     IRQ_SET_BURST;          // IRQ auf Burst Mode stellen
550                     delay_ms(32);           // Entspricht bei 16MHz -> 2 ms
551
552
553                     // Interrupt freischalten
554                     CC1101_GDO2_CLEAR_IRQ;
555                     CC1101_GDO2_IRQ_ENABLE;
556                     TEST_PIN_ON;
557
558                     command_recived = 0;
559                 } break;
560
561                 /************************************************************
562                 ** Kommando wenn vorher BURST MODE war, um CLK unzustellen
563                 ************************************************************/
564                 case COMMAND_BACK_FROM_BURST: {
565
566                     _DINT();                // Globale Interrups ausschalten
567                     DCOCTL  = 0;            // Select lowest DCOx and MODx settings
568                     BCSCTL1 = CALBC1_1MHZ;  // Set DCO to 8MHz
569                     DCOCTL  = CALDCO_1MHZ;
570                     _EINT();                // Globale Interrups einschalten
571
572                     // IRQ TimerA fortsetzen
573                     if(IRQ_TIMERA_IS_BALANCING) {
574                         TIMER_A_START_UP_MODE;
575                         TIMER_A_0_CM_IRQ_ENABLE;
576                     }
577
578                     rx(HEADER_LENGTH + rx_data_length);
579                 } break;
580
581                 /************************************************************
582                 ** Leersendung
583                 ************************************************************/
584                 case COMMAND_DOWNLINK_BURST_CHECK: {
585
586                     rx(HEADER_LENGTH + rx_data_length);
587                 } break;
588
589
590                 /************************************************************
591                 ** Anfrage der Burst Daten
592                 ************************************************************/
593                 case COMMAND_DOWNLINK_BURST_DATA_RQ: {
```

```
594                     //Berechnung der Anzahl der Frames
595                     uint16_t index = burst_counter;
596                     burst_counter = 0;          // Zurücksetzen
597                     burst_frame_counter = 0;    // Zurücksetzen
598
599                     while(index >= (burst_frame_lenght/2)) {
600                         burst_frame_counter++;
601                         index = index - (burst_frame_lenght/2);
602                     }
603
604                     uint8_t packet[4];
605
606                     packet[0] = ADDRESS_BASE_STATION;
607                     packet[1] = ADDRESS_THIS_SENSOR;
608                     packet[2] = burst_frame_counter;  // Anzahl der Frames
609                     packet[3] = burst_frame_lenght;   // Länge der Frames ohne Header
610
611
612                     tx_packet(4,packet);
613
614
615                     rx(HEADER_LENGTH + rx_data_length);
616                 } break;
617
618                 /*************************************************************
619                 ** Senden der Burst Daten
620                 *************************************************************/
621                 case COMMAND_DOWNLINK_BURST_DATA_RX: {
622
623                     uint16_t seq_number      = 0;
624                     uint8_t sample_counter   = 0;
625                     uint8_t packages_counter = 0;
626                     uint8_t packet[(HEADER_LENGTH + burst_frame_lenght)];
627
628                     packet[0] = ADDRESS_BASE_STATION;
629                     packet[1] = ADDRESS_THIS_SENSOR;
630                     packet[2] = seq_number;             // Anzahl der Frames
631                     packet[3] = frame_number;           // Länge der Frames ohne Header
632
633                     // Hier werden die zu sendenden Daten berechnet
634                     for(packages_counter=0;packages_counter < burst_frame_lenght;packages_counter++) {
635                         packet[HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(burst_frame_lenght/2))+
                                sample_counter] >> 8) & 0x0F);
636                         seq_number++;       // Sequenznummer hochzählen
637                         packages_counter++;
638                         packet[HEADER_LENGTH + packages_counter] = (uint8_t)((sample_burst_buf[(frame_number*(burst_frame_lenght/2))+
                                sample_counter] >> 0) & 0xFF);
639                         seq_number++;       // Sequenznummer hochzählen
640                         sample_counter++;
641                     }
642
643                     tx_packet((HEADER_LENGTH + burst_frame_lenght),packet);
644
645
646                     rx(HEADER_LENGTH + rx_data_length);
647                 } break;
648
649                 /*************************************************************
650                 ** Kommando SLEEP empfangen
651                 *************************************************************/
652                 case COMMAND_DOWNLINK_SLEEP: {
653
654                     led_on(LED_ALL);
655                     delay_ms(1000);
656                     led_off(LED_ALL);
657
658                     init_for_sleep();
659
660                     ENTER_LPM4;
661
662                 } break;
663
664                 /*************************************************************
665                 **
666                 *************************************************************/
667                 case COMMAND_WAIT: {
668
669                     rx(HEADER_LENGTH + rx_data_length);
670                 } break;
671             }
672         }
673     }
674     return 0;
675 }
676
```

```
677 /*****************************************************************************************
678 ** Empfangsmodus
679 *****************************************************************************************/
680 void rx(uint8_t packages) {
681     cc1101_set_rx(brate);
682     IRQ_SET_RX;
683     CC1101_GDO2_CLEAR_IRQ;      // Clear the sync word detected interrupt
684     CC1101_GDO2_IRQ_ENABLE;     // Enable the sync word detected interrupt
685     cc1101_rx(packages);                // Transceiver in RX state
686     command_recived = 0;
687 }
688 /*****************************************************************************************
689 ** Senden eines Dauerträgers: Wird nur für Testzwecke benötigt
690 *****************************************************************************************/
691 void tx_carrier(void) {
692     //led_on(LED_TX);
693
694     cc1101_init();
695     cc1101_reset();         // Reset chip and go to idle state
696     cc1101_config_no_packet();
697     cc1101_tx_carrier();
698     P1DIR |= BIT6;
699     P1OUT |= 1;
700     while(1);
701 }
702 /*****************************************************************************************
703 ** Empfangen eines Pakets
704 *****************************************************************************************/
705 void tx_packet(uint8_t packetes, uint8_t *txbuf) {
706     /************************************
707      * Sendezeit optimierung 26.03.13 NS
708      ************************************/
709     cc1101_set_tx(brate);
710     IRQ_SET_TX;                 // Sendeflag setzen
711     CC1101_GDO2_CLEAR_IRQ;      // Clear the sync word detected interrupt
712     CC1101_GDO2_IRQ_ENABLE;     // Enable the sync word detected interrupt
713
714     if(IRQ_TIMERA_IS_BALANCING) {
715         TIMER_A_STOP;
716         TIMER_A_0_CM_IRQ_DISABLE;
717     }
718
719     led_on(LED_TX);
720     cc1101_fill_tx_fifo(txbuf, packetes);
721     cc1101_tx(packetes);
722
723     while(!irq_send_done_flag);
724
725     if(IRQ_TIMERA_IS_BALANCING) {
726         TIMER_A_0_CM_IRQ_ENABLE;
727         TIMER_A_START_UP_MODE;
728     }
729
730     led_off(LED_TX);
731
732     irq_send_done_flag = 0;
733
734     cc1101_idle();
735
736 }
```

F.1.23. main.h

```
1   /*****************************************************************************
2   **     Description:    main.h
3   **     Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **     Date:           06/03/2013
5   **     Author:         Nico Sassano
6   **                     modified taken from Phillip Durdaut
7   *****************************************************************************/
8
9   #ifndef MAIN_H_
10  #define MAIN_H_
11
12  #include <msp430x23x.h>
13  #include <signal.h>
14  #include <stdio.h>
15
16  #include "types.h"
17  #include "adc12.h"
18  #include "adg918.h"
19  #include "as3930.h"
20  #include "balancing.h"
21  #include "cc1101.h"
22  #include "delay.h"
23  #include "led.h"
24  #include "i2c.h"
25  #include "temp_sensor.h"
26  #include "timer.h"
27  #include "clk.h"
28  #include "init.h"
29
30  /*-------------------------------------------------------------------------
31    Fix error in msp430x23x.h
32  -------------------------------------------------------------------------*/
33
34  #define __MSP430_HAS_ADC12__
35  #include <msp430/adc12.h>
36
37  /*-------------------------------------------------------------------------
38    Defines
39  -------------------------------------------------------------------------*/
40
41  // Address definitions
42  #define BROADCAST                                   0x00    // Adresse des Broadcast
43  #define ADDRESS_BASE_STATION                        0xFF    // Adresse der Basisstation
44  #define ADDRESS_THIS_SENSOR                         0x01    // Adresse des Sensors
45
46  #define ENABLE_LEDS
47
48  // Frequency offset added to the base frequency (in units of 1.59 kHz - 1.65 kHz)
49  #define FREQUENCY_OFFSET                6
50
51  // Testport deklaration
52  #define TEST_PORT_PxDIR                             (P2DIR)
53  #define TEST_PORT_PxOUT                             (P2OUT)
54  #define TEST_PORT_PIN                               (BIT7)
55
56  #define TEST_PIN_ON         TEST_PORT_PxOUT |= TEST_PORT_PIN
57  #define TEST_PIN_OFF        TEST_PORT_PxOUT &=~TEST_PORT_PIN
58
59  // TPS61201 Control
60  #define TPS61201_PxDIR                              (P5DIR)
61  #define TPS61201_PxOUT                              (P5OUT)
62  #define TPS61201_PIN                                (BIT1)
63
64  #define TPS61201_ENABLE         TPS61201_PxOUT |= TPS61201_PIN
65  #define TPS61201_DISABLE        TPS61201_PxOUT &=~TPS61201_PIN
66
67
68
69  #define ON                                          0x01
70  #define OFF                                         0x00
71
72  /*** Downlink commands ************************************/
73  #define COMMAND_WAKEUP                              0x00
74  #define COMMAND_WAKEUP_DONE                         0x01
75  #define COMMAND_DOWNLINK_IS_AWAKE                   0x02
76  #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE             0x03
77  #define COMMAND_DOWNLINK_SEND_VOLTAGE               0x04
78  #define COMMAND_DOWNLINK_SLEEP                      0x05
79  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE         0x06
80  #define COMMAND_DOWNLINK_SEND_TEMPERATURE           0x07
81  #define COMMAND_DOWNLINK_BALANCING_ON               0x08
82  #define COMMAND_DOWNLINK_BALANCING_OFF              0x09
83  #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE   0x0A
```

```
84  #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE      0x0B
85  #define COMMAND_DOWNLINK_BURST_MODE                      0x0C
86  #define COMMAND_DOWNLINK_BURST_DATA_RQ                   0x0D
87  #define COMMAND_DOWNLINK_BURST_DATA_RX                   0x0E
88  #define COMMAND_DOWNLINK_BURST_CHECK                     0x0F
89  #define COMMAND_DOWNLINK_CONFIG_SET                      0x10
90  #define COMMAND_DOWNLINK_CONFIG                          0x11
91  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102       0x12
92  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102         0x13
93  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_TMP102_CALI  0x14
94  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_TMP102_CALI    0x15
95  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430       0x16
96  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430         0x17
97  #define COMMAND_DOWNLINK_SAMPLE_TEMPERATURE_MSP430_CALI  0x18
98  #define COMMAND_DOWNLINK_SEND_TEMPERATURE_MSP430_CALI    0x19
99  #define COMMAND_DOWNLINK_CALIBRATION_TMP102              0x1A
100 #define COMMAND_DOWNLINK_CALIBRATION_MSP430              0x1B
101 #define COMMAND_DOWNLINK_CALIBRATION_ADC                 0x1C
102 #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATURE_ALL    0x1D
103 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATURE_ALL  0x1E
104
105 #define COMMAND_BACK_FROM_BURST                          0xFD
106 #define COMMAND_WAIT                                     0xFE
107 /*** TEST DOWNLINKS ***************************************/
108 #define COMMAND_DOWNLINK_ERROR_TEST                      0x99
109
110 #define COMMAND_DOWNLINK_UNKWOWN                         0xFF
111
112 #define HEADER_LENGTH                                    0x04
113
114 /*************************************************************
115 ** Frequenzen für die Burstmessung
116 *************************************************************/
117 #define BURST_FREQ_7500HZ                                0x1A
118 #define BURST_FREQ_10000HZ                               0x19
119 #define BURST_FREQ_8000HZ                                0x18
120 #define BURST_FREQ_6000HZ                                0x17
121 #define BURST_FREQ_4000HZ                                0x15
122 #define BURST_FREQ_2000HZ                                0x16
123 #define BURST_FREQ_1000HZ                                0x01
124 #define BURST_FREQ_950HZ                                 0x02
125 #define BURST_FREQ_900HZ                                 0x03
126 #define BURST_FREQ_850HZ                                 0x04
127 #define BURST_FREQ_800HZ                                 0x05
128 #define BURST_FREQ_750HZ                                 0x06
129 #define BURST_FREQ_700HZ                                 0x07
130 #define BURST_FREQ_650HZ                                 0x08
131 #define BURST_FREQ_600HZ                                 0x09
132 #define BURST_FREQ_550HZ                                 0x0A
133 #define BURST_FREQ_500HZ                                 0x0B
134 #define BURST_FREQ_450HZ                                 0x0C
135 #define BURST_FREQ_400HZ                                 0x0D
136 #define BURST_FREQ_350HZ                                 0x0E
137 #define BURST_FREQ_300HZ                                 0x0F
138 #define BURST_FREQ_250HZ                                 0x10
139 #define BURST_FREQ_200HZ                                 0x11
140 #define BURST_FREQ_150HZ                                 0x12
141 #define BURST_FREQ_100HZ                                 0x13
142 #define BURST_FREQ_50HZ                                  0x14
143
144 #define BURST_VALUES_50                                  0x01
145 #define BURST_VALUES_100                                 0x02
146 #define BURST_VALUES_150                                 0x03
147 #define BURST_VALUES_200                                 0x04
148 #define BURST_VALUES_250                                 0x05
149 #define BURST_VALUES_300                                 0x06
150 #define BURST_VALUES_350                                 0x07
151 #define BURST_VALUES_400                                 0x08
152 #define BURST_VALUES_450                                 0x09
153 #define BURST_VALUES_500                                 0x0A
154 #define BURST_VALUES_550                                 0x0B
155 #define BURST_VALUES_600                                 0x0C
156 #define BURST_VALUES_650                                 0x0D
157 #define BURST_VALUES_700                                 0x0E
158 #define BURST_VALUES_750                                 0x0F
159 #define BURST_VALUES_800                                 0x10
160 #define BURST_VALUES_850                                 0x11
161 #define BURST_VALUES_900                                 0x12
162
163 /*************************************************************
164 ** Interrupt Modes
165 *************************************************************/
166 #define TX_MODE                                          0x00
167 #define RX_MODE                                          0x01
168 #define BURST_MODE                                       0x02
```

```
169
170 #define ALERT_HIGH                                  0x00
171 #define ALERT_LOW                                   0x01
172
173 #define TEMP_NORMAL                                 0x00
174 #define TEMP_HIGH                                   0x01
175
176 /*****************************************************
177 ** TIMER Macros
178 *****************************************************/
179 #define TIMERA_BALANCING                            0x01
180
181 #define TIMERB_BURST                                0x01
182 #define TIMERB_BURST_END                            0x02
183
184 // Clock frequencies
185 #define DCOCLK                          1000000  // DCO Clock frequency
186 #define MCLK                            1000000  // Main System Clock frequency
187 #define SMCLK                           1000000  // Sub System Clock frequency
188
189 /*****************************************************
190 ** IRQ Macros
191 *****************************************************/
192 #define IRQ_SET_RX                      (irq_mode = RX_MODE)
193 #define IRQ_IS_RX                       (irq_mode == RX_MODE)
194 #define IRQ_SET_TX                      (irq_mode = TX_MODE)
195 #define IRQ_IS_TX                       (irq_mode == TX_MODE)
196 #define IRQ_SET_BURST                   (irq_mode = BURST_MODE)
197 #define IRQ_IS_BURST                    (irq_mode == BURST_MODE)
198
199 /*****************************************************
200 ** IRQ_ALERT_SET_HIGH    bei zu hoher Temperatur
201 ** IRQ_ALERT_IS_LOW      bei normaler Temperatur
202 *****************************************************/
203 #define IRQ_ALERT_SET_HIGH              (irq_alert = ALERT_HIGH)
204 #define IRQ_ALERT_IS_HIGH               (irq_alert == ALERT_HIGH)
205 #define IRQ_ALERT_SET_LOW               (irq_alert = ALERT_LOW)
206 #define IRQ_ALERT_IS_LOW                (irq_alert == ALERT_LOW)
207
208 #define TEMP_IS_NORMAL                  (temp_state == TEMP_NORMAL)
209 #define TEMP_SET_NORMAL                 (temp_state = TEMP_NORMAL)
210 #define TEMP_IS_HIGH                    (temp_state == TEMP_HIGH)
211 #define TEMP_SET_HIGH                   (temp_state = TEMP_HIGH)
212
213 /*****************************************************
214 ** IRQ TIMER
215 *****************************************************/
216 #define IRQ_TIMERA_SET_BALANCING        (irq_timera = TIMERA_BALANCING)
217 #define IRQ_TIMERA_UNSET_BALANCING      (irq_timera = 0x00)
218 #define IRQ_TIMERA_IS_BALANCING         (irq_timera == TIMERA_BALANCING)
219
220 #define IRQ_TIMERB_SET_BURST            (irq_timerb = TIMERB_BURST)
221 #define IRQ_TIMERB_UNSET_BURST          (irq_timerb = 0x00)
222 #define IRQ_TIMERB_IS_BURST             (irq_timerb == TIMERB_BURST)
223
224 /*****************************************************
225 ** Burst Flag Macros
226 *****************************************************/
227 #define BURST_ERROR_FLAG_SET            (burst_error_flag = 0x01)
228 #define BURST_ERROR_FLAG_UNSET          (burst_error_flag = 0x00)
229 #define BURST_ERROR_FLAG_IS_SET         (burst_error_flag == 0x01)
230 #define BURST_ERROR_FLAG_IS_UNSET       (burst_error_flag == 0x00)
231 /*-------------------------------------------------------------------
232   Types
233 -------------------------------------------------------------------*/
234
235 typedef enum { S_INIT,
236                S_SLEEP,
237                S_RX,
238                S_WAKEUP,
239                S_WAKEUP_RX,
240                S_WAKEUP_DONE,
241                S_TX_AWAKE,
242                S_SAMPLE,
243                S_TX_SAMPLE,
244                S_BALANC_FIRST_START,
245                S_BALANC_ON,
246                S_BALANC_OFF
247              } state_t ;
248
249 /*-------------------------------------------------------------------
250   Macros
251 -------------------------------------------------------------------*/
252
253 #define ENTER_LPM4                      __bis_SR_register(LPM4_bits + GIE);
```

```
254 #define EXIT_LPM4                       __bic_SR_register_on_exit(LPM4_bits);
255
256 void rx(uint8_t);
257
258 #endif /* MAIN_H_ */
```

F.1.24. tempsensor.c

```
1  /*****************************************************************************
2  **    Description:    temp_sensor.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           06/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Niels Jegenhorst
7  *****************************************************************************/
8
9  #include "main.h"
10
11 extern uint16_t config_value;
12
13 void temp_sensor_init(void) {
14     temp_sensor_get_contr_reg();
15 }
16
17 void temp_sensor_get_contr_reg() {
18     uint8_t data[2];
19
20     /*************************************************************
21     * Pointer wird auf das Config Register ausgerichtet
22     *************************************************************/
23     data[0] = TMP102_REG_CONF; // Value for Pointer-Register
24
25     i2c_write(ADDR_TMP102, 1, data);      // Set Pointer-Register
26
27     i2c_read(ADDR_TMP102, 2, data);       // Read Control-Register
28
29     config_value = (uint16_t) data[0] << 4;
30     config_value |= data[1] >> 4;
31
32 }
33
34 void temp_sensor_config_reg() {
35     uint8_t data[3];
36
37     /*************************************************************
38     * Hier wird der Temperatursensor konfiguriert
39     *************************************************************/
40     TMP102_EM_OFF;
41     TMP102_CON_RATE_4;
42     TMP102_SD_OFF;
43     TMP102_COMPERATOR_MODE;
44     TMP102_POL_INV;
45     TMP102_FAULTS_6;
46
47     data[0] = TMP102_REG_CONF;                    // Value for Pointer-Register
48     data[1] = (uint8_t) (config_value >> 8);      // Byte 1 at first
49     data[2] = (uint8_t) (config_value & 0x00FF);  // Byte 2 at last
50
51     i2c_write(ADDR_TMP102, 3, data);              // Write Control-Register
52
53 }
54
55 /*****************************************************************
56 ** Setzen der Alarmtemperatur
57 ** 1 Digit -> 0,0625°C
58 ** 0x1900  -> 25°C
59 *****************************************************************/
60 void temp_sensor_set_alert(uint16_t temp_low, uint16_t temp_high) {
61     uint8_t data[3];
62
63     /*************************************************************
64     * Low Daten setzen
65     *************************************************************/
66     data[0] = TMP102_REG_LOW;                     // Value for Pointer-Register
67     data[1] = (uint8_t) (temp_low >> 8);          // Byte 1 at first
68     data[2] = (uint8_t) (temp_low & 0x00FF);      // Byte 2 at last
69
70     i2c_write(ADDR_TMP102, 3, data);              // Write Control-Register
71
72     /*************************************************************
73     * High Daten setzen
74     *************************************************************/
75     data[0] = TMP102_REG_HIGH;                    // Value for Pointer-Register
76     data[1] = (uint8_t) (temp_high >> 8);         // Byte 1 at first
77     data[2] = (uint8_t) (temp_high & 0x00FF);     // Byte 2 at last
78
79     i2c_write(ADDR_TMP102, 3, data);              // Write Control-Register
80 }
81
82 uint16_t temp_sensor_get_temp(void) {
```

```
83
84      uint8_t data[2];
85      uint16_t temp_value = 0x0000;
86
87      data[0] = TMP102_REG_TEMP;                      // Value for Pointer-Register
88      i2c_write(ADDR_TMP102, 1, data);                // Set Pointer-Register
89      temp_value = 0x0000;
90      i2c_read(ADDR_TMP102, 2, data);
91
92      temp_value = (uint16_t) data[0] << 4;
93      temp_value |= data[1] >> 4;
94
95      return temp_value;
96
97  }
```

F.1.25. tempsensor.h

```
1   /*******************************************************************************
2   **    Description:    temp_sensor.h
3   **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **    Date:           06/03/2013
5   **    Author:         Nico Sassano
6   **                    modified taken from Niels Jegenhorst
7   *******************************************************************************/
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          P2REN
18  #define TMP102_ALERT_PxOUT          P2OUT
19  #define TMP102_ALERT_PxDIR          P2DIR
20  #define TMP102_ALERT_PxIES          P2IES
21  #define TMP102_ALERT_PxIE           P2IE
22  #define TMP102_ALERT_PxIFG          P2IFG
23  #define TMP102_ALERT_PIN            BIT4
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_ */
```

F.1.26. timer.c

```
1   /*******************************************************************************
2   **    Description:    timer.c
3   **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **    Date:           13/03/2013
5   **    Author:         Nico Sassano
6   *******************************************************************************/
7
8   #include "main.h"
9
10  extern volatile uint8_t irq_timera;           // Status TimerA
11  extern volatile uint8_t irq_timerb;           // Status TimerB
12  extern volatile uint8_t burst_error_flag;
13
14  void timer_a_init(void) {
15      /*************************************************************
16      * Timer A initialisierung
17      *************************************************************/
18      // timer clear
19      TACTL = TACLR;
20      // no grouping, counter length = 16bit, clock source = SMCLK, div = 8
21      TACTL |= (TASSEL1 | ID1 | ID0);
22      // Capture/Compare Reg 0 set
23      TACCR0 = 62500;
24      // Compare-mode, IRQ enable
25      TACCTL0 |= CCIE;
26  }
27
28  /*************************************************************
29  ** Timer A Initialisierung für die Balancierung
30  ** 1MHz; DIV=8; TACCR0 = 62500 -> 500ms
31  *************************************************************/
32  void timer_a_init_balanc(void) {
33      IRQ_TIMERA_SET_BALANCING;
34      TIMER_A_STOP;
35      TIMER_A_RESET;
36      TIMER_A_SOURCE_SMCLK;
37      TIMER_A_SOURCE_DIV_8;
38
39      TACCR0 = 62500;
40      TIMER_A_0_CM_IRQ_ENABLE;
41  }
42
43  /*************************************************************
44  ** Timer B Initialisierung für die Burstmessung
45  **
46  *************************************************************/
47  void timer_b_init_burst(uint8_t freq) {
48      IRQ_TIMERB_SET_BURST;
49
50      TIMER_B_STOP;
51      TIMER_B_RESET;
52      TIMER_B_SOURCE_SMCLK;
53
54      switch(freq) {
55
56          /************************
57          ** 16MHz; TimerDiv 1
58          ** Periode  -> 100us
59          ** 1472 (ISR)-> 92us
60          ************************/
61          case BURST_FREQ_10000HZ: {
62              TIMER_B_SOURCE_DIV_1;
63
64              if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
65                  TBCCR0 = 2000 - 1472;   // 125us - 92us (ISR)
66              } else {
67                  TBCCR0 = 1600 - 184;    // 100 us - 11,5us (ISR)
68                  BURST_ERROR_FLAG_UNSET;
69              }
70
71              // Interrupts für TimerB freischalten
72              TIMER_B_0_CM_IRQ_ENABLE;
73              TIMER_B_1_CM_IRQ_DISABLE;
74          }break;
75
76          /************************
77          ** 16MHz; TimerDiv 1
78          ** Periode      -> 125us
79          ** 1472 (ISR)   -> 92us
80          ************************/
81          case BURST_FREQ_8000HZ: {
82              TIMER_B_SOURCE_DIV_1;
83
```

```
84              if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
85                  TBCCR0 = 2400 - 1472;    // 150us - 92us (ISR)
86              } else {
87                  TBCCR0 = 2000 - 184;     // 125 us - 11,5us (ISR)
88                  BURST_ERROR_FLAG_UNSET;
89              }
90
91              // Interrupts für TimerB freischalten
92              TIMER_B_0_CM_IRQ_ENABLE;
93              TIMER_B_1_CM_IRQ_DISABLE;
94          }break;
95
96          /************************
97          ** 16MHz; TimerDiv 1
98          ** Periode       -> 133us
99          ** 1472 (ISR)    -> 92us
100         ************************/
101         case BURST_FREQ_7500HZ: {
102             TIMER_B_SOURCE_DIV_1;
103
104             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
105                 TBCCR0 = 2400 - 1472;    // 150us - 92us (ISR)
106             } else {
107                 TBCCR0 = 2000 - 184;     // 125 us - 11,5us (ISR)
108                 BURST_ERROR_FLAG_UNSET;
109             }
110
111             // Interrupts für TimerB freischalten
112             TIMER_B_0_CM_IRQ_ENABLE;
113             TIMER_B_1_CM_IRQ_DISABLE;
114         }break;
115
116         /************************
117         ** 16MHz; TimerDiv 1
118         ** Periode  -> 166us
119         ** 1472 (ISR)-> 92us
120         ************************/
121         case BURST_FREQ_6000HZ: {
122             TIMER_B_SOURCE_DIV_1;
123
124             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
125                 TBCCR0 = 3200 - 1472;     // 200 us - 92us (ISR)
126             } else {
127                 TBCCR0 = 2656 - 184;      // 166 us - 11,5us (ISR)
128                 BURST_ERROR_FLAG_UNSET;
129             }
130
131             // Interrupts für TimerB freischalten
132             TIMER_B_0_CM_IRQ_ENABLE;
133             TIMER_B_1_CM_IRQ_DISABLE;
134         }break;
135
136         /************************
137         ** 16MHz; TimerDiv 1
138         ** Periode  -> 250us
139         ** 1472 (ISR)-> 92us
140         ************************/
141         case BURST_FREQ_4000HZ: {
142             TIMER_B_SOURCE_DIV_1;
143
144             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
145                 TBCCR0 = 4400 - 1472;    // 275us - 92us (ISR)
146             } else {
147                 TBCCR0 = 4000 - 184;     // 250 us - 11,5us (ISR)
148                 BURST_ERROR_FLAG_UNSET;
149             }
150
151             // Interrupts für TimerB freischalten
152             TIMER_B_0_CM_IRQ_ENABLE;
153             TIMER_B_1_CM_IRQ_DISABLE;
154          }break;
155
156          /************************
157          ** 16MHz; TimerDiv 1
158          ** Periode  -> 500us
159          ** 1472 (ISR)-> 92us
160          ************************/
161         case BURST_FREQ_2000HZ: {
162             TIMER_B_SOURCE_DIV_1;
163
164             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
165                 TBCCR0 = 9600 - 1472;    // 600us - 92us (ISR)
166             } else {
167                 TBCCR0 = 8000 - 184;     // 500 us - 11,5us (ISR)
168                 BURST_ERROR_FLAG_UNSET;
```

```
169             }
170
171             // Interrupts für TimerB freischalten
172             TIMER_B_0_CM_IRQ_ENABLE;
173             TIMER_B_1_CM_IRQ_DISABLE;
174         }break;
175
176         /************************
177         ** 16MHz; TimerDiv 1
178         ** Periode  -> 1ms
179         ** 12800    -> 800us
180         ** 19200    -> 1.2ms
181         ** 1472 (ISR)-> 92us
182         ************************/
183         case BURST_FREQ_1000HZ: {
184             TIMER_B_SOURCE_DIV_1;
185
186             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
187                 TBCCR1 = 12800 - 1472;   // 800us  - 92us (ISR)
188                 TBCCR0 = 19200 - 1472;   // 1200us - 92us (ISR)
189             } else {
190                 TBCCR1 = 8600 ;          // 600 us
191                 TBCCR0 = 16000;          // 1000 us
192                 BURST_ERROR_FLAG_UNSET;
193             }
194
195             // Interrupts für TimerB freischalten
196             TIMER_B_0_CM_IRQ_ENABLE;
197             TIMER_B_1_CM_IRQ_ENABLE;
198
199         }break;
200
201         /************************
202         ** 16MHz; TimerDiv 1
203         ** Periode  -> 1.052ms
204         ** 13632    -> 0.852 ms
205         ** 20032    -> 1.252 ms
206         ** 1472 (ISR)-> 92us
207         ************************/
208         case BURST_FREQ_950HZ: {
209             TIMER_B_SOURCE_DIV_1;
210             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
211                 TBCCR1 = 13632 - 1472;  // 0.852 ms - 33 us
212                 TBCCR0 = 20032 - 1472;   // 1.252 ms - 33 us
213             } else {
214                 TBCCR1 = 10432;          // 652 us
215                 TBCCR0 = 16832;          // 1052 us
216                 BURST_ERROR_FLAG_UNSET;
217             }
218
219             // Interrupts für TimerB freischalten
220             TIMER_B_0_CM_IRQ_ENABLE;
221             TIMER_B_1_CM_IRQ_ENABLE;
222         }break;
223
224         /************************
225         ** 16MHz; TimerDiv 1
226         ** Periode  -> 1.10 ms
227         ** 14400    -> 0.90 ms
228         ** 20800    -> 1.30 ms
229         ** 1472 (ISR)-> 92us
230         ************************/
231         case BURST_FREQ_900HZ: {
232             TIMER_B_SOURCE_DIV_1;
233             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
234                 TBCCR1 = 14400 - 1472;
235                 TBCCR0 = 20800 - 1472;
236             } else {
237                 TBCCR1 = 11200;          // 700 us
238                 TBCCR0 = 17600;          // 1.1 ms
239                 BURST_ERROR_FLAG_UNSET;
240             }
241
242             // Interrupts für TimerB freischalten
243             TIMER_B_0_CM_IRQ_ENABLE;
244             TIMER_B_1_CM_IRQ_ENABLE;
245         }break;
246
247         /************************
248         ** 16MHz; TimerDiv 1
249         ** Periode  -> 1.176 ms
250         ** 15616    -> 0.976 ms
251         ** 22016    -> 1.376 ms
252         ** 1472 (ISR)-> 92us
253         ************************/
```

```
254             case BURST_FREQ_850HZ: {
255                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
256                     TIMER_B_SOURCE_DIV_1;
257                     TBCCR1 = 15616 - 1472;
258                     TBCCR0 = 22016 - 1472;
259                 } else {
260                     TBCCR1 = 12416;          // 0.776 ms
261                     TBCCR0 = 18816;          // 1.176 ms
262                     BURST_ERROR_FLAG_UNSET;
263                 }
264
265                 // Interrupts für TimerB freischalten
266                 TIMER_B_0_CM_IRQ_ENABLE;
267                 TIMER_B_1_CM_IRQ_ENABLE;
268             }break;
269
270             /************************
271             ** 16MHz; TimerDiv 1
272             ** Periode  -> 1.25 ms
273             ** 16800    -> 1.05 ms
274             ** 23200    -> 1.45 ms
275             ** 1472 (ISR)-> 92us
276             ************************/
277             case BURST_FREQ_800HZ: {
278                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
279                     TIMER_B_SOURCE_DIV_1;
280                     TBCCR1 = 16800 - 1472;
281                     TBCCR0 = 23200 - 1472;
282                 } else {
283                     TBCCR1 = 13600;          // 850us
284                     TBCCR0 = 20000;          // 1.25 ms
285                     BURST_ERROR_FLAG_UNSET;
286                 }
287
288                 // Interrupts für TimerB freischalten
289                 TIMER_B_0_CM_IRQ_ENABLE;
290                 TIMER_B_1_CM_IRQ_ENABLE;
291             }break;
292
293             /************************
294             ** 16MHz; TimerDiv 1
295             ** Periode  -> 1.33 ms
296             ** 18080    -> 1.13 ms
297             ** 24480    -> 1.53 ms
298             ** 1472 (ISR)-> 92us
299             ************************/
300             case BURST_FREQ_750HZ: {
301                 TIMER_B_SOURCE_DIV_1;
302                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
303                     TBCCR1 = 18080 - 1472;
304                     TBCCR0 = 24480 - 1472;
305                 } else {
306                     TBCCR1 = 14880;          // 0.930 ms
307                     TBCCR0 = 21280;          // 1.330 ms
308                     BURST_ERROR_FLAG_UNSET;
309                 }
310
311                 // Interrupts für TimerB freischalten
312                 TIMER_B_0_CM_IRQ_ENABLE;
313                 TIMER_B_1_CM_IRQ_ENABLE;
314             }break;
315
316             /************************
317             ** 16MHz; TimerDiv 1
318             ** Periode  -> 1.49 ms
319             ** 20640    -> 1.29 ms
320             ** 27040    -> 1.69 ms
321             ** 1472 (ISR)-> 92us
322             ************************/
323             case BURST_FREQ_700HZ: {
324
325                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
326                     TIMER_B_SOURCE_DIV_1;
327                     TBCCR1 = 20640 - 1472;
328                     TBCCR0 = 27040 - 1472;
329                 } else {
330                     TBCCR1 = 17440;          // 1.09 ms
331                     TBCCR0 = 23840;          // 1.49 ms
332                     BURST_ERROR_FLAG_UNSET;
333                 }
334
335                 // Interrupts für TimerB freischalten
336                 TIMER_B_0_CM_IRQ_ENABLE;
337                 TIMER_B_1_CM_IRQ_ENABLE;
338             }break;
```

```
339
340             /************************
341             ** 16MHz; TimerDiv 1
342             ** Periode  -> 1.54 ms
343             ** 21440    -> 1.34 ms
344             ** 27840    -> 1.74 ms
345             ** 1472 (ISR)-> 92us
346             ************************/
347             case BURST_FREQ_650HZ: {
348                 TIMER_B_SOURCE_DIV_1;
349
350                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
351                     TBCCR1 = 21440 - 1472;
352                     TBCCR0 = 27840 - 1472;
353                 } else {
354                     TBCCR1 = 18240;          // 1.14 ms
355                     TBCCR0 = 24640;          // 1.54 ms
356                     BURST_ERROR_FLAG_UNSET;
357                 }
358
359                 // Interrupts für TimerB freischalten
360                 TIMER_B_0_CM_IRQ_ENABLE;
361                 TIMER_B_1_CM_IRQ_ENABLE;
362             }break;
363
364             /************************
365             ** 16MHz; TimerDiv 1
366             ** Periode  -> 1.60 ms
367             ** 22400    -> 1.40 ms
368             ** 28800    -> 1.80 ms
369             ** 1472 (ISR)-> 92us
370             ************************/
371             case BURST_FREQ_600HZ: {
372                 TIMER_B_SOURCE_DIV_1;
373
374                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
375                     TBCCR1 = 22400 - 1472;
376                     TBCCR0 = 28800 - 1472;
377                 } else {
378                     TBCCR1 = 19200;          // 1.2 ms
379                     TBCCR0 = 25600;          // 1.6 ms
380                     BURST_ERROR_FLAG_UNSET;
381                 }
382
383                 // Interrupts für TimerB freischalten
384                 TIMER_B_0_CM_IRQ_ENABLE;
385                 TIMER_B_1_CM_IRQ_ENABLE;
386             }break;
387
388             /************************
389             ** 16MHz; TimerDiv 1
390             ** Periode  -> 1.80 ms
391             ** 25600    -> 1.60 ms
392             ** 32000    -> 2.00 ms
393             ** 1472 (ISR)-> 92us
394             ************************/
395             case BURST_FREQ_550HZ: {
396                 TIMER_B_SOURCE_DIV_1;
397
398                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
399                     TBCCR1 = 25600 - 1472;
400                     TBCCR0 = 32000 - 1472;
401                 } else {
402                     TBCCR1 = 22400;          // 1.4 ms
403                     TBCCR0 = 28800;          // 1.8 ms
404                     BURST_ERROR_FLAG_UNSET;
405                 }
406
407                 // Interrupts für TimerB freischalten
408                 TIMER_B_0_CM_IRQ_ENABLE;
409                 TIMER_B_1_CM_IRQ_ENABLE;
410             }break;
411
412             /************************
413             ** 16MHz; TimerDiv 1
414             ** Periode  -> 2.00 ms
415             ** 28800    -> 1.80 ms
416             ** 35200    -> 2.20 ms
417             ** 1472 (ISR)-> 92us
418             ************************/
419             case BURST_FREQ_500HZ: {
420                 TIMER_B_SOURCE_DIV_1;
421
422                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
423                     TBCCR1 = 28800 - 1472;
```

```
424                 TBCCR0 = 35200 - 1472;
425             } else {
426                 TBCCR1 = 25600;          // 1.6 ms
427                 TBCCR0 = 32000;          // 2.0 ms
428                 BURST_ERROR_FLAG_UNSET;
429             }
430
431             // Interrupts für TimerB freischalten
432             TIMER_B_0_CM_IRQ_ENABLE;
433             TIMER_B_1_CM_IRQ_ENABLE;
434         }break;
435
436         /************************
437         ** 16MHz; TimerDiv 1
438         ** Periode  -> 2.22 ms
439         ** 32320    -> 2.02 ms
440         ** 38720    -> 2.42 ms
441         ** 1472 (ISR)-> 92us
442         ************************/
443         case BURST_FREQ_450HZ: {
444             TIMER_B_SOURCE_DIV_1;
445
446             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
447                 TBCCR1 = 32320 - 1472;
448                 TBCCR0 = 38720 - 1472;
449             } else {
450                 TBCCR1 = 29120;          // 1.82 ms
451                 TBCCR0 = 35520;          // 2.22 ms
452                 BURST_ERROR_FLAG_UNSET;
453             }
454
455             // Interrupts für TimerB freischalten
456             TIMER_B_0_CM_IRQ_ENABLE;
457             TIMER_B_1_CM_IRQ_ENABLE;
458         }break;
459
460         /************************
461         ** 16MHz; TimerDiv 1
462         ** Periode  -> 2.50 ms
463         ** 36800    -> 2.30 ms
464         ** 43200    -> 2.00 ms
465         ** 1472 (ISR)-> 92us
466         ************************/
467         case BURST_FREQ_400HZ: {
468             TIMER_B_SOURCE_DIV_1;
469
470             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
471                 TBCCR1 = 36800 - 1472;
472                 TBCCR0 = 43200 - 1472;
473             } else {
474                 TBCCR1 = 33600;          // 2.1 ms
475                 TBCCR0 = 40000;          // 2.5 ms
476                 BURST_ERROR_FLAG_UNSET;
477             }
478
479             // Interrupts für TimerB freischalten
480             TIMER_B_0_CM_IRQ_ENABLE;
481             TIMER_B_1_CM_IRQ_ENABLE;
482         }break;
483
484         /************************
485         ** 16MHz; TimerDiv 1
486         ** Periode  -> 2.85 ms
487         ** 42400    -> 2.65 ms
488         ** 48800    -> 3.05 ms
489         ** 1472 (ISR)-> 92us
490         ************************/
491         case BURST_FREQ_350HZ: {
492             TIMER_B_SOURCE_DIV_1;
493
494             if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
495                 TBCCR1 = 42400 - 1472;
496                 TBCCR0 = 48800 - 1472;
497             } else {
498                 TBCCR1 = 39200;          // 2.45 ms
499                 TBCCR0 = 45600;          // 2.85 ms
500                 BURST_ERROR_FLAG_UNSET;
501             }
502
503             // Interrupts für TimerB freischalten
504             TIMER_B_0_CM_IRQ_ENABLE;
505             TIMER_B_1_CM_IRQ_ENABLE;
506         }break;
507
508         /************************
```

```
509             ** 16MHz; TimerDiv 1
510             ** Periode  -> 3.33 ms
511             ** 50080    -> 3.13 ms
512             ** 56480    -> 3.53 ms
513             ** 1472 (ISR)-> 92us
514             ************************/
515             case BURST_FREQ_300HZ: {
516                 TIMER_B_SOURCE_DIV_1;
517
518                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
519                     TBCCR1 = 50080 - 1472;
520                     TBCCR0 = 56480 - 1472;
521                 } else {
522                     TBCCR1 = 46880;          // 2.93 ms
523                     TBCCR0 = 53280;          // 3.33 ms
524                     BURST_ERROR_FLAG_UNSET;
525                 }
526
527                 // Interrupts für TimerB freischalten
528                 TIMER_B_0_CM_IRQ_ENABLE;
529                 TIMER_B_1_CM_IRQ_ENABLE;
530             }break;
531
532             /************************
533             ** 16MHz; TimerDiv 2
534             ** Periode  -> 4.00 ms
535             ** 30400    -> 3.80 ms
536             ** 33600    -> 4.20 ms
537             ** 264 (ISR)-> 92us
538             ************************/
539             case BURST_FREQ_250HZ: {
540                 TIMER_B_SOURCE_DIV_2;
541
542                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
543                     TBCCR1 = 30400 - 264;
544                     TBCCR0 = 33600 - 264;
545                 } else {
546                     TBCCR1 = 28800;          // 3.6 ms
547                     TBCCR0 = 32800;          // 4.1 ms
548                     BURST_ERROR_FLAG_UNSET;
549                 }
550
551                 // Interrupts für TimerB freischalten
552                 TIMER_B_0_CM_IRQ_ENABLE;
553                 TIMER_B_1_CM_IRQ_ENABLE;
554             }break;
555
556             /************************
557             ** 16MHz; TimerDiv 2
558             ** Periode  -> 5.00 ms
559             ** 38400    -> 4.80 ms
560             ** 41600    -> 5.20 ms
561             ** 264 (ISR)-> 92us
562             ************************/
563             case BURST_FREQ_200HZ: {
564                 TIMER_B_SOURCE_DIV_2;
565
566                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
567                     TBCCR1 = 38400 - 264;
568                     TBCCR0 = 41600 - 264;
569                 } else {
570                     TBCCR1 = 36800;          // 4.6 ms
571                     TBCCR0 = 40000;          // 5.0 ms
572                     BURST_ERROR_FLAG_UNSET;
573                 }
574
575                 // Interrupts für TimerB freischalten
576                 TIMER_B_0_CM_IRQ_ENABLE;
577                 TIMER_B_1_CM_IRQ_ENABLE;
578             }break;
579
580             /************************
581             ** 16MHz; TimerDiv 2
582             ** Periode  -> 6.66 ms
583             ** 51680    -> 6.46 ms
584             ** 54880    -> 6.86 ms
585             ** 264 (ISR)-> 92us
586             ************************/
587             case BURST_FREQ_150HZ: {
588                 TIMER_B_SOURCE_DIV_2;
589
590                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
591                     TBCCR1 = 51680 - 264;
592                     TBCCR0 = 54880 - 264;
593                 } else {
```

```
594                     TBCCR1 = 50080;          // 6.26 ms
595                     TBCCR0 = 53280;          // 6.66 ms
596                     BURST_ERROR_FLAG_UNSET;
597                 }
598
599                 // Interrupts für TimerB freischalten
600                 TIMER_B_0_CM_IRQ_ENABLE;
601                 TIMER_B_1_CM_IRQ_ENABLE;
602             }break;
603
604             /*************************
605             ** 16MHz; TimerDiv 4
606             ** Periode  -> 10.0 ms
607             ** 39200    -> 9.80 ms
608             ** 40800    -> 10.2 ms
609             ** 132 (ISR)-> 92us
610             *************************/
611             case BURST_FREQ_100HZ: {
612                 TIMER_B_SOURCE_DIV_4;
613
614                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
615                     TBCCR1 = 39200 - 132;
616                     TBCCR0 = 40800 - 132;
617                 } else {
618                     TBCCR1 = 38400;          //  9.6 ms
619                     TBCCR0 = 40000;          // 10.0 ms
620                     BURST_ERROR_FLAG_UNSET;
621                 }
622
623                 // Interrupts für TimerB freischalten
624                 TIMER_B_0_CM_IRQ_ENABLE;
625                 TIMER_B_1_CM_IRQ_ENABLE;
626             }break;
627
628             /*************************
629             ** 16MHz; TimerDiv 8
630             ** Periode  -> 20.0 ms
631             ** 39600    -> 19.8 ms
632             ** 40400    -> 20.2 ms
633             ** 66 (ISR) -> 92us
634             *************************/
635             case BURST_FREQ_50HZ: {
636                 TIMER_B_SOURCE_DIV_8;
637
638                 if(BURST_ERROR_FLAG_IS_UNSET) { // Wurde ein Fehler dedektiert?
639                     TBCCR1 = 39600 - 66;
640                     TBCCR0 = 40400 - 66;
641                 } else {
642                     TBCCR1 = 39200;          // 19.6 ms
643                     TBCCR0 = 40000;          // 20.0 ms
644                     BURST_ERROR_FLAG_UNSET;
645                 }
646
647                 // Interrupts für TimerB freischalten
648                 TIMER_B_0_CM_IRQ_ENABLE;
649                 TIMER_B_1_CM_IRQ_ENABLE;
650             }break;
651         }
652     }
653
654     void timer_b_init(void) {
655         /*************************************************************
656         * Timer B initialisierung
657         *************************************************************/
658
659         // timer clear
660         TBCTL = TACLR;
661         // no grouping, counter length = 16bit, clock source = SMCLK, div = 8
662         TBCTL |= (TASSEL1 | ID1 | ID0);
663         // Capture/Compare Reg 0 set
664         TBCCR0 = 62500;
665         // Compare-mode, IRQ enable
666         TBCCTL0 |= CCIE;
667
668     }
```

F.1.27. timer.h

```
1   /*****************************************************************************
2   **   Description:    timer.h
3   **   Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
4   **   Date:           13/03/2013
5   **   Author:         Nico Sassano
6   *****************************************************************************/
7
8   #ifndef TIMER_H_
9   #define TIMER_H_
10
11  #include "main.h"
12
13  /************************************************************
14  **  Defines Timer A
15  ************************************************************/
16  #define TIMER_A_START_UP_MODE         (TACTL |= MC0)
17  #define TIMER_A_START_CM_MODE         (TACTL |= MC1)
18  #define TIMER_A_START_UD_MODE         (TACTL |=(MC1 | MC0))
19
20  #define TIMER_A_STOP                  (TACTL &= ~(MC0 | MC1))
21
22  #define TIMER_A_RESET                 (TACTL = TACLR)
23  #define TIMER_A_FLAG_RESET0           (TACCTL0 &= ~(CCIFG|COV))
24
25  #define TIMER_A_SOURCE_TACLK          (TACTL = TASSEL_0)
26  #define TIMER_A_SOURCE_ACLK           (TACTL = TASSEL_1)
27  #define TIMER_A_SOURCE_SMCLK          (TACTL = TASSEL_2)
28
29  #define TIMER_A_SOURCE_DIV_8          (TACTL |= (ID1 | ID0))
30  #define TIMER_A_SOURCE_DIV_4          (TACTL |= ID1); \
31                                        (TACTL &= ~ID0);
32  #define TIMER_A_SOURCE_DIV_2          (TACTL &= ~ID1); \
33                                        (TACTL |= ID0);
34  #define TIMER_A_SOURCE_DIV_1          (TACTL &= ~ID1); \
35                                        (TACTL &= ~ID0);
36
37  #define TIMER_A_0_OUTMOD_RESET_SET    (TACCTL0 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
38  #define TIMER_A_1_OUTMOD_RESET_SET    (TACCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
39  #define TIMER_A_2_OUTMOD_RESET_SET    (TACCTL2 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
40
41  #define TIMER_A_0_OUTMOD_TOGGLE_SET   (TACCTL0 |= (OUTMOD2 | OUTMOD1)); \
42                                        (TACCTL0 &= ~OUTMOD0);
43  #define TIMER_A_1_OUTMOD_TOGGLE_SET   (TACCTL1 |= (OUTMOD2 | OUTMOD1)); \
44                                        (TACCTL1 &= ~OUTMOD0);
45  #define TIMER_A_2_OUTMOD_TOGGLE_SET   (TACCTL2 |= (OUTMOD2 | OUTMOD1)); \
46                                        (TACCTL2 &= ~OUTMOD0);
47
48  #define TIMER_A_0_OUTMOD_RESET        (TACCTL0 |= (OUTMOD2 | OUTMOD0)); \
49                                        (TACCTL0 &= ~OUTMOD1);
50  #define TIMER_A_1_OUTMOD_RESET        (TACCTL1 |= (OUTMOD2 | OUTMOD0)); \
51                                        (TACCTL1 &= ~OUTMOD1);
52  #define TIMER_A_2_OUTMOD_RESET        (TACCTL2 |= (OUTMOD2 | OUTMOD0)); \
53                                        (TACCTL2 &= ~OUTMOD1);
54
55  #define TIMER_A_0_CM_IRQ_ENABLE       (TACCTL0 |= CCIE);
56  #define TIMER_A_1_CM_IRQ_ENABLE       (TACCTL1 |= CCIE);
57  #define TIMER_A_2_CM_IRQ_ENABLE       (TACCTL2 |= CCIE);
58
59  #define TIMER_A_0_CM_IRQ_DISABLE      (TACCTL0 &= ~CCIE);
60  #define TIMER_A_1_CM_IRQ_DISABLE      (TACCTL1 &= ~CCIE);
61  #define TIMER_A_2_CM_IRQ_DISABLE      (TACCTL2 &= ~CCIE);
62
63
64  /************************************************************
65  **  Defines Timer B
66  ************************************************************/
67  #define TIMER_B_START_UP_MODE         (TBCTL |= MC0)
68  #define TIMER_B_START_CM_MODE         (TBCTL |= MC1)
69  #define TIMER_B_START_UD_MODE         (TBCTL |=(MC1 | MC0))
70
71  #define TIMER_B_STOP                  (TBCTL &= ~(MC0 | MC1))
72
73  #define TIMER_B_RESET                 (TBCTL = TBCLR)
74  #define TIMER_B_FLAG_RESET            (TBCCTL1 &= ~(CCIFG|COV))
75
76  #define TIMER_B_SOURCE_TACLK          (TBCTL = TBSSEL_0)
77  #define TIMER_B_SOURCE_ACLK           (TBCTL = TBSSEL_1)
78  #define TIMER_B_SOURCE_SMCLK          (TBCTL = TBSSEL_2)
79
80  #define TIMER_B_SOURCE_DIV_8          (TBCTL |= (ID1 | ID0))
81  #define TIMER_B_SOURCE_DIV_4          (TBCTL |= ID1); \
82                                        (TBCTL &= ~ID0);
83  #define TIMER_B_SOURCE_DIV_2          (TBCTL &= ~ID1); \
```

```
84                                      (TBCTL |= ID0);
85  #define TIMER_B_SOURCE_DIV_1        (TBCTL &= ~ID1); \
86                                      (TBCTL &= ~ID0);
87
88  #define TIMER_B_CM_RISING_EDGE      (TBCCTL0 |= CM1)
89
90  #define TIMER_B_0_OUTMOD_RESET_SET  (TBCCTL0 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
91  #define TIMER_B_1_OUTMOD_RESET_SET  (TBCCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
92  #define TIMER_B_2_OUTMOD_RESET_SET  (TBCCTL2 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
93
94  #define TIMER_B_0_OUTMOD_TOGGLE_SET (TBCCTL0 |= (OUTMOD2 | OUTMOD1)); \
95                                      (TBCCTL0 &= ~OUTMOD0);
96  #define TIMER_B_1_OUTMOD_TOGGLE_SET (TBCCTL1 |= (OUTMOD2 | OUTMOD1)); \
97                                      (TBCCTL1 &= ~OUTMOD0);
98  #define TIMER_B_2_OUTMOD_TOGGLE_SET (TBCCTL2 |= (OUTMOD2 | OUTMOD1)); \
99                                      (TBCCTL2 &= ~OUTMOD0);
100
101 #define TIMER_B_0_OUTMOD_RESET      (TBCCTL0 |= (OUTMOD2 | OUTMOD0)); \
102                                     (TBCCTL0 &= ~OUTMOD1);
103 #define TIMER_B_1_OUTMOD_RESET      (TBCCTL1 |= (OUTMOD2 | OUTMOD0)); \
104                                     (TBCCTL1 &= ~OUTMOD1);
105 #define TIMER_B_2_OUTMOD_RESET      (TBCCTL2 |= (OUTMOD2 | OUTMOD0)); \
106                                     (TBCCTL2 &= ~OUTMOD1);
107
108 #define TIMER_B_0_CM_IRQ_ENABLE     (TBCCTL0 |= CCIE);
109 #define TIMER_B_1_CM_IRQ_ENABLE     (TBCCTL1 |= CCIE);
110 #define TIMER_B_2_CM_IRQ_ENABLE     (TBCCTL2 |= CCIE);
111
112 #define TIMER_B_0_CM_IRQ_DISABLE    (TBCCTL0 &= ~CCIE);
113 #define TIMER_B_1_CM_IRQ_DISABLE    (TBCCTL1 &= ~CCIE);
114 #define TIMER_B_2_CM_IRQ_DISABLE    (TBCCTL2 &= ~CCIE);
115
116
117 /**** Prototyp declaration ************************************************/
118 void timer_a_init(void);
119 void timer_a_init_balanc(void);
120 void timer_a_init_test(void);
121 void timer_b_init_burst(uint8_t);
122 void timer_b_init_burst_end(uint8_t);
123 void timer_b_init(void);
124
125
126 #endif
```

F.2. Batteriesteuergerät

F.2.1. cc1101.c

```
1   /*****************************************************************************
2   **    Name:           cc1101.c
3   **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4   **    Date:           07/03/2013
5   **    Author:         Nico Sassano
6   **                    modified taken from Phillip Durdaut
7   *****************************************************************************/
8   #include "main.h"
9
10  /* Output power table (433 MHz): -30, 10 (dBm) */
11  uint8_t cc1101_patable[] = { 0x12, 0xc0 };
12  uint8_t cc1101_patable_a[] = {0xc0 };
13  uint8_t cc1101_patablelen = 2;
14
15  extern volatile uint8_t irq_data_received_flag;
16
17  /*** Command Strobes (Table 42 in datasheet) ******************************/
18  #define CC1101_CS_SRES              0x30
19  #define CC1101_CS_SFSTXON           0x31
20  #define CC1101_CS_SXOFF             0x32
21  #define CC1101_CS_SCAL              0x33
22  #define CC1101_CS_SRX               0x34
23  #define CC1101_CS_STX               0x35
24  #define CC1101_CS_SIDLE             0x36
25  #define CC1101_CS_SAFC              0x37
26  #define CC1101_CS_SWOR              0x38
27  #define CC1101_CS_SPWD              0x39
28  #define CC1101_CS_SFRX              0x3A
29  #define CC1101_CS_SFTX              0x3B
30  #define CC1101_CS_SWORRST           0x3C
```

```
31  #define CC1101_CS_SNOP              0x3D
32
33  /*** Configuration Registers **************************************************/
34  #define CC1101_CR_IOCFG2            0x00    /* GDO2 output pin configuration */
35  #define CC1101_CR_IOCFG1            0x01    /* GDO1 output pin configuration */
36  #define CC1101_CR_IOCFG0            0x02    /* GDO0 output pin configuration */
37  #define CC1101_CR_FIFOTHR           0x03    /* RX FIFO and TX FIFO thresholds */
38  #define CC1101_CR_SYNC1             0x04    /* Sync word, high byte */
39  #define CC1101_CR_SYNC0             0x05    /* Sync word, low byte */
40  #define CC1101_CR_PKTLEN            0x06    /* Packet length */
41  #define CC1101_CR_PKTCTRL1          0x07    /* Packet automation control */
42  #define CC1101_CR_PKTCTRL0          0x08    /* Packet automation control */
43  #define CC1101_CR_ADDR              0x09    /* Device address */
44  #define CC1101_CR_CHANNR            0x0A    /* Channel number */
45  #define CC1101_CR_FSCTRL1           0x0B    /* Frequency synthesizer control */
46  #define CC1101_CR_FSCTRL0           0x0C    /* Frequency synthesizer control */
47  #define CC1101_CR_FREQ2             0x0D    /* Frequency control word, high byte */
48  #define CC1101_CR_FREQ1             0x0E    /* Frequency control word, middle byte */
49  #define CC1101_CR_FREQ0             0x0F    /* Frequency control word, low byte */
50  #define CC1101_CR_MDMCFG4           0x10    /* Modem configuration */
51  #define CC1101_CR_MDMCFG3           0x11    /* Modem configuration */
52  #define CC1101_CR_MDMCFG2           0x12    /* Modem configuration */
53  #define CC1101_CR_MDMCFG1           0x13    /* Modem configuration */
54  #define CC1101_CR_MDMCFG0           0x14    /* Modem configuration */
55  #define CC1101_CR_DEVIATN           0x15    /* Modem deviation setting */
56  #define CC1101_CR_MCSM2             0x16    /* Main Radio Cntrl State Machine config */
57  #define CC1101_CR_MCSM1             0x17    /* Main Radio Cntrl State Machine config */
58  #define CC1101_CR_MCSM0             0x18    /* Main Radio Cntrl State Machine config */
59  #define CC1101_CR_FOCCFG            0x19    /* Frequency Offset Compensation config */
60  #define CC1101_CR_BSCFG             0x1A    /* Bit Synchronization configuration */
61  #define CC1101_CR_AGCCTRL2          0x1B    /* AGC control */
62  #define CC1101_CR_AGCCTRL1          0x1C    /* AGC control */
63  #define CC1101_CR_AGCCTRL0          0x1D    /* AGC control */
64  #define CC1101_CR_WOREVT1           0x1E    /* High byte Event 0 timeout */
65  #define CC1101_CR_WOREVT0           0x1F    /* Low byte Event 0 timeout */
66  #define CC1101_CR_WORCTRL           0x20    /* Wake On Radio control */
67  #define CC1101_CR_FREND1            0x21    /* Front end RX configuration */
68  #define CC1101_CR_FREND0            0x22    /* Front end TX configuration */
69  #define CC1101_CR_FSCAL3            0x23    /* Frequency synthesizer calibration */
70  #define CC1101_CR_FSCAL2            0x24    /* Frequency synthesizer calibration */
71  #define CC1101_CR_FSCAL1            0x25    /* Frequency synthesizer calibration */
72  #define CC1101_CR_FSCAL0            0x26    /* Frequency synthesizer calibration */
73  #define CC1101_CR_RCCTRL1           0x27    /* RC oscillator configuration */
74  #define CC1101_CR_RCCTRL0           0x28    /* RC oscillator configuration */
75  #define CC1101_CR_FSTEST            0x29    /* Frequency synthesizer cal control */
76  #define CC1101_CR_PTEST             0x2A    /* Production test */
77  #define CC1101_CR_AGCTEST           0x2B    /* AGC test */
78  #define CC1101_CR_TEST2             0x2C    /* Various test settings */
79  #define CC1101_CR_TEST1             0x2D    /* Various test settings */
80  #define CC1101_CR_TEST0             0x2E    /* Various test settings */
81
82  /*** Status Registers *********************************************************/
83  #define CC1101_SR_PARTNUM           0x30    /* Part number */
84  #define CC1101_SR_VERSION           0x31    /* Current version number */
85  #define CC1101_SR_FREQEST           0x32    /* Frequency offset estimate */
86  #define CC1101_SR_LQI               0x33    /* Demodulator estimate for link quality */
87  #define CC1101_SR_RSSI              0x34    /* Received signal strength indication */
88  #define CC1101_SR_MARCSTATE         0x35    /* Control state machine state */
89  #define CC1101_SR_WORTIME1          0x36    /* High byte of WOR timer */
90  #define CC1101_SR_WORTIME0          0x37    /* Low byte of WOR timer */
91  #define CC1101_SR_PKTSTATUS         0x38    /* Current GDOx status and packet status */
92  #define CC1101_SR_VCO_VC_DAC        0x39    /* Current setting from PLL cal module */
93  #define CC1101_SR_TXBYTES           0x3A    /* Underflow and # of bytes in TXFIFO */
94  #define CC1101_SR_RXBYTES           0x3B    /* Overflow and # of bytes in RXFIFO */
95  #define CC1101_SR_RCCTRL1_STATUS    0x3C    /* Last RC oscilator calibration results */
96  #define CC1101_SR_RCCTRL0_STATUS    0x3D    /* Last RC oscilator calibration results */
97
98  /*** Single / Burst access ****************************************************/
99  #define CC1101_WRITE_BURST          0x40
100 #define CC1101_READ_SINGLE          0x80
101 #define CC1101_READ_BURST           0xC0
102
103 /* Memory locations */
104
105 #define CC1101_ML_PATABLE           0x3E
106 #define CC1101_ML_TXFIFO            0x3F
107 #define CC1101_ML_RXFIFO            0x3F
108
109 /******************************************************************************
110 **  Prototypen deklaration
111 ******************************************************************************/
112
113 void cc1101_spi_setup(void);
114 void cc1101_spi_write_register(uint8_t address, uint8_t value);
115 void cc1101_spi_write_register_burst(uint8_t address, uint8_t * buffer, uint8_t count);
```

```
116 char cc1101_spi_read_register(uint8_t address);
117 void cc1101_spi_read_register_burst(uint8_t address, uint8_t *buffer, uint8_t count);
118 uint8_t cc1101_spi_read_status(uint8_t address);
119 void cc1101_spi_command_strobe(uint8_t strobe);
120
121 void cc1101_set_tx(uint8_t brate) {
122     cc1101_init_tx();               // init for TX
123     cc1101_reset();                 // Reset chip and go to idle state
124     cc1101_config_packet(brate);    // Configure the transceiver for sending packets
125 }
126
127 void cc1101_set_rx(uint8_t brate) {
128     irq_data_received_flag = 0;     // Reset flag
129     cc1101_init_rx();               // Init RX
130     cc1101_reset();                 // Reset chip and go to idle state
131     cc1101_config_packet(brate);    // Configure the transceiver for sending packets
132 }
133
134 void cc1101_init_rx(void) {
135     cc1101_spi_setup();             // Set SPI
136     CC1101_GDO2_DIR_IN;             // Set direction
137     CC1101_GDO2_IRQ_RISING_EDGE;    // Set rising edge
138     CC1101_GDO2_IRQ_DISABLE;        // IRQ disable
139     CC1101_GDO2_CLEAR_IRQ;
140 }
141
142 void cc1101_init_tx(void) {
143     cc1101_spi_setup();             // Set SPI
144     CC1101_GDO2_DIR_IN;             // Set direction
145     CC1101_GDO2_IRQ_FALLING_EDGE;   // Set falling edge
146     CC1101_GDO2_IRQ_DISABLE;        // IRQ disable
147     CC1101_GDO2_CLEAR_IRQ;
148 }
149
150 void cc1101_power_up_reset(void) {
151     // Manual power-on reset
152     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;        // Chip enable
153     delay_100us(1);
154     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;         // Chip disable
155     delay_100us(1);
156     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;        // Chip enable
157     delay_100us(5);
158     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;         // Chip disable
159     delay_100us(5);
160
161     cc1101_reset();
162 }
163
164 void cc1101_reset(void) {
165     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;        // Chip enable
166     while (!(IFG1 & UTXIFG0));                  // Wait for TX to finish
167     U0TXBUF = CC1101_CS_SRES;                   // Send reset strobe
168     while(!(UTCTL0 & TXEPT));                   // Wait for TX to finish
169     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;         // Chip disable
170     delay_ms(1);
171 }
172
173 void cc1101_config_no_packet(void) {
174     // GDO0 at high impedance
175     cc1101_spi_write_register(CC1101_CR_IOCFG0, 0x2E);
176
177     // GDO2 at high impedance
178     cc1101_spi_write_register(CC1101_CR_IOCFG2, 0x2E);
179
180     // Asynchronous serial mode, Infinite packet length mode
181     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, 0x32);
182
183     // Channel 0
184     cc1101_spi_write_register(CC1101_CR_CHANNR, 0x00);
185
186     // Carrier frequency: 433.999969 MHz
187     cc1101_spi_write_register(CC1101_CR_FREQ2, 0x10);
188     cc1101_spi_write_register(CC1101_CR_FREQ1, 0xB1);
189     cc1101_spi_write_register(CC1101_CR_FREQ0, 0x3B);
190
191     cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0xFD); //DATA_E = 13, CHANBW_M = 4, CHANBW_E = 4
192
193     // 250 kBaud, OOK
194     cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x3B); //DATA_M = 59
195     cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x30);
196
197     cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x00); //???
198
199     cc1101_spi_write_register(CC1101_CR_DEVIATN, 0x15); //???
200     cc1101_spi_write_register(CC1101_CR_MCSM0, 0x18);   //???
```

```
201
202         // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
203         cc1101_spi_write_register(CC1101_CR_FOCCFG, 0x16);
204
205         // 10 dBm output power
206         cc1101_spi_write_register_burst(CC1101_ML_PATABLE, cc1101_patable, 2);
207         cc1101_spi_write_register(CC1101_CR_FREND0, 0x11);
208     }
209
210     void cc1101_config_packet(uint8_t brate) {
211         // Rising edge on GDO0 when packet received and CRC check OK
212         // (Deasserted when first byte is read from RX FIFO)
213         cc1101_spi_write_register(CC1101_CR_IOCFG0, 0x07);
214
215         // Rising edge on GDO2 when sync word has been received
216         cc1101_spi_write_register(CC1101_CR_IOCFG2, 0x06);
217
218         // The 4 SYNC bytes: 0x12 0x09 (repeated once)
219         cc1101_spi_write_register(CC1101_CR_SYNC1, 0x81);
220         cc1101_spi_write_register(CC1101_CR_SYNC0, 0x81);
221
222         // Flush RX packets when CRC is not OK, Address check and 0x00 broadcast
223         cc1101_spi_write_register(CC1101_CR_PKTCTRL1, 0x0A);
224
225         // No whitening, FIFO mode, CRC disable, Fixed packet length
226         cc1101_spi_write_register(CC1101_CR_PKTCTRL0, 0x00);
227
228         // Device Address
229         cc1101_spi_write_register(CC1101_CR_ADDR, ADDRESS_BASE_STATION);
230
231         // Channel 0
232         cc1101_spi_write_register(CC1101_CR_CHANNR, 0x00);
233
234         // IF frequency: 152.34375 kHz
235         cc1101_spi_write_register(CC1101_CR_FSCTRL1, 0x06);
236
237         // Carrier frequency: 433.999969 MHz
238         cc1101_spi_write_register(CC1101_CR_FREQ2, 0x10);
239         cc1101_spi_write_register(CC1101_CR_FREQ1, 0xB1);
240         cc1101_spi_write_register(CC1101_CR_FREQ0, 0x3B);
241
242         switch(brate) {
243             /*************************************************************************************
244             ** 40 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
245             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
246             *************************************************************************************/
247             case 40: {
248                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8A);
249                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
250                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
251                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
252                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
253             }break;
254
255             /*************************************************************************************
256             ** 59.906 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
257             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
258             *************************************************************************************/
259             case 60: {
260                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
261                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x2E);
262                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
263                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
264                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
265             }break;
266
267             /*************************************************************************************
268             ** 79.9408 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
269             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
270             *************************************************************************************/
271             case 80: {
272                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
273                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
274                 cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
275                 cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
276                 cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
277             }break;
278
279             /*************************************************************************************
280             ** 99.9756 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 203.125000 kHz
281             ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
282             *************************************************************************************/
283             case 100: {
284                 cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8B);
285                 cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0xF8);
```

```
286             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
287             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
288             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
289         }break;
290
291         /*****************************************************************************
292         ** 119.812 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
293         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
294         *****************************************************************************/
295         case 120: {
296             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
297             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x2E);
298             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
299             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
300             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
301         }break;
302
303         /*****************************************************************************
304         ** 140.045 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
305         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
306         *****************************************************************************/
307         case 140: {
308             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
309             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x61);
310             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
311             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
312             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
313         }break;
314
315         /*****************************************************************************
316         ** 159.882 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
317         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
318         *****************************************************************************/
319         case 160: {
320             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
321             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0x93);
322             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
323             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
324             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
325         }break;
326
327         /*****************************************************************************
328         ** 180.115 kBaud, OOK, Channel spacing: 149.963379 kHz, RX filter bandwidth: 406.250000 kHz
329         ** No Manchester en/decoding, 30/32 SYNC bits detected, No FEC, 4 Preamble bytes
330         *****************************************************************************/
331         case 180: {
332             cc1101_spi_write_register(CC1101_CR_MDMCFG4, 0x8C);
333             cc1101_spi_write_register(CC1101_CR_MDMCFG3, 0xC6);
334             cc1101_spi_write_register(CC1101_CR_MDMCFG2, 0x33);
335             cc1101_spi_write_register(CC1101_CR_MDMCFG1, 0x22);
336             cc1101_spi_write_register(CC1101_CR_MDMCFG0, 0x7A);
337         }break;
338
339     }
340
341     // Calibrate when going from IDLE to RX or TX, Crystal off when in SLEEP state
342     cc1101_spi_write_register(CC1101_CR_MCSM0, 0x10);
343
344     // FCL gain: 3000, Saturation point for the frequency offset compensation algorithm -> SmartRF Studio
345     cc1101_spi_write_register(CC1101_CR_FOCCFG, 0x16);      //Verändern
346
347     //cc1101_spi_write_register(CC1101_CR_FREND1, 0x00);
348     cc1101_spi_write_register_burst(CC1101_ML_PATABLE, cc1101_patable, cc1101_patablelen);
349     cc1101_spi_write_register(CC1101_CR_FREND0, 0x11);     // 10 dBm output power
350 }
351
352 void cc1101_fill_tx_fifo(uint8_t * buffer, uint8_t length) {
353     // Clear TX FIFO
354     cc1101_clear_tx_fifo();
355
356     // Transfer the bytes via SPI to the TX fifo of the transceiver
357     cc1101_spi_write_register_burst(CC1101_ML_TXFIFO, buffer, length);
358 }
359
360 void cc1101_read_rx_fifo(uint8_t * buffer, uint8_t length) {
361     // Disable the receiver
362     cc1101_idle();
363
364     // Transfer the bytes via SPI from the RX fifo
365     cc1101_spi_read_register_burst(CC1101_ML_RXFIFO, buffer, length);
366 }
367
368 void cc1101_enable_crc(void) {
369     uint8_t current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
370     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, current | BIT2);
```

```
371 }
372
373 void cc1101_disable_crc(void) {
374     uint8_t current = cc1101_spi_read_register(CC1101_CR_PKTCTRL0);
375     cc1101_spi_write_register(CC1101_CR_PKTCTRL0, current & ~BIT2);
376 }
377
378 void cc1101_clear_tx_fifo(void) {
379     cc1101_spi_command_strobe(CC1101_CS_SFTX);
380 }
381
382 void cc1101_clear_rx_fifo(void) {
383     cc1101_spi_command_strobe(CC1101_CS_SFRX);
384 }
385
386 uint8_t cc1101_get_rxbytes(void) {
387     return (cc1101_spi_read_status(CC1101_SR_RXBYTES) & 0x7F);
388 }
389
390 void cc1101_idle(void) {
391     cc1101_spi_command_strobe(CC1101_CS_SIDLE);
392 }
393
394 void cc1101_tx_asynchronous_mode(void) {
395     cc1101_spi_command_strobe(CC1101_CS_STX);
396 }
397
398 void cc1101_tx(uint8_t packetes_to_tx) {
399     // DATA packet length
400     cc1101_spi_write_register(CC1101_CR_PKTLEN, packetes_to_tx);
401
402     // Enable CRC calculation when transmitting data
403     cc1101_enable_crc();
404
405     // TX state
406     cc1101_spi_command_strobe(CC1101_CS_STX);
407 }
408
409 void cc1101_rx(uint8_t packetes_to_rx) {
410     // DATA packet length
411     cc1101_spi_write_register(CC1101_CR_PKTLEN, packetes_to_rx);
412
413     // Enable CRC check when receiving data
414     cc1101_enable_crc();
415
416     // Clear RX FIFO
417     cc1101_clear_rx_fifo();
418
419     // RX state
420     cc1101_spi_command_strobe(CC1101_CS_SRX);
421 }
422
423 uint8_t cc1101_get_partnum(void) {
424     return cc1101_spi_read_status(CC1101_SR_PARTNUM);
425 }
426
427 uint8_t cc1101_get_version(void) {
428     return cc1101_spi_read_status(CC1101_SR_VERSION);
429 }
430
431 uint8_t cc1101_get_marcstate(void) {
432     return (cc1101_spi_read_status(CC1101_SR_MARCSTATE) & 0x1F);
433 }
434
435
436 void cc1101_spi_setup(void) {
437     CC1101_CSn_PxDIR |= CC1101_CSn_PIN;    // nCS is output
438     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;    // Unselect CC1101 chip
439
440     ME1 |= USPIE0;                         // Enable USART0 SPI
441     U0CTL |= SWRST;                        // USART logic held in reset state
442     U0CTL |= (CHAR | SYNC | MM);           // 8-bit, SPI, Master
443     U0TCTL |= (CKPH | SSEL1 | SSEL0 | STC);  // UCLK delayed, SMCLK, 3-pin SPI mode
444
445     U0BR0 = 2;  // BRCLK/2
446     U0BR1 = 0;
447     U0MCTL = 0; // Always 0 in SPI mode
448
449     // SPI functionality for pins
450     CC1101_SPI_PxSEL |= (CC1101_SPI_SI_PIN | CC1101_SPI_SO_GDO1_PIN | CC1101_SPI_SCLK_PIN);
451     CC1101_SPI_PxDIR |= (CC1101_SPI_SI_PIN | CC1101_SPI_SCLK_PIN); // SI and SCLK are outputs
452
453     UCTL0 &= ~SWRST; // Initialize USART state machine
454 }
455
```

```
456 void cc1101_spi_write_register(uint8_t address, uint8_t value) {
457     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;       // Chip enable
458     while (!(IFG1&UTXIFG0));                   // Wait for TX to finish
459     U0TXBUF = address;                         // Send address
460     while (!(IFG1&UTXIFG0));                   // Wait for TX to finish
461     U0TXBUF = value;                           // Send value
462     while(!(UTCTL0&TXEPT));                    // Wait for TX complete
463     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;        // Chip disable
464 }
465
466 void cc1101_spi_write_register_burst(uint8_t address, uint8_t * buffer, uint8_t count) {
467     uint8_t i;
468
469     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;       // Chip enable
470     while (!(IFG1 & UTXIFG0));                 // Wait for TX to finish
471     U0TXBUF = address | CC1101_WRITE_BURST;    // Send address
472
473     for (i = 0; i < count; i++) {
474         while (!(IFG1 & UTXIFG0));             // Wait for TX to finish
475         U0TXBUF = buffer[i];                   // Send data
476     }
477
478     while(!(UTCTL0 & TXEPT));
479     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;        // Chip disable
480 }
481
482 char cc1101_spi_read_register(uint8_t address) {
483     uint8_t x;
484
485     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;          // Chip enable
486     while (!(IFG1 & UTXIFG0));                    // Wait for TX to finish
487     U0TXBUF = (address | CC1101_READ_SINGLE);     // Send address
488     while (!(IFG1 & UTXIFG0));                    // Wait for TX to finish
489     U0TXBUF = 0;                                  // Dummy write so we can read data
490     while(!(UTCTL0 & TXEPT));                     // Wait for TX complete
491     x = U0RXBUF;                                  // Read data
492     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;           // Chip disable
493
494     return x;
495 }
496
497 void cc1101_spi_read_register_burst(uint8_t address, uint8_t * buffer, uint8_t count) {
498     uint16_t i;
499
500     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;       // Chip enable
501     while (!(IFG1 & UTXIFG0));                 // Wait for TXBUF ready
502     U0TXBUF = (address | CC1101_READ_BURST);   // Send address
503     while(!(UTCTL0 & TXEPT));                  // Wait for TX complete
504
505     U0TXBUF = 0;
506     while (!(IFG1&URXIFG0));
507
508     for (i = 0; i < count; i++) {
509         U0TXBUF = 0;                           // Initiate next data RX, meanwhile
510         while (!(IFG1&URXIFG0));               // Wait for end of TX data byte
511         buffer[i] = U0RXBUF;                   // Store data from last data RX
512     }
513
514     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;        // Chip disable
515 }
516
517 uint8_t cc1101_spi_read_status(uint8_t address) {
518     uint8_t status;
519
520     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;       // Chip enable
521     while (!(IFG1 & UTXIFG0));                 // Wait for TX to finish
522     U0TXBUF = (address | CC1101_READ_BURST);   // Send address
523     while (!(IFG1 & UTXIFG0));                 // Wait for TX to finish
524     U0TXBUF = 0;                               // Dummy write so we can read data
525     while(!(UTCTL0 & TXEPT));                  // Wait for TX complete
526     status = U0RXBUF;                          // Read data
527     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;        // Chip disable
528
529     return status;
530 }
531
532 void cc1101_spi_command_strobe(uint8_t strobe) {
533     CC1101_CSn_PxOUT &= ~CC1101_CSn_PIN;       // Chip enable
534     while (!(IFG1 & UTXIFG0));                 // Wait for TX to finish
535     U0TXBUF = strobe;                          // Send strobe
536     IFG1 &= ~URXIFG0;                          // Clear flag
537     while(!(UTCTL0 & TXEPT));                  // Wait for TX complete
538     CC1101_CSn_PxOUT |= CC1101_CSn_PIN;        // Chip disable
539 }
```

F.2.2. cc1101.h

```
/*******************************************************************************
**     Name:           cc1101.h
**     Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
**     Date:           07/03/2013
**     Author:         Nico Sassano
**                     modified taken from Phillip Durdaut
*******************************************************************************/

#ifndef CC1101_H_
#define CC1101_H_

#include "main.h"

/*******************************************************************************
**  Defines
*******************************************************************************/
#define CC1101_CSn_PxDIR            P3DIR
#define CC1101_CSn_PxOUT            P3OUT
#define CC1101_CSn_PIN              BIT0

#define CC1101_SPI_PxSEL            P3SEL
#define CC1101_SPI_PxDIR            P3DIR
#define CC1101_SPI_PxIN             P3IN
#define CC1101_SPI_SI_PIN           BIT1
#define CC1101_SPI_SO_GDO1_PIN      BIT2
#define CC1101_SPI_SCLK_PIN         BIT3

#define CC1101_GDO0_PxDIR           P1DIR
#define CC1101_GDO0_PxIN            P1IN
#define CC1101_GDO0_PIN             BIT0
#define CC1101_GDO0_POS             0

#define CC1101_GDO2_PxDIR           P1DIR
#define CC1101_GDO2_PxIN            P1IN
#define CC1101_GDO2_PxIES           P1IES
#define CC1101_GDO2_PxIE            P1IE
#define CC1101_GDO2_PxIFG           P1IFG
#define CC1101_GDO2_PIN             BIT1
#define CC1101_GDO2_POS             1

/*******************************************************************************
**  Macros
*******************************************************************************/
#define CC1101_GDO0_DIR_IN              (CC1101_GDO0_PxDIR &= ~CC1101_GDO0_PIN)
#define CC1101_GDO0_IN                  ((CC1101_GDO0_PxIN & CC1101_GDO0_PIN) >> CC1101_GDO0_POS)

#define CC1101_GDO2_DIR_IN              (CC1101_GDO2_PxDIR &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IN                  ((CC1101_GDO2_PxIN & CC1101_GDO2_PIN) >> CC1101_GDO2_POS)
#define CC1101_GDO2_IRQ_RISING_EDGE     (CC1101_GDO2_PxIES &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_FALLING_EDGE    (CC1101_GDO2_PxIES |= CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_ENABLE          (CC1101_GDO2_PxIE |= CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_DISABLE         (CC1101_GDO2_PxIE &= ~CC1101_GDO2_PIN)
#define CC1101_GDO2_IRQ_PENDING         ((CC1101_GDO2_PxIFG & CC1101_GDO2_PIN) == CC1101_GDO2_PIN)
#define CC1101_GDO2_CLEAR_IRQ           (CC1101_GDO2_PxIFG &= ~(CC1101_GDO2_PIN))

void cc1101_set_tx(uint8_t);
void cc1101_set_rx(uint8_t);
void cc1101_init_rx(void);
void cc1101_init_tx(void);
void cc1101_power_up_reset(void);
void cc1101_reset(void);
void cc1101_config_no_packet_tx(void);
void cc1101_config_no_packet_rx(void);
void cc1101_config_no_packet(void);
void cc1101_config_packet(uint8_t);
void cc1101_config_packet_tx(void);
void cc1101_config_packet_rx(void);
void cc1101_fill_tx_fifo(uint8_t * buffer, uint8_t length);
void cc1101_read_rx_fifo(uint8_t * buffer, uint8_t length);
void cc1101_enable_crc(void);
void cc1101_disable_crc(void);
void cc1101_clear_tx_fifo(void);
void cc1101_clear_rx_fifo(void);
uint8_t cc1101_get_rxbytes(void);
void cc1101_idle(void);
void cc1101_tx_asynchronous_mode(void);
void cc1101_tx(uint8_t);
void cc1101_rx(uint8_t);
uint8_t cc1101_get_partnum(void);
uint8_t cc1101_get_version(void);
uint8_t cc1101_get_marcstate(void);

#endif /* CC1101_H_ */
```

F.2.3. clk.c

```
1  /*****************************************************************************
2  **     Name:              clk.c
3  **     Hardware:          BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **     Date:              07/04/2013
5  **     Author:            Nico Sassano
6  **                        modified taken from Simon Puettjer
7  *****************************************************************************/
8  #include "main.h"
9
10 void osc_configuration(void) {
11     uint16_t max_tries = 1000;
12     uint8_t i;
13
14     _BIC_SR(OSCOFF);                    // Switch LFXT on
15     BCSCTL1 &= ~XT2OFF;                 // XT2on
16     do {
17         IFG1 &= ~OFIFG;                 // Clear OSCFault flag
18         for (i = 0xFF; i > 0; i--);     // Time for flag to set
19         if (!(--max_tries)) {           // Stop after 1k tries, error with ext osc!
20         }
21     }
22     while ((IFG1 & OFIFG));             // OSCFault flag still set?
23     IFG1 &= ~OFIFG;                     // Clear OSCFault flag
24
25     BCSCTL2 |= SELM_2 | SELS;           // MCLK = SMCLK = XT2 (safe)
26
27     // MCLK = 8 MHz
28     BCSCTL2 &= ~(DIVS_3 | DIVM_3);      // SMCLK = MCLK = XT2 / 1
29 }
```

F.2.4. clk.h

```
1  /*******************************************************************************
2  **    Name:              clk.h
3  **    Hardware:          BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:              07/03/2013
5  **    Author:            Nico Sassano
6  **                       modified taken from Simon Puettjer
7  *******************************************************************************/
8
9  #ifndef CLK_H_
10 #define CLK_H_
11
12 #include "main.h"
13
14 void osc_configuration(void);
15
16 #endif /* CLK_H_ */
```

F.2.5. delay.c

```
1  /*******************************************************************************
2  **    Name:           delay.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           07/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *******************************************************************************/
8
9  #include "main.h"
10
11 void delay(uint16_t i);
12
13 void delay_ms(uint16_t delay_ms)
14 {
15     while (delay_ms > 0) {
16         delay(DELAY_CYCLES_PER_MS);
17         delay_ms--;
18     }
19 }
20
21 void delay_100us(uint16_t delay_100us)
22 {
23     while (delay_100us > 0) {
24         delay(DELAY_CYCLES_PER_100US);
25         delay_100us--;
26     }
27 }
28
29 void delay(uint16_t i)
30 {
31     while (i > 0) {
32         _NOP();
33         i--;
34     }
35 }
```

F.2.6. delay.h

```
1  /*******************************************************************************
2  **    Name:           delay.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           07/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *******************************************************************************/
8
9  #ifndef DELAY_H_
10 #define DELAY_H_
11
12 #include "main.h"
13
14 #define DELAY_CYCLES_PER_MS      610
15 #define DELAY_CYCLES_PER_100US   57
16
17 void delay_ms(uint16_t delay_ms);
18 void delay_100us(uint16_t delay_100us);
19
20 #endif /* DELAY_H_ */
```

F.2.7. isr.c

```
1  /*******************************************************************************
2  **     Name:           isr.c
3  **     Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 04/2013
4  **     Date:           02/04/2013
5  **     Author:         Nico Sassano
6  *******************************************************************************/
7  #include "main.h"
8
9  /*********************************************
10 * Data Buffer
11 *********************************************/
12 extern volatile uint8_t rx_data[64];
13
14 /*********************************************
15 ** Frames
16 *********************************************/
17 extern volatile uint8_t rx_data_length;
18
19 /*********************************************
20 * Interruptflags
21 *********************************************/
22 extern volatile uint8_t irq_data_received_flag;
23 extern volatile uint8_t irq_mode;
24 extern volatile uint8_t irq_send_done_flag;
25
26 extern volatile uint8_t waiting_time;        // Wartezeit zählen
27 extern volatile uint16_t asynch_counter;     // Zählt die Perioden im asynchronen Modus
28
29 interrupt (PORT1_VECTOR) isr_port1(void) {
30
31     /*****************************************************************
32     ** Daten wurden im Tranciver empfangen
33     *****************************************************************/
34     if ((CC1101_GDO2_IRQ_PENDING) && (IRQ_IS_RX)) {  // Sync word detected
35         CC1101_GDO2_IRQ_DISABLE;
36         CC1101_GDO2_CLEAR_IRQ;
37
38         // Wait until packet completely received
39         while(CC1101_GDO2_IN);
40
41         // Check whether received data is valid
42         if ((CC1101_GDO0_IN ) && cc1101_get_rxbytes() == HEADER_LENGTH + rx_data_length) {
43             uint8_t rx_buf[HEADER_LENGTH + rx_data_length];
44
45             cc1101_read_rx_fifo(rx_buf, HEADER_LENGTH + rx_data_length);
46
47             // Data packet BASE_STATION
48             if (rx_buf[0] == ADDRESS_BASE_STATION) {
49
50                 uint8_t index;
51                 for (index = 0; index < (HEADER_LENGTH + rx_data_length); index++) {
52                     rx_data[index] = rx_buf[index];
53                 }
54                 irq_data_received_flag = 1;
55             }
56         }
57         cc1101_idle();       // Transceiver back to IDLE state
58     }
59     /*****************************************************************
60     ** IRQ wenn gesendet wurde
61     *****************************************************************/
62     if ((CC1101_GDO2_IRQ_PENDING) && (IRQ_IS_TX)) {
63         CC1101_GDO2_IRQ_DISABLE;
64         CC1101_GDO2_CLEAR_IRQ;
65         irq_send_done_flag = 1; // Flag setzten
66     }
67 }
68
69 /*******************************************************************************
70 ** ISR TIMERA0: Asynchron Counter
71 *******************************************************************************/
72 interrupt ( TIMERA0_VECTOR ) isr_timer_A0 (void) {
73
74     P1OUT |= BIT2;
75     TIMER_A_0_CM_IRQ_DISABLE
76     TIMER_A_1_CM_IRQ_ENABLE;
77     TAR = 0;
78     asynch_counter++;
79 }
80
81 /*******************************************************************************
82 ** ISR TIMERA1: Asynchron Counter
83 *******************************************************************************/
```

```
84  interrupt ( TIMERA1_VECTOR ) isr_timer_A1 (void) {
85      P1OUT &= ~BIT2;
86      TIMER_A_FLAG1_RESET;
87  }
88
89  /*************************************************************************
90  ** ISR TIMERB0: Asynchron Counter
91  *************************************************************************/
92  interrupt ( TIMERB0_VECTOR ) isr_timer_B0 (void) {
93      waiting_time++;
94  }
```

F.2.8. lcd16x2.c

```
1  /*-------------------------------------------------------------------
2      Project:              LCD16x2 functions
3      Used components:      MSP430-169STK
4      Date:                 10/28/2007
5      Last update:          04/04/2008
6      Author:               Stephan Plaschke
7
8      Modified by:          Nico Sassano
9      Last modification:    13/03/2013
10 -------------------------------------------------------------------*/
11
12 /*-------------------------------------------------------------------
13     Headerfiles
14 -------------------------------------------------------------------*/
15 //#include "lcd16x2.h"
16 //#include "types.h"
17 //#include <msp430x16x.h>
18 #include "main.h"
19
20 /*-------------------------------------------------------------------
21     x ms delay loop
22 -------------------------------------------------------------------*/
23 void LCD_delay_ms(uint16_t LCD_delay_ms)
24 /*-------------------------------------------------------------------*/
25 {
26     uint16_t LCD_delay_loop;
27
28
29     LCD_delay_loop = 610;    //(610*b) cycles (for 8MHz)
30
31     for (; LCD_delay_ms>0; LCD_delay_ms--)
32     {
33     LCD_delay(LCD_delay_loop);
34     }
35 }
36
37 /*-------------------------------------------------------------------
38     delay loop
39     counts x times
40 -------------------------------------------------------------------*/
41 void LCD_delay(uint16_t LCD_delay_loop)
42 /*-------------------------------------------------------------------*/
43 {
44     for(; LCD_delay_loop>0; LCD_delay_loop--) _NOP();
45 }
46
47 /*-------------------------------------------------------------------
48     Init 16x2LCD
49 -------------------------------------------------------------------*/
50 void LCD_init(void)
51 /*-------------------------------------------------------------------*/
52 {
53     LCD_DATA_PORT &= ~LCD_RS_PIN;        // sets LCD in COMMAND MODE
54     LCD_delay_ms(15);                    // Delay ca.15ms
55
56     LCD_DATA_PORT |= BIT4 | BIT5;        // D7-D4 = 0011
57     LCD_DATA_PORT &= ~BIT6 & ~BIT7;
58     LCD_enable();
59     LCD_delay_ms(5);
60     LCD_enable();
61     LCD_delay_ms(1);
62     LCD_enable();
63     LCD_delay_ms(1);
64     LCD_DATA_PORT &= ~BIT4;              // D7-D4 = 0010
65     LCD_enable();
66
67     LCD_send_cmd(LCD_ON);                //   switch LCD on
68     LCD_send_cmd(LCD_CLR);               //   clear LCD
69     LCD_send_cmd(LCD_LINE1);             //   set cursor to x=0, y=0
70 }
71
72 /*-------------------------------------------------------------------
73     Toggle LCD_E pin
74 -------------------------------------------------------------------*/
75 void LCD_enable(void)
76 /*-------------------------------------------------------------------*/
77 {
78     LCD_DATA_PORT |= LCD_E_PIN;
79     _NOP();
80     _NOP();
81     LCD_DATA_PORT &= ~LCD_E_PIN;
82 }
83
```

```
84  /*-------------------------------------------------------------------------
85      Send command to LCD
86  -------------------------------------------------------------------------*/
87  void LCD_send_cmd (uint8_t LCD_out_byte)
88  /*-------------------------------------------------------------------------*/
89  {
90      uint8_t LCD_tmp;
91
92      LCD_delay_ms(1);
93      LCD_tmp = LCD_out_byte & 0xf0;       // get upper nibble
94      LCD_DATA_PORT &= 0x0f;
95      LCD_DATA_PORT |= LCD_tmp;            // send CMD to LCD
96      LCD_DATA_PORT &= ~LCD_RS_PIN;        // sets LCD in COMMAND MODE
97      LCD_enable();
98      LCD_tmp = LCD_out_byte & 0x0f;
99      LCD_tmp = LCD_tmp << 4;              // get down nibble
100     LCD_DATA_PORT &= 0x0f;
101     LCD_DATA_PORT |= LCD_tmp;
102     LCD_DATA_PORT &= ~LCD_RS_PIN;        // sets LCD in COMMAND MODE
103     LCD_enable();
104 }
105
106 /*-------------------------------------------------------------------------
107     Send DATA to LCD
108 -------------------------------------------------------------------------*/
109 void LCD_send_data (uint8_t LCD_out_byte)
110 /*-------------------------------------------------------------------------*/
111 {
112     uint8_t LCD_tmp;
113
114     LCD_delay_ms(1);
115     LCD_tmp = LCD_out_byte & 0xf0;       // get upper nibble
116     LCD_DATA_PORT &= 0x0f;
117     LCD_DATA_PORT |= LCD_tmp;            // send DATA to LCD
118     LCD_DATA_PORT |= LCD_RS_PIN;         // sets LCD in DATA MODE
119     LCD_enable();
120     LCD_tmp = LCD_out_byte & 0x0f;
121     LCD_tmp = LCD_tmp << 4;              // get lower nibble
122     LCD_DATA_PORT &= 0x0f;
123     LCD_DATA_PORT |= LCD_tmp;
124     LCD_DATA_PORT |= LCD_RS_PIN;         // sets LCD in DATA MODE
125     LCD_enable();
126 }
127
128 /*-------------------------------------------------------------------------
129     Send string to LCD
130 -------------------------------------------------------------------------*/
131 void LCD_send_text(char *out_string)
132 /*-------------------------------------------------------------------------*/
133 {
134     while (*out_string)                  // *out_string != '\0'
135     {
136         LCD_send_data(*out_string);      // send byte to LCD
137         out_string++;
138     }
139 }
140
141 /*-------------------------------------------------------------------------*/
142 void LCD_send_int(int input)
143 /*-------------------------------------------------------------------------*/
144 {
145     unsigned short output = 0;
146     unsigned short leading_zeros = 1;
147     int divider = 10000;
148
149
150     if (input < 0)  // print 5 digits via uart
151     {
152         input = -1*(input);
153         LCD_send_text("-");
154     }
155
156     // if-else structure to print only needed digits
157     while (divider > 9)
158     {
159         output = input/divider;
160         if ((output!=0)||(leading_zeros!=1))
161         {
162             LCD_send_data((uint8_t) output+48);
163             leading_zeros = 0;
164         }
165         input = input-output*divider;
166         divider = divider/10;
167     }
168     LCD_send_data((uint8_t) input+48);
```

```
169 }
170
171 /*-------------------------------------------------------------------*/
172 void LCD_send_unsigned_int(int input)
173 /*-------------------------------------------------------------------*/
174 {
175     unsigned short output = 0;
176     unsigned short leading_zeros = 1;
177     int divider = 10000;
178
179     // if-else structure to print only needed digits
180     while (divider > 9)
181     {
182         output = input/divider;
183         if ((output!=0)||(leading_zeros!=1))
184         {
185             LCD_send_data((uint8_t) output+48);
186             leading_zeros = 0;
187         }
188         input = input-output*divider;
189         divider = divider/10;
190     }
191     LCD_send_data((uint8_t) input+48);
192 }
193
194 /*-------------------------------------------------------------------*/
195 void LCD_send_long(long input)
196 /*-------------------------------------------------------------------*/
197 {
198     unsigned short output = 0;
199     unsigned short leading_zeros = 1;
200     long divider = 1000000000;
201
202
203     if (input < 0)  // print 5 digits via uart
204     {
205         input = -1*(input);
206         LCD_send_text("-");
207     }
208
209     // if-else structure to print only needed digits
210     while (divider > 9)
211     {
212         output = input/divider;
213         if ((output!=0)||(leading_zeros!=1))
214         {
215             LCD_send_data((uint8_t) output+48);
216             leading_zeros = 0;
217         }
218         input = input-output*divider;
219         divider = divider/10;
220     }
221     LCD_send_data((uint8_t) input+48);
222 }
223
224 /*-------------------------------------------------------------------*/
225 void LCD_send_unsigned_long(long input)
226 /*-------------------------------------------------------------------*/
227 {
228     unsigned short output = 0;
229     unsigned short leading_zeros = 1;
230     long divider = 1000000000;
231
232     // if-else structure to print only needed digits
233     while (divider > 9)
234     {
235         output = input/divider;
236         if ((output!=0)||(leading_zeros!=1))
237         {
238             LCD_send_data((uint8_t) output+48);
239             leading_zeros = 0;
240         }
241         input = input-output*divider;
242         divider = divider/10;
243     }
244     LCD_send_data((uint8_t) input+48);
245 }
```

F.2.9. lcd16x2.h

```
1  /*-------------------------------------------------------------------
2      Project:              LCD16x2 module header
3      Used components:      MSP430-169STK
4      Date:                 10/28/2007
5      Last update:          10/28/2007
6      Author:               Stephan Plaschke
7
8      Modified by:          Nico Sassano
9      Last modification:    13/03/2013
10 -------------------------------------------------------------------*/
11
12 #ifndef LCD16X2_H_
13 #define LCD16X2_H_
14
15 /*-------------------------------------------------------------------
16     Headerfiles
17 -------------------------------------------------------------------*/
18 //#include "types.h"
19 #include "main.h"
20
21 /*-------------------------------------------------------------------
22     Global variables
23 -------------------------------------------------------------------*/
24
25 /*extern u8_t BATMON_control_reg;
26 extern u8_t SENSOR_last_data[8][40];// last received data from each sensor
27 extern u8_t CURRENT_IN_OUT;      // 1 = charge 0 = discharge
28 extern u8_t SENSOR_address[6];   // array with sensor address
29 extern u8_t SOC;                 // State of charge
30 extern u8_t SOH;                 // State of health
31
32 extern u8_t CURRENT_channel; //used sensor channel (1: ch1(low), 2: ch2(high))*/
33
34
35 /*-------------------------------------------------------------------
36     Defined values, change for each project
37 -------------------------------------------------------------------*/
38
39 #define    LCD_E_PIN          BIT1
40 #define    LCD_RS_PIN         BIT3
41 #define    LCD_LIGHT_PIN      BIT0
42 #define    LCD_DATA_PORT      P4OUT
43
44 /*-------------------------------------------------------------------
45     Defined values, equal in each project, don't change
46 -------------------------------------------------------------------*/
47
48 #define    LCD_LIGHT_ON       (LCD_DATA |= LCD_LIGHT_PIN) // sets pin4.0 HIGH
49 #define    LCD_LIGHT_OFF      (LCD_DATA &= ~LCD_LIGHT_PIN)     // clears pin4.0
50
51 #define    LCD_ON             0x0c     //  LCD control command, ON
52 #define    LCD_OFF            0x08     //  LCD control command, OFF
53 #define    LCD_CLR            0x01     //  LCD control command, CLEARSCREEN
54 #define    LCD_LINE1          0x80
55     //  LCD control command, sets cursor on y=0,x=0
56 #define    LCD_LINE2          0xc0
57     //  LCD control command, sets cursor on y=1,x=0
58 #define    LCD_ENDLINE1       0x8f
59     //  LCD control command, sets cursor on y=0,x=15
60 #define    LCD_ENDLINE2       0xcf
61     //  LCD control command, sets cursor on y=1,x=15
62
63 /*-------------------------------------------------------------------
64     Prototypes
65 -------------------------------------------------------------------*/
66
67 void LCD_delay_ms(uint16_t);
68 void LCD_delay(uint16_t);
69 void LCD_send_cmd (uint8_t);
70 void LCD_send_data (uint8_t);
71 void LCD_send_text(char *);
72 void LCD_enable(void);
73 void LCD_init(void);
74 //void LCD_show_voltage(uint8_t);
75 void LCD_send_int(int);
76 void LCD_send_unsigned_int(int);
77 void LCD_send_long(long);
78 void LCD_send_unsigned_long(long);
79
80 #endif /*16X2LCD_H_*/
```

F.2.10. main.c

```
1  /*****************************************************************************
2  **    Name:           main.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           07/03/2013
5  **    Author:         Nico Sassano
6  **                    modified taken from Phillip Durdaut
7  *****************************************************************************/
8  #include "main.h"
9
10 /*****************************************************************************
11 **  Global variables
12 *****************************************************************************/
13 uint8_t brate_start = 40;       // Standart Übertragungsrate
14 uint8_t brate = 100;            // Übertragungsrate nach der Konfiguration
15
16 const uint8_t sensor_address[] = { ADDRESS_SENSOR_1, ADDRESS_SENSOR_2,
17                                    ADDRESS_SENSOR_3, ADDRESS_SENSOR_4 };
18 volatile state_t state = S_INIT;
19 uint8_t current_sensor = 0;
20
21 volatile uint16_t asynch_counter = 0;     // Zählt die Perioden im asynchronen Modus
22
23 /*********************************************
24 * Data Buffer
25 *********************************************/
26 uint16_t burst_data[50] = {0};      // Speichert die Burstdaten
27 volatile uint8_t rx_data[64];       // Empfangsbuffer
28 uint16_t sample_buf_volt[4];        // Zwischenspeicher für Spannungswerte
29 uint16_t sample_buf_temp[4];        // Zwischenspeicher für Temperaturwerte TMP102
30 uint16_t sample_buf_temp_msp[4];    // Zwischenspeicher für Temperaturwerte MSP430
31
32 /*********************************************
33 ** Frames
34 *********************************************/
35 volatile uint8_t rx_data_length = 0;
36 volatile uint8_t tx_data_length = 0;
37 volatile uint8_t frame_counter  = 0;
38
39 /*********************************************
40 ** Burst Mode
41 *********************************************/
42 uint8_t burst_freq = BURST_FREQ_1000HZ; // Burst Frequenz
43 uint8_t burst_values = 0;               // Anzahl der Burst Werte
44 uint16_t seq_number = 0;                // Zählt die Seq.nummer
45
46 /*********************************************
47 **
48 *********************************************/
49 volatile uint8_t number_of_sample = 0;
50 volatile uint8_t packetes_per_sample = 0;
51
52 /*********************************************
53 ** Kalibrierung
54 *********************************************/
55 uint16_t cali_nominal_tmp102    =   0;
56 uint16_t cali_nominal_msp430    =   0;
57 uint8_t cali_pro                =   0;
58 uint8_t cali_neg                =   0;
59
60 /*********************************************
61 * Balancing
62 *********************************************/
63 volatile uint8_t balancing_on       = 0;
64 volatile uint8_t rx_balancing_state = 0;
65 volatile uint16_t upper_balanc_temp = 0x3B0;
66 volatile uint16_t lower_balanc_temp = 0x220;
67 volatile uint16_t upper_alarm_temp  = 0x1A00;
68 volatile uint16_t lower_alarm_temp  = 0x1900;
69
70 /*********************************************
71 * Interruptflags
72 *********************************************/
73 volatile uint8_t irq_data_received_flag = 0;
74 volatile uint8_t irq_mode = TX_MODE;
75 volatile uint8_t irq_send_done_flag = 0;
76 volatile uint8_t irq_button2_flag = 0;
77 volatile uint8_t irq_button3_flag = 0;
78 volatile uint8_t irq_balanc_on_flag = 0;
79 volatile uint8_t irq_balanc_off_flag = 0;
80
81 volatile uint8_t irq_timer_a_wakeup_flag = 0;
82 volatile uint8_t irq_timer_a_test_flag = 0;
```

```
83
84  /*********************************************
85  * Errorbuffer
86  *********************************************/
87  volatile uint8_t error_check_awake = 0;
88  volatile uint8_t error_tx = 0;
89
90  volatile uint8_t reciveString[10];
91  volatile uint8_t string_index = 0;
92
93  volatile uint8_t waiting_time = 0;
94
95  /*****************************************************************************
96  ** Prototypen deklaration
97  *****************************************************************************/
98  void osc_configuration(void);
99  void rx(uint8_t);
100 void tx_wakeup(void);
101 void rx_packet(uint8_t);
102 void tx_packet(uint8_t, uint8_t *);
103 void tx_uart(void);
104
105 //-------------------------------------------------------
106 int main (void) {
107 //-------------------------------------------------------
108
109     //-------------------------------------------------------
110     // OSC configuration & startup (8 MHz)
111     //-------------------------------------------------------
112     osc_configuration();
113
114     /*-------------------------------------------------------
115     ** Port konfiguration
116     -------------------------------------------------------*/
117     LED1_DIR_OUT;
118     LED1_OFF;
119
120     LED2_DIR_OUT;
121     LED2_OFF;
122
123     LCD_DIR_OUT;
124     LCD_OUT_0x00;
125
126     // TX pin should be input at first because the transceiver sends
127     // a clock signal on this pin on startup
128     TXPIN_MAN;
129     TXPIN_DIR_IN;
130
131     //-------------------------------------------------------
132     // LCD konfiguration
133     //-------------------------------------------------------
134     LCD_init();
135     LCD_send_cmd(LCD_CLR);
136     LCD_send_cmd(LCD_LINE1);
137     LCD_send_text("Please start    ");
138     LCD_send_cmd(LCD_LINE2);
139     LCD_send_text("the terminal    ");
140
141     //-------------------------------------------------------
142     // CC1101 konfiguration
143     //-------------------------------------------------------
144     cc1101_init_tx();
145     cc1101_power_up_reset();
146     cc1101_config_packet(brate);
147     cc1101_rx(HEADER_LENGTH);
148
149     CC1101_GDO2_CLEAR_IRQ;
150     CC1101_GDO2_IRQ_ENABLE;
151
152     /*-------------------------------------------------------
153     * Timer konfiguration
154     -------------------------------------------------------*/
155     timer_b_init(); //Timer for waiting
156
157     TIMER_B_RESET;
158     TIMER_B_STOP;
159
160     // Global interrupt enable
161     state = S_START;
162     _EINT();
163
164     uart_init();
165
166     while(1) {
167
```

```
168         //Erneute initialisierung
169         uart_init(); // Prüfen ob nötig
170
171         for(string_index = 0; string_index != 10; string_index++)
172             reciveString[string_index] = 0;
173
174         string_index = 0;
175
176         do {
177             string_index++;
178             while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
179             reciveString[string_index] = U0RXBUF;
180
181         }while(reciveString[string_index]);
182
183
184         uart_menue();
185
186          switch(state) {
187              /*********************************************************************
188              ** Wakeup senden und Konfigurieren
189              *********************************************************************/
190              case S_WAKEUP_ALL: {
191
192                  // Senden des WAKEUP Signal
193                  state = S_WAKEUP_BROADCAST;
194                  cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
195                  tx_wakeup();
196                  delay_ms(1000); // Zeit zum aufwachen
197
198                  state = S_WAKEUP_ADR;
199                  uint8_t packet[14];
200                  for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
201                      // Paket zusammenstellen
202                      packet[0] = sensor_address[current_sensor];
203                      packet[1] = COMMAND_WAKEUP;
204                      packet[2] = 0x00;
205                      packet[3] = 0x00;
206
207                      tx_data_length = 0; // Datenlänge = 0
208
209                      cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
210                      tx_packet(HEADER_LENGTH + tx_data_length, packet);
211                      delay_ms(100);
212                  }
213
214                  state = S_WAKEUP_DONE;
215                  // WAKEUP DONE senden
216                  // Paket zusammenstellen
217                  packet[0] = BROADCAST;
218                  packet[1] = COMMAND_WAKEUP_DONE;
219                  packet[2] = 0x00;
220                  packet[3] = 0x00;
221
222                  tx_data_length = 0; // Datenlänge = 0
223
224                  cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
225                  tx_packet(HEADER_LENGTH + tx_data_length, packet);
226
227                  delay_ms(1000);
228                  state = S_TX_CONFIG_SET;
229
230                  // Paket zusammenstellen
231                  packet[0] = BROADCAST;
232                  packet[1] = COMMAND_DOWNLINK_CONFIG_SET;
233                  packet[2] = 0x0A; // Länge der nächsten Datensendung
234                  packet[3] = 0x00;
235
236                  tx_data_length = 0; // Datenlänge = 0
237
238                  cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
239                  tx_packet(HEADER_LENGTH + tx_data_length, packet);
240
241                  delay_ms(1000);
242                  state = S_TX_CONFIG;
243                  // Paket zusammenstellen
244                  packet[0]   = BROADCAST;
245                  packet[1]   = COMMAND_DOWNLINK_CONFIG;
246                  packet[2]   = 0x00;
247                  packet[3]   = 0x00;
248
249                  packet[4]   = brate;       // Brate
250                  packet[5]   = (uint8_t)((upper_balanc_temp >> 8) & 0x0F);     // obere Bal. Temperatur MSB
251                  packet[6]   = (uint8_t)((upper_balanc_temp >> 0) & 0xFF);     // obere Bal. Temperatur LSB
252                  packet[7]   = (uint8_t)((lower_balanc_temp >> 8) & 0x0F);     // untere Bal. Temperatur MSB
```

```
253                 packet[8]   = (uint8_t)((lower_balanc_temp >> 0) & 0xFF);     // untere Bal. Temperatur LSB
254                 packet[9]   = (uint8_t)((upper_alarm_temp >> 8) & 0x0F);      // obere Alarm Temperatur MSB
255                 packet[10]  = (uint8_t)((upper_alarm_temp >> 0) & 0xFF);      // obere Alarm Temperatur LSB
256                 packet[11]  = (uint8_t)((lower_alarm_temp >> 8) & 0x0F);      // untere Alarm Temperatur MSB
257                 packet[12]  = (uint8_t)((lower_alarm_temp >> 0) & 0xFF);      // untere Alarm Temperatur LSB
258
259                 packet[13]  = 50;       // Burst Framelänge
260
261                 tx_data_length = 10;    // Datenlänge = 10
262
263                 cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
264                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
265
266                 delay_ms(1000); // 1sek warten
267                 state = S_TX_IS_AWAKE;
268
269                 // Überprüfen ob alle Sensoren wach sind
270                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
271                     // Paket zusammenstellen
272                     packet[0] = sensor_address[current_sensor];
273                     packet[1] = COMMAND_DOWNLINK_IS_AWAKE;
274                     packet[2] = 0x01;
275                     packet[3] = 0x00;
276
277                     tx_data_length = 0; // Datenlänge = 0
278
279                     cc1101_set_tx(brate);
280                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
281
282                     error_check_awake = 0;      // Error zurücksetzen
283                     rx_data_length = 0;     // Datenlänge = 0
284                     cc1101_set_rx(brate);   // Auf Empfang einstellen
285                     rx_packet(HEADER_LENGTH + rx_data_length);
286
287                     tx_uart();  //Fehlerausgabe
288                 }
289
290             } break;
291
292             /*********************************************************************
293             ** Wakeup senden und adressiert aufwecken
294             *********************************************************************/
295             case S_WAKEUP: {
296                 LCD_send_cmd(LCD_CLR);
297                 LCD_send_cmd(LCD_LINE1);
298                 LCD_send_text("Sending the     ");
299                 LCD_send_cmd(LCD_LINE2);
300                 LCD_send_text("wakeup now      ");
301
302                 // Senden des WAKEUP Signal
303                 state = S_WAKEUP_BROADCAST;
304                 cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
305                 tx_wakeup();
306                 delay_ms(1000); // Zeit zum aufwachen
307
308                 tx_uart();
309
310                 state = S_WAKEUP_ADR;
311                 uint8_t packet[4];
312                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
313                     // Paket zusammenstellen
314                     packet[0] = sensor_address[current_sensor];
315                     packet[1] = COMMAND_WAKEUP;
316                     packet[2] = 0x00;
317                     packet[3] = 0x00;
318
319                     tx_data_length = 0; // Datenlänge = 0
320
321                     cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
322                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
323                     delay_ms(100);
324                 }
325
326                 state = S_WAKEUP_DONE;
327                 // WAKEUP DONE senden
328                 // Paket zusammenstellen
329                 packet[0] = BROADCAST;
330                 packet[1] = COMMAND_WAKEUP_DONE;
331                 packet[2] = 0x00;
332                 packet[3] = 0x00;
333
334                 tx_data_length = 0; // Datenlänge = 0
335
336                 cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
337                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
```

```
338
339                 LCD_send_cmd(LCD_CLR);
340                 LCD_send_cmd(LCD_LINE1);
341                 LCD_send_text("The wakeup is    ");
342                 LCD_send_cmd(LCD_LINE2);
343                 LCD_send_text("done now         ");
344             } break;
345
346             /*****************************************************************
347             ** Kontrollieren ob die Sensoren aufgewacht sind
348             *****************************************************************/
349             case S_TX_IS_AWAKE: {
350                 LCD_send_cmd(LCD_CLR);
351                 LCD_send_cmd(LCD_LINE1);
352                 LCD_send_text("Checking all");
353                 LCD_send_cmd(LCD_LINE2);
354                 LCD_send_text("sensors");
355
356                 uint8_t packet[4];
357                 //Check all sensors, if there awake 20.03.13 NS
358                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
359
360                     // Paket zusammenstellen
361                     packet[0] = sensor_address[current_sensor];
362                     packet[1] = COMMAND_DOWNLINK_IS_AWAKE;
363                     packet[2] = 0x01;
364                     packet[3] = 0x00;
365
366                     tx_data_length = 0; // Datenlänge = 0
367
368                     cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
369                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
370
371                     error_check_awake = 0;          // Error zurücksetzen
372
373                     rx_data_length = 0;             // Datenlänge = 0
374                     cc1101_set_rx(brate_start);     // Auf Empfang einstellen
375                     rx_packet(HEADER_LENGTH + rx_data_length);
376
377                     tx_uart();  //Fehlerausgabe
378                 }
379
380                 LCD_send_cmd(LCD_CLR);
381                 LCD_send_cmd(LCD_LINE1);
382                 LCD_send_text("Sensors are");
383                 LCD_send_cmd(LCD_LINE2);
384                 LCD_send_text("ready");
385
386             } break;
387
388             /*****************************************************************
389             ** Kommando zur Konfiguration Set
390             **
391             *****************************************************************/
392             case S_TX_CONFIG_SET: {
393                 uint8_t packet[4];
394                 // Paket zusammenstellen
395                 packet[0] = BROADCAST;
396                 packet[1] = COMMAND_DOWNLINK_CONFIG_SET;
397                 packet[2] = 0x0A; // Länge der nächsten Datensendung
398                 packet[3] = 0x00;
399
400                 tx_data_length = 0; // Datenlänge = 0
401
402                 cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
403                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
404
405                 LCD_send_cmd(LCD_CLR);
406                 LCD_send_cmd(LCD_LINE1);
407                 LCD_send_text("Config set       ");
408                 LCD_send_cmd(LCD_LINE2);
409                 LCD_send_text("is done          ");
410             } break;
411
412             /*****************************************************************
413             ** Kommando zur Konfiguration
414             **
415             *****************************************************************/
416             case S_TX_CONFIG: {
417                 uint8_t packet[5];
418                 // Paket zusammenstellen
419                 packet[0]   = BROADCAST;
420                 packet[1]   = COMMAND_DOWNLINK_CONFIG;
421                 packet[2]   = 0x00;
422                 packet[3]   = 0x00;
```

```
423
424                 packet[4]   = brate;       // Brate
425                 packet[5]   = (uint8_t)((upper_balanc_temp >> 8) & 0x0F);     // obere Bal. Temperatur MSB
426                 packet[6]   = (uint8_t)((upper_balanc_temp >> 0) & 0xFF);     // obere Bal. Temperatur LSB
427                 packet[7]   = (uint8_t)((lower_balanc_temp >> 8) & 0x0F);     // untere Bal. Temperatur MSB
428                 packet[8]   = (uint8_t)((lower_balanc_temp >> 0) & 0xFF);     // untere Bal. Temperatur LSB
429                 packet[9]   = (uint8_t)((upper_alarm_temp >> 8) & 0x0F);      // obere Alarm Temperatur MSB
430                 packet[10]  = (uint8_t)((upper_alarm_temp >> 0) & 0xFF);      // obere Alarm Temperatur LSB
431                 packet[11]  = (uint8_t)((lower_alarm_temp >> 8) & 0x0F);      // untere Alarm Temperatur MSB
432                 packet[12]  = (uint8_t)((lower_alarm_temp >> 0) & 0xFF);      // untere Alarm Temperatur LSB
433
434                 packet[13]  = 50;        // Burst Framelänge
435
436                 tx_data_length = 10;     // Datenlänge = 10
437
438                 cc1101_set_tx(brate_start); // CC1101 auf Senden stellen
439                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
440
441                 LCD_send_cmd(LCD_CLR);
442                 LCD_send_cmd(LCD_LINE1);
443                 LCD_send_text("Config is done  ");
444                 LCD_send_cmd(LCD_LINE2);
445                 LCD_send_text("                ");
446             } break;
447
448             /*****************************************************************
449             ** Kalibrierung des TMP102
450             *****************************************************************/
451             case S_TX_CALIBRATION_TMP102: {
452
453                 uint8_t packet[4];
454                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
455                     // Paket zusammenstellen
456                     packet[0] = sensor_address[current_sensor];
457                     packet[1] = COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_TMP102_CALI;
458                     packet[2] = 0x00;
459                     packet[3] = 0x00;
460
461                     tx_data_length = 0; // Datenlänge = 0
462                     cc1101_set_tx(brate);
463                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
464
465                     delay_ms(100);
466                     // Paket zusammenstellen
467                     packet[0] = sensor_address[current_sensor];
468                     packet[1] = COMMAND_DOWNLINK_SEND_TEMPERATUR_TMP102_CALI;
469                     packet[2] = 0x00;
470                     packet[3] = 0x00;
471
472                     tx_data_length = 0; // Datenlänge = 0
473                     cc1101_set_tx(brate);
474                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
475
476                     // Temperatur empfangen
477                     rx_data_length = 2;      // Datenlänge = 2
478                     cc1101_set_rx(brate);    // Auf Empfang einstellen
479                     rx_packet(HEADER_LENGTH + rx_data_length);
480
481                     cali_pro = 0;     // Werte zurücksetzen
482                     cali_neg = 0;     // Werte zurücksetzen
483
484                     // Differenz berechnen
485                     if(sample_buf_temp[current_sensor] < cali_nominal_tmp102) { // Ist Sollwert größer?
486                         cali_pro = cali_nominal_tmp102 - sample_buf_temp[current_sensor];
487                     }else if(sample_buf_temp[current_sensor] > cali_nominal_tmp102) { // Ist Sollwert kleiner?
488                         cali_neg = sample_buf_temp[current_sensor] - cali_nominal_tmp102;
489                     }
490
491                     delay_ms(100);
492                     // Paket zusammenstellen
493                     packet[0] = BROADCAST;
494                     packet[1] = COMMAND_DOWNLINK_CALIBRATION_TMP102;
495                     packet[2] = cali_pro;
496                     packet[3] = cali_neg;
497
498                     tx_data_length = 0; // Datenlänge = 0
499                     cc1101_set_tx(brate);
500                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
501                 }
502
503                 LCD_send_cmd(LCD_CLR);
504                 LCD_send_cmd(LCD_LINE1);
505                 LCD_send_text("Calibration     ");
506                 LCD_send_cmd(LCD_LINE2);
507                 LCD_send_text("TMP102 is done  ");
```

```
508             } break;
509
510             /*****************************************************************
511             ** Kalibrierung des MSP430
512             *****************************************************************/
513             case S_TX_CALIBRATION_MSP430: {
514                 uint8_t packet[4];
515                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
516                     // Paket zusammenstellen
517                     packet[0] = sensor_address[current_sensor];
518                     packet[1] = COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_MSP430_CALI;
519                     packet[2] = 0x00;
520                     packet[3] = 0x00;
521
522                     tx_data_length = 0; // Datenlänge = 0
523                     cc1101_set_tx(brate);
524                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
525
526                     delay_ms(100);
527                     // Paket zusammenstellen
528                     packet[0] = sensor_address[current_sensor];
529                     packet[1] = COMMAND_DOWNLINK_SEND_TEMPERATUR_MSP430_CALI;
530                     packet[2] = 0x00;
531                     packet[3] = 0x00;
532
533                     tx_data_length = 0; // Datenlänge = 0
534                     cc1101_set_tx(brate);
535                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
536
537                     // Temperatur empfangen
538                     rx_data_length = 2;     // Datenlänge = 2
539                     cc1101_set_rx(brate);   // Auf Empfang einstellen
540                     rx_packet(HEADER_LENGTH + rx_data_length);
541
542                     cali_pro = 0;    // Werte zurücksetzen
543                     cali_neg = 0;    // Werte zurücksetzen
544
545                     // Differenz berechnen
546                     if(sample_buf_temp[current_sensor] < cali_nominal_msp430) { // Ist Sollwert größer?
547                         cali_pro = cali_nominal_msp430 - sample_buf_temp[current_sensor];
548                     }else if(sample_buf_temp[current_sensor] > cali_nominal_msp430) { // Ist Sollwert kleiner?
549                         cali_neg = sample_buf_temp[current_sensor] - cali_nominal_msp430;
550                     }
551
552                     delay_ms(100);
553                     // Paket zusammenstellen
554                     packet[0] = BROADCAST;
555                     packet[1] = COMMAND_DOWNLINK_CALIBRATION_MSP430;
556                     packet[2] = cali_pro;
557                     packet[3] = cali_neg;
558
559                     tx_data_length = 0; // Datenlänge = 0
560                     cc1101_set_tx(brate);
561                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
562                 }
563
564                 LCD_send_cmd(LCD_CLR);
565                 LCD_send_cmd(LCD_LINE1);
566                 LCD_send_text("Calibration     ");
567                 LCD_send_cmd(LCD_LINE2);
568                 LCD_send_text("MSP430 is done  ");
569             } break;
570
571
572             /*****************************************************************
573             ** Kalibrierung des MSP430
574             *****************************************************************/
575             case S_TX_CALIBRATION_ADC: {
576                 uint8_t packet[4];
577                 // Paket zusammenstellen
578                 packet[0] = BROADCAST;
579                 packet[1] = COMMAND_DOWNLINK_CALIBRATION_ADC;
580                 packet[2] = cali_pro;
581                 packet[3] = cali_neg;
582
583                 tx_data_length = 0; // Datenlänge = 0
584
585                 cc1101_set_tx(brate);
586                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
587
588                 LCD_send_cmd(LCD_CLR);
589                 LCD_send_cmd(LCD_LINE1);
590                 LCD_send_text("Calibration of  ");
591                 LCD_send_cmd(LCD_LINE2);
592                 LCD_send_text("the ADC is done ");
```

```
593             } break;
594
595             /*****************************************************************
596             ** Kommando zur einfachen Spannungsmessung
597             *****************************************************************/
598             case S_TX_SAMPLE_VOLTAGE: {
599                 uint8_t packet[4];
600                 // Paket zusammenstellen
601                 packet[0] = BROADCAST;
602                 packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE;
603                 packet[2] = 0x00;
604                 packet[3] = 0x00;
605
606                 tx_data_length = 0;  // Datenlänge = 0
607
608                 LED1_ON;
609                 cc1101_set_tx(brate);
610                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
611                 LED1_OFF;
612
613                 LCD_send_cmd(LCD_CLR);
614                 LCD_send_cmd(LCD_LINE1);
615                 LCD_send_text("Sampling Voltage");
616                 LCD_send_cmd(LCD_LINE2);
617                 LCD_send_text("is done          ");
618             } break;
619
620             /*****************************************************************
621             ** Kommando zum senden der einfachen Spnnungsmessung
622             *****************************************************************/
623             case S_TX_SEND_VOLTAGE: {
624
625                 uint8_t packet[4];
626
627                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
628
629                     sample_buf_volt[current_sensor] = 0xFFFF;    //Zurücksetzen
630                     // Paket zusammenstellen
631                     packet[0] = sensor_address[current_sensor];
632                     packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE;
633                     packet[2] = 0x00;
634                     packet[3] = 0x00;
635
636                     tx_data_length = 0;  // Datenlänge = 0
637                     cc1101_set_tx(brate);
638                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
639
640                     rx_data_length = 2;     // Datenlänge = 0
641                     cc1101_set_rx(brate);   // Auf Empfang einstellen
642                     rx_packet(HEADER_LENGTH + rx_data_length);
643                 }
644                 tx_uart();
645
646             } break;
647
648             /*****************************************************************
649             ** Kommando zur einfachen Spannungsmessung und
650             ** Temperaturmessung
651             *****************************************************************/
652             case S_TX_SAMPLE_VOLTAGE_TEMPERATUR_ALL: {
653                 uint8_t packet[4];
654                 // Paket zusammenstellen
655                 packet[0] = BROADCAST;
656                 packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATUR_ALL;
657                 packet[2] = 0x00;
658                 packet[3] = 0x00;
659
660                 tx_data_length = 0;  // Datenlänge = 0
661
662                 cc1101_set_tx(brate);
663                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
664
665             } break;
666
667             /*****************************************************************
668             ** Kommando zum senden der einfachen Spnnungsmessung
669             ** und Temperaturmessung
670             *****************************************************************/
671             case S_TX_SEND_VOLTAGE_TEMPERATUR_ALL: {
672
673                 uint8_t packet[4];
674
675                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
676
677                     sample_buf_volt[current_sensor] = 0xFFFF;    //Zurücksetzen
```

```
678                     sample_buf_temp[current_sensor] = 0xFFFF;    //Zurücksetzen
679                     sample_buf_temp_msp[current_sensor] = 0xFFFF;    //Zurücksetzen
680                     // Paket zusammenstellen
681                     packet[0] = sensor_address[current_sensor];
682                     packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATUR_ALL;
683                     packet[2] = 0x00;
684                     packet[3] = 0x00;
685
686                     tx_data_length = 0;  // Datenlänge = 0
687                     cc1101_set_tx(brate);
688                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
689
690                     rx_data_length = 6;      // Datenlänge = 6
691                     cc1101_set_rx(brate);    // Auf Empfang einstellen
692                     rx_packet(HEADER_LENGTH + rx_data_length);
693                 }
694                 tx_uart();
695             } break;
696
697             /*****************************************************************
698             ** Kommando zur einfachen Spannungsmessung und
699             ** Temperaturmessung
700             *****************************************************************/
701             case S_TX_SAMPLE_VOLTAGE_TEMPERATUR: {
702                 uint8_t packet[4];
703                 // Paket zusammenstellen
704                 packet[0] = BROADCAST;
705                 packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATUR;
706                 packet[2] = 0x00;
707                 packet[3] = 0x00;
708
709                 tx_data_length = 0;  // Datenlänge = 0
710
711                 cc1101_set_tx(brate);
712                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
713
714             } break;
715
716             /*****************************************************************
717             ** Kommando zum senden der einfachen Spnnungsmessung
718             ** und Temperaturmessung
719             *****************************************************************/
720             case S_TX_SEND_VOLTAGE_TEMPERATUR: {
721
722                 uint8_t packet[4];
723
724                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
725
726                     sample_buf_volt[current_sensor] = 0xFFFF;    //Zurücksetzen
727                     sample_buf_temp[current_sensor] = 0xFFFF;    //Zurücksetzen
728                     // Paket zusammenstellen
729                     packet[0] = sensor_address[current_sensor];
730                     packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATUR;
731                     packet[2] = 0x00;
732                     packet[3] = 0x00;
733
734                     tx_data_length = 0;  // Datenlänge = 0
735                     cc1101_set_tx(brate);
736                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
737
738                     rx_data_length = 4;      // Datenlänge = 4
739                     cc1101_set_rx(brate);    // Auf Empfang einstellen
740                     rx_packet(HEADER_LENGTH + rx_data_length);
741                 }
742                 tx_uart();
743             } break;
744
745             /*****************************************************************
746             ** Kommando zur einfachen Temperaturmessung
747             ** mit MSP430
748             *****************************************************************/
749             case S_TX_SAMPLE_TEMPERATUR_TMP102: {
750                 uint8_t packet[4];
751                 // Paket zusammenstellen
752                 packet[0] = BROADCAST;
753                 packet[1] = COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_TMP102;
754                 packet[2] = 0x00;
755                 packet[3] = 0x00;
756
757                 tx_data_length = 0;  // Datenlänge = 0
758
759                 cc1101_set_tx(brate);
760                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
761             } break;
762
```

```
763             /*****************************************************************
764             ** Kommando zum senden der einfachen Temperaturmessung
765             ** mit TMP102
766             *****************************************************************/
767             case S_TX_SEND_TEMPERATUR_TMP102: {
768                 uint8_t packet[4];
769
770                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
771                     // Paket zusammenstellen
772                     packet[0] = sensor_address[current_sensor];
773                     packet[1] = COMMAND_DOWNLINK_SEND_TEMPERATUR_TMP102;
774                     packet[2] = 0x00;
775                     packet[3] = 0x00;
776
777                     tx_data_length = 0;  // Datenlänge = 0
778                     cc1101_set_tx(brate);
779                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
780
781                     rx_data_length = 2;      // Datenlänge = 2
782                     cc1101_set_rx(brate);    // Auf Empfang einstellen
783                     rx_packet(HEADER_LENGTH + rx_data_length);
784                 }
785                 tx_uart();
786             } break;
787
788             /*****************************************************************
789             ** Kommando zur einfachen Temperaturmessung
790             ** mit MSP430
791             *****************************************************************/
792             case S_TX_SAMPLE_TEMPERATUR_MSP430: {
793                 uint8_t packet[4];
794                 // Paket zusammenstellen
795                 packet[0] = BROADCAST;
796                 packet[1] = COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_MSP430;
797                 packet[2] = 0x00;
798                 packet[3] = 0x00;
799
800                 tx_data_length = 0;  // Datenlänge = 0
801
802                 cc1101_set_tx(brate);
803                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
804             } break;
805
806             /*****************************************************************
807             ** Kommando zum senden der einfachen Temperaturmessung
808             ** mit MSP430
809             *****************************************************************/
810             case S_TX_SEND_TEMPERATUR_MSP430: {
811                 uint8_t packet[4];
812
813                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
814                     // Paket zusammenstellen
815                     packet[0] = sensor_address[current_sensor];
816                     packet[1] = COMMAND_DOWNLINK_SEND_TEMPERATUR_MSP430;
817                     packet[2] = 0x00;
818                     packet[3] = 0x00;
819
820                     tx_data_length = 0;  // Datenlänge = 0
821                     cc1101_set_tx(brate);
822                     tx_packet(HEADER_LENGTH + tx_data_length, packet);
823
824                     rx_data_length = 2;          // Datenlänge = 2
825                     cc1101_set_rx(brate);        // Auf Empfang einstellen
826                     rx_packet(HEADER_LENGTH + rx_data_length);
827                 }
828                 tx_uart();
829             } break;
830
831             /*****************************************************************
832             ** Kommando zum anschalten der Balancierung
833             **
834             *****************************************************************/
835             case S_TX_BALANCING_ON: {
836                 uint16_t balancing_volt = 0x0A19;   // Bis zu diesem Wert balancieren
837                 uint8_t packet[4];
838                 uint8_t volt_msb = 0x00;
839                 uint8_t volt_lsb = 0x00;
840
841                 volt_msb = balancing_volt >> 8;
842                 volt_lsb = balancing_volt >> 0;
843                 // Paket zusammenstellen
844                 packet[0] = BROADCAST;
845                 packet[1] = COMMAND_DOWNLINK_BALANCING_ON;
846                 packet[2] = volt_msb;
847                 packet[3] = volt_lsb;
```

```
848
849                 tx_data_length = 0;  // Datenlänge = 0
850
851                 cc1101_set_tx(brate);
852                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
853
854                 delay_ms(500);
855
856                 /*************************************************************
857                 ** NachfolgendeZeilen sind für den Balancierungstest
858                 *************************************************************/
859 //                while(1) {
860 //                    uint8_t packet[4];
861 //                    // Paket zusammenstellen
862 //                    packet[0] = BROADCAST;
863 //                    packet[1] = COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATUR;
864 //                    packet[2] = 0x00;
865 //                    packet[3] = 0x00;
866 //
867 //                    tx_data_length = 0;  // Datenlänge = 0
868 //
869 //                    cc1101_set_tx(brate);
870 //                    tx_packet(HEADER_LENGTH + tx_data_length, packet);
871 //
872 //                    delay_ms(300);
873 //
874 //                    for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
875 //                        // Paket zusammenstellen
876 //                        packet[0] = sensor_address[current_sensor];
877 //                        packet[1] = COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATUR;
878 //                        packet[2] = 0x00;
879 //                        packet[3] = 0x00;
880 //
881 //                        tx_data_length = 0;  // Datenlänge = 0
882 //                        cc1101_set_tx(brate);
883 //                        tx_packet(HEADER_LENGTH + tx_data_length, packet);
884 //
885 //                        rx_data_length = 4;     // Datenlänge = 4
886 //                        cc1101_set_rx(brate);   // Auf Empfang einstellen
887 //                        rx_packet(HEADER_LENGTH + rx_data_length);
888 //                    }
889 //
890 //                    LED1_ON;
891 //                    tx_uart();
892 //                    LED1_OFF;
893 //                    delay_ms(300);
894 //                }
895
896             } break;
897
898             /*************************************************************
899             ** Kommando zum ausschalten der Balancierung
900             **
901             *************************************************************/
902             case S_TX_BALANCING_OFF: {
903                 uint8_t packet[4];
904                 // Paket zusammenstellen
905                 packet[0] = BROADCAST;
906                 packet[1] = COMMAND_DOWNLINK_BALANCING_OFF;
907                 packet[2] = 0x00;
908                 packet[3] = 0x00;
909
910                 tx_data_length = 0;  // Datenlänge = 0
911
912                 cc1101_set_tx(brate);
913                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
914
915                 delay_ms(20);
916                 state = S_TX_SEND_VOLTAGE_TEMPERATUR;
917             } break;
918
919             /*************************************************************
920             ** Kommando zum Burst-Mode
921             **
922             *************************************************************/
923             case S_TX_BURST_MODE: {
924                 LED1_ON;
925                 // Komando zum Burst senden
926                 uint8_t packet[4];
927                 // Paket zusammenstellen
928                 packet[0] = BROADCAST;
929                 packet[1] = COMMAND_DOWNLINK_BURST_MODE;
930                 packet[2] = burst_freq;
931                 packet[3] = BURST_VALUES_700;
932
```

```
933                 tx_data_length = 0;  // Datenlänge = 0
934
935                 cc1101_set_tx(brate);
936                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
937
938                 // Bursttakt starten
939                 CC1101_GDO2_IRQ_DISABLE;
940                 CC1101_GDO2_CLEAR_IRQ;
941
942                 TXPIN_DIR_IN;                   // TX pin is input until CC1101 is in TX state
943                 cc1101_reset();                 // Reset all registers to their default values and go to idle state
944                 cc1101_config_no_packet();      // Configure the transceiver for sending the wakeup signal
945                 cc1101_tx_asynchronous_mode();  // Change to TX state
946                 TIMER_A_STOP;                   // Make sure timer for TX pin toggling is not running
947                 timer_a_init_burst(burst_freq);
948                 TXPIN_DIR_OUT;                  // TX pin ist Ausgang
949                 P1OUT |= BIT2;
950
951
952                 // delay_ms(28 - 3);               // Wartezeit@1MHz
953                 delay_ms(15 - 3);               // Wartezeit@16MHz des ZS
954
955                 LED1_OFF;           // Gibt Triggersignal aus
956
957                 TXPIN_MAN;
958                 TIMER_A_CM;         // Start the timer that is toggling the TX pin
959
960                 while(asynch_counter < 701);
961
962                 asynch_counter = 0; // Zurücksetzen
963                 P1OUT |= BIT2;
964                 TXPIN_MAN;          // TX auf manuell stellen
965                 TIMER_A_STOP;       // Timer stoppen
966                 delay_ms(10);       // Wartezeit
967
968                 TXPIN_MAN;
969                 TXPIN_DIR_IN;
970
971                 delay_ms(500);      // Wateteit
972                 // Paket zusammenstellen
973                 // Leersendung -> nötig für Umstellung
974                 packet[0] = BROADCAST;
975                 packet[1] = COMMAND_DOWNLINK_BURST_CHECK;
976                 packet[2] = 0x00;
977                 packet[3] = 0x00;
978
979                 tx_data_length = 0;  // Datenlänge = 0
980
981                 cc1101_set_tx(brate);
982                 tx_packet(HEADER_LENGTH + tx_data_length, packet);
983
984             } break;
985
986             /*********************************************************************
987             ** Kommando zur Anfrage der BURST DATEN
988             ** Antwort von ZS:
989             ** Anzahl Frames und Framelänge ohne Header
990             *********************************************************************/
991             case S_BURST_DATA_RQ: {
992                 uint8_t packet[4];
993
994                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
995                     // Paket zusammenstellen
996                     packet[0] = sensor_address[current_sensor];
997                     packet[1] = COMMAND_DOWNLINK_BURST_DATA_RQ;
998                     packet[2] = 0x00;
999                     packet[3] = 0x00;
1000
1001                    tx_data_length = 0;  // Datenlänge = 0
1002
1003                    cc1101_set_tx(brate);
1004                    tx_packet(HEADER_LENGTH + tx_data_length, packet);
1005
1006                    // Antwort abwarten
1007                    rx_data_length = 0;     // Datenlänge = 0
1008                    cc1101_set_rx(brate);   // Auf Empfang einstellen
1009                    rx_packet(HEADER_LENGTH + rx_data_length);
1010                    delay_ms(100);
1011                }
1012            } break;
1013
1014            /*********************************************************************
1015            ** Burstdaten empfangen
1016            *********************************************************************/
1017            case S_BURST_DATA_RX: {
```

```
1018
1019                    //current_sensor = 0;
1020
1021                    uint8_t frame_number = 0;
1022                    uint8_t packet[4];
1023
1024                    for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
1025                        for(frame_number = 0;frame_number<frame_counter;frame_number++) {
1026                            // Paket zusammenstellen
1027                            packet[0] = BROADCAST;
1028                            packet[1] = COMMAND_DOWNLINK_BURST_DATA_RX;
1029                            packet[2] = frame_number;
1030                            packet[3] = 0x00;
1031
1032                            tx_data_length = 0;  // Datenlänge = 0
1033
1034                            cc1101_set_tx(brate);
1035                            tx_packet(HEADER_LENGTH + tx_data_length, packet);
1036
1037                            cc1101_set_rx(brate);    // Auf Empfang einstellen
1038                            rx_packet(HEADER_LENGTH + rx_data_length);  // Paket empfangen
1039
1040                            tx_uart();       // Daten an UART senden
1041                            delay_ms(100);
1042                        }
1043                        rx_data_length = 0; // Datenlänge = 0
1044                    }
1045                } break;
1046
1047                default: break;
1048            }
1049        }
1050        return 0;
1051    }
1052
1053    /*****************************************************************************************
1054    ** Auf Empfang einstellen
1055    *****************************************************************************************/
1056    void rx(uint8_t packetes) {
1057        CC1101_GDO2_CLEAR_IRQ;       // Clear the sync word detected interrupt
1058        CC1101_GDO2_IRQ_ENABLE;      // Enable the sync word detected interrupt
1059        IRQ_SET_RX;
1060        cc1101_rx(packetes);         // Transceiver in RX state
1061    }
1062
1063    /*****************************************************************************************
1064    ** Pakete empfangen
1065    *****************************************************************************************/
1066    void rx_packet(uint8_t expect_packetes) {
1067
1068        rx(expect_packetes);
1069
1070        waiting_time = 0;
1071        TIMER_B_RESET;  // Timer zurücksetzen
1072        TIMER_B_START;  // Timer starten
1073        while((!irq_data_received_flag) && (waiting_time <= MAX_RECIVE_TIME));// Warten bis daten empfangen
1074        CC1101_GDO2_IRQ_DISABLE;                        //Set IRQ off
1075        CC1101_GDO2_CLEAR_IRQ;                          //Set IRQ off
1076        TIMER_B_STOP;   // Timer stoppen
1077
1078        if(irq_data_received_flag) {
1079            // Answer from the correct sensor?
1080            if (rx_data[1] == sensor_address[current_sensor]) {
1081                /*************************************************************************
1082                * Hier werden die Empfangenen Pakete dekodiert
1083                *************************************************************************/
1084                switch(state) {
1085                    /*************************************************************************
1086                    ** Dekodierung der IS AWAKE Anfrage
1087                    ** Paketzusammenstellung:
1088                    ** (Adresse Basisstation | Adresse Sensor | 1 | 0x00)
1089                    *************************************************************************/
1090                    case S_TX_IS_AWAKE: {
1091                        error_check_awake = rx_data[2];
1092                    }break;
1093
1094                    /*************************************************************************
1095                    ** Dekodierung der einfachen Spannungsübertragung
1096                    ** Paketzusammenstellung:
1097                    ** (Adr. Basisstation | Adr. Sensor | Befehl | 0x00 | Volt_MSB | Volt_LSB)
1098                    *************************************************************************/
1099                    case S_TX_SEND_VOLTAGE: {
1100                        uint16_t volt_msb = 0;
1101                        uint16_t volt_lsb = 0;
1102
```

```
1103                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1104                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1105
1106                    sample_buf_volt[current_sensor] = volt_msb | volt_lsb;
1107
1108                    delay_ms(5);
1109                } break;
1110
1111                /*****************************************************************************
1112                ** Dekodierung der einfachen Spannungs- und Temeraturmessung
1113                ** Paketzusammenstellung:
1114                ** (Adr. BS | Adr. ZS | Befehl | 0x00 | Volt_MSB | Volt_LSB | Temp_MSB | Temp_LSB)
1115                *****************************************************************************/
1116                case S_TX_SEND_VOLTAGE_TEMPERATUR: {
1117                    uint16_t volt_msb = 0;
1118                    uint16_t volt_lsb = 0;
1119
1120                    uint16_t temp_msb = 0;
1121                    uint16_t temp_lsb = 0;
1122
1123                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1124                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1125
1126                    temp_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 2] & 0x0F)) << 8) & 0x0F00;
1127                    temp_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 3] & 0xFF)) << 0) & 0x00FF;
1128
1129                    sample_buf_volt[current_sensor] = volt_msb | volt_lsb;
1130                    sample_buf_temp[current_sensor] = temp_msb | temp_lsb;
1131
1132                    delay_ms(5);
1133                } break;
1134
1135
1136                /*****************************************************************************
1137                ** Dekodierung der einfachen Spannungs- und Temeraturmessung
1138                ** Paketzusammenstellung:
1139                ** (Adr. BS | Adr. ZS | Befehl | 0x00 | Volt_MSB | Volt_LSB | Temp_MSB_TMP | Temp_LSB_TMP | Temp_MSB_MSP | Temp_LSB_MSP
                       )
1140                *****************************************************************************/
1141                case S_TX_SEND_VOLTAGE_TEMPERATUR_ALL: {
1142                    uint16_t volt_msb = 0;
1143                    uint16_t volt_lsb = 0;
1144
1145                    uint16_t temp_msb_tmp = 0;
1146                    uint16_t temp_lsb_tmp = 0;
1147
1148                    uint16_t temp_msb_msp = 0;
1149                    uint16_t temp_lsb_msp = 0;
1150
1151                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1152                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1153
1154                    temp_msb_tmp = (((uint16_t)(rx_data[HEADER_LENGTH + 2] & 0x0F)) << 8) & 0x0F00;
1155                    temp_lsb_tmp = (((uint16_t)(rx_data[HEADER_LENGTH + 3] & 0xFF)) << 0) & 0x00FF;
1156
1157                    temp_msb_msp = (((uint16_t)(rx_data[HEADER_LENGTH + 4] & 0x0F)) << 8) & 0x0F00;
1158                    temp_lsb_msp = (((uint16_t)(rx_data[HEADER_LENGTH + 5] & 0xFF)) << 0) & 0x00FF;
1159
1160                    sample_buf_volt[current_sensor] = volt_msb | volt_lsb;
1161                    sample_buf_temp[current_sensor] = temp_msb_tmp | temp_lsb_tmp;
1162                    sample_buf_temp_msp[current_sensor] = temp_msb_msp | temp_lsb_msp;
1163
1164                    delay_ms(5);
1165                } break;
1166
1167                /*****************************************************************************
1168                ** Dekodierung der einfachen Temeraturmessung
1169                ** Paketzusammenstellung:
1170                ** (Adr. BS | Adr. ZS | Befehl | 0x00 | Temp_MSB | Temp_LSB)
1171                *****************************************************************************/
1172                case S_TX_SEND_TEMPERATUR_TMP102: {
1173                    uint16_t temp_msb = 0;
1174                    uint16_t temp_lsb = 0;
1175
1176                    temp_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1177                    temp_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1178
1179                    sample_buf_temp[current_sensor] = temp_msb | temp_lsb;
1180
1181                    delay_ms(5);
1182                } break;
1183
1184                /*****************************************************************************
1185                ** Balancierung einschalten
1186                *****************************************************************************/
```

```
1187                case S_TX_BALANCING_ON: {
1188                    uint16_t volt_msb = 0;
1189                    uint16_t volt_lsb = 0;
1190
1191                    uint16_t temp_msb = 0;
1192                    uint16_t temp_lsb = 0;
1193
1194                    volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1195                    volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1196
1197                    temp_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 2] & 0x0F)) << 8) & 0x0F00;
1198                    temp_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 3] & 0xFF)) << 0) & 0x00FF;
1199
1200                    sample_buf_volt[current_sensor] = volt_msb | volt_lsb;
1201                    sample_buf_temp[current_sensor] = temp_msb | temp_lsb;
1202                } break;
1203
1204                /*****************************************************************************
1205                ** Dekodierung der einfachen Temeraturmessung ohne Offset
1206                ** Paketzusammenstellung:
1207                ** (Adr. BS | Adr. ZS | Befehl | 0x00 | Temp_MSB | Temp_LSB)
1208                *****************************************************************************/
1209                case S_TX_CALIBRATION_TMP102: {
1210                    uint16_t temp_msb = 0;
1211                    uint16_t temp_lsb = 0;
1212
1213                    temp_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1214                    temp_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1215
1216                    sample_buf_temp[current_sensor] = temp_msb | temp_lsb;
1217
1218                    delay_ms(5);
1219                } break;
1220
1221                /*****************************************************************************
1222                ** Dekodierung der einfachen Spannungs– und Temeraturmessung
1223                ** Paketzusammenstellung:
1224                ** (Adr. BS | Adr. ZS | Befehl | 0x00 | Temp_MSB | Temp_LSB)
1225                *****************************************************************************/
1226                case S_TX_SEND_TEMPERATUR_MSP430: {
1227                    uint16_t temp_msb = 0;
1228                    uint16_t temp_lsb = 0;
1229
1230                    temp_msb = (((uint16_t)(rx_data[HEADER_LENGTH + 0] & 0x0F)) << 8) & 0x0F00;
1231                    temp_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + 1] & 0xFF)) << 0) & 0x00FF;
1232
1233                    sample_buf_temp_msp[current_sensor] = temp_msb | temp_lsb;
1234
1235                    delay_ms(5);
1236                } break;
1237
1238                /*****************************************************************************
1239                ** Dekodierung der BURST DATA Anfrage
1240                ** Paketzusammenstellung:
1241                ** (Adresse Basisstation | Adresse Sensor | Anzahl Frames | Framelänge ohne Header)
1242                *****************************************************************************/
1243                case S_BURST_DATA_RQ: {
1244                    frame_counter   = rx_data[2];
1245                    rx_data_length  = rx_data[3];
1246                    seq_number = 0;             // Sequenznummer zurücksetzen
1247
1248                } break;
1249
1250                /*****************************************************************************
1251                ** Dekodierung der BURST DATEN
1252                ** Paketzusammenstellung:
1253                ** (Adresse Basisstation | Adresse Sensor | Seq. Nummer | )
1254                *****************************************************************************/
1255                case S_BURST_DATA_RX: {
1256
1257                    uint16_t volt_msb = 0;
1258                    uint16_t volt_lsb = 0;
1259
1260                    uint8_t index = 0;
1261
1262                        for(seq_number = 0; seq_number <= rx_data_length; seq_number++) {
1263                            volt_msb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0x0F)) << 8) & 0x0F00;
1264                            seq_number++;
1265                            volt_lsb = (((uint16_t)(rx_data[HEADER_LENGTH + seq_number] & 0xFF)) << 0) & 0x00FF;
1266
1267                            burst_data[index] = volt_msb | volt_lsb;
1268                            index++;
1269                    }
1270                } break;
1271
```

```
1272                 default: break;
1273             }
1274         }
1275     }
1276 }
1277 /*****************************************************************************************
1278 ** Paket Senden
1279 *****************************************************************************************/
1280 void tx_packet(uint8_t packetes, uint8_t *txbuf) {
1281     TXPIN_MAN;                  // TX pin controlled manually
1282     TXPIN_DIR_IN;               // TX pin is input
1283 
1284     CC1101_GDO2_CLEAR_IRQ;      // Clear the sync word detected interrupt
1285     CC1101_GDO2_IRQ_ENABLE;     // Enable the sync word detected interrupt
1286     IRQ_SET_TX;
1287 
1288     cc1101_fill_tx_fifo(txbuf, packetes);   // Daten an FIFO senden
1289     cc1101_tx(packetes);                    // Daten senden
1290 
1291     waiting_time = 0;                       // Wartezeit zurücksetzen
1292     TIMER_B_RESET;                          // Timer zurücksetzen
1293     TIMER_B_START;                          // Timer starten
1294     while((!irq_send_done_flag) && (waiting_time <= MAX_RECIVE_TIME));// Warten bis daten empfangen
1295     TIMER_B_STOP;                           // Timer stoppen
1296 
1297     tx_data_length = 0;                     // Datenlänge = 0
1298 
1299     if(!irq_send_done_flag)                 // Wartezeit ausgelöst
1300         error_tx = ERROR_TX_WAITING_TO_LONG;
1301     else
1302         irq_send_done_flag = 0;             // Kein fehler
1303 
1304     cc1101_idle();
1305 }
```

F.2.11. main.h

```
1  /*****************************************************************************
2  **     Name:           main.h
3  **     Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
4  **     Date:           07/03/2013
5  **     Author:         Nico Sassano
6  **                     modified taken from Phillip Durdaut
7  *****************************************************************************/
8  #ifndef MAIN_H_
9  #define MAIN_H_
10
11 #include <msp430x16x.h>
12 #include <signal.h>
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 #include "delay.h"
17 #include "cc1101.h"
18 #include "uart.h"
19 #include "lcd16x2.h"
20 #include "timer.h"
21 #include "uart_menue.h"
22 #include "wakeup.h"
23 #include "clk.h"
24
25 /*************************************************************
26 **  Defines
27 *************************************************************/
28 #define NUMBER_OF_SENSORS                                1
29
30 #define WAKEUP_DURATION_MS                               1000
31
32 #define HEADER_LENGTH                                    4
33
34 // Address definitions
35 #define BROADCAST                                        0x00
36 #define ADDRESS_BASE_STATION                             0xFF
37 #define ADDRESS_SENSOR_1                                 0x01
38 #define ADDRESS_SENSOR_2                                 0x02
39 #define ADDRESS_SENSOR_3                                 0x03
40 #define ADDRESS_SENSOR_4                                 0x04
41
42 /*** Downlink commands ***************************************/
43 #define COMMAND_WAKEUP                                   0x00
44 #define COMMAND_WAKEUP_DONE                              0x01
45 #define COMMAND_DOWNLINK_IS_AWAKE                        0x02
46 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE                  0x03
47 #define COMMAND_DOWNLINK_SEND_VOLTAGE                    0x04
48 #define COMMAND_DOWNLINK_SLEEP                           0x05
49 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATUR               0x06
50 #define COMMAND_DOWNLINK_SEND_TEMPERATUR                 0x07
51 #define COMMAND_DOWNLINK_BALANCING_ON                    0x08
52 #define COMMAND_DOWNLINK_BALANCING_OFF                   0x09
53 #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATUR         0x0A
54 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATUR       0x0B
55 #define COMMAND_DOWNLINK_BURST_MODE                      0x0C
56 #define COMMAND_DOWNLINK_BURST_DATA_RQ                   0x0D
57 #define COMMAND_DOWNLINK_BURST_DATA_RX                   0x0E
58 #define COMMAND_DOWNLINK_BURST_CHECK                     0x0F
59 #define COMMAND_DOWNLINK_CONFIG_SET                      0x10
60 #define COMMAND_DOWNLINK_CONFIG                          0x11
61 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_TMP102        0x12
62 #define COMMAND_DOWNLINK_SEND_TEMPERATUR_TMP102          0x13
63 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_TMP102_CALI   0x14
64 #define COMMAND_DOWNLINK_SEND_TEMPERATUR_TMP102_CALI     0x15
65 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_MSP430        0x16
66 #define COMMAND_DOWNLINK_SEND_TEMPERATUR_MSP430          0x17
67 #define COMMAND_DOWNLINK_SAMPLE_TEMPERATUR_MSP430_CALI   0x18
68 #define COMMAND_DOWNLINK_SEND_TEMPERATUR_MSP430_CALI     0x19
69 #define COMMAND_DOWNLINK_CALIBRATION_TMP102              0x1A
70 #define COMMAND_DOWNLINK_CALIBRATION_MSP430              0x1B
71 #define COMMAND_DOWNLINK_CALIBRATION_ADC                 0x1C
72 #define COMMAND_DOWNLINK_SEND_VOLTAGE_TEMPERATUR_ALL     0x1D
73 #define COMMAND_DOWNLINK_SAMPLE_VOLTAGE_TEMPERATUR_ALL   0x1E
74 /*** TEST DOWNLINKS ******************************************/
75 #define COMMAND_DOWNLINK_ERROR_TEST                      0x99
76
77 #define COMMAND_DOWNLINK_UNKWOWN                         0xFF
78
79 /*** Error definition ****************************************/
80 #define NO_ERROR                                         0x00
81 #define ERROR_1                                          0x01
82 #define ERROR_TX_WAITING_TO_LONG                         0x02
83
```

```
84  #define ERROR_NO_ANS_SENS_1                         0x01
85  #define ERROR_NO_ANS_SENS_2                         0x02
86  #define ERROR_NO_ANS_SENS_3                         0x03
87  #define ERROR_NO_ANS_SENS_4                         0x04
88
89  #define ERROR_RX_DATA_NO_VALID                      0x07
90  #define ERROR_RX_WRONG_ADRESS                       0x08
91
92  // Interrupt Modes
93  #define TX_MODE                                     0x00
94  #define RX_MODE                                     0x01
95
96  #define MAX_RECIVE_TIME                             0xF4
97
98  #define CONTROL_FRAME                               0x00
99  #define DATA_FRAME                                  0x01
100
101 /*************************************************************
102 ** Types
103 *************************************************************/
104
105 typedef enum { S_WAIT,
106                S_START,
107                S_RESTART,
108                S_INIT,
109               S_WAKEUP,
110               S_WAKEUP_ALL,
111               S_WAKEUP_BROADCAST,
112               S_WAKEUP_ADR,
113               S_WAKEUP_DONE,
114               S_TX_IS_AWAKE,
115               S_RX_IS_AWAKE,
116               S_TX_SAMPLE_TEMPERATUR_TMP102,
117               S_TX_SEND_TEMPERATUR_TMP102,
118               S_TX_SAMPLE_TEMPERATUR_MSP430,
119               S_TX_SEND_TEMPERATUR_MSP430,
120               S_TX_SAMPLE_VOLTAGE_TEMPERATUR_ALL,
121               S_TX_SEND_VOLTAGE_TEMPERATUR_ALL,
122               S_TX_SAMPLE_VOLTAGE_TEMPERATUR,
123               S_TX_SEND_VOLTAGE_TEMPERATUR,
124               S_TX_SAMPLE_VOLTAGE,
125               S_TX_SEND_VOLTAGE,
126               S_BROADCAST_SLEEP,
127               S_TX_BALANCING_ON,
128               S_TX_BALANCING_OFF,
129               S_TX_TEST_UEBERTRAGUNG,
130               S_TX_BURST_MODE,
131               S_BURST_DATA_RQ,
132               S_BURST_DATA_RX,
133               S_TX_CONFIG_SET,
134               S_TX_CONFIG,
135               S_TX_CALIBRATION_TMP102,
136               S_TX_CALIBRATION_MSP430,
137               S_TX_CALIBRATION_ADC,
138             } state_t ;
139
140 /*** LEDs ************************************************/
141 #define LEDx_DIR_OUT              (P3DIR |= (BIT7 | BIT6))
142 #define LED1_DIR_OUT              (P3DIR |= BIT6)
143 #define LED1_ON                   (P3OUT &= ~BIT6)
144 #define LED1_OFF                  (P3OUT |= BIT6)
145 #define LED1_TOGGLE               (P3OUT ^= BIT6)
146 #define LED2_DIR_OUT              (P3DIR |= BIT7)
147 #define LED2_ON                   (P3OUT &= ~BIT7)
148 #define LED2_OFF                  (P3OUT |= BIT7)
149 #define LED2_TOGGLE               (P3OUT ^= BIT7)
150
151 /** LCD ***************************************************/
152 #define LCD_DIR_OUT               (P4DIR = 0xFF)
153 #define LCD_OUT_0x00              (P4OUT = 0x00)
154
155 /*** Flags ***********************************************/
156 #define BALANCE_ON_FLAG_ON          (irq_balanc_on_flag = 1)
157 #define BALANCE_ON_FLAG_OFF         (irq_balanc_on_flag = 0)
158 #define BALANCE_ON_FLAG_IS_SET      (irq_balanc_on_flag == 1)
159
160 #define BALANCE_OFF_FLAG_ON         (irq_balanc_off_flag = 1)
161 #define BALANCE_OFF_FLAG_OFF        (irq_balanc_off_flag = 0)
162 #define BALANCE_OFF_FLAG_IS_SET     (irq_balanc_off_flag == 1)
163
164 #define TIMER_A_WAKEUP_FLAG_ON      (irq_timer_a_wakeup_flag = 1)
165 #define TIMER_A_WAKEUP_FLAG_OFF     (irq_timer_a_wakeup_flag = 0)
166 #define TIMER_A_WAKEUP_FLAG_IS_SET  (irq_timer_a_wakeup_flag == 1)
167
168 #define TIMER_A_TEST_FLAG_ON      (irq_timer_a_test_flag = 1)
```

```
169 #define TIMER_A_TEST_FLAG_OFF     (irq_timer_a_test_flag = 0)
170 #define TIMER_A_TEST_FLAG_IS_SET  (irq_timer_a_test_flag == 1)
171
172 /*** CC1101 asynchronous TX ********************************/
173 #define TXPIN_TIMER               (P1SEL |= BIT2)
174 #define TXPIN_MAN                 (P1SEL &= ~BIT2)
175 #define TXPIN_DIR_IN              (P1DIR &= ~BIT2)
176 #define TXPIN_DIR_OUT             (P1DIR |= BIT2)
177 #define TXPIN_LOW                 (P1OUT &= ~BIT2)
178 #define TXPIN_HIGH                (P1OUT |= BIT2)
179
180 // Clock source is SMCLK / 1
181 // PWM period 8 us
182 #define TIMERA_CONFIG_125         (TACTL |= TASSEL1); \
183                                   (TACCR0 = 64 - 1); \
184                                   (TACCR1 = 32 - 1); \
185                                   (TACCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
186 #define TIMERA_UP                 (TACTL |= MC0)
187
188 /*** Defines IRQ ********************************************/
189 #define IRQ_SET_RX                (irq_mode = RX_MODE)
190 #define IRQ_IS_RX                 (irq_mode == RX_MODE)
191 #define IRQ_SET_TX                (irq_mode = TX_MODE)
192 #define IRQ_IS_TX                 (irq_mode == TX_MODE)
193
194 /***********************************************
195 ** Frequenzen für die Burstmessung
196 ***********************************************/
197 #define BURST_FREQ_7500HZ           0x1A
198 #define BURST_FREQ_10000HZ          0x19
199 #define BURST_FREQ_8000HZ           0x18
200 #define BURST_FREQ_6000HZ           0x17
201 #define BURST_FREQ_4000HZ           0x15
202 #define BURST_FREQ_2000HZ           0x16
203 #define BURST_FREQ_1000HZ           0x01
204 #define BURST_FREQ_950HZ            0x02
205 #define BURST_FREQ_900HZ            0x03
206 #define BURST_FREQ_850HZ            0x04
207 #define BURST_FREQ_800HZ            0x05
208 #define BURST_FREQ_750HZ            0x06
209 #define BURST_FREQ_700HZ            0x07
210 #define BURST_FREQ_650HZ            0x08
211 #define BURST_FREQ_600HZ            0x09
212 #define BURST_FREQ_550HZ            0x0A
213 #define BURST_FREQ_500HZ            0x0B
214 #define BURST_FREQ_450HZ            0x0C
215 #define BURST_FREQ_400HZ            0x0D
216 #define BURST_FREQ_350HZ            0x0E
217 #define BURST_FREQ_300HZ            0x0F
218 #define BURST_FREQ_250HZ            0x10
219 #define BURST_FREQ_200HZ            0x11
220 #define BURST_FREQ_150HZ            0x12
221 #define BURST_FREQ_100HZ            0x13
222 #define BURST_FREQ_50HZ             0x14
223
224 #define BURST_VALUES_50             0x01
225 #define BURST_VALUES_100            0x02
226 #define BURST_VALUES_150            0x03
227 #define BURST_VALUES_200            0x04
228 #define BURST_VALUES_250            0x05
229 #define BURST_VALUES_300            0x06
230 #define BURST_VALUES_350            0x07
231 #define BURST_VALUES_400            0x08
232 #define BURST_VALUES_450            0x09
233 #define BURST_VALUES_500            0x0A
234 #define BURST_VALUES_550            0x0B
235 #define BURST_VALUES_600            0x0C
236 #define BURST_VALUES_650            0x0D
237 #define BURST_VALUES_700            0x0E
238 #define BURST_VALUES_750            0x0F
239 #define BURST_VALUES_800            0x10
240 #define BURST_VALUES_850            0x11
241 #define BURST_VALUES_900            0x12
242
243 #endif /* MAIN_H_ */
```

F.2.12. timer.c

```
1 /*******************************************************************************
2 **    Name:           timer.c
3 **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4 **    Date:           13/03/2013
```

```
5   **    Author:         Nico Sassano
6   ********************************************************************************/
7   #include "main.h"
8
9   extern volatile uint8_t irq_timer_a_wakeup_flag;
10  extern volatile uint8_t irq_timer_a_test_flag;
11
12  /************************************************************
13  ** Timer A für das 125kHz WakeUp Signal
14  ************************************************************/
15  void timer_a_wakeup_init(void) {
16
17      TIMER_A_RESET;
18      TACTL |= TASSEL1;
19      TIMER_A_SOURCE_DIV_1;
20      TACCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0);
21
22      /***********************************
23      ** TACCRx ist 16bit Register
24      ***********************************/
25      TACCR0 = 62 - 1;
26      TACCR1 = 32 - 1;
27
28      TIMER_A_WAKEUP_FLAG_ON;
29  }
30
31  /************************************************************
32  ** Timer A für Burst Frequenz
33  ************************************************************/
34  void timer_a_init_burst(uint8_t burst_freq) {
35      TIMER_A_RESET;
36      TIMER_A_SOURCE_SMCLK;
37      TIMER_A_SOURCE_DIV_8;
38
39      /***********************************
40      ** TACCRx ist 16bit Register
41      ** CLK --> 8MHz
42      ***********************************/
43      switch(burst_freq) {
44
45          case BURST_FREQ_10000HZ: {
46              TACCR0 = 100 - 7;
47              TACCR1 = 50 - 7;
48          }break;
49
50          case BURST_FREQ_8000HZ: {
51              TACCR0 = 125 - 8;
52              TACCR1 = 50 - 8;
53          }break;
54
55          case BURST_FREQ_7500HZ: {
56              TACCR0 = 133 - 8;
57              TACCR1 = 60 - 8;
58          }break;
59
60          case BURST_FREQ_6000HZ: {
61              TACCR0 = 166 - 8;
62              TACCR1 = 66 - 8;
63          }break;
64
65          case BURST_FREQ_4000HZ: {
66              TACCR0 = 250 - 8;
67              TACCR1 = 100 - 8;
68          }break;
69
70          case BURST_FREQ_2000HZ: {
71              TACCR0 = 500 - 1;
72              TACCR1 = 100 - 1;
73          }break;
74
75          case BURST_FREQ_1000HZ: {
76              TACCR0 = 1000 - 1;
77              TACCR1 = 400 - 1;
78          }break;
79
80          case BURST_FREQ_950HZ: {
81              TACCR0 = 1052;
82              TACCR1 = TACCR0 - 500;
83          }break;
84
85          case BURST_FREQ_900HZ: {
86              TACCR0 = 1111;
87              TACCR1 = TACCR0 - 500;
88          }break;
89
```

```
90              case BURST_FREQ_850HZ: {
91                  TACCR0 = 1176;
92                  TACCR1 = TACCR0 - 500;
93              }break;
94
95              case BURST_FREQ_800HZ: {
96                  TACCR0 = 1250;
97                  TACCR1 = TACCR0 - 500;
98              }break;
99
100             case BURST_FREQ_750HZ: {
101                 TACCR0 = 1333;
102                 TACCR1 = TACCR0 - 500;
103             }break;
104
105             case BURST_FREQ_700HZ: {
106                 TACCR0 = 1428;
107                 TACCR1 = TACCR0 - 500;
108             }break;
109
110             case BURST_FREQ_650HZ: {
111                 TACCR0 = 1538;
112                 TACCR1 = TACCR0 - 500;
113             }break;
114
115             case BURST_FREQ_600HZ: {
116                 TACCR0 = 1666;
117                 TACCR1 = TACCR0 - 500;
118             }break;
119
120             case BURST_FREQ_550HZ: {
121                 TACCR0 = 1818;
122                 TACCR1 = TACCR0 - 500;
123             }break;
124
125             case BURST_FREQ_500HZ: {
126                 TACCR0 = 2000;
127                 TACCR1 = TACCR0 - 500;
128             }break;
129
130             case BURST_FREQ_450HZ: {
131                 TACCR0 = 2222;
132                 TACCR1 = TACCR0 - 500;
133             }break;
134
135             case BURST_FREQ_400HZ: {
136                 TACCR0 = 2500;
137                 TACCR1 = TACCR0 - 500;
138             }break;
139
140             case BURST_FREQ_350HZ: {
141                 TACCR0 = 2857;
142                 TACCR1 = TACCR0 - 500;
143             }break;
144
145             case BURST_FREQ_300HZ: {
146                 TACCR0 = 3333;
147                 TACCR1 = TACCR0 - 500;
148             }break;
149
150             case BURST_FREQ_250HZ: {
151                 TACCR0 = 4000;
152                 TACCR1 = TACCR0 - 500;
153             }break;
154
155             case BURST_FREQ_200HZ: {
156                 TACCR0 = 5000;
157                 TACCR1 = TACCR0 - 500;
158             }break;
159
160             case BURST_FREQ_150HZ: {
161                 TACCR0 = 6666;
162                 TACCR1 = TACCR0 - 500;
163             }break;
164
165             case BURST_FREQ_100HZ: {
166                 TACCR0 = 10000;
167                 TACCR1 = TACCR0 - 500;
168             }break;
169
170             case BURST_FREQ_50HZ: {
171                 TACCR0 = 20000;
172                 TACCR1 = TACCR0 - 500;
173             }break;
174         }
```

```
175
176         //TIMER_A_OUTMOD_RESET_SET;
177         TIMER_A_TEST_FLAG_ON;
178
179         TIMER_A_0_CM_IRQ_ENABLE;
180         TIMER_A_1_CM_IRQ_ENABLE;
181     }
182
183
184     void timer_b_init(void){
185         /*************************************************************
186         * Timer B initialisierung 13.03.13 NS
187         *************************************************************/
188
189         // timer clear
190         TBCTL = TACLR;
191         // no grouping, counter length = 16bit, clock source = SMCLK, div = 8
192         TBCTL |= (TASSEL1 | ID1 | ID0);
193         // Capture/Compare Reg 0 set
194         TBCCR0 = 1000;
195         // Compare-mode, IRQ enable
196         TBCCTL0 |= CCIE;
197         // Capture/Compare Reg 1 set
198         //TBCCR1 = TIME_TO_SAMPLE;
199         // Compare-mode, IRQ enable
200         //TBCCTL1 |= CCIE;
201         // Capture/Compare Reg 2 set
202         //TBCCR2 = TIME_TO_TX;
203         // Compare-mode, IRQ enable
204         //TBCCTL2 |= CCIE;
205     }
```

F.2.13. timer.h

```
1  /*****************************************************************************
2  **    Name:           timer.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **    Date:           13/03/2013
5  **    Author:         Nico Sassano
6  *****************************************************************************/
7
8  #ifndef TIMER_H_
9  #define TIMER_H_
10
11 #include "main.h"
12
13 /************************************************************
14 **  Defines Timer A
15 ************************************************************/
16 #define TIMER_A_UP                  (TACTL |= MC0)
17 #define TIMER_A_CM                  (TACTL |= MC1); \
18                                     (TACTL &= ~MC0);
19 #define TIMER_A_STOP                (TACTL &= ~(MC0 | MC1))
20 #define TIMER_A_RESET               (TACTL = TACLR);
21
22 #define TIMER_A_FLAG0_RESET         (TACCTL0 &= ~(CCIFG|COV))
23 #define TIMER_A_FLAG1_RESET         (TACCTL1 &= ~(CCIFG|COV))
24
25 #define TIMER_A_SOURCE_SMCLK        (TACTL |= TASSEL1)
26 #define TIMER_A_SOURCE_DIV_8        (TACTL |= (ID1 | ID0))
27 #define TIMER_A_SOURCE_DIV_4        (TACTL |= ID1); \
28                                     (TACTL &= ~ID0);
29 #define TIMER_A_SOURCE_DIV_2        (TACTL &= ~ID1); \
30                                     (TACTL |= ID0);
31 #define TIMER_A_SOURCE_DIV_1        TACTL &= ~ID1; \
32                                     TACTL &= ~ID0;
33 #define TIMER_A_OUTMOD_RESET_SET    (TACCTL1 |= (OUTMOD2 | OUTMOD1 | OUTMOD0))
34 #define TIMER_A_OUTMOD_TOGGLE_SET   (TACCTL1 |= (OUTMOD2 | OUTMOD1)); \
35                                     (TACCTL1 &= ~OUTMOD0);
36 #define TIMER_A_OUTMOD_RESET        (TACCTL1 |= (OUTMOD2 | OUTMOD0)); \
37                                     (TACCTL1 &= ~OUTMOD1);
38
39 #define TIMER_A_0_CM_IRQ_ENABLE     (TACCTL0 |= CCIE); \
40                                     (TACTL |= TAIE);
41 #define TIMER_A_1_CM_IRQ_ENABLE     (TACCTL1 |= CCIE);\
42                                     (TACTL |= TAIE);
43 #define TIMER_A_2_CM_IRQ_ENABLE     (TACCTL2 |= CCIE);
44
45 #define TIMER_A_0_CM_IRQ_DISABLE    (TBCCTL0 &= ~CCIE);
46
47 #define TIMER_A_1_CM_IRQ_DISABLE    (TBCCTL1 &= ~CCIE);
48
49 #define TIMER_A_2_CM_IRQ_DISABLE    (TBCCTL2 &= ~CCIE);
50
51 /************************************************************
52 **  Defines Timer B
53 ************************************************************/
54 #define TIMER_B_START               (TBCTL |= (MC0 | ID1 | ID0))
55 #define TIMER_B_STOP                (TBCTL &= ~(MC0 | MC1))
56 #define TIMER_B_RESET               (TBCTL |= TBCLR)
57
58 /**** Prototyp declaration ************************************************/
59 void timer_a_wakeup_init(void);
60 void timer_a_init_burst(uint8_t);
61 void timer_a_second_init(void);
62 void timer_b_init(void);
63
64 #endif
```

F.2.14. uart.c

```
/*******************************************************************************
**    Name:           uart.c
**    Hardware:       BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 03/2013
**    Date:           07/03/2013
**    Author:         Nico Sassano
**                    modified taken from Phillip Durdaut
*******************************************************************************/

#include "main.h"

void uart_init(void) {

      U0CTL |= 0x01;          // UART Reset für Konfiguration
      P3DIR &= ~(0x20);       // P3.5 als Input
      P3DIR |= 0x10;          // P3.4 als Output
      P3SEL |= 0x20 | 0x10;   // Alternativ-Funktion von P3.5 + 4 auswählen (UART)

      U0CTL &= ~(0x80);       // keine Parität
      U0CTL &= ~(0x20);       // ein Stoppbit
      U0CTL |= 0x10;          // 8-Bit-Daten
      U0CTL &= ~(0x08);       // kein Listen-Mode
      U0CTL &= ~(0x04);       // UART-Mode auswählen
      U0CTL &= ~(0x02);       // kein Mulitprozessor-Mode

      U0TCTL &= ~CKPL;                // UCLKI = UCLK
      U0TCTL |= (SSEL1 | SSEL1);      // BRCLK = SMCLK

      // Baudrate: 28800 mit SMCLK (8 MHz)
      UBR10 = 0x01;
      UBR00 = 0x15;
      UMCTL0 = 0xbb;

      ME1 |= 0x80;       // Senden aktivieren

      ME1 |= 0x40;       // Empfangen aktivieren

      U0CTL &= ~(0x01); // UART Reset zurücksetzen
}

void uart_tx(uint8_t *buffer, uint8_t length) {
    uint8_t index;

    for (index = 0; index < length; index++) {

        TXBUF0 = buffer[index];          // Send data
        while (!(IFG1 & UTXIFG0));       // Wait for TX to finish
        delay_100us(10);
    }
    delay_100us(2);
}
```

F.2.15. uart.h

```
1  /*****************************************************************************
2  **     Name:              uart.c
3  **     Hardware:          BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 03/2013
4  **     Date:              07/03/2013
5  **     Author:            Nico Sassano
6  **                        modified taken from Phillip Durdaut
7  *****************************************************************************/
8  #ifndef UART_H_
9  #define UART_H_
10
11 #include "main.h"
12
13 /*-------------------------------------------------------------------------
14   Defines -> Need to be changed depending on hardware
15  -------------------------------------------------------------------------*/
16 #define UART_TX_PxDIR                   P3DIR
17 #define UART_TX_PxSEL                   P3SEL
18 #define UART_TX_PIN                     BIT4
19
20 /*-------------------------------------------------------------------------
21   Public functions
22  -------------------------------------------------------------------------*/
23 void uart_init(void);
24 void uart_tx(uint8_t * buffer, uint8_t length);
25
26 #endif /* UART_H_ */
```

F.2.16. uartmenue.c

```
/*******************************************************************************
**     Name:            uart_menue.c
**     Hardware:        BATSEN ZS Klasse 3 v0.2 – Nico Sassano – 06/2013
**     Date:            12/06/2013
**     Author:          Nico Sassano
*******************************************************************************/
#include "main.h"

uint16_t uart_counter = 0x0000;

extern volatile uint8_t reciveString[];
extern volatile uint8_t string_index;
extern volatile state_t state;
extern volatile uint8_t frame_counter;
extern uint16_t cali_nominal_tmp102;
extern uint16_t cali_nominal_msp430;

extern uint8_t current_sensor;

/*********************************************
* Data Buffer
*********************************************/
extern uint16_t burst_data[50];
extern uint16_t sample_buf_volt[4];
extern uint16_t sample_buf_temp[4];
extern uint16_t sample_buf_temp_msp[4];

/*********************************************
** Burst Mode
*********************************************/
extern uint8_t burst_freq; // Burst Frequenz

/*********************************************
** Frames
*********************************************/
extern volatile uint8_t rx_data_length;
extern volatile uint8_t frame_counter;

/*********************************************
* Errorbuffer
*********************************************/
extern volatile uint8_t error_check_awake;

void uart_menue(void) {
    uint8_t uartbuf[50];
    uint16_t msb = 0;
    uint16_t lsb = 0;

    if((reciveString[1] == '0') && (reciveString[2] == 0x0d)) {
        /*********************************************************************
        ** Wakeup ALL
        *********************************************************************/

        state = S_WAKEUP_ALL;


    } else if((reciveString[1] == '1') && (reciveString[2] == 0x0d)) {
        /*********************************************************************
        ** Wakeup
        *********************************************************************/

            state = S_WAKEUP;

            sprintf(uartbuf,"Send wakeup");
            uartbuf[11] = 13;
            uart_tx(uartbuf, 12);
    } else if((reciveString[1] == 'h') && (reciveString[2] == 'e') && (reciveString[3] == 'l') && (reciveString[4] == 'p') && (
        reciveString[5] == 0x0d)) {
        /*********************************************************************
        ** Help Menue
        *********************************************************************/
        state = S_START;

        sprintf(uartbuf,"Help menue:");
        uartbuf[11] = 13;
        uart_tx(uartbuf, 12);

        sprintf(uartbuf,"0  -> Wake-up and config");
        uartbuf[24] = 13;
        uart_tx(uartbuf, 25);

        sprintf(uartbuf,"1  -> Sending wake-up only");
        uartbuf[26] = 13;
```

```
83          uart_tx(uartbuf, 27);
84
85          sprintf(uartbuf,"2  -> Checking wake-up");
86          uartbuf[22] = 13;
87          uart_tx(uartbuf, 23);
88
89          sprintf(uartbuf,"3  -> Set sensor for config");
90          uartbuf[27] = 13;
91          uart_tx(uartbuf, 28);
92
93          sprintf(uartbuf,"4  -> Sending the config");
94          uartbuf[24] = 13;
95          uart_tx(uartbuf, 25);
96
97          sprintf(uartbuf,"5  -> Sample voltage");
98          uartbuf[20] = 13;
99          uart_tx(uartbuf, 21);
100
101         sprintf(uartbuf,"6  -> Sending voltage");
102         uartbuf[21] = 13;
103         uart_tx(uartbuf, 22);
104
105         sprintf(uartbuf,"7  -> Sample voltage and temperature of the TMP102");
106         uartbuf[50] = 13;
107         uart_tx(uartbuf, 51);
108
109         sprintf(uartbuf,"8  -> Send voltage and temperature of the TMP102");
110         uartbuf[48] = 13;
111         uart_tx(uartbuf, 49);
112
113         sprintf(uartbuf,"9  -> Sample temperature of the TMP102");
114         uartbuf[38] = 13;
115         uart_tx(uartbuf, 39);
116
117         sprintf(uartbuf,"10 -> Send temperature of the TMP102");
118         uartbuf[36] = 13;
119         uart_tx(uartbuf, 37);
120
121         sprintf(uartbuf,"11 -> Sample temperature of the MSP430");
122         uartbuf[38] = 13;
123         uart_tx(uartbuf, 39);
124
125         sprintf(uartbuf,"12 -> Send temperature of the MSP430");
126         uartbuf[36] = 13;
127         uart_tx(uartbuf, 37);
128
129         sprintf(uartbuf,"13 -> Start burst measurement");
130         uartbuf[29] = 13;
131         uart_tx(uartbuf, 30);
132
133         sprintf(uartbuf,"14 -> Get config data for burst measurement");
134         uartbuf[43] = 13;
135         uart_tx(uartbuf, 44);
136
137         sprintf(uartbuf,"15 -> Start burst data transmission");
138         uartbuf[35] = 13;
139         uart_tx(uartbuf, 36);
140
141         sprintf(uartbuf,"16 -> Balancing on");
142         uartbuf[18] = 13;
143         uart_tx(uartbuf, 19);
144
145         sprintf(uartbuf,"17 -> Balancing off");
146         uartbuf[19] = 13;
147         uart_tx(uartbuf, 20);
148
149
150     } else if((reciveString[1] == 0x73) && (reciveString[2] == 0x74) && (reciveString[3] == 0x61) && (reciveString[4] == 0x72) && (
            reciveString[5] == 0x74) &&(reciveString[6] == 0x0d)) {
151         /*********************************************************************
152         ** Wakeup
153         *********************************************************************/
154
155         state = S_WAKEUP;
156
157         sprintf(uartbuf,"Send wakeup");
158         uartbuf[11] = 13;
159         uart_tx(uartbuf, 12);
160
161     } else if((reciveString[1] == 0x32) && (reciveString[2] == 0x0d)) {
162         /*********************************************************************
163         ** Check Wakeup
164         *********************************************************************/
165         state = S_TX_IS_AWAKE;
166
```

```
167             sprintf(uartbuf,"Check wakeup");
168             uartbuf[12] = 13;
169             uart_tx(uartbuf, 13);
170
171         } else if((reciveString[1] == 0x33) && (reciveString[2] == 0x0d)) {
172             /*********************************************************************
173             ** Set Config
174             *********************************************************************/
175             state = S_TX_CONFIG_SET;
176
177             sprintf(uartbuf,"Set Config");
178             uartbuf[10] = 13;
179             uart_tx(uartbuf, 11);
180
181         } else if((reciveString[1] == 0x34) && (reciveString[2] == 0x0d)) {
182             /*********************************************************************
183             ** Send Config
184             *********************************************************************/
185             state = S_TX_CONFIG;
186
187             sprintf(uartbuf,"Send Config");
188             uartbuf[11] = 13;
189             uart_tx(uartbuf, 12);
190
191         } else if((reciveString[1] == 0x35) && (reciveString[2] == 0x0d)) {
192             /*********************************************************************
193             ** Sample Voltage
194             *********************************************************************/
195             state = S_TX_SAMPLE_VOLTAGE;
196
197             sprintf(uartbuf,"Sample Voltage");
198
199             uartbuf[14] = 13;
200             uart_tx(uartbuf, 15);
201
202         } else if((reciveString[1] == 0x36) && (reciveString[2] == 0x0d)) {
203             /*********************************************************************
204             ** Send Voltage
205             *********************************************************************/
206             state = S_TX_SEND_VOLTAGE;
207             sprintf(uartbuf,"Send Voltage");
208
209             uartbuf[12] = 13;
210             uart_tx(uartbuf, 13);
211
212         } else if((reciveString[1] == 0x37) && (reciveString[2] == 0x0d)) {
213             /*********************************************************************
214             ** Sample Voltage & Temperature
215             *********************************************************************/
216             state = S_TX_SAMPLE_VOLTAGE_TEMPERATUR;
217
218             sprintf(uartbuf,"Sample Voltage and Temperature");
219
220             uartbuf[30] = 13;
221             uart_tx(uartbuf, 31);
222
223         } else if((reciveString[1] == 0x38) && (reciveString[2] == 0x0d)) {
224             /*********************************************************************
225             ** Send Voltage & Temperature
226             *********************************************************************/
227             state = S_TX_SEND_VOLTAGE_TEMPERATUR;
228
229             sprintf(uartbuf,"Send Voltage and Temperature");
230
231             uartbuf[28] = 13;
232             uart_tx(uartbuf, 29);
233
234         } else if((reciveString[1] == 0x39) && (reciveString[2] == 0x0d)) {
235             /*********************************************************************
236             ** Sample Temperature TMP102
237             *********************************************************************/
238             state = S_TX_SAMPLE_TEMPERATUR_TMP102;
239
240             sprintf(uartbuf,"Sample Temperature TMP102");
241
242             uartbuf[25] = 13;
243             uart_tx(uartbuf, 26);
244
245         } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x30) && (reciveString[3] == 0x0d)) {
246             /*********************************************************************
247             ** Send Temperature TMP102
248             *********************************************************************/
249             state = S_TX_SEND_TEMPERATUR_TMP102;
250
251             sprintf(uartbuf,"Send Temperature TMP102");
```

```
252
253             uartbuf[23] = 13;
254             uart_tx(uartbuf, 24);
255
256         } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x31) && (reciveString[3] == 0x0d)) {
257             /*********************************************************************
258             ** Sample Temperature MSP430
259             *********************************************************************/
260             state = S_TX_SAMPLE_TEMPERATUR_MSP430;
261
262             sprintf(uartbuf,"Sample Temperature MSP430");
263
264             uartbuf[25] = 13;
265             uart_tx(uartbuf, 26);
266
267         } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x32) && (reciveString[3] == 0x0d)) {
268             /*********************************************************************
269             ** Send Temperature MSP430
270             *********************************************************************/
271             state = S_TX_SEND_TEMPERATUR_MSP430;
272
273             sprintf(uartbuf,"Send Temperature MSP430");
274
275             uartbuf[23] = 13;
276             uart_tx(uartbuf, 24);
277         } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x33) && (reciveString[3] == 0x0d)) {
278             /*********************************************************************
279             ** Burst mode
280             *********************************************************************/
281             sprintf(uartbuf,"Bitte Burstfrequenz eingeben:");
282             uartbuf[29] = 0x0D;
283             uart_tx(uartbuf, 30);
284
285             sprintf(uartbuf," 1 ->  50Hz");
286             uartbuf[11] = 0x0D;
287             uart_tx(uartbuf, 12);
288
289             sprintf(uartbuf," 2 -> 100Hz");
290             uartbuf[11] = 0x0D;
291             uart_tx(uartbuf, 12);
292
293             sprintf(uartbuf," 3 -> 150Hz");
294             uartbuf[11] = 0x0D;
295             uart_tx(uartbuf, 12);
296
297             sprintf(uartbuf," 4 -> 200Hz");
298             uartbuf[11] = 0x0D;
299             uart_tx(uartbuf, 12);
300
301             sprintf(uartbuf," 5 -> 250Hz");
302             uartbuf[11] = 0x0D;
303             uart_tx(uartbuf, 12);
304
305             sprintf(uartbuf," 6 -> 300Hz");
306             uartbuf[11] = 0x0D;
307             uart_tx(uartbuf, 12);
308
309             sprintf(uartbuf," 7 -> 350Hz");
310             uartbuf[11] = 0x0D;
311             uart_tx(uartbuf, 12);
312
313             sprintf(uartbuf," 8 -> 400Hz");
314             uartbuf[11] = 0x0D;
315             uart_tx(uartbuf, 12);
316
317             sprintf(uartbuf," 9 -> 450Hz");
318             uartbuf[11] = 0x0D;
319             uart_tx(uartbuf, 12);
320
321             sprintf(uartbuf,"10 -> 500Hz");
322             uartbuf[11] = 0x0D;
323             uart_tx(uartbuf, 12);
324
325             sprintf(uartbuf,"11 -> 550Hz");
326             uartbuf[11] = 0x0D;
327             uart_tx(uartbuf, 12);
328
329             sprintf(uartbuf,"12 -> 600Hz");
330             uartbuf[11] = 0x0D;
331             uart_tx(uartbuf, 12);
332
333             sprintf(uartbuf,"13 -> 650Hz");
334             uartbuf[11] = 0x0D;
335             uart_tx(uartbuf, 12);
336
```

```
337             sprintf(uartbuf,"14 -> 700Hz");
338             uartbuf[11] = 13;
339             uart_tx(uartbuf, 12);
340
341             sprintf(uartbuf,"15 -> 750Hz");
342             uartbuf[11] = 0x0D;
343             uart_tx(uartbuf, 12);
344
345             sprintf(uartbuf,"16 -> 800Hz");
346             uartbuf[11] = 0x0D;
347             uart_tx(uartbuf, 12);
348
349             sprintf(uartbuf,"17 -> 850Hz");
350             uartbuf[11] = 0x0D;
351             uart_tx(uartbuf, 12);
352
353             sprintf(uartbuf,"18 -> 900Hz");
354             uartbuf[11] = 0x0D;
355             uart_tx(uartbuf, 12);
356
357             sprintf(uartbuf,"19 -> 950Hz");
358             uartbuf[11] = 0x0D;
359             uart_tx(uartbuf, 12);
360
361             sprintf(uartbuf,"20 -> 1kHz");
362             uartbuf[10] = 0x0D;
363             uart_tx(uartbuf, 11);
364
365             sprintf(uartbuf,"21 -> 2kHz");
366             uartbuf[10] = 0x0D;
367             uart_tx(uartbuf, 11);
368
369             sprintf(uartbuf,"22 -> 4kHz");
370             uartbuf[10] = 0x0D;
371             uart_tx(uartbuf, 11);
372
373             sprintf(uartbuf,"23 -> 6kHz");
374             uartbuf[10] = 0x0D;
375             uart_tx(uartbuf, 11);
376
377             sprintf(uartbuf,"24 -> 8kHz");
378             uartbuf[10] = 0x0D;
379             uart_tx(uartbuf, 11);
380
381             sprintf(uartbuf,"25 -> 10kHz");
382             uartbuf[11] = 0x0D;
383             uart_tx(uartbuf, 12);
384
385             for(string_index = 0; string_index != 10; string_index++)
386                 reciveString[string_index] = 0;
387
388             string_index = 0;
389
390             do {
391                 string_index++;
392                 while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
393                 reciveString[string_index] = U0RXBUF;
394
395             }while(reciveString[string_index] != 13);
396
397             if((reciveString[1] == 0x31) && (reciveString[2] == 0x0d))
398                 burst_freq = BURST_FREQ_50HZ;
399             else if((reciveString[1] == 0x32) && (reciveString[2] == 0x0d))
400                 burst_freq = BURST_FREQ_100HZ;
401             else if((reciveString[1] == 0x33) && (reciveString[2] == 0x0d))
402                 burst_freq = BURST_FREQ_150HZ;
403             else if((reciveString[1] == 0x34) && (reciveString[2] == 0x0d))
404                 burst_freq = BURST_FREQ_200HZ;
405             else if((reciveString[1] == 0x35) && (reciveString[2] == 0x0d))
406                 burst_freq = BURST_FREQ_250HZ;
407             else if((reciveString[1] == 0x36) && (reciveString[2] == 0x0d))
408                 burst_freq = BURST_FREQ_300HZ;
409             else if((reciveString[1] == 0x37) && (reciveString[2] == 0x0d))
410                 burst_freq = BURST_FREQ_350HZ;
411             else if((reciveString[1] == 0x38) && (reciveString[2] == 0x0d))
412                 burst_freq = BURST_FREQ_400HZ;
413             else if((reciveString[1] == 0x39) && (reciveString[2] == 0x0d))
414                 burst_freq = BURST_FREQ_450HZ;
415             else if((reciveString[1] == 0x31) && (reciveString[2] == 0x30) && (reciveString[3] == 0x0d))
416                 burst_freq = BURST_FREQ_500HZ;
417             else if((reciveString[1] == 0x31) && (reciveString[2] == 0x31) && (reciveString[3] == 0x0d))
418                 burst_freq = BURST_FREQ_550HZ;
419             else if((reciveString[1] == 0x31) && (reciveString[2] == 0x32) && (reciveString[3] == 0x0d))
420                 burst_freq = BURST_FREQ_600HZ;
421             else if((reciveString[1] == 0x31) && (reciveString[2] == 0x33) && (reciveString[3] == 0x0d))
```

```
422             burst_freq = BURST_FREQ_650HZ;
423         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x34) && (reciveString[3] == 0x0d))
424             burst_freq = BURST_FREQ_700HZ;
425         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x35) && (reciveString[3] == 0x0d))
426             burst_freq = BURST_FREQ_750HZ;
427         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x36) && (reciveString[3] == 0x0d))
428             burst_freq = BURST_FREQ_800HZ;
429         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x37) && (reciveString[3] == 0x0d))
430             burst_freq = BURST_FREQ_850HZ;
431         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x38) && (reciveString[3] == 0x0d))
432             burst_freq = BURST_FREQ_900HZ;
433         else if((reciveString[1] == 0x31) && (reciveString[2] == 0x39) && (reciveString[3] == 0x0d))
434             burst_freq = BURST_FREQ_950HZ;
435         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x30) && (reciveString[3] == 0x0d))
436             burst_freq = BURST_FREQ_1000HZ;
437         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x31) && (reciveString[3] == 0x0d))
438             burst_freq = BURST_FREQ_2000HZ;
439         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x32) && (reciveString[3] == 0x0d))
440             burst_freq = BURST_FREQ_4000HZ;
441         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x33) && (reciveString[3] == 0x0d))
442             burst_freq = BURST_FREQ_6000HZ;
443         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x34) && (reciveString[3] == 0x0d))
444             burst_freq = BURST_FREQ_8000HZ;
445         else if((reciveString[1] == 0x32) && (reciveString[2] == 0x35) && (reciveString[3] == 0x0d))
446             burst_freq = BURST_FREQ_10000HZ;
447
448         sprintf(uartbuf,"Burstmode has start");
449         uartbuf[19] = 13;
450         uart_tx(uartbuf, 20);
451
452         state = S_TX_BURST_MODE;
453
454     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x34) && (reciveString[3] == 0x0d)) {
455         state = S_BURST_DATA_RQ;
456
457     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x35) && (reciveString[3] == 0x0d)) {
458         state = S_BURST_DATA_RX;
459
460         sprintf(uartbuf,"%03X",frame_counter);
461         uartbuf[6] = 13;
462         uart_tx(uartbuf, 7);
463     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x36) && (reciveString[3] == 0x0d)) {
464
465         /*********************************************************************
466         ** Balancierung an
467         *********************************************************************/
468         state = S_TX_BALANCING_ON;
469
470         sprintf(uartbuf,"Balancing on ");
471         uartbuf[14] = 13;
472         uart_tx(uartbuf, 15);
473
474     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x37) && (reciveString[3] == 0x0d)) {
475         /*********************************************************************
476         ** Balancierung aus
477         *********************************************************************/
478         state = S_TX_BALANCING_OFF;
479
480         sprintf(uartbuf,"Balancing off");
481         uartbuf[14] = 13;
482         uart_tx(uartbuf, 15);
483
484     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x38) && (reciveString[3] == 0x0d)) {
485         /*****************************************************
486          ** Kalibrierung TMP102 (TEST)
487          *****************************************************/
488         state = S_TX_CALIBRATION_TMP102;
489
490         sprintf(uartbuf,"Please input nominal value for the TMP102:");
491         uartbuf[43] = 13;
492         uart_tx(uartbuf, 44);
493
494         while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
495         msb = (((uint16_t)(U0RXBUF & 0x0F)) << 8) & 0x0F00;
496         while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
497         lsb = (((uint16_t)(U0RXBUF & 0xFF)) << 0) & 0x00FF;
498
499         cali_nominal_tmp102 = msb | lsb;
500
501     } else if((reciveString[1] == 0x31) && (reciveString[2] == 0x39) && (reciveString[3] == 0x0d)) {
502         /*****************************************************
503         ** Kalibrierung MSP430 (TEST)
504         *****************************************************/
505
506         state = S_TX_CALIBRATION_MSP430;
```

```
507
508             sprintf(uartbuf,"Please input nominal value for the MSP430:");
509             uartbuf[43] = 13;
510             uart_tx(uartbuf, 44);
511
512             while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
513             msb = (((uint16_t)(U0RXBUF & 0x0F)) << 8) & 0x0F00;
514             while( (IFG1 & URXIFG0) == 0); // Auf Zeichen Warten
515             lsb = (((uint16_t)(U0RXBUF & 0xFF)) << 0) & 0x00FF;
516
517             cali_nominal_msp430 = msb | lsb;
518
519         } else if((reciveString[1] == 0x32) && (reciveString[2] == 0x30) && (reciveString[3] == 0x0d)) {
520             /*********************************************************************
521             ** Sample Voltage and Temp ALL
522             *********************************************************************/
523
524              state = S_TX_SAMPLE_VOLTAGE_TEMPERATUR_ALL;
525
526
527          }  else if((reciveString[1] == 0x32) && (reciveString[2] == 0x31) && (reciveString[3] == 0x0d)) {
528              /*********************************************************************
529              ** Send Voltage and Temp ALL
530              *********************************************************************/
531
532               state = S_TX_SEND_VOLTAGE_TEMPERATUR_ALL;
533
534          }
535     }
536
537     /*****************************************************************************************************
538     ** Send data via UART
539     *****************************************************************************************************/
540     void tx_uart(void) {
541
542         uint8_t uartbuf[50] = {0};
543
544         switch(state) {
545
546             case S_WAKEUP_BROADCAST: {
547
548                 uart_init();
549
550                 sprintf(uartbuf,"Wakeup done");
551                 uartbuf[11] = 13;
552                 uart_tx(uartbuf, 12);
553             }break;
554
555             /*********************************************************************
556             ** UART Ausgabe ob Sensor er Wach ist
557             *********************************************************************/
558             case S_TX_IS_AWAKE: {
559
560                 uart_init();
561
562                 if(error_check_awake) {
563                     sprintf(uartbuf,"Sensor %03d is awake",current_sensor+1);
564                     uartbuf[19] = 13;
565                     uart_tx(uartbuf, 20);
566                 } else {
567                     sprintf(uartbuf,"No answer from sensor %03d",current_sensor+1);
568                     uartbuf[25] = 13;
569                     uart_tx(uartbuf, 26);
570                 }
571
572
573             } break;
574
575             /*********************************************************************
576             ** Ausgabe der Spannungswerte
577             *********************************************************************/
578             case S_TX_SEND_VOLTAGE: {
579
580                 uart_init();
581
582                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
583                     sprintf(uartbuf,"%04d, V%04d", uart_counter, sample_buf_volt[current_sensor]);
584                     uartbuf[11] = 13;
585                     uart_tx(uartbuf, 12);
586                 }
587
588                 uart_counter++;
589             } break;
590
591             /*********************************************************************
```

```
592             ** Ausgabe der Spannung- und Temperaturwerte
593             ********************************************************************/
594             case S_TX_SEND_VOLTAGE_TEMPERATUR: {
595
596                 uart_init();
597
598                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
599                     sprintf(uartbuf,"%04d, V%04d, T%04d", uart_counter, sample_buf_volt[current_sensor], sample_buf_temp[current_sensor]);
600                     uartbuf[18] = 13;
601                     uart_tx(uartbuf, 19);
602                 }
603
604                 uart_counter++;
605
606             } break;
607
608             /********************************************************************
609             ** Ausgabe der Spannung und den beiden Temperaturen (TMP und MSP430)
610             ********************************************************************/
611             case S_TX_SEND_VOLTAGE_TEMPERATUR_ALL: {
612
613                 uart_init();
614
615                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
616                     sprintf(uartbuf,"%04d, V%04d, T%04d, T%04d", uart_counter, sample_buf_volt[current_sensor], sample_buf_temp[
                            current_sensor],sample_buf_temp_msp[current_sensor]);
617                     uartbuf[25] = 13;
618                     uart_tx(uartbuf, 26);
619                 }
620
621                 uart_counter++;
622
623             } break;
624
625             /********************************************************************
626             ** Ausgabe der Temperatur TMP102
627             ********************************************************************/
628             case S_TX_SEND_TEMPERATUR_TMP102: {
629
630                 uart_init();
631
632                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
633                     sprintf(uartbuf,"%04d T%04d", uart_counter, sample_buf_temp[current_sensor]);
634                     uartbuf[11] = 13;
635                     uart_tx(uartbuf, 12);
636                 }
637
638                 uart_counter++;
639
640
641             } break;
642
643             /********************************************************************
644             ** Ausgabe der Temperatur MSP430
645             ********************************************************************/
646             case S_TX_SEND_TEMPERATUR_MSP430: {
647
648                 uart_init();
649
650                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++)
651                     sprintf(uartbuf,"%03X T%04X", uart_counter, sample_buf_temp_msp[current_sensor]);
652
653                 uart_counter++;
654
655                 uartbuf[9] = 13;
656                 uart_tx(uartbuf, 16);
657             } break;
658
659             /********************************************************************
660             ** Ausgabe der Burst Daten
661             ********************************************************************/
662             case S_BURST_DATA_RX: {
663
664                 uart_init();
665                 uint8_t index = 0;
666                 for(index = 0;index<(rx_data_length/2);index++) {
667                     sprintf(uartbuf,"%04d V%04d", uart_counter, burst_data[index]);
668
669                     uartbuf[11] = 13;
670                     uart_tx(uartbuf, 12);
671                     uart_counter++;
672                 }
673             }break;
674
675             /********************************************************************
```

```
676             ** Ausgabe der Balancierungsdaten
677             ************************************************************/
678             case S_TX_BALANCING_ON: {
679
680                 uart_init();
681
682                 for(current_sensor = 0; current_sensor < NUMBER_OF_SENSORS; current_sensor++) {
683                     sprintf(uartbuf,"%04d, V%04d, T%04d", uart_counter, sample_buf_volt[current_sensor], sample_buf_temp[current_sensor]);
684                     uartbuf[18] = 13;
685                     uart_tx(uartbuf, 19);;
686                 }
687
688                 uart_counter++;
689             } break;
690
691             default: break;
692         }
693     }
```

F.2.17. uartmenue.h

```
1  /*******************************************************************************
2  **     Name:              uart_menue.h
3  **     Hardware:          BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 06/2013
4  **     Date:              12/06/2013
5  **     Author:            Nico Sassano
6  *******************************************************************************/
7  #ifndef UART_MENUE_H_
8  #define UART_MENUE_H_
9
10 void uart_menue(void);
11 void tx_uart(void);
12
13 #endif /* UART_MENUE_H_ */
```

F.2.18. wakeup.c

```
1  /*******************************************************************************
2  **    Name:           wakeup.c
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 06/2013
4  **    Date:           24/06/2013
5  **    Author:         Nico Sassano
6  *******************************************************************************/
7  #include "main.h"
8  /*****************************************************************************************************
9  ** Wake Up senden
10 *****************************************************************************************************/
11 void tx_wakeup(void) {
12     CC1101_GDO2_IRQ_DISABLE;          // Disable Interrupt
13     CC1101_GDO2_CLEAR_IRQ;
14
15     TXPIN_DIR_IN;                     // TX pin is input until CC1101 is in TX state
16     TIMER_A_STOP;                     // Make sure timer for TX pin toggling is not running
17     TIMER_A_RESET;                    // Reset timer for TX pin toggling
18
19     cc1101_reset();                   // Reset all registers to their default values and go to idle state
20     cc1101_config_no_packet();        // Configure the transceiver for sending the wakeup signal
21     cc1101_tx_asynchronous_mode();    // Change to TX state
22     timer_a_wakeup_init();            //Timer for wakeup 02.04.13 NS
23
24     TXPIN_DIR_OUT;                    // TX pin is output
25     TXPIN_TIMER;                      // TX pin driven by the timer
26     TIMER_A_UP;                       // Start the timer that is toggling the TX pin
27
28     delay_ms(WAKEUP_DURATION_MS);     // Transmit the wakeup signal
29
30     TIMER_A_STOP;                     // Stop the timer again
31     TIMER_A_RESET;
32     TXPIN_MAN;                        // TX pin controlled manually
33     TXPIN_DIR_IN;                     // TX pin is input again
34
35     cc1101_idle();                    // Transceiver back to IDLE state
36 }
```

F.2.19. wakeup.h

```
1  /*******************************************************************************
2  **    Name:           wakeup.h
3  **    Hardware:       BATSEN ZS Klasse 3 v0.2 - Nico Sassano - 06/2013
4  **    Date:           24/06/2013
5  **    Author:         Nico Sassano
6  *******************************************************************************/
7
8  #ifndef WAKEUP_H_
9  #define WAKEUP_H_
10
11 #include "main.h"
12
13 void tx_wakeup(void);
14
15 #endif /* WAKEUP_H_ */
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 16. August 2013	
Ort, Datum	Unterschrift