Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Diplomarbeit

Simon Püttjer

Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellsensoren

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Simon Püttjer

Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellsensoren

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Informations- und Elektrotechnik
Studienrichtung Kommunikationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr.Ing. Jürgen Vollmer

Abgegeben am 17. Juni 2011

Simon Püttjer

Thema der Diplomarbeit

Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellsensoren

Stichworte

Sensorik, Starterbatterie, Batterie-Management, Ladezustand, drahtlos, Automobile Elektronik

Kurzzusammenfassung

Die Zuverlässigkeit der Energieversorgung von Automobil-Bordnetzen wird in den nächsten Jahren zusehends wichtiger werden. Besonders bei der Versorgung sicherheitsrelevanter Systeme (X-by-Wire) ist dies zu beachten. Entgegen bereits etablierter Methoden der Batterieüberwachung über ihre Gesamtspannung erzielt die Überwachung jeder einzelnen Zelle bessere Ergebnisse bei der Beurteilung der verbleibenden Kapazität. Diese Arbeit setzt auf bisherige Erkenntnisse der drahtlosen Zellüberwachung auf und behandelt dabei ihre Umsetzung auf die Starterbatterie.

Simon Püttjer

Title of the paper

Wireless Battery Diagnostic of Automotive SLI Batteries

Keywords

battery, SLI, diagnostic, wireless, monitoring, automotive electronics, battery management, on-board power supply

Abstract

The demand on the reliability of automotive vehicles power supply will steadily increase in the future. This is for security related systems (X-by-Wire) especially important to note. Contrary to already established methods of battery management by its overall voltage, monitoring of each single cell, results in better evaluation of its remaining capacity. This thesis builds on previous knowledge of wireless cell monitoring and deals with its implementation on automotive SLI batteries.

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 8. Februar 2011

Diplomarbeit Simon Püttjer:

Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellensensoren

Motivation

Starterbatterien liefern die Energie für den Startvorgang der Verbrennungsmotoren der Automobile. Darüber hinaus puffern sie das Automobil-Bordnetz mit immer mehr elektrischen Verbrauchern. In dieser Rolle als Pufferspeicher kommt in wenigen Jahren eine sicherheitskritische Funktion hinzu, wenn mit elektrischer Energie gelenkt und gebremst werden wird (X-by-Wire).
An der HAW Hamburg ist im Rahmen eines vom Bundesministerium für Bildung und Forschung geförderten Forschungsvorhabens BATSEN (Drahtlose Zellensensoren für Fahrzeugbatterien) ein Konzept entstanden, mögliche Funktionseinschränkungen frühzeitig zu erkennen. Bei dem im Vorhaben verfolgten neuartigen Lösungsansatz werden Messwerte von jeder einzelnen Batteriezelle aufgenommen und drahtlos übertragen. Dabei müssen Teile der Vorverarbeitung der Messwerte in den Sensor selbst als Mikrocontroller-Software implementiert werden.
In Vorarbeiten ist erkannt worden, dass eine statistisch mittelnde Erfassung von Zellenwerten mit einer Datenerfassung im Minutenbereich nicht ausreicht, den wechselnden Lang- und Kurzzeitbelastungen der Starterbatterie zu genügen. Beispielsweise stehen sich gegensätzlich gegenüber: sehr langzeitige Stromentnahmen in der Ruhephase im Bereich von wenigen mA und sehr kurzzeitige Ereignisse mit Strömen über 100 A, jedoch mit weniger als 1 Sekunde Gesamtdauer.
Motivation der Diplomarbeit ist, die Konzepte und Rahmenbedingungen für entsprechende Betriebsarten der Zellen-Sensorik zu finden und praktische Erfahrungen in dieser Aufgabe zu sammeln.

Aufgabe

Herr Püttjer erhält die Aufgabe, die Rahmenbedingungen der Sensorik für den kurzeitigen Hochstrombetriebs- am Beispiel des Startvorgangs - zu untersuchen und eine wesentlich zeitlich dichtere Messwerterfassung zu konzipieren. Dazu passend ist eine Untersuchung der entstehenden höheren Belastung des drahtlosen Kommunikationskanals notwendig, sowie die Erarbeitung von Konzepten, diesen günstig auszunutzen.
Die Zellenüberwachung erfolgt also in einem Zusammenwirken aus verteilter Datenerfassung und Vorverarbeitung und der Kommunikationsprotokolle. Herr Püttjer erhält die Aufgabe ausgewählte Varianten der Sensor- und Datenerfassungsoftware neu zu implementieren und hinsichtlich realisierbarem Aufwand, erreichbarer Messgegenauigkeit und aussagefähiger Ergebnisse zu untersuchen. Bei der Umsetzung sollen Controller der MSP430-Familie, der GNU C-Compiler, die Entwicklungsumgebung Eclipse und das Matlab-Tool-Paket sowie ein eigener Datenlogger-Aufbau eingesetz werden.
Es ist eine Funktionsdemonstration mit einem praktischen Aufbau an einer Starterbatterie abschließend vorgesehen.
Zur Untersuchung der entstehenden höheren Belastung des drahtlosen Kommunikationskanals sollen entsprechende Kommunikationsteuerungen in die Controller der Sensoren implementiert werden.

Für die Diplomarbeit sind die folgenden Arbeitspakete geplant:

1) **Einführung, Konzeption und Rahmenbedingungen**
    - Einarbeitung in die Projektzielstellung mit besonderer Bezug auf die Starterbatterie
    - Analyse der Vorarbeiten und theoretische Darstellung der bisherigen Betriebsart
    - Abschätzungen von Anforderungen und Grenzen
2) **Modifikation einer Starterbatterie, Vorbereitung der Hardware für die Messungen**
    - Umbau und Ausrüstung einer Starterbatterie mit Zellensensoren
    - Anpassung, Fehlerbeseitigung und Verbessung des Datenloggers
    - Praktischer Aufbau einer verbesserten Variante des Datenloggers, insbesondere mit einer Stromerfassung für die Lade- und Entladevorgänge in zwei Messbereichen
    - Erstellung von Matlab-Skripten zur Analyse von Oszilloskop-Messreihen
    - Erstellung und Inbetriebnahme von Messaufbauten
3) **Praktische Erfassung der Rahmenbedingungen der Hochstromereignisse**
    - Erfassung der zeitlichen Abläufe von Batteriestrom und Spannung an verschieden Fahrzeugen beim Startvorgang
    - Aufbau eines Anschlusses zum externen Betrieb einer messtechnisch präparierten Batterie
    - Zellenweise Erfassung durch angepasste Adapterhardware
    - Berücksichtigung einer guten Auflösung (zeitlich und Spannung) auch unter hohen Offsetspannungen
    - Auswertung der Daten mit gekoppelten Speicheroszilloskopen und Matlab
4) **Verfahrensanalyse und Optimierung**
    - Allgemeine Darstellung des unsynchronisiertem Kommunikationsverfahrens mit Übertragungskonflikten
    - Messtechnische Untersuchung der Übertragungseffizienz und der Verlustraten an Messwerten
    - Gegenüberstellung von Verfahren mit ungesteuerten Mess- und Kommunikationsverfahren gegen Verfahren mit lokaler Triggerung durch Ereignisdetektion (z.B. schneller Spannungsabfall bei Hochstrom)
5) **Verfahrensimplementierung, Laborerprobung und Variantenvergleich**
    - Einbringen der neuen Verfahren in die Software der Sensoren und Datenloggers
    - Funktionsnachweis durch Signaleinspeisung in den Sensor, z.B. von einer variablen Spannungsquelle oder von einem arbiträren Signalgenerator
    - Durchführung von Laborerprobung und Softwaretests
    - Darstellung der Software-Struktur der Modifikation
6) **Erprobung und Messung mit der modifizierten Batterie**
    - Durchführung ausgewählter Messreihen mit Hochstrom-, Normalbetriebs- und Ruhezuständen
    - Vergleich dieser Messungen
7) **Auswertung und Bewertung**
    - Grafische Darstellung und Dokumentation der Messergebnisse
    - Vergleich der Verfahren, ev. Abweichungen der Ergebnisse u.a. Besonderheiten
    - Diskussion und Bewertung des Sensorsignals anhand der Erprobungserfahrung und ggf. der Messdaten

Dokumentation

Die Fachliteratur, die Vorarbeiten und die kommerziellen Unterlagen sind zielgerichtet zu recherchieren. Dabei sind wichtige Grundlagen und die vorgesehene Anwendung näher zu betrachten. Ebenso ist die Berücksichtigung der Messverfahrens-Varianten und -Parameter notwendig, um deren Potential beurteilen zu können. Die Funktionsweise und die Struktur der Software-Module ist 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. Sie sind auszuwerten und als Diagramme zusammenfassend darzustellen. Die realisierten Lösungen und die Ergebnisse sind kritisch einordnend zu bewerten. Ansätze für Verbesserungen und weitere Arbeiten sind zu nennen.

Danksagung

Das Erstellen umfangreicherer Arbeiten erfordert immer auch die Unterstützung durch weitere Personen. Sei es durch technische, fachliche oder auch menschliche Kompetenzen. An dieser Stelle möchte ich deshalb allen Menschen danken, durch die ich in der Lage war diese Arbeit zu bewältigen.

An erster Stelle bedanke ich mich bei meinem Erstprüfer, Herrn Prof. Dr.-Ing. Karl-Ragmar Riemschneider. Sein fachliches Wissen und stetiges Engagement halfen mir besonders beim Herangehen an auftauchende Problemstellungen. Ohne ihn wäre zudem die Erstellung dieser Arbeit nicht möglich gewesen.
Herr Prof. Dr. Jürgen Vollmer war besonders bei nachrichtentechnischen Fragen eine große Hilfe. Er nahm sich stets die Zeit Problemstellungen zu diskutieren und so zu deren Lösung beizutragen. Für die Übernahme der Bewertung meiner Arbeit als Zweitprüfer danke ich ihm sehr.

Nicht zu vergessen ist der technische Beistand durch die Mitarbeiter des Labors für Informationstechnik Herrn Gerhard Wolff und Herrn Jörg Pflüger. Sie waren immer bemüht, mir Materialien und Werkzeug zur Verfügung zu stellen. Gerade ihre Erfahrung in der praktischen Konstruktion von Messaufbauten schätze ich sehr.

Dank auch an Dipl.-Ing. Günter Müller, der mir bei den Messungen und der Korrektur half.

Sehr lobenswert ist auch das unermüdliche Engagement von Herrn Dipl.-Ing. Matthias Schneider, der dem Forschungsprojekt BATSEN in diesem Jahr als wissenschaftlicher Mitarbeiter hinzugestoßen ist. Gerade gegen Ende dieser Arbeit stand er mir immer mit Rat und Tat zur Seite.

Dank gebührt ebenso allen weiteren Kommilitonen, die zur selben Zeit ihre Abschlussarbeiten bearbeiteten. Sie trugen maßgeblich zu der sehr entspannten Stimmung im Labor bei. Besonders danken möchte ich an dieser Stelle Niels Jegenhorst, der zur selben Zeit mit seiner Master-Arbeit beschäftigt war. Mit ihm konnte ich viele technische Details diskutieren, was ich als sehr wertvoll erachte.

Zuletzt danke ich noch meinen Freunden und meiner Familie, die während der Erstellung dieser Arbeit das eine oder andere Mal auf mich verzichten mussten. Ihre moralische Unterstützung bedeutet mir sehr viel.

Inhaltsverzeichnis

Tabellenverzeichnis 10

Abbildungsverzeichnis 11

1. Motivation und Einführung 14

1.1. Starterbatterien . . . 15

1.1.1. Aufgaben und Anforderungen im Kfz-Bordnetz . . . 16

1.1.2. Aufbau und Eigenschaften von Blei-Batterien . . . 21

1.2. Batteriesensorik und –management . . . 24

1.2.1. Ladezustandsbestimmung der Batterie . . . 25

1.2.2. Einsatz von Batteriesensorik . . . 26

1.3. Einzelzellüberwachung . . . 28

1.3.1. Konzept von drahtlosen Sensornetzen . . . 29

1.3.2. Realisierung des Sensors . . . 30

1.3.3. Realisierung des Empfängers . . . 34

1.4. Anwendung der drahtloser Sensorik auf Kfz Starterbatterien . . . 38

1.4.1. Vorüberlegungen zu Anforderungen an das System . . . 39

1.4.2. Vorbewertung des Sendeverfahrens . . . 39

2. Analyse 41

2.1. Betriebsverhalten von Starterbatterien . . . 41

2.1.1. Batterieverhalten im Hochstrombetrieb . . . 42

2.1.2. Messaufbau und Durchführung . . . 43

2.1.3. Auswertung der Messdaten . . . 44

2.2. Limitierung realer Systeme . . . 51

2.2.1. Sender/Zellsensor . . . 51

2.2.2. Empfänger/Datenlogger . . . 55

2.3. Übertragungskanal . . . 58

2.3.1. Framefehler . . . 58

2.3.2. Messung der Framefehlerrate . . . 60

2.4. Auswertung und Konzeption . . . 64

3. Realisierung **70**
3.1. Konstruktion und Änderung verwendeter Hardwarekomponenten 70
3.1.1. Sensorsystem 70
3.1.2. Modifikation der Starterbatterie und Konstruktion von Messaufbauten . 74
3.2. Verfahrensimplementation 77
3.2.1. Sensor 77
3.2.2. Datenlogger 81
3.3. Erprobung 82
3.3.1. Laborbetrieb 82
3.3.2. Betrieb am Kfz 85

4. Auswertung und Fazit **92**
4.1. Zusammenfassung 92
4.2. Bewertung erzielter Ergebnisse 93
4.3. Fazit 95

5. Ausblick **97**

Literaturverzeichnis **99**

Anhang **101**
A. Quelltext 101
A.1. Sensor 101
A.2. Datenlogger 114
B. Matlab Quellcode 205
B.1. Framefehler Auswertung 205
B.2. Auswertung zu den Messungen an der Starterbatterie 209
B.3. Auswertung und Analyse der Sensordaten 211
C. Sonstiger Anhang 217

Tabellenverzeichnis

1.1. Eigenschaften häufig genutzter Akkumulatoren 20
1.2. Ladezustand über die Ruhespannung 26
1.3. Übertragungsprotokoll des Zellensensors 34
1.4. Biterkennung nach Regeln 37

2.1. Einstellungen der Oszilloskope 44
2.2. Vergleich der Fahrzeuge 49

3.1. Messbereiche des DHAB S/24 72
3.2. Verwendete Starterbatterien 74
3.3. Änderung des Übertragungsprotokoll für das neue Übertragungsverfahren (ohne RunIn und SOF) 81
3.4. Umrechnung der Schwellwerte zur Wahl des Messbereichs 82

Abbildungsverzeichnis

1.1. ADAC Pannenstatistik 2010 14
1.2. Allgemeine Zusammenhänge im Kfz-Bordnetz 16
1.3. Beispiel Bordnetz: VW Phaeton 17
1.4. Das Funktionsprinzip des Bleiakkumulators 21
1.5. Aufbau einer Starterbatterie 23
1.6. Batterie- und Energiemanagement im Zusammenhang 25
1.7. BMW, Hella, Autokabel: intelligenter Batteriesensor 27
1.8. Blockschaltbild des IBS2006 28
1.9. Ladezustand einzelner Zellen 28
1.10. Konzept drahtloser Batteriesensorik 29
1.11. Aufbau des Zellsensors, modifiziert nach [14] Abb. 2.2 30
1.12. Realisierung des Zellensensors 31
1.13. Flussdiagramm der Senderoutine 33
1.14. Aufbau des Steuergeräts 35
1.15. Praktischer Aufbau des Steuergeräts 36
1.16. Erkennung der Bits aus dem Empfangssignal 37
1.17. Ablauf des Timerinterrupts im Steuergerät 38
1.18. Anforderungen in verschiedenen Betriebszuständen 39
1.19. Darstellung des Übertragungsverfahrens auf Frame-Ebene 40

2.1. Schematische Darstellung der Teilspannungen an der Batterie 42
2.2. Messaufbau der Hochstrommessung 43
2.3. Zellspannungen und Laststrom im Startmoment eines VW Passat Diesel über 10s 45
2.4. Zellspannungen und Laststrom im Startmoment eines Mercedes Benz Vito Diesel über 4s 46
2.5. Zellspannungen im Startmoment eines Audi A4 über 10s 47
2.6. Zellspannungen im Startmoment eines VW Passat Diesel über 10s mit geschwächter Zelle (Zelle 3) 48
2.7. Vergleich abschnittsweiser zu „starrer“ Mittelung des Hochstromereignisses . 52
2.8. Darstellung der Versorgungsspannung des Sensors im Sendefall 53
2.9. ASH Block Diagramm 55

2.10. Nutzdatensignal und Störungen auf dem Kanal: Sendeseite (oben), Empfängerseite (unten) . . . 56
2.11. Rekonstruktion des Digitalsignals am Empfänger . . . 57
2.12. Abschätzung der Framefehlerrate . . . 59
2.13. Überlagerung der Zeitfenster . . . 59
2.14. Verhältnisse verschiedener Fehlerfälle zum Gesamtverlust an Frames pro Sensor (Empfänger ohne Modifikation) . . . 61
2.15. Darstellung theoretischer und gemessener Framefehlerraten mit modifiziertem Empfänger . . . 62
2.16. Timings der Frames bei Annahme fester Sendezeiten von 500ms . . . 63
2.17. Schichtenmodell der Kommunikation zwischen Sensor und Datenlogger . . . 65
2.18. Simulation des Datenpuffers im Sensor, Passat 40s . . . 67
2.19. Simulation des Datenpuffers im Sensor, Audi A4 40s . . . 68
2.20. Simulation des Datenpuffers im Sensor, Audi A4 3x an bzw. aus 40s . . . 69

3.1. Schaltplan der DR5100 Empfängerplatine . . . 71
3.2. Änderungen der Strommessung am Developmentboard . . . 73
3.3. Verwendeter Hall-Sensor mit Anschlusskabel . . . 73
3.4. Geöffnete Starterbatterie von Moll . . . 75
3.5. Modifizierte Starterbatterie mit Zellsensoren . . . 75
3.6. Messadapter für Hochstrommessungen mit dem Oszilloskop . . . 76
3.7. Flussdiagramm der Messwertaufnahme . . . 78
3.8. Darstellung der Queue als Ringspeicher . . . 79
3.9. Zustandsdiagramme des Übertragungsteils . . . 80
3.10. Schematischer Aufbau bei der Laborerprobung . . . 83
3.11. Gemessenes Zeitverhalten der Zwischenspeichers . . . 84
3.12. Erprobung am Signalgenerator mit 6 Sensoren parallel . . . 85
3.13. Praktischer Aufbau der Erprobung in der Fahrzeughalle . . . 86
3.14. Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 20s . . . 88
3.15. Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 20s, gezoomt 89
3.16. Sensormessdaten beim Startvorgang im Vergleich, VW Passat über 40s . . . 90
3.17. Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 40s, 3-fach-Start . . . 91

4.1. Darstellung der Abhängigkeiten des gesamten Systems im Überblick . . . 95

5.1. Mögliches Sendeverfahren ohne festen Zeitanteil . . . 98

2. Konstruktionszeichnung des Batterie-Deckels der Starterbatterie von Banner (Verhältnis 1:2) . . . 218

3. Konstruktionszeichnung des Batterie-Deckels der Starterbatterie von Moll (Verhältnis 1:2) 219

1. Motivation und Einführung

Starterbatterien zählen seit je her zu den wichtigsten Energiespeichern am Markt. Gerade in der heutigen Zeit, deren Fortschritt immer mehr Energie aus dem Bordnetz fordert, wird die Starterbatterie ein immer wichtigerer Bestandteil im Auto. Ohne einen zuverlässigen Energiespeicher fielen bei einem Defekt sämtliche elektrische Verbraucher unvorhergesehen aus. – Es käme zur Panne. Schon heutzutage zählt die Kfz-Elektrik neben Motor und Motormanagement zu den häufigsten Ursachen einer Panne . Laut der aktuellen Pannenstatistik des ADAC belegte im Jahr 2010 defekte Elektrik einen Anteil von 53,7% (Abb. 1.1). Hiervon erwies sich in 75% der Fälle die Starterbatterie als Auslöser.

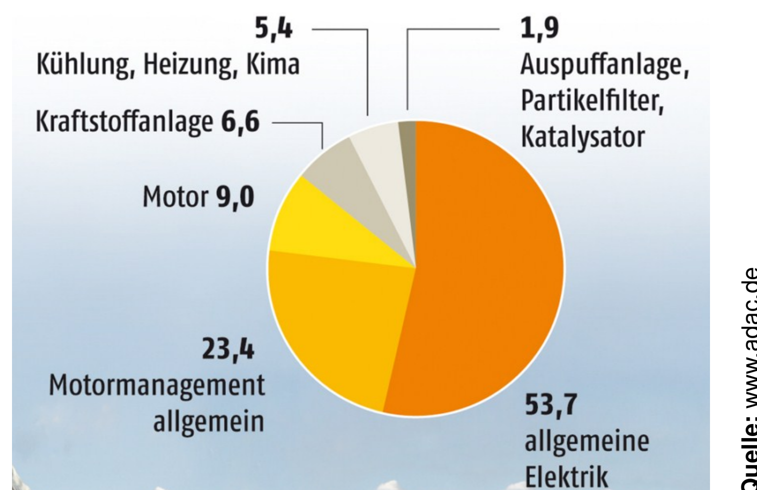


Abbildung 1.1.: ADAC Pannenstatistik 2010

Die schiere Abhängigkeit elektrischer Systeme von der Starterbatterie als Quelle lassen sie so als schwächstes Glied im Bordnetz erscheinen. Gerade bei sicherheitsrelevanten Aspekten von X-by-Wire[1] wird das Fahrzeughandling immer abhängiger von einer zuverlässig funktionierenden Elektronik. Trotz vielversprechender Kostenvorteile bei der Fertigung lassen sich angesichts dieser Unsicherheit mit der X-by-Wire Technik langfristig keine Erfolge

[1] Begriff der elektrischen Kopplung von Kfz-Funktionseinheiten wie Bremsen,Kuppeln, Lenken usw. Ersatz mechan. Steuerung durch elektrische/opt. Signale.

auf dem Markt erzielen [1].
Die Lösung des Batterieproblems durch Weiterentwicklung neuer Energiespeicher wird sich in naher Zukunft nicht ergeben. Zu groß ist die Vielzahl von Abhängigkeiten gegenüber Witterung, Fahrverhalten, Teilkomponenten und Alter. Als folglich kritisches System wird die Starterbatterie immer bezüglich ihrer Grundfunktionen überwacht werden müssen. Lebensdauer und Effektivität lassen sich durch entsprechendes Management entscheidend verbessern. Zudem kann ein voraussichtlicher Fehlerfall vorzeitig erkannt und möglicherweise rechtzeitig behoben werden. Die Anforderungen an Batteriesensorik sind dem Markt entsprechend hoch und auch in Hinblick auf die Elektromobilität sehr gefragt. Gerade die Minimierung der Kosten, in Verbindung zu hohen Stückzahlen und zum Erhalt rentabler Verkaufserlöse, treibt die Entwicklung in diesem Bereich immer weiter an. Die vorliegende Arbeit soll hier durch die Untersuchung neuer drahtloser Technologien im Bereich der Batteriesensorik zur Kosteneffizienz einen Beitrag leisten.

1.1. Starterbatterien

Die Starterbatterie ist ein essenziell wichtiger Teil für den reibungslosen Betrieb von Verbrennungsmotoren aller Art. Zum Starten von Verbrennungsmotoren muss das Kraftstoff-Luftgemisch zunächst zum optimalen Zündzeitpunkt vorverdichtet werden. Das hierfür benötigte Antreiben der Kurbelwelle wird heutzutage elektrisch durch den sogenannten Starter bzw. Anlasser erreicht. Dieser ist meistens ein Elektromotor, der beim Starten kurzfristig durch einen Magnetschalter über ein Zahnradtrieb mit dem Verbrennungsmotor verbunden wird. Ist der Motor gestartet, wird die Batterie durch die überschüssige Energie des Motors über einen Generator wieder geladen. Sie ist ein Akkumulator.
Neben dem Startvorgang dient die Starterbatterie außerdem dem Puffern besonderer Lastspitzen und versorgt bei Motorstillstand permanent elektrische Verbraucher wie z.B. Bordcomputer, Radio, Licht usw. Abb. 1.2 zeigt den Zusammenhang der Starterbatterie und wichtiger Teilkomponenten.

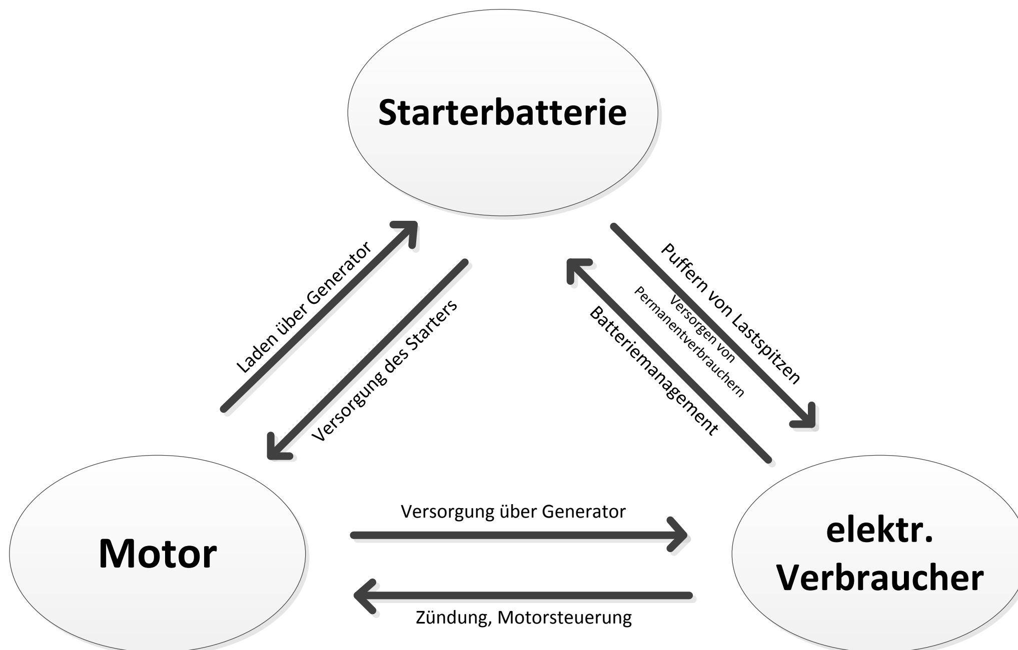
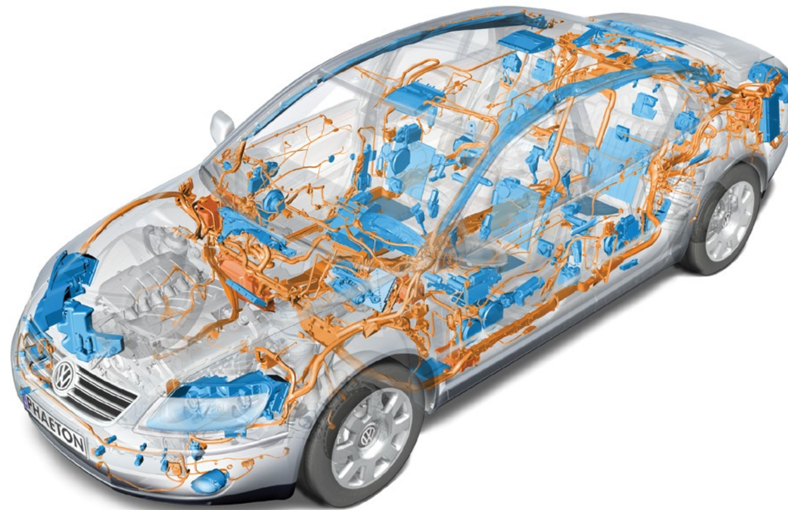


Abbildung 1.2.: Allgemeine Zusammenhänge im Kfz-Bordnetz

1.1.1. Aufgaben und Anforderungen im Kfz-Bordnetz

Das Kfz-Bordnetz bezeichnet die Gesamtheit aller verbauten, elektrischen Verbraucher, sowie ihre versorgenden Komponenten wie Generator und Batterie. Je nach Fahrzeuggröße und Ausstattung ergibt sich eine beachtliche Anzahl von Komponenten, die in ihrer Gesamtheit Kabelverbindungen von bis zu mehreren Kilometern erfordern. Abb. 1.3 zeigt beispielhaft das Bordnetz des VW Phaeton. Aus der Abbildung wird deutlich mit welcher Komplexität sich die Verkabelung über die gesamte Karosserie erstreckt.



Quelle: www.bosch-service.de

Abbildung 1.3.: Beispiel Bordnetz: VW Phaeton

Die Hauptaufgabe der Starterbatterie ist neben der Versorgung des Starters die Versorgung der im Auto befindlichen elektrischen Verbraucher, sobald der Generator nicht mehr genügend Leistung zur Verfügung stellen kann (Leerlauf oder Motorstillstand). Ebenso schützt ihre hohe Kapazität das Bordnetz vor eventuell auftretenden Spannungsspitzen, durch die die Elektronik gefährdet würde. Es lassen sich hieraus 3 Betriebszustände ableiten:

Hochstromentladung Steht der Motor still, fordert der Starter zum Erzeugen des benötigten Drehmoments kurzzeitig einen hohen Strom von mehreren hundert Ampere. Der Zustand des Motors und die Umgebungstemperatur beeinflussen den Startvorgang zusätzlich: Bei niedrigen Temperaturen sind sämtliche Schmiermittel sehr zähflüssig und erhöhen zusätzlich den Kraftaufwand für den Starter.

Lade-/Entladebetrieb Liefert der Generator bei laufendem Motor genügend Energie für das Bordnetz, wird die Batterie kontinuierlich geladen. Ein entsprechender Laderegler stellt dazu die benötigten Ladespannungen bereit. Die hierzu aufgebrachte Leistung ist jedoch stark von der Motordrehzahl abhängig. Die im Generator induzierte Spannung erzeugt erst bei etwa $6000 min^{-1}$ Umdrehungen die entsprechende Nennleistung[2]. Bei niedrigen Drehzahlen (Leerlauf bei <$1000 min^{-1}$) wird nur noch ein geringer Teil der Nennleistung erreicht. Gerade bei Fahrzyklen mit hohen Leerlaufanteilen, etwa im dichten Stadtverkehr oder häufigen Kurzstrecken, kann der Generator die benötigte Leistung nicht bereitstellen und die Batterie wird entladen.

[2]die typische Nennleistung von Lichtmaschinen beträgt bei Kleinwagen etwa 1kW bis 3 kW in der Oberklasse

Ruhe Ist der Motor ausgeschaltet wird die Batterie i.d.R. nur sehr schwach belastet. Permanentverbraucher wie Radio, Bordcomputer, Steuergerät usw. benötigen hier über längere Standzeiten kontinuierlich nicht mehr als 3-10mA Ruhestrom.

Optimalerweise sollte die Batterie nicht kontinuierlich während der Fahrt entladen werden, sodass das Starten des Motors auch bei starker Nutzung von Komfortverbrauchern stets gewährleistet ist. Bei Fahrzeugen der Oberklasse wird deshalb häufig eine 2. Servicebatterie verbaut, die für starke, zyklische Belastungen ausgelegt ist.

Neben behandelten, technischen Gegebenheiten spielt ebenso die Wahl der Starterbatterie eine wichtige, ökonomische Rolle. Bei praktizierter Massenherstellung von Fahrzeugen rücken verstärkt Produktions-, Anschaffungs- und Wartungskosten in den Vordergrund, die unter Abwägung von eingesetzten Batterien erfüllt werden müssen. Die Kosten hängen dabei von der Art der gewählten Batterie ab, die wiederum unterschiedliche Merkmale und Eigenschaften aufweist. Die wichtigsten Merkmale verschiedener Batterietypen sind dabei in Tabelle 1.1 aufgelistet. Im Vergleich besonders auffälliger Werte sind dabei entsprechend gekennzeichnet.
Als Starterbatterien scheiden hierbei Nickel-Cadmium und Nickel-Metallhydrid aus. Trotz der guten Zyklenzahl und annehmbarem Ladeverhalten eignen sich diese Typen durch ihren hohen Innenwiderstand und Wartungsbedarf nicht für den Einsatz als Starterbatterie. Die zudem geringe Zellspannung von 1,2 V erfordern zum Erreichen höherer Spannungen entsprechend viele Zellen, was die Wartung und Kosten im Vergleich zur Blei-Säure Batterie in die Höhe treibt. Durch die relativ gute Energiedichte und der im Vergleich zu Nickel-Cadmium viel geringeren Toxizität finden NiMH-Akkumulatoren eher Anwendung in Kleinstgeräten und dem Modellbau. Ihre guten Hochstromeigenschaften unter Vollast lassen auch den Einsatz in Hybridfahrzeugen zu, zuletzt serienmäßig z.B. im Toyota Prius.
Weit höheres Potenzial hat die Lithium-Ionen Technik: Wie in Tabelle 1.1 ersichtlich, bietet sie die meisten Vorteile im Vergleich zu herkömmlichen Techniken. Der größte Vorteil liegt dabei in der höheren Energiedichte, ihrer Wartungsfreiheit und im Schnelladerverhalten. Im Vergleich zum Blei-Säure-Akkumulator ist eine deutliche Steigerung in nahezu allen Eigenschaften gegeben. Die Lithium-Ionen-Technik ist somit die derzeit erfolgversprechendste Akku-Technologie. Mit neuen Werkstoffen und Nanotechnologie sind heutzutage schon erhebliche Verbesserungen zu den genannten Eigenschaften möglich. In Bezug zur Starterbatterie stehen schon Lithium-Ion-Varianten zum Verkauf. Der Automobilhersteller Porsche vertrieb Anfang des Jahres 2010 die ersten Starterbatterien[3] mit Lithium-Ion-Technik, die aufgrund von Gewichtsersparnissen neben Langlebigkeit höhere Reichweiten und bessere Fahreigenschaften versprechen.
Die Lithium Ionen Technik birgt jedoch auch Nachteile: Seltene Werkstoffe, sowie hohe Empfindlichkeit gegenüber Über- und Tiefentladung schlagen sich erheblich auf die Kosten

[3] Porsche Leichbau-Starterbatterie, Nennkapazität: 18Ah, Gewicht: 6kg, www.Porsche.de

nieder. Derzeit in Elektrofahrzeugen verbaute Lithium-Ion-Akkupakete nehmen häufig den Großteil der Fahrzeugkosten ein. So liegt beispielsweise der Anteil der Batterie im Karabag 500E[4] mit 10.000 - 20.000 € bei etwa 30%-50%.
Zuletzt bleibt noch der herkömmliche Blei-Säure Akkumulator als Starterbatterie übrig. Die schon seit Jahrzehnten im Automobil bewährte Technik bietet sich vor allem wegen den geringen Herstellungskosten und ihrer Robustheit an. Die im Vergleich zu anderen Technologien ersichtlichen Nachteile aus Tabelle 1.1 lassen sich bezogen auf die Anforderungen schlichtweg ertragen: Die Hauptaufgabe des Starters fordert im Allgemeinen keine längeren Ladezyklen. Kurz nach dem Start entlastet ihn i.d.R. der Generator. Unter kurzen Belastungen wirkt sich seine relativ hohe Robustheit gegenüber unsachgemäßer Ladung positiv auf die Lebenszeiten aus. Die einfache Technik wird aufgrund ihrer Verbreitung und mangelnder Alternativen noch lange auf dem Markt bestehen. In heutigen Fahrzeugen wird die Starterbatterie häufig vernachlässigt und trägt zu Pannen bei, wie anfangs erwähnt, sogar zu einem erheblichen Teil. Schon durch eine einfache Überwachung und Sicherstellung ausreichender Aufladung der Batterie lässt sich eine z.T. erhebliche Erhöhung der Lebensdauer erreichen [15], weshalb die Blei-Säure-Starterbatterie weiter in dieser Arbeit behandelt wird.

[4] basierend auf Fiat 500, 30kW/41PS,22kWh Kapazität, www.karabag.de

	NiCd	NiMH	Blei-Säure	Lithium-Ion Kobalt	Lithium-Ion Mangan	Lithium-Ion Phosphat
spezifische Energie Dichte (Wh/kg)	45-80	60-120	30-50 -	150-190 +	100-135	90-120
Innen-Widerstand in $m\Omega$	100-200 6V pack	200-300 6V pack -	<100 12V pack +	150-300 7,2V pack	25-75 +	25-50 + +
Anzahl Zyklen (für 80% der Nennkapazität)	1000	300-500	200-300 -	500-1000	500-1000	1000-2000 + +
Schnellladezeit	1h typisch +	2-4h	8-16h -	2-4h	<1h +	<1h +
Überladetoleranz	moderat	gering	hoch +	sehr gering –		
Selbstentladung pro Monat (bei Raumtemp.)	20,00%	30,00% -	5,00% +	<10%		
Zellenspannung (nominal)	1,2V	1,2V	2V	3,6V	3,8V	3,3V
Ladestrom[5] max C-Wert für bestes Resultat	20C 1C	5C 0,5C	5C 0,2C	>3C <1C	>30C <10C	>30C <10C
Betriebstemp. (nur Entladung, in °C)	-40 bis 60	-20 bis 60	-20 bis 60	-20 bis 60		
Wartungsbedarf	30-60 Tage	60-90 Tage	3-6 Monate	nicht nötig ++		
Sicherheitsbedarf	thermisch stabil	thermisch stabil, Sicherung üblich +		Überwachungs-Schaltung benötigt -		
Kommerziell verwendet seit	1950	1990	späten 1800er	1991	1996	2006
Toxizität	sehr hoch	gering	sehr hoch	gering		

Quelle: www.battery-university.com

Tabelle 1.1.: Eigenschaften häufig genutzter Akkumulatoren

[5] Lade- und Entladeströme sind meist auf die Nennkapazität bezogen und in C ausgedrückt. 1 C bezeichnet die Ent- oder Aufnahme der auf die Nennkapazität bezogenen Stromstärke. Zu hohe Ladeströme wirken sich negativ auf die Zyklenlebensdauer aus. Bessere Ergebnisse werden bei längeren Ladevorgängen mit geringer Stromstärke erzielt.

1.1.2. Aufbau und Eigenschaften von Blei-Batterien

Die Starterbatterie ist seit jeher eine Bleibatterie: Sie besteht prinzipiell aus zwei Elektroden, die über einen Elektrolyten miteinander Ionen austauschen. Die gespeicherte chemische Energie wird hierbei durch eine elektrochemische Redoxreaktion in elektrische Energie umgewandelt. Beim Laden verläuft die Reaktion entsprechend gegenläufig ab.
Bei der Bleibatterie besteht die negative Elektrode aus Blei und die positive Elektrode aus Bleidioxid. Als Elektrolyt dient mit Wasser verdünnte Schwefelsäure (ca. 38%). Abb. 1.4 zeigt den prinzipiellen Aufbau mit chemischen Hauptreaktionen für Lade- und Entladevorgang.

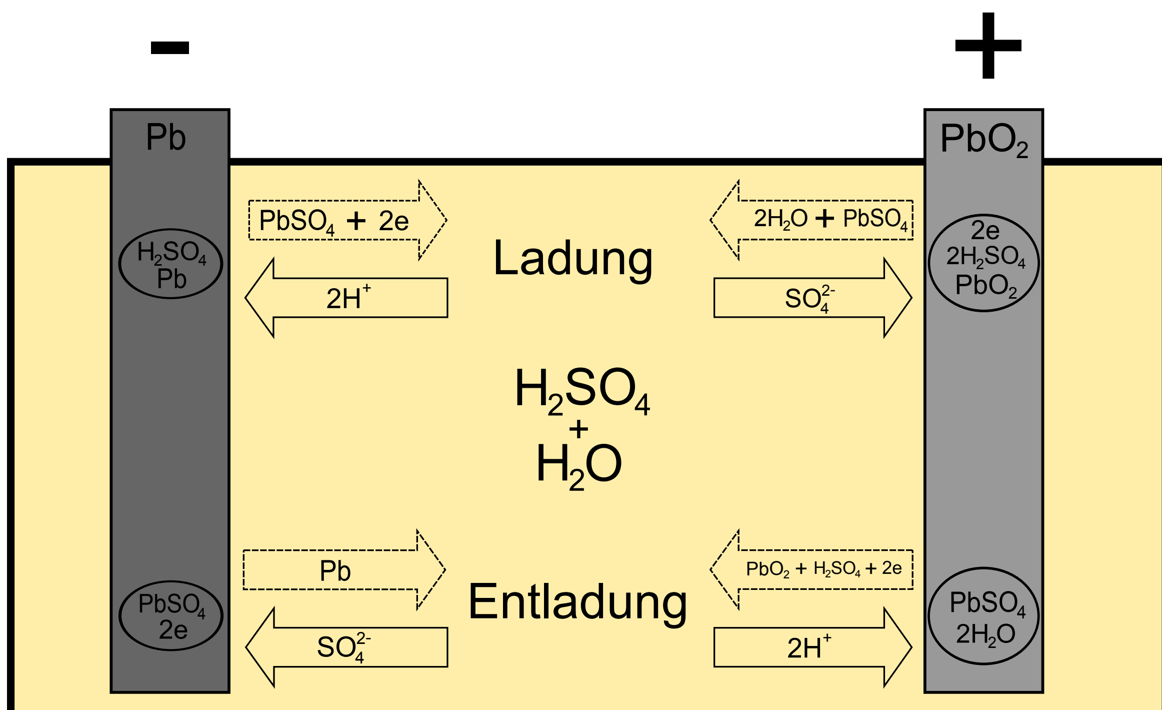
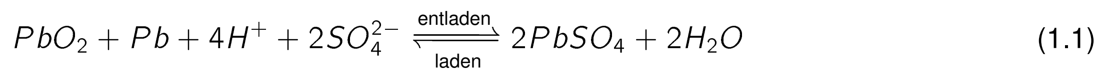


Abbildung 1.4.: Das Funktionsprinzip des Bleiakkumulators

Während der Hauptreaktion wandert beim Laden des Akku das negativ geladene Sulfation SO_4^{2-} aus dem Elektrolyt zur positiven Elektrode ab. Es wird dabei unter Bildung von Schwefelsäure Wasser verbraucht. Zudem werden 2 neue Ladungen e frei, die über den äußeren Ladestromkreis zur negativen Elektrode gelangen. Die Elektrode wird entsprechend positiver geladen. An der negativen Elektrode hingegen bildet sich mit dem abgewanderten Wasserstoffion H^+ ebenfalls Schwefelsäure. Es werden dabei 2 Ladungen aufgenommen. Die Elektrode wird negativer geladen. Bei Belastung des Akku laufen die entsprechenden Reaktionen gegenläufig ab. Die folgende Reaktionsgleichung fasst die chemischen Prozes-

se zusammen:



$$PbO_2 + Pb + 4H^+ + 2SO_4^{2-} \underset{\text{laden}}{\overset{\text{entladen}}{\rightleftharpoons}} 2PbSO_4 + 2H_2O \tag{1.1}$$

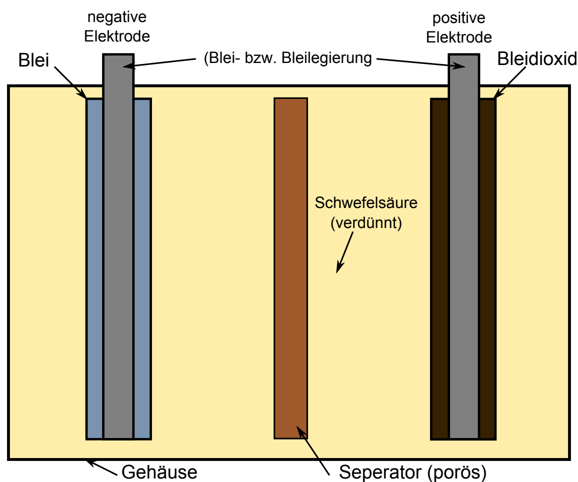
Wie bei vielen Redoxreaktionen spielt auch hier die Temperatur eine wichtige Rolle. Bei hohen Temperaturen werden chemische Reaktionen i.d.R. begünstigt, wobei ihr zeitlicher Ablauf z.T. erheblich verkürzt wird. Ebenso verlangsamt ein entsprechender Temperatursturz die chemischen Reaktionen. Im Falle des Bleiakkus ist stets eine mittlere Temperatur für einen stabilen und schonenden Betrieb vorteilhaft. Temperaturen außerhalb des Bereichs von -20 - +60°C wirken sich in jedem Fall negativ aus, ob akut bei sehr niedrigen Temperaturen oder unter Hitze langfristig auf die Lebensdauer, bei denen chemische Nebenreaktionen die Kapazität verringern. U.a. gehören dazu:

Gasung Wird der Akku bis etwa 2,5V Zellenspannung geladen tritt eine Gasbildung ein (der Akku "kocht"). An der positiven Elektrode entsteht dabei Sauerstoff und an der negativen Elektrode die doppelte Menge Wasserstoff. Es wird Wasser zersetzt und es bildet sich flammbares Knallgas. Bei häufiger Gasung sollte Wasser nachgefüllt werden.

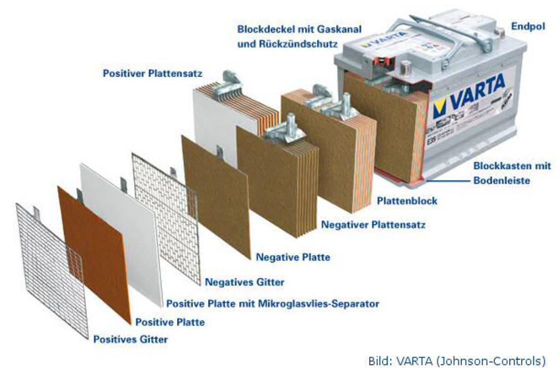
Gitterkorrosion Die positive Elektrode wird aufgrund von Korrosion langsam zersetzt. Dies trägt zur Selbstentladung bei.

Sauerstoffreduktion ist die Reduzierung von Sauerstoffs im Elektrolyten bei längeren Ruhephasen.

Im Laufe der Entwicklung von Bleibatterien lag es stets nahe, die Nebenreaktionen konstruktionsbedingt zu minimieren. Etwa wird in speziellen Rekombinationsstopfen mit Katalysatoren der Verbrauch von Wasser verringert. Abb. 1.5 zeigt den Aufbau einer modernen Blei-Starterbatterie.



(a) Aufbau einer Zelle



(b) Aufbau einer Varta Starterbatterie

Abbildung 1.5.: Aufbau einer Starterbatterie

Die Elektroden werden als Schutz vor Kurzschlüssen meistens durch einen porösen Separator getrennt. Je nach Elektrodenfläche ergibt sich die jeweilige Zellkapazität. Dabei bilden mehrere Elektrodenpaare parallel eine Zelle. Mehrere Zellen in Reihe geschaltet, ergeben höhere Spannungen. 6 Zellen in Reihe mit jeweils einer 2 V Zellenspannung, ergeben die 12 V einer Starterbatterie.

Über konstruktionsbedingte Änderungen lassen sich Effizienz und Kosten weiter verbessern. So ist eine Auslegung der Elektroden als beschichtete Gitter durch eine folglich erhöhte Oberfläche effizienter und kostengünstiger in der Herstellung als eine massive Platte[6]. Des weiteren lässt sich noch die Leitfähigkeit der Materialien verbessern. Verschiedene Bleilegierungen der Gitter lassen so weniger Leistung am Innenwiderstand verloren gehen. Auch wirken sich z.B. das Festsetzen des Elektrolyts in Flies oder Gel positiv auf Sicherheit und Wartung aus. Eine detailliertere Darstellung hierzu findet sich in einschlägiger Literatur [11],[23],[24].

[6]sogenannte Panzerplatten werden für “high duty”Akkus mit hohen zyklischen Belastungen eingesetzt

1.2. Batteriesensorik und –management

Die in vorangegangenen Abschnitten 1.1.1 und 1.1.2 dargestellten Zusammenhänge des Kfz-Bordnetzes und der verwendeten Technik von Starterbatterien spielen im Automotive Bereich mit fortschreitender Entwicklung eine zusehends wichtigere Rolle. In Hinblick auf Zuverlässigkeit und fortschreitende Sicherheitsansprüche durch Einsatz neuer Technologien im Elektronikbereich, ist ein intelligentes Batteriemanagement unabdingbar.
Das Batteriemanagement stellt in intelligenten Systemen die Funktionen der Starterbatterie als Energiepuffer für Spitzenlasten sicher. Wichtige Grundfunktionen wie das Versorgen vom Starter und Permanentverbrauchern bei Motorstillstand gehören ebenfalls dazu. Durch Überwachung von Einflussgrößen wie Strom, Spannung und Temperatur kann so auf eventuelle Engpässe frühzeitig reagiert und die Batterie schonend betrieben werden, um ihre Lebensdauer maximal auszunutzen. Batterie- und Energiemanagement des Fahrzeugs arbeiten dabei sehr eng zusammen. Abb.1.6 zeigt den Zusammenhang in der Übersicht.
Batterie- und Energiemanagement sind dabei grundsätzlich zu unterscheiden. Batteriemanagement überwacht die Batterie bezüglich ihrer Parameter. Aus gewonnenen Messdaten wird der Zustand bestimmt. Hierzu zählen u.a.:

State of Charge (SoC) bezeichnet den Ladezustand der Batterie in Bezug zur Nennkapazität.

State of Health (SoH) ist der Alterungszustand der Batterie.

Crank Capability (CC) gibt die Fähigkeit an, den Motor zu starten.

Charge Acceptance (CA) bezeichnet die Fähigkeit der Batterie Ladung aufzunehmen.

Auf Basis von SoC und SoH wird geprüft, ob noch genügend Leistung zum Erfüllen bestimmter Funktionen verfügbar ist. Unter die State of Function (SoF) fallen dabei auch CC und CA.

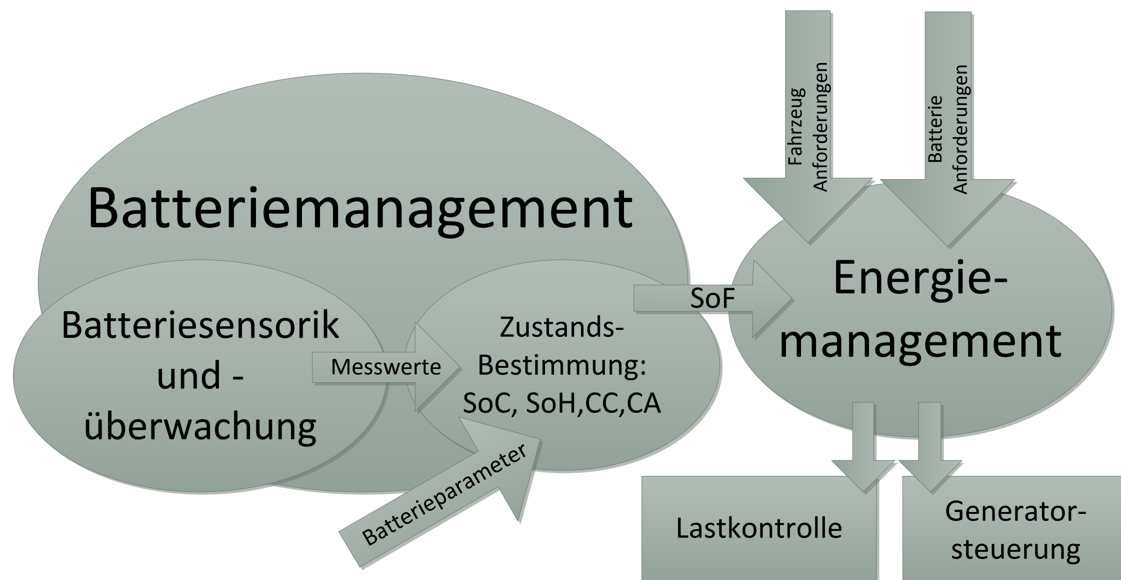


Abbildung 1.6.: Batterie- und Energiemanagement im Zusammenhang

Der Zustand der Batterie wird dem Energiemanagement mitgeteilt, das entsprechende Handlungen vornimmt: Durch das Lastenmanagement können momentan unwichtige Funktionen des Fahrzeugs deaktiviert werden, falls die Generatorleistung nicht für das Laden der schwachen Batterie ausreicht. Das Energiemanagement hat auch Zugriff auf die Generaltorsteuerung: Abhängig vom Batteriezustand kann die Batterie so vor Überladung geschützt werden. Ebenso lassen sich Ladespannungen der Batterietemperatur anpassen. Ein schonender und effizienter Betrieb ist somit möglich.

1.2.1. Ladezustandsbestimmung der Batterie

Der Ladezustand einer Batterie ist nicht trivial. Genaue Aussagen lassen sich hier nur bei der Berücksichtigung aller Einflussgrößen treffen. Unterschiedliche Belastungszustände stellen dabei zusätzlich Anspruch an die Sensorik und Messwerterfassung. Aus Abschnitt 1.1.1 gingen bereits zu Grunde liegende Anforderungen an die Starterbatterie hervor. Neben Standzeiten des Kfzs, bei denen nur sehr wenig Energie an die Ruhestromverbraucher verloren geht, sind ebenfalls hoch dynamische Messwertverläufe im Startbetrieb und während der Fahrt zu erwarten. Beide Situationen liefern hierbei unterschiedliche Aussagen über den Ladezustand:

Ruhephase Nach Lade- oder Entladevorgängen stellt sich ein chemisches Spannungsgleichgewicht in jeder Zelle ein. Dieser Vorgang kann bis zu einigen Tagen dauern. Dennoch aussagekräftig ist hier der Zeitpunkt einer hinreichend stabilen Spannung nach etwa 3-4 Stunden bei der sich Fehler entsprechend kalibrieren lassen. Zu berücksichtigen sind dabei unbedingt Ruhestromverbraucher, die ebenfalls einen geringen Spannungsabfall verursachen. Die Ruhespannung der Batterie korreliert direkt mit ihrer Säuredichte, die eine genaue Einschätzung des Ladezustands zulässt (vgl. Abschnitt 1.4). Die Tabelle 1.2 zeigt den ungefähren Ladezustand zu der entsprechenden Polspannungen und Säuredichte. Die Einteilung ist nach BCI[7] angegeben.

Betriebsphase Der Zustand der belasteten Batterie ist nur durch Ladungsbilanzierung bestimmbar. Hierbei werden stets zu- und abfließende Ladungen überwacht. Mit Bezug zur jeweiligen Gesamtkapazität ist eine genaue Aussage über verbleibende Ladungen möglich. Dieses Vorgehen erfordert eine genaue Strommessung. Wichtig ist hierbei eine genaue zeitliche Auflösung, da über den Strom integriert werden muss. Aufgrund von Säureschichtung[8] innerhalb der Bleibatterie, starker Gasung oder anderen altersbedingten Effekten, kann es hier zu Abweichungen von der tatsächlichen Kapazität kommen, weswegen ein Abgleich mit der Ruhespannungsmessung idealerweise erfolgen sollte. Störfälle sind so ebenfalls leicht erkennbar.

Ruhespannung in Volt	**Säuredichte in g/cm3**	**Ladezustand in %**
12,65	1,28	100
12,45	1,24	75
12,24	1,20	50
12,06	1,17	25
11,89	1,10	0

Tabelle 1.2.: Ladezustand über die Ruhespannung

1.2.2. Einsatz von Batteriesensorik

Das Problem der Ladezustandsbestimmung ist schon einige Zeit der Automobilindustrie bekannt. Fast jeder Automobilhersteller rüstet derzeit Fahrzeuge der Mittel- und Oberklasse

[7] Battery Council International: Vereinigung zur Angleichung industrieller Standards, weltweit mehr als 190 Mitglieder

[8] Die an den Elektroden entstandene Säure ist schwerer als Wasser und sinkt durch Gravitation nach unten ab. Es entsteht so eine unterschiedliche Säurekonzentration im Elektrolyten. Beim starken Entladen setzt sich nun weniger Säure an den Platten um.

mit einer entsprechenden Batteriesensorik aus. So entwickelte im vergangenen Jahrzehnt z.B. BMW in Kooperation mit seinen Partnern, Hella und Autokabel, den intelligenten Batteriesensor IBS. Die Bauform wurde dabei an die nach DIN 72311 normierte Polniesche der Starterbatterien angepasst, welches den Betrieb in allen Kfz sicherstellen soll (Abb. 1.7).

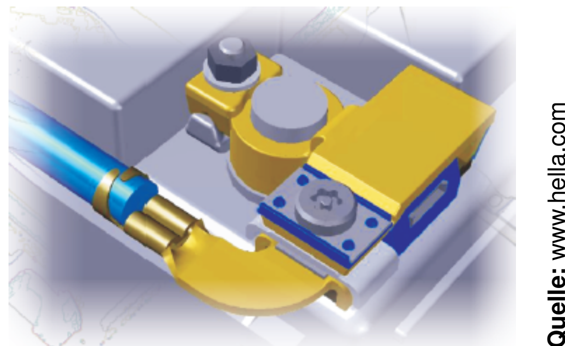
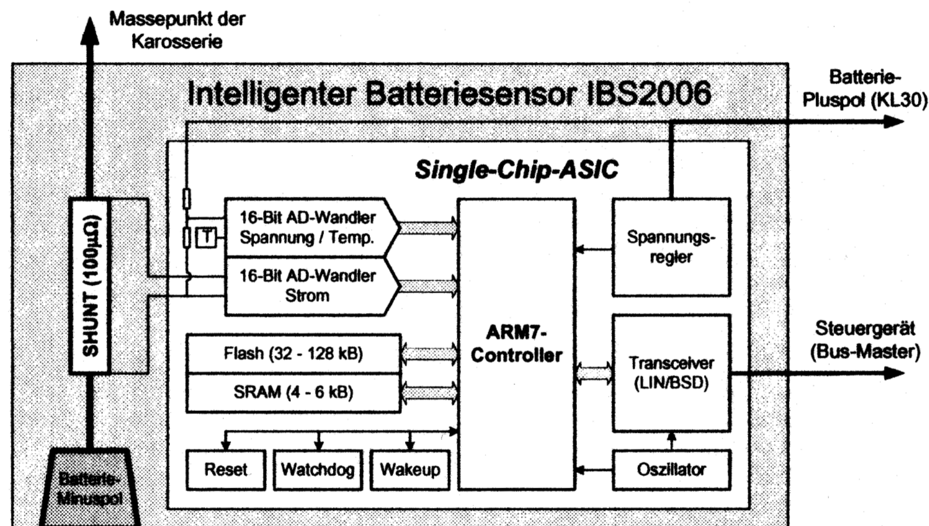


Abbildung 1.7.: BMW, Hella, Autokabel: intelligenter Batteriesensor

Bei dem IBS2006 werden Strom, Spannung und Temperatur[9] der Starterbatterie erfasst. Die Strommessung geschieht dabei sehr präzise über einen Shunt im 200$\mu\Omega$-Bereich und umfasst 3 Messbereiche von $\pm 1A$ (Ruhe) über $\pm 200A$ (Lade-/Entladebetrieb) bis $+1500A$. Das Spannungssignal am Shunt erreicht hier beispielsweise bei einem Ruhestrom von 30 mA einen sehr geringen Wert von $6\mu V$. Dies liegt gerade 3 Zehnerpotenzen über der Thermospannung und erfordert eine entsprechend hohe Messauflösung. Strom und Spannung werden synchron mit einer Abtastrate von 1 - 8 kHz aufgenommen. Der Sensor ist dabei über den LIN Bus mit dem Energiemanagement des Kfz verbunden. Um Zeit auf dem Bus einzusparen, benötigt der IBS2006 einen leistungsstarken Mikrocontroller für die Vorverarbeitung. Alles in allem wird hier in Hinblick auf den serienmäßigen Einsatz ein beachtliches Maß an Hardware eingesetzt, um den Anforderungen zu genügen. Abb. 1.8 zeigt das Blockschaltbild des IBS2006 mit allen Komponenten.

[9] Die thermische Ankopplung erfolgt über den massiven Batteriepol.



Quelle: [17] Seite 53ff

Abbildung 1.8.: Blockschaltbild des IBS2006

1.3. Einzelzellüberwachung

Bisherige Realisierungen von Batteriemanagement Systemen setzen auf möglichst einfache Batteriesensoren, die zentral an der Polklemme Messwerte erfassen. Dieses Vorgehen hat jedoch einen erheblichen Nachteil: Mögliche Unterschiede in den einzelnen Zellen können bei Betrachtung der Gesamtspannung nicht erfasst werden. Es steht quasi nur der Mittelwert zur Verfügung, der keine Aussage über den Zustand einzelner Zellen zulässt. Die Tatsache schwacher Teilzellen wird bei Betrachtung der Gesamtspannung durch Zellen besseren Zustands verdeckt. Treten weiter hohe Entladeströme kommt es zum Ausfall der Batterie ohne das die überwachende Sensorik frühzeitig Maßnahmen ergreift (Abb. 1.9).

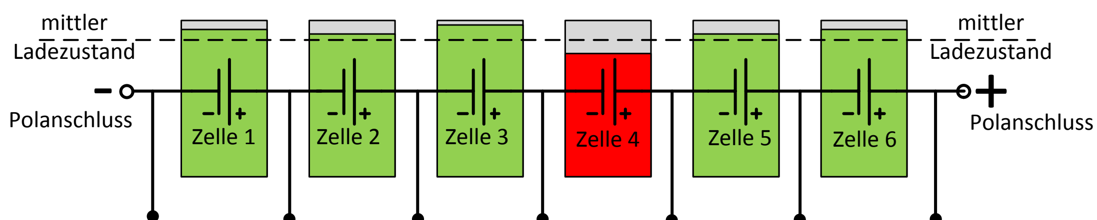


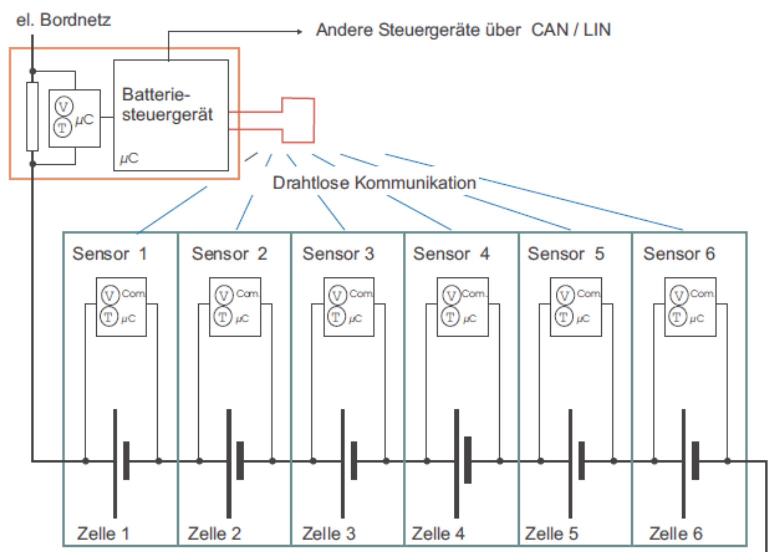
Abbildung 1.9.: Ladezustand einzelner Zellen

Die Überwachung jeder einzelnen Zelle lässt weitaus genauere Vorhersagen zu. Unterschiedliche Zellen würden problemlos erkannt. Auch ein Angleichen der Zellladungen (Cell

Balancing) durch etwa gezielte Entnahme von Ladungen wäre möglich. Ein Konzept mit drahtlosen Zellsensoren soll im Verlauf dieses Abschnittes weiter behandelt werden.

1.3.1. Konzept von drahtlosen Sensornetzen

Dem Überwachen einzelner Zellspannungen stehen ganz klar der höhere Aufwand und die damit verbundenen Kosten entgegen. Zusätzliche Elektronik, erforderliche Verkabelung, Steckkontakte und der erhöhte Aufwand in einer zentralen Messwerterfassung machen das Thema des serienmäßigen Einsatz bei Starterbatterien gerade auf dem Automobilmarkt eher unattraktiv. Bisher lohnte die Einzelzellüberwachung nur bei Hochleistungs-Traktionsbatterien auf Lithium-Ion Basis in Elektrofahrzeugen. Besonders wegen Sicherheitsaspekten kann hier nicht darauf verzichtet werden. Die drahtlose Sensorik gilt daher als ein guter Ansatz, um ein effektives Maß an Integrierbarkeit zu erreichen. Abb. 1.10 stellt dafür das Konzept dar.



Quelle: Förderantrag 'FHprofUnt 2010' [14]

Abbildung 1.10.: Konzept drahtloser Batteriesensorik

An den Anschlüssen jeder Zelle wird dabei ein drahtloser Sensor angebracht. Die Sensoren messen die Zellspannung sowie die Temperatur und übertragen die Information an eine zentrale Empfangseinheit. Zentral mit dem Datenempfang wird der hinzu- oder abfließende Strom gemessen, der bei einer Reihenschaltung jeder Zelle entspricht. Empfangene Zelldaten können nun ausgewertet und an das Energiemanagement des Fahrzeugs weitergeleitet werden.

1.3.2. Realisierung des Sensors

Im Rahmen des Projekt BATSEN an der Hochschule für Angewandte Wissenschaften Hamburg wurden in Vorarbeiten bereits experimentelle Aufbauten entwickelt [10][6]. Alle Realisierungen arbeiten im freien ISM (**I**ndustrial, **S**ientific and **M**edical) Band. Teilweise wurden dabei Aufbauten mit RFID realisiert, die einen sparsamen Betrieb ermöglichen und die Batterie nicht zusätzlich belasten. Der Aufbau der Variante in Abb. 1.11 sendet ohne Downlink im Frequenzbereich um 433 MHz und wird über die Zellspannung versorgt.

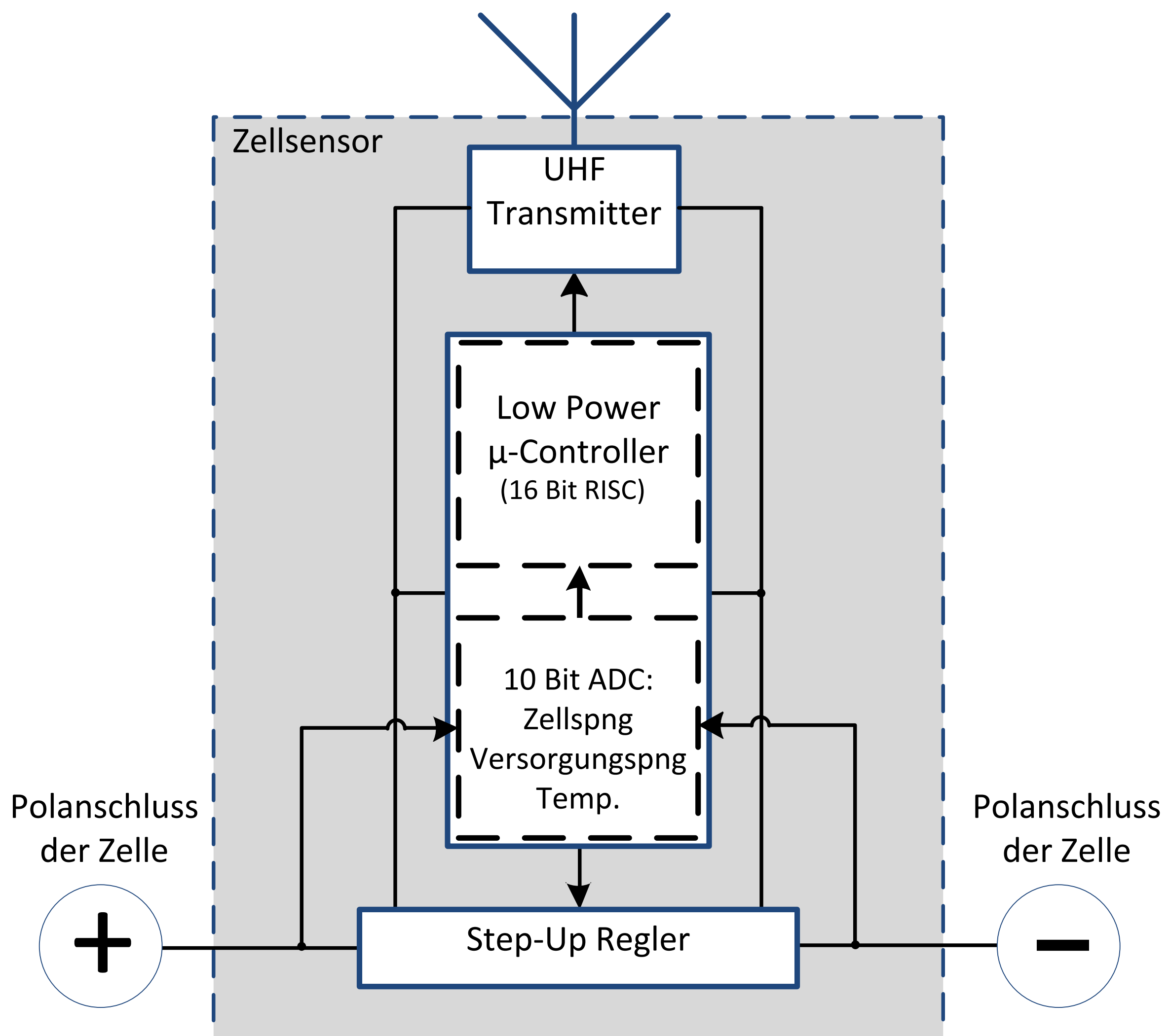


Abbildung 1.11.: Aufbau des Zellsensors, modifiziert nach [14] Abb. 2.2

Beim Hardwaredesign wurde auf Bauteile mit geringem Stromverbrauch im SMD Format geachtet. Die Sensorplatine wird direkt über entsprechende Kontaktflächen (5) mit der Zelle

verbunden. Als Mikrocontroller zur Steuerung der Peripherie und Auswertung der Messdaten dient ein Low Power Mikrocontroller der MSP430xx2 Familie von Texas Instruments. Enthalten ist bereits ein 10 Bit ADC zur Messwertaufnahme. Für die Kommunikation mit der Empfangseinheit werden die Sendedaten direkt als OOK Basisbandsignal der Sende-PLL TDA5100A von Infineon zur Umsetzung in den 434MHz Bereich übergeben. Gespeist werden beide Bauelemente von einem Step-Up-Konverter L6920 der Firma STMicroelectronics. Der Step-Up-Konverter passt die Zellspannung, bei Bleibatterien typisch 1,7V...2,4V, der nötigen Betriebsspannung von 3,3V an. Möglich ist hier sogar der Betrieb bei niedrigen Eingangspannungen von bis zu 0,8V, da dies die minimal nötige Spannung zum Starten des Bausteins darstellt. In Abb. 1.12 ist der fertige Aufbau zu sehen.

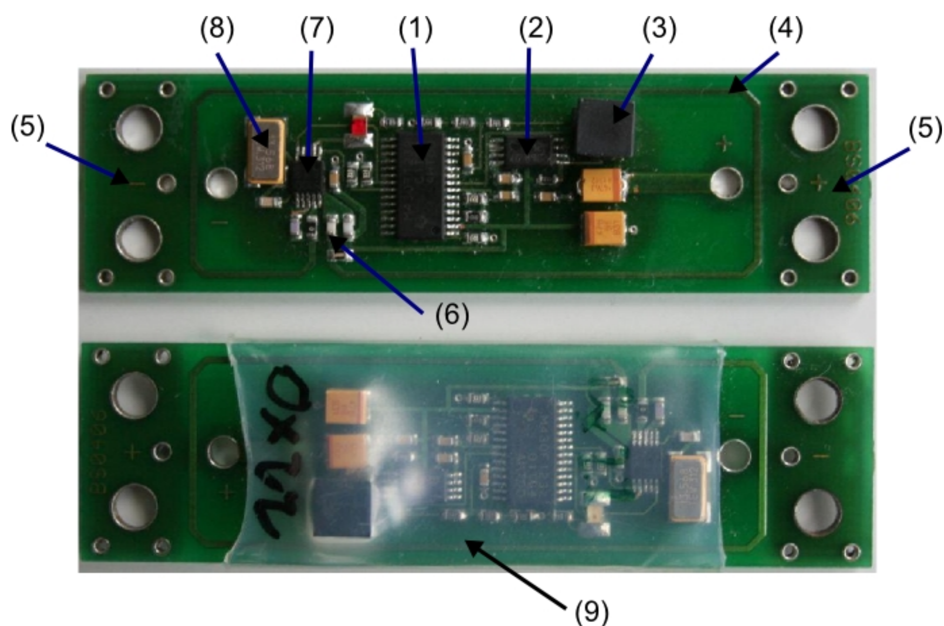


Abbildung 1.12.: Realisierung des Zellensensors

Links und rechts vom Mikrocontroller (1), in der Mitte von Abb. 1.12, sind Step-Up-Konverter (2) und Sende-PLL (7) angeordnet. Bei diesem Sensor ist trotz des gesetzten Kostenziels ein Quarz (8) für den Transmitter unumgänglich. Nur bei dem MSP430x1232 konnte auf einen zusätzlichen Quarz verzichtet werden. Sein interner DCO stellt einen Takt von 2 MHz zur Verfügung. Der DCO ist zwar stark temperaturabhängig, jedoch fällt ein abweichender Takt bei der bisherigen Anwendung nicht sonderlich ins Gewicht. Der Wirkungsgrad des Step-Up-Konverters ist stark von der Güte seiner Ladespule (3) abhängig. Hier wurde deshalb eine große Spule mit Ferritkern gewählt. Der Transmitter ist über ein Anpassnetzwerk (6) mit einer kurzen Schleifenantenne (4) verbunden. Dabei ist die Antenne kostengünstig als Leiterbahn ausgelegt. Die andere Seite der Platine beinhaltet nur noch Kontaktflächen für einen Nadeladapter, über den der Mikrocontroller programmiert werden kann. Zum Schutz gegen

Korrosion der Bauteile ist der Sensor mit Schrumpfschlauch (9) geschützt. Der Schaltplan zu diesem Aufbau ist im Anhang von [10] zu finden.

Software: Übertragungsprotokoll und -verfahren

Aufgrund des gewählten On-Off-Keyings, kann die Sende-PLL im einfachen Ein-Aus-Betrieb genutzt werden. Dabei ist der Timer des Mikrocontrollers auf die entsprechende Pulsbreiten ausgelegt. Die Datenleitung wird für die nötige Zeit auf den entsprechenden Pegel der Sendedaten eingestellt. Abb.1.13 stellt den Ablauf der Senderoutine in der Übersicht dar.

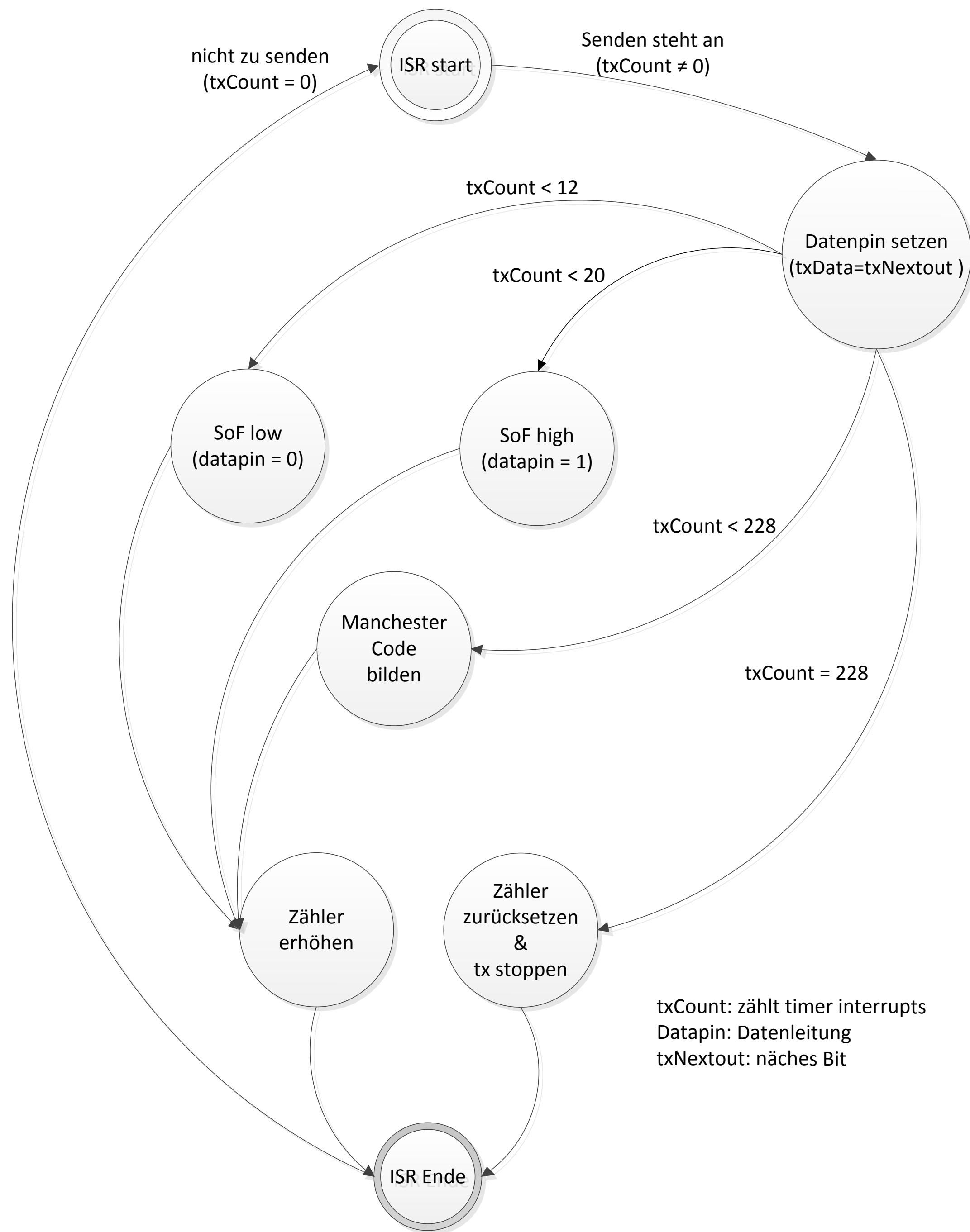


Abbildung 1.13.: Flussdiagramm der Senderoutine

Die verwendete Bitrate ist 10 kBit/s, was eine Bitdauer von 100μs ergibt. Mögliche Schwankungen des Sendetaktes machten den Empfang der Daten sehr aufwändig. Es wurde deshalb eine Manchester Codierung gewählt, die allerdings die effektive Bitrate um die Hälfte reduziert, also 5 kBit/s. Das Übertragungsprotokoll hat dabei den Aufbau aus Tabelle 1.3. Ein Sendeframe umfasst 88 Bit brutto. Dabei wird ein Frame durch eine definierte, uncodier-

Start of Frame	RunIn	Parameter	Adresse	Daten	CRC
definierte Bitdauer	16 Bit	12 Bit	20 Bit	48 Bit	8 Bit

Tabelle 1.3.: Übertragungsprotokoll des Zellensensors

te Bitdauer von etwa 800μs angekündigt. Danach folgt die RunIn, eine definierte Bitsequenz (0x0001), die zur Kalibration der Bitdauer dient. Im Protokoll sind noch neben den eigentlichen Messdaten und dem CRC Byte zusätzliche Informationen angedacht, die jedoch noch nicht genutzt werden. Zukünftige Kalibrierdaten oder sonstige Statusflags könnten so leicht eingebunden werden.
Als Übertragungsverfahren ist eine Art ALOAH[10] Verfahren gewählt worden. Die Tatsache, dass alle Sensoren auf der selben Frequenz senden und ohne Downlink völlig unsynchronisiert sind, begründet dies. Als Folge senden alle Sensoren in zufälligen Zeitabständen einen Datenframe ab. Kollidieren zwei oder mehrere Frames, gehen Informationen unwiderruflich verloren. Aus diesem Grund dürfen die Zeitspannen nicht zu kurz gewählt werden. Zugleich sollte die Framerate der Anwendung entsprechend ausgelegt sein. Bei der Erprobung mit Gabelstapler-Traktionsbatterien (24V Polspannung, 12 Zellen) lieferten minütliche Messwerte erste Ergebnisse. Zur Minimierung der Verluste wurde dabei pro Sensor im Mittel alle 2 Sekunden einmal gesendet [10].

1.3.3. Realisierung des Empfängers

Alle Messwerte der Sensoren laufen zentral in einem Steuergerät zusammen. Abb. 1.14 zeigt den Aufbau des Steuergeräts als Blockdiagramm.

[10]Stochastisches Übertragungsverfahren für Kommunikationsnetze ohne Kanalabtastung

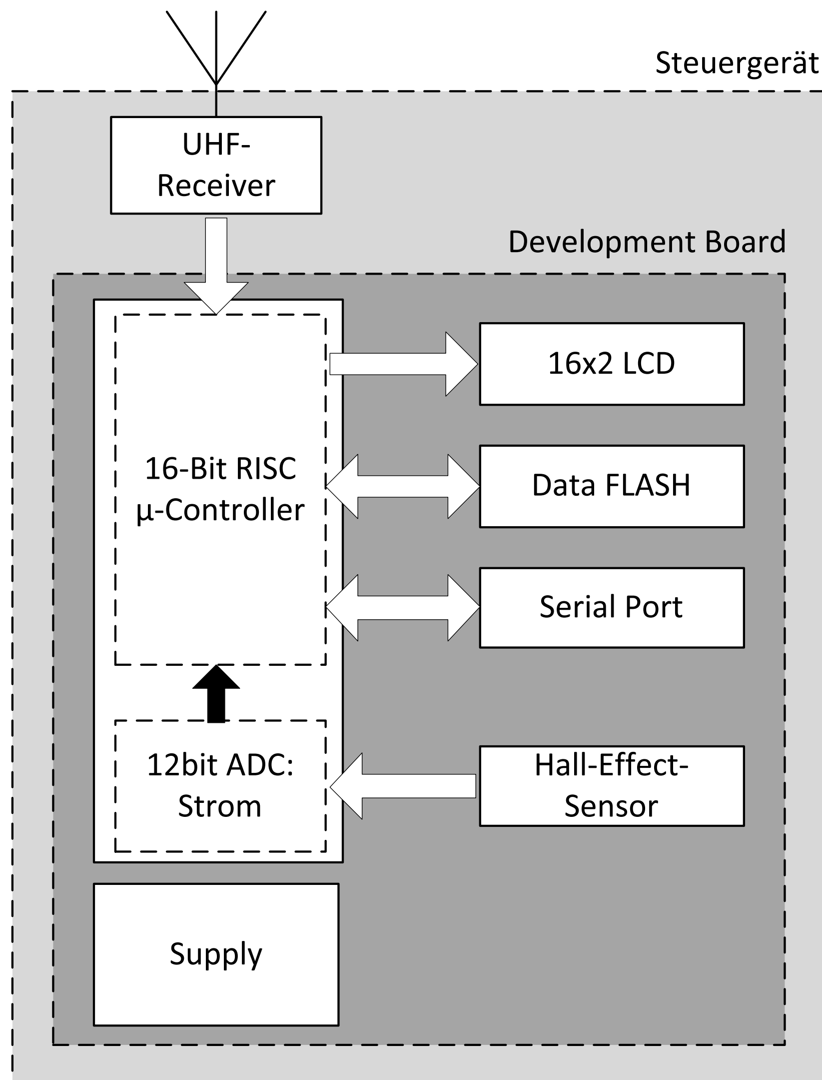


Abbildung 1.14.: Aufbau des Steuergeräts

Abb. 1.15 zeigt dazu den praktischen Aufbau. Der dazu nötige Empfänger (3) ist der Einfachheit halber an ein Development Board der Firma Olimex Ltd. (1) angeschlossen. Die Entwicklungsplattform bring schon von Haus aus zusätzliche Peripherie mit, wie ein 16x2 Display (2) zur Ausgabe von Informationen, sowie einen Flashspeicher zum Speichern der Messwerte. Zum Auslesen der Messwerte wird die serielle Schnittstelle (5) bereitgestellt und erlaubt so eine Auswertung am PC. Der Hybrid Chip RX5000L der Firma RFM Inc übernimmt den Datenempfang. Über eine Datenleitung gibt er das rekonstruierte Digitalsignal an den Mikrocontroller weiter. Sein Ausgang ist sehr hochohmig und benötigt daher ein zusätzlichen Baustein zum Treiben des Dateneingangs des Mikrocontrollers. Dazu dient ein einfaches X-OR-Gatter. Der MSP430x169 von Texas Instruments kümmert sich neben der Auswertung

der Messdaten um die Biterkennung und legt Daten im Flashspeicher ab. Es stehen dabei 8 MByte zur Verfügung. Programing und Debuging der Mikrocontrollers ist über die JTAG Schnittstelle (6) möglich. Die Zentrale Stromerfassung (7) ist über einen Hall-Effekt-Sensor (LM HTFS 400-P/SP2) realisiert.

Die Strommessung wurde erstmalig in der Arbeit von A. Hoops [5] in die Funktionen des Steuergeräts eingefügt. Sie erlaubt durch das Stromintegral über je eine Minute das einfache Anzeigen des Ladezustands durch SoC und SoH. Die Strominformationen werden dabei auch mit den Sensordaten im Flash abgelegt.

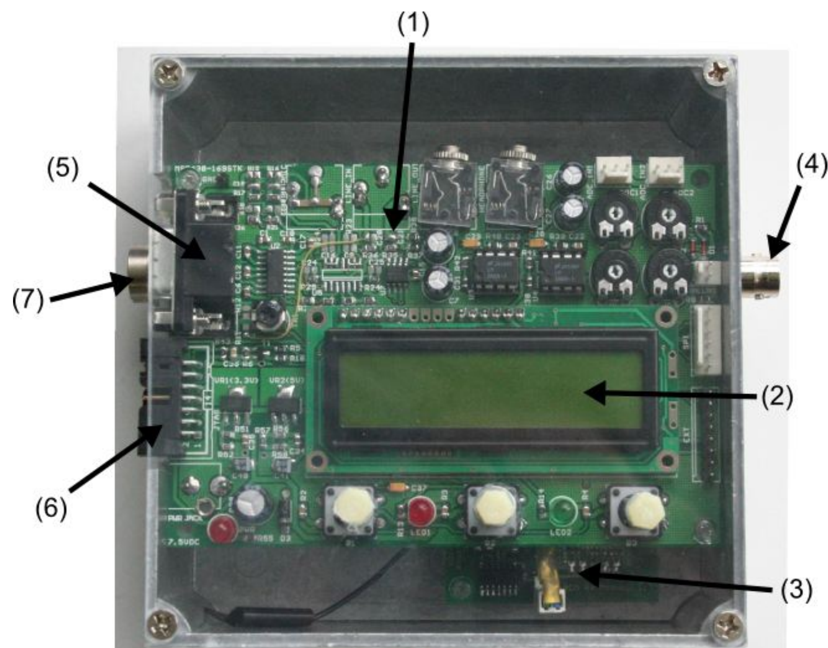


Abbildung 1.15.: Praktischer Aufbau des Steuergeräts

Software: Empfangsroutine

Die Biterkennung aus dem digitalen Basisbandsignal des UHF-Receivers übernimmt der Mikrocontroller. Der Empfangsalgorithmus basiert auf einem Timer-Capture-Interrupt: Tritt am Dateneingangspin eine steigende Flanke auf, wird die Timer Interruptroutine aufgerufen. Hier errechnet sich die jeweilige Zeitspanne zwischen 2 Interrupts aus der Differenz der Zählerwerte des Captures. Die bestimmten Zeitspannen geben Rückschluss auf die empfangenden Bits. Das Digitalsignal in Abb. 1.16 beinhaltet alle möglichen Zeitspannen. Hieraus lassen sich auf Regeln zur Biterkennung schließen. Die Entscheidungsschwellen müssen dabei Toleranzen beinhalten, da durch Temperatureinfluss der Sendetakt stets leicht verändert wird. Dieses Verfahren erfordert bei jedem Empfang eine Kalibration der Bitbreiten. Hierzu dient

die RunIn Sequenz. Durch das Senden von Nullen entstehen, Manchester codiert, steigende Flanken mit der jeweiligen Bitbreite. Der Durchschnittswert stellt die Referenz dar. Eine detaillierte Darstellung dazu findet sich in [10].

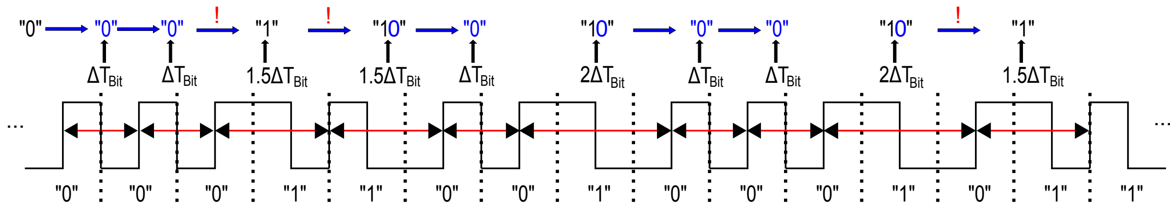


Abbildung 1.16.: Erkennung der Bits aus dem Empfangssignal

Es lassen sich die Regeln[11] je nach auftretender Zeitspanne $x \cdot \Delta T_{Bit}$ ableiten. Dabei wird ein ideales Empfangssignal angenommen.

$x \cdot \Delta T_{Bit}$	zuletzt erkannt	erkannt
1	1	1
	0	0
1,5	1	10
	0	1
2	1	10
	0	10

Tabelle 1.4.: Biterkennung nach Regeln

Die gesamte Biterkennung, sowie die Auswertung findet nach folgender Abb. 1.17 in der Interruptroutine statt.

[11] In Abb.1.16 sind nicht alle Möglichkeiten enthalten. Fehlende Fälle befinden sich in Tab. 1.4.

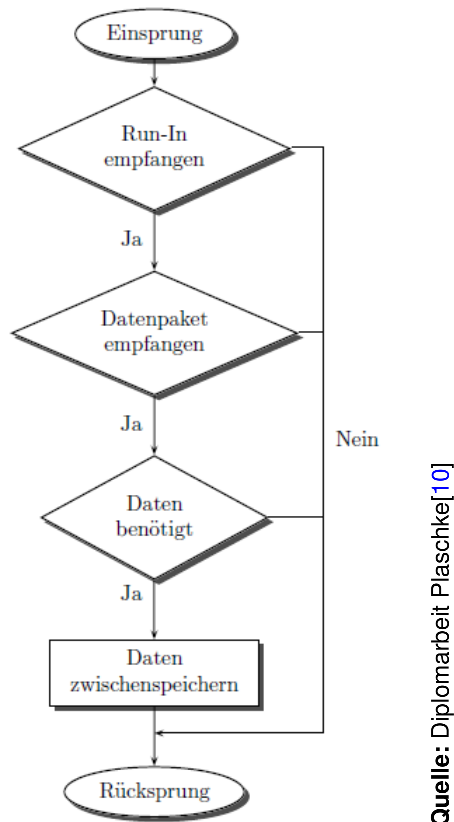


Abbildung 1.17.: Ablauf des Timerinterrupts im Steuergerät

1.4. Anwendung der drahtloser Sensorik auf Kfz Starterbatterien

Im letzten Abschnitt 1.3 wurden die Vorzüge drahtloser Zellüberwachung gegenüber herkömmliche Realisierungen von Batterieüberwachungssystemen dargestellt. Die bisher dabei erzielten Fortschritte zeigten schon erste Ergebnisse bei der Anwendung auf Traktionsbatterien für Gabelstapler. Die gewählten Parameter für die Übertragung der Messdaten an die zentrale Empfangseinheit erwiesen sich dabei als praktikabel, jedoch wurde im Zuge der abgeschlossenen Vorarbeiten das Potential der Aufbauten nicht weiter untersucht. Der Gesamteindruck lässt hier die Wiederverwendbarkeit in Bezug auf andere Anwendungen vermuten. Dieser Frage soll mit besonderem Bezug zur Starterbatterie in folgenden Kapiteln nachgegangen werden.

1.4.1. Vorüberlegungen zu Anforderungen an das System

Wichtigstes Kriterium für die Umsetzung auf Starterbatterien stellt die zeitliche Messauflösung des Systems dar. In Abschnitt 1.1 zeigten sich bereit grob 3 Betriebszustände der Starterbatterie. Diese 3 Zustände stellen dabei unterschiedliche Ansprüche an das System. Die Obere Grenze hängt dabei vom Hochstromereignis bei Starten des Motors ab. Dieses Ereignis dauert bei gängigen Fahrzeugen etwa 1-2 Sekunden. In dieser Zeit müssen die Sensoren genügend Messdaten sammeln, um eine fundierte Aussage über den Ladezustand treffen zu können. Der Energieumsatz ist hier am höchsten, es steigen die Anforderungen an die erforderliche Datenrate.
Während den anderen Betriebsphasen wird von weitaus geringeren Strömen ausgegangen. Die Zellspannungen bzw. der Strom nimmt eine geringere Dynamik an und lässt größere Messabweichungen zu. In Folge ist eine niedrigere Datenrate zulässig. In der Ruhephase ist sogar eine niedrige Datenrate erwünscht: Einher geht damit der Stromverbrauch der Sensorik, da ein Sendevorgang die Zelle zusätzlich belastet. Idealerweise ist somit eine dynamische Datenrate erwünscht. Das System muss sich dabei selbstständig auf den jeweiligen Betrieb einstellen. Abb. 1.18 stellt die Ansprüche in der Übersicht dar.

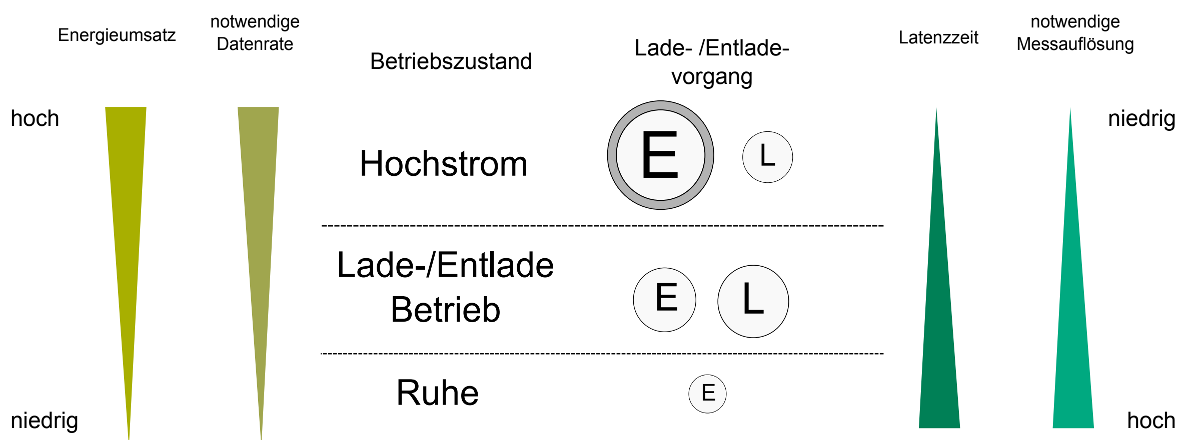


Abbildung 1.18.: Anforderungen in verschiedenen Betriebszuständen

1.4.2. Vorbewertung des Sendeverfahrens

Als Sendeverfahren wurde ein stochastisches Verfahren gewählt. Jeder Sensor sendet dabei seine Messdaten, völlig unbewusst seine Messdaten in einem Datenframe ab. Gleichzeitiges Senden mehrerer Sensoren führt zu einer Kollision und somit zum Datenverlust. Um die Kollisionswahrscheinlichkeit zu minimieren, wählt jeder Sensor zufällig seinen Sendezeitpunkt aus. Übertragungsfehler hängen dabei maßgeblich ab von:

- Anzahl der Sensoren im Sensornetz
- Zeitfenster für Übertragungen
- Bitrate bzw. Framedauer

In Abb. 1.19 ist die Überlagerung aller 6 Sensoren in einem Zeitabschnitt dargestellt. Die Zeitfenster sind, der Darstellung halber, extrem klein gewählt. Die Reihenfolge der Datenframes ist entsprechend der Sendezeitpunkte innerhalb eines Zeitfensters zufällig. Die Zeitfenster wandern mit der Zeit und überlagern sich. Das Übertragungsverfahren ist vollkommen asynchron.

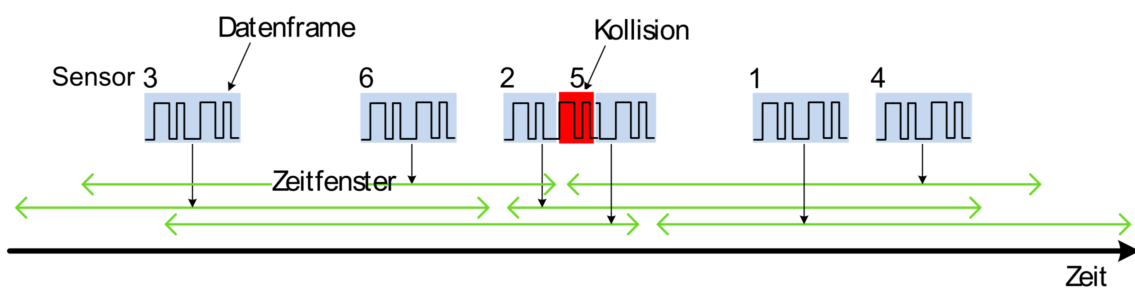


Abbildung 1.19.: Darstellung des Übertragungsverfahrens auf Frame-Ebene

Die letzte Darstellung zeigt, dass die Wartezeiten zur Vermeidung von Kollisionen direkt die Datenrate betreffen. Die Wahl der Parameter wird demnach ein Kompromiss zwischen Daten- und Fehlerrate ergeben. In Verbindung mit den Ansprüchen einer hinreichend hohen Messgenauigkeit muss u.a. dieser Aspekt in dem folgenden Kapitel weiter untersucht werden.

2. Analyse

In Hinblick auf die gesetzten Ziele dieser Arbeit soll hier analysiert werden, inwiefern die vorgegebenen Mittel und Anforderungen der Hard- sowie Software für die Problemstellung angepasst werden müssen. Besonders wichtig sind dabei die Analyse des Übertragungskanals und die Eigenschaften der Sende- und Empfängerseite, von denen maßgeblich die Zuverlässigkeit des Gesamtsystems abhängt.
Die genauen Anforderungen der Sensorik richtet sich nach dem Verhalten der Zellspannungen in den bereits vorgestellten Betriebssituationen. Die Analyse der Signaldynamik gibt Aufschluss über die genaue Dauer der Hochstromsituation und hilft bei der Wahl der erforderlichen Parameter am Sensor.

2.1. Betriebsverhalten von Starterbatterien

Wie in den vorangegangenen Kapiteln beschrieben, werden eine Reihe von Anforderungen an die Starterbatterie gestellt. Sie dient in erster Linie als Versorger des Anlassers zum Starten des Motors. Hinzu kommt die Aufgabe vom Puffern des Bordnetzes, um bei geringen Drehzahlen der Lichtmaschine die Energieversorgung sonstiger Verbraucher zu unterstützen. Als Folge dieser Anforderungen sind die 3 Betriebsarten Ruhe-, Lade-/Entlade- und Hochstrombetrieb näher zu untersuchen. Während im Ruhezustand nur sehr kleine Lasten angeschlossen sind, bei denen sich die Zellspannungen sehr langsam ändern, erhalten sie im Lade-/Entladebetrieb bis hin zum Hochstrombetrieb ein gewisses Maß an Dynamik im Zeitbereich. Ströme von bis zu einigen hundert Ampere belasten die Starterbatterie erheblich und tragen maßgeblich zur Ladungsbilanz bei. Gerade bei gealterten oder schwach geladenen Starterbatterien variieren die einzelnen Zellspannungen und -kapazitäten z.T. erheblich, was sich wiederum auf das Verhältnis der Zellen untereinander negativ auswirkt.
Für einen Eindruck der zu erfassenden Größen, sowie der späteren Parametrisierung der Messalgorithmen ist eine genaue Aufzeichnung des Startvorgangs wertvoll und soll in diesem Abschnitt behandelt werden.

2.1.1. Batterieverhalten im Hochstrombetrieb

Die Erfassung der Zellspannungen kann sehr genau mit dem Oszilloskop vorgenommen werden. Hierbei ist jedoch aufgrund der internen Massekopplung aller Kanäle, die Messung der Zellspannungen direkt pro Kanal nicht möglich. Die Zellen würden so über das Oszilloskop kurzgeschlossen. Eine direkte Messung erforderte zusätzliche Elektronik (Differenzmessverstärker) für eine Potentialtrennung der Zellspannungen. Ein einfacherer Weg ist die Messung der Teilpotentiale mit Bezug zum Massepol der Batterie wie in Abb. 2.1 schematisch dargestellt. Die einzelnen Zellspannungen ergeben sich später aus deren Differenzen: $U_{C_1} = U_1; U_{C_2} = U_2 - U_1; U_{C_3} = U_3 - U_2; \ldots; U_{C_6} = U_6 - U_5$.

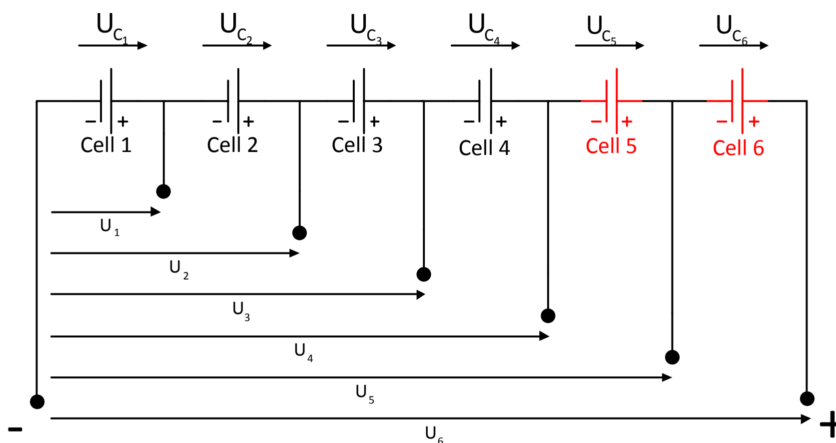


Abbildung 2.1.: Schematische Darstellung der Teilspannungen an der Batterie

Auf diese Weise treten jedoch höhere Spannungen auf, die nach der Digitalisierung durch das Oszilloskop nur eine geringere vertikale Auflösung zulassen. Bei der höchsten, auftretenden Teilspannung von 14,4 V (Bordnetz) an der letzten Zelle würde die tatsächliche Auflösung, bei 20V/div $\Delta U_{mess} = \frac{20V}{255} \approx 80mV$ betragen. Da die Zellspannungen beim Startvorgang schon um einige hundert mV einbrechen können, ist eine derart grobe Auflösung nicht tragbar. Abhilfe schafft hier das Nutzen der Offset-Funktion am Oszilloskop. Beim hier verwendeten Tektronix MSO3034 lassen sich stufenweise Offsetspannungen von teilweise bis über 100V einstellen. Der maximale Offset richtet sich hierbei nach der verwendeten, vertikalen Auflösung. Bei der Auflösung ab 100 - 500mV/div ist ein Offset von bis zu 10V einstellbar. Zum Erhöhen der vertikalen Auflösung bietet sich der High Resolution Modus des Oszilloskops an: Durch Überabtastung und anschließender Mittlung erhält man somit eine ungefähre Auflösung von 11 Bit. Die letzten 2 Zellen erfordern daher das Messen der Zellspannungen über einen Spannungsteiler.

2.1.2. Messaufbau und Durchführung

Für die Messung von 6 Zellen werden 2 Oszilloskope (je 4 Kanäle) benötigt, die über den externen Trigger zu synchronisieren sind. Die in Kapitel 3, Abschnitt 3.1.2 vorbereitete Starterbatterie wurde über Adapterplatinen mit BNC Kabeln angeschlossen. Jeder Adapter der Zellen 1-4 führt die jeweilige Zellspannung über 50Ω Schutzwiderstände zum BNC Anschluss. Die Adapterplatinen der Zellen 5-6 (CH1-2 scope 2) sind zusätzlich mit einem Spannungsteiler versehen, der die Zellspannungen über je $10k\Omega$ halbiert. Für den Massebezug müssen die Spannungsteiler zusätzlich mit dem Minuspol der Batterie verbunden werden. Channel 3-4 des 2. Oszilloskops bleiben übrig für die Messung des Gesamtstroms durch eine Stromzange und den Trigger zum Synchronisieren. Die Strommessung kann direkt Spannungsbezogen mit entsprechendem Verhältnis als Strom ausgegeben werden. Abb. 2.2 zeigt den gesamten Messaufbau.

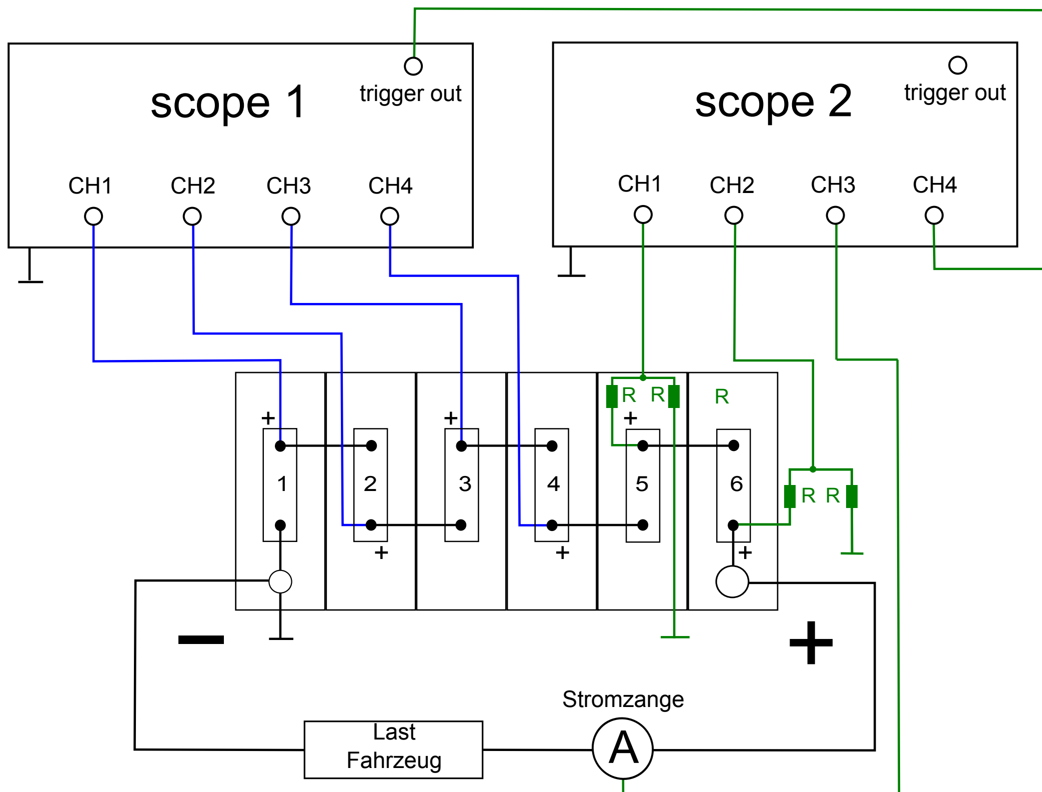


Abbildung 2.2.: Messaufbau der Hochstrommessung

Im Ruhezustand werden die Teilspannungen mittels Offsetfunktion in die Nulllage verschoben. Die Aufzeichnung kann über die Force Trigger Taste im Single-Shot Modus gestartet werden. Die Einstellungen der Oszilloskope zeigt Tab. 2.1.

	scope 1	**scope 2**
Messgr.	Zellspannung	Strom
horiz.	400ms-4s/div	
vert.	200-500mV/div	100A/div
mode	High Res	
trigger	Ch1	Ch4
Anz.Punkte	1M	

Tabelle 2.1.: Einstellungen der Oszilloskope

2.1.3. Auswertung der Messdaten

Nach der Aufnahme des Startvorgangs gemäß dem vorherigen Abschnitt 2.1.2 können die Messwerte in Matlab ausgewertet werden. Die Messdaten geben Aufschluss über maximale Abtastrate und zu implementierende Messgenauigkeit. Anhand der Ergebnisse lassen sich mögliche Algorithmen simulativ erproben und anpassen. Dieser Abschnitt beschäftigt sich mit der Darstellung der aus 2.1.2 gewonnen Daten und derer Auswertung mit Matlab.

Darstellung der Messdaten

Da es sich bei den Messwerten um Teilspannungen gegen Masse handelt, ist zunächst eine Vorverarbeitung in Matlab notwendig. Ebenso müssen die durch den Spannungsteiler halbierten Teilspannungen wieder angepasst werden. Die Messdaten sind entweder im binären Tektronix Format *.isf oder in Textform als *.csf abgespeichert. Nach dem Herausrechnen des Spannungsteilers werden dabei die Differenzspannungen aus den einzelnen Teilspannungen gebildet und entsprechend dargestellt. Die Abb 2.3 - 2.6 zeigen das Ergebnis mit unterschiedlichen Fahrzeugen. Testweise wurde in Abb. 2.6 absichtlich Zelle 3 einzeln entladen. Im Gegensatz zu den anderen Zellen fehlen hier bereits mit 35Ah über ein Drittel der verfügbaren Kapazität. In den Abbildungen sind deutlich die unterschiedlichen Spannungseinbrüche der Zellen beim Start des PKW zu sehen. Anfangs befinden sich alle Zellen in der Ruhelage bei etwa 2V. Nach dem Umdrehen des Zündschlüssels schaltet zunächst der Magnetschalter zum Starten des Anlassers (kurze Peaks). Der folgende Lastmoment des Starters fordert der Batterie einen zunächst sehr hohen Peakstrom ab, infolge dessen die Zellspannungen entsprechend bis zu 600mV einbrechen. Es folgt nun ein wellenförmiger Verlauf, der den unterschiedlichen Kraftaufwand zum Drehen der Kurbelwelle zeigt. Je

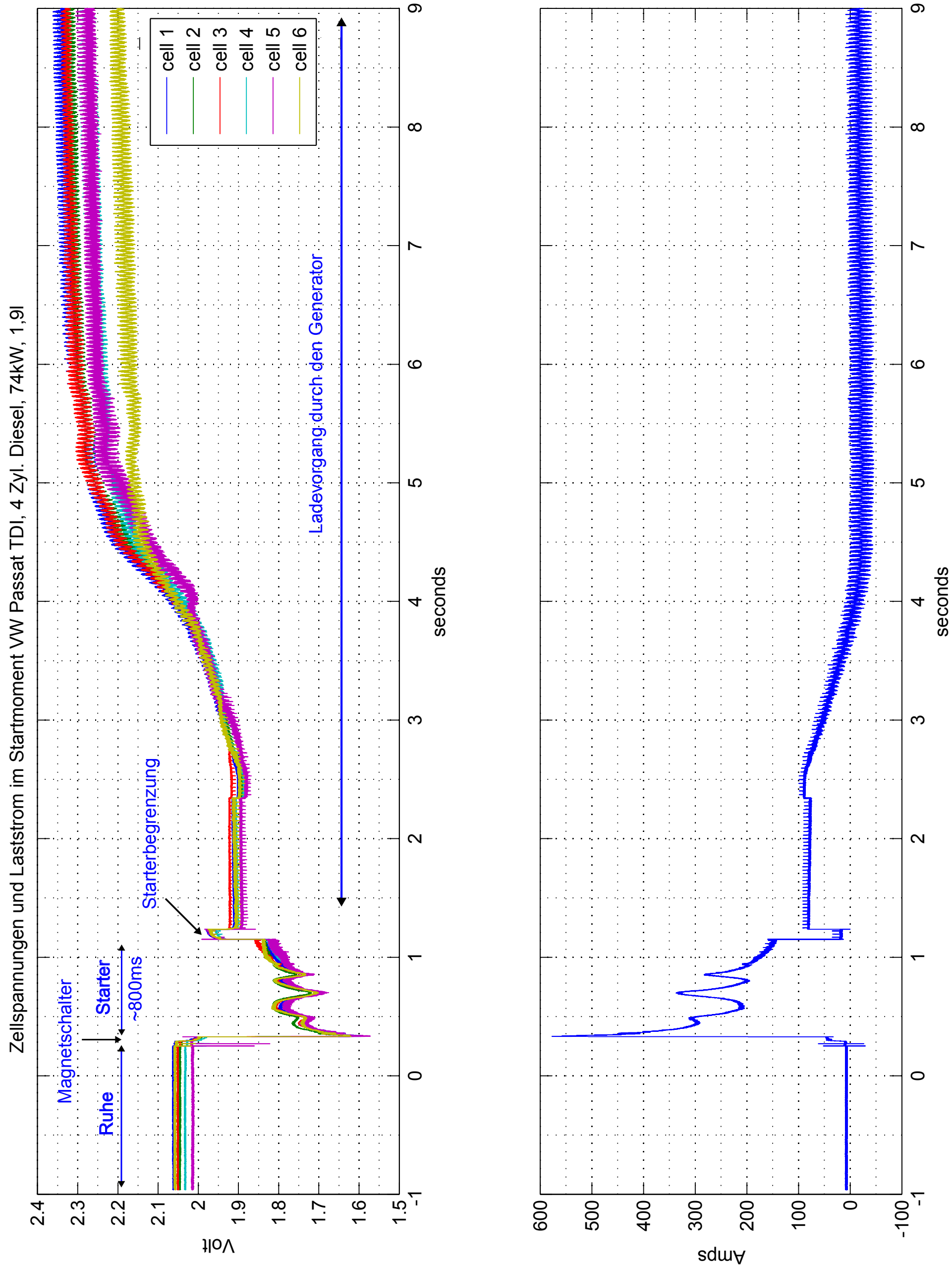


Abbildung 2.3.: Zellspannungen und Laststrom im Startmoment eines VW Passat Diesel über 10s

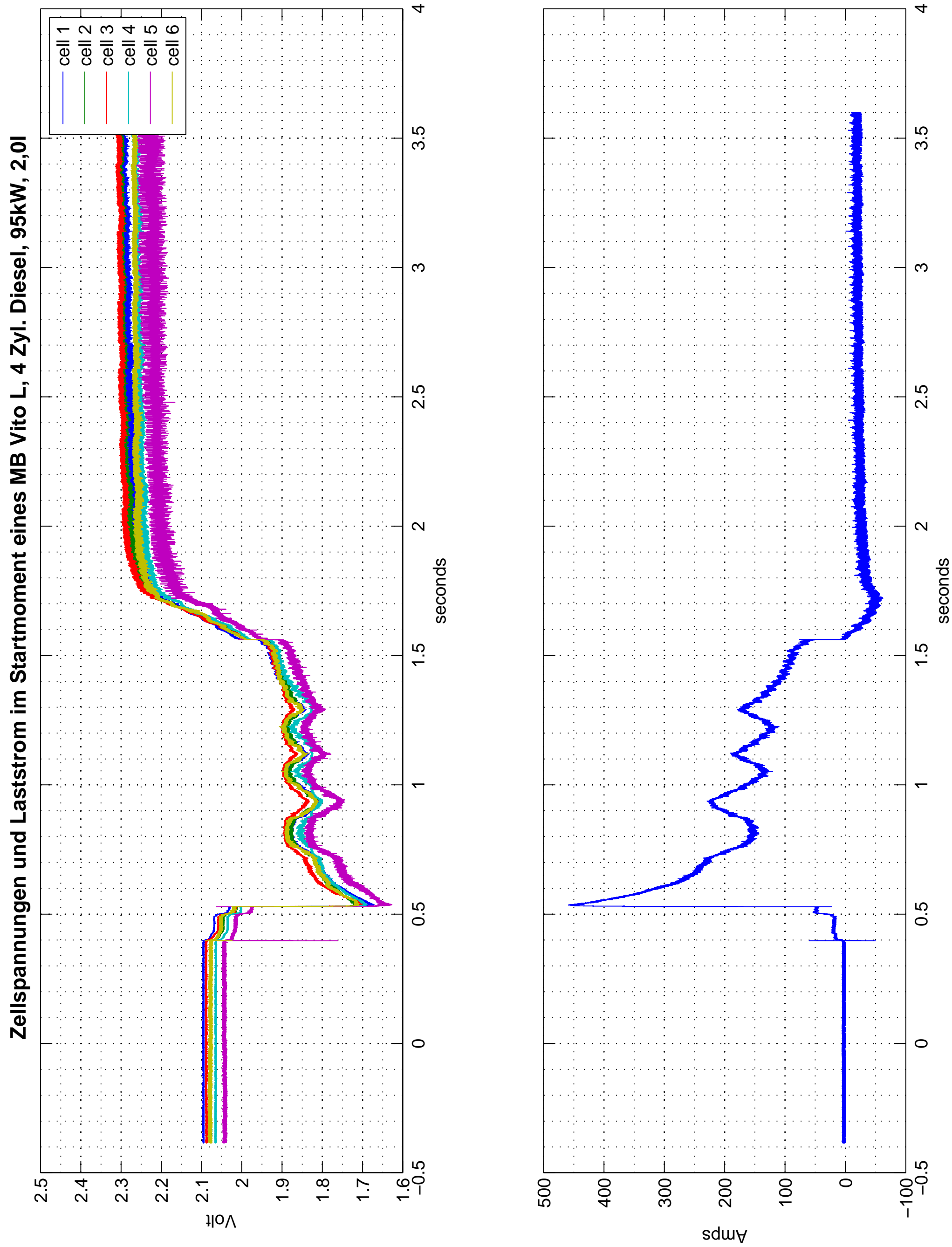


Abbildung 2.4.: Zellspannungen und Laststrom im Startmoment eines Mercedes Benz Vito Diesel über 4s

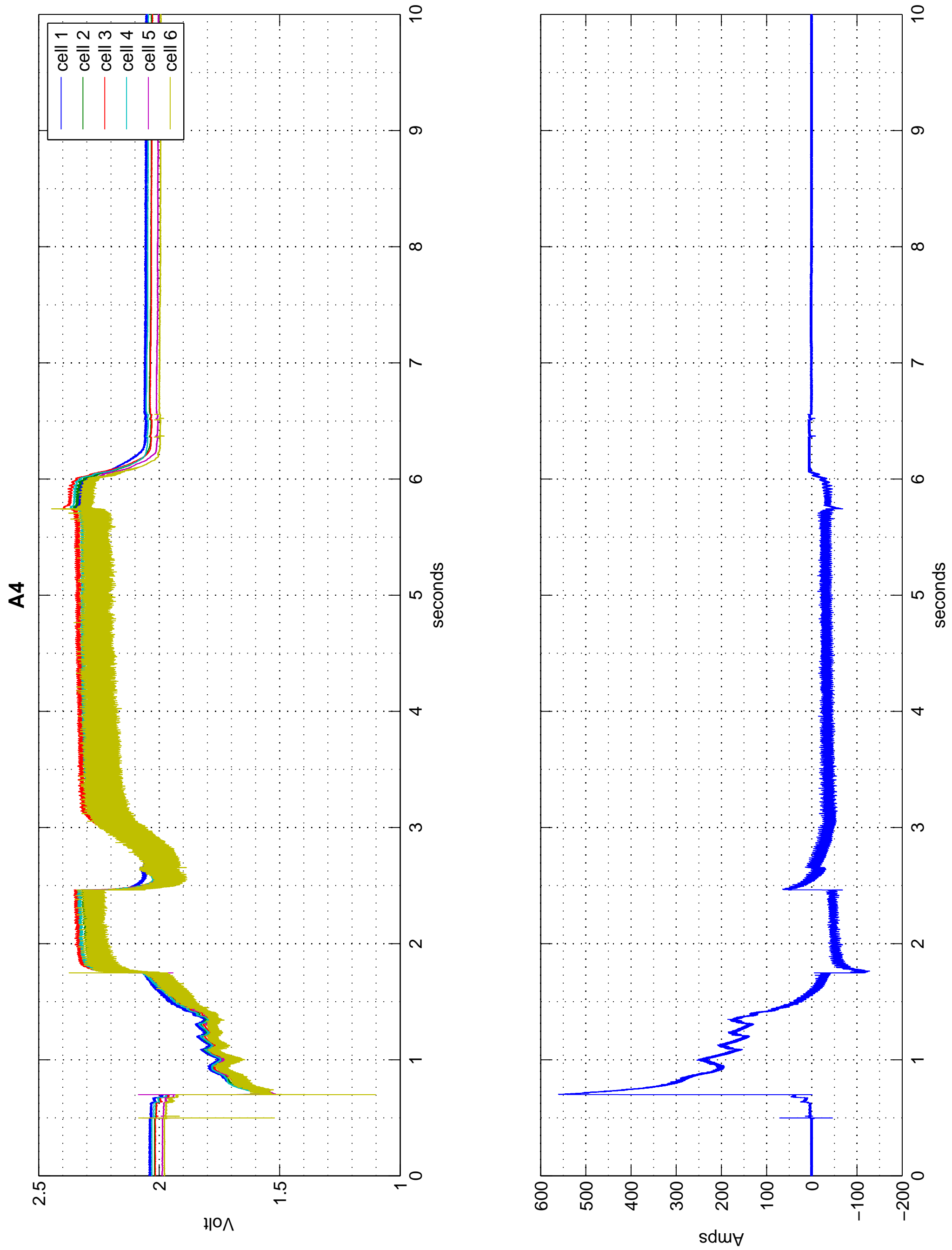


Abbildung 2.5.: Zellspannungen im Startmoment eines Audi A4 über 10s

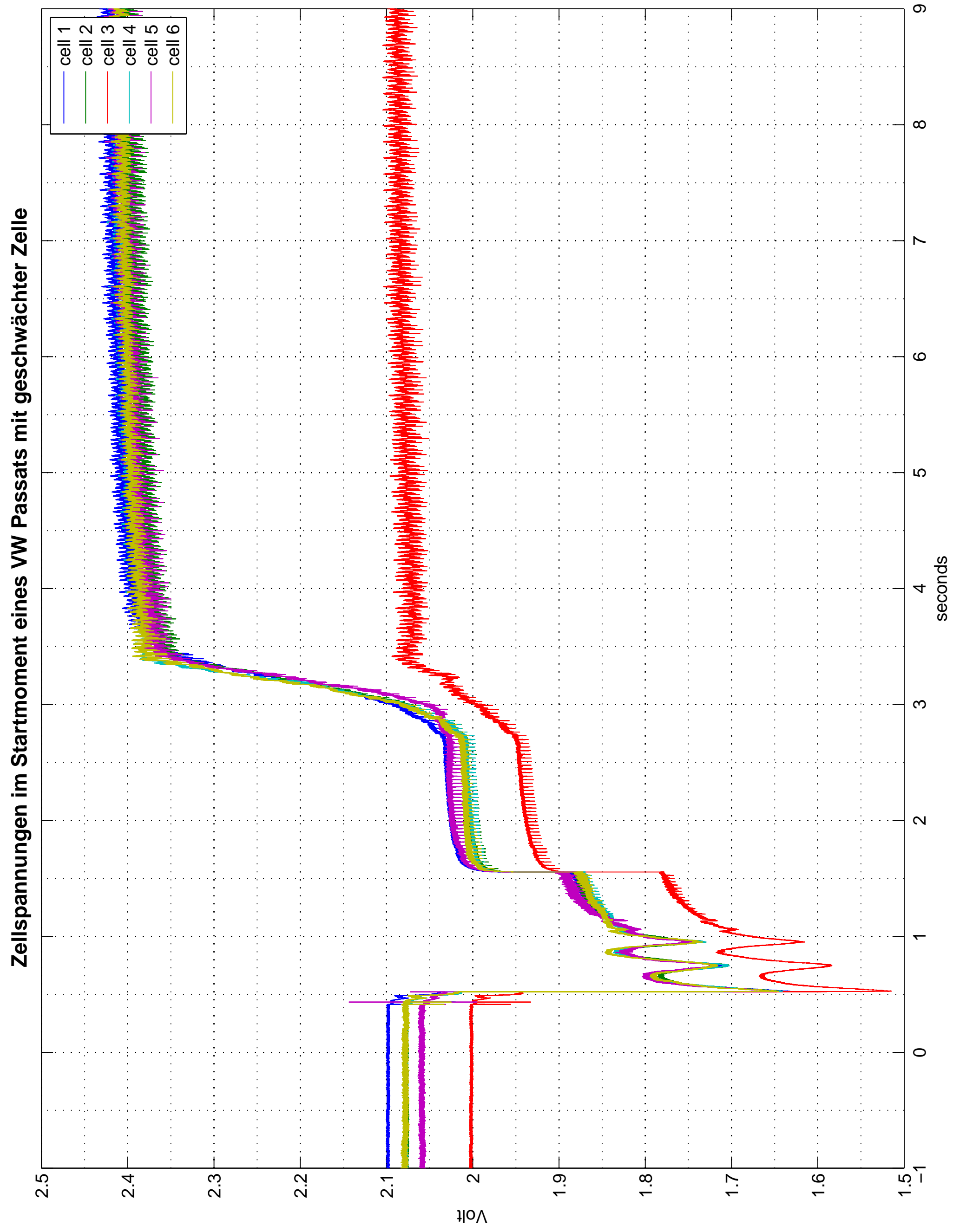


Abbildung 2.6.: Zellspannungen im Startmoment eines VW Passat Diesel über 10s mit geschwächter Zelle (Zelle 3)

nach eingebautem Starter bzw. Motor gestaltet sich der Stromverlauf, ebenfalls abhängig von Kolbenstellung und Motortemperatur, höchst unterschiedlich. In Abbildung 2.3 wird die wellenförmige Starterphase abrupt beendet. Der Wagen scheint hier nach dem Abschalten des Starters die durch Selbstinduktion auftretenden Ströme in die Batterie zurückzuführen. Die steile Flanke am Ende des Startvorgangs deutet darauf hin. Bei den anderen Fahrzeugen prägt allein das Zuschalten der Lichtmaschine diesen Zeitabschnitt.
Die Startcharakteristik des Audi A4 in Abb. 2.5 zeigt im Gegensatz zu den anderen Fahrzeugen einen erhöhten Rauschpegel. Bei einem Otto-Motor entstehen hohe Störimpulse u.a. durch die Zündspule, die hohe Spannungen zum Zünden des Kraftstoffluftgemisches bereitstellt. Ebenso wird kurz nach dem Zuschalten des Generators ein weiterer Verbraucher zugeschaltet, der kurzzeitig die gesamte Leistung der Lichtmaschine beansprucht. Ist der Motor gestartet, setzt nach dem Anlaufen zusätzlicher Verbraucher wie Bordelektronik, Kraftstoffpumpe usw. allmählich der Regelvorgang der Lichtmaschine ein wobei die Batterie wieder geladen wird.
Abb. 2.6 zeigt die Belastung der Batterie mit geschwächter Zelle. Dieser Vorgang bildet zwar keine tatsächlich gealterte Batterie nach, zeigt aber jedoch das unterschiedliche Ladeverhalten im Gegensatz zu den anderen Zellen. Wie im Bild zu sehen, verhält sich die Zelle 3 zunächst wie die voll geladenen Zellen. Erst zum Wechsel in den Ladevorgang durch die Lichtmaschine fällt der viel geringere Spannungsanstieg der Zelle gegenüber den anderen auf. Der Anstieg der Zellspannung ist etwa 3 mal geringer. Aufgrund der geringeren Sättigung nimmt die Zelle 3 einen erhöhten Strom auf. Der begrenzte Strom durch die Lichtmaschine lässt die Zelle nicht schnell genug nachladen, weshalb sie zunächst nur entgegen der 2,4 Volt Ladespannung auf ca. 2,07 Volt im Messbereich verharrt.
Zusammenfassend sind die Zeiten der eigentlichen Hochstromphasen interessant. Den Vergleich der gemessenen Fahrzeuge stellt Tabelle 2.2 dar.

Modell	**Motor**	**Nennleistung [kW]**	**Hubraum [l]**	**Startzeit [ms]**
VW Passat TDI	Diesel	74	1,9	820
MB Vito	Diesel	95	2	1200
Audi A4	Benziner V6	142	2,8	1000

Tabelle 2.2.: Vergleich der Fahrzeuge

Aus der Tabelle ist eine mittlere Hochstromdauer von etwa eine Sekunde erkennbar, wobei die Angaben von der ersten fallenden Flanke bis zum Zuschalten des Generators betrachtet wurden. Die Dauer erhöht sich entsprechend der Größe bzw. Leistung des Motors. Grundsätzliche Unterschiede zwischen Benziner und Diesel sind dabei zu erwarten.

Bewertung der Messmethodik

Die dargestellten Messdaten sind in Hinblick auf Genauigkeit folgendermaßen zu betrachten: Bei der verwendeten Messmethode wurden Teilspannungen gemessen. Nach dem Errechnen der Zellspannungen weisen die Werte mit ansteigender Zellennummer einen steigenden Fehler auf. So ist in Abb. 2.4 ein erhöhtes Rauschen der Zellspannungen 5 und 6 gegenüber niedrigeren Zellen zu sehen. Grundsätzliches Rauschen auf der Zellenspannung verstärkt diesen Effekt zusätzlich (vgl. Abb 2.5), weswegen die Teilspannungen vor dem Verarbeiten gefiltert werden müssen. Toleranzen der Spannungsteiler, sowie Messtoleranzen (Hi-Res ca.11bit) und Offsetfehler (±50mV) der Oszilloskope addieren sich ebenso mit steigender Zelle. Auch die Strommessungen sind skeptisch zu betrachten: Die verwendete Stromzange wurde außerhalb der Spezifikation betrieben. Es sind nur Gleichstrom und Wechselstrommessungen bei 50Hz vorgesehen. Das Messen von zeitlich änderbaren Strömen verfälscht so möglicherweise das Endergebnis durch Überschwinger oder Rauschen.

2.2. Limitierung realer Systeme

Aus den gewonnen Erkenntnissen des letzten Abschnitts ging die mittlere Dauer der Hochstromphase von etwa einer Sekunde hervor. In Bezug auf eine hinreichend gute Erfassung durch die Batteriesensorik stellt diese Dauer den Richtwert dar. Im Folgenden muss untersucht werden, inwiefern die gegebene Messsensorik aus Abschnitt 1.3 den Anforderungen des Startvorgangs genügt. Das gegebene System beschränkt sich auf die Sensoren und die zentrale Empfangseinheit, welche nachfolgend einzeln bezüglich der erforderlichen Datenraten geprüft werden.

2.2.1. Sender/Zellsensor

Der Zellsensor besteht, wie in Abschnitt 1.3.2 dargestellt, aus einem Step-Up Converter zum Anpassen der Versorgungsspannung aus der Zelle, einem Mikrocontroller zur Aufnahme der Messwerte und einem UHF Transmitter zum Übertragen der Messwerte. Das Sendeprotokoll wird durch den Mikrocontroller gesteuert und dem Transmitter zum Umsetzten ins ISM Band übergeben. Neben der maximalen Datenrate des UHF Transmitters von 115 kbps richtet sich daher die eigentliche Datenrate nach dem Takt des Mikrocontrollers. Der Mikrocontroller generiert das zu sendende Datenframe mit den nötigen Bitbreiten. Sein Takt ist auf 2 MHz durch den internen DCO[1] festgelegt. Mit den gewählten Einstellungen aus 1.3.2 ergeben sich etwa 21,6ms für einen Datenframe. Die Übertragung mehrerer Sensoren verringert die Datenrate nochmals. Das Sendeverfahren sieht die zufällige Übertragungen der Datenframes vor, um systematischen Überlagerungen am Empfänger entgegen zu wirken. Das Verringern der Wartezeiten bei der Übertragung geht jedoch einher mit einer höheren Verlustrate. Eine Messwertaufnahme mit der maximalen Datenrate scheidet somit aus. Wie genau die Wartezeiten angepasst werden können ist dennoch für die generelle Übertragung ein wichtiger Aspekt. Die Zuverlässigkeit des Sendeverfahrens und die Betrachtung von Übertragungsfehlern wird in Abschnitt 2.3 behandelt.
Eine Möglichkeit den Startvorgang zu erkennen, ist nun nur noch die Vorverarbeitung im Sensor selbst. Die auftretenden Datenmengen bei der Messung können so auf wenige, aussagekräftige Informationen zusammengefasst werden. Die zu übertragenen Informationen über Zellspannung, Versorgungsspannung und Temperatur unterliegen dabei ohnehin unterschiedlichen zeitkritischen Anforderungen. So wird meistens von langsamen Temperaturänderungen ausgegangen. Die Übertragung der Versorgungsspannung ist in erster Linie zur Überprüfung der Funktionalität der Sensors vorgesehen. Das Übertragen dieser Information reicht in Ruhephasen i.d.R. völlig aus. Kritisch ist hier die Zellspannung:
Der Ladezustand der Batterie basiert auf einer Ladungsbilanzierung. Dazu wird häufig das

[1] Digital Controlled Oszillator: RC Oszillator durch den temperaturabhängigen Widerstand lässt sich kein stabiler Takt erzeugen. Es ist deshalb für die Übertragung stets ein Quarz Oszillator notwendig.

Integral von Strom und Spannung über eine bestimmte Zeitdauer bestimmt. Die Information über die Stromstärke ist im Sensor nicht vorhanden. Nur die Zellspannung kann vorher im Sensor verarbeitet werden und führt zu der nötigen Datenkomprimierung. Dies setzt jedoch einen rechenstarken Mikrocontroller auf jedem Sensor voraus. Unter den Voraussetzung des niedrigen Taktes von 2 MHz und den zeitkritischen Vorgängen der Aufbereitung von Datenframes ist eine zeitgenaue Vorverarbeitung durch Integration im Sensor hier jedoch nicht gegeben. Aufwendige mathematische Operationen müssen dabei vermieden werden. Das Bilden des arithmetischen Mittels über eine bestimmte Anzahl von Messproben ist jedoch möglich. Kritisch ist hierbei das Mittel über viele Proben, da so vergleichsweise starke Einbrüche der Zellspannung im Endergebnis eine geringere Bewertung finden. Zu klein gewählte Intervalle dagegen erfordern wiederum höhere Datenraten. Dieser Effekt erfordert wiederum aufwendigere Berechnungen. Anstatt des arithmetischen Mittels erzielt das quadratische Mittel unter diesen Gesichtspunkten das bessere Ergebnis, was jedoch mehr Rechenaufwand erfordert. Eine weitere Möglichkeit besteht in der Mittelung von Messproben geringerer Differenz: Im Hochstrommoment bricht die Zellspannung bis auf den ersten, kurzen Peak für die gesamte Dauer auf das etwa gleiche Niveau ein. Bei solch geringer Streuung der Messwerte ergibt das Mittel einen entsprechend geringeren Fehler. Dieses Vorgehen erfordert dabei eine zuverlässige Erkennung des Hochstromereignisses. Abb. 2.7 verdeutlicht dabei den Unterschied von abschnittsweiser zu „starrer“ Mittelung.

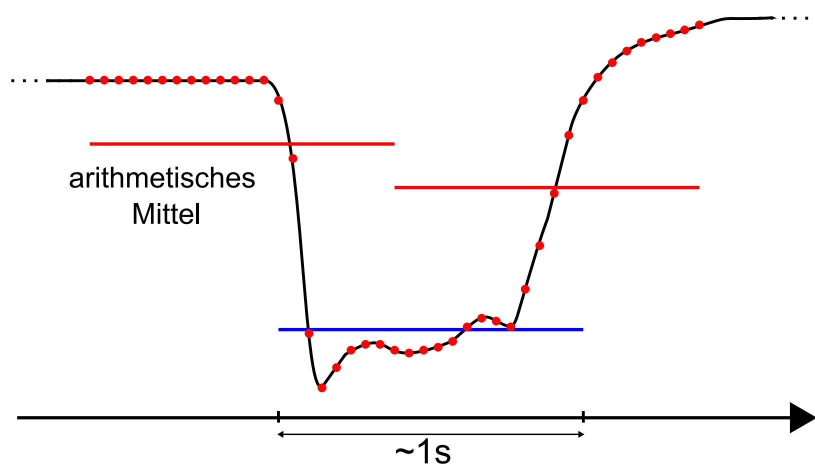


Abbildung 2.7.: Vergleich abschnittsweiser zu „starrer“ Mittelung des Hochstromereignisses

Die Grafik zeigt, dass das abgepasste Mittel über den tatsächlichen Hochstrombereich einen viel geringeren Fehler verursacht, als das starre Mitteln. Trifft das Mittel-Intervall nicht exakt genug mit dem eigentlichen Hochstromfall überein, kommt es zu gravierenden Abweichungen. Zudem treffen alle Sensoren zufällig auf den Spannungseinbruch. Als Folge entstehen starke Ladeschwankungen der Zellen.
Das abschnittsgenaue Starten der Mittlung setzt eine sehr zuverlässige Flankenerkennung

voraus. Störschwankungen auf den Zellen (Fehlstart, Motorstörung usw) können einzelne Sensoren ebenfalls fälschlich auslösen. Sogar im Sensor selbst ist darauf zu achten, dass beim Aufnehmen von Messwerten die Versorgungsspannung nicht durch den Step-Up-Converter gestört wird (Abb. 2.8). In diesem Fall lässt sich dieser für eine kurze Zeitspanne abschalten. Der Sensor wird für diesen Augenblick über einen Kondensator versorgt. Jeder Peak stellt einen Schaltvorgang des Step-Up-Konverters dar. Dies ist jedoch nicht bei Sendevorgängen möglich.
Im Sendefall belastet der Transmitter mit zusätzlicher Sendeleistung die Spannungsversorgung. Der Kondensator kann in diesem Fall nicht genügend Energie liefern, mit der Folge, dass der Step-Up-Converter in diesem Augenblick aktiv sein muss. Die Messwertaufnahme ist somit nicht möglich.

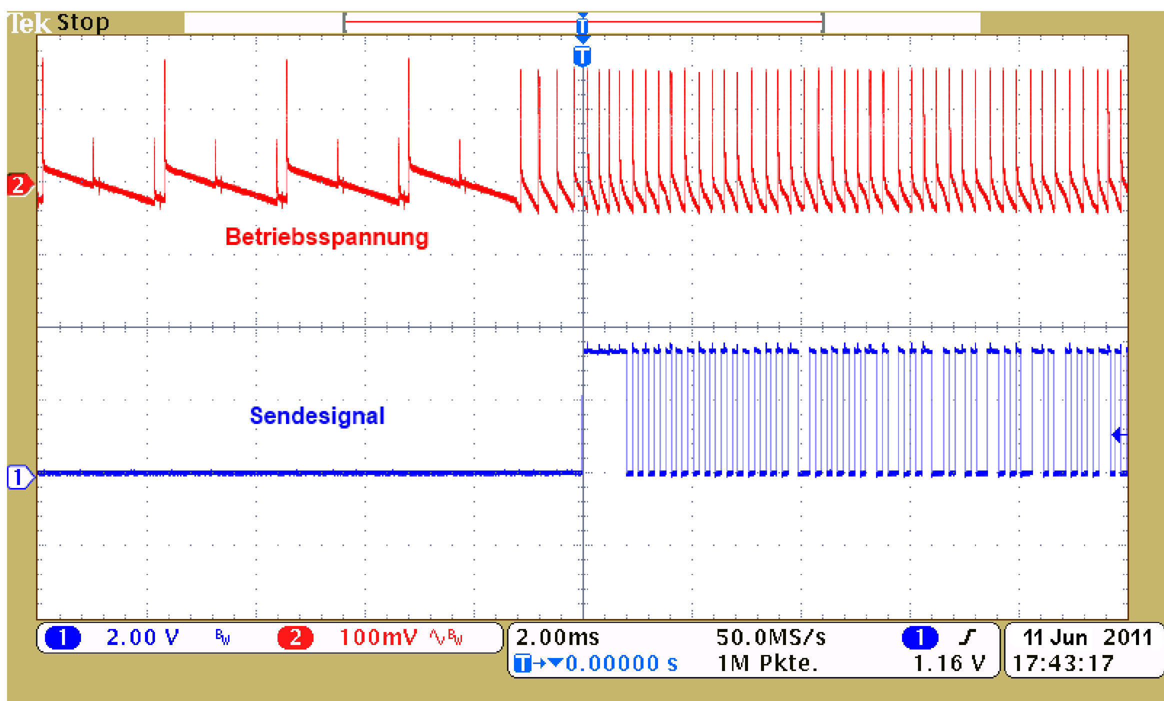


Abbildung 2.8.: Darstellung der Versorgungsspannung des Sensors im Sendefall

Im Falle der kontinuierlichen Messwertaufnahme ist der Sensor bei der Übertragung folglich „blind“. Dieser Punkt spielt ebenfalls bei der Wahl der Datenrate einen große Rolle, da viele Sendevorgänge zum Verlust wichtiger Messwerte beitragen.

Generell ist die Vorverarbeitung durch Zusammenfassung mehrerer Messproben sehr sensibel gegenüber Übertragungsfehlern. Im Fehlerfall gehen so folglich gleich viele Messwerte verloren. Die folgende Gegenüberstellung fasst nochmals die Problematiken der Datenübertragung zusammen.

mittelnde Verfahren	Übertragung einzelner Messwerte
• Erhöhter Rechenaufwand bei quadratischem Mittel im Sensor. Möglicherweise Störung zeitkritischer Aufgaben (Übertragung). • Eine hohe Abtastrate der Zellspannung ist möglich. Situationen mit langsamen Spannungsänderungen werden sehr genau erfasst. • *Mittelnde Verfahren verfälschen die Messspannung. Für niedrigere Abweichungen muss gezielt über das Hochstromereignis gemittelt werden.* • Starke Änderungen der Zellspannungen werden unterbewertet und gehen im Mittel unter. • Der Beginn und das Ende der Hochstromereignisse müssen genau mit über dem zu mittelnden Intervall übereinstimmen, da sonst das Ergebnis zellenweise auseinander läuft. • Der Sendevorgang lässt aufgrund von Störungen der Zellspannung in diesem Moment keine Messwertaufnahmen zu. Der Sensor ist in diesem Moment „blind“. • Übertragungsfehler betreffen viele Messwerte. • *Zu übertragende Daten werden zusammengefasst. Die Ansprüche an die Datenrate sind geringer.*	• *Die Messauflösung hängt direkt von der Datenrate ab. Hohe Datenraten sind nicht verfügbar.* • Es ist keine Vorverarbeitung notwendig. Die Auswertung findet im Empfänger statt. • Verluste einzelner Messwerte fallen nicht ins Gewicht. • Störungen durch den Sendevorgang sind unkritisch, da Sende- und Messvorgänge sequenziell ablaufen.

2.2.2. Empfänger/Datenlogger

In Abschnitt 1.3.3 wurde bereits der Aufbau des Empfängers dargestellt. Im Zuge der Anforderungen des vorherigen Abschnitts über den Zellsensor steht die Untersuchung und Einstellung des Empfängers noch aus.
Der verwendete Empfänger der Firma RF Monolithics basiert auf dem ASH[2] Prinzip. Wie bei üblichen Überlagerungsempfängern wir das Eingangssignal stufenweise verstärkt bzw. gefiltert. Bei ASH geschieht dies über die Zeit, anstatt über die Frequenz: Zur Vermeidung von Rückkopplungseffekten zwischen den Verstärkerstufen werden diese abwechselnd betrieben. Über einen Pulsgenerator wird zunächst die erste Verstärkerstufe (RFA1) angesteuert und ihr Ausgangssignal über eine Verzögerungsglied gehalten. In Folge des Abschaltens des ersten Verstärkers schaltet der 2. Verstärker (RFA2) zu. Auf diese Weise wird das System über die Zeit stabilisiert. Abb. 2.9 zeigt das Blockschaltbild des Empfängers.

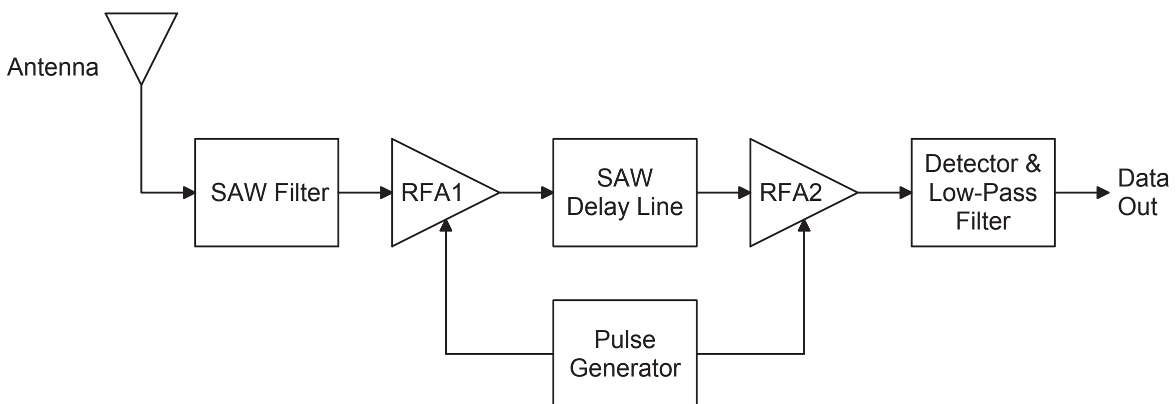


Abbildung 2.9.: ASH Block Diagramm

Die Vorselektion übernimmt dabei ein SAW-Filter[3]. Zum Schluss wird das Digitalsignal noch durch verschiedene Schwellwerterkennungen zurückgewonnen. Der auf diese Weise realisierte Empfänger unterstützt die Modulation ASK[4] bis zu einer Bitrate von 115,2kbps. Hier ist der Empfänger durch externe Beschaltung für den OOK[5] Betrieb eingestellt und ist für Bitraten bis zu 2,4kbps angepasst. Im OOK Betrieb sind zwar nur geringere Bitraten möglich, jedoch ist diese Form der Modulation sicherer bezüglich äußerer Störungen.

[2] Amplifier Sequenced Hybrid
[3] Surface Accustic Wave: Oberflächenwellenfilter basierend auf Interferenzen von Signalen verschiedener Laufzeiten, realisiert mit dem Piezoeffekt
[4] Amplitude-Shift-Keying: Modulation des Digitalsignals auf die Amplitude des Trägerssignals
[5] On-Off-Keying: Einfachste Form von ASK, das Trägersignal wird dabei entsprechend dem Digitalsignal an- bzw. ausgeschaltet

Die Bewertung des Empfangssignals kann zunächst leicht im Zeitbereich erfolgen. Das Signal wurde dabei direkt am Datenpin des Empfängers im Datenlogger abgegriffen und in Abb. 2.10 dargestellt.

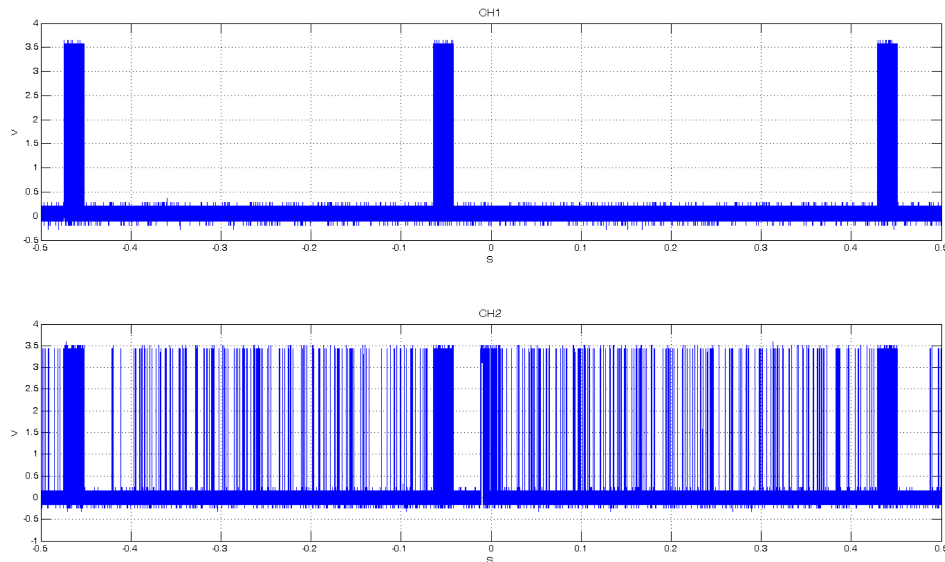


Abbildung 2.10.: Nutzdatensignal und Störungen auf dem Kanal: Sendeseite (oben), Empfängerseite (unten)

Die Abbildung zeigt oben 3 Frames, die innerhalb einer Sekunde gesendeten wurden. Unten ist das zurückgewonnene, digitale Signal am Empfänger dargestellt. Deutlich sind hier Störungen zwischen den eigentlichen Frames zu erkennen. Es fällt weiter auf, dass direkt nach jedem Frame für eine ebenso lange Dauer keine Störungen auftreten. Dies könnte an einer adaptiv angepassten Verstärkung des Empfangssignals liegen. Der Empfänger-Hybridchip enthält zwar eine AGC[6], jedoch ist diese bei der eingestellten Datenrate und der verwendeten Modulation eher optional und abgeschaltet. Nach Messungen im Funkmessraum konnten Störungen durch den Funkkanal ebenso ausgeschlossen werden. Naheliegend ist aufgrund der hohen Streuung fehlerhaft erkannter Bits eine zu sensibel eingestellte Entscheiderstufe der Digitalsignalrückgewinnung. Abb. 2.11 zeigt das analoge Basisbandsignal und das daraus zurückgewonnene Digitalsignal.

[6] Automatic Gain Control: adaptive Vorverstärkung zur Normalisierung schwankungsbehafteter Eingangssignale

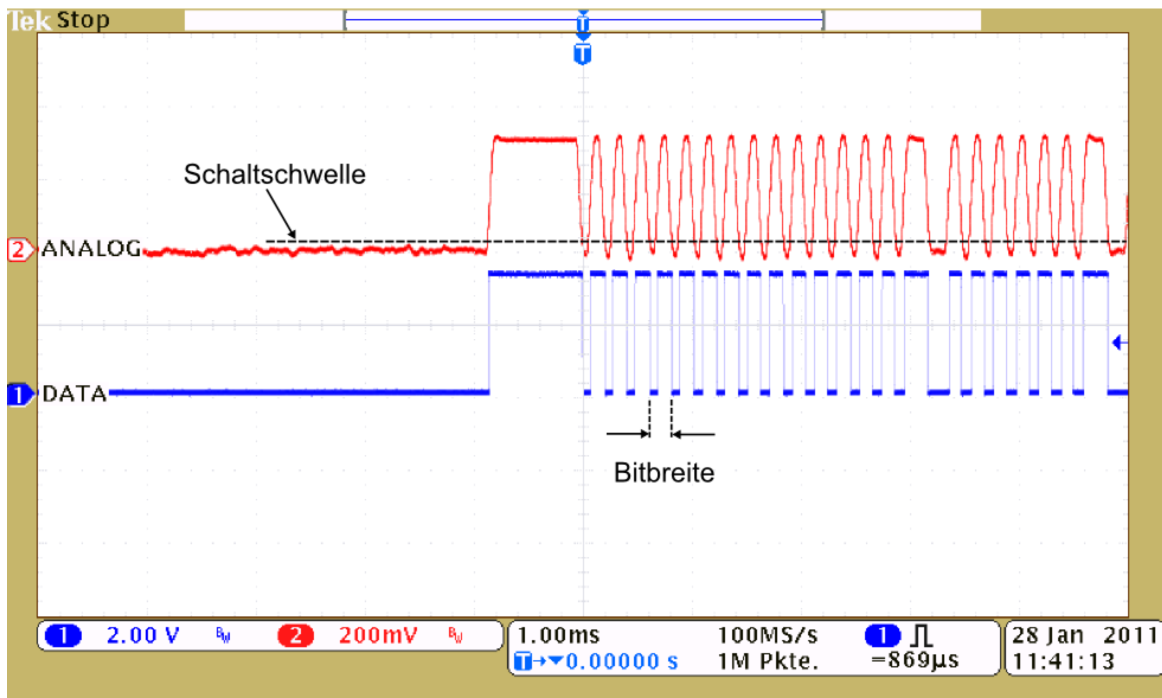


Abbildung 2.11.: Rekonstruktion des Digitalsignals am Empfänger

Die Entscheidung über ein Bit fällt anhand der einstellbaren Schaltschwelle. Das analoge Basisbandsignal wurde zuvor durch einen Tiefpass von hochfrequenten Störanteilen befreit (vgl. Abb. 2.9). Seine Form hängt dabei von der Grenzfrequenz des Filters ab. In Abb. 2.11 zeigt sich ebenso die Änderung des Tastverhältnisses der Bits. Im Gegensatz zum Tastverhältnis des Sendesignals von 50%, steigt es nach dem Empfänger auf etwa 60% an. Dies lässt auf eine zu niedrig eingestellte Grenzfrequenz schließen. Die Form eines Pulses wird zu stark verformt. Die Tatsache, dass der Empfänger laut Datenblatt [12] standardmäßig auf 2,4 kbps Bitrate eingestellt ist, bestätigt dies. Die vorab eingestellte 3dB Bandbreite des Tiefpasses beträgt dabei etwa 4,4 kHz. Die verwendete Datenrate beträgt jedoch 10 kbps und erfordert somit das Anpassen des Tiefpass. Dies kann durch geringe Änderungen der Hardware passend eingestellt werden. Die genau erforderlichen Änderungen sind im folgenden Kapitel: Realisierung unter Abschnitt 3.1 dokumentiert.

Strommessung mittels Hall-Effekt Sensor

Die Funktion der Strommessung wurde vorab in [5] implementiert. Es wurde dazu der Open-Loop-Hall-Effekt Sensor HTFS 400-P/SP2 von LEM eingesetzt. Der Sensor deckt einen Messbereich von ± 400A ab und gibt den entsprechenden Spannungswert mit einem Offset der halben Betriebsspannung aus. Die Messgenauigkeit ist dabei abhängig von seinem

Eigenrauschen. Dieser Wert beträgt laut Datenblatt etwa ± 25mV. Umgerechnet ergibt sich somit eine Schwankung des Nullpunkts von ± 8A. Eine zuverlässige Strommessung ist somit nur bei höheren Strömen jenseits von 8A möglich.
Die vorhandene Messgenauigkeit lieferte bei Traktionensbatterien mit hohen Lastströmen hinreichend gute Werte. Für den Betrieb an Starterbatterien ist jedoch ebenfalls eine gute Auflösung bei sehr geringen Strömen gefordert, welche der HTFS 400-P/SP2 nicht bietet. Als Alternative schlägt [5] einen Hall-Effekt Sensor mit zusätzlichem Messbereich für geringere Ströme vor, welcher extra für Messungen am Kfz entwickelt wurde. Als weitere Verbesserungsmöglichkeit wird im gleichen Zuge der direkte Anschluss des Sensorausgangs über einen Spannungsteiler an den ADC des Mikrocontrollers erwähnt. Die bisherige Anbindung über die Vorverstärkung des Development Boards ist unnötig abhängig von der Temperatur und sollte nach Möglichkeit umgangen werden.
Die letzten Endes vorgenommenen Modifikationen des Datenloggers sind ebenfalls in Abschnitt 3.1 behandelt.

2.3. Übertragungskanal

Wie bei jedem drahtlosen Übertragungssystem müssen die erforderlichen Übertragungsparameter entsprechend den zu erfüllenden Anforderungen abgewägt und eingestellt werden. Bei dem hier verwendeten UHF System stellt die maximale Übertragungsrate eine harte Grenze der Messgenauigkeit dar. Als Fortsetzung der in Abschnitt 2.2.1 begonnenen Untersuchung des Sensors wird in diesem Abschnitt der Übertragungskanal hinsichtlich auftretender Verluste untersucht. Sonstige Untersuchungen bezüglich auftretender atmosphärische Störungen usw sollen hier aufgrund kurzer Übertragungswege keine Beachtung geschenkt werden.

2.3.1. Framefehler

Bereits in Abschnitt 1.4.2 kam der Begriff von Framefehlern auf. Framefehler entstehen durch die Kollision mindestens zweier Datenframes infolge sich überlappender Sendezeiten der Sensoren. Das verwendete Übertragungsverfahren verhindert dabei durch zufällige Übertragungen innerhalb eines vorgegebenen Zeitfensters systematische Überlagerungen. Die zu erwartende Übertragungseffizienz lässt sich durch die Betrachtung eines Zeitabschnitts X, in dem N Datenframes der Länge T verschiedener Sensoren auftreten, näherungsweise bestimmen. Als Voraussetzung der Betrachtung muss $X \gg N \cdot T$ gelten. Abb. 2.12 zeigt dabei den Ausgangspunkt der Betrachtung mit zwei Frames.

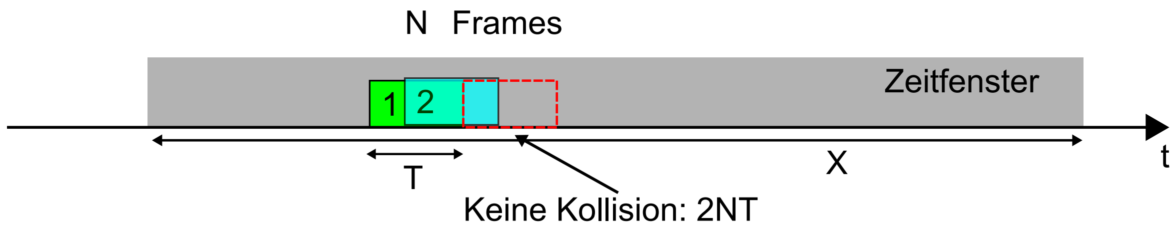


Abbildung 2.12.: Abschätzung der Framefehlerrate

Die Kollisionswahrscheinlichkeit bestimmt sich aus dem Verhältnis der Framelängen zu der Betrachtungsdauer. Die Bedingung keiner Kollision bei dieser Betrachtung erfordert die Verdopplung des Verhältnisses. Es ergibt sich folglich:

$$P_{E,1} = \frac{2(N-1)T}{X} \tag{2.1}$$

Diese Rechnung stellt einen Worst-Case-Fall dar: Es wird hier entgegen des eigentlich in Abschnitt 1.4.2 vorgestellten Sendeverfahrens die Überlagerung der Zeitfenster der Sensoren angenommen. Aufgrund der fehlenden Synchronisation der Sensoren ist eine volle Überlagerung der Zeitfenster nicht immer gegeben (Abb. 2.13).

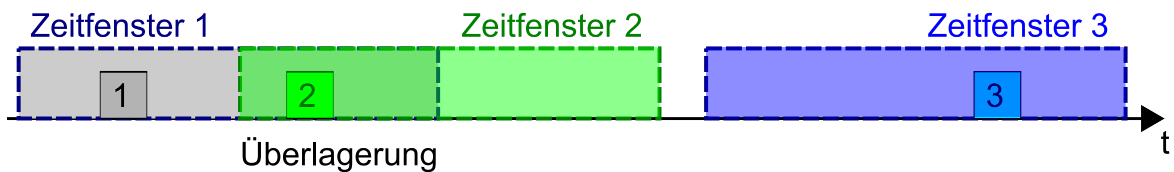


Abbildung 2.13.: Überlagerung der Zeitfenster

Am Beispiel bisher verwendeter Parameter ergibt Gl. 2.1 mit X=2s, T=23ms,N=6 die Kollisionswahrscheinlichkeit $P_{E,1} = 0,115 = 11,5\%$. Der bisherige Einsatz des Sensorsystems sah zudem den Empfang eines Frames innerhalb einer Minute vor. Es konnten so bei der mittleren Sendezeit von 2s bis zu 30 Wiederholungen stattfinden. Im Falle von W Wiederholungen ändert sich 2.1 zu:

$$P_{E,W} = P_{E,1}^{W} = \left(\frac{2(N-1)T}{X}\right)^{W} \tag{2.2}$$

Die Übertragung mit 30 Wiederholungen gilt somit laut Gl. 2.2 als quasi ideal.
Für die hier verfolgte Anwendung auf Starterbatterien gelten jedoch andere Anforderungen. Die Untersuchung der Startvorgänge in Abschnitt 2.1.1 ließ auf die Dauer des zu erfassenden Hochstromereignisses von etwa einer Sekunde schließen. Das Anpassen der Datenrate

auf diesen Bereich lässt die Fehlerrate deutlich steigen. Außerdem muss hier auf das wiederholte Senden von Frames verzichtet werden. Eine deutliche Senkung der Sendezeit wird somit nicht möglich sein. Es muss dabei zwischen Verlusten von Datenframes und Verlust von Messwerten abgewogen werden.
Eine Größe des Zeitfensters von etwa 366ms hat sich bisher als guter Kompromiss erwiesen. Nach Gl. 2.1 ergibt sich dann im schlimmsten Fall eine Fehlerrate von 69%. Anpassungen der Datenrate können dann durch Einbringen von Mindestwartezeiten vorgenommen werden dies hat keinen Einfluss auf die Kollisionswahrscheinlichkeit.

2.3.2. Messung der Framefehlerrate

Das Abschätzen der Framefehlerrate aus dem letzten Abschnitt berücksichtigt keine sonstigen Störungen auf dem Kanal. Die dargestellten Gleichungen sind dabei nur für den schlimmsten Fall totaler Überlagerung aller Zeitfenster gültig. Die sich daraus ergebenden Einschätzungen der Übertragungseffizienz sind daher eher als pessimistisch zu sehen. Für eine genauere Bestimmung der Fehlerrate muss entsprechender mathematischer Aufwand getrieben werden. Ebenso ist eine simulative Erprobung in Software möglich, bei der das Übertragungsverfahren nachgebildet wird. Eine einfache Lösung bietet eine Fehlermessung direkt am Empfänger.
Durch das Senden einzelner Datenframes sind Fehlübertragungen leicht bei der Auswertung bereits empfangener Frames möglich. Jeder Frame bedarf dabei einer eigenen Nummer. Das Übertragungsprotokoll, aus Tabelle 1.3 bietet dafür noch genügend Platz. Die überdimensionierte Adresse benötigt tatsächlich nur 1 Byte und lässt noch 2 Byte für die Nummerierung offen. Im Empfänger ergeben sich dann die Verluste durch die Differenz aus den beiden zuletzt empfangenen Nummern gleicher Adresse.
Das beschriebene Vorgehen ist trotz seiner Einfachheit sehr zeitkritisch und stellt zusätzliche Anforderungen an den Empfänger. Die Auswertung der Verluste erfolgt direkt als Teil der Empfangsroutine, dargestellt in Abschnitt 1.3.3. Die Dauer der Empfangsroutine hat direkten Einfluss auf die zu messenden Framefehler. Dauert sie zu lange, können eintreffende Frames nicht ausgewertet werden und gelten als verloren. Unter diesem Umstand ist besonders die in 1.3.3 erwähnte Problematik des undefinierten Ende der Datenframes zu berücksichtigen: Die Empfangsroutine wertet eintreffende Frames anhand steigender Flanken aus. Endet dabei ein Datenframe auf eine fallende Flanke bzw. ist das zuletzt empfangende Bit eine „1“, startet der Empfang erst bei der nächsten positiven Flanke neu. Erst in diesem Moment gilt das vorige Frame als empfangen. Da die Auswertung selbst Rechenzeit benötigt, dauert die Interruptroutine entsprechend länger und überschneidet sich mit dem nächsten Frame. Dieser Frame gilt dann als verloren. Um diesen Fall zu verhindern, wurde eine abschließe „0“ dem Protokoll hinzugefügt.
Die abschließende Auswertung kann am PC erfolgen. Dazu ist entweder die direkte Über-

tragung über die serielle Schnittstelle oder das Ablegen der Information im Flash des Datenloggers möglich. Beide Fälle bieten die Möglichkeit neben verlorenen Frames zusätzliche Informationen über den Ausstieg aus der Empfangsroutine im Fehlerfall zu speichern. Der frühzeitige Ausstieg tritt anhand fehlerhaft erkannter Bits auf. Ist die Zeitspanne zwischen 2 Flanken außerhalb der Toleranzen, ergibt sich, bis auf den CRC[7], je nach Zustand einer der folgenden Fehler:

- Zeitspanne zu kurz
- Zeitspanne zu lang
- CRC Fehler

Abb. 2.14 zeigt die Anteile einzelner Fehlerfälle am Gesamtverlust der Frames beim Empfang. Es wurden dabei 1000 Frames korrekt übertragen und ausgewertet. In diesem Fall ist der Empfänger noch nicht entstört (vgl. Abschnitt 1.3.3). Es fällt zunächst der hohe Verlust an Frames auf. Pro Sensor gehen fast 80% verloren. Der Anteil tatsächlicher Empfangsfehler ist vergleichsweise gering. Es fällt gerade der verschwindend geringe Anteil an CRC Fehlern auf. Demnach kommt es häufig schon vorher zu Empfangsfehlern. Der Rest an Verlusten entsteht durch Störungen durch den zu empfindlich einstellten Entscheider des Empfangsmoduls.

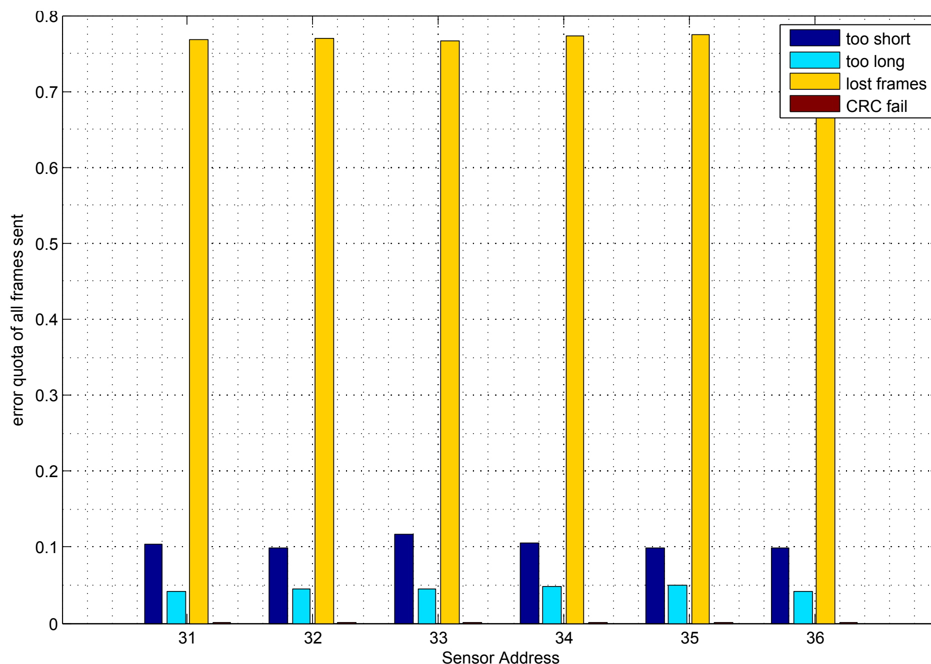


Abbildung 2.14.: Verhältnisse verschiedener Fehlerfälle zum Gesamtverlust an Frames pro Sensor (Empfänger ohne Modifikation)

[7]Cyclic Redundancy Check, zyklisches Prüfsummenverfahren zur Fehlerkorrektur

Abbildung 2.15 zeigt den Anstieg der Fehlerrate mit zunehmender Anzahl teilnehmender Sensoren. Dabei wurden 1000 korrekt empfangene Frames ausgewertet. Nach der Änderung der Empfindlichkeit des Empfängers und deren Anpassung an die verwendete Bitrate (Abschnitt 3.1.1), tritt erwartungsgemäß bei nur einem Sensor keine Kollision auf. Senden dagegen 2 Teilnehmer auf dem Kanal, so lässt sich bereits eine Fehlerrate von 11,3% messen. Schätzungsweise steigt die Fehlerrate um etwa 8% mit jedem weiteren Sensor an. Mit 6 Sensoren ergibt sich eine Fehlerrate von 41,3%. Der Vergleich zu der theoretischen Betrachtung bestätigt eindeutig die damit erreichte Nachbildung beim Fall maximaler Überlappung der Sende-Zeitfenster, in dem jeder Sensor einen Datenframe sendet.

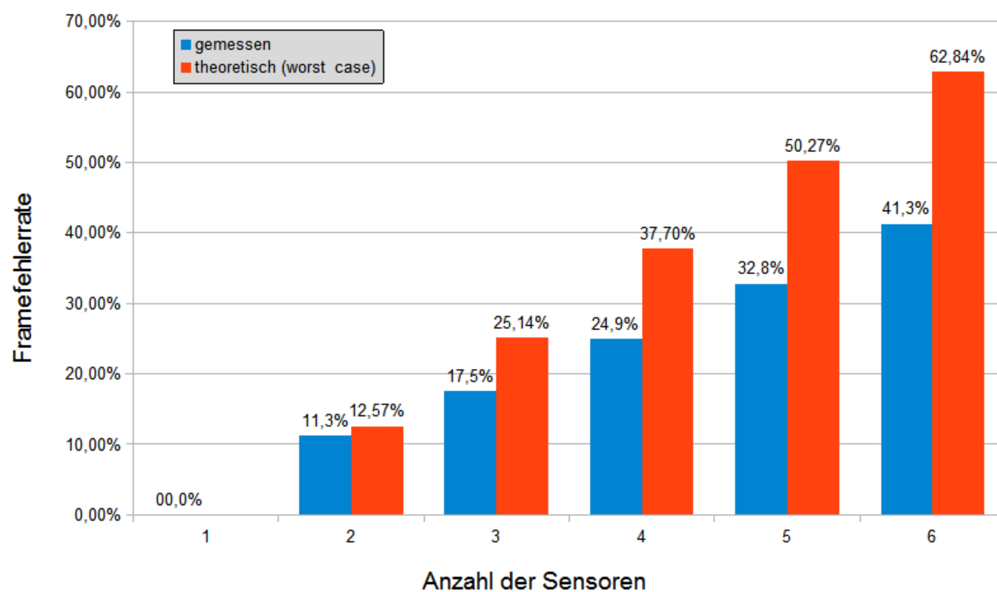


Abbildung 2.15.: Darstellung theoretischer und gemessener Framefehlerraten mit modifiziertem Empfänger

In Abbildung 2.16 sind die aufgenommenen Frames auf der Zeitachse dargestellt. Mit der Information über die verlorenen Frames kann ihre zeitliche Reihenfolge rekonstruiert werden. Dabei wurde ein festes Sendeintervall von 500ms angenommen. Bei dieser Darstellung fallen die Frames verschiedener Sensoren auf einen Zeitpunkt zusammen. Die Anzahl der Frames zu einem Zeitpunkt ließt sich dabei an der Y-Achse ab. Durch diese Darstellung fällt die Häufigkeit von Verlusten besser auf. In 2.16 fallen keine größeren Lücken auf, die auf systematische Kollisionen hindeuten.

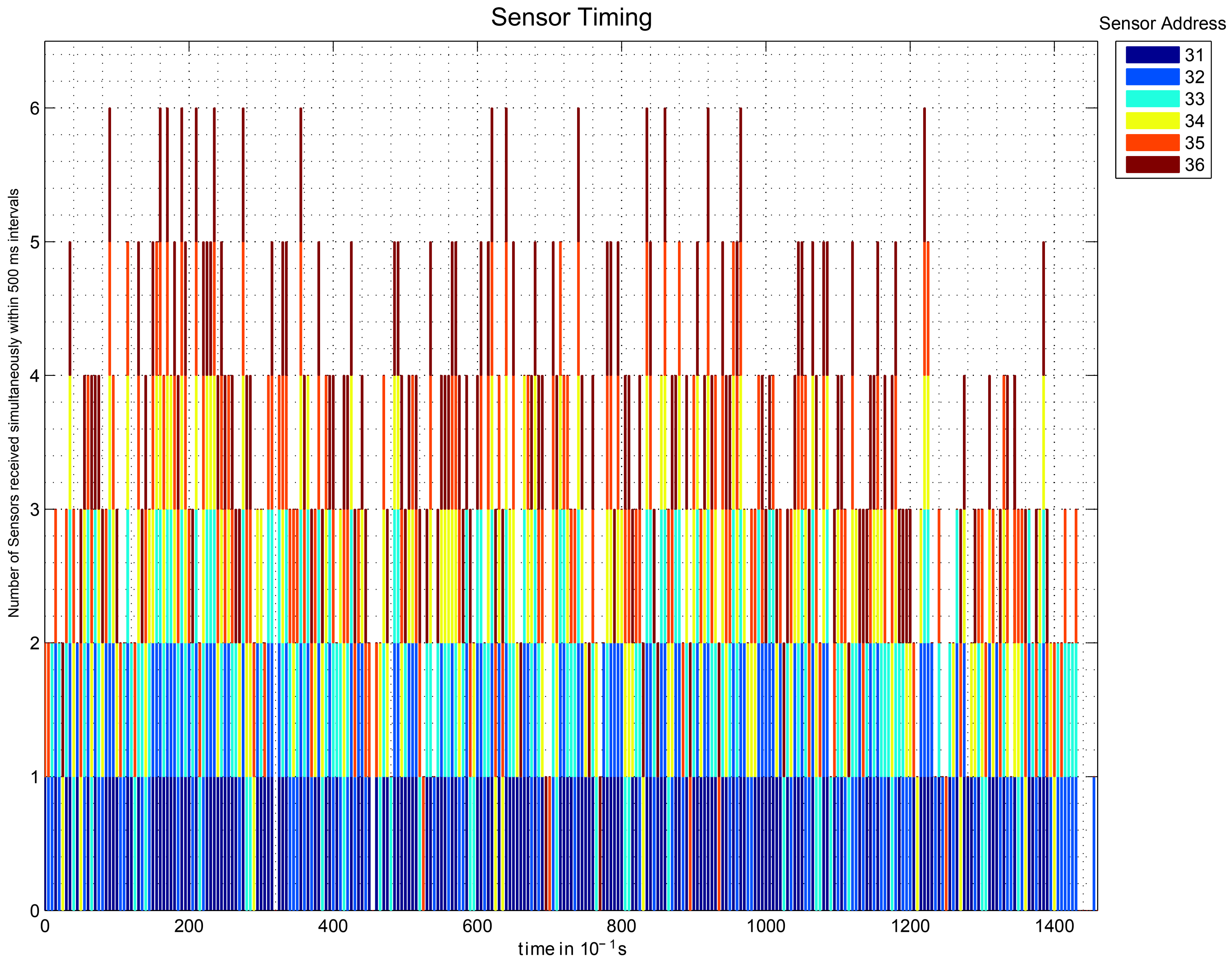


Abbildung 2.16.: Timings der Frames bei Annahme fester Sendezeiten von 500ms

2.4. Auswertung und Konzeption

Auf Basis letzterer Untersuchungen des Sensorsystems hinsichtlich Framefehler und Möglichkeiten der zur Verfügung stehenden Hardware, können die untersuchten Ansätze aus Abschnitt 1.3.2 weiter verfolgt werden. Dabei wurden Vor- und Nachteile einer Vorverarbeitung der Messwerte im Sensor gegenüber der einzelnen Übertragung der Messpunkte behandelt. Während die Vorverarbeitung eine entsprechende Intelligenz des Sensors erfordert, ist die Einzelwertübertragung aufgrund von hohen Framefehlerraten laut Abschnitt 2.3 weniger geeignet. Die Übertragung mehrerer Daten ist dabei jedoch weniger kritisch, da in diesem Fall der Datenverlust einzelner Frames nicht viele Werte, wie bei einer Vorverarbeitung, betrifft.
Eine scheinbar gute Lösung ist hier die Kombination beider Strategien:

- Zur Datenkompression wird über die Messwerte gemittelt. Die Intervalle dürfen nicht zu groß gewählt werden, um die Signaldynamik zu erhalten. Ein Abtastintervall von 1ms über 10 Messwerte ist dabei angesichts der zu erwartenden Spannungsverläufe ein guter Richtwert. Dies erforderte bei direkter Übertragung 100 Frames pro Sekunde.
- Die Datenrate ist stark begrenzt. Durch das Sendeverfahren steigt die Fehlerrate überproportional mit der Datenrate an. Ein Zeitfenster von etwa 333ms ergibt mit 6 Sensoren laut Abschnitt 2.3.2 eine Fehlerrate von 41%. Die Datenrate liegt folglich mit der festen Verzögerung (333ms) zwischen 1-3 Frames pro Sekunde.

Die verfügbare Framerate ist viel zu gering. Die hohe Anzahl entstehender Messwerte kann nicht schnell genug abgearbeitet werden. In diesem Fall führt Quellcodierung im Sensor weiter: Bereits Abb. 1.18 zeigte die unterschiedlichen Anforderungen der drei Betriebszustände der Starterbatterie. Nur Zustände mit hohem Energieumsatz erfordern dabei eine genauere Erfassung durch die Sensorik. In Bezug auf eine Datenkompression lassen sich die gemessenen Daten nach ihrer Bedeutsamkeit beurteilen. Eine Entscheidung lässt sich diesbezüglich nach der Signaldynamik fällen. Treten hiernach größere Änderungen der Zellspannungen zwischen 2 Messwerten auf, deutet dies auf ein Hochstromereignis hin. In diesem Fall müssen die Messwerte entsprechend schnell übertragen werden. Übrige Werte werden dabei weggelassen. In Ruhephasen, konstanter Zellenspannung genügt wieder eine langsame Übertragungsrate.
Die verlustbehaftete Datenkompression erzeugt dennoch phasenweise ein hohes Datenaufkommen und erfordert einen Puffer bzw. Zwischenspeicher zum Angleichen der Datenrate. Dieser ist so auszulegen, dass während einer Hochstromphase hinreichend viele Messwerte aufgenommen werden können und der Puffer nicht überläuft. Die Problematik beim Puffern liegt wiederum beim erhöhten Aufwand der Synchronisation der Messdaten. Zum Erhalt der Zeitbasis muss jedem Messwert ein Zeitstempel beigefügt werden, der ebenfalls den zusätzlichen Jitter der Senderoutine enthält. Im Empfänger ist der Zeitpunkt der Messwertaufnahme entsprechend zu rekonstruieren.

Für die Übersicht des Verfahrens ist eine Darstellung der erforderlichen Instanzen als Schichtmodell sinnvoll (Abb. 2.17).

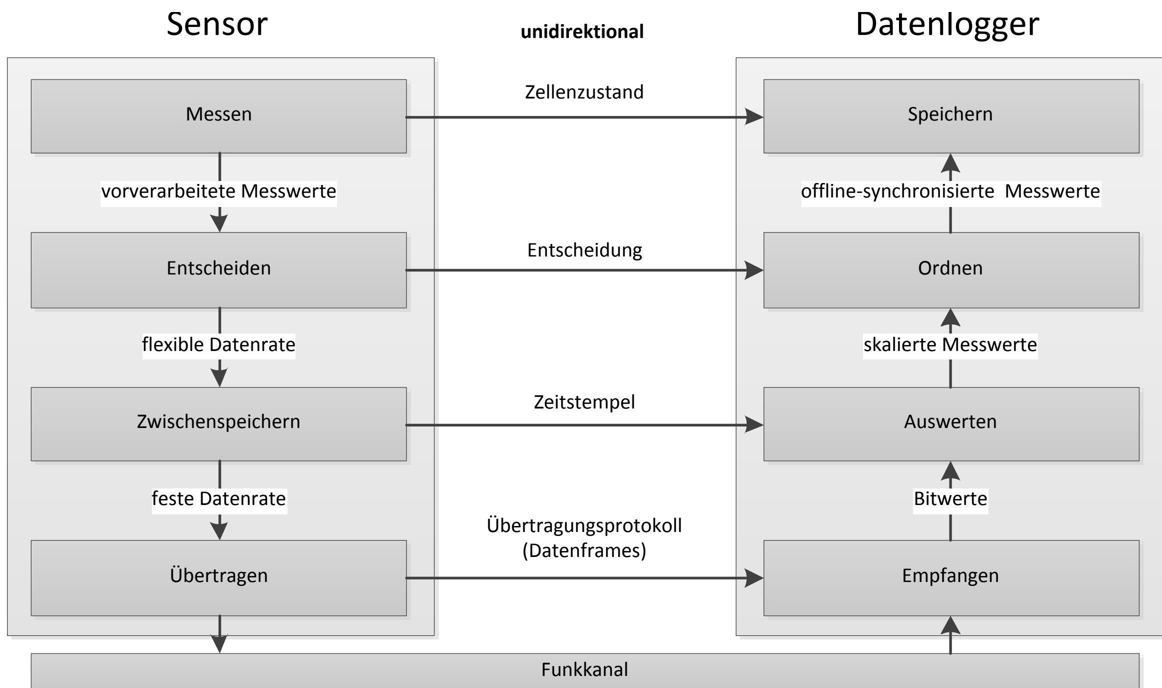
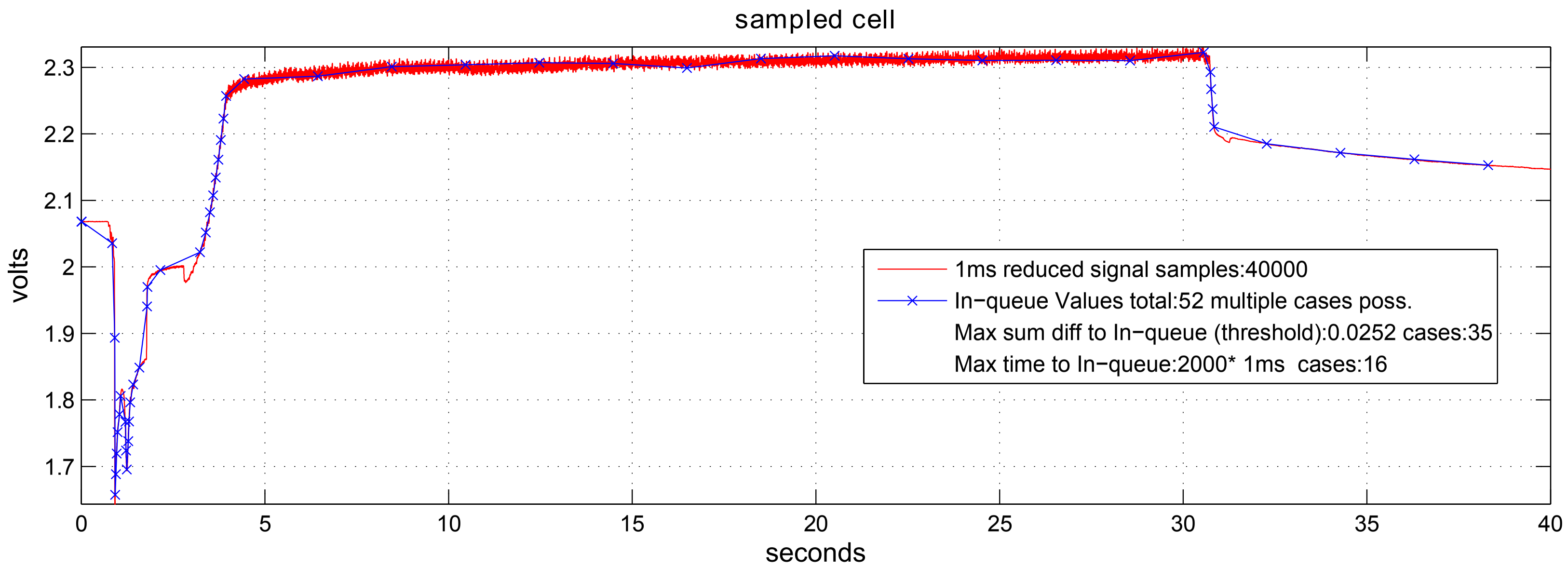


Abbildung 2.17.: Schichtenmodell der Kommunikation zwischen Sensor und Datenlogger

Besonders kritisch ist die Beurteilung der Messwerte hinsichtlich der Hochstromphase. Sie entscheidet über das Messwertaufkommen, das im Puffer gehalten werden soll. Dieser darf nicht überlaufen, um Informationsverlust zu vermeiden. Im Störungsfall bzw. starken Spannungsschwankungen auf den Zellen wird der Puffer unbeabsichtigt belastet. Folgt darauf beispielsweise ein Hochstromereignis, können folgende Werte nicht mehr aufgenommen werden und gehen somit verloren. Die Bedingungen für das Zwischenspeicherung eines Messwertes müssen demnach störungsfest ausgelegt werden. Die letztliche Größe des Datenpuffers ist durch den RAM Speicher des Mikrocontrollers begrenzt (256Byte).
Für die Auslegung der Parameter eignen sich sehr gut die bereits aufgenommenen Startvorgänge aus Abschnitt 2.1.1. Sie erlauben die Simulation des vorgestellten Konzepts. Als Bedingung für das Aufnehmen eines Messwerts in die Queue wurde ein Schwellwert von 25mV eingestellt. Für die Zeitbedingung sind zwei Sekunden gewählt worden. Wie bereits erwähnt, werden die Messwerte über 10ms Intervalle gemittelt. Dies macht die Entscheidung über das Zwischenspeichern robuster gegenüber kurzen Störimpulsen. Abb. 2.18 zeigt das Ergebnis der Simulation anhand des Startvorgangs des VW Passats über 40 Sekunden. Es lassen sich bereits mit den gewählten Parametern für die In-Queue-Bedingung gute Ergebnisse erzielen. Der hohe Spannungseinbruch der Zelle durch den Anlasser wird

entsprechend häufig abgetastet. Hier gelangen schlagartig viele Messwerte in die Queue. Zwischenzeitlich werden schon einige Daten im Mittel alle 500ms abgebaut. Im Übergang zum Ladezustand durch den Generator erhöht sich das Datenaufkommen auf das Maximum von 19 gespeicherten Messwerten. In der langen Ladephase bleibt die Zellspannung, trotz der Störungen durch den Laderegler, unterhalb der festgelegten Toleranz, was genügend Zeit zum Leeren der Queue bietet.
In der Regel folgt auf den Startvorgang eine, bis auf die Störungen des Ladereglers, konstante Ladephase. Folgt jedoch kurz nach dem Starten ein weiterer Einbruch durch Zuschalten weiterer Verbraucher oder durch das erneute Starten des Motors nach einem Fehlstart, ist ein Überlaufen der Queue sehr wahrscheinlich. Der Sensor benötigt stets genügend Zeit um seinen Speicher wieder zu leeren. Abb. 2.19 zeigt im Vergleich dazu den Startvorgang des Audi A4.
Der Audi A4 zeigt eine andere Startcharakteristik. Kurz nach der Belastung der Batterie durch den Starter und des Übergangs zur Ladephase setzt die Ladung für einen kurzen Augenblick aus. Dies erzeugt einen tiefen Spannungseinbruch. In diesem Fall erhöht sich das Datenaufkommen abermals auf bis zu 31 Werten. Noch stärker wird die Queue im Falle mehrerer Startversuche ausgelastet (Abb. 2.20). Die kurzen Ruhephasen zwischen den Starts erlauben kein Abbauen der Daten. Die maximale Auslastung steigt auf 50 an. Viele steile Fanken im Spannungsverlauf erzeugen zudem hohe Abweichungen zum tatsächlichen Verlauf.
Bei nur 256 Byte RAM-Speicher ist laut der Simulation die Größe der Queue maximal auf etwa 30 einzustellen. Bei angenommenen 30 Speicherzellen tritt ein Überlauf des Speichers nur unter extremen Umständen, gerade im letzten Fall, auf. Da durch einen Überlauf keine weiteren Daten abgespeichert werden können, würde im letzten Fall nur der erste Startversuch aufgenommen werden. Für die spätere Ladezustandsbestimmung entstehen so hohe Abweichungen. Die tatsächlich mögliche Größe der Queue muss im nächsten Schritt der Implementation des Verfahrens erprobt werden.

Zuletzt bleibt noch die Strommessung zu erwähnen: Die Strommessung im Datenlogger läuft völlig asynchron neben dem Datenempfang ab. Die korrekte Zuordnung eintreffender Zellspannungswerte ist mit erheblichem Aufwand verbunden, weswegen im Moment darauf verzichtet wird. Diesbezüglich wird im letzten Kapitel 5 ein Ansatz genannt.



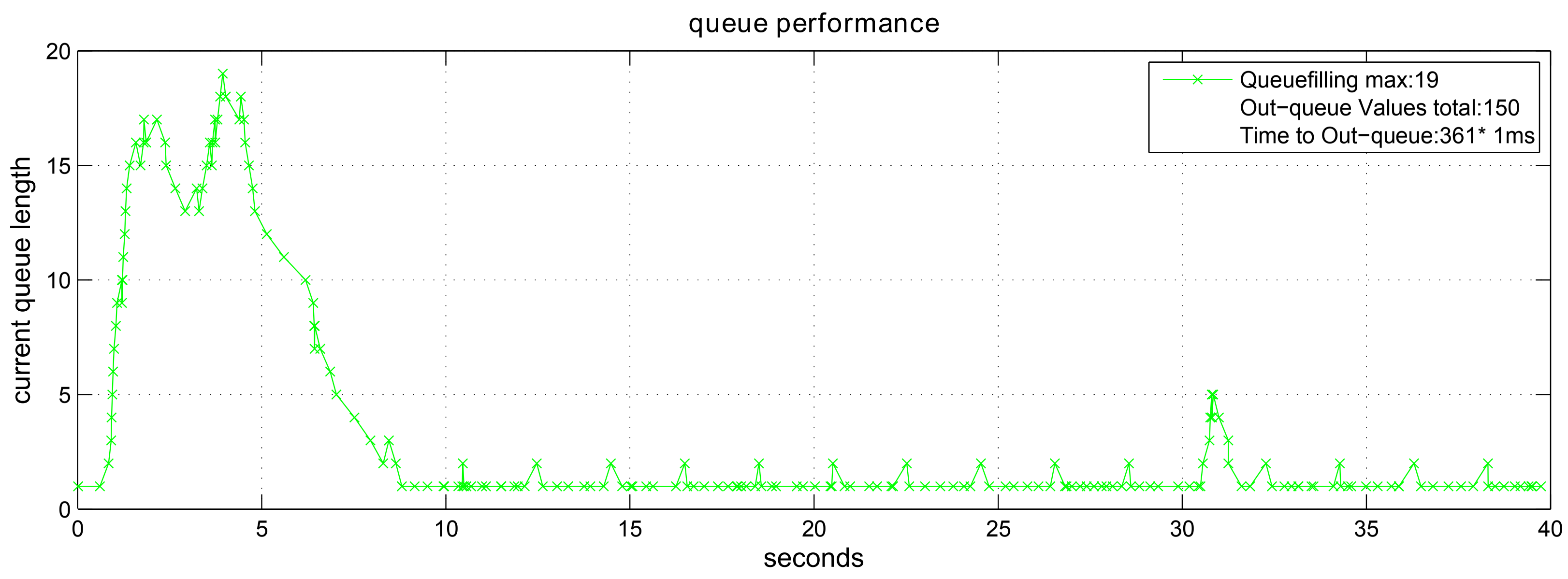
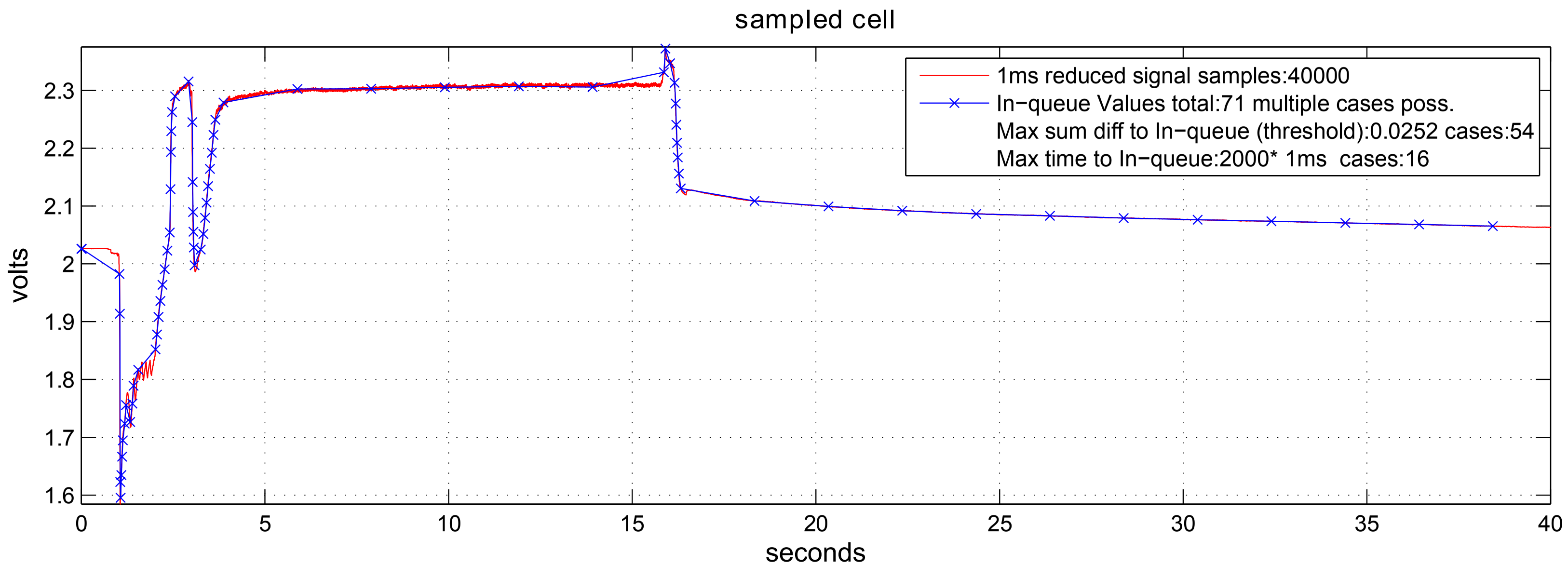


Abbildung 2.18.: Simulation des Datenpuffers im Sensor, Passat 40s



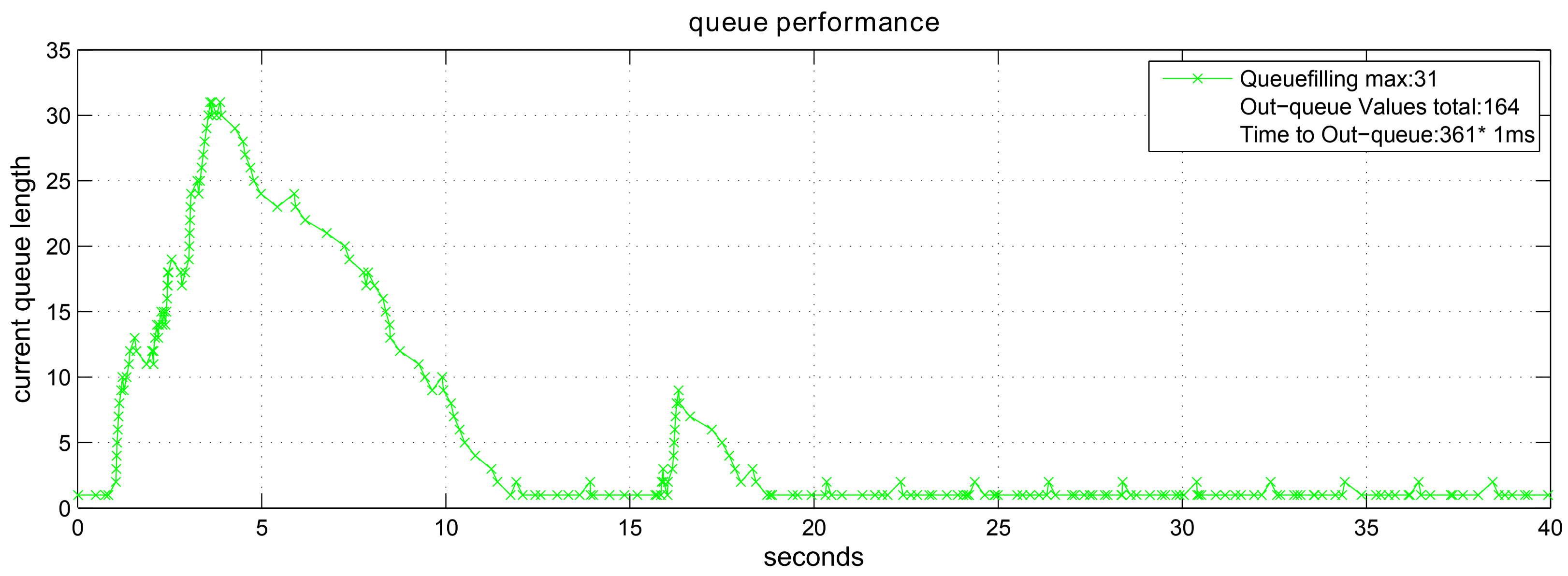


Abbildung 2.19.: Simulation des Datenpuffers im Sensor, Audi A4 40s

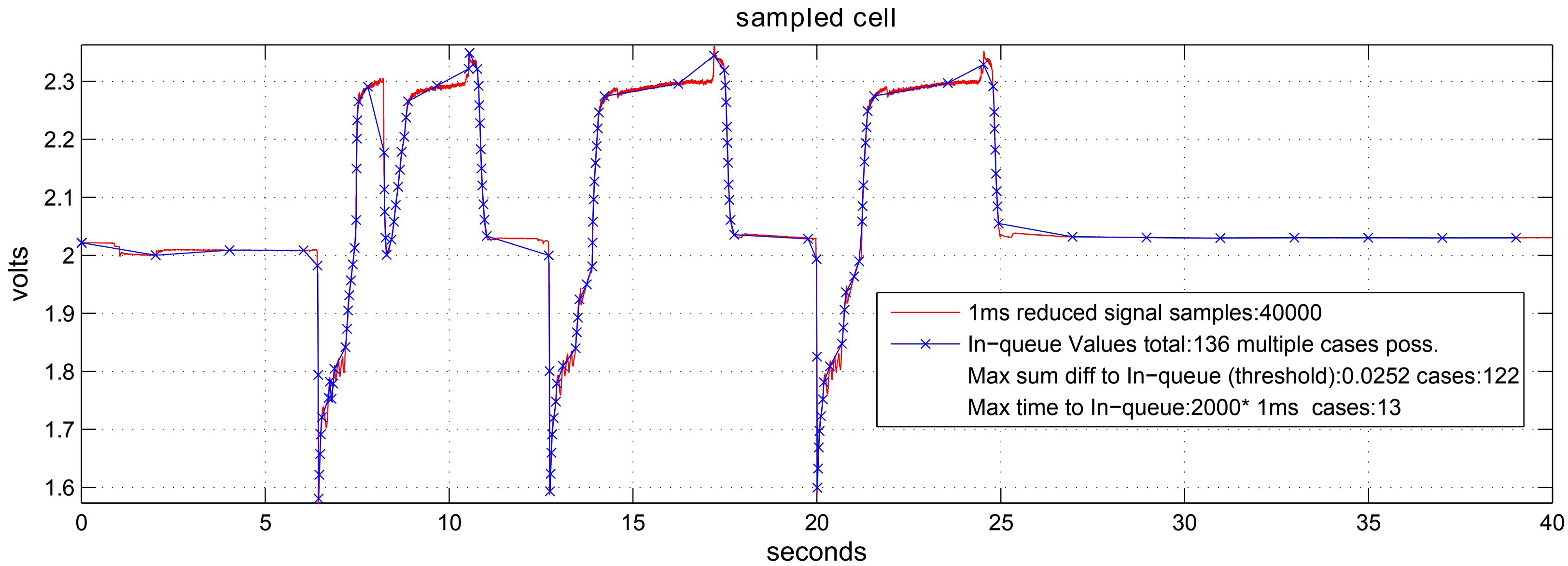


Abbildung 2.20.: Simulation des Datenpuffers im Sensor, Audi A4 3x an bzw. aus 40s

3. Realisierung

In diesem Kapitel werden auf Basis der in den anderen Kapiteln gewonnenen Erkenntnisse mögliche Problemlösungen in Hard- sowie Software umgesetzt. Konkret wird auf Algorithmen von Sensor und Basis, sowie auf mechanische Arbeiten an der Batterie und sonstiger Peripherie eingegangen, die im Laufe der Arbeit angefertigt wurden. Die Funktion ist anschließend im letzten Abschnitt dieses Kapitels nachgewiesen.

3.1. Konstruktion und Änderung verwendeter Hardwarekomponenten

In Kapitel 2 machten schon erste Untersuchungen kleinere Modifikationen der Hardware nötig. Für die durchgeführten Messungen am Fahrzeug war es notwendig entsprechendes Messequipment zu erstellen. Dazu gehört auch der Umbau der Starterbatterie selbst.

3.1.1. Sensorsystem

Für die Anpassung des Sensorsystems an die Anwendung auf Starterbatterien wurden durch neue Anforderungen Änderungen an der Hardware notwendig.

Sensor

Der vorgegebene Zellsensor bedarf keiner großen Änderungen. Seine Funktion an sich ließ sich vollständig durch die Software bestimmen. Für die Montage auf der Starterbatterie waren verzinnte Kontaktflächen zum Anschluss der Zellspannung sinnvoll und wurden nachträglich angebracht. Als Schutz gegen austretende Säuredämpfen beim Laden der Batterie, der die Sensorplatine und deren Bauteile angreift, ist der Einsatz von Schrumpfschlauch sinnvoll. Aufgrund offener Kontaktflächen für den Zellanschluss ist jedoch nicht die gesamte Sensorplatine geschützt. Diese Methode gilt somit nur als Zwischenlösung und erfordert eine gute Abdichtung der Batterie. Für eine permanente Lösung ist die Kapselung der Sensoren notwendig.

Datenlogger

Der Datenlogger basiert auf einem Entwicklungsboard und einer zusätzlich angefertigten Platine für den Empfängerhybrid. Im Laufe von Abschnitt 1.3.3 wurde die Notwendigkeit der Änderung der Empfindlichkeit behandelt. Der Empfänger lässt sich dabei nur durch externe Beschaltung beeinflussen. Abb. 3.1 zeigt die Beschaltung des Empfängerhybridchips für OOK.

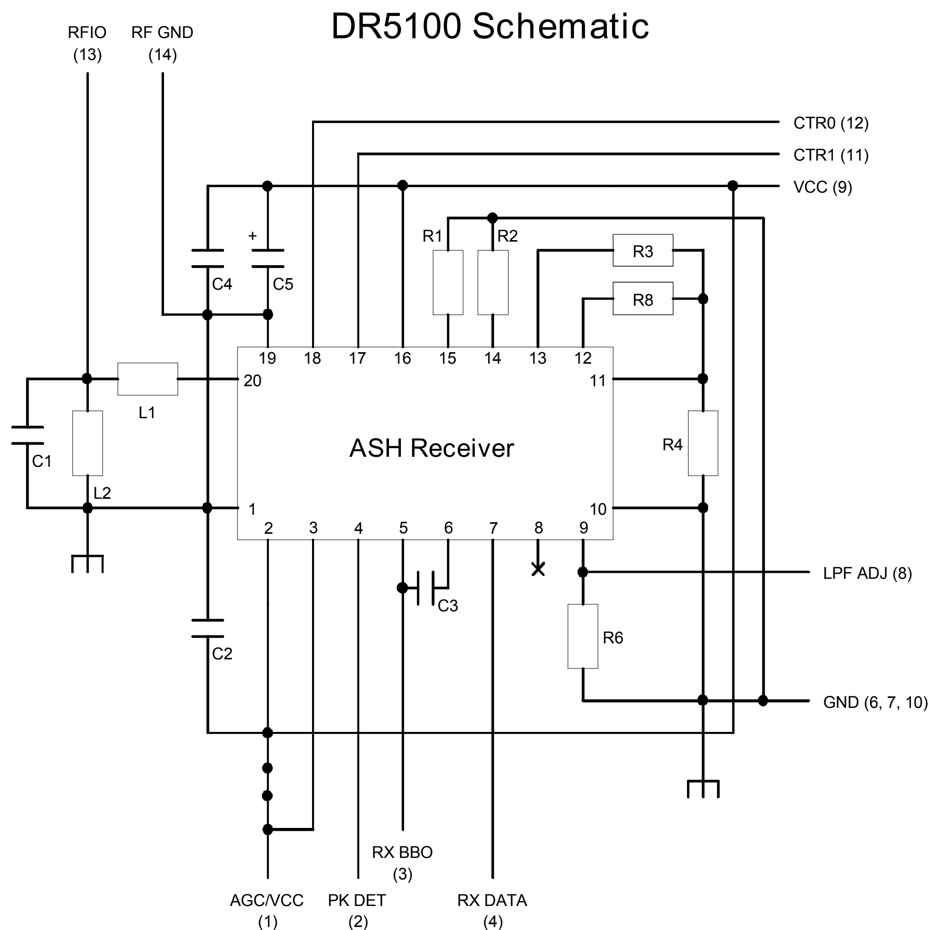


Abbildung 3.1.: Schaltplan der DR5100 Empfängerplatine

Gemäß der Betriebsart ASK sind hier 2 Entscheidungsschwellen einstellbar. Die Minimalschwelle ist hier beim verwendeten OOK ausschlaggebend. Sie ist von der maximalen Empfindlichkeit von 0mV bis zu 90mV über einen externen Spannungsteiler einstellbar. Das Erhöhen von R_3 auf $100k\Omega$ zieht hier erst ab einer Schwellspannung von 90mV den Datenpin auf High und entstört somit den Empfang.

Die Anpassung an die verwendetet Bitrate richtet sich nach der geringsten Pulsdauer des

Sendesignals. Es ergibt sich eine Bitrate von 10 kbps. Das Empfangsmodul wurde bisher bei einer maximalen Bitrate von 2,4 kbps betrieben. Es zeigte sich die damit verbundene Verformung eines Sendepulses eine Erhöhung des Tastgrads auf bis zu 60% nach der Rekonstruktion des Digitalsignals. Dies wird wesentlich durch die Tiefpassfilterung des Basisbandsignals bestimmt. Laut Datenblatt [12] betrug die 3dB Bandbreite des Empfangstiefpass etwa 4,4 kHz. Diese lässt sich über einen zusätzlichen Widerstand parallel zu R_6 einstellen. Für die Bitrate von 10 kbps erhöht hier ein Widerstand von $100k\Omega$ die Bandbreite des Tiefpass auf etwa 18,8 kHz. Die übrigen Werte aus Abb. 3.1 sind dem Datenblatt [13] entnehmen.

Strommessung

Weitere Änderungen des Datenloggers beziehen sich auf die Verbesserung der Strommessung. Gemäß den Ergebnissen aus Abschnitt 1.3.3 ist der Einsatz eines neuen Hall-Effekt-Sensors nötig. Der gewählte Sensor DHAB S/24 von LEM ist ein Open-Loop Hall-Effekt Sensor für automotive Anwendungen. Er besitzt dafür zwei Messbereiche unterschiedlicher Auflösung (Tab. 3.1) und ist bereits mit einem robusten Stecker (Delphi GT500) versehen.

Parameter	Messbereich	
	1	2
Spanne in A	-75 - +75	-500 - +500
Empfindlichkeit in mV/A	26,7	4
Ausgangsspng max in V	4,76	4,76
Ausgangsspng min in V	0,24	0,24

Tabelle 3.1.: Messbereiche des DHAB S/24

Beide Messbereiche sind über jeweils einen Pin herausgeführt. Die maximale Ausgangsspannung des Sensors überschreitet dabei die Referenzspannung des ADC von 2,5V. Die nötige Vorskalierung der Eingangsspannung ist bisher über die Verstärkerstufe des Developmentboards realisiert. Diese Skalierung hängt zusätzlich von der Temperatur ab. Hier ist deshalb die Anbindung des Stromsensors über einen hochohmigen 1:1 Spannungsteiler realisiert. Die Änderungen am Developmentboard sind in Abb. 3.2 dargestellt. Alle entfernten Bauteile sind rot umrandet. Überbrückungen sind schwarz gekennzeichnet. Die Widerstände R_{17} und R_{18} sind durch jeweils $10k\Omega$ ersetzt. Angebunden wird der Stromsensor über die Anschlüsse Ch1 und Ch2. Als Anschluss des Sensors an den Datenlogger wurde ein 4 poliger DIN Stecker gewählt. In Abb. 3.3 ist dies ersichtlich.

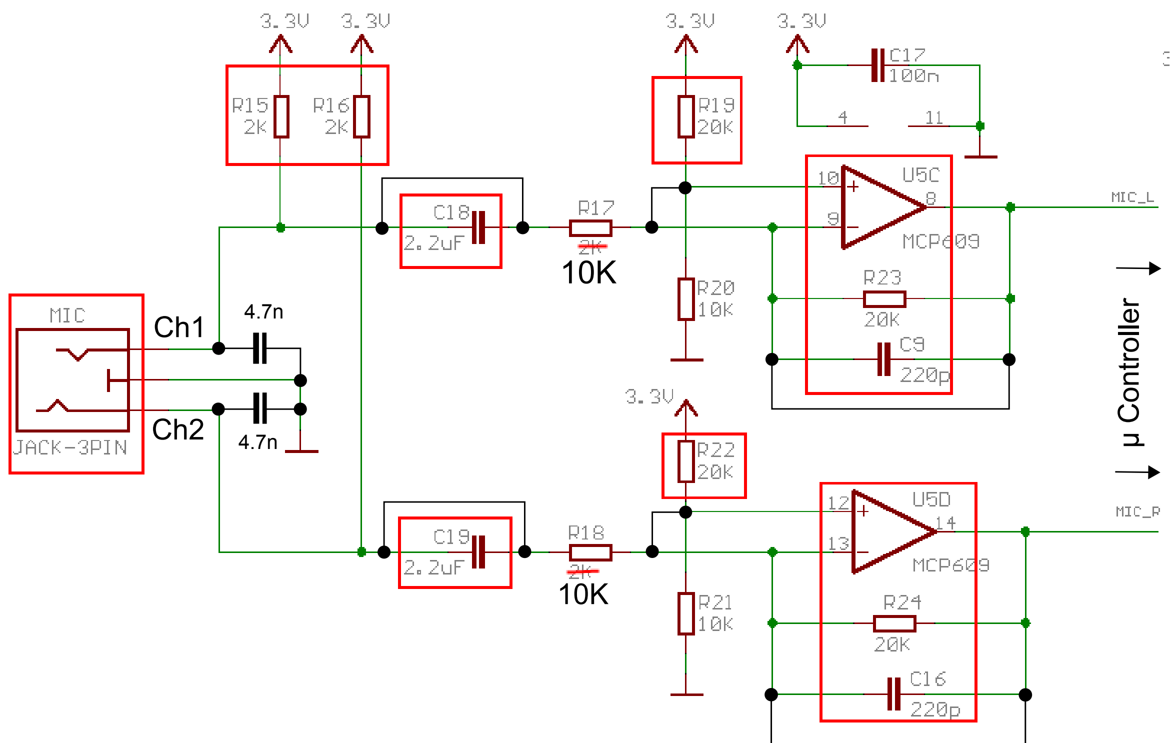


Abbildung 3.2.: Änderungen der Strommessung am Developmentboard

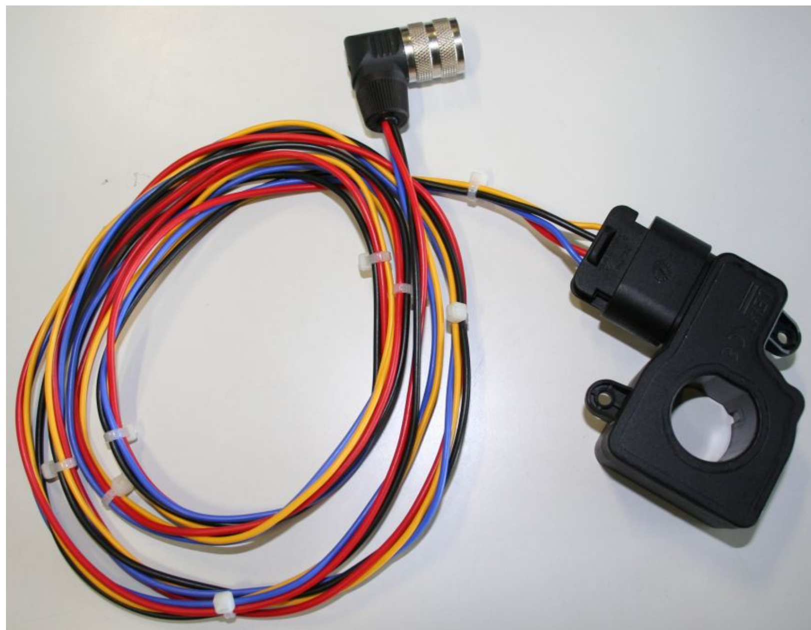


Abbildung 3.3.: Verwendeter Hall-Sensor mit Anschlusskabel

3.1.2. Modifikation der Starterbatterie und Konstruktion von Messaufbauten

Für den Betrieb der vorhandenen Zellsensoren an der Starterbatterie müssen die Zellanschlüsse herausgeführt werden. Dies erfordert das Öffnen der Batterie. Ihren generellen Aufbau zeigte bereits Abb. 1.5b. Die beiden Plattensätze jeder Zelle sind über massive Bleischienen zellenweise vergossen. An ihnen ist die jeweilige Zellspannung zusammengeführt. Das Batteriegehäuse ist meist aus Polypropylen, einem weichen Kunststoff, gefertigt. Jede Zelle ist dabei einzeln verschweißt, sodass das Elektrolyt sich nicht ungleichmäßig über die Zellen verteilen kann. Die einzige Verbindung ist über die Zentralentgasung im Deckel der Batterie, damit stets Druckausgleich herrscht.
Für die Erprobung der Zellsensorik standen zwei Starterbatterien zur Verfügung (Tabelle 3.2).

Hersteller	Banner	Moll
Typ	588 027 064	600 038 085
Nennkapazität	88Ah	100Ah
Kaltstartstrom (EN)	640A	850A
Gehäusegröße	H8	H8

Tabelle 3.2.: Verwendete Starterbatterien

Die folgende Abbildung 3.4 zeigt die unbefüllte Batterie von Moll im geöffneten Zustand. Hierbei wurden die Zugänge der Zellverbinder freigelegt. Die Entgasungskanäle und Zufüllpropfen im Deckel sollen weiterhin erhalten bleiben. An den Bleischienen in jeder Zelle wurden die Verbindungen nach außen angebracht. Hierfür bietet sich Bleifahnen an, die direkt verlötet werden können und beständig in Kontakt mit Batteriesäure sind. Die Anschlüsse für die Sensoren sind am negativen Pol pro Zelle je mit einem Lötstift versehen. Mit der passenden Aussparung an einer Kontaktfläche jedes Sensors kann dieser nicht falsch verpolt montiert werden und verhindert somit seine Beschädigung.

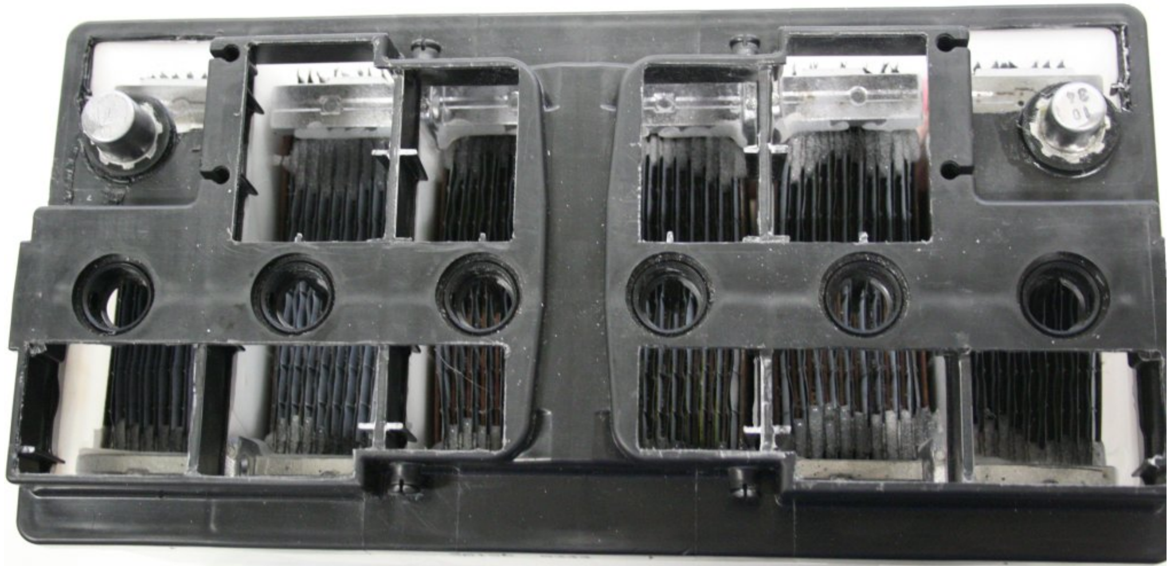


Abbildung 3.4.: Geöffnete Starterbatterie von Moll

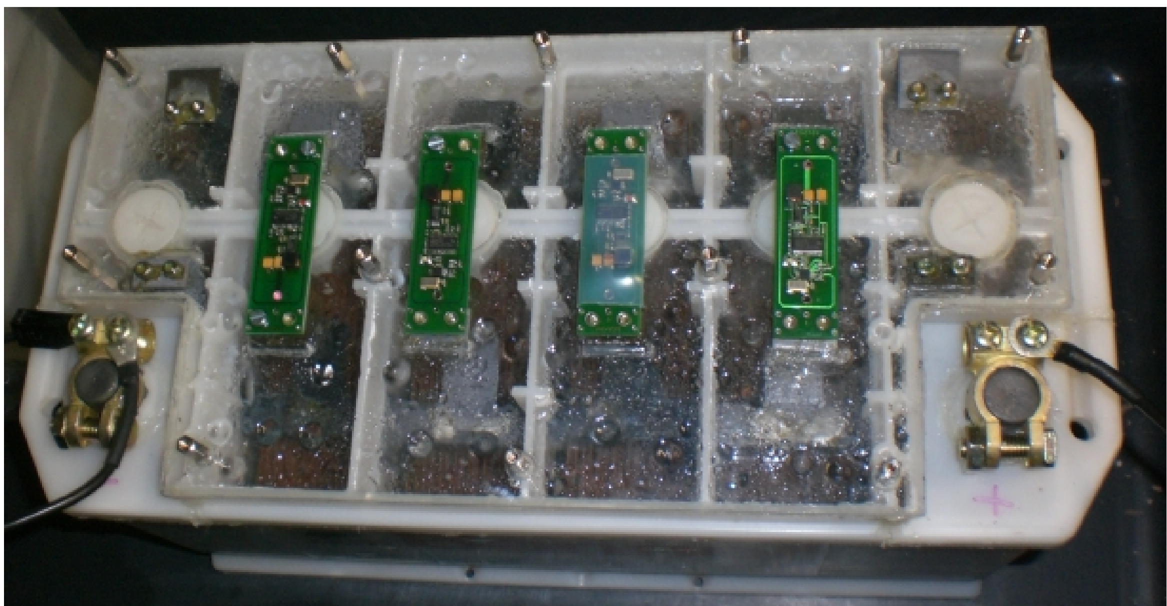


Abbildung 3.5.: Modifizierte Starterbatterie mit Zellsensoren

Die fertig modifizierte Starterbatterie von Banner ist in Abb. 3.5 dargestellt. Sie ist bereits befüllt und mit Sensoren ausgestattet. Die Batterie wurde wieder mit einem Deckel aus Plexiglas geschlossen. Die Zellanschlüsse wurden dabei durch Schlitze im Deckel herausgeführt und mit Abstandsbolzen mit dem Deckel verschraubt. Die Nähe zur Batteriesäure lässt

die verzinkten Abstandsbolzen schnell korrodieren, weswegen sie zusätzlich verlötet wurden, um den Kontakt zur Zelle sicherzustellen. Als zusätzlichen Korrosionsschutz kommt hier Polfett zum Einsatz. Auf den Abstandsbolzen lassen sich die Sensoren verschrauben. Als Dichtmittel eignetet sich Heißkleber, der sich als besonders säurebeständig erwiesen hat.
Für die Anbindung der Oszilloskope bei der Hochstrommessung wurden entsprechende Adapter-Anschlüsse konstruiert (Abb. 3.6). Sie beinhalten je zwei BNC Anschlüsse[1], die über 50Ω Schutzwiderstände mit der Zelle verbunden sind. Im Fall eventueller Kurzschlüsse schützen die Eingänge der Oszilloskope vor Beschädigungen (siehe Abb. 2.2). Für die beiden Zellen höherer Potentiale ist zusätzlich ein Spannungsteiler angebracht. Über die Lötstifte ist der Massenbezug anzuschließen.

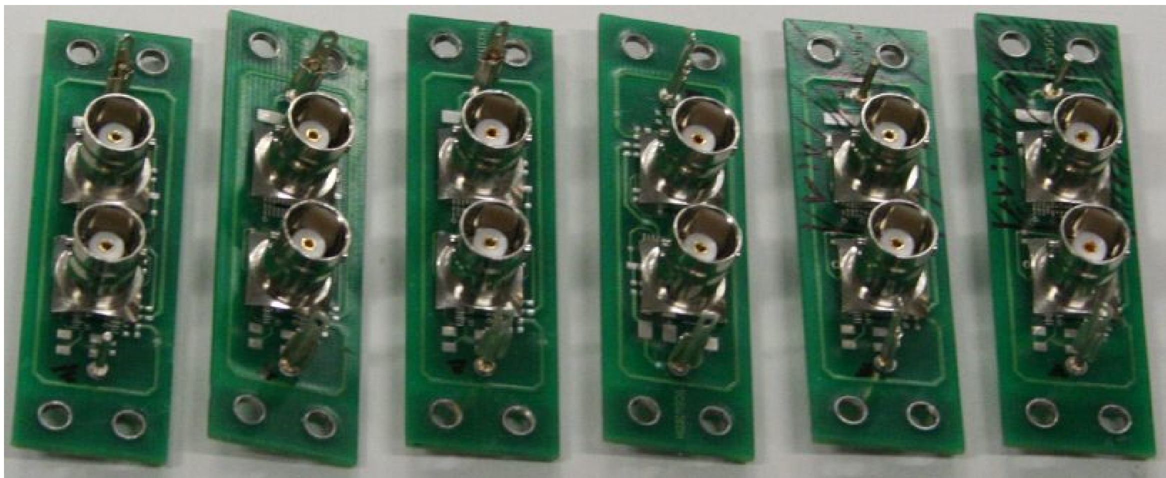


Abbildung 3.6.: Messadapter für Hochstrommessungen mit dem Oszilloskop

[1] Messungen mit Differenzverstärkern benötigen einen zusätzlichen Anschluss pro Zelle.

3.2. Verfahrensimplementation

Das in Abschnitt 2.4 vorgestellte Konzept soll hier implementiert werden und konzentriert sich dabei auf den Sensor. Für die ersten Schritte der Konzepterprobung genügt der Datenlogger vorerst als Schnittstelle zum PC und soll noch nicht die Aufgabe der weiteren Auswertung der Datenframes übernehmen. Dabei ist nur die vorhandene Software an den entsprechenden Stellen anzupassen, um die Daten an den PC weiterzureichen.

3.2.1. Sensor

Das in Abbildung 2.17 dargestellte Schichtenmodell zeigt bereits die im Sensor ablaufenden Instanzen. Das Kernstück des Verfahrens ist die Realisierung des Datenpuffers, im folgenden Queue genannt. In der Queue häufen sich, im Zuge der langsamen Datenrate bei der Übertragung und der schnelleren Aufnahme von Messwerten, bei Hochstromereignissen sehr schnell viele Werte an. In der Queue muss, im Sinne des Verfahrens, pro Messwert zusätzlich eine Zeitmarke ablegt werden, um später die Zeitbasis wiederherzustellen. Der Messwert an sich benötigt zwei Byte (Summe über 10 Messwerte bei 10 Bit Auflösung). Die Größe der Zeitmarke richtet sich nach der maximalen Zeitdifferenz zwischen zwei empfangenen Frames. Läuft der Zeitstempel dort mehr als ein Mal über, entstehen Fehler bei der Rekonstruktion der Zeitbasis. Dies ist bei der hohen Framefehlerrate nicht unwahrscheinlich. Zwei Byte für den Zeitstempel sind daher angebracht. Pro Messung sind also 4 Byte Speicher nötig. Der auf dem Sensor eingesetzte MSP430x1232 stellt 256 Byte RAM zur Verfügung. Die Größe der Queue ist folglich, je nach Auslastung des Stack, stark begrenzt. Die tatsächliche, maximale Größe des Stack lässt sich schlecht genau schätzen und hängt vom weiteren Programmverlauf ab. Es ist sinnvoll in diesem Zusammenhang Operationen, die den Stack unnötig auslasten, möglichst zu vermeiden. Dazu gehören u.a. viele Funktionsaufrufe und komplexe, mathematische Operationen.
Der Ablauf der Sensorsoftware teilt sich in 2 Zustände auf:

- Messwertaufnahme
- Messwertübertragung

Beide Zustände sind als zeitkritisch zu betrachten und finden daher in der Interrupt Service Routine des Timers statt. Die Messwertübertragung ist an die feste Bitdauer eines Halbbits (Manchester-Codierung) von $100\mu s$ geknüpft. Das Timing der Messwertaufnahme ist dadurch über entsprechende Zählvariablen einstellbar. 10 Aktivierungen des Interrupt-Handlers ergeben demnach das Abtastintervall von 1ms.
In Abb. 3.7 ist der genaue Ablauf der Messwertaufnahme dargestellt. Um Zeit im Interrupt zu sparen, wird der ADC Wert erst im nächsten Aufruf abgeholt.

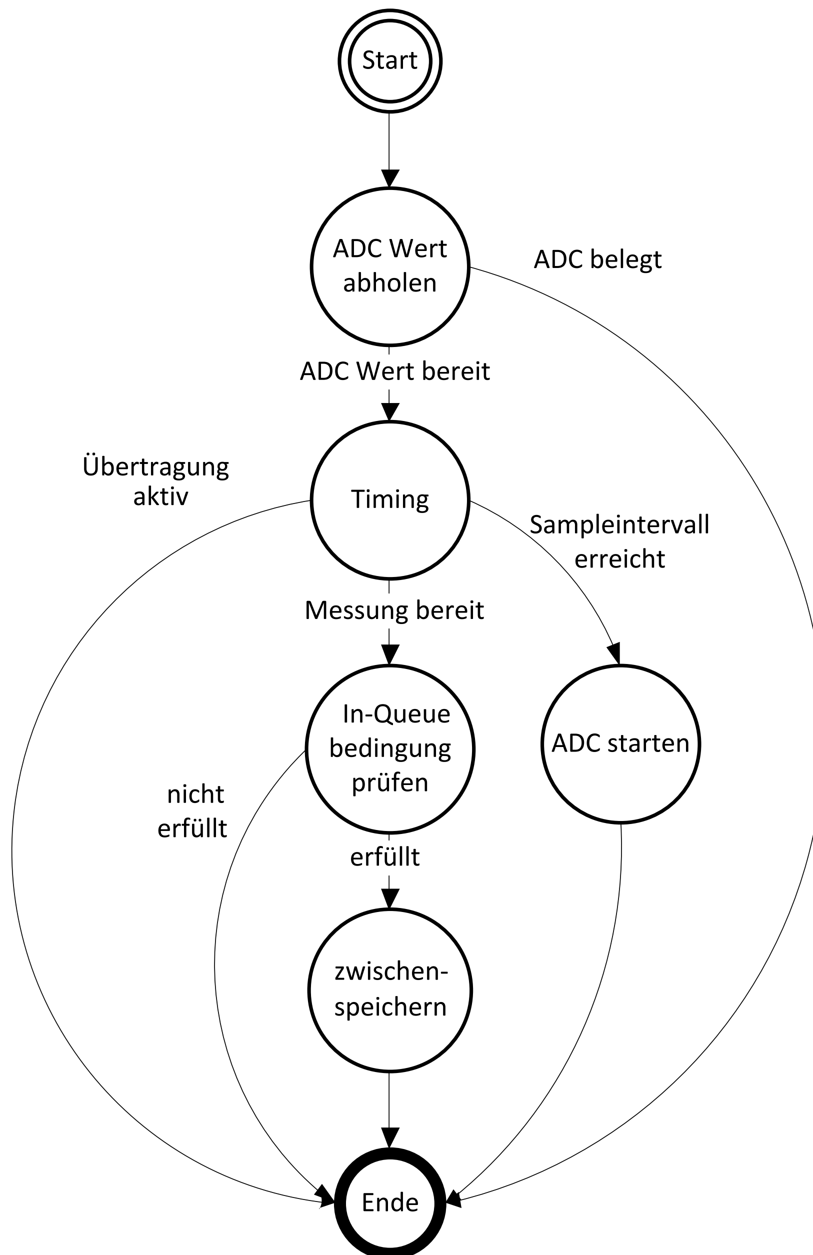


Abbildung 3.7.: Flussdiagramm der Messwertaufnahme

Wie bereits in Abschnitt 2.2.1 gezeigt, ist die Aufnahme eines Messwerts während eines Sendevorgangs nicht möglich. Zu hoch ist in diesem Fall der Energieverbrauch des Transmitters, der sich ohne den Step-Up-Konverter nicht decken lässt. Die übergangsweise Versorgung über einen Kondensator reicht in dieser Zeit nicht aus. In jedem Fall muss dennoch das Generieren der Zeitstempel gewährleistet sein.
Ist das bestimmte Messintervall abgelaufen und die Summe über 10 Messwerte gebildet,

werden die Bedingungen zum Weiterleiten in die Queue geprüft. Treffen die Bedingungen nicht zu, wird das Messinterval verworfen.
Die Queue basiert auf einem Ringspeicher (Abb. 3.8). Auf dem Ring werden durch zwei Indizes Abgänge und Zugänge des Speichers koordiniert. Der linksseitige Abstand von Schreib- zu Leseindex stellt den aktuellen Speicherverbrauch dar. Der Speicher läuft dann über, wenn der rechtsseitige Abstand null ist bzw. der linksseitige Abstand den maximalen Speicherplatz einnimmt. Es wird erst dann wieder ein Platz im Speicher frei, wenn ein Element in den Sendepuffer geladen wird. Andernfalls muss der jeweilige Zugang verworfen werden. Das Entleeren der Queue wird durch den Übertragungsteil bestimmt (Abb. 3.9).

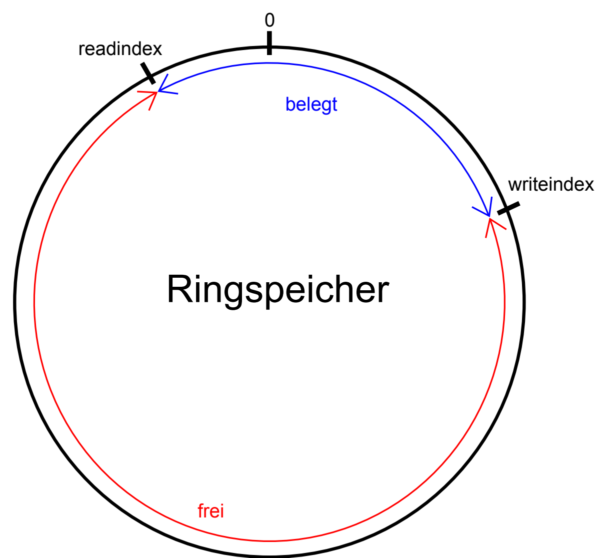


Abbildung 3.8.: Darstellung der Queue als Ringspeicher

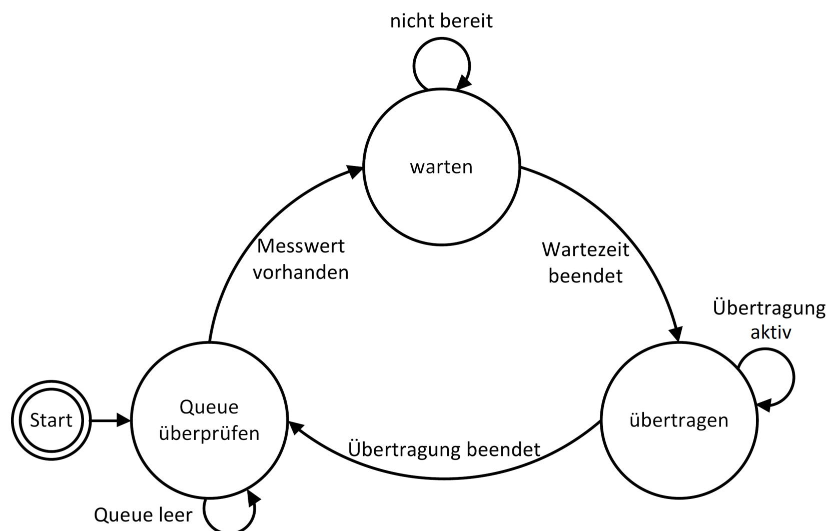


Abbildung 3.9.: Zustandsdiagramme des Übertragungsteils

Der Übertragungsteil enthält wiederum Unterzustände:

Queue überprüfen Dieser Zustand stellt sich solange ein, wie die Queue leer ist bzw. kein Messwert in den Speicher aufgenommen wurde. Vor dem Verlassen des Zustands wird der jeweilige Speicherinhalt in den Sendepuffer geladen und eine zufällige Wartezeit für den nächsten Zustand generiert. Die Wartezeit ist ein Vielfaches des periodisch ausgelösten Interrupt-Handlers des Timers und wird durch ein Zufallszahlengenerator gebildet.

warten Der Wartezustand ist Teil des Übertragungsverfahrens und besteht bis die jeweilige Wartezeit abgelaufen ist.

übertragen Das Übertragen der eigentlichen Information über die Messwerte nach dem in Abb. 1.3.2 dargestellten Protokoll geschieht in diesem Zustand.

Die dargestellten Programmverläufe müssen alle innerhalb des Timer-Interrupt ablaufen. Bei dem bisher verwendeten Takt von 2 MHz zeigen sich bereits Auswirkungen auf das Sendesignal. Die Pulsbreiten des Sendesignals variieren bei 2 MHz schon mit dem unterschiedlichen Rechenaufwand, der während der Timer-ISR auftritt. Abhilfe schafft hier nur die Erhöhung des Taktes auf 3 MHz.
Zu Testzwecken fügt der Sendeteil dem zu sendenden Datenframe zusätzliche Informationen zu. Interessant ist die aktuelle Auslastung der Queue und der jeweilige Framestatus. Demnach kann ein Datenframe entweder einen aktuellen oder älteren Messwert enthalten, der aus dem Zwischenspeicher geladen wird, enthalten. Aktuelle Messwerte und Werte aus dem Zwischenspeicher werden demnach mit einem „Livebit“ gekennzeichnet. Im Fall eines einzigen Bits ist noch genügend Platz im Protokoll vorhanden. Alle anderen Informationen

finden zu Lasten der Messwerte von Temperatur und Betriebspannung Platz. Beide Werte sind eher als zeit-unkritisch zu betrachten und spielen im Moment keine große Rolle. Tabelle 3.3 enthält noch den die zufällig generierte Wartezeit. Hiermit ist es möglich, den tatsächlichen Messzeitpunkt von direkt gesendeten Frames zu bestimmen. Dies ist später eventuell interessant, um die Messwerte noch genauer zu synchronisieren, soll aber im Moment nicht weiter behandelt werden. Der komplette Quelltexte ist im Anhang A.1 zu finden.

Livebit+sonstige	Jitter	Adresse	Zellspng	Timelabel	Queue	CRC
8Bit	16Bit	8 Bit	16 Bit	16 Bit	16Bit	8Bit

Tabelle 3.3.: Änderung des Übertragungsprotokoll für das neue Übertragungsverfahren (ohne RunIn und SOF)

3.2.2. Datenlogger

Die Änderung der Software des Datenloggers beschränkt sich im Wesentlichen auf Optimierungen bezüglich des Zeitverhaltens und des Anpassens der Strommessung auf zwei Messbereiche. Im Verlauf dieser Arbeit wurden einige Funktionen aus den Arbeiten A. Hoops [5] und S. Plaschke [10] modifiziert oder gänzlich entfernt. Dazu gehören:

- Temperaturkalibrierung
- Datenspeicherung der Messwerte pro Minute
- Datenübertragung der Messwerte aus dem Flash-Speicher über die serielle Schnittstelle
- Entprellen der Taster zur Steuerung des Datenloggers
- Darstellung der Strommessung in A/min

Die entsprechenden Änderungen sind im Quelletext unter Anhang A.2 vermerkt.

Strommessung

Die Strommessung mit dem Hall-Effekt-Sensor DHAB/S24 erfordert zwei Messbereiche (vgl. Tab. 3.1). Der auf dem Datenlogger eingesetzte MSP430F169 stellt dazu für die Messwertaufnahme einen 12Bit ADC zur Verfügung. Die Umwandlung der Sensorausgänge erfolgt periodisch mit dem ADC- eigenen Taktgeber von etwa 5MHz. Durch den entsprechend gewählten Taktteiler ist die Aufnahme von Messwerten alle 1,03 ms möglich. Die Umwandlung der ADC-Eingänge geschieht parallel im repeated sequenzed mode des ADCs und endet

mit dem Auslösen des ADC Interrupt-Handlers. In seiner ISR muss über die Wahl des Messbereichs entschieden werden.
Die Wahl des Messbereichs wird anhand des empfindlicheren Messbereichs 1 bestimmt. Wird dieser Messbereich überschritten, ist der Messbereich 2 zu benutzten. Um Rechenzeit in der ISR zu sparen, wurden die entsprechenden ADC Werte bereits umgerechnet. Tab 3.4 zeigt die berechneten Umschaltschwellen der Messbereiche:

max. Sensor Ausgang	ADC Wert[2]	Stromwert
4,369V	3578	+70
0,631V	517	-70

Tabelle 3.4.: Umrechnung der Schwellwerte zur Wahl des Messbereichs

Zur Vermeidung von Schwankungen zwischen den Messbereichen an dient eine entsprechende Schalthysterese über etwa 500mA.
Erst nach der Wahl des Messbereichs wird der Messwert entsprechend skaliert. Die Erfassung des Stroms sieht zusätzlich den Abgleich des Nullwertes vor und beinhaltet demnach neben der normalen Messwertaufnahme einen Tariermodus. Zu diesem Zweck wird über 20 Messwerte gemittelt. Auf Basis dieses Nullwertes kann der tatsächliche Strom berechnet werden.

3.3. Erprobung

Das behandelte Konzept des Zwischenpufferns von Messwerten zur Angleichung der Messrate an die Datenrate wurde im letzten Abschnitt 3.2 implementiert. Die Simulation des Zwischenspeichers half bei der Parameter-Findung für die In-Queue-Bedingungen und die dazu nötige Größe des Speichers. Im Folgenden sollen die aufgezeigten Schritte direkt am Sensor erprobt werden. Bevor das System am Automobil betrieben wird, ist die vorherige Überprüfung auf Funktionalität im Labor sinnvoll.

3.3.1. Laborbetrieb

Der praktische Betrieb des Konzepts am Sensor ist zuvor unter Laborbedingungen zu prüfen. Das Verhalten des Sensors gibt so Aufschluss über eventuelle Programmfehler und ist besonders für die Überprüfung der Parameter für den Zwischenspeicher wichtig, um einen Stack-Überlauf zu vermeiden.

[2] errechnet nach $A_{ADC} = V_{out} \frac{4095}{2 \cdot 2{,}5V}$

Der Laborbetrieb des Systems setzt wieder auf die aufgenommenen Startvorgänge auf. Diese können durch einen arbiträren Signalgenerator beliebig ausgegeben werden. Zum Treiben der Sensoren wird zusätzlich ein Leistungsverstärker benötigt. Das Einbeziehen der Fehlerrate in die Ergebnisse wird durch das Parallelschalten der Sensoren erreicht Abb. 3.10 zeigt schematisch den Aufbau der Laborerprobung.

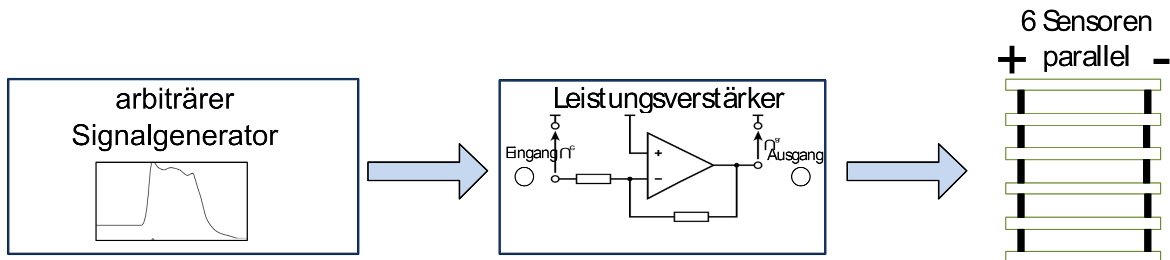


Abbildung 3.10.: Schematischer Aufbau bei der Laborerprobung

Die Aufnahme der Startvorgänge konnte mit hoher, zeitlicher Auflösung durchgeführt werden. Dem Signalgenerator ist es jedoch nur möglich bis zu etwa 130.000 Werte aufzunehmen. Dies erfordert das entsprechende Dezimieren der Messdaten. Das Signal wird dadurch störungsfreier und gilt somit nicht als vollwertiger Ersatz zu einer Erprobung am Kfz.
Abb.fig:messqueueperf zeigt die gemessene Auslastung des Zwischenspeichers im Sensor. Es zeigt sich sehr die Ähnlichkeit im Vergleich zu der Simulation in Abb. 2.19 und bestätigt die erfolgreiche Implementierung des Konzepts aus Abschnitt 1.10. Das Ergebnis ist noch nicht als ideal zu betrachten. Die Messung durch den Sensor zeigt bei genauerer Betrachtung zu Beginn einen geringen Spannungsfehler. Das Signal wurde hierbei der tatsächlichen Spannung durch einen Offset angenähert und durch Verschieben auf der Zeitachse am tatsächlichen Verlauf ausgerichtet. Der tiefste Spannungseinbruch der Sensormesswerte diente dabei als Bezugspunkt . Die genauere Messung, erfordert die noch nicht Implementierte Kalibration der Spannungsmessung. Ebenso ist beim Betrieb mehrerer Sensoren der Datenverlust durch Übertragungsfehler immer noch zu erwarten. Treten solche Fehler bei steileren Signalverläufen auf, entstehen große Abweichungen zum tatsächlichen Zellsignal. Solch einen Fall zeigt Abb. 3.12. Bei 6 verwendeten Sensoren fallen hier deutlich fehlende Messwerte auf.

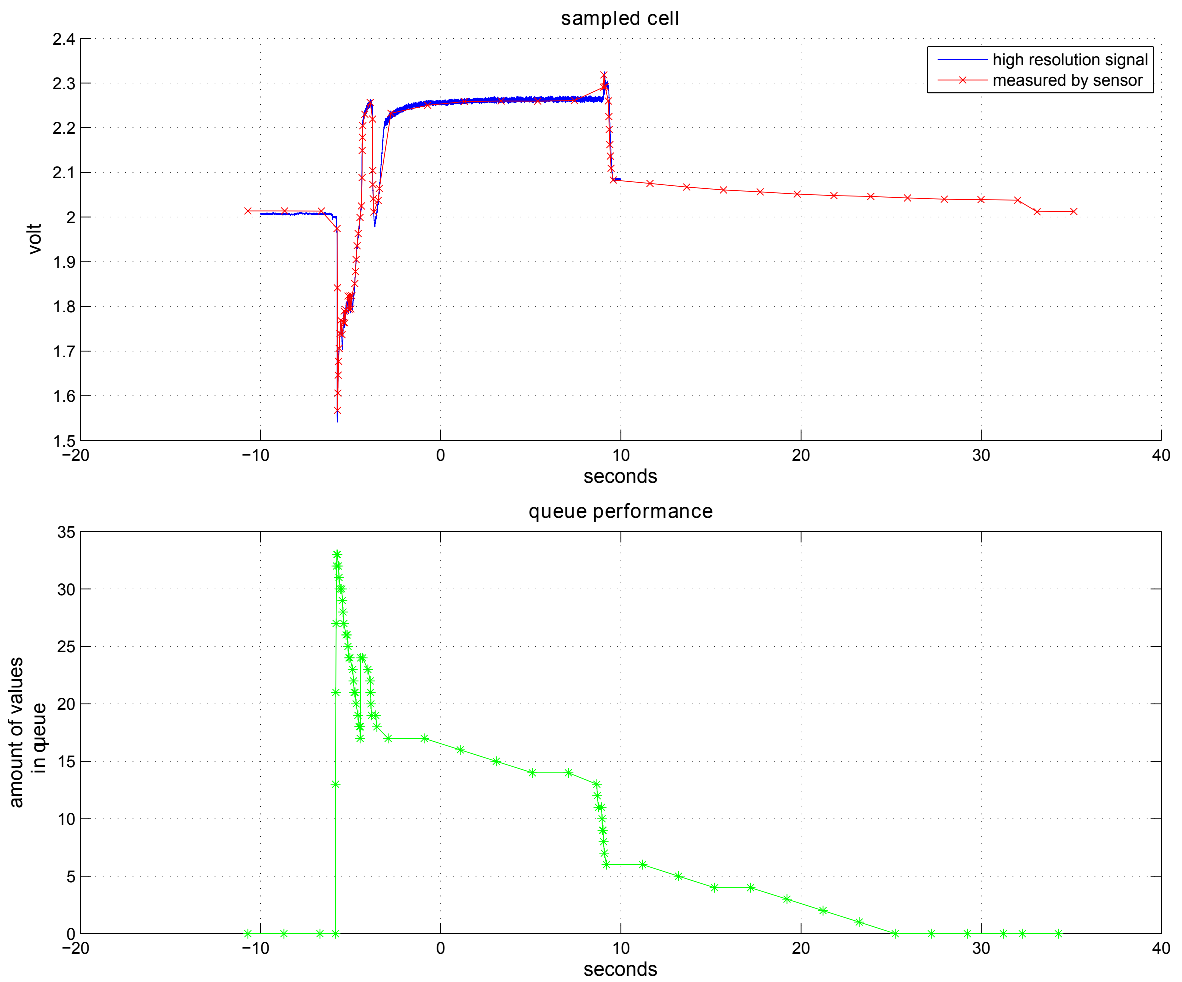


Abbildung 3.11.: Gemessenes Zeitverhalten der Zwischenspeichers

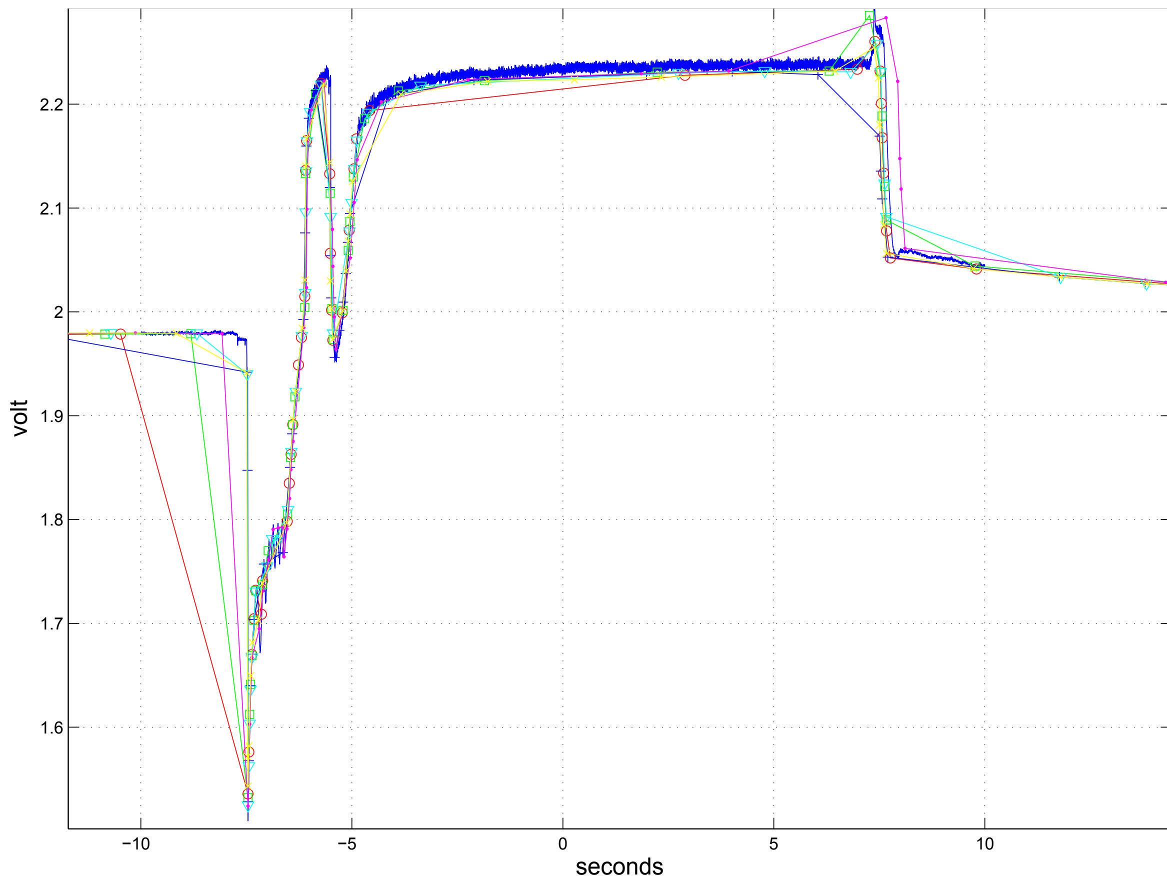


Abbildung 3.12.: Erprobung am Signalgenerator mit 6 Sensoren parallel

3.3.2. Betrieb am Kfz

Die Erprobung am Kfz bietet im Gegensatz zur Laborerprobung mit arbiträrem Signalgenerator viel realitätsnähere Bedingungen. Das Zellsignal enthält wieder alle Störungen und Schmutzeffekte. Jeder Sensor wird zudem an einer unterschiedlichen Zellspannung betrieben. Dies zeigt, inwiefern die Messdaten der Sensoren auseinander laufen können.
Zum Vergleich der Signaltreue der mit den Sensoren aufgenommenen Signalverläufe bietet sich wieder die Parallelmessung mit den Oszilloskopen an. Zu diesem Zweck wird der gleiche Aufbau wie aus Abschnitt 2.1.2 benutzt. Die verwendeten Messadapter lassen sich hierbei über Abstandsbolzen parallel zu den Sensoren auf der Starterbatterie anbringen. Zum Empfang der Sensordaten dient der Datenlogger. Die Messwerte werden in diesem Fall direkt über die serielle Schnittstelle an den PC weitergereicht. Start- und Messvorgang der Sensoren lassen sich bei der Durchführung nicht miteinander synchronisieren. Der Start des Datenempfangs durch den PC wird per Hand abgepasst. Es entsteht somit ein fester Zeitversatz beider Messungen. Die Sensordaten müssen später den Daten der Oszilloskope

angeglichen werden. Abbildung 3.13 zeigt ein Bild des Messaufbaus in der Fahrzeughalle der Hochschule für Angewandte Wissenschaften Hamburg.

Abbildung 3.13.: Praktischer Aufbau der Erprobung in der Fahrzeughalle

Die Messergebnisse 3.14 - 3.17 sind die Ergebnisse der Erprobung. Sie zeigen jeweils die Messwerte der 6 Sensoren und aus Gründen der Übersichtlichkeit nur Zelle 1, gemessen mit dem Oszilloskop. Sensor und Oszilloskop-Messung wurden wieder anhand des minimalen Sensormesswertes ausgerichtet. Erprobt wurde das System am Audi A4 und am VW Passat.
Abb. 3.14 zeigt die Messungen am Audi A4 über 20s. Die Sensor-Werte steigen am Ende der Oszilloskop-Werte, nach 20s, leicht an, bevor sie schlagartig abfallen. In diesem Fall liefen die Sensoren etwas länger und erfassten zusätzlich den Überschwinger beim Abschalten des Motors. Generell ist eine geringe Abweichung der Messgenauigkeit zu erkennen. Gerade Sensor 3 zeigt deutliche Abweichungen zum Oszilloskop. Hier sei nochmals zu erwähnen, dass zum Vergleich nur Zelle 1 dargestellt wurde. Die Zellspannungen sind generell unterschiedlich, sodass der Fehler bei Zelle 3 im Vergleich nicht zu schwerwiegend eingeschätzt werden sollte. Bei den verwendeten wurde bisher jedoch noch keine Kalibrierung der Spannungsmessung durchgeführt, weswegen kleinere Abweichungen nicht überraschen.
Die Entscheidung über ein Hochstromereignis wird stets im Vergleich zum zuletzt in die

Queue aufgenommenen Wert gefällt. Dies hat zur Folge, dass steile Flanken im Zellsignal bei weit auseinanderliegenden Queue-Werten größere Fehler in der Signaltreue verursachen. Hinzu kommt noch die Verschlechterung durch Übertragungsfehler in diesem Moment. Abb. 3.15 zeigt dazu den Startzeitpunkt in hineingezoomter Ansicht. Es fällt hier auf, dass die Messwerte der Sensoren nicht exakt auf das tatsächliche Zellsignal ausgerichtet wurden. Die Zeitbasis der Messung weicht zum Teil noch sehr vom ideal ab. Die Rekonstruktion der Messintervalle nach der Zwischenspeicherung in der Queue ist mit dem Taktfehler des Sensors behaftet, der sich über die Zeit durch die verwendeten Zeitstempel aufaddiert. Der interne DCO des Sensors schwankt über die Zeit z.T. um etwa 2-3% und hängt von Temperatur und Versorgungsspannung ab. Dies fällt besonders bei Abb. 3.16 auf. Der Verlauf der Sensor-Messwerte fällt hier nach dem Abschalten des Motors etwas später ab. Der Fehler bleibt dabei jedoch im Bereich von Millisekunden. Abb. 3.17 zeigt besonders den Fall von hohen Übertragungsverlusten, da hier über eine größere Zeitspanne die Zellsignale stark schwanken . In diesem Fall wurde der VW Passat innerhalb von 40 Sekunden dreimal gestartet. Auf die kurzen Ladephasen zwischen den Starts entfallen zu wenig Messwerte, sodass größere Teile des Integrals verloren gehen.

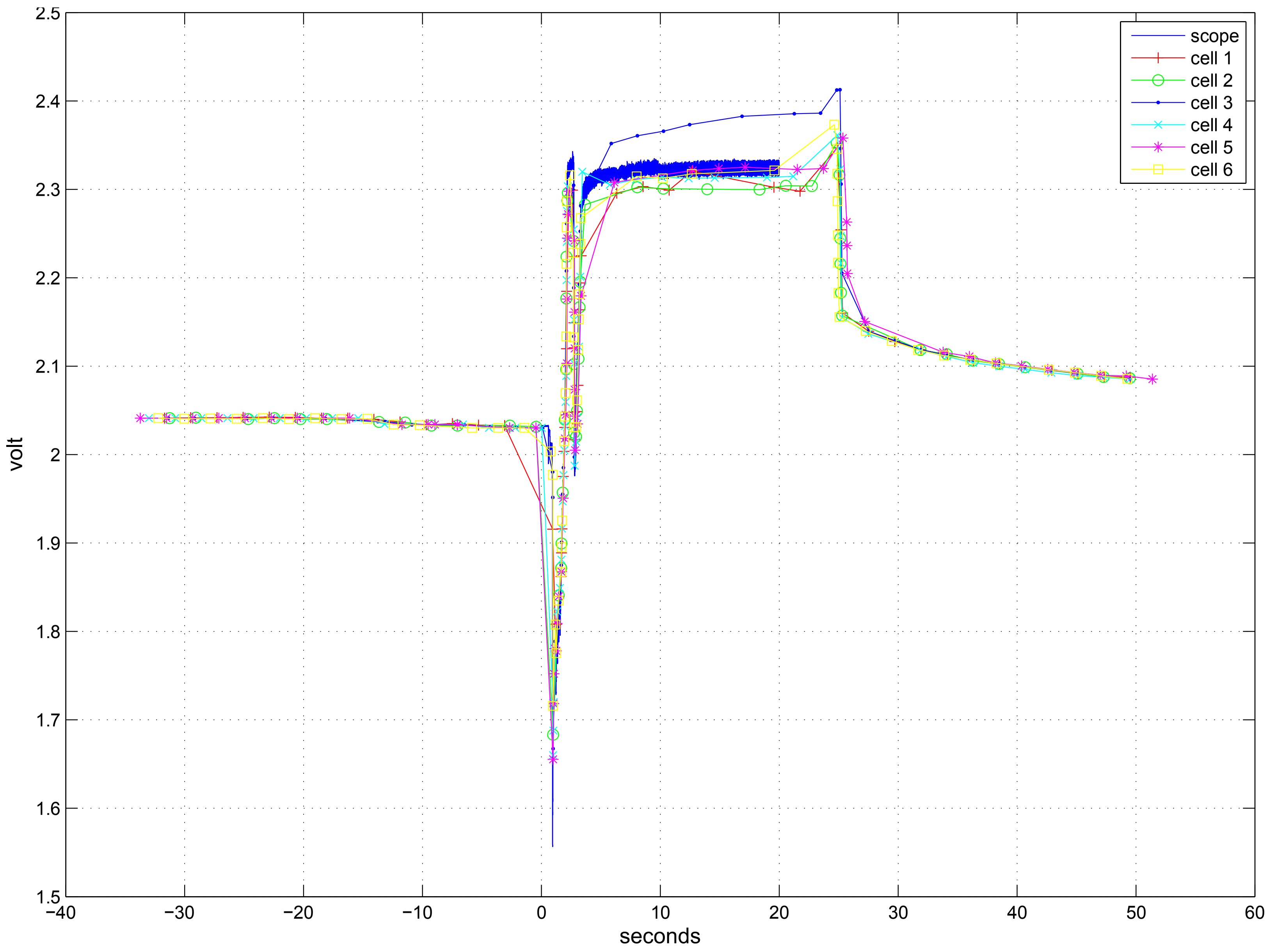


Abbildung 3.14.: Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 20s

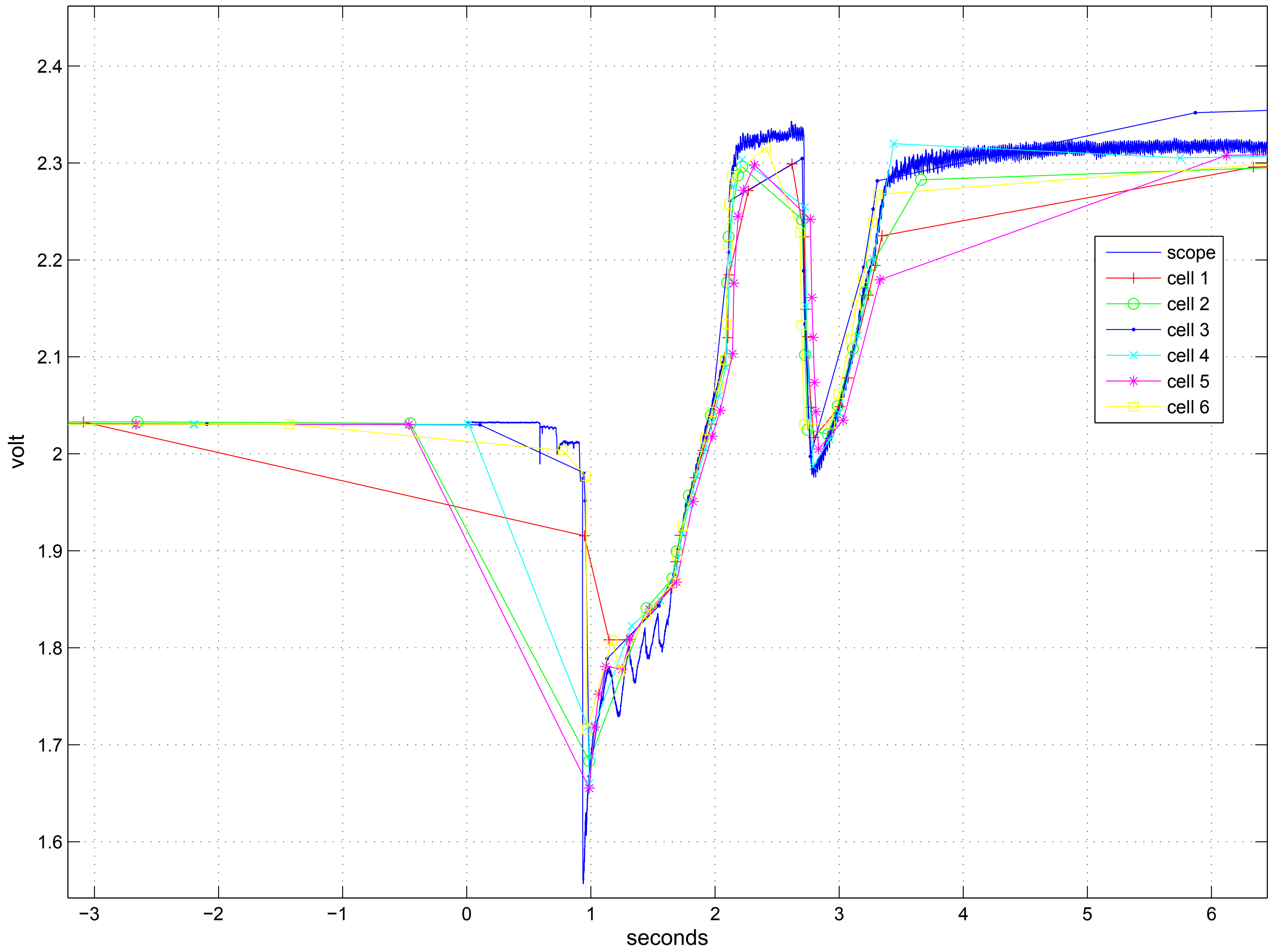


Abbildung 3.15.: Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 20s, gezoomt

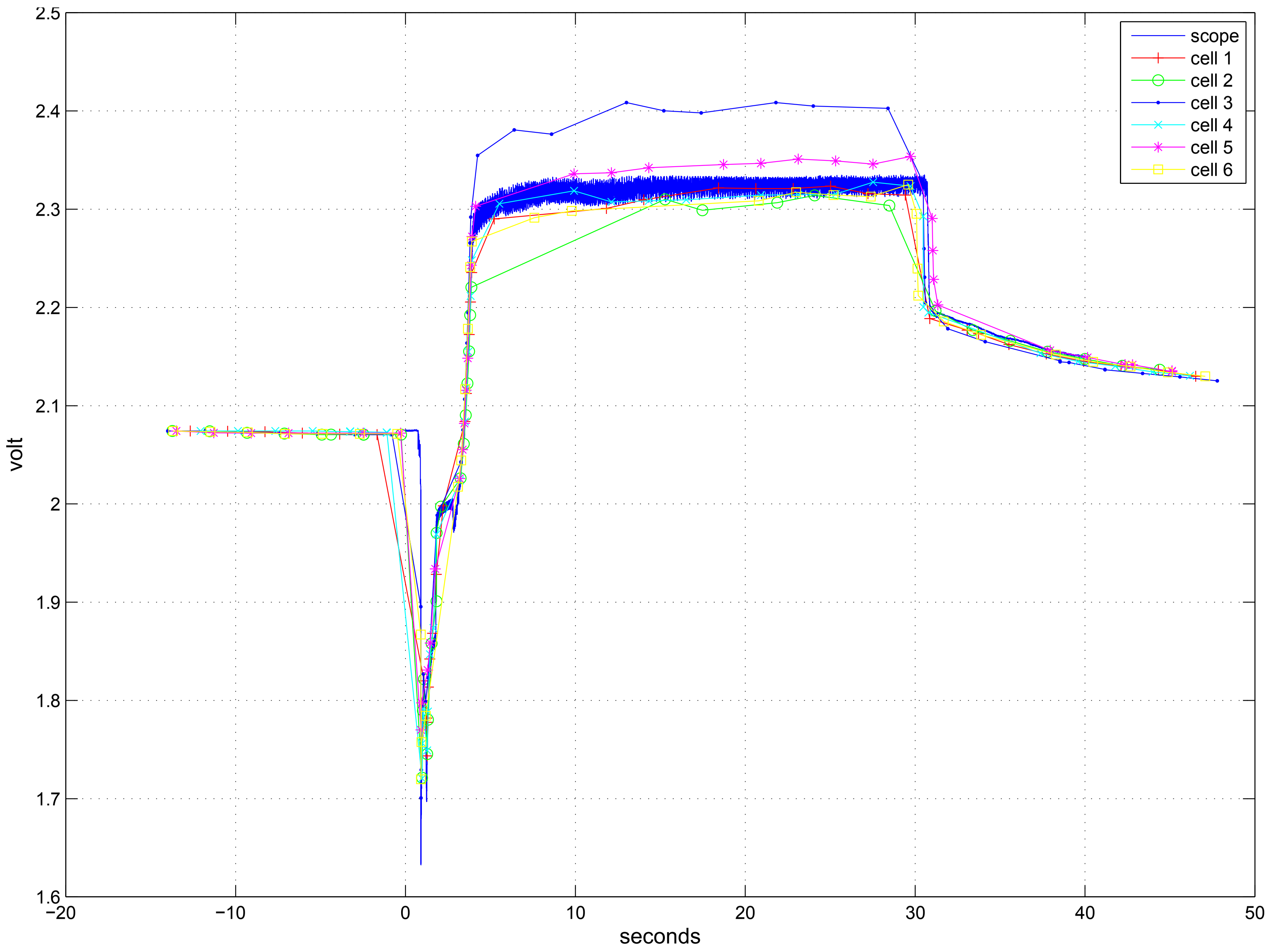


Abbildung 3.16.: Sensormessdaten beim Startvorgang im Vergleich, VW Passat über 40s

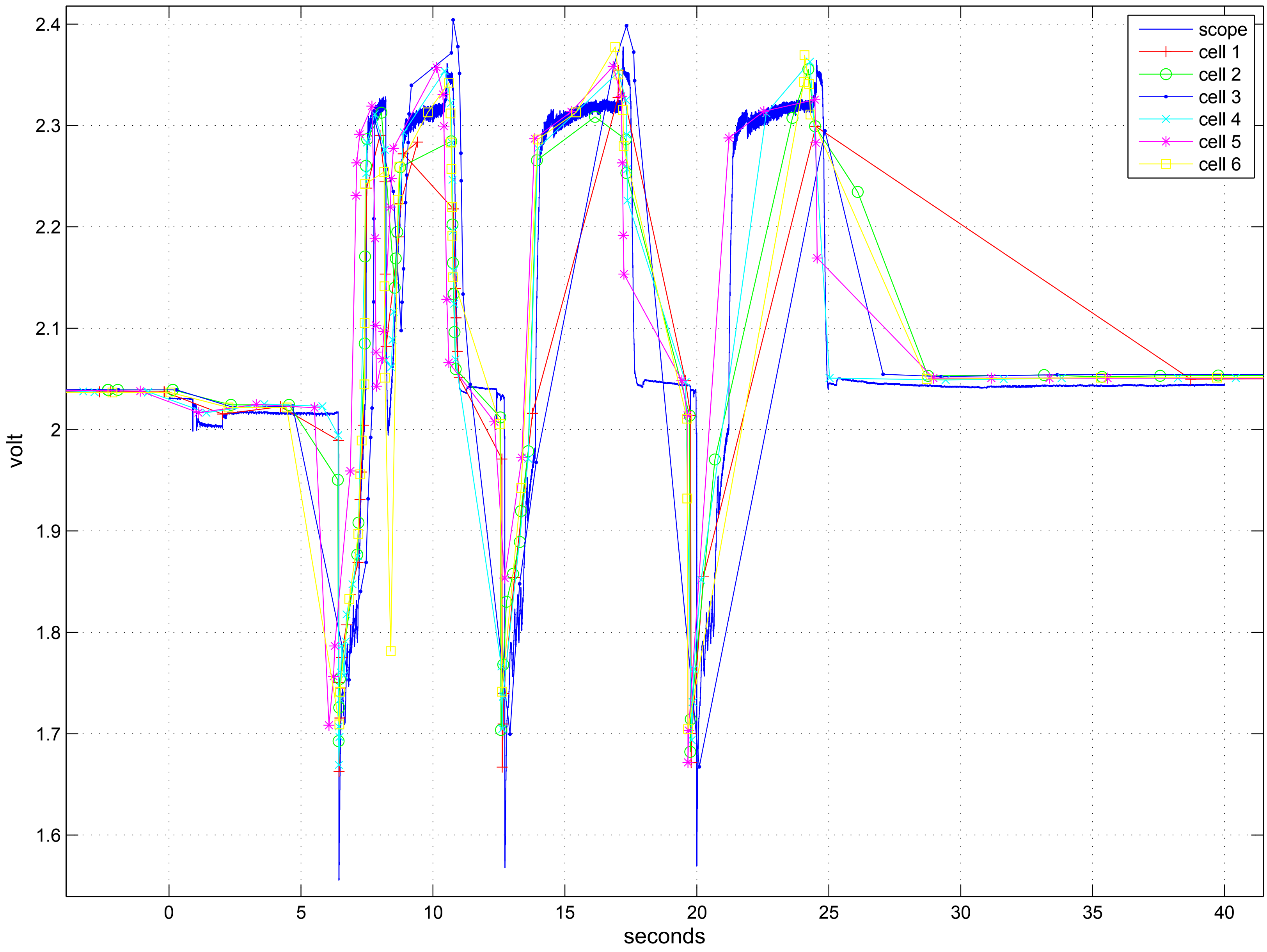


Abbildung 3.17.: Sensormessdaten beim Startvorgang im Vergleich, Audi A4 über 40s, 3-fach-Start

4. Auswertung und Fazit

Zum Abschluss dieser Arbeit sollen in diesem Kapitel, nach einer kurzen Zusammenfassung, erzielte Ergebnisse ausgewertet und im Gesamtüberblick dargestellt werden. Zuletzt wird daraus das Fazit abgeleitet.

4.1. Zusammenfassung

Am Anfang der vorliegenden Diplomarbeit stand ein drahtloses Sensorsystem, das im Rahmen vorangegangener Diplomarbeiten weiterentwickelt wurde. Das System zeigte bei der Anwendung auf Traktionsbatterien für Gabelstapler bereits seine Funktionalität. Umstand dieser Diplomarbeit war es, die Ausweitung der Anwendung auf Starterbatterien im Automotive-Bereich zu untersuchen. Das übernommene Sensorsystem sollte diesbezüglich entsprechend angepasst und erprobt werden.
Am Anfang der Untersuchungen stellte sich zunächst die Frage an die Geschwindigkeit der Datenübertragung. Um eine Aussage über die genauen, zeitlichen Anforderungen der Sensorik beim Starten von Kfz-Motoren zu erhalten war es wichtig, eine genaue Referenz zu erlangen. Zu diesem Zweck bot sich die hoch aufgelöste Aufnahme jeder Zellspannungen mittels Oszilloskopen an. Das praktische Messen an der Starterbatterie erforderte die Konstruktion entsprechender Messaufbauten und das Modifizieren der Batterie selbst. Um den Abgriff einzelner Zellspannungen zu ermöglichen, musste dazu die Batterie geöffnet werden. Nach dem Anbringen von Kontaktfahnen an jeder Zelle, ließ sich die Batterie anschließend mit einer eigens konstruierten Deckelplatte wieder verschließen, um das Austreten von Säure zu verhindern.
Die Ergebnisse der Batteriemessung ergaben in etwa eine Startzeit von 0,8 - 1,2s auf Basis von 3 verschiedenen Automobilen. Der Spannungsverlauf zeigte zudem das zu erwartende Störverhalten während und nach dem Start des Motors, wonach der Sensor entsprechend auszulegen war. Auf Basis der Zellmessungen konnten dann entsprechende Anforderungen an die Datenrate gestellt werden. Die bereits verwendete Übertragung eines Messwertes pro Zelle in einer Minute reichte nicht aus, um das kurze Hochstromereignis beim Start zu erfassen. Entsprechende Reserven des Systems waren jedoch zu erwarten, sodass die höchst mögliche Datenrate zu bestimmen war.
Die Datenübertragung erfolgt durch ein total asynchrones, statistisches Sendeverfahren,

ähnlich dem ALOHA Prinzip. Die Tatsache, dass jeder Sensor keine Daten empfangen kann, erfordert dabei das zufällige Senden eines Datenpakets innerhalb eines kurzen Zeitfensters. Maßgeblich für die Übertragungsgeschwindigkeit ist dabei die, aufgrund von Kollisionen auf dem Kanal, entstehende Framefehlerrate. Theoretische Untersuchungen des Sendeverfahrens und den damit verbundenen Anforderung an die Messgenauigkeit zeigten erste Ansätze zur Auslegung des Systems. Die errechnete Fehlerrate galt dabei als zu pessimistisch, da sie vom Senden eines jeden Sensors im selben Zeitfenster ausging. Bei dieser Betrachtung ergibt sich ein Worst-Case Fall extremster Überlagerung. Der Abgleich mit praktischen Messungen war dazu erforderlich. Die Messergebnisse sollten diesbezüglich ohne Einfluss sonstiger Faktoren aufgenommen werden. Demnach ergab sich die Notwendigkeit der Überprüfung der Hardware auf uneingeschränkte Funktionalität.
Die Darstellung des Zeitverhaltens vom zurückgewonnenen Digitalsignal am Empfänger zeigte ein hohes Maß an Störungen. Es stellte sich heraus, dass die Empfangseinheit an das Sendesignal falsch angepasst war. Das Ändern der Tiefpassfilterung und der Empfindlichkeit der Entscheiderstufe im Empfänger entstörten das zurückgewonnene Digitalsignal. Anschließende Messungen der Framefehlerrate ergaben im Gegensatz zum theoretischen Worst-Case-Fall von etwa 70% eine tatsächliche Framefehlerrate von 41,3%.
Durch die vorangegangenen Erkenntnisse über Daten- und Fehlerrate ließen sich die Anforderungen an die Messgenauigkeit weiter untersuchen. Das Sendeintervall war im Vergleich zur tatsächlichen Dauer des Hochstromereignisses zu gering. Das direkte Übertragen eines Messwerts erreichte somit keine hinreichend genaue Messgenauigkeit. Eine Möglichkeit viele Messpunkte zu übertragen bietet in diesem Zusammenhang die Vorverarbeitung im Sensor. Durch Quellkodierung, d.h. Datenkompression (Mittelung über 10ms) und Angleichung der Messdatenrate über einen Zwischenspeicher ermöglichen eine viel genauere zeitliche Auflösung der Zellspannungen. Das Konzept ließ sich durch die Simulation am PC, hinsichtlich der zur Verfügung stehenden Mittel, überprüfen. Der geringe Arbeitsspeicher des Mikrocontrollers auf dem Sensor stellt dabei die Grenze dar. Die simulativ erprobten Parameter wurden für die Implementierung im Sensor genutzt und im Labor mit einem arbiträren Signalgenerator überprüft. Schlussendlich ließ sich die Erprobung am Fahrzeug, parallel zur bereits durchgeführten Batteriemessung durchführen und zeigte weiterführende Ergebnisse.

4.2. Bewertung erzielter Ergebnisse

Ziel dieser Arbeit war die Untersuchung und Anpassung des zuvor entwickelten, drahtlosen Sensorkonzepts auf Automobil-Starterbatterien. Die drahtlos-Zellsensorik befindet sich wie zuvor in einem starken Entwicklungsprozess und erlaubte bisher das Erfassen der für eine Ladezustandsbestimmung nötigen Messwerte von Spannung, Temperatur und Strom bei Traktionsbatterien für Gabelstapler. Die hier verwendete Sensorik basiert auf drahtlosen

Sensoren ohne jegliche Synchronisation durch die Möglichkeit des Datenempfangs einer zentralen Steuereinheit. Das dadurch eingesetzte, statistische Sendeverfahren lässt schnellere Datenübertragungen nur in Verbindung mit hohen Fehlerraten zu. Messungen an der Starterbatterie zeigten jedoch sehr genau die Anforderungen an die Sensorik in Hinblick auf die erforderliche Messgenauigkeit. Gerade die kurze Hochstromphase erfordert dafür höhere Übertragungsgeschwindigkeiten, die von dem hier eingesetztem System nicht unterstützt werden können. Die Erprobung des im Laufe dieser Arbeit entwickelten Übertragungskonzepts zeigte jedoch, dass trotz der geringen Datenrate eine bereits gute Erfassung der Zellsignale möglich ist. Durch die Verwendung einer verlustbehafteten Quellkodierung im Sensor wurde die nötige Datenkompression erreicht. Messwerte mit besonderer Bedeutung werden dabei weiterhin mit hinreichender Genauigkeit übertragen. Als Gütefaktor für die zeitliche Messauflösung ergibt sich dadurch die Größe des RAM-Speichers des Mikrocontrollers auf dem Sensor. Das Datenaufkommen wird im Speicher über die notwendige Gesamtübertragungszeit gehalten. Schwächen zeigen sich bei dieser Methode besonders in Hinblick auf eine zeitnahe Beurteilung der Messdaten an der Empfangseinheit. Jeder Sensor benötigt entsprechend der Anzahl zwischengespeicherter Messwerte Zeit für die gesamte Übertragung des letzten Hochstromereignisses. Im Fall der Ladezustandsbestimmung der Starterbatterie ist dies jedoch eher unkritisch. Ist der Zwischenspeicher nur groß genug, werden auch folgende Spannungsschwankungen erfasst und übertragen.

Probleme bereitet das Verfahren bei der Wiederanordnung zuvor im Zwischenspeicher abgelegter Messwerte. Das Hinzufügen einer Zeitmarke zu jedem Messwert reicht nur bedingt aus. Das Generieren der Zeitmarken, sowie alle anderen Aktionen sind dem Takt des Mikrocontrollers auf dem Sensor unterworfen. Der Mikrocontroller bezieht seinen Takt durch seinen internen DCO. Sein Takt läuft somit nicht hinreichend stabil und belegt die erzeugte Zeitmarke mit einem zusätzlichen Fehler. Das Kalibrieren des Sensortaktes ist mit zusätzlichem Aufwand verbunden und erfordert entsprechende Reaktionen auf dem Empfänger. Durch das Messen der Sendeintervalle von unverzögert durch die Queue gesendeter Messwerte wurde bisher ein Korrekturfaktor bei der Auswertung errechnet. Dieser konnte die Abweichung des Taktes weiter verringern, ließ aber eine völlige Korrektur nicht zu. Eine Takt-Kalibration direkt bei der Auswertung im Empfänger wird hier als effektiver gesehen. An dieser Stelle wird auf Kapitel 5 verwiesen, das mögliche Lösungsansätze vorschlägt, die in der Bearbeitungszeit dieser Abschlussarbeit nicht mehr Erprobt werden konnte. Eine korrekte Zeitbasis stellt für das erstellte Verfahren ein wichtiges Kriterium dar. Besonders gilt dies bei der Weiterverwendung der Strommessung, die bisher völlig unsynchronisiert, zentral von dem Datenlogger erfasst wird. Ohne die korrekte zeitliche Anordnung der Messwerte entstehen erhebliche Abweichungen des Zellverlaufs. Spätere Auswertungen erzeugen so große Fehler bei der Ladezustandsbestimmung der Batterie. Den abschließenden Überblick aller auszulegenden Größen der letzten Diskussion zeigt Abb. 4.1.

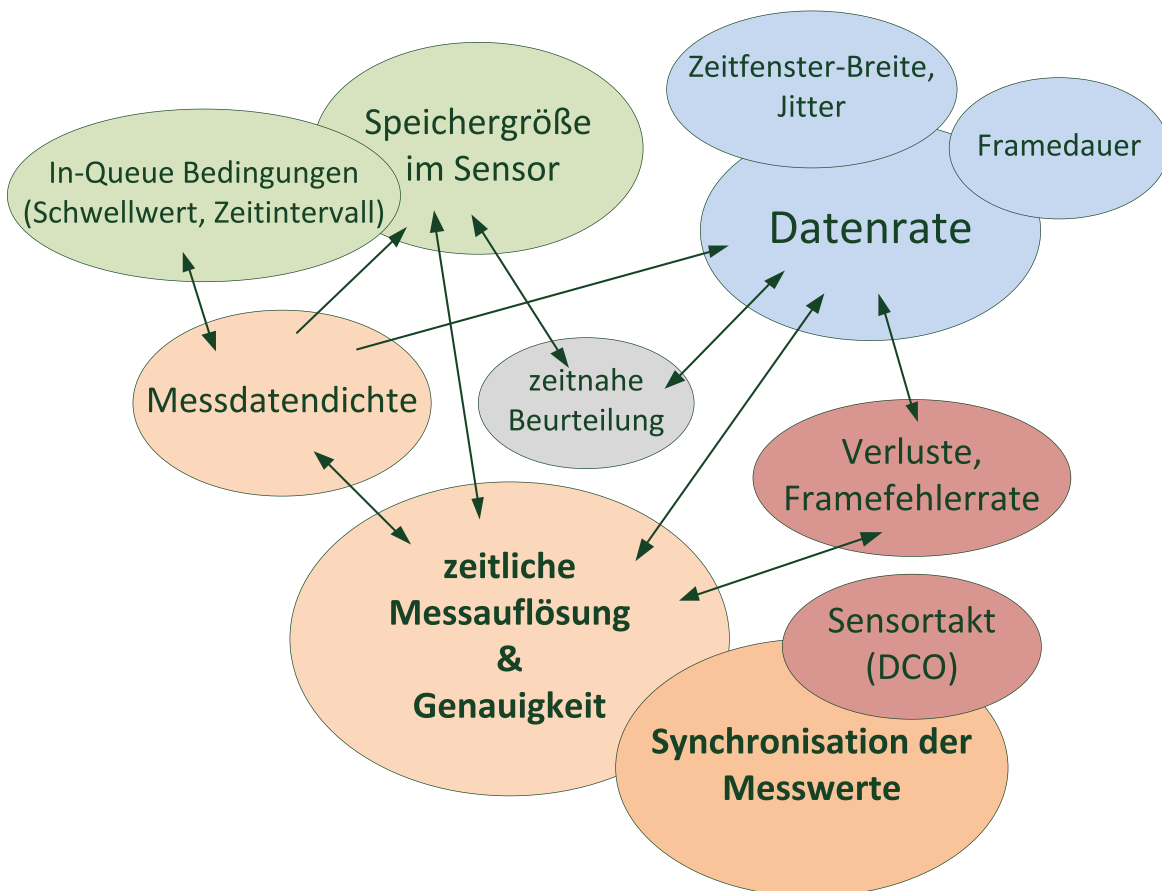


Abbildung 4.1.: Darstellung der Abhängigkeiten des gesamten Systems im Überblick

4.3. Fazit

Für das Forschungsprojekt BATSEN an der Hochschule für Angewandte Wissenschaften Hamburg hat diese Abschlussarbeit den ersten Schritt in Richtung Starterbatterie vollzogen. Es wurde gezeigt, dass das Konzept der einfachen Zellsensorik ohne Downlink durchaus Potenzial für die Anwendung im Automotive-Bereich besitzt. Die Durchführung und Erstellung der entwickelten Messmethodik ergab wertvolle Informationen über das Zellverhalten der Batterie im Betriebsfall. In Hinblick auf die Erhöhung der Datenrate wurde das Übertragungsverfahren weiterführend untersucht und die Hardware auf eine schnellere und fehlerfreie Übertragung angepasst. Es wurden die Anforderungen der Starterbatterie an die Sensorik aufgezeigt und ins Verhältnis zu den bestehenden Möglichkeiten gesetzt. Im Laufe der Arbeit entstand ein Konzept, um trotz geringer Datenraten eine hinreichend genaue Abtastung der Zellspannungen zu erreichen. Die Dokumentation der Ergebnisse zeigte hierbei

entstandene Schwächen und Probleme auf und gab Lösungsansätze an. Diese Arbeit war im Rahmen des Forschungsprojekts einem stetigen Entwicklungsprozess unterstellt, der neben motivierenden Erfolgen und auch ernüchternden Momente enthielt, die jedoch die Grundlagen für weiterführende Untersuchungen bildeten und den Ehrgeiz bei der Problemlösung weiter antrieben. Persönlich bereitete mich diese Abschlussarbeit durch ihren sehr praxisnahen Bezug sehr gut auf das anschließende Arbeitsleben vor und erlaubte einen guten Eindruck der projektbezogenen Arbeit.

5. Ausblick

Im Umfang des verwendeten Drahtlos-Systems sind noch an vielen Stellen Verbesserungen möglich, die durch diese Arbeit nicht mehr möglich waren. Im folgenden werden abschließend Ansätze aufgezeigt, anhand derer Optimierungen durchgeführt werden können.

- Die bisher entstandenen Quelltexte von Datenlogger und Sensor sollten weiter optimiert werden. Gerade die Software des Datenloggers ist bereits durch mehrere Arbeiten modifiziert und erfordert im Sinne des Forschungsprojektes und der Übersichtlichkeit eine Typenbereinigung.
- Die hier angepasste Strommessung wurde aufgrund von Schwierigkeiten bei der Zuordnung der Messwerte des Stroms zu den entsprechenden Zellspannungen nicht weiter untersucht. Ein Ansatz stellt das Puffern entsprechend vieler Stromwerte dar, mit dem ein Vorlauf erreicht wird, da Spannungswerte durch das Übertragungsverfahren verzögert werden. Die genaue Zuordnung kann offline nach dem Empfang geschehen. Das Verfahren ist in der Implementierung erwartungsgemäß sehr aufwendig und erfordert den häufigen, zwischenzeitlichen Abgleich zu jedem Sensor.
- Die Quellkodierung im Sensor erschwert die Rekonstruktion der Messwerte bezüglich zeitlicher Genauigkeiten. Der Sensortakt spielt dabei eine große Rolle. Die Kalibration des Taktfehlers ist ein weiterer Punkt, den es zu untersuchen gilt. Bisher entstand der Ansatz der Implementierung einer genaueren Zeitbasis im Empfänger. Im Fall von unverzögerten Datenframes ist es möglich Rückschlüsse auf den Sensortakt zu erlangen. Bei der Auswertung der Messdaten am PC flossen diese Informationen bereits in die Rückgewinnung der Zeitbasis ein. Die Genauigkeit lässt sich dabei noch erhöhen, indem dieser Abgleichprozess während des Datenempfangs vom Empfänger ausgeführt wird.
- Die realisierte Quellkodierung erfordert für eine hohe Messgenauigkeit einen ausreichend großen Zwischenspeicher. Die Abwägung des Kosten-Nutzen-Verhältnisses spricht in diesem Fall für den Einsatz erweiterter Hardware mit mehr physikalischen Möglichkeiten.
- Der Sensor lässt sich bezüglich der Bauteilkosten weiter optimieren. Ein quarz-loser Transmitter auf der Basis eines Silizium-LC-Oszillator wurde bereits im Laufe anderen Arbeiten erprobt und hat sich bewährt.

- Im Allgemeinen ist die Bitrate der Datenübertragung zu erhöhen. Diese hängt maßgeblich von der Taktrate des Mikrocontrollers ab. Der Empfänger an der Basisstation muss die Daten wieder auswerten. Die Folge einer Takterhöhung schlägt sich somit auch auf den auswertenden Mikrocontroller der Empfangseinheit nieder. Die Änderung der Bitrate erfordert hinterher die erneute Untersuchung auftretender Fehlerraten.
- Die Fehlerrate bei der Datenübertragung ist aufgrund des Sendeverfahrens sehr hoch und lässt sich nur zu Lasten der Übertragungsgeschwindigkeit senken. Die Messgenauigkeit ist jedoch stark von der Fehlerrate abhängig. Eine geeignete Kanalcodierung stellt hierfür einen guten Ansatz dar. Das Hinzufügen von Redundanzen kann die Fehlerrate bei dem verwendeten Sendeverfahren um einige Potenzen verringern. Dadurch wird zwangsweise das Sendeprotokoll vergrößert. In diesem Fall ändert sich die Betrachtung der Fehlerraten erneut, was im Verhältnis trotzdem einen Gewinn versprechen sollte..
- Der derzeitige Aufbau des Sensors erlaubt es nicht kontinuierlich, ohne Unterbrechungen die Zellspannung zu messen. Die Messvorgänge werden stets durch den Sendevorgang unterbrochen. Dies ist hardware-bedingt und lässt sich durch die großen Störungen des vorhandenen Step-Up-Konverters nicht vermeiden. Die kontinuierliche Messaufnahme ist jedoch zum Erreichen höherer Messgenauigkeit wichtig und bedarf daher der weiteren Untersuchung und einem eventuellen Redesign der Spannungsversorgung.
- Die Implementierung des zufälligen Sendens innerhalb eines Zeitfensters ist derzeit mit einem festen Zeitanteil versehen. Es wird so sichergestellt, die Zeitfenster eines Sensors sich nicht überlagern. Ein besserer Ansatz stellt möglicherweise das zusätzliche Warten über die verbleibende Zeit innerhalb des jeweiligen Zeitfensters dar (Abb. 5.1). Die Sendeintervalle können so deutlich reduziert und die Bitrate erhöht werden. Bezüglich der Fehlerrate bedarf diese Änderung einer neuen Untersuchung.

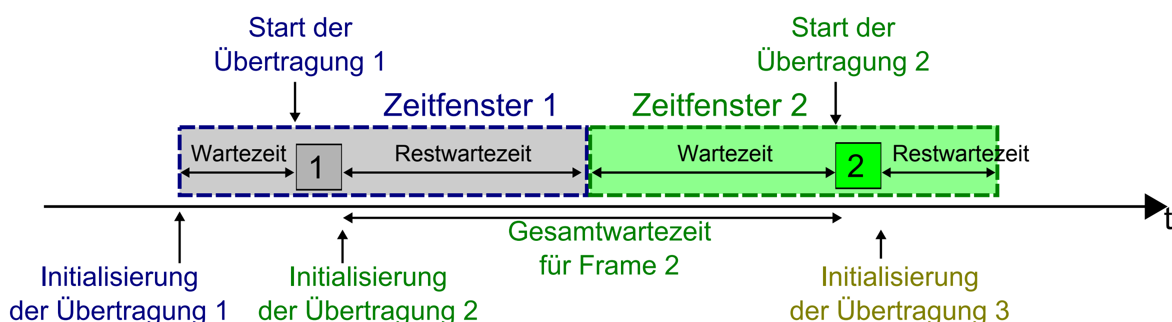


Abbildung 5.1.: Mögliches Sendeverfahren ohne festen Zeitanteil

Literaturverzeichnis

[1] ANLAUFT, Anna: *X-by-wire: Marktwachstum durch Vorteile entlang der gesamten Wertschöpfungskette*. Version: April 2008. http://frosttest.com/uat/servlet/press-release.pag?docid=127858849

[2] BOHLEN, Oliver ; BLANKE, Holger ; BULLER, Stephan ; FRICKE, Birger ; HAMMOUCHE, Abderrezak ; LINZEN, Dirk ; THELE, Marc ; DONCKER, Rik W. D. ; SAUER, Dirk U.: *Battery Monitoring for Automotive Batteries is not a Miracle*. http://www.isea.rwth-aachen.de. Version: Mai 2005

[3] BUCHMANN, Isidor: *Battery University*. http://www.batteryuniversity.com. Version: 2005

[4] FORM, Thomas: *Vorlesung Fahrzeugelektronik*. http://www.ifr.ing.tu-bs.de. Version: Februar 2007

[5] HOOPS, Alexander: *Praxissemesterbericht - Drahtlose Batteriesensorik und Batterietest*. Februar 2010

[6] KRANNICH, Tobias: *Experimentalsystem für einen Sensor-Controller mit drahtloser Energie- und Datenübertragung*, HAW Hamburg, Diplomarbeit, Oktober 2008. http://opus.haw-hamburg.de/volltexte/2008/615/

[7] LEM (Hrsg.): *DHAB S/24 Current Transducer Datenblatt*. LEM, 2008. http://www.lem.com

[8] MEISSNER, Eberhard ; RICHTER, Gerolf: Battery Monitoring and Electrical Energy Management Precondition for future vehicle electric power systems. In: *Journal of Power Sources* (2003), 79-98. http://www.elsevier.com/locate/jpowsour

[9] OLIMEX LTD. (Hrsg.): *StarterKit MSP430-169STK schematics*. Olimex Ltd., 2004. http://www.olimex.com/dev

[10] PLASCHKE, Stephan: *Experimentalsystem für drahtlose Batteriesensorik*, HAW Hamburg, Diplomarbeit, Oktober 2008. http://opus.haw-hamburg.de/volltexte/2008/620/

[11] REIF, Conrad: *Batterien, Bordnetze und Vernetzung*. Vieweg+Teubner, 2010. – ISBN 978–3–8348–1310–7

[12] RF MONOLITHICS, INC (Hrsg.): *RFM DevKit DR5100 Datenblatt*. RF Monolithics, Inc, 2008. http://www.rfm.com

[13] RF MONOLITHICS, INC (Hrsg.): *RFM RX5000 Datenblatt*. RF Monolithics, Inc, 2008. http://www.rfm.com

[14] RIEMSCHNEIDER, Karl-Ragmar ; VOLLMER, Jürgen: BATSEN Förderantrag im Programm 'FHprofUnt 2010' des Bundesministeriums für Bildung und Forschung. (2010)

[15] SAUER, Dirk U. ; KARDEN, Eckhard ; FRICKE, Birger ; BLANKE, Holger ; THELE, Marc ; BOHLEN, Oliver ; SCHIFFER, Julia ; GERSCHLER, Jochen B. ; KAISER, Rudi: Charging performance of automotive batteries. In: *Journal of Power Sources* (2007), 22-30. http://www.isea.rwth-aachen.de

[16] SAUER, Dirk U. ; SCHIFFER, Julia ; THELE, Marc ; BOHLEN, Oliver: Review of Monitoring Technologies for Lead-acid Batteries. In: *Advanced Automotive Battery Conference*, 2005

[17] SCHÖLLMANN, Matthias (Hrsg.): *Energiemanagement und Bordnetze*. Expert-Verlag, 2005 (Haus der Technik Fachbuch 41). – ISBN 3–8169–2466–2

[18] SCHÖLLMANN, Matthias (Hrsg.): *Energiemanagement und Bordnetze II*. Expert-Verlag, 2007 (Haus der Technik 71). – ISBN 978–3–8169–2649–8

[19] STURM, Matthias (Hrsg.): *Mikrocontrollertechnik : Am Beispiel der MSP430-Familie*. München : Hanser, 2011. – ISBN 978–3–446–42231–5

[20] TEXAS IMSTRUMENTS (Hrsg.): *MSP430x1xx Family User's Guide*. F. Texas Imstruments, Februar 2006

[21] TEXAS INSTRUMENTS (Hrsg.): *MSP430F12x2 Mixed Signal Microcontroller Datasheet*. D. Texas Instruments, August 2004

[22] TEXAS INSTRUMENTS (Hrsg.): *MSP430F16x Mixed Signal Microcontroller Datasheet*. Texas Instruments, März 2011. http://www.msp430.com

[23] WEYDANZ, Wolfgang ; JOSSEN, Andreas: *Moderne Akkumulatoren richtig einsetzen*. Reichardt Verlag, 2006. – ISBN 978–3–939359–11–1

[24] WITTE, Erich: *Blei- und Stahlakkumulatoren*. Bd. 3. Krausskopf-Verlag, 1967

[25] ZVEI: Publikationen / Fachverband Batterien. Version: April 2005. http://www.zvei.org/fachverbaende/batterien/publikationen/. 2005. – Forschungsbericht

Anhang

A. Quelltext

A.1. Sensor

Listing 1: Quellcode des Sensorsystems (main.c)

```
/*
    -------------------------------------------------------------------------------------
  Project:      BATSEN Sensorclass I
  Discription:
  Used components:  Wireless Battery Sensor without downlink

  Author:       Simon Püttjer
  Date:       04/01/2011
  Last update:    16/04/2010

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

-------------------------------------------------------------------------------------
  Headerfiles
-----------------------------------------------------------------------------------*/
#include <io.h>
#include <signal.h>
#include "inline.h"
#include "main.h"
#include "isr.h"
/*-----------------------------------------------------------------------------------
  Globals
//-------------------------------------------------------------------------------*/
//tx stuff
//****************************************
unsigned short temp = 0;  //temporary adc10 mem register
state_1 txstate_main = CHECK_QUEUE_1; //transmission main state
unsigned char txCount = 0;  //transmit count variable
unsigned char txNextOut = 0;  //transmit level flag
unsigned char txByte = 0; //byte index
unsigned char txBit = 0x80; //bit index msb first
tx txState = SOF;
// init of random shift register
static unsigned long sr_r = 0x08000000;
//waiting queue - struct_array
volatile queue waiting_queue[QUEUE_LENGTH];
```

```
38  //sample stuff
39  //***************************************
40  unsigned char writeindex=0, readindex =0;
41  unsigned char livebit = 0;
42  sa saState = TIMING;
43  adc adc10_state = ADC_IDLE;

45  //transmission protocol
46  //-----------------------
47  unsigned char txData[14] = {

49      0x00, 0x01, // RunIn (16 Bit)
50      0x00, 0x00, 0x00, 0x00, // parameters (version , live_bit, misc (1 byte)), jitter (2
         byte), sensor address (1 byte)
51      //for frame error measurement the address is shortened to 1 byte
52      0x00, 0x00, // cell voltage (2 byte)
53      0x00, 0x00, // timelabel (2 byte)
54      0x00, 0x00, // current queuelength
55      0x00, // CRC XOR[2 .. 11]
56      0x00 //for stopbit
57      };

59  //sensor info
60  struct InfoBlock {
61    unsigned long magicID;
62    unsigned long sonsorID;
63    // Daten für Kalibrierung
64  //  unsigned short vFactor;
65  //  unsigned short tOffset;
66  //  unsigned short tFactor;
67  };

69  //store sensor information at memory address 0x1000
70  const struct InfoBlock * const infoBlock = (const struct InfoBlock *) 0x1000;

72  //variables for sensor address validation
73  const unsigned long magicID = 0x64490F01;
74  const unsigned char version = 0x01;

76  //inqueue variables
77  unsigned short val_diff = 1,  //value difference
78      time_diff;        //time difference

80  //prototypes
81  //-----------------------------------------------------------------------
82  void ADC10init(void); // adc initialization
83  inline unsigned short transmit(unsigned short temperature, unsigned short battery,
        unsigned short supply); //fill transmit protocol


86  //functions
87  /*----------------------------------------------------------------------
88   * prepare data buffer and initiate transmission
89  ----------------------------------------------------------------------*/
90  unsigned short transmit(unsigned short temperature, unsigned short battery, unsigned
        short supply)
91   {
92    unsigned char n;  //loop index for parity
93    unsigned short max; //maximum wait time count
```

```
95    //transmission variables reset
96    //---------------------------------
97    P1OUT &= ~0x01; // P1.0 clear data pin
98    txNextOut = 0; //reset data output to low
99    txCount = 0; //initiate transmission
100   txByte = 0; //reset byte index
101   txBit = 0x80; //reset bit index
102   txState = SOF;  //reset protocol
103   //waiting queue init
104   //never count transmission index
105   //    number of interrupts to wait for next transmission
106   SR(); //random shift register macro (see main.h)

108   //calculate the current jitter from randum number sr_r, (TXMID - (TXMID / TXFAC)) is
          the steady part
109   //
          ********************************************************************************************

110   max = (unsigned short) (sr_r % (2 * (TXMID / TXFAC)) + (TXMID - (TXMID / TXFAC)));

112   //livebit - indicates a transmission without jitter
113   if(livebit) //check routine: see isr.c
114     txData[2] = livebit + BIT0;
115   else
116     txData[2] = BIT0;
117     //careful! txData[2] is for version ID! Leave the MSB alone! It's important for the
            receiver!

119   //Jitter
120   //************************************************
121   txData[3] = (unsigned char)(max>>8);
122   txData[4] = (unsigned char)(max & 0x00FF);
123   //************************************************
124   // temperature is used for cell voltage
125   txData[6] = (unsigned char) (temperature >> 8);
126   txData[7] = (unsigned char) (temperature & 0x00FF);
127   // cell  is used for timelabel
128   txData[8] = (unsigned char) (battery >> 8);
129   txData[9] = (unsigned char) (battery & 0x00FF);
130   //supply is used for current inqueue length
131   txData[10] = (unsigned char) (supply >> 8);
132   txData[11] = (unsigned char) (supply & 0x00FF);

134   //generate parity
135   //**************************
136   txData[12] = 0;
137   for (n = 2; n < 12; n++)
138     txData[12] ^= txData[n];
139   //**************************

141   livebit = 0;  //reset livebit, next value one might be from queue
142   return max;   //return maximum wait time
143 }

145 /*-------------------------------------------------------------
146  * initialize transmission
147  -------------------------------------------------------------*/
148 void initTransmit(unsigned long id)
149 {
```

```
150   // P1.0 Data Transmit
151   // P1.1 Select Transmit
152   P1OUT &= ~(BIT1 + BIT0); // P1.0 + P1.1 clear
153   P1SEL &= ~(BIT1 + BIT0); // P1.0 + P1.1 => IO Port
154   P1DIR |= BIT1 + BIT0; // P1.0 + P1.1 => outputs

156   txCount = 0;
157   txNextOut = 0;

159   txData[2] = (unsigned char) (id >> 24); //version ID
160   txData[3] = (unsigned char) (id >> 16); //cal. values/address (--> is used for FER
          calc.)
161   txData[4] = (unsigned char) (id >> 8);  //address --> (used for FER calc.)
162   txData[5] = (unsigned char) (id & 0x000000FF);  //address
163   txData[12] = 0;
164 }
165 /*-------------------------------------------------------------------------------
166  * initialize ADC10
167  -------------------------------------------------------------------------------*/
168 void ADC10init(void/*unsigned char mode_adc*/) {

170   //initialize adc measurements
171   //----------------------------------------------------------------------------
172   ADC10CTL0 &= ~ENC; // reset ENC
173   //  ADC10CTL1 = INCH_2 + ADC10DIV_3;       // Ch 2 ADC10CLK/4
174   //  ADC10CTL1 = INCH_2 + ADC10DIV_2;       // Ch 2 ADC10CLK/3
175   ADC10CTL1 = INCH_2 + ADC10DIV_0 + CONSEQ_0 + ADC10SSEL_3; // Ch 2 + ADC10CLK/1+ single
           conversion mode + SMCLK
176   //  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + REF2_5V + ADC10ON;
177   //  ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + REF2_5V + ADC10ON;
178   ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + REF2_5V + ADC10ON;
179   //  ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + ADC10ON + REF2_5V + ADC10IE;  //
          reference: Vss, hold-time 8*ADC10CLK; reference on, ADC10 on
180   //  ADC10CTL1 = INCH_2 + + ADC10DIV_7 + CONSEQ_2; //ADC10 Channel 2, ADC10CLK/8,
          repeating single channel

182   //old code: temperature and supply voltage currently not used
183 /*  switch (mode_adc) {

185   case SET_TEMP:
186     //initialize temperature measurement
187     //-------------------------------------------------------------------------
188     ADC10CTL0 &= ~ENC; // reset ENC
189     //    ADC10CTL1 = INCH_10 + ADC10DIV_3;       // Temp Sensor ADC10CLK/4
190     //    ADC10CTL1 = INCH_10 + ADC10DIV_2;       // Temp Sensor ADC10CLK/3
191     ADC10CTL1 = INCH_10 + ADC10DIV_2 + CONSEQ_0; // Temp Sensor
192     //    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE;
193     //    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
194     //    ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + ADC10ON;
195     ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + ADC10ON;
196     break;
197   case SET_SUPP:
198     //initialize supply voltage measurement
199     //-------------------------------------------------------------------------
200     ADC10CTL0 &= ~ENC; // reset ENC
201     //    ADC10CTL1 = INCH_11 + ADC10DIV_3;       // Ch 11 ADC10CLK/4
202     //    ADC10CTL1 = INCH_11 + ADC10DIV_2;       // Ch 11 ADC10CLK/3
203     ADC10CTL1 = INCH_11 + ADC10DIV_2 + CONSEQ_0; // Ch 11
204     //    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + REF2_5V + ADC10ON;
205     //    ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + REF2_5V + ADC10ON;
```

```
206       ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + REF2_5V + ADC10ON;
207       for (n = 0; n < 3; n++)
208         x++;
209       break;
210     case SET_CELL:
211       //initialize cell voltage measurement
212       //-------------------------------------------------------------------------------
213       ADC10CTL0 &= ~ENC; // reset ENC
214       //    ADC10CTL1 = INCH_2 + ADC10DIV_3;        // Ch 2 ADC10CLK/4
215       //    ADC10CTL1 = INCH_2 + ADC10DIV_2;        // Ch 2 ADC10CLK/3
216       ADC10CTL1 = INCH_2 + ADC10DIV_0 + CONSEQ_2; // Ch 2 + ADC10CLK/1
217       //    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + REF2_5V + ADC10ON;
218       //    ADC10CTL0 = SREF_1 + ADC10SHT_1 + REFON + REF2_5V + ADC10ON;
219       ADC10CTL0 = SREF_1 + ADC10SHT_2 + REFON + REF2_5V + ADC10ON;
220       for (n = 0; n < 3; n++)
221         x++;
222       break;
223     }}*/
224   }

226   int xy;
227   /*-------------------------------------------------------------------------------------
228    *  main routine
229   -------------------------------------------------------------------------------------*/
230   int main(void) {

232     //init queue with 0 (for testing)
233     //***************************************
234     for (xy = 0; xy < QUEUE_LENGTH; xy++) {
235       waiting_queue[xy].value = 0;
236       waiting_queue[xy].val_reg = 0;
237     }

239     //set up sensor
240     //*****************************************
241     //no match with magic ID --> sensor address 0
242     if (infoBlock->magicID != magicID)
243       initTransmit(0x00000000);
244     else
245     { //match with magic ID --> set sensor address written by debugger
246       unsigned long sonsorID = infoBlock->sonsorID; //take address from memory address
247       sonsorID &= 0x000FFFFF; //Sensor address: 20 bits
248       unsigned long versionID = version;
249       versionID <<= 24; //version 1 byte
250       sonsorID |= versionID;

252       sr_r = (sonsorID << 1) | 0x08000000; //init shift register
253       initTransmit(sonsorID); //initialize transmit with sensoraddres etc.
254     }

256     // avert Watchdog reset
257     WDTCTL = WDTPW + WDTHOLD;

259     // set DCO to 3MHz; RSEL=7 DOC=3 MOD=0 DCOR=0
260     DCOCTL = DCO1 | DCO0; // DOC=3 MOD=0
261     BCSCTL1 = RSEL2 | RSEL1 | RSEL0; // RSEL=7
262     BCSCTL2 = 0; // DCOR=0 (no external R)

264     // set up Timer A
265     //  want SMCLK
```

```
266   //  count up (sawtooth), period = COUNTS
267   TACTL = MC_1 | ID_0 | TASSEL_2;

269   // compare mode, interrupt enabled
270   TACCTL0 = CCIE; // other CCTLx regs not used

272   TACCR0 = 325; // cycle length

274   // other CCRx regs not used
275   // other CCTLx regs not used

277   ADC10init();  // initialize ADC10

279   eint();     // global interrupt enable

281   // P2.0 Output controls  Step-Up-Converter
282   P2OUT |= 0x01; // P2.0 set - enable StepUp
283   P2SEL &= ~0x01; // P2.0 IO Port
284   P2DIR |= 0x01; // P2.0 output
285   // P3.3 Output for timing test pin (for testing)
286   P3OUT |= 0x08; // P3.3 set
287   P3SEL &= ~0x08; // P3.3 IO Port
288   P3DIR |= 0x08; // P3.3 output
289   // P3.6 Output für LED (function indication)
290   P3SEL &= ~BIT6; // P3.6 IO Port
291   P3DIR |= BIT6; // P3.6 output
292   LED_OFF;  //init LED turned off

294   //main loop
295   //**************************************
296   for (;;);

298   //LED blinks when transmission is set
299 }
```

Listing 2: Quellcode des Sensorsystems (isr.c)

```
1   /*
      -------------------------------------------------------------------------------------
2     Project:      BATSEN Sensorclass I
3     Discription:
4     Used components:  Wireless Basttery Sensor without downlink

6     Author:       Simon Püttjer
7     Date:       04/01/2011
8     Last update:    16/04/2010

10    File:       isr.c
11  ------------------------------------------------------------------------------------------

12    Timer ISR
13  ------------------------------------------------------------------------------------
14    Headerfiles
15  ----------------------------------------------------------------------------------*/
16  #include <io.h>
17  #include <signal.h>
18  #include "inline.h"
19  #include "main.h"
20  #include "isr.h"
```

```
22  //globals
23  //-------------------------------------------------------------------
24  extern unsigned char txCount; //transmit count variable
25  extern unsigned char txNextOut; //transmit level flag
26  extern unsigned char txByte; //byte index
27  extern unsigned char txBit; //bit index msb first
28  extern unsigned char mod_adc; //select adc mode register
29  extern unsigned char txData[14]; //transmit protocol
30  extern queue waiting_queue[QUEUE_LENGTH]; //queue strucarray
31  extern unsigned char writeindex, readindex; //indices for queuing
32  extern unsigned long sr(void); //random shift register
33  extern tx txState;  //transmission main state
34  extern sa saState;  //sampling main state
35  extern adc adc10_state; //lower sampling state
36  extern state_1 txstate_main;
37  extern unsigned char livebit;
38  extern unsigned short temp;
39  extern unsigned short val_diff, time_diff;

41  //prototypes
42  //--------------------------------------------------------------------
43  extern unsigned long sr(void); //shift register
44  extern inline unsigned short transmit(unsigned short temperature,
45      unsigned short battery, unsigned short supply); //write measured data to tx buffer
46  /*--------------------------------------------------------------------
47   * Timer Interrupt sets output level for transmitter (see timer init in main function
        (~100us))
48   --------------------------------------------------------------------*/
49  interrupt (TIMERA0_VECTOR) timerRoutine() {
50    static unsigned char adc10_count = 0, interval_count = 0; //time counter-variables
51    static unsigned short last_sample = 0;  /*35000 is ~2V for high mode detection*/
52    static unsigned short timelabel = 0;  //timelabel for buffered data
53    static unsigned short random_wait = 0;  //random wait counter
54    /*################## sampling ########################*/

56    switch(adc10_state)
57    {
58      case SAMPLE_BUSY:
59        if(ADC10CTL1 & ADC10BUSY)//still busy --> maybe in next ISR
60        {
61          saState = IDLE_STATE;
62          adc10_state = SAMPLE_BUSY;
63        }
64        else//pick sample
65          {
66            temp +=ADC10MEM;
67            saState = TIMING;
68            STEPUP_ON;
69            adc10_state = ADC_IDLE;
70          }
71        break;
72      case ADC_IDLE:
73        break;
74    }
75  //  saState = IDLE_STATE;
76  if(!txCount){

78  //  switch(adc10_state)
79  //  {
```

```
80  //    case SAMPLE_BUSY:
81  //      if(ADC10CTL1 & ADC10BUSY)//still busy --> maybe in next ISR
82  //      {
83  //        saState = IDLE_STATE;
84  //        adc10_state = SAMPLE_BUSY;
85  //      }
86  //      else//pick sample
87  //        {
88  //          temp +=ADC10MEM;
89  //          saState = TIMING;
90  //          STEPUP_ON;
91  //          adc10_state = ADC_IDLE;
92  //        }
93  //      break;
94  //    case ADC_IDLE:
95  //      break;
96  //  }

98     switch (saState)
99     {
100      case IDLE_STATE:
101        //do nothing
102        break;

104      case TIMING:
105        if (interval_count < SAMPLE_INTERVAL)//samples ready?
106        {
107          if (adc10_count < SAMPLE_TIME-1)//time to sample?
108          {
109            adc10_count++;
110          }
111          else
112          {
113            adc10_count = 0; //reset adc counter
114            interval_count++;
115            STEPUP_OFF;
116            ADC10CTL0 |= ENC + ADC10SC; //start adc10 sampling
117            adc10_state = SAMPLE_BUSY;
118          }
119          break;
120        }
121        //transmission in progress! do not compare! just count timelabel!
122 //      else if(txCount)
123 //      {
124 //        interval_count = 0;
125 //        timelabel++;
126 //        break;
127 //      }
128        else
129        {
130        interval_count = 0;
131        timelabel++;
132        //no break here! compare samples right away!
133 //      saState = SAMPLE_COMPARE;
134        }

136      case SAMPLE_COMPARE:
137        //compare samples to detect a voltage drop due to high current
138        if(temp > last_sample)
139          val_diff = temp - last_sample;
```

```
140        else
141          val_diff = last_sample - temp;

143        if(timelabel > waiting_queue[writeindex].val_reg)
144          time_diff = timelabel - waiting_queue[writeindex].val_reg;
145        else
146          time_diff = waiting_queue[writeindex].val_reg - timelabel;

148        if(val_diff>THRESHOLD)
149          //high current detected
150        {
151          //store sample in queue
152          val_diff = 0;
153          saState = SAVETOQUEUE;
154        }
155        else if(time_diff>=TIME_DIFF) //time difference since last inqueue
156        {
157          //normal transmission when transmitter is ready
158          saState = SAVETOQUEUE;
159        }
160        else
161        {
162          //transmission in progress --> discard sample
163  //        last_sample = temp; //save last sample
164          saState = TIMING;
165          temp = 0;
166          break;
167        }

169      case SAVETOQUEUE:
170        //save sample to queue until queue is full
171        if (((writeindex + 1) % QUEUE_LENGTH) != readindex)
172        {
173          writeindex = (writeindex + 1) % QUEUE_LENGTH;
174          waiting_queue[writeindex].value = temp; //put element to buffer
175          waiting_queue[writeindex].val_reg = timelabel; //put time label
176        }
177        saState = TIMING;
178        last_sample = temp; //save last inqueue
179        temp = 0; //reset sum variable

181    }

183  }
184  else
185  {
186    if (interval_count < SAMPLE_INTERVAL)//samples ready?
187    {
188      if (adc10_count < SAMPLE_TIME-1)//time to sample?
189      {
190        adc10_count++;
191      }
192      else
193      {
194        adc10_count = 0; //reset adc counter
195        interval_count++;
196      }
197    }
198    else
199    {
```

```
200      timelabel++;
201      interval_count = 0;
202    }
203    /*################## sampling  end########################*/
204  }


207    /*################## transmission ###################*/
208    //%%%%%%%%%%%%%%% tx main  %%%%%%%%%%%%%%%%%%%%
209    switch (txstate_main)
210    {
211      case CHECK_QUEUE_1:
212        //check queue for data to transmit
213        if ((readindex % QUEUE_LENGTH) != writeindex)
214          {
215          readindex = (readindex +1) % QUEUE_LENGTH;
216          if((timelabel == waiting_queue[readindex].val_reg))   //not delayed by queue
217            livebit = 2;

219          //actual queue volume
220          //**********************************************************
221          if(writeindex<readindex)  //righthandside from writeindex
222          { //ignite transmission an put element to tx buffer
223            random_wait = transmit(waiting_queue[readindex].value,
224                waiting_queue[readindex].val_reg,(unsigned short)QUEUE_LENGTH-(readindex-
                     writeindex));
225          }
226          else            //lefthandside from writecount
227          { //ignite transmission an put element to tx buffer
228            random_wait = transmit(waiting_queue[readindex].value,
229                waiting_queue[readindex].val_reg,(unsigned short)(writeindex-readindex));
230          }
231          //**********************************************************

233          txstate_main = WAIT_1;  //wait state set
234        }
235        else
236        {
237  //          saState = SAMPLE_BUSY;  //new data please!
238          break;
239        }
240        //when queue contains a value do not break!

242          txstate_main = WAIT_1;  //wait state set
243          //TEST_TOGGLE;

245      case WAIT_1:
246        //wait for random interval
247        if (random_wait)
248        {
249          random_wait--;
250          break;
251        }
252        else
253        {
254        //ignite transmission
255        //TEST_TOGGLE;

257        txCount = 1;
258        TX_ENABLE; //enable transmitter
```

```
259        txstate_main = TX_NOW;
260        //do not break after wait count is 0!
261        }
262      case TX_NOW:
263        //first set data out
264        if (txNextOut)
265          TXDATA_ON;  //tx pin high
266        else
267          TXDATA_OFF; //tx pin low
268        //protocol timings
269        switch (txCount) {
270        //begin SOF low
271          case SOF_LOW:
272            txNextOut = 0;

274            LED_ON; //LED on
275            break;
276            //after 6 * 2 SOF high
277          case SOF_HIGH:
278            txNextOut = 1;
279            break;
280          //after SOF begin DATA transmission
281          case DATA_START:
282            txState = DATA;
283            break;
284            //end of frame
285          case EOF:
286            txState = END;
287        }
288      //%%%%%%%%%%%%%%% transmission protocol %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
289        switch (txState)
290        {
291          case DATA:
292            //take every second half of a bit
293            if (txCount & 0x01)
294              // Manchester Code
295              // second half of bit => negate signal
296              txNextOut = !txNextOut;
297              //other half is the data to transmit
298            else
299            {
300              txNextOut = txData[txByte] & txBit; //take bit X (txByte) of a byte Y (txBit
                  )
301              //end of byte reached --> reset bit index, count up byte index
302              if (txBit == 0x01)
303              {
304                txBit = 0x80; //reset bit index to msb
305                txByte++; //count up byte index
306              }
307              //no lsb --> next bit
308              else
309                txBit = txBit >> 1; //next bit
310            }
311            break;
312          case END:
313            //end of transmission, reset all counters
314            txCount = 0;
315            TXDATA_OFF; //data output low
316            TX_DISABLE; //disable transmitter
317            txstate_main = CHECK_QUEUE_1; //go check waiting queue
```

```
318             saState = TIMING;
319             LED_OFF; //LED off
320             interval_count = 0;
321             adc10_count = 0;
322             temp = 0;
323             break;
324           case SOF:
325             break;
326         }
327         if (txState != END)
328           txCount++;
329         break;
330       //%%%%%%%%%%%%%%% transmission protocol END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
331     }
332     //%%%%%%%%%%%%%%%%%%%% tx main END %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

334 }
```

Listing 3: Quellcode des Sensorsystems (main.h)

```
1  /*
       ----------------------------------------------------------------------------------------
2    Project:       BATSEN Sensorclass I
3    Discription:
4    Used components:  Wireless Battery Sensor without downlink

6    Author:        Simon Püttjer
7    Date:        04/01/2011
8    Last update:     16/04/2010

10   File:        main.h
11 ------------------------------------------------------------------------------------------

13 ------------------------------------------------------------------------------------
14   Defines
15 ----------------------------------------------------------------------------------*/
16 #ifndef MAIN_H_
17 #define MAIN_H_

19 #define TXFAC 3   //factor the time varies around average transmission time
20 #define TXMID 5000  //number of interrupt to wait
21 #define QUEUE_LENGTH 35 //waiting queue length

23 #define LED_OFF (P3OUT |= BIT6) // P3.6 set - LED off
24 #define LED_ON (P3OUT &= ~BIT6) // P3.6 clear - LED on
25 #define STEPUP_ON (P2OUT |= BIT0) // P2.0 set - StepUp on
26 #define STEPUP_OFF (P2OUT &= ~BIT0) // P2.0 clear - StepUp off
27 #define TEST_TOGGLE (P3OUT ^= BIT3) // P3.3 toggle
28 #define TXDATA_OFF (P1OUT &= ~BIT0) // P1.0 clear - data low
29 #define TXDATA_ON (P1OUT |= BIT0) // P1.0 set - data high
30 #define TX_ENABLE (P1OUT |= BIT1) // P1.1 set - enable transmitter
31 #define TX_DISABLE (P1OUT &= ~BIT1) // P1.1 clear - disable transmitter

33 //random shift register macro
34 //****************************
35 #define SR() sr_r <<= 1;\
36   sr_r |= (((sr_r >> 28) ^ (sr_r >> 19)) & 0x00000001);\
37   sr_r &= 0x0FFFFFFF;
```

```
38 //****************************

40 //waiting queue struct
41 typedef struct {
42   unsigned short value; //measured value
43   unsigned short val_reg; //flagregister (high current (bit7: 1) and rest (bit7: 0))
44 }queue;

46 #endif /*MAIN_H_*/
```

Listing 4: Quellcode des Sensorsystems (isr.h)

```
1  /*
     -----------------------------------------------------------------------------------------
2    Project:      BATSEN Sensorclass I
3    Discription:
4    Used components:  Wireless Basttery Sensor without downlink

6    Author:       Simon Püttjer
7    Date:       04/01/2011
8    Last update:    16/04/2010

10   File:       isr.h
11 ------------------------------------------------------------------------------------------

13 ------------------------------------------------------------------------------------
14 ----------------------------------------------------------------------------------*/
15 #ifndef ISR_H_
16 #define ISR_H_
17 //######### transmission ###############

19 //transmission main states
20 typedef enum
21 {
22   CHECK_QUEUE_1,  //check queue for new data
23   WAIT_1,     //wait given time interval
24   TX_NOW      //transmit
25 }state_1;

27 /*%%%%%%%%% state: transmit %%%%%%%%%%%%*/
28 //protocol timing defines (multiples of timer interrupts)
29 #define SOF_LOW 1 //StartOfFrame (frame delay)
30 #define SOF_HIGH (6 * 2)
31 #define DATA_START ((6 + 4) * 2)
32 #define EOF (((6 + 4) + 13 * 8) * 2 + 2+1)  //EndOfFrame

34 //data pin status
35 typedef enum
36 {
37   SOF,  //StartOfFrame
38   DATA, //data part
39   END   //end
40 }tx;

42 /*########## sampling ###################*/

44 //timings (multiples of timer interrupts)
45 #define SAMPLE_TIME 10           //high sampling steps of 100us
```

```
#define SAMPLE_INTERVAL 10          //number of samples to summerize along (64 is max)
#define THRESHOLD (SAMPLE_INTERVAL * 7)    //threshold for inqueueing (x * (1.5 * 2.4)mV)
#define TIME_DIFF 200 //inqeue condition: threshold of time difference since last inqeue

//sampling main states
typedef enum
{
  IDLE_STATE,      //do nothing
  TIMING,        //count timing variables
  SAMPLE_COMPARE,    //compare samples
  SAVETOQUEUE      //save to queue
}sa;

//adc states
typedef enum
{
  SAMPLE_BUSY,   //check if adc ready and read value
  ADC_IDLE     //adc is idle
}adc;

#endif /*ISR_H_*/
```

A.2. Datenlogger

Listing 5: Quellcode des Datenloggers

```
/*-----------------------------------------------------------------------------------
  Project:      MSP430 project source file
  Used components:    MSP430-169STK
  Date:       10/28/2007
  Last Update:       04/04/2008
  Author:       Stephan Plaschke

  Modified by:     Alexander Hoops
  Last modification:  19/01/2010
  Modified by:     Simon Püttjer
  Last modification:  05/05/2011
-------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------
  Headerfiles
-----------------------------------------------------------------------------------*/
#include <ISR.h>
#include <RTC.h>
#include <ADC.h>
#include <main.h>
#include <flash.h>
#include <stdio.h>
#include <signal.h>
#include <lcd16x2.h>
#include <uart_menu.h>
#include <msp430x16x.h>
#include <MSP_functions.h>
#include <math.h>

/*-----------------------------------------------------------------------------------
```

```
31    Global variables
32  ---------------------------------------------------------------------------------*/
33  unsigned char BATMON_control_reg; // global control register
34  unsigned char BUTTON_state;      // actual button state
35  unsigned char RADIO_rx_state;    // data receive state
36  unsigned char UART_menu_position; // UART_menue position
37  unsigned char WDT_function;    // WDT function, BLINKY for RTC SETUP or RTC
38  unsigned char RTC_menue_position; // menu position in RTC setup
39  unsigned char Enable_clear_lcd;   // clear first LCD row
40  unsigned char show_sensornmb;    // choice of LCD screening
41  unsigned char measuring=0;     // Datalogging activated = 1
42  unsigned char scan_mode=0;     // standard_measure =>0 /
43            // scan mode sensor(1)/temp cal (2)
44  unsigned long temp_correct_value;    // temp value for calibration
45  unsigned char SENSOR_addr_req[5]; // array for continuous mode,
46            // shows required address
47  unsigned char SENSOR_addr_receiv[5];  // array for checking if all sensors have
48            // send a value
49  unsigned char SENSOR_address[40]; // array with sensor address
50  unsigned char SENSOR_active;     // active sensor count
51  unsigned char SENSOR_last_data[8][40];  // last received data from each sensor
52  unsigned char SENSOR_low;    // number of sensors under min voltage
53  /*----Sensor temperature calibration data----*/
54  signed char SENSOR_temp_cal_b[41]={
55      0,27,22,-5,10,15,-14,2,9,
56      -34,81,32,0,-24,38,0,0,0,0,0,
57      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
58      0,0,0,0,0}; // sensor address 0..40

60  /*-------------Current Measuring--------------*/
61  unsigned long CURRENT_last_data;     // stored current value
62  unsigned char CURRENT_IN_OUT;      // 1 = charge 0 = discharge
63  unsigned long CURRENT_tara;      // tara for current value reset = 0
64  unsigned char CURRENT_INVERT=0;      // variable to setup sensor
65  unsigned long AH_last_minute=0;      // AH last minute
66  unsigned long AH_last_sec=0;       //AH last second
67  unsigned char AH_last_minute_IN_OUT=0;  // 1 = charge 0 = discharge
68  unsigned long AH_MASTER;          // AH picker *0.44 = AH
69  unsigned char U_MASTER_CAPACITY;     // Capacity created by voltage
70              // values in percent DOC
71  unsigned char CURRENT_channel = 0; //used sensor channel (1: ch1(low), 2: ch2(high))

73  /*-------------Battery Parameters-------------*/
74  unsigned long CELL_VOLT_MAX=214;     // Cell voltage max 2.10V = 210
75  unsigned long CELL_VOLT_MIN=189;     // Cell voltage min 1.88V = 188
76  unsigned long BATT_CAPACITY=88;    // Battery capacity 500Ah = 500

78  RADIO_FRAME_STRUCT  RADIO_frame;     // received data frame
79  RTC_MSP430       RTC_values; // global RTC values

81  unsigned char global_string[MAX_STRING_LENGTH]; // global command string

83  //alles für den Flash
84  unsigned char  WRITE_BUF [BUF_SIZE];
85  unsigned char  READ_BUF [BUF_SIZE];
86  unsigned char  Ring_Full;
87  unsigned short Index_Ring;
88  unsigned short Count_Ring;
89  unsigned short Index_Last;
```

```
91    //frame error calculations
92  //  int long txErrorCountActual[6];
93  //  int long txErrorCountOld[6] = {0,0,0,0,0,0};
94  //  int long txError[6][2];
95  unsigned char fer = 0;


98  /*--------------------------------------------------------------------------------
99    MAIN LOOP
100 ------------------------------------------------------------------------------*/
101 int main (void)
102 {
103   unsigned char OSC_error;
104   unsigned char UART_error;

106   RTC_MSP430 RTC_tmp={0,0,0,0};

108 /*--------------------------------------------------------------------------------
109   Set local/global variables
110 ------------------------------------------------------------------------------*/
111     // initialize clock
112   RTC_values.day     = 22;
113   RTC_values.month   = 01;
114   RTC_values.year    = 2009;
115     RTC_values.wday    = TUE;
116     RTC_values.hr    = 23;
117     RTC_values.min     = 59;
118     RTC_values.sec     = 55;

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

123   UART_menu_position = UART_MENU_FIRSTCHAR;
124   Enable_clear_lcd = STATUS;
125   show_sensornmb = 0;
126 //--------------------------------------------------------------------------------

128   //OSC_error = DCO_set(uC_FREQUENCY);     // internal oscillator
129     OSC_error = XT2_set(uC_FREQUENCY);     // external oscillator
130   // initialize USART, enable RX&TX interrupt and RX&TX modul
131   USART0_init(RX_INT, RX_TX);
132   // initialize PORT direction/function
133   PORTS_init();
134   // initialize 16x2 character LCD
135   LCD_init();
136   // initialize TIMERB, debounce function for button B1-B3
137 //  TIMERB_init();
138   // initialize LEDs+Tara value
139   LED1_OFF;
140   LED2_OFF;
141   // global interrupt enable
142   _EINT();
143   //CURRENT_tara = 0;
144   // initialize RTC
145   RTC_startup();
146   // initialize Flash
147   init_Flash();
148   // Initialize radio unit
149   RADIO_init();
150   // read control data from flash?
```

```
151     FLASH_read_at_startup();
152     // display time on LCD
153     LCD_send_time();
154     // stop measurement
155     UART_menu_stop();
156     // switch LEDS off
157     ADC_init();
158     // initialize ADC

160     LED1_OFF;
161     LED2_OFF;
162     while(1)
163     {
164       RTC_compare(&RTC_tmp);
165       // look if real time clock values changed

167       if (BATMON_control_reg & GLOBAL_STRING_RECEIVED)
168       // string from UART received
169       {
170         UART_error = UART0_cmp_string();
171         // compare string with valid commands

173         if (UART_error)
174           UART0_send_text("NOK\r\n");
175         else
176           UART0_send_text("OK\r\n");

178         BATMON_control_reg &= !GLOBAL_STRING_RECEIVED;
179         // clear global value
180       }

182       if (BATMON_control_reg & VALID_DATA_RECEIVED)
183       // data received from sensor
184       {

186         if (scan_mode < 1)
187         {

189 //          RADIO_store_frame();
190           // store valid frame in flash
191         }
192         RADIO_enable();
193         // enable radio module
194         if (show_sensornmb == 0)
195         {
196           LCD_send_sensor(RADIO_frame.frame_byte[3], RADIO_frame.frame_byte[6],
                  RADIO_frame.frame_byte[7]);
197           // show sensor address on LCD was 3
198         }


201         BATMON_control_reg &= !VALID_DATA_RECEIVED;
202           // clear global value
203       }

205       if (BATMON_control_reg & CLEAR_FIRST_LCD_ROW)
206         // show sensor data in first LCD row
207       {

209         LCD_show_voltage(show_sensornmb);
```

```
      BATMON_control_reg &= !CLEAR_FIRST_LCD_ROW;
    }

    if((BATMON_control_reg & B1) ||
        (BATMON_control_reg & B2))
        // Button pressed jump into menu
    {
      BATMON_control_reg &= ~(B1|B2|B3);
      // clear button state
      BATMON_menu_options();
    }
    if(BATMON_control_reg & B3)
    // Button pressed show next info menu
    {
      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;
      }
    }
  }
  return (0);
}
```

Listing 6: Quellcode des Datenloggers (adc.c)

```
/*----------------------------------------------------------------------------------
  Project:      ADC functions
  Used components:    MSP430-169STK
  Date:       11/04/2009
  Last update:      12/02/2010
  Author:       Alexander Hoops
------------------------------------------------------------------------------------

------------------------------------------------------------------------------------
  Headerfiles
--------------------------------------------------------------------------------*/
#include <ISR.h>
#include <main.h>
#include <signal.h>
#include <msp430x16x.h>
#include <MSP_functions.h>
#include <ADC.h>

/*----------------------------------------------------------------------------------
  Init ADC
--------------------------------------------------------------------------------*/
void ADC_init(void)
/*------------------------------------------------------------------------------*/
{
  P6SEL = 0x0f; /* P6.0 from P6.3 ADC12 function */
  ADC12CTL0 = 0x0000;
  ADC12CTL1 = 0x0000;
  ADC12MCTL2 = 0x0000;
  ADC12IE = 0x0000;
  ADC12CTL1 = CSTARTADD_0 + ADC12DIV_4 + SHP + CONSEQ_3;
  /* Conv start add = 0, use SH,  */
  ADC12MCTL0 = SREF_1 + INCH_0;
  ADC12MCTL1 = SREF_1 + INCH_1;
```

```
34    /* Vr+ = Vref+ and Vr- = AVss, Input Channel = A2 */
35    ADC12IE = 0x03;
36    /* enable Interrupt A0-1 */
37    _EINT();
38    ADC12CTL0 = SHT0_12 + REF2_5V + REFON + ADC12ON + MSC;
39    /* SH-Time 1024*ADC12CLK, URef+=2,5V, Uref ON, ADC12 ON */
40  }

42  /*---------------------------------------------------------------------------------
43    Start ADC
44  -------------------------------------------------------------------------------*/
45  void ADC_start(void)
46  /*-------------------------------------------------------------------------------*/
47  {
48    ADC12CTL0 |= ADC12SC + ENC;
49    /* Enable conversion, Start sample and conversion */
50  }

52  /*---------------------------------------------------------------------------------
53    Stop ADC
54  -------------------------------------------------------------------------------*/
55  void ADC_stop(void)
56  /*-------------------------------------------------------------------------------*/
57  {
58    ADC12CTL0 &= ~(ADC12SC + ENC);
59    /* Disable conversion, Start sample and conversion */
60    LED1_OFF;
61  }

63  /*---------------------------------------------------------------------------------
64    Disable ADC
65  -------------------------------------------------------------------------------*/
66  void ADC_disable(void)
67  /*-------------------------------------------------------------------------------*/
68  {
69    ADC12CTL0 &= ~(ADC12ON + REFON);
70    /* ADCOFF + U REF OFF */
71  }

73  /*---------------------------------------------------------------------------------
74    Check BAT
75  -------------------------------------------------------------------------------*/
76  void Check_BAT(void)
77  /*-------------------------------------------------------------------------------*/
78  {
79    unsigned char v, t=0;
80    unsigned long test_value, test_value_2;
81    static unsigned char index_last_minutes_low_consumption = 0;
82    static unsigned char Capacity_write = 0;


85    // check consumption of last minute > 0,2 AH
86    if (AH_last_minute < 20)
87    {
88      index_last_minutes_low_consumption++;
89      if (index_last_minutes_low_consumption >= 3)
90      {
91        index_last_minutes_low_consumption = 3;
92        Capacity_write = 1;
93        // write voltage based cap. in main cap.
```

```
      }
    }
    else
    {
      index_last_minutes_low_consumption = 0;
      Capacity_write = 0;
    }
    // find depth cell
    test_value = (SENSOR_last_data[3][t]<<8);
    test_value |= (SENSOR_last_data[4][t]);

    for (v=1;v<SENSOR_active;v++)
    {
      test_value_2 = (SENSOR_last_data[3][v]<<8);
      test_value_2 |= (SENSOR_last_data[4][v]);
      if (test_value_2 < test_value)
      {
        t=v;
        test_value = test_value_2;
      }
    }

    // check if lowest is depth discharged => capacity = 0
    if ((test_value/40.96) < CELL_VOLT_MIN)
    {
      U_MASTER_CAPACITY = 0;
    }
    // voltage value from lowest sensor over max voltage (charging for example)
    // capacity = 100
    else if ((test_value/40.96) > CELL_VOLT_MAX)
    {
      U_MASTER_CAPACITY = 100;
    }
    // generate capacity based on voltage value from lowest cell
    else
    {
      U_MASTER_CAPACITY = (((test_value/40.96)-CELL_VOLT_MIN)/
            (CELL_VOLT_MAX-CELL_VOLT_MIN)*100);
    }
    // if low current consumption write voltage based capacity to main cap.
    if (Capacity_write == 1)
    {

      AH_MASTER = U_MASTER_CAPACITY * BATT_CAPACITY / 44 ;
      Capacity_write = 0;
    }
  }
```

Listing 7: Quellcode des Datenloggers (Flash.c)

```
/*--------------------------------------------------------------------------------
  Project:      MSP430-169STK onboard nand-flash functions
  Used components:    MSP430-169STK
  Date:       12/01/2007
  Last update:      09/05/2008
  Author:       Tobias Krannich
  Modified:     Stephan Plaschke

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

```
---------------------------------------------------------------------------

---------------------------------------------------------------------------
  Headerfiles
-------------------------------------------------------------------------*/
#include <main.h>
#include <Flash.h>
#include <stdio.h>
#include <lcd16x2.h>
#include <msp430x16x.h>
#include <MSP_functions.h>



static unsigned char *FLASH_clear_frame[] = {
    "                ",
    "  Clear flash   "
};

//----------------NAND FLASH---------------------------------------------------
//---------------------------------------------------------------------------
void init_Flash (void)
//---------------------------------------------------------------------------
{
  P2OUT = 0x07;                    //NAND FLASH ini
    P2DIR = 0x1F;
}

//---------------------------------------------------------------------------
// pull flash pins to inactive condition
void Inactive_Flash(void)
//---------------------------------------------------------------------------
{
  IO_DIR=INPUT;           //IO is inputs
  _CE_OFF;                //=1
  _RE_OFF;                //=1
  _WE_OFF;                //=1
  ALE_OFF;                //=0
  CLE_OFF;                //=0
}

//---------------------------------------------------------------------------
void Write_Data(unsigned char a)
//---------------------------------------------------------------------------
{
  IO_DIR=OUTPUT;          //IO is outputs
  _WE_ON;
  OUT_PORT=a;
  _WE_OFF;                //latch data
}

//---------------------------------------------------------------------------
unsigned char Read_Data(void)
//---------------------------------------------------------------------------
{
  unsigned char f;
  IO_DIR=INPUT;           //IO is inputs
  _RE_ON;
  f=IN_PORT;
  _RE_OFF;                //read data
```

```
71    return (f);
72  }

74  //-----------------------------------------------------------------------------------
75  unsigned char Programm_Bytes(unsigned char COL_ADD,
76                               unsigned char ROW_ADDL,
77                               unsigned char ROW_ADDH,
78                               unsigned char NUMBER)
79  {
80  //-----------------------------------------------------------------------------------
81          unsigned char k, l;
82          Inactive_Flash();
83      CLE_ON;
84      _CE_ON;
85      Write_Data(WRITE_PAGE);
86      CLE_OFF;
87      ALE_ON;
88          Write_Data(COL_ADD);
89          Write_Data(ROW_ADDL);
90          Write_Data(ROW_ADDH);
91          ALE_OFF;
92          for (k=0; k != NUMBER; k++)
93            {
94               l=WRITE_BUF[k];
95               Write_Data(l);
96            }
97          CLE_ON;
98          Write_Data(WRITE_AKN);
99          while ((R_B) == 0);
100         Write_Data(READ_STATUS);
101         CLE_OFF;
102         l = Read_Data();
103         Inactive_Flash();
104         return(l);
105 }

107 //-----------------------------------------------------------------------------------
108 void Read_Bytes (unsigned char COL_ADD,
109                  unsigned char ROW_ADDL,
110                  unsigned char ROW_ADDH,
111                  unsigned char NUMBER)
112 {
113 //-----------------------------------------------------------------------------------
114         unsigned char n, r;
115         Inactive_Flash();
116         CLE_ON;
117         _CE_ON;
118         Write_Data(READ_0);
119         CLE_OFF;
120         ALE_ON;
121         Write_Data(COL_ADD);
122         Write_Data(ROW_ADDL);
123         Write_Data(ROW_ADDH);
124         ALE_OFF;
125         while ((R_B) == 0);
126         for (n=0; n != NUMBER; n++)
127           {
128              r=Read_Data();
129              READ_BUF[n] = r;
130           }
```

```
131         Inactive_Flash();
132 }

134 //--------------------------------------------------------------------------------
135 void Data_Struct_To_Write_Buffer(DATA_STRUCT *pData)
136 //--------------------------------------------------------------------------------
137 {
138   WRITE_BUF[0] = pData->C0;
139   WRITE_BUF[1] = pData->C1;
140   WRITE_BUF[2] = pData->C2;
141   WRITE_BUF[3] = pData->C3;
142   WRITE_BUF[4] = pData->C4;
143   WRITE_BUF[5] = pData->C5;
144   WRITE_BUF[6] = pData->C6;
145   WRITE_BUF[7] = pData->C7;
146   WRITE_BUF[8] = pData->C8;
147   WRITE_BUF[9] = pData->C9;
148   WRITE_BUF[10] = pData->C10;
149   WRITE_BUF[11] = pData->C11;
150   WRITE_BUF[12] = pData->C12;
151   WRITE_BUF[13] = pData->C13;
152   WRITE_BUF[14] = pData->C14;
153   WRITE_BUF[15] = pData->C15;
154 }

156 //--------------------------------------------------------------------------------
157 void Val_To_Data(DATA_STRUCT *pData, unsigned long i)
158 //--------------------------------------------------------------------------------
159 {
160   if(i >= 527)
161   {
162     _NOP();
163   }
164   pData->C0 = (unsigned char)((i>>0) & 0xFF);
165   pData->C1 = (unsigned char)((i>>8) & 0xFF);
166   pData->C2 = (unsigned char)((i>>16) & 0xFF);
167   pData->C3 = (unsigned char)((i>>24) & 0xFF);
168 }

170 //--------------------------------------------------------------------------------
171 // Es wird der Block gelöscht, in dem die Adresse liegt
172 unsigned char Erase_Flash (unsigned char BLOCK_ADDL, unsigned char BLOCK_ADDH)
173 //--------------------------------------------------------------------------------
174 {
175         unsigned char m;
176         Inactive_Flash();
177     CLE_ON;
178     _CE_ON;
179     Write_Data(ERASE_BLOCK);
180     CLE_OFF;
181     ALE_ON;
182         Write_Data(BLOCK_ADDL);
183         Write_Data(BLOCK_ADDH);
184         ALE_OFF;
185         CLE_ON;
186         Write_Data(ERASE_AKN);
187         while ((R_B) == 0);
188         Write_Data(READ_STATUS);
189         CLE_OFF;
190         m = Read_Data();
```

```
191         Inactive_Flash();
192         return (m);
193 }

195 //-----------------------------------------------------------------------------------
196 DATA_STRUCT * Read_Ring_Flash(void)
197 //-----------------------------------------------------------------------------------
198 {
199   //unsigned short Index_Last = 0;
200   unsigned short ADD = 0;
201   unsigned char ADDH = 0;
202   unsigned char ADDL = 0;
203   unsigned char COL = 0;
204   static DATA_STRUCT Data = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
205   DATA_STRUCT *pData = &Data;

207   // Letztes Element im Ringbuffer ermitteln
208   if(Ring_Full == 0)  // Ringbuffer hatte noch keinen Überlauf
209   {
210     // Keine Daten im Ringspeicher???
211     if(Count_Ring == 0)
212     {
213       return NULL;
214     }
215     Index_Last = Index_Ring - Count_Ring;
216   }
217   else // Es gab schon einen Überlauf -> lese ab dem ersten Struct
218   {
219     // Keine Daten im Ringspeicher???
220     if((Count_Ring - 255) == 0) //
221     {
222       return NULL;
223     }
224     Index_Last = Index_Ring - (Count_Ring + 1 - 256)/* -
225     (Index_Ring%256)*/; // +1 um auf 0x10000 zu kommen
226   }
227   Count_Ring--;

229   // Adresse berechnen
230   ADD = Index_Last / 16;
231   COL = Index_Last % 16;
232   ADDL = (unsigned char)(ADD & 0xFF);
233   ADDH = (unsigned char)((ADD>>8) & 0xFF);

235   // Speicher auslesen
236   Read_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);

238   // Read_BUF in das Datenstruct schreiben
239   pData->C0  = READ_BUF[0];
240   pData->C1  = READ_BUF[1];
241   pData->C2  = READ_BUF[2];
242   pData->C3  = READ_BUF[3];
243   pData->C4  = READ_BUF[4];
244   pData->C5  = READ_BUF[5];
245   pData->C6  = READ_BUF[6];
246   pData->C7  = READ_BUF[7];
247   pData->C8  = READ_BUF[8];
248   pData->C9  = READ_BUF[9];
249   pData->C10 = READ_BUF[10];
250   pData->C11 = READ_BUF[11];
```

```
251   pData->C12 = READ_BUF[12];
252   pData->C13 = READ_BUF[13];
253   pData->C14 = READ_BUF[14];
254   pData->C15 = READ_BUF[15];

257   return pData;
258 }

260 //-----------------------------------------------------------------------------------
261 //-----------------------------------------------------------------------------------
262 //    These functions are modified/created by Stephan Plaschke
263 //-----------------------------------------------------------------------------------
264 //-----------------------------------------------------------------------------------

266 //-----------------------------------------------------------------------------------
267 void Erase_Flash_All(void)
268 //-----------------------------------------------------------------------------------
269 {
270   int i = 0;
271   unsigned char j=0;
272   unsigned char ADDL;
273   unsigned char ADDH;

275   // show splash screen on LCD and send via UART
276   LCD_send_cmd(LCD_LINE1);
277   LCD_send_text(FLASH_clear_frame[1]);
278   UART0_send('.');

280   for(i=0; i<0x1000; i++)
281   {
282     //ADD = i * 16;
283     ADDL = (unsigned char)(i & 0xFF);
284     ADDH = (unsigned char)((i>>8) & 0xFF);
285     Erase_Flash(ADDL,ADDH);

287     // show splash screen on LCD and send via UART
288     if (!(i%200))
289     {
290       if(j^=0x01)
291       {
292         LCD_send_cmd(LCD_LINE1);
293         LCD_send_text(FLASH_clear_frame[1]);
294         UART0_send('.');
295       }
296       else
297       {
298         LCD_send_cmd(LCD_LINE1);
299         LCD_send_text(FLASH_clear_frame[0]);
300       }
301     }
302   }
303   // set all index to the first frame in flash
304   Count_Ring = 0;
305   Ring_Full = 0;
306   // set pointer on the address 256, first block used for controll data
307   Index_Ring = 256;
308   Index_Last = 256;

310   LCD_send_cmd(LCD_LINE1);
```

```
311    LCD_send_text(FLASH_clear_frame[0]);
312    UART0_send_text("\r\n");

314    RADIO_init_frame();      // set first frame
315  }

317  //----------------------------------------------------------------------------------
318  void Write_Ring_Flash(DATA_STRUCT *pData)
319  //----------------------------------------------------------------------------------
320  {
321    unsigned char ADDL;
322    unsigned char ADDH;
323    unsigned short ADD;
324    unsigned char COL;


327    ADD = Index_Ring / 16;
328    COL = Index_Ring % 16;
329    ADDL = (unsigned char)(ADD & 0xFF);
330    ADDH = (unsigned char)((ADD>>8) & 0xFF);

332    // Lösche den nächsten Block
333    if( (Index_Ring%256) == 0)
334    {
335      Erase_Flash(ADDL,ADDH);
336    }

338    Data_Struct_To_Write_Buffer(pData);
339    Programm_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);

341    if(Count_Ring < 65279) // address space: 65535 - 256
342    {
343      Count_Ring++;
344    }
345    else
346    {
347      Ring_Full = 1;
348    }

350    Index_Ring++;
351    if(Index_Ring >= 65535)
352    {
353      Index_Ring = 256;
354           // set pointer on the address 256, first block used for controll data
355    }
356  }

358  //----------------------------------------------------------------------------------
359  void FLASH_Write_Controll_Data(void)
360  //----------------------------------------------------------------------------------
361  {
362    Erase_Flash(0,0);

364    WRITE_BUF[0]  = (unsigned char)(Index_Ring >> 8);
365    WRITE_BUF[1]  = (unsigned char)Index_Ring;
366    WRITE_BUF[2]  = (unsigned char)(Index_Last >> 8);
367    WRITE_BUF[3]  = (unsigned char)Index_Last;
368    WRITE_BUF[4]  = (unsigned char)(Count_Ring >> 8);
369    WRITE_BUF[5]  = (unsigned char)Count_Ring;
370    WRITE_BUF[6]  = Ring_Full;
```

```
371    WRITE_BUF[7]  = SENSOR_active;
372    WRITE_BUF[8]  = SENSOR_address[0];
373    WRITE_BUF[9]  = SENSOR_address[1];
374    WRITE_BUF[10] = SENSOR_address[2];
375    WRITE_BUF[11] = SENSOR_address[3];
376    WRITE_BUF[12] = SENSOR_address[4];
377    WRITE_BUF[13] = SENSOR_address[5];
378    WRITE_BUF[14] = SENSOR_address[6];
379    WRITE_BUF[15] = SENSOR_address[7];

381    Programm_Bytes(0,0,0,BUF_SIZE);

383    WRITE_BUF[0]  = SENSOR_address[8];
384    WRITE_BUF[1]  = SENSOR_address[9];
385    WRITE_BUF[2]  = SENSOR_address[10];
386    WRITE_BUF[3]  = SENSOR_address[11];
387    WRITE_BUF[4]  = SENSOR_address[12];
388    WRITE_BUF[5]  = SENSOR_address[13];
389    WRITE_BUF[6]  = SENSOR_address[14];
390    WRITE_BUF[7]  = SENSOR_address[15];
391    WRITE_BUF[8]  = SENSOR_address[16];
392    WRITE_BUF[9]  = SENSOR_address[17];
393    WRITE_BUF[10] = SENSOR_address[18];
394    WRITE_BUF[11] = SENSOR_address[19];
395    WRITE_BUF[12] = SENSOR_address[20];
396    WRITE_BUF[13] = SENSOR_address[21];
397    WRITE_BUF[14] = SENSOR_address[22];
398    WRITE_BUF[15] = SENSOR_address[23];

400    Programm_Bytes(0,1,0,BUF_SIZE);

402    WRITE_BUF[0]  = SENSOR_address[24];
403    WRITE_BUF[1]  = SENSOR_address[25];
404    WRITE_BUF[2]  = SENSOR_address[26];
405    WRITE_BUF[3]  = SENSOR_address[27];
406    WRITE_BUF[4]  = SENSOR_address[28];
407    WRITE_BUF[5]  = SENSOR_address[29];
408    WRITE_BUF[6]  = SENSOR_address[30];
409    WRITE_BUF[7]  = SENSOR_address[31];
410    WRITE_BUF[8]  = SENSOR_address[32];
411    WRITE_BUF[9]  = SENSOR_address[33];
412    WRITE_BUF[10] = SENSOR_address[34];
413    WRITE_BUF[11] = SENSOR_address[35];
414    WRITE_BUF[12] = SENSOR_address[36];
415    WRITE_BUF[13] = SENSOR_address[37];
416    WRITE_BUF[14] = SENSOR_address[38];
417    WRITE_BUF[15] = SENSOR_address[39];

419    Programm_Bytes(0,2,0,BUF_SIZE);

421    WRITE_BUF[0]  = SENSOR_temp_cal_b[1];
422    WRITE_BUF[1]  = SENSOR_temp_cal_b[2];
423    WRITE_BUF[2]  = SENSOR_temp_cal_b[3];
424    WRITE_BUF[3]  = SENSOR_temp_cal_b[4];
425    WRITE_BUF[4]  = SENSOR_temp_cal_b[5];
426    WRITE_BUF[5]  = SENSOR_temp_cal_b[6];
427    WRITE_BUF[6]  = SENSOR_temp_cal_b[7];
428    WRITE_BUF[7]  = SENSOR_temp_cal_b[8];
429    WRITE_BUF[8]  = SENSOR_temp_cal_b[9];
430    WRITE_BUF[9]  = SENSOR_temp_cal_b[10];
```

```
431    WRITE_BUF[10] = SENSOR_temp_cal_b[11];
432    WRITE_BUF[11] = SENSOR_temp_cal_b[12];
433    WRITE_BUF[12] = SENSOR_temp_cal_b[13];
434    WRITE_BUF[13] = SENSOR_temp_cal_b[14];
435    WRITE_BUF[14] = SENSOR_temp_cal_b[15];
436    WRITE_BUF[15] = SENSOR_temp_cal_b[16];

438    Programm_Bytes(0,3,0,BUF_SIZE);

440    WRITE_BUF[0]  = SENSOR_temp_cal_b[17];
441    WRITE_BUF[1]  = SENSOR_temp_cal_b[18];
442    WRITE_BUF[2]  = SENSOR_temp_cal_b[19];
443    WRITE_BUF[3]  = SENSOR_temp_cal_b[20];
444    WRITE_BUF[4]  = SENSOR_temp_cal_b[21];
445    WRITE_BUF[5]  = SENSOR_temp_cal_b[22];
446    WRITE_BUF[6]  = SENSOR_temp_cal_b[23];
447    WRITE_BUF[7]  = SENSOR_temp_cal_b[24];
448    WRITE_BUF[8]  = SENSOR_temp_cal_b[25];
449    WRITE_BUF[9]  = SENSOR_temp_cal_b[26];
450    WRITE_BUF[10] = SENSOR_temp_cal_b[27];
451    WRITE_BUF[11] = SENSOR_temp_cal_b[28];
452    WRITE_BUF[12] = SENSOR_temp_cal_b[29];
453    WRITE_BUF[13] = SENSOR_temp_cal_b[30];
454    WRITE_BUF[14] = SENSOR_temp_cal_b[31];
455    WRITE_BUF[15] = SENSOR_temp_cal_b[32];

457    Programm_Bytes(0,4,0,BUF_SIZE);

459    WRITE_BUF[0]  = SENSOR_temp_cal_b[33];
460    WRITE_BUF[1]  = SENSOR_temp_cal_b[34];
461    WRITE_BUF[2]  = SENSOR_temp_cal_b[35];
462    WRITE_BUF[3]  = SENSOR_temp_cal_b[36];
463    WRITE_BUF[4]  = SENSOR_temp_cal_b[37];
464    WRITE_BUF[5]  = SENSOR_temp_cal_b[38];
465    WRITE_BUF[6]  = SENSOR_temp_cal_b[39];
466    WRITE_BUF[7]  = SENSOR_temp_cal_b[40];
467    WRITE_BUF[8]  = (unsigned char)(AH_MASTER>>8);
468    WRITE_BUF[9]  = (unsigned char)AH_MASTER;
469    WRITE_BUF[10] = (unsigned char)(CELL_VOLT_MAX>>8);
470    WRITE_BUF[11] = (unsigned char)CELL_VOLT_MAX;
471    WRITE_BUF[12] = (unsigned char)(CELL_VOLT_MIN>>8);
472    WRITE_BUF[13] = (unsigned char)CELL_VOLT_MIN;
473    WRITE_BUF[14] = (unsigned char)(BATT_CAPACITY>>8);
474    WRITE_BUF[15] = (unsigned char)BATT_CAPACITY;

476    Programm_Bytes(0,5,0,BUF_SIZE);

478    WRITE_BUF[0]  = (unsigned char)(CURRENT_tara>>8);
479    WRITE_BUF[1]  = (unsigned char)CURRENT_tara;
480    WRITE_BUF[2]  = CURRENT_INVERT;
481    WRITE_BUF[3]  = 0;
482    WRITE_BUF[4]  = 0;
483    WRITE_BUF[5]  = 0;
484    WRITE_BUF[6]  = 0;
485    WRITE_BUF[7]  = 0;
486    WRITE_BUF[8]  = 0;
487    WRITE_BUF[9]  = 0;
488    WRITE_BUF[10] = 0;
489    WRITE_BUF[11] = 0;
490    WRITE_BUF[12] = 0;
```

```
491    WRITE_BUF[13] = 0;
492    WRITE_BUF[14] = 0;
493    WRITE_BUF[15] = 0;

495    Programm_Bytes(0,6,0,BUF_SIZE);

497  }
498  //-----------------------------------------------------------------------------------
499  void FLASH_Read_Controll_Data(void)
500  //-----------------------------------------------------------------------------------
501  {
502    //unsigned short i;
503    // Speicher auslesen
504    Read_Bytes(0,0,0,BUF_SIZE);

506    Index_Ring        = (READ_BUF[0] << 8) | READ_BUF[1];
507    Index_Last        = (READ_BUF[2] << 8) | READ_BUF[3];
508    Count_Ring        = (READ_BUF[4] << 8) | READ_BUF[5];
509    Ring_Full         = READ_BUF[6];
510    SENSOR_active      = READ_BUF[7];
511    SENSOR_address[0]  = READ_BUF[8];
512    SENSOR_address[1]  = READ_BUF[9];
513    SENSOR_address[2]  = READ_BUF[10];
514    SENSOR_address[3]  = READ_BUF[11];
515    SENSOR_address[4]  = READ_BUF[12];
516    SENSOR_address[5]  = READ_BUF[13];
517    SENSOR_address[6]  = READ_BUF[14];
518    SENSOR_address[7]  = READ_BUF[15];

520    // Speicher auslesen
521    Read_Bytes(0,1,0,BUF_SIZE);

523    SENSOR_address[8]  = READ_BUF[0];
524    SENSOR_address[9]  = READ_BUF[1];
525    SENSOR_address[10] = READ_BUF[2];
526    SENSOR_address[11] = READ_BUF[3];
527    SENSOR_address[12] = READ_BUF[4];
528    SENSOR_address[13] = READ_BUF[5];
529    SENSOR_address[14] = READ_BUF[6];
530    SENSOR_address[15] = READ_BUF[7];
531    SENSOR_address[16] = READ_BUF[8];
532    SENSOR_address[17] = READ_BUF[9];
533    SENSOR_address[18] = READ_BUF[10];
534    SENSOR_address[19] = READ_BUF[11];
535    SENSOR_address[20] = READ_BUF[12];
536    SENSOR_address[21] = READ_BUF[13];
537    SENSOR_address[22] = READ_BUF[14];
538    SENSOR_address[23] = READ_BUF[15];

540    // Speicher auslesen
541    Read_Bytes(0,2,0,BUF_SIZE);

543    SENSOR_address[24] = READ_BUF[0];
544    SENSOR_address[25] = READ_BUF[1];
545    SENSOR_address[26] = READ_BUF[2];
546    SENSOR_address[27] = READ_BUF[3];
547    SENSOR_address[28] = READ_BUF[4];
548    SENSOR_address[29] = READ_BUF[5];
549    SENSOR_address[30] = READ_BUF[6];
550    SENSOR_address[31] = READ_BUF[7];
```

```
551   SENSOR_address[32] = READ_BUF[8];
552   SENSOR_address[33] = READ_BUF[9];
553   SENSOR_address[34] = READ_BUF[10];
554   SENSOR_address[35] = READ_BUF[11];
555   SENSOR_address[36] = READ_BUF[12];
556   SENSOR_address[37] = READ_BUF[13];
557   SENSOR_address[38] = READ_BUF[14];
558   SENSOR_address[39] = READ_BUF[15];

560   // Speicher auslesen
561   Read_Bytes(0,3,0,BUF_SIZE);

563   SENSOR_temp_cal_b[1] = READ_BUF[0];
564   SENSOR_temp_cal_b[2] = READ_BUF[1];
565   SENSOR_temp_cal_b[3] = READ_BUF[2];
566   SENSOR_temp_cal_b[4] = READ_BUF[3];
567   SENSOR_temp_cal_b[5] = READ_BUF[4];
568   SENSOR_temp_cal_b[6] = READ_BUF[5];
569   SENSOR_temp_cal_b[7] = READ_BUF[6];
570   SENSOR_temp_cal_b[8] = READ_BUF[7];
571   SENSOR_temp_cal_b[9] = READ_BUF[8];
572   SENSOR_temp_cal_b[10] = READ_BUF[9];
573   SENSOR_temp_cal_b[11] = READ_BUF[10];
574   SENSOR_temp_cal_b[12] = READ_BUF[11];
575   SENSOR_temp_cal_b[13] = READ_BUF[12];
576   SENSOR_temp_cal_b[14] = READ_BUF[13];
577   SENSOR_temp_cal_b[15] = READ_BUF[14];
578   SENSOR_temp_cal_b[16] = READ_BUF[15];

580   // Speicher auslesen
581   Read_Bytes(0,4,0,BUF_SIZE);

583   SENSOR_temp_cal_b[17] = READ_BUF[0];
584   SENSOR_temp_cal_b[18] = READ_BUF[1];
585   SENSOR_temp_cal_b[19] = READ_BUF[2];
586   SENSOR_temp_cal_b[20] = READ_BUF[3];
587   SENSOR_temp_cal_b[21] = READ_BUF[4];
588   SENSOR_temp_cal_b[22] = READ_BUF[5];
589   SENSOR_temp_cal_b[23] = READ_BUF[6];
590   SENSOR_temp_cal_b[24] = READ_BUF[7];
591   SENSOR_temp_cal_b[25] = READ_BUF[8];
592   SENSOR_temp_cal_b[26] = READ_BUF[9];
593   SENSOR_temp_cal_b[27] = READ_BUF[10];
594   SENSOR_temp_cal_b[28] = READ_BUF[11];
595   SENSOR_temp_cal_b[29] = READ_BUF[12];
596   SENSOR_temp_cal_b[30] = READ_BUF[13];
597   SENSOR_temp_cal_b[31] = READ_BUF[14];
598   SENSOR_temp_cal_b[32] = READ_BUF[15];

600   // Speicher auslesen
601   Read_Bytes(0,5,0,BUF_SIZE);

603   SENSOR_temp_cal_b[33] = READ_BUF[0];
604   SENSOR_temp_cal_b[34] = READ_BUF[1];
605   SENSOR_temp_cal_b[35] = READ_BUF[2];
606   SENSOR_temp_cal_b[36] = READ_BUF[3];
607   SENSOR_temp_cal_b[37] = READ_BUF[4];
608   SENSOR_temp_cal_b[38] = READ_BUF[5];
609   SENSOR_temp_cal_b[39] = READ_BUF[6];
610   SENSOR_temp_cal_b[40] = READ_BUF[7];
```

```
611   AH_MASTER          = (READ_BUF[8]<<8) | READ_BUF[9];
612   CELL_VOLT_MAX       = (READ_BUF[10]<<8) | READ_BUF[11];
613   CELL_VOLT_MIN       = (READ_BUF[12]<<8) | READ_BUF[13];
614   BATT_CAPACITY       = (READ_BUF[14]<<8) | READ_BUF[15];

616   // Speicher auslesen
617   Read_Bytes(0,6,0,BUF_SIZE);

619   CURRENT_tara       = (READ_BUF[0]<<8) | READ_BUF[1];
620   CURRENT_INVERT      = READ_BUF[2];
621 //   0 = READ_BUF[3];
622 //   0 = READ_BUF[4];
623 //   0 = READ_BUF[5];
624 //   0 = READ_BUF[6];
625 //   0 = READ_BUF[7];
626 //   0 = READ_BUF[8];
627 //   0 = READ_BUF[9];
628 //   0 = READ_BUF[10];
629 //   0 = READ_BUF[11];
630 //   0 = READ_BUF[12];
631 //   0 = READ_BUF[13];
632 //   0 = READ_BUF[14];
633 //   0 = READ_BUF[15];
634 //   0 = READ_BUF[16];

636 }

638 /*---------------------------------------------------------------------------------
639   read data at startup
640 -------------------------------------------------------------------------------*/
641 void FLASH_read_at_startup(void)
642 /*-----------------------------------------------------------------------------*/
643 {
644   LCD_send_cmd(LCD_LINE1);
645   LCD_send_text("Read contr.data ");
646   LCD_send_cmd(LCD_LINE2);
647   LCD_send_text("YES    NO        ");

649   while(1)
650   {
651     if(BATMON_control_reg & B1)
652     // Button 1 = YES, read controll data from flash
653     {
654       UART0_send_text("Read data\r\n");
655       FLASH_Read_Controll_Data();
656       BATMON_control_reg &= ~(B1|B2|B3);
657       // clear button state
658       break;
659       // exit while loop
660     }
661     if(BATMON_control_reg & B2)
662       // Button 2 = NO, write init sequence
663     {
664       RADIO_init_frame();
665       // write first frame with actual time and 0 as data
666       BATMON_control_reg &= ~(B1|B2|B3);
667       // clear button state
668       break;
669       // exit while loop
670     }
```

```
671     }
672 }
```

Listing 8: Quellcode des Datenloggers (ISR.c)

```
1  /*-------------------------------------------------------------------------------------
2    Project:       ISR functions
3    Used components:    MSP430-169STK
4    Date:        12/15/2007
5    Last update:       04/04/2008
6    Author:        Stephan Plaschke

8    Modified by:     Alexander Hoops
9    Last modification:  12/02/2010
10   Modified by:     Simon Püttjer
11   Last modification:  05/05/2011
12 -------------------------------------------------------------------------------------

14 -------------------------------------------------------------------------------------
15   Headerfiles
16 -----------------------------------------------------------------------------------*/
17 #include <ISR.h>
18 #include <ADC.h>
19 #include <main.h>
20 #include <signal.h>
21 #include <stdio.h>
22 #include <lcd16x2.h>
23 #include <msp430x16x.h>
24 #include <MSP_functions.h>
25 #include <uart_menu.h>
26 #include <math.h>

28 /*-------------------------------------------------------------------------------------
29   Global variables
30 -----------------------------------------------------------------------------------*/
31 extern unsigned char BUTTON_state;    // actual button state
32 unsigned char BATMON_control_reg;   //control register
33 extern unsigned char RADIO_rx_state;    // data receive state
34 extern unsigned char global_string[MAX_STRING_LENGTH];
35 extern RADIO_FRAME_STRUCT RADIO_frame;  // received data frame
36 extern unsigned char SENSOR_addr_req[5];  // required sensor addresses
37 extern unsigned char SENSOR_addr_receiv[5]; // check all sensors send value
38 extern unsigned char SENSOR_address[40];  // stored sensor addresses
39 extern unsigned char SENSOR_last_data[8][40]; // last received data from each sensor
40 extern unsigned char CURRENT_INVERT;    // variable to setup sensor
41 extern unsigned long CURRENT_last_data;   //stored current value
42 extern unsigned char CURRENT_IN_OUT;    // 1 = charge 0 = discharge
43 extern unsigned long AH_last_minute;    // AH last minute
44 extern unsigned long AH_last_sec;   // AH last second
45 extern unsigned char AH_last_minute_IN_OUT; // 1 = charge 0 = discharge
46 extern unsigned long AH_MASTER;     // AH picker
47 extern unsigned char scan_mode;     // scan mode => 1
48 extern unsigned long temp_correct_value;  // temp value for calibration
49 extern signed char SENSOR_temp_cal_b[41]; // Sensor calibration data b

51 extern unsigned long CELL_VOLT_MAX;   // Cell voltage max 2.78V = 278
52 extern unsigned long CELL_VOLT_MIN;   // Cell voltage min 1.88V = 188
53 extern unsigned long BATT_CAPACITY;   // Battery capacity 500Ah = 500
54 extern unsigned char SENSOR_low;    // Sensor under min voltage
55 extern unsigned char SOC;     // State of charge
```

```
56  extern unsigned char SOH;      // State of health
57  extern int long lost_frames[6]; //number of lost frames for each sensor
58  extern unsigned char fer; //Frame error calc. control register

60  extern unsigned char CURRENT_channel;

62  //calculate current frame error
63  //------------------------------------------------------------------
64  void countFER(unsigned char sensor_number, unsigned char crc_error,
65      unsigned short bit_too_long, unsigned short bit_too_short,
66      unsigned short wrong_RunIn) {

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

71    switch (fer) {

73    //init frame error calc.
74    case 1:
75      fer = 2;
76    //frame error calc.
77    case 2:
78      temp_error = (short)(RADIO_frame.frame_byte[1]);
79      temp_error <<= 8;
80      temp_error |= (short)(RADIO_frame.frame_byte[2]);
81      temp_error_old = (short)(SENSOR_last_data[1][sensor_number]);
82      temp_error_old <<= 8;
83      temp_error_old |= (short)(SENSOR_last_data[2][sensor_number]);

85      //lost frames is the difference of the current and the last transmission count
86      lost_frames = temp_error - temp_error_old - 1;

88      //matlab binblock protocoll #<N><D><A>
89      UART0_send('#'); //doublecross terminates transmission
90      UART0_send('2'); //number of digits that follows in D
91      UART0_send('1'); //number of bytes that follows in A
92      UART0_send('0');
93      UART0_send(SENSOR_address[sensor_number]);
94      UART0_send((unsigned char) (bit_too_long >> 8));
95      UART0_send((unsigned char) (bit_too_long & 0x00FF));
96      UART0_send((unsigned char) (bit_too_short >> 8));
97      UART0_send((unsigned char) (bit_too_short & 0x00FF));
98      UART0_send((unsigned char) (wrong_RunIn >> 8));
99      UART0_send((unsigned char) (wrong_RunIn & 0x00FF));
100     UART0_send((unsigned char) (lost_frames >> 8));
101     UART0_send((unsigned char) (lost_frames & 0x00FF));
102     UART0_send(crc_error);

104     //send new line
105     UART0_send_text("\n");


108     break;
109   default:
110     fer = 0;
111   }

113 }

115 //------------------------------------------------------------------
```

```
118 /*-----------------------------------------------------------------------------------
119   Timer A0 interrupt service routine, Capture/compare0 interrupt  (bit recognition)
120 ---------------------------------------------------------------------------------*/
121 interrupt (TIMERA0_VECTOR) TIMER_A0(void)
122 /*-------------------------------------------------------------------------------*/
123 {
124   static unsigned int RADIO_capture_old, RADIO_capture_act;
125   static unsigned int RADIO_capture_interval;
126 //  unsigned long cal_temperature;
127   static unsigned int RADIO_bit_length, RADIO_one_bit_length;
128   static unsigned int RADIO_two_bit_length;
129   static unsigned int RADIO_three_bit_length, RADIO_four_bit_length;
130 //  unsigned long check_voltage;
131   static unsigned char RADIO_capture_counter;
132   static unsigned char Temp_cal_counter[41]=
133   {0,0,0,0,0,0,0,0,0,0,
134    0,0,0,0,0,0,0,0,0,0,
135    0,0,0,0,0,0,0,0,0,0,
136    0,0,0,0,0,0,0,0,0,0,
137    0};
138   signed long temp_diff, real_Temp;
139   unsigned char *address_sensor;
140   address_sensor = &SENSOR_address[0];
141   // compare values for the received address in global array,
142   // byte = position in array
143   unsigned char address_bit, address_byte;
144   unsigned char i, j, frame_error, ex, sens, sensor_calibrated;

146   //FER log
147   //------------------------------------------------------------
148   static unsigned short bit_too_short, bit_too_long;
149   static unsigned short wrong_RunIn;
150   static unsigned short crc_error[6] = {0}; //crc_error count for each sensor address
151   //------------------------------------------------------------

153   //calculate bit width
154   //---------------------------------------------------------------
155   RADIO_capture_old = RADIO_capture_act;
156   RADIO_capture_act = CCR0;



160   /* overflow occured (actual interrupt occured in next counter intervall)*/
161   if (RADIO_capture_act < RADIO_capture_old)
162     RADIO_capture_interval = ((65535-RADIO_capture_old)
163             + RADIO_capture_act);

165   /* calculate difference (no overflow occured) */
166   else

168     /* calculate difference */
169     RADIO_capture_interval = RADIO_capture_act - RADIO_capture_old;

171   //----------------------------------------------------------------

173   //case differentiation - bit recognition
174   //----------------------------------------------------------------
175   switch(RADIO_rx_state)
```

```
176   {
177   case SE_SEQ_PREPARE:

179     // first interrupt, nothing to do
180     RADIO_frame.frame_bit_counter = 0;  // actual bit position
181     RADIO_frame.frame_byte_counter = 0; // actual byte in frame
182     BATMON_control_reg &= !VALID_DATA_RECEIVED; // clear global value
183     RADIO_rx_state = SE_SEQ_START;  //start sequence next time
184     //break;  // exit switch-case block

186   case SE_SEQ_START:
187     // look if start sequence detected, SoF width is 700..1200us
188     if((RADIO_capture_interval < START_SEQ_LENGTH_MAX)
189       && (RADIO_capture_interval > START_SEQ_LENGTH_MIN))
190     {
191       RADIO_rx_state = SE_SEQ_PRE_WAIT; //set wait flag
192       RADIO_capture_counter = 14;
193       // waitcounter 14 bit until second byte received
194     }
195     break;  // exit switch-case block

197   case SE_SEQ_PRE_WAIT:
198     // check next 14 bits for valid length
199     /*if((RADIO_capture_interval < TWO_BIT_LENGTH)
200       && (RADIO_capture_interval > ONE_BIT_LENGTH))
201     {
202       RADIO_capture_counter--;
203       if(RADIO_capture_counter == 0)
204         RADIO_rx_state = SE_SEQ_DATA_START;
205     }
206     else {
207       RADIO_rx_state = SE_SEQ_PREPARE;
208       break;
209     }*/

211     RADIO_capture_counter--;

213     //calibrate bitrate - cal. bit pattern is 0x00,0x01
214     //

216    if (RADIO_capture_counter == 0)
217     {
218       RADIO_bit_length += RADIO_capture_interval;
219       RADIO_bit_length /= 2;

221       RADIO_one_bit_length   = RADIO_bit_length - 50;
222       RADIO_two_bit_length   = RADIO_bit_length + 50;
223       RADIO_three_bit_length = RADIO_bit_length * 2 - 50;
224       RADIO_four_bit_length  = RADIO_bit_length * 2 + 50;

226       RADIO_rx_state = SE_SEQ_DATA_START;
227     }
228     else
229     {
230       //floating average?
231       RADIO_bit_length += RADIO_capture_interval;
232       RADIO_bit_length /= 2;
233     }

235     break;  // exit switch-case block
```

```
237  case SE_SEQ_DATA_START:
238    // end of second byte (cal. pattern), check second byte (detect a '1' at the end)

240    //min < bitwidth < max double bitwidth --> '10' is detected case
241    if((RADIO_capture_interval < RADIO_four_bit_length)
242      && (RADIO_capture_interval > RADIO_three_bit_length))
243    {
244      RADIO_rx_state = SE_SEQ_RXDATA;
245      //enable receive data flag
246      RADIO_frame.frame_bit_counter = 1;
247      // actual bit position
248      RADIO_frame.frame_byte_counter = 0;
249      // actual byte in frame
250      RADIO_frame.frame_last_bit = 0;
251      // last bit state, equals actual bit
252      RADIO_frame.frame_byte[0] = 0x00;
253      // set actual bit
254    }
255    //nothing detected
256    else
257      RADIO_rx_state = SE_SEQ_PREPARE;
258      // second byte not 0x01, error
259      wrong_RunIn++;
260    break;  // exit switch-case block

262  case SE_SEQ_RXDATA:
263    // receive data, for this cases see documentation

265    if(RADIO_capture_interval < RADIO_one_bit_length)
266    // intervall to short, exit
267    {
268      bit_too_short++;
269      RADIO_rx_state = SE_SEQ_PREPARE;  //error start again
270      break;  // exit switch-case block
271    }

273    else if(RADIO_capture_interval < RADIO_two_bit_length)
274    // same value like the one before
275    {
276      RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8; //count every
         8 bit up
277      RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1; //go to next bit
278      RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] |= RADIO_frame.
         frame_last_bit; //set bit
279      RADIO_frame.frame_bit_counter++;  //count up bit counter
280    }

282    else if(RADIO_capture_interval < RADIO_three_bit_length)
283    {
284      if(RADIO_frame.frame_last_bit)
285      // a one and a following zero detected '10'
286      {
287        //set the '1'
288        RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
289        RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter]<<= 1;
290        RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] |= 0x01; //set bit
291        RADIO_frame.frame_bit_counter++;

293        //set the '0'
```

```
294         RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
295         RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1;
296         RADIO_frame.frame_last_bit = 0;
297         RADIO_frame.frame_bit_counter++;
298       }
299       else  // a '1' detected
300       {
301         RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
302         RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1;
303         RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] |= 0x01;
304         RADIO_frame.frame_last_bit = 1;
305         RADIO_frame.frame_bit_counter++;
306       }
307     }

309     else if(RADIO_capture_interval < RADIO_four_bit_length)
310       // a one with a following zero detected '10'
311     {
312       //set the '1'
313       RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
314       RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1;
315       RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] |= 0x01;
316       RADIO_frame.frame_bit_counter++;

318       //set the '0'
319       RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
320       RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1;
321       RADIO_frame.frame_bit_counter++;
322     }

324     else  // intervall to long, exit with error
325     {
326       if (RADIO_frame.frame_bit_counter == 87)
327       // last byte(CRC) is odd
328       {
329         RADIO_frame.frame_byte_counter = RADIO_frame.frame_bit_counter / 8;
330         RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] <<= 1;
331         RADIO_frame.frame_byte[RADIO_frame.frame_byte_counter] |= 0x01;
332         RADIO_frame.frame_bit_counter++;
333       }
334       //error
335       else
336       {
337         RADIO_rx_state = SE_SEQ_PREPARE;
338         bit_too_long++;
339         break;  // exit switch-case block
340       }
341     }

343     if (RADIO_frame.frame_bit_counter == 88)
344       // end of bitstream
345     {
346       i=0;  // init address counter

348       //scan mode
349       //-------------------------------------------------------
350       if (scan_mode == 1)
351           // Scan mode started
352       {
```

```
353          while((SENSOR_address[i] != 0 ) && (SENSOR_address[i] != RADIO_frame.frame_byte
                 [3]))
354          // check if address valid ( if it is already existent ), skip Address 0
355          {
356            i++;
357          }

359          //Sensor not yet existant - print info to uart
360          if (SENSOR_address[i] != RADIO_frame.frame_byte[3])
361          {
362            PRINTF_UART;
363            printf("Sensor: %2x => Adresse: %2x\r\n",
364              i+1, RADIO_frame.frame_byte[3]);
365          }

367          SENSOR_address[i] = RADIO_frame.frame_byte[3];  //save sensor address

369          //desired number of sensors found
370          if (i >= SENSOR_active-1)
371          {
372            scan_mode = 0;  //stop scan mode
373            LCD_send_cmd(LCD_LINE1);
374            UART_menu_stop();
375            // disable radio unit
376            LCD_send_text(" Scan complete  ");
377            UART0_send_text("Scan complete \r\n");
378            BATMON_control_reg &= ~VALID_DATA_RECEIVED;
379            // set global value
380            FLASH_Write_Controll_Data();  //write gathered control data to flash
381            RADIO_disable();  //stop reveiver
382            for (i=0; i<SENSOR_active; i++)
383            {
384              SENSOR_last_data[0][i] = SENSOR_address[i];
385            }
386            // set sensors active

388            switch(SENSOR_active)
389            {
390            case 12:
391              SENSOR_addr_req[0] = 0xFF;
392              // 12 sensors are configured
393              SENSOR_addr_req[1] = 0x0F;
394              SENSOR_addr_receiv[0] = 0xFF;
395              SENSOR_addr_receiv[1] = 0x0F;
396              break;
397            case 24:
398              SENSOR_addr_req[0] = 0xFF;
399              // 24 sensors are configured
400              SENSOR_addr_req[1] = 0xFF;
401              SENSOR_addr_req[2] = 0xFF;
402              SENSOR_addr_req[3] = 0x00;
403              SENSOR_addr_req[4] = 0x00;
404              SENSOR_addr_receiv[0] = 0xFF;
405              SENSOR_addr_receiv[1] = 0xFF;
406              SENSOR_addr_receiv[2] = 0xFF;
407              SENSOR_addr_receiv[3] = 0x00;
408              SENSOR_addr_receiv[4] = 0x00;
409              break;
410            case 40:
411              SENSOR_addr_req[0] = 0xFF;
```

```
412              // 40 sensors are configured
413              SENSOR_addr_req[1] = 0xFF;
414              SENSOR_addr_req[2] = 0xFF;
415              SENSOR_addr_req[3] = 0xFF;
416              SENSOR_addr_req[4] = 0xFF;
417              SENSOR_addr_receiv[0] = 0xFF;
418              SENSOR_addr_receiv[1] = 0xFF;
419              SENSOR_addr_receiv[2] = 0xFF;
420              SENSOR_addr_receiv[3] = 0xFF;
421              SENSOR_addr_receiv[4] = 0xFF;
422              break;
423            default:
424              SENSOR_addr_req[0] = 0x3F;
425              // 6 sensors are configured
426              SENSOR_addr_receiv[0] = 0x3F;
427              break;
428          }
429        }
430        //--------------------------------------------------------

432      }
433      else if(scan_mode == 2)
434        // Calibration initialization
435        //----------------------------------------------------------
436      {
437        //clear temp variables
438        for (sens=0;sens<41;sens++)
439        {
440          Temp_cal_counter[sens]=0;
441        }
442        scan_mode = 3;
443        RADIO_enable();
444      }
445      else if(scan_mode == 3)
446        // Calibration started
447      {
448        real_Temp = (RADIO_frame.frame_byte[4]);
449        real_Temp = real_Temp << 8;
450        real_Temp |= RADIO_frame.frame_byte[5];
451        ex = Temp_cal_counter[RADIO_frame.frame_byte[3]];
452        if (ex==0)
453        {
454          SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]]=
455            temp_correct_value-real_Temp;
456        }
457        else if((ex>=1) && (ex<20))
458        {
459          temp_diff = (temp_correct_value-real_Temp)+
460          (ex*SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]]);
461          SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]] =
462            temp_diff/(ex+1);
463        }
464        Temp_cal_counter[RADIO_frame.frame_byte[3]]++;
465        sensor_calibrated=0;
466        for (sens=0;sens<SENSOR_active;sens++)
467        {
468          if (Temp_cal_counter[SENSOR_address[sens]]>5)
469          {
470            sensor_calibrated++;
471          }
```

```
473         }
474         if (sensor_calibrated == SENSOR_active)
475         {
476           scan_mode = 0;
477           LCD_send_cmd(LCD_LINE1);
478           UART_menu_stop();
479           // disable radio unit
480           LCD_send_text("Calibr. complete");
481           UART0_send_text("Calibration complete \r\n");

483           RADIO_init_frame();
484           FLASH_Write_Controll_Data();

486         }
487         else
488         {
489           BATMON_control_reg &= ~VALID_DATA_RECEIVED;
490           // clear global value
491         }
492       }
493       //-------------------------------------------------------

495       //scan mode '0' no certain mode
496       else
497       {

499         do    // check if valid sensor
500         {
501           if (SENSOR_address[i] == RADIO_frame.frame_byte[3]) //compare if sensor
                  address is allowed
502             break;
503               // exit do-while loop
504           i++;
505         }
506         while(i<40);  //get sensor address index


509         if (i<40)   // if sensor address valid, maximum of 40 sensors
510         {
511           j = 0;
512           frame_error = 0;

514           while(j<11)   // check CRC
515           {
516             frame_error = frame_error ^ RADIO_frame.frame_byte[j];
517             j++;
518           }

520           if (frame_error)  // CRC not valid, exit
521           {
522             // CRC not valid return to the beginning
523             RADIO_rx_state = SE_SEQ_PREPARE;
524             crc_error[i]++;
525             break;  // exit switch-case block
526           }

528         if(fer==1)
529         {
530           //reset all error count variables
```

```
531            //for startup
532            bit_too_short = 0;
533            bit_too_long = 0;
534            wrong_RunIn = 0;
535          }

537          if(fer>0)
538          {
539            countFER(i,crc_error[i], bit_too_long, bit_too_short, wrong_RunIn); //FER calc
                 .
540          }

542          //reset error count variables
543          crc_error[i] = 0;
544          wrong_RunIn = 0;
545          bit_too_short = 0;
546          bit_too_long = 0;

548            // calculate position in sensor_addr_req array
549            //address_bit = (SENSOR_address[i] - 1) % 8 ;
550            //address_byte = (SENSOR_address[i] - 1) / 8;
551            address_bit = i % 8 ; //--> sensor addr bit is 'i'
552            address_byte = i / 8; //--> sensor addr byte is every 8 bit

554            // check if sensor data needed
555            /*if ((SENSOR_addr_req[address_byte] >> address_bit) & 0x01)
556            {*/
557              // clear sensor request and receive flag
558              SENSOR_addr_req[address_byte] &= ~(0x01 << address_bit);
559              SENSOR_addr_receiv[address_byte] &= ~(0x01 << address_bit);


562              //save last data received for sensor X
563              for (j=0; j<8; j++)
564              {
565  //              if (j==1)
566  //              {
567                  // temperature offset
568  //                if (RADIO_frame.frame_byte[3]!=0)    //offset?
569  //                {
570  //                  cal_temperature = (RADIO_frame.frame_byte[j+3]);
571  //                  cal_temperature = cal_temperature << 8;
572  //                  cal_temperature |= RADIO_frame.frame_byte[j+4];
573  //
574  //                  cal_temperature += SENSOR_temp_cal_b[SENSOR_address[i]];
575  //                  SENSOR_last_data[j][i] = (cal_temperature>>8);
576  //                  RADIO_frame.frame_byte[j+3] = cal_temperature>>8; //overwrite temp
       with cal. temp
577  //                }
578  //                else
579  //                {
580  //                  SENSOR_last_data[j][i] = RADIO_frame.frame_byte[j+3];
581  //                }
582  //              }
583  //              else if (j==2)
584  //              {
585  //                if (RADIO_frame.frame_byte[3]!=0)
586  //                {
587  //                  SENSOR_last_data[j][i] = (unsigned char)cal_temperature;
```

```
588 //                  RADIO_frame.frame_byte[j+3] = (unsigned char)cal_temperature; //
        overwrite temp with cal. temp
589 //                }
590 //                else
591 //                {
592 //                  SENSOR_last_data[j][i] = RADIO_frame.frame_byte[j+3];
593 //                }
594 //              }
595 //              else
596 //              {
597 //                SENSOR_last_data[j][i] = RADIO_frame.frame_byte[j+3];
598 //              }
599               SENSOR_last_data[j][i]=RADIO_frame.frame_byte[j];
600             }

602 //            // check depth discharge
603 //            check_voltage = (SENSOR_last_data[3][i]<<8);
604 //            check_voltage |= SENSOR_last_data[4][i];
605 //            if ((check_voltage/4.096) <=
606 //              ((CELL_VOLT_MIN*10)+100))
607 //            {
608 //              if ((check_voltage/4.096) <=
609 //                  (CELL_VOLT_MIN*10))
610 //              {
611 //                SENSOR_low++;
612 //                // don t leave next value
613 //                // SENSOR_addr_req[address_byte] |=
614 //                  // (0x01 << address_bit);
615 //              }
616 //            }

618             //RADIO_disable();  // disable radio unit
619             unsigned short tempus = RADIO_frame.frame_byte[4]<<8;
620               tempus |= RADIO_frame.frame_byte[5];
621               unsigned short tempus1 = RADIO_frame.frame_byte[6]<<8;
622               tempus1 |= RADIO_frame.frame_byte[7];
623               unsigned short tempus2 = RADIO_frame.frame_byte[8]<<8;
624               tempus2 |= RADIO_frame.frame_byte[9];
625             printf("ADD: %x LIVE: %x VOLTAGE: %i COUNTER: %i TEMP: %i\r\n",RADIO_frame.
                frame_byte[3], RADIO_frame.frame_byte[0]&BIT6,tempus, tempus1, tempus2)
                ;


628             BATMON_control_reg |= VALID_DATA_RECEIVED;
629                   // set global value
630           //}
631         }

633       }
634       // Receiving frame completed(valid or not),
635       // return to the beginning
636       RADIO_rx_state = SE_SEQ_PREPARE;

638     }
639     break;  // exit switch-case block

641   default: RADIO_rx_state = SE_SEQ_PREPARE;
642     break;  // exit switch-case block
643   }
644   TACCTL0 &= ~(CCIFG|COV);  /* reset timer occured flag, for up mode */
```

```
645     /*CCR0 += 410;  */    /* add value to timer, continous mode  */

647 }

649 /*---------------------------------------------------------------------------------
650   Timer A1 interrupt service routine
651 -------------------------------------------------------------------------------*/
652 interrupt (TIMERA1_VECTOR) TIMER_A1(void)
653 /*-----------------------------------------------------------------------------*/
654 {
655   switch(TAIV)
656   {
657     /* Capture/compare1 interrupt */
658     case 0x02:
659         TACCTL1 &= ~(CCIFG|COV);
660         /* reset timer occured flag   */
661         /*TACCR1 += 410;  */
662         /* add value to timer, continous mode */
663         break;

665     /* Capture/compare2 interrupt */
666     case 0x04:
667         TACCTL2 &= ~(CCIFG|COV);
668         /* reset timer occured flag */
669         /*TACCR2 += 410;  */
670         /* add value to timer, continous mode */
671         break;

673     /* Timeroverflow interrupt  */
674     case 0x0A: TACTL &= ~TAIFG;
675         /* reset timer occured flag */
676         break;
677   }

679 }

681 /*---------------------------------------------------------------------------------
682   Timer B0 interrupt service routine (button debounce)
683 -------------------------------------------------------------------------------*/
684 interrupt (TIMERB0_VECTOR) TIMER_B0(void)
685 /*-----------------------------------------------------------------------------*/
686 {
687   static char TB0_ct0, TB0_ct1;
688   char TB0_i;
689   static unsigned char x,y;


692   TB0_i = BUTTON_state ^ BUTTON_INPUT;
693     // input changed ?
694   TB0_ct0 = ~( TB0_ct0 & TB0_i );
695     // reset or count ct0
696   TB0_ct1 = (TB0_ct0 ^ (TB0_ct1 & TB0_i));
697     // reset or count ct1
698   TB0_i &= (TB0_ct0 & TB0_ct1);
699     // count until roll over
700   BUTTON_state ^= TB0_i;
701     // toggle debounced state
702   BATMON_control_reg |= BUTTON_state & TB0_i;
703     // only 0->1:key pressing detect
704   TBCCTL0 &= ~CCIFG;
```

```
705     // reset timer occured flag, for up mode
706     //TBCCR0 += 410;
707     // add value to timer, continous mode
708   if (x<4)
709   {
710     x++;
711   }
712   else
713   {
714     x=1;
715     //ADC12CTL0 |= ADC12SC;
716     // start ADC-conversion
717     if (y<8)
718     {
719       y++;
720     }
721     else
722     {
723       y=1;
724     }

726   }
727 }

729 /*------------------------------------------------------------------------------------
730   Timer B1 interrupt service routine
731 ----------------------------------------------------------------------------------*/
732 interrupt (TIMERB1_VECTOR) TIMER_B1(void)
733 /*--------------------------------------------------------------------------------*/
734 {
735   switch(TBIV)
736   {
737     /* Capture/compare1 interrupt */
738     case 0x02:

740         TBCCTL1 &= ~(CCIFG|COV);
741           /* reset timer occured flag   */
742         //TBCCR1 += 2;
743         /* add value to timer, continous mode */
744         break;

746     /* Capture/compare2 interrupt */
747     case 0x04:
748         TBCCTL2 &= ~(CCIFG|COV);
749         /* reset timer occured flag */
750         /*TBCCR2 += 410;  */
751         /* add value to timer, continous mode */
752         break;

754     /* Timeroverflow interrupt  */
755     case 0x0E:
756         TBCTL &= ~TAIFG;
757         /* reset timer occured flag */
758         break;
759   }

761 }

763 /*------------------------------------------------------------------------------------
764   UART0 RX interrupt service routine
```

```
765 /*---------------------------------------------------------------------------------*/
766 interrupt (USART0RX_VECTOR) USART0_RX(void)
767 /*---------------------------------------------------------------------------------*/
768 {
769   static unsigned char rx_string[MAX_STRING_LENGTH];
770   static unsigned char i;

772   switch (RXBUF0)
773   {
774   case (RETURN):
775     // if carriage return jump to next line first position and
776     // store string global
777     UART0_send(NEWLINE);
778     UART0_send(RETURN);

780     rx_string[i] = NULLTERMINATOR;     // terminate string
781     i++;

783     for (; i>0; i--)
784     // copy string
785     {
786       global_string[i-1] = rx_string[i-1];
787     };
788     BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
789     // set control register to global string received
790     break;

792   case ('%'):
793     // if '%' jump to next line first position and store string global
794     UART0_send(NEWLINE);
795     UART0_send(RETURN);

797     rx_string[i] = NULLTERMINATOR;
798     // terminate string
799     i++;

801     for (; i>0; i--)
802     // copy string
803     {
804       global_string[i-1] = rx_string[i-1];
805     };
806     BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
807     // set control register to global string received
808     break;

810   case (BACKSPACE):
811     // backspace, delete last char
812     UART0_send(BACKSPACE);
813     UART0_send(SPACE);
814     UART0_send(BACKSPACE);
815     i--;
816     break;

818   default:
819     // echo
820     if(i<(MAX_STRING_LENGTH-1))
821     // string length reached?
822     {
823       UART0_send(RXBUF0);
824       // string length not reached store received char
```

```
825         rx_string[i] = RXBUF0;
826         i++;
827       }
828       else
829         // string length reached, save string in global variable
830       {
831         rx_string[i] = NULLTERMINATOR;
832         // terminate string
833         for (; i>0; i--)
834         // copy string
835         {
836           global_string[i-1] = rx_string[i-1];
837         };
838         BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
839         // set control register to global string received
840       }
841       break;
842     }
843 }

845 /*-------------------------------------------------------------------------------------
846   UART0 TX interrupt service routine
847 -----------------------------------------------------------------------------------*/
848 interrupt (USART0TX_VECTOR) USART0_TX(void)
849 /*-----------------------------------------------------------------------------------*/
850 {

852 }
853 /*

855 -----------------------------------------------------------------------------------
856   PORT1 interrupt service routine
857 -----------------------------------------------------------------------------------
858 interrupt (PORT1_VECTOR) PORT_1(void)
859 ---------------------------------------------------------------------------------
860 {

862 }

864 -----------------------------------------------------------------------------------
865   PORT2 interrupt service routine
866 -----------------------------------------------------------------------------------
867 interrupt (PORT2_VECTOR) PORT_2(void)
868 ---------------------------------------------------------------------------------
869 {

871 }
872 */

874 /*-------------------------------------------------------------------------------------
875   DAC12 interrupt service routine
876 -----------------------------------------------------------------------------------*/
877 interrupt (DACDMA_VECTOR) DAC12(void)
878 /*-----------------------------------------------------------------------------------*/
879 {

881 }

883 /*-------------------------------------------------------------------------------------
884   ADC12 interrupt service routine (current measuring)
```

```
885 -----------------------------------------------------------------------------*/
886 interrupt (ADC12_VECTOR) ADC12(void)
887 /*---------------------------------------------------------------------------*/
888 {
889   //actual tared current value
890   unsigned long CURRENT_work;
891   //current sum for tara
892   static unsigned long current_sum;
893   //counter for tara
894   static char current_sum_counter;

896   static char x_low=0,x_mid=0,x_high=0,x_min=0;

898   //actual current directions for each interval
899   static char x_low_IN_OUT, x_mid_IN_OUT, x_high_IN_OUT, x_min_IN_OUT;
900   //current value sum for each interval and sensor channel
901   static unsigned long x_low_level_sum, x_mid_level_sum;
902   unsigned long x_mid_level_sum_tmp = 0;
903   static unsigned long x_min_level_sum, x_high_level_sum[2];
904   static unsigned short x_high_level_sum_count[2];

906   static char i,w;
907   static char x,z;
908   static long y;
909   unsigned short j=0;
910   PRINTF_UART;    // set printf to UART

912   //****************************************************************************
913   // measuring range decision
914   //****************************************************************************

916   static unsigned int channel1, channel2;
917   static unsigned long current_value;

919   channel2 = (unsigned int)(ADC12MEM0);
920   channel1 = (unsigned int)(ADC12MEM1);

922   //switch to channel 2 if channel 1 out of range (channel1: -70 - +70 A)
923   if((channel1<3578) && (channel1>1033))
924   {
925     current_value = channel1;

927     if((current_value < 3560) && (current_value > 1048))
928     {
929       current_value = channel1;
930       //set channel flag to ch1
931       CURRENT_channel = 1;
932     }
933   }
934   else
935   {
936     current_value = channel2;
937     //set channel flag to ch2
938     CURRENT_channel = 2;
939   }

941   //*****************************************************************************
942   // init Current tara
943   //*****************************************************************************
944   if (scan_mode == 4)
```

```
945   {
946     scan_mode = 5;
947     current_sum = current_value;
948     current_sum_counter = 0;
949   }
950   // Current tara 20 values (scan_mode 5)
951   else if (scan_mode == 5)
952   {
953     if (current_sum_counter==0) //first current value (no current value so far)
954     {
955       current_sum += current_value;
956       current_sum_counter++;
957     }
958     else if((current_sum_counter>=1) && (current_sum_counter<19)) //take 19 values left
959     {
960       current_sum += current_value;
961       current_sum_counter++;
962     }
963     else  //all data received --> take average of 20 values as tara value
964     {
965       current_sum /= (current_sum_counter + 1);
966       CURRENT_tara = current_sum;
967       scan_mode = 0;  //reset scan mode to normal
968       printf("Current Tara value: %3li \r\n", CURRENT_tara);
969       if (measuring == 0)
970       {
971         ADC_stop();
972       }
973     }
974   }
975   //*****************************************************************************
976   // normal consumption recording (scan_mode 0)
977   //*****************************************************************************
978   else
979   {

981     // init new tara
982     if (CURRENT_tara == 0)
983     {
984       i=0;
985       x=0;
986       z=0;
987       y=0;
988       w=0;
989       CURRENT_tara = current_sum; //2007 --DEFAULT= 0V
990       AH_last_minute = 0;
991     }

993     // consumption recording
994     else
995     {
996       // current inverted?
997       if (CURRENT_INVERT == 0)
998       {
999         CURRENT_work = current_value;
1000      }
1001      else
1002      {
1003        //invert current value
1004        if(current_value<=CURRENT_tara)
```

```
1005          CURRENT_work =  CURRENT_tara + (CURRENT_tara - current_value);
1006        else
1007          CURRENT_work =  CURRENT_tara - (current_value - CURRENT_tara);
1008      }
1009      // Zero Point definition
1010      if (((CURRENT_work)<=(CURRENT_tara+10))&&((CURRENT_work)>=(CURRENT_tara-10)))
1011      {
1012        CURRENT_work = CURRENT_tara;
1013      }
1014      //actual current! For LCD!
1015      //Attention! CURRENT_last_data is not signed!
1016      if(CURRENT_work > CURRENT_tara)
1017        CURRENT_last_data = CURRENT_work - CURRENT_tara;
1018      else
1019        CURRENT_last_data = CURRENT_tara - CURRENT_work;

1021      //kHz level
1022      if (x_high < 29)
1023      {
1024        x_high++;
1025        x_high_level_sum_count[CURRENT_channel-1]++;
1026        // discharge
1027        if ((CURRENT_work) >= CURRENT_tara)
1028        {
1029          CURRENT_last_data = CURRENT_work - CURRENT_tara;
1030          CURRENT_IN_OUT = 0; //set global discharge
1031          //high level measurement is ~30ms interval
1032          //get first value, set this kHz level to discharge

1034          if (x_high == 1)
1035          {
1036            x_high_level_sum[0] = 0;
1037            x_high_level_sum[1] = 0;
1038            x_high_level_sum_count[0] = 0;
1039            x_high_level_sum_count[1] = 0;
1040            x_high_level_sum_count[CURRENT_channel-1]++;

1042            x_high_level_sum[CURRENT_channel-1] = CURRENT_last_data;

1044            x_high_IN_OUT = 0;  //high level is set to discharge
1045          }
1046          //if discharge in this level, summarize measured data
1047          else if(x_high_IN_OUT == 0)
1048          {
1049            x_high_level_sum[CURRENT_channel-1] += CURRENT_last_data;
1050          }
1051          //if charge state changes within the kHz periode high
1052          //necessary to avoid negatives
1053          else
1054          {
1055            if (x_high_level_sum[CURRENT_channel-1] > CURRENT_last_data)
1056            {
1057              x_high_level_sum[CURRENT_channel-1] -= CURRENT_last_data;
1058            }

1060            else
1061            {
1062              x_high_level_sum[CURRENT_channel-1] = CURRENT_last_data - x_high_level_sum
                      [CURRENT_channel-1];
```

```
1064                 x_high_IN_OUT = 0;  //set state to discharge
1065               }
1066             }
1067           }
1068           //charge
1069           else
1070           {
1071             CURRENT_last_data = CURRENT_tara - CURRENT_work;
1072             CURRENT_IN_OUT = 1; //set charge state
1073             if (x_high == 1)
1074             {
1075               x_high_level_sum[0] = 0;
1076               x_high_level_sum[1] = 0;
1077               x_high_level_sum_count[0] = 0;
1078               x_high_level_sum_count[1] = 0;
1079               x_high_level_sum_count[CURRENT_channel-1]++;

1081               x_high_level_sum[CURRENT_channel-1] = CURRENT_last_data;

1083               x_high_IN_OUT = 1;  //high level is set to charge
1084             }
1085             //summarize measured data in charge mode
1086             else if(x_high_IN_OUT == 1)
1087             {
1088               x_high_level_sum[CURRENT_channel-1] += CURRENT_last_data;
1089             }
1090             //distinguish between charge states
1091             //again: avoid the negatives!
1092             else
1093             {
1094               if (x_high_level_sum[CURRENT_channel-1] > CURRENT_last_data)
1095               {
1096                 x_high_level_sum[CURRENT_channel-1] -= CURRENT_last_data;
1097               }
1098               else
1099               {
1100                 x_high_level_sum[CURRENT_channel-1] = CURRENT_last_data - x_high_level_sum
                         [CURRENT_channel-1];

1102                 x_high_IN_OUT = 1;
1103               }
1104             }
1105           }
1106         }
1107         else
1108         {

1110           //medium khz level
1111           if (x_mid < 33)
1112           {
1113             x_mid++;

1115             //calculate average of high level
1116             //don't divide by 0!
1117             for(j=0;j<2;j++)
1118             {
1119               if(x_high_level_sum_count[j]>0)
1120                 x_high_level_sum[j] += x_high_level_sum[j]/x_high_level_sum_count[j];
1121               else
1122                 x_mid_level_sum_tmp += 0;
```

```
1123             if(j == 0)
1124               x_high_level_sum[0] *= 45.7;
1125             else
1126               x_high_level_sum[1] *= 305.3;

1128             x_mid_level_sum_tmp += x_high_level_sum[j];
1129           }

1131           if(x_high_level_sum[0] && x_high_level_sum[1])
1132             x_mid_level_sum_tmp /= 2;

1134           x_high_level_sum[0] = 0;
1135           x_high_level_sum[1] = 0;


1138           // discharge
1139           if (x_high_IN_OUT == 0)
1140           {
1141             // first value
1142             if (x_mid == 1)
1143             {
1144               x_mid_level_sum = x_mid_level_sum_tmp;

1146               x_mid_IN_OUT = 0;
1147             }
1148             else if(x_mid_IN_OUT == 0)
1149             {
1150               x_mid_level_sum += x_mid_level_sum_tmp;
1151             }
1152             else
1153             {
1154               if (x_mid_level_sum > x_mid_level_sum_tmp)
1155               {
1156                 x_mid_level_sum -= x_mid_level_sum_tmp;
1157               }
1158               else
1159               {
1160                 x_mid_level_sum = x_mid_level_sum_tmp-x_mid_level_sum;

1162                 x_mid_IN_OUT = 0;
1163               }
1164             }
1165           }
1166           //charge
1167           else
1168           {
1169             if (x_mid == 1)
1170             {
1171               x_mid_level_sum = x_mid_level_sum_tmp;

1173               x_mid_IN_OUT = 1;
1174             }
1175             else if(x_mid_IN_OUT == 1)
1176             {
1177               x_mid_level_sum += x_mid_level_sum_tmp;
1178             }

1180             else
1181             {
1182               if (x_mid_level_sum > x_mid_level_sum_tmp)
```

```
1183               {
1184                 x_mid_level_sum -= x_mid_level_sum_tmp;
1185               }
1186               else
1187               {
1188                 x_mid_level_sum = x_mid_level_sum_tmp - x_mid_level_sum;

1190                 x_mid_IN_OUT = 1;
1191               }
1192             }
1193           }
1194       }
1195       else
1196         //low khz level
1197         {
1198           AH_last_sec = x_mid_level_sum/33;

1200           BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
1201           if (x_low < 30)
1202           {
1203             x_low++;
1204             // discharge
1205             if (x_mid_IN_OUT == 0)
1206             {
1207               if (x_low == 1)
1208               {
1209                 x_low_level_sum = AH_last_sec;
1210                 x_low_IN_OUT = 0;
1211               }
1212               else if(x_low_IN_OUT == 0)
1213               {
1214                 x_low_level_sum += AH_last_sec;
1215               }
1216               else
1217               {
1218                 if (x_low_level_sum > AH_last_sec)
1219                 {
1220                   x_low_level_sum -= AH_last_sec;
1221                 }
1222                 else
1223                 {
1224                   x_low_level_sum = AH_last_sec - x_low_level_sum;
1225                   x_low_IN_OUT = 0;
1226                 }
1227               }
1228             }
1229             //charge
1230             else
1231             {
1232               if (x_low == 1)
1233               {
1234                 x_low_level_sum = AH_last_sec;
1235                 x_low_IN_OUT = 1;
1236               }
1237               else if(x_low_IN_OUT == 1)
1238               {
1239                 x_low_level_sum += AH_last_sec;
1240               }
1241               else
1242               {
```

```
1243                    if (x_low_level_sum > AH_last_sec)
1244                    {
1245                      x_low_level_sum -= AH_last_sec;
1246                    }
1247                    else
1248                    {
1249                      x_low_level_sum = AH_last_sec - x_low_level_sum;
1250                      x_low_IN_OUT = 1;
1251                    }
1252                  }
1253                }
1254              }
1255              else
1256              {

1258                if (x_min == 0)
1259                {
1260                  x_min_level_sum = x_low_level_sum/30;
1261                  x_min_IN_OUT = x_low_IN_OUT;
1262                  x_min++;
1263                }
1264                else
1265                {
1266                  //same current direction
1267                  LED1_TOGGLE;

1269                  if (x_low_IN_OUT == x_min_IN_OUT)
1270                  {
1271                    AH_last_minute_IN_OUT = x_min_IN_OUT;
1272                    AH_last_minute = x_min_level_sum + x_low_level_sum/30;
1273                  }
1274                  else
1275                  {
1276                    if (x_min_level_sum > x_low_level_sum/30)
1277                    {
1278                      AH_last_minute_IN_OUT = x_low_IN_OUT;
1279                      AH_last_minute = ((x_low_level_sum/30)-x_min_level_sum);
1280                    }

1282                    else
1283                    {
1284                      AH_last_minute_IN_OUT = x_min_IN_OUT;
1285                      AH_last_minute = (x_min_level_sum-(x_low_level_sum/30));
1286                    }
1287                  }
1288                  x_min = 0;
1289                }
1290                x_low = 0;
1291              }
1292              x_mid = 0;
1293            }
1294            x_high = 0;
1295          }
1296        }
1297      }
1298    }

1300    /*------------------------------------------------------------------------------*/
1301    /*  DAC12 interrupt service routine
1302    ------------------------------------------------------------------------------*/
```

```
1303 interrupt (COMPARATORA_VECTOR) COMPERATOR(void)
1304 /*---------------------------------------------------------------------------------*/
1305 {

1307 }
```

Listing 9: Quellcode des Datenloggers (isr_ ports.c)

```
1  /*---------------------------------------------------------------------------------
2    Project:       MSP430-169STK onboard nand-flash functions
3    Used components:    MSP430-169STK
4    Date:        12/01/2007
5    Last update:       09/05/2008
6    Author:        Tobias Krannich
7    Modified:      Stephan Plaschke

9    Modified by:     Alexander Hoops
10   Last modification:  12/02/2010
11 ---------------------------------------------------------------------------------

13 ---------------------------------------------------------------------------------
14   Headerfiles
15 ---------------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <Flash.h>
18 #include <stdio.h>
19 #include <lcd16x2.h>
20 #include <msp430x16x.h>
21 #include <MSP_functions.h>



25 static unsigned char *FLASH_clear_frame[] = {
26     "                ",
27     "  Clear flash   "
28 };

30 //----------------NAND FLASH-------------------------------------------------------
31 //---------------------------------------------------------------------------------
32 void init_Flash (void)
33 //---------------------------------------------------------------------------------
34 {
35   P2OUT = 0x07;               //NAND FLASH ini
36     P2DIR = 0x1F;
37 }

39 //---------------------------------------------------------------------------------
40 // pull flash pins to inactive condition
41 void Inactive_Flash(void)
42 //---------------------------------------------------------------------------------
43 {
44   IO_DIR=INPUT;          //IO is inputs
45   _CE_OFF;               //=1
46   _RE_OFF;               //=1
47   _WE_OFF;               //=1
48   ALE_OFF;               //=0
49   CLE_OFF;               //=0
50 }

52 //---------------------------------------------------------------------------------
```

```
53  void Write_Data(unsigned char a)
54  //--------------------------------------------------------------------------------------
55  {
56    IO_DIR=OUTPUT;          //IO is outputs
57    _WE_ON;
58    OUT_PORT=a;
59    _WE_OFF;                //latch data
60  }

62  //--------------------------------------------------------------------------------------
63  unsigned char Read_Data(void)
64  //--------------------------------------------------------------------------------------
65  {
66    unsigned char f;
67    IO_DIR=INPUT;           //IO is inputs
68    _RE_ON;
69    f=IN_PORT;
70    _RE_OFF;                //read data
71    return (f);
72  }

74  //--------------------------------------------------------------------------------------
75  unsigned char Programm_Bytes(unsigned char COL_ADD,
76                               unsigned char ROW_ADDL,
77                               unsigned char ROW_ADDH,
78                               unsigned char NUMBER)
79  {
80  //--------------------------------------------------------------------------------------
81          unsigned char k, l;
82          Inactive_Flash();
83      CLE_ON;
84      _CE_ON;
85      Write_Data(WRITE_PAGE);
86      CLE_OFF;
87      ALE_ON;
88          Write_Data(COL_ADD);
89          Write_Data(ROW_ADDL);
90          Write_Data(ROW_ADDH);
91          ALE_OFF;
92          for (k=0; k != NUMBER; k++)
93            {
94               l=WRITE_BUF[k];
95               Write_Data(l);
96            }
97          CLE_ON;
98          Write_Data(WRITE_AKN);
99          while ((R_B) == 0);
100         Write_Data(READ_STATUS);
101         CLE_OFF;
102         l = Read_Data();
103         Inactive_Flash();
104         return(l);
105 }

107 //--------------------------------------------------------------------------------------
108 void Read_Bytes (unsigned char COL_ADD,
109                  unsigned char ROW_ADDL,
110                  unsigned char ROW_ADDH,
111                  unsigned char NUMBER)
112 {
```

```
113 //---------------------------------------------------------------------------------
114         unsigned char n, r;
115         Inactive_Flash();
116         CLE_ON;
117         _CE_ON;
118         Write_Data(READ_0);
119         CLE_OFF;
120         ALE_ON;
121         Write_Data(COL_ADD);
122         Write_Data(ROW_ADDL);
123         Write_Data(ROW_ADDH);
124         ALE_OFF;
125         while ((R_B) == 0);
126         for (n=0; n != NUMBER; n++)
127           {
128              r=Read_Data();
129              READ_BUF[n] = r;
130           }
131         Inactive_Flash();
132 }

134 //---------------------------------------------------------------------------------
135 void Data_Struct_To_Write_Buffer(DATA_STRUCT *pData)
136 //---------------------------------------------------------------------------------
137 {
138   WRITE_BUF[0] = pData->C0;
139   WRITE_BUF[1] = pData->C1;
140   WRITE_BUF[2] = pData->C2;
141   WRITE_BUF[3] = pData->C3;
142   WRITE_BUF[4] = pData->C4;
143   WRITE_BUF[5] = pData->C5;
144   WRITE_BUF[6] = pData->C6;
145   WRITE_BUF[7] = pData->C7;
146   WRITE_BUF[8] = pData->C8;
147   WRITE_BUF[9] = pData->C9;
148   WRITE_BUF[10] = pData->C10;
149   WRITE_BUF[11] = pData->C11;
150   WRITE_BUF[12] = pData->C12;
151   WRITE_BUF[13] = pData->C13;
152   WRITE_BUF[14] = pData->C14;
153   WRITE_BUF[15] = pData->C15;
154 }

156 //---------------------------------------------------------------------------------
157 void Val_To_Data(DATA_STRUCT *pData, unsigned long i)
158 //---------------------------------------------------------------------------------
159 {
160   if(i >= 527)
161   {
162     _NOP();
163   }
164   pData->C0 = (unsigned char)((i>>0) & 0xFF);
165   pData->C1 = (unsigned char)((i>>8) & 0xFF);
166   pData->C2 = (unsigned char)((i>>16) & 0xFF);
167   pData->C3 = (unsigned char)((i>>24) & 0xFF);
168 }

170 //---------------------------------------------------------------------------------
171 // Es wird der Block gelöscht, in dem die Adresse liegt
172 unsigned char Erase_Flash (unsigned char BLOCK_ADDL, unsigned char BLOCK_ADDH)
```

```
173 //--------------------------------------------------------------------------------
174 {
175         unsigned char m;
176         Inactive_Flash();
177     CLE_ON;
178     _CE_ON;
179     Write_Data(ERASE_BLOCK);
180     CLE_OFF;
181     ALE_ON;
182         Write_Data(BLOCK_ADDL);
183         Write_Data(BLOCK_ADDH);
184         ALE_OFF;
185         CLE_ON;
186         Write_Data(ERASE_AKN);
187         while ((R_B) == 0);
188         Write_Data(READ_STATUS);
189         CLE_OFF;
190         m = Read_Data();
191         Inactive_Flash();
192         return (m);
193 }

195 //--------------------------------------------------------------------------------
196 DATA_STRUCT * Read_Ring_Flash(void)
197 //--------------------------------------------------------------------------------
198 {
199   //unsigned short Index_Last = 0;
200   unsigned short ADD = 0;
201   unsigned char ADDH = 0;
202   unsigned char ADDL = 0;
203   unsigned char COL = 0;
204   static DATA_STRUCT Data = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
205   DATA_STRUCT *pData = &Data;

207   // Letztes Element im Ringbuffer ermitteln
208   if(Ring_Full == 0)  // Ringbuffer hatte noch keinen Überlauf
209   {
210     // Keine Daten im Ringspeicher???
211     if(Count_Ring == 0)
212     {
213       return NULL;
214     }
215     Index_Last = Index_Ring - Count_Ring;
216   }
217   else // Es gab schon einen Überlauf -> lese ab dem ersten Struct
218   {
219     // Keine Daten im Ringspeicher???
220     if((Count_Ring - 255) == 0) //
221     {
222       return NULL;
223     }
224     Index_Last = Index_Ring - (Count_Ring + 1 - 256)/* -
225     (Index_Ring%256)*/; // +1 um auf 0x10000 zu kommen
226   }
227   Count_Ring--;

229   // Adresse berechnen
230   ADD = Index_Last / 16;
231   COL = Index_Last % 16;
232   ADDL = (unsigned char)(ADD & 0xFF);
```

```
233   ADDH = (unsigned char)((ADD>>8) & 0xFF);

235   // Speicher auslesen
236   Read_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);

238   // Read_BUF in das Datenstruct schreiben
239   pData->C0  = READ_BUF[0];
240   pData->C1  = READ_BUF[1];
241   pData->C2  = READ_BUF[2];
242   pData->C3  = READ_BUF[3];
243   pData->C4  = READ_BUF[4];
244   pData->C5  = READ_BUF[5];
245   pData->C6  = READ_BUF[6];
246   pData->C7  = READ_BUF[7];
247   pData->C8  = READ_BUF[8];
248   pData->C9  = READ_BUF[9];
249   pData->C10 = READ_BUF[10];
250   pData->C11 = READ_BUF[11];
251   pData->C12 = READ_BUF[12];
252   pData->C13 = READ_BUF[13];
253   pData->C14 = READ_BUF[14];
254   pData->C15 = READ_BUF[15];


257   return pData;
258 }

260 //-----------------------------------------------------------------------------------
261 //-----------------------------------------------------------------------------------
262 //    These functions are modified/created by Stephan Plaschke
263 //-----------------------------------------------------------------------------------
264 //-----------------------------------------------------------------------------------

266 //-----------------------------------------------------------------------------------
267 void Erase_Flash_All(void)
268 //-----------------------------------------------------------------------------------
269 {
270   int i = 0;
271   unsigned char j=0;
272   unsigned char ADDL;
273   unsigned char ADDH;

275   // show splash screen on LCD and send via UART
276   LCD_send_cmd(LCD_LINE1);
277   LCD_send_text(FLASH_clear_frame[1]);
278   UART0_send('.');

280   for(i=0; i<0x1000; i++)
281   {
282     //ADD = i * 16;
283     ADDL = (unsigned char)(i & 0xFF);
284     ADDH = (unsigned char)((i>>8) & 0xFF);
285     Erase_Flash(ADDL,ADDH);

287     // show splash screen on LCD and send via UART
288     if (!(i%200))
289     {
290       if(j^=0x01)
291       {
292         LCD_send_cmd(LCD_LINE1);
```

```
293          LCD_send_text(FLASH_clear_frame[1]);
294          UART0_send('.');
295        }
296        else
297        {
298          LCD_send_cmd(LCD_LINE1);
299          LCD_send_text(FLASH_clear_frame[0]);
300        }
301      }
302    }
303    // set all index to the first frame in flash
304    Count_Ring = 0;
305    Ring_Full = 0;
306    // set pointer on the address 256, first block used for controll data
307    Index_Ring = 256;
308    Index_Last = 256;

310    LCD_send_cmd(LCD_LINE1);
311    LCD_send_text(FLASH_clear_frame[0]);
312    UART0_send_text("\r\n");

314    RADIO_init_frame();      // set first frame
315  }

317  //------------------------------------------------------------------------------------
318  void Write_Ring_Flash(DATA_STRUCT *pData)
319  //------------------------------------------------------------------------------------
320  {
321    unsigned char ADDL;
322    unsigned char ADDH;
323    unsigned short ADD;
324    unsigned char COL;


327    ADD = Index_Ring / 16;
328    COL = Index_Ring % 16;
329    ADDL = (unsigned char)(ADD & 0xFF);
330    ADDH = (unsigned char)((ADD>>8) & 0xFF);

332    // Lösche den nächsten Block
333    if( (Index_Ring%256) == 0)
334    {
335      Erase_Flash(ADDL,ADDH);
336    }

338    Data_Struct_To_Write_Buffer(pData);
339    Programm_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);

341    if(Count_Ring < 65279) // address space: 65535 - 256
342    {
343      Count_Ring++;
344    }
345    else
346    {
347      Ring_Full = 1;
348    }

350    Index_Ring++;
351    if(Index_Ring >= 65535)
352    {
```

```
353      Index_Ring = 256;
354           // set pointer on the address 256, first block used for controll data
355    }
356  }

358  //--------------------------------------------------------------------------------
359  void FLASH_Write_Controll_Data(void)
360  //--------------------------------------------------------------------------------
361  {
362    Erase_Flash(0,0);

364    WRITE_BUF[0]  = (unsigned char)(Index_Ring >> 8);
365    WRITE_BUF[1]  = (unsigned char)Index_Ring;
366    WRITE_BUF[2]  = (unsigned char)(Index_Last >> 8);
367    WRITE_BUF[3]  = (unsigned char)Index_Last;
368    WRITE_BUF[4]  = (unsigned char)(Count_Ring >> 8);
369    WRITE_BUF[5]  = (unsigned char)Count_Ring;
370    WRITE_BUF[6]  = Ring_Full;
371    WRITE_BUF[7]  = SENSOR_active;
372    WRITE_BUF[8]  = SENSOR_address[0];
373    WRITE_BUF[9]  = SENSOR_address[1];
374    WRITE_BUF[10] = SENSOR_address[2];
375    WRITE_BUF[11] = SENSOR_address[3];
376    WRITE_BUF[12] = SENSOR_address[4];
377    WRITE_BUF[13] = SENSOR_address[5];
378    WRITE_BUF[14] = SENSOR_address[6];
379    WRITE_BUF[15] = SENSOR_address[7];

381    Programm_Bytes(0,0,0,BUF_SIZE);

383    WRITE_BUF[0]  = SENSOR_address[8];
384    WRITE_BUF[1]  = SENSOR_address[9];
385    WRITE_BUF[2]  = SENSOR_address[10];
386    WRITE_BUF[3]  = SENSOR_address[11];
387    WRITE_BUF[4]  = SENSOR_address[12];
388    WRITE_BUF[5]  = SENSOR_address[13];
389    WRITE_BUF[6]  = SENSOR_address[14];
390    WRITE_BUF[7]  = SENSOR_address[15];
391    WRITE_BUF[8]  = SENSOR_address[16];
392    WRITE_BUF[9]  = SENSOR_address[17];
393    WRITE_BUF[10] = SENSOR_address[18];
394    WRITE_BUF[11] = SENSOR_address[19];
395    WRITE_BUF[12] = SENSOR_address[20];
396    WRITE_BUF[13] = SENSOR_address[21];
397    WRITE_BUF[14] = SENSOR_address[22];
398    WRITE_BUF[15] = SENSOR_address[23];

400    Programm_Bytes(0,1,0,BUF_SIZE);

402    WRITE_BUF[0]  = SENSOR_address[24];
403    WRITE_BUF[1]  = SENSOR_address[25];
404    WRITE_BUF[2]  = SENSOR_address[26];
405    WRITE_BUF[3]  = SENSOR_address[27];
406    WRITE_BUF[4]  = SENSOR_address[28];
407    WRITE_BUF[5]  = SENSOR_address[29];
408    WRITE_BUF[6]  = SENSOR_address[30];
409    WRITE_BUF[7]  = SENSOR_address[31];
410    WRITE_BUF[8]  = SENSOR_address[32];
411    WRITE_BUF[9]  = SENSOR_address[33];
412    WRITE_BUF[10] = SENSOR_address[34];
```

```
413    WRITE_BUF[11] = SENSOR_address[35];
414    WRITE_BUF[12] = SENSOR_address[36];
415    WRITE_BUF[13] = SENSOR_address[37];
416    WRITE_BUF[14] = SENSOR_address[38];
417    WRITE_BUF[15] = SENSOR_address[39];

419    Programm_Bytes(0,2,0,BUF_SIZE);

421    WRITE_BUF[0]  = SENSOR_temp_cal_b[1];
422    WRITE_BUF[1]  = SENSOR_temp_cal_b[2];
423    WRITE_BUF[2]  = SENSOR_temp_cal_b[3];
424    WRITE_BUF[3]  = SENSOR_temp_cal_b[4];
425    WRITE_BUF[4]  = SENSOR_temp_cal_b[5];
426    WRITE_BUF[5]  = SENSOR_temp_cal_b[6];
427    WRITE_BUF[6]  = SENSOR_temp_cal_b[7];
428    WRITE_BUF[7]  = SENSOR_temp_cal_b[8];
429    WRITE_BUF[8]  = SENSOR_temp_cal_b[9];
430    WRITE_BUF[9]  = SENSOR_temp_cal_b[10];
431    WRITE_BUF[10] = SENSOR_temp_cal_b[11];
432    WRITE_BUF[11] = SENSOR_temp_cal_b[12];
433    WRITE_BUF[12] = SENSOR_temp_cal_b[13];
434    WRITE_BUF[13] = SENSOR_temp_cal_b[14];
435    WRITE_BUF[14] = SENSOR_temp_cal_b[15];
436    WRITE_BUF[15] = SENSOR_temp_cal_b[16];

438    Programm_Bytes(0,3,0,BUF_SIZE);

440    WRITE_BUF[0]  = SENSOR_temp_cal_b[17];
441    WRITE_BUF[1]  = SENSOR_temp_cal_b[18];
442    WRITE_BUF[2]  = SENSOR_temp_cal_b[19];
443    WRITE_BUF[3]  = SENSOR_temp_cal_b[20];
444    WRITE_BUF[4]  = SENSOR_temp_cal_b[21];
445    WRITE_BUF[5]  = SENSOR_temp_cal_b[22];
446    WRITE_BUF[6]  = SENSOR_temp_cal_b[23];
447    WRITE_BUF[7]  = SENSOR_temp_cal_b[24];
448    WRITE_BUF[8]  = SENSOR_temp_cal_b[25];
449    WRITE_BUF[9]  = SENSOR_temp_cal_b[26];
450    WRITE_BUF[10] = SENSOR_temp_cal_b[27];
451    WRITE_BUF[11] = SENSOR_temp_cal_b[28];
452    WRITE_BUF[12] = SENSOR_temp_cal_b[29];
453    WRITE_BUF[13] = SENSOR_temp_cal_b[30];
454    WRITE_BUF[14] = SENSOR_temp_cal_b[31];
455    WRITE_BUF[15] = SENSOR_temp_cal_b[32];

457    Programm_Bytes(0,4,0,BUF_SIZE);

459    WRITE_BUF[0]  = SENSOR_temp_cal_b[33];
460    WRITE_BUF[1]  = SENSOR_temp_cal_b[34];
461    WRITE_BUF[2]  = SENSOR_temp_cal_b[35];
462    WRITE_BUF[3]  = SENSOR_temp_cal_b[36];
463    WRITE_BUF[4]  = SENSOR_temp_cal_b[37];
464    WRITE_BUF[5]  = SENSOR_temp_cal_b[38];
465    WRITE_BUF[6]  = SENSOR_temp_cal_b[39];
466    WRITE_BUF[7]  = SENSOR_temp_cal_b[40];
467    WRITE_BUF[8]  = (unsigned char)(AH_MASTER>>8);
468    WRITE_BUF[9]  = (unsigned char)AH_MASTER;
469    WRITE_BUF[10] = (unsigned char)(CELL_VOLT_MAX>>8);
470    WRITE_BUF[11] = (unsigned char)CELL_VOLT_MAX;
471    WRITE_BUF[12] = (unsigned char)(CELL_VOLT_MIN>>8);
472    WRITE_BUF[13] = (unsigned char)CELL_VOLT_MIN;
```

```
473   WRITE_BUF[14] = (unsigned char)(BATT_CAPACITY>>8);
474   WRITE_BUF[15] = (unsigned char)BATT_CAPACITY;

476   Programm_Bytes(0,5,0,BUF_SIZE);

478   WRITE_BUF[0]  = (unsigned char)(CURRENT_tara>>8);
479   WRITE_BUF[1]  = (unsigned char)CURRENT_tara;
480   WRITE_BUF[2]  = CURRENT_INVERT;
481   WRITE_BUF[3]  = 0;
482   WRITE_BUF[4]  = 0;
483   WRITE_BUF[5]  = 0;
484   WRITE_BUF[6]  = 0;
485   WRITE_BUF[7]  = 0;
486   WRITE_BUF[8]  = 0;
487   WRITE_BUF[9]  = 0;
488   WRITE_BUF[10] = 0;
489   WRITE_BUF[11] = 0;
490   WRITE_BUF[12] = 0;
491   WRITE_BUF[13] = 0;
492   WRITE_BUF[14] = 0;
493   WRITE_BUF[15] = 0;

495   Programm_Bytes(0,6,0,BUF_SIZE);

497 }
498 //----------------------------------------------------------------------------------
499 void FLASH_Read_Controll_Data(void)
500 //----------------------------------------------------------------------------------
501 {
502   //unsigned short i;
503   // Speicher auslesen
504   Read_Bytes(0,0,0,BUF_SIZE);

506   Index_Ring        = (READ_BUF[0] << 8) | READ_BUF[1];
507   Index_Last        = (READ_BUF[2] << 8) | READ_BUF[3];
508   Count_Ring        = (READ_BUF[4] << 8) | READ_BUF[5];
509   Ring_Full         = READ_BUF[6];
510   SENSOR_active       = READ_BUF[7];
511   SENSOR_address[0]   = READ_BUF[8];
512   SENSOR_address[1]   = READ_BUF[9];
513   SENSOR_address[2]   = READ_BUF[10];
514   SENSOR_address[3]   = READ_BUF[11];
515   SENSOR_address[4]   = READ_BUF[12];
516   SENSOR_address[5]   = READ_BUF[13];
517   SENSOR_address[6]   = READ_BUF[14];
518   SENSOR_address[7]   = READ_BUF[15];

520   // Speicher auslesen
521   Read_Bytes(0,1,0,BUF_SIZE);

523   SENSOR_address[8]   = READ_BUF[0];
524   SENSOR_address[9]   = READ_BUF[1];
525   SENSOR_address[10]  = READ_BUF[2];
526   SENSOR_address[11]  = READ_BUF[3];
527   SENSOR_address[12]  = READ_BUF[4];
528   SENSOR_address[13]  = READ_BUF[5];
529   SENSOR_address[14]  = READ_BUF[6];
530   SENSOR_address[15]  = READ_BUF[7];
531   SENSOR_address[16]  = READ_BUF[8];
532   SENSOR_address[17]  = READ_BUF[9];
```

```
533    SENSOR_address[18] = READ_BUF[10];
534    SENSOR_address[19] = READ_BUF[11];
535    SENSOR_address[20] = READ_BUF[12];
536    SENSOR_address[21] = READ_BUF[13];
537    SENSOR_address[22] = READ_BUF[14];
538    SENSOR_address[23] = READ_BUF[15];

540    // Speicher auslesen
541    Read_Bytes(0,2,0,BUF_SIZE);

543    SENSOR_address[24] = READ_BUF[0];
544    SENSOR_address[25] = READ_BUF[1];
545    SENSOR_address[26] = READ_BUF[2];
546    SENSOR_address[27] = READ_BUF[3];
547    SENSOR_address[28] = READ_BUF[4];
548    SENSOR_address[29] = READ_BUF[5];
549    SENSOR_address[30] = READ_BUF[6];
550    SENSOR_address[31] = READ_BUF[7];
551    SENSOR_address[32] = READ_BUF[8];
552    SENSOR_address[33] = READ_BUF[9];
553    SENSOR_address[34] = READ_BUF[10];
554    SENSOR_address[35] = READ_BUF[11];
555    SENSOR_address[36] = READ_BUF[12];
556    SENSOR_address[37] = READ_BUF[13];
557    SENSOR_address[38] = READ_BUF[14];
558    SENSOR_address[39] = READ_BUF[15];

560    // Speicher auslesen
561    Read_Bytes(0,3,0,BUF_SIZE);

563    SENSOR_temp_cal_b[1] = READ_BUF[0];
564    SENSOR_temp_cal_b[2] = READ_BUF[1];
565    SENSOR_temp_cal_b[3] = READ_BUF[2];
566    SENSOR_temp_cal_b[4] = READ_BUF[3];
567    SENSOR_temp_cal_b[5] = READ_BUF[4];
568    SENSOR_temp_cal_b[6] = READ_BUF[5];
569    SENSOR_temp_cal_b[7] = READ_BUF[6];
570    SENSOR_temp_cal_b[8] = READ_BUF[7];
571    SENSOR_temp_cal_b[9] = READ_BUF[8];
572    SENSOR_temp_cal_b[10] = READ_BUF[9];
573    SENSOR_temp_cal_b[11] = READ_BUF[10];
574    SENSOR_temp_cal_b[12] = READ_BUF[11];
575    SENSOR_temp_cal_b[13] = READ_BUF[12];
576    SENSOR_temp_cal_b[14] = READ_BUF[13];
577    SENSOR_temp_cal_b[15] = READ_BUF[14];
578    SENSOR_temp_cal_b[16] = READ_BUF[15];

580    // Speicher auslesen
581    Read_Bytes(0,4,0,BUF_SIZE);

583    SENSOR_temp_cal_b[17] = READ_BUF[0];
584    SENSOR_temp_cal_b[18] = READ_BUF[1];
585    SENSOR_temp_cal_b[19] = READ_BUF[2];
586    SENSOR_temp_cal_b[20] = READ_BUF[3];
587    SENSOR_temp_cal_b[21] = READ_BUF[4];
588    SENSOR_temp_cal_b[22] = READ_BUF[5];
589    SENSOR_temp_cal_b[23] = READ_BUF[6];
590    SENSOR_temp_cal_b[24] = READ_BUF[7];
591    SENSOR_temp_cal_b[25] = READ_BUF[8];
592    SENSOR_temp_cal_b[26] = READ_BUF[9];
```

```
593    SENSOR_temp_cal_b[27] = READ_BUF[10];
594    SENSOR_temp_cal_b[28] = READ_BUF[11];
595    SENSOR_temp_cal_b[29] = READ_BUF[12];
596    SENSOR_temp_cal_b[30] = READ_BUF[13];
597    SENSOR_temp_cal_b[31] = READ_BUF[14];
598    SENSOR_temp_cal_b[32] = READ_BUF[15];

600    // Speicher auslesen
601    Read_Bytes(0,5,0,BUF_SIZE);

603    SENSOR_temp_cal_b[33] = READ_BUF[0];
604    SENSOR_temp_cal_b[34] = READ_BUF[1];
605    SENSOR_temp_cal_b[35] = READ_BUF[2];
606    SENSOR_temp_cal_b[36] = READ_BUF[3];
607    SENSOR_temp_cal_b[37] = READ_BUF[4];
608    SENSOR_temp_cal_b[38] = READ_BUF[5];
609    SENSOR_temp_cal_b[39] = READ_BUF[6];
610    SENSOR_temp_cal_b[40] = READ_BUF[7];
611    AH_MASTER          = (READ_BUF[8]<<8) | READ_BUF[9];
612    CELL_VOLT_MAX       = (READ_BUF[10]<<8) | READ_BUF[11];
613    CELL_VOLT_MIN       = (READ_BUF[12]<<8) | READ_BUF[13];
614    BATT_CAPACITY       = (READ_BUF[14]<<8) | READ_BUF[15];

616    // Speicher auslesen
617    Read_Bytes(0,6,0,BUF_SIZE);

619    CURRENT_tara       = (READ_BUF[0]<<8) | READ_BUF[1];
620    CURRENT_INVERT      = READ_BUF[2];
621  //   0 = READ_BUF[3];
622  //   0 = READ_BUF[4];
623  //   0 = READ_BUF[5];
624  //   0 = READ_BUF[6];
625  //   0 = READ_BUF[7];
626  //   0 = READ_BUF[8];
627  //   0 = READ_BUF[9];
628  //   0 = READ_BUF[10];
629  //   0 = READ_BUF[11];
630  //   0 = READ_BUF[12];
631  //   0 = READ_BUF[13];
632  //   0 = READ_BUF[14];
633  //   0 = READ_BUF[15];
634  //   0 = READ_BUF[16];

636  }

638  /*-----------------------------------------------------------------------------------
639    read data at startup
640  ---------------------------------------------------------------------------------*/
641  void FLASH_read_at_startup(void)
642  /*-------------------------------------------------------------------------------*/
643  {
644    LCD_send_cmd(LCD_LINE1);
645    LCD_send_text("Read contr.data ");
646    LCD_send_cmd(LCD_LINE2);
647    LCD_send_text("YES    NO        ");

649    while(1)
650    {
651      if(BATMON_control_reg & B1)
652      // Button 1 = YES, read controll data from flash
```

```
653      {
654        UART0_send_text("Read data\r\n");
655        FLASH_Read_Controll_Data();
656        BATMON_control_reg &= ~(B1|B2|B3);
657        // clear button state
658        break;
659        // exit while loop
660      }
661      if(BATMON_control_reg & B2)
662        // Button 2 = NO, write init sequence
663      {
664        RADIO_init_frame();
665        // write first frame with actual time and 0 as data
666        BATMON_control_reg &= ~(B1|B2|B3);
667        // clear button state
668        break;
669        // exit while loop
670      }
671    }
672  }
```

Listing 10: Quellcode des Datenloggers (MSP_ functions.c)

```
1  /*-------------------------------------------------------------------------------------
2    Project:      Specialfunctions for the MSP430x1232 and
3            MSP430x169
4    Used components:    MSP430-169STK
5    Date:       10/28/2007
6    Last update:      09/05/2008
7    Author:       Stephan Plaschke

9    Modified by:    Alexander Hoops
10   Last modification:  12/02/2010
11 --------------------------------------------------------------------------------------

13 --------------------------------------------------------------------------------------
14   Headerfiles
15 ------------------------------------------------------------------------------------*/
16 #include <ISR.h>
17 #include <ADC.h>
18 #include <RTC.h>
19 #include <main.h>
20 #include <flash.h>
21 #include <stdio.h>
22 #include <flash.h>
23 #include <stdlib.h>
24 #include <string.h>
25 #include <signal.h>
26 #include <lcd16x2.h>
27 #include <msp430x16x.h>
28 #include <uart_menu.h>
29 #include <MSP_functions.h>

31 /*-------------------------------------------------------------------------------------
32   Static
33 ------------------------------------------------------------------------------------*/
34 static unsigned char *BATMON_menu_msg[17]={
35     " Show sensors    ",
36     " Start recording",
37     " Stop recording ",
```

```
38      " Load contr.data",
39      " Save contr.data",
40      " Set date/time  ",
41      " Backlight      ",
42      " Current tara   ",
43      " Current invert ",
44      " Scan sensors   ",
45      " Calibrate temp ",
46      "NEXT   OK   EXIT",
47      "YES    NO       ",
48      "OFF    ON       ",
49      "6      12   MORE",
50      "24     40   EXIT",
51      "+      OK   EXIT"
52  };

54  /*------------------------------------------------------------------------------------
55    Functions
56  ----------------------------------------------------------------------------------*/

58  /*------------------------------------------------------------------------------------
59    Set DCO to selected frequency
60    clk_multi is the multiplier of the  A_clk (32768KHz)
61    Error_code:
62    0 ->  no error
63    1 ->  oscillator error
64  ----------------------------------------------------------------------------------*/
65  unsigned char DCO_set (unsigned int clk_multi)
66  /*--------------------------------------------------------------------------------*/
67  {
68      unsigned int Compare, Oldcapture = 0;
69      unsigned int Max_turns=10000;
70      unsigned char Error_code=0;

72      BCSCTL1 |= DIVA_3;                  // ACLK= LFXT1CLK/8
73      CCTL2 = CM_1 + CCIS_1 + CAP;          // CAP, ACLK
74      TACTL = TASSEL_2 + MC_2 + TACLR;      // SMCLK, cont-mode, clear

76      while (1)
77      {
78        while (!(CCIFG & CCTL2));      // Wait until capture occured
79        CCTL2 &= ~CCIFG;                 // Clear flag
80        Compare = CCR2;                  // Get current captured SMCLK
81        Compare = Compare - Oldcapture;    // SMCLK difference
82        Oldcapture = CCR2;                     // get current captured SMCLK
83        if (clk_multi == Compare)    // clk_multi = compare then exit
84        {
85          Error_code = 0;        // no error occurs
86          break;
87        }
88        else if (clk_multi < Compare)    // too fast, slow it down
89        {
90            DCOCTL--;
91  /* decrease DCOCTL register
92  * if DCOCTL overrun and basic clock register RSELx bits are not equal 0
93  * decrease RSELx bits                  */
94            if ((DCOCTL == 0xFF) && (!(BCSCTL1 == (XT2OFF + DIVA_3))))
95              BCSCTL1--;
96        }
97        else          // too slow, fast it up
```

```
98              {
99                  DCOCTL++;
100                 /* increase DCOCTL register if DCOCTL overrun and basic clock
101                 register RSELx bits are equal 1 increase RSELx bits           */
102                 if ((DCOCTL == 0x00) && (!(BCSCTL1 == (XT2OFF + 0x07 + DIVA_3))))
103                     BCSCTL1++;
104             }
105
106             if (!(--Max_turns)) // if 10k turns reached exit with error
107             {
108               Error_code = 1;
109               break;
110             }
111         }
112         CCTL2 = 0;              // Stop CCR2
113         TACTL = 0;                // Stop Timer_A
114       BCSCTL1 &= ~DIVA_3;         // restore BCSCTL1
115       BCSCTL2 &= ~(DIVS_3 | DIVM_3);  // restore BCSCTL2
116
117         return (Error_code);
118     }
119
120     /*----------------------------------------------------------------------------------
121       Set XT2 to selected frequency
122       clk_multi is the multiplier of the  A_clk (32768KHz)
123       Error_code:
124       0 ->  no error
125       1 ->  external oscillator error
126       2 ->  clock wrong, set to 4MHZ
127     --------------------------------------------------------------------------------*/
128     unsigned char XT2_set (unsigned int clk_multi)
129     /*------------------------------------------------------------------------------*/
130     {
131       unsigned int Max_turns = 1000;
132       unsigned char i, Error_code = 0;
133
134       _BIC_SR(OSCOFF);          // switch LFXT on
135       BCSCTL1 &= ~XT2OFF;               // XT2on
136
137       do
138       {
139           IFG1 &= ~OFIFG;               // Clear OSCFault flag
140           for (i = 0xFF; i > 0; i--); // Time for flag to set
141
142           if (!(--Max_turns))   // stop after 1k tries, error with ext osc!
143           {
144             Error_code = 1;
145             break;
146           }
147       }
148       while ((IFG1 & OFIFG));           // OSCFault flag still set?
149
150       BCSCTL2 |= SELM_2 | SELS;    // MCLK = SMCLK = XT2 (safe)
151
152       switch (clk_multi)
153       {
154       case MHZ_8:
155         BCSCTL2 &= ~(DIVS_3 | DIVM_3);  // SMCLK&MCLK = XT2 / 1
156         break;
157       case MHZ_4:
```

```
158     BCSCTL2 |= DIVS_1 | DIVM_1;   // SMCLK&MCLK = XT2 / 2
159     break;
160   case MHZ_2:
161     BCSCTL2 |= DIVS_2 | DIVM_2;   // SMCLK&MCLK = XT2 / 4
162     break;
163   case MHZ_1:
164     BCSCTL2 |= DIVS_3 | DIVM_3;   // SMCLK&MCLK = XT2 / 8
165     break;
166   default:
167     Error_code = 2;
168     BCSCTL2 |= DIVS_1 | DIVM_1;   // SMCLK&MCLK = XT2 / 2
169     break;
170   }
171   return (Error_code);
172 }

174 /*------------------------------------------------------------------------------------
175   putchar() defined for use with printf
176   PRINTF_LCD :  output on LCD of the MSP430x169STK board
177   PRINTF_UART:  output through RS232
178 ----------------------------------------------------------------------------------*/
179 int putchar(int c)
180 //-----------------------------------------------------------------------------------
181 {
182   UART0_send((unsigned char)c);
183   return(c);
184 }

186 //*********************************************************************************/
187 /*------------------------------------------------------------------------------------
188   MSP430x169 specific functions
189 ----------------------------------------------------------------------------------*/

191 /*-----------------------------------------------------------------------------------
192   PORTS init
193 ----------------------------------------------------------------------------------*/
194 void PORTS_init(void)
195 /*--------------------------------------------------------------------------------*/
196 {

198   P1SEL = PORT1_SEL;    // Port1 I/O Funktion
199   P1DIR = PORT1_DIR;    // Port1 Ausgang
200   P1IES = PORT1_IES;    // Port1 interrupt edge select
201   P1IE  = PORT1_IE;   // Port1 interrupt enable

203   P2SEL = PORT2_SEL;    // Port2 I/O Funktion
204   P2DIR = PORT2_DIR;    // Port2 Ausgang
205   P2IES = PORT2_IES;    // Port2 interrupt edge select
206   P2IE  = PORT2_IE;   // Port2 interrupt enable

208   P3SEL = PORT3_SEL;    // enable special funktion on PIN3.4, 3.5
209   P3DIR = PORT3_DIR;    // set direction, PIN3.4 OUTPUT

211   P4OUT = PORT4_SEL;    // LCD ini, set PORT4 as output
212     P4DIR = PORT4_DIR;

214     P5SEL = PORT5_SEL;    // Port5 I/O Funktion
215     P5DIR = PORT5_DIR;    // Port5 Ausgang

217   P6SEL = PORT6_SEL;      // P6.x ADC option select
```

```
218     P6DIR = PORT6_DIR;
219   }

221   /*-------------------------------------------------------------------------------
222     ADC0 init
223   -----------------------------------------------------------------------------*/
224   void ADC0_init(void)
225   /*-----------------------------------------------------------------------------*/
226   {
227       ADC12CTL0 = ADC12ON+SHT0_0+REFON+REF2_5V;// ADC12ON / reference on 2.5V
228     ADC12CTL1 = SHP;                  // SHP benutzen
229     ADC12MCTL0 = SREF_1+ADC0_BS;       // Vr+=Vref+, ADC0 bitselect
230   }

232   /*-------------------------------------------------------------------------------
233     get ADC0 result
234   -----------------------------------------------------------------------------*/
235   int ADC0_sampling(void)
236   /*-----------------------------------------------------------------------------*/
237   {
238       ADC12CTL0 |= ADC12SC + ENC;        // Sampling open
239       ADC12CTL0 &= ~ADC12SC;             // Sampling closed, start conversion
240       while ((ADC12CTL1 & ADC12BUSY) == 1);    // ADC12 busy?
241       return(*ADC12MEM);       // return the value read from ADC
242   }

244   /*-------------------------------------------------------------------------------
245     Init USART0, automatic baud rate generation for selected uC frequency
246   -----------------------------------------------------------------------------*/
247   void USART0_init(unsigned char USART0_IRQ, unsigned char USART0_MODE)
248   /*-----------------------------------------------------------------------------*/
249   {
250     UCTL0    =  SWRST + CHAR;  // set 8bit, none parity and 1 stoppbit
251     UTCTL0   |= SSEL_2;     // set SMCLK as USART0_CLK

253     switch (uC_FREQUENCY)    // set UART to selected mode with selected
254             // uC frequency
255     {
256     case MHZ_1:
257       switch (USART_BAUD)
258       {
259       case BAUD_19k2: UBR00 = 0x34;      // see manual
260         UBR10 = 0x00;
261         UMCTL0 = 0x20;
262         break;
263       case BAUD_9600: UBR00 = 0x68;      // see manual
264         UBR10 = 0x00;
265         UMCTL0 = 0x04;
266         break;
267       case BAUD_4800: UBR00 = 0xd0;      // see manual
268         UBR10 = 0x00;
269         UMCTL0 = 0x92;
270         break;
271       default: UBR00 = 0x68;         // see manual
272         UBR10 = 0x00;
273         UMCTL0 = 0x04;
274       break;
275       };
276       break;
277     case MHZ_2:
```

```
278       switch (USART_BAUD)
279       {
280       case BAUD_115k2: UBR00 = 0x11;      // see manual
281         UBR10 = 0x00;
282         UMCTL0 = 0x52;
283         break;
284       case BAUD_19k2: UBR00 = 0x68;      // see manual
285         UBR10 = 0x00;
286         UMCTL0 = 0x04;
287         break;
288       case BAUD_9600: UBR00 = 0xD0;      // see manual
289         UBR10 = 0x00;
290         UMCTL0 = 0x92;
291         break;
292       case BAUD_4800: UBR00 = 0xA0;      // see manual
293         UBR10 = 0x01;
294         UMCTL0 = 0x6D;
295         break;
296       default: UBR00 = 0x68;          // see manual
297         UBR10 = 0x00;
298         UMCTL0 = 0x04;
299       break;
300       };
301       break;
302     case MHZ_4:
303       switch (USART_BAUD)
304       {
305       case BAUD_115k2: UBR00 = 0x22;      // see manual
306         UBR10 = 0x00;
307         UMCTL0 = 0xDD;
308         break;
309       case BAUD_19k2: UBR00 = 0xD0;      // see manual
310         UBR10 = 0x00;
311         UMCTL0 = 0x92;
312         break;
313       case BAUD_9600: UBR00 = 0xA0;      // see manual
314         UBR10 = 0x01;
315         UMCTL0 = 0x6D;
316         break;
317       case BAUD_4800: UBR00 = 0x41;      // see manual
318         UBR10 = 0x03;
319         UMCTL0 = 0x92;
320         break;
321       default: UBR00 = 0x68;          // see manual
322         UBR10 = 0x00;
323         UMCTL0 = 0x04;
324       break;
325       };
326       break;
327     case MHZ_8:
328       switch (USART_BAUD)
329       {
330       case BAUD_115k2: UBR00 = 0x45;      // see manual
331         UBR10 = 0x00;
332         UMCTL0 = 0xAA;
333         break;
334       case BAUD_19k2: UBR00 = 0xa0;      // see manual
335         UBR10 = 0x01;
336         UMCTL0 = 0x6D;
337         break;
```

```
338        case BAUD_9600: UBR00 = 0x41;      // see manual
339          UBR10 = 0x03;
340          UMCTL0 = 0x92;
341          break;
342        case BAUD_4800: UBR00 = 0x82;      // see manual
343          UBR10 = 0x6;
344          UMCTL0 = 0x6D;
345          break;
346        default: UBR00 = 0x68;         // see manual
347          UBR10 = 0x00;
348          UMCTL0 = 0x04;
349        break;
350        };
351        break;
352      }

354      switch (USART0_MODE)
355        {
356        case RX: ME1 |= URXE0;       // enable UART0 RX
357            break;
358        case TX: ME1 |= UTXE0;       // enable UART0 TX
359            break;
360        case RX_TX: ME1 |= (URXE0 | UTXE0);    // enable UART0 TX&RX
361            break;
362        default: ME1 &= ~(UTXE0 | URXE0); // disable UART0 TX&RX
363            break;
364        }

366      UCTL0   &= ~SWRST;         // clear SWRST

368      switch (USART0_IRQ)
369        {
370        case RX_INT: IE1 |= URXIE0;      // enable UART0 RX ISR
371            break;
372        case TX_INT: IE1 |= UTXIE0;        // enable UART0 TX ISR
373            break;
374        case RX_TX_INT: IE1 |= UTXIE0+URXIE0;   // enable UART0 TX&RX ISR
375            break;
376        default: IE1 &= ~UTXIE0+URXIE0;      // disable UART0 TX&RX ISR
377            break;
378        }
379    }

381    /*-------------------------------------------------------------------------------------
382      send character
383    -----------------------------------------------------------------------------------*/
384    void UART0_send(unsigned char out_char)
385    /*---------------------------------------------------------------------------------*/
386    {
387      while(!(UTCTL0&TXEPT));    // look if USART0 still busy
388      TXBUF0 = out_char;       // write char in transmit buffer
389    }

391    /*-------------------------------------------------------------------------------------
392      Send string to USART
393    -----------------------------------------------------------------------------------*/
394    void UART0_send_text(char *out_string)
395    /*---------------------------------------------------------------------------------*/
396    {
397        while (*out_string)        // *out_string != '\0'
```

```
398      {
399          UART0_send(*out_string);     // send byte to LCD
400          out_string++;
401      }
402  }

404  /*------------------------------------------------------------------------------
405    TIMERB init
406  ------------------------------------------------------------------------------*/
407  void TIMERB_init(void)
408  /*----------------------------------------------------------------------------*/
409  {
410    // initialise TIMERB0, ACLK/1, up mode, capture/compare0 int
411    TBCTL   &= ~MC_3;      // disable Timer_B
412    TBCTL   = TBSSEL_1 | TBCLR;    //  set timer register, clear TAR
413    TBCCTL0 = CCIE;        //  set control register
414    TBCCR0  = 500;
415    TBCTL   |= MC_1;       // start Timer_B

417    _EINT();         // enable interrupts
418  }

420  /*------------------------------------------------------------------------------
421    store receive frame
422  ------------------------------------------------------------------------------*/
423  void RADIO_store_frame(void)
424  /*----------------------------------------------------------------------------*/
425  {
426    static DATA_STRUCT Data;
427    DATA_STRUCT *pData = &Data;

429    pData->C0  = RTC_values.day;
430    pData->C1  = RTC_values.month;
431    pData->C2  = (unsigned char)(RTC_values.year >> 8);
432    pData->C3  = (unsigned char)RTC_values.year;
433    pData->C4  = RTC_values.wday;
434    pData->C5  = RTC_values.hr;
435    pData->C6  = RTC_values.min;
436    pData->C7  = RTC_values.sec;
437    pData->C8  = RADIO_frame.frame_byte[3]; //Address
438    pData->C9  = RADIO_frame.frame_byte[4]; //Temp
439    pData->C10 = RADIO_frame.frame_byte[5]; //Temp
440    pData->C11 = RADIO_frame.frame_byte[6]; //Supp
441    pData->C12 = RADIO_frame.frame_byte[7]; //Supp
442    pData->C13 = RADIO_frame.frame_byte[8]; //Cell
443    pData->C14 = RADIO_frame.frame_byte[9]; //Cell
444    pData->C15 = RADIO_frame.frame_byte[10];//CRC

446    Write_Ring_Flash(pData);
447  }

449  /*------------------------------------------------------------------------------
450    init receive frame
451  ------------------------------------------------------------------------------*/
452  void RADIO_init_frame(void)
453  /*----------------------------------------------------------------------------*/
454  {
455    static DATA_STRUCT Data;
456    DATA_STRUCT *pData;
```

```
458   pData = &Data;

460   // store time and data
461   pData->C0  = RTC_values.day;
462   pData->C1  = RTC_values.month;
463   pData->C2  = (unsigned char)(RTC_values.year >> 8);
464   pData->C3  = (unsigned char)RTC_values.year;
465   pData->C4  = RTC_values.wday;
466   pData->C5  = RTC_values.hr;
467   pData->C6  = RTC_values.min;
468   pData->C7  = RTC_values.sec;
469   pData->C8  = 0x00;
470   pData->C9  = 0x00;
471   pData->C10 = 0x00;
472   pData->C11 = 0x00;
473   pData->C12 = 0x00;
474   pData->C13 = 0x00;
475   pData->C14 = 0x00;
476   pData->C15 = 0x00;

478   // write data to flash
479   Write_Ring_Flash(pData);
480 }

482 /*------------------------------------------------------------------------------------
483   init radio unit
484 ----------------------------------------------------------------------------------*/
485 void RADIO_init(void)
486 /*----------------------------------------------------------------------------------*/
487 {
488   // initialise TIMERA0, SMCLK/8, continous mode, capture on rising edge,
489   // capture/compare0 int
490   TACTL    = TASSEL_2 | ID_3 | TACLR;  //  set timer register, clear TAR
491   TACCTL0 = CM_1 | CAP | SCS | CCIE;  //  set control register
492   TACTL   |= MC_2;            // start Timer_A
493 }

495 /*------------------------------------------------------------------------------------
496   enable radio unit
497 ----------------------------------------------------------------------------------*/
498 void RADIO_enable(void)
499 /*----------------------------------------------------------------------------------*/
500 {
501   //  begin with new frame
502   global_string[0] = NULLTERMINATOR;
503   BATMON_control_reg &= ~VALID_DATA_RECEIVED; // clear global value
504   RADIO_rx_state = SE_SEQ_PREPARE;

506   TACCTL0 |= CCIE | CAP;  // enable capture interrupt
507   TACTL   |= MC_2;     // stop Timer_A
508 }

510 /*------------------------------------------------------------------------------------
511   disable radio unit
512 ----------------------------------------------------------------------------------*/
513 void RADIO_disable(void)
514 /*----------------------------------------------------------------------------------*/
515 {
516   TACTL   &= ~MC_2;    // stop Timer_A
517   TACCTL0 &= ~CAP;
```

```
518    TACCTL0 &= ~CCIE;      // disable interrupt
519  }

521  /*--------------------------------------------------------------------------------
522    batmon main menu
523  --------------------------------------------------------------------------------*/
524  void BATMON_menu_options(void)
525  /*--------------------------------------------------------------------------------*/
526  {
527    unsigned char BATMON_menu_position;
528    unsigned char BATMON_menu_counter;
529    unsigned char time_tmp, i;

531    BATMON_menu_position = BATMON_MENU_MAIN;
532    BATMON_menu_counter = 0;

534    while(BATMON_menu_position != BATMON_MENU_EXIT)
535    {
536      switch(BATMON_menu_position)
537      {
538      case BATMON_MENU_START:
539        LCD_send_cmd(LCD_LINE1);
540        LCD_send_text(BATMON_menu_msg[1]);
541        LCD_send_cmd(LCD_LINE2);
542        LCD_send_text(BATMON_menu_msg[12]);

544        while(1)
545        {
546          if(BATMON_control_reg & B1) // Button 1 = YES,
547          {
548            UART_menu_start();
549            BATMON_control_reg &= ~(B1|B2|B3);       // clear button state
550            break;     // exit while loop
551          }
552          if(BATMON_control_reg & B2) // Button 2 = NO,
553          {
554            BATMON_control_reg &= ~(B1|B2|B3);       // clear button state
555            break;     // exit while loop
556          }
557        }

559        BATMON_menu_position = BATMON_MENU_EXIT;
560        break;     // exit switch case

562      case BATMON_MENU_STOP:
563        LCD_send_cmd(LCD_LINE1);
564        LCD_send_text(BATMON_menu_msg[2]);
565        LCD_send_cmd(LCD_LINE2);
566        LCD_send_text(BATMON_menu_msg[12]);

568        while(1)
569        {
570          if(BATMON_control_reg & B1) // Button 1 = YES,
571          {
572            UART_menu_stop();
573            BATMON_control_reg &= ~(B1|B2|B3);
574            // clear button state
575            break;     // exit while loop
576          }
577          if(BATMON_control_reg & B2) // Button 2 = NO,
```

```
578          {
579            BATMON_control_reg &= ~(B1|B2|B3);
580            // clear button state
581            break;     // exit while loop
582          }
583        }

585        BATMON_menu_position = BATMON_MENU_EXIT;
586        break;     // exit switch case

588      case BATMON_MENU_SAVE:
589        LCD_send_cmd(LCD_LINE1);
590        LCD_send_text(BATMON_menu_msg[4]);
591        LCD_send_cmd(LCD_LINE2);
592        LCD_send_text(BATMON_menu_msg[12]);

594        while(1)
595        {
596          if(BATMON_control_reg & B1) // Button 1 = YES
597          {
598            UART_menu_write_controll();
599            BATMON_control_reg &= ~(B1|B2|B3);
600            // clear button state
601            break;       // exit while loop
602          }
603          if(BATMON_control_reg & B2) // Button 2 = NO
604          {
605            BATMON_control_reg &= ~(B1|B2|B3);
606            // clear button state
607            break;       // exit while loop
608          }
609        }

611        BATMON_menu_position = BATMON_MENU_EXIT;
612        break;     // exit switch case

614      case BATMON_MENU_LOAD:
615        LCD_send_cmd(LCD_LINE1);
616        LCD_send_text(BATMON_menu_msg[3]);
617        LCD_send_cmd(LCD_LINE2);
618        LCD_send_text(BATMON_menu_msg[12]);

620        while(1)
621        {
622          if(BATMON_control_reg & B1) // Button 1 = YES
623          {
624            UART_menu_read_controll();
625            BATMON_control_reg &= ~(B1|B2|B3);
626            // clear button state
627            break;       // exit while loop
628          }
629          if(BATMON_control_reg & B2) // Button 2 = NO,
630          {
631            BATMON_control_reg &= ~(B1|B2|B3);
632            // clear button state
633            break;       // exit while loop
634          }
635        }

637        BATMON_menu_position = BATMON_MENU_EXIT;
```

```
638         break;     // exit switch case

640       case BATMON_MENU_RTC:
641         LCD_send_cmd(LCD_LINE1);  // clear first LCD row
642         LCD_send_text("                ");
643         RTC_init();      // enter RTC init mode

645         BATMON_menu_position = BATMON_MENU_EXIT;
646         break;     // exit switch case

648       case BATMON_MENU_SHOW:
649         time_tmp = RTC_values.sec + 5;

651         if (time_tmp > 59)
652           time_tmp -= 60;

654         LCD_send_cmd(LCD_LINE1);
655         LCD_send_text("                ");
656         LCD_send_cmd(LCD_LINE2);
657         LCD_send_text("                ");
658         LCD_send_cmd(LCD_LINE1);

660         while(time_tmp != RTC_values.sec)
661         {
662           for(i=0; i<SENSOR_active; i++)
663           {
664             // jump to second line if maximum of line
665             // one reached
666             if(((i%16) == 0) &&
667               (i!=0)) LCD_send_cmd(LCD_LINE2);

669             // i/8 = byte-, i%8 = bit position in array
670             if((SENSOR_addr_req[(i/8)] >> (i%8)) & 0x01)
671               LCD_send_data('.');
672             else
673               LCD_send_data('*');
674           }

676           LCD_send_cmd(LCD_LINE1);
677         }

679         LCD_send_cmd(LCD_LINE1);
680         LCD_send_text("                ");

682         BATMON_menu_position = BATMON_MENU_EXIT;
683         break;     // exit switch case

685       case BATMON_MENU_MAIN:
686         LCD_send_cmd(LCD_LINE1);
687         LCD_send_text(BATMON_menu_msg[0]);
688         LCD_send_cmd(LCD_LINE2);
689         LCD_send_text(BATMON_menu_msg[11]);

691         while(1)
692         {
693           // Button B1 pressed, scroll through menu
694           if(BATMON_control_reg & B1)
695           {
696             if (++BATMON_menu_counter > 10)
697             {
```

```
698              BATMON_menu_counter = 0;
699            }

701            LCD_send_cmd(LCD_LINE1);
702            LCD_send_text(BATMON_menu_msg
703                [BATMON_menu_counter]);
704            BATMON_control_reg &= ~(B1|B2|B3);
705            // clear button state
706          }

708          // Button B2 pressed, accept menu item
709          else if(BATMON_control_reg & B2)
710          {
711            switch (BATMON_menu_counter)
712            {
713            case 0:
714              BATMON_menu_position =
715                BATMON_MENU_SHOW;
716              break;  // exit switch case
717            case 1:
718              BATMON_menu_position =
719                BATMON_MENU_START;
720              break;  // exit switch case
721            case 2:
722              BATMON_menu_position =
723                BATMON_MENU_STOP;
724              break;  // exit switch case
725            case 3:
726              BATMON_menu_position =
727                BATMON_MENU_LOAD;
728              break;  // exit switch case
729            case 4:
730              BATMON_menu_position =
731                BATMON_MENU_SAVE;
732              break;  // exit switch case
733            case 5:
734              BATMON_menu_position =
735                BATMON_MENU_RTC;
736              break;  // exit switch case
737            case 6:
738              BATMON_menu_position =
739                BATMON_MENU_BL;
740              break;  // exit switch case
741            case 7:
742              BATMON_menu_position =
743                BATMON_MENU_CTAR;
744              break;  // exit switch case
745            case 8:
746              BATMON_menu_position =
747                BATMON_MENU_CINV;
748              break;  // exit switch case
749            case 9:
750              BATMON_menu_position =
751                BATMON_MENU_SCAN;
752              break;  // exit switch case
753            case 10:
754              BATMON_menu_position =
755                BATMON_MENU_CALI;
756              break;  // exit switch case
757            }
```

```
759            BATMON_control_reg &= ~(B1|B2|B3);
760              // clear button state
761            break;     // exit while loop
762          }
763          // Button B3 pressed, exit menu
764          else if(BATMON_control_reg & B3)
765          {
766            if (Enable_clear_lcd == ON)
767            {
768              LCD_send_cmd(LCD_LINE1);
769              LCD_send_text("                ");
770            }
771            else if (Enable_clear_lcd == STATUS)
772            {
773              LCD_send_cmd(LCD_LINE1);
774              LCD_send_text(" Radio disabled ");
775            }
776            LCD_send_time();
777            BATMON_menu_position = BATMON_MENU_EXIT;
778            BATMON_control_reg &= ~(B1|B2|B3);
779              // clear button state
780            break;     // exit while loop
781          }
782        }

784        break;     // exit switch case
785      // switch Backlight ON/OFF
786      case BATMON_MENU_BL:
787        LCD_send_cmd(LCD_LINE1);
788        LCD_send_text(BATMON_menu_msg[6]);
789        LCD_send_cmd(LCD_LINE2);
790        LCD_send_text(BATMON_menu_msg[13]);

792        while(1)
793        {
794          // Button B1 pressed, Backlight ON
795          if(BATMON_control_reg & B1)
796          {
797            BL_OFF;
798            LCD_send_time();
799            UART0_send_text("Backlight OFF\r\n");
800            BATMON_menu_position = BATMON_MENU_EXIT;
801            BATMON_control_reg &= ~(B1|B2|B3);
802              // clear button state
803            break;     // exit while loop
804          }

806          // Button B2 pressed, Backlight OFF
807          else if(BATMON_control_reg & B2)
808          {
809            BL_ON;
810            LCD_send_time();
811            UART0_send_text("Backlight ON\r\n");
812            BATMON_menu_position = BATMON_MENU_EXIT;
813            BATMON_control_reg &= ~(B1|B2|B3);
814              // clear button state
815            break;     // exit while loop
816          }
817        }
```

```
818      break;
819    // Set actual current value at zero  Current TARA
820    case BATMON_MENU_CTAR:
821      LCD_send_cmd(LCD_LINE1);
822      LCD_send_text(BATMON_menu_msg[7]);
823      LCD_send_cmd(LCD_LINE2);
824      LCD_send_text(BATMON_menu_msg[12]);

826      while(1)
827      {
828      // Button B1 pressed, set zero
829        if(BATMON_control_reg & B1)
830        {
831          CURRENT_tara = 0;
832          scan_mode = 4;
833          ADC_start();
834          LCD_send_time();
835          BATMON_menu_position = BATMON_MENU_EXIT;
836          BATMON_control_reg &= ~(B1|B2|B3);
837            // clear button state
838          break;    // exit while loop
839        }

841      // Button B2 pressed, exit menu
842        else if(BATMON_control_reg & B2)
843        {
844          LCD_send_time();
845          BATMON_menu_position = BATMON_MENU_EXIT;
846          BATMON_control_reg &= ~(B1|B2|B3);
847            // clear button state
848          break;    // exit while loop
849        }
850      }
851      break;
852    // invert current values "sensor type specific"
853    case BATMON_MENU_CINV:
854      LCD_send_cmd(LCD_LINE1);
855      LCD_send_text(BATMON_menu_msg[8]);
856      LCD_send_cmd(LCD_LINE2);
857      LCD_send_text(BATMON_menu_msg[12]);

859      while(1)
860      {
861      // Button B1 pressed, invert current values
862        if(BATMON_control_reg & B1)
863        {
864          if (CURRENT_INVERT == 0)
865          {
866            CURRENT_INVERT = 1;
867          }
868          else
869          {
870            CURRENT_INVERT = 0;
871          }
872          LCD_send_time();
873          BATMON_menu_position = BATMON_MENU_EXIT;
874          BATMON_control_reg &= ~(B1|B2|B3);
875            // clear button state
876          break;    // exit while loop
877        }
```

```
879          // Button B2 pressed, exit menu
880            else if(BATMON_control_reg & B2)
881            {
882              LCD_send_time();
883              BATMON_menu_position = BATMON_MENU_EXIT;
884              BATMON_control_reg &= ~(B1|B2|B3);
885                // clear button state
886              break;     // exit while loop
887            }
888          }
889          break;
890        // scan for sensors
891        case BATMON_MENU_SCAN:
892          LCD_send_cmd(LCD_LINE1);
893          LCD_send_text(BATMON_menu_msg[9]);
894          LCD_send_cmd(LCD_LINE2);
895          LCD_send_text(BATMON_menu_msg[14]);

897          while(1)
898          {
899          // Button B1 pressed, scan 6 sensors
900            if(BATMON_control_reg & B1)
901            {
902              BATMON_menu_position = BATMON_MENU_EXIT;
903              BATMON_control_reg &= ~(B1|B2|B3);
904                // clear button state
905              SCAN(6);
906              break;     // exit while loop
907            }

909          // Button B2 pressed, scan 12 sensors
910            else if(BATMON_control_reg & B2)
911            {
912              BATMON_menu_position = BATMON_MENU_EXIT;
913              BATMON_control_reg &= ~(B1|B2|B3);
914                // clear button state
915              SCAN(12);
916              break;     // exit while loop
917            }
918          // Button B3 pressed, more sensors submenu
919            else if(BATMON_control_reg & B3)
920            {
921              LCD_send_cmd(LCD_LINE2);
922              LCD_send_text(BATMON_menu_msg[15]);
923              BATMON_control_reg &= ~(B1|B2|B3);
924                // clear button state
925              while(1)
926              {
927              // Button B1 pressed, scan 24 sensors
928                if(BATMON_control_reg & B1)
929                {
930                  BATMON_menu_position =
931                    BATMON_MENU_EXIT;
932                  BATMON_control_reg &=
933                    ~(B1|B2|B3);
934                  // clear button state
935                  SCAN(24);
936                  break;  // exit while loop
937                }
```

```
939            // Button B2 pressed, scan 40 sensors
940              else if(BATMON_control_reg & B2)
941              {
942                BATMON_menu_position =
943                  BATMON_MENU_EXIT;
944                BATMON_control_reg &=
945                  ~(B1|B2|B3);
946                // clear button state
947                SCAN(40);
948                break;   // exit while loop
949              }
950            // Button B3 pressed, exit menu
951              else if(BATMON_control_reg & B3)
952              {
953                LCD_send_time();
954                BATMON_menu_position =
955                  BATMON_MENU_EXIT;
956                BATMON_control_reg &=
957                  ~(B1|B2|B3);
958                // clear button state
959                break;   // exit while loop
960              }
961            }
962            break;     // exit while loop
963          }
964        }
965        break;
966      // calibrate temperature sensors
967      case BATMON_MENU_CALI:
968        LCD_send_cmd(LCD_LINE1);
969        LCD_send_text(BATMON_menu_msg[10]);
970        LCD_send_cmd(LCD_LINE2);
971        LCD_send_text(BATMON_menu_msg[12]);
972        unsigned long Temperature=200;
973        unsigned char tenner,ones,dezi;
974        while(1)
975        {
976        // Button B1 pressed, submenu
977          if(BATMON_control_reg & B1)
978          {
979            LCD_send_cmd(LCD_LINE1);
980            LCD_send_text(" Temp:");
981            tenner = (Temperature%1000)/100;
982            ones = (Temperature%100)/10;
983            dezi = Temperature %10;
984            LCD_send_data(tenner+48);
985            LCD_send_data(ones+48);
986            LCD_send_text(".");
987            LCD_send_data(dezi+48);
988            LCD_send_data(223);
989            LCD_send_text("C     ");
990            LCD_send_cmd(LCD_LINE2);
991            LCD_send_text(BATMON_menu_msg[16]);
992            BATMON_control_reg &= ~(B1|B2|B3);
993              // clear button state
994            while(1)
995            {
996            // Button B1 pressed, increment start value
997              if(BATMON_control_reg & B1)
```

```
998           {
999             Temperature+=5;
1000            if (Temperature > 500)
1001            {
1002              Temperature = 100;
1003            }
1004            LCD_send_cmd(LCD_LINE1);
1005            LCD_send_text(" Temp:");
1006            tenner =
1007            (Temperature%1000)/100;
1008            ones = (Temperature%100)/10;
1009            dezi = Temperature %10;
1010            LCD_send_data(tenner+48);
1011            LCD_send_data(ones+48);
1012            LCD_send_text(".");
1013            LCD_send_data(dezi+48);
1014            LCD_send_data(223);
1015            LCD_send_text("C    ");
1016            BATMON_control_reg &=
1017              ~(B1|B2|B3);
1018            // clear button state
1019          }

1021          // Button B2 pressed, calibrate
1022          else if(BATMON_control_reg & B2)
1023          {
1024            BATMON_menu_position =
1025              BATMON_MENU_EXIT;
1026            BATMON_control_reg &=
1027              ~(B1|B2|B3);
1028            // clear button state
1029            CALIBRATE(Temperature);
1030            break;  // exit while loop
1031          }
1032          // Button B3 pressed, exit
1033          else if(BATMON_control_reg & B3)
1034          {
1035            LCD_send_time();
1036            BATMON_menu_position =
1037              BATMON_MENU_EXIT;
1038            BATMON_control_reg &=
1039              ~(B1|B2|B3);
1040            // clear button state
1041            break;  // exit while loop
1042          }
1043        }
1044      }
1045      // Button B2 pressed, EXIT
1046      else if(BATMON_control_reg & B2)
1047      {
1048        BATMON_menu_position = BATMON_MENU_EXIT;
1049        BATMON_control_reg &= ~(B1|B2|B3);
1050            // clear button state
1051        break;    // exit while loop
1052      }
1053      else if (BATMON_menu_position == BATMON_MENU_EXIT)
1054      {
1055        break;    // exit while loop
1056      }
1057    }
```

```
1058        }
1059      }
1060      // set text in first LCD line
1061      if (Enable_clear_lcd == STATUS)
1062      {

1064        LCD_send_cmd(LCD_LINE1);
1065        LCD_send_text(" Radio disabled ");
1066      }
1067      else if (Enable_clear_lcd == ON)
1068      {
1069        LCD_send_cmd(LCD_LINE1);
1070        LCD_send_text("                ");
1071      }
1072      // set time on second LCD line
1073      LCD_send_time();
1074    }
1075    /*----------------------------------------------------------------------------------
1076      Scan function - scans for sensors
1077    --------------------------------------------------------------------------------*/
1078    void SCAN(unsigned char sensors)
1079    /*--------------------------------------------------------------------------------*/
1080    {
1081      unsigned char i;
1082      if ((sensors == 6) || (sensors == 12) || (sensors == 24) || (sensors == 40))
1083      {
1084        //first clear all addresses
1085        for (i=0; i < 40; i++)
1086        {
1087          SENSOR_address[i] = 0;
1088        }

1090        SENSOR_active = sensors;  //change numer of active sensors
1091        UART_menu_stop();   //stop uart so scanning cannot be interrupted
1092        LCD_send_cmd(LCD_LINE1);  //set cursor to first line of display
1093        LCD_send_text("  Scanning...   ");  //write text to display
1094        UART0_send_text("Scan enabled \r\n"); //write to uart
1095        LCD_send_cmd(LCD_LINE2);  //set cursor to second line of display
1096        LCD_send_text("Stop            ");  //write to display
1097        Enable_clear_lcd = OFF; //disable clear lcd flag
1098        scan_mode = 1;  //enable scanmode in interrupt
1099        RADIO_enable(); //enable receiver

1101        //Button 1 stopps scan
1102        while (scan_mode == 1)
1103        {
1104          if(BATMON_control_reg & B1)
1105          {
1106            BATMON_control_reg &= ~(B1|B2|B3);
1107            // clear button state
1108            scan_mode = 0;
1109            RADIO_disable();
1110            Enable_clear_lcd = ON;
1111            LCD_send_cmd(LCD_LINE1);
1112            LCD_send_text(" Scan halted    ");
1113            UART0_send_text("Scan halted \r\n");
1114          }
1115        }
1116        LCD_send_time();
1117      }
```

```
1118    //wrong number of sensors --> error try again
1119    else
1120    {
1121      LCD_send_cmd(LCD_LINE1);
1122      LCD_send_text(" Scan impossible");
1123      UART0_send_text("Scan impossible \r\n");
1124      Enable_clear_lcd = ON;
1125    }
1126  }
1127  /*-----------------------------------------------------------------------------------
1128    Calibration function
1129  ---------------------------------------------------------------------------------*/
1130  void CALIBRATE(unsigned long Temperature)
1131  /*---------------------------------------------------------------------------------*/
1132  {
1133    UART_menu_stop();
1134    temp_correct_value = Temperature*8/10;
1135    UART0_send_text("Calibrating... \r\n");
1136    LCD_send_cmd(LCD_LINE1);
1137    LCD_send_text(" Calibrating... ");
1138    LCD_send_cmd(LCD_LINE2);
1139    LCD_send_text("Stop            ");
1140    Enable_clear_lcd = OFF;
1141    scan_mode = 2;
1142    RADIO_enable();
1143    while (scan_mode > 1)
1144    {
1145      if(BATMON_control_reg & B1)
1146      {
1147        BATMON_control_reg &= ~(B1|B2|B3);
1148        // clear button state
1149        scan_mode = 0;
1150        RADIO_disable();
1151        Enable_clear_lcd = ON;
1152        LCD_send_cmd(LCD_LINE1);
1153        LCD_send_text(" Cali. halted   ");
1154        UART0_send_text("Calibration halted \r\n");
1155      }
1156    }
1157    LCD_send_time();
1158  }
```

Listing 11: Quellcode des Datenloggers (RTC.c)

```
1   /*-------------------------------------------------------------------------------
2     Project:       MSP430 RTC source file
3     Used components:    MSP430-169STK
4     Date:        02/08/2008
5     Last update:       09/05/2008
6     Author:        Stephan Plaschke

8     Modified by:     Alexander Hoops
9     Last modification:  12/02/2010
10  -----------------------------------------------------------------------------------

12  -----------------------------------------------------------------------------------
13    Headerfiles
14  ---------------------------------------------------------------------------------*/
15  #include <RTC.h>
16  #include <ADC.h>
```

```
17 #include <main.h>
18 #include <stdio.h>
19 #include <signal.h>    // needed for interrupt
20 #include <lcd16x2.h>
21 #include <msp430x16x.h>
22 #include <MSP_functions.h>

24 /*-------------------------------------------------------------------------------------
25   Static
26 -----------------------------------------------------------------------------------*/
27 static unsigned char *RTC_days[7]={
28     "MON ",
29     "TUE ",
30     "WED ",
31     "THU ",
32     "FRY ",
33     "SAT ",
34     "SUN "
35 };

37 static unsigned char *RTC_months[12]={
38     "Jan",
39     "Feb",
40     "Mar",
41     "Apr",
42     "May",
43     "Jun",
44     "Jul",
45     "Aug",
46     "Sep",
47     "Oct",
48     "Nov",
49     "Dec"
50 };
51 /*-------------------------------------------------------------------------------------
52   RTC init (initialze watchdog timer)
53 -----------------------------------------------------------------------------------*/
54 void RTC_init(void)
55 /*---------------------------------------------------------------------------------*/
56 {
57   unsigned char time_char;

59   RTC_menue_position = RTC_MENUE_DAY;
60   BATMON_control_reg &= ~(B1 | B2 | B3);

62   /* initialise watchdog timer used for BLINKY, intervall mode, 1/4sec  */
63   WDTCTL = WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS_1;   /* 1/4sec intervall mode
       */
64   WDT_function = BLINKY;
65   IE1 |= WDTIE;      /* enable Watchdog interrupt */

67   /* initialise TIMERB0, ACLK/1, up mode, capture/compare0 int*/
68   /*TBCTL   &= ~MC_3;            // disable Timer_B
69   TBCTL    = TBSSEL_1 | TBCLR;        //  set timer register, clear TAR
70   TBCCTL0 = CCIE;              //  set control register
71   TBCCR0   = 500;
72   TBCTL    |= MC_1;            // start Timer_B
73   */
74   /* global interrupt enable  */
75   _EINT();
```

```
77    /* display real time and weekday */
78    LCD_send_date();
79  /*-----------------------------------------------------------------------------------
80   Edit the date
81  -----------------------------------------------------------------------------------*/
82    do
83    {
84  /* if B1 and B2 are pressed exit the RTC_EDIT_MODE  */
85      if ((BATMON_control_reg & B1) && (BATMON_control_reg & B2))
86      {
87        BATMON_control_reg &= ~(B1 | B2);
88        RTC_menue_position = RTC_MENUE_EXIT;
89      }
90  /* only B1 is pressed decrement the editable value(day,sec,min,hr)  */
91      else if(BATMON_control_reg & B1)
92      {
93          BATMON_control_reg &= ~B1;     // clear button state variable
94          LED2_TOGGLE;
95          switch(RTC_menue_position)
96          {
97          case RTC_MENUE_DAY: if(RTC_values.day == 1) RTC_values.day = 31;
98            else RTC_values.day--;
99            break;
100         case RTC_MENUE_MONTH: if(RTC_values.month == 0) RTC_values.month = 11;
101           else RTC_values.month--;
102           break;
103         case RTC_MENUE_YEAR: if(RTC_values.year == 0) RTC_values.year = 4000;
104           else RTC_values.year--;
105           break;
106         default: break;
107         }
108     }
109 /* only B2 is pressed increment the editable value(day,sec,min,hr)  */
110     else if(BATMON_control_reg & B2)
111     {
112         BATMON_control_reg &= ~B2;     // clear button state variable
113         LED2_TOGGLE;
114         switch(RTC_menue_position)
115         {
116         case RTC_MENUE_DAY: if(RTC_values.day == 31) RTC_values.day = 1;
117           else RTC_values.day++;
118           break;
119         case RTC_MENUE_MONTH: if(RTC_values.month == 11) RTC_values.month = 0;
120           else RTC_values.month++;
121           break;
122         case RTC_MENUE_YEAR: if(RTC_values.year == 4000) RTC_values.year = 0;
123           else RTC_values.year++;
124           break;
125         default: break;
126         }
127     }
128 /* only B3 is pressed change the editable value */
129     else if(BATMON_control_reg & B3)
130     {
131       BATMON_control_reg &= ~B3;  // clear button state variable
132       LED2_TOGGLE;
133       /* display previews editable value */
134       switch(RTC_menue_position)
135       {
```

```
136         case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
137           time_char = (RTC_values.day / 10) + 48;
138           LCD_send_data(time_char);
139           time_char = (RTC_values.day % 10) + 48;
140           LCD_send_data(time_char);
141           break;
142         case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
143           LCD_send_text(RTC_months[RTC_values.month]);
144           break;
145         case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
146           time_char = RTC_values.year / 1000;
147           LCD_send_data(time_char+48);
148           time_char = (RTC_values.year / 100) - ((RTC_values.year/1000) * 10);
149           LCD_send_data(time_char+48);
150           time_char = (RTC_values.year / 10) - ((RTC_values.year/100) * 10);
151           LCD_send_data(time_char+48);
152           time_char = RTC_values.year % 10;
153           LCD_send_data(time_char+48);
154           break;
155         default: break;
156       }
157       /* change editable value */
158       if(RTC_menue_position == RTC_MENUE_YEAR)
159         RTC_menue_position = RTC_MENUE_DAY;
160       else
161         RTC_menue_position++;
162     }
163   }
164   while(RTC_menue_position != RTC_MENUE_EXIT);

166 /*-------------------------------------------------------------------------------------
167  Edit the weekday and time
168 -------------------------------------------------------------------------------------*/
169   /* display real time and weekday */
170   IE1 &= ~WDTIE;              // disable Watchdog interrupt
171   RTC_menue_position = RTC_MENUE_WDAY;
172   LCD_send_time();
173   IE1 |= WDTIE;       // enable Watchdog interrupt

175   do
176   {
177 /* if B1 and B2 are pressed exit the RTC_EDIT_MODE  */
178     if ((BATMON_control_reg & B1) && (BATMON_control_reg &B2))
179     {
180       BATMON_control_reg &= ~(B1 | B2);
181       RTC_menue_position = RTC_MENUE_EXIT;
182     }
183 /* only B1 is pressed decrement the editable value(day,sec,min,hr)  */
184     else if(BATMON_control_reg & B1)
185     {
186         BATMON_control_reg &= ~B1;     // clear button state variable
187         LED2_TOGGLE;
188         switch(RTC_menue_position)
189         {
190         case RTC_MENUE_WDAY: if(RTC_values.wday == MON) RTC_values.wday = SUN;
191           else RTC_values.wday--;
192           break;
193         case RTC_MENUE_HR: if(RTC_values.hr == 0) RTC_values.hr = 23;
194           else RTC_values.hr--;
195           break;
```

```
196         case RTC_MENUE_MIN: if(RTC_values.min == 0) RTC_values.min = 59;
197           else RTC_values.min--;
198           break;
199         case RTC_MENUE_SEC: if(RTC_values.sec == 0) RTC_values.sec = 59;
200           else RTC_values.sec--;
201           break;
202         default: break;
203         }
204     }
205 /* only B2 is pressed increment the editable value(day,sec,min,hr)  */
206     else if(BATMON_control_reg & B2)
207     {
208         BATMON_control_reg &= ~B2;     // clear button state variable
209         LED2_TOGGLE;
210         switch(RTC_menue_position)
211         {
212         case RTC_MENUE_WDAY: if(RTC_values.wday == SUN) RTC_values.wday = MON;
213           else RTC_values.wday++;
214           break;
215         case RTC_MENUE_HR: if(RTC_values.hr == 23) RTC_values.hr = 0;
216           else RTC_values.hr++;
217           break;
218         case RTC_MENUE_MIN: if(RTC_values.min == 59) RTC_values.min = 0;
219           else RTC_values.min++;
220           break;
221         case RTC_MENUE_SEC: if(RTC_values.sec == 59) RTC_values.sec = 0;
222           else RTC_values.sec++;
223           break;
224         default: break;
225         }
226     }
227 /* only B3 is pressed change the editable value */
228     else if(BATMON_control_reg & B3)
229     {
230       BATMON_control_reg &= ~B3;  // clear button state variable
231       LED2_TOGGLE;
232       /* display previews editable value */
233       switch(RTC_menue_position)
234       {
235         case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
236           LCD_send_text(RTC_days[RTC_values.wday]);
237           break;
238         case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
239           time_char = (RTC_values.hr / 10) + 48;
240           LCD_send_data(time_char);
241           time_char = (RTC_values.hr % 10) + 48;
242           LCD_send_data(time_char);
243           break;
244         case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
245           time_char = (RTC_values.min / 10) + 48;
246           LCD_send_data(time_char);
247           time_char = (RTC_values.min % 10) + 48;
248           LCD_send_data(time_char);
249           break;
250         case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
251           time_char = (RTC_values.sec / 10) + 48;
252           LCD_send_data(time_char);
253           time_char = (RTC_values.sec % 10) + 48;
254           LCD_send_data(time_char);
255           break;
```

```
256           default: break;
257         }
258         /* change editable value */
259         if(RTC_menue_position == RTC_MENUE_SEC)
260           RTC_menue_position = RTC_MENUE_WDAY;
261         else
262           RTC_menue_position++;
263       }
264     }
265     while(RTC_menue_position != RTC_MENUE_EXIT);

267     /* initialise watchdog timer used for RTC, intervall mode, 1sec */
268     WDTCTL = WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL;    /* 1sec intervall mode  */
269     WDT_function = RTC;
270     IE1 |= WDTIE;                    /* enable Watchdog interrupt  */

272     /* global interrupt disable */
273     //_DINT();

275     // switch LEDS off
276     LED1_OFF;
277     LED2_OFF;

279     /* display real time and day */
280     LCD_send_time();

282     if (Enable_clear_lcd == STATUS)
283     {
284       LCD_send_cmd(LCD_LINE1);
285       LCD_send_text(" Radio disabled ");
286     }
287     else if(Enable_clear_lcd == ON)
288     {
289       LCD_send_cmd(LCD_LINE1);
290       LCD_send_text("                ");
291     }
292   }

294   /*-----------------------------------------------------------------------------------
295     RTC compare tmp and real time
296   ---------------------------------------------------------------------------------*/
297   void RTC_compare(RTC_MSP430 *RTC_tmp)
298   /*---------------------------------------------------------------------------------*/
299   {
300     unsigned char time_char;

302     /* compare old RTC values with real values */
303     if(RTC_values.wday != RTC_tmp->wday)
304     {
305       RTC_tmp->wday = RTC_values.wday;
306       LCD_send_cmd(LCD_RTC_WDAY);
307       LCD_send_text(RTC_days[RTC_values.wday]);
308     }
309     if(RTC_values.hr != RTC_tmp->hr)
310     {
311       RTC_tmp->hr = RTC_values.hr;
312       LCD_send_cmd(LCD_RTC_HR);
313       time_char = (RTC_values.hr / 10) + 48;
314       LCD_send_data(time_char);
315       time_char = (RTC_values.hr % 10) + 48;
```

```
316      LCD_send_data(time_char);
317    }
318    if(RTC_values.min != RTC_tmp->min)
319    {
320      RTC_tmp->min = RTC_values.min;
321      LCD_send_cmd(LCD_RTC_MIN);
322      time_char = (RTC_values.min / 10) + 48;
323      LCD_send_data(time_char);
324      time_char = (RTC_values.min % 10) + 48;
325      LCD_send_data(time_char);
326    }
327    if(RTC_values.sec != RTC_tmp->sec)
328    {
329      RTC_tmp->sec = RTC_values.sec;
330      LCD_send_cmd(LCD_RTC_SEC);
331      time_char = (RTC_values.sec / 10) + 48;
332      LCD_send_data(time_char);
333      time_char = (RTC_values.sec % 10) + 48;
334      LCD_send_data(time_char);
335    }
336  }

338  /*-----------------------------------------------------------------------------------
339    RTC startup, set time/date or init timer only
340  -----------------------------------------------------------------------------------*/
341  void RTC_startup(void)
342  //-----------------------------------------------------------------------------------
343  {
344    LCD_send_cmd(LCD_LINE1);
345    LCD_send_text(" Set date/time  ");
346    LCD_send_cmd(LCD_LINE2);
347    LCD_send_text("YES    NO       ");

349    while(1)
350    {
351      if(BATMON_control_reg & B1) // Button 1 = YES,
352      {
353        RTC_init();
354        BATMON_control_reg &= ~(B1|B2|B3);       // clear button state
355        break;           // exit while loop
356      }
357      if(BATMON_control_reg & B2) // Button 2 = NO,
358      {
359        /* initialise watchdog timer used for RTC, intervall mode, 1sec */
360        WDTCTL = WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL;   /* 1sec intervall mode  */
361        WDT_function = RTC;
362        IE1 |= WDTIE;                        /* enable Watchdog interrupt  */

364        /* display real time and day */
365        LCD_send_time();
366        BATMON_control_reg &= ~(B1|B2|B3);       // clear button state
367        break;           // exit while loop
368      }
369    }
370  }

372  /*-----------------------------------------------------------------------------------
373    Watchdog interrupt service routine
374  -----------------------------------------------------------------------------------*/
```

```
376 interrupt (WDT_VECTOR) WATCH_DOG(void)
377 /*---------------------------------------------------------------------------------*/
378 {
379   static uint8_t WDT_tmp;
380   static uint8_t BLINKY_state;

382   unsigned char time_char, j, address_byte, address_bit, i;

384   //store measurments every 1 sec if measure mode is set
385   //-----------------------------------------------------------------
386   if (measuring == 1) {
387     for (i = 0; i < (SENSOR_active + 1); i++) //for each active sensor + current sensor
388     {
389       if (i < SENSOR_active)
390       {
391         address_byte = i / 8; //count address byte index every 8 bit
392         address_bit = i % 8; //count bit index

394         //only store data of requested sensors
395         if (SENSOR_addr_req[address_byte] & (0x01 << address_bit))
396         {
397           for (j = 0; j < 8; j++)
398           {
399             RADIO_frame.frame_byte[j + 3] = SENSOR_last_data[j][i];
400           }
401           RADIO_store_frame();
402         }
403       }
404       //store current sensor
405       else {
406         RADIO_frame.frame_byte[3] = 0;  //Current sensor has address '0'
407         RADIO_frame.frame_byte[4] = 0;
408         RADIO_frame.frame_byte[5] = (unsigned char) AH_last_minute_IN_OUT;
409         RADIO_frame.frame_byte[6] = 0;
410         RADIO_frame.frame_byte[7] = 0;
411         RADIO_frame.frame_byte[8] = AH_last_sec >> 8;
412         RADIO_frame.frame_byte[9] = (unsigned char) AH_last_sec & 0xff;
413         RADIO_frame.frame_byte[10] = 0;
414         RADIO_store_frame();
415         Check_BAT();
416       }
417     }
418   }
419   switch (SENSOR_active) {
420   case 12: // 12 sensors are configured
421     SENSOR_addr_req[0] = 0xFF; // set required sensors values
422     SENSOR_addr_req[1] = 0x0F;
423     SENSOR_addr_receiv[0] = 0xFF;
424     SENSOR_addr_receiv[1] = 0x0F;
425     break;
426   case 24: // 24 sensors are configured
427     SENSOR_addr_req[0] = 0xFF; // set required sensors
428     SENSOR_addr_req[1] = 0xFF;
429     SENSOR_addr_req[2] = 0xFF;
430     SENSOR_addr_req[3] = 0x00;
431     SENSOR_addr_receiv[0] = 0xFF;
432     SENSOR_addr_receiv[1] = 0xFF;
433     SENSOR_addr_receiv[2] = 0xFF;
434     SENSOR_addr_receiv[3] = 0x00;
435     break;
```

```
436      case 40: // 24 sensors are configured
437        SENSOR_addr_req[0] = 0xFF; // set required sensors
438        SENSOR_addr_req[1] = 0xFF;
439        SENSOR_addr_req[2] = 0xFF;
440        SENSOR_addr_req[3] = 0xFF;
441        SENSOR_addr_req[4] = 0xFF;
442        SENSOR_addr_receiv[0] = 0xFF;
443        SENSOR_addr_receiv[1] = 0xFF;
444        SENSOR_addr_receiv[2] = 0xFF;
445        SENSOR_addr_receiv[3] = 0xFF;
446        SENSOR_addr_receiv[4] = 0xFF;
447        break;
448      default: // 6 sensors are configured
449        SENSOR_addr_req[0] = 0x3F; // set required sensors
450        SENSOR_addr_receiv[0] = 0x3F;
451        break;
452      }
453      //--------------------------------------------------------

455      switch (WDT_function) {
456    /* 1 sec timer for the real time clock  */

458          // set sensors active every 60 seconds

460      case RTC:
461        if(++RTC_values.sec == 60)        // 60 seconds reached?  --> save data to flash
              when measuring is enabled
462        {
463          RTC_values.sec = 0;

465          if(++RTC_values.min == 60)          // 60 minutes reached?
466          {
467            RTC_values.min = 0;
468            if(++RTC_values.hr == 24)        // 0 o'clock reached?
469            {
470              RTC_values.hr = 0;
471              if ((RTC_values.year % 4) == 0)       // leap year
472              {
473                RTC_values.day++;
474                if((RTC_values.month == 1)&&(RTC_values.day==30)) // february
475                {
476                  RTC_values.month++;
477                  RTC_values.day = 1;
478                }
479              }
480              else                        // normal year
481              {
482                RTC_values.day++;
483                if((RTC_values.month == 1)&&(RTC_values.day==29))
484                {
485                  RTC_values.month++;
486                  RTC_values.day = 1;
487                }
488              }

490              // 30 day update (April, June, August, October, December)
491              if (((RTC_values.month == 3) || (RTC_values.month == 5) || (RTC_values.month
                    == 7) ||
492                    (RTC_values.month == 9) || (RTC_values.month == 11)) && RTC_values.day ==
                        31)
```

```
493            {
494              RTC_values.day = 1;
495              RTC_values.month++;
496            }

498            // 31 day update (january, March, Mai, July, September, November)
499            if (RTC_values.day == 32)
500            {
501              RTC_values.day = 1;
502              RTC_values.month++;
503            }

505            // year update
506            if(RTC_values.month == 12)
507            {
508              RTC_values.month = 0;
509              RTC_values.year++;
510            }

512            // weekday update
513            if(++RTC_values.wday == (SUN+1))
514              RTC_values.wday = MON;
515          }
516        }
517      }
518      if (Enable_clear_lcd & ON)    // if radio enabled clear LCD/reset rx process
519      {
520        // Clear LCD active message every 4 seconds
521        if( (!((RTC_values.sec+1) % 4)))
522          BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
523        // reset rx process every 30 seconds
524        if(!((RTC_values.sec+1) % 30))
525        {
526          if (measuring == 1)
527          {
528            RADIO_enable();
529          }
530        }
531      }
532      break;
533  /* switch every 0,5 sec the editable value on or off, only in RTC_EDIT_MODE */
534    case BLINKY:
535      if(++WDT_tmp == 2)
536      {
537        WDT_tmp = 0;
538        LED1_TOGGLE;
539        if(BLINKY_state ^= 0x01)
540        {
541          switch (RTC_menue_position)
542          {
543          case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
544            LCD_send_text(RTC_days[RTC_values.wday]);
545            break;
546          case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
547            time_char = (RTC_values.hr / 10) + 48;
548            LCD_send_data(time_char);
549            time_char = (RTC_values.hr % 10) + 48;
550            LCD_send_data(time_char);
551            break;
552          case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
```

```
553             time_char = (RTC_values.min / 10) + 48;
554             LCD_send_data(time_char);
555             time_char = (RTC_values.min % 10) + 48;
556             LCD_send_data(time_char);
557             break;
558           case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
559             time_char = (RTC_values.sec / 10) + 48;
560             LCD_send_data(time_char);
561             time_char = (RTC_values.sec % 10) + 48;
562             LCD_send_data(time_char);
563             break;
564           case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
565             time_char = (RTC_values.day / 10) + 48;
566             LCD_send_data(time_char);
567             time_char = (RTC_values.day % 10) + 48;
568             LCD_send_data(time_char);
569             break;
570           case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
571             LCD_send_text(RTC_months[RTC_values.month]);
572             break;
573           case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
574             time_char = RTC_values.year / 1000;
575             LCD_send_data(time_char+48);
576             time_char = (RTC_values.year / 100) - ((RTC_values.year/1000) * 10);
577             LCD_send_data(time_char+48);
578             time_char = (RTC_values.year / 10) - ((RTC_values.year/100) * 10);
579             LCD_send_data(time_char+48);
580             time_char = RTC_values.year % 10;
581             LCD_send_data(time_char+48);
582             break;
583           }
584         }
585         else
586         {
587           // update screen
588           BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
589           switch (RTC_menue_position)
590           {
591           case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
592             LCD_send_text("   ");
593             break;
594           case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
595             LCD_send_text("  ");
596             break;
597           case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
598             LCD_send_text("  ");
599             break;
600           case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
601             LCD_send_text("  ");
602             break;
603           case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
604             LCD_send_text("  ");
605             break;
606           case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
607             LCD_send_text("   ");
608             break;
609           case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
610             LCD_send_text("    ");
611             break;
612           }
```

```
613         }
614       }
615       break;
616     default: break;
617     }
618     IFG1 &= ~WDTIFG;  /* clear watchdog interrupt flag  */
619 }
```

Listing 12: Quellcode des Datenloggers (lcd16x2.c)

```
1  /*------------------------------------------------------------------------------
2    Project:      LCD16x2 functions
3    Used components:    MSP430-169STK
4    Date:       10/28/2007
5    Last update:      04/04/2008
6    Author:       Stephan Plaschke

8    Modified by:    Alexander Hoops
9    Last modification:  12/02/2010
10 ------------------------------------------------------------------------------*/

12 /*------------------------------------------------------------------------------
13   Headerfiles
14 ------------------------------------------------------------------------------*/
15 #include <main.h>
16 #include <lcd16x2.h>
17 #include <msp430x16x.h>
18 #include <ADC.h>
19 //#include <MSP_functions.h>

21 /*------------------------------------------------------------------------------
22   Static
23 ------------------------------------------------------------------------------*/
24 static unsigned char *RTC_days[7]={
25     "MON ",
26     "TUE ",
27     "WED ",
28     "THU ",
29     "FRY ",
30     "SAT ",
31     "SUN "
32 };

34 static unsigned char *RTC_months[12]={
35     "Jan",
36     "Feb",
37     "Mar",
38     "Apr",
39     "May",
40     "Jun",
41     "Jul",
42     "Aug",
43     "Sep",
44     "Oct",
45     "Nov",
46     "Dec"
47 };

49 /*------------------------------------------------------------------------------
50   x ms delay loop
```

```
51   ----------------------------------------------------------------------------------*/
52   void LCD_delay_ms(unsigned int LCD_delay_ms)
53   /*--------------------------------------------------------------------------------*/
54   {
55     unsigned int LCD_delay_loop;

57     switch (uC_FREQUENCY)
58     {
59           case MHZ_1: LCD_delay_loop = 73;  //(88*b) cycles (for 1MHz)
60                 break;
61       case MHZ_2: LCD_delay_loop = 150; //(175*b) cycles (for 2MHz)
62             break;
63       case MHZ_4: LCD_delay_loop = 304; //(350*b) cycles (for 4MHz)
64             break;
65       case MHZ_8: LCD_delay_loop = 610; //(610*b) cycles (for 8MHz)
66             break;
67       default: break;
68     }

70       for (; LCD_delay_ms>0; LCD_delay_ms--)
71       {
72         LCD_delay(LCD_delay_loop);
73       }
74   }

76   /*----------------------------------------------------------------------------------
77       delay loop
78     counts x times
79   ----------------------------------------------------------------------------------*/
80   void LCD_delay(unsigned int LCD_delay_loop)
81   /*--------------------------------------------------------------------------------*/
82   {
83     for(; LCD_delay_loop>0; LCD_delay_loop--) _NOP();
84   }

86   /*----------------------------------------------------------------------------------
87     Init 16x2LCD
88   ----------------------------------------------------------------------------------*/
89   void LCD_init(void)
90   /*--------------------------------------------------------------------------------*/
91   {
92     LCD_DATA_PORT &= ~LCD_RS_PIN;    // sets LCD in COMMAND MODE
93       LCD_delay_ms(15);            // Delay ca.15ms

95       LCD_DATA_PORT |= BIT4 | BIT5;        // D7-D4 = 0011
96       LCD_DATA_PORT &= ~BIT6 & ~BIT7;
97       LCD_enable();
98       LCD_delay_ms(5);
99       LCD_enable();
100      LCD_delay_ms(1);
101      LCD_enable();
102      LCD_delay_ms(1);
103      LCD_DATA_PORT &= ~BIT4;              // D7-D4 = 0010
104      LCD_enable();

106      LCD_send_cmd(LCD_ON);        //  switch LCD on
107      LCD_send_cmd(LCD_CLR);         //  clear LCD
108      LCD_send_cmd(LCD_LINE1);       //  set cursor to x=0, y=0
109  }
```

```
112  /*------------------------------------------------------------------------------
113    Toggle LCD_E pin
114  ------------------------------------------------------------------------------*/
115  void LCD_enable(void)
116  /*------------------------------------------------------------------------------*/
117  {
118    LCD_DATA_PORT |= LCD_E_PIN;
119    _NOP();
120    _NOP();
121    LCD_DATA_PORT &= ~LCD_E_PIN;
122  }


125  /*------------------------------------------------------------------------------
126    Send command to LCD
127  ------------------------------------------------------------------------------*/
128  void LCD_send_cmd (unsigned char LCD_out_byte)
129  /*------------------------------------------------------------------------------*/
130  {
131    unsigned char LCD_tmp;

133    LCD_delay_ms(1);
134    LCD_tmp = LCD_out_byte & 0xf0;     // get upper nibble
135    LCD_DATA_PORT &= 0x0f;
136    LCD_DATA_PORT |= LCD_tmp;            // send CMD to LCD
137    LCD_DATA_PORT &= ~LCD_RS_PIN;    // sets LCD in COMMAND MODE
138    LCD_enable();
139    LCD_tmp = LCD_out_byte & 0x0f;
140    LCD_tmp = LCD_tmp << 4;              // get down nibble
141    LCD_DATA_PORT &= 0x0f;
142    LCD_DATA_PORT |= LCD_tmp;
143    LCD_DATA_PORT &= ~LCD_RS_PIN;    // sets LCD in COMMAND MODE
144    LCD_enable();
145  }


148  /*------------------------------------------------------------------------------
149    Send DATA to LCD
150  ------------------------------------------------------------------------------*/
151  void LCD_send_data (unsigned char LCD_out_byte)
152  /*------------------------------------------------------------------------------*/
153  {
154    unsigned char LCD_tmp;

156    LCD_delay_ms(1);
157    LCD_tmp = LCD_out_byte & 0xf0;     // get upper nibble
158    LCD_DATA_PORT &= 0x0f;
159    LCD_DATA_PORT |= LCD_tmp;            // send DATA to LCD
160    LCD_DATA_PORT |= LCD_RS_PIN;     // sets LCD in DATA MODE
161    LCD_enable();
162    LCD_tmp = LCD_out_byte & 0x0f;
163    LCD_tmp = LCD_tmp << 4;              // get lower nibble
164    LCD_DATA_PORT &= 0x0f;
165    LCD_DATA_PORT |= LCD_tmp;
166    LCD_DATA_PORT |= LCD_RS_PIN;     // sets LCD in DATA MODE
167    LCD_enable();
168  }

170  /*------------------------------------------------------------------------------
```

```
171     Send string to LCD
172   ------------------------------------------------------------------------------*/
173   void LCD_send_text(char *out_string)
174   /*------------------------------------------------------------------------------*/
175   {
176       while (*out_string)          // *out_string != '\0'
177       {
178           LCD_send_data(*out_string);    // send byte to LCD
179           out_string++;
180       }
181   }


184   /*-------------------------------------------------------------------------------
185     send date
186   ------------------------------------------------------------------------------*/
187   void LCD_send_time(void)
188   /*------------------------------------------------------------------------------*/
189   {
190     unsigned char time_char;

192     LCD_send_cmd(LCD_LINE2);
193     LCD_send_text("              >>");

195     LCD_send_cmd(LCD_RTC_WDAY);
196     LCD_send_text(RTC_days[RTC_values.wday]);

198     time_char = (RTC_values.hr / 10) + 48;
199     LCD_send_data(time_char);
200     time_char = (RTC_values.hr % 10) + 48;
201     LCD_send_data(time_char);
202     LCD_send_data(':');

204     time_char = (RTC_values.min / 10) + 48;
205     LCD_send_data(time_char);
206     time_char = (RTC_values.min % 10) + 48;
207     LCD_send_data(time_char);
208     LCD_send_data(':');

210     time_char = (RTC_values.sec / 10) + 48;
211     LCD_send_data(time_char);
212     time_char = (RTC_values.sec % 10) + 48;
213     LCD_send_data(time_char);
214   }


217   /*-------------------------------------------------------------------------------
218     send date
219   ------------------------------------------------------------------------------*/
220   void LCD_send_date(void)
221   /*------------------------------------------------------------------------------*/
222   {
223     unsigned char time_char;

225     LCD_send_cmd(LCD_LINE2);
226     LCD_send_text("                ");

228     LCD_send_cmd(LCD_RTC_DAY);
229     time_char = (RTC_values.day / 10) + 48;
230     LCD_send_data(time_char);
```

```
231   time_char = (RTC_values.day % 10) + 48;
232   LCD_send_data(time_char);

234   LCD_send_data('-');
235   LCD_send_text(RTC_months[RTC_values.month]);

237   LCD_send_data('-');
238   time_char = RTC_values.year / 1000;
239   LCD_send_data(time_char+48);
240   time_char = (RTC_values.year / 100) - ((RTC_values.year/1000) * 10);
241   LCD_send_data(time_char+48);
242   time_char = (RTC_values.year / 10) - ((RTC_values.year/100) * 10);
243   LCD_send_data(time_char+48);
244   time_char = RTC_values.year % 10;
245   LCD_send_data(time_char+48);
246 }


249 /*-----------------------------------------------------------------------------------
250   send sensor address + voltage modified by A.H. and S.P.
251 ---------------------------------------------------------------------------------*/
252 void LCD_send_sensor(unsigned char address_value, unsigned char voltage_value_high,
253             unsigned char voltage_value_low)
254 /*--------------------------------------------------------------------------------*/
255 {
256   unsigned char address_hundred, address_tenner, address_ones;
257   unsigned char voltage_ones, voltage_dezi, voltage_cent;
258   unsigned long voltage_work;
259   unsigned long voltage_complete;
260   voltage_work = voltage_value_high;
261   voltage_work = voltage_work << 8;
262   voltage_work |= voltage_value_low;

264   address_hundred = address_value / 100;
265   address_value %= 100;
266   address_tenner = address_value / 10;
267   address_value %= 10;
268   address_ones = address_value / 1;
269   voltage_work = voltage_work / 4.096;
270   voltage_complete = voltage_work;
271   if ((voltage_work % 10) < 5){
272       voltage_work = (voltage_work + 10);
273   }

275   voltage_ones = (unsigned char)(voltage_work / 1000);
276   voltage_work %= 1000;
277   voltage_dezi = (unsigned char)(voltage_work / 100);
278   voltage_work %= 100;
279   voltage_cent = (unsigned char)(voltage_work / 10);

281   LCD_send_cmd(LCD_LINE1);

283   LCD_send_text("Sens ");
284   LCD_send_data(address_hundred+48);
285   LCD_send_data(address_tenner+48);
286   LCD_send_data(address_ones+48);
287   LCD_send_text(" ");
288   if ((voltage_complete) < ((CELL_VOLT_MIN*10)+100))
289   {
290     if ((voltage_complete) > (CELL_VOLT_MIN*10))
```

```
291      {
292        LCD_send_text("lowvolt");
293      }
294      else
295      {
296        LCD_send_text("depth d");
297      }
298    }
299    else
300    {
301      LCD_send_data(voltage_ones+48);
302      LCD_send_text(",");
303      LCD_send_data(voltage_dezi+48);
304      LCD_send_data(voltage_cent+48);
305      LCD_send_text("V   ");
306    }

308  }
309  /*-------------------------------------------------------------------------------------
310    send sensor address/voltage from senosornmb by A.H.
311  -------------------------------------------------------------------------------------*/
312  void LCD_show_voltage(unsigned char sensornmb)
313  /*-------------------------------------------------------------------------------------*/
314  {
315    unsigned char value1_hundred, value1_tenner, value1_ones, value1_centi, value1_deci,
         value1_milli;
316    unsigned char value2_tenner, value2_ones, value2_deci;
317    unsigned char value2_cent, value2_mill;
318    unsigned char sensornmb_tenner, sensornmb_ones;
319    unsigned long voltage_work, address_work, sensornmb_work, voltage_sum;
320    static unsigned long voltage_old, sensornmb_old, AH_last_minute_old;
321    static unsigned long current_old;
322    unsigned long current_work;
323    unsigned char o, Battery_octa, U_CAP;

325    voltage_work = SENSOR_last_data[3][(sensornmb-1)]<<8;
326    voltage_work |= SENSOR_last_data[4][(sensornmb-1)];
327    if ((voltage_work != voltage_old)||(sensornmb!=sensornmb_old)||
328    (current_old!=CURRENT_last_data))
329    {

331      // show active sensor address + voltage
332      if ((sensornmb > 0) && (sensornmb <= SENSOR_active))
333      {
334        if ((sensornmb != sensornmb_old)||(voltage_work !=
335          voltage_old))
336        {
337          voltage_old = voltage_work;
338          address_work = SENSOR_address[sensornmb-1];
339          value1_hundred = address_work / 100;
340          address_work %= 100;
341          value1_tenner = address_work / 10;
342          address_work %= 10;
343          value1_ones = address_work / 1;
344          voltage_work = voltage_work / 4.096;
345          if ((voltage_work % 10) >= 5){
346            voltage_work += 5;
347          }
348          value2_ones = (unsigned char)(voltage_work / 1000);
349          voltage_work %= 1000;
```

```
350          value2_deci = (unsigned char)(voltage_work / 100);
351          voltage_work %= 100;
352          value2_cent = (unsigned char)(voltage_work / 10);

354          sensornmb_work = sensornmb;
355          sensornmb_tenner = sensornmb_work / 10;
356          sensornmb_work %= 10;
357          sensornmb_ones = sensornmb_work;
358          LCD_send_cmd(LCD_LINE1);
359          LCD_send_text("Sens");
360          LCD_send_data(sensornmb_tenner+48);
361          LCD_send_data(sensornmb_ones+48);
362          LCD_send_text("/");
363          LCD_send_data(value1_hundred+48);
364          LCD_send_data(value1_tenner+48);
365          LCD_send_data(value1_ones+48);
366          LCD_send_text(" ");
367          LCD_send_data(value2_ones+48);
368          LCD_send_text(",");
369          LCD_send_data(value2_deci+48);
370          LCD_send_data(value2_cent+48);
371          LCD_send_text("V");
372        }
373      }
374      // show overall voltage
375      else if (sensornmb > SENSOR_active)
376      {
377        if (sensornmb == SENSOR_active+1)
378        {
379          if ((sensornmb != sensornmb_old)||(voltage_work !=
380              voltage_old))
381          {
382            voltage_sum = 0;
383            voltage_old = voltage_work;

385            //summerize cell voltages
386            for (o=0; o<SENSOR_active; o++)
387            {
388              voltage_work = SENSOR_last_data[3][o]<<8;
389              voltage_work |= SENSOR_last_data[4][o];
390              voltage_sum += voltage_work;
391            }

393            //scale voltage value
394            voltage_sum = voltage_sum / 4.096;

396            //round up
397 //           if ((voltage_sum % 10) >= 5){
398 //             voltage_sum += 5;
399 //           }

401 //           voltage_sum %= 10000;
402            value2_tenner = (unsigned char)(voltage_sum / 10000);
403            voltage_sum %= 10000;
404            value2_ones = (unsigned char)(voltage_sum / 1000);
405            voltage_sum %= 1000;
406            value2_cent = (unsigned char)(voltage_sum / 100);
407            voltage_sum %= 100;
408            value2_deci = (unsigned char)(voltage_sum / 10);
409            voltage_sum %= 10;
```

```
410             value2_mill = (unsigned char)(voltage_sum);
411             LCD_send_cmd(LCD_LINE1);
412             LCD_send_text("U ges: ");
413             LCD_send_data(value2_tenner+48);
414             LCD_send_data(value2_ones+48);
415             LCD_send_text(",");
416             LCD_send_data(value2_cent+48);
417             LCD_send_data(value2_deci+48);
418             LCD_send_data(value2_mill+48);
419             LCD_send_text("V  ");
420           }
421         }
422         // show consumption last minute
423         else if (sensornmb == SENSOR_active+2)
424         {
425           if ((sensornmb != sensornmb_old)||
426           (AH_last_minute != AH_last_minute_old))
427           {
428             current_old = CURRENT_last_data;
429             AH_last_minute_old = AH_last_minute;
430             LCD_send_cmd(LCD_LINE1);
431             LCD_send_text("Co: ");
432             if (AH_last_minute_IN_OUT == 1)
433             {
434               current_work = AH_last_minute;
435               LCD_send_text(" ");
436             }
437             else
438             {
439               current_work = AH_last_minute;
440               LCD_send_text("-");
441             }
442             /*if (current_work%10>=5)
443             {
444               current_work += 5;
445             }*/
446             value1_hundred = current_work / 100000;
447             current_work %= 100000;
448             value1_tenner = current_work / 10000;
449             current_work %= 10000;
450             value1_ones = current_work / 1000;
451             current_work %= 1000;
452             value1_centi = current_work / 100;
453             current_work %= 100;
454             value1_deci = current_work / 10;
455             current_work %= 10;
456             value1_milli = current_work;

458             LCD_send_data(value1_hundred+48);
459             LCD_send_data(value1_tenner+48);
460             LCD_send_data(value1_ones+48);
461             LCD_send_text(",");
462             LCD_send_data(value1_centi+48);
463 /*            LCD_send_data(value1_deci+48);
464             LCD_send_data(value1_milli+48)*/;
465             LCD_send_text("Ah/min");
466           }
467         }
468         // show actual batterie current
469         else if (sensornmb == SENSOR_active+3)
```

```
470        {
471          if ((sensornmb != sensornmb_old)||
472          (current_old != CURRENT_last_data))
473          {
474            current_old = CURRENT_last_data;
475            LCD_send_cmd(LCD_LINE1);
476            LCD_send_text(" Ibat :");
477            //charge --> negative
478            if (CURRENT_IN_OUT)
479              LCD_send_text("-");
480            //discharge --> positive
481            else
482              LCD_send_text(" ");

484            //calc actual current value dependent of sensor channel
485            if(CURRENT_channel==1)  //use channel 1
486              current_work = (unsigned long)(CURRENT_last_data*45.7);
487            else  //use channel 2
488              current_work = (unsigned long)(CURRENT_last_data*305.3);

490            //round up
491            /*if (current_work%10>=5)
492              current_work += 5;*/

494            value1_hundred = current_work / 100000;
495            current_work %= 100000;
496            value1_tenner = current_work / 10000;
497            current_work %= 10000;
498            value1_ones = current_work / 1000;
499            current_work %= 1000;
500            value1_centi = current_work / 100;
501            current_work %= 100;
502            value1_deci = current_work / 10;
503            current_work %= 10;
504            value1_milli = current_work;

506            LCD_send_data(value1_hundred+48);
507            LCD_send_data(value1_tenner+48);
508            LCD_send_data(value1_ones+48);
509            LCD_send_text(",");
510            LCD_send_data(value1_centi+48);
511            LCD_send_data(value1_deci+48);
512            LCD_send_data(value1_milli+48);
513            LCD_send_text("A  ");
514          }
515        }
516        // show batterie capacity
517        else if (sensornmb == SENSOR_active+4)
518        {
519          if ((sensornmb != sensornmb_old)||
520            (AH_last_minute_old != AH_last_minute))
521          {
522            LCD_send_cmd(LCD_LINE1);
523            LCD_send_text(" Cbat: ");
524            AH_last_minute_old = AH_last_minute;
525            current_work = AH_MASTER*44;
526            LCD_send_text(" ");

528            AH_last_minute_old = current_work;
529            if (current_work%10>=5)
```

```
530         {
531           current_work += 5;
532         }
533         value1_hundred = current_work / 10000;
534         current_work %= 10000;
535         value1_tenner = current_work / 1000;
536         current_work %= 1000;
537         value1_ones = current_work / 100;
538         current_work %= 100;
539         value1_deci = current_work / 10;
540         LCD_send_data(value1_hundred+48);
541         LCD_send_data(value1_tenner+48);
542         LCD_send_data(value1_ones+48);
543         LCD_send_text(",");
544         LCD_send_data(value1_deci+48);
545         LCD_send_text("Ah ");
546       }
547     }
548     // show batterie capacity segment
549     else if (sensornmb == SENSOR_active+5)
550     {
551       if ((sensornmb != sensornmb_old)||
552       (AH_last_minute_old != AH_last_minute))
553       {
554         AH_last_minute_old = AH_last_minute;
555         Battery_octa =
556         (AH_MASTER*0.44/BATT_CAPACITY)*9;
557         LCD_send_cmd(LCD_LINE1);
558         LCD_send_text(" Bat :");
559         LCD_send_data(91);
560         if (Battery_octa==0)
561         {
562           LCD_send_text(" !LOW!  ");
563         }
564         else
565         {
566           for(o=0; o<Battery_octa && o<8; o++)
567           {
568             LCD_send_data(1);
569           }

571           for(o=Battery_octa; o<8; o++)
572           {
573             LCD_send_text(" ");
574           }
575         }
576         LCD_send_data(93);
577       }

579     }

581     // show battery capacity voltage based in percent
582     else
583     {
584       if ((sensornmb != sensornmb_old)||
585       (AH_last_minute_old != AH_last_minute))
586       {
587         AH_last_minute_old = AH_last_minute;
588         LCD_send_cmd(LCD_LINE1);
589         U_CAP = U_MASTER_CAPACITY;
```

```
590            value1_hundred = U_CAP / 100;
591            U_CAP %= 100;
592            value1_tenner = U_CAP / 10;
593            U_CAP %= 10;
594            value1_ones = U_CAP / 1;
595            LCD_send_text(" Cap_U:");
596            LCD_send_data(value1_hundred+48);
597            LCD_send_data(value1_tenner+48);
598            LCD_send_data(value1_ones+48);
599            LCD_send_text("% ");
600            LCD_send_text("    ");
601          }
602        }
603      }
604      // battery meter percent
605      else
606      {
607        if ((sensornmb != sensornmb_old)||(AH_last_minute_old !=
608          AH_last_minute))
609        {
610          AH_last_minute_old = AH_last_minute;
611          Battery_octa = (AH_MASTER*0.44/BATT_CAPACITY/1)*100;
612          value1_hundred = Battery_octa/100;
613          Battery_octa %= 100;
614          value1_tenner = Battery_octa/10;
615          Battery_octa %= 10;
616          value1_ones = Battery_octa/1;
617          LCD_send_cmd(LCD_LINE1);
618          LCD_send_text(" Bat%  ");
619          if (value1_hundred == 1)
620          {
621            LCD_send_text("100");
622          }
623          else
624          {
625            LCD_send_data(value1_hundred+48);
626            LCD_send_data(value1_tenner+48);
627            LCD_send_data(value1_ones+48);
628          }
629          LCD_send_text("%");
630          LCD_send_text("     ");
631        }
632      }
633      sensornmb_old = sensornmb;
634    }
635  }
```

B. Matlab Quellcode

B.1. Framefehler Auswertung

Listing 13: M-File zum Auslesen der Informationen über die Framefehler aus dem Datenlogger

```
%this derivate of ferread reads the output of the uC's memory
clear all
close all;
%init number of steps to be recorded
steps = 1001;
%init error counter
errorcount = 0;
%% initialize com port and start recording
com_port = sprintf('COM%s', input('Enter COM-PORT:','s'));
% open COM-PORT with 115200 baud and return as terminator
serial_port = serial(com_port,'BaudRate',115200,'Terminator','CR');
fopen(serial_port);
set(serial_port,'Timeout',10);
%send data request
fprintf(serial_port, 'start');  %start a measurement
pause(5);
fprintf(serial_port, 'fer');    %start frame error output
%% read data
%create waitbar
h = waitbar(0,'1','Name','getting FER data...');

i = 1;
while i < steps
    %try

        [data,count,msg] = binblockread(serial_port,'uint8');
        if ~isempty(msg)
            break
        end
        data2(i,:) = data';
    %if an error occurs
    %catch me
     %errorcount = errorcount+1;
     %disp(['Error in step: ' num2str(i)]);
    %end
    %increase waitcount od waitbar
    message = sprintf('%i frames received',i);
    waitbar(i/steps,h, message);
    i = i+1;
end
%% close connection
fprintf(serial_port, 'fer');   %stop FER output
pause(1);
fprintf(serial_port, 'stop');   %stop measurement
fclose(serial_port);
delete(serial_port);
clear serial_port;
%save data matrix
save('feranalyse-6sensoren.mat','data2')
```

Listing 14: M-File zur Analyse und Darstellung der Information über die Framefehler

```
 %fer analysis of basen receiver
%******************************
%this function is supposed to present the frame error stuff in a "nice to
%look at plot". The input is the recorded data from batmon_fer function.
%
%The recorded data is formatted as follows:
%##########################################################################
```

```
8   %Sensor Address||BitTooShort||BitTooLong||WrongRunIn||LostFrames||CrcError
9   %##########################################################################
10  %
11  %The data batmon_fer gives back has no time reference hence this function
12  %tries to create an approximate one. This is given from the minimum frame
13  %rate of a transmission. Despite this time reference is not very accurate
14  %it at least gives an impression of transmission delays caused of
15  %frameerrors. Additionally having an adopted time reference of fixed
16  %intervals is an easier method than saving timer values for now.

18  %Attention: This script expects the Sensor Addresses 31-36, 6 Sensors and
19  %1000 row input! Keep in mind when editing!

21  %convert input data to double
22  %data2 = double(data2);
23  %determine length of input
24  datalength = length(data2);
25  %used sensor addresses
26  SENSOR_ADDRESS = 31:1:36;

28  %% create general histogram for input data
29  %count sensor reception rate
30  receive_count = histc(data2(:,1),SENSOR_ADDRESS);

32  %% sort screen data sensor by sensor, blank out
33  %preallocate array for sorted sensors
34  sensor = cell(1,6);
35  sensor_screened = cell(1,6);
36  sensor_lost = zeros(datalength,6);

38  sensor_error = zeros(5,6);

40  %search data for each sensor and blank out others
41  for i = 1:6
42      sensor_screened(i) = { double(data2(:,1) == SENSOR_ADDRESS(i)) };
43      sensor(i) = { repmat(sensor_screened{i},1,6).*data2 };
44      sensor_lost(:,i) = sensor{i}(:,5);

46      for j = 1:5
47          sensor_error(j,i) = sum(sensor{i}(:,j+1));
48      end
49  end
50  %% create the time reference for the input data.
51  %It is assumed that the fixed transmission time interval is 500ms for
52  %eatch sensor. The maximum duration of the fer recording varies according
53  %to actual the amount of frame errors. For that the amount of transmissions
54  %of the last received sensor must be taken care of.

56  %get received data rate
57  sensor_rate = sum(sensor_lost)+receive_count';

59  %determine the maximum rate
60  [max_rate I] = max(sensor_rate);

62  %preallocate timing matrix, first column is the time reference, the rest
63  %will be for sensor timing
64  sensor_timed = zeros(max_rate+1,7);
65  sensor_timed(:,1) = 0:5:5*max_rate;

67  %calculate sensor timings
```

```
68  %preallocate sensor timing cache
69  timing_cache = zeros(1,6);

71  %loop for element index of sorted input data
72  for i = 1:datalength

74  %determine cache index
75  index_cur = data2(i,1)-30;

77  %put sensor timing flag in right position
78  timing_cache(index_cur) = data2(i,5)+timing_cache(index_cur)+1;
79  sensor_timed(timing_cache(index_cur),index_cur+1) = 1;

81  end
82  %% plot the timing results
83  %get screen size
84  scrsz = get(0,'ScreenSize');
85  %plot stuff
86  figure('Name','sensor timing','Position',...
87      [1 scrsz(4)/2-75 scrsz(3)/2 scrsz(4)/2]);
88  bar(sensor_timed(:,1),sensor_timed(:,2:7),0.5,'stacked');
89  axis([min(sensor_timed(:,1)) max(sensor_timed(:,1)) 0 6.5]);
90  h = legend('31','32','33','34','35','36','orientation','vertical',...
91      'location','NorthEastOutside');
92  %legend title
93  v = get(h,'title');
94  set(v,'string','Sensor Address');
95  grid minor;
96  %label axis
97  xlabel('time in $10^{-1}$s','Interpreter','LaTex','fontsize',10);
98  ylabel('Number of Sensors received simultaneously ','fontsize',8);
99  %title
100 title('Sensor Timing','fontsize',14);

102 %% create histogram for eatch sensor
103 % sort input data by sensor addess row wise
104 figure('Name','Sensor Histogram','Position',...
105     [scrsz(3)/2+10 scrsz(4)/2-75 scrsz(3)/2-50 scrsz(4)/2]);
106 %calc relative sensor rate
107 receive_rel = receive_count./sensor_rate';
108 %plot sensor rate + overall
109 receive_all_rel = [sum(receive_count)/sum(sensor_rate);zeros(6,1)];
110 SENSOR_ADDRESS_all = [30 SENSOR_ADDRESS];
111 bar(SENSOR_ADDRESS, receive_rel);
112 hold on;
113 bar(SENSOR_ADDRESS_all,receive_all_rel,'r');
114 %label axis
115 xlabel('Sensor Address','fontsize',10);
116 ylabel('Frame Rate','fontsize',10);
117 %title
118 title('received Sensor Rates','fontsize',14);
119 %legend
120 legend('individual Sensor','overall');
121 %grid
122 grid minor
123 hold off;
124 %relative sensor errors
125 mask = ones(4,1)*sensor_error(4,:);
126 mask(3,:) = sensor_rate;
127 %treat RunIn separately
```

```
wrongRunIn = sensor_error(3,:);
sensor_error = [sensor_error(1:2,:);sensor_error(4:5,:)];
sensor_error_rel = sensor_error./mask;
figure('Name','Sensor Statistic','Position',...
    [scrsz(3)/2+10 0 scrsz(3)/2-50 scrsz(4)/2]);
bar(SENSOR_ADDRESS,sensor_error_rel');
legend('too short','too long','lost frames','CRC fail');
grid minor
xlabel('Sensor Address','fontsize',10);
ylabel('error quota of all frames sent','fontsize',10);
receive_quota = sum(receive_count)/(sum(wrongRunIn)+sum(receive_count));
disp(['amount of successful received frames out of number of tries: ',...
num2str(receive_quota)]);
average_tries = sum(wrongRunIn)/sum(receive_count);
disp(['The receiver needs ',num2str(average_tries),' tries on average',...
    ' to receive a frame successfully']);
```

B.2. Auswertung zu den Messungen an der Starterbatterie

Listing 15: M-File zur Analyse und Darstellung der Information über die Framefehler

```
%cellanalysis_csv.m - is for interpreting the scopedata recorded for the
%cellmeasurement of the SLI battery.
%input: scopedata *.csv (tektronix)
%output: plots, cell voltages

%description: As input 2 csv files are necessary. These have to be formated
%with textronix standard where the data is placed column by column
%channelwise. The first file contains the first few cells, the other one
%the rest. Both files must be named the same except of a last
%symbol/number attached to the second filename to work properly.

%% get file path and import data
[filename, pathname, filterindex] = uigetfile( ...
{'*.csv','test files (*.csv)'}, ...
'Pick a datafile from a battery meassurement');
if isequal(filename,0) || isequal(pathname,0)
    disp('Canceled by user in fileselect box');
    return
else
    filename_compl = fullfile(pathname, filename);
    %list related files
    list_csv = dir([filename_compl(1:end-4) '*.csv']);
    if length(list_csv) < 2
        disp(' ');
        disp('Abort! There is no data for the 2. scope');
        disp(' ');
        return
    else
        disp(' ');
        disp('User has selected the files:');
        disp(' ');
        disp(list_csv(1).name);
        disp(list_csv(2).name);
        disp(' ');
    end
```

```
36 end
37 %data import
38 dataa = importdata(fullfile(pathname,list_csv(1).name),',',15);
39 datab = importdata(fullfile(pathname,list_csv(2).name),',',15);
40 %merging data (first column is time)
41 data = [dataa.data datab.data(:,2:end)];
42 %% calculate cell voltages
43 %scale Cell 5-6 (voltage divider)
44 data(:,6:7) = data(:,6:7)*2;
45 data_temp = data;
46 %cell voltages
47 [data_rlength data_clength] = size(data);
48 i=1;
49 while i < 6
50     data(:,i+2) = data_temp(:,i+2)-data_temp(:,i+1);
51     i=i+1;
52 end
53 %% plot data and label graphs
54 h = figure('Name',fullfile(pathname, filename));
55 % subplot(2,1,1);
56 plot(data(:,1),data(:,2:7))
57 legend('cell 1', 'cell 2', 'cell 3', 'cell 4', 'cell 5', 'cell 6')
58 ylabel('Volt')
59 xlabel('seconds')
60 grid minor;
61 set(h,'PaperUnits','normalized');
62 set(h,'PaperPosition', [0 0 1 1]);
63 set(h,'PaperOrientation','landscape');
64 % request title
65 titlename = input('Please give me a title name: ','s');
66 % format title
67 title(titlename,'fontsize',12,'fontweight','bold');
68 % output
69 choice = input('save as PDF? (if not leave blank) ','s');
70 if ~strcmp(choice,'')
71     file_pdf= [choice '.pdf'];
72     saveas(h, file_pdf);
73     close(h);
74     winopen(file_pdf);
75 end
76 % subplot(2,1,2);
77 % plot(data(:,1),data(:,8));
78 % ylabel('Amps')
79 % xlabel('seconds')
80 % grid minor;
```

Listing 16: M-File für die Darstellung des Datenformats des Oszilloskops

```
1  %Kurvenauswertung der Strommessung am Auto für Textronic Oszilloskop
2  %Input: csv Datei mit Messdaten aus Oszilloskop
3  %Output: Plot der Messdaten

5  %open data file
6  fid = fopen('tek0007.csv');
7  %fid = fopen('test.csv');

9  %header = textscan(fid,'%s','delimiter','\t');

11 %scan data file for header
12 header = textscan(fid, '%s',14, 'delimiter','\t');
```

```
%scan data file for data
data = textscan(fid, '%f,%f');

%close data file
fclose(fid);

%convert array to matrix
dataA = [data{1,:}];

%find start value
i=1;
while i<length(dataA) && (abs(dataA(i+1,2)-dataA(i,2)))<2
    i=i+1;
end

%shift signal to new origin
dataNewX = [(dataA(:,1)-dataA(i,1)), dataA(:,2)];

%plot data
plot(dataNewX(:,1),dataNewX(:,2));
grid;
axis([-1e-4 max(dataNewX(:,1)) min(dataNewX(:,2)) (max(dataNewX(:,2))+50)])
title('Anlassstrom VW Passat Diesel','fontsize',14);
xlabel('sec','fontsize',24);
ylabel('A','fontsize',24);
```

B.3. Auswertung und Analyse der Sensordaten

Listing 17: M-File für die Auswertung und Darstellung der Sensordaten

```
%file: batsen_plot_ms_0806.m
%
%author: Matthias Schneider, Simon Püttjer
%last modification: 08.06.2011
%
%description: This script file rearranges and displays the recorded data, the sensor
%measured during a battery test. As input a *.mat file is needed or data
%which is formated as follows:
% 6 columns:
% ||versionID||AddressWord||SenAdd||CellVoltage||timelabel||Pos.inQueue||
% measurements row-wise

%% clear workspace
clear all

%% some adjustment factors
%time_offset = [-18.8 -12 -14.5 -14.5 -15.5 -16];
time_offset = [0 0 0 0 0 0];
time_cor_fac = [1 1 1 1 1 1];
voltage_cor_fac = [1 1 1 1 1 1];
frame_length = 0.024;

%% initialize input data
display('Pick a *.mat file!');
% get file path and import data
```

```
26 [filename, pathname, filterindex] = uigetfile( ...
27 {'*.mat','matlab mat file (*.mat)'; ...
28 '*.*', 'All Files (*.*)'}, ...
29 'Pick a datafile from a battery meassurement');
30 if isequal(filename,0) || isequal(pathname,0)
31     disp('Canceled by user in fileselect box');
32     return
33 else
34     filename_compl = fullfile(pathname, filename);
35     disp(' ');
36     disp('User has selected the file:');
37     disp(filename_compl);
38     disp(' ');
39 end
40 % load file path as matrix file
41 input = load(filename_compl,'-mat');
42 input = input.data3;
43 input1 = double(input); %new input
44 input2 = input1;


47 scope_data = load('C:\Users\simon\Documents\MATLAB\Batteriemessung_1_6_2011_sorted_data\
       A4\Messung1\A4_M1_arb_gen2.mat','-mat');
48 scope_data = scope_data.data;

50 %% find and sort the data of each sensor into a seperate vector inside an array

52 % find the sensor adresses
53 sen_ids = [input2(1,3)] %init id list with adresses from first packet
54 id_already_known = 0;
55 for i=2:1:size(input2,1) %loop for all packets
56     id_already_known = 0; %reset flag for known ids
57     for k=1:1:size(sen_ids,2) %loop for actual id list
58         if input2(i,3) == sen_ids(k) %check if id in packet matches id in list
59             id_already_known = 1; %set known id flag
60             break;
61         end
62     end
63     if id_already_known == 0
64         sen_ids(size(sen_ids,2)+1) = input2(i,3); %place the unknown id at the end of id
               list
65     end
66 end
67 sorted_sen_ids = sort(sen_ids) %sort the sensor ids

69 %%seperate data of each sensor into an array of vectors
70 for i = 1:1:size(sorted_sen_ids,2) % loop for the sensor ids
71     j = 1; %counter variable for vector element
72     for k = 1:1:size(input2,1) %loop for the packets to be sorted
73         if input2(k,3) == sorted_sen_ids(i) % check if packet sensor matches actual
              sensor id
74         sep_data{i}(j,:) = input2(k,:); %write packet into vector of actual id
75         j=j+1; %increase vector element counter
76         end
77     end
78 end

80 sep_data_sorted = sep_data;

82 %% transform voltage values
```

```
83  for i = 1:1:size(sep_data,2) % loop for the sensor ids
84      start_ts = sep_data{i}(1,5);
85      start_tsb = (sep_data{i}(1,8)*2^16+sep_data{i}(1,9))/1e6;%-(5000-sep_data{i}(1,2))
            /10000
86      for k = 1:1:size(sep_data{i},1)
87          sep_data{i}(k,4) = sep_data{i}(k,4)*1.5*(2.5/1023)/10*voltage_cor_fac(i);
88          sep_data{i}(k,5) = (sep_data{i}(k,5) - start_ts);
89          if sep_data{i}(k,5) < 0
90              sep_data{i}(k,5) = sep_data{i}(k,5)+2^16;
91          end
92          sep_data{i}(k,2) = (sep_data{i}(k,2))/10000;
93          sep_data{i}(k,5) = ((sep_data{i}(k,5)/100));
94          sep_data{i}(k,8) = (sep_data{i}(k,8)*2^16+sep_data{i}(k,9))/1e6-start_tsb;%-(sum
                (sep_data{i}(1:k,2)));
95          %if sep_data{i}(k,1) == 1
96          %    sep_data{i}(k,5) = sep_data{i}(k,5) + frame_length;
97          %end
98      end
99      sep_data{i}(1,8) = 0;
100     m = 1;
101     for k = 2:1:size(sep_data{i},1)
102         if (sep_data{i}(k,1) == 3) && (sep_data{i}(k-1,1) == 3);
103 %            delta_time_lb(m) = sep_data{i}(k,8)-(sep_data{i}(k-1,8)-sep_data{i}(k-1,2))
        ;
104 %            adj(m) = delta_time_lb(m)/(sep_data{i}(k,5)-(sep_data{i}(k-1,5)));
105             delta_time_lb(m) = sep_data{i}(k,5)-sep_data{i}(k-1,5) +(sep_data{i}(k,2)-
                    sep_data{i}(k-1,2));
106             sep_data{i}(k,10) = delta_time_lb(m);
107             sep_data{i}(k,11) = (sep_data{i}(k,8)-(sep_data{i}(k-1,8)));
108             adj{i}(m) = delta_time_lb(m)/(sep_data{i}(k,8)-(sep_data{i}(k-1,8)));
109             madj(i) = mean(adj{i}(:));
110             sep_data{i}(k,12) = adj{i}(m);

112             m = m+1;
113         end
114     end

116     for k = 1:1:size(sep_data{i},1)
117         sep_data{i}(k,13) = (sep_data{i}(k,5)*1/madj(i));
118     end

120     voffset(i) = abs(sep_data{i}(1,4)-scope_data(1,2));
121     [min_u(i) min_ts(i)] = min(sep_data{i}(:,4));
122     [min_u_scope(i) min_ts_scope(i)] = min(scope_data(:,2));
123     time_offset(i) = sep_data{i}(min_ts(i),13)-scope_data(min_ts_scope(i),1)
124     for k = 1:1:size(sep_data{i},1)
125         sep_data{i}(k,13) = (sep_data{i}(k,13)-time_offset(i));
126         sep_data{i}(k,4)  =  sep_data{i}(k,4)- voffset(i);

128     end
129 end



133 %% plot data
134 figure(1);
135 plot(scope_data(:,1),scope_data(:,2));
136 cstring='rgbcmyk'; % color string
137 hold on
138 for i=1:1:length(sep_data)
```

```
139 %     figure(i)
140 %     hold
141     %plot(sep_data{i}(:,5),sep_data{i}(:,4),'k');
142     %plot(sep_data{i}(:,8),sep_data{i}(:,4));
143     %plot(scope_data(:,1),scope_data(:,3));
144     plot(sep_data{i}(:,13),sep_data{i}(:,4),cstring(mod(i,7)));
145 end
```

Listing 18: M-File für die Simulation des Zwischenspeichers

```
1  clear all; close all
2  % Step 1: Load file and define the raw oversampling_factor
3  % all scope samplings are reduced to 1ms sampling
4  % Select by delete comments of only two lines

6  %input_data=load('vito_4s.mat');
7  %oversampling_factor_raw=250;  % raw duration 4s / 1 Mio => 4us ->  1ms

9  %input_data=load('vito_10s.mat');
10 %oversampling_factor_raw=100;  % raw duration 10s / 1 Mio => 10us ->  reduced 1ms

12 %input_data=load('vito_20s.mat');
13 %oversampling_factor_raw=50;  % raw duration 20s / 1 Mio => 20us ->  reduced 1ms

15 input_data=load('vito_40s.mat');
16 oversampling_factor_raw=25;  % raw duration 40s / 1 Mio => 40us ->  reduced 1ms


19 % Step 2 select one of six sensors 1 to 6 !
20 %  %%%%%%% Select Sensor %%%%%%%


23 sensor_number=2;

25 input_data = input_data.data_scaled;
26 % Cell Voltage raw Cv_raw
27 Cv_raw=input_data(:,1+sensor_number);

29 % Step 3 Downsample to 1ms Cell_voltage_decimated CVD
30 % Cvd sampling scaled in every case reduced to 1ms
31 Cvd=downsample(Cv_raw,oversampling_factor_raw);


34 % by the way ... this is a problem ... don't care  ... Cv1=decimate(Cell1_voltage_raw,
       oversampling_faktor_raw);

36 % Step 4 Average or filter a Block
37 %Cvd=filter(Rectwin(8),sum(Rectwin(8)), Cvd);  has problem with the
38 % groupdelay of the first values ... minor bad effects in plot scaling
39 Cvd=filtfilt(rectwin(8),sum(Rectwin(8)), Cvd);

41 % Alternatives ...
42 Cvd=filtfilt(gausswin(13),sum(gausswin(13)), Cvd);
43 % Step 5 reduce to 8 ms
44 % Cv=downsample(Cvd,10);
45 Cv = Cvd;
46 % find the lengt of reduced data ... not fix !
47 maxCv=length(Cv);
48 %  %%%%%%% Here you change the starting index; default =1  %%%%%%%
49 Startindex=1;
```

```
50  Cv=Cv(Startindex:maxCv);
51  maxCv=length(Cv);


54  %  %%%%%%% Now Sample level preprocessing  completed %%%%%%%



58  % The Parameters of Input Queue xx_iq and Output of Queue xx _oq
59  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% important parameter settings here in relation to
         8ms !!  %%%%%%%%%%%%%%%%%%%%
60  stretch_in=2; % here is to modify the denisity of transmitting
61  stretch_out=1;
62  max_time_distance_iq =round(1000*stretch_in); % 1.024s *stretch  ... a max value ...
        could shorter
63  time_distance_oq=round(500*stretch_out); % 512ms *stretch a fix integer value (later
        with articifal random offset)
64  var_time = round(time_distance_oq / 3);
65  fix_time = time_distance_oq;
66  time_distance_oq=randi([-var_time var_time],1,1) + fix_time;
67  % The three threshold as conditions  fast In-queue !!!
68  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% important parameter settings
        %%%%%%%%%%%%%%%%%%%%
69  value_diff_threshold=0.01; % the max value difference from last In-queue allowed without
         an new In-queue
70  sum_value_diff_threshold=0.3; % the max sum of differences from last In-queue allowed
        without an new In-queue
71  value_diff_multiply_time_diff_threshold=0.5; % the max value difference*time difference
        from last In-queue allowed without an new In-queue

73  % Variables for Plots later

75  % For observation of quentries
76  qv=zeros(100000,1);  % queue entries value  ... here all values stored NOT really a
        queue
77  qts=zeros(100000,1); % queue entries time stamp  (in_queue time stamp) ...  here all
        values stored NOT really a queue

79  % For observation of queue behaviour only
80  loq=zeros(100000,1); % current length of queue (loq) in the moment of inqueue or
        outqueue
81  tsq=zeros(100000,1); % times stamps in the moment of inqueue or outqueue

83  % Indizees and initialisation
84  i=1; % run index of last reduced sampled (8ms) value
85  j=1;  % index of recent in-queue value array  ...  contains at the end the number of all
         inqueue values
86  k=0;  % index of recent queue change (both in- or outqueue)-value ... contains at the
        number of all queue operations
87  ko=0; % counter of out queues
88  l=0;  % current lenght of queue
89  l_max=0;  % max length of queue

91  % Only for statistics if later coming conditions are true or false
92  c1counter=0; c2counter=0; c3counter=0; c4counter=0;

94  % some variables to initalisize
95  qv(1)=Cv(2);
96  qts(j)=0;
97  sumdiff=0;
```

```
99  while (i<maxCv),
100     % grad=abs((Cv1(i)-rv(j))/(i-j)) not in use ... first idea

102     diff=abs((Cv(i)-qv(j)));  % difference to last inqueue value
103     sumdiff=sumdiff+diff;     % sum of all difference to last inqueue value since last
            inque
104     time_diff=i-qts(j);        % difference to last inqueue value

106     i=i+1;

108     % One of this criteria to select  of more in OR combination

110     % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% important method modifikations
            %%%%%%%%%%%%%%%%%%%%

112     cond1 =    (abs(diff)>value_diff_threshold) ;% needed differences reached
113     cond2 =    (abs((diff*time_diff))>value_diff_multiply_time_diff_threshold); % needed
            differences * times reached
114     cond3 =    (abs(sumdiff)>sum_value_diff_threshold);  % intergrated values reaches
            threshold
115     cond4 =    (time_diff>max_time_distance_iq);  % needed differences reached

117     % only statitics counting of true or false
118     if (cond1);c1counter=c1counter+1; end
119     if (cond2); c2counter=c2counter+1; end
120     if (cond3); c3counter=c3counter+1; end
121     if (cond4); c4counter=c4counter+1; end


124     if (cond1 || cond4)
125     % recuce at the end  if effective
126     % p.e.  if (cond3 | cond4)
127     % OLDCODE if  ( sumdiff>sum_value_diff_threshold) |  ( time_diff>
            max_time_distance_iq)  % needed differences reached

129          % Storing a value = inqueue
130          j=j+1;  % index in storing array for observation
131          qv(j)=Cv(i); % Voltage value
132          qts(j)=i; % time stamp
133          % only if sumdiff is in use
134          sumdiff=0;

136          % Only for que observation
137          l=l+1; % increase current queue length
138          k=k+1; % increase array use for queue observation
139          loq(k)=l; % store current length
140          tsq(k)=i; % store time stamp in 8ms units

142     end

144     if(mod(i,10)==0)

146     end


149     % out of queue period reached
150     if (0==mod(i,time_distance_oq)) % longest defined time period to transmit
151         if (l>l_max)    % collect the longest queue for obsverations
152             l_max=l;
```

```
153         end
154         if (l>1)       %  if the queue longer then 1 then reduce it
155             l=l-1;     % a queue of one is never reduce to empty
156         end
157         ko=ko+1;
158         k=k+1; % increase array use for queue observation
159         loq(k)=l; % store current length
160         tsq(k)=i; % store time stamp in 8ms units

162     end
163    time_distance_oq=randi([-var_time var_time],1,1) + fix_time;
164 end;


167 % Olotting routines
168 figure(1);
169 title('Non equistand sample reducing');
170 subplot(2,1,1);
171 plot(1:maxCv,Cv,'r',qts(1:j),qv(1:j),'-xb',0,0,'w',0,0,'w',0,0,'w',0,0,'w');
172  rs_string=strcat('8ms reduced signal samples:  ',num2str(maxCv));
173  iq_string=strcat('In-queue Values total:  ',num2str(j),' multiple cases poss.');
174  t1q_string=strcat('Max diff to In-queue (threshold):  ',num2str(value_diff_threshold),
         ' cases:',  num2str(c1counter));
175  t2q_string=strcat('Max diff*time to In-queue (threshold):  ',num2str(
         value_diff_multiply_time_diff_threshold), ' cases:',  num2str(c2counter));
176  t3q_string=strcat('Max sum diff to In-queue (threshold):  ',num2str(
         sum_value_diff_threshold), ' cases:',  num2str(c3counter));
177  t4q_string=strcat('Max time to In-queue:  ',num2str(max_time_distance_iq),'* 8ms
         cases:',  num2str(c4counter) );


180 legend(rs_string,  iq_string,t1q_string,t2q_string,t3q_string,t4q_string);
181 subplot(2,1,2);
182 plot(tsq(1:k),loq(1:k),'-xg',0,0,'w',0,0,'w');
183 l_string=strcat('Queuefilling max: ',num2str(l_max));
184 oq_string=strcat('Out-queue Values total:  ',num2str(ko));
185 t1q_string=strcat('Time to Out-queue:  ',num2str(time_distance_oq),'* 8ms');

187 legend(l_string,oq_string, t1q_string);

189 % Example of ohter plots

191 figure(2);
192 [AX,H1,H2]=plotyy(qts(1:j),qv(1:j), tsq(1:k),loq(1:k));

194 set(get(AX(1),'Ylabel'),'String','Voltage signal value')
195 set(get(AX(2),'Ylabel'),'String','Length of queue')
196 xlabel('Time index (in 8ms)')
197 title('Queue-filling')
198 set(H1,'LineStyle','-')
199 set(H1,'Marker','x')
200 set(H2,'LineStyle','-')
201 set(H2,'Marker','x')
```

C. Sonstiger Anhang

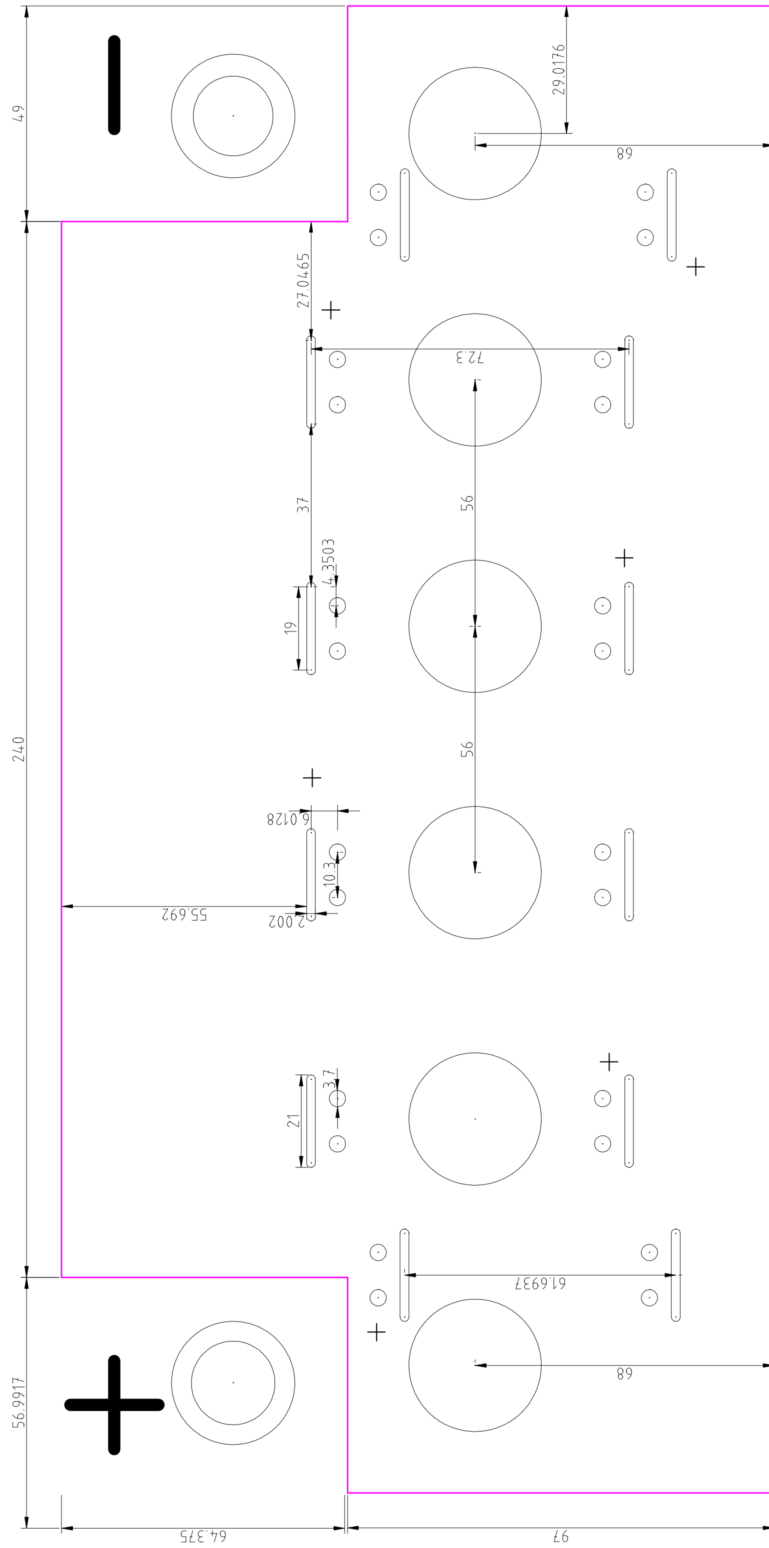


Abbildung 2.: Konstruktionszeichnung des Batterie-Deckels der Starterbatterie von Banner (Verhältnis 1:2)

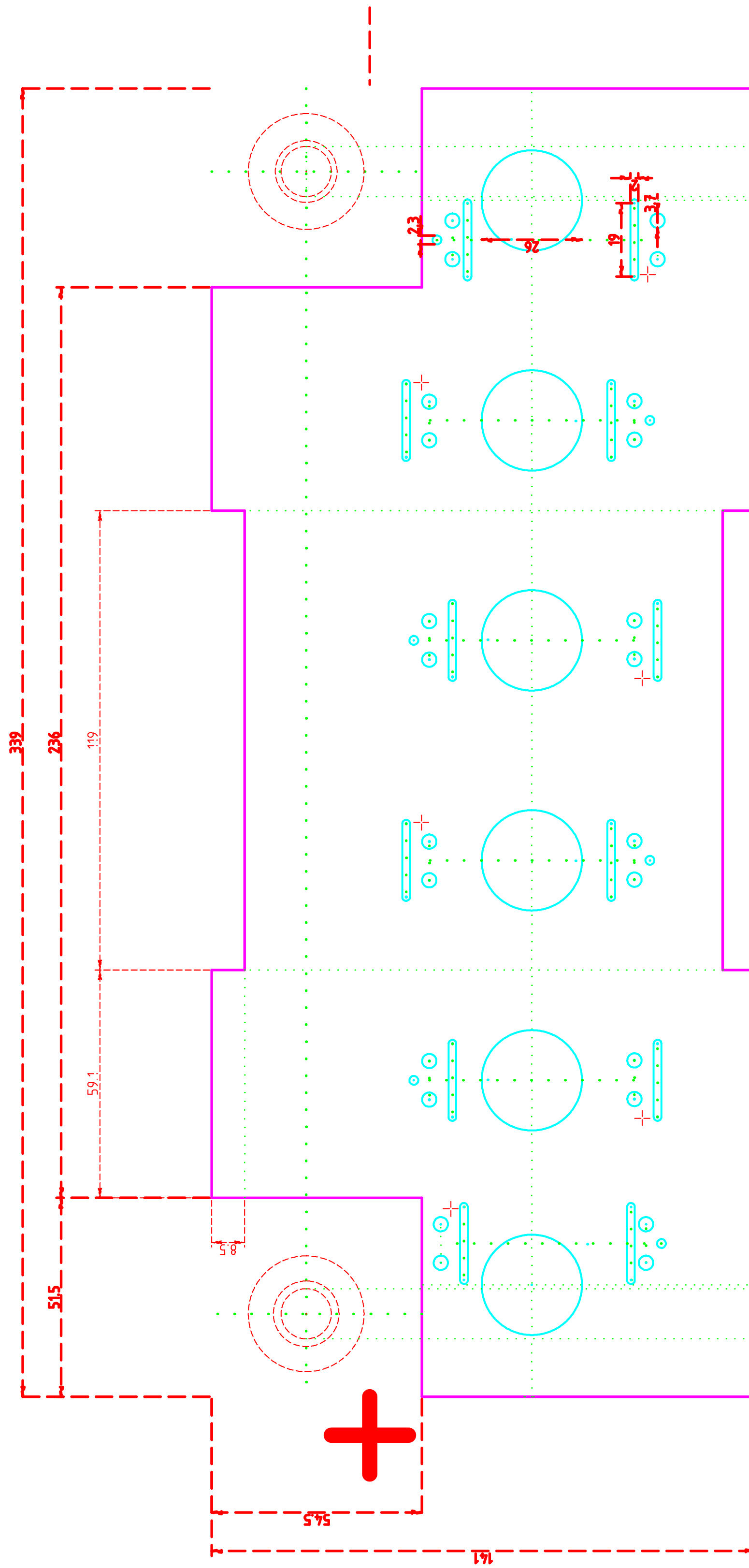


Abbildung 3.: Konstruktionszeichnung des Batterie-Deckels der Starterbatterie von Moll (Verhältnis 1:2)

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) 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, 17. Juni 2011	
Ort, Datum	Unterschrift