Bachelorthesis

Michael Meinzer

Hard- und Softwareentwicklung sowie Erprobung drahtloser Zellensensoren für Fahrzeugbatterien

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Michael Meinzer

Hard- und Softwareentwicklung sowie Erprobung drahtloser Zellensensoren für Fahrzeugbatterien

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

Betreuender Prüfer : Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr. rer.nat. Henning Dierks

Abgegeben am 08. Oktober 2013

Michael Meinzer

Thema der Bachelorthesis

Hard- und Softwareentwicklung sowie Erprobung drahtloser Zellensensoren für Fahrzeugbatterien

Stichworte

Sensorik, Fahrzeugbatterie, drahtlose Sensoren, Erprobung, Warteschlange, Mikrocontroller

Kurzzusammenfassung

Defekte Fahrzeugbatterien sind eine häufige Ursache für Pannen am Kraftfahrzeug. Um die Sicherheit und die Zuverlässigkeit der Batterie zu erhöhen ist es notwendig, die Spannung und die Temperatur jeder einzelnen Zelle zu erfassen. Die drahtlose Datenübertragung ermöglicht eine kostengünstige, robuste und galvanisch getrennte Erfassung der Daten. Um bei Hochstromereignissen eine hohe Messpunktdichte zu ermöglichen, wird eine Zwischenspeicherung der Messergebnisse im Sensor durchgeführt. Die zwischengespeicherten Werte werden nach einem Warteschlangen-ähnlichen Prinzip im Anschluß an das Hochstromereignis übertragen. Diese Arbeit basiert auf Vorarbeiten zur drahtlosen Batteriesensorik und umfasst die Weiterentwicklung eines Sensors. Die Konzeption, Hard- und Softwareentwicklung sowie die Erprobung sind die wesentlichen Bestandteile dieser Arbeit.

Michael Meinzer

Title of the paper

Hard- and software development of cell sensors and its proving on car batteries

Keywords

Sensors, car battery, wireless sensors, proving, queue, micro-controller

Abstract

Defect car batteries are a frequent cause of breakdowns on the motor vehicles. To increase the safety and reliability of the battery it is necessary to detect the voltage and the temperature of each cell. The wireless data transfer enables affordable, rugged and galvanically isolated detection of the data. To allow high current events a high point density, the sensor caches the measurement results. The cached values are queue-like principle transferred after the high-current event. This thesis is based on preparatory work on the wireless battery-sensor and includes the further development of a sensor. The design, hardware and software development and testing are key components of this thesis.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AC	Alternating Current
ACK	Acknowledgement
ADAC	Allgemeiner Deutscher Automobil Club
ADC	Analog Digital Converter
BMBF	Bundesministerium für Bildung und Forschung
BMW	Bayerische Motoren Werke
CRC	Cyclic Redundancy Check
ESD	Electrostatic Discharge
FSK	Frequency Shift Keying
GPIO	General-Purpose Input/Output
I²C	Inter-Integrated Circuit
ISM	Industrial, Scientific and Medical Band
JTAG	Joint Test Action Group
LSB	Least Significant Bit
MSOP	Mini Small Outline Package
OOK	On-Off Keying
RAM	Random-Access Memory
SI	Silicon Labs
SMD	Surface-Mounted Device
SON	Small Outline No-Lead
TAB	Büro für Technikfolgen-Abschätzung beim Deutschen Bundestag
TI	Texas Instruments
TSSOP	Thin-Shrink Small Outline Package
UART	Universal Asynchronous Receiver Transmitter

1 Einführung

1.1 Einleitung

Die Automobilbranche stellt den größten Wirtschaftszweig in Deutschland dar. Mit über 1,8 Millionen Erwerbstätigen erwirtschaftet diese Branche jährlich etwa 350 Milliarden Euro und somit ca. ein Fünftel des Gesamtumsatzes des verarbeitenden Gewerbes in Deutschland. Das Branchensegment zur Herstellung von Kraftfahrzeugen und der dazugehörigen vor- und nachgelagerten Bereiche stellt knapp ein Drittel aller jährlichen Forschungs- und Entwicklungsaufwendungen in Gesamtdeutschland zur Verfügung [1].

Bei der Betrachtung von Pannenstatistiken deutscher Automobilclubs lassen sich Schwerpunkte für das Vorkommen eines Fahrzeugstillstandes ausmachen. Die Kfz-Elektrik zählt hierbei neben mechanischen Motorschäden und Problematiken im Motormanagement zu den häufigsten Ursachen eines liegengebliebenen Fahrzeuges. Im Jahr 2008 waren 40 % aller Einsätze des Allgemeinen Deutschen Automobilclubs auf Störungen der allgemeinen Fahrzeugelektrik zurückzuführen [2]. Als größter deutscher Automobilclub identifiziert der ADAC in seiner Aufstellung aus dem Jahr 2013 mit ebenfalls etwa 40 Prozent weiterhin insbesondere die Elektronik als größte Fehlerursache für ein liegengebliebenes Automobil. Ein tieferer Blick in die Statistik 2013 zeigt deutlich, dass über 31 Prozent aller ausgefallenen Fahrzeuge und somit ca. 87 Prozent aller Elektronikfehler allein auf einen Batteriedefekt und dem dazugehörigen Verbrauchsmanagement zurückzuführen sind [3].

In der modernen Fahrzeugtechnik steigt die Anzahl elektronischer Verbraucher im Automobil kontinuierlich an. Neben zunehmenden elektronischen Sicherheits- und Fahrerassistenzsystemen steigt ebenfalls die Zahl an elektrischen Komfortsystemen. Die Komplexität der Fahrzeugelektronik ist somit zunehmend. Die damit verbundenen Baugruppen, beispielsweise die Kraftfahrzeugbatterie, erfahren eine wachsende Beanspruchung. Eine Fahrzeugbatterie muss mehr Verbraucher mit Energie versorgen und ist zu einem noch wichtigeren Bestandteil des Kraftfahrzeugbaus geworden. Ein zuverlässiger Energiespeicher ist daher unerlässlich, um einen störungsfreien Betrieb des Automobils zu gewährleisten [2].

Die Daten der Pannenstatistik zeigen, dass im Zeitraum 2008 bis 2013 keine signifikanten Verbesserungen der Elektronikschwierigkeiten in der Automobilbranche erzielt werden konnten. Die Anzahl an ausgefallenen Fahrzeugen aufgrund eines Batteriedefekts verharrt weiterhin auf konstant hohem Niveau. Die Politik hat darüber hinaus die Weichen in Richtung E-Mobilität gestellt. Im nationalen Entwicklungsplan Elektromobilität des Bundesverkehrsministeriums werden bis zum Jahr 2020 eine Million Elektroautos auf deutschen Straßen angestrebt. Zielsetzung der Bundesregierung ist es hierbei, dass

mindestens die Hälfte dieser Fahrzeuge aus deutscher Produktion stammt. Die Thematik einer verlässlichen Energieversorgung wird somit auch weiterhin von bedeutender Stellung für die inländische Automobilindustrie sein [4].
Deutsche Kraftfahrzeughersteller widmen sich daher der Problematik einer zuverlässigen Energieversorgung. Die Bayrischen Motorenwerke haben beispielsweise in den vergangenen Jahren einen Batteriesensor an der Batterieklemme integriert, um frühzeitig einen Ausfall der Batterie prognostizieren zu können. Hiermit kann aufgrund einer rechtzeitigen Warnmeldung und eines frühzeitigen Batterieaustausches ein Liegenbleiben des Kraftfahrzeuges unterbunden werden [5]. Die Überwachungstiefe dieser Warnsysteme kann in ihrer aktuellen Bauweise nur einen Teil möglicher Batterieausfälle identifizieren, so dass weiterhin mit ausfallenden Fahrzeugen zu kalkulieren ist.
Die beschriebenen Aspekte zeigen, dass es erforderlich ist, die Tiefe der Sensorik weiter zu verbessern. Ziel der vorliegenden Bachelorthesis ist es daher, mittels einer drahtlosen Verbindungsschnittstelle einen Sensor zu entwickeln, der keine räumliche Nähe zur Fahrzeugelektronik benötigt und zusammen mit der Batterie einbaulagenunabhängig verbaut werden kann. Hierdurch soll ein frühzeitiger Wartungseinsatz für einen Batterieaustausch noch genauer prognostiziert und die Zahl an liegengebliebenen Fahrzeugen aufgrund eines Batteriedefekts reduziert werden.

1.2 BATSEN

Im Rahmen des Projektes „BATSEN – Drahtlose Sensoren für Fahrzeugbatterien“ wird der Einsatz von drahtlosen Sensoren in den Batteriezellen untersucht. Das BATSEN Projekt wird vom BMBF gefördert und wird an der HAW Hamburg durchgeführt. Dabei werden Temperatur und Spannung in der Zelle gemessen und versendet. Die Messwerte werden als Datenpakete drahtlos an das Batteriesteuergerät übertragen. Dort werden die Daten der Zellensensoren empfangen sowie der Strom an der Batterie gemessen. Anschließend werden mit diesen Messdaten Lade- und Alterungszustand jeder einzelnen Zelle bestimmt. Dazu sollen technologie- und typenspezifische Verfahren und Modelle zum Einsatz kommen. Die verteilte Messung und Vorverarbeitung der Sensordaten sowie eine zentrale Auswertung aller Messdaten ist ein Gegenstand des Forschungsvorhabens [6].

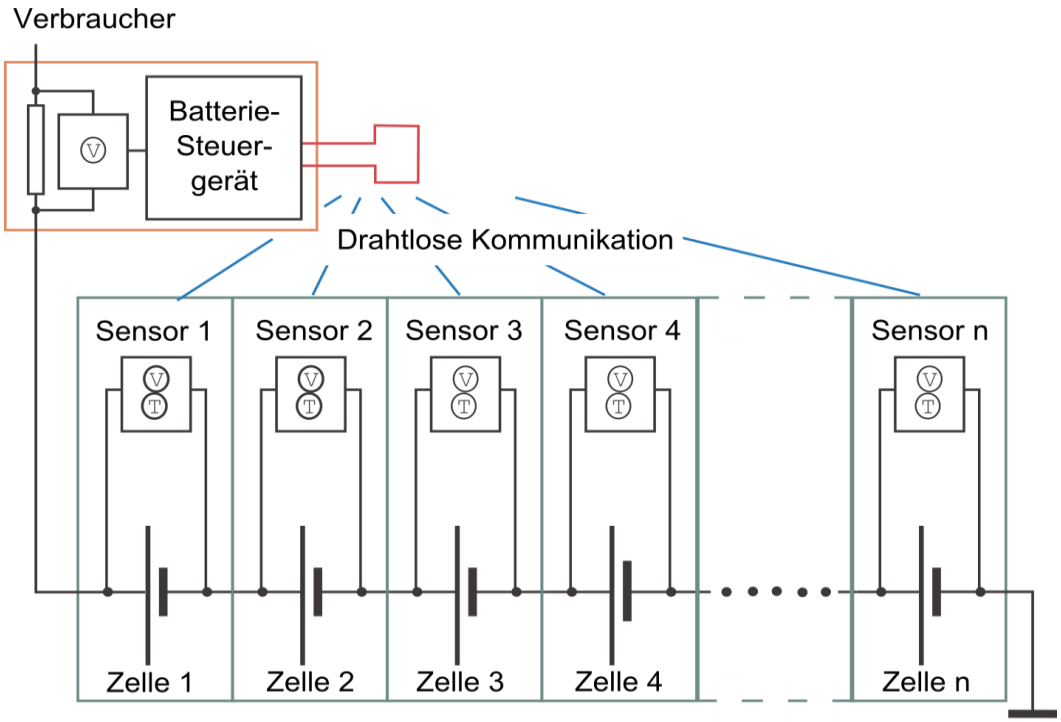

Abb. 1.1 Schematische Darstellung der drahtlosen Zellen-Überwachung

zeigt die schematische Darstellung der drahtlosen Zellen-Überwachung.

Das Steuergerät misst dabei über einen Shunt-Widerstand[1] den Strom der Batterie. Die einzelnen Sensoren messen dabei die Spannung und die Temperatur der jeweiligen Zelle.

1.3 Aufgabenstellung

Ziel dieser Arbeit ist die Hard- und Softwareentwicklung von Drahtlosen Zellensensoren und die Erprobung dieser Sensoren. Es soll ein Sensor mit einer Sicherheitsschaltung entwickelt werden, welche den Sensor gegen Verpolung, Kurzschluss und Überspannung schützt. Folgende Schritte sollen dabei durchgeführt werden: Auswahl der Bauelemente, Erstellung eines Schaltplans sowie Erstellung eines Platinenlayouts. Es ist eine Kleinserie von sechs bis zwölf Sensoren in Betrieb zu nehmen. Die Störeinflüsse des Spannungswandlers sowie die Messauflösung sind zu untersuchen. Es gilt die Messgenauigkeit und den Eingangsspannungsbereich, insbesondere im Zusammenwirken mit der Sicherheitsschaltung zu testen. Die Sensoren sind in der Spannungs- und Temperaturmessung zu kalibrieren. Ein Warteschlangenverfahren ist in die Sensorsoftware zu integrieren. Messwerte sollen im Sensor zwischengespeichert werden. Ein Zeitstempel-Verfahren für die Rekonstruktion der Messzeitpunkte ist zu

[1] Der Shunt-Widerstand ist ein niederohmiger elektrischer Widerstand und dient zur Strommessung.

implementieren. Eine Erprobung der Sensoren am Fahrzeug ist durchzuführen. Die Aufgaben dieser Arbeit sind sinngemäß aus der Aufgabenstellung im Anhang A entnommen. Details sind dort beschrieben.

2 Hardware

Aus der Aufgabenbeschreibung für den Sensor geht hervor, dass ein Ultra Low Power Mikrocontroller der Firma Texas Instruments und ein quarzloser Transmitter der Firma Silicon Labs verwendet werden sollen. Zusätzlich sind die Sensoren für eine höhere Spannung von bis zu 5 Volt auszulegen, um sie ebenfalls für andere Batteriezellen als die Bleisäurebatteriezelle verfügbar zu machen.
Der zu entwickelnde Sensor sollte die Maße des aus der Bachelorarbeit von Ilgin [7] entstandenen Sensors einhalten. Die Platinengröße ist somit auf 71mm x 20mm beschränkt. Zusätzlich ist zu der Platinengröße auch die Kompatibilität zu dem im Zuge des BATSEN Projektes entwickelten Programmieradapters einzuhalten.

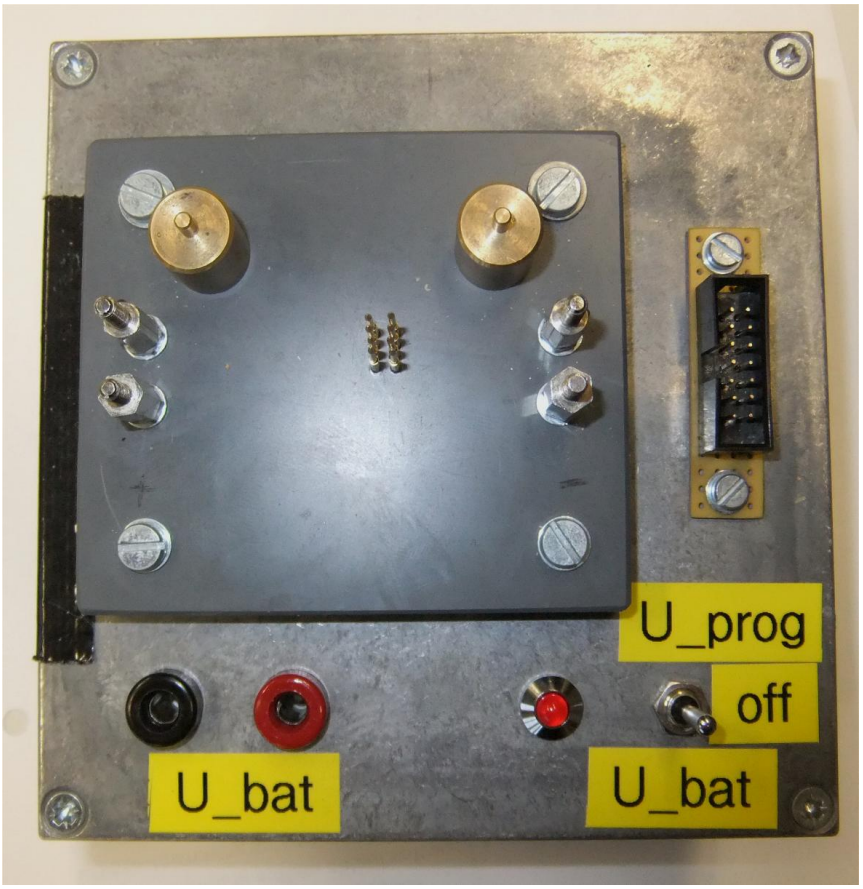

Abb. 2.1 Programmieradapter der BATSEN Projektes

Abb. 2.1 zeigt den Programmieradapter des BATSEN Projektes. Dieser dient der Programmierung der Sensoren. Die Nadelstifte leiten Takt- und Informationssignale des Programmiergeräts an den Mikrocontroller weiter. Mit dieser Konstruktion ist ein schneller Wechsel der Platinen möglich und es lassen sich Wannenstecker auf der Platine einsparen. Die Position der Programmier-Pads ist somit fest vorgegeben.

Für die Bauteile kamen nur SMD-Bauteile in Betracht, da sich diese durch ihre kleine Bauform und durch ihren geringen Energiebedarf auszeichnen.

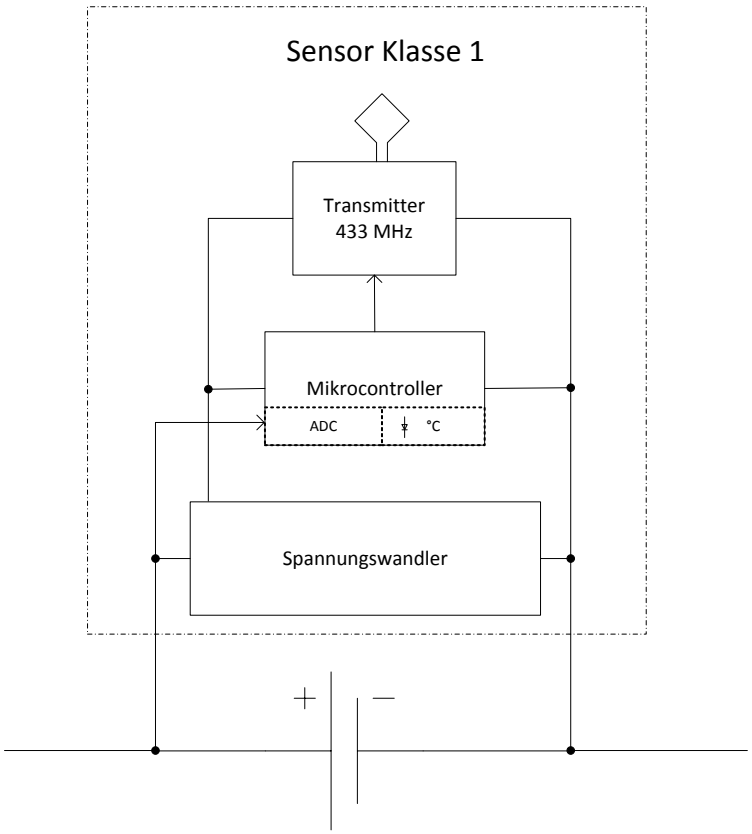

Abb. 2.2 Sensormodell

Die Abb. 2.2 zeigt das Sensormodell. Der Spannungswandler dient der Anpassung der Zellenspannung an die Versorgungsspannung der elektrischen Bauteile.
Der Mikrocontroller misst die Zellenspannung sowie die Temperatur. Diese Messdaten werden dann an den Transmitter übergeben um sie drahtlos zu übertragen.

2.1 Mikrocontroller

Als Mikrocontroller für den Sensor wurde der MSP430G2553 von TI ausgewählt.
Die MSP430 Serie ist für Anwendungen mit niedrigstem Stromverbrauch optimiert. Der ausgewählte Typ MSP430G2553 zeichnet sich durch folgende Merkmale aus:

- 16 Bit Mikrocontroller
- 512 Byte RAM
- 16 kByte Flash
- I²C Bus

- mehrere externe ADC Eingänge
- CPU unabhängiger ADC
- zwei 16 Bit Timer
- interner RC-Oszillator
- interner Temperatursensor
- 10 Bit ADC

Der Mikrocontroller lässt sich hierbei komfortabel über die Joint Test Action Group - Schnittstelle programmieren. Unter Berücksichtigung des einfacheren Layouts wurde die Variante Spy-Bi-Wire [8] gewählt. Es werden in diesem Fall lediglich zwei Signalleitungen vom Programmiergerät benötigt, üblicherweise müsste jedoch 4-Wire JTAG [8] genutzt werden. Die Versorgungspannung und der Massebezug müssen in beiden Varianten zusätzlich mitgeführt werden. Der Nachteil der Spy-Bi-Wire Variante ist die geringe Programmiergeschwindigkeit gegenüber der 4-Wire JTAG Variante [9], jedoch kann dieser Punkt aufgrund der zu erwartenden Programmgröße als vernachlässigbar angesehen werden. Aufgrund der geringen Peripherie, welche an den Mikrocontroller angeschlossen wird, konnte auf die Chipgehäuseform TSSOP mit ihren 20 zu Verfügung stehenden Pins zurückgegriffen werden. Dies stellt die kleinste verfügbare Bauform des Mikrocontrollers dar. Innerhalb der Produktfamilie verfügt dieses über den größten RAM Speicher mit 512 Byte. In der Arbeit von Püttjer [5] war das RAM eine einschränkende Größe, dort betrug der verfügbare RAM Speicher nur 256 Byte. Die Messung der Zellenspannung wurde über einen Spannungsteiler realisiert, damit auch andere Zellen mit einer höheren Zellenspannung verwendet werden können. Für die Eingangsspannungsmessung wurde die interne Referenzspannung von 2,5 Volt des Mikrocontrollers verwendet. Diese wird wahlweise auf 2,5 Volt oder 1,5 Volt stabilisiert.

2.2 Transmitter

Beim Transmitter fiel die Wahl auf den von Silicon Labs gefertigten Typ SI 4012. Dieser zeichnet sich insbesondere durch seinen quarzlosen Betrieb aus. Somit lässt sich der Einsatz eines externen Quarzes vermeiden, was eine entscheidende Kostenersparnis ermöglicht. Dieses Sende-IC wurde schon in der Bachelorarbeit von Ilgin [7] verwendet. Der Sende-IC SI 4012 lässt sich komfortabel über den I²C-Bus ansteuern. Der Transmitter ist für die Anwendung in Batterieapplikationen entwickelt und optimiert worden. Er beherrscht die Modulationsart FSK sowie die in dieser Arbeit verwendete Modulation OOK. Der gewählte Transmitter besitzt eine integrierte, automatische

Antennenabstimmung. Der Transmitter kann zudem im Betrieb abgeschaltet werden, um Energie zu sparen. Die gedruckte Antenne von dem von Ilgin [7] entwickelten Sensor sollte übernommen werden, da mit dieser bereits Erfahrungen gesammelt werden konnten. Die Antenne wurde aus einem Datenblatt entnommen [10] und für die hier verwendeten Sensoren verkleinert.

2.3 Spannungsversorgung

Für die Spannungsversorgung der Elektronik des Sensors wurde der Spannungswandler vom Typ TPS61201 von TI gewählt. Dieser Boost-Converter[2] arbeitet gemäß Datenblatt mit einer hohen Effizienz von bis zu 90 % [11]. Sein Eingangsspannungsbereich reicht von 0,3 Volt bis zu 5,5 Volt. Dabei kann er autonom zwischen den Betriebsarten Boost Mode und Down Conversion Mode umschalten, um dauerhaft die voreingestellte Ausgangsspannung von 3,3 Volt bereitzustellen. Der Spannungswandler besitzt einen Schutz gegen ausgangsseitigen Kurzschluss. Zum Zweck der Energieeinsparung und der Verringerung der Störungen auf die Eingangsspannung lässt sich der Spannungswandler im laufenden Betrieb abschalten.

Der Spannungswandler erzeugt durch seine Funktionsweise Störungen auf der Eingangsspannungsseite. Diese Störungen entstehen durch die von ihm erzeugten Schaltvorgänge. Die Messung der Zellenspannung ist durch diese Einflüsse gestört. Eine Möglichkeit diese Störungen zu minimieren ist darin gegeben, die Spannungsversorgung für den Zeitpunkt der Messung abzuschalten. Die Abschaltung der Spannungsversorgung gestaltete sich als schwierig, um die Funktion des Sensors im weiteren Verlauf nicht zu beeinträchtigen. Für die Abschaltung und Reaktivierung müssen bestimmte Zeiten eingehalten werden, damit die über Kondensatoren gepufferte Spannung nicht zusammenbricht und die Störungen vom Spannungswandler keine Auswirkungen mehr zeigen. Der Mikrocontroller schaltet bei dem Wegfall seiner Versorgungsspannung ab und seine GPIO-Pins kehren in Folge dessen in den hochohmigen Zustand zurück. Die Art der Abschaltung der Spannungsversorgung wurde über einen Pull-Up Widerstand realisiert, dafür schaltet der Mikrocontroller den entsprechenden Pin auf Massepotential. Bei der Rückkehr in den hochohmigen Zustand wirkt dieses Potential wieder gegen die Zellenspannung und schaltet den Spannungswandler wieder zu. Der Mikrocontroller würde dann, bedingt durch den Neustart, alle vorherigen Messwerte verlieren. Die Abschaltung der

[2] Boost Converter – Aufwärtsregler, Gleichspannungswandler

Spannungsversorgung wurde im Rahmen dieser Bachelorarbeit nicht weiter betrachtet, da die beschriebene Softwarelösung nicht dafür ausgelegt ist. Um die Störungen zu minimieren, wurde die gemessene Zellenspannung über 2 Werte gemittelt und geglättet.

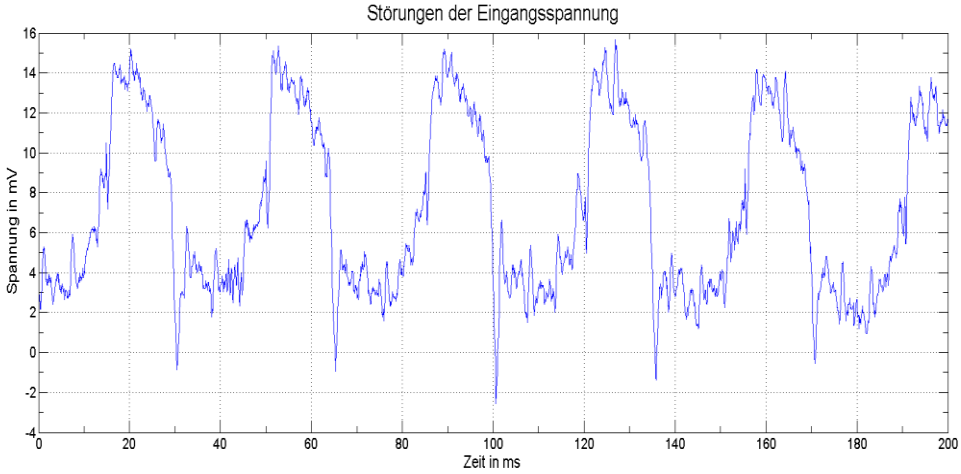

Abb. 2.3 Störungen der Eingangsspannung

Die Abb. 2.3 zeigt den Spannungsverlauf der Eingangsspannung bei aktivem Spannungswandler. Die Spannung wurde mit einem Oszilloskop in der AC-Einstellung gemessen, um den Gleichspannungsanteil auf 0 Volt zu bringen. Die Störungen im Wechselspannungsbereich erreichen einen Spitzenwert von bis zu 16 mV.

2.4 Sicherheitsschaltung

Gemäß der Aufgabenstellung ist ein Verpolungs-, Kurzschluss- und Überspannungsschutz wünschenswert.

Der Verpolungsschutz des Sensors wird durch die Diode und zusätzlich durch die erste der beiden Sicherungen gewährleistet wie in Abb. 2.4 gezeigt. Bei der Diode handelt es sich um eine Suppressordiode[3] vom Hersteller Vishay. Sie ist in Sperrrichtung vom Pluspol zum Minuspol der Zelle geschaltet und ist für eine Versorgungsspannung von bis zu 6,5 Volt [12] vorgesehen. Diese Dioden dienen zum Schutz von empfindlichen Bauteilen und Signaleingängen vor einmaligen, zeitlich begrenzten Überspannungen, wie z.B. ESD Ereignissen.

Bei Verpolung des Sensors vom Minuspol zum Pluspol der Zelle, wird die Diode leitend und der dabei entstehende Strom wird über die Diode und die erste Sicherung abgeleitet.

[3] Suppressordiode , Transient Voltage Suppressor Diode (TVS Diode)

Erreicht der Strom in einem Fehlerfall den Nennstrom von 250 mA der Sicherung, so trennt diese den Sensor von der Batteriezelle. Damit wird gewährleistet, dass die Batteriezelle intakt bleibt. Der Sensor hingegen ist erst nach einem Sicherungsaustausch wieder einsatzfähig.
Bei der Entwicklung dieser Schaltung war der Schutz der Batteriezelle vorrangig. Ein Ausfall der Zelle hätte weitreichendere Folgen als ein ausgefallener Sensor. Eine defekte Zelle kann zu einem Ladungsungleichgewicht führen und die Funktion der Batterie nachhaltig zum Negativen beeinträchtigen. Ein Ausfall der Batterie könnte zudem zu einer Schädigung der angeschlossenen Elektronik führen.

Die zweite Sicherung dient dem Schutz der Schaltung bei Überspannung. Die Diode benötigt für die Ableitung dieser eingangsseitig auftretenden Spannung eine gewisse Zeit. Der Mikrocontroller misst, bedingt durch die Funktion der Überwachung der Zellenspannung, direkt hinter den Sicherungen. Im Fehlerfall würde sonst der Mikrocontroller zerstört oder aber leitend werden und als dauerhafter Verbraucher innerhalb der Zelle fungieren.

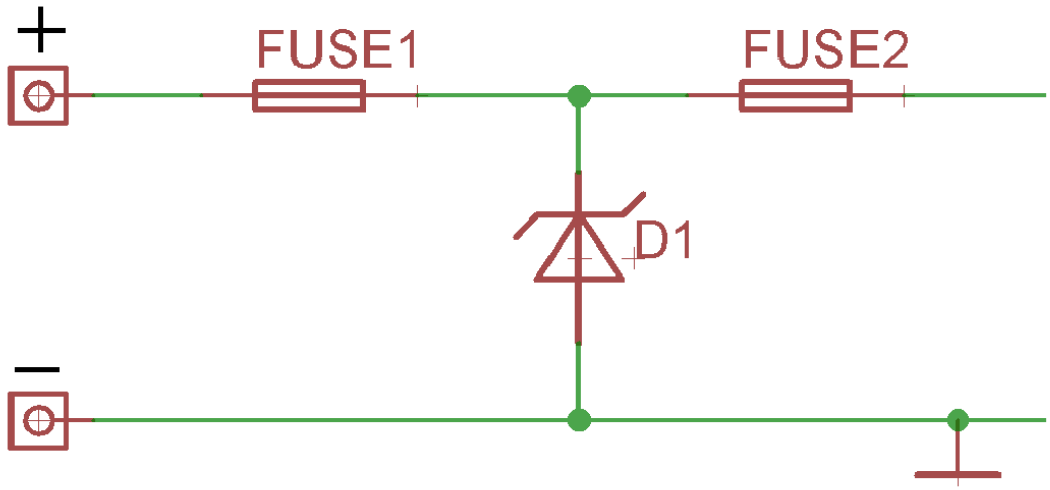

Abb. 2.4 Sicherheitsschaltung des Sensors

Die Abb. 2.4 zeigt die für diesen Sensor entworfene Sicherheitsschaltung, bestehend aus SMD Sicherungen sowie der Supressordiode.

Die beiden in Reihe geschalteten Sicherungen haben ebenfalls negative Einflüsse auf die Schaltung. Die ersten für den Sensor verwendeten Sicherungen hatten einen

Innenwiderstand von 2,8 Ohm [13]. Dieser hohe Innenwiderstand schränkt den Eingangsspannungsbereich des Spannungswandlers sehr stark ein. Die untere Spannungsgrenze von 0,3 Volt wurde mit diesen Sicherungen auf 1,4 Volt erhöht. Unterhalb dieser Spannung war der Transmitter und Mikrocontroller noch aktiv aber nicht in der Lage, Datenpakete an das Steuergerät zu senden. Das Steuergerät zeigte zwar einen sehr schwachen Empfang an, jedoch war keines der Datenpakete gültig, weder in der Länge noch in der CRC-Prüfsumme. Diese untere Spannungsgrenze wurde für den Einsatz an einer Batteriezelle als zu hoch eingestuft.

Die Erkenntnisse über die untere Spannungsgrenze führten zu einer Bestellung von neuen Sicherungen des Herstellers Littlefuse mit einem deutlich geringeren Innenwiderstand. Dieser betrug bei den neuen Sicherungen bei gleichem Nennstrom nur noch 0,54 Ohm [14]. Diese Sicherungen ermöglichten es, den Sensor mit einer minimalen Eingangsspannung von 1 Volt stabil zu betreiben. Zum Vergleich wurde ein Sensor ohne Sicherungen bestückt. Dieser ermöglichte einen stabilen Betrieb bis zu einer auf 0,35 Volt abgesenkten Eingangsspannung.

2.5 Sensorplatine

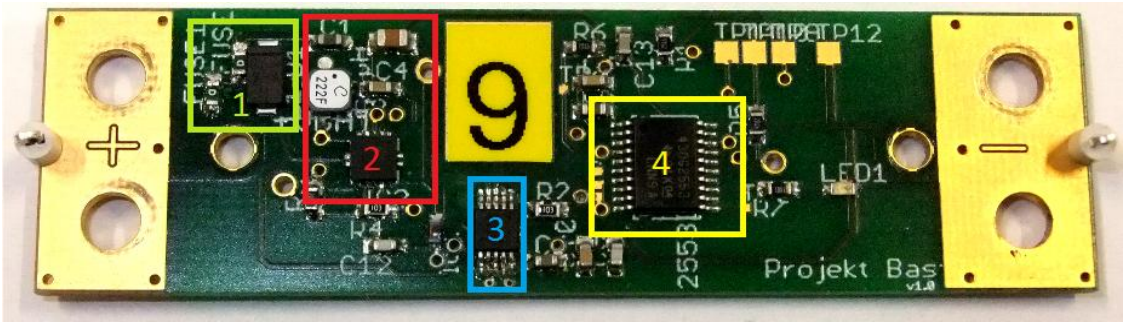

Abb. 2.5 Platinen-Vorderseite des entwickelten Sensors

Abb. 2.5 zeigt einen fertig bestückten Sensor mit Sicherheitsschaltung (1), Spannungsversorgung (2), Transmitter (3) und Mikrocontroller (4).

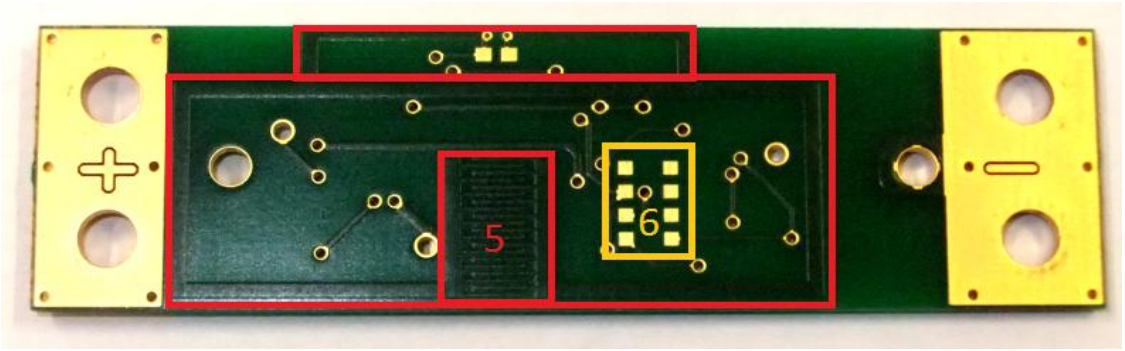

Abb. 2.6 Platinen-Rückseite des entwickelten Sensors

In Abb. 2.6 ist die Platinenrückseite des Sensors mit der Antenne (5) und den Programmier-Pads (6) dargestellt.

Bauteil / Baugruppe	*Position auf dem Sensor*
Spannungsversorgung	1
Schutzschaltung	2
Transmitter	3
Mikrocontroller	4
Antenne	5
Programmier-Pads	6

Tabelle 2-1 Baugruppen Sensor

Der Transmitter wurde aufgrund seiner Funktion in die Nähe des Einspeisepunktes der Antenne platziert. Der Mikrocontroller wurde unweit der Programmier-Pads angeordnet. Die Spannungsversorgung sowie die Schutzschaltung sitzen in Abb. 2.5 im linken Bereich des Sensors.

3 Software

In diesem Kapitel wird die für den Sensor erstellte Software erläutert und auf die Grundfunktionen näher eingegangen. Der Ablauf der Software wurde durch zwei Zustandsautomaten realisiert. Diese beiden Zustandsautomaten beeinflussen sich dabei gegenseitig. Der erste Zustandsautomat beinhaltet die Ansteuerung des ADCs, der zweite Zustandsautomat beinhaltet die Datenaufbereitung und die Steuerung des Transmitters. Die wichtigste Anforderung an die Sensorsoftware war, dass die Zellenspannung alle 100 µs zu messen ist. Alle Aktionen, wie das Messen der Spannung und der Temperatur, Daten aufbereiten und Daten an den Transmitter senden, mussten deshalb innerhalb des 100 µs Timer-Interrupts abgeschlossen werden.

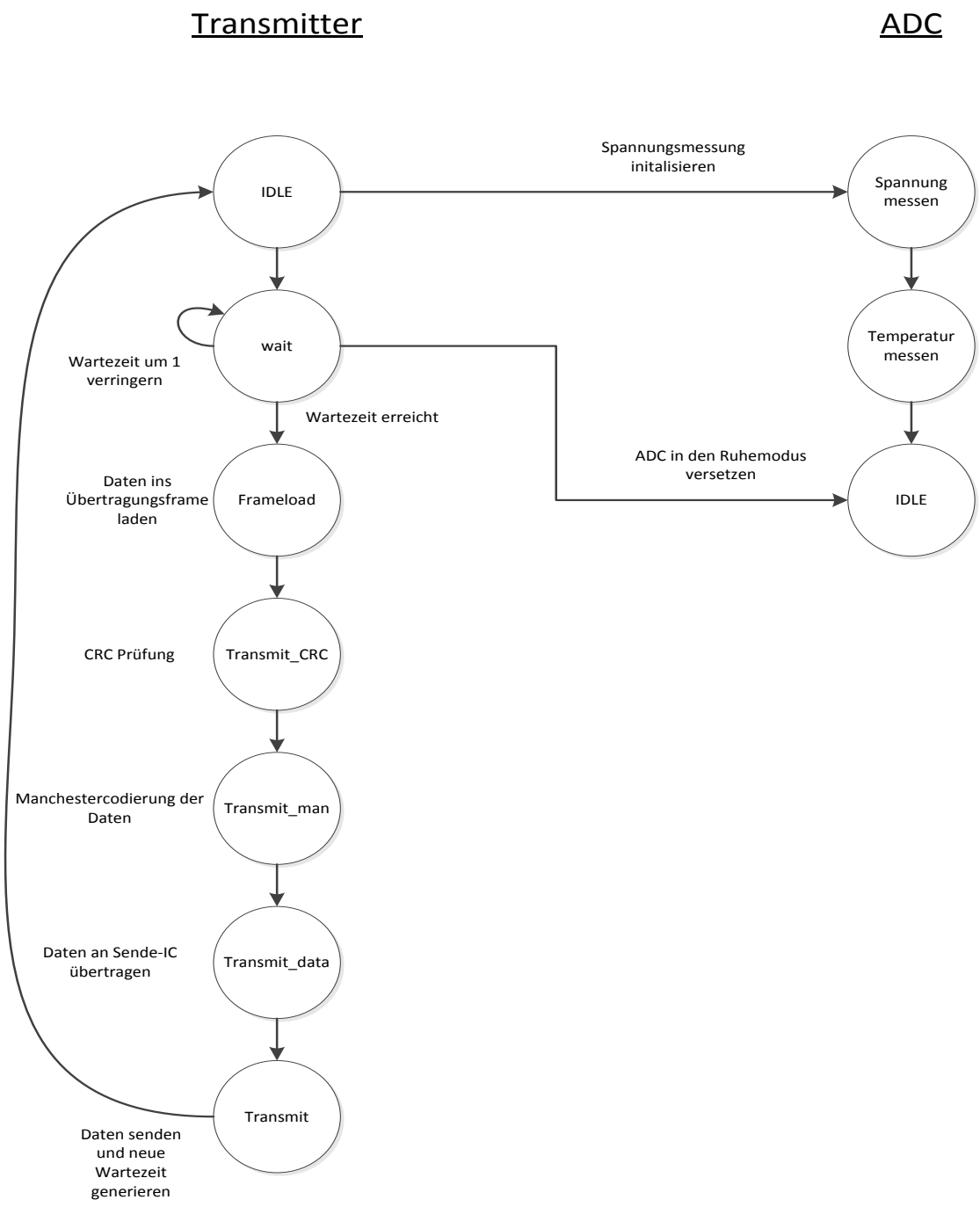

Abb. 3.1 Zustandsautomaten im Timer-Interrupt

Abb. 3.1 zeigt die beiden Zustandsautomaten in dem Timer-Interrupt. Die Funktionen wurden in Transmitter- und ADC-Handling aufgeteilt. Der Transmitter darf hierbei den ADC in den Ruhemodus versetzen.

3.1 Schematische Darstellung der Grundfunktionen

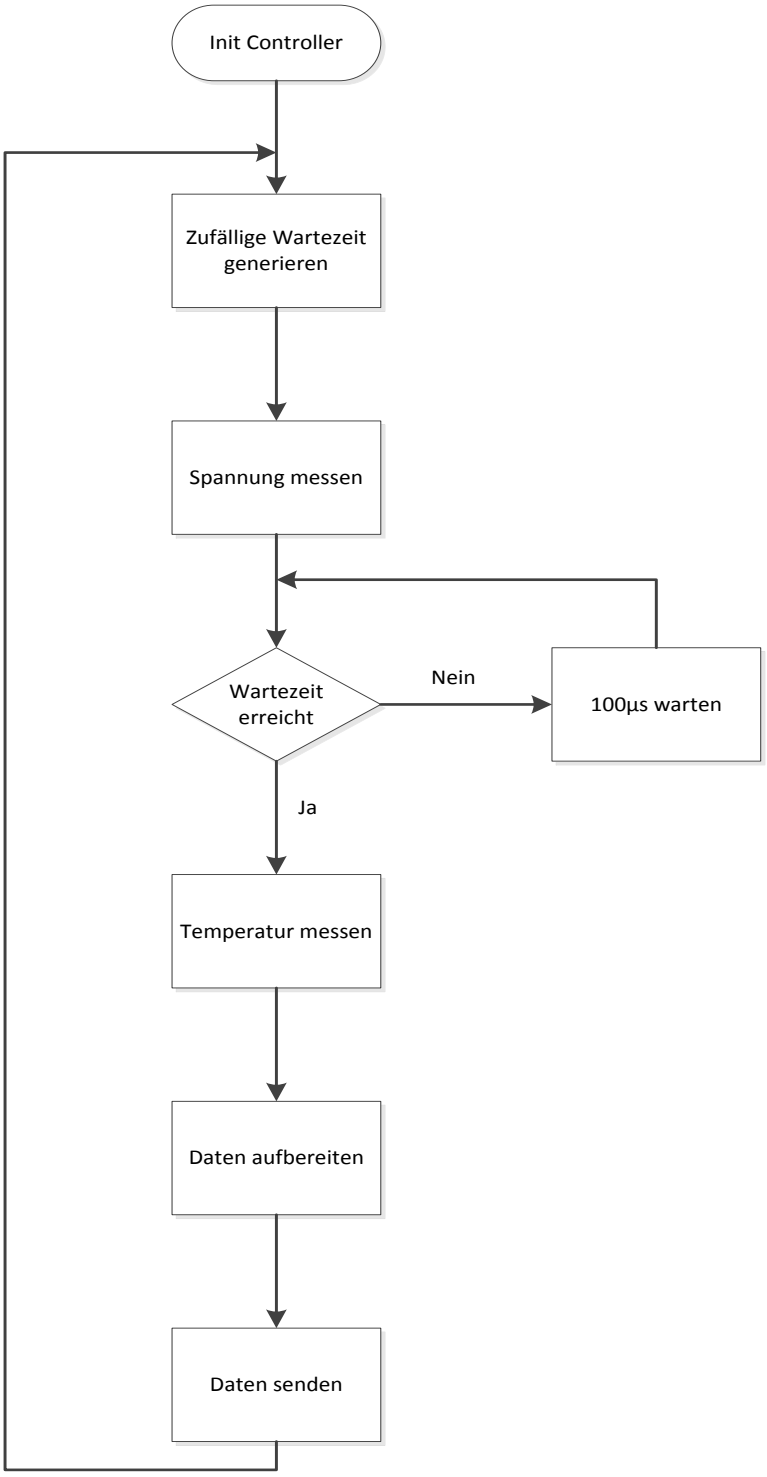

Abb. 3.2 Schematischer Ablauf der Grundfunktionen

Abb. 3.2 zeigt den Ablauf der Grundfunktion des Sensors. Nach der Initialisierung der Funktionseinheiten des Mikrocontrollers wird eine zufällige Wartezeit erzeugt. Im

Anschluss daran wird die Batteriespannung gemessen und nach Ablauf der Messung werden die beiden Werte Spannung und Wartezeit temporär zwischengespeichert. Bei dem Erreichen der zufälligen Wartezeit wird die Temperatur mithilfe des integrierten Temperatursensors gemessen. Im Anschluss an diese Messung werden die Ergebnisse von Spannung, Temperatur und die Wartezeit in einem Übertragungsrahmen gespeichert. Dieser wird nach einer CRC Prüfung nach Manchester codiert wie in den vorherigen Arbeiten von Ilgin [7] und Püttjer [5]. Im Anschluss daran werden diese Daten an den Transmitter über die I²C-Schnittstelle gesendet. Der Transmitter erhält nachfolgend den Sendebefehl und überträgt dann das Übertragungsframe an das Steuergerät. Zeitgleich zum Senden wird eine neue Wartezeit erzeugt und der Ablauf beginnt erneut.

3.2 Grundfunktionen im Einzelnen

3.2.1 Wartezeit

Bei der Erzeugung der zufälligen Wartezeit handelt es sich um ein Schieberegister, welches pseudozufällige Zahlen anhand der Sensor ID erzeugt. Diese Funktion kam schon in den Bachelor- und Diplomarbeiten von Püttjer [5] und Ilgin [7] zum Einsatz. Es werden Zahlen im Bereich von $-2.1475 \cdot 10^9$ bis $2.1474 \cdot 10^9$ generiert.

```
// RND SHIFT MAKRO
#define  SR() sr_r<<=1;sr_r|=(((sr_r >> 28)^(sr_r>>19))&0x00000001);sr_r&=0x0FFFFFFF
// INIT sr_r
long sr_r;
sr_r = (Sensoradresse | 0x08000000);
```

Mithilfe der Rechnung

```
#define TXFAC 3
#define TXMID 5000
time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 2;
```

wird das Ergebnis des Zahlengenerators auf Werte im Bereich von 13336 und 26660 eingegrenzt. Diese Werte werden anschließend, durch die alle 100 µs auftretenden Timer-Interrupts, dekrementiert. Die dadurch entstehen Zeiteinheiten bewegen sich im Mittel von zwei Sekunden. Auf diese Weise steht ein relativ einfaches Verfahren zur Erzeugung von zufälligen Sendezeitpunkten zur Verfügung. Die zwei Sekunden wurden gewählt, damit die Sensoren bei längeren Standzeiten die Batterie nicht zu sehr belasten und dennoch genügend Werte über den Zustand der Zellen liefern.

Der maximale Energieverbrauch des Sensors beträgt ca. 80 mW und teilt sich wie folgt auf:

- Transmitter im Sendevorgang ca. 50 mW [15]
- Mikrocontroller bei der Taktfrequenz 4 MHz ca. 6,6 mW [16]
- Leuchtdiode ca. 6,6 mW

Der Spannungswandler arbeitet mit einer Effizienz von ca. 80 % [11] bei der Eingangsspannung von zwei Volt.

3.2.2 Spannungs- und Temperaturmessung

Der Analog-Digital-Converter des Mikrocontrollers wurde so konfiguriert, dass für die Spannungsmessung die intern stabilisierten 2,5 Volt als Referenzspannungsquelle verwendet werden. Der ADC wurde hardwareseitig mit einem Spannungsteiler versehen, damit dieser in der Lage ist, Spannungen von bis zu 5 Volt zu erfassen. Auf den Vorteil dieser höheren möglichen Eingangsspannung folgt der Nachteil der verringerten Auflösung. Die Maßnahme des Spannungsteilers setzt die Wertigkeit des LSB von 2,44 mV auf 4,88 mV. Die Spannung konnte bedingt durch die Busbreite von 16 Bit und den Zeiteinheiten von 100 µs nicht im Mikrocontroller umgerechnet werden. Die ADC Werte werden daher direkt übertragen und anschließend in dem Steuergerät umgerechnet. Rechnungen mit dem Mikrocontroller auf 32 Bit Breite haben zu Verzögerungen von bis zu 2 ms geführt und somit den Programmablauf stark verzerrt. Die 16 Bit Integer-Multiplikation benötigt auf dem Mikrocontroller 100 Takte [17]. Bei Fließkommaoperationen erhöht sich die Anzahl der benötigten Takte nochmals auf bis zu 400 Takte [17]. Zusätzlich steigt auch der RAM Verbrauch an. Die Rechnung mit Fließkommawerten ist bedingt durch das LSB notwendig, dieses ist nicht ganzzahlig. Die Verschiebung der Nachkommastellen auf ganzzahlige Integer-Werte brachte keinen Erfolg, da die Werte durch die Busbreite von 16 Bit auf maximal 65535 beschränkt sind. Ein Wechsel von 16 Bit Variablen auf 32 Bit Variablen führte erneut zu Verzögerungen im Programmablauf. Die Rechnung mit 16 Bit Festkommawerten wurde nicht weiter untersucht. Die Implementierung der Festkommarechnung hätte zu Rundungsfehlern geführt, weshalb dieser Ansatz nicht weiter verfolgt wurde. Das Rechnen mit ganzzahligen aufgerundeten LSB-Werten hätte zu zusätzlichen Prüfungen geführt, um den Versatz durch das Aufrunden zu korrigieren.

3.2.3 I²C-Bus

Zur Kommunikation zwischen Mikrocontroller und Transmitter dient die I²C Schnittstelle. Diese wurde im Mikrocontroller auf eine Geschwindigkeit von 100 kBit/s konfiguriert. Dabei arbeitet der Mikrocontroller als Busmaster und der Sende IC als Slave. Der Mikrocontroller stößt in diesem Fall die Kommunikation an. Bei der Übertragung der Daten muss der Timer-Interrupt-Betrieb deaktiviert werden. Der Grund sind die Zeitschlitze von 100 µs für die Anforderung die Spannung in diesem Zeitabstand zu messen und der zu übertragenden Daten bei der Geschwindigkeit von 100 kBit/s. Die Anzahl der Daten beträgt 144 Bit pro Frame. In einem Zeitfenster von 100 µs und einer Geschwindigkeit von 100 kBit/s würden maximal 10 Bit übertragen werden können. Der Transmitter erwartet einen vorangestellten Befehl, um die Daten entgegenzunehmen. Dieser Befehl selbst ist 8 Bit breit. Pro Zeitfenster würden somit nur maximal 2 Bit an Nutzdaten übertragen werden können. Um der Verzerrung bei der Wartezeit entgegen zu wirken, werden bei der Datenaufbereitung vor dem Senden die Übertragungszeit der I²C Schnittstelle als fester Bestandteil von 2 ms dazugerechnet. Diese Zeit wurde messtechnisch ermittelt. Die I²C Schnittstelle erwartet zusätzlich vor dem Transfer von Datenpaketen eine Authentifizierung. Der Beginn einer Übertragung wird mit dem Start-Signal vom Master aufgezeigt. Dann folgt die Adresse des anzusprechenden Slaves. Dies wird durch das ACK-Bit vom entsprechenden Slave bestätigt. Der I²C Bus lässt sich auch mit einer Geschwindigkeit von 400 kBit/s betreiben. Tests damit waren nicht erfolgreich, da der Transmitter nach einem einmaligen Senden keine Daten mehr vom Mikrocontroller akzeptierte, weswegen dieser Ansatz wieder verworfen wurde.

3.2.4 CRC Prüfung

Die Daten werden vor dem Versand mit einer zusätzlichen CRC Prüfsumme versehen.
Diese wird im Mikrocontroller mit dem Startwert „0x00“ durchgeführt. Die Daten werden dabei byteweise mit dem Startwert *exklusiv oder* verknüpft. Dies stellt eine einfache Methode da, um Übertragungsfehler zu erkennen. Das Steuergerät führt eine Identische Prüfung durch. Nur wenn das Ergebnis des Steuergerätes korrekt ist, werden die Daten als gültig erkannt. Diese Methode der Fehlerprüfung bietet jedoch keine Fehlerkorrektur der Daten, sondern lediglich eine Prüfung der korrekten Übermittlung.

3.2.5 Manchestercodierung

Der Manchestercode, welcher bereits in den Arbeiten von Püttjer [5] und Ilgin [7] zum Einsatz kam, ist eine Codierung, die das Taktsignal erhält.

Eine wesentliche Eigenschaft dieser Codierung ist die Gleichanteilfreiheit des resultierenden Signals. Der Nachteil dieser Codierung ist allerdings, dass sich bei der Datenübertragung die Bitanzahl verdoppelt und somit auch die Übertragungszeit.

3.2.6 Übertragungsprotokoll

Im Laufe der Arbeit wurde ein neues Übertragungsprotokoll eingeführt, um den Anforderungen der Aufgabenstellung besser gerecht zu werden. Das vorhandene Protokoll besaß eine Datenlänge von 108 Bit und war für die vorgesehene Daten zur Übertragung überdimensioniert.

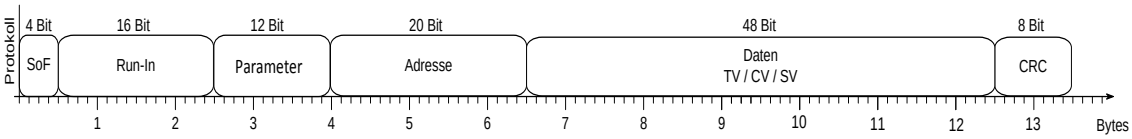

Abb. 3.3 Protokoll aus vorhandenen Arbeiten [5]

Die Abb. 3.3 zeigt das Protokoll, welches in den Vorgängerarbeiten zum Einsatz kam. Dabei wurde immer die Länge des Protokolls beibehalten die Aufteilung der Bits innerhalb des Protokolls hingegen variierte.

Das in dieser Arbeit neu entstandene Protokoll besitzt eine Datenlänge von 72 Bit.

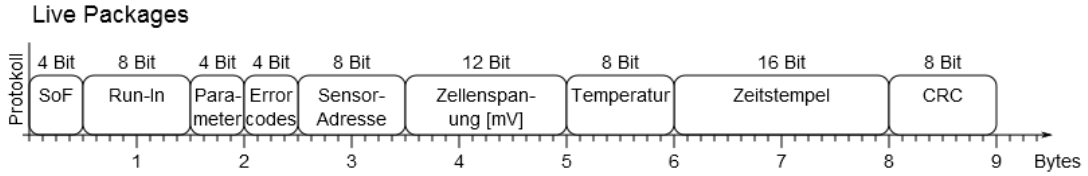

Abb. 3.4 Übertragungsframe Livedaten

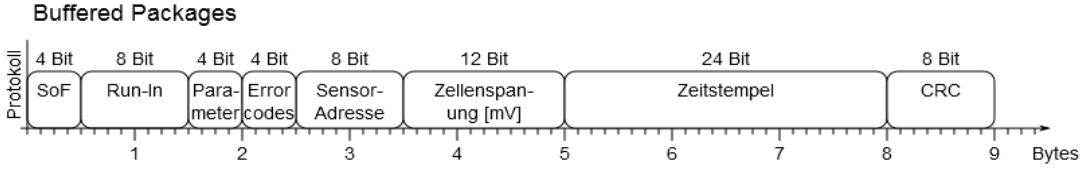

Abb. 3.5 Übertragungsframe Queuedaten

Abb. 3.4 und Abb. 3.5 zeigen die in dieser Arbeit entstandenen Protokolle. Die Einsparung beträgt 36 Bit gegenüber dem bisherigen Protokoll. Die Abb. 3.4 zeigt das Übertragungsframe für den normalen Betrieb, beim zufälligen Senden im Mittel alle zwei Sekunden. Abb. 3.5 zeigt das Übertragungsframe für den Warteschlangen-Betrieb, welcher im vierten Kapitel beschrieben wird.

3.2.7 Aufbau des neuen Protokolls

Bezeichnung	Altes Protokoll	Neues Protokoll	Erklärung
SOF	4 Bit	4 Bit	Start of Frame
RUN-IN	16 Bit	8 Bit	Lernsequenz für Steuergerät
Parameter	12 Bit	4 Bit	Parameterinformationen
Errorcodes	-entfällt-	4 Bit	Fehlerzustände im Sensor
Sensoradresse	20 Bit	8 Bit	Adresse des Sensors
Daten	48 Bit	36 Bit	Mess- und Zeitwerte
CRC	8 Bit	8 Bit	CRC-Prüfsumme

Tabelle 3-1 Übersicht der Protokolle

Tabelle 3-1 Übersicht der Protokolle zeigt den Unterschied zwischen dem vorhanden Protokoll von dem neu implemementieren Protokoll. Der RUN-IN wurde um die Hälfte gekürzt von 16 Bit auf 8 Bit. Die Anzahl der zulässigen Sensor-IDs verringert sich auf 255. In einer 12 Volt Fahrzeugbatterie kommen maximal 6 Sensoren zu Einsatz, somit stehen noch genügend Reserven für andere Batterietypen mit einer höheren Anzahl an Sensoren zur Verfügung. Für Parameter wurden 4 Bit vorgesehen, welche 15 zusätzliche Statusinformationen zulassen. In der Arbeit wurde nur die zusätzliche Information für die Kennzeichnung von Livedaten verwendet, damit die beiden entwickelten Protokollimplementierungen unterschieden werden können. Für Fehlerzustände wurden 4 Bit vorgesehen. Dem Sensor ist es dadurch möglich, bestimmte Zustände über die Zelle oder seine Peripherie dem Steuergerät mitzuteilen. Solche Zustände könnten einen durchgeführten Software-Reset oder die Abschaltung des Transmitters nach dem Erreichen einer kritischen Minimalspannung aufzeigen. Für den Spannungswert wurden 12 Bit reserviert. Hiermit wurde der Anforderung Rechnung getragen, dass die

Spannungswerte direkt im Sensor umgerechnet werden. Durch die 12 Bit wären 4.096 Werte in Millivolt ausgedrückt möglich. Die Parameterbits könnten diesen Bereich noch weiter vergrößern. Für den Temperaturwert wurden 8 Bit vorgesehen. Die zufällige Wartezeit wird mit 16 Bit übertragen. Auf diese Weise lässt sich eine Zeitdifferenz von bis zu 6 Sekunden übermitteln. Die Zeitdifferenz dient der Rekonstruktion des Messzeitpunktes zu dem Sendezeitpunkt. Das Steuergerät kann aus dem Zeitpunkt des Empfangs auf den Zeitpunkt der Messung schließen. Die Zeit, wann der Transmitter die Daten exakt aussendet, ist nicht bekannt. Vor jedem Senden führt dieser einen Abgleich der Antenne durch.

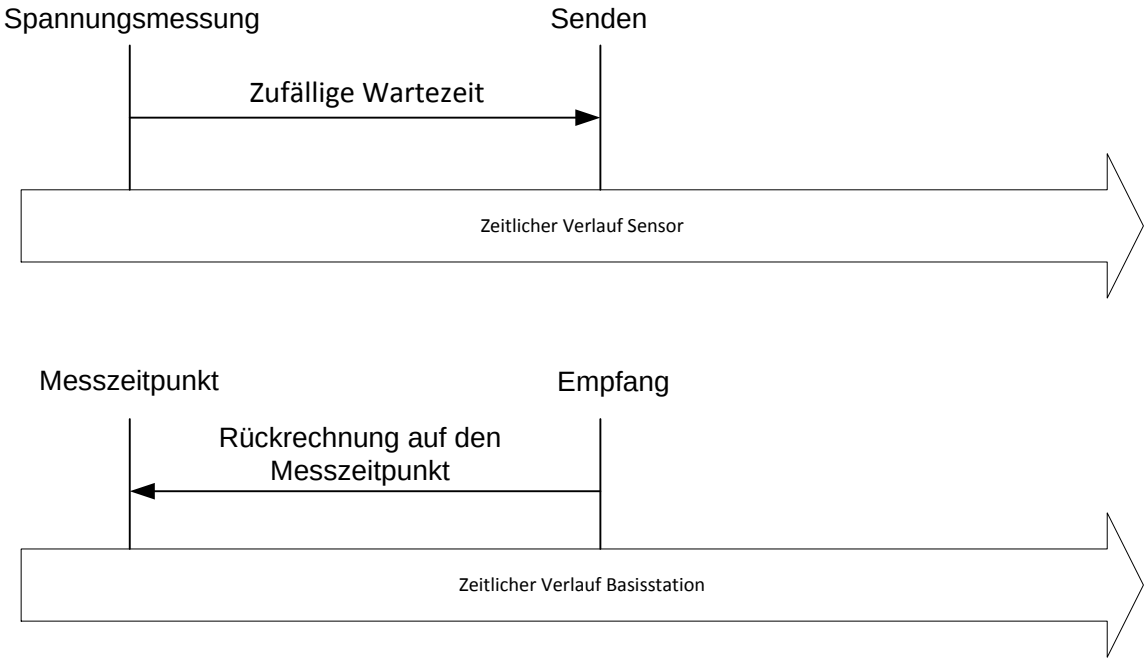

Abb. 3.6 Schematische Darstellung der Rekonstruktion des Messzeitpunktes

In Abb. 3.6 ist der zeitliche Verlauf von Sensor und Steuergerät sowie der Ablauf der Rekonstruktion des Messzeitpunktes dargestellt. Der Sensor misst die Spannung und speichert diese zusammen mit der zufälligen Wartezeit temporär ab. Das Steuergerät kann nach dem Empfang der Daten und ihrer eigenen Zeit auf den Messzeitpunkt rückschließen.

3.2.8 Empfangsroutine in dem Steuergerät

In dem Steuergerät wird beim Empfang erst der Start of Frame auf die korrekte Zeitlänge geprüft. Im Anschluss daran wird mithilfe der RUN-IN Sequenz die Zeitdauer der „0“ und „1“ Übergänge ermittelt und mit einem Aufschlag von ± 25 % versehen. An diese Sequenz folgen die Nutzdaten. Nach dem vollständigen Empfang der Datenbits werden die vorliegenden Daten entsprechend Manchester decodiert. Nach der Decodierung wird die CRC Prüfsumme gebildet. Wenn die Prüfsumme gültig ist, werden die Daten temporär in in einem neuen Frame im Steuergerät gespeichert. Dieses Frame beinhaltet zusätzlich zu den Daten des Sensors wie Spannung, Wartezeit der Sendung, Temperatur und Sensor-ID, noch den Zeitstempel des Empfangs in dem Steuergerät und eine Framenummer, welche die Anzahl aller gültig empfangenen Frames in dem Steuergerät seit dem Start repräsentiert. Im Anschluss an die temporäre Speicherung wird das Frame zur Übertragung an die UART Schnittstelle freigegeben. Kommt es zwischen der Freigabe und der Übertragung an die UART Schnittstelle zu einem erneuten Empfang, so geht das vorherige Frame verloren.

3.3 Messung der Framefehlerrate im Abstand von 2 Sekunden

Der Klasse-1 Sensor verfügt nur über einen Uplink-Kanal zum Steuergerät. Dieser kann somit seine Daten nur senden und hat keine Kenntnis darüber, ob der Übertragungskanal bereits belegt ist. Bei dem Einsatz von 6 Sensoren kommt es zwangsläufig zu Kollisionen auf dem Übertragungskanal. Bei Kollisionen kommt es zu dem unwiederbringlichen Verlust von Messdaten.

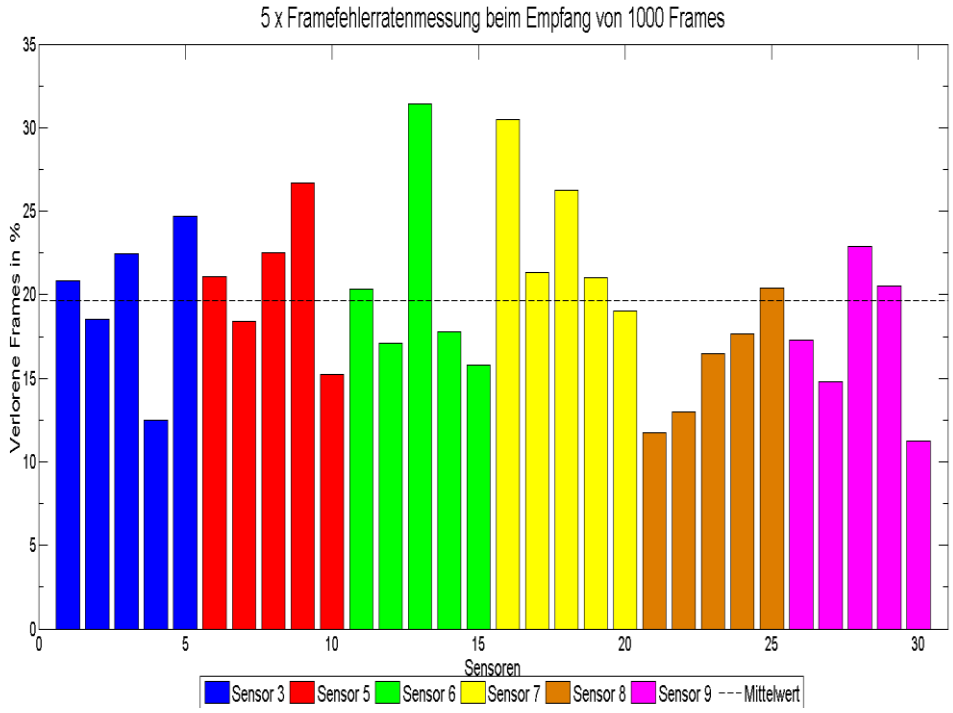

Abb. 3.7 Messung Framefehlerrate Sensoren bei 2 Sekunden Senderate mit 5 maliger Wiederholung

Die Abb. 3.7 zeigt den prozentualen Verlust beim Empfang von 1.000 gültigen Übertragungsframes an. Die prozentuale Ermittlung der Datenverluste wurde fünf Mal wiederholt. Im Anschluss wurde ein Mittelwert über die Verluste berechnet, dieser beträgt 19,64 %.

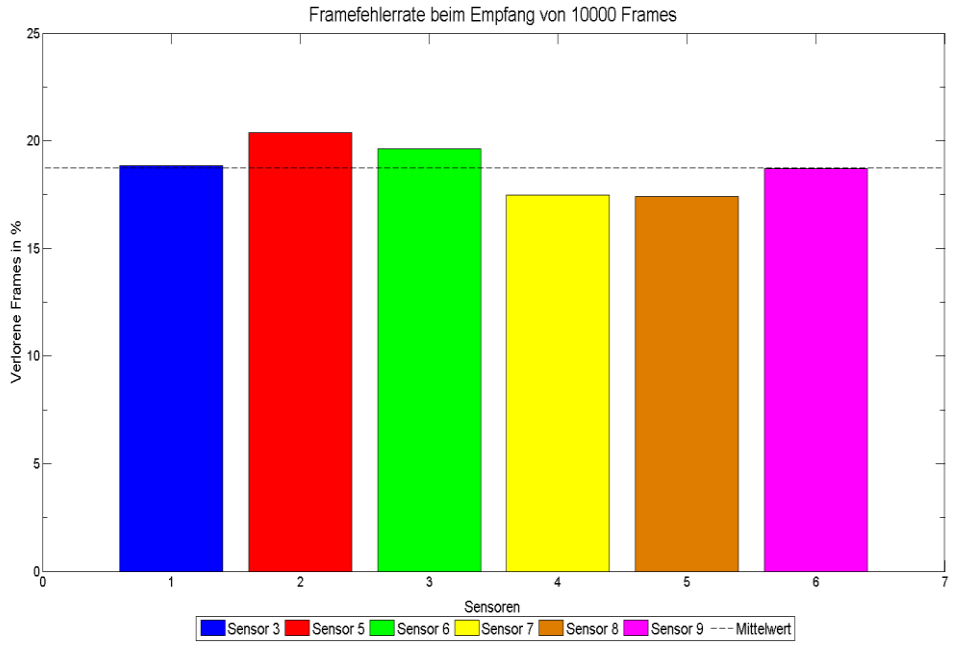

Abb. 3.8 Messung Framefehlerrate Sensoren bei 2 Sekunden Senderate

Bei Abb. 3.8 handelt es sich um die gleiche Messung mit der Maßgabe, dass hierbei einmalig 10.000 zusammenhängende Übertragungsframes ausgewertet wurden. Die Ausreißer von Sensor 6 und Sensor 7 aus Abb. 3.7 mildern sich dabei stark ab. Bei 10.000 Frames beträgt der Mittelwert der verlorenen Frames 18,74 %. Zu den Verlusten bei gleichzeitigem Senden der Sensoren kommt noch der Verlust an Daten durch das Steuergerät selbst. Dieser Verlust ist bedingt durch den Ablauf beim Empfang am Steuergerät und der Weiterleitung der Daten über die UART-Schnittstelle.

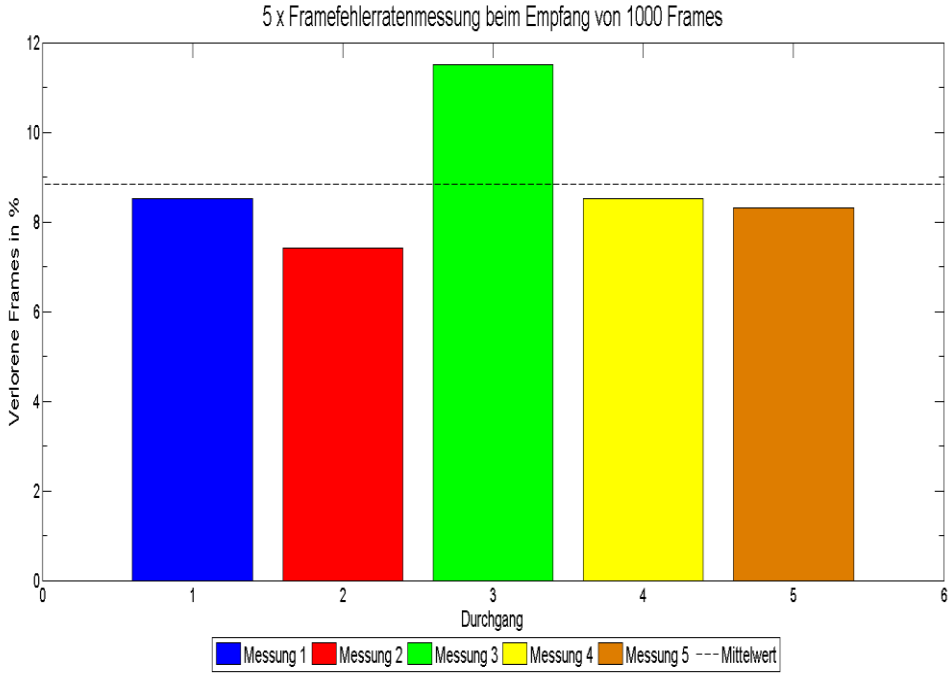

Abb. 3.9 Messung Framefehlerrate Steuergerät bei 2 Sekunden Senderate, mit 5 Wiederholungen

Der Verlust an Daten durch das Steuergerät wird in Abb. 3.9 aufgezeigt. Bei 1.000 Übertragungsframes und fünf Wiederholungen beträgt der Verlust 8,85 %. Bei dem Empfang von 10.000 Datenpaketen beträgt der Verlust 8,82 %.

4 Warteschlangenverfahren

In diesem Kapitel wird das implementierte Warteschlangenverfahren beschrieben. Die Implementierung wurde notwendig, da die zeitliche Erfassung der Zellenspannung alle zwei Sekunden für Hochstromereignisse, wie dem Motorstart, zu ungenau und langsam ist. Ein Motorstart dauert im Normalfall 1 bis 1,5 Sekunden.

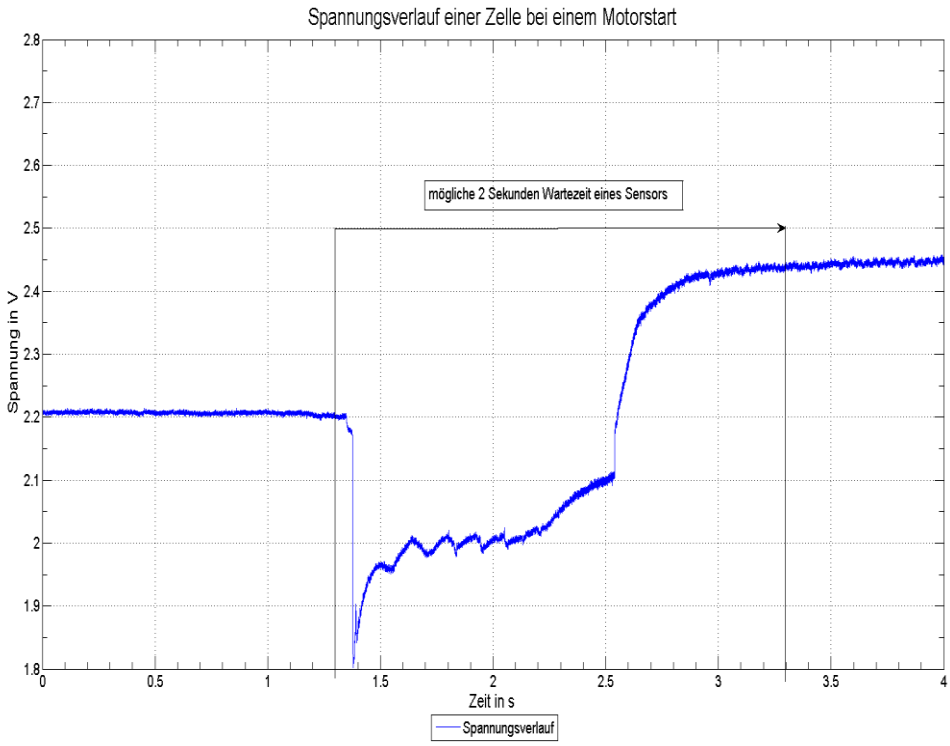

Abb. 4.1 Spannungsverlauf einer Batteriezelle beim Motorstart

Die Abb. 4.1 zeigt den Spannungsverlauf eines Startvorgangs an einem Fahrzeug. In die Abbildung wurde ein mögliches Messintervall des Sensors von zwei Sekunden eingezeichnet. Aus der Abbildung wird deutlich, dass eine Spannungsmessung für kritische Ereignisse wie dem Motorstart im Mittel von zwei Sekunden zeitlich zu ungenau ist. Im schlimmsten Fall würde ein Sensor ein Hochstromereignis kaum detektieren können. Der Startvorgang selbst dauert kaum länger als zwei Sekunden. Allerdings ist der Startvorgang relevant für die Bestimmung des Zellenzustandes. Aufgrund des Umstandes der zeitlich zu geringen Erfassung der Zellenspannung, wurde ein Warteschlangenverfahren in die Sensorsoftware implementiert.

4.1 Ablauf Warteschlangenverfahren

Im Anschluss an die Spannungsmessung wird der gemessene Wert mit dem vorherigen Messwert verglichen. Kommt es bei dem Vergleich zu einem Überschreiten einer vorher festgelegten Spannungsdifferenz, wird von einem Hochstromereignis ausgegangen. In diesem Fall wird die Messwertaufnahme erhöht, um das Ereignis gut abbilden zu können. Im Zuge der Erprobung hat sich eine Differenz von 8 LSB, ca. 40 mV, als zuverlässig erwiesen. Dieser hohe Wert ist bedingt durch die Störungen des Spannungswandlers. Ein zu niedrig eingestellter Wert würde durch die Störungen des Spannungswandlers ein dauerhaftes enqueuing auslösen.

Die Größe der Warteschlange ist begrenzt durch den RAM-Speicher des Mikrocontrollers. Der Controller verfügt über 512 Byte RAM, hiermit konnte eine Warteschlangenlänge von 65 Werten realisiert werden. In der Warteschlange werden der Spannungswert und der zugehörige Zeitwert gespeichert. Der Zeitwert wird in 24 Bit aufgelöst, wodurch sich versetzte Zeitpunkte von dem Speichern in der Warteschlange bis zum Senden dieses Wertepaares von bis zu 27 Minuten ergeben. In dieser Konfiguration müssen 36 Bit pro Messwert gespeichert werden. Die Messwerte und der Zeitwert werden in einer Struktur von 5 x 8 Bit Integer-Variablen gespeichert. Durch die Architektur des Mikrocontrollers müssen allerdings 40 Bit reserviert werden.

Die vier zusätzlichen Bits lassen sich für weitere Informationen zu dem Messwert nutzen. Diese Informationen könnten aus dem niedrigsten Spannungswert in der Queue oder aber einer Sendewiederholung eines markanten Wertes bestehen. Mithilfe von Bitshifting[4] könnte die Länge der Queue um mindestens 7 zusätzliche Werte erhöht werden. In dieser Art der Implementierung würden die vier frei bleibenden Bits für zusätzliche Informationen verloren gehen. Im Enqueue-Fall wird neben der erhöhten Messwertaufnahme auch die Senderate der Übertragungsframes erhöht. Dies wurde durch die Senkung der Wartezeit realisiert. Die Wartezeit wurde von zwei Sekunden auf 500 ms gesenkt. Die erhöhte Senderate dient dem schnelleren Abbau der Queue. Bei einer Senderate von zwei Sekunden würde der Sensor nach einem Enqueue-Fall die Warteschlange nur sehr langsam bis niemals abbauen können. Die Warteschlange wurde als Ringspeicher implementiert.

[4] Bitshifting – bitweise Schiebeoperation

Variablenbreite	Timerticks	Millisekunden	Sekunden
16 Bit	65535	6553	ca. 6,5
20 Bit	1048575	104857	ca. 104
24 Bit	16777215	1677721	ca. 1667

Tabelle 4-1 Zeit - Repräsentation Timerticks

In der Tabelle 4-1 sind die Anzahl der möglichen Timer-Interrupts bei der entsprechend breiten Zählvariablen aufgezeigt.

4.2 Messung der Framefehlerrate im Abstand von 500 Millisekunden

Die gesenkte Wartezeit führt durch die erhöhte Senderate zu mehr Kollisionen auf dem Kanal. Aufgrund dieser Tatsache wurde die Messung zur Ermittlung der Framefehler aus Kapitel 3 wiederholt.

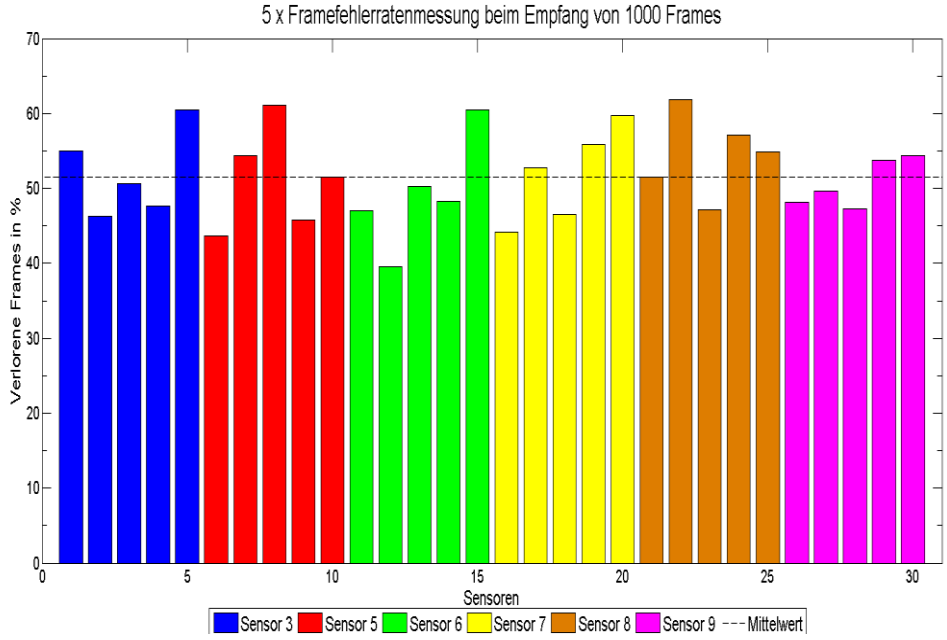

Abb. 4.2 Messung Framefehler Sensor bei 0,5 Sekunden Senderate

Abb. 4.2 zeigt den prozentualen Verlust beim Empfang von 1.000 Übertragungsframes an dem Steuergerät. Die Ermittlung der Datenverluste wurde fünf Mal wiederholt. Im Anschluss wurde ein Mittelwert über die Verluste berechnet. Dieser beträgt 51,55 %.

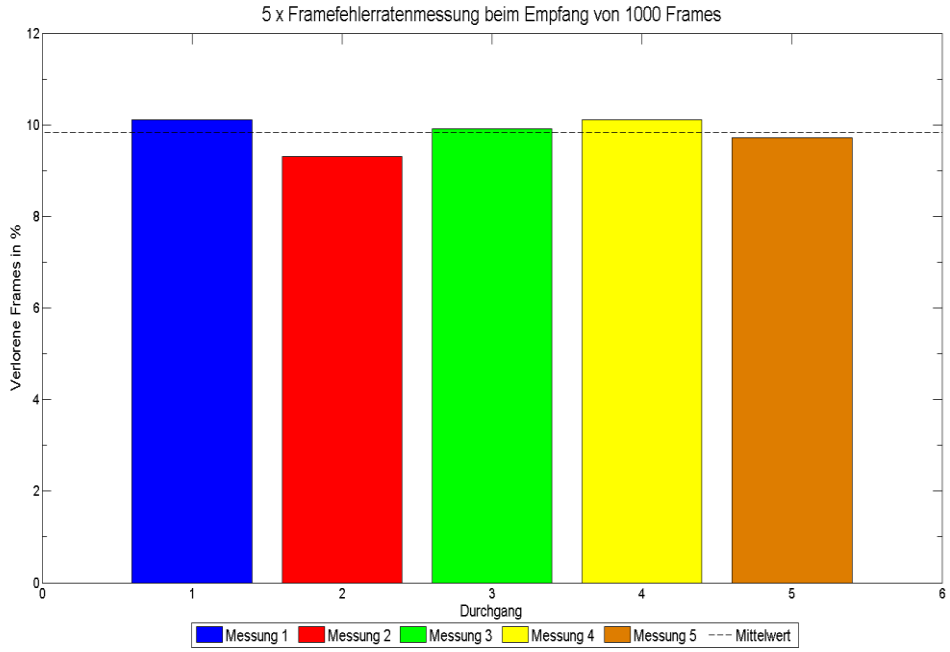

Abb. 4.3 Messung Framefehler Steuergerät bei 0,5 Sekunden Senderate

Abb. 4.3 zeigt den prozentualen Verlust der Übertragungsframes im Steuergerät beim schnelleren Senden der Daten. Der Verlust ist mit 9,9 % gegenüber den 8,85 % beim langsameren Sendemodus kaum gestiegen.

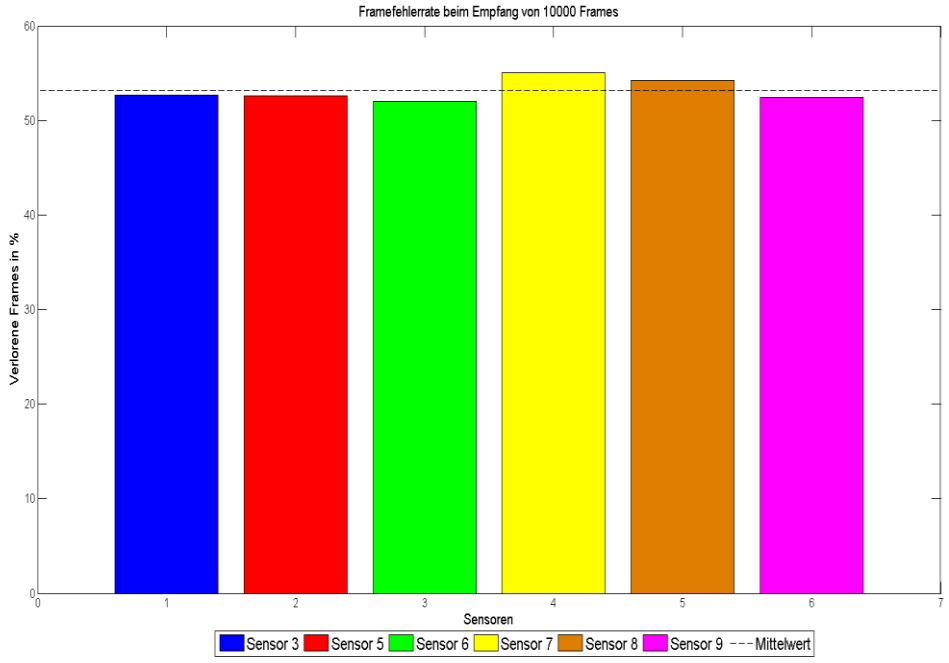

Abb. 4.4 Messung Framefehler Sensor und Steuergerät bei 0,5 Sekunden Senderate

Abb. 4.4 zeigt den Empfang von 10.000 Frames im schnelleren Sendemodus. Der Verlust bei der Übertragung beträgt 53,17 %. und der Verlust des Steuergeräts 10,1 %.

Die Abbildungen zeigen, dass der Verlust an Daten im schnelleren Sendemodus ca. 51 % beträgt. Es geht somit jedes zweite Datenpaket der Sensoren verloren. Die Verluste beim Senden im langsameren Sendemodus betrugen nur ca. 18,74 %.

Im Laufe der Implementierung des Warteschlangenverfahrens sind mehrere Konzepte über die Realisierung der Enqueue-Bedingung entstanden.

4.3 Warteschlangenverfahren Implementierungen

Die erste Realisierung bestand aus einer Überprüfung der Enqueue-Bedingung bei jeder Messung. Für ein erfolgreiches Einspeichern in die Warteschlange musste die Spannungsdifferenz bei jedem Messpunkt erfüllt werden. Der Startvorgang konnte dabei nicht zuverlässig aufgezeichnet werden, entscheidende Werte gingen verloren. Die Gründe dafür waren einerseits die Mittelung der Spannungswerte durch die Software von nicht direkt aufeinander folgenden Zeitpunkten. Die Mittelung sorgt so für eine Glättung des realen Spannungswertes. Ein weiterer Grund für die lückenhafte Aufzeichnung war die nicht zu jedem Prüfzeitpunkt erreichte Spannungsdifferenz. Die Sensorsoftware kehrte somit in den Normalbetrieb zurück.

Das zweite Konzept sah vor, dass bei dem einmaligen Erfüllen der Enqueue-Bedingung eine Fest vorgegebene Anzahl von Messpunkten ohne Prüfung in die Queue gespeichert werden. Diese Art der Implementierung wurde mit nur einem Sensor an dem Zellensimulator getestet. Der Zeitpunkt der Messwertaufnahme variierte ebenso wie die Anzahl der festen Messpunkte. Für Tests an dem Zellensimulator musste die Sensorsoftware erneut angepasst werden. Der Zellensimulator kann maximal 140.000 Spannungspunkte [19] nacheinander ausgeben. Eine zusätzliche Einschränkung hierbei ist das vorgegebene Zeitintervall, in welchem es immer 20.000 Spannungspunkte [19] pro Sekunde sein müssen. Es ergibt sich eine theoretisch maximale Simulationsdauer von sieben Sekunden. Im Anschluss an diese Ausgabe besteht die Möglichkeit, dass der Zellensimulator entweder in seine Ausgangslage von 0 Volt zurückkehrt, das Signal kontinuierlich wiederholt oder einen definierten Spannungspegel ausgibt. Die maximale Simulationsdauer von 7 Sekunden mit anschließender Rückkehr in die Ausgangslage von 0 Volt sind für den Sensor zu kurz, um alle aufgenommenen Messwerte an das Steuergerät zu senden. Innerhalb dieser Zeit können maximal 14 Datenpakete gesendet werden. Der Zellensimulator benötigt nach Ablauf der Simulation eine Umschaltzeit von ca. 5 ms.

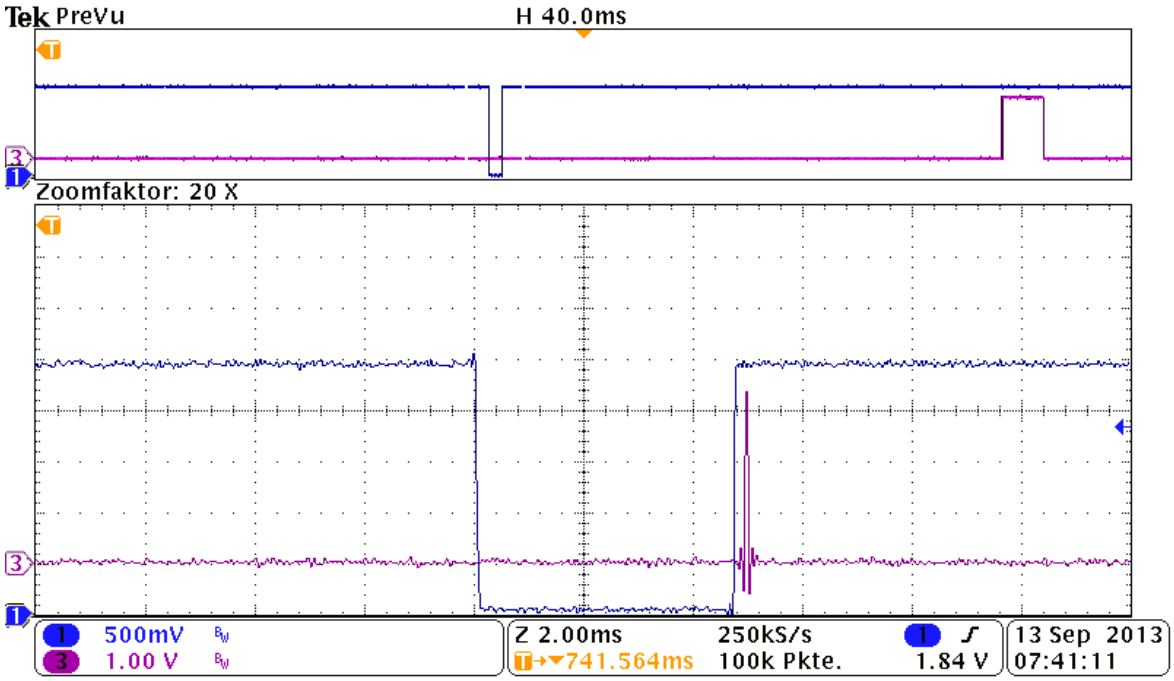

Abb. 4.5 Umschaltzeit des Zellensimulators

Die Abb. 4.5 zeigt den Zeitpunkt der Umschaltung zwischen dem Simulationssignal und dem festen Spannungswert. Die Zeitspanne von 5 ms ohne Versorgungsspannung ist für den Sensor ein kritischer Zustand. Im Zeitpunkt der Umschaltung muss der Stromverbrauch des Sensors auf ein Minimum reduziert werden, sonst kommt es zu undefinierten Zuständen bei dem Mikrocontroller und bei dem Transmitter. In diesem Zustand sind alle vom Transmitter gesendeten Pakete unvollständig und werden von dem Steuergerät nicht akzeptiert. Dieser Zustand lässt sich nur durch einen Reset der Hardware des Sensors beheben.

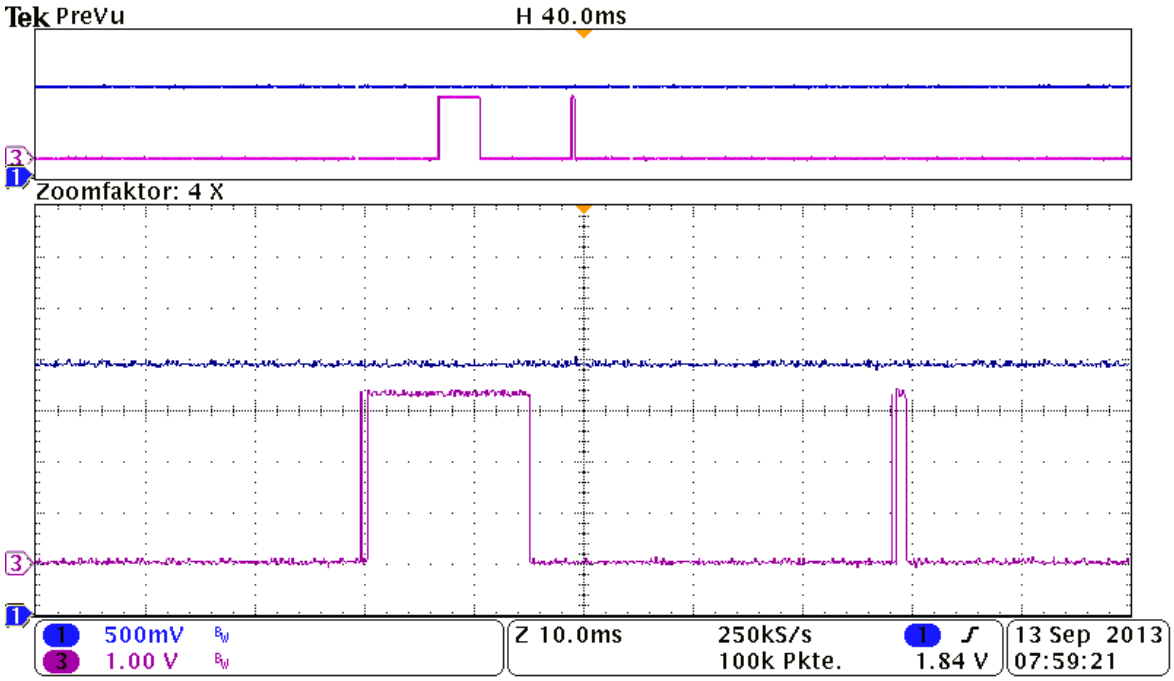

Abb. 4.6 Empfang fehlerhaftes Übertragungsframe

Auf Abb. 4.6 ist der Fehlerzustand des Sensors erkennbar. Das Steuergerät empfängt hierbei nur einen Datenblock. Dieser Datenblock besitzt nach Manchestercodierung keine Informationen und wird durch die Empfangsroutine des Steuergerätes als ungültig erklärt und verworfen.

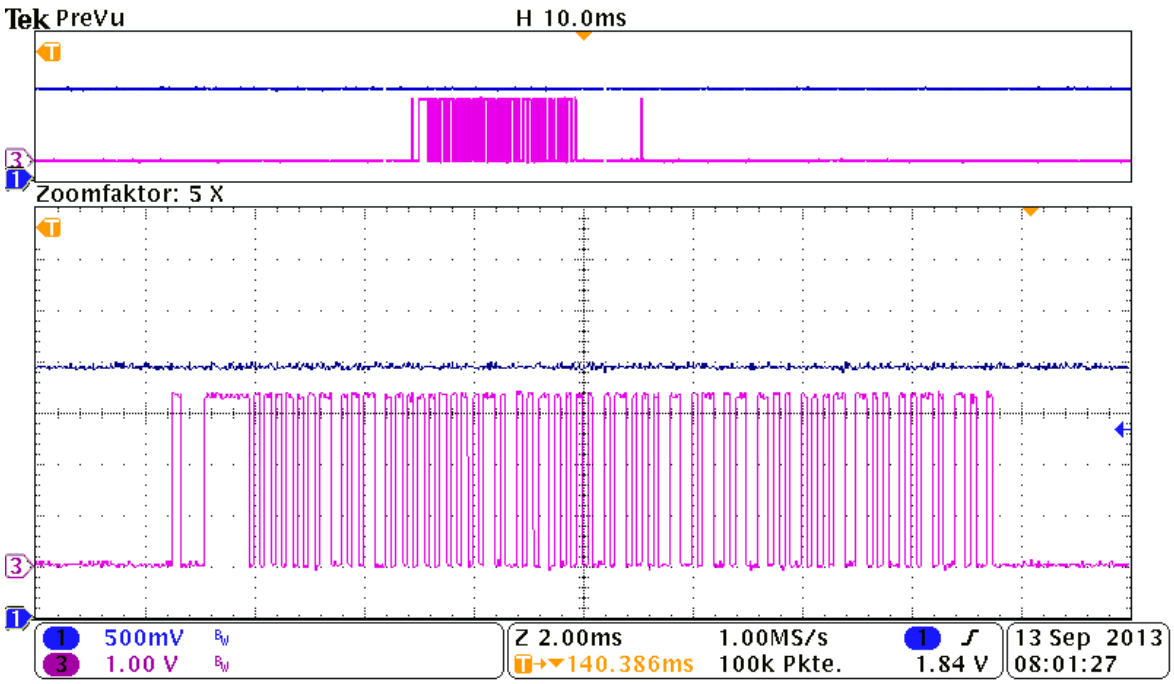

Abb. 4.7 Empfang korrektes Übertragungsframe

Die Abb. 4.7 zeigt den korrekten Empfang des Übertragungsframes am Steuergerät. Deutlich sind der SOF sowie die folgenden Daten des Übertragungsframes zu erkennen. Um den fehlerhaften Zustand und den daraufhin notwendigen Reset mit dem Verlust aller im Mikrocontroller gespeicherten Messwerte zu vermeiden, wurde die Sensorsoftware angepasst. Detektiert die Spannungsmessung einen Spannungspegel von unter einem Volt (ADC Wert 210) wird der Transmitter deaktiviert. Der Sensor kann ab diesem Moment keine Daten mehr senden. Die Messwertaufnahme und das Aufnehmen von Werten in die Warteschlange werden fortgesetzt. Detektiert die Spannungsmessung nachfolgend eine Spannung von mehr als einem Volt wird der Transmitter aktiviert und neu initialisiert.

Die jetzt folgenden Abbildungen von den Erprobungsmessungen wurden nach 100 gültig empfangenen Frames beendet. Außerdem wurde ein Spannungsoffset von 50 mV integriert, um die Spannungsverläufe beider Signale kenntlich zu machen.

	Abstand der Messpunkte in ms	Anzahl der Messpunkte
Enqueue-Bedingung 1	50	30
Enqueue-Bedingung 2	75	30
Enqueue-Bedingung 3	100	30
Enqueue-Bedingung 4	50	60
Enqueue-Bedingung 5	75	60
Enqueue-Bedingung 6	100	60

Tabelle 4-2 Übersicht der erprobten Enqueue-Bedingungen

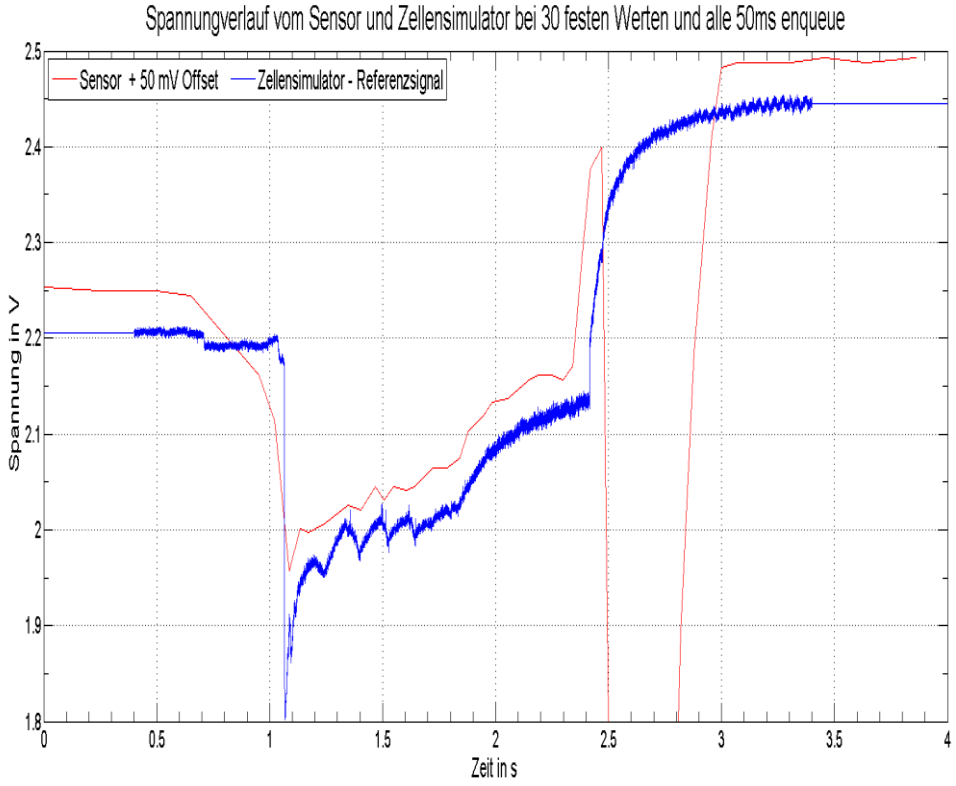

Abb. 4.8 Spannungsverlauf Enqueue-Bedingung 1

Abb. 4.8 zeigt das Aufnehmen von Messwerten bei einem simulierten Startvorgang am Zellengenerator. Nach einmaligem Erfüllen der Enqueue-Bedingung wird alle 50 ms die Spannung gemessen und ohne Prüfung in der Queue gespeichert. Dieser Vorgang wird 30-mal wiederholt. Erst im Anschluss an die 30 Messwerte wird geprüft, ob die Enqueue-Bedingung noch erfüllt wird. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators. Die Messwerterfassung im Abstand von 50 ms führt zu einer hohen Signaltreue zum Referenzsignal. Der Einbruch

ab Sekunde 2,5 ist bedingt durch die Umschaltzeit des Zellensimulators von dem Simulationssignal auf einen definierten Spannungspegel.

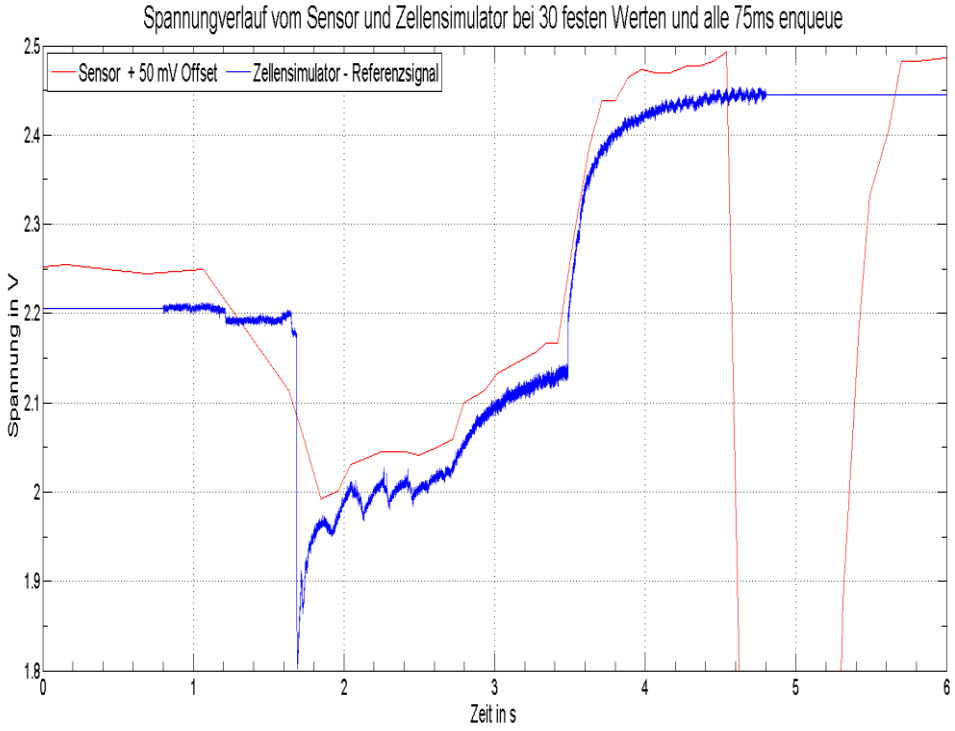

Abb. 4.9 Spannungsverlauf Enqueue-Bedingung 2

Bei Abb. 4.9 wurde der Messzeitpunkt nach dem einmaligen Erfüllen der Enqueue-Bedingung auf 75 ms gesetzt. Die Anzahl der festen Messpunkte wurde dabei nicht verändert. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators. Die Umschaltung des Zellensimulators tritt hier nach ca. 4,6 Sekunden auf.

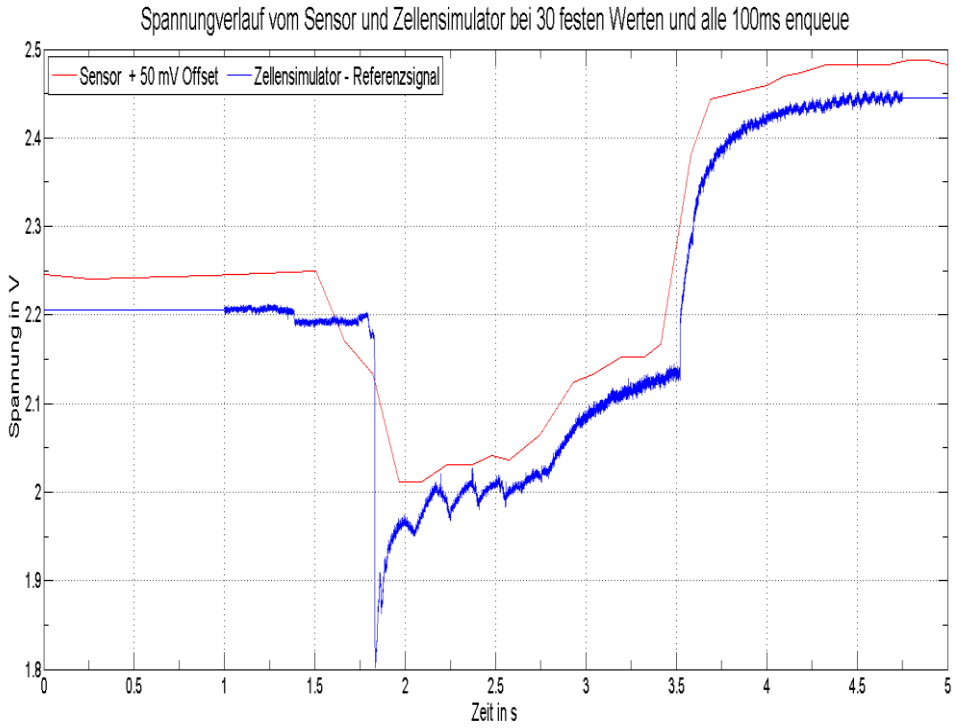

Abb. 4.10 Spannungsverlauf Enqueue-Bedingung 3

Bei Abb. 4.10 wurde der Messzeitpunkt nach dem einmaligen Erfüllen der Enqueue-Bedingung auf 100 ms gesetzt. Die Anzahl der festen Messpunkte wurde dabei nicht verändert. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators. Die festen Messpunkte alle 100 ms und die Begrenzung der Aufzeichnung auf 100 gültige Frames sorgen für die Nichterfassung des Umschaltpunktes des Zellensimulators.

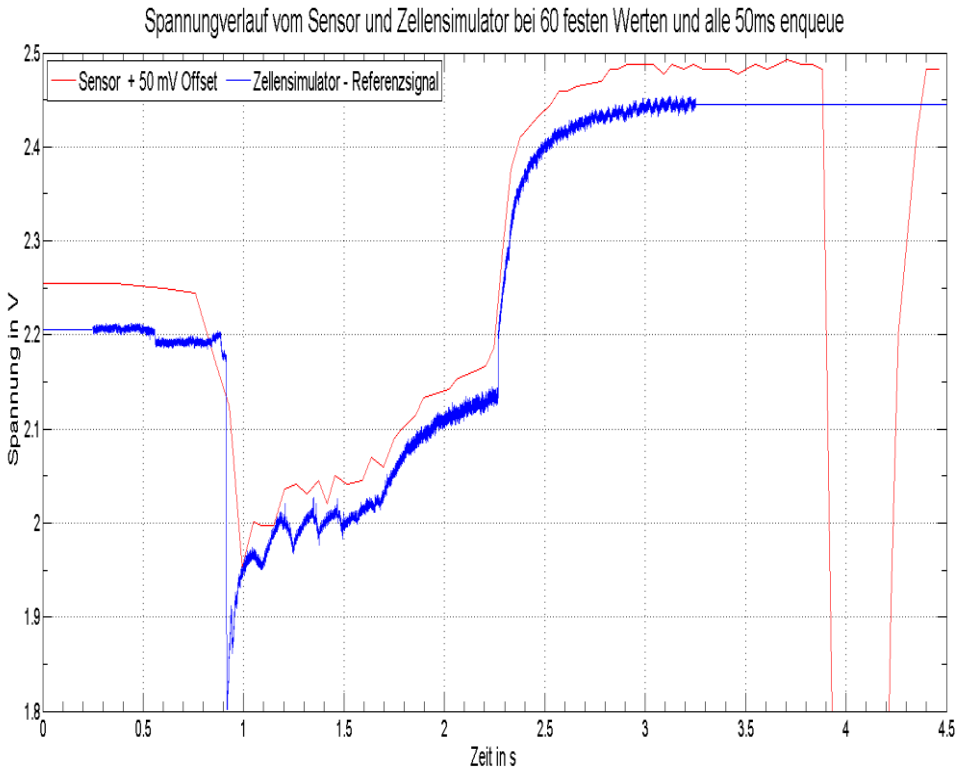

Abb. 4.11 Spannungsverlauf Enqueue-Bedingung 4

Bei Abb. 4.11 wurde der Messzeitpunkt nach dem einmaligen Erfüllen der Enqueue-Bedingung auf 50 ms gesetzt. Die Anzahl der festen Messpunkte wurde auf 60 Werte verändert. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators. Der Einbruch ab Sekunde 3,9 ist bedingt durch die Umschaltzeit des Zellensimulators von dem Simulationssignal auf einen definierten Spannungspegel.

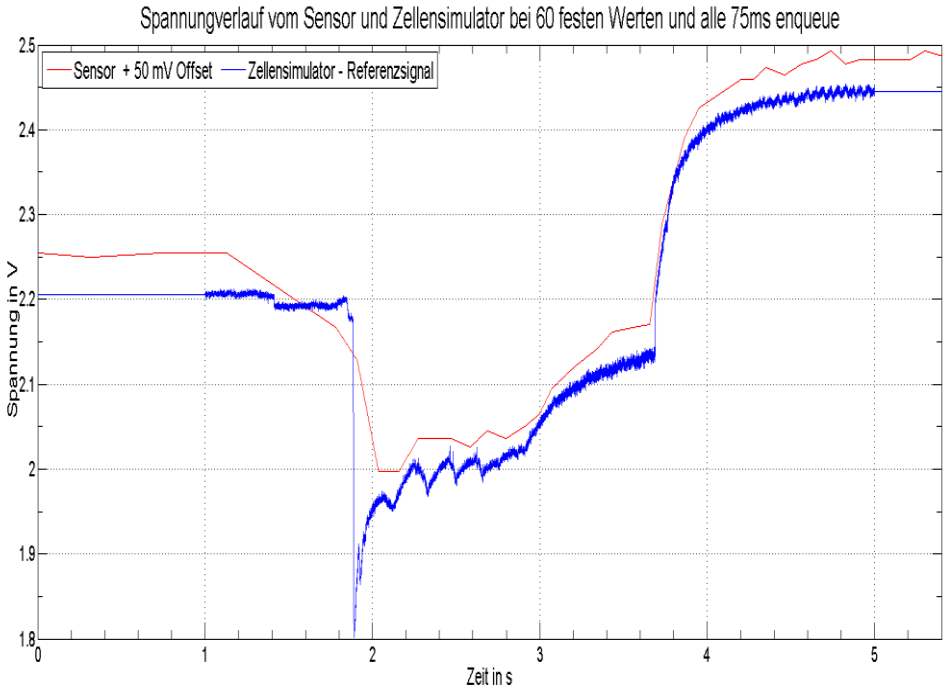

Abb. 4.12 Spannungsverlauf Enqueue-Bedingung 5

Bei Abb. 4.12 wurde der Messzeitpunkt nach dem einmaligen Erfüllen der Enqueue-Bedingung auf 75 ms gesetzt. Die Anzahl der festen Messpunkte wurde auf 60 Werte verändert. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators.

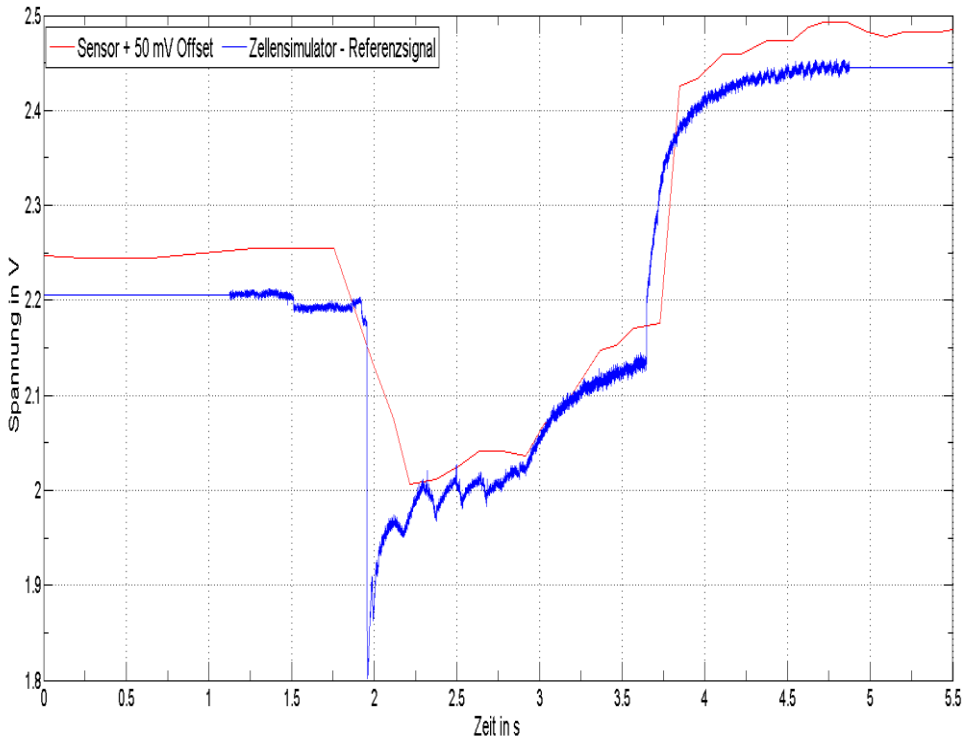

Abb. 4.13 Spannungsverlauf Enqueue-Bedingung 6

Bei Abb. 4.13 wurde der Messzeitpunkt nach dem einmaligen Erfüllen der Enqueue-Bedingung auf 100 ms gesetzt. Die Anzahl der festen Messpunkte wurde auf 60 Werte verändert. Der rote Graph zeigt den Spannungsverlauf des Sensors und der blaue Graph das Ausgangssignal des Zellensimulators.

In Anbetracht der Messwerte und der nötigen Genauigkeit ist eine Parameterwahl von 100 ms und 30 festen Messpunkten vorzuziehen. Ein Grund für die Wahl der Einstellungen ist die Problematik, welche entsteht, wenn der erste Startvorgang eines Fahrzeuges erfolglos bleibt und ein zweiter Startvorgang innerhalb kürzester Zeit folgt. Die Queue wäre bei der festen Messwertaufnahme von 60 Messpunkten schon zu ca. 83 % gefüllt. Die maximale Länge der Queue beträgt 65 Messwerte. Somit könnten anschließend nur noch weitere 5 - 10 Messpunkte aufgenommen werden. Die stark verringerte Messwertaufnahme führt dann zu einem verzerrten Signalverlauf. Die Sensorsoftware ist so konfiguriert, dass nur dann Werte in der Queue gespeichert werden, wenn Platz vorhanden ist. Alle anderen Werte werden dabei verworfen. Eine Messwertaufnahme im Inqueuefall von 50 ms ist nicht optimal. Einerseits wird bei 30 Messpunkten eine hohe Auflösung im Bereich von 1,5 Sekunden erzielt. Nach dieser Messwertaufnahme würde jedoch eine erneute Prüfung stattfinden. Im Falle der Nichterfüllung der Enqueue-Bedingung kehrt die Sensorsoftware

zurück in den Normalbetrieb. Der Startvorgang kann dabei in seiner gesamten Zeitdauer nicht zuverlässig aufgezeichnet werden. Die gewählten 100 ms als feste Messzeitpunkte können bei einer festen Messwertaufnahme von 30 Werten einen Start zuverlässig abbilden. Die Kombination aus 100 ms und 30 Werten ergibt eine resultierende Aufnahmezeit von drei Sekunden. Im Falle eines fehlgeschlagenen oder länger andauernden Startvorgangs kann eine zuverlässige Aufzeichnung gewährleistet werden. Der Nachteil dieser Realisierung ist, dass die Enqueue-Bedingung schon von kleinen Störungen ausgelöst werden kann, bei der kein wichtiges Ereignis in der Zelle oder an der Batterie anliegt. Im schlimmsten Fall könnte diese Störung kurz vor einem Hochstromereignis auftreten und die Queue schon anteilig auslasten. Ein weiterer Nachteil dieser Realisierung könnte eine zu knapp eingestellte Spannungsdifferenz für den Enqueue-Fall sein. Sollten sich die Störungen des Spannungswandlers potenzieren, käme es zu einer Dauerauslastung der Queue. Der Transmitter würde dann in diesem Fall die Zelle kontinuierlich belasten und die Messwerte aus der Queue wären dann nicht mehr zusammenhängend.

Die dritte Realisierung besteht darin, dass nach einmaligem Erfüllen der Enqueue-Bedingung die Spannungsdifferenz für den Enqueue-Fall herab gesetzt wird. Diese Realisierung würde die Queue dann dynamischer auslasten. Diese Implementierung wurde nicht mehr am Zellensimulator erprobt oder getestet. Diese Art der Realisierung würde die Nachteile von kurzen Störungen, welche die Enqueue-Bedingung erfüllen mindern. In dieser Realisierung wird bei jedem Spannungswert die Enqueue-Bedingung geprüft. Die Spannungsdifferenz wird bei einmaligem Erfüllen anschließend herabgesetzt. Erfüllt die Spannungsmessung diese Differenz nicht mehr, so kehrt der Sensor nach Abbau der entstandenen Queue wieder in den Normalbetrieb zurück. Störungen kurzer Dauer würden somit die Queue nicht mehr auslasten, wie es bei der festen Anzahl von Messwerten geschieht. Das Herabsetzen der Spannungsdifferenz ist im Bezug auf ein mögliches, dauerhaftes einreihen von Messwerten in die Warteschlange gefährlich und stellt eine zusätzliche Belastung für die Zelle, vor allem im Ruhezustand des Fahrzeuges, dar.

Für alle Realisierungen gilt, dass die Spannungsdifferenz für den Enqueue-Fall nicht zu knapp ausgelegt werden sollte, damit sich kleine Störungen oder ähnliche Effekte nicht auf das Warteschlangenverfahren auswirken oder aber ein dauerhaftes Speichern der Werte in die Warteschlange auslösen.

5 Kalibrierung

Die ADC Werte sind eine digitale Repräsentation von einem analogen Spannungswert. Dieser Spannungswert ist durch den ADC fehlerbehaftet. Im Zuge dessen wurden die Sensoren auf die Spannungsmessung kalibriert. Die Sicherungen des Sensors haben einen nicht unerheblichen Anteil an der Verzerrung der gemessenen Spannung. Beim Digitalisieren von analogen Größen kommt es durch den ADC zu Fehlern.

Es tritt ein Offsetfehler auf. Dieser bewirkt eine konstante Abweichung in allen Bereichen vom realen Messwert. Dieser lässt sich durch eine einfache Korrekturrechnung kompensieren.

Der auftretende Quantisierungsfehler ließe sich durch die Entfernung des Eingangsspannungsteilers oder durch einen höher auflösenden ADC minimieren. Der Quantisierungsfehler tritt, bedingt durch die endliche Auflösung des ADC, immer auf undbeträgt ± ½ LSB.
Bedingt durch den Spannungsteiler beträgt er bei dem Sensor 2,44 mV.

$\frac{5V}{1023} = 4{,}88mV$, $\frac{4{,}88mV}{2} = 2{,}44mV$

Ohne Spannungsteiler würde der Quantisierungsfehler 1,22 mV betragen.

$\frac{2{,}5V}{1023} = 2{,}44mV$, $\frac{2{,}44mV}{2} = 1{,}22mV$

Der auftretende Verstärkungsfehler ist die Abweichung der tatsächlichen Übertragungsfunktion von der Idealen unter Ausschluss des Offsetfehlers. Im Datenblatt des Mikrocontrollers ist der maximal auftretende Verstärkungsfehler mit 2 LSB angegeben [16].
Der Linearitätsfehler eines ADC ist die Abweichung seiner realen Kennlinie von der idealen Kennlinie. Bei dem verwendeten Mikrocontroller beträgt dieser maximal 1 LSB.

Für die Kalibrierung der Sensoren kam der Zellensimulator [19] zum Einsatz.
Dieser wurde so konfiguriert, dass er definierte Spannungspegel im Bereich von 1 Volt bis 2,4 Volt in 100 mV Schritten ausgibt. In den Sensoren wurde das enqueuing deaktiviert und eine einfache Spannungsmessroutine implementiert. Innerhalb dieser Routine wird die Spannung zwei Mal in kurzen Zeitabständen gemessen und über diese beiden Werte gemittelt. Im Anschluss an diese Messungen wurde der Spannungswert an das Steuergerät zur Weiterverarbeitung im Softwareprogramm Matlab gesendet. Die Messung wurde für jeden Spannungspunkt zehnfach wiederholt. Zu dem Sensor kam parallel ein

12 Bit ADC zum Einsatz. Die Messwerte dieses ADCs dienten der Ermittlung der angelegten Spannung.

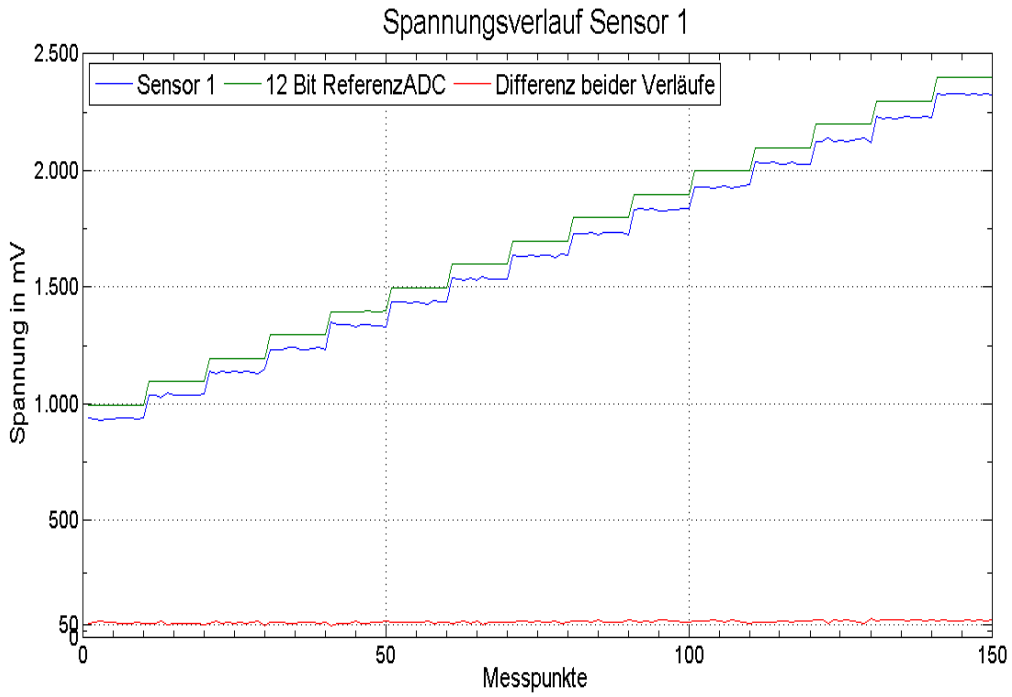

Abb. 5.1 Spannungsverlauf Sensor und Referenz ADC

Die Abb. 5.1 zeigt den Verlauf der gemessenen Spannung über einen Sensor sowie den 12 Bit ADC. Der grüne Graph zeigt den Spannungsverlauf des Referenz ADC und der blaue Graph den Sensor 1. Der rote Verlauf zeigt den Unterschied zwischen dem 12 Bit ADC und dem Sensor auf und somit den Offset zwischen beiden Messwerten. Dieser Offset beträgt ca. 50 mV.

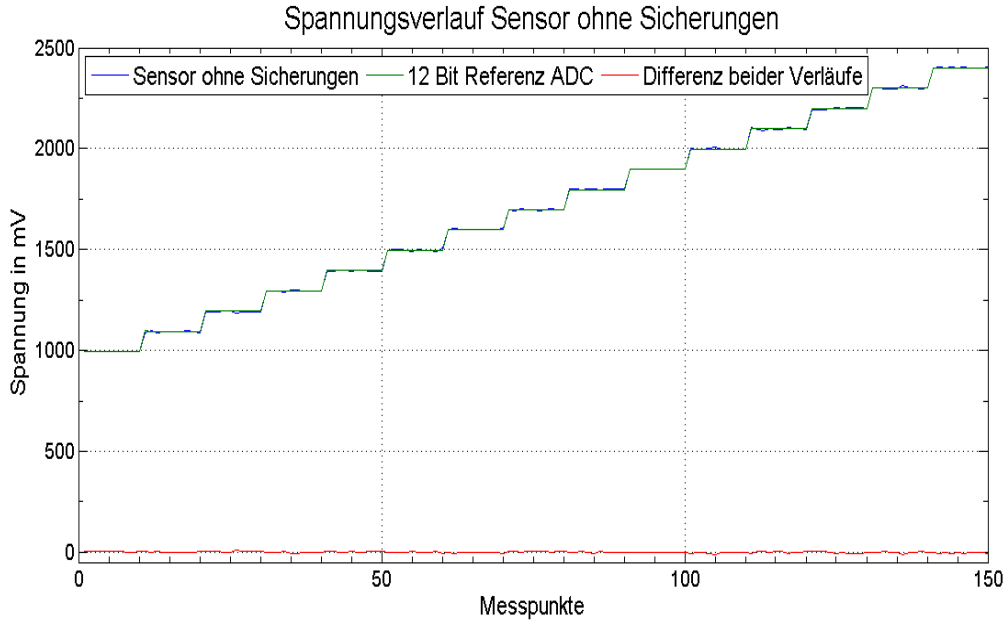

Abb. 5.2 Spannungsverlauf Sensor ohne Sicherungen und Referenz ADC

Die Abb. 5.2 zeigt den Spannungsverlauf vom 12 Bit ADC zu einem Sensor ohne Sicherungen. Die beiden Kurven sind nahezu deckungsgleich und die Differenz der beiden Spannungsverläufe ist nahe 0 mV.

Im Anschluss an diese Messreihen wurden mithilfe des Softwareprogramms Matlab die Parameter der Regressionsgeraden bestimmt. Die Methode der Ausgleichsrechnung nach dem Gauß‘schen Prinzip der kleinsten Quadrate dient zur Bestimmung der Parameter.
Die Regressionsgerade ist definiert als:

$$y = m \cdot x + b$$

$$y = m \cdot (ADC - WERT \cdot 2) + b$$

	m	b in µV
Sensor 1	1.0082	$5.0126 \cdot 10^4$
Sensor 2	0.9890	$4.6894 \cdot 10^4$
Sensor 3	0.9961	$5.0171 \cdot 10^4$
Sensor 5	1.0111	$5.4232 \cdot 10^4$
Sensor 6	1.0077	$4.5674 \cdot 10^4$
Sensor 7	0.9924	$4.9317 \cdot 10^4$
Sensor 8	1.0076	$5.0589 \cdot 10^4$
Sensor 9	0.9843	$5.0126 \cdot 10^4$
Sensor ohne Sicherungen	0.9965	$5.8939 \cdot 10^3$

Tabelle 5-1 Kalibrierungsfaktoren

Die Tabelle 5-1 zeigt die erstellten Korrekturfaktoren für die Spannungsmessung. Der Zellensimulator nimmt Einstellungen für die Ausgangsspannung nur im Mikrovoltbereich entgegen. Aufgrund dieser Tatsache sind die Parameter in b entsprechend hoch. Der Sensor 4 wurde nicht kalibriert, denn er war zum Zeitpunkt der Kalibrierung noch mit den hochohmigen Sicherungen bestückt.

6 Fahrzeugerprobung

In diesem Kapitel wird der Ablauf der Fahrzeugerprobung beschrieben. Dabei ging es darum, eine Fahrzeugbatterie mit den Sensoren auszurüsten und Messdaten von bestimmten Szenarien wie der Startvorgang und dem Zuschalten von Verbrauchern, zu erhalten. Das Fahrzeug, ein BMW 3.16, wurde vom Department Fahrzeugtechnik gestellt. Für die Fahrzeugerprobung wurden die Sensoren 3, 5, 6, 7, 8 und 9 verwendet.

6.1 Aufbau an der Batterie

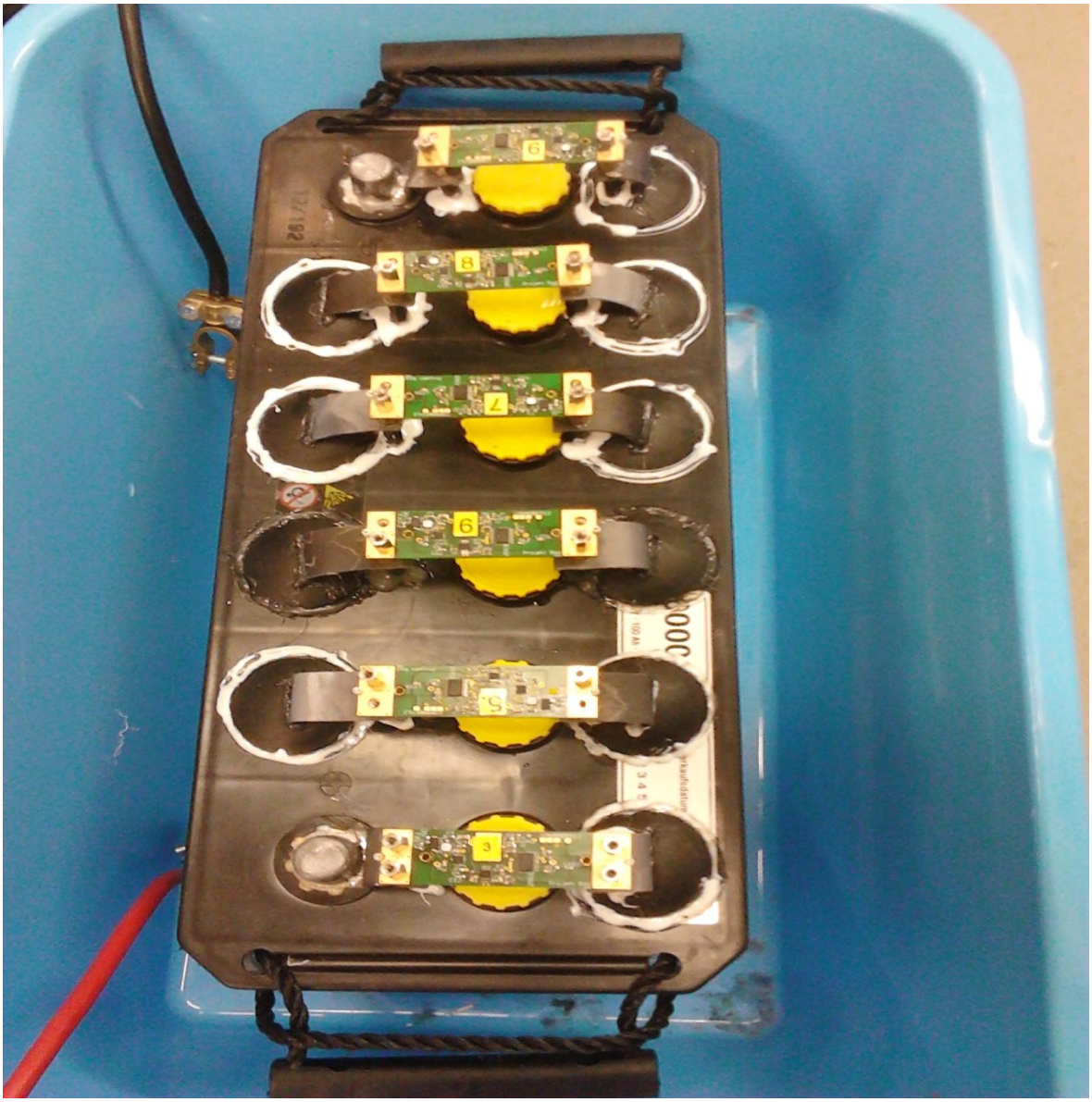

Abb. 6.1 Batterie mit Sensoren

Die Abb. 6.1 zeigt die mit den Sensoren bestückte Fahrzeugbatterie.

Die Batterie wurde vorher präpariert, damit die Sensoren extern montiert werden können. Dieser Aufbau ermöglicht eine einfache Veränderung der Konfiguration der Sensoren. Die Zellenspannung wurde mithilfe von Bleianschlüssen nach außen geführt und mit Abstandshaltern versehen. Für Messungen im Stand wurde die vorhandene Autobatterie abgeklemmt und die Kabel der Fahrzeugelektrik an der präparierten Batterie angeschlossen.

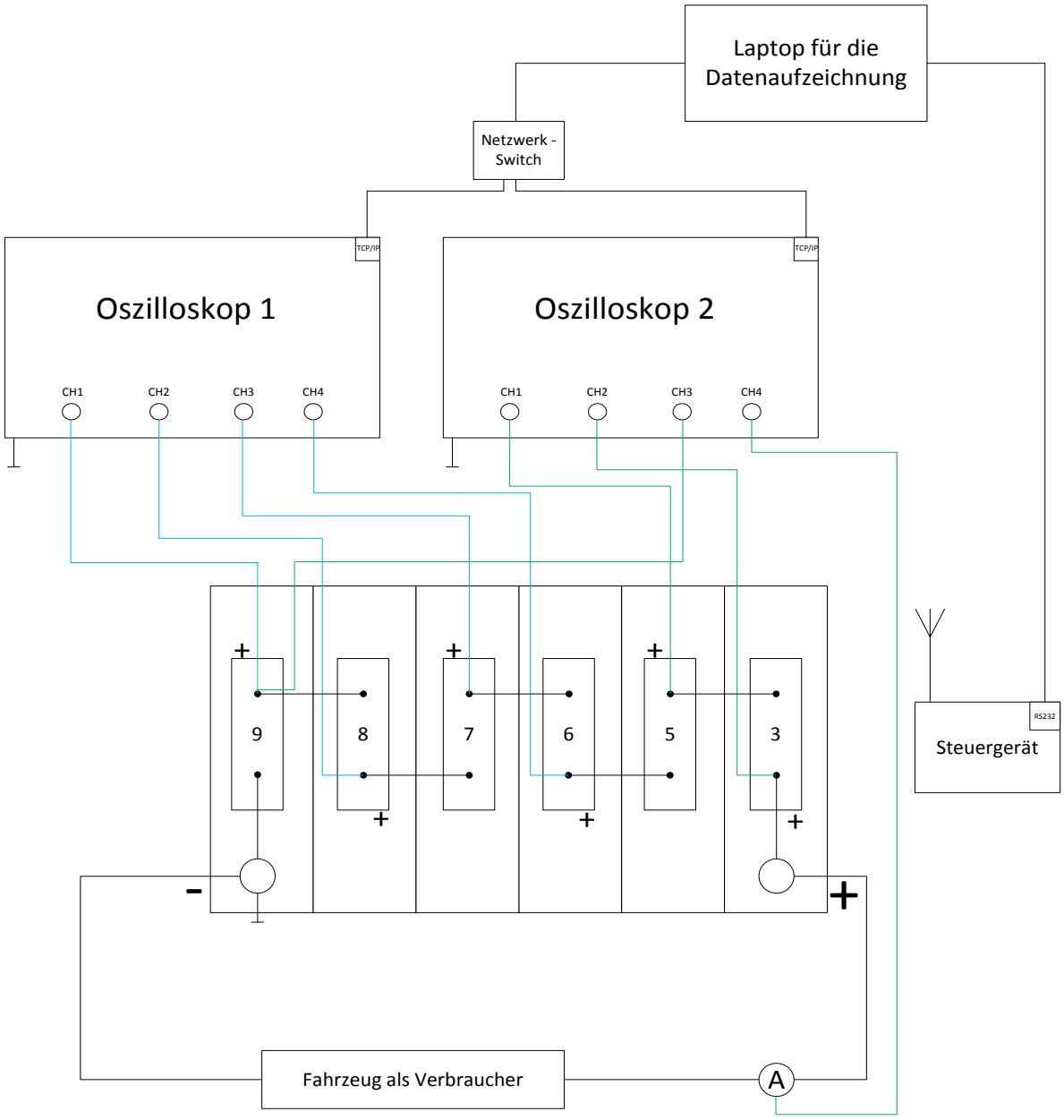

Abb. 6.2 Schematische Darstellung vom Messaufbau

Abb. 6.2 zeigt den schematischen Messaufbau an der Fahrzeugbatterie mit den beiden Oszilloskopen, dem Steuergerät sowie dem Laptop für die Datenaufzeichnung.

Abb. 6.3 Adaptiere Fahrzeugbatterie und Messgeräte am Fahrzeug

Die Abb. 6.3 zeigt das Fahrzeug sowie den Messaufbau. Im Vordergrund sind die zwei Oszilloskope zu sehen, welche für die Aufzeichnung der Spannungen der einzelnen Zellen sowie des Gesamtstromes zuständig sind. Die Oszilloskope wurden parallel zu den Zellensensoren angeschlossen. Der Laptop diente zur Datenaufzeichnung der Sensoren über das Steuergerät sowie der im Netzwerk angeschlossenen Oszilloskope.

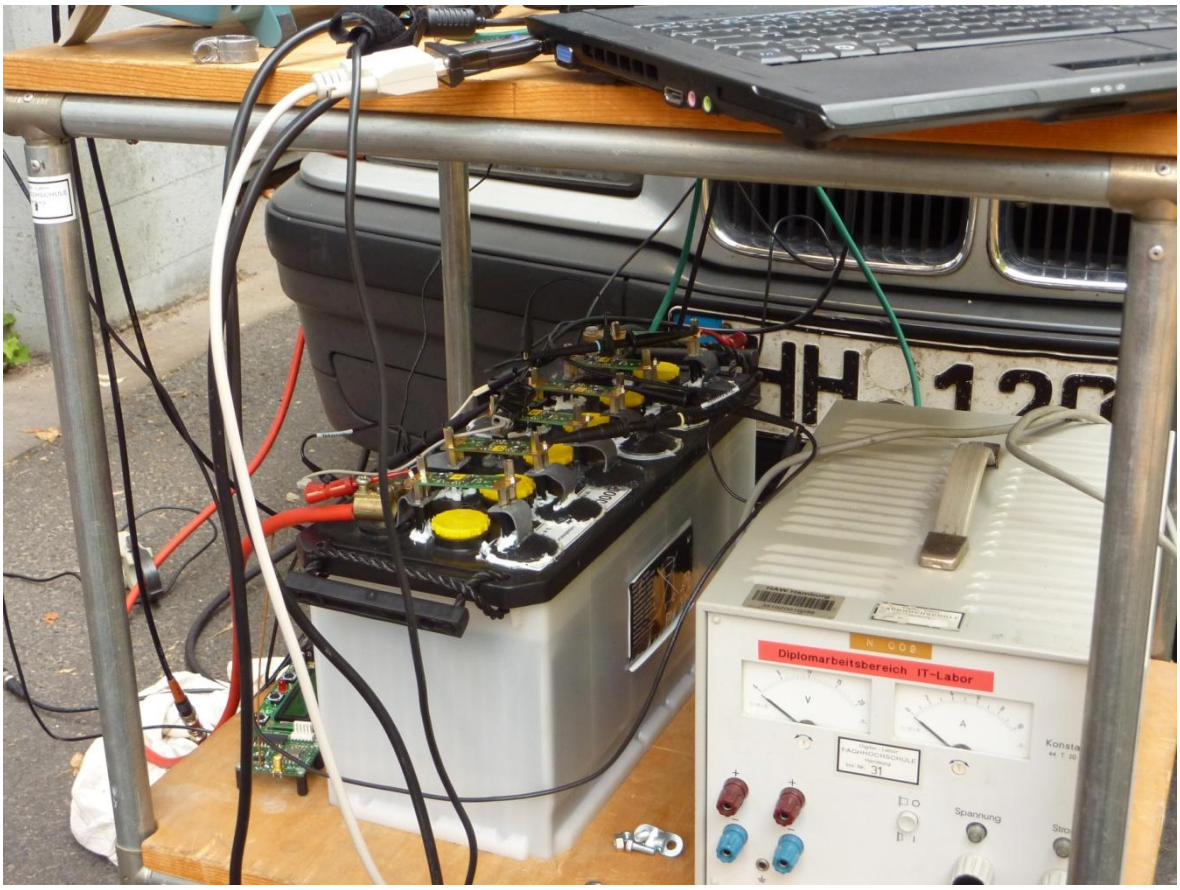

Abb. 6.4 Erprobungsbatterie mit Sensoren und Tastköpfen

Die Abb. 6.4 zeigt die Fahrzeugbatterie mit den Sensoren und den Tastköpfen der Oszilloskope. Aufgrund der internen Masseverschaltung der Oszilloskope wurde nur eine Masse angeschlossen. Die Zellen würden ansonsten über die Eingänge der Oszilloskope kurzgeschlossen werden.

6.2 Szenarien am stehenden Fahrzeug

Als erster Test wurde das Fahrzeug mehrfach gestartet. Zusätzlich wurden elektrische Verbraucher wie die Heckscheibenheizung und das Abblendlicht zugeschaltet.

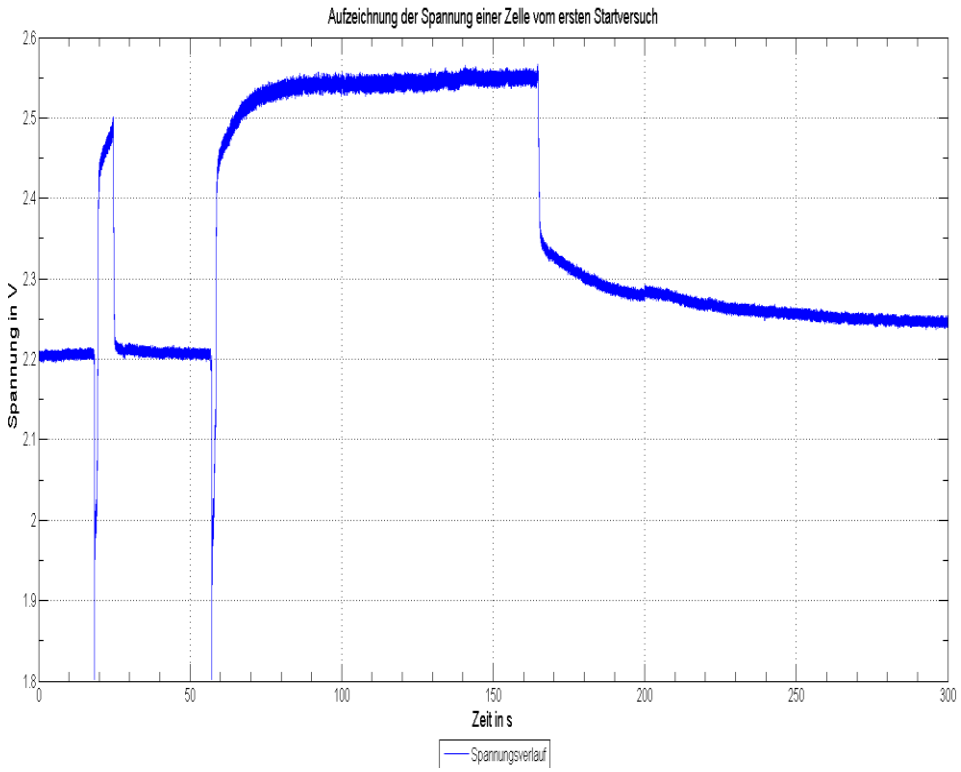

Abb. 6.5 Spannungsverlauf am Oszilloskop einer Zelle über den gesamten ersten Testlauf

Die Abb. 6.5 zeigt den gesamten Spannungsverlauf einer Zelle über den kompletten ersten Testlauf an. Das Fahrzeug wurde hierbei zweimal gestartet.

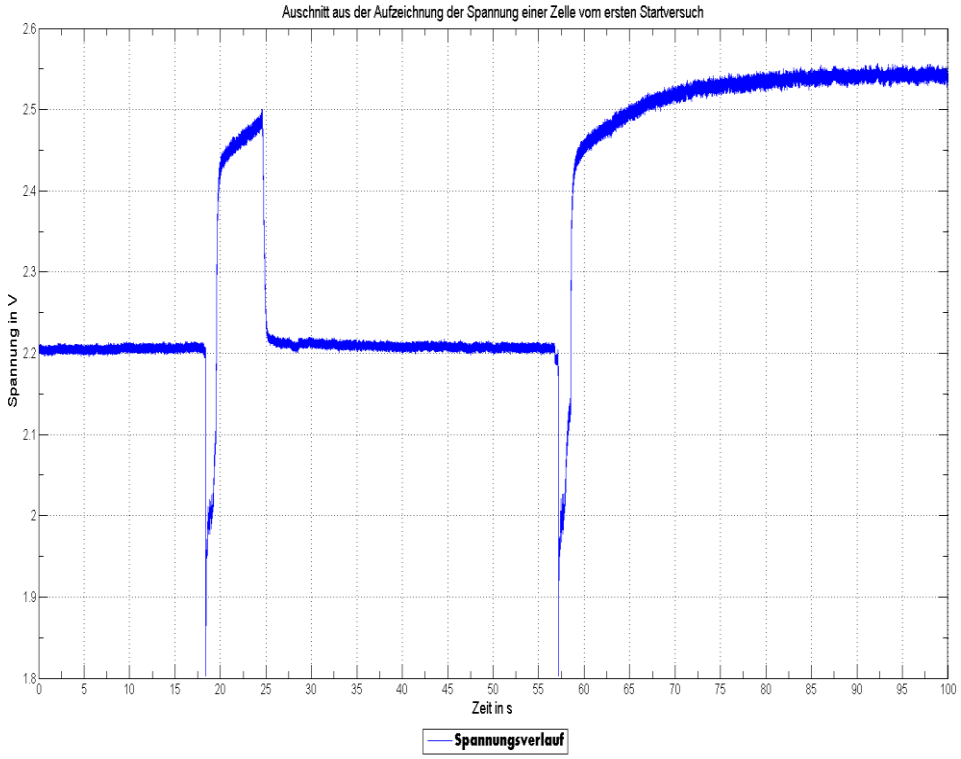

Abb. 6.6 Ausschnitt Spannungsverlauf am Oszilloskop Kanal 1 beim ersten Testlauf

In Abb. 6.6 sind die beiden Startvorgänge von dem BMW zu sehen. Aufgenommen wurde dieser Spannungsverlauf an der ersten Zelle der Batterie.

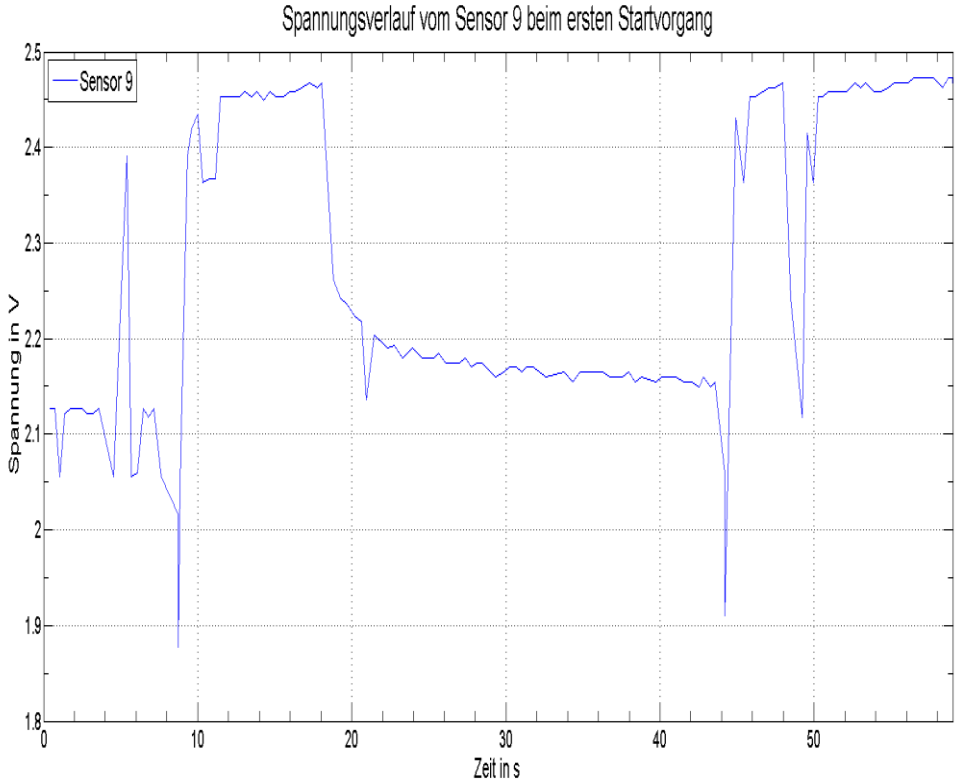

Abb. 6.7 Spannungsverlauf vom Sensor 9 über den kompletten ersten Testlauf

Die Abb. 6.7 zeigt den kompletten Ablauf des ersten Tests im Stand am Sensor. Die Aufzeichnung zeigt die Nachteile der ersten Implementierung der Enqueue-Bedingung auf. Die beiden Startvorgänge sind nur grob erkennbar. Durch die zeitlich nicht eindeutig zusammenhängende Prüfung der Spannungsdifferenz und der fehlerhaften Glättung der Messwerte ist der Spannungsverlauf nicht eindeutig mit dem der Oszilloskop-Daten vergleichbar.

Im zweiten Test wurden bei laufendem Motor die Heckscheibenheizung und die Lichthupe mehrfach betätigt.

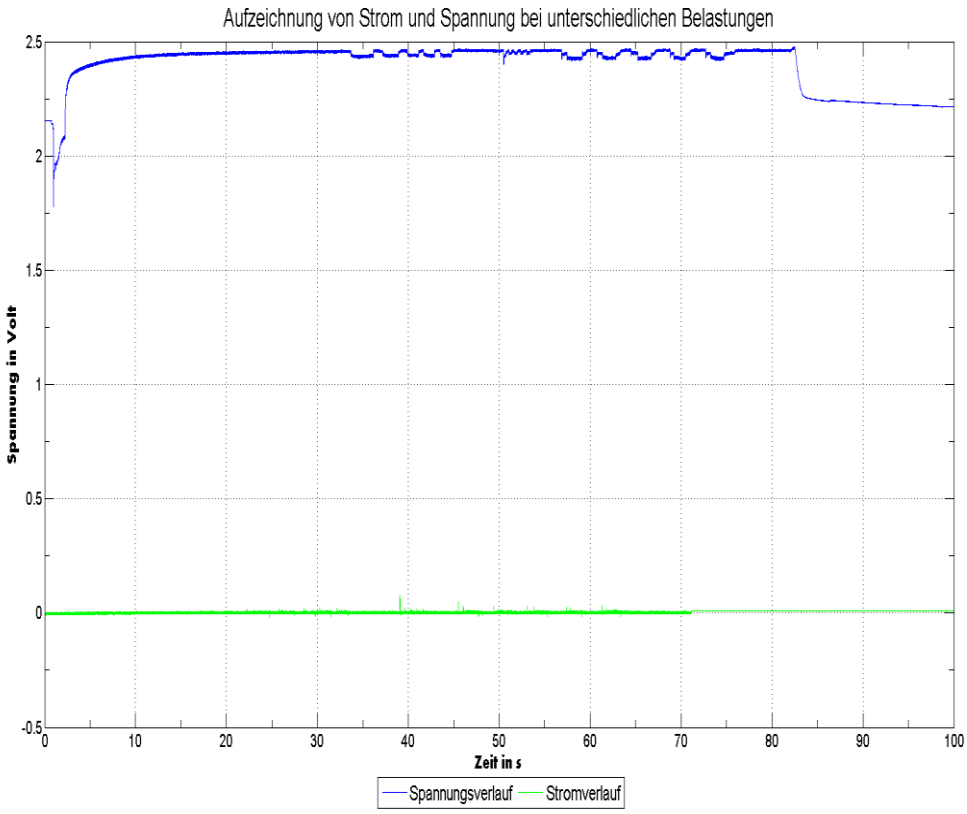

Abb. 6.8 Strom- und Spannungsverlauf am Oszilloskop - zweiter Test am Fahrzeug

Die Abb. 6.8 zeigt den Strom- und Spannungsverlauf an der Batterie für den zweiten Test am Fahrzeug. Für diesen Test wurde bei laufendem Motor die Lichthupe 5-mal betätigt und dann die Heckscheibenheizung ebenfalls 5-mal ein- und ausgeschaltet. Die Schaltvorgänge sind gut im Spannungsverlauf durch die Einbrüche erkennbar. Anschließend wurde während der Lichthupe die Heckscheibenheizung zugeschaltet.

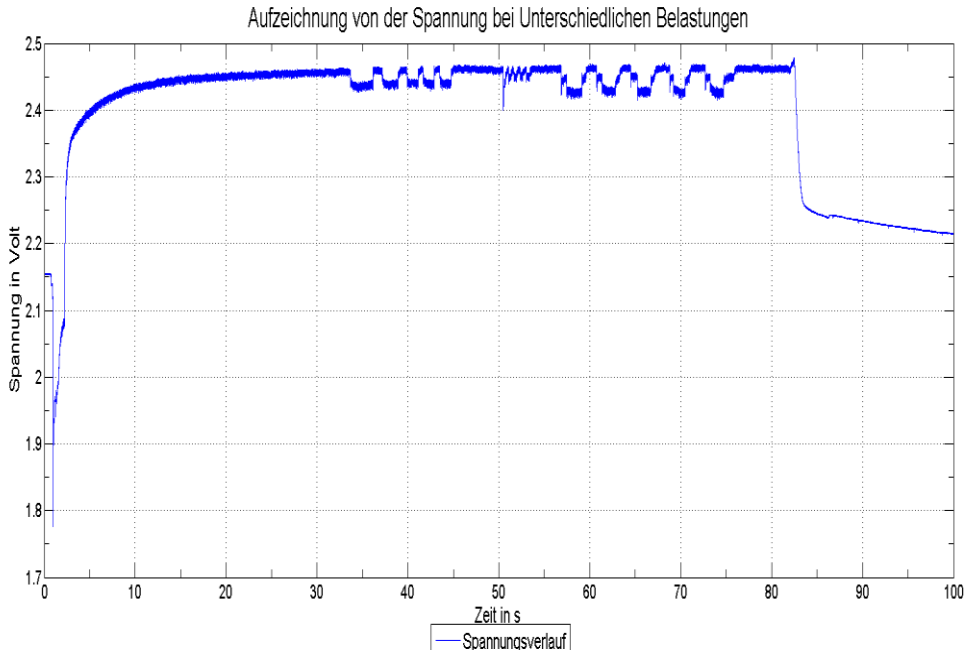

Abb. 6.9 Spannungsverlauf am Oszilloskop - zweiter Test

Die Abb. 6.9 zeigt nur den Spannungsverlauf für den zweiten Test am Fahrzeug. Das Zuschalten der Verbraucher wird durch die Spannungseinbrüche deutlich.

6.3 Erprobungsfahrt

Für die Erprobungen auf dem Verkehrsübungsplatz musste das Fahrzeug umgerüstet werden. Die ersten Tests fanden im Stand an der Fahrzeughalle statt. Für die Erprobungsfahrt wurde das Fahrzeug am Verkehrsübungsplatz umgerüstet. Die Versorgungsleitungen der Fahrzeugelektrik wurden über das geöffnete Fenster der Beifahrertür in den Fahrgastraum gelegt. Die Batterie mit den Sensoren musste platzbedingt hinter dem Beifahrersitz installiert werden.

Abb. 6.10 Fahrzeugumbau auf dem Verkehrsübungsplatz

Die Abb. 6.10 zeigt die Versorgungsleitungen der Fahrzeugelektrik, welche von dem Motorraum über das Fenster der Beifahrertür in den Innenraum gelegt wurden.

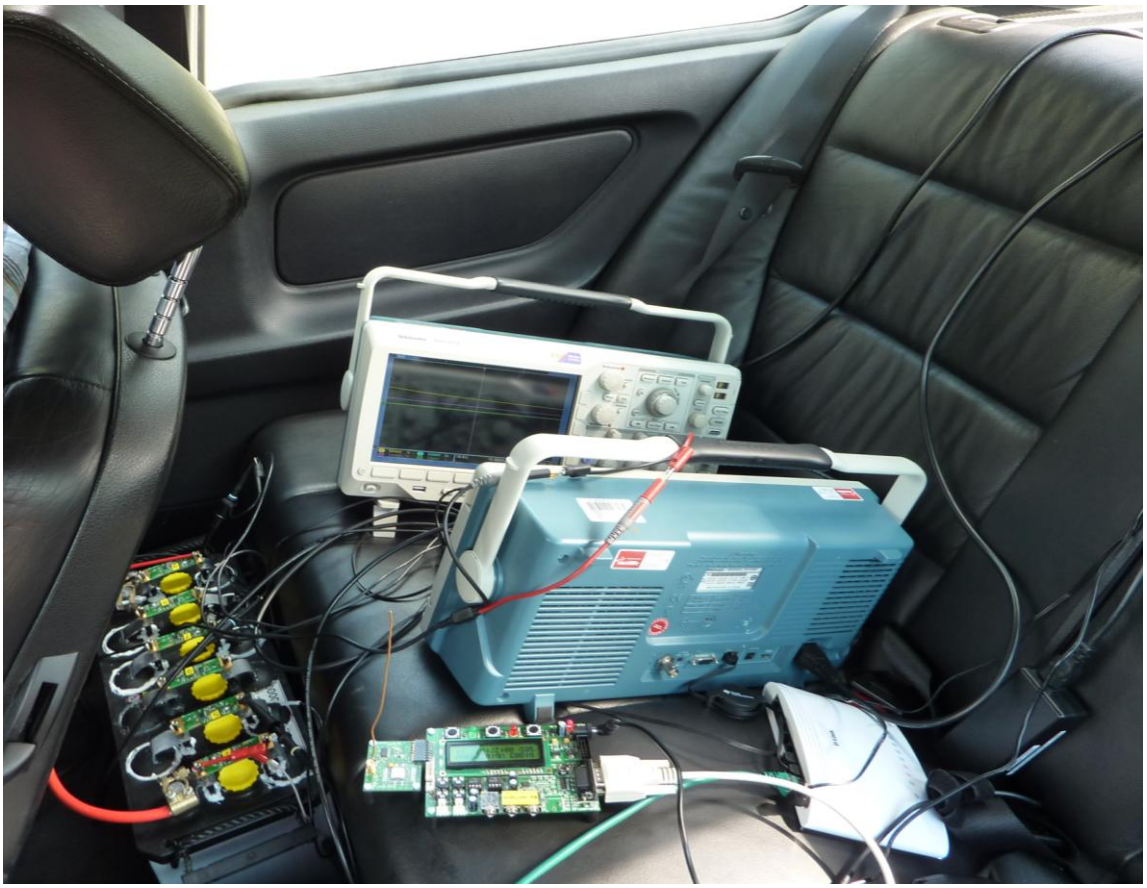

Abb. 6.11 Platzierungen der Messgeräte und der Erprobungsbatterie

Die Abb. 6.11 zeigt die Platzierungen der beiden Oszilloskope und des Steuergerätes auf der Rücksitzbank sowie die mit Sensoren bestückte Batterie hinter dem Beifahrersitz. Die Messgeräte, der Laptop für die Datenaufzeichnung von allen Quellen sowie das Steuergerät wurden von einer zusätzlichen Batterie mit Wechselrichter aus dem Kofferraum versorgt. Diese Maßnahme erfolgte, damit die Messgeräte keine Störungen auf die Sensoren verursachen.

Abb. 6.12 Zusatzbatterie und Wechselrichter

Die Abb. 6.12 zeigt den Aufbau im Fahrzeug aus Blickrichtung des Kofferraums und die Versorgungsbatterie mit Wechselrichter für die Mess- und Aufzeichnungsgeräte.

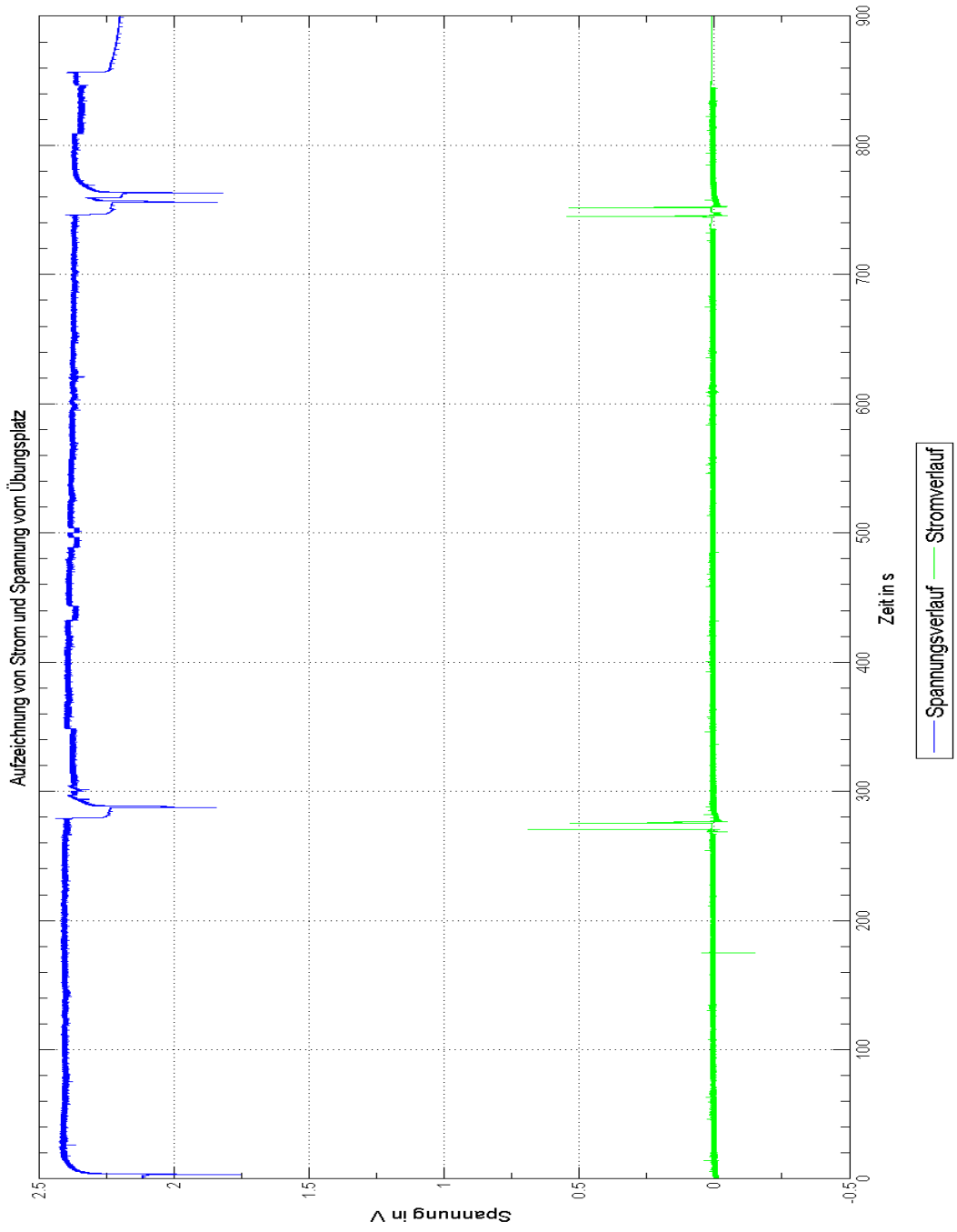

Abb. 6.13 Oszilloskop-Daten der Erprobungsfahrt

Abb. 6.13 zeigt den kompletten Verlauf der Erprobungsfahrt auf dem Verkehrsübungsplatz. Der Motor des Fahrzeuges wurde während der Erprobungsfahrt drei Mal gestartet. Die Heckscheibenheizung wurde im Laufe der Fahrt mehrfach ein- und ausgeschaltet. Die Betätigung der Heizung ist in den Spannungseinbrüchen deutlich ersichtlich. Allerdings zeigt der Stromverlauf in diesem Bereich keine Einbrüche oder Spitzen. Nur die Startvorgänge sind im Stromverlauf erkennbar. Die Ursache für die fehlenden Einbrüche beim Zuschalten der Verbraucher ist der verwendete Messbereich

der Strommesszange. Die Messzange wurde im größtmöglichen Messbereich verwendet, um den Strom beim Motostart zuverlässig aufnehmen zu können. Der Verlauf von Spannung und Strom sind dabei nicht synchron. Grund hierfür ist das Aufzeichnungsskript in dem Softwareprogramm Matlab. Dieses Skript liest im 70 Sekunden Abständen den gesamten Bildschirminhalt der Oszilloskope aus. Durch diese Methodik kommt es zum Versatz der einzelnen Kanäle untereinander. Die verwendeten Oszilloskope besitzen eine maximale Zeitauflösung von 100 Sekunden. Um keine Messwerte zu verlieren, wird bereits nach 70 Sekunden der Bildschirminhalt gespeichert.

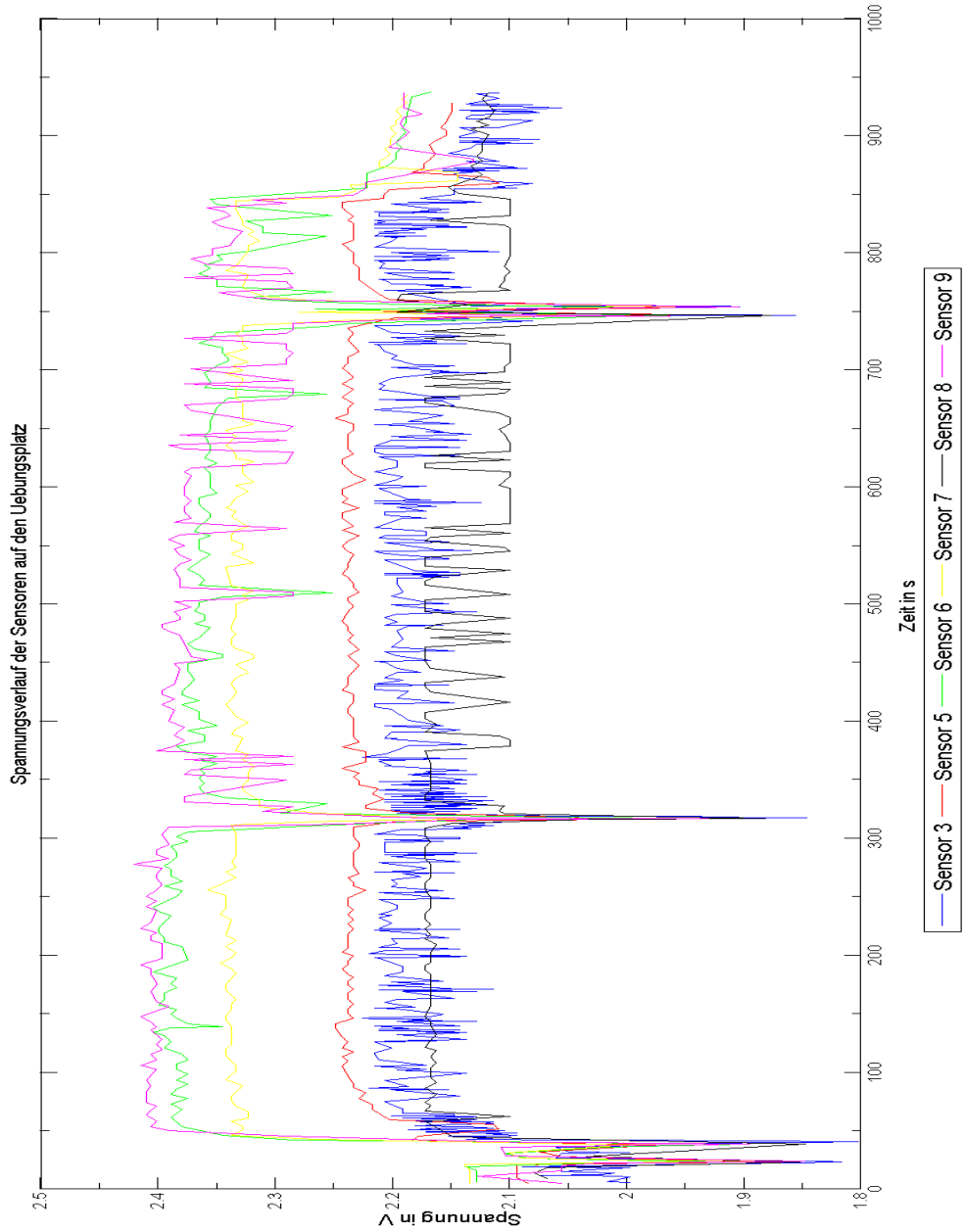

Abb. 6.14 Sensoren-Daten über die komplette Erprobungsfahrt

Die Abb. 6.14 zeigt die Sensordaten über den kompletten Verlauf der Erprobungsfahrt. Aus diesen Daten lassen sich nur die Startvorgänge erkennen. Der Sensor 3 befand sich während der Probefahrt im dauerhaften enqueuing. Die Werte vom Sensor 3 sind trotz

des dauerhaften Einreihens in die Warteschlange nicht mit den Oszilloskop-Daten vergleichbar. Bei der Erprobungsfahrt wurde die identische Sensorsoftware wie zuvor bei den Testvorgängen im Stand genutzt. Diese Software war zum Zeipunkt der Erprobung am Fahrzeug noch mit der ersten Implementierung der Enqueue-Bedingung ausgestattet.

7 Fazit und Ausblick

Im Rahmen dieser Arbeit konnten zehn Klasse-1 Sensoren erfolgreich in Betrieb genommen werden. Neun von diesen Sensoren wurden zusätzlich mit einer Sicherheitsschaltung ausgestattet. Diese Schaltung schützt die Sensoren vor Überspannungen, Kurzschlüssen sowie vor Verpolung bei der Montage. Weiterhin ist ein neues, verkürztes Protokoll entstanden und erfolgreich implementiert worden. Dieses Protokoll spart 33 Prozent der Bits gegenüber dem bisher vorhandenen Protokoll ein. In die Sensoren konnte ein Warteschlangenspeicher von 65 Werten integriert werden. Der Motorstart kann von der Software der Sensoren zuverlässig detektiert und aufgezeichnet werden. Eine Rekonstruktion des Spannungsverlaufes ist anhand der Messwerte möglich. Die Sensoren konnten erfolgreich kalibriert werden und der negative Einfluss der Sicherungen auf die Spannungsmessung kompensiert werden. Der eingesetzte Spannungswandler ermöglicht den Einsatz der Sensoren auf anderen Zellentypen mit einer anderen Zellenspannung. Der Sensor ist in der Lage, zuverlässig Spannungen im Bereich von 1 Volt bis 5 Volt zu erfassen. Die Sensordaten werden über den Transmitter erfolgreich an das Steuergerät übermittelt. Ein Zeitstempelverfahren, um die Messzeitpunkte zuverlässig zu rekonstruieren, konnte erfolgreich in die Software implementiert werden.

Weiterführend sollten folgende Aspekte näher betrachtet werden:

Die Temperaturmessung des Sensors wurde nicht kalibriert. Die für die Spannungsmessung verwendeten Kalibrierungsfaktoren wurden in dem Steuergerät hinterlegt. Der Mikrocontroller konnte innerhalb der 100 µs Zeitfenster die Spannung nicht umrechnen. Eine mögliche Abhilfe hierfür wäre das Senden der Kalibrierungsfaktoren in regelmäßigen Abständen. Diese Faktoren müssten dann nur noch temporär in dem Steuergerät hinterlegt werden, um für neue Sensoren die Software des Steuergerätes nicht zu verändern. Im Laufe der Arbeit stellte sich heraus, dass die Software des Steuergerätes eine komplette Neugestaltung benötigt. Die Software wurde nun über sechs Jahre von fünf Personen immer wieder modifiziert und ihren entsprechenden Aufgaben angepasst. Der Quellcode ist daher immer schwerer nachvollziehbar und benötigt dadurch mehr Einarbeitungszeit. Die Dokumentation des Steuergerätes ist in manchen Bereichen unvollständig.

Die Sensoren können Startvorgänge und Ruhephasen des Fahrzeuges aufzeichnen und die Daten an das Steuergerät senden. Im Laufe der Arbeit sind neue Konzepte zu dem Warteschlangenverfahren entstanden. Zum einen das beim einmaligen Erfüllen der Enqueue-Bedingung, eine feste Anzahl von Messpunkten ohne Prüfung in die Warteschlange gespeichert werden und zum anderen ein Konzept, welches nach einmaligem Erfüllen der Enqueue-Bedingung das Herabsetzen der Spannungsdifferenz vorsieht. Diese konnten nicht alle vollständig getestet werden. Die Software der Sensoren hat noch keinen finalen Stand erreicht, weitere Tests und Modifikationen, z.B. eine Abschaltung der Spannungs-versorgung für eine störfreie Messung der Zellenspannung, wären hier notwendig.
In dieser Arbeit wurde die Temperaturmessung im Startmoment nicht weiter betrachtet. Aus diesem Grund existieren keine Temperaturdaten von diesem Ereignis. Mithilfe kleinerer Softwareanpassungen könnte neben der Spannung auch die Temperatur im Startvorgang aufgezeichnet werden.

Ein weiterer Lösungsansatz, der in der Arbeit aufgekommen ist, ist die Daten der Sensoren über Lichtimpulse zu übertragen. Es wäre dabei zu prüfen, ob sich durch diesen Ansatz die Kosten der einzelnen Sensoren weiter reduzieren lassen. Der Einsatz von farbigem Licht könnte dabei die Kollisionen auf dem Übertragungskanal verringern. Offen ist bei diesem Ansatz der zuverlässige Transport der Lichtimpulse durch das verschlossene Batteriegehäuse.

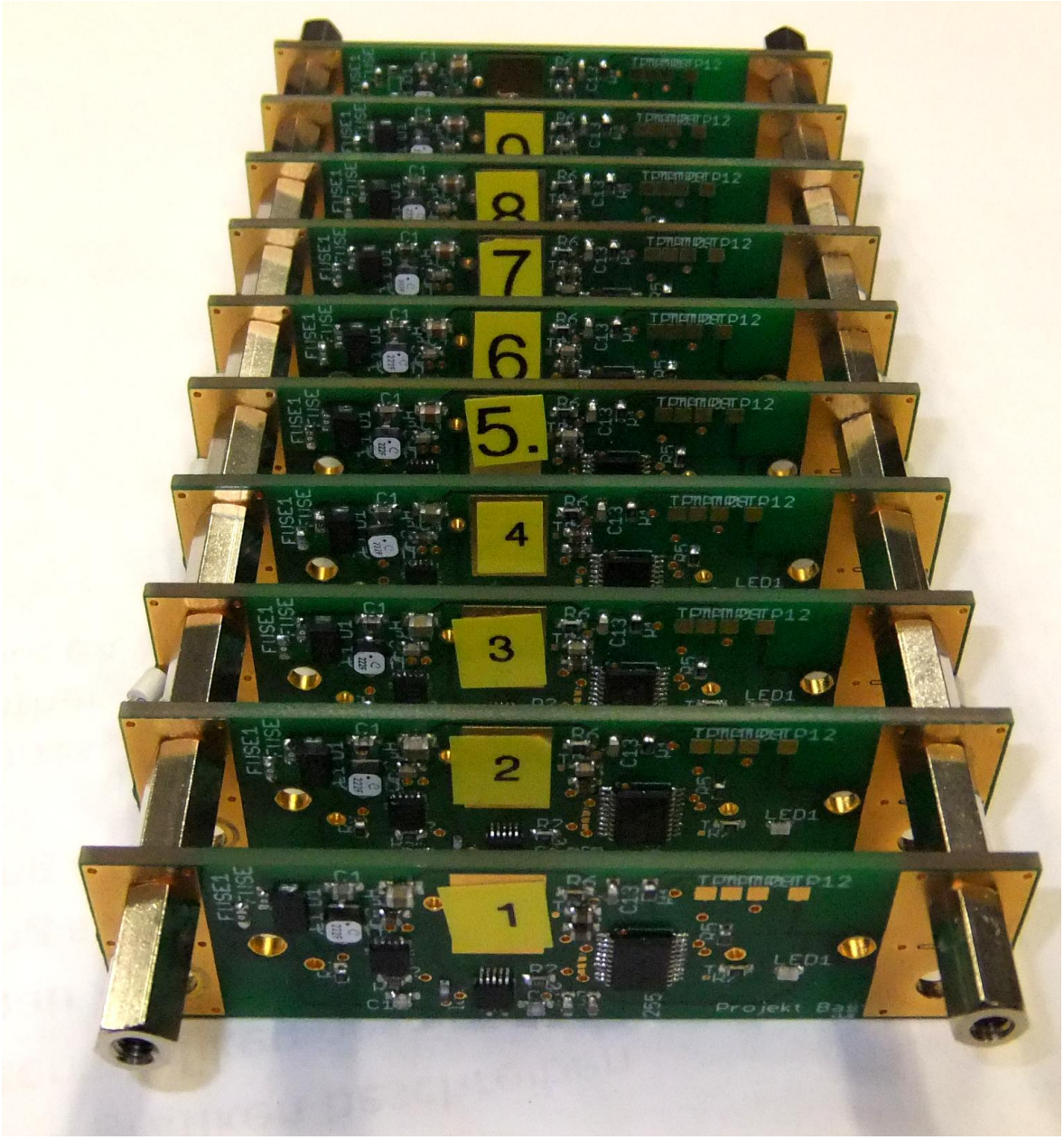

Abb. 7.1 Zehn Sensoren, bestückt und betriebsbereit

In Abb. 7.1 sind zehn Sensoren zu sehen, welche im Rahmen dieser Arbeit entstanden sind und in Betrieb genommen wurden.

Literaturverzeichnis

[1] TAB Arbeitsbericht Nr. 152; Zukunft der Automobilindustrie, http://www.tab-beim-bundestag.de/de/publikationen/berichte/ab152.html; Abruf 03.10.2013

[2] ADAC Pannenstatistik 2008; http://www.adac.de/_mmm/pdf/Pannenstatistik_2008_33402.pdf; Abruf 03.10.2013

[3] ADAC Pannenstatistik 2013; http://www.adac.de/_mmm/pdf/Pannenstatistik_2013_169836.pdf; Abruf 03.10.2013

[4] BM VGS Entwicklungsplan Elektromobilität; http://www.bundesregierung.de/Webs/Breg/DE/Themen/Energiewende/Mobilitaet/elektromobilitaet/_node.html; Abruf 03.10.2013

[5] S. Püttjer; Diagnosefunktion für Automobil-Starterbatterien; Diplomarbeit HAW Hamburg 2011

[6] Drahtlose Sensoren in den Zellen von Fahrzeug-Batterien;http://www.haw-hamburg.de/fileadmin/user_upload/TI-IE/Daten/Docs/ESZ-ASP/IWKM21_Batsen.pdf; Abruf 03.10.2013

[7] S. Ilgin; Drahtlose Sensoren für Batteriemodule - Konzeption und Kalibrierung; Bachelorthesis HAW Hamburg 2011

[8] Texas Instruments; MSP430 Hardware Tools User´s Guide; SLAU278P; http://www.ti.com/lit/ug/slau278p/slau278p.pdf; Abruf 03.10.2013

[9] Texas Instruments; MSP430 Programming Via the JTAG Interface: SLAU320L; http://www.ti.com/lit/ug/slau320l/slau320l.pdf; Abruf 03.10.2013

[10] Silicon Labs; Si4010 Antenna Interface and Matching Network Guide; AN369; http://www.silabs.com/Support%20Documents/TechnicalDocs/AN369.pdf; Abruf 03.10.2013

[11] Texas Instruments; TPS 61201 Datasheet; SLVS577D; http://www.ti.com/lit/ds/slvs577d/slvs577d.pdf; Abruf 03.10.2013

[12] Vishay; SMA6F5.0A thru SMA6F20A Datasheet; 89458; http://www.vishay.com/docs/89458/sma6f5.pdf; Abruf 03.10.2013

[13] Multicomp; SMD Fuse; http://www.farnell.com/datasheets/1693108.pdf; Abruf 03.10.2013

[14] Littlefuse; Surface Mount Fuses; 467 Series; http://www.littelfuse.com/products/fuses/surface-mount-fuses/thin-film-chip-fuses/~/media/Files/Littelfuse/Technical%20Resources/Documents/Data%20Sheets/467.pdf; Abruf 03.10.2013

[15] Silicon Labs; Si4012 Crystal-less FKS/Ook RF Transmitter; Datasheet; http://www.silabs.com/Support%20Documents/TechnicalDocs/Si4012.pdf; Abruf 03.10.2013

[16] Texas Instruments; Mixed Signal Mircocontroller Datasheet; SLAS735J; http://www.ti.com/lit/ds/symlink/msp430g2553.pdf; Abruf 03.10.2013

[17] Texas Instruments; Efficient Multiplication and Division Using MSP430;SLAA329; http://www.ti.com/lit/an/slaa329/slaa329.pdf; Abruf 03.10.2013

[18] N. Jegenhorst; Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation; Masterthesis HAW Hamburg 2011

[19] T. Steinmann; Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator; Bachelorthesis HAW Hamburg 2012

[20] Coilcraft; Shielded Power Inductors - LPS3015 Series; http://www.coilcraft.com/pdfs/lps3015.pdf; Abruf 03.10.2013

Anhang

A 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

1. Juli 2013

Bachelorarbeit: Michael Meinzer

Drahtlose Zellensensoren für Fahrzeugbatterien - Hard- und Softwareentwicklung sowie Erprobung

Motivation

Für den Erfolg der Elektrofahrzeuge, aber auch zur Verbesserung der Zuverlässigkeit von konventionellen Fahrzeugen, ist die Batterie eine wichtige Komponente. In dem BMBF-geförderten Projekt "Drahtlose Zellensensoren für Fahrzeugbatterien - BATSEN" werden verschiedene Klassen von Funksensoren untersucht, mit deren Hilfe Aussagen über den Lade- und Alterungszustand von mehrzelligen Fahrzeugbatterien möglich werden sollen. Die Sensoren selbst werden dabei im Inneren der jeweiligen Batteriezellen platziert. Dort messen sie Spannung und Temperatur und übertragen diese an ein Steuergerät außerhalb der Batterie. Dieses misst den Strom durch die Batterie, kombiniert diesen Wert mit den Daten der Sensoren und ermittelt daraus den Lade- und Alterungszustand der Batterie.

Aufgabe

Herr Meinzer erhält die Aufgabe einen vorhandenen Prototyp eines aufwandgünstigen Sensors ohne Rückkanal (Klasse 1) weiterzuentwickeln und zu testen. Dabei soll auf aktuelle Hardwarekomponenten, u.a. auf eine moderne Controllervariante aus der MSP430-Familie der Fa. Texas Instruments sowie auf einen quarzlosen Transmitter der Fa. Silicon Labs zurückgegriffen werden. Es sollen eine Kleinserie (6-12 Stück) von Sensormustern für verschiedene Erprobungen gefertigt werden. Der Schwerpunkt der Arbeit liegt auf der Erweiterung der Sensorsoftware und deren Erprobung sowohl im Labor als auch am Fahrzeug.
Die Aufgabe gliedert sich wie folgt:

1) **Einarbeitung und Analyse der Rahmenbedingungen**
   - Einordnung und Beitrag zur Projektzielstellung
   - Recherche in Fach- und Firmenliteratur
   - Einarbeiten in die Vorarbeiten im Projekt
   - Recherche Bauelementen in Bezug auf Anforderungen
2) **Schaltungskonzept und Hardwareentwurf**
   - Erarbeitung des Schaltungskonzeptes,
   - Einbindung einer Sicherheitsschaltung (Verpolung, Kurzschluss und Überspannung)
   - Konkrete Auswahl von Bauelementen und Schaltunganpassung an die Pinbelegung
   - Antennengestaltung passend zur Bauform des Sensors
   - Entwurf des Schaltplans und Erstellen des Platinenlayouts

3) **Inbetriebnahme und Laborerprobung**

- Aufbau der Kleinserie von Labormustern
- Schrittweise Inbetriebnahme der Sensoren im Labor
- Test der Schaltung sowie der drahtlosen Übertragung durch Testsoftware und Debugbetrieb
- Untersuchen von Störeinflüssen (z.B. DC/DC-Converter) sowie Messauflösung und Messrauschen
- Test auf Messgenauigkeit und Eingangsspannungsbereich, insbesondere im Zusammenwirken mit der Sicherheitschaltung

4) **Kalibrierung**

- Durchführung von Temperatur- und Spannungs-Kalibrierungen
- Festlegung von Kalibrierungsbereich und Kalibrierungsschrittweite
- Erfassung der Kalibierdaten, Eintrag von Kompensationwerten in jeden Sensoren
- Kompensationsrechnung in der Sensorsoftware realisieren

5) **Softwareanpassungen und Warteschlangenverfahren**

- Anpassung der Software an den neuen Controller und Transmitter
- Zwischenspeichern von Messwerten in einer Warteschlange
- Zeitstempel-Verfahren für Rekonstruktion von Messzeitpunkten
- Test der Warteschlangenfunktion und der Sensorfunktionen im Labor

6) **Fahrzeug-Erprobung**

- Anpassen der Steuergerätesoftware für die neuen Sensoren
- Durchführung von Versuchen am Fahrzeug, optional Nutzung eines Rollenprüfstandes für verschiedene Fahrzeugzustände oder exemplarischer Start- und Fahrbetrieb
- Auswertung und Analyse der Erprobungs-Daten

Dokumentation

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

B Sensor

B.1 Sensor Schaltung

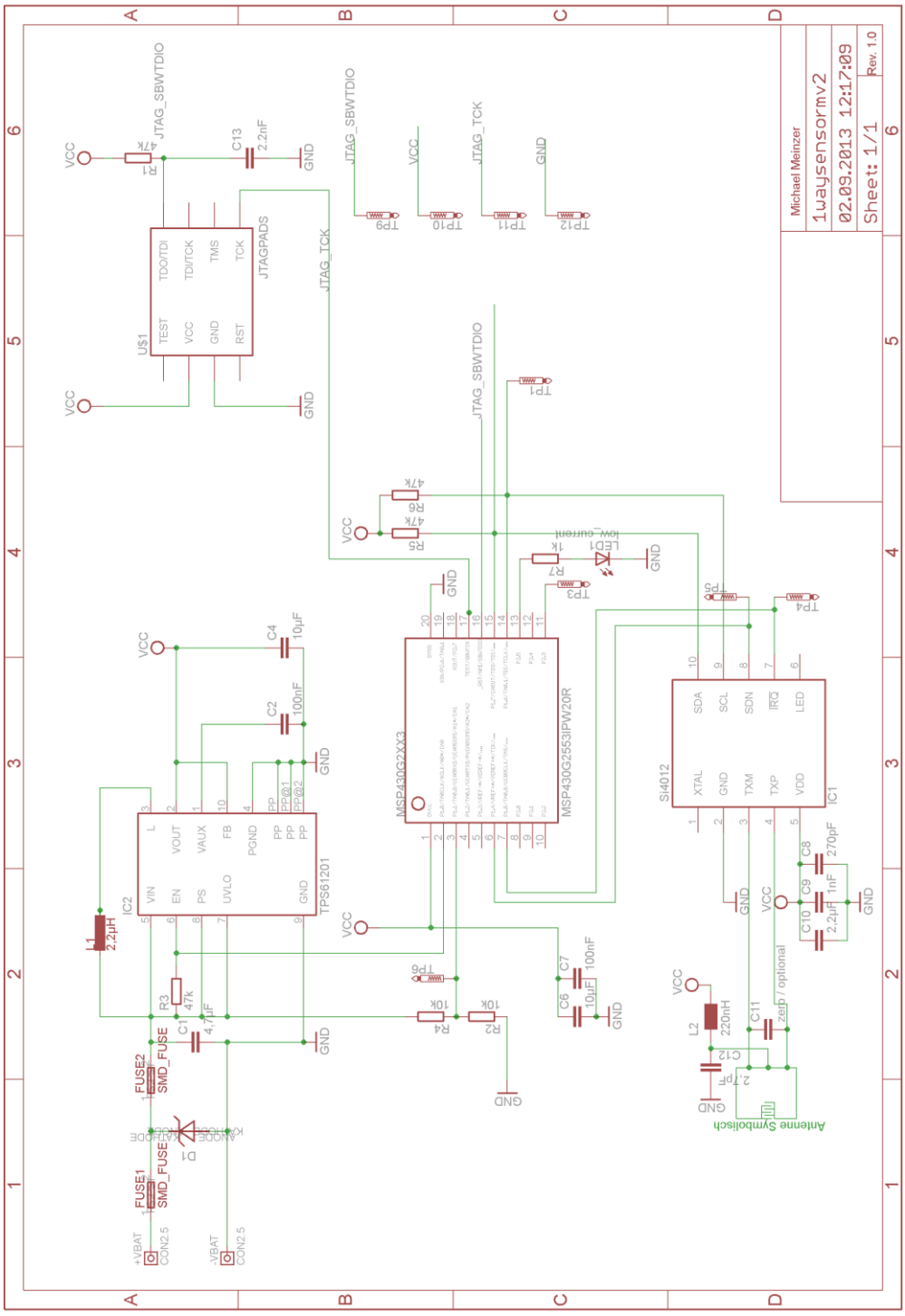

Abb. B.1 Sensorschaltung

B.2 Top-Layer Platine

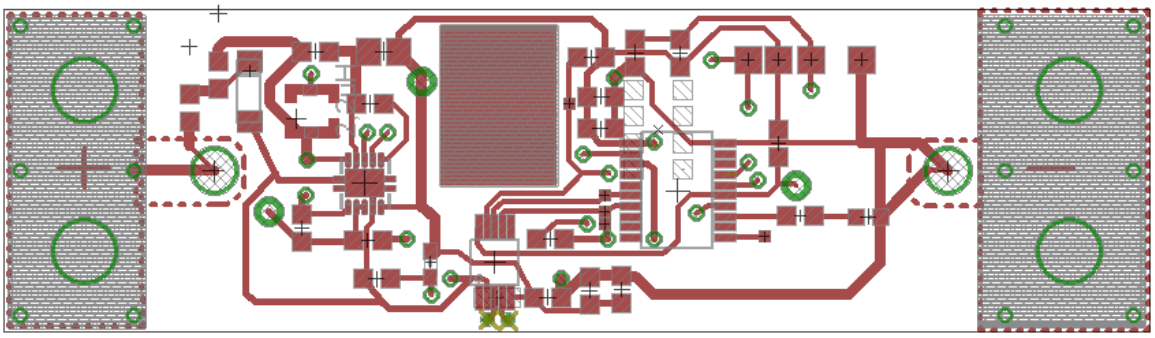

Abb. B.2 Top Layer der Sensorplatine

B.3 Bottom-Layer Platine

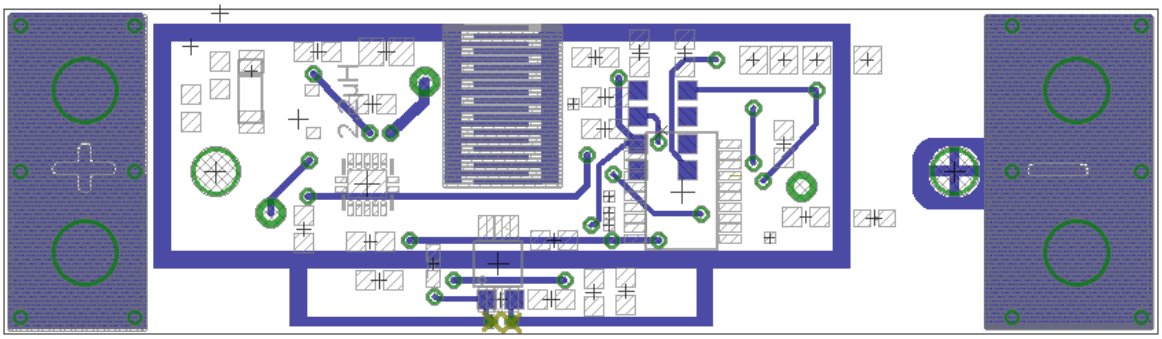

Abb. B.3 Bottom-Layer der Sensorplatine

B.4 Bauteilliste Sensor

Bauteil	Wert / Funktion	Bauform
MSP430G2553	Mikrocontroller	20-Pin TSSOP
SI4012	Transmitter	10-Pin MSOP
TPS61201	Spannungsversorgung	10-Pin SON
C1	4,7 µF	0603
C2	100 nF	0603
C4	10 µF	0805
C6	10 µF	0603
C7	100 nF	0603
C8	270 pF	0603
C9	1 nF	0603
C10	2,2 µF	0603
C11	optional - nicht verwendet	0603
C12	2,7 pF	0603
C13	2,2 nF	0603
LED1	low_current rot	0603
R1	47 k	0603
R2	10 k	0603
R3	47 k	0603
R4	10 k	0603
R5	47 k	0603
R6	47 k	0603
R7	1 k	0603
F1_alt	250 mA	0603
F1_neu	250 mA	0603
F2_alt	200 mA	0603
F2_neu	250 mA	0603
L1	2,2 µH	LPS3015_Package [20]
L2	220 nH	0603
TVS Diode	Überspannungsschutz	DO-221AC [12]

Tabelle B-1 Bauteilliste

C Quellcode

C.1 Sensor

C.1.1 Sensor_0.h

```
/*
 * Sensor_0.h
 *
 *  Created on: 26.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_0_H_
#define SENSOR_0_H_

#define Sensoradresse		0x10
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_0_H_ */
```

C.1.2 Sensor_1.h

```
/*
 * Sensor_1.h
 *
 *  Created on: 22.07.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_1_H_
#define SENSOR_1_H_

#define Sensoradresse		0x01
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_1_H_ */
```

C.1.3 Sensor_2.h

```
/*
 * Sensor_2.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_2_H_
#define SENSOR_2_H_

#define Sensoradresse		0x02
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_2_H_ */
```

C.1.4 Sensor_3.h

```
/*
 * Sensor_3.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_3_H_
#define SENSOR_3_H_

#define Sensoradresse		0x03
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_3_H_ */
```

C.1.5 Sensor_4.h

```
/*
 * Sensor_4.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_4_H_
#define SENSOR_4_H_

#define Sensoradresse		0x04
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_4_H_ */
```

C.1.6 Sensor_5.h

```
/*
 * Sensor_5.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_5_H_
#define SENSOR_5_H_

#define Sensoradresse		0x05
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_5_H_ */
```

C.1.7 Sensor_6.h

```
/*
 * Sensor_6.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_6_H_
#define SENSOR_6_H_

#define Sensoradresse		0x06
#define Voltagecal		0x01
#define Temperaturcal		0x01

#endif /* SENSOR_6_H_ */
```

C.1.8 Sensor_7.h

```
/*
 * Sensor_7.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_7_H_
#define SENSOR_7_H_

#define Sensoradresse           0x07
#define Voltagecal              0x01
#define Temperaturcal           0x01

#endif /* SENSOR_7_H_ */
```

C.1.9 Sensor_8.h

```
/*
 * Sensor_8.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_8_H_
#define SENSOR_8_H_

#define Sensoradresse           0x08
#define Voltagecal              0x01
#define Temperaturcal           0x01

#endif /* SENSOR_8_H_ */
```

C.1.10 Sensor_9.h

```
/*
 * Sensor_9.h
 *
 *  Created on: 02.08.2013
 *      Author: Michael
 *      Headerfile mit den Kalbierkoeffizienten und der Sensoradresse
 *      Parameterauslagerung für ein einfacheres Handling zwischen den Sensoren
 */

#ifndef SENSOR_9_H_
#define SENSOR_9_H_

#define Sensoradresse           0x09
#define Voltagecal              0x01
#define Temperaturcal           0x01

#endif /* SENSOR_9_H_ */
```

C.1.11 mainheader.h

```
/*
 * mainheader.h
 *
 *  Created on: 22.07.2013
 *      Author: Michael
 */

#ifndef MAINHEADER_H_
#define MAINHEADER_H_

#include <stdint.h>
#include <msp430g2553.h>

/*
 * Definition für I2C
 * Übernommen von Masterarbeit Jegenhorst
 */
        #define UCB0BR_BIT_CLK_100          200.0                       // UCB0 Bit Clock [kHz]
        #define SMCLK                       1000.0                      // SubMain Clock (from DCO) [kHz]
        #define UCB0BR_0_SET_100  ((uint16_t)( SMCLK / UCB0BR_BIT_CLK_100))

/*
```

```
* Allgemeine Makros und Enums
*/
        // TX 433 On/Off Handling
        #define TX433_ON (P1OUT &= ~BIT4)
        #define TX433_OFF (P1OUT |= BIT4)

        // LED Handling
        #define LED_ON (P2OUT |= BIT5)
        #define LED_OFF (P2OUT &= ~BIT5)
        #define LED_toggle (P2OUT ^= BIT5)

        // Messpin Handling
        #define Messpin_ON (P2OUT |= BIT3)
        #define Messpin_OFF (P2OUT &= ~BIT3)
        #define Messpin_toggle (P2OUT ^= BIT3)

        // Enableeingang des DCDC Wandlers
        #define DCDC_ON (P1OUT |= BIT0)
        #define DCDC_OFF (P1OUT &= ~BIT0)
        #define DCDC_Toggle (P1OUT ^= BIT0)

        // TX Frame Informationen
        // gemäss neuem Frame Maximallänge 72 Bit / ohne RUN In SOF 60 Bit
        // Makronamen von der Masterarbeit Jegenhorst übernommen

        #define     TX433_FRAME_BYTES                   8       // TX data bytes = 60 / 8 + 0.5 Nur ganzzahlige Bytes..
        #define     TX433_FRAME_BYTES_MCH               18      // Manchester-coded TX bytes SOF + RunIn + Framebytes ((4 + 8 + 60)*2(BIT)) / 8
                                                                //              = (TX433_FRAME_BITS*2) / 8
        #define TX433_FRAME_BYTES_MCH_half              9       // die hälfte der Bytes für die gesplittete Übertragung der Daten an den Transmitter

        #define TX433_LIVEBIT                           0x10    // Kennzeichnung für ein Liveframe
        #define TX433_BUFFERED                          0x00    // Kennzeichnung für ein Bufferedframe

        // Buffered Frame Setup
        #define TX433_FR_PARA_BYTE0                     0       // obere 4 Bit Parameter / untere 4 Bits Errorcodes
        #define TX433_FR_ADDR_BYTE0                     1       // Volle 8 Bit für die Adresse
        #define TX433_FR_VOLTAGE_BYTE0                  2       // Zellspannung
        #define TX433_FR_TIMESTAMP24_BYTE               3       // eigentlich bei 3.5 Byte. beim Laden der Werte wird angepasst
        #define TX433_FR_CRC_BYTE0                      6       // angepasst mit Korrekturrechnung von 3.5 Byte..eigentlich 6.5 Byte bis 7.5 Byte

        // Live Frame Setup nur zusätzliche Angaben
        #define TX433_FR_TEMPERATUR_BYTE0               3
        #define TX433_FR_TIMESTAMP16_BYTE               4

                                                                        // Data-Alignment in txdata[]: Verschiebung um 1 da Set_Fifo zuerst
        #define TX433_TX_SOF_BYTE0                      1               // Set FIFO macht die Verschiebung notwendig
        #define TX433_TX_PARA_BYTE0                     4               // Parameter Bytes Start - Verschiebung durch Set FIFO
        #define TX433_TX_ERR_BYTE0                      5               // Error Bytes Start - Verschiebung durch Set FIFO
        #define TX433_TX_ADDR_BYTE0                     6               // Sensoradresse - Verschiebung durch Set FIFO
        #define TX433_TX_DATA_BYTE0                     8               // Zellspannung - Verschiebung durch Set FIFO
        #define TX433_TX_TIMESTAMP_BYTE                 11              // Zeitstempel(Ausgangslage buffered Frame) - Verschiebung durch Set FIFO
        #define TX433_TX_CRC_BYTE0                      17              // CRC Ergebnis - Verschiebung durch Set FIFO


        // Übernommen aus der Arbeit von Puettjer
        // RND SHIFT MAKRO
        #define  SR() sr_r<<=1;sr_r|=(((sr_r >> 28)^(sr_r>>19))&0x00000001);sr_r&=0x0FFFFFFF
        #define TXFAC 3
        #define TXMID 5000

        // Definitionen für die LSB Werte des ADC und untere Grenzspannung
        // Korrektur da Floatingpointoperationen zu teuer sind
        //          #define UBatt_LSB (2.5/1023)
        #define LowVoltage 210                  // Untere Spannungsgrenze für den Transmitter eingeführt für den Zellensimulator
        #define Voltagediff    8                // Spannungsdifferenz in LSB 7 ca. 17mV / 9 ca. 22 mv / 6 ca. 14mv / 20 ca. 48mv / 8 ca. 20mv

        // Queue_Length
        #define Queue_Length 65

        // Zeit pro Messwertaufnahme in die Queue ca. 50ms
        #define queuevaluetime          750     // Wartezeit bis beim Inqueuefall der nächste Wert in die Queue kommt 750 ca. 75ms
        #define queuereturntime 20000           // Werte Aufnahme alle  2s da das RND Makro Werte um 2s generiert
        #define queue_fix_length 30             // Fixe Anzahl an Werte welche im Zeitraster von 50ms in die Queue kommen bevor erneut geprüft wird

        // Sendewiederholungsbit
        #define Sendewdh 0x01


        // Enum für Schleifen und Status
        enum boolean {
                        FALSE = 0x00,
                        TRUE  = 0x01
        };

        // Definition Transmitter Statemachine
        typedef enum{
                IDLE,
                Standby,
                Frameload,
                Transmit,
                wait,
                Transmit_crc,
                Transmit_man,
                Transmit_data_1,
                Transmitter_off,
                Transmitter_on
        }transmitstate;

        // Definition ADC Statemachine
        typedef enum{
                ADC_busy_voltage,
```

```
                ADC_busy_temperature,
                ADC_idle,
                ADC_inqueuewait
        }adcstates;

        // Queue-Struct
        typedef struct {
                uint8_t timelabel24_1;          // oberer Teil des 24 Bit Timers
                uint8_t timelabel24_2;          // mittlerer Teil des 24 Bit Timers
                uint8_t timelabel24_3;          // unterer Teil des 24 Bit Timers
                uint8_t voltage12_1;            // oberer Teil des 12 Bit Spannungswertes
                uint8_t voltage12_2;            // unterer Teil des 12 Bit Spannungswertes und 4 Bit Status Sendewdh zb.
        }queue;

        //      Return-Codes
        #define EXIT_SUCCESS        0x01
        #define EXIT_FAILURE        0x00

// Funktionen

// Funktionen für die Initalisierung
void initFrequency(void);
void initGPIO(void);
void initI2C(void);
void initADC(void);
void initTimerA0(void);

// Funktionen für das Übertragungsframe
void frame_tx433_init(void);
void frame_tx433_code_manchester(void);
void frame_tx433_calc_crc(void);
void frame_tx433_buffered(void);
void frame_tx433_live(void);

#endif /* MAINHEADER_H_ */
```

C.1.12 main.c

```
/*
 * main.c
 */

#define EXTERN

#include <msp430g2553.h>

#include "mainheader.h"
#include "global.h"
#include "Sensors\Sensor_3.h"
#include "tx_433.h"
#include "adc.h"
#include "queue.h"

int main(void) {
   WDTCTL = WDTPW | WDTHOLD;        // Stop watchdog timer


   // Initalisierung der Grundfunktionen der Hardware
   initFrequency();
   initGPIO();
   initI2C();
   initADC();
   initTimerA0();

   // INIT sr_r
   sr_r = (Sensoradresse | 0x08000000);

   initqueue();

   // TODO Framefehlermessung
   // Counter welcher im Testfall in jedem Liveframe inkrementiert wird
   // framecounter = 0;

   // Ausschalten des TX
   TX433_OFF;
   // Warten bis off
   while((P1IN & BIT5) != BIT5);
   // wiedereinschalten
   TX433_ON;

   // Frame Initalisieren
   frame_tx433_init();

   // Zwangspause derzeit bis der TX reagiert
   __delay_cycles(250000);
   LED_OFF;

   // Messen der Spannung vor dem Tx_Init damit Zeit vergeht bis der ADC fertig ist.
   sample_voltage_batt();
   tx433_init();

   // Init der Variablen damit die Startbedingungen gegeben sind
   tmp = ADC10MEM;
   battvoltage_old = tmp;
   readindex = 0;
   writeindex = 0;
```

```
    tx_status = 0;

    // Set timervalue to zero
    timer24 = 0;
    timervalue = 0;


    // Init der States auf IDLE
    transmit = IDLE;
    ADC = ADC_idle;

    // Statischer Zustand
    Messpin_OFF;
    LED_OFF;

    // Generate First Rnd Sendtime
    // TODO RND SHIFT MAKRO
    //wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 3;          // slow im Mittel 4 Sekunden
    wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 2 ;          // slow im Mittel 2 Sekunden...
    SR();
    // Globale Interrupts zulassen
    // Da Timer läuft
  _EINT();

      while(1){

              // Alle Aktionen finden in der Timer_isr.c statt

              // VoltageCal-Loop mit _Delay_cycles realisiert
              // Ablauf
              // warten 50ms / Dann Spg messen / Dann Werte ins Frame / Daten an Transmitter / Transmitter senden lassen / warten 50ms...
//            __delay_cycles(2000000);                  // Sollten 50ms entsprechen bei 4 MHz
//            sample_voltage_batt();                    // ADC antriggern
//            __delay_cycles(2000);
//            battvoltage += ADC10MEM;                  // ADC Wert ins Framepuffer
//            sample_voltage_batt();                    // ADC antriggern
//            __delay_cycles(2000);
//            battvoltage += ADC10MEM;
//            battvoltage >>= 1;
//            frame_tx433_live();                       // Wert ins Frame
//            frame_tx433_calc_crc();                   // CRC über das Frame
//            frame_tx433_code_manchester();            // Manchestercodierung der Daten
              // Messpin_ON;
//            tx433_cmd_fifo_set();                     // Frame an den Transmitter senden
//            tx433_cmd_tx_start();                     // Transmitter Commando fürs Senden schicken
              // Messpin_OFF;
//            LED_toggle;
//            battvoltage = 0;


    }
}
```

C.1.13 init.c

```
/*
 * init.c
 *
 *  Created on: 22.07.2013
 *      Author: Michael
 *      Sourcefile für die Initalisierung des MSP430G2553
 */

#include <msp430g2553.h>
#include "mainheader.h"
#include "global.h"

// Frequenzeinstellungen
// 4MHz Mainclk und 1000kHz SMCLK
// Quelle und Ablauf der Frequenzeinstellungen Gracetool von TI
void initFrequency(){

              /*
    * Basic Clock System Control 2
    *
    * SELM_0 -- DCOCLK
    * DIVM_1 -- Divide by 2
    * ~SELS -- DCOCLK
    * DIVS_3 -- Divide by 8
    * ~DCOR -- DCO uses internal resistor
    *
    * Note: ~<BIT> indicates that <BIT> has value zero
    */
              // Quelle für MCLK -> DCOCLK / Teilung 2 bei 8 MHz / Quelle für SMCLK pauschal auf DCOCLK(alternativen nur Extern) / Teilung 8 bei 8 MHz
    BCSCTL2 = SELM_0 | DIVM_1 | DIVS_3;

    // Quelle Gracetool
    if (CALBC1_8MHZ != 0xFF) {
        /* Adjust this accordingly to your VCC rise time */
        __delay_cycles(100000);

        // Follow recommended flow. First, clear all DCOx and MODx bits. Then
        // apply new RSELx values. Finally, apply new DCOx and MODx bit values.
        DCOCTL = 0x00;
        BCSCTL1 = CALBC1_8MHZ;      /* Set DCO to 8MHz */
        DCOCTL = CALDCO_8MHZ;
```

```
  }
  /*
   * Basic Clock System Control 1
   *
   * XT2OFF -- Disable XT2CLK
   * ~XTS -- Low Frequency
   * DIVA_0 -- Divide by 1
   *
   * Note: ~XTS indicates that XTS has value zero
   */
  BCSCTL1 |= XT2OFF | DIVA_3;

  /*
   * Basic Clock System Control 3
   *
   * XT2S_0 -- 0.4 - 1 MHz
   * LFXT1S_0 -- If XTS = 0, XT1 = 32768kHz Crystal ; If XTS = 1, XT1 = 0.4 - 1-MHz crystal or resonator
   * XCAP_1 -- ~6 pF
   */
  //BCSCTL3 = XT2S_0 | LFXT1S_0 | XCAP_1;
  /*
          //DCOCTL  = CALBC1_1MHZ;
          DCOCTL  = CALBC1_8MHZ;
          // XT2 aus, 1 MHz, sowie Taktteiler für ACLK auf 1 Voller Takt auf der ACLK
          BCSCTL1 = (XT2OFF | CALBC1_1MHZ | DIVA_0);
          // MCLK = DCOCLK, div MCLK = X, SMCLK = DCOCLK, div SMCLK = X
          BCSCTL2 = SELM_0 | DIVM_0 | DIVS_1;
          */

}
// Portinitalisierung
void initGPIO(){

          /* Port 1 */
          P1OUT = BIT4 + BIT0;          // GPIO Pin 1.4 für SDN Pin vom SI4012
          // Port Selection 1 für Pin 1.6 / 1.7 beide Selectionregister auf 1 für die Funktion als I2C
          P1SEL = BIT6 + BIT7;
          P1SEL2 = BIT6 + BIT7;
          // Direction Pin 1.6 / 1.7 werden vom UCI konfiguriert
          P1DIR = BIT4 + BIT0;          // BIT 4 = SDN Funktion des SendeIC´s
          //Save Line
          DCDC_ON;
          // Port 1 Interrupt Edge Select Register
          P1IES = 0;
          // Port 1 Interrupt Flag Register
          P1IFG = 0;
          /* Port 2 */
          P2OUT = BIT3 + BIT5;          // GPIO Pin 2.3 für Messpin & GPIO 2.5 für LED als Ausgang
          P2DIR = BIT3 + BIT5;          // GPIO Pin 2.3 für Messpin & GPIO 2.5 für LED als Ausgang
}
// I2C Einstellungen übernommen von Masterarbeit Jegenhorst
void initI2C(){

          // first software reset
          UCB0CTL1 |= UCSWRST;
          // Own adress 7-bit, slave adress 7-bit, single master, I2C-mode, synch mode
          UCB0CTL0  = (UCMST | UCMODE_3 | UCSYNC);
          // Clock source = SMCLK
          UCB0CTL1  = (UCSSEL_2 | UCSWRST);
          // Baud rate = 100kbps, f_bitclk = f_brclk / UCB0RX = 100kHz,
          //          f_brclk = SMCLK = 0.5MHz
          //          --> UCB0RX = 5
          UCB0BR0 = UCB0BR_0_SET_100;       // low byte, UCB0RX = (UCB0R0 + UCB0R1 x 128)
          UCB0BR1 = 0x00;                   // high byte
          // IRQ Flags UCNACKIFG, UCSTPIFG, UCSTTIFG, UCALIFG clears by sw-reset
          // I2C own 7-bit address = 0x7B (123)
          UCB0I2COA |= 0x007B;
          // no IRQs enabled for NACK, STOP, START, ARB
          UCB0I2CIE = 0x00;
}
// ADC Grundeinstellungen die genauen Einstellungen erfolgen in der ADC.c für die entsprechende Messung
void initADC(){

          // ADC10CTL0 Registersetup
          // Referenz auf interne Referenz umschalten
          ADC10CTL0 |= SREF_1;
          // Ref- Spg. 2.5 V
          ADC10CTL0 |= REF2_5V;
          // Referenz Generator an
          ADC10CTL0 |= REFON;
          // Sample and Hold time für Temperaturmessung verlängert. Der ADC benötigt dort 30µS für die Messung.
          ADC10CTL0 |= ADC10SHT_3;

          // ADC10CTL1 Registersetup
          // SHSx Sample and Hold Source Select
          ADC10CTL1 |= SHS_0;
          // ADC10DIV Clock divider
          ADC10CTL1 |= ADC10DIV_5;
          // ADC10SSEL Clock Source select
          ADC10CTL1 |= ADC10SSEL_0;
          // Conversion sequence mode select Singlemode.
          ADC10CTL1 |= CONSEQ_0;

          // ADC "einschalten"
          ADC10CTL0 |= ADC10ON;
}
// TimerA Init und ISR Anmeldung
void initTimerA0(){

          // Funktion mit SMCLK gegeben, die ACLK ist nicht lauffähig
          // Reset First
```

```
	TACTL = TACLR;
	// SMCLK + Inputclockteiler = 1 + Upmode
	TACTL = TASSEL_2 | ID_0 | MC_1;
	// Comparemode mit Interrupt Enable
	TACCTL0 = CCIE;
	// SMCLK läuft mit 1000kHz
	// 100 entsprechen im 100µS Kontrolle durch TogglePin in der Timerroutine
	// 50 entsprechen im 50µS Kontrolle durch TogglePin in der Timerroutine
	TACCR0 = 100;
}
```

C.1.14 global.h

```
/*
 * global.h
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 */

#ifndef GLOBAL_H_
#define GLOBAL_H_

#ifndef EXTERN
		#define EXTERN extern
	#endif

		// Struct zu grossen Teilen von Masterarbeit Jegenhorst übernommen
		struct {
			uint8_t		return_status;
			uint8_t		state;
			uint8_t		idlemode;
			uint8_t		actdatatsize;
			uint8_t		framedata[TX433_FRAME_BYTES];
			uint8_t		txdata[TX433_FRAME_BYTES_MCH+1];  // + 1Byte for I2C Command
		}EXTERN volatile g_tx433;

		EXTERN volatile uint16_t battvoltage_old;		// alter Spannungswert ADC zum Vergleich
		EXTERN volatile uint16_t battvoltage;		// Spannungswert fürs Liveframe
		EXTERN volatile uint16_t tmp;		// Spannungswert
		EXTERN volatile uint16_t temperatur;		// Temperaturwer fürs Liveframe
		EXTERN volatile uint16_t queuewait_counter;		// Counter für Inqueue Spannungsmessung
		EXTERN volatile transmitstate transmit;		// Statemachine Transmitter
		EXTERN volatile adcstates ADC;		// Statemachine ADC
		EXTERN volatile uint32_t timer24;		// TimerTicks in TimerISR
		EXTERN volatile uint32_t timervalue;		// Speicher für Zeitzwischenrechnungen

		// Variablen für das zufällige Senden
		EXTERN volatile long sr_r;		// Zufallszahl Startwert
		EXTERN volatile uint16_t wait_rnd_time;		// Counter für zufälliges Senden
		EXTERN volatile uint16_t wait_save_time;		// Speicher für zufälliges Senden
		EXTERN volatile uint16_t framecounter;		// Framecounter für Framefehlertests
		EXTERN volatile uint8_t statusbitarray;		// Statusarray 8 Bit

		// Queue Handling
		EXTERN volatile uint8_t readindex;		// Leseindex Queue
		EXTERN volatile uint8_t writeindex;		// Schreibindex Queue
		EXTERN volatile queue waitingqueue[Queue_Length];		// Array der (struct)Queue
		// Queue Handling 2
		EXTERN volatile uint8_t queue_counter;		// Queuezähler für das Inqueue von 30 aufeinander folgendnen Werten
		EXTERN volatile uint8_t Voltagedifferenz;		// für das dynamische Herabsetzen der Inqueuespannungsdifferenz

		EXTERN volatile uint8_t tx_status;		// Debug Zellensimulator Transmitterstatus 1/0 == aus/an


#endif /* GLOBAL_H_ */
```

C.1.15 adc.h

```
/*
 * adc.h
 *
 *  Created on: 27.06.2013
 *      Author: Michael
 */

#ifndef ADC_H_
#define ADC_H_
// Funktionen

void sample_voltage_batt(void);
void sample_temperature(void);

#endif /* ADC_H_ */
```

C.1.16 adc.c

```
/*
 * adc.c
 *
 *  Created on: 27.06.2013
 *      Author: Michael
 */
#include "mainheader.h"
#include "global.h"
#include "adc.h"

// Spannung messen
void sample_voltage_batt(){

        // ADC Stoppen, damit Konfiguration der ADC Register möglich wird
        ADC10CTL0 &= 0xFFFD;
        // ADC Umkonfigurieren
        ADC10CTL0 |= SREF_1;
        // Ref- Spg. 2.5 V
        ADC10CTL0 |= REF2_5V;
        // Referenz Generator an
        ADC10CTL0 |= REFON;
        // Löschen der Einstellung
        ADC10CTL1 &= 0x0FFF;
        ADC10CTL1 |= INCH_1;
        // Analog Input für A1
        ADC10AE0 |= BIT1;
        ADC10CTL0 |= ADC10ON;
        // ADC "Scharf_schalten"
        ADC10CTL0 |= (ENC + ADC10SC);
}
// Temperatur messen
void sample_temperature(){

        // ADC Stoppen, damit Konfiguration der ADC Register möglich wird
        ADC10CTL0 &= 0xFFFD;
        // Interne Ref setzen
        ADC10CTL0 |= SREF_1;
        // ADC Ref auf 1.5 Volt gesetzt
        ADC10CTL0 &= ~REF2_5V;
        // Inputchannel wechseln auf Temperatursensor
        ADC10CTL1 &= 0x0FFF;
        ADC10CTL1 |= INCH_10;
        ADC10AE0 &= 0x00;
        // ADC "Scharf_schalten"
        ADC10CTL0 |= (ENC + ADC10SC);

}
```

C.1.17 tx_433.h

```
/*
 * tx_433.h
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 */
// Makrodefinition für den SI4012
// Quelle Masterarbeit, Jegenhorst

#include <stdint.h>

#ifndef TX_433_H_
#define TX_433_H_

        #define ADDR_TX433                      0x70            // Address of 433MHz transmitter

        //___ Commands: ____________________________________
        #define TX433_CMD_GET_REV               0x10            // Return product and revision info
        #define TX433_CMD_SET_PROPERTY          0x11            // Set property
```

```
        #define TX433_CMD_GET_PROPERTY          0x12            // Get property
        #define TX433_CMD_LED_CTRL              0x13            // Turn LED on/off
        #define TX433_CMD_CHANGE_STATE          0x60            // Change power state
        #define TX433_CMD_GET_STATE             0x61            // Get state
        #define TX433_CMD_TX_START              0x62            // Start transmission
        #define TX433_CMD_TX_STOP               0x67            // Stop transmission
        #define TX433_CMD_SET_INT               0x63            // Interrupt control
        #define TX433_CMD_GET_INT_STATUS        0x64            // Get interrupt status
        #define TX433_CMD_INIT_FIFO             0x65            // Clear TX FIFO
        #define TX433_CMD_SET_FIFO              0x66            // Store data in TX FIFO
        #define TX433_CMD_GET_BAT_STATUS        0x68            // Get battery status (Vdd voltage)

        //___ Properties: ____________________________________
        #define TX433_PROP_CHIP_CONFIG          0x10    // FSK dev, LSB first, ext. XO
        #define TX433_PROP_LED_INTENSITY        0x11    // LED current drive strength
        #define TX433_PROP_MODULATION_FSKDEV    0x20    // MOD type and FSK dev
        #define TX433_PROP_TUNE_INTERVAL        0x21    // Tuning interval in sec
        #define TX433_PROP_FIFO_THRESHOLD       0x30    // FIFO threshold
        #define TX433_PROP_BITRATE_CONFIG       0x31    // Date rate and ramp if OOK
        #define TX433_PROP_TX_FREQUENCY         0x40    // Carrier if OOK, upper if FSK
        #define TX433_PROP_LBD_CONFIG           0x41    // Low battery voltage threshold
        #define TX433_PROP_XO_CONFIG            0x50    // XO config
        #define TX433_PROP_PA_CONFIG            0x60    // PA config

        //___ Data Values for Properties: (MSB in DATA1) _____
        #define TX433_PROP_CHIP_CONFIG_DATA     0x00    // stand for: intern oscillator,
                                                        //  MSB first
        #define TX433_PROP_CHIP_CONFIG_BCNT     1       // Number of Bytes = 1

        #define TX433_PROP_LED_INTENSITY_DATA   0x03    // stand for: LED current =0.97mA
        #define TX433_PROP_LED_INTENSITY_BCNT   1       // Number of Bytes = 1

        #define TX433_PROP_MODULATION_FSKDEV_DATA       0x0000  // stand for: OOK-Modulation
//      #define TX433_PROP_MODULATION_FSKDEV_DATA       0x013F  // stand for: FSK-Modulation
//                                                                                              //
                biFSKDev = 63
        #define TX433_PROP_MODULATION_FSKDEV_BCNT       2       // Number of Bytes = 2

        #define TX433_PROP_TUNE_INTERVAL_DATA           0x000A  // stand for: 10s tuning interval
        #define TX433_PROP_TUNE_INTERVAL_BCNT           2       // Number of Bytes = 2

        #define TX433_PROP_FIFO_THRESHOLD_DATA  0x701020        // stand for: almost full = 240
                                                                //  almost empt = 16
                                                                //  auto tx    = 32
        #define TX433_PROP_FIFO_THRESHOLD_BCNT  3               // Number of Bytes = 3

        #define TX433_PROP_BITRATE_CONFIG_DATA  0x006402        // stand for: 10.000bps,
//      #define TX433_PROP_BITRATE_CONFIG_DATA  0x01F402        // stand for: 50.000bps,
                                                                //  ramp rate = 2us
        #define TX433_PROP_BITRATE_CONFIG_BCNT  3               // Number of Bytes = 3

        #define TX433_PROP_TX_FREQUENCY_DATA    0x19DDC7C8      // stand for: 433.965.000 Hz
        #define TX433_PROP_TX_FREQUENCY_BCNT    4               // Number of Bytes = 4

        #define TX433_PROP_LBD_CONFIG_DATA      0x09C4003C      // stand for: threshold = 2500mV
                                                                //  interval  = 60s
        #define TX433_PROP_LBD_CONFIG_BCNT      4               // Number of Bytes = 4

                                                                // TX433_PROP_PA_CONFIG_DATA = 0x014600807D7F
                                                                // splitted into 3 parts, stands for:
        #define TX433_PROP_PA_CONFIG_DATA_P2    0x0146          // max current drive,
                                                                // PA level    = 70
//      #define TX433_PROP_PA_CONFIG_DATA_P2    0x0023          // limit current drive
                                                                // PA level    = 35

        #define TX433_PROP_PA_CONFIG_DATA_P1    0x0080          // PA cap      = 128,
        #define TX433_PROP_PA_CONFIG_DATA_P0    0x7D7F          // fAlphaSteps = 125,
                                                                // fBetaSteps  = 127,
        #define TX433_PROP_PA_CONFIG_BCNT       6               // Number of Bytes = 6

        //___ States & IdleModes: ____________________________
        #define TX433_STATE_IDLE                0x00
        #define TX433_STATE_SHUTDOWN            0x01
        #define TX433_STATE_TX                  0x10
        #define TX433_IDLEMODE_STANDBY          0x00
        #define TX433_IDLEMODE_SENSOR           0x01
        #define TX433_IDLEMODE_TUNE             0x02

        //___ DataTransmission Mode: _________________________
        #define TX433_DTMOD_FIFO                0x00
        #define TX433_DTMOD_CW                  0x01
        #define TX433_DTMOD_PN90                0x02
        #define TX433_DTMOD_PN91                0x03

        //___ Interrupts: ____________________________________
        #define TX433_IRQ_FIFO_UFLOW            0x80
        #define TX433_IRQ_FIFO_OFLOW            0x10
        #define TX433_IRQ_FIFO_AEMTY            0x20
        #define TX433_IRQ_FIFO_AFULL            0x40
        #define TX433_IRQ_PACKET_SENT           0x08
        #define TX433_IRQ_LOW_BAT               0x04
        #define TX433_IRQ_TUNE                  0x02
        #define TX433_IRQ_POR                   0x01

        //___ Return Status: _________________________________
        #define TX433_RESTATUS_CTS_OK           0x80

        //--- Prototyps: ------------------------------------------------
        uint8_t     tx433_init(void);
        uint8_t tx433_set_prop(const uint8_t);
        uint8_t tx433_get_prop(const uint8_t);
```

```
uint8_t tx433_cmd_tx_start(void);
uint8_t tx433_cmd_tx_stop(void);
uint8_t tx433_cmd_led_ctrl(const uint8_t);
uint8_t tx433_cmd_fifo_init(void);
uint8_t tx433_cmd_fifo_set(void);
uint8_t tx433_cmd_set_int(void);
uint8_t tx433_cmd_get_int_status(void);
uint8_t tx433_cmd_get_state(void);
uint8_t tx433_cmd_change_state(void);

uint8_t tx433_cmd_fifo_set_part1(void);
uint8_t tx433_cmd_fifo_set_part2(void);


#endif /* TX_433_H_ */
```

C.1.18 tx_433.c

```
/*
 * tx_433.c
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 *      Quelle Masterarbeit Jegenhorst
 */
#include "mainheader.h"
#include "global.h"
#include "tx_433.h"
#include "i2c_bus.h"

//--- Function: tx433_init() ------------------------------------------------------
uint8_t tx433_init(void) {

        uint8_t ok;

//      g_tx433.idlemode = TX433_IDLEMODE_TUNE;                 // --> 370us response to TX
        g_tx433.idlemode = TX433_IDLEMODE_STANDBY;              // --> 6.6ms response to TX
        ok = tx433_cmd_tx_stop();
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
        }
        ok = tx433_set_prop(TX433_PROP_CHIP_CONFIG);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
        }
        ok = tx433_set_prop(TX433_PROP_TUNE_INTERVAL);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }
        ok = tx433_set_prop(TX433_PROP_MODULATION_FSKDEV);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }

        ok = tx433_set_prop(TX433_PROP_TX_FREQUENCY);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }
        ok = tx433_set_prop(TX433_PROP_PA_CONFIG);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }
        ok = tx433_set_prop(TX433_PROP_BITRATE_CONFIG);
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }
        ok = tx433_cmd_fifo_init();
        if(ok != EXIT_SUCCESS){
                return EXIT_FAILURE;
                }

        return EXIT_SUCCESS;
}
//--- Function: tx433_set_prop() ------------------------------------------------------
uint8_t tx433_get_prop(uint8_t property) {

        // not needed
        return 1;
}
//--- Function: tx433_set_prop() ------------------------------------------------------
uint8_t tx433_set_prop(uint8_t property) {

        uint8_t ok, bytes;
        uint8_t data[8];                                        // max 2+6 byte data to sent

        data[0] = TX433_CMD_SET_PROPERTY;                       // Command = Set_Property
        data[1] = property;                                     // Property ID

                                                                // Splitting of the property data into
                                                                // single bytes is replaced as constants
                                                                // by the compiler.
                                                                // data[2] <-- MSB of property data
                                                                                        // data[n] <-- LSB, nmax = 7
        switch(property) {
                case TX433_PROP_CHIP_CONFIG:
```

```
                                bytes   = TX433_PROP_CHIP_CONFIG_BCNT;
                                data[2] = TX433_PROP_CHIP_CONFIG_DATA;
                                break;
                        case TX433_PROP_LED_INTENSITY:
                                bytes   = TX433_PROP_CHIP_CONFIG_BCNT;
                                data[2] = TX433_PROP_LED_INTENSITY_DATA;
                                break;
                        case TX433_PROP_MODULATION_FSKDEV:
                                bytes   = TX433_PROP_MODULATION_FSKDEV_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_MODULATION_FSKDEV_DATA >> 8);
                                data[3] = (uint8_t)(TX433_PROP_MODULATION_FSKDEV_DATA & 0xFF);
                                break;
                        case TX433_PROP_TUNE_INTERVAL:
                                bytes   = TX433_PROP_TUNE_INTERVAL_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_TUNE_INTERVAL_DATA >> 8);
                                data[3] = (uint8_t)(TX433_PROP_TUNE_INTERVAL_DATA & 0xFF);
                                break;
                        case TX433_PROP_FIFO_THRESHOLD:
                                bytes   = TX433_PROP_FIFO_THRESHOLD_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA >> 16);
                                data[3] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA >> 8);
                                data[4] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA & 0xFF);
                                break;
                        case TX433_PROP_BITRATE_CONFIG:
                                bytes = TX433_PROP_BITRATE_CONFIG_BCNT;
                                data[2] = (uint8_t)((uint32_t)TX433_PROP_BITRATE_CONFIG_DATA >> 16);
                                data[3] = (uint8_t)(TX433_PROP_BITRATE_CONFIG_DATA >> 8);
                                data[4] = (uint8_t)(TX433_PROP_BITRATE_CONFIG_DATA & 0xFF);
                                break;
                        case TX433_PROP_TX_FREQUENCY:
                                bytes = TX433_PROP_TX_FREQUENCY_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 24);
                                data[3] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 16);
                                data[4] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 8);
                                data[5] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA & 0xFF);
                                break;
                        case TX433_PROP_LBD_CONFIG:
                                bytes = TX433_PROP_LBD_CONFIG_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 24);
                                data[3] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 16);
                                data[4] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 8);
                                data[5] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA & 0xFF);
                                break;
                        case TX433_PROP_PA_CONFIG:
                                bytes = TX433_PROP_PA_CONFIG_BCNT;
                                data[2] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P2 >> 8);
                                data[3] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P2 & 0xFF);
                                data[4] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P1 >> 8);
                                data[5] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P1 & 0xFF);
                                data[6] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P0 >> 8);
                                data[7] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P0 & 0xFF);
                                break;
                        default:
                                return EXIT_FAILURE;
                }
                bytes = bytes + 2;                                              // add 2 for command and property

                ok = i2c_bus_write(ADDR_TX433, bytes, data);        // Write Data
                if(ok != EXIT_SUCCESS)
                        return EXIT_FAILURE;

                ok = i2c_bus_read(ADDR_TX433, 1, data);                         // Read Reply
                if(ok != EXIT_SUCCESS)
                        return EXIT_FAILURE;
                g_tx433.return_status = data[0];

                return EXIT_SUCCESS;
}
//--- Function: tx433_cmd_tx_start() ------------------------------------------------------
uint8_t tx433_cmd_tx_start(void) {

                        // Time to transmit the Start Command over SMBUS ~ 1.14ms @ 100kbit/s
                        // Time to TX after received Start Command = 370us @ Idlemode = Tune
                        // = 6.6ms @ Idlemode = Standby

                uint8_t ok;
                uint8_t data[6];

                data[0] = TX433_CMD_TX_START;                           // Command = TX_START
                data[1] = 0x00;                                         // PacketSize[15:8]
                data[2] = 0x13; // 1b                                   // PacketSize[ 7:0]
                                                                        //= TX433_FRAME_BYTES_MCH = 27
                data[3] = (0x02 & g_tx433.state);                       // Sensor State,
                                                                        // FIFO Auto-TX disable
                data[4] = g_tx433.idlemode;                             // Idle Mode
//              data[5] = TX433_DTMOD_CW;                               // DataTransmission Mode = CW
                data[5] = TX433_DTMOD_FIFO;                             // DataTransmission Mode = FIFO

                ok = i2c_bus_write(ADDR_TX433, sizeof(data), data);     // Write Data
                if(ok != EXIT_SUCCESS){
                        return EXIT_FAILURE;
                        }else {
                                return EXIT_SUCCESS;
                        }
}
//--- Function: tx433_cmd_tx_stop() -------------------------------------------------------
uint8_t tx433_cmd_tx_stop(void) {

                uint8_t ok;
                uint8_t data[3];
```

```
	data[0] = TX433_CMD_TX_STOP;				// Command = TX_STOP
	data[1] = TX433_STATE_IDLE;				// Sensor State = Idle
	data[2] = g_tx433.idlemode;				// Idle Mode


	ok = i2c_bus_write(ADDR_TX433, sizeof(data), data);	// Write Data
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;

	ok = i2c_bus_read(ADDR_TX433, 1, data);			// Read Reply
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;
	g_tx433.return_status = data[0];

	return EXIT_SUCCESS;
}
//--- Function: tx433_cmd_fifo_init() ------------------------------------------------
uint8_t tx433_cmd_fifo_init(void) {

	uint8_t ok;
	uint8_t data[1];

	data[0] = TX433_CMD_INIT_FIFO;				// Command = Init_FIFO

	ok = i2c_bus_write(ADDR_TX433, sizeof(data), data);	// Write Data
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;

	ok = i2c_bus_read(ADDR_TX433, 1, data);			// Read Reply
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;
	g_tx433.return_status = data[0];

	return EXIT_SUCCESS;
}
//--- Function: tx433_cmd_fifo_set() -------------------------------------------------
uint8_t tx433_cmd_fifo_set(void) {

	uint8_t ok;
	uint8_t data[1];					// Write Data
	ok = i2c_bus_write(ADDR_TX433, TX433_FRAME_BYTES_MCH+1, (uint8_t *) g_tx433.txdata);
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;

	ok = i2c_bus_read(ADDR_TX433, 1, data);			// Read Reply
	if(ok != EXIT_SUCCESS)
		return EXIT_FAILURE;
	g_tx433.return_status = data[0];
	return EXIT_SUCCESS;
}

uint8_t tx433_cmd_fifo_set_part1(void){

		uint8_t ok;
		// Write Data
		ok = i2c_bus_write(ADDR_TX433, TX433_FRAME_BYTES_MCH+1, (uint8_t *) g_tx433.txdata);
		if(ok != EXIT_SUCCESS)
			return EXIT_FAILURE;
		return EXIT_SUCCESS;
}
```

C.1.19 queue.h

```
/*
 * queue.h
 *
 *  Created on: 21.08.2013
 *      Author: Michael
 */

#ifndef QUEUE_H_
#define QUEUE_H_

void writequeue(void);
void readqueue(void);
void initqueue(void);
void copyqueue(void);
void clearentry(void);

#endif /* QUEUE_H_ */
```

C.1.20 queue.c

```
/*
 * queue.c
 *
 *  Created on: 21.08.2013
 *      Author: Michael
 */

// Handling mit der Queue

#include "mainheader.h"
#include "global.h"
#include "queue.h"

// Init der Queue mit 0 Werten
void initqueue(){

		// Freerun des Randomshift Makros und init Queue
		for (timer24 = 0;timer24 < Queue_Length;timer24++){
		   SR();
		   waitingqueue[timer24].timelabel24_1 = 0;
		   waitingqueue[timer24].timelabel24_2 = 0;
		   waitingqueue[timer24].timelabel24_3 = 0;
		   waitingqueue[timer24].voltage12_1 = 0;
		   waitingqueue[timer24].voltage12_2 = 0;
		   }
		timer24 = 0;

 }
// Schreiben in die Queue an die nächste verfügbare Queueposition
void writequeue(){

		waitingqueue[writeindex].timelabel24_1 = (uint8_t) (timer24 >> 16UL);	// 16UL - 16Bit System und LongValue
		waitingqueue[writeindex].timelabel24_2 = (uint8_t) (timer24 >> 8);
		waitingqueue[writeindex].timelabel24_3 = (uint8_t) timer24;
		// Voltagevalue
		waitingqueue[writeindex].voltage12_1 = (uint8_t) (tmp >> 4);
		waitingqueue[writeindex].voltage12_2 = (uint8_t)(tmp << 4);
		//waitingqueue[writeindex].voltage12_2 |= Sendewdh;		// WDH Bit setzen
		writeindex++;

		// Reset des Schreibindexes
		if(writeindex >= Queue_Length){
		writeindex = 0;
		// Überschreibschutz Write
		if ((statusbitarray & 0x40) == 1){
			statusbitarray &= 0xB0;
		} else {
			statusbitarray |= 0x40;
		}

		}
}
 void readqueue(){
 }

 // Kopieren des aktuellen Queuewertes an die nächste verfügbare Schreibposition und löschen des WDH Bits
 void copyqueue(){

		waitingqueue[writeindex].timelabel24_1 = waitingqueue[readindex].timelabel24_1;
		waitingqueue[writeindex].timelabel24_2 = waitingqueue[readindex].timelabel24_2;
		waitingqueue[writeindex].timelabel24_3 = waitingqueue[readindex].timelabel24_3;
		waitingqueue[writeindex].voltage12_1 = waitingqueue[readindex].voltage12_1;
		waitingqueue[writeindex].voltage12_2 = waitingqueue[readindex].voltage12_2 & 0xF0; // Wdh Bit löschen
 }
 void clearentry() {
		waitingqueue[readindex].timelabel24_1 = 0;
		waitingqueue[readindex].timelabel24_1 = 0;
		waitingqueue[readindex].timelabel24_1 = 0;
		waitingqueue[readindex].voltage12_1 = 0;
		waitingqueue[readindex].voltage12_2 = 0;
}
```

C.1.21 frame_tx.c

```
/*
 * frame_tx.c
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 *      Quelle Masterarbeit Jegenhorst
 */


#include "Sensors\Sensor_3.h"                                   // Enthält die Sensoradresse
#include "mainheader.h"
#include "global.h"
#include "queue.h"
#include <stdint.h>

#define TX433_CMD_SET_FIFO                      0x66            // Store data in TX FIFO

/*
 * Initalisierung des Datenframes und des TX-Übertragungsframes
 */
void frame_tx433_init(void){

        uint8_t i;
        g_tx433.framedata[0] = 0x00;                            // setzen der Parameter und Errorcodes auf Default 00
        g_tx433.framedata[1] = Sensoradresse;                   // Sensoradresse

        for ( i = 2; i < TX433_FRAME_BYTES; i++){               // Daten und CRC Prüfsumme auf 0 setzen
                g_tx433.framedata[i] = 0x00;
        }
        // TX Übertragungsframe
        g_tx433.txdata[0] = TX433_CMD_SET_FIFO;                 // Command zum schreiben in den Fifo
        g_tx433.txdata[TX433_TX_SOF_BYTE0]              = 0xFF; // RUN-IN Seq 1111 1111 Manchestercoded
        g_tx433.txdata[TX433_TX_SOF_BYTE0 + 1]          = 0x55; // Synch-Seq 0101 0101 --> "0000" Manchestercoded
        g_tx433.txdata[TX433_TX_SOF_BYTE0 + 2]  = 0x56;         // Synch-Seq 0101 0110 --> "0001" Manchestercoded

        // Die anderen Datenbytes werden von frame_code_manchester codiert
}

// Übernommen aus Jegenhorst Masterarbeit
void frame_tx433_code_manchester(void) {

        uint8_t byte_fr, byte_tx;
        uint8_t shift_tx;
        uint8_t shift_fr;

        byte_tx = TX433_TX_PARA_BYTE0;                          // set startbyte after SOF bytes
                                                                // run through bytes of framedata:
        for(byte_fr = TX433_FR_PARA_BYTE0; byte_fr < TX433_FRAME_BYTES; byte_fr++) {
                g_tx433.txdata[byte_tx] = 0x00;                 // clear databyte
                shift_tx = 0x80;                                // shift to MSB first
                                                                // run through 8-bit dataword:
                for(shift_fr = 0x80; shift_fr > 0x00; shift_fr >>= 1) {
                        if(g_tx433.framedata[byte_fr] & shift_fr) {     // if value = 1 --> 1/0 trans
                                g_tx433.txdata[byte_tx] |= shift_tx;    // shift in "10"
                        }
                        else {                                  // if value = 0         --> 0/1 trans
                                g_tx433.txdata[byte_tx] |= (shift_tx >> 1);     // shift in "01"
                        }
                        if(shift_fr == 0x10) {
                                byte_tx++;                      // goto next tx databyte
                                shift_tx = 0x80;                // shift to MSB first
                                g_tx433.txdata[byte_tx] = 0x00; // clear databyte
                        }
                        else {
                                shift_tx >>= 2;                 // goto next pair
                        }
                }
                byte_tx++;                                      // goto next tx databyte
        }
}

// Übernommen aus Jegenhorst Masterarbeit mit Anpassungen
void frame_tx433_calc_crc(void) {

                // This function calculates the CRC checksum over the frame, in dependence of
                // the function "parity()" from the firmware of the old sensor.

        uint8_t i;
        uint8_t tmp;
        tmp = 0x00;                                             // Start mit Generatorpolynom = 0

        // Schleife über die ersten 6 Bytes des Frames..
        // XOR Byteweise mit Framedata verknüpft
        for(i = 0; i < TX433_FRAME_BYTES-1; i++){
                tmp ^= g_tx433.framedata[i];
        }
        // CRC Wert nach g_tx433.framedata.crc schreiben
        g_tx433.framedata[TX433_FR_CRC_BYTE0] = (tmp >> 4) + g_tx433.framedata[TX433_FR_TIMESTAMP24_BYTE + 3];
        g_tx433.framedata[TX433_FR_CRC_BYTE0 + 1] = (tmp << 4);
}

// Funktionen welche die Daten ins Frame lädt :
void frame_tx433_buffered(void){

        uint8_t i;
        // Reset für Bitshifting notwendig
```

```
	for(i = TX433_FR_VOLTAGE_BYTE0; i < TX433_FR_CRC_BYTE0+2;i++)
	{
		g_tx433.framedata[i] = 0x00;
	}

	// Bufferd Framebit
	g_tx433.framedata[TX433_FR_PARA_BYTE0] = TX433_BUFFERED;
	// Laden des Spannungswertes 12 Bit
	g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0] = (uint8_t) (waitingqueue[readindex].voltage12_1);
	g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0 + 1] = (uint8_t) (waitingqueue[readindex].voltage12_2) & 0xF0;

	timervalue = 0;
	timervalue = (waitingqueue[readindex].timelabel24_1 );
	timervalue = (timervalue << 16UL);
	timervalue += (waitingqueue[readindex].timelabel24_2 << 8);
	timervalue += (waitingqueue[readindex].timelabel24_3);

	timervalue = (uint32_t) timer24 - timervalue;

	// Der Konstante Wert von 20(2ms) repräsentiert die Zeit die benötigt wird für die Übertragung an den Transmitter und für das Sendecommando (MSP
Seitig)
	timervalue = timervalue + 20;
			//(uint32_t) timer24- timervalue + 20;
	g_tx433.framedata[TX433_FR_TIMESTAMP24_BYTE] = (uint8_t) (timervalue >> 20) + g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0 + 1];
	g_tx433.framedata[TX433_FR_TIMESTAMP24_BYTE + 1] = (uint8_t) (timervalue >> 12);
	g_tx433.framedata[TX433_FR_TIMESTAMP24_BYTE + 2] = (uint8_t) (timervalue >> 4);
	g_tx433.framedata[TX433_FR_TIMESTAMP24_BYTE + 3] = (uint8_t) (timervalue << 4);

}

void frame_tx433_live(void){
	uint8_t i;
	uint16_t timevalue;
	// Reset für Bitshifting notwendig
	for(i = TX433_FR_VOLTAGE_BYTE0; i < TX433_FR_CRC_BYTE0+2;i++)
	{
		g_tx433.framedata[i] = 0x00;
	}
	// TODO Framefehlermessung
	// framecounter++;

	// Livebit setzen
	g_tx433.framedata[TX433_FR_PARA_BYTE0] = TX433_LIVEBIT;
	// Laden des Spannungswertes 12 Bit
	g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0] = (uint8_t) (battvoltage >> 4);
	g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0 + 1] = (uint8_t) (battvoltage << 4);

	// Laden des Temperaturwertes 8 Bit
	g_tx433.framedata[TX433_FR_TEMPERATUR_BYTE0] = (uint8_t) (temperatur >> 4) + g_tx433.framedata[TX433_FR_VOLTAGE_BYTE0 + 1];
	g_tx433.framedata[TX433_FR_TEMPERATUR_BYTE0 + 1] = (uint8_t) (temperatur << 4);

	/*
	// Framecounter wird für den Zeitwert eingefügt
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE ] = (uint8_t) (framecounter >> 12) + g_tx433.framedata[TX433_FR_TEMPERATUR_BYTE0 + 1];
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 1] = (uint8_t) (framecounter >> 4);
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 2] = (uint8_t) (framecounter << 4);
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE ] = (uint8_t) (timer24 >> 12) + g_tx433.framedata[TX433_FR_TEMPERATUR_BYTE0 + 1];
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 1] = (uint8_t) (timer24 >> 4);
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 2] = (uint8_t) (timer24 << 4);*/

	// Laden des Zeitstempels 16 Bit
	// Umschreiben der wait_save_time sowie des festen Versatzes im Moment 2ms ca. 20Timer Ticks
	timevalue = (uint16_t)wait_save_time + (timer24-wait_save_time) + 20;
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE ] = (uint8_t) (timevalue >> 12) + g_tx433.framedata[TX433_FR_TEMPERATUR_BYTE0 + 1];
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 1] = (uint8_t) (timevalue >> 4);
	g_tx433.framedata[TX433_FR_TIMESTAMP16_BYTE + 2] = (uint8_t) (timevalue << 4);
}
```

C.1.22 i2c_bus.h

```
/*
 * i2c_bus.h
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 *      Quelle Masterarbeit Jegenhorst
 */

#include <stdint.h>

#ifndef I2C_BUS_H_
#define I2C_BUS_H_

	uint8_t i2c_bus_write(uint8_t div_address, uint8_t data_length, uint8_t *data);
	uint8_t i2c_bus_read(uint8_t div_address, uint8_t data_length, uint8_t *data);

#endif /* I2C_BUS_H_ */
```

C.1.23 i2c_bus.c

```
/*
 * i2c_bus.c
 *
 *  Created on: 25.06.2013
 *      Author: Michael
 *      Quelle Masterarbeit Jegenhorst
 */

#include "i2c_bus.h"
#include "mainheader.h"
#include "tx_433.h"
#include "global.h"

//--- Function: i2c_bus_write() ----------------------------------------------------------
uint8_t i2c_bus_write(uint8_t div_address, uint8_t data_length, uint8_t *data) {

	uint8_t i;

	if((UCB0CTL1 & UCSWRST) != UCSWRST) {				// when I2C is activ
		return EXIT_FAILURE;					//		then exit
	}
	UCB0CTL1 |= UCTR;						// set transmitter-mode
	UCB0I2CSA = div_address;					// set slave address of sensor
	UCB0CTL1 &= ~UCSWRST;						// unset SWRESET
	UCB0CTL1 |= UCTXSTT;						// send START-CON
	UCB0TXBUF = data[0];						// then write data

	for(i = 1; i < data_length; i++) {
		while(TRUE) {						// wait until ...
			if((UCB0STAT & UCNACKIFG) == UCNACKIFG){	// NACK from slave
				UCB0CTL1 |= UCTXSTP;			// then send STOP-CON
				UCB0STAT &= ~UCNACKIFG;			// reset flag
				UCB0CTL1 |= UCSWRST;			// set SWRESET
				return EXIT_FAILURE;			// and exit
			}
			if((IFG2 & UCB0TXIFG) == UCB0TXIFG) {// data / start-con was send
				UCB0TXBUF = data[i];			// then write data
				break;
			}
		}
	}
	while((IFG2 & UCB0TXIFG) != UCB0TXIFG);				// wait until data/start-con was send
	UCB0CTL1 |= UCTXSTP;						// send STOP-COND

	while((UCB0CTL1 & UCTXSTP) == UCTXSTP);				// wait until STOP-con was send
	UCB0CTL1 |= UCSWRST;						// set SWRESET

	return EXIT_SUCCESS;
}
//--- Function: i2c_bus_read() -----------------------------------------------------------
uint8_t i2c_bus_read(uint8_t div_address, uint8_t data_length, uint8_t *data) {

	uint8_t i;

	if((UCB0CTL1 & UCSWRST) != UCSWRST) {				// when I2C is activ
		return EXIT_FAILURE;					// then exit
	}
	UCB0CTL1 &= ~UCTR;						// reset transmitter-mode
	UCB0I2CSA = div_address;					// set slave address of sensor
	UCB0CTL1 &= ~UCSWRST;						// unset SWRESET
	UCB0CTL1 |= UCTXSTT;						// send START-CON

	if(data_length > 1) {
		for(i = 0; i < data_length; i++) {
			while(TRUE) {					// wait until ...
				if((UCB0STAT & UCNACKIFG) == UCNACKIFG){	// NACK from slave
					UCB0CTL1 |= UCTXSTP;		// then send STOP-CON
					UCB0STAT &= ~UCNACKIFG;		// reset flag
					UCB0CTL1 |= UCSWRST;		// set SWRESET
					return EXIT_FAILURE;		// and exit
				}
				if((IFG2 & UCB0RXIFG) == UCB0RXIFG) {			// data / start-con was send
					data[i] = UCB0RXBUF;				// readout data
					if(i == data_length-2)				// if next byte will be the last one
						UCB0CTL1 |= UCTXSTP;			// send STOP-CON after next receive
					break;
				}
			}
		}
	}
	else {
		while((UCB0CTL1 & UCTXSTT) == UCTXSTT);			// wait for acknowledge of slave,
		UCB0CTL1 |= UCTXSTP;					// then send STOP-COND immediately
		if((UCB0STAT & UCNACKIFG) == UCNACKIFG) {	// if NACK from slave
			UCB0CTL1 |= UCSWRST;				// set SWRESET
			return EXIT_FAILURE;				// and exit
		}
		data[0] = UCB0RXBUF;					// readout data
	}

	while((UCB0CTL1 & UCTXSTP) == UCTXSTP);				// wait until STOP-con was send
	UCB0CTL1 |= UCSWRST;						// set SWRESET

	return EXIT_SUCCESS;
}
//-------------------------------------------------------------------------------------
```

C.1.24 Timer_isr.c

```
/*
 * Timer_isr.c
 *
 *  Created on: 24.07.2013
 *      Author: Michael
 */
#include "mainheader.h"
#include "global.h"
#include "queue.h"
#include "adc.h"
#include "tx_433.h"

/*
 * Zuerst wird der ADC gestartet und die Spannung gemessen
 */
// ISR wird ausgelöst

#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_A0(void){

	// Timervalue für alle Zeitwerte
	timer24++;
	// Fehlerabsicherung Allgemein
	if (timer24 >= 0x7FFFFF){
		timer24 = 0;
	}

	switch (ADC) {
	case (ADC_idle) :
			// do nothing
	break;

	// Spannungsmessung wurde angetriggert
	case ADC_busy_voltage :

		// Prüfen ob ADC ready ist
		if(ADC10CTL1 & ADC10BUSY){
			// do nothing. ADC not ready
		}
		else {
			// ADC fertig
			// Alten Spannungswert in ADC Wert für den Vergleich bereitstellen
			battvoltage_old = tmp;
			// Umstellung auf das Senden der ADC Werte da Multiplikationen und Divisionen von 32Bit Werten auf dem MSP430G2553 zu teuer
			in der Zeit und im Speicher sind
			tmp += ADC10MEM;
			tmp >>= 1;

			// Sonderprüfung notwendig für den Zellensimulator
			// Prüfung ob die gemessene Spannung unter 1 Volt gefallen ist
			if(tmp < LowVoltage){
				// ist unter 1 Volt Transmitter abschalten
				transmit = Transmitter_off;
			} else if ((tmp > LowVoltage) && (tx_status == 1)) {
				// ist über 1 Volt und Transmitter ist vorher ausgeschaltet worden
				transmit = Transmitter_on;
			}

			// Vergleich ob Inqueue - Bedingung zutrifft
			if (((battvoltage_old + Voltagediff) < tmp) || (((battvoltage_old - Voltagediff) > tmp))) {

				ADC = ADC_inqueuewait;

				if(!(statusbitarray & 0x02)) {
					// Inqueuebit setzen einmalig
					statusbitarray |= 0x02;
					timer24 = 0;
					SR();
					// TODO RND SHIFT MAKRO
					wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC))));	// Schnelles
Senden
					// Counter mit Queuetime laden
					queuewait_counter = queuevaluetime;
					writequeue();
				} else {
					// Schutz das sich die Queue selbst überschreibt. Wenn Queue voll dann werden die Werte verworfen.
					if(writeindex == readindex){

					} else {
						writequeue();
					}
				}
			} else {
				// Inqueue trifft nicht zu
				if(statusbitarray & 0x02){
					// Erste Rückkehr
					if(!(statusbitarray & 0x12)){
						wait_save_time = queuereturntime;		// Sample Wiederholung auf 2s setzen temporär
						statusbitarray |= 0x12;				// Gegen erneutes Setzen schützen
						queue_counter = 0;				// Zurücksetzen des 30 Werte Inqueues
					}
					// Rückkehr aus Inqueuebedingung Übergang Normalbetrieb
					// Alle Werte aus der Queue senden, noch immer schnell.
					if(writeindex != readindex){
						wait_save_time--;
						if(wait_save_time == 0){
```

```
                                                wait_save_time = queuereturntime; // Queuereturntime sollte 2 Sekunden entsprechen
                                                // Werte verlangsamt in die Queue schreiben, dadurch kann sich die Queue abbauen
                                                writequeue();
                                        }
                                } else if (((!(statusbitarray & 0x40)) && (!(statusbitarray & 0x20))) || ((statusbitarray & 0x40) && (statusbitarray &
0x20))) {
                                //(((statusbitarray & 0x40) && (statusbitarray & 0x20)) || ((!(statusbitarray & 0x40) && (!(statusbitarray & 0x20))) {
                                        // Reset Status Rückkehr zu Normalbetrieb
                                        statusbitarray = 0;
                                        // Neue Wartezeit ermitteln
                                        SR();
                                        // TODO RND SHIFT MAKRO
                                        //wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 3 ; //
                                        slow im Mittel 4 Sekunden...
                                        wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 2 ; //
                                        slow im Mittel 2 Sekunden...
                                        // Reset timer
                                        timer24 = 0;
                                }
                        } else {
                                // Fehlerabsicherung Normalbetrieb allerdings Wertebeschränkung auf 63000 / 0xF618
                                if(timer24 >= 0xF618){
                                timer24 = 0;
                                }
                        }
                        // Sicherung gegen erneutes auslösen
                        if(!statusbitarray) {
                                wait_save_time = wait_rnd_time;
                                battvoltage = tmp;
                                statusbitarray = 0x01;
                        }
                        // Nun Temperatur messen,
                        sample_temperature();
                        ADC = ADC_busy_temperature;
                }
        }
        break;
case (ADC_inqueuewait) :
                queuewait_counter--;
// Status setzen damit die Senderoutine nicht den Ablauf verändert
statusbitarray |= 0x80;
// Neuen Wert messen
// Neuen Wert zwischenpuffern
if(queuewait_counter == 2)
{
        sample_voltage_batt();

} else if (queuewait_counter == 1) {
        // Überschreiben des Wertes damit die Mittelung den Wert nicht zu sehr verzerrt
        tmp = ADC10MEM;
        sample_voltage_batt();
} else if(queuewait_counter == 0) {
        // ADC Wert einlesen und Mitteln
        tmp += ADC10MEM;
        tmp >>= 1;
        // Sonderprüfung notwendig für den Zellensimulator
        // Prüfung ob die gemessene Spannung unter 1 Volt gefallen ist
        if(tmp < LowVoltage){
                // ist unter 1 Volt Transmitter abschalten
                transmit = Transmitter_off;
        } else if ((tmp > LowVoltage) && (tx_status == 1)) {
                // ist über 1 Volt und Transmitter ist vorher ausgeschaltet worden
                transmit = Transmitter_on;
        }
        sample_voltage_batt();
        // Inqueue mit 30 Werten FIX, Counter erhöhen und Wartezeit zurücksetzen
        queue_counter++;
        queuewait_counter = queuevaluetime;
        // Schutz das sich die Queue selbst überschreibt. Wenn Queue voll dann werden die Werte verworfen.
        if(writeindex == readindex){

        } else {
                // Wert speichern
                writequeue();
        }
} else if (queue_counter == queue_fix_length){
        // Prüfen ob überhaupt noch Inqueue Bedingung vorliegt
        sample_voltage_batt();
        // Zurücksetzen
        statusbitarray &= 0x7F;
        ADC = ADC_busy_voltage;
}
break;

case (ADC_busy_temperature) :
                if(ADC10CTL1 & ADC10BUSY){
                // do nothing. ADC not ready
                }
                else {
                        temperatur = ADC10MEM;
                        sample_voltage_batt();
                        ADC = ADC_busy_voltage;
                }
break;
default :
        break;
}

switch(transmit){
```

```
case (IDLE):
// Antriggern des ADC
sample_voltage_batt();
if (statusbitarray & 0x80) {
        // Rückkehr zur Inqueuewaitfunktion
        ADC = ADC_inqueuewait;
} else {
        ADC = ADC_busy_voltage;
}
transmit = wait;
break;
case (wait):
// Reduzierung der Sendewartezeit um 1.
// Beim erreichen eines Wertes von 0 darf erneut gesendet werden mit vorheriger Aufbereitung
wait_rnd_time--;
// Prüfung
if(wait_rnd_time == 0){
        // Daten ins Frame laden lassen
        transmit = Frameload;
        ADC = ADC_idle;
}
break;
case (Frameload) :
        // Prüfung ob Inqueue vorliegt
        if(statusbitarray & 0x02){
                // liegt vor
                // Werte ins Frame
                frame_tx433_buffered();
                readindex++;
                if(readindex >= Queue_Length){
                        readindex = 0;
                        // Überschreibschutz Read mit Toggle wenn Ende der Queue erreicht wurde
                        if((statusbitarray & 0x20)== 1){
                                statusbitarray &= 0xD0;
                        } else {
                                statusbitarray |= 0x20;
                        }
                }
        } else {
                // liegt nicht vor
                frame_tx433_live();
        }
transmit = Transmit_crc;
break;
case (Transmit_crc) :
frame_tx433_calc_crc();
transmit = Transmit_man;
break;
case (Transmit_man) :
frame_tx433_code_manchester();
transmit = Transmit_data_1;
                break;
case (Transmit_data_1) :
        // Dauer der Interruptabschaltung und des Sendens des Frames an den Transmitter : 1.96ms < x < 2.08ms
        _DINT();
        // Daten an Transmitter senden
        tx433_cmd_fifo_set();
        transmit = Transmit;
        //break;
        // Zwangspause erzeugen sonst sendet der Transmitter nicht.
        _EINT();
        //transmit = Transmit;
//break;
case (Transmit) :
_DINT();
// Dem Transmitter das Sendekommando übermitteln
tx433_cmd_tx_start();
LED_toggle;
// Neue Zufallszahl
SR();
// Prüfung ob Inqueue vorliegt
if(!(statusbitarray & 0x02)){
        // liegt nicht vor
        // Neue Wartezeit
        // TODO RND SHIFT MAKRO
        wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 2 ; // slow im Mittel 2 Sekunden...
        //wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))) << 3;          // slow im Mittel 4 Sekunden
        timer24 = 0;
        statusbitarray = 0x00;
} else {
        // liegt vor
        // Neue Wartezeit
        wait_rnd_time = ((uint16_t)(sr_r % (2 * (TXMID / TXFAC)) + ( TXMID - (TXMID / TXFAC)))); // fast
}
transmit = IDLE;
_EINT();
break;
case (Transmitter_off) :
        // Flag setzen für Transmitterabschaltung
        tx_status = 1;
        TX433_OFF;
break;
case (Transmitter_on) :
        // Globale Interrupts deaktivieren
        _DINT();
        // Transmitter anschalten
        TX433_ON;
        // Zwangspause Transmitter ON != Ready
         __delay_cycles(250000);
         // Initalisierung des Transmitters
```

```
                    tx433_init();
                    // Flag zurücksetzen für Transmitterabschaltung
                    tx_status = 0;
                    // Globale Interrupts aktivieren
                    _EINT();
                    // Rückkehr zum normalen Status
                    transmit = IDLE;
                    break;
        default :
                            break;
        }

}
```

C.2 Steuergerät

Es werden hier nur die geänderten Dateien aufgeführt.

Alle Quellcodedateien befinden sich in der Masterarbeit Jegenhorst [18].

C.2.1 main.c

```
/*---------------------------------------------------------------------------
        Project:                BATSEN
        Discription:            MSP430 project source file
        Used components:    MSP430-169STK, MSP-GCC 20100218, Eclipse Helios
        Date:                   10/28/2007

        Author:                 Stephan Plaschke
        Last Update:            04/04/2008

        Modified by:            Alexander Hoops
        Last modification:      19/01/2010

        Modified by:            Niels Jegenhorst
        Last modification:      23/05/2011

        Modified by:            Sergej Ilgin
        Last modification:      17/08/2011

        Modified by:            Michael Meinzer
        Last modification:      20/09/2013

        File:                   main.c
---------------------------------------------------------------------------

---------------------------------------------------------------------------
        Headerfiles
---------------------------------------------------------------------------*/
#include "main.h"
#include "stdio.h"
#include <signal.h>
#include <math.h>
#include <msp430x16x.h>
#include "headerfiles/rtc.h"
#include "headerfiles/adc.h"
#include "headerfiles/flash.h"
#include "headerfiles/lcd16x2.h"
#include "headerfiles/uart_menu.h"
#include "headerfiles/msp_functions.h"
#include "headerfiles/sensor_data_proc.h"
#include "headerfiles/system_handling.h"

#define EXTERN                                                  // define globals here
#define INIT_GLOBALS                                            // init global constants here
#include "headerfiles/globals.h"

/*---------------------------------------------------------------------------
||      MAIN LOOP
---------------------------------------------------------------------------*/
int main (void) {

        unsigned char OSC_error;
        unsigned char UART_error;
        // Einschub Meinzer
        framecounter = 0;
        calvoltage = 0;                 // Cal Voltage disabled
        sendvoltage = 0;        // SendVoltage disabled
        RTC_MSP430 RTC_tmp = {0,0,0,0};

        /*---------------------------------------------------------------------------
        ||      Initialization
        //
        ---------------------------------------------------------------------------*/
        //OSC_error = DCO_set(uC_FREQUENCY);            // internal oscillator
        OSC_error = XT2_set(uC_FREQUENCY);              // external oscillator
```

```
	globals_init();					// Set global variables


	PORTS_init();					// initialize PORT direction/function
	LCD_init();					// initializ-e 16x2 character LCD
	LED1_OFF;					// initialize LEDs+Tara value
	LED2_OFF;
	BL_OFF;						// LCD Backlight OFF

	eint();						// IRQs enable

	RTC_startup();					// initialize RTC
	init_Flash();					// initialize Flash
	RADIO_init();					// Initialize radio unit

	FLASH_read_at_startup();			// read control data from flash?
	LCD_send_time();				// display time on LCD
	recording_stop();				// stop measurement
	#ifdef ENABLE_CURRENT_MEASURE
		ADC_init();				// initialize ADC
	#endif
	LED1_OFF;					// switch LEDS off
	LED2_OFF;
	USART0_init(RX_INT, RX_TX);// initialize USART, enable RX&TX irq and RX&TX modul
	UART0_send_text("\n System started ----\n");

	/*-------------------------------------------------------------------------
	||		Infinite Loop
	-------------------------------------------------------------------------*/
	while(TRUE) {
							// look if real time clock
		RTC_compare(&RTC_tmp);			// values has changed

		// string from UART received? ----------
		if (BATMON_control_reg & GLOBAL_STRING_RECEIVED) {
			UART_error = UART0_cmp_string();		// compare string with
									// valid commands
			if (UART_error){
				UART0_send_text("  NOK\n");
			}
			else if (calvoltage == 1){
					// send nothing
			}
			else
				UART0_send_text("  OK\n");

			BATMON_control_reg &= !GLOBAL_STRING_RECEIVED;	// clear global value
		}
		// data received from sensor? ----------
		if (BATMON_control_reg & VALID_DATA_RECEIVED) {
			TP_ADC2_ON;
			BATMON_control_reg &= !VALID_DATA_RECEIVED;	// clear global value
			g_rx_status = INACTIVE;				// set flag

			// Für Debug entfernt Fehlersuche : Verlorene Frames / Doppelte Frames und Falsch laufenende Timer
			// Update : Keine Veränderung
			/*if (scan_mode < 1) {
				RADIO_store_frame();		// store valid frame in flash
			}
*/
			if((show_sensornmb == 0) && (g_show_sensor_lcd == TRUE)) {

				//show sensor address and battery voltage on display
			  LCD_send_sensor_v2(g_sensor_data.id & 0xFF, g_sensor_data.batteryvoltage);

			}
```

```
// Debug umschreiben der Methode auf Binblockwrite für Matlab...
// show sensor data on UART in decimal-mode:
if(g_show_sensor_data_uart == DEC) {
        if(calvoltage == 0){
        // Binblockwrite Modus : Derzeit werden 22 Byte geschrieben daher D = 2 / A = 2 2
        // Startcommando
        UART0_send('#');
        UART0_send('2'); //number of digits that follows in D
        UART0_send('2'); //number of bytes that follows in A
        UART0_send('1');

        // TODO Anpassungen an Binblockwrite für Matlab
        // Zeitstempel wird gesplittet in mehreren Variablen übertragen
        // Parameter und Errorbits zuerst 8 Bit
        UART0_send((unsigned char)(g_sensor_data.parameter_error));
        // ID des Sensors 8 Bit
        UART0_send((unsigned char)(g_sensor_data.id));
        // Spannungswert 12 Bit Value in einer 32 Bit Variablen
        // 31 - 24
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 24));
        // 23 - 16
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 16));
        // 15 - 8
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 8));
        // 7 - 0
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage & 0x000000FF));
        // Temperaturwert (nur bei Liveframe Wert > 0) 8 Bit
        UART0_send((unsigned char)(g_sensor_data.temperature & 0x000000FF));
        // Timerwert max. 24 Bit Value (Liveframe 12 Bit) in einer 32 Bit Variablen
        // 31 - 24
        UART0_send((unsigned char)(g_sensor_data.timervalue >> 24));
        // 23 - 16
        UART0_send((unsigned char)(g_sensor_data.timervalue >> 16));
        // 15 - 8
        UART0_send((unsigned char)(g_sensor_data.timervalue >> 8));
        // 7 - 0
        UART0_send((unsigned char)(g_sensor_data.timervalue & 0x000000FF));

        // Zeitstempel der Basisstation gesplittet in Tag / Stunde / Minute / Sekunde / Millisekunde
        // Day
        UART0_send((unsigned char)(g_sensor_data.day));
        // Hour
        UART0_send((unsigned char)(g_sensor_data.hour));
        // Minute
        UART0_send((unsigned char)(g_sensor_data.min));
        // Second
        UART0_send((unsigned char)(g_sensor_data.second));
        // Msecond 15 - 7
        UART0_send((unsigned char)(g_sensor_data.msecond >> 8));
        // Msecond 7 - 0
        UART0_send((unsigned char)(g_sensor_data.msecond & 0x00FF));
        // TODO DEBUG TIMER UND MEHR
        UART0_send((unsigned char)(g_sensor_data.framecounter>> 24));
        // 23 - 16
        UART0_send((unsigned char)(g_sensor_data.framecounter >> 16));
        // 15 - 8
        UART0_send((unsigned char)(g_sensor_data.framecounter >> 8));
        // 7 - 0
        UART0_send((unsigned char)(g_sensor_data.framecounter & 0x000000FF));
}else if (calvoltage == 1 && sendvoltage == 1){
        // Send via UART BinBlockWrite - Voltage only 32Bit / 8 Bit
        UART0_send('#');
        UART0_send('1'); //number of digits that follows in D
        UART0_send('4'); //number of bytes that follows in A
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 24));
        // 23 - 16
        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 16));
```

```
                                        // 15 - 8
                                        UART0_send((unsigned char)(g_sensor_data.batteryvoltage >> 8));
                                        // 7 - 0
                                        UART0_send((unsigned char)(g_sensor_data.batteryvoltage & 0x000000FF));


                                        // Reset
                                        sendvoltage = 0;

                                }
                                }
                                TP_ADC2_OFF;
                        }
                // show sensor data in first LCD row? -------
                        if (BATMON_control_reg & CLEAR_FIRST_LCD_ROW) {

                                //show sensor address and battery voltage on display
                                LCD_send_sensor_v2(g_sensor_data.id , g_sensor_data.batteryvoltage);
                                BATMON_control_reg &= !CLEAR_FIRST_LCD_ROW;
                        }
                        // Button pressed jump into menu? -----------
                        if((BATMON_control_reg & B1) || (BATMON_control_reg & B2)) {
                                BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
                                BATMON_menu_options();
                        }
                        // Button pressed show next info menu -------
                        if(BATMON_control_reg & B3)  {
                                show_sensornmb++;
                                BATMON_control_reg &= (~B3); // clear Button3 flag
                                BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
                                if (show_sensornmb > (SENSOR_active+6)) {
                                        show_sensornmb = 0;
                                }
                        }
        }               // End of while(1)
        return (0);
}
//-------------------------------------------------------------------------------
```

C.2.2 globals_init.c

```
/*-------------------------------------------------------------------------------
        Project:                BATSEN
        Discription:
        Used components:    MSP430-169STK

        Author:                 Niels Jegenhorst
        Date:                   20/04/2011
        Last update:            20/04/2011

        Modified by:            Michael Meinzer
        Last modification:      20/09/2013

        File:                                   globals_init.c
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
        Headerfiles
-------------------------------------------------------------------------------*/
#include <main.h>
```

```
#include <msp430x16x.h>
#include "headerfiles/msp_functions.h"
#include "headerfiles/uart_menu.h"
#include "headerfiles/lcd16x2.h"
#include "headerfiles/adc.h"
#include "headerfiles/flash.h"
#include "headerfiles/system_handling.h"
#include "headerfiles/globals.h"

/*-------------------------------------------------------------------------
        void globals_init(void)
-------------------------------------------------------------------------*/
void globals_init(void) {

        uint8_t i;

        RTC_values.day      = 22;                       // initialize clock
        RTC_values.lastday  = 21;
        RTC_values.month    = 01;
        RTC_values.year             = 2009;
        RTC_values.wday     = TUE;
        RTC_values.hr               = 23;
        RTC_values.min              = 59;
        RTC_values.sec              = 55;
        RTC_values.msec             = 0;

        Index_Ring = 256;                               // default address in flash
        Index_Last = 256;

        UART_menu_position = UART_MENU_FIRSTCHAR;
        Enable_clear_lcd   = STATUS;
        show_sensornmb     = 0;
        g_show_sensor_lcd  = TRUE;
        g_show_sensor_data_uart = DEC;

        measuring = 0;
        scan_mode = 0;

/*      SENSOR_temp_cal_b = { 0,27,22,-5,10,15,-14,2,9,-34,
 *                                                      81,32,0,-24,38,0,0,0,0,0,
 *                                                      0,0,0,0,0,0,0,0,0,0,
 *                                                      0,0,0,0,0,0,0,0,0,0};
*/
        SENSOR_temp_cal_b[0]  = 0;
        SENSOR_temp_cal_b[1]  = 27;
        SENSOR_temp_cal_b[2]  = 22;
        SENSOR_temp_cal_b[3]  = -5;
        SENSOR_temp_cal_b[4]  = 10;
        SENSOR_temp_cal_b[5]  = 15;
        SENSOR_temp_cal_b[6]  = -14;
        SENSOR_temp_cal_b[7]  = 2;
        SENSOR_temp_cal_b[8]  = 9;
        SENSOR_temp_cal_b[9]  = -34;
        SENSOR_temp_cal_b[10] = 81;
        SENSOR_temp_cal_b[11] = 32;
        SENSOR_temp_cal_b[12] = 0;
        SENSOR_temp_cal_b[13] = -24;
        SENSOR_temp_cal_b[14] = 38;
        for(i=15; i<41; i++) {
                SENSOR_temp_cal_b[i] = 0x00;
        }

        //      g_currmeas.current_tara = 0;
        g_currah.AH_last_minute = 0;
        g_currah.AH_last_sec    = 0;
        g_currah.AH_last_minute_IN_OUT = 0;
        g_currmeas.current_invert = 0;
```

```
        g_currmeas.channel = 0;
        g_currmeas.x_low  = 0;
        g_currmeas.x_mid  = 0;
        g_currmeas.x_high = 0;
        g_currmeas.x_min  = 0;
        g_currmeas.finish = FALSE;

        CELL_VOLT_MAX   = 214;
        CELL_VOLT_MIN   = 189;
        BATT_CAPACITY   = 88;
        fer = 0;

        g_rx_error.bit_error   = 0x0000;
        g_rx_error.wrong_RunIn = 0x0000;
        g_rx_error.timeout     = 0x0000;
        for(i=0; i<6; i++) {
                g_rx_error.crc_error[i] = 0x0000;
        }
        g_rx_status      = INACTIVE;
        g_rx_en                         = FALSE;


        // Laden der Kal Werte in das Struct Array mit Verschiebung um 1
        // Sensor 1
        g_voltage_cal_data[1].m = 10082;                        // 1.0082 / 5.0126*10^4
        g_voltage_cal_data[1].b = 501260;

        // Sensor 2
        g_voltage_cal_data[2].m = 9890;                         // 0.9890 / 4.6894*10^4
        g_voltage_cal_data[2].b = 468940;

        // Sensor 3
        g_voltage_cal_data[3].m = 9961;                         // 0.9961 / 5.0171*10^4
        g_voltage_cal_data[3].b = 501710;

        // Sensor 4 (nicht kalibriert Sensor 4 ausgestattet mit den alten Sicherungen)
        g_voltage_cal_data[4].m = 1;
        g_voltage_cal_data[4].b = 1;

        // Sensor 5
        g_voltage_cal_data[5].m = 10111;                        // 1.0111 / 5.4232*10^4
        g_voltage_cal_data[5].b = 542320;

        // Sensor 6
        g_voltage_cal_data[6].m = 10077;                        // 1.0077 / 4.5674*10^4
        g_voltage_cal_data[6].b = 456740;

        // Sensor 7
        g_voltage_cal_data[7].m = 9924;                         // 0.9924 / 4.9317*10^4
        g_voltage_cal_data[7].b = 493170;

        // Sensor 8
        g_voltage_cal_data[8].m = 10076;                        // 1.0076 / 5.0589*10^4
        g_voltage_cal_data[8].b = 505890;

        // Sensor 9
        g_voltage_cal_data[9].m = 9843;                         // 0.9843 / 5.413*10^4
        g_voltage_cal_data[9].b = 541300;

        // Sensor 10 Dummy Sensor ohne Sicherungen
        g_voltage_cal_data[0].m = 9965;                         // 0.9965 / 5.8939*10^3
        g_voltage_cal_data[0].b = 58939;



}
```

```
//-------------------------------------------------------------------------------
```

C.2.3 sensor_data_proc.c

```
/*-------------------------------------------------------------------------------
        Project:                BATSEN
        Discription:            Sourcecode for sensor data processing
        Used components:    MSP430-169STK

        Author:                 Niels Jegenhorst
        Date:                   20/04/2011
        Last update:            20/04/2011

        Modified by:            Sergej Ilgin
        Last modification:      17/08/2011 Function: convert_sensor_data()

        Modified by:            Michael Meinzer
        Last modification:      20/09/2013

        File:                                   sensor_data_proc.c
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
        Headerfiles
-------------------------------------------------------------------------------*/
#include <main.h>
#include <msp430x16x.h>
#include "headerfiles/globals.h"

#define SENSOR_LSB_Voltage (2.5/1023)


/*-------------------------------------------------------------------------------
        void check_depth_discharge(uint8_t index)
-------------------------------------------------------------------------------*/
void check_depth_discharge(uint8_t index) {

                // Author:              Alexander Hoops
                // Last modification:   12/02/2010

        unsigned long check_voltage;

        check_voltage = (SENSOR_last_data[3][index]<<8);
        check_voltage |= SENSOR_last_data[4][index];
        if ((check_voltage/4.096) <= ((CELL_VOLT_MIN*10)+100)) {
                if ((check_voltage/4.096) <= (CELL_VOLT_MIN*10)) {
                        SENSOR_low++;
                        // don t leave next value
                        // SENSOR_addr_req[address_byte] |=
                                // (0x01 << address_bit);
                }
        }
}
/*-------------------------------------------------------------------------------
        void convert_sensor_data(void)
-------------------------------------------------------------------------------*/
void convert_sensor_data(void) {

        framecounter++;

        uint32_t tmp = 0x00;
        // Anpassungen an neues Protokoll der Sensoren

        // Speicherung der Parameterbits und Errorbits
        g_sensor_data.parameter_error = RADIO_frame.frame_byte[0];
        // Speicherung der Sensoradresse
        g_sensor_data.id = RADIO_frame.frame_byte[1];
        // Speicherung des Spannungswertes
        g_sensor_data.batteryvoltage = ((uint8_t)RADIO_frame.frame_byte[2] << 4) + ((uint8_t)RADIO_frame.frame_byte[3] >> 4);

        // Prüfung aufs Livebit
        if(g_sensor_data.parameter_error & 0x10)
        {
                // Liveframe
                // Speicherung des 12 Bit Temperaturwertes Korrektur des Offsets in Matlab
                tmp = (uint8_t)(RADIO_frame.frame_byte[3] & 0x0F);
                g_sensor_data.temperature = tmp << 4;
                tmp = (uint8_t)(RADIO_frame.frame_byte[4] & 0xF0);
                g_sensor_data.temperature += tmp >> 4;
                // Speicherung des 16 Bit Timerwertes
                tmp = (uint8_t)(RADIO_frame.frame_byte[4] & 0x0F);
                g_sensor_data.timervalue = tmp << 12;
                tmp = (uint8_t)RADIO_frame.frame_byte[5];
                g_sensor_data.timervalue += tmp << 4;
                tmp = (uint8_t)(RADIO_frame.frame_byte[6] & 0xF0);
                g_sensor_data.timervalue += tmp >> 4;
        }
        else {
        // Buffered Frame
        // Speicherung des 24 Bit Timerwertes
        tmp = (uint8_t)(RADIO_frame.frame_byte[3] & 0x0F);
        g_sensor_data.timervalue = tmp << 20;
        tmp = (uint8_t)(RADIO_frame.frame_byte[4]);
        g_sensor_data.timervalue += tmp << 12;
        tmp = (uint8_t)(RADIO_frame.frame_byte[5]);
```

```
g_sensor_data.timervalue += tmp << 4;
tmp = (uint8_t)(RADIO_frame.frame_byte[6] & 0xF0);
g_sensor_data.timervalue += tmp >> 4;
// da kein Temperaturwert im Buffered Frame übertragen wird, aber beim Transfer an UART
g_sensor_data.temperature = 0;
}
g_sensor_data.framecounter = framecounter;

// Neue Speicherung des Zeitstempels
g_sensor_data.day = g_rx_timing.rtc_sof.day;                    // day
g_sensor_data.hour = g_rx_timing.rtc_sof.hr;                    // hour
g_sensor_data.min = g_rx_timing.rtc_sof.min;                    // minute
g_sensor_data.second = g_rx_timing.rtc_sof.sec;                 // second
g_sensor_data.msecond = g_rx_timing.rtc_sof.msec;               // msecond

// Einschub am 23.08
// Korrektur der Spannungswerte und Temperaturwerte(später) da die Sensorrechenleistung nicht ausreicht
// Die KAL Daten stehen im Speicher für die Sensoren 1/2/3/5/6/7/8/9/10(Sensor ohne Sicherungen)
// y = mx + b Rechnung. Linear

// TODO keine Korrektur für den Vergleich und den Einfluss bei der Framefehlermessung.
tmp = (g_sensor_data.batteryvoltage*2*1000*SENSOR_LSB_Voltage) ; // *1000 für die mV Korrektur
g_sensor_data.batteryvoltage = (uint32_t)(g_voltage_cal_data[g_sensor_data.id].m * tmp + g_voltage_cal_data[g_sensor_data.id].b)/10000;

tmp = 0;
/* Alte Speicherung der Werte vom Radioframe
g_sensor_data.vid  = RADIO_frame.frame_byte[0];
g_sensor_data.id   = RADIO_frame.frame_byte[3];
g_sensor_data.id  |= (uint32_t) RADIO_frame.frame_byte[2] << 8;
g_sensor_data.id  |= ((uint32_t) RADIO_frame.frame_byte[1] << 16) & 0xF0;
g_sensor_data.dataset = 0x00;
*/
// Switch(CASE) für Sensor_Version entfällt. Im neuen Protokoll ist dafür nichts vorgesehen.

// Select the correct conversion on the
/*switch (g_sensor_data.vid) {   //          basis of the sensor VID:
        case 0x00:
                break;

        case 0x01:                                      // VID of BS0406 ____________________________________
                                                                        // Conversion of temperature value:
                g_sensor_data.temperature  = (uint16_t) RADIO_frame.frame_byte[4] << 8;
                g_sensor_data.temperature |= RADIO_frame.frame_byte[5];
                                                                        // no Info?!
                g_sensor_data.temperature = g_sensor_data.temperature;
                RADIO_frame.frame_byte[4] = g_sensor_data.temperature >> 8;
                RADIO_frame.frame_byte[5] = g_sensor_data.temperature & 0xFF;

                                                                        // Conversion of cell-voltage:
                g_sensor_data.v_cell  = (uint16_t) RADIO_frame.frame_byte[6] << 8;
                g_sensor_data.v_cell |= RADIO_frame.frame_byte[7];
                                                                        // data * 4.096 --> [mV]
                g_sensor_data.v_cell  = g_sensor_data.v_cell * 40.96;
                RADIO_frame.frame_byte[6] = g_sensor_data.v_cell >> 8;
                RADIO_frame.frame_byte[7] = g_sensor_data.v_cell & 0xFF;

                                                                        // Conversion of supply-voltage:
                g_sensor_data.v_supply  = (uint16_t) RADIO_frame.frame_byte[8] << 8;
                g_sensor_data.v_supply |= RADIO_frame.frame_byte[9];
                                                                        // data * 4.096 --> [mV]
                g_sensor_data.v_supply  = g_sensor_data.v_supply * 4.096;
                RADIO_frame.frame_byte[8] = g_sensor_data.v_supply >> 8;
                RADIO_frame.frame_byte[9] = g_sensor_data.v_supply & 0xFF;
                break;

        case 0x03:                                      // VID of ZS v0.1 ___________________________________

                g_sensor_data.dataset = (RADIO_frame.frame_byte[1] & 0xF0) >> 4;
                switch (g_sensor_data.dataset) {
                        case SET_1:                                     // When standard sensor data received: ----------
                                                                                // Conversion of cell-voltage:
                                g_sensor_data.v_cell  = (uint16_t) RADIO_frame.frame_byte[6] << 8;
                                g_sensor_data.v_cell |= RADIO_frame.frame_byte[7];
                                                                                // data / 2^12 * 25000 --> [10 x mV]
                                g_sensor_data.v_cell  = (g_sensor_data.v_cell * 25000) >> 12;
                                RADIO_frame.frame_byte[6] = g_sensor_data.v_cell >> 8;
                                RADIO_frame.frame_byte[7] = g_sensor_data.v_cell & 0xFF;

                                                                                // Conversion of temperature value:
                                g_sensor_data.temperature  = (uint16_t) RADIO_frame.frame_byte[4]<<8;
                                g_sensor_data.temperature |= RADIO_frame.frame_byte[5];
                                                                                // data*100*0.0625 --> [100 x °C]
                                g_sensor_data.temperature = (g_sensor_data.temperature * 100) >> 4;
                                RADIO_frame.frame_byte[4] = g_sensor_data.temperature >> 8;
                                RADIO_frame.frame_byte[5] = g_sensor_data.temperature & 0xFF;

                                                                                // Conversion of supply-voltage:
                                g_sensor_data.v_supply  = (uint16_t) RADIO_frame.frame_byte[8] << 8;
                                g_sensor_data.v_supply |= RADIO_frame.frame_byte[9];
                                                                                // data / 2^12 * 2500 --> [mV]
                                g_sensor_data.v_supply  = (g_sensor_data.v_supply * 2500) >> 12;
                                RADIO_frame.frame_byte[8] = g_sensor_data.v_supply >> 8;
                                RADIO_frame.frame_byte[9] = g_sensor_data.v_supply & 0xFF;

                                g_sensor_data.v_actin = 0;
                                g_sensor_data.v_lp    = 0;
                                break;
                        case SET_2:                                     // When alternative sensor data received: -------
                                                                                // Conversion of lowpass-voltage:
                                g_sensor_data.v_lp  = (uint16_t) RADIO_frame.frame_byte[4] << 8;
```

```
                                        g_sensor_data.v_lp |= RADIO_frame.frame_byte[5];
                                                                                        // data / 2^12 * 2500 --> [mV]
                                        g_sensor_data.v_lp  = (g_sensor_data.v_lp * 2500) >> 12;
                                        RADIO_frame.frame_byte[4] = g_sensor_data.v_lp >> 8;
                                        RADIO_frame.frame_byte[5] = g_sensor_data.v_lp & 0xFF;

                                                                                        // Conversion of Activator-voltage:
                                        g_sensor_data.v_actin  = (uint16_t) RADIO_frame.frame_byte[8] << 8;
                                        g_sensor_data.v_actin |= RADIO_frame.frame_byte[9];
                                                                                        // data / 2^12 * 2500 --> [mV]
                                        g_sensor_data.v_actin  = (g_sensor_data.v_actin * 2500) >> 12;
                                        RADIO_frame.frame_byte[8] = g_sensor_data.v_actin >> 8;
                                        RADIO_frame.frame_byte[9] = g_sensor_data.v_actin & 0xFF;

                                                                                        // Conversion of cell-voltage:
                                        g_sensor_data.v_cell  = (uint16_t) RADIO_frame.frame_byte[6] << 8;
                                        g_sensor_data.v_cell |= RADIO_frame.frame_byte[7];
                                                                                        // data / 2^12 * 25000 --> [10 x mV]
                                        g_sensor_data.v_cell  = (g_sensor_data.v_cell * 25000) >> 12;
                                        RADIO_frame.frame_byte[6] = g_sensor_data.v_cell >> 8;
                                        RADIO_frame.frame_byte[7] = g_sensor_data.v_cell & 0xFF;

                                        g_sensor_data.v_supply    = 0;
                                        g_sensor_data.temperature = 0;
                                        break;
                                case REPLY_CMD:
                                        g_sensor_data.rx_last_cmd[0] = RADIO_frame.frame_byte[4];
                                        g_sensor_data.rx_last_cmd[1] = RADIO_frame.frame_byte[5];
                                        g_sensor_data.rx_last_parameter[0]  =
                                                                                (uint16_t) RADIO_frame.frame_byte[6] << 8;
                                        g_sensor_data.rx_last_parameter[0] |= RADIO_frame.frame_byte[7];
                                        g_sensor_data.rx_last_parameter[1]  =
                                                                                (uint16_t) RADIO_frame.frame_byte[8] << 8;
                                        g_sensor_data.rx_last_parameter[1] |= RADIO_frame.frame_byte[9];
                                        break;
                                default:
                                        break;
                        }
                        break;
                        case 0x04:              // Sensor VID of BSBM20V rev.1  S.I

                                        g_sensor_data.dataset = (RADIO_frame.frame_byte[1] & 0xF0) >> 4;
                                        switch (g_sensor_data.dataset) {
                                                case SET_1:
                                                  // convert battery voltage:
                                                        g_sensor_data.v_cell  = (uint16_t) RADIO_frame.frame_byte[6] << 8;
                                                        g_sensor_data.v_cell |= RADIO_frame.frame_byte[7];

                                                        //overwrite with converted voltage
                                                        RADIO_frame.frame_byte[6] = g_sensor_data.v_cell >> 8;
                                                        RADIO_frame.frame_byte[7] = g_sensor_data.v_cell & 0xFF;

                                                        // conversion of temperature value:
                                                        g_sensor_data.temperature  = (uint16_t) RADIO_frame.frame_byte[4]<<8;
                                                        g_sensor_data.temperature |= RADIO_frame.frame_byte[5];

                                                        //overwrite with converted temperature
                                                        RADIO_frame.frame_byte[4] = g_sensor_data.temperature >> 8;
                                                        RADIO_frame.frame_byte[5] = g_sensor_data.temperature & 0xFF;

                                                        //conversion of supply-voltage:
                                                        g_sensor_data.v_supply  = (uint16_t) RADIO_frame.frame_byte[8] << 8;
                                                        g_sensor_data.v_supply |= RADIO_frame.frame_byte[9];

                                                        ////overwrite with converted supply-voltage
                                                        RADIO_frame.frame_byte[8] = g_sensor_data.v_supply >> 8;
                                                        RADIO_frame.frame_byte[9] = g_sensor_data.v_supply & 0xFF;

                                                        g_sensor_data.v_actin = 0;
                                                        g_sensor_data.v_lp    = 0;
                                                        break;
                                                case REPLY_CMD:
                                                        g_sensor_data.rx_last_cmd[0] = RADIO_frame.frame_byte[4];
                                                        g_sensor_data.rx_last_cmd[1] = RADIO_frame.frame_byte[5];
                                                        g_sensor_data.rx_last_parameter[0]  =
                                                                                                (uint16_t)
RADIO_frame.frame_byte[6] << 8;
                                                        g_sensor_data.rx_last_parameter[0] |= RADIO_frame.frame_byte[7];
                                                        g_sensor_data.rx_last_parameter[1]  =
                                                                                                (uint16_t)
RADIO_frame.frame_byte[8] << 8;
                                                        g_sensor_data.rx_last_parameter[1] |= RADIO_frame.frame_byte[9];
                                                        break;
                                                default:
                                                        break;
                                        }
                                        break;

                        default:
                                break;
                }*/
                // end of switch case



                /*
                // --- Generate Timestamp: ------------------------------------------------------
                // Timestamp = [DAY HR MIN SEC MSEC] (32-bit):
                g_sensor_data.timestamp   = g_rx_timing.rtc_sof.day;            // days:    5-bit used
```

```
        g_sensor_data.timestamp <<= 5;                  // shift 5-bit for hours
        g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.hr;     // hours:   5-bit used
        g_sensor_data.timestamp <<= 6;                  // shift 6-bit for minutes
        g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.min;    // minutes: 6-bit used
        g_sensor_data.timestamp <<= 6;                  // shift 6-bit for seconds
        g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.sec;    // seconds: 6-bit used
        g_sensor_data.timestamp <<= 10;                 // shift 10-bit for msec
        g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.msec;   // milliseconds: 10-bit used
        */

}
//-------------------------------------------------------------------------------
```

C.2.4 isr_timera.c

```
/*-------------------------------------------------------------------------------
        Project:                BATSEN
        Discription:
        Used components:    MSP430-169STK

        Author:                 Stephan Plaschke
        Date:                   12/15/2007
        Last update:            04/04/2008

        Modified by:            Alexander Hoops
        Last modification:      12/02/2010

        Modified by:            Niels Jegenhorst
        Last modification:      01/06/2011

        Modified by:            Michael Meinzer
        Last modification:      20/09/2013

        File:                   isr_timera.c
-------------------------------------------------------------------------------

-------------------------------------------------------------------------
        Headerfiles
-------------------------------------------------------------------------*/
#include <main.h>
#include <signal.h>
#include <stdio.h>
#include <msp430x16x.h>
#include "headerfiles/lcd16x2.h"
#include "headerfiles/msp_functions.h"
#include "headerfiles/uart_menu.h"
#include "headerfiles/system_handling.h"
#include "headerfiles/sensor_data_proc.h"
#include "headerfiles/isr_timera.h"
#include "headerfiles/globals.h"

//-------------------------------------------------------------------------------
// calculate current frame error
//-------------------------------------------------------------------------------
void countFER(unsigned char sensor_number, unsigned char crc_error,
                                    unsigned short bit_too_long, unsigned short bit_too_short,
                                    unsigned short wrong_RunIn) {

        volatile short lost_frames = 0;
        unsigned short temp_error = 0, temp_error_old = 0;

        switch (fer) {

                //init frame error calc.
                case 1:

                        fer = 2;

                //frame error calc.
                case 2:

                        temp_error = (short)(RADIO_frame.frame_byte[1]);
                        temp_error <<= 8;
                        temp_error |= (short)(RADIO_frame.frame_byte[2]);
                        temp_error_old = (short)(SENSOR_last_data[1][sensor_number]);
                        temp_error_old <<= 8;
                        temp_error_old |= (short)(SENSOR_last_data[2][sensor_number]);

                        //lost frames is the difference of the current and the last transmission count
                        lost_frames = temp_error - temp_error_old - 1;
                        UART0_send('#'); //doublecross terminates transmission
                        UART0_send('2'); //number of digits that follows in D
                        UART0_send('1'); //number of bytes that follows in A
                        UART0_send('0');
                        UART0_send(SENSOR_address[sensor_number]);
                        UART0_send((unsigned char) (bit_too_long >> 8));
                        UART0_send((unsigned char) (bit_too_long & 0x00FF));
                        UART0_send((unsigned char) (bit_too_short >> 8));
                        UART0_send((unsigned char) (bit_too_short & 0x00FF));
                        UART0_send((unsigned char) (wrong_RunIn >> 8));
                        UART0_send((unsigned char) (wrong_RunIn & 0x00FF));
                        UART0_send((unsigned char) (lost_frames >> 8));
                        UART0_send((unsigned char) (lost_frames & 0x00FF));
                        UART0_send(crc_error);
```

```
                                //send new line
                                UART0_send_text("\n");
                                break;

                        default:
                                fer = 0;
                }
}
//-------------------------------------------------------------------------------


/*-------------------------------------------------------------------------
        Timer A0 interrupt service routine, Capture/compare0 interrupt   (bit recognition)
-------------------------------------------------------------------------*/
interrupt (TIMERA0_VECTOR) TIMER_A0(void)
/*-------------------------------------------------------------------------*/
{
        static unsigned char Temp_cal_counter[41]=
                {0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,
                 0};
        signed long temp_diff, real_Temp;
        // compare values for the received address in global array,
        // byte = position in array
        unsigned char address_bit, address_byte;
        unsigned char i, j, frame_error, ex, sens, sensor_calibrated, xor_backup;

        //-------------------------------------------------------------
        //calculate bit width
        //-------------------------------------------------------------
        TA0CTL &= ~(MC1 | MC0);                                 // stop  timer A
        TA0CTL |= TACLR;                                        // reset timer A
        TA0CTL |= (ID1 | ID0 | MC1);                            // start timer A first/again
        g_rx_timing.capture = TA0CCR0;                          // readout capture time
        g_rx_timing.rx_in   = (TACCTL0 & CCI) >> 3;             // read CCI0A input state

        //-------------------------------------------------------------
        //case differentiation - bit recognition
        //-------------------------------------------------------------
        switch(RADIO_rx_state)
        {
        case SE_SEQ_PREPARE:                    //_____________________________________________________

                // TODO Anpassungen Empfang für das neue Sendeprotokoll der Sensoren
                // first interrupt of a new rx frame

                #ifdef PIN_DEBUG
                        TP_ADC1_ON;
                        TP_INR_OFF;
                #endif
                g_rx_timing.shift_in = 0;                       // reset shift counter
                g_rx_timing.byte_ctr = 0;                       // reset byte counter
                g_rx_timing.rtc_sof.day  = RTC_values.day;      // Save actual RTC value:
                g_rx_timing.rtc_sof.hr   = RTC_values.hr;
                g_rx_timing.rtc_sof.min  = RTC_values.min;
                g_rx_timing.rtc_sof.sec  = RTC_values.sec;
                g_rx_timing.rtc_sof.msec = RTC_values.msec / 10;

                BATMON_control_reg &= !VALID_DATA_RECEIVED;     // clear global value
                g_rx_status    = ACTIVE;                        // set status
                RADIO_rx_state = SE_SEQ_START;                  // start sequence next time
                break;                                          // exit switch-case block

        case SE_SEQ_START:                              //_____________________________________________________

                // look if start sequence detected, SoF width is 700..1200us

                if( (g_rx_timing.capture < START_SEQ_LENGTH_MAX) &&
                    (g_rx_timing.capture > START_SEQ_LENGTH_MIN) ) {
                        RADIO_rx_state = SE_SEQ_PRE_WAIT;  // next RX state
                        // Debug meinzer
                        //g_rx_timing.shift_in = (14*2)+1;      // waitcounter set for 15 1/2 bits until
                        g_rx_timing.shift_in = (6*2)+1;         // Veränderung auf den verkürzten Run - IN.
                }                                               //              second byte received
                else {                                          // when time not equals:
                        TA0CTL &= ~(MC1 | MC0);                 // stop timer A
                        TA0CTL |= TACLR;                        // reset timer A
                        g_rx_status    = RUN_IN_ERROR;
                        RADIO_rx_state = SE_SEQ_PREPARE;   // set RX state for next start of frame
                        #ifdef ENABLE_RX_ERROR_LOG
                                g_rx_error.wrong_RunIn++;
                        #endif
                        #ifdef PIN_DEBUG
                                TP_ADC1_OFF;
                                TP_INR_ON;
                        #endif
                }
                break;                                          // exit switch-case block

        case SE_SEQ_PRE_WAIT:                   //_____________________________________________________

                // calibrate bitrate - cal. bit pattern is 0x00,0x01
                g_rx_timing.shift_in --;

                if (g_rx_timing.shift_in == 0) {                // calc bit lengths:

                        g_rx_timing.bit_tmax = (g_rx_timing.halfbit_tavg << 1) +        // t_halfbit * 2
                                                (g_rx_timing.halfbit_tavg >> 1);                        // +25%
                        g_rx_timing.bit_tmin = (g_rx_timing.halfbit_tavg << 1) -        // t_halfbit * 2
```

```
                                                    (g_rx_timing.halfbit_tavg >> 1);          // -25%

            g_rx_timing.halfbit_tmax = g_rx_timing.halfbit_tavg +       // t_halfbit
                                       (g_rx_timing.halfbit_tavg >> 2); // +25%
            g_rx_timing.halfbit_tmin = g_rx_timing.halfbit_tavg -       // t_halfbit
                                       (g_rx_timing.halfbit_tavg >> 1); // -50%

            RADIO_rx_state = SE_SEQ_DATA_START;                 // next RX state
        }
        else {                                                  // calc floating average:
            g_rx_timing.halfbit_tavg += g_rx_timing.capture;
            g_rx_timing.halfbit_tavg >>= 1;
        }
        break;                                                  // exit switch-case block

case SE_SEQ_DATA_START:         //_____________________________________________________

        // end of second byte (cal. pattern), check second byte (detect a '1' at the end)

                                                                // time for long high of transition "01":
        if( (g_rx_timing.capture < g_rx_timing.bit_tmax) &&
            (g_rx_timing.capture > g_rx_timing.halfbit_tmax) ){
            RADIO_rx_state = SE_SEQ_RXDATA;                     // next RX state
        }
        else {                                                  // nothing detected:
            TA0CTL &= ~(MC1 | MC0);                             // stop timer A
            TA0CTL |= TACLR;                                    // reset timer A
            g_rx_status    = SYNCH_ERROR;
            RADIO_rx_state = SE_SEQ_PREPARE;   // set RX state for next start of frame
            #ifdef ENABLE_RX_ERROR_LOG
                g_rx_error.wrong_RunIn++;
            #endif
            #ifdef PIN_DEBUG
                TP_ADC1_OFF;
                TP_INR_ON;
            #endif
        }
        break;                                                  // exit switch-case block

case SE_SEQ_RXDATA:                     //_____________________________________________________

        // catch manchester: ---------------------------------------------------------
                                                                // time for half bit length detected: -------
        if( (g_rx_timing.capture < g_rx_timing.halfbit_tmax) &&
            (g_rx_timing.capture > g_rx_timing.halfbit_tmin) ) {
                                                                // rising edge  --> shift in "1"
                                                                // falling edge --> shift in "0"
            g_rx_timing.rx_seq  <<= 1;
            g_rx_timing.rx_seq   |= g_rx_timing.rx_in;
            g_rx_timing.shift_in += 1;
        }                                                       // time for one bit length detected: --------
        else if( (g_rx_timing.capture < g_rx_timing.bit_tmax) &&
                 (g_rx_timing.capture > g_rx_timing.bit_tmin) ) {
                                                                // rising edge  --> shift in "01"
                                                                // falling edge --> shift in "10"
            g_rx_timing.rx_seq  <<= 1;
            g_rx_timing.rx_seq   |= ((!g_rx_timing.rx_in) & 0x01);
            g_rx_timing.rx_seq  <<= 1;
            g_rx_timing.rx_seq   |= g_rx_timing.rx_in;
            g_rx_timing.shift_in += 2;
        }
        else {                                                  // time to short or long, exit: ------------
            TA0CTL &= ~(MC1 | MC0);                             // stop timer A
            TA0CTL |= TACLR;                                    // reset timer A
            g_rx_status    = BIT_ERROR;
            RADIO_rx_state = SE_SEQ_PREPARE;   // set RX state for next start of frame
            #ifdef ENABLE_RX_ERROR_LOG
                g_rx_error.bit_error++;
            #endif
            #ifdef PIN_DEBUG
                TP_ADC1_OFF;
                TP_INR_ON;
            #endif
            break;                                              // exit switch-case block
        }
        // Zusatz : Sonderfall nur ein halbes Byte prüfen
            if(g_rx_timing.byte_ctr == 7) {
                if(g_rx_timing.shift_in == 8) {     // one byte complete: ----------------------
                    RADIO_frame.frame_byte[g_rx_timing.byte_ctr] = 0x00;
                    g_rx_timing.rx_in    = 0x01;                // init for bit setting in frame_byte
                    g_rx_timing.shift_in = 0x0002;              // init for manchester sequenz "10"
                    while(TRUE) {
                                // manchester sequenz "10" --> 1:
                                if(g_rx_timing.rx_seq & g_rx_timing.shift_in) {
                                //              then set bitvalue in byte
                                RADIO_frame.frame_byte[g_rx_timing.byte_ctr] |= g_rx_timing.rx_in;
                        }       // manchester sequenz "01" --> 0:
                                //              then nothing to set in byte
                                if(g_rx_timing.shift_in != 0x8000) {
                                    g_rx_timing.rx_in <<= 1;        // shift to next possible bit in byte
                                    g_rx_timing.shift_in <<= 2;     // shift to next manchester sequenz
                                }
                                else        // when manchster sequenz decoded:
                    break;      //   exit loop while(TRUE)
                    }
                                g_rx_timing.shift_in = 0;                       // reset counter for next sequenz
                                g_rx_timing.byte_ctr ++;                        // increment byte counter
                                // Anpassung Aufgrund der Manchesterdecodierung
                                // Diese schiebt die Bits von rechts nach links ins Byte
                                // durch das halbe Byte kommt es sonst zu Positionsfehlern.
```

```
                                        RADIO_frame.frame_byte[7] = (RADIO_frame.frame_byte[7] << 4);
                                        }
                        }
                        else {      // decode manchester: ------------------------------------------------------------
                        if(g_rx_timing.shift_in == 16) {// one byte complete: ----------------------
                        RADIO_frame.frame_byte[g_rx_timing.byte_ctr] = 0x00;
                        g_rx_timing.rx_in    = 0x01;                // init for bit setting in frame_byte
                        g_rx_timing.shift_in = 0x0002;              // init for manchester sequenz "10"
                        while(TRUE) {       // manchester sequenz "10" --> 1:
                                if(g_rx_timing.rx_seq & g_rx_timing.shift_in) {
                                //              then set bitvalue in byte
                                RADIO_frame.frame_byte[g_rx_timing.byte_ctr] |= g_rx_timing.rx_in;
                                }           // manchester sequenz "01" --> 0:
                                //  then nothing to set in byte
                                if(g_rx_timing.shift_in != 0x8000) {
                                        g_rx_timing.rx_in <<= 1;        // shift to next possible bit in byte
                                        g_rx_timing.shift_in <<= 2;     // shift to next manchester sequenz
                                }
                                else        // when manchster sequenz decoded:
                                        break;      //   exit loop while(TRUE)
                        }
                        g_rx_timing.shift_in = 0;                   // reset counter for next sequenz
                        g_rx_timing.byte_ctr ++;                    // increment byte counter
                }
        }


                //---------------------------------------------------------------------------
                // end of bitstream: ------------------------------------------------------
                // Es muss 8 sein damit das letzte halbe BYTE noch durchkommt
                if (g_rx_timing.byte_ctr == 8) {
                        TA0CTL &= ~(MC1 | MC0);             // stop timer A
                        TA0CTL |= TACLR;                    // reset timer A
                        i=0;                                // init address counter
                        switch (scan_mode) {
                                //---------------------------------------------------------------------------
                                // scan mode started
                                //---------------------------------------------------------------------------
                                // Scanmode verändert, Adresse des Sensors in Byte 1
                                case 1:
                                        // check if address valid (if it is already existent),skip Address 0
                                        while( (SENSOR_address[i] != 0 ) &&
                                                (SENSOR_address[i] != RADIO_frame.frame_byte[1]) ) {
                                                i++;
                                        }

                                        //Sensor not yet existent - print info to UART
                                        if (SENSOR_address[i] != RADIO_frame.frame_byte[1]) {
                                                PRINTF_UART;
                                                printf("Sensor: %2x => Adresse: %2x\n",
                                                                i+1, RADIO_frame.frame_byte[1]);
                                        }

                                        SENSOR_address[i] = RADIO_frame.frame_byte[1]; //save sensor address

                                        //desired number of sensors found
                                        if (i >= SENSOR_active-1) {
                                                scan_mode = 0;                                          // stop scan mode
                                                LCD_send_cmd(LCD_LINE1);
                                                recording_stop();                                       // disable radio unit
                                                LCD_send_text(" Scan complete  ");
                                                UART0_send_text("Scan complete\n");
                                                BATMON_control_reg &= ~VALID_DATA_RECEIVED;             // set global value
                                                FLASH_Write_Controll_Data();                            // write gathered control
                                                                                                        // data to flash
                                                RADIO_disable();                                        // stop reveiver
                                                g_rx_status = INACTIVE;

                                                for (i=0; i<SENSOR_active; i++) {                       // set sensors active:
                                                        SENSOR_last_data[0][i] = SENSOR_address[i];
                                                }

                                                switch(SENSOR_active) {
                                                        case 12:
                                                                SENSOR_addr_req[0] = 0xFF;  // 12 sensors are configured
                                                                SENSOR_addr_req[1] = 0x0F;
                                                                SENSOR_addr_receiv[0] = 0xFF;
                                                                SENSOR_addr_receiv[1] = 0x0F;
                                                                break;
                                                        case 24:
                                                                SENSOR_addr_req[0] = 0xFF;  // 24 sensors are configured
                                                                SENSOR_addr_req[1] = 0xFF;
                                                                SENSOR_addr_req[2] = 0xFF;
                                                                SENSOR_addr_req[3] = 0x00;
                                                                SENSOR_addr_req[4] = 0x00;
                                                                SENSOR_addr_receiv[0] = 0xFF;
                                                                SENSOR_addr_receiv[1] = 0xFF;
                                                                SENSOR_addr_receiv[2] = 0xFF;
                                                                SENSOR_addr_receiv[3] = 0x00;
                                                                SENSOR_addr_receiv[4] = 0x00;
                                                                break;
                                                        case 40:
                                                                SENSOR_addr_req[0] = 0xFF;  // 40 sensors are configured
                                                                SENSOR_addr_req[1] = 0xFF;
                                                                SENSOR_addr_req[2] = 0xFF;
                                                                SENSOR_addr_req[3] = 0xFF;
                                                                SENSOR_addr_req[4] = 0xFF;
                                                                SENSOR_addr_receiv[0] = 0xFF;
                                                                SENSOR_addr_receiv[1] = 0xFF;
                                                                SENSOR_addr_receiv[2] = 0xFF;
```

```
                                                SENSOR_addr_receiv[3] = 0xFF;
                                                SENSOR_addr_receiv[4] = 0xFF;
                                                break;
                                        default:
                                                SENSOR_addr_req[0] = 0x3F;  // 6 sensors are configured
                                                SENSOR_addr_receiv[0] = 0x3F;
                                                break;
                                }
                        }                       // End if desired number of sensors found
                        #ifdef PIN_DEBUG
                                TP_ADC1_OFF;
                        #endif
                        break;

                //-----------------------------------------------------------------
                // Calibration initialization
                //-----------------------------------------------------------------
                case 2:
                        for (sens=0;sens<41;sens++)    {                        // clear temp variables
                                Temp_cal_counter[sens]=0;
                        }
                        scan_mode = 3;
                        RADIO_enable();
                        break;
                //-----------------------------------------------------------------
                // Calibration started
                //-----------------------------------------------------------------
                case 3:
                        real_Temp = (RADIO_frame.frame_byte[4]);
                        real_Temp = real_Temp << 8;
                        real_Temp |= RADIO_frame.frame_byte[5];
                        ex = Temp_cal_counter[RADIO_frame.frame_byte[3]];
                        if (ex==0) {
                                SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]] =
                                                temp_correct_value - real_Temp;
                        }
                        else if((ex>=1) && (ex<20)) {
                                temp_diff = (temp_correct_value - real_Temp) +
                                                        (ex * SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]]);
                                SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]] = temp_diff /(ex+1);
                        }
                        Temp_cal_counter[RADIO_frame.frame_byte[3]]++;
                        sensor_calibrated = 0;
                        for (sens=0;sens<SENSOR_active;sens++) {
                                if (Temp_cal_counter[SENSOR_address[sens]]>5) {
                                        sensor_calibrated++;
                                }
                        }
                        if (sensor_calibrated == SENSOR_active) {
                                scan_mode = 0;
                                LCD_send_cmd(LCD_LINE1);
                                recording_stop();                                                       // disable radio
unit
                                LCD_send_text("Calibr. complete");
                                UART0_send_text("Calibration complete\n");
                                g_rx_status = INACTIVE;
                                RADIO_init_frame();
                                FLASH_Write_Controll_Data();
                        }
                        else {
                                BATMON_control_reg &= ~VALID_DATA_RECEIVED;             // clear global value
                        }
                        break;
                //-----------------------------------------------------------------
                // scan mode '0' no certain mode
                //-----------------------------------------------------------------
                case 0:
                        do {    // check if valid sensor:
                                // compare if sensor address is allowed
                                // Debug am 25.07 Neues Protokoll
                                if (SENSOR_address[i] == RADIO_frame.frame_byte[1])
                                        break;  // exit loop, get sensor address index
                                i++;
                        } while(i<40);

                        if (i<40) {             // if sensor address valid, maximum of 40 sensors
                                RADIO_frame.sensor_index = i;

                                j = 0;
                                frame_error = 0;
                                // Korrektur des CRC Polynoms notwendig.. da Position 6.5 - 7.5 Byte..
                                // Anpassungen an das CRC Polynom, da sich dieses in 2 Bytes befindet
                                xor_backup = 0x00;
                                xor_backup = (RADIO_frame.frame_byte[6] & 0x0F) << 4;
                                xor_backup = ((RADIO_frame.frame_byte[7] & 0xF0) >> 4) + xor_backup;
                                // Rücksetzen des CRC Wertes im Frame
                                RADIO_frame.frame_byte[7] = 0x00;
                                RADIO_frame.frame_byte[6] &= 0xF0;

                                while(j<7) {                                    // check CRC
                                        frame_error ^= RADIO_frame.frame_byte[j];
                                        j++;
                                }
                                // Zusatzprüfung des CRC Bytes
                                frame_error ^= xor_backup;

                                if (frame_error) {      // CRC not valid, exit
                                        g_rx_status    = CRC_FAILURE;
                                        RADIO_rx_state = SE_SEQ_PREPARE;
                                        #ifdef ENABLE_RX_ERROR_LOG
```

```
                                                    g_rx_error.crc_error[RADIO_frame.sensor_index]++;
                                                #endif
                                                #ifdef PIN_DEBUG
                                                    TP_ADC1_OFF;
                                                    TP_INR_ON;
                                                #endif
                                                break;                                  // exit switch-case block
                                            }
                                            #ifdef ENABLE_RX_ERROR_LOG
                                                if(fer==1) {        // reset all error variables for startup
                                                    g_rx_error.bit_error = 0;
                                                    g_rx_error.wrong_RunIn = 0;
                                                }
                                                if(fer>0) {                 // FER calc.
                                                    countFER(RADIO_frame.sensor_index,
                                                        g_rx_error.crc_error[RADIO_frame.sensor_index],
                                                        g_rx_error.bit_error,
                                                        g_rx_error.bit_error,
                                                        g_rx_error.wrong_RunIn);
                                                }                           // reset error count variables:

                                                convert_sensor_data();      // convert sensor data
                                                g_rx_status = SUCCESS;      // set flag
                                                BATMON_control_reg |= VALID_DATA_RECEIVED; // set flag
                                                #ifdef PIN_DEBUG
                                                    TP_ADC1_OFF;
                                                #endif
                                            //}
                                        }                   // end of "if (i<40) {}"
                                        break;
                                default:
                                        scan_mode = 0;      }                   // end of switch case (scan mode)
                                                                                // Receiving frame completed (valid or not),
                            RADIO_rx_state = SE_SEQ_PREPARE;// return to the beginning
                    }                                                           // end of "if (.frame_bit_counter == 88) {}"
                    break;                                                      // exit switch-case block

            default:                                    //_____________________________________________________
                    RADIO_rx_state = SE_SEQ_PREPARE;
                    break;                                                      // exit switch-case block
            }
            TA0CCTL0 &= ~CCIFG;                                                 // reset IRQ flag, deletes unwanted captures
                                                                                // (reset of IRQ flag is done by calling ISR)
}
/*-------------------------------------------------------------------------------
            Timer A1 interrupt service routine for compare register 1 interrupt
-------------------------------------------------------------------------------*/
interrupt (TIMERA1_VECTOR) TIMER_A1(void)
/*-------------------------------------------------------------------------------*/
{

                    // This ISR stops the timer A after a runtime of START_SEQ_LENGTH_MAX
                    // without any capture on CCI0A.

            TA0CTL &= ~(MC1 | MC0);                                 // stop timer A
            TA0CTL |= TACLR;                                        // reset timer A

            g_rx_status    = TIMEOUT;
            RADIO_rx_state = SE_SEQ_PREPARE;            // set RX state for next start of frame
            #ifdef ENABLE_RX_ERROR_LOG
                    g_rx_error.timeout ++;
            #endif
            #ifdef PIN_DEBUG
                    TP_INR_ON;
                    TP_ADC1_OFF;
            #endif
            TA0CCTL1 &= ~CCIFG;                                             // reset IRQ flag
}
//-------------------------------------------------------------------------------
```

D Matlab-Skript

D.1 Skript für die Aufzeichnung der Daten

```
% Testfile zum Auslesen der Basistation mithilfe von Matlab.
% Ziel empfangene Messwerte auslesen, dieses Skrip erwartet eine
% vorher gehende Konfiguration mit dem Programm HTerm für die
% Aufzeichnung der Daten
% Quelle Arbeit Puettjer

function serial_read_binblock(comport,size,dateiname)

%clear all;
%close all;

%% Comport connect
% Comport mit 115200 Baud und Terminator öffnen
serial_port = serial(comport,'Baudrate',115200,'Terminator','CR');
fopen(serial_port);
set(serial_port,'Timeout',50);
% Startcommando senden für Aufzeichnungsbeginn
% fprintf(serial_port, 'start');
pause(4);

h = waitbar(0,'empfange Daten');
%% Einleseschleife
i = 1;
x = 0;
z = 0;
while 1
    [data,count,msg] = binblockread(serial_port,'uint8');
    if ~isempty(msg);
        break;
    end
    data2(i,:) = uint8(data');

    %% try to cast Data for Save and Edit
    % Edit am 20.08 auf neuen Zeitstempel
    data3(i,:) = [uint32(data2(i,1)),uint32(data2(i,2))...
..,uint32(swapbytes(typecast(data2(i,3:6),'uint32')))...
..,uint32(data2(i,7)),uint32(swapbytes(typecast(data2(i,8:11),'uint32')))
,uint32(data2(i,12)),uint32(data2(i,13)),uint32(data2(i,14))...
...,uint32(data2(i,15)),uint32(swapbytes(typecast(data2(i,16:17),'uint16'
))),uint32(swapbytes(typecast(data2(i,18:21),'uint32')))];
    i = i+1;
    % Zwischenspeichern
    if x == 30
        save(['dump_' num2str(z) dateiname],'data3');
        x = 0;
        z = z + 1;
    end
    % Einfache Aktivitätsanzeige
    message = sprintf('%i gültige Frames empfangen',i);
    waitbar(i/size,h,message);

     if(i >= size)
         break;
     end
     x = x + 1;
```

```
end
close(h);

%% Comport schliessen
fclose(serial_port);
delete(serial_port);
clear serial_port;

%% In eine CSV Datei schreiben
csvwrite(dateiname,data3);
clear all;
close all;
```

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, 08. Oktober 2013

Ort, Datum Unterschrift