Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Niels Jegenhorst

Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

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

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

Niels Jegenhorst

Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Studiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. -Ing. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr. -Ing. Stephan Hußmann

Abgegeben am 27. Oktober 2011

Niels Jegenhorst

Thema der Masterarbeit

Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation

Stichworte

Batteriezellensensor, Sensornetz, Mikrocontroller, Sensorsignalverarbeitung

Kurzzusammenfassung

Diese Arbeit behandelt die Konzeption und Realisierung eines Zellensensors mit bidirektionaler drahtloser Kommunikation. Der Uplink-Kanal verläuft von den Sensoren zur zentralen Batteriemanagementeinheit, der Downlink-Kanal in der entgegengesetzten Richtung. Es erfolgt eine vielfältige Verwendung des Downlink-Kanals, für die Wake-Up-Funktion, zur Synchronisation und zur Datenübertragung. Die Besonderheit, eine begrenzte Menge an Energie aus der Batteriezelle für den Betrieb der Sensoren beziehen zu können, wird durch ein energiesparendes Konzept gelöst. Die Batteriemanagementeinheit stellt hierbei die zentrale Gegenstelle zu den Sensoren dar, so dass in der Gesamtheit ein Spezialfall eines sternförmig angeordneten Sensornetzes entsteht.

Niels Jegenhorst

Title of the master thesis

Development of a cell sensor for automotive batteries with bidirectional wireless communication

Keywords

battery cell sensor, sensor nodes, microcontroller, sensor signal processing

Abstract

This paper discusses the conception and realization of a cell sensor with bidirectional wireless communication. The uplink channel runs from the sensors to the central battery management unit, the downlink channel in the opposite direction. There is a varied use of the downlink channel for the wake-up feature, for synchronization and for data transmission. Relate the particularity of a limited amount of energy from the battery cell for the operation of the sensors can will be solved by an energy-saving concept. The battery management unit in this case represents the central counterpart to the sensors, so that in the entirety is formed a special case of a star-shaped arranged sensor network.

Danksagung

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. -Ing. Karl-Ragmar Riemschneider, betreuender Prüfer und Projektleiter des Forschungsvorhabens BATSEN, für die Ermöglichung dieser Masterarbeit und den immerzu sehr engagierten Einsatz bedanken.
Ein weiterer Dank geht an Herrn Prof. Dr. -Ing. Stephan Hußmann, den Zweitgutachter dieser Arbeit.

Außerdem möchte ich Herrn Prof. Dr. -Ing. Jürgen Vollmer, ebenfalls Projektleiter des Forschungsvorhabens BATSEN, und Herrn Dipl. -Ing. Günter Müller für die stets gute Unterstützung meinen Dank aussprechen.
Für die fachliche und tatkräftige Unterstützung bedanke ich ebenfalls bei Herrn Dipl. -Ing. Helmut Otte und Herrn Dipl. -Ing. Matthias Schneider

Besonders möchte ich mich ebenso bei Herrn Dipl. -Ing. (FH) Martin Krey bedanken, der mich stets mit guten Ratschlägen ermutigen konnte.
Gleichsam danke ich meinen Kommilitonen Simon Püttjer und Raik Kube, welche ebenfalls im Rahmen des Projektes BATSEN ihre Abschlussarbeit verfasst haben, sowie Kalin Ivanov und Markus Piorek für ihre Unterstützungen.

Ein ganz besonderer Dank geht an meine Verlobte Maike Weber, die mich stets mit all ihren Kräften unterstützt.

Inhaltsverzeichnis

1 Einführung

In vielen Anwendungsgebieten werden mehrzellige Batterien als Energiespeicher eingesetzt. Zum einen als Antriebs- und Traktionsbatterie in elektrischen Fahr- und Förderzeugen (Gabelstapler, Flurförderfahrzeuge), zum anderen als Starter- und Pufferbatterie in konventionellen Automobilen oder in unterbrechungsfreien Stromversorgungen.
Für einen gesicherten und optimierten Betrieb dieser Batterien ist eine Überwachung der einzelnen Batteriezellen von großer Bedeutung. Im Gegensatz zu einer Überwachung der Batterie, als eine geschlossene Einheit von Zellen, lassen sich so bedeutend mehr wichtige Informationen gewinnen. Mit diesen ist es möglich eine detailliertere und genauere Aussage bezüglich der Betriebssicherheit und der Verfügbarkeitsprognose zu treffen.

Um einen messtechnischen Zugang zu den einzelnen Batteriezellen zu erhalten, gibt es verschiedene Lösungsansätze. Zu einem die drahtgebundenen Systeme, zum anderen die drahtlosen Systeme. An der HAW Hamburg gibt es hierzu das von dem Bundesministerium für Bildung und Forschung geförderte Forschungsvorhaben „BATSEN“, welches sich mit Konzepten der drahtlosen Systeme auseinandersetzt.
Das folgende Bild 1.1 veranschaulicht das Prinzip eines drahtlosen Systems für die Überwachung der einzelnen Batteriezellen.

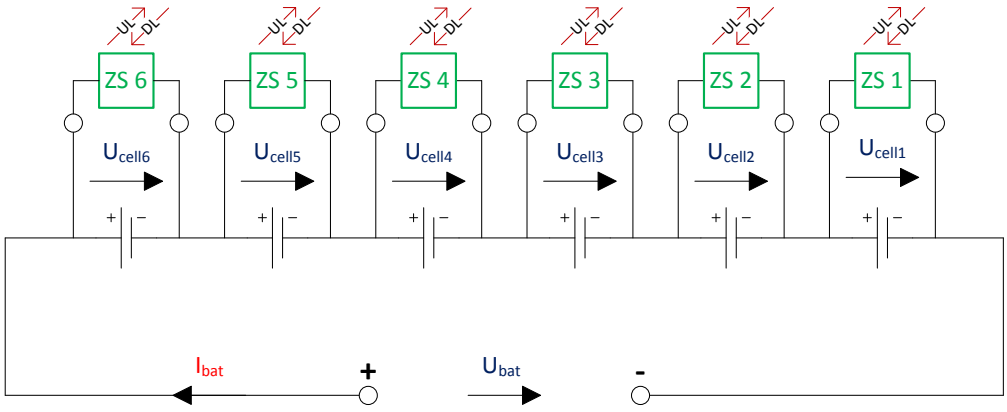


Bild 1.1: *Schematische Darstellung einer Batterie mit Zellensensoren*

Jede Zelle der Batterie wird hierbei mit einem Zellensensor bestückt, welcher mittels einer noch zu entwickelnden bidirektionalen drahtlosen Schnittstelle mit einer zentralen Batteriemanagementeinheit kommunizieren kann. Die Erfassung des Stromflusses hingegen kann zentral durch die Batteriemanagementeinheit erfolgen. Im Gegensatz zu herkömmlichen drahtlosen Sensoren, können in dieser Anwendung die Zellensensoren eine begrenzte Menge an Energie für den Betrieb aus der Batteriezelle entnehmen. Die Menge der verfügbaren Energie variiert abhängig von dem Anwendungsgebiet.

1.1 Motivation

Innerhalb dieses Forschungsvorhabens sind bereits einige Vorarbeiten[1] durchgeführt worden. Diese Abschlussarbeiten beschäftigten sich mit einem drahtlosem System von Zellensensoren, welche nur über einen Nachrichtenkanal zu der Batteriemanagementeinheit verfügen. Daraus resultiert, dass diese Sensoren nur mit einer statistischen Erfassung und unkoordinierten Übertragung von Messwerten arbeiteten können. Es stellte sich heraus, dass speziell bei wechselnden Lang- und Kurzzeitbelastungen diese Arbeitsweise nachteilig sein kann.
Innerhalb dieser Masterarbeit soll ein neuer Zellensensor entwickelt werden, der über einen bidirektionalen drahtlosen Nachrichtenkanal verfügt und somit die Nachteile der bisherigen unkoordinierten Betriebsart vermeidet. Durch die bidirektionale Kommunikation können zudem diverse Funktionalitäten ermöglicht werden, die ohne diese nicht möglich sind. Für den energiesparenden Betrieb der Zellensensoren wäre beispielsweise eine Aktivierung dieser, ausgelöst durch die drahtlose Kommunikation zu den Sensoren, ein angestrebtes Ziel. Die bidirektionale Kommunikation würde ebenso eine Steuerung von Effektoren auf den Zellensensoren, zum Zweck der Ladungsbalancierung, ermöglichen. Eine Ladungsbalancierung dient dem Ausgleich von Unterschieden im Lade- und Alterungszustand zwischen den Zellen einer Batterie. Erst die Überwachung der einzelnen Zellen einer Batterie durch die Zellensensoren ermöglicht es, derartige Differenzen überhaupt zu erkennen und zu bewerten.

1.2 Kategorisierung der Zellensensoren

Aufgrund von differenzierten Anwendungsgebieten sind innerhalb des Projektes „BATSEN“ drei Klassen von Zellensensoren definiert. Diese Klassen umfassen au-

[1] siehe Plaschke (2008), Püttjer (2011)

tonome, teilautonome und zentral kommandierte Zellensensoren, welche im Rahmen des Projektes teils noch zu entwickeln und zu untersuchen sind. In der folgenden Tabelle 1.1 befindet sich eine Gegenüberstellung der Eigenschaften von den Klassen.

	Klasse 1	Klasse 2	Klasse 3
Sensorfunktion / Moduswechsel	autonom	teilautonom	zentral kommandiert
Zeitliche Netzorganisation	ohne Synchronisation	einfachste Synchronisation	komplexe Synchronisation
Übertragung vom Steuergerät zum Sensor	kein Downlink	Downlink mit Broadcast-Wake-up	Downlink mit Broadcast und adressierte Kommandos
Ladungs-balancierung in der Zelle	autonome Steuerbarkeit, bisher kein Ansatz erkennbar	teilautonome Steuerbarkeit denkbar	zentrale Steuerbarkeit möglich
Hardwareaufwand im Sensor und Steuergerät	sehr gering	hoch	sehr hoch

Tabelle 1.1: *Klassifikation der Zellensensoren, entnommen aus Riemschneider u. Vollmer (2010)*

Inwieweit eine Klasse einen bestimmten Anwendungsgebiet, beispielsweise den Fahr- oder Förderzeugen, zugeordnet werden kann, bedarf es noch an Analysen. Tendenzen lassen sich jedoch aufgrund der zur Verfügung stehenden Energie für den Betrieb der Sensoren, insbesondere aber anhand der preislichen Relation, aufzeigen. Die Kosten für eine Antriebs- und Traktionsbatterie, wie sie in Fahr- oder Förderzeugen verwendet werden, übersteigen üblicherweise ein Vielfaches der Kosten von Starterbatterien in herkömmlichen Kraftfahrzeugen. Ein mögliches Anwendungsgebiet der Klasse 3 wäre somit folglich innerhalb dieser Kategorie von Batterien denkbar.

1.3 Einordnung der Aufgabenstellung zum Gesamtprojekt

Der in dieser Masterarbeit zu konzipierende Zellensensor soll der erste realisierte Sensor für die Klasse 2 innerhalb des Projektes sein. Da der Konzeption zunächst

ein wesentlicher Anteil an Recherche und Analyse vorausgeht und so auch abweichende Lösungsmöglichkeiten vorgestellt werden, wird ein grundlegender Beitrag für weitere Entwicklungen im Projekt geschaffen.
Mit der Realisierung des neuen Zellensensors kann erstmals der zusätzliche Hardwareaufwand für einen Sensor mit bidirektionaler Kommunikation veranschaulicht werden. Mit den dabei erzielten Erkenntnissen, sowie weiteren Erprobungen, lassen sich Optimierungsmöglichkeiten für zukünftige Versionen, möglicherweise auch für Sensoren anderer Klassen, ableiten.
Durch die Realisierung des Nachrichtenkanals zu den Sensoren, wird erstmals eine Steuerung der Zellensensoren ermöglicht. Hierdurch eröffnet sich die Möglichkeit diverse neue Funktionalitäten für die Zellen-Sensorik zu entwickeln, welche zuvor nicht realisierbar waren. Ein Schwerpunkt für zukünftige Arbeiten könnte sich in dem Bereich der Erprobung von neu hinzuzufügenden Funktionalitäten befinden.
Aufgrund der erstmals synchronisierbaren Messwerterfassung ergeben sich neue Aspekte für die Verarbeitung der Messwerte. Diese Messwerte dienen als Basis für noch, im weiteren Verlauf des Forschungsvorhabens, zu implementierende Batteriemodelle. Diese sollen dazu verwendet werden Aussagen bezüglich der Betriebssicherheit und der Verfügbarkeitsprognose von Batterien vorzunehmen.

2 Analyse

2.1 Aktueller Stand der Zellen-Sensorik im Projekt

Der in den Vorarbeiten verwendete Zellensensor der Klasse 1 ist so konzipiert, dass er ohne einen Nachrichtenkanal zwischen Basisstation und Zellensensor (*Downlink (DL)*) auskommt. Er ist bewusst sehr minimalistisch aufgebaut (siehe Plaschke (2008)) und besteht im wesentlichem aus einem *Mikrocontroller (MCU)*, einem *Gleichspannungswandler (DC-DC-Wandler)* und einem 433 MHz Transmitter für den Nachrichtenkanal zur Basisstation (*Uplink (UL)*).
Als Übertragungsverfahren für den *UL* dient eine einfache *On-Off Keying (OOK)* Modulation. Da der Zellensensor aufgrund des nicht vorhandenen *DL* keine Möglichkeit der Synchronisation für den *UL* hat, kommt für die Übertragung ein unkoordiniertes Verfahren zum Einsatz. Dieses Verfahren besteht darin, dass jeder Zellensensor seine aufgenommenen Messwerte innerhalb einer mittleren Zeitdauer t_{sr} sendet, die um eine pseudozufällige Zeit $\pm\Delta t_{ps}$ variiert. So wird erreicht, dass jedes gesendete Paket mit einer gewissen Wahrscheinlichkeit (siehe Püttjer (2011) und Kube (2011)) bei dem Empfänger, der Basisstation, ankommt. Abhängig von der Anzahl der Zellensensoren, die den *UL*-Kanal belegen, der Übertragungsdauer bzw. Framelänge, sowie einer vorgegebenen *Frame-Fehlerrate (FER)*, ergibt sich eine maximal mögliche mittlere Senderate, mit der die Sensoren ihre Messwerte senden können.

Bei der Überwachung von Ereignissen mit geringer zeitlicher Veränderung, entsprechend einer möglichst konstanten Belastung der Batteriezellen, zeigte sich, dass das unkoordinierte Verfahren der Klasse 1 zur Messwertaufnahme und Übertragung sehr gut geeignet ist. Sobald jedoch die zeitliche Dauer von Ereignissen in die Größenordnung von der mittleren Senderate der Zellensensoren kommt, welche zugleich die Messrate darstellt, sind derartige Ereignisse nicht mehr zuverlässig bzw. nur unvollständig zu erfassen.
Bei einer mittleren Senderate von 500 ms, wie bei sechs Zellensensoren einer Starterbatterie bei Voruntersuchungen verwendet[1], ist ein Hochstromereignis wie der Startvorgang eines Verbrennungsmotors, mit einer Dauer von nur etwa 1,5 s, kaum aussagekräftig messbar. Zudem ist aufgrund der unkoordinierten Übertragung

[1] vgl. Püttjer (2011) sowie Kube (2011)

zusätzlich eine *FER* von 41 % stets vorhanden. Es können somit im Mittel nur 1,18 Frames pro Sekunde eines Zellensensors erfolgreich übermittelt werden.
Ein Hochstromereignis wie der Startvorgang, welcher aufgrund des hohen stattfindenden Energieumsatzes vermutlich einen entscheidenden Beitrag für die Zustandsdiagnose der Batterie liefert, ist somit nur teilweise erfassbar.

Obwohl dem Zellensensor der Klasse 1 kein *DL* für die Koordination zur Verfügung steht, ist im Rahmen einer Diplomarbeit (siehe Püttjer (2011)) ein Verfahren entwickelt, realisiert und erprobt worden, welches die Erkennung von Hochstromereignissen gestattet. Ist ein solches Ereignis anhand der hohen Dynamik der Zellenspannung erkannt, zeichnet der Sensor eine vermehrte Anzahl von Messwerten auf, welche anschließend mit der maximal vorgegebenen Senderate über den *UL*-Kanal übertragen werden. Anschließend nach der Transmission aller zusätzlichen Messwerte reduziert sich wiederum die Senderate auf ein geringeres Niveau.
Als problematisch erwies sich hierbei unter anderem die Rekonstruktion der Aufnahmezeitpunkte von den zusätzlichen Messwerten, da jeder Zellensensor einen unterschiedlich schnell laufenden Zeitgeber aufweist. Zudem ist für die Beobachtung der Dynamik von der Zellenspannung eine kontinuierliche Abtastung und Bewertung notwendig, was zu einem erhöhtem Energiebedarf führt. Der Schwellwert, ab wann ein Hochstromereigniss erkannt wird, lässt sich zudem nur zum Zeitpunkt der Programmierung einmalig vorgeben, eine spätere Anpassung im laufendem Betrieb ist nicht möglich.

Würde der Zellensensor hingegen über einen *DL*-Kanal verfügen, wäre über diesen einerseits eine Synchronisation der Zeitgeber möglich, sowie könnten diese durch einen Broadcast getriggert, Hochstromereignisse synchron aufzeichnen. Darüber hinaus ist durch den *DL*-Kanal eine Parametrisierung der Messwertaufnahme, sowie insbesondere eine Koordination der *UL*-Kanal Kommunikation für die Übertragung der Messwerte des Ereignisses, durchführbar.

2.2 Abschätzung der Anforderungen

Für die Konzeption des zu entwickelnden Zellensensors, wie auch bereits für die Auswahl der vorzustellenden Transceiver Konzepte, gilt es die Anforderungen an die Zellen-Sensorik abzuschätzen. Diese Anforderungen sollten im Idealfall möglichst konkret und detailliert verfasst werden, entsprechend einer funktionellen Spezifikation. Praktisch ist dies, wie es sich bei den folgenden Unterpunkten zeigt, nicht immer eindeutig möglich. Grund hierfür ist die Tatsache, dass das Projekt „BATSEN“ erst noch am Anfang steht und der neue Zellensensor erstmalig dazu dient, Untersuchungen im Bereich der Klasse 2 durchzuführen.

2.2.1 Spannungsbereich

Als Messobjekt für den neuen Zellensensor soll neben den bisher im Projekt stets verwendeten Bleibatterien auch moderne Lithiumbatterien verwendbar sein, die einen höheren Spannungsbereich aufweisen. In der Tabelle 2.1 sind die relevanten Zellenspannungen[2] für verschiedene Batterietypen aufgelistet.

Batterietyp	Nenn-spannung U_{z0} in V	Entladeschluss-spannung U_{zm} in V	Ladeschluss-spannung U_{zl} in V
Bleibatterie	$2,1$	$\approx 1,75$	$\approx 2,42$
Gelbatterie	$2,1$	$\approx 1,85$	≈ 2.35
AGM-Batterie	$2,1$	$\approx 1,85$	$\approx 2,4$
Lithium-Ionen-Batterie (typ.)	$3,6$	$\approx 2,5$	$\approx 4,2$
Lithium-Eisen-Phosphat ($LiFePO_4$)	$3,3$	-	-
Lithium-Titanat ($Li_{4/3}Ti_{5/3}O_4$)	$2,3$	-	-

Tabelle 2.1: *Batterietypen und deren jeweiligen Zellenspannungen bei* 25 °C

Der benötigte Spannungsbereich um alle aufgeführten Batterietypen verwenden zu können, liegt demnach bei $1,75$ bis $4,2$ V. Da die Zellensensoren eine Überwachung der Batteriezellen übernehmen sollen, ist es zweckmäßig, den Spannungsbereich zu erweitern. So führen tiefentladene oder überladene Batteriezellen, genauso wie Hochstromereignisse, zu Extremwerten, die ebenfalls durch die Zellensensoren messbar sein müssen.

Als maximal zulässige Zellenspannung $U_{z,max}$ sollte demnach $> 4,5$ V angenommen werden, sowie als minimale Zellenspannung $U_{z,min} < 1,0$ V.

2.2.2 Messgenauigkeit und -intervall

Die Genauigkeit mit der die Messung der Zellenspannung zu erfolgen hat, soll in dem Bereich von wenigen Millivolt liegen. Konkrete Vorgaben hierzu sind aufgrund des noch fehlenden Batteriemodells noch nicht vorhanden.
Ebenso lässt sich die erforderliche zeitliche Genauigkeit (*Jitter*) und das Intervall

[2] entnommen aus Reif (2010), Linden u. Reddy (2010), Kuchling (2004)

der Spannungsmessung nur grob abschätzen. Bei einem Hochstromereigniss beispielsweise, muss das Intervall, als auch der *Jitter* der Zellensensoren zueinander, sehr viel genauer sein als während einer Messung bei Ruhestrom.
Sowohl bei der Messgenauigkeit als auch dem Messintervall, gilt es für die spätere Auslegung den erforderlichen Aufwand mit dem resultierenden Nutzen zu vergleichen und anschließend daraus einen Kompromiss zu schließen.

2.2.3 Energiebedarf

Für die Energieversorgung der Zellensensoren dient ihr jeweiliges Messobjekt, die Batteriezelle. Wie viel Energie von der Batteriezelle entnommen werden kann, hängt von den Betriebsbedingungen der Batterie in Relation zur ihrer Kapazität ab. Die folgende Aufstellung soll hierzu einen Überblick geben:

- Antriebs- und Traktionsbatterien:
  - Typische Nennkapazität $\approx 460 \ldots 920\,\mathrm{Ah}$
  - Anwendung in Fahr- oder Förderzeugen
  - Betrieb vorwiegend in Abfolge von Lade- und Entladephasen
- Starter- und Pufferbatterien:
  - Typische Nennkapazität $\approx 36 \ldots 120\,\mathrm{Ah}$
  - Anwendung z. B. in konventionellen Automobilen
  - Betrieb in Abfolge von Lade- und Entladephasen sowie längeren Ruhephasen
- Pufferbatterien:
  - Typische Nennkapazität $\approx 5 \ldots 260\,\mathrm{Ah}$
  - Anwendung in unterbrechungsfreien Stromversorgungen
  - Betrieb vorwiegend im Ladezustand mit kurzzeitigen Entladephasen

Um zu verdeutlichen wie viel Energie von einem Zellensensor in etwa verbraucht werden darf, folgt ein vereinfachtes Beispiel, anhand einer herkömmlichen Bleibatterie mit $n = 6$ Zellen.
Die mittlere Stromaufnahme eines Zellensensors soll beispielsweise 10 mA betragen. Die Leistungsaufnahme P_{zs} je Zellensensor folgt damit zu:

$$P_{zs} = I_{zs} \cdot U_z = 10\,\mathrm{mA} * 2\,\mathrm{V} = 20\,\mathrm{mW} \tag{2.1}$$

mit

I_{zs} : Stromaufnahme des Zellensensors
U_z : Zellenspannung

Demzufolge haben alle $n = 6$ Zellensensoren einer Batterie eine Leistungsaufnahme von:

$$P_{zs,ges} = n \cdot P_{zs} = 120\,\text{mW} \tag{2.2}$$

Soll die Batterie beispielsweise ein Jahr lang gelagert werden, benötigen die Zellensensoren alleine folgende Energie:

$$E = P_{zs,ges} \cdot 24\,\text{h} \cdot 365 = 1051{,}2\,\text{Wh} \tag{2.3}$$

$$\text{oder} \quad Q = \frac{P_{zs,ges} \cdot 24\,\text{h} \cdot 365}{6 \cdot U_z} = 87{,}6\,\text{Ah} \tag{2.4}$$

Üblicherweise hat eine Starterbatterie von Kraftfahrzeugen eine Nennkapazität von $Q_n = 36$ bis 120 Ah. Entsprechend erkennt man, dass eine Starterbatterie durchaus innerhalb eines Jahres entladen werden kann, wenn diese nicht zwischendurch nachgeladen wird. Insbesondere da zu der Stromentnahme der Zellensensoren auch noch die Selbstentladung der Batterie hinzukommt.
Die Selbstentladung einer Batterie ist abhängig von verschiedenen Parametern, wie die Temperatur, der Ruhespannung und dem Batterietyp. Typischerweise wird hierbei eine Größenordnung von etwa 0,2 %/Tag der Kapazität angegeben (vgl. Reif (2010)). Die Entladung, durch die in dem Beispiel angenommen Zellensensoren, beträgt bei einer 80 Ah Batterie 0,3 %/Tag und ist damit bereits größer als die Selbstentladung.
Neben der Selbstentladung ist zusätzlich der permanente Ruhestrom von etwa 35 mA, verursacht durch die Bordelektronik eines Kraftfahrzeugs, relevant. Bezogen auf eine 80 Ah Batterie ergibt sich somit 1,05 %Tag zusätzliche Endladung.

Anhand dieses Beispiels erkennt man, dass insbesondere für Starterbatterien es wichtig ist, die durchschnittliche Leistungsaufnahme der Zellensensoren möglichst minimal ($P_{zs} \ll 20\,\text{mW}$) zu halten. Auf die Größen Selbstentladung und Ruhestrombedarf kann hingegen kein Einfluss genommen werden. Folglich wäre es sinnvoll, wenn die Zellensensoren nur dann Energie verbrauchen, falls diese auch aktiv benötigt werden.

Weiterhin ist für den Betrieb der Zellen-Sensorik eine zentrale Reader-Datenlogger-Einheit (Basisstation) als zentrale Gegenstelle erforderlich, welche ebenfalls Energie von der Batterie benötigt. Diese wird im Gegensatz zu den Zellensensoren, direkt mit der externen Steuerelektronik des Fahrzeugs verbunden und somit von dieser,

nur bei Bedarf aktiviert. Zwar ist davon auszugehen, dass der Energiebedarf der aktiven Basisstation wesentlich größer wie der eines einzelnen Zellensensors ist, hingegen sind jedoch stets mehrere Zellensensoren gegenüber nur einer Basisstation erforderlich.
Vereinfacht lässt sich dieser Zusammenhang anhand der benötigten mittleren Leistungsaufnahme $P_{tx,ges}$ für die Transmission von Informationen innerhalb eines Zyklus, dargestellt in der Gleichung 2.5, veranschaulichen. Diese Gleichung stellt eine Vereinfachung dar, welche beispielsweise die grundlegende Leistungsaufnahme für den Betrieb (vgl. Abschnitt 4.4.1) nicht berücksichtigt. Dabei kann man zudem annehmen, abhängig von dem gewähltem Übertragungsverfahren, dass die Basisstation nur alle s_{bs} Zyklen Informationen zu den Sensoren sendet.

$$P_{tx,ges} = n \cdot P_{tx,zs} + \frac{1}{s_{bs}} \cdot P_{tx,bs} \tag{2.5}$$

Nimmt man vereinfacht an, die Leistung $P_{tx,zs}$ wäre etwa gleich $P_{tx,bs}$, folgt offensichtlich eine um den Faktor ns_{bs} erhöhter Leistungsbedarf innerhalb eines Zyklus, für die Übertragung von Informationen seitens der Zellensensoren .
Da der Schwerpunkt der Arbeit auf die Entwicklung der Zellensensoren gelegt ist, wird der Energiebedarf der *Basisstation* nicht weiter betrachtet.

2.2.4 Nachrichtenkanal

Für die Anforderungen an den Nachrichtenkanal muss zwischen dem *UL*-Kanal und dem *DL*-Kanal unterschieden werden.

Der *UL*-Kanal dient hauptsächlich dazu, die Messdaten der Zellensensoren an die Basisstation zu übermitteln. Hierbei werden, je nach Betriebsart und Anzahl der Zellensensoren die den Kanal belegen, Übertragungsraten im Bereich von einigen kbit/s benötigt.
Das bisherige System der Zellen-Sensorik verwendet eine Datenrate von 5 kBit/s in Verbindung mit einer Manchester-Codierung, entsprechend einer Übertragungsrate von 10 kBit/s. Ein einzelnes Frame bestehend aus 108 Bit (siehe Abschnitt 4.3.1.2), benötigt hierbei 21,6 ms für die Übertragung der Messwerte zur Basisstation. Die effektive Datenrate des jetzigen Systems beträgt, unter Berücksichtigung[3] einer mittleren Senderate von 500 ms und einem *FER* von 41 %, sowie einer Anzahl von sechs Zellensensoren, entsprechend 127,44 Bit/s je Zellensensor. Diesen Wert gilt es, für das zu entwickelnde System, mindestens zu erreichen. Sowohl das bereits verwendete Frame, als auch die Datenrate sollen für den zu konzipierenden Zellensensor nach Möglichkeit aus Kompatibilitätsgründen übernommen werden.

[3] vgl. Püttjer (2011) sowie Kube (2011)

Hingegen dient der *DL*-Kanal vorrangig zur Steuerung der Zellensensoren, wobei, abhängig von der Betriebsart, geringere Übertragungsraten als für den *UL*-Kanal nötig sind.
Der Empfänger für den *DL*-Kanal sollte möglichst derart gestaltet sein, dass dieser jederzeit in der Lage ist eine Nachrichtenübertragung zu detektieren. Hierbei soll bewusst nur gefordert werden, jederzeit eine Übertragung erkennen zu können und nicht auch zu demodulieren oder gar zu dekodieren. Diese Eigenschaft ist zwingend erforderlich, da die Zellensensoren, beispielsweise unmittelbar vor einem Hochstromereigniss, entsprechend konfiguriert werden müssen.
Durch diese Forderung kann der Zellensensor so ausgelegt werden, dass dieser über möglichst wenige aktive Komponenten verfügt, die dauerhaft aktiv von der Batteriezelle mit Leistung versorgt werden müssen. Somit ließe sich der Energiebedarf, welcher möglichst minimal sein sollte, reduzieren.

2.2.5 Wirkungsbereich und Umgebungseinflüsse der Funkübertragung

Für die benötigte Reichweite der drahtlosen Übertragung zwischen der Basisstation und den Zellensensoren lassen sich momentan nur grobe Richtwerte angeben. Die benötigte Reichweite hängt zum einen von der Geometrie des verwendeten Messobjektes (der Batterie) ab, zum anderen von der Einbaulage der Antennen von der Basisstation und den Zellensensoren zueinander. Entscheidend ist zudem die Umgebung, in welcher die drahtlose Übertragung erfolgen soll. So beeinflussen Metallflächen in der näheren Umgebung maßgeblich die Wellenausbreitung. In Folge einer metallischen Umgebung kann eine Dämpfung der Übertragung auftreten, ebenso wie die Verstimmung der verwendeten Antennen, was eine Reduzierung der Güte zur Folge haben kann.

Als konkretes Messobjekt wird zunächst vereinfacht die Antriebs- und Traktionsbatterie, des für dieses Projekt zur Verfügung stehenden Gabelstaplers, angenommen. Die maximale Distanz zwischen einer mittig positionierten Antenne der Basisstation und eines Zellensensors beträgt hier 31,5 cm. Ein anderes konkretes Messobjekt ist eine modifizierte Starterbatterie (siehe Püttjer (2011)), bei der die maximale Distanz 13 cm beträgt.
Somit wird für die benötigte Reichweite, als Richtwert bei der Auswahl der Konzepte, ein Bereich von 35 bis 50 cm angenommen.

2.2.6 Realisierungsaufwand und Integrierbarkeit

Insbesondere die Starterbatterien für herkömmliche Kraftfahrzeuge sind ein Massenprodukt und damit sehr kostensensitiv. Insbesondere diese Arten von Batterien

stellen erhöhte Anforderungen bezüglich des Energiebedarfs, da ihre Kapazität, im Vergleich zum Beispiel zu Traktionsbatterien, gering ist. Zudem muss der Zellensensor die Funktionalität dazu haben auch kurzzeitige Ereignisse, wie den Startvorgang des Verbrennungsmotors, entsprechend zu behandeln.
Der Zellensensor muss also einerseits besonders kostengünstig produzierbar sein, andererseits jedoch diverse Eigenschaften aufweisen, welche hohe Anforderungen an den hardwaretechnischen Aufbau stellen.

Ein wichtiger Punkt zur Kostenreduktion des Zellensensors soll sein, dass er möglichst keinen Quarzoszillator als Taktgeber für die *MCU*, als auch für den *UL*-Kanal, benötigt. Zusätzlich zu den relativ hohen Kosten eines Quarzoszillators, ist hierbei die Forderung nach Robustheit relevant. Insbesondere dieser Punkt bestimmt maßgeblich die Konzeption des Zellensensors, da für ein herkömmliches System stets ein präziser Taktgeber nötig ist.

Weiterhin ist der Aspekt der projektierten Integration der Analog-, Digital- und *Kurzwellen (engl. High Frequency) (HF)*-Schaltungen des Zellensensors in einen Mikrochip von großer Bedeutung. Hierfür ist es vorteilhaft, wenn einige Randbedingungen, wie zum Beispiel der Verzicht auf Quarzoszillatoren, beachtet werden.
Durch die geplante spätere Integration sind die Aspekte der besonders kostengünstigen Realisierung, in Verbindung mit minimalen geometrischen Ausmaßen, für den zu realisierenden Zellensensor, nicht maßgebend. Entscheidend ist hierbei der Entwurf eines „erprobungsfreundlichen“ Konzeptes, welches gestattet, systematische Entscheidungen für die Integration zu treffen.

2.3 Passive Transceiver Konzepte

Für die Realisierung des neuen Zellensensors, welcher gleichzeitig über einen *DL*-Kanal und einen *UL*-Kanal verfügen soll, gilt es ein für diese Anwendung passendes Sender- und Empfängerkonzept auszuwählen.
Von besonderem Interesse ist dabei zunächst der Empfänger. Dieser sollte dauerhaft empfangsbereit sein und ist somit maßgeblich für den Energiebedarf des Zellensensors. Grundsätzlich gibt es zunächst zwei Kategorien von Empfängerkonzepten, zwischen denen zu unterscheiden ist:

- aktive Empfänger:
  - Ermöglichen den Empfang von Informationen über größere Distanzen auch bei relativ schwachem Signalpegel.
  - Benötigen typischerweise eine permanente Energieversorgung, mit Spannungen größer als die minimal angesetzte Zellenspannung, in dem Bereich von einigen Milliampere.
- passive Empfänger:
  - Empfang von Informationen nur bei relativ hohen Signalpegel und kleineren Distanzen.
  - Benötigen hingegen meist keine permanente Energieversorgung, oder alternativ Verwendung von Spannungen im Bereich der minimal angesetzten Zellenspannung.
  - Implizieren mitunter mit geringem zusätzlichem Aufwand einen *UL*-Kanal.

Aufgrund der Forderung nach einem möglichst geringem Energiebedarf des Zellensensors, werden die aktiven Empfänger bei der Konzeption zunächst nicht weiter berücksichtigt.
Vielmehr sollen die passiven Empfänger Konzepte näher untersucht werden. Diese finden oftmals Anwendung in dem Bereich der *Radio-Frequency Identification (RFID)*-Systeme und implizieren meist einen Transmitter, wovon die Bezeichnung passive Transceiver folgt. *RFID*-Systeme haben die besondere Eigenschaft, neben der Übertragung von Informationen, auch die Übertragung von Energie über den *DL*-Kanal ermöglichen. Für die Umsetzung der *Wake-Up*-Funktionalität, zum Aufwecken der Zellensensoren durch den *DL*-Kanal, kann diese Eigenschaft ausgenutzt werden (vgl. Abschnitt 3.2).

2.3.1 Grundlagen über RFID-Systeme

Ein *RFID*-System besteht, entsprechend dem Bild 2.1, aus einem zentralen Erfassungs- oder Lesegerät (*Reader*) und einen oder mehreren kontaktlosen Datenträgern (*Transponder*). Diese Systeme, die zur Kategorie der automatischen Identifikationsverfahren zählen, dienen der kontaktlosen Bereitstellung von Informationen über Objekte jeglicher Art, wie zum Beispiel der Warenkennzeichnung, sowie als Ersatz für kontaktbehaftete Chipkarten. Da es sich hierbei oftmals um Anwendungen für den Massenmarkt handelt, sind die verwendeten Konzepte für die Transponder dementsprechend minimalistisch, um kostensparend zu sein.

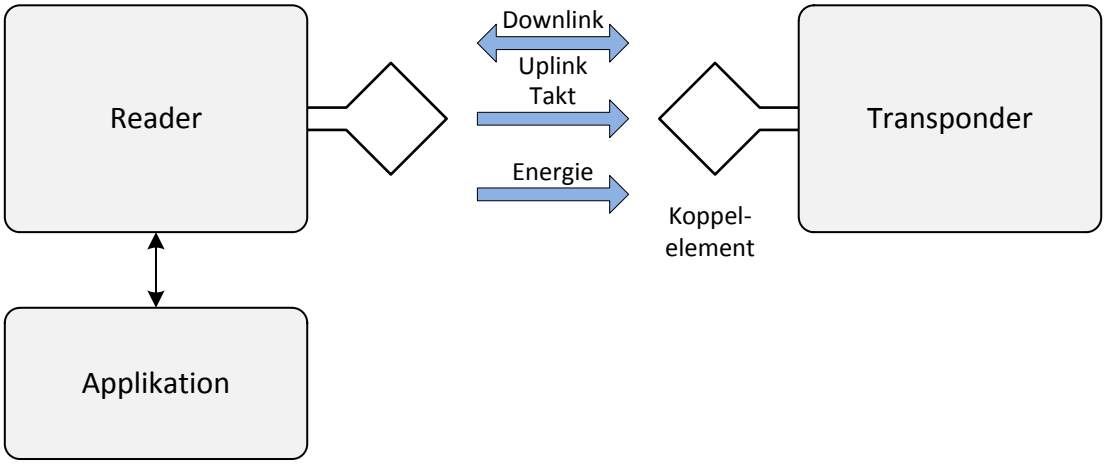


Bild 2.1: *Komponenten eines RFID-Systems, entnommen aus Finkenzeller (2008)*

Der Reader dient typischerweise dazu, Informationen von dem Transponder auszulesen und einer übergeordneten Anwendung bereitzustellen. Hierzu stellt dieser dem Transponder wiederum einen Takt, sowie Energie für die Datenverarbeitung als auch für die Datentransmission zum Reader (*UL*), zur Verfügung. Oftmals beinhaltet der Reader einen *DL*-Kanal zu dem Transponder, um Steuerbefehle oder Daten zu transferieren.

Hingegen dient der Transponder meist nur der reinen Speicherung von Informationen. Üblicherweise beinhaltet er einen sehr einfachen Empfänger, eine einfache logische Schaltung, oder einen Mikrocontroller, sowie zur Transmission von Informationen einen Sender. Den Teil der Hardware, der dabei als Empfänger und Sender, sowie eventuell zur Spannungsversorgung dient, wird im Allgemeinen als *Front-End* bezeichnet. Abhängig davon, ob der Transponder zudem eine eigene unabhängige Spannungsversorgung beinhaltet, bezeichnet man diese als aktive oder passive Transponder. Bei den aktiven Transponder stellt diese die Energie für den Betrieb der logische Schaltung, oder des Mikrocontrollers ganz oder teilweise zur

Verfügung. Dagegen beziehen die passiven Transponder ihre gesamte benötigte Energie aus dem von dem Reader bereitgestellten elektrischen oder magnetischen Feld.

Ein Unterscheidungsmerkmal von *RFID*-Systemen ist die Betriebsart[4], hierbei lassen sich drei verschiedene Verfahren unterscheiden:

- *Vollduplex (FDX)*-Systeme:
  - Simultane Kommunikation via *DL*- und *UL*-Kanal
  - Feld des Reader muss für *UL*-Kommunikation aktiviert sein
  - *UL*-Kommunikation mittels einer harmonischen[5], subharmonischen[6] oder anharmonischen[7] Frequenz der Trägerfrequenz des Reader
- *Halbduplex (HDX)*-Systeme:
  - Zeitversetzte Kommunikation via *DL*- und *UL*-Kanal
  - Feld des Reader muss für *UL*-Kommunikation aktiviert sein
  - *UL*-Kommunikation mittels Lastmodulation oder Lastmodulation mit Hilfsträger der Trägerfrequenz des Reader
- *Sequentielle (SEQ)*-Systeme:
  - Zeitversetzte Kommunikation via *DL*- und *UL*-Kanal
  - Feld des Reader ist deaktiviert während *UL*-Kommunikation
  - Speicherung von Energie aus dem Feld des Reader für die *UL*-Kommunikation
  - *UL*-Kommunikation mittels Trägerfrequenz des Reader

Diese Verfahren unterscheiden sich folglich darin, wann Energie und Informationen übertragen werden (siehe Bild 2.2), sowie welche Verfahren für die *UL*-Kommunikation dienen.

[4] entnommen aus Finkenzeller (2008)
[5] vielfachen
[6] ganzteiligen
[7] unabhängigen

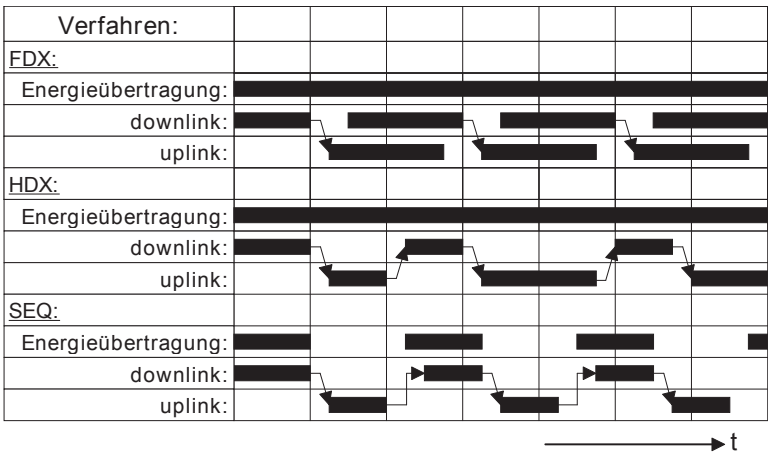


Bild 2.2: *Zeitliche Abläufe bei FDX- HDX- und SEQ Systemen, entnommen aus Finkenzeller (2008)*

2.3.1.1 Kopplungsarten

Für die drahtlose Übertragung von Energie und Informationen zwischen dem Reader und dem Transponder existieren verschiedene Arten zur Verkopplung[8]:

- kapazitive oder magnetische Kopplung
  - Anwendung in *Close coupling Systeme*n, mit Reichweiten von etwa 1 cm.
  - Einsatz beliebiger Frequenzen im Bereich von 0 . . . 30 MHz.
- induktive Kopplung
  - Verwendung in *Remote-coupling Systeme*n, bei typischen Reichweiten bis zu 1 m.
  - Betrieb auf den Frequenzen $\leq$ 135 kHz, 13,56 MHz sowie 27,125 MHz.
- elektrische Kopplung
  - Ebenfalls Verwendung in *Remote-coupling Systeme*en, siehe Induktive Kopplung.
- elektromagnetische Backscatter-Kopplung
  - Einsatz in *Long-range Systeme*n deren Reichweite größer 1 m ist.
  - Betrieb auf den *Mikrowellen (engl. Ultra High Frequency) (UHF)*-Frequenzen 868 MHz (Europa) bzw. 915 MHz (USA), sowie den Mikrowellenfrequenzen 2,5 GHz und 5,8 GHz.

[8] entnommen aus Finkenzeller (2008)

Aufgrund der benötigten Reichweite für das zu entwickelte System ist die kapazitive oder magnetische Kopplung nicht anwendbar. Ebenso ist die elektrische Kopplung nicht praktikabel, da für diese auf der Seite des Readers hohe Spannungen von einigen hundert Volt, an einer relativ großen elektrisch leitfähigen Fläche, als Koppelelement nötig sind. Die elektromagnetische Backscatter-Kopplung hat zwar eine hohe Reichweite, jedoch den Nachteil, dass die Transponder über eine eigene Energieversorgung verfügen müssen, da keine transformatorische Kopplung mehr vorhanden und somit keine ausreichende Energie übertragbar ist.

Für die geplante Anwendung ist lediglich die induktive Kopplung geeignet, da hiermit die erforderliche Distanz erreichbar ist und zugleich Energie für den Betrieb eines passiven Frontends zur Verfügung steht.
Die Funktionsweise eines induktiv gekoppelten Systems (siehe Bild 2.3) besteht darin, dass der Reader in seiner Antennenspule ein hochfrequentes elektromagnetisches Wechselfeld erzeugt, welches in die Antennenspule des Transponders zum geringen Teil einkoppelt und somit eine Spannung, die zur Informationsübertragung und Energieversorgung dient, induziert.

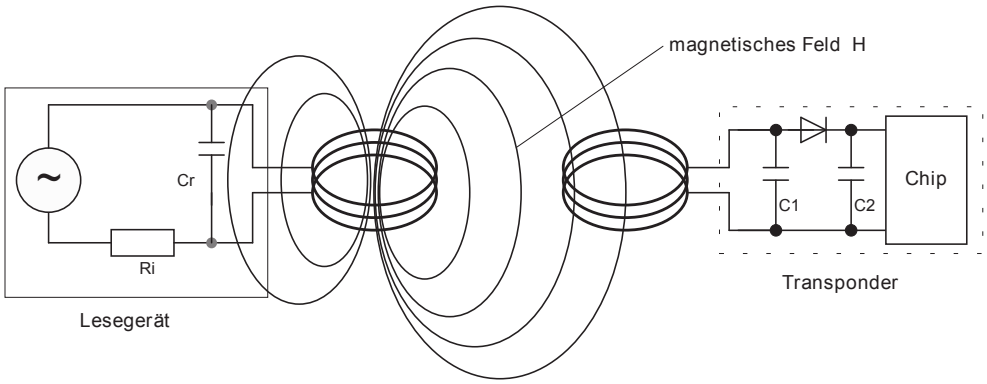


Bild 2.3: *Prinzipschaltung induktiv gekoppelter Transponder, entnommen aus Finkenzeller (2008)*

Aufgrund dessen, dass die Distanz zwischen den Antennenspulen des Readers und Transponders sehr viel kleiner ist, als die Wellenlängen der hierbei verwendeten Frequenzen (siehe Abschnitt 2.3.2.1), kann das elektromagnetische Wechselfeld als rein magnetisches Feld betrachtet werden. In dem Bild 2.4 ist beispielhaft aufgetragen, wie sich die magnetische Feldstärke H, über der Distanz d entlang der Spulenachse, bemessen zur Mitte einer Spule, verhält.

Dieser Feldstärkeverlauf für eine „kurze“ Zylinderspule, innerhalb des Nahfeldes (siehe Abschnitt 2.3.2.1), ist anhand folgender Formel 2.6, entnommen aus Finkenzeller (2008), berechnet worden. Dabei ist zu beachten, dass die verwendete Formel eine Vereinfachung darstellt und nur für Frequenzen von 135 kHz bis etwa in den *HF*-Bereich gültig ist. Als Parameter dienten hierbei die Daten der Readerspule

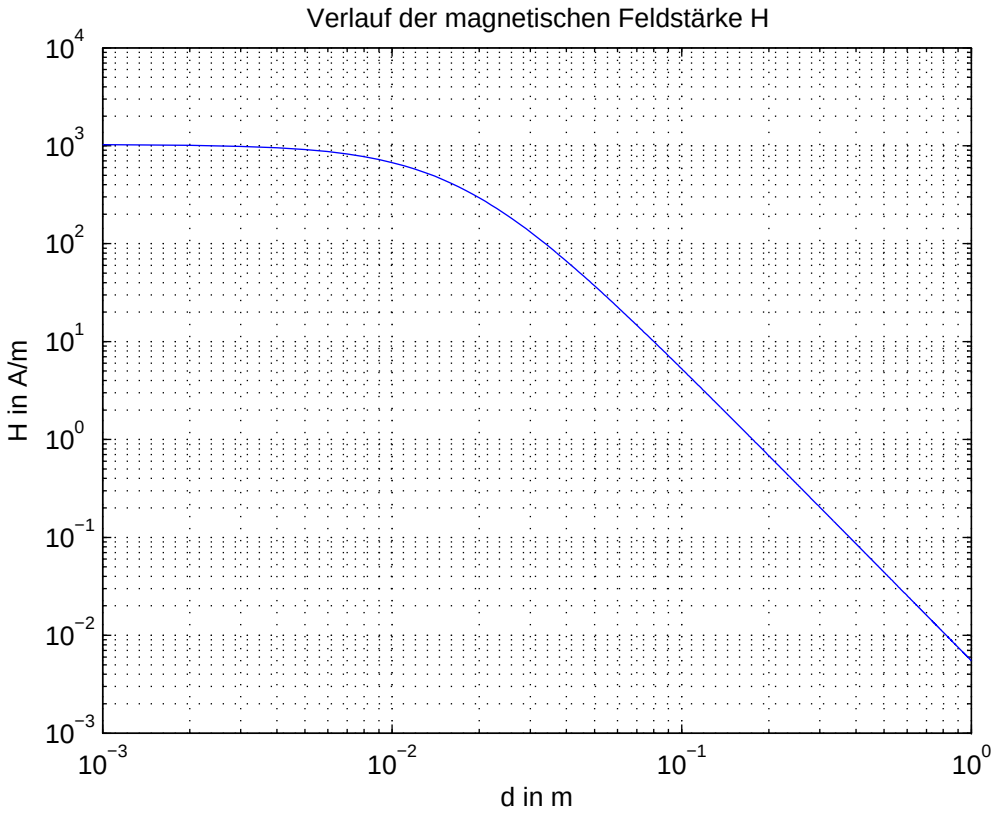


Bild 2.4: *Verlauf der magnetischen Feldstärke H abhängig von der Distanz d zur Readerspule*

vom Entwicklungskit aus Abschnitt 3.1.1.

$$H = \frac{I \cdot N \cdot R^2}{2\sqrt{\left(R^2 + d^2\right)^3}} \tag{2.6}$$

mit

$$\begin{aligned} I &= 500\,\text{mA} \quad \text{(Spulenstrom)} \\ N &= 72 \quad \text{(Windungszahl)} \\ R &= 17{,}5\,\text{mm} \quad \text{(Spulenradius)} \\ d &= \text{Distanz zur Spulenmitte} \end{aligned}$$

Wie man in dem Bild 2.4 erkennen kann, ändert sich die Feldstärke mit der Distanz zunächst nur gering, solange diese etwa gleich dem Spulenradius ist. Anschließend fällt die Feldstärke mit etwa 60 dB/Dekade ab.

Wie entscheidend hierbei die verwendeten Frequenzen sind, wird in dem Abschnitt 2.3.2.1 näher erläutert.

2.3.1.2 Kopplungsfaktor

Um zu beschreiben, welcher Anteil des von einer Spule emittierten gesamten magnetischen Fluss Φ eine weitere Spule durchsetzt und somit die beiden Spulen verkoppelt, dient zum einen die Gegeninduktivität M als quantitative Beschreibung, zum anderen qualitativ der Kopplungsfaktor k.

Die Gegeninduktivität M_{21} definiert, nach Finkenzeller (2008), das Verhältnis aus dem umfassten magnetischen Koppelfluss Ψ_{21} einer Spule 2 (Fläche A_2) zu dem Strom I_1 einer Spule 1 (Fläche A_1):

$$M_{21} = \frac{\Psi_{21}(I_1)}{I_1} = \int_{A_2} \frac{B_2(I_1)}{I_1} \cdot dA_2 \tag{2.7}$$

Alternativ lässt sich diese Größe, wie in Formel 2.7 gezeigt, anhand der magnetischen Flussdichte B_2, bezogen auf die Fläche A_2 der Spule 2, ermitteln.
Mit dieser Definition lässt sich ebenso aufgrund der Reziprozität der in einer Spule 1 gebildete Koppelfluss Ψ_{12} von dem Strom I_2 einer Spule 2 berechnen, es gilt folglich:

$$M = M_{12} = M_{21} \tag{2.8}$$

Der Kopplungsfaktor k ist hingegen nach Finkenzeller (2008) eine Größe, die unabhängig von der Geometrie der Spulen eine prozentuale Aussage über deren Verkopplung trifft:

$$k = \frac{M}{\sqrt{L_1 \cdot L_2}} \tag{2.9}$$

Für den Fall $k = 1$ tritt eine totale Verkopplung auf, wie sie beispielsweise in einem Transformator angewandt wird. Näherungsweise lässt sich der Kopplungsfaktor für zwei auf der Spulenachse angeordnete Spulen nach Gleichung 2.10 berechnen, unter der Einschränkung, dass der Radius r_{tp} der Transponderspule kleiner gleich dem Radius r_{rd} der Readerspule ist.

$$k \approx \frac{r_{tp}^2 \cdot r_{rd}^2}{\sqrt{r_{tp} \cdot r_{rd} \cdot \left(\sqrt{d^2 + r_{rd}^2}\right)^3}} \tag{2.10}$$

In dem Bild 2.5 ist die vorhandene Abhängigkeit des Kopplungsfaktors k von der Distanz d zur Mitte der Readerspule, deren Radius r_{rd} beträgt, dargestellt.

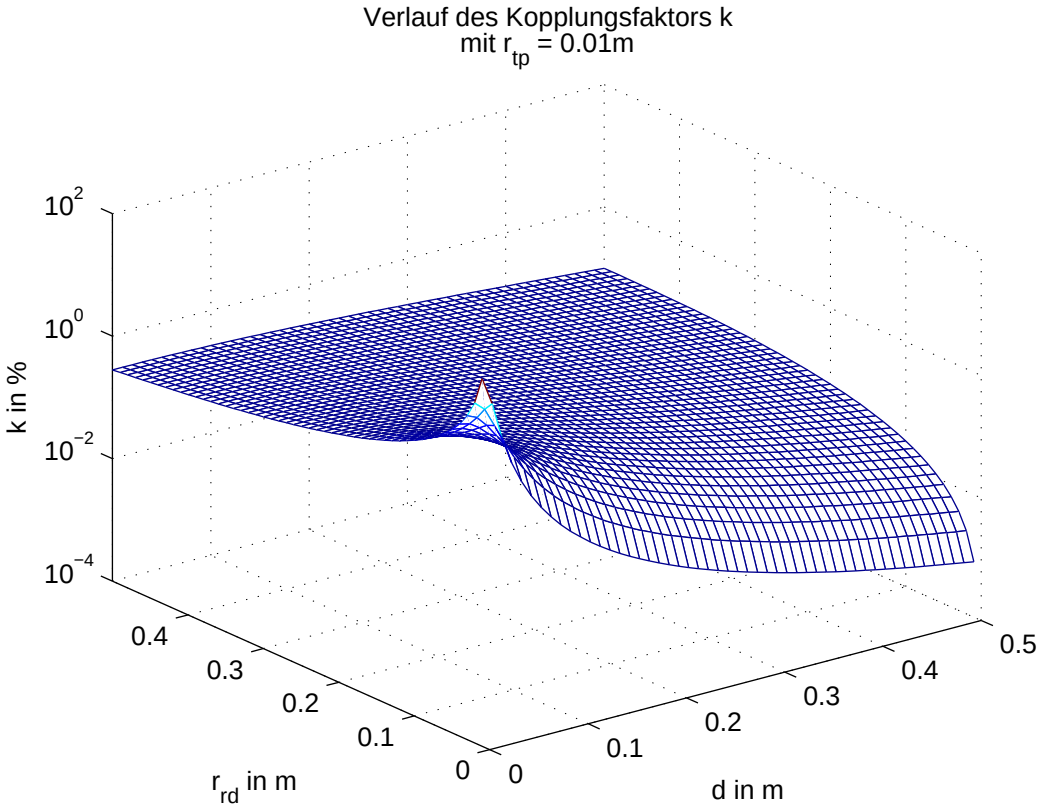


Bild 2.5: *Verlauf des Kopplungsfaktors k abhängig von der Distanz d und Radius r_{rd} der Readerspule*

Bei einem minimalen Radius r_{rd} der Readerspule gleich dem Radius r_{tp} der Transponderspule, sowie einer Distanz $d = 0$, ergibt sich ein Kopplungsfaktor von $k = 100\,\%$, das heißt beide Spulen werden von dem gleichem magnetischen Fluss Φ durchsetzt. Vergrößert man weiterhin bei der Distanz $d = 0$ den Radius r_{rd}, reduziert sich der Kopplungsfaktor k. Dies entspricht der Tatsache, dass eine geometrisch zu große Readerspule eine kleinere Transponderspule bei einer zu geringen Distanz d ebenfalls nicht durchfluten kann.

2.3.1.3 Resonanz

Sowohl die Antennenspule des Readers, als auch die des Transponders, wird in einem Schwingkreis betrieben. Dies bietet den Vorteil, dass der Wirkungsgrad verbessert und gleichzeitig eine gewisse Frequenzselektivität erreicht wird.

Dabei bildet die Readerspule L_1 in Reihenschaltung mit einer Kapazität C_1 eine Reihenresonanz, um eine Spannungsüberhöhung über der Readerspule L_1 zu erreichen. Für den Transponder wird hingegen eine Parallelresonanz aus der Transponderspule L_2 und einer Kapazität C_2 gebildet, dies dient für eine äußere Spannungsüberhöhung, um den Betrieb des Transponders zu erreichen.
Für einen idealen Schwingkreis berechnet sich die Resonanzfrequenz f_{res} wie folgt:

$$f_{res} = \frac{1}{2\pi\sqrt{L \cdot C}} \tag{2.11}$$

Als Maß für die Spannungs- und Stromüberhöhung in einem Schwingkreises steht die Güte Q. Für eine Reihenresonanz mit dem ohmschen Serienwiderstand R_1 für die Verluste in der Spule L_1 und im Kondensator C_1, berechnet diese sich wie folgt:

$$Q(f_{res}) = \frac{1}{R_1}\sqrt{\frac{L_1}{C_1}} \tag{2.12}$$

In einer Parallelresonanz hingegen berechnet sich die Güte Q mit dem ohmschen Verlustwiderstand R_{L2} der Spule L_2 entsprechend:

$$Q(f_{res}) = \frac{1}{R_{L2}\sqrt{\frac{C_2}{L_2}}} \tag{2.13}$$

Belastet man die Parallelresonanz, so wie es auf einem Transponder erfolgt, mit einem Parallelwiderstand R_L, ändert sich die Güte:

$$Q(f_{res}) = \frac{1}{R_{L2}\sqrt{\frac{C_2}{L_2}} + \frac{1}{R_L} \cdot \sqrt{\frac{L_2}{C_2}}} \tag{2.14}$$

Gleichzeitig bestimmt die Güte die Bandbreite B, der sich ergebenen Resonanzkurve, anhand der folgenden Formel 2.15:

$$B = \frac{f_{res}}{Q} \tag{2.15}$$

Für die Informationsübertragung ist diese Größe relevant, da sie Einfluss auf die nutzbaren Datenraten hat. Eine hohe Güte Q hat eine geringe Bandbreite B und auch eine geringe Datenrate zur Folge.

2.3.1.4 Ansprechfeldstärke

Ein wichtiges Kriterium für die Unterscheidung von Transpondern ist die Ansprechfeldstärke. Diese gibt an, wie groß die magnetische Feldstärke mindestens sein muss, um den Betrieb eines bestimmten Transponders zu gewährleisten.
Für die Berechnung der Ansprechfeldstärke muss zunächst die induzierte Spannung u_i bestimmt werden, welche von einer Transponderspule mit der Fläche A und einer Windungszahl N bei einer bestimmten effektiven magnetischen Feldstärke H_{eff} erzeugt wird, siehe Gleichung 2.16 (entnommen aus Finkenzeller (2008)).

$$u_i = \mu_0 \cdot A \cdot N \cdot \omega \cdot H_{eff} \tag{2.16}$$

Nimmt man nun an, der Transponder bestünde vereinfacht aus einer Transponderspule L_{res}, welche einen ohmschen Widerstand R_{res} aufweist, einer parallelen Kapazität C_{ges}, die alle parallelen Kapazitäten zusammenfasst und einem Lastwiderstand R_L. So lässt sich bei vorgegebener Spannung u_e am Ausgang des Parallelschwingkreises die Ansprechfeldstärke H_{min} nach Finkenzeller (2008), wie folgt, berechnen:

$$H_{min} = \frac{u_e \cdot \sqrt{\left(\frac{\omega L_{res}}{R_L} + \omega R_{res} C_{ges}\right)^2 + \left(1 - \omega^2 L_{res} C_{ges} + \frac{R_{res}}{R_L}\right)^2}}{\omega \cdot \mu_0 \cdot A \cdot N} \tag{2.17}$$

Die Ansprechfeldstärke H_{min} ist folglich unter anderem abhängig von der Antennenfläche A, der Windungszahl N und der Trägerfrequenz ω, sowie insbesondere mit dem übereinstimmen der Resonanzfrequenz mit der Trägerfrequenz.
Dadurch, dass die induzierte Spannung u_i abhängig von der Frequenz ω ist, lässt sich zeigen, dass mit steigender Frequenz ω die nötige Ansprechfeldstärke H_{min} sinkt. Dieser Effekt ist jedoch begrenzt, da bei höheren Frequenzen das Verhältnis der Induktivität zur Kapazität des Parallelschwingkreises sich verschlechtert.

Ist die Ansprechfeldstärke H_{min} für einen Transponder bekannt, so lässt sich auch die maximale Distanz zwischen diesem und einem bestimmten Reader bestimmen. Die als Energiereichweite bezeichnete Größe bestimmt sich aus dem Verlauf der ausgesendeten magnetischen Feldstärke des Readers, wie sie beispielsweise nach Gleichung 2.6 berechenbar ist.
Allerdings geht man bei den bisherigen Berechnungen stets davon aus, dass die Transponderspule parallel zur Readerspule auf deren Spulenachse angeordnet ist. Praktisch ist dies jedoch selten der Fall, die Transponderspule kann gegenüber der Readerspule verkippt und/oder verschoben sein. Zudem ist die Krümmung der magnetischen Feldlinien im Raum ortsabhängig, was, abhängig von der Lage der Transponderspule zu den Feldlinien, positive oder negative Auswirkungen hat.

Weiterhin gilt es bei der Ansprechfeldstärke zu unterscheiden zwischen der Feldstärke, die nötig ist für den Betrieb des Transponders selber und derer, die für eine fehlerfrei empfangende Rückantwort benötigt wird. Optimal wäre es, wenn sowohl der Betrieb als auch die Rückantwort die gleiche Feldstärke voraussetzen würden, in der Regel ist jedoch für die Rückantwort eine höhere Feldstärke erforderlich.

2.3.2 Frequenzbereiche

2.3.2.1 Frequenzbereiche für induktiv gekoppelte RFID-Systeme

Für die Auswahl des passenden Frequenzbereichs für ein induktiv gekoppeltes *RFID*-System gibt es verschiedene Punkte, die für die gewünschte Anwendung von Bedeutung sind.
Ein sehr wichtiger Punkt ist hierbei die zur Verfügung stehende magnetische Feldstärke. Sie bestimmt wie viel Energie überhaupt genutzt werden kann. Abhängig von der Wellenlänge λ der betrachteten Frequenz f verschiebt sich der Punkt, in dem das magnetische Feld von dem Nah- zum Fernfeld übergeht. Diese Distanz d lässt sich nach Finkenzeller (2008) näherungsweise angeben zu:

$$d = \frac{\lambda}{2\pi} \tag{2.18}$$

mit

$$\lambda = \frac{c_0}{f}$$

wobei

c_0 : Lichtgeschwindigkeit

Somit gilt also:

$$d < \frac{\lambda}{2\pi} \rightarrow \text{Nahfeld} \tag{2.19}$$

$$d \geq \frac{\lambda}{2\pi} \rightarrow \text{Fernfeld} \tag{2.20}$$

Innerhalb des Nahfeldes beträgt der Feldstärkeabfall zunächst 60 dB/Dekade, im Fernfeld dann 20 dB/Dekade. Außer dem reinen magnetischem Feld breitet sich zudem kontinuierlich das elektromagnetische Feld aus, welches ab dem Übergang zum Fernfeld vollständig ausgebildet ist und sich von der Antenne ablöst. Dies heißt aber auch, dass, wenn das elektromagnetische Feld vollständig ausgebildet ist, keine transformatorische (induktive) Kopplung mehr möglich ist.

Somit gibt die Grenze zum Fernfeld die absolute maximale Reichweite für induktiv gekoppelte Systeme an. Dabei gilt es zu beachten, dass in anderen Bereichen der Kommunikationstechnik abweichende Größenordnungen für den Übergang zum Fernfeld angegeben werden. Die folgende Tabelle 2.2 gibt für ausgewählte Frequenzen die Distanz d_f für den Übergang zum Fernfeld an.

Frequenz	Wellenlänge λ	Übergang Fernfeld d_f
135 kHz	2220,67 m	353 m
6,78 MHz	44,22 m	7,04 m
13,56 MHz	22,11 m	3,52 m
27,125 MHz	11,05 m	1,76 m
433 MHz	0,69 m	0,11 m

Tabelle 2.2: *Wellenlängen und deren Übergang zum Fernfeld für verschiedene Frequenzen*

Ein weiterer entscheidender Punkt bei der Auswahl des Frequenzbereichs für ein induktiv gekoppeltes *RFID*-System ist, dass auch *RFID*-Systeme rechtlich gesehen als Funkanlagen betrachtet werden und somit den üblichen Regulierungen unterliegen. So dürfen nur gewisse Frequenzbereiche mit unterschiedlicher Bandbreite und Sendeleistung genutzt werden. Neben den Frequenzbereich unter 135 kHz sind die sogenannten *Industrial Scientific and Medical (ISM)*-Frequenzen und die Frequenzen für *Short Range Devices (SRD)*-Anwendungen nutzbar. Die maximal erlaubte ausgesendete magnetische Feldstärke bezieht sich stets auf eine Entfernung von 10 m zum Sender. Abhängig von der verwendeten Wellenlänge hat dies zur Folge, da der Feldstärkeabfall innerhalb des Nahfeldes 60 dB/Dekade beträgt, dass die Feldstärke in der näheren Umgebung des Senders deutlich höher sein darf. Die folgenden ausgewählten Frequenzen[9] sind für die Anwendung nutzbar:

- <135 kHz:
  - Hohe magnetische Feldstärke verfügbar.
  - Sendeleistung bei 119 . . . 135 kHz maximal 66 dBμA/m.
  - Relativ große Induktivität für den Transponderschwingkreis erforderlich $\approx 1 \ldots 10$ mH.
  - Sowie ebenfalls große Kapazität nötig, damit schlecht integrierbar.
  - Ferritspulen für den Transponderschwingkreis bei Miniaturisierung nötig.
  - Niedrige Übertragungsraten.

[9] inkl. weiterer Angaben entnommen aus Finkenzeller (2008)

  - Kostengünstige Transponder wenn keine Ferritspulen erforderlich.
  - Einfache Realisierung von Frontends für Labormuster.
- 6,78 MHz:
  - Geringere magnetische Feldstärke verfügbar als bei 135 kHz.
  - Keine weltweite Nutzung möglich.
  - Sendeleistung allgemein $42\,\mathrm{dB}\mu\mathrm{A/m}$.
  - Induzierte Leistung bei gleicher Antennenfläche und magnetischer Feldstärke größer als bei 135 kHz.
  - Für kommerzielle *RFID*-Anwendungen nicht so sehr gebräuchlich.
- 13,56 MHz:
  - Verfügbar Magnetische Feldstärke geringer als bei 6,78 MHz.
  - Als *ISM*- und *SRD*-Frequenz weltweit verfügbar.
  - Sendeleistung $60\,\mathrm{dB}\mu\mathrm{A/m}$ für *RFID*-Anwendungen.
  - Kleine Spulen nötig $\approx 1 \ldots 10\,\mu\mathrm{H}$.
  - Kapazität für den Transponderschwingkreis integrierbar.
  - Frontend somit vollständig integrierbar.
  - Spule für den Transponderschwingkreis in gedruckter Form realisierbar.
  - Übertragungsraten höher als bei 135 kHz.
  - Induzierte Leistung bei gleicher Antennenfläche und magnetischer Feldstärke größer als bei 135 kHz.
- 27,125 MHz:
  - Verfügbar Magnetische Feldstärke geringer als bei 13,56 MHz.
  - Keine weltweite Nutzung möglich, aber als *ISM*-Frequenz ausgewiesen.
  - Sendeleistung allgemein $42\,\mathrm{dB}\mu\mathrm{A/m}$.
  - Induzierte Leistung bei gleicher Antennenfläche und magnetischer Feldstärke etwas kleiner als bei 13,56 MHz.
  - Gleiche positive Eigenschaften wie bei 13,56 MHz.
  - Für kommerzielle *RFID*-Anwendungen unüblich.

Für die geplante Anwendung sind die Frequenzbereiche kleiner als 135 kHz, sowie der Bereich von 13,56 MHz relevant, da hier neben der hohen Sendeleistung eine Vielzahl von *RFID*-Anwendungen bereits vorhanden und somit ein Zugriff auf kommerzielle Lösungen möglich ist.

2.3.2.2 Frequenzbereiche für elektromagnetisch gekoppelte Systeme

Bei Frequenzen oberhalb von einigen 10 MHz wird die Verwendung von induktiv gekoppelten Systemen für größere Reichweiten zunehmend ungünstiger, da die Grenze des notwendigen Nahfeldes schnell erreicht und somit keine ausreichende transformatorische Kopplung mehr möglich ist.
Angewendet wird vielmehr die elektromagnetische Kopplung. Diese ermöglicht zwar wesentlich größere Reichweiten als bei induktiver Kopplung, vermindert jedoch die Möglichkeit einer effizienten Energieübertragung und somit den Betrieb passiver Transponder. Es folgt eine Auflistung der hierfür zur Verfügung stehenden Frequenzbereiche, da sie für den bisher vorhandenen *UL*-Kanal des bestehenden Zellensensors verwendet werden.

- 433 MHz:
  - Als *ISM*-Frequenz weltweit verfügbar.
  - Belegt durch eine Vielzahl von *ISM*-Anwendungen wie „Keyless Entry"-Systeme im KFZ-Bereich oder Funkübertragung für z. B. Thermometer, Schalter und Handfunkgeräte.
- 868 MHz:
  - Frequenzbereich 868 … 870 MHz für *SRD*-Anwendungen, sowie auch *RFID*-Systeme mit geringer Sendeleistung zulässig.
  - Frequenzbereich 865 … 868 MHz speziell für *RFID*-Systeme mit größerer Sendeleistung.
- 2,45 GHz:
  - Weltweit als *ISM*-Frequenz nutzbar, *SRD*-Anwendungen dürfen höhere Sendeleistung haben.
  - Verwendung für *RFID*-Systeme, ebenso belegt durch WLAN und Telemetriesender sowie Amateurfunk- und Ortungsfunkdienst.
- 5,8 GHz:
  - Ebenso als *ISM*-Frequenz weltweit nutzbar, sowie für *SRD*-Anwendungen höherer Sendeleistung.

- Anwendungen wie Bewegungsmelder oder als *RFID*-System für die Erfassung von Mautgebühren.
- Überlappung mit Amateurfunk- und Ortungsfunkdienst.

2.3.3 Uplink-Kanal mittels Lastmodulation

Induktiv gekoppelte *RFID*-Systeme, deren Kommunikation nach der Betriebsart *FDX* oder *HDX* abläuft, nutzen für die Informationsübertragung auf dem *UL*-Kanal üblicherweise die sogenannte Lastmodulation. Dieses Verfahren wird, aufgrund der besonders einfachen und somit auch kostengünstigen Realisierung auf Seiten des Transponders, bei der Mehrzahl der Transponder angewendet.

Aufgrund der transformatorischen Kopplung kann ausgenutzt werden, dass sich eine veränderte Impedanz des Transponders auf den Stromfluss in der Spule des Readers mittels der Gegeninduktivität *M* auswirkt. Dabei unterscheidet man zwischen der ohmschen Lastmodulation und der kapazitiven Lastmodulation.
Bei ersterer verändert der Transponder durch Schalten eines Lastwiderstands im Takt des Modulationssignals seine Leistungsaufnahme und damit die Belastung der Readerspule.
Hingegen wird bei der kapazitiven Lastmodulation eine Parallelkapazität im Takt geschaltet, welche wiederum die Resonanzfrequenz des Parallelschwingkreises variiert und somit ebenfalls eine Last- und Phasenänderung in der Readerspule verursacht.

2.3.4 Verwendung sequentieller Verfahren

Eine andere alternative Betriebsart, deren Kommunikation nach dem *SEQ*-Verfahren abläuft (siehe Abschnitt 2.3.1), nutzt nicht die Lastmodulation für den *UL*-Kanal, sondern einen durch den Transponder eigens erzeugten modulierten Träger. Hierzu wird Energie aus dem vom Reader ausgesendeten Feld in einem Ladekondensator zwischengespeichert und für die Transmission genutzt.
Dieses Verfahren bietet den Vorteil eines höheren *Störabstand (SNR)* des am Reader empfangenen *UL*-Signals, sowie die Möglichkeit verschiedene Modulationsverfahren, wie z. B. Amplituden- oder Frequenzmodulation, anzuwenden. Zudem erfolgt bei den hier beschriebenen passiven Transpondern die Transmission für den *UL* auf der gleichen Trägerfrequenz wie für den *DL*, wodurch hierfür der gleiche Parallelschwingkreis mit der gleichen Transponderspule nutzbar ist.
Ein weiterer großer Vorteil gegenüber *FDX*- und *HDX*-Systemen ist die Tatsache, dass der Transponder erst dann aktiv werden muss, wenn der Ladekondensator

einen gewünschten Spannungswert erreicht hat. Somit steht für den Betrieb eine höhere Spannung zu Verfügung, da bei einem nahezu aufgeladenen Ladekondensator Spannungsanpassung vorhanden ist, gegenüber der meist benötigten Leistungsanpassung der beiden anderen Systeme. Folglich ergibt sich theoretisch eine reduzierte Ansprechfeldstärke verbundenen mit einer erhöhten Reichweite.

Die Problematik der *SEQ* Systeme ist jedoch die Tatsache, dass für den *UL* ein frequenzgebendes Bauteil erforderlich ist.

2.3.4.1 Trägerfrequenzerzeugung für den Uplink-Kanal

Zur Erzeugung der Trägerfrequenz sind verschiedene Methoden denkbar, diese wären:

- Ausnutzung des Parallelschwingkreises
- Erzeugung durch den Mikrocontroller (ohne Quarzoszillator)
- Quarzoszillator
- Alternativen zum Quarzoszillator

Die Ausnutzung des Parallelschwingkreises bietet den Vorteil, dass dieser bereits auf dem Transponder vorhanden und somit einer geringere Anzahl zusätzlicher Bauelemente notwendig ist. Allerdings sind die Bauelemente des Schwingkreises üblicherweise Fertigungstoleranzen und Umgebungseinflüssen ausgesetzt, was unweigerlich zu einer Verstimmung und folglich zu einer Ablage von der gewollten Trägerfrequenz führt. Um diesen Effekt zu reduzieren, kann man, einmalig bei der Produktion oder auch fortlaufend im Betrieb, einen Resonanzabgleich unter Verwendung von Trimmkondensatoren vornehmen. Jedoch ist auch dies nur bedingt möglich, da für einen präzisen Abgleich aufwendige Schaltungstechnik erforderlich ist.
Die Abhängigkeit der Resonanzfrequenz nach Gleichung 2.11 von den Bauelementwerten lässt sich anhand der Verstimmung v, wie folgt, ausdrücken:

$$v = \left(\frac{\omega_1}{\omega_0} - 1\right) \cdot 100\% \tag{2.21}$$

$$= \left(\sqrt{\frac{C_0}{C_0 + \Delta C_0}} - 1\right) \cdot 100\% \tag{2.22}$$

bzw.

$$v = \left(\sqrt{\frac{L_0}{L_0 + \Delta L_0}} - 1\right) \cdot 100\% \tag{2.23}$$

Vereinfacht lässt sich auch für eine prozentuale Abweichung ΔC bzw. ΔL die Verstimmung v angeben:

$$v = \left(\frac{1}{\sqrt{1 + \frac{\Delta C}{100\%}}} - 1 \right) \cdot 100\% \tag{2.24}$$

bzw.

$$v = \left(\frac{1}{\sqrt{1 + \frac{\Delta L}{100\%}}} - 1 \right) \cdot 100\% \tag{2.25}$$

So führt beispielsweise eine Änderung der Kapazität oder Spule von $-5\,\%$ zu einer Verstimmung v von $+2{,}6\,\%$. Eine gewünschte Resonanzfrequenz von 135 kHz würde sich so um $+3{,}5$ kHz ändern, eine Resonanzfrequenz von 13,56 MHz dagegen bereits um $+352$ kHz. Letzteres würde zu einer Verletzung der entsprechenden Funkzulassungsvorschrift (REC 70-03) führen, da bei 13,56 MHz nur eine Abweichung von ± 7 kHz zulässig ist.
Aus diesem Grund werden derartige Systeme, die den Schwingkreis als frequenzgebendes Bauteil nutzen, nur bei Trägerfrequenzen < 135 kHz angewendet.

Alternativ wäre ein Verfahren denkbar, welches mit einer *MCU* den nötigen Takt generiert. Hierzu könnte diese die Frequenz des *DL*-Signals als Vorgabe nutzen, da die *MCU* selber nur über einen relativ ungenauen RC-Oszillator als Taktgeber verfügen sollte.
Für Trägerfrequenzen von < 135 kHz wäre eine einfache von einem Zeitgeber gesteuerte Takterzeugung möglich, da die Taktfrequenz der *MCU* üblicherweise um ein Vielfaches größer ist.
Wenn sich jedoch die Trägerfrequenz in der Größenordnung der Taktfrequenz der *MCU* befindet, wie z. B. bei 13,56 MHz, ist dies so nicht mehr machbar. Man müsste vielmehr die Taktfrequenz der *MCU* exakt an die Trägerfrequenz des *DL*-Signals anpassen. So bieten beispielsweise einige *MCU* der MSP430 3xx bzw. 4xx Serie des Herstellers *Texas Instruments (TI)* die Möglichkeit mit einer internen *Frequency Locked Loop (FLL)* den internen Taktgeber an einen externen Takt anzupassen. Für den benötigten Frequenzbereich von 13,56 MHz oder größer, sind jedoch keine entsprechenden *MCU* verfügbar, vielmehr müssten Anpassungen verbunden mit zusätzlichem Schaltungsaufwand vorgenommen werden. Welche für den Takt zu erreichende Genauigkeit und Stabilität damit zu erreichen wäre, bedarf einer gezielten Untersuchung.

Das übliche Verfahren für die Erzeugung des Taktes für das Trägersignal des *DL* wäre mittels eines Quarzoszillators, da diese insbesondere für höhere Trägerfrequenzen die nötige Genauigkeit und Stabilität ermöglichen. Jedoch sind diese ver-

gleichsweise kostenintensiv und zudem nicht integrierbar, folglich nicht für die geplanten Anwendung in dem Zellensensor geeignet.

Daher bedarf es einer Alternative, welche die gestellten Anforderungen erfüllt.
Eine weitere Art von Oszillatoren sind diskret aufgebaute LC-Oszillatoren, wie z. B. der Meißner- oder Clapp- bzw. Colpitts-Oszillator, beschrieben in Seifart (2003) oder auch Hayward u. a. (2003). Diese haben jedoch ebenfalls die Forderung nach sehr präzisen Bauelementen, so dass es, insbesondere bei Temperaturänderungen, einer Kompensationsschaltung bedarf und somit ein diskreter Aufbau sehr aufwändig wäre.
Als Alternative zu Quarzoszillatoren existieren die MEMS[10]-Oszillatoren, welche in Multi-Chip-Module integrierbar sind. Diese bestehen aus einem mikromechanisch gefertigten Resonator aus Polysilizium, kombiniert mit einem separatem Die, der eine analogen Schaltung für die Anregung des Resonators, eine Ausgangsstufe und insbesondere eine Temperaturkompensation beinhaltet. Zwar haben diese eine hohe mechanischen Schockfestigkeit gegenüber herkömmlichen Quarzen, jedoch benötigt der Die des Resonators eine hermetische vakuumierte Kapselung. Mittlerweile ist diese Kapselung auch in CMOS-Herstellungsprozessen realisierbar. Beispielhaft sei hier der „SiT8208“ des Herstellers SiTime (siehe SiTime (2011)) aufzuführen, dessen Frequenzstabilität von ± 50 ppm, genügt beispielsweise der erforderlichen Stabilität von ± 514 ppm für eine Trägerfrequenz bei 13,56 MHz. Der Strombedarf des Oszillators ist zwar mit 32 mA bei 3,3 V relativ hoch, jedoch benötigt man den Takt nur für die *UL*-Kommunikation und könnte diesen anderenfalls deaktivieren.
Im Gegensatz zu den MEMS- und Quarz-Oszillatoren benötigten integrierte LC-Oszillatoren[11] keine hermetische Kapselung des Resonators (siehe Bild 2.6) und sind somit in Standard-CMOS-Prozessen kostengünstig realisierbar.

Primäres Element ist hierbei ein integrierter LC-Schwingkreis, der mit einer Frequenz von einigen Gigahertz arbeitet und mittels einer Kompensation gegen Temperatureinflüsse stabilisiert wird. Dessen Ausgangssignal wird entsprechend der gewünschten Vorgabe geteilt und durch einen Treiber ausgegeben. Beispielhaft sei hier der „Si500“ von dem Hersteller *Silicon Laboratories (SI)* neben den Oszillatoren des Herstellers *Integrated Device Technology (IDT)* aufzuführen, dessen Funktionsweise in dem Blockschaltbild 2.7 dargestellt ist.

Die erreichbare Frequenzstabilität, beispielsweise des „Si500“ vom Hersteller *SI* (siehe Silicon Laboratories (2011)), wird angegeben mit ± 250 ppm und genügt somit ebenfalls den Anforderungen. In Anbetracht der Integrierbarkeit gilt es diesen Typ von Oszillatoren bevorzugt zu betrachten.

[10] Mikro Elektro-Mechanisches System
[11] oftmals als Silizium-Oszillatoren beworben

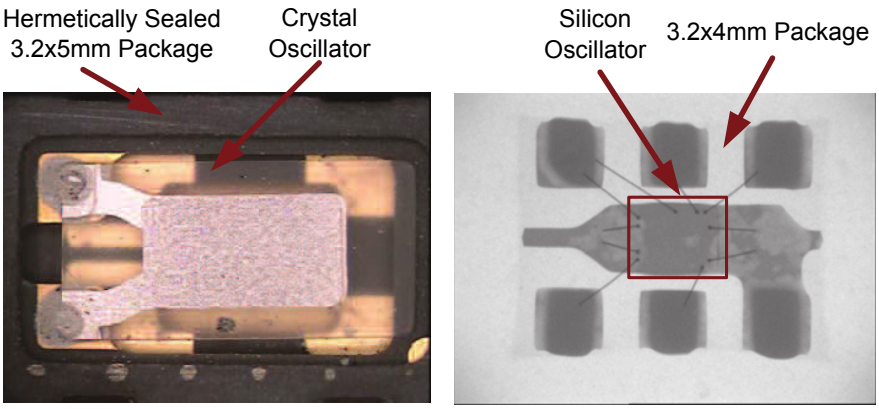


Bild 2.6: *Vergleich zwischen Quarzoszillator und integrierten „Si500“ LC-Oszillator, entnommen aus Silicon Laboratories (2008)*

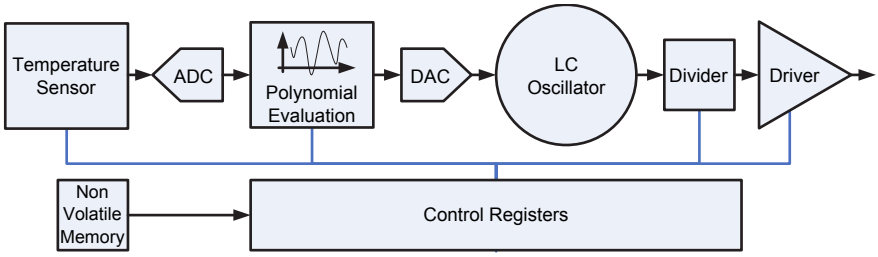


Bild 2.7: *Blockschaltbild des integrierten „Si500“ LC-Oszillators, entnommen aus Silicon Laboratories (2008)*

2.3.4.2 Uplink-Kanal mittels Frequenzmodulation

Eine Variante für ein *SEQ* System ist die Anwendung der Frequenzmodulation für den *UL*-Kanal, welcher mit der gleichen Trägerfrequenz wie für den *DL*-Kanal betrieben wird. Als frequenzgebendes Bauelement dient, für Systeme mit einer Trägerfrequenz kleiner 135 kHz, der Parallelschwingkreis mit der Transponderspule, wie es z. B. bei dem Transponder „TMS37157“ des Herstellers *TI* (siehe Texas Instruments (2009b)) erfolgt.
Durch die Anwendung der Frequenzmodulation kann ein solches System effizient betrieben werden, da die Modulation durch das Aufschalten einer Parallelkapazität realisierbar ist. Im Gegensatz zur Amplitudenmodulation lässt sich eine höhere Anzahl an Bits pro Transmission übertragen.
Eine genauere Behandlung des genannten Transponders „TMS37157“ erfolgt in dem späterem Abschnitt 3.1.1.

2.4 Aktive Transceiver Konzepte

Aufgrund der dem Zellensensor zur Verfügung stehende Energie, durch die zu überwachende Batteriezelle, ist neben den reinen passiven Transceiver Konzepten aus dem vorherigem Abschnitt 2.3 die Anwendung aktiver Transceiver Konzepte möglich - mit dem Vorteil eines erhöhten Wirkungsbereichs der drahtlosen Übertragung.
Da jedoch der mögliche Energiebedarf des Zellensensors (vgl. Abschnitt 2.2.3) stark limitiert ist, sind keine gebräuchlichen Konzepte, wie z. B. Bluetooth oder ZigBee, anwendbar. Diese verfügen zwar über verschiedene Techniken, um Energie einzusparen, jedoch basieren diese stets darauf, dass der *DL*-Kanal nur zeitweilig und nicht permanent verfügbar ist.
Aus diesem Grund gilt weiterhin die Verwendung passiver oder sehr energiesparender Empfängerkonzepte, wie zuvor im Abschnitt 2.3 vorgestellt. Für den Transmitter hingegen ist die Forderung bezüglich des Energiebedarfs sekundär, da dieser nur für den Moment einer Transmission auf dem *UL*-Kanal aktiv sein muss.

2.4.1 Gemeinsames Frequenzband für Up- und Downlink

Ein mögliches Konzept wäre die Abwandlung der bereits zuvor vorgestellten *SEQ* System aus dem Abschnitt 2.3.4, bei denen für den *UL*- und *DL*-Kanal das gleiche Frequenzband genutzt wird.

2.4.1.1 Sender mit Hilfsenergie

Bei den betrachteten *SEQ* Systemen wurde für die Transmission des *UL* der vorhandene Schwingkreis des *DL* mit der zuvor gespeicherten Energie aus dem Feld angeregt. Wenn jedoch die gespeicherte Energie nicht ausreicht, ist keine oder nur eine unvollständige Transmission möglich. Zudem beeinträchtigt der Aufladevorgang ebenfalls den *DL*-Kanal.
So ist es naheliegend, Energie aus der Batteriezelle für die Anregung des Schwingkreises zu verwenden. Hierfür ist jedoch eine Schaltung nötig, welche über den geforderten Spannungsbereich (siehe Abschnitt 2.2.1) die Anregung des Schwingkreises mit genügend hoher Leistung ausführt. Neben der Besonderheit, bei sehr niedrigen Spannungen von kleiner einem Volt die Funktion zu gewährleisten, bedarf es der unipolaren Anregung eines Parallelschwingkreises. Denn im Gegensatz zu einer Reihenresonanz, wie sie im Reader als Sender verwendet wird, stellt sich

bei einer spannungsgesteuerten Anregung der Parallelresonanz keine Spannungsüberhöhung über der Transponderspule ein. Gleichzeitig muss zudem die Funktion, sprich die Güte und die Lage der Resonanzfrequenz, des Schwingkreises für den *DL*-Empfänger erhalten bleiben.

Als Taktgeber für den Träger des *UL* könnte weiterhin für Trägerfrequenzen kleiner gleich 135 kHz der Schwingkreis ausgenutzt oder für höhere Frequenzen ein aktiver Taktgeber (siehe Abschnitt 2.3.4.1) verwendet werden.

2.4.1.2 Empfänger mit Hilfsenergie

Für Systeme, bei denen der passive *DL*-Empfänger den Wirkungsbereich der drahtlosen Übertragung unzulässig beschränkt, wäre die Verwendung eines von der Batteriezelle gestützten Vorverstärkers für den Empfängers denkbar.
Dieser Vorverstärker könnte derart dimensioniert werden, dass er nur von der unstabilisierten Zellenspannung versorgt wird und somit sehr wenig Energie benötigt. Konkretere Realisierungsaspekte siehe Abschnitt 3.3.1.3.

2.4.2 Separate Frequenzbänder für Up- und Downlink

Alternativ können zu den *SEQ* Systemen, die für den *UL* und *DL* einen gemeinsamen Frequenzbereich und damit auch einen gemeinsamen Antennen-Schwingkreis nutzen, ebenso getrennte Frequenzbereiche für diese Systeme verwendet werden.
Der Vorteil ist, dass für beide Kanäle separate Empfangs- und Sendeschaltungen zur Verfügung stehen, die einerseits eine *FDX*-Kommunikation erlauben und andererseits entsprechend der Anwendung optimierbar sind.
Nachteilig ist hingegen der zusätzliche Bedarf nach einer zweiten Antenne, welche möglicherweise miteinander kombinierbar werden können.

Aufgrund der permanent nötigen Empfangsbereitschaft des *DL*-Empfängers, ist es weiterhin wegen der Limitierung der Energieaufnahme sinnvoll, passive bzw. mit Hilfsenergie betriebene Empfängerkonzepte anzuwenden. Wegen der Trennung benötigen diese nunmehr keine gleichzeitige Anpassung der Eigenschaften als *UL*-Sender.

Mittels der Separation können für den *UL*-Sender vollkommen getrennte Konzepte verfolgt werden. Hierdurch ist eine vorteilhafte Anwendung kommerziell verfügbarer Lösungen möglich, was dem gelegten Schwerpunkt der Arbeit entspricht. Wegen der Vorgabe, den Zellensensor möglichst ohne einen Quarz zu realisieren, ist die Auswahl an Lösungen beschränkt.
Möglich wäre es zum einen, herkömmliche integrierte Transmitter einzusetzen

und den üblichen Quarz als Taktgeber durch beispielsweise einen integrierten LC-Oszillator zu ersetzen. Zum anderen die Verwendung von Transmittern mit einem bereits integrierten LC-Oszillator. Letztere Lösung wird später in dem Abschnitt 3.3.2 angewendet.

2.5 Analyse kommerzieller Lösungen

In den vorherigen beiden Abschnitten 2.3 und 2.4 wurden verschiedene Konzepte für passive und aktive Transceiver vorgestellt, welche relevant für die Konzeption des Zellensensors sein könnten.

Anhand dieser betrachteten Konzepte und den vorhandenen Rahmenbedingungen für die Realisierung, sind diverse kommerziell verfügbare integrierte Transponder *Front-End*s recherchiert worden.
Dies dient dazu einen Überblick zu erhalten, inwieweit auf kommerzielle Lösungen für die Realisierung zurückgreifbar, oder gegebenenfalls eine eigene Lösung notwendig ist.

2.5.1 Integrierte Transponder Front-End

Zunächst wurden integrierte Transponder *Front-End*s recherchiert, welche nach Möglichkeit über kein, beziehungsweise ein einfaches *Back-End* verfügen und somit keine *MCU* enthalten.
Die folgende Auflistung beinhaltet die Ergebnisse der Recherche verbunden mit einer kurzen Bewertung:

- U3280M von Atmel:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End:
    - *Electrically Erasable Programmable Read-Only Memory (EEPROM)*-Speicher
    - *Inter-Integrated Circuit (I^2C)*-Schnittstelle
  - Front-End:
    - Ein- und Ausgabe für Modulation und Demodulation
    - Ausgabe des *Radio Frequenz (RF)*-Taktes der Trägerfrequenz
    - Bereitstellung einer Spannungsversorgung
  - Frequenzbereich: *DL* & *UL* auf 125 kHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: bedingt, da rein passiver *UL*

- AT24RF08C von Atmel:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End:
    - *EEPROM*-Speicher
    - I^2C-Schnittstelle
  - Front-End: kein Zugang
  - Frequenzbereich: *DL* & *UL* auf 125 kHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: nein, Zugriff auf *Front-End* nur über *EEPROM* möglich
- IPMS_RFE125 des Fraunhofer-Institut Photonische Mikrosysteme:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End: nicht vorhanden
  - Front-End:
    - Ein- und Ausgabe für Modulation und Demodulation
    - Ausgabe des *RF*-Taktes der Trägerfrequenz
    - Bereitstellung einer Spannungsversorgung
  - Frequenzbereich: *DL* & *UL* auf 135 kHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: bedingt, da rein passiver *UL*
- MLX90129 von Melexis:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End:
    - Datenlogger mit A/D Interface für externe Sensoren
    - *EEPROM*-Speicher
    - *Serial Peripheral Interface (SPI)*-Schnittstelle
  - Front-End:
    - Bereitstellung einer Spannungsversorgung
  - Frequenzbereich: *DL* & *UL* auf 13,56 MHz

  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: nein, da kein direkter Zugriff auf *Front-End* möglich

- M24LR64-R von *STMicroelectronics (ST)*:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End:
    - *EEPROM*-Speicher
    - I^2C-Schnittstelle
  - Front-End: kein Zugang
  - Frequenzbereich: *DL* & *UL* auf 13,56 MHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: nur für Voruntersuchungen brauchbar, da Zugriff auf *Front-End* nur über *EEPROM* möglich

- TMS37157 von *TI*:
  - *SEQ* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Frequenzmodulation
  - Back-End:
    - *EEPROM*-Speicher
    - *SPI*-Schnittstelle
  - Front-End:
    - Informationsübertragung via *SPI*-Schnittstelle
    - Ausgabe des *RF*-Taktes der Trägerfrequenz
    - Bereitstellung einer Spannungsversorgung
  - Frequenzbereich: *DL* & *UL* auf 135 kHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: aufgrund der *SEQ* Betriebsart möglich, wenn auch *UL* rein passiv

- CC11x1-Q1 von *TI*:
  - Vollständiger Transceiver, steuerbar mittels *SPI*-Schnittstelle
  - *HDX* Betriebsart, *DL* & *UL* via Amplituden- oder Frequenzmodulation
  - Frequenzbereich: *DL DL* & *UL* auf 433/868 MHz

- Energieversorgung: aktiver Betrieb bei $1,8 \ldots 3{,}3\,\mathrm{V}$
  - Strombedarf bei Empfangsbereitschaft $\approx 18\,\mathrm{mA}$
  - Strombedarf bei Transmission $13 \ldots 35\,\mathrm{mA}$

$\rightarrow$ Eignung: ungeeignet

Keines der zurzeit bekannten kommerziellen *Front-End*s lässt einen aktiven oder mit Hilfsenergie gestützten Betrieb zu, wie er für die geplante Anwendung möglich wäre. Somit ist bei allen gefundenen Lösungen mit einer Problematik bezüglich des Wirkungsbereichs zu rechnen.

Von den aufgelisteten Bauelementen entspricht das *Front-End* „TMS37157“ vom Hersteller *TI* am ehesten den Anforderungen. Dessen Kommunikation läuft nach der *SEQ* Betriebsart mit Frequenzmodulation für den *UL* ab und bietet damit im Vergleich zu den übrigen Bauelementen, welche allesamt die Lastmodulation nutzen, den wohl größten Wirkungsbereich. Zudem ist ein direkter Zugriff auf das *Front-End* mittels der *SPI*-Schnittstelle möglich. Aufgrund der möglichen Eignung für die Realisierung und für die weitere Konzeptfindung, wird dieses *Front-End* in dem Abschnitt 3.1 näher untersucht.
Als weiteres Bauelement ist das „M24LR64-R“ des Herstellers *ST* von Relevanz, es eignet sich zwar nicht für die konkrete Realisierung des Zellensensors, jedoch ist es aufgrund des höheren verwendeten Frequenzbereichs von 13,56 MHz für die Konzeption von belang. Dieses Bauelement wird ebenso in dem Abschnitt 3.1 für Voruntersuchungen verwendet.

Für eine beispielhafte Abschätzung, welcher Energiebedarf hingegen für einen aktiven Empfänger nötig wäre, dient der Transceiver „CC11x1-Q1“ vom Hersteller *TI*. Obwohl dieser bereits als sehr energiesparend beworben wird, sind für den empfangsbereiten Betrieb $\approx 18\,\mathrm{mA}$ erforderlich und ist somit vollkommen ungeeignet bezüglich der gestellten Anforderungen aus Abschnitt 2.2.3.

2.5.2 Integrierte Transponder Front-End mit Mikrocontroller

Weiterhin gibt es, im Gegensatz zu den vorherig aufgelisteten integrierten *Front-Ends*, welche mit bereits integrierter *MCU*. Da der Zellensensor einen solchen benötigt, wäre dies für die Anwendung von Vorteil.
Folgende relevante Bauelemente wurden hierzu recherchiert:

- ATA6286 von Atmel:
  - *FDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Amplituden- oder Frequenzmodulation
  - Back-End:
    - vollständiger 8-Bit RISC Mikrocontroller
    - *SPI*-Schnittstelle
    - Sensor-Interface
  - Front-End:
    - *Langwellen (engl. Low Frequency) (LF)*-Empfänger mit Wake-up Funktion und Manchester Dekodierung
    - *UHF*-Sender
  - Frequenzbereich: *DL* 125 kHz und *UL* 433 kHz
  - Energieversorgung:
    - *Front-End* aktiver Betrieb
    - *Back-End* aktiver Betrieb

  → Eignung: *MCU* ungeeignet
- U9280M-H von Atmel:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation
  - Back-End:
    - 4-Bit Mikrocontroller
    - I^2C-Schnittstelle
  - Front-End: U3280M, siehe Abschnitt 2.5.1
  - Frequenzbereich: *DL DL* & *UL* auf 125 kHz
  - Energieversorgung:
    - *Front-End* passiver Betrieb
    - *Back-End* passiver oder aktiver Betrieb

  → Eignung: *MCU* ungeeignet, *Front-End* nur passiv betreibbar
- IPMS_ST1 des Fraunhofer-Institut Photonische Mikrosysteme:
  - *HDX* Betriebsart, *DL* via Amplitudenmodulation und *UL* via Lastmodulation

- Back-End:
  - 16-Bit RISC Mikrocontroller
  - *Analog Digital Konverter (ADC)* mit vier Eingängen
  - I^2C-Schnittstelle
- Front-End: IPMS_RFE125, siehe Abschnitt 2.5.1
- Frequenzbereich: *DL DL* & *UL* auf 125 kHz
- Energieversorgung:
  - *Front-End* passiver Betrieb
  - *Back-End* passiver oder aktiver Betrieb

→ Eignung: ungeeignet da *Front-End* nur passiv betreibbar

Alle drei recherchierten Bauelemente sind für die Anwendung nicht geeignet. Der „ATA6286“ von dem Herstellers Atmel verfügt zwar über einen aktiven *UL*-Sender, jedoch ist die *MCU* recht unflexibel einsetzbar, da diese offensichtlich sehr für die Anwendung in einem Reifenluftdrucksensor optimiert ist.

3 Konzept- und Schaltungsentwurf

In dem vorherigen Kapitel 2 sind neben den Anforderungen an den zu entwickelnden Zellensensor verschiedene Konzepte für passive und aktive Transceiver mitsamt einigen kommerziellen Lösungen behandelt worden.
Anhand dieser Vorgaben und Konzepte, sowie zwei Voruntersuchungen (siehe Abschnitt 3.1), gilt es ein konkretes Konzept, verbunden mit dem Entwurf der benötigten Schaltungen für den Zellensensor, zu entwickeln.

3.1 Voruntersuchungen zur Konzeptfindung

Für die spätere Konzeption ist die Eignung von kommerziellen Lösungen für das *Front-End* des Zellensensors von fundamentaler Bedeutung. Leider zeigte sich jedoch (siehe Abschnitt 2.5), dass nur das *LF-Front-End* „TMS37157“ des Herstellers *TI* näherungsweise geeignet ist. Neben diesem wird noch das *HF-Front-End* „M24LR64-R“ vom Hersteller *ST* in den folgenden beiden Abschnitten weiter untersucht.

3.1.1 LF Transponder

Für die Untersuchung des 135 kHz *LF* Transponders *Front-End* „TMS37157“ von *TI* diente das von *TI* verfügbare Entwicklungskit „eZ430-TMS37157“ (siehe Texas Instruments (2009a)). Dieses besteht aus einem Transponder, welcher das besagte *Front-End* mit einer *MCU* enthält und einem Reader, welcher mittels einer Demonstrationssoftware, gesteuert werden kann.

Die in dem *Front-End* enthaltenen Komponenten sind in dem Bild 3.1 dargestellt. Neben dem eigentlichen analogem *Front-End* ist eine Kontrolleinheit[1] zur Steuerung, Speicher und eine Einheit für das Energiemanagement[2] enthalten.

[1] engl. Control Unit
[2] engl. Power Management

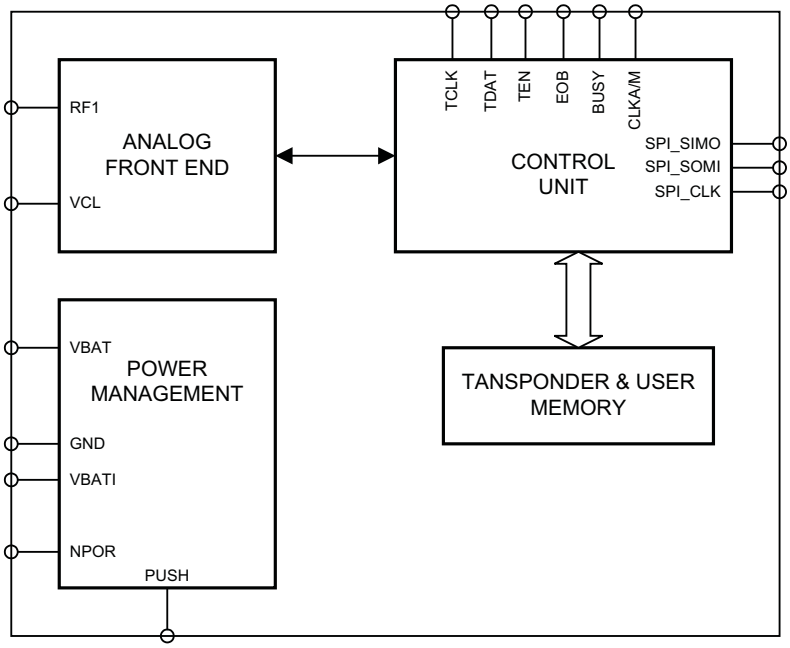


Bild 3.1: *Blockschaltbild zur Systemübersicht des „TMS37157“ von TI, entnommen aus Texas Instruments (2009b)*

Ein genaueres Prinzipschaltbild, siehe Bild 3.2, zeigt den Aufbau des Energiemanagements. Wenn sich der Transponder innerhalb des aktiven *RF*-Feldes des Readers befindet, lädt dieses zunächst den Kondensator C_L auf und erzeugt so die Spannung V_{CL} (vgl. Bild 2.2). Erreicht diese die Schwellenspannung von 5,75 V, wird die Kontrolleinheit aktiv und ist somit bereit für den Empfang auf dem *DL*-Kanal. Die Kontrolleinheit bezieht hierzu ihre Energie entweder direkt aus dem *RF*-Feld, oder optional von einer Batterie, angeschlossen an den Punkt V_{BAT}. Nebenbei schaltet diese den Ausgang V_{BATI} frei, für die Spannungsversorgung einer angeschlossenen *MCU* aus der Batterie. Für eine Versorgung direkt aus dem *RF*-Feld müsste dieser hingegen mit dem Anschluss V_{BAT} verbunden sein. Anschließend, nach dem die Kontrolleinheit ein Abschalten des *RF*-Feldes detektiert hat, startet die Transmission auf dem *UL*-Kanal via Frequenzmodulation. Als frequenzgebendes Element dient hierzu der Transponderschwingkreis, welcher automatisch abgeglichen werden kann.

Die Transponderspule des Entwicklungskits ist eine zylindrische SMD-Ferritspule mit 2,66 mH des Herstellers Neosid[3]. Durch den Ferritkern kann trotz der relativ großen Induktivität eine kleine Bauform, von etwa 11,6 mm × 3,6 mm bei einer Höhe von 2,5 mm, erreicht werden. Nachteilig ist jedoch der damit verbundene hohe Kostenfaktor.

[3] Herst. Bez. : MS32ka/2.66mH

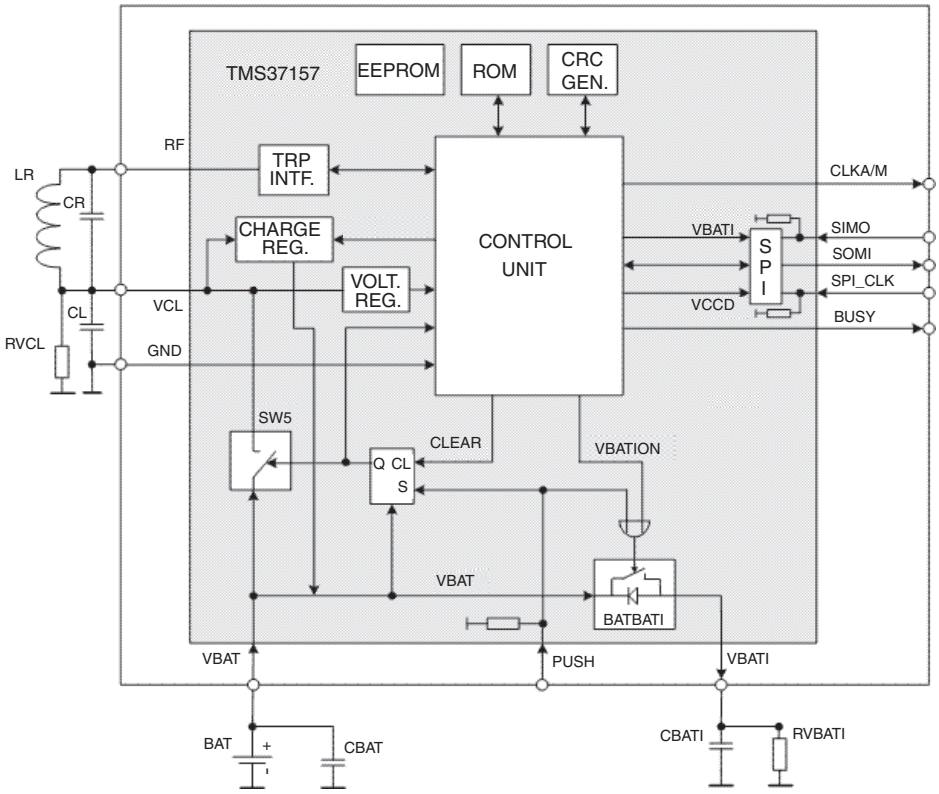


Bild 3.2: *Prinzipschaltbild für das Energiemanagement des „TMS37157“ von TI, entnommen aus Texas Instruments (2009b)*

Als Readerspule dient hingegen eine kreisförmige Luftspule mit 445 μH und einem Durchmesser von etwa 35 mm.

Die mit dem unmodifizierten Entwicklungskit erreichbare Distanz zwischen Reader- und Transponderspule, bei einer Ausrichtung entsprechend der Spulenachse und eine erfolgreiche Übertragung mit gültiger Rückantwort möglich ist, konnte experimentell auf ≈ 7 cm ermittelt werden. Diese Reichweite ist, trotz der Verwendung des batteriegestützten Betriebs, für die geplante Anwendung viel zu gering.
Um eine größere Distanz zu erhalten, wurde zunächst die Güte des Schwingkreises von dem Reader durch den Einsatz einer Ferritspule (siehe Bild 3.3) erhöht. Zusätzlich erfolgte ein Austausch der Serienwiderstände von 4,7 Ω auf 1,0 Ω am Ausgang des Treibers für die Spule. Somit konnte die Distanz für eine erfolgreiche Kommunikation auf ≈ 13 cm erhöht werden, welche aber noch zu gering ist.
Als weitere Modifikation konnte die SMD-Ferritspule des Transponders durch eine selbstgewickelte Luftspule gleicher Induktivität ersetzt werden. Die damit erreichbare Distanz betrug damit zwar ≈ 22 cm, jedoch hatte dies eine geometrisch sehr große Transponderspule (≈ 182 Wdg·) zur Folge, welche nicht für die Realisierung

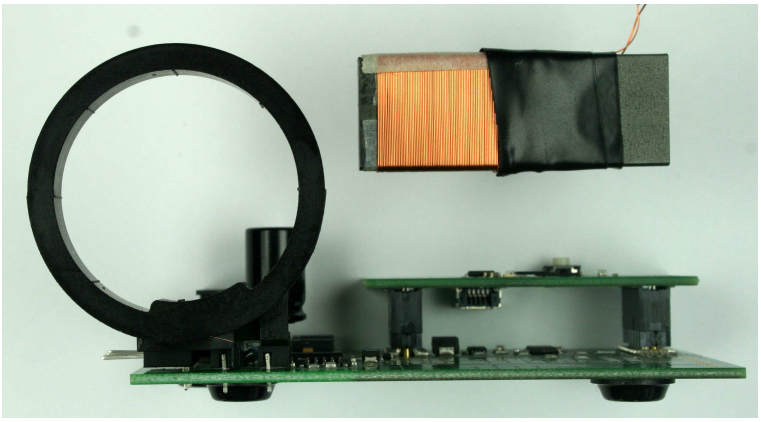


Bild 3.3: *Reader des Entwicklungskits eZ430-TMS37157 von TI, mit Luftspule (links) und Ferritspule (rechts)*

geeignet ist.
Weiterhin wurde mit der Energieversorgung des *Front-End* vom Transponder experimentiert, da einerseits offensichtlich für die Transmission des *UL* nicht die Energie aus der Batterie des Transponder genutzt wird, sondern die von dem *RF*-Feld empfangene und in dem Kondensator C_L gespeicherte Energie. Andererseits muss die Spannung V_{CL} über dem Kondensator C_L für die Aktivierung des *Front-End* erst den nötigen Schwellwert erreichen, was aber bei größerer Distanz zunehmend länger dauert bzw. nicht möglich ist.
So liegt es nahe, die Spannung V_{CL} mittels einer äußeren Spannungsquelle so zu stützen, damit das *Front-End* schneller aktiviert wird und gleichzeitig mehr Energie für die Rückantwort zur Verfügung steht.
Jedoch zeigte sich, dass das Energiemanagement des *Front-End* stets nach einer durchgeführten Transmission, als auch in regelmäßigen Zeitabständen, automatisch den Kondensator C_L zu entladen versucht. Dieses Verhalten führt zudem zu einer Störung des Antennensignals, so dass abhängig von der Intensität der äußeren Energiezufuhr, keine erfolgreiche Kommunikation möglich ist.

Aufgrund des zu geringen experimentell festgestellten Wirkungsbereichs der drahtlosen Kommunikation ist die Verwendung des *Front-End* „TMS37157“ von *TI* für die Realisierung des Zellensensors nicht geeignet.
Des weiteren hat die Verwendung von Trägerfrequenzen in dem Bereich von 135 kHz den Nachteil, dass die benötigten Transponderspulen einen hohen Induktivitätswert von 2,66 mH benötigen. Diese Spulen sind nur als Ferritspulen in kleiner Bauform herstellbar und lassen sich nicht als gedruckte Spulen auf Leiterplatten realisieren.
Zudem ist die Energieübertragung zum Transponder, im Vergleich zu einer höheren Trägerfrequenz von 13,56 MHz, geringer, wie in dem folgendem Abschnitt zur Untersuchung eines *HF-Front-End* gezeigt wird.

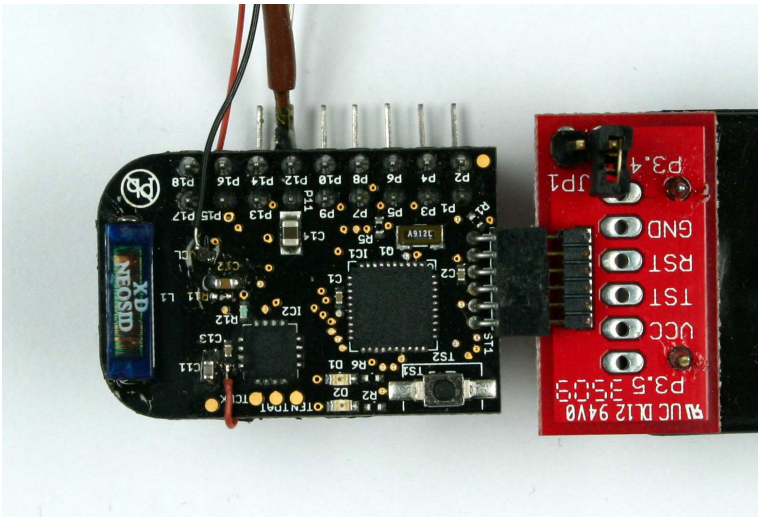


Bild 3.4: *Modifizierter Transponder des Entwicklungskits „eZ430-TMS37157“ von TI, mit angestecktem Batteriemodul*

3.1.2 HF Transponder

Weiterhin wird ein 13,56 MHz *HF* Transponder *Front-End* des Typs „M24LR64-R“ von *ST* näher untersucht, wofür das Entwicklungskit „STARTKIT-M24LR-A“ von *ST* (siehe STMicroelectronics (2010b)) zu Verfügung steht.
Neben einem ebenfalls von einer Demonstrationssoftware steuerbaren Reader, sind mehrere Transponder verschiedener Bauform in dem Entwicklungskit enthalten. Zwei der Transponder sind auf Leiterplatten mit gedruckten Transponderspulen aufgebracht, ein weiterer auf einer Leiterplatte mit einer Ferritspule.
Der Transponder „M24LR64-R“ von *ST* (siehe Bild 3.5) beinhaltet, neben dem *HF-Front-End*, eine Kontrolleinheit mit angeschlossener I^2C-Schnittstelle, ein Energiemanagement, sowie einen *EEPROM*-Speicher. Auf den *EEPROM*-Speicher kann sowohl mittels der *RF*- als auch mit der I^2C-Schnittstelle zugegriffen werden. Ein direkter Zugriff von der I^2C- auf die *RF*-Schnittstelle, oder andersherum, ist hingegen nicht möglich.

Die gedruckten rechteckigen Transponderspulen, mit den Abmessungen 48×75 mm bzw. 20×40 mm, weisen eine Induktivität von etwa 4,2 μH auf. Die erforderliche Parallelkapazität für den Transponderschwingkreis von etwa 32,8 pF, ist bei allen drei Transpondern durch die interne Abgleichkapazität, sowie die externen parasitären Kapazitäten, realisiert.

Für einen konkreten Vergleich der benötigten Ansprechfeldstärke für ein System mit einer Trägerfrequenz von 135 kHz, mit einem System der Frequenz 13,56 MHz, wird bei beiden Systemen die gleiche Spulenfläche mit den Maßen 48×75 mm von

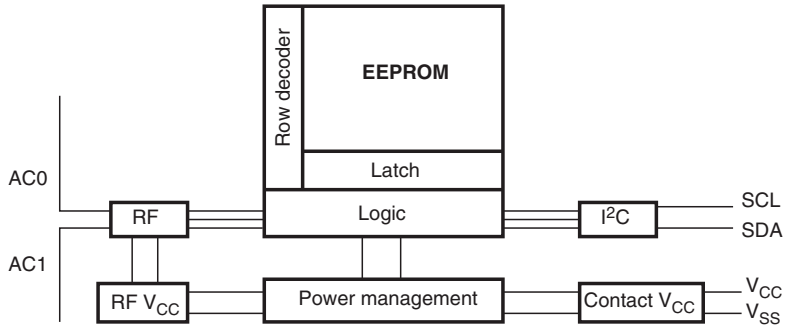


Bild 3.5: *Blockschaltbild zur Systemübersicht des „M24LR64-R“ von ST, entnommen aus STMicroelectronics (2010a)*

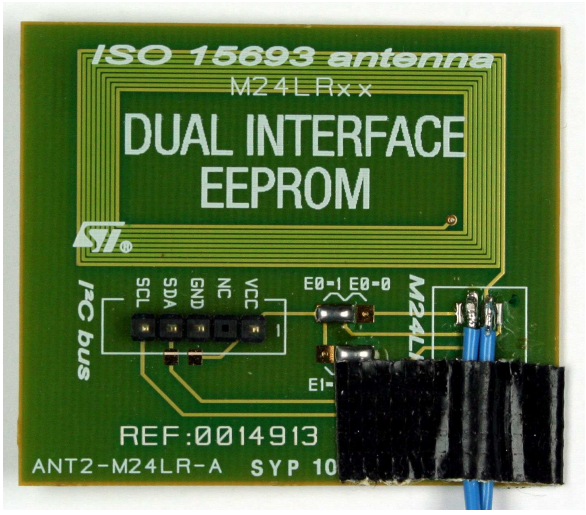


Bild 3.6: *Modifizierte Leiterplatte mit Antennenspule (20 × 40 mm) eines Transponders aus dem Entwicklungskit „STARTKIT-M24LR-A“ von ST*

einem Transponder des Entwicklungskits angenommen. Nach der Gleichung 2.17 ergeben sich, unter Verwendung entsprechender Parameter anhand der beiden betrachteten Entwicklungskits, folgende beispielhafte Werte:

$$H_{min,135kHz} = 0{,}2292\,\mathrm{A/m} \tag{3.1}$$

$$H_{min,13.56MHz} = 0{,}0255\,\mathrm{A/m} \tag{3.2}$$

Die benötigte Feldstärke für ein 13,56 MHz System ist hiernach um etwa den Faktor neun geringer. Folglich ergibt sich, unter Annahme des gleichen Feldstärkeverlaufs, eine erhöhte Reichweite des Systems.

Die mit dem Entwicklungskit zu erzielende Reichweite eines auf der Spulenachse angeordneten Transponders betrug etwa 7,5 cm bei Verwendung der 20 × 40 mm

großen Transponderspule. Mit der $48 \times 75\,\text{mm}$ großen Transponderspule waren hingegen etwa 14 mm zu erzielen.
Durch das Anlegen einer äußeren Betriebsspannung konnte keine Erhöhung der Reichweite erzielt werden, denn diese dient nur der Versorgung der Kontrolleinheit des *Front-End* bei der Anwendung der I^2C-Kommunikation.
An dem Reader des Entwicklungskits sind keine Modifikationen durchgeführt worden, da dieser über einen vollständig integriertes Reader-IC verfügt, woran keine einfachen Änderung vorzunehmen sind. Die Sendeleistung des als „Short Range“ deklarierten Readers beträgt etwa 200 mW.

Gegenüber dem zuvor untersuchtem *LF*-System, ergab sich mit dem *HF*-System nur ein geringfügig vergrößerter Wirkungsbereich. Jedoch sind die Reader der beiden Systeme nicht vergleichbar, der des *LF*-Systems sendet mit ungefähr 4 W, hingegen der des *HF*-Systems nur mit etwa 200 mW.
Folglich ist das *HF*-System bezüglich der Reichweite offensichtlich besser geeignet. Weiterhin hat dieses den erheblichen Vorteil, der einfach zu realisierenden und geometrisch kleinen gedruckten Transponderspulen.

3.2 Ableitung des Implementationskonzeptes

Auf Basis der Voruntersuchungen, sowie der gezeigten passiven und aktiven Transceiver Konzepte, gilt es nun, unter Berücksichtigung der gestellten Anforderungen, ein für die geplante Anwendung optimiertes Konzept zu erstellen.

Zunächst werden verschiedene denkbare Varianten für das Konzept gegenübergestellt, um deren Eignung abzuschätzen.

3.2.1 Varianten für die Konzeption

Die folgenden vorgestellten Konzepte unterscheiden sich hauptsächlich in dem Bereich des *Front-End*.
Das *Back-End* bildet bei allen Konzepten eine *MCU*, welche für die Dekodierung des *DL*-Signals, für die Messwertaufnahme der Spannungs- und Temperaturmessung und die Ansteuerung des *UL*-Kanals verwendet wird. Zwar ist anstelle der *MCU* ebenfalls ein *FPGA* einsetzbar, dieser benötigt allerdings gegenüber einer *MCU* meist zusätzliche Peripherie, wie zum Beispiel einen *ADC*. Außerdem ergeben sich Vorteile bezüglich der Programmierung. Die Software für eine *MCU* lässt sich oftmals schneller entwickeln und modifizieren, wohingegen das Systemdesign eines *FPGA* mittels einer Hardwarebeschreibungssprache meist aufwendiger und unflexibler ist.
Weiterhin beinhaltet das *Back-End* einen *DC-DC-Wandler* für die Spannungsversorgung der aktiven Komponenten, da die Zellenspannung der zu messenden Batteriezelle (siehe Abschnitt 2.2.1) an die benötige Betriebsspannung für den Betrieb der aktiven Komponenten angepasst werden muss.
Da der Zellensensor, entsprechend den Anforderungen, möglichst wenig Energie verbrauchen soll, ist es zweckmäßig den *DC-DC-Wandler* nur dann einzuschalten, wenn dieser benötigt wird. Somit lässt sich der durchschnittliche Energieverbrauch gegenüber einer ansonsten gängigen Lösung, die nur die *MCU* in einen Ruhezustand versetzt, drastisch reduzieren. Für das Ein- und Ausschalten des *DC-DC-Wandlers* dient eine Detektor- und Halteschaltung, welche durch den *DL*-Kanal ausgelöst wird und zusätzlich von der *MCU* steuerbar ist.
Eine genauere Beschreibung der verwendeten Komponenten erfolgt in dem Abschnitt 3.3, schließlich gilt es zunächst verschiedene Konzepte vorzustellen.

Das erste denkbare Konzept mit dem geringstem Realisierungsaufwand, ist das in dem Bild 3.7 gezeigte Konzept I, dessen Kommunikation nach der *HDX*-Betriebsart ablaufen würde und für den *UL*-Kanal die Lastmodulation des *RF*-Feldes vom Reader verwendet.

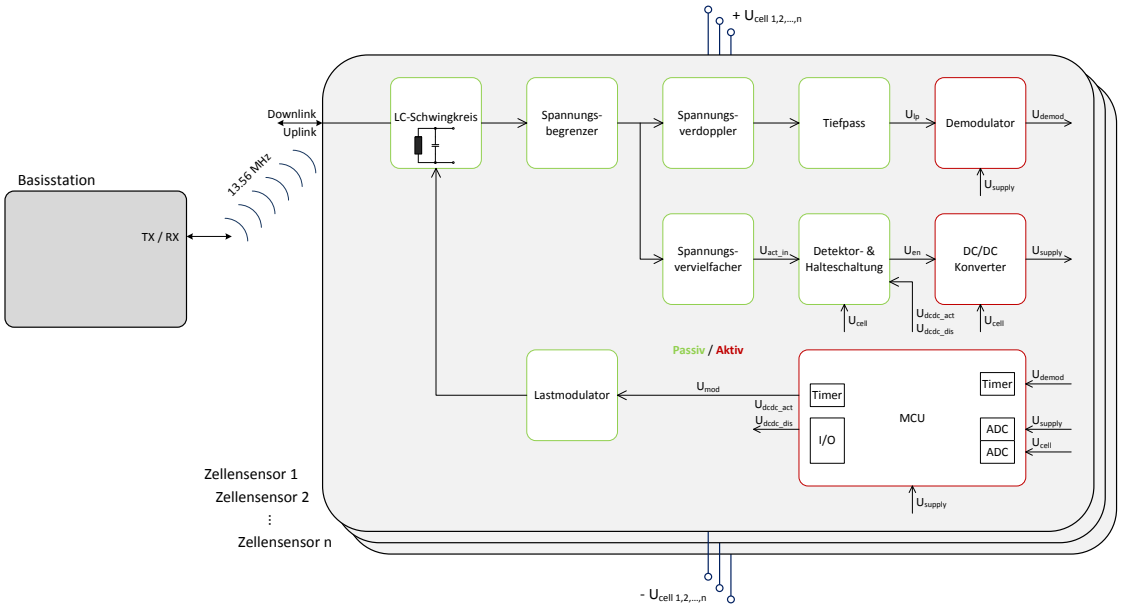


Bild 3.7: *Blockschaltbild des Konzeptes I:*
HDX-System unter Anwendung der Lastmodulation für den UL-Kanal

Als Trägerfrequenzen wären die Frequenzen kleiner gleich 135 kHz, als auch 13,56 MHz, geeignet, da für den *UL*-Kanal durch die Anwendung der Lastmodulation seitens des Zellensensors kein hochpräzises frequenzgebendes Bauteil erforderlich ist.
Ein derartiges System hätte aufgrund des verwendeten *UL*-Kanals einen Wirkungsbereich zur Folge, der nicht den Anforderungen entsprechen würde. Diese Systeme arbeiten, beispielsweise der Transponder „M24LR64-R“ von *ST*, unter Verwendung von Transponderspulen in der Größe von Chipkarten, üblicherweise mit Reichweiten von kleiner als 10 cm. Zwar erreicht man mit einer Vergrößerung der Readerspule höhere Reichweiten, dies hat jedoch eine mögliche Abschattung der sich in unmittelbarer Nähe befindlichen Transpondern zur Folge. Für die Realisierung des *Front-End* sind, neben einer relativ simplen Eigenrealisierung, verschiedene kommerzielle Lösungen einsetzbar.

Die Grundlage für das zweite Konzept, siehe Bild 3.8, ist die *SEQ* Betriebsart der Kommunikation, wie sie auch bei dem zuvor untersuchtem „TMS37157“ von *TI* angewendet wird.
Hierdurch ergibt sich der Vorteil eines verbesserten *SNR* für den *UL*-Kanal und eine geringfügig größere Reichweite als bei einem *HDX*-System mit Lastmodulation. Da für den *SEQ* Betrieb das Trägersignal für den *UL* von dem Zellensensor selber erzeugt werden muss, ist ein frequenzgebendes Bauteil erforderlich - in diesem Fall der Parallelschwingkreis, bestehend aus der Antennenspule und der

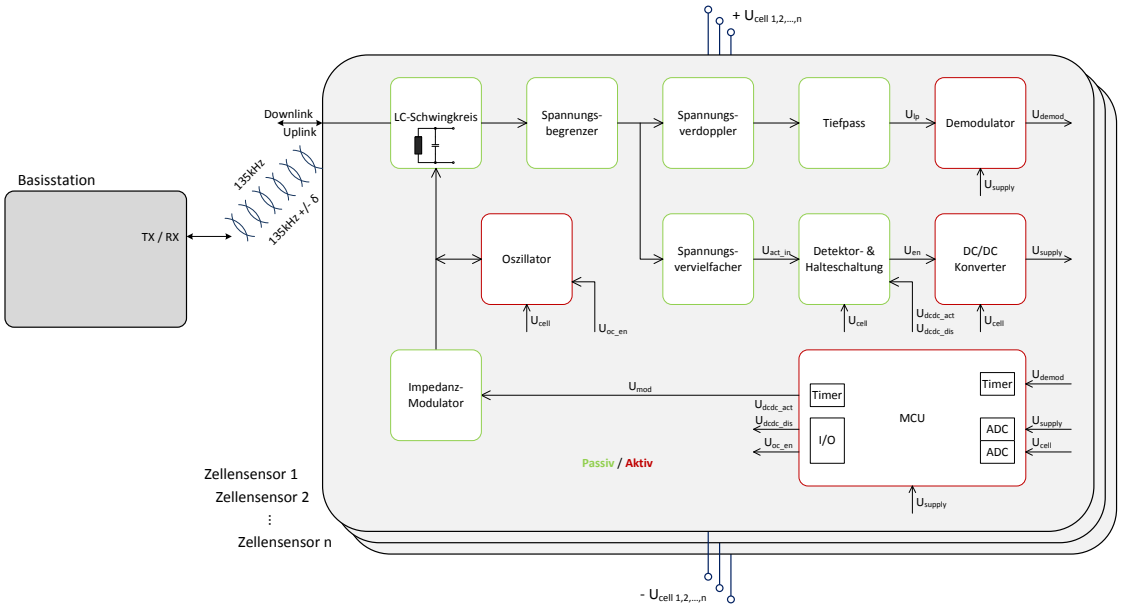


Bild 3.8: *Blockschaltbild des Konzeptes II:*
SEQ System, Transponderschwingkreis als frequenzgebendes Bauteil für den UL-Kanal

Parallelkapazität.
Wie bereits zuvor im Abschnitt 2.3.4 beschrieben, sind nur Trägerfrequenzen, die kleiner gleich 135 kHz sind, für ein derartiges System zulässig. In Folge dieser Limitierung beschränkt sich zum einen der Wirkungsbereich eines solchen Systems gegenüber höheren Trägerfrequenzen, zum anderen bestünde wieder die Problematik mit der Größe der Antennenspule des Zellensensors. Als *Front-End* ist das „TMS37157“ von *TI* nicht direkt einsetzbar, da es einen permanent aktiven *DC-DC-Wandler* zur Anpassung der Betriebsspannung an die Zellenspannung benötigt, wodurch eine Eigenentwicklung für das *Front-End* zu bevorzugen ist. Diese könnte zudem, anstelle der aus dem *HF*-Feld empfangenen Energie, die Energie der Batteriezelle für die Trägererzeugung nutzen und somit die Reichweite verbessern.

Das dritte Konzept, siehe Bild 3.9, setzt die Eigenentwicklung eines *Front-End* voraus. Ähnlich wie das Konzept zuvor, verwendet dieses die *SEQ* Betriebsart, nutzt jedoch einen separaten Oszillator für die Generierung des für den *UL* benötigten Trägersignals.
Ist die Frequenzstabilität dieses Oszillators entsprechend präzise (siehe Abschnitt 2.3.4.1), sind im Gegensatz zu dem Konzept II auch Trägerfrequenzen größer als 135 kHz für den *UL*-Kanal verwendbar. Durch die Verwendung von zum Beispiel 13,56 MHz als Trägerfrequenz, ließe sich der Wirkungsbereich vergrößern und gleichzeitig die Größe der Antennenspule des Zellensensors reduzieren, so dass diese als gedruckte Spule realisierbar ist. Als Energiequelle für den *UL*, bietet es sich

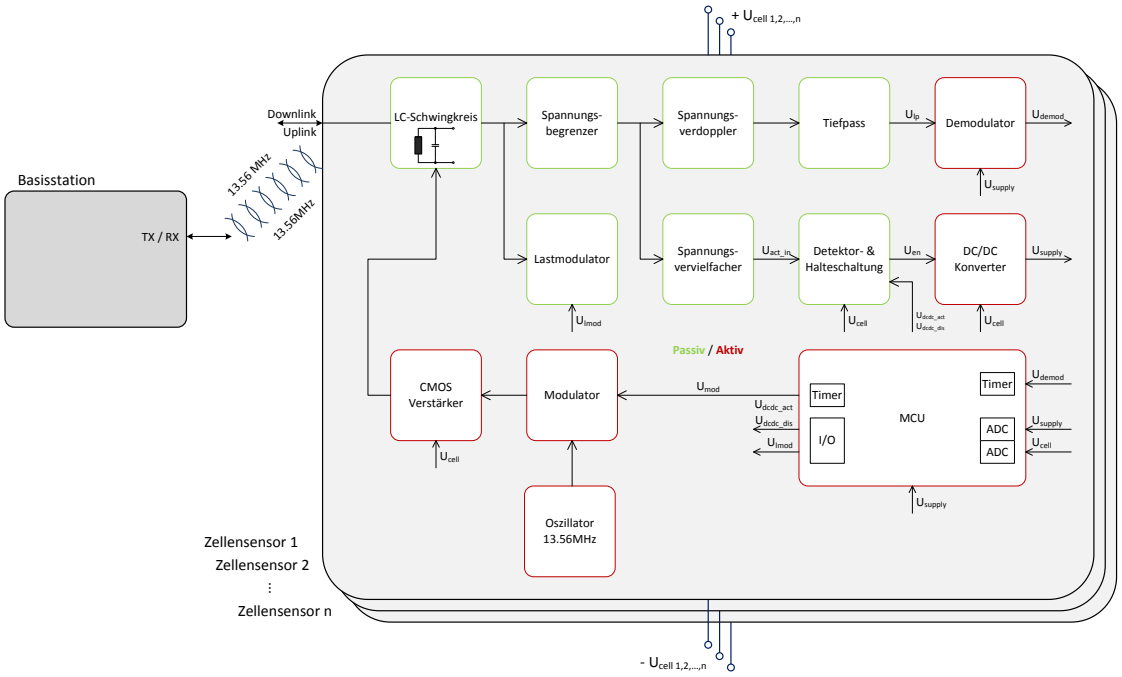


Bild 3.9: *Blockschaltbild des Konzeptes III: SEQ System, Oszillator als frequenzgebendes Bauteil für den UL-Kanal*

an, den CMOS Verstärker direkt mit der unangepassten Zellenspannung zu betreiben, um so den *DC-DC-Wandler* kleiner dimensionieren zu können. Dieses Prinzip entspräche einem Sender mit Hilfsenergie, inklusive den bereits in Abschnitt 2.4.1.1 beschriebenen Problemen. Entsprechende FETs beziehungsweise CMOS Bauelemente für den Betrieb bei Spannungen von unter einem Volt, kleinen Ausgangskapazitäten, die den Parallelschwingkreis nicht verstimmen und gleichzeitig eine genügend hohe Schaltfrequenz aufweisen, sind als diskrete Bauelemente nur sehr begrenzt verfügbar. Aus diesem Grund ist auch dieses Konzept als solches belassen worden.

3.2.2 Finales Konzept

Die zuvor dargestellten Konzepte I bis III nutzen ein gemeinsames Frequenzband für den *DL*- & *UL*-Kanal. Eine Aufspaltung in getrennte Frequenzbänder bietet hingegen, wie in Abschnitt 2.4.2 beschrieben, verschiedene Vorteile.
Wie in dem Bild 3.10 dargestellt, ist es für einen Transponder bzw. Zellensensor zweckmäßig, einen *LF*-Empfänger in Kombination mit einem *UHF*-Sender zu verwenden. Der Reader bzw. die Basisstation beinhaltet folglich einen *LF*-Sender neben einem *UHF*-Empfänger.

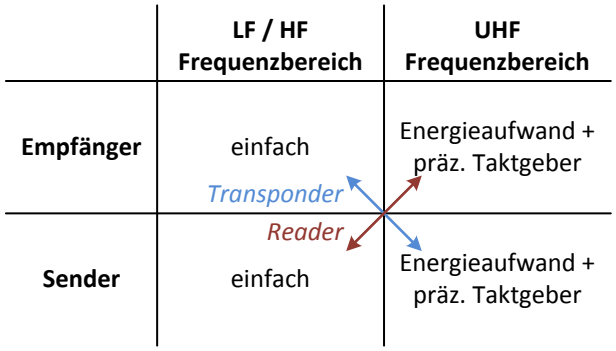


Bild 3.10: *Anwendung von LF und HF bzw. UHF Frequenzbereichen für die Realisierung*

Diese Zweckmäßigkeit beruht darauf, dass ein *LF*- oder *HF*- gegenüber einem *UHF*-Empfänger weniger Energie verbraucht und keinen Taktgeber benötigt, was sich deutlich auswirkt, da dieser auf jedem Zellensensor permanent Empfangsbereit sein muss. Zudem ist die Realisierung von Eigenentwicklungen, insbesondere bei Labormustern, deutlich vereinfacht. Der *UHF*-Sender benötigt zwar im Moment der Übertragung mehr Energie, ermöglicht aber aufgrund der elektromagnetischen Kopplung einen ausreichend großen Wirkungsbereich in Kombination mit einer höheren Kanalkapazität für den *UL*-Kanal. Seitens der Basisstation wird zwar ein *UHF*-Empfänger benötigt, je nach Betriebsart muss dieser jedoch nicht dauerhaft aktiv sein.
Betrachtet man das gesamte System, benötigt man nur einen aufwendigen aktiven *UHF*-Empfänger für die Basisstation, gegenüber n einfachen passiven *LF*- bzw. *HF*-Empfängern für n Batteriezellen.
Der *UHF*-Sender für den *UL*-Kanal auf den Zellensensoren kann, aufgrund moderner integrierter Taktgeber (siehe Abschnitt 2.3.4.1) mittels kommerziellen integrierten Transmittern, realisiert werden. Somit entspricht diese Aufteilung der Anforderung, den Zellensensor ohne einen Quarzoszillator betreiben zu können.

Als endgültiges Konzept ergibt sich, durch die Verwendung von separaten Frequenzbereichen in Kombination mit den zuvor dargestellten Konzepten, das in dem Bild 3.11 dargestellte System eines Zellensensors mit bidirektionaler Kommunikation.

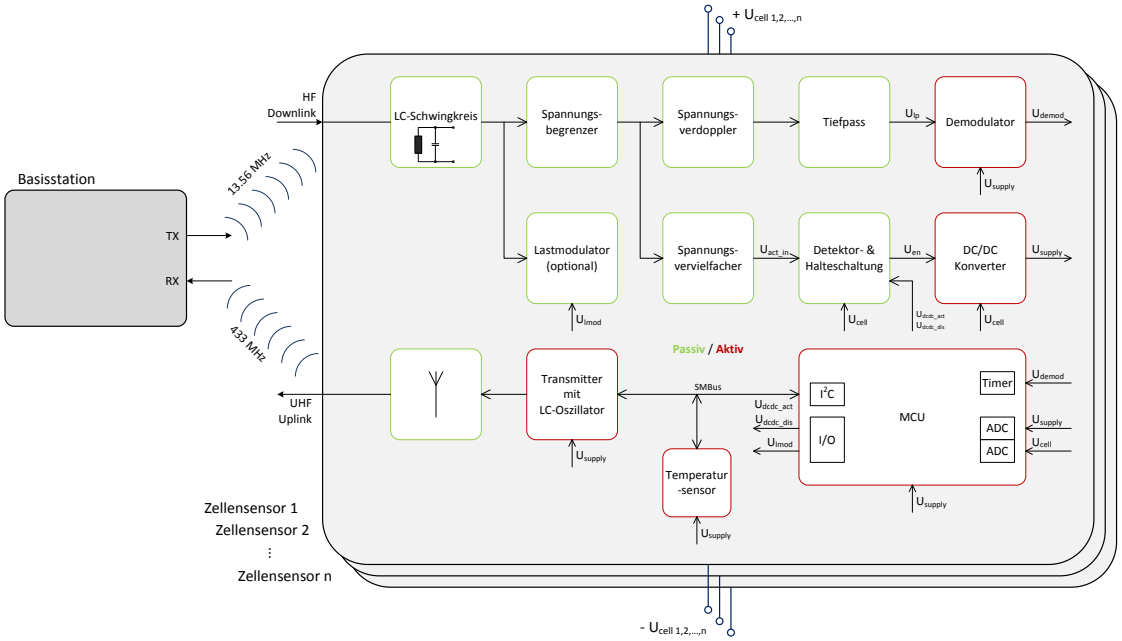


Bild 3.11: *Blockschaltbild des Konzeptes IV:*
Finales System unter Verwendung von separaten DL- & UL-Kanälen

Der *HF DL*-Empfänger ist in diesem Konzept nahezu komplett passiv ausgelegt. Für die Aktivierung des Zellensensors generiert dieser bei einem angelegten externen *HF*-Feld ein Signal für die Detektor- & Halteschaltung, die schließlich den *DC-DC-Wandler* steuert. Erst wenn der *DC-DC-Wandler* aktiv ist, steigt der Energieverbrauch nennenswert an. Weiterhin ist als alternativer *UL*-Kanal zusätzlich die Lastmodulation des *DL*-Signals experimentell vorgesehen. Als Transmitter für den *UHF UL*-Kanal dient der „SI4012“ von *SI*, welcher bereits einen LC-Oszillator integriert hat. Weitere Details des Konzeptes und der Realisierung sind dem nächsten Abschnitt 3.3 zu entnehmen.

Das finale Konzept IV ließe sich alternativ um einen mit Hilfsenergie betriebenen Empfänger (vgl. Abschnitt 2.4.1) ergänzen, wenn ein erweiterter Wirkungsbereich erforderlich beziehungsweise der des Konzeptes IV ungenügend ist. Wie in dem Bild 3.12 dargestellt, wäre neben des Vorverstärkers, welcher mit sehr wenig Energie direkt von der Batteriezelle betreibbar ist, ein aktiver Verstärker für die Demodulation des *DL*-Signals notwendig.
Gegenüber dem Konzept IV, steigt der Realisierungsaufwand für dieses Konzept

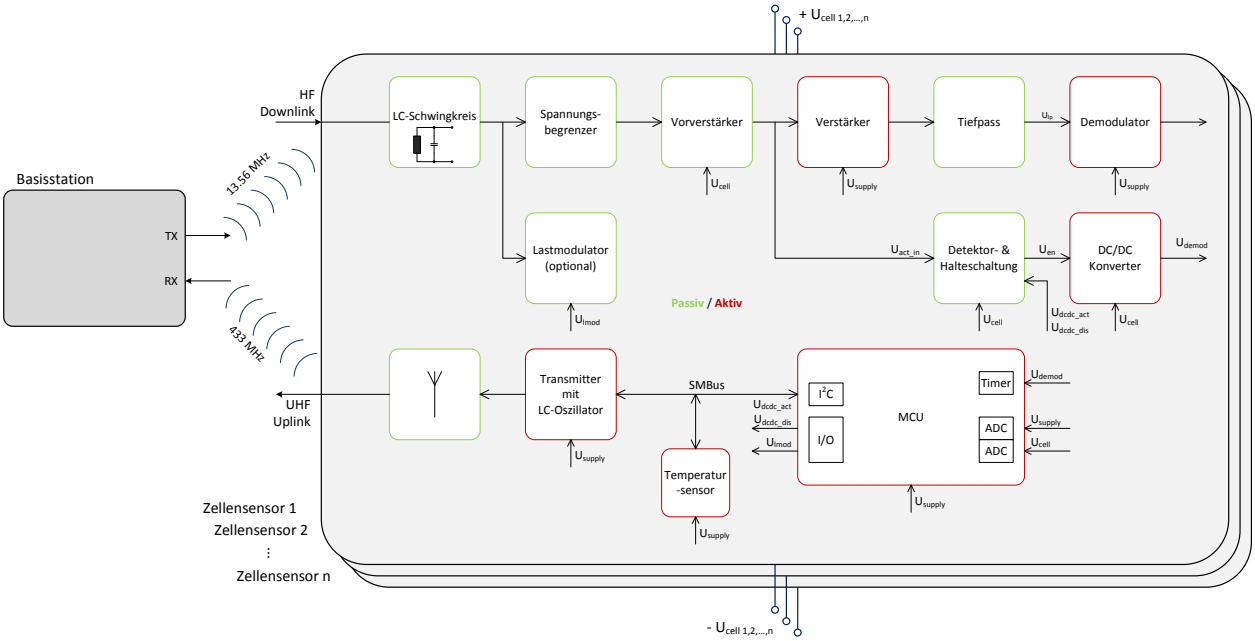


Bild 3.12: *Blockschaltbild des Konzeptes V:*
Finales System mit separatem DL- & UL-Kanal und zusätzlichem Vorverstärker

deutlich an.
Aufgrund von Voruntersuchungen an Teilen des *DL*-Empfängers von Konzept IV mit einem Labormuster (siehe Abschnitt 3.3), ist letztlich die Entscheidung für die Realisierung nach Konzept IV, ohne den zusätzlichem Vorverstärker, gefallen.
In dem folgenden Abschnitt 3.3 wird dennoch zusätzlich der Vorverstärker für den Empfänger mit Hilfsenergie näher erläutert, da dieser für die Konzeption relevant war, beziehungsweise vielleicht für zukünftige Realisierungen noch ist.

3.3 Schaltungsentwurf

Anhand des zuvor vorgestellten Konzeptes IV, für die Realisierung des Zellensensors und unter Beachtung der gestellten Anforderungen, werden im Folgenden die jeweiligen benötigten Schaltungsteile entworfen.
Dies erfolgt vielfach unter Verwendung der Software „PSpice“ für die Schaltungssimulation, da einerseits einige Problemstellungen analytisch nur näherungsweise lösbar sind, andererseits somit der benötigte Zeitaufwand für den Entwurf insgesamt reduziert werden kann. Weiterhin dienen Labormuster von Teilen der Schaltung für die Verifikation der Simulationsergebnisse.

3.3.1 HF-Empfänger

Die Empfängerschaltung des Zellensensors besteht primär aus einem Parallelschwingkreis, welcher aus der Antennenspule und einer Parallelkapazität gebildet wird. Die Induktivität der Antennenspule soll mit etwa 4,2 μH den Antennenspulen des Entwicklungskits „STARTKIT-M24LR-A“ des Herstellers *ST* (vgl. STMicroelectronics (2010b)) ähneln, um bereits vorab mit Labormustern die Schaltungsentwicklung verifizieren zu können. Weiterhin beträgt die erforderliche Parallelkapazität für die geplante Resonanzfrequenz, entsprechend der Trägerfrequenz von 13,56 MHz, nur etwa 32,8 pF und ist somit bereits zum Teil durch parasitäre Kapazitäten der nachfolgenden Schaltungsteile gedeckt.

Dem Parallelschwingkreis folgt eine Diodenschaltung, siehe Bild 3.13, zur Begrenzung der Eingangsspannung für die nachfolgenden Schaltungsteile. Dies ist notwendig, da mit Verringerung des Abstandes zwischen Reader- und Transponderspule der Kopplungsfaktor steigt und so bei entsprechender Anpassung durchaus Spannungen von einigen hundert Volt in die Transponderspule induziert werden können. Eine robuste Spannungsbegrenzung ist folglich elementar.
Die verwendete Diodenschaltung aus Silizium-Dioden vom Typ 1N914 begrenzt die positive als auch negative Spannung am Schwingkreis auf etwa 2,16 V. Aus verschiedenen Gründen sind bewusst keine Zenerdioden hierfür verwendet worden. Als Kriterium für die Auswahl dieses Diodentyps, galt zu einem eine kleine Kapazität von 4 pF, in Kombination mit einem möglichst geringen Leckstrom, schnellem Schaltverhalten und hohem Durchlassstrom. Diese Parameter sind typischerweise bei der Verwendung von Zenerdioden nicht zu erreichen. Im Hinblick auf die geplante Integration gilt zudem, dass Zenerdioden im Normalfall durch eine integrative Serienschaltung von herkömmlichen Dioden realisiert werden.

Parallel zu der Diodenschaltung befindet sich eine Transistorschaltung für die optionale Lastmodulation als Alternative zum *UHF UL*-Kanal. Die Ansteuerung der

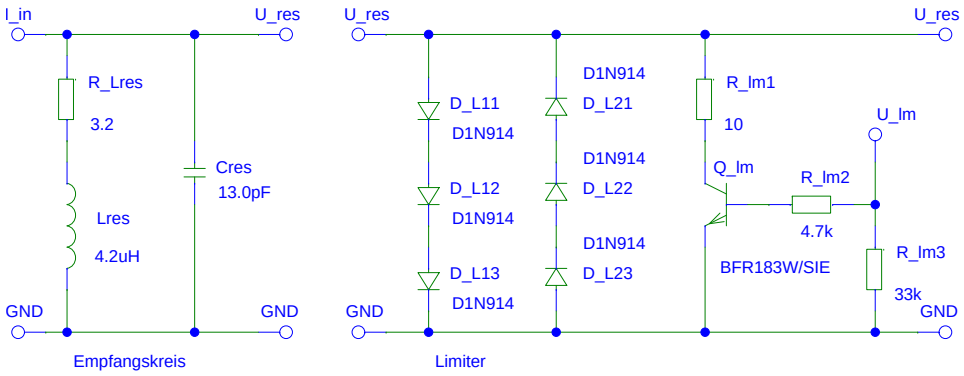


Bild 3.13: *Parallelschwingkreis des DL-Empfängers in Verbindung mit der Spannungsbegrenzung und der Lastmodulation*

Schaltung erfolgt durch die *MCU*, standardmäßig ist jedoch der Transistor nichtleitend, um den Schwingkreis nicht zu beeinflussen. Bei der Auswahl des Transistors vom Typ BFR183W waren ebenfalls, neben einem geringem Leckstrom, die parasitären Kapazitäten, in diesem Fall 0,7 pF, entscheidend.

Die Diodenschaltung zur Spannungsbegrenzung und die Transistorschaltung besitzen bereits zusammen eine parasitäre Parallelkapazität von etwa 8,7 pF. Hinzukommen die Parallelkapazitäten, die von der Demodulations- und Wake-up-Schaltung gebildet werden. So bedarf es in der Simulation lediglich einer Parallelkapazität C_{res} von 13 pF, für die Abstimmung des Schwingkreises. Bei der späteren Realisierung genügten 11,6 pF.

Zur Simulation der *DL*-Empfängerschaltung bedarf es in „PSpice“ einer Nachbildung der magnetischen Kopplung zwischen der Antennenspule der Basisstation und der Spule des Zellensensors. Wie in dem Bild 3.14 gezeigt, erfolgt dies durch eine stromgesteuerte Stimulation mit der Stromquelle I_1, welche den übertragenen Stromfluss in der Spule des Zellensensors nachbildet.
Die übrigen Elemente dienen zur Steuerung des übertragenen Spulenstroms, um neben einem dauerhaft aktivem Trägersignal auch einen mit *OOK* modulierten Träger zu generieren.

3.3.1.1 Wake-up

Entsprechend dem entwickelten Konzept IV soll der Zellensensor aus einem energiesparenden passiven Zustand in einen aktiven Zustand, initiiert durch den *DL*-Kanal, versetzt werden können. Üblicherweise wird diese Funktionalität als *Wake-Up* durch den *DL*-Kanal bezeichnet.
Der Zellensensor soll also in der Lage sein, in Folge eines ausreichend lange aktiven

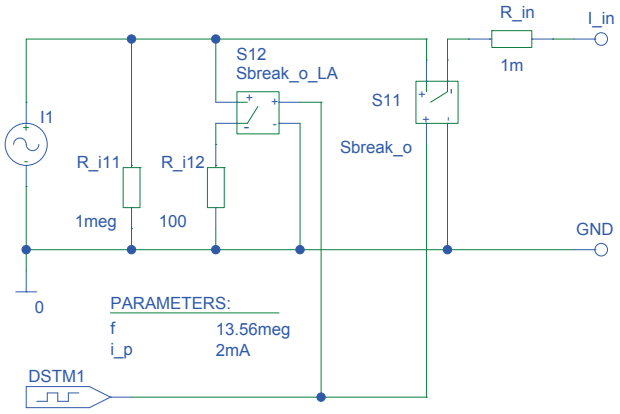


Bild 3.14: *Nachbildung der Stimulation für den Parallelschwingkreis des DL-Empfängers in der Schaltungssimulation*

HF-Feldes des *DL*-Kanals, die Spannungsversorgung für die eigenen aktiven Komponenten einzuschalten, ohne zuvor nennenswert viel Energie zu verbrauchen. Hierfür bedarf es einen Detektor, welcher angibt, ob ein äußeres *HF*-Feld der Frequenz 13,56 MHz in die Antennenspule des Parallelschwingkreises einkoppelt oder nicht. Woraufhin eine Schaltung folgt, welche das Signal des Detektors speichert und an die nachfolgende Schaltung, den *DC-DC-Wandler*, anpasst.

Als Eingangssignal für die beschriebene Detektorschaltung dient die durch die Diodenschaltung bereits begrenzte Ausgangsspannung des Parallelschwingkreises. Um die Ansprechfeldstärke zu minimieren, wird diese Spannung mit Hilfe einer Spannungsvervielfacherschaltung (siehe Bild 3.15) gleichgerichtet und vergrößert.

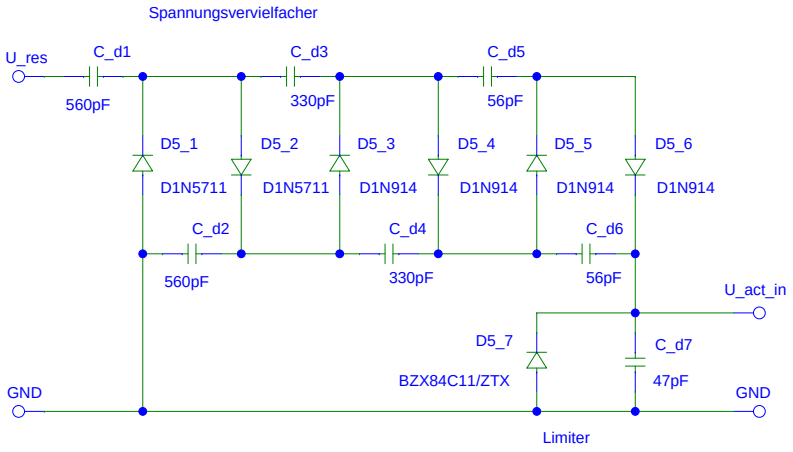


Bild 3.15: *Spannungsvervielfacherschaltung zur Generierung des Eingangssignals der Detektor- & Halteschaltung*

Die hierfür verwendete dreistufige Villard-Kaskadenschaltung, entnommen aus Lindner u. a. (2005), generiert theoretisch aus der Effektivspannung U_e am Eingang eine Gleichspannung der Größe $2n\sqrt{2} \cdot U_e$, bzw. in diesem Fall $8,48 \cdot U_e$. In der Praxis ist jedoch die Gleichspannung kleiner, da in jeder Kaskade der Spannungsabfall über den Dioden berücksichtigt werden muss und es sich bei dem Faktor zudem um die reine Leerlaufspannung handelt. Schließlich kann nur die Spannung und nicht die Leistung vergrößert werden.
So ist es für die Erzeugung einer möglichst großen Gleichspannung zweckmäßig, insbesondere bei sehr kleinen Eingangsspannungen, Schottky-Dioden mit einer kleineren Flussspannung anstelle der üblichen Silizium-Dioden für die Schaltung zu verwenden. Dabei ist entsprechend der Anforderung auszuwählen, ob alle Kaskaden oder nur die ersten Kaskaden mit Schottky-Dioden versehen werden. Weiterhin sind die Parameter der Dioden, wie kurze Schaltzeiten entsprechend der Trägerfrequenz und minimale Leckströme, entscheidend.
Um die Ausgangsspannung „U_act_in“ der Spannungsvervielfacherschaltung für die nachfolgenden Schaltungsteile zu begrenzen, ist in diesem Fall eine Zenerdiode eingesetzt.
Die Kapazitätswerte der einzelnen Kaskaden sind, unter Berücksichtigung der nachfolgenden Schaltung, simulativ optimiert. Eine numerische Berechnung wäre in diesem Fall nur näherungsweise möglich.

Eine nachfolgende Transistorschaltung (siehe Bild 3.16), die Detektor- & Halteschaltung, nutzt die Spannung „U_act_in“ als Eingangssignal, um das Signal „U_act“ für die Steuerung des *DC-DC-Wandler* zu schalten.
Sobald das Ausgangssignal „U_act“ einmal geschaltet ist, hält sich dieser Zustand über eine Rückkopplung durch die Transistoren Q_3 und Q_4 selber. Mittels der Transistoren Q_2 und Q_5 kann zudem das Ausgangssignal „U_act“ gesetzt oder rückgesetzt werden, um so den *DC-DC-Wandler* mittels der *MCU* an- bzw. auszuschalten.
Wegen der hochohmigen Auslegung der Schaltung, verbunden mit entsprechenden Typen von Transistoren, konnte die Stromaufnahme der Schaltung sehr weit reduziert werden. So beträgt die Stromaufnahme im nicht geschaltetem Zustand 35,5 μA bzw. im geschaltetem 13 μA, bei einer Zellenspannung von 2 V.
Dies ist insofern entscheidend, da diese Schaltung, neben dem deaktiviertem *DC-DC-Wandler* und dem Spannungsteiler für die Messung der Zellenspannung, die einzige des Zellensensors ist, welche permanent Energie verbraucht.

Zur Verifikation der Simulationsergebnisse ist ein Labormuster (siehe Bild 3.17), bestehend aus der Spannungsvervielfacherschaltung, einer abgewandelten Detektor- & Halteschaltung, sowie dem im nächsten Abschnitt gezeigten Hüllkurvendetektor (Bild 3.18), aufgebaut worden.
Die verwendete Halteschaltung unterscheidet sich lediglich darin, dass diese durch eine *Lichtemittierende Diode (LED)* anzeigt, ob ein *Wake-Up* erfolgt ist oder nicht. Als Antennenspule für den Sender und den Empfänger diente jeweils eine modifi-

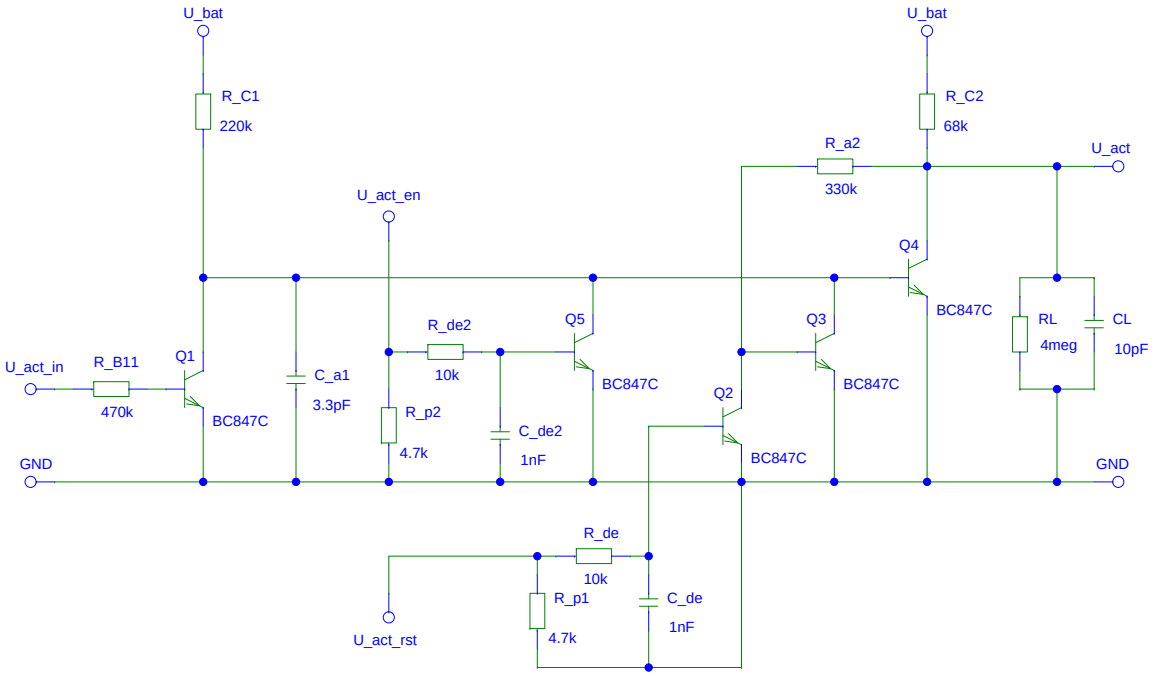


Bild 3.16: *Detektor- & Halteschaltung für die Aktivierung des DC-DC-Wandlers*

zierte Leiterplatte von den Transpondern aus dem beschriebenen Entwicklungskit „STARTKIT-M24LR-A“ von *ST*. Das Trägersignal für den *UL* konnte mittels eines Funktionsgenerators vom Typ „AFGU“ von Rohde & Schwarz generiert und durch einen weiteren Funktionsgenerator des Typs „3500FG“ von PeakTech entsprechend der gewünschten Modulationsfrequenz *OOK* moduliert werden.
Mit Hilfe dieses Aufbaus war es möglich, die Simulationsergebnisse soweit zu bestätigen, dass die Funktion der geplanten Schaltungen gegeben ist. Ein absoluter Vergleich bei einer bestimmten Ausgangsspannung des Parallelschwingkreises ist nicht möglich gewesen, da hierfür eine direkte Messung dieser Spannung erforderlich ist. Diese Messung ist jedoch stets mit einer Verstimmung des Parallelschwingkreises verbunden, aufgrund der Kapazität des Tastkopfes von einem Oszilloskops in der Größenordnung der Parallelkapazität des Schwingkreises.

Die Verwendung von derartigen Schaltungen für die Erzeugung des *Wake-Up* ist auch in anderen Bereichen mit drahtloser Kommunikation geläufig. So wurde beispielsweise in Ansari u. a. (2008) eine ähnliche Schaltung beschrieben. Diese nutzt jedoch einen Komparator, der einer *MCU* signalisiert in einen aktiven Zustand zu wechseln. Der Schaltvorgang des Komparators ist ebenfalls abhängig von der Ausgangsspannung einer Spannungsvervielfacherschaltung.
Hierbei wird allerdings eine stabile und ausreichend hohe Betriebsspannung für den Komparator vorausgesetzt. Die eigens entwickelte Transistorschaltung hinge-

gen stellt diese Anforderung nicht, der Betrieb ist in einem Bereich von 0.6 bis 5 V, entsprechend den gestellten Anforderungen, möglich.

3.3.1.2 Demodulation

Für einen *DL*-Empfänger mit möglichst minimalistischen Schaltungsaufwand ist unter anderem die Wahl des Modulationsverfahrens für die Informationsübertragung zwischen der Basisstation und dem Zellensensor von entscheidender Bedeutung.
Naheliegend ist darum die Verwendung der Modulation mit *Amplitudenumtastung (ASK)* mit einem Modulationsgrad von 100 % (*OOK*). Dies ermöglicht die Verwendung eines sehr einfachen Hüllkurvendetektors, der mit einem nachfolgenden Entscheider für die Rekonstuktion des Nutzsignals verbunden ist. Diese inkohärente Demodulation von Amplitudenmodulierten Signalen findet ebenfalls in dem Bereich der beschriebenen *RFID*-Systeme häufig Anwendung, da auch hier ein minimaler Schaltungsaufwand meist gefordert ist.
Diese Art der Demodulation entspricht zudem den gestellten Anforderungen, da im Gegensatz zu den kohärenten Demodulationsverfahren kein Taktsignal, also

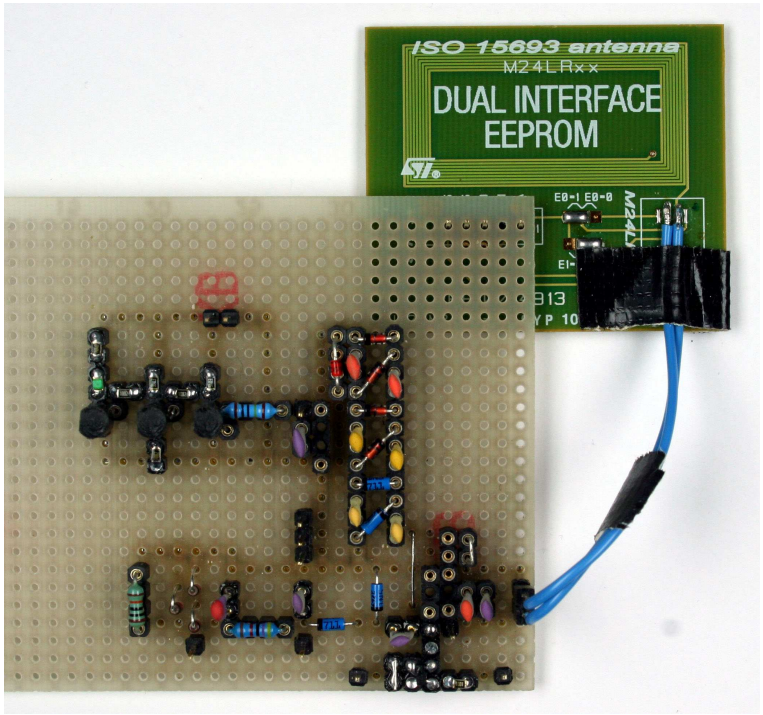


Bild 3.17: *Labormuster zur Verifikation der Simulationsergebnisse von Teilen der DL-Empfängerschaltung des Zellensensors*

kein Quarzoszillator, seitens des Empfängers erforderlich ist.
Zwar wäre auch die Verwendung der *Frequenzumtastung (FSK)* als Modulation denkbar, da auch hier inkohärente Empfänger möglich sind. Diese erfordern jedoch üblicherweise aufwändige Filter zur Selektion der Frequenzlinien.

Der für den Zellensensor entworfene Hüllkurvendetektor, wie in dem Bild 3.18 dargestellt, besteht aus einer Spannungsverdopplerschaltung nach Villard, einem nachfolgendem Tiefpassfilter und einer erneuten Diodenschaltung zur Spannungsbegrenzung.

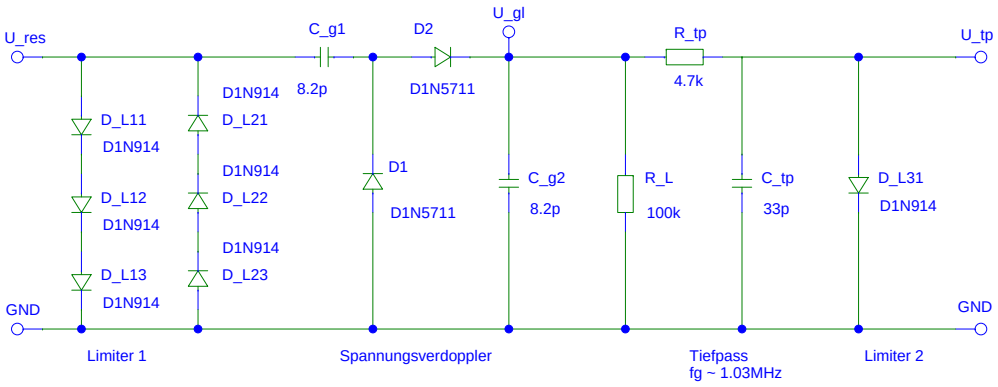


Bild 3.18: *Hüllkurvendetektor für die Demodulation, bestehend aus Spannungsverdopplerschaltung und Tiefpassfilter*

Gegenüber der Spannungsvervielfacherschaltung für den *Wake-Up* sind hingegen bei dieser die Kapazitäten $C_{g1,2}$ entsprechend der Modulationsfrequenz, abhängig von der möglichst hochohmigen nachfolgenden Impedanz, auszulegen. Die Optimierung der Kapazitätswerte von $C_{g1,2}$ für eine geplante Modulationsfrequenz von 10 kHz erfolgte auch in diesem Fall simulativ.
Bei der Auslegung des Tiefpassfilters, welcher die restlichen Signalanteile der Trägerfrequenz filtern soll, ist neben der Grenzfrequenz die Zeitkonstante für die Entladung des Kondensators, bedingt durch die Verwendung von *OOK* und des einfachen Hüllkurvendetektor, entscheidend.
Nach dem Tiefpassfilter erfolgt, trotz der Spannungsbegrenzung am Parallelschwingkreis, eine erneute Begrenzung mittels einer Silizium-Diode. Dies dient dazu, einen definierten maximalen Spannungspegel für den nachfolgenden Entscheider zur Rekonstruktion des Nutzsignals, zu erhalten.

Das von dem Tiefpassfilter ausgegebene Spannungssignal könnte für die Rekonstruktion des Nutzsignals am einfachsten direkt mit einem Logikeingang der *MCU* verbunden werden (siehe Krannich (2008)). Bei einem rein passiv betriebenen System, welches die Energie für den Betrieb der *MCU* ebenso aus dem *RF*-Feld ableitet, wie für die Informationsübertragung, ist dieser Ansatz praktikabel.
Für den konzipierten Zellensensor, welcher die Energie aus dem *RF*-Feld nur für

den *Wake-Up* und den *DL*-Kanal benötigt, würde dies eine Verkleinerung des Wirkungsbereichs zur Folge haben. Denn die Verwendung eines Logikeingangs ist aufgrund der zu geringen Eingangsimpedanz oft nicht optimal, insbesondere setzt dieser entsprechende Logikpegel des Ausgangssignals vom Tiefpassfilter voraus.
Eine Alternative wäre hierzu die Verwendung einer variablen Schaltschwelle, angepasst an den jeweiligen Spannungsbereich des Tiefpasssignals, anhand derer ein Komparator die Entscheidung zur Rekonstruktion vornimmt. Die Erzeugung der Schaltschwelle mittels eines *Digital Analog Konverter (DAC)*, sowie die Beobachtung des Signals für deren Justage durch einen *ADC*, wären mit einer *MCU* zu realisieren (siehe Jegenhorst (2009)). Vorteilhaft ist hierbei die hohe Flexibilität eines solchen Systems. Der schaltungstechnische Aufwand und die zusätzlich benötigte Prozessorzeit der *MCU* sind hingegen nachteilig. Dennoch werden derartige Verfahren im Bereich der *RFID*-Systeme eingesetzt.
Als weitere Alternative dient ein Verfahren, welches ohne eine vorzugebende Schaltschwelle die Rekonstruktion durchführen kann. Bei diesem Verfahren erfolgt ebenfalls der Vergleich zweier Spannungen. Eine folgt direkt dem Tiefpasssignal, die andere hingegen ergibt sich aus diesem Signal mit einer leichten Verzögerung. Durch Betrachtung der Differenz dieser beiden Signale, lässt sich somit jede steigende und fallende Flanke des Tiefpasssignals, entsprechend der *OOK* Modulation, erkennen.
Dieses Prinzip, entnommen aus Gebhart (2008), ist zu einer konkreten Realisierungsmöglichkeit ausgearbeitet worden und ist wie im Folgenden (siehe Bild 3.19) dargestellt.

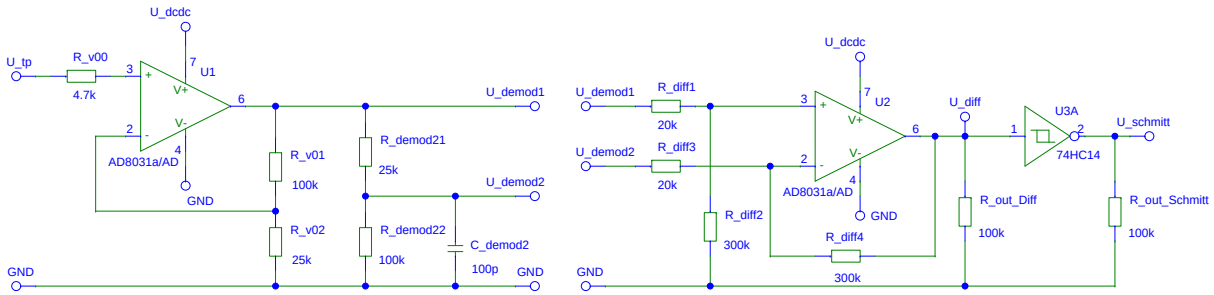


Bild 3.19: *Demodulationsschaltung für eine doppelte Flankendetektion mit einem nichtinvertierenden Verstärker und Differenzverstärker*

Das von dem Tiefpassfilter ausgegebene spannungsbegrenzte Signal wird zunächst durch einen nichtinvertierenden Verstärker vergrößert, bevor es einmal direkt und einmal durch ein RC-Glied verzögert an einen Differenzverstärker weitergegeben wird. Die Zeitkonstante des RC-Gliedes muss entsprechend der Modulationsfrequenz von 10 kHz angepasst sein. Der nichtinvertierende Verstärker sorgt neben einer leichten Verstärkung für eine Entkopplung der nachfolgenden Impedanz. Mittels des Differenzverstärkers (entnommen aus Lindner u. a. (2005)) lässt sich die

Differenz zwischen den beiden Signalen soweit verstärken, dass das Ausgangssignal direkt einem Schmitt-Trigger zugeführt werden kann. Andererseits kann somit eine minimale Differenz zur Unterdrückung von Störungen vorgegeben werden. Entsprechende simulierte Spannungsverläufe für diese Schaltung sind in dem Bild 3.20 dargestellt.

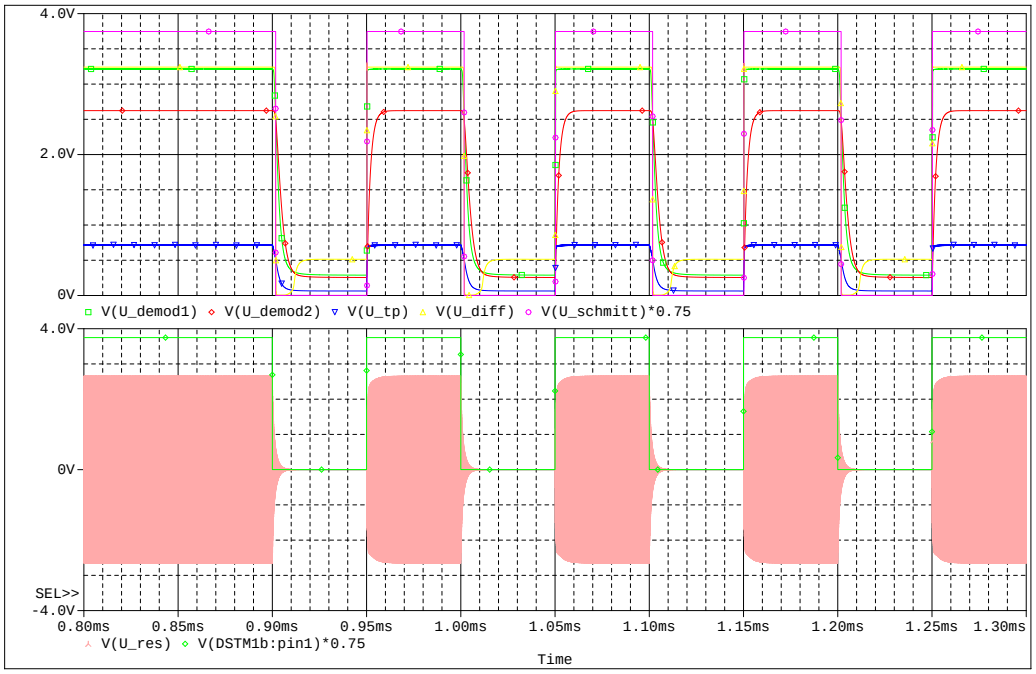


Bild 3.20: *Simulierte Spannungsverläufe von der Demodulationsschaltung mit doppelter Flankendetektion, bei einer Modulationsfrequenz von* 10 kHz

Wie man hierbei anhand der Spannung $U_{schmitt}$ erkennen kann, ist eine vollständige Rekonstruktion des modulierenden Signals „$DSTM1b : pin1$“ möglich. Die Spannung U_{demod2} folgt der Spannung U_{demod1} verzögert und geteilt, wodurch sich mit entsprechender Verstärkung ($v_d \approx 15$) die Differenzspannung U_{diff} einstellt.

Um den Realisierungsaufwand der Schaltung für die Rekonstruktion des Nutzsignals weiter zu reduzieren, ist die beschriebene Schaltung weiter abgewandelt worden.
Die so entstandene Schaltung (siehe Bild 3.21) benötigt als aktives Bauelement lediglich einen Komparator, welcher, wie die Operationsverstärker der vorherigen Schaltung, nur nach einem erfolgten *Wake-Up* durch den *DC-DC-Wandler* mit Spannung versorgt werden muss.
Die Spannung U_{demod1}, die ohne zusätzliche Zeitkonstante direkt der Ausgangsspannung vom Tiefpassfilter folgt, ist im Ruhezustand, bedingt durch einen Spannungsteiler, stets kleiner als die Spannung U_{demod2}. Erhöht sich die Ausgangsspannung des Tiefpassfilters, folgt die Spannung U_{demod1} direkt und übersteigt somit die

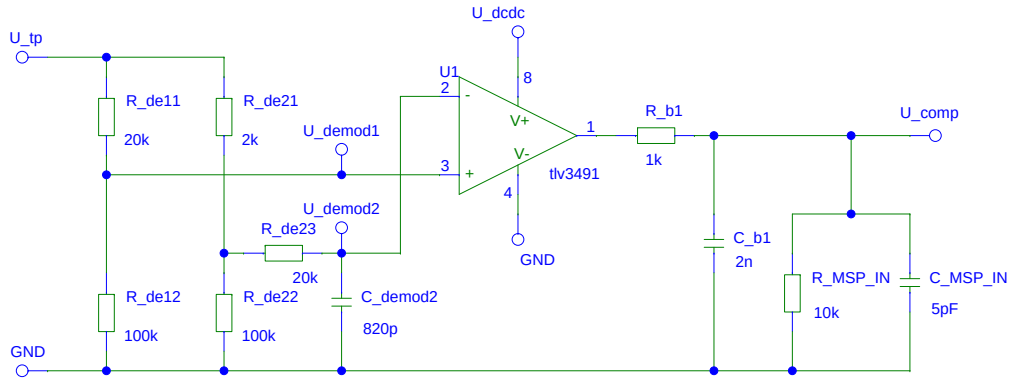


Bild 3.21: *Demodulationsschaltung mittels eines Komparators für eine einfache Flankendetektion*

Spannung U_{demod2}, da diese, wie auch bei der Schaltung zuvor, durch ein RC-Glied verzögert ist. Wie in dem Bild 3.22 dargestellt, schaltet in diesem Moment der Komparator (U_{comp}) entsprechend um.

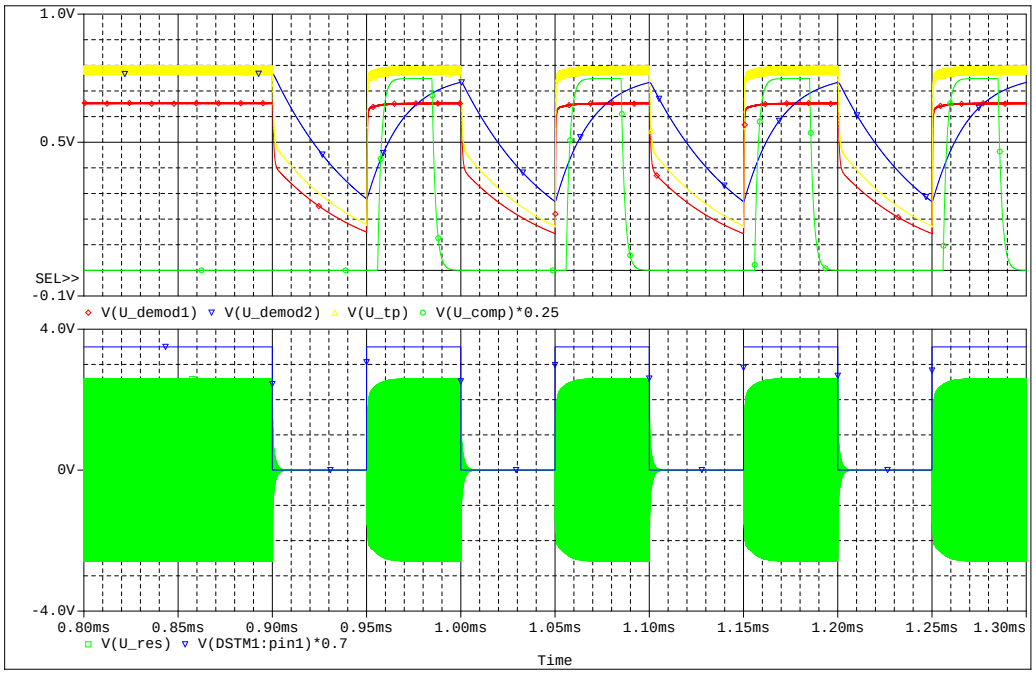


Bild 3.22: *Simulierte Spannungsverläufe von der Demodulationsschaltung mit einfacher Flankendetektion, bei einer Modulationsfrequenz von* 10 kHz

Bedingt durch den Spannungsteiler, welcher für einen definierten Anfangszustand nötig ist, ändert sich das Verhältnis der beiden Spannungen $U_{demod1,2}$ nach kurzer Zeit zueinander und der Komparator schaltet zurück.
Das Ausgangssignal des Komparators entspricht dem nötigen Logikpegel und kann somit direkt mit der *MCU* verbunden werden.

Mit dieser Schaltung ist im Gegensatz zu der etwas aufwendigeren Schaltung (siehe Bild 3.19) nur die Detektion von positiven Flanken der *OOK* Modulation möglich. So halbiert sich zwar die mögliche Übertragungsrate, jedoch ist der Realisierungsaufwand und der Energiebedarf im aktivem Betrieb reduziert.
Aus diesem Grund erfolgt die Realisierung der Empfängerschaltung des Zellensensors, mit dem Kompromiss zwischen Realisierungsaufwand und Übertragungsrate, nach der zuletzt beschriebenen Schaltungsvariante. Das gesamte Schaltbild der Empfängerschaltung in Verbindung mit dem *Wake-Up*, aber ohne den Parallelschwingkreis, ist im Anhang B.1.1 dargestellt. Der Schaltplan des Parallelschwingkreises befindet sich, aufgrund der später im Abschnitt 4.1 beschriebenen Realisierung des Zellensensors, im Anhang B.1.2.
Zusätzlich zu der bereits beschriebenen Funktionalität der Empfängerschaltung, besteht die Möglichkeit die Ausgangsspannung des Tiefpassfilters und die der Spannungsvervielfacherschaltung mittels des *ADC* der *MCU* zu messen. Dies ist vorgesehen, um auf diese Weise eine genauere Aussage über den Betriebszustand treffen zu können. Beispielsweise kann hiermit der Abgleich des Parallelschwingkreises, mit Hilfe einer Vergleichsmessung für verschiedene Kapazitätswerte, durchgeführt werden. Ebenso ist eine relativ messbare Aussage über den Zustand der Versorgung aus dem *HF*-Feldes möglich, im Gegensatz zu einer reinen Ja-Nein-Entscheidung, ob der Zellensensor betriebsbereit ist oder nicht.

3.3.1.3 Vorverstärker

Eine Abwandelung vom finalen Konzept IV, stellt das Konzept V (siehe Abschnitt 3.2.2) dar. Dieses verfügt zur Reduzierung der Ansprechfeldstärke über einen mit Hilfsenergie betriebenen Vorverstärker.
Obwohl die Realisierung des Vorverstärkers auf dem Zellensensor aus genannten Gründen im Nachhinein nicht durchgeführt wurde, ist im Vorfeld eine realisierbare Schaltung entwickelt worden.

Bei dem Entwurf des Vorverstärkers galt es, die im Folgenden aufgelisteten Rahmenbedingungen zu beachten. Diese Bedingungen erschließen sich aus den Anforderungen des Abschnittes 2.2, verbunden mit Erprobungen an einem Labormuster der Empfängerschaltung des Konzeptes IV.

- Energieaufnahme P_{zs} des Zellensensors bei Empfangsbereitschaft $\ll 20\,\mathrm{mW}$
- Betriebsspannungsbereich von $U_{zs} \leq 1\,\mathrm{V}$ bis $U_{zs} \geq 4{,}5\,\mathrm{V}$
- Frequenzbereich des Trägersignals $13{,}56\,\mathrm{MHz}$
- Modulationsfrequenz $1 \ldots 10\,\mathrm{kHz}$

- Eingangsspannungsbereich von $U_{res} \ll 0{,}5\,\mathrm{V}$ bis $U_{res} > 50\,\mathrm{V}$
- Verzerrung des Ausgangssignals gestattet, solange Demodulation möglich

Neben dem Gegensatz mit möglichst wenig Energieaufwand eine relative hohe Signalfrequenz zu verstärken, stellt der weite Betriebsspannungsbereich die größere Problematik dar.
Aus diesem Grund können keine der sonst üblichen Schaltungen, bestehend aus Operationsverstärkern, angewandt werden, vielmehr ist die Entwicklung einer Transistorschaltung notwendig. Durch die variable Betriebsspannung ist jedoch keine herkömmliche Festlegung des Arbeitspunktes durch Spannungs- oder Stromeinstellung für einen Transistor möglich. Wie in dem Bild 3.23 dargestellt, wird vielmehr eine stabilisierte Spannungsgegenkopplung angewendet.

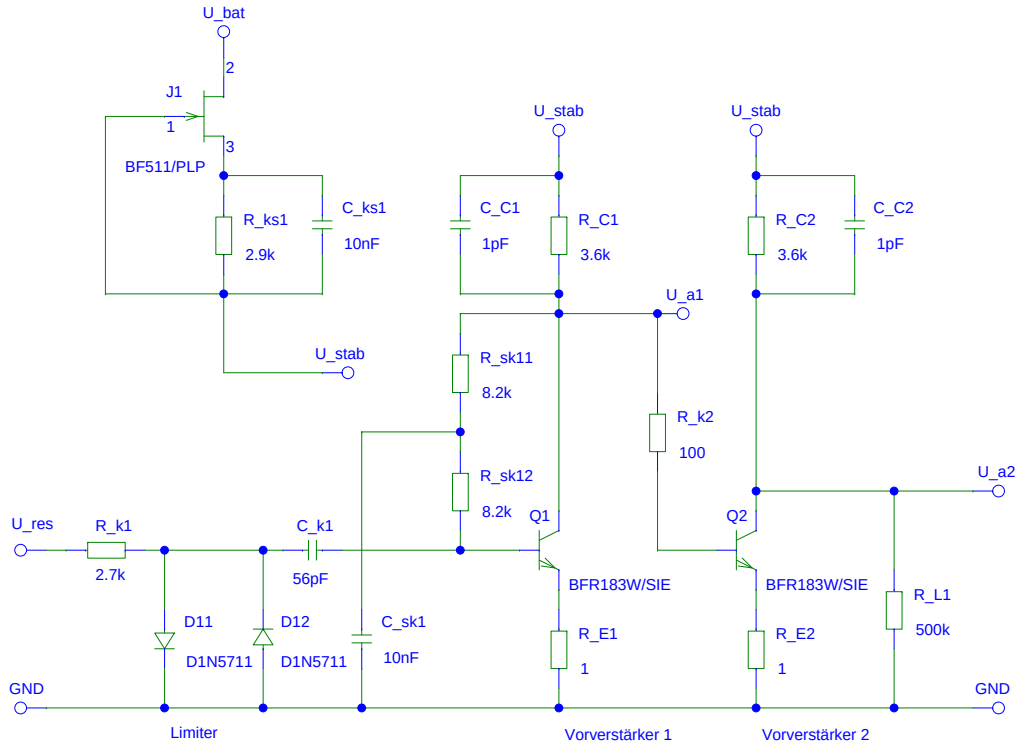


Bild 3.23: *Zweistufiger Vorverstärker für das Ausgangssignal des Parallelschwingkreises mit Spannungsbegrenzer am Eingang*

Aufgrund der Gegenkopplung reduziert sich die mögliche Verstärkung, sodass zwei gleichspannungsgekoppelte Transistoren in Emitterschaltung, zur Erzielung einer ausreichend hohen Gesamtverstärkung, erforderlich sind. Zur Minimierung des Energiebedarfs bei höheren Betriebsspannungen ist weiterhin eine Konstantstromquelle zur Begrenzung vorgesehen, von welcher die beiden Transistorstufen versorgt werden. Vor der Wechselspannungskopplung für die erste Transistorstufe

befindet sich eine Diodenschaltung, welche die maximale Eingangsspannung soweit reduziert, dass keine Übersteuerung der Verstärkerschaltung möglich ist. Die Auswahl der Transistoren ist hierbei von großer Bedeutung, neben dem Betrieb bei möglichst kleiner Spannung sind auch kleine parasitäre Kapazitäten, eine hohe Schaltfrequenz und die mögliche Verstärkung, entscheidend.

Aufgrund der begrenzten Amplitude der Ausgangsspannung von der Vorverstärkerschaltung, bei Betriebsspannungen kleiner als einem Volt, sowie der begrenzten Ausgangsleistung durch die sehr energiesparende Dimensionierung, kann für die Demodulation kein passiver Hüllkurvendetektor, wie zuvor bei dem Konzept IV, verwendet werden.
Vielmehr dient ein wechselspannungsgekoppelter nichtinvertierender Verstärker (siehe Bild 3.24) zur Vergrößerung des *HF*-Ausgangssignals der Vorverstärkerschaltung.

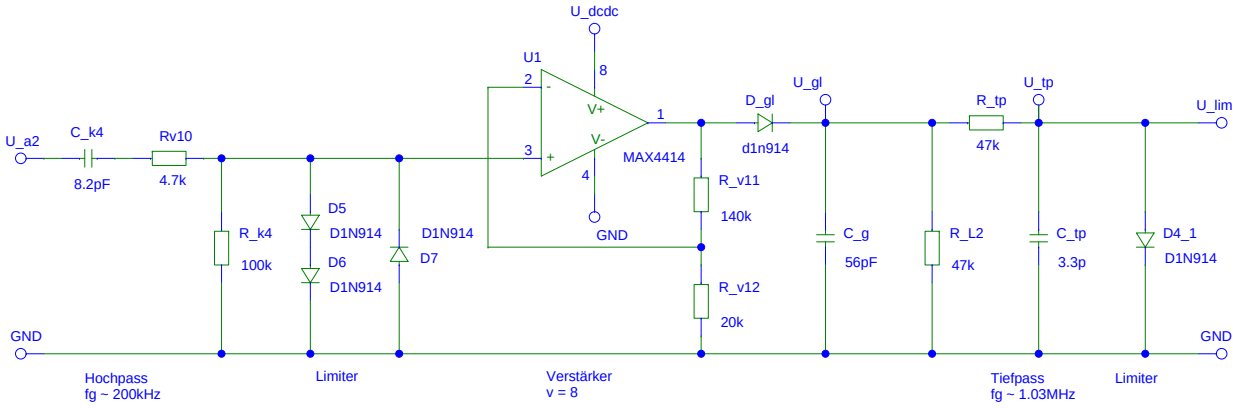


Bild 3.24: *Hüllkurvendetektor für die Demodulation, mit vorgeschalteten nichtinvertierenden Verstärker und Tiefpassfilter mit Spannungsbegrenzung am Ausgang*

Zur Begrenzung der *HF*-Eingangsspannung ist auch hier wiederum eine Diodenschaltung notwendig, da die Höhe der Amplitude von der Ausgangsspannung des Vorverstärkers abhängig von der variablen Zellenspannung ist. Für die eigentliche Gleichrichterschaltung genügt in diesem Fall eine einfache Einweggleichrichtung, die entsprechend nach der gewünschten Modulationsfrequenz dimensioniert ist. Wie zuvor in der im Bild 3.18 dargestellten Schaltung, ist weiterhin ein Tiefpassfilter mit anschließender Spannungsbegrenzung erforderlich.
Für die Rekonstruktion des Nutzsignals aus dem Ausgangssignal des Tiefpassfilters sind wiederum beide Schaltungen (siehe Bild 3.19 bzw. 3.21), aus dem Konzept IV, verwendbar.
Als Detektor- und Halteschaltung für den *Wake-Up* ist eine Abwandlung, der im Konzept IV verwendeten Schaltung (siehe Bild 3.16), möglich. Daher ist eine Spannungsvervielfacherschaltung (vgl. Bild 3.15) nicht mehr notwendig.

Zur Veranschaulichung der Funktion des Vorverstärkers, in Kombination mit dem Hüllkurvendetektor aus Bild 3.24, sind in dem Bild 3.25 einige relevante simulierte Spannungsverläufe dargestellt.

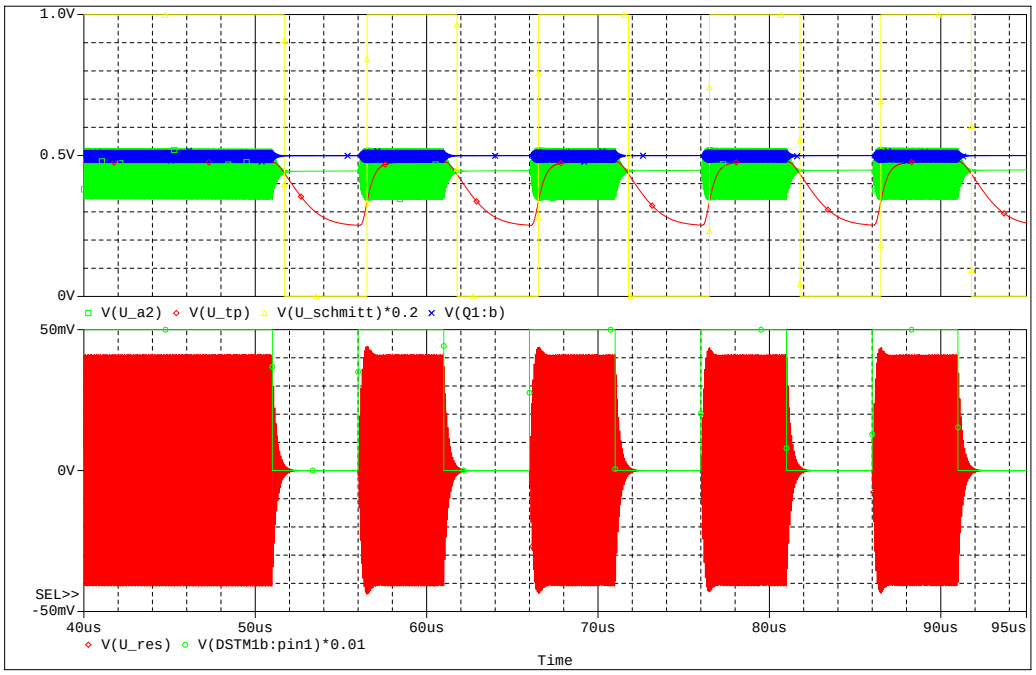


Bild 3.25: *Simulierte Spannungsverläufe der Vorverstärkerschaltung und der nachgeschalteten Demodulationsschaltung mit doppelter Flankendetektion, bei einer Modulationsfrequenz von* 100 kHz

Das mit einer Frequenz von 100 kHz modulierte Ausgangssignal U_{res} des Parallelschwingkreises, wird über die Wechselspannungskopplung an die Basis des ersten Transistors Q_1 als das Signal $U_{Q1:b}$ weitergereicht, sowie durch die Gegenkopplung mit einen Offset von etwa 0,5 V versehen. Das Ausgangssignal der ersten Transistorstufe generiert verstärkt durch die zweite Stufe, direkt das Ausgangssignal U_{a2} des Vorverstärkers, bei einer Zellenspannung von nur 0,8 V.
Durch den beschriebenen Hüllkurvendetektor entsteht aus diesem Ausgangssignal wiederum das Tiefpasssignal U_{tp}, welches durch eine entsprechende Schaltung (siehe Bild 3.19) zu dem eigentlichen Nutzsignal $U_{schmitt}$ fehlerfrei - bei einer Modulationsfrequenz von 100 kHz - rekonstruiert wird.

Die Stromaufnahme der Vorverstärkerschaltung im empfangsbereitem Betrieb beträgt (anhand von Simulationen) etwa 64 bis 470 μA, in einem Bereich der Zellenspannung von 0.8 bis 5 V. Dies entspricht einer Leistungsaufnahme von etwa 52 μW bis 2,35 mW und erfüllt somit die gestellte Anforderung.
Als minimale Spannung für das Eingangssignal U_res konnte ein Wert von $\approx$ 30 mV, bei einem zugeführten Spulenstrom von 10,6 μA, ermittelt werden. Im direkten Vergleich zu dem Konzept IV reduziert sich somit der nötige Spulenstrom um den

Faktor 13,4 und die Spannung am Parallelschwingkreis um den Faktor 14,6. Hier beträgt die minimale Spannung U_{res} etwa 438 mV bei 142 μA. Entsprechend dieser Faktoren ist somit ebenfalls die nötige Ansprechfeldstärke verkleinert. Die Distanz zwischen den Spulen des Zellensensors und der Basisstation vergrößert sich hingegen, aufgrund der hohen Feldstärkeabnahme im Nahfeld, etwa um den Faktor 2,4. Abhängig von der Ausgangslage ist somit die Distanz deutlich zu vergrößern, wenn beispielsweise anstelle von 30 cm durch die Verwendung des Vorverstärkers 72 cm erzielbar sind - oder alternativ eine Erhöhung der Beständigkeit der drahtlosen Übertragung beispielsweise in metallischen Umgebungen.
Folglich ist die Verwendung des Vorverstärkers durchaus eine Möglichkeit zur Minimierung der Ansprechfeldstärke und die damit verbundene Erhöhung der Reichweite. Betrachtet man das gesamte System der Zellen-Sensorik, ist es kritisch zu betrachten, ob der erhöhte Realisierungsaufwand und Energiebedarf für die der Anzahl n erforderlichen Zellensensoren, gegenüber einem leistungsstärkeren Transmitter in der Basisstation, gerechtfertigt ist.

3.3.2 UHF-Sender

Das finale Konzept IV sieht die Verwendung von getrennten Frequenzbereichen für den *DL*- und den *UL*-Kanal vor. Entsprechend ist ein vom *HF*-Empfänger vollkommen getrennter aktiver *UHF*-Sender, für den *UL*-Kanal eingeplant.

Wie bereits in dem Abschnitt 2.4.2 angedeutet, soll hierfür ein integrierter *UHF*-Transmitter mit bereits integriertem LC-Oszillator verwendet werden. Nach umfangreicher Recherche fiel die Wahl auf den Transmitter „Si4012“ von *SI* (siehe Bild 3.26), den einzigen bekannten und verfügbaren Transmitter dieser Art, welcher die im Folgenden aufgelisteten Eigenschaften (aus Silicon Laboratories (2010b)) aufweist.

- Trägerfrequenz: einstellbar 27 ... 960 MHz
- Frequenzstabilität: $\pm$250 ppm bei $-40 \ldots 85\,^{\circ}$C
- Modulationsverfahren: *FSK* und *OOK*
- Übertragungsrate: 100 kBaud - *FSK*, 50 kBaud - *OOK*
- Schnittstelle zu *MCU*: *System Management Bus (SMBus)* ($\approx I^2C$)
- 255 Byte FIFO-Speicher
- Automatische Antennenabstimmung

- maximal 14,2 mA Stromaufnahme bei *OOK* Modulation
- 600 μA Stromaufnahme im Ruhezustand unter Beibehaltung der Einstellungen

Als Trägerfrequenz sollen weiterhin 433 MHz, in Anlehnung an den bestehenden Zellensensor der Klasse 1, eingesetzt werden, ebenso wie die *OOK* Modulation. Die Frequenzstabilität genügt hierbei den Anforderungen von $\pm$863 ppm für das gewählte 433 MHz *ISM*-Frequenzband.

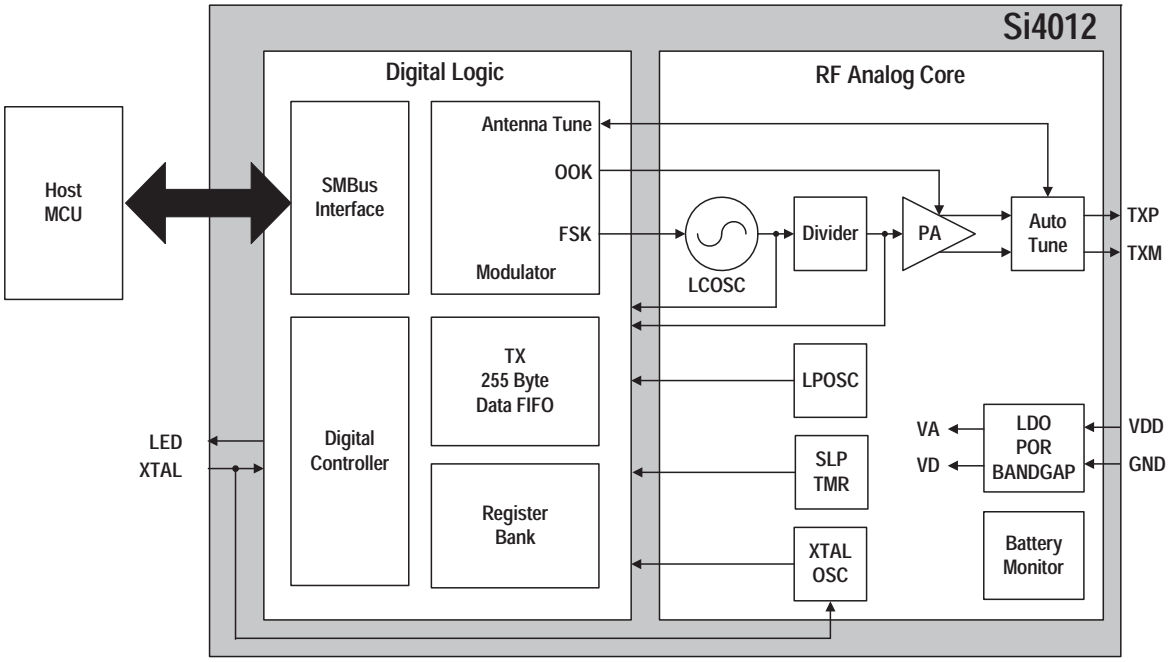


Bild 3.26: *Blockschaltbild des UHF Transmitters „Si4012“ vom Hersteller SI mit integriertem LC-Oszillator, entnommen aus Silicon Laboratories (2010b)*

Im Gegensatz zu dem bestehenden Zellensensor der Klasse 1 müssen bei diesem die zu transmittierenden Daten nur in den FIFO-Speicher geladen werden. Die Transmission selber, getriggert durch ein entsprechendes Kommando, erfolgt automatisch entsprechend der eingestellten Übertragungsrate. In Folge des internen LC-Oszillators ist diese nunmehr mit einem wesentlich kleineren Fehler behaftet, gegenüber einer direkten Modulation durch die *MCU*, getaktet mit einem unpräzisen RC-Oszillator.

Die Beschaltung des Transmitters (siehe Anhang B.1.2) bedarf nur einer minimalen Anzahl von Bauelementen und ist entsprechend den Empfehlungen aus Silicon Laboratories (2010a) vorgenommen worden. Als Antenne wird eine gedruckte Schleifenantenne verwendet, Details hierzu siehe Abschnitt 4.1.2, welche entsprechend den Anforderungen, kostengünstig realisierbar ist.

3.3.3 Mikrocontroller

Aus in dem Abschnitt 3.2 genannten Gründen soll für den konzipierten Zellensensor eine *MCU* für das *Back-End* verwendet werden. Folgende Vorgaben für die Auswahl einer passenden *MCU* ergeben sich unter anderem anhand des Konzeptes IV:

- *SMBus*- bzw. I^2C-Schnittstelle
- 12 Bit *ADC* mit ≥ 4 Kanälen und interner Referenzspannungsquelle
- ≥ 2 Zeitgeber
- ≥ 2 kB RAM und ≥ 8 kB ROM
- ≥ 7 I/O-Pins
- interner RC-Oszillator mit ≥ 8 MHz für den Systemtakt
- *Supply Voltage Supervisor (SVS)* zur Überwachung der Betriebsspannung
- JTAG-Programmierschnittstelle

Die Dimensionierung der *MCU* soll bewusst nicht zu minimalistisch sein, um für die Erprobung des Konzeptes, sowie später nachfolgenden Algorithmen, ausreichend Reserven zu haben. Eine Optimierung bezüglich der *MCU* steht hierbei nicht im Vordergrund.
Entsprechend den Anforderungen (siehe Abschnitt 2.2) soll eine Messung der Zellenspannung auf wenige Millivolt genau, in einem Messbereich von ≤ 1 bis 5 V, möglich sein. Bei der Verwendung eines *ADC* mit 10 Bit Auflösung würde dieser Messbereich zu einer Genauigkeit von nur 4,88 mV führen, weshalb eine Auflösung von12 Bit, verbunden mit einer Genauigkeit von 1,22 mV, gefordert ist.
Bei der Auswahl gilt es weiterhin zu beachten, dass die I^2C-Schnittstelle nicht die gleichen Pins der Programmierschnittstelle belegt, um so die Debug-Möglichkeiten nicht zu beeinträchtigen. Ebenso ist die Art der I^2C-Schnittstelle von Bedeutung, da es es Schnittstellen gibt, welche nur die nötigsten Funktionen in Hardware realisiert haben, wodurch die benötigte Prozessorzeit für Realisierung in Software zwangsweise vergrößert wird.
Die Anforderung an die Taktfrequenz ergibt sich unter anderem anhand der verwendeten Modulationsfrequenz des *DL*-Kanals. Beträgt diese beispielsweise wie angedacht 10 kHz, so stehen bei einer Taktfrequenz von 1 MHz lediglich 100 Takte für die Dekodierung des Demodulationssignals zur Verfügung. Aus diesem Grund ist eine höhere Taktfrequenz zu bevorzugen, schließlich ist die Reduktion im Nachhinein, im Gegensatz zu einer Erhöhung, meist möglich.

Um einen sicheren Betrieb der *MCU* gewährleisten zu können, ist eine *SVS* notwendig. Diese überwacht stets die Betriebsspannung und führt bei Unterschreitung einer einstellbaren Schwelle eine gewünschte Aktion aus. Beispielsweise, wenn der *DC-DC-Wandler* durch den *Wake-Up* aktiviert wird und die Betriebsspannung hochfährt, ist es sinnvoll, dass die *MCU* sich erst dann vollständig konfiguriert, wenn die Spannung den gewünschten Schwellwert erreicht hat.

Für die Auswahl der *MCU* steht die MSP430-Serie des Herstellers *TI* zur Verfügung, da diese bereits im Projekt „BATSEN“ erfolgreich eingesetzt wird und somit die Portabilität von Programmcode bestehen bleibt. Aus dieser Serie ist schließlich der „MSP430F235“ (siehe Texas Instruments (2007a)) ausgewählt worden, welcher die Anforderungen bei weitem erfüllt.
Da der integrierte *ADC* eine interne Referenzspannung von 2,5 V verwendet, ist für die Messung der Zellenspannung ein $1:1$ Spannungsteiler erforderlich, welcher Zwecks Minimierung des Energieverbrauchs hochohmig ausgelegt ist. Gleiches gilt für die Messung der Betriebsspannung, der Ausgangsspannung des *DC-DC-Wandler*. Die gesamte Beschaltung der *MCU* ist dem Anhang B.1.1 zu entnehmen.

3.3.4 Spannungsversorgung

Für den Betrieb der aktiven Komponenten des Zellensensors wird eine stabilisierte Betriebsspannung benötigt. Diese Komponenten sind:

- Mikrocontroller
- *UHF*-Transmitter
- Komparator der *HF*-Empfängerschaltung
- Temperatursensor (siehe Abschnitt 3.3.6)

Aufgrund des geforderten Bereichs der zulässigen Zellenspannung von $< 1{,}0\,\mathrm{V}$ bis $> 4{,}5\,\mathrm{V}$ (siehe Abschnitt 2.2.1) für den Betrieb des Zellensensors, sowie einer festgelegten Betriebsspannung von 3,3 V für die aktiven Komponenten, ergibt sich die Notwendigkeit der Verwendung eines *DC-DC-Wandlers*. Da die Eingangsspannung (Zellenspannung) sowohl kleiner, als auch größer als die Ausgangsspannung (Betriebsspannung) sein kann, muss der *DC-DC-Wandler* eine sogenannte Step-Up-Down-Funktionalität aufweisen. Was heißt, dass dieser einerseits als Step-Up-Wandler fungiert, wenn die Eingangsspannung vergrößert werden muss (Boost Mode). Andererseits als Step-Down-Wandler, wenn diese zu verkleinern ist (Down Conversion Mode). Der zu erzeugende Ausgangsstrom für den Betrieb der Komponenten sollte mindestens etwa 30 mA betragen.

Weiterhin ist durch die *Wake-Up*-Funktionalität gefordert, dass der *DC-DC-Wandler* über einen Eingang zum Ein- und Ausschalten verfügt und, dass, wenn dieser Ausgeschaltet ist, der Ausgang vom Eingang entkoppelt wird.

Entsprechend dieser Anforderungen ist der *DC-DC-Wandler* „TPS61201“ des Herstellers *TI* ausgewählt worden. Dieser verfügt über eine sehr geringe Anlaufspannung von 0,5 V und ermöglicht den Betrieb bei bis zu 5,5 V Zellenspannung. Der Wirkungsgrad, welcher abhängig von der Eingangsspannung ist, beträgt bei 0,9 V Zellenspannung bereits über 50 %, was für einen energiesparenden Betrieb des Zellensensors relevant ist.

Die Beschaltung des *DC-DC-Wandler* (siehe Anhang B.1.1) ist entsprechend den Vorgaben aus dem dazugehörigem Datenblatt (siehe Texas Instruments (2007c)) vorgenommen worden. Als untere Grenze der Zellenspannung ist ein Schwellwert von 0,6 V zur Sicherung eines zuverlässigen Betriebs eingestellt worden, ab welcher der *DC-DC-Wandler* sich selbständig abschaltet.

3.3.5 Ladungsbalancierung

Der konzipierte Zellensensor soll, neben der Überwachung von der Zellenspannung und Temperatur, auch über einen durch den *DL*-Kanal steuerbaren Effektor für die Ladungsbalancierung verfügen.
Dieser Effektor ist im wesentlichen ein Schalter, welcher einen zusätzlichen Nebenstrompfad schaltet. Durch den Nebenstrompfad ist es möglich, den Energieverbrauch des Zellensensors zu vervielfachen. In der Regel wirkt dieser folglich wie eine zusätzliche Last. Auf diese Weise ist es möglich die Zellen einer Batterie zu balancieren, indem alle Zellen beispielsweise auf den gleichen Spannungslevel, gesteuert durch die Batteriemanagementeinheit, entladen werden.

Entsprechend einfach ist die Schaltung zur Ladungsbalancierung aufgebaut. Der Schalter ist, siehe Anhang B.1.1, durch einen N-Kanal Mosfet in Sourceschaltung realisiert, verbunden mit einem Drainwiderstand zur Strombegrenzung. Zur Erhöhung des Energieverbrauchs ist eine Parallelschaltung dieser Schaltung möglich. In der geplanten Schaltung ist somit die Verdopplung des Stromflusses auf etwa 88 mA bei 1,5 V bis 5 V Zellenspannung durchführbar.
Abhängig von der Kapazität der Batterie und der zur Verfügung stehenden Zeitdauer für eine Ladungsbalancierung, ist der Effektor entsprechend zu dimensionieren. Allerdings gibt es, abhängig von der Geometrie des Zellensensors und der Umgebung, stets eine Obergrenze für die Geschwindigkeit der Balancierung, aufgrund der freigesetzten Verlustleistung. Für den konzipierten Zellensensor ist die Dimensionierung von untergeordneter Bedeutung, vielmehr ist das Prinzip und die Steuerung der Ladungsbalancierung entscheidend.

3.3.6 Temperaturmessung

Für die Überwachung einer Batteriezelle ist neben der Spannungsmessung die Messung der Temperatur von Bedeutung.
Bisher existieren jedoch keine Abschätzungen über die geforderte Genauigkeit der Temperaturinformation, aufgrund eines noch fehlenden Batteriemodells. Vorhergehende Experimente mit dem bereits existierenden Zellensensor der Klasse 1 haben gezeigt, dass eine Temperaturmessung anhand der internen Temperaturdiode eine Toleranz von mehr als 1 °C aufweist.
Im Rahmen des Projektes später folgende Untersuchungen, bezüglich der Temperaturinformation, auch die Möglichkeit einer genaueren Temperaturmessung zu ermöglichen, ist die Verwendung eines separaten vollständig integrierten Temperatursensors vorgesehen. Da für den *UHF*-Transmitter bereits ein *SMBus* zum Datenaustausch benötigt wird, kann an diesen zusätzlich ein solcher Sensor parallel angeschlossen werden.

Ausgewählt für diese Anwendung wurde der Temperatursensor „TMP102“ des Herstellers *TI* (siehe Texas Instruments (2007b)), der eine Auflösung von 0,0625 °C bei einer Genauigkeit von 0,5 °C in dem Bereich von -25 °C bis $+85$ °C ermöglicht. Zudem verfügt dieser Sensor über eine sehr geringe Stromaufnahme von nur 15 μA, verbunden mit einer sehr kleinen Bauform von $1,6 \times 1,6$ mm, was den Anforderungen an den Zellensensor gerecht wird. Die Beschaltung des Sensors ist dargestellt in dem Anhang B.1.1.

4 Realisierung

In dem vorherigem Kapitel 3 ist neben der Konzeption auch der Schaltungsentwurf des neuen Zellensensors vorgenommen worden.
Dieses Kapitel beschäftigt sich hingegen zunächst mit der hardwaretechnischen Realisierung des Zellensensors und der bisher nicht weiter erörterten Gegenstelle, der Basisstation. Anschließend erfolgt das Softwaredesign für den Zellensensors und die Basisstation, inklusive der Umsetzung, verbunden mit einem Konzept für die Koordination der Zellen-Sensorik.
Für die Verifikation des realisierten Systems muss natürlich eine Erprobung vorgenommen werden, woraufhin zuletzt eine Analyse des Systems und ein Ausblick auf Optimierungen sich ergeben.

4.1 Praktischer Aufbau des Zellensensors

Der praktische Aufbau des neuen Zellensensors soll sich nach Möglichkeit an der Geometrie, siehe Anhang B.1, des bereits bestehenden Zellensensors der Klasse 1 orientieren. Die bereits verwendeten Abmessungen der Platine von $71 \times 20\,\text{mm}$, zeigten sich zumindest für die Integration in einer Starterbatterien als geeignet. Weiterhin existiert bereits ein Nadeladapter für die Programmierung der Sensoren, sowie verschiedene Adapter, welche allesamt auf die vorhandene Geometrie der Anschlussflächen abgestimmt sind.
Da sowohl die *DL*-, als auch die *UL*-Antenne in gedruckter Form realisiert werden sollen, ist aus geometrischen Gründen angedacht, eine Realisierung bestehend aus zwei Platinen vorzusehen. Eine Platine, die nur die beiden Antennen inklusive Anpassung und *UHF*-Transmitter beherbergt. Eine andere Platine wird hingegen für alle übrigen Komponenten eingesetzt. Hiermit erhält man zugleich den Vorteil des modularen Aufbaus. Wenn beispielsweise eine Änderung der Antennengeometrie vorzunehmen ist, bedarf es nur der geringfügigen Anpassung einer Platine mit wenigen Bauelementen. Man erhält somit, wie gefordert, einen erprobungsfreundlichen Aufbau.
Für die spätere Endanwendung des Zellensensors ist der geplante Aufbau wahrscheinlich nicht geeignet. Da eine möglichst umfassende Integration aller Analog- und Digitalschaltungen in einen Chip geplant ist, welcher wesentlich weniger

Platzbedarf als die jetzigen Schaltungen haben dürfte, ist die jetzige Konzeption mit zwei gedruckten Antennen als machbar anzusehen.

4.1.1 Antenne Downlink-Kanal

Die Realisierung der Antennenspule für den *DL*-Kanal des Zellensensors soll, wie bereits beschrieben, mittels einer gedruckten Leiterschleife erfolgen.
In Vorversuchen ist bereits mit einem Labormuster, von Teilen des entworfenen *HF*-Empfängers, experimentiert worden. Als Antennenspule diente hierbei eine modifizierte Leiterplatte von einem Transponder aus dem Entwicklungskit „STARTKIT-M24LR-A“ von *ST*. Da sich der Induktivitätswert von der verwendeten Antennenspule bei der Schaltungsentwicklung als geeignet darstellte, soll ebenso die zu entwerfende Antennenspule den gleichen Wert aufweisen.
Für die Realisierung muss die Geometrie der Antennenspule an die verfügbare Fläche auf der Leiterplatte des Zellensensors angepasst werden. Als optimale Geometrie ergibt sich somit eine einlagige planare rechteckige Spule. Der Induktivitätswert derartiger Spulen lässt sich relativ aufwendig abhängig von der Geometrie berechnen, wie in Microchip (2003) dargestellt.
Alternativ zu einer numerischen Berechnung wäre eine elektromagnetische Feldsimulation der Antennenspule mit beispielsweise „CST Microwave Studio“ oder „Ansoft HFSS“ möglich. Diese hätte zudem den Vorteil, dass neben der Geometrie der Antennenspule auch die übrigen Strukturen auf der Leiterplatte berücksichtigt werden, insbesondere unter Einwirkung frequenzabhängiger Effekte wie den Skin und Proximity Effekt.

Zur Minimierung des Zeitaufwands für den Entwurf wird die benötigte Geometrie der Antennenspule mit der Software „Antenna Design“ des Herstellers *ST* (siehe STMicroelectronics (2009)) näherungsweise bestimmt. Hierzu erfolgt eine Variation der Geometrie bis der gewünschte Induktivitätswert erreicht ist. Die Software berechnet numerisch den Induktivitätswert für eine gegebene Geometrie einer planaren rechteckigen Leiterschleife. Zur Verifikation der verwendeten Software diente im Vorfeld abermals eine Transponderspule von dem Entwicklungskit „STARTKIT-M24LR-A“ von *ST*, deren Geometrie vermessen und der berechnete Induktivitätswert mit einer Messung an einer LCR-Messbrücke verglichen wurde.
In der Tabelle 4.1 sind die ermittelten Parameter für die benötigte Antennenspule des Zellensensors dargestellt.
Anhand dieser Parameter erfolgte anschließend die Realisierung der Antennenspule (siehe Bild 4.1) im Layout-Editor „EAGLE“. Das Bild 4.2 zeigt die entstandene Leiterplatte. Die in dem Bild ersichtliche Kupferfläche in der Mitte der Antennenspule, wird für die *UL*-Antenne (siehe Abschnitt 4.1.2) benötigt.
Zwar konnte die Dimensionierung der Antennenspule nur näherungsweise für den

Parameter	Größe
Induktivitätswert (soll)	4,2 μH
Länge der Spule	44,3 mm
Breite der Spule	20,3 mm
Leiterbahnabstand	0,15 mm
Leiterbahnbreite	0,35 mm
Leiterbahndicke	35 μm
Substratdicke	1,0 mm
Windungszahl	9 Wdg.
Segmentanzahl	36 Sgt.

Tabelle 4.1: *Parameter der Antennenspule für den DL-Kanal des Zellensensors*

gewünschten Induktivitätswert vorgenommen werden, aufgrund der Möglichkeit des Resonanzabgleichs mit der Parallelkapazität des Schwingkreises, kann ein bis zu etwa $\pm 25\,\%$ abweichender Induktivitätswert kompensiert werden.
Der Resonanzabgleich des Schwingkreises wäre ohnehin erforderlich, da die Eingangsimpedanz der *HF*-Empfängerschaltung nur näherungsweise anhand der Schaltungssimulation bekannt ist. Weiterhin entstehen zudem parasitäre Kapazitäten abhängig, von dem Layout der Realisierung, die ebenfalls zu einer Beeinflussung der Parallelresonanz führen können.

Bei der Erprobung des vollständigen Zellensensors konnte sukzessive, mit Hilfe der berührungslosen Messung (siehe Abschnitt 3.3.1.2) der Empfangsparameter,

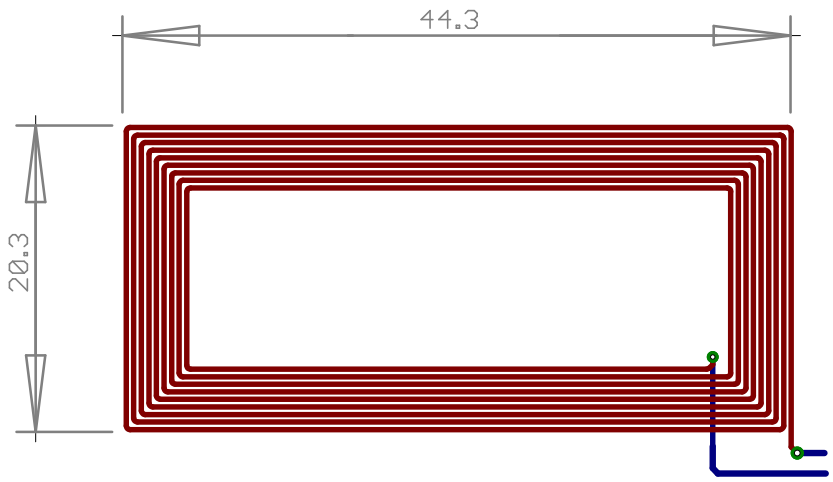


Bild 4.1: *Antennenspule des Zellensensors für den Downlink-Kanal, Bild skaliert um Faktor $s = 2.0$, Angaben in mm*

die benötigte diskrete Parallelkapazität für den Abgleich des Resonanzkreises auf einen Wert von etwa 11,6 pF angenähert werden. Zum Vergleich, der simulativ ermittelte Wert (siehe 3.3) betrug 13 pF.
Somit ist anzunehmen, dass der berechnete Induktivitätswert von der Antennenspule hinreichend genau, mit dem der gefertigten Antennenspule, übereinstimmt.

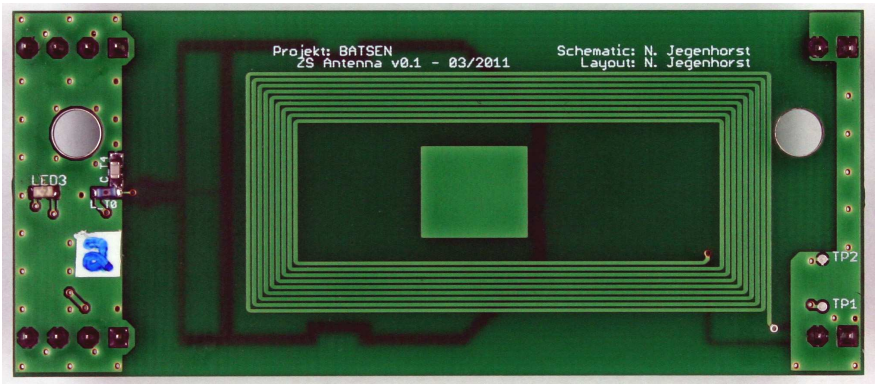


Bild 4.2: *Realisierte Antennenspule auf der Leiterplatte „BATSEN ZS Antenna v0.1“ für den Downlink-Kanal des Zellensensors*

4.1.2 Antenne Uplink-Kanal

Die für den *UHF*-Transmitter „SI4012“ des Herstellers *SI* benötigte Antenne ist im Vergleich zu der Antennenspule des *DL*-Kanals wesentlich komplexer.
Der Grund hierfür ist, dass der differentielle Verstärkerausgang des Transmitters als optimale Lastimpedanz einen Wert von $(33.8 + j126.8)\,\Omega$ erfordert. Üblicherweise wäre eine derartige Impedanz mit Hilfe eines diskreten Anpassungsnetzwerkes, in Verbindung mit einer Schleifenantenne, zu erzielen. Zur Reduzierung der benötigten Bauelemente ist ein Verzicht dieses Netzwerkes wünschenswert.
Im Rahmen der näheren Untersuchung zur Eignung des *UHF*-Transmitters ist ein Demokit (4010-DASKF 434) von *SI* untersucht worden. Dieses enthält Zwecks Demonstration neben einem *UHF*-Empfänger eine Fernbedienung, die unter anderem mit dem gleichen *UHF*-Transmitter kombiniert wird und mit einer *MCU* in einem Multichipgehäuse aufgebaut ist. Als Antenne für den Transmitter wird hierbei eine sogenannte „inductively tapped loop antenna“ bzw. „transformer matched loop antenna“ verwendet. Derartige Antennen (z. B. beschrieben in Microchip (2004)) bieten den Vorteil, dass diese durch Veränderung ihre Geometrie an komplexe Impedanzen anpassbar sind. Neben einer diskreten Spule und einem Kondensator ist lediglich eine weitere Kapazität, realisiert als sogenannter „printed interdigital capacitor“ am Kopf der Leiterschleife, erforderlich.

Die Details zum Entwurf der Antenne von dem betrachtetem Demokits sind dargestellt in Silicon Laboratories (2010a). Dieser erfolgte mit Unterstützung eines elektromagnetischen Feldsimulators. Weitere Details zu dieser Antenne werden in einer im Projekt parallel laufenden Masterarbeit Kube (2011) behandelt.

Für die Realisierung der *UHF*-Antenne des Zellensensors ist, da dies nicht den Schwerpunkt der Arbeit bilden sollte, die in dem Demokit verwendete „inductively tapped loop antenna“ leicht modifiziert (siehe Bild 4.3) übernommen worden.

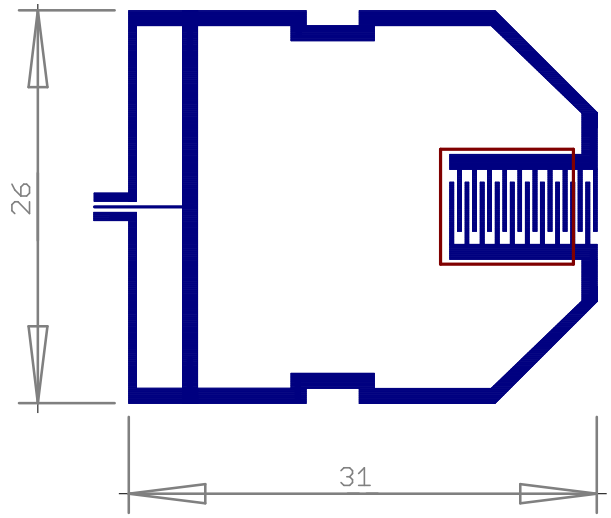


Bild 4.3: *UHF-Antenne auf dem Zellensensor für den Uplink-Kanal, Bild skaliert um Faktor* $s = 2.0$, *Angaben in mm*

Dies konnte insofern durchgeführt werden, da die Geometrie der Antenne zu der Geometrie des geplanten Zellensensors, inklusive der bereits vorgestellten Antennenspule für den *DL*-Kanal, kompatibel ist. Eine eventuell auftretende Abweichung von der optimalen Antennenimpedanz ist, durch die automatische Antennenanpassung des Transmitters, in einem gewissen Bereich kompensierbar.
Da der verwendete *UHF*-Transmitter auf beliebige Trägerfrequenzen in dem Bereich von 27 bis 960 MHz einstellbar ist, kann ebenso alternativ das *ISM*-Frequenzband 868 MHz mit einer entsprechend angepassten Antenne (siehe Silicon Laboratories (2010a)) genutzt werden.
Das erstellte Layout der Platine für die beiden Antennen der *DL*- und *UL*-Kanäle ist dargestellt im Anhang B.3, sowie der dazugehörige Schaltplan im Anhang B.1.2. Zur Illustration ist in dem Bild 4.4 die Unterseite der realisierten Leiterplatte veranschaulicht.

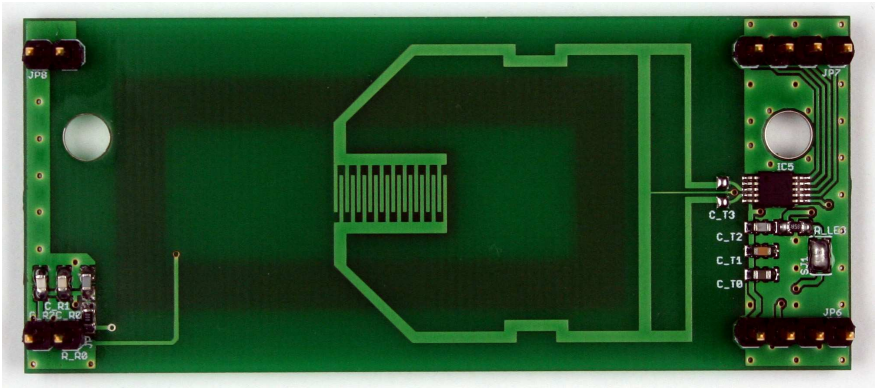


Bild 4.4: *Realisierte Antenne auf der Unterseite der Leiterplatte „BATSEN ZS Antenna v0.1“ für den Uplink-Kanal des Zellensensors, mit UHF-Transmitter (rechts) und Anpassungsnetzwerk für die DL-Antennenspule (links)*

4.1.3 Bauelemente

Für die Realisierung des entwickelten Zellensensors sind die folgenden, bereits beschriebenen, integrierten Halbleiterbauelementen erforderlich:

- 1× Mikrocontroller MSP430F235 von *TI*
- 1× *UHF*-Transmitter SI4012 von *SI*
- 1× Komparator TLV3491 von *TI*
- 1× Temperatursensor TMP102 von *TI*
- 1× *DC-DC-Wandler* TPS61201 von *TI*

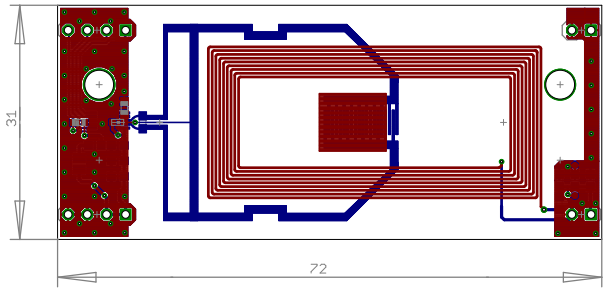
Weiterhin notwendig sind die nachfolgend aufgelisteten diskreten Halbleiterbauelemente:

- 6× Schottkydiode 1N5711
- 9× Siliziumdiode FDLL914
- 1× Zenerdiode BZX84-C11
- 1× *HF*-Transistor BFR183W
- 5× Kleinsignal-Transistoren BC847C
- 2× N-Kanal Mosfet 2N7002

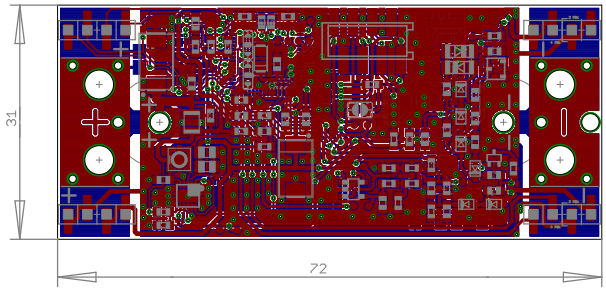
Sowie die übrigen diskreten Bauelemente, zwei Spulen, 46 Kondensatoren, 45 Widerstände und drei optionale *LED*. In der Summe sind somit insgesamt 125 Bauelemente für die Realisierung des Zellensensors notwendig. Dies entspricht zwar nicht unmittelbar dem Ziel eines möglichst minimalen Realisierungsaufwands, im Hinblick auf eine spätere Chip-Integration lässt sich dieser vermutlich, aufgrund der verwendeten Bauelemente, stark reduzieren.

4.1.4 Platinenentwurf

Wie bereits beschrieben, sollen zwei getrennte Leiterplatten für den Aufbau des Zellensensors erstellt werden. Für die elektrische Verbindung zwischen den beiden Leiterplatten ist, um die vorhandene Geometrie der Anschlussflächen beizubehalten, eine leichte Verbreiterung der Platinen um 11 mm nötig gewesen. Wie in dem Bild 4.5 dargestellt, lassen sich so vier SMD-Buchsenleisten auf der unteren Platine, für die Kontaktierung der oberen Platine positionieren.



(a) Obere Leiterplatte „BATSEN ZS Antenna v0.1“

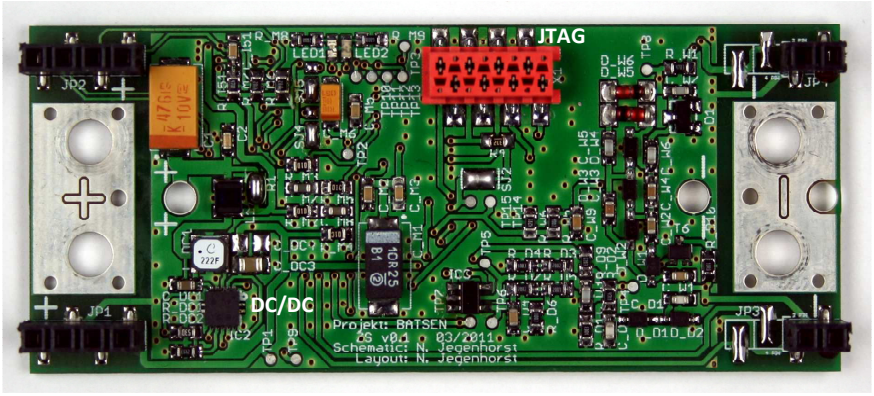


(b) Untere Leiterplatte „BATSEN ZS v0.2“

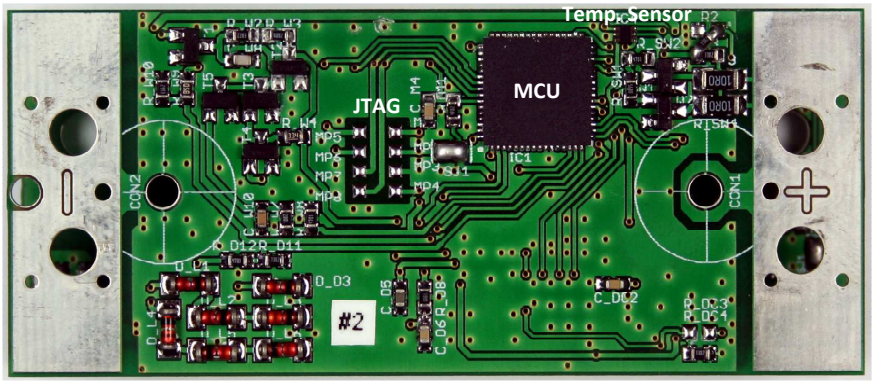
Bild 4.5: *Layout der Leiterplatten für den konzipierten Zellensensor, Bild skaliert um Faktor* $s = 1.0$*, Angaben in mm*

Neben den beibehaltenen Anschlussflächen für den Nadeladapter, ist zusätzlich eine SMD-Federleiste für die Kontaktierung des Programmieradapters vorgesehen.

Gegenüber dem bisherigen Zellensensor beinhaltet der jetzige zudem einen mechanischen Verpolungsschutz, der durch eine zusätzliche Bohrung in der Kontaktfläche für die negative Zellenspannung realisiert ist.
Das Bild 4.6 zeigt die untere aufgebaute Leiterplatte des Zellensensors mit den beschriebenen Elementen. Die obere Leiterplatte ist bereits zuvor im Bild 4.2 und Bild 4.4 gezeigt worden.



(a) Oberseite



(b) Unterseite

Bild 4.6: *Realisierte Leiterplatte „BATSEN ZS v0.1“ des konzipierten Zellensensors*

Sämtliche erstellte Schaltpläne und Layouts für den Zellensensor sind in dem Anhang B.1 dargestellt.

4.2 Praktischer Aufbau der Reader-Datenlogger-Einheit

In den vorherigen Abschnitten und Kapiteln ist stets der Zellensensor als primäres Element betrachtet worden. Für die Komplettierung der Zellen-Sensorik bedarf es zusätzlich einer zentralen Gegenstelle, der Reader-Datenlogger-Einheit, zur Datenerfassung und Koordination der Zellensensoren, sowie der zentralen Strommessung.

Diese Einheit, dargestellt in dem Bild 4.7, besteht aus einer Datenlogger-Einheit, einem *UHF*-Empfänger und einer Reader-Einheit, welche allesamt in den folgenden Abschnitten näher erläutert werden.

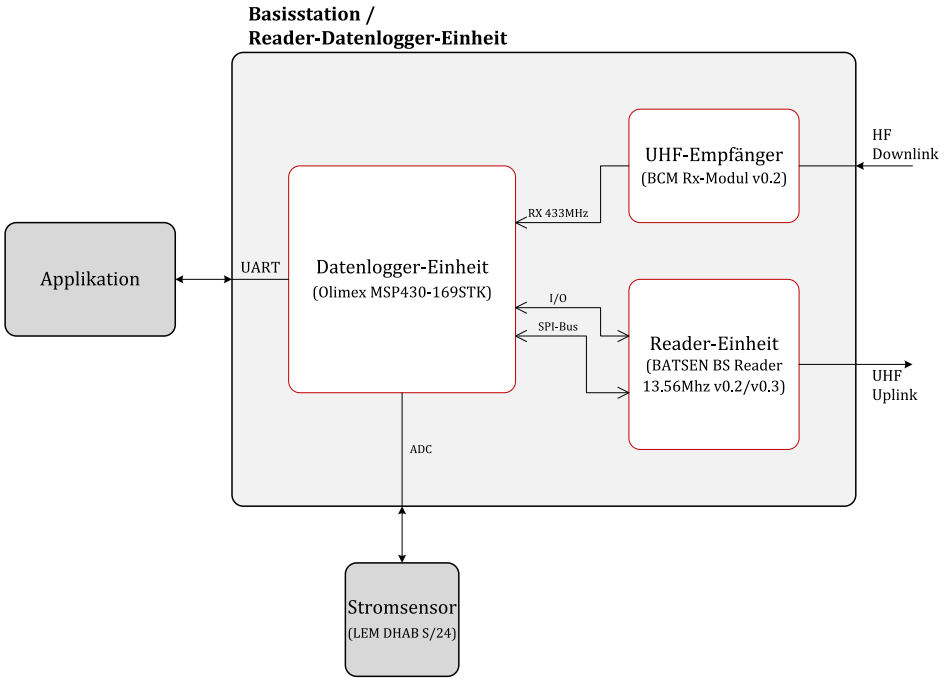


Bild 4.7: *Blockschaltbild der Reader-Datenlogger-Einheit (Basisstation) mit den verwendeten Komponenten*

Zur Steuerung der Reader-Datenlogger-Einheit und zur Weitergabe der Messdaten, verfügt diese über eine serielle Schnittstelle, wodurch die Anbindung an Matlab oder ein Terminal möglich ist.

4.2.1 Aufbau der Datenlogger-Einheit

Die Datenlogger-Einheit ist realisiert durch ein modifiziertes Entwicklungsboard vom Typ „MSP430-169STK“ des Herstellers Olimex. Der ursprüngliche Aufbau

entstammt der Vorarbeit Plaschke (2008) und ist anschließend weiter modifiziert worden in der Arbeit Püttjer (2011).

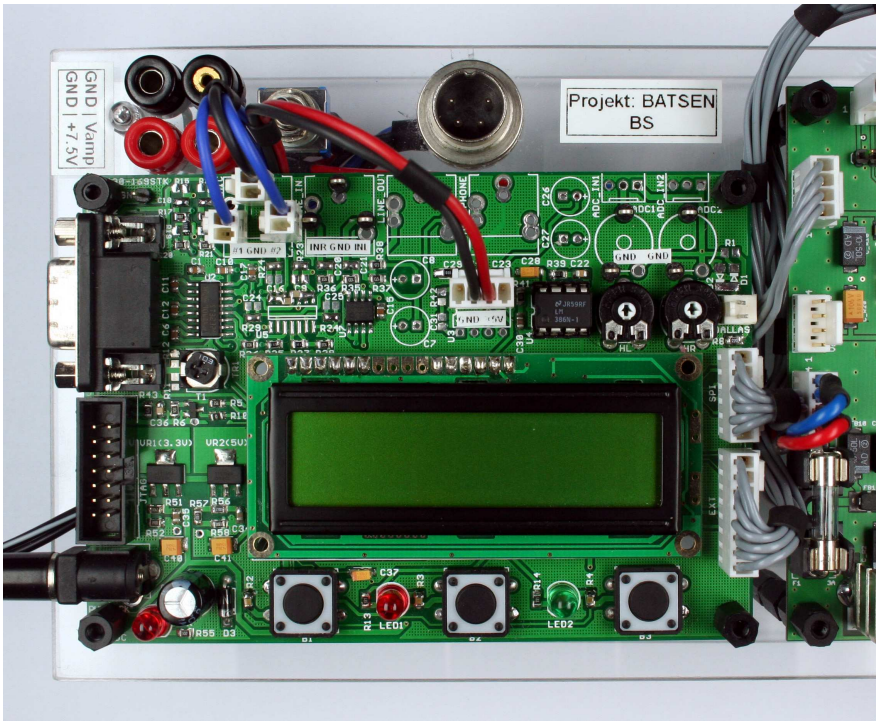


Bild 4.8: *Datenlogger-Einheit der aufgebauten Basisstation*

Die wichtigsten Bestandteile des Entwicklungsboards für die Anwendung sind:

- Mikrocontroller der MSP430-Serie (MSP430F169) von *TI*
- Anzeigeeinheit, bestehend aus einem LC-Display
- Serielle RS-232 Schnittstelle
- JTAG-Programmierschnittstelle
- 3× Taster für Anwendereingaben

An dem Entwicklungsboard sind weiterhin im Rahmen dieser Arbeit einige Modifikationen vorgenommen worden. Alle nötige Veränderungen, vom Originalzustand ausgehend, sind dargestellt im Anhang B.2.5 in dem zugehörigen Schaltplan. In dem Bild 4.8 ist der realisierte Aufbau der Datenlogger-Einheit dargestellt.
Für den Datenempfang auf dem *UL*-Kanal ist mit der Datenlogger-Einheit ein *UHF*-Empfänger-Modul verbunden, welches ebenfalls der Vorarbeit Plaschke (2008) entstammt. Dieses Modul verwendet einen 433 MHz Hybrid Empfänger Modul vom Typ „DR5100“ bzw. „DR3100“ des Herstellers RFM. Der dazugehörige Schaltplan

und das Layout (siehe Anhang B.2.4) sind ebenfalls nochmals geringfügig modifiziert worden, einerseits vorab in der Arbeit Püttjer (2011), sowie in dieser Arbeit.
An die Datenlogger-Einheit lässt sich zudem ein Stromsensor vom Typ „DHAB S/24“ des Herstellers LEM (siehe Püttjer (2011)) für die zentrale Messung des Batteriestroms anschließen.

4.2.2 Aufbau der Reader-Einheit

Die Reader-Einheit der Basisstation ist notwendig für die Realisierung des *DL*-Kanals zum Zellensensor. Steuerbar durch die Datenlogger-Einheit muss diese ein mit *OOK* modulierbares *HF*-Feld, der Frequenz 13,56 MHz, mit ausreichender Feldstärke erzeugen.
Wie nachfolgend dargestellt, erfolgt der Aufbau unter der Verwendung von aus dem *RFID*-Bereich üblichen integrierten Bausteinen für Reader. Diese haben den Vorteil, dass neben einer digitalen Kontrolleinheit bereits die analogen Komponenten für ein *Front-End* in einem Baustein zusammengefasst sind. Zusätzlich unterstützen diese üblicherweise die Erkennung von Lastmodulation für den frequenzgleichen *UL*-Kanal, welche von dem konzipierten Zellensensor ebenso als Alternative zum *UHF*-Uplink unterstützt wird.

4.2.2.1 Auswahl des Reader-IC

Folgende Anforderungen ergaben sich bei der Auswahl eines passenden kommerziellen Reader-IC:

- Trägerfrequenz: 13,56 MHz
- Modulationsverfahren: *OOK*
- Übertragungsrate: 1 . . . 10 kBaud
- Ausgangsleistung: $\geq$ 200 mW
- Direkter Zugriff auf das *Front-End*
- Schnittstelle zu MCU: *SPI* und/oder Parallel

Die benötigte Ausgangsleistung des *HF*-Senders stützt sich hierbei auf Vorversuche mit dem bereits beschriebenem Labormuster in Verbindung mit einem Funktionsgenerator für die Erzeugung des Trägersignals. Ab einer Leistung von etwa 110 mW konnte ein *Wake-Up* ausgelöst werden.
Ein direkter Zugriff aus das *Front-End* ist erforderlich, um die Kodierung, sowie

den Frameaufbau für den *DL*-Kanal unabhängig von standardisierten Übertragungsverfahren, wie z. B. ISO15693, vornehmen zu können. Als Schnittstelle für den Informationsaustausch stehen seitens der Datenlogger-Einheit nur die beiden genannten Möglichkeiten zur Auswahl.

Entsprechend dieser Vorgaben ist schließlich das Reader-IC „TRF7960“ des Herstellers *TI* ausgewählt worden. Für die externe Beschaltung existieren entsprechende Vorgaben in dem dazugehörigem Datenblatt (siehe Texas Instruments (2006b)), an denen sich der durchgeführte Schaltungsentwurf (siehe Anhang B.2.1) orientiert. Die so entstandene Reader-Einheit ist in dem Bild 4.9 dargestellt.

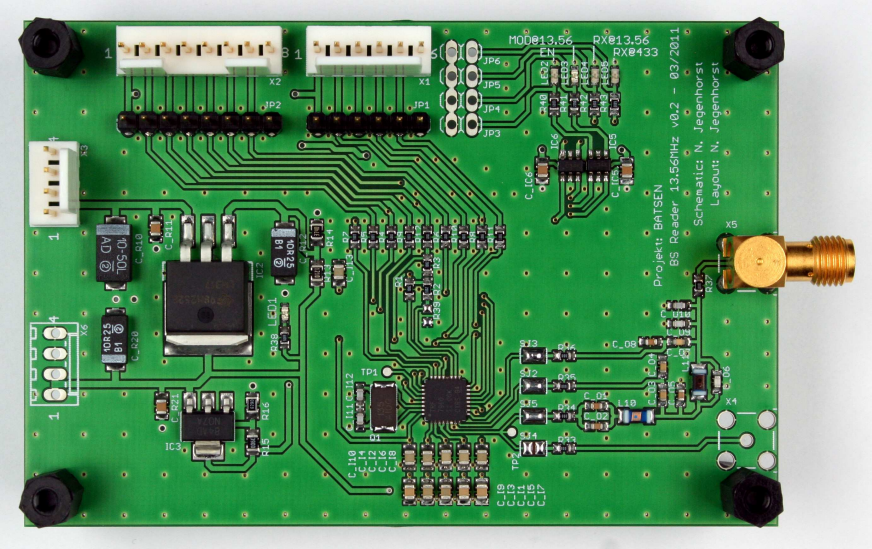


Bild 4.9: *Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.2“*

4.2.2.2 Leistungsverstärker

In Folge eines fehlgeschlagenen Funktionstestes (siehe Abschnitt 4.4.3) mit der realisierten Zellen-Sensorik, ist als Problemlösung eine Erhöhung der Sendeleistung der Reader-Einheit festgelegt worden. Es zeigte sich in einem Versuch mit einem Leistungs-Funktionsgenerator, dass eine Sendeleistung von etwa 750 mW für die gewünschte Umgebung ausreichend ist.

Da eine Recherche nach passenden kommerziell verfügbaren Leistungsverstärkern nicht die gewünschten Resultate brachte, fiel die Entscheidung zur Realisierung eines eigens aufgebauten Leistungsverstärkers. Als schaltungstechnische Vorgabe diente hierzu eine auf das verwendete Reader-IC bezogene Applikationsschrift

(siehe Texas Instruments (2008)) von *TI*. Der darin beschriebene Leistungsverstärker verfügt über eine maximale Ausgangsleistung von etwa 4 W, sodass eine ausreichende Leistungsreserve vorhanden ist.
Entsprechend der Applikationsschrift konnte der Schaltplan, sowie das Layout der bereits realisierten Reader-Einheit (siehe Anhang B.2.1, B.2.6), um den Leistungsverstärker (siehe Bild 4.10) ergänzt werden. Der so entstandene Schaltplan befindet sich im Anhang B.2.2 mit dem dazugehörigem Layout im Anhang B.2.7.

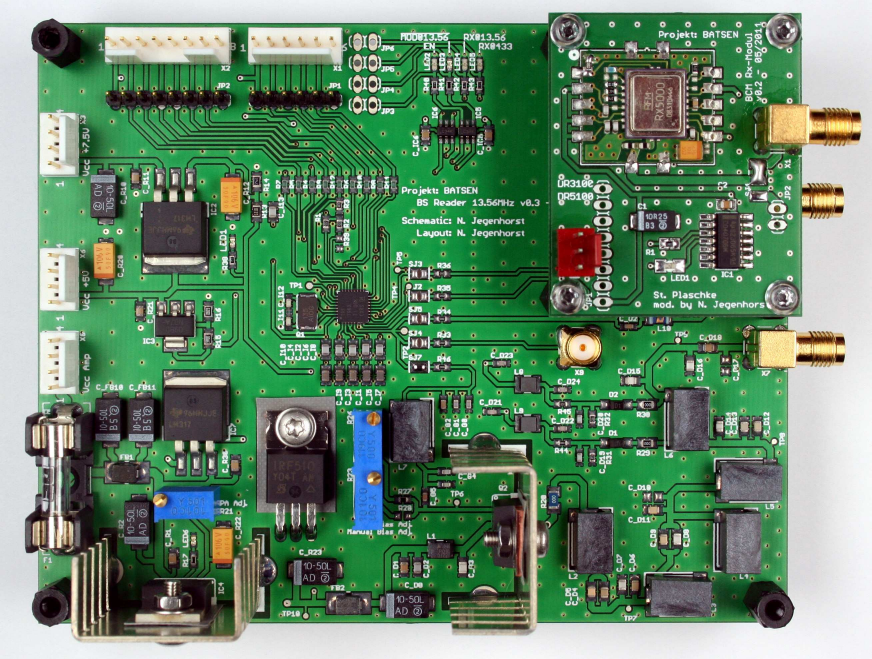


Bild 4.10: *Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.3“ mit Leistungsverstärker und aufmontierten UHF-Empfänger-Modul „BCM Rx-Modul v0.2“*

4.2.2.3 Antenne Downlink-Kanal

Die beiden realisierten Reader-Einheiten mit und ohne Leistungsverstärker, haben die Gemeinsamkeit, dass sie als Antenne eine an 50 Ω angepasste Antennenspule in Reihenresonanz benötigen.
Der Entwurf von der Geometrie für die Antennenspule erfolgte nach dem gleichen Verfahren, wie für die Antennenspule des *DL*-Kanals vom Zellensensor bereits im Abschnitt 4.1.1 gezeigt worden ist. Hierbei ergaben sich die in der Tabelle 4.2 aufgelisteten Parameter, die zu der im Bild 4.11 gezeigten Geometrie führen.

Zur Erhöhung der Reichweite des Systems könnte man annehmen, dass man die Fläche der Antennenspule noch größer dimensioniert. Dabei muss jedoch die Größe der Antennenspule des Zellensensors im Verhältnis betrachtet werden. Hierzu

Parameter	Größe
Induktivitätswert (soll)	4,2 μH
Länge der Spule	80,0 mm
Breite der Spule	53,0 mm
Leiterbahnabstand	0,5 mm
Leiterbahnbreite	1,0 mm
Leiterbahndicke	35 μm
Substratdicke	1,0 mm
Windungszahl	6 Wdg.
Segmentanzahl	24 Sgt.

Tabelle 4.2: *Parameter der Antennenspule für den Downlink-Kanal der Reader-Einheit*

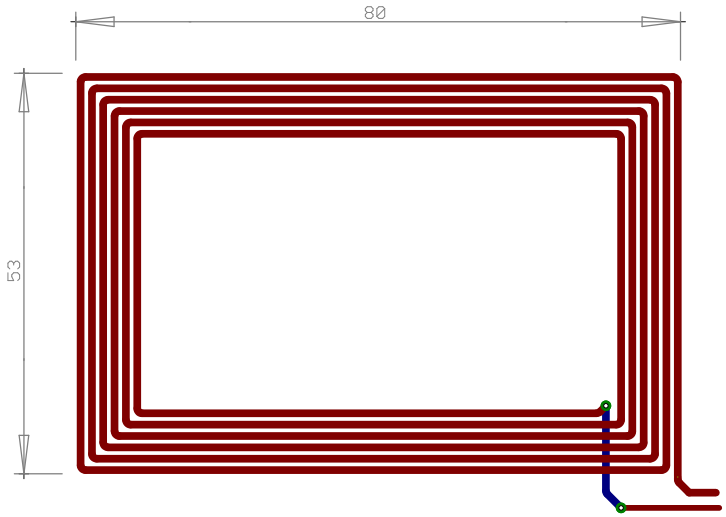


Bild 4.11: *Antennenspule für den Downlink-Kanal der Reader-Einheit, Bild skaliert um Faktor $s = 1.0$, Angaben in mm*

sind in dem Bild 4.12 die beiden realisierten Antennenspulen nebeneinander dargestellt. Eine wesentlich vergrößerte Spule der Basisstation könnte bei einem ungünstig angeordneten System die Spule des Zellensensors verdecken. Das heißt, es würde zu einer Abschattung kommen.

Um die Anpassung der Impedanz der Reihenresonanz an die 50 Ω Eingangsimpedanz zu erreichen, dient ein sogenanntes Pi-Glied, entnommen aus (Krischke, 2001, S. 133). Für die Berechnung des Pi-Gliedes ist eine ideal auf die Trägerfrequenz von 13,56 MHz abgestimmte Reihenresonanz angenommen worden, deren Impedanz in etwa den Kupferverlusten ($\approx 3{,}66\,\Omega$) entspricht. Für die Abstimmung der Reihen-

resonanz bedarf es nach praktischer Erprobung an der aufgebauten Leiterplatte, einer Serienkapazität von 23,3 pF.

Das folgende Bild 4.12 zeigt die realisierte Leiterplatte mit der Antennenspule von der Reader-Einheit, im direktem Vergleich zu der Antennenspule des Zellensensors. Wie man dabei erkennen kann, ist das Größenverhältnis der Antennenspule zueinander derart gewählt, dass die der Reader-Einheit nicht wesentlich größer ist. Anderenfalls könnte es zur sogenannten Abschattung der Spule des Zellensensors kommen, wenn sich diese unmittelbar unter der größeren Spule der Reader-Einheit befindet.

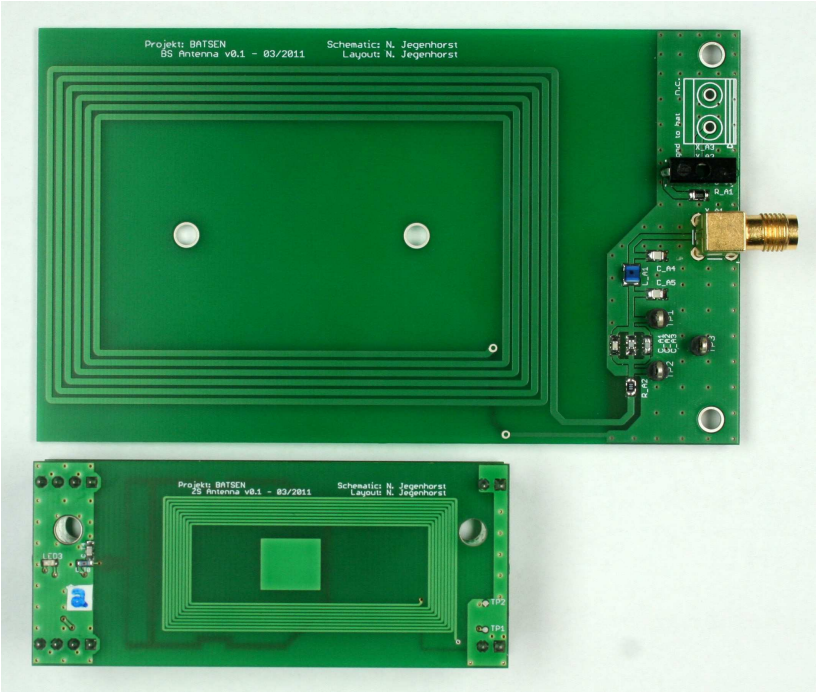


Bild 4.12: *Realisierte Antennenspule für den Downlink-Kanal der Reader-Einheit (oben) und Antennenspule für den Downlink-Kanal des Zellensensors (unten)*

Der Schaltplan für die Leiterplatte der Antennenspule befindet sich im Anhang B.2.3, sowie das dazugehörige Layout im Anhang B.2.8.

4.2.3 Zusammenschaltung der Komponenten

Das Bild 4.13 zeigt den realisierten Aufbau der Reader-Datenlogger-Einheit (Basisstation) mit den relevanten Anschlüssen. Bei der Verdrahtung ist darauf geachtet worden, dass der Aufbau möglichst modular bleibt und einen einfachen Austausch der Komponenten erlaubt.

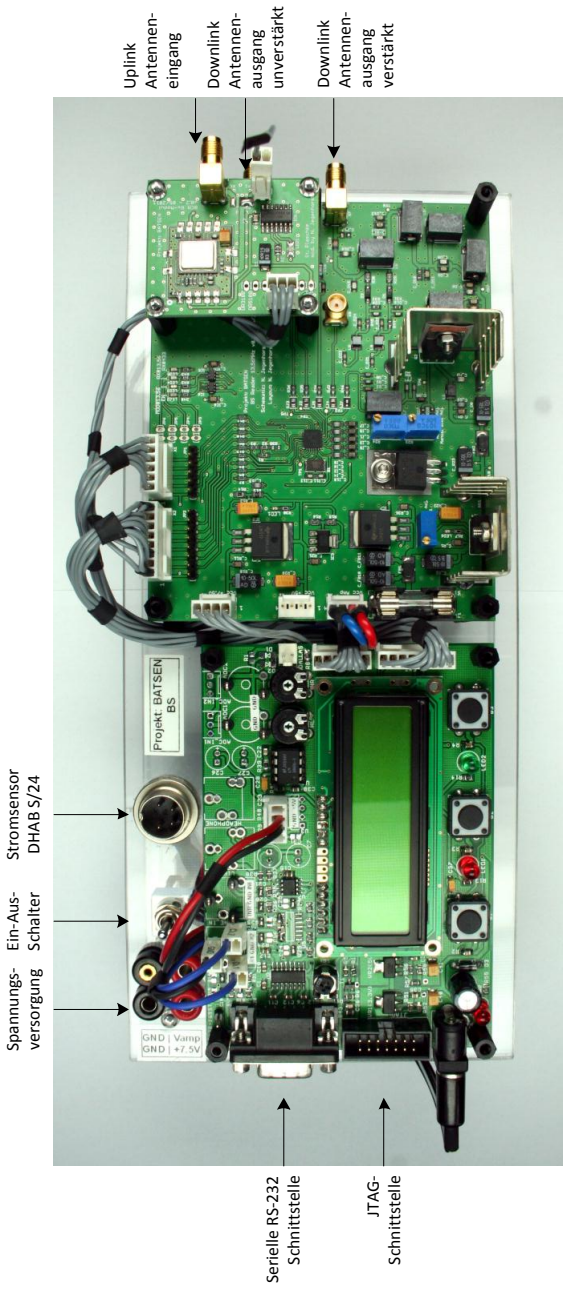


Bild 4.13: *Zusammenschaltung der Komponenten zur Reader-Datenlogger-Einheit (Basisstation)*

Für den Betrieb der Basisstation wird eine Spannungsversorgung von 7,5 V bei 250 mA für die Datenlogger-Einheit und Teilen der Reader-Einheit benötigt. Bei Verwendung des Leistungsverstärkers benötigt dieser zusätzlich eine Spannungsversorgung wahlweise von 5 bis 18 V bei 1 A. Das *UHF*-Empfänger-Modul, sowie der Stromsensor, beziehen ihre Betriebsspannung von der Datenlogger-Einheit.

4.3 Softwaredesign und -realisierung

Nach der praktischen Realisierung der Zellen-Sensorik im vorherigen Kapitel 4 folgt in diesem Kapitel die Beschreibung des Softwaredesigns inklusive der wichtigsten Aspekte für deren Umsetzung.
Behandelt wird dabei unter anderem der Frameaufbau für den *DL*- und *UL*-Kanal, die Koordination der Zellen-Sensorik, sowie die Messdatenverarbeitung mit Matlab.

4.3.1 Software für den Mikrocontroller des Zellensensors

Der Mikrocontroller des Zellensensors soll verschiedene Funktionalitäten beinhalten. Dazu zählt die Dekodierung des *DL*-Signals von der Empfängerschaltung, die Messwertaufnahme der Spannungs- und Temperaturmessung und die Ansteuerung des *UHF*-Transmitters für den *UL*-Kanal. Neben diesen grundlegenden Funktionalitäten bedarf es einer von empfangenen Kommandos abhängigen Ablaufsteuerung, welche vordefinierte Funktionen ausführen kann, die ebenfalls durch Kommandos in ihren Parametern einstellbar sein können.

4.3.1.1 Dekodierung des Downlink-Empfangssignals

Neben der reinen *Wake-Up*-Funktionalität des realisierten *DL*-Empfängers verfügt dieser zusätzlich über eine Demodulationsschaltung, wie bereits im Abschnitt 3.3.1.2 gezeigt. Somit ist es möglich, von der Basisstation gesendete komplexere Informationen mit einer Übertragungsrate von 10 kHz zu empfangen.

Frameaufbau

Für die geordnete Übertragung der Informationen ist ein Frame, angelehnt an den Frame des *UL*-Kanals (siehe nächsten Abschnitt 4.3.1.2), entworfen worden. Dieser Frame setzt sich aus den folgenden Elementen zusammen:

- *Wake-Up*-Präambel:
  - Dauer: 5 ms Dauerstrich gefolgt von 0,2 ms Unterbrechung
  - Wird für den *Wake-Up* benötigt, falls Zellensensor(en) sich im deaktiviertem Zustand befinden
  - Zeitdauer setzt sich zusammen aus:
    - Aufladen der Spannungsvervielfacherschaltung $t_{on,wk}$

  - Hochfahren der Betriebsspannung durch den *DC-DC-Wandler* $t_{on,dcdc}$
  - Initialisieren der *MCU* $t_{on,mcu}$

- Start of Frame (SOF):
  - Dauer: 2,4 ms
  - Zunächst 800 μs Dauerstrich als Run-In
  - Gefolgt von 8 Bit mit dem dezimalen Wert eins zur Synchronisation der Zeitbasis
- Adresse:
  - Dauer: 4,0 ms
  - 20 Bit Zieladresse für Adressierung separiert nach:
    - Unicast - einzelner Zellensensoren
    - Multicast - einer Gruppe von Zellensensoren (nicht realisiert)
    - Broadcast - aller Zellensensoren
- Kommando:
  - Dauer: 1,6 ms
  - 8 Bit zur Kodierung von 256 Kommandos und Nachrichten
- Parameter:
  - Dauer: 2,4 ms
  - 12 Bit zur Kodierung von Parametern eines zugehörigem Kommandos bzw. einer Nachricht
- Paritätsprüfung (CRC):
  - Dauer: 1,6 ms
  - 8 Bit Paritätswert aus einer byteweise Verknüpfung der Nutzdaten

Zusammengefasst ergeben diese Elemente den in dem Bild 4.14 dargestellten Frameaufbau.

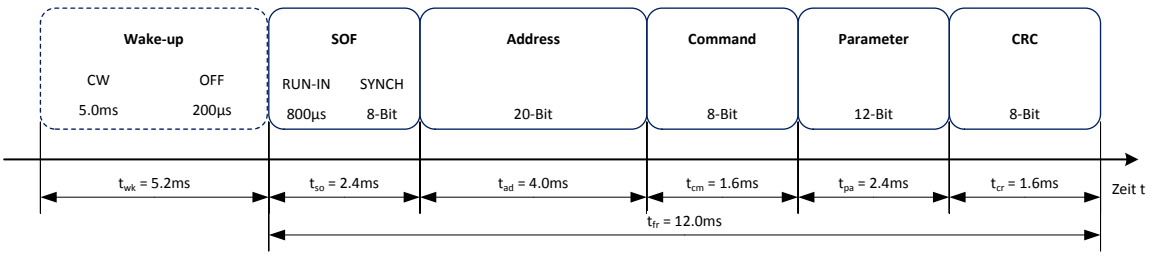


Bild 4.14: *Aufbau des Frames für die Informationsübertragung zwischen der Basisstation und den Zellensensoren (Downlink)*

Dekodierung

Als Codierungsverfahren für den *DL*-Kanal wird, wie auch für den *UL*-Kanal, die Manchester-Codierung verwendet. Dieses bietet zum einen den Vorteil, dass das übertragene Signal keinen Gleichanteil aufweist, zum anderen lässt sich das Taktsignal aus dem Modulationssignal rekonstruieren. Nachteilig ist hingegen die Halbierung der verfügbaren Datenrate, entsprechend 5 kBaud bei diesem System.
Wie in dem Bild 4.15 bereits angedeutet, ist es nur möglich, bedingt durch die Demodulationsschaltung mit einfacher Flankendetektion (siehe Abschnitt 3.3.1.2), die steigenden Flanken des Modulationssignals zu erkennen. Somit ergeben sich die

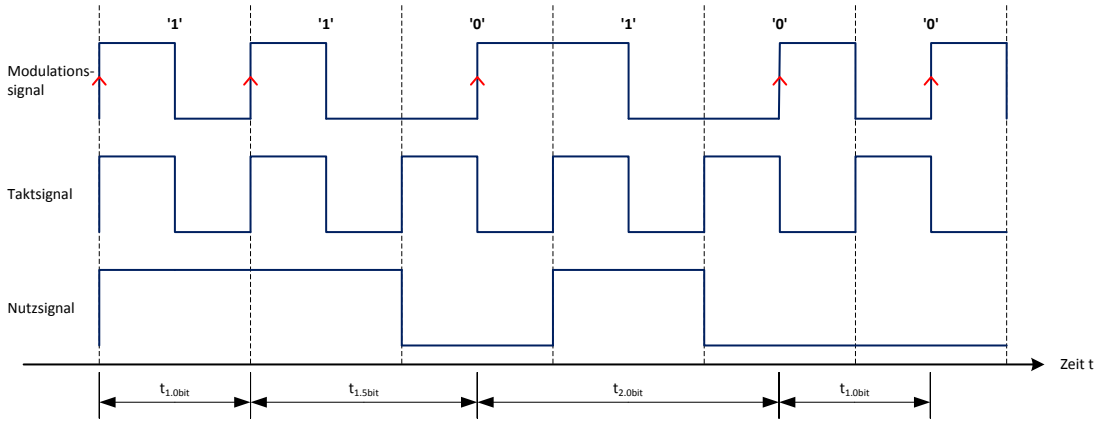


Bild 4.15: *Zustände der Manchester-Codierung, insbesondere bei einfacher Flankendetektion*

drei dargestellten Zeiten zwischen den steigenden Flanken, anhand denen das Modulationssignal zu decodieren ist. Der *Start of Frame (SOF)*-Block in dem Frame sorgt dabei für einen bekannten definierten Anfangszustand, von dem ausgehend eine Zuordnung der gemessenen Zeiten zu den einzelnen Bits des Nutzsignals möglich ist. Hierzu ergibt sich die folgende logische Zuordnung:

- Zeitdauer $1.0 \times t_{bit}$:
  - aktueller Wert entspricht letztem Wert
- Zeitdauer $1.5 \times t_{bit}$:
  - letzter Wert gleich '0', dann zwei Bits der Folge [1 0] erkannt
  - letzter Wert gleich '1', dann ein Bit des Wertes '1' erkannt
- Zeitdauer $2.0 \times t_{bit}$:
  - zwei Bits der Folge [1 0] erkannt

Dabei ist anzumerken, dass aufgrund der einfachen Flankendetektion eine logische eins immer erst mit dem folgendem Bit erkannt werden kann. So ist es notwendig einen definierten Endzustand zu erzeugen, wenn also das letzte Bit in einem Frame

eine logische eins ist, bedarf es am Ende einer zusätzlichen steigenden Flanke durch ein weiteres Bit.
Diese Art der Dekodierung entstammt der ursprünglichen Software von der Basisstation (siehe Plaschke (2008)), welche in einer weiteren Arbeit (siehe Püttjer (2011)) modifiziert wurde.

Implementierung

Das von der Demodulationsschaltung mit einem kompatiblem Logikpegel ausgegebene Signal, wird direkt an den Eingang des Zeitgeber „Timer A“ weitergeleitet. Dieser Zeitgeber, eingestellt auf eine Schrittweite von 2 μs, wird im sogenannten „Capture-Mode“ betrieben. Das heißt bei jeder steigenden Flanke wird der Zählerstand gespeichert und ein *Interrupt Request (IRQ)* ausgelöst.
Die mit dieser Interruptquelle verbundene *Interrupt Service Routine (ISR)* (siehe Anhang C.14) beinhaltet einen Zustandsautomaten (vgl. Bild 4.16), der die gemessenen Werte des Zeitgebers auswertet und entsprechend der Zuordnung das Nutzsignal rekonstruiert. Innerhalb des *SOF*-Blocks wird anhand der Synchronisationssequenz die mittlere Bitdauer t_{avg}, für die Berechnung der folgenden benötigen Schwellwerte, bestimmt.

$$\begin{aligned} t_{1.0bit,min} &= 1,0 \cdot t_{avg} - 25\% \\ t_{1.0bit,max} &= 1,0 \cdot t_{avg} + 25\% \\ t_{1.5bit,max} &= 1,5 \cdot t_{avg} + 25\% \\ t_{2.0bit,max} &= 2,0 \cdot t_{avg} + 25\% \end{aligned}$$

Somit sind die Schwellwerte bekannt, in denen sich die gemessenen Zeiten t_{cap} für die Rekonstruktion bewegen müssen.

$$\begin{aligned} t_{1.0bit,min} < t_{cap} < t_{1.0bit,max} &\quad \rightarrow \text{Zeit } t_{1.0bit} \text{ detektiert} \\ t_{1.0bit,max} < t_{cap} < t_{1.5bit,max} &\quad \rightarrow \text{Zeit } t_{1.5bit} \text{ detektiert} \\ t_{1.5bit,max} < t_{cap} < t_{2.0bit,max} &\quad \rightarrow \text{Zeit } t_{2.0bit} \text{ detektiert} \end{aligned}$$

Anderenfalls werden die bereits empfangenden Daten verworfen und es wird auf die nächste *SOF*-Sequenz gewartet. Sind alle 48 Bits eines Frames erkannt, folgt zur Verifizierung eine Paritätsprüfung.
Anschließend muss nach einer erfolgreichen Verifizierung anhand der Zieladresse entschieden werden, ob die empfangenen Informationen für den jeweiligen Zellensensor bestimmt sind. Hierzu muss entweder die Zieladresse mit der Sensoradresse übereinstimmten (Unicast), oder es handelt sich um eine definierte Adresse für einen Broadcast.

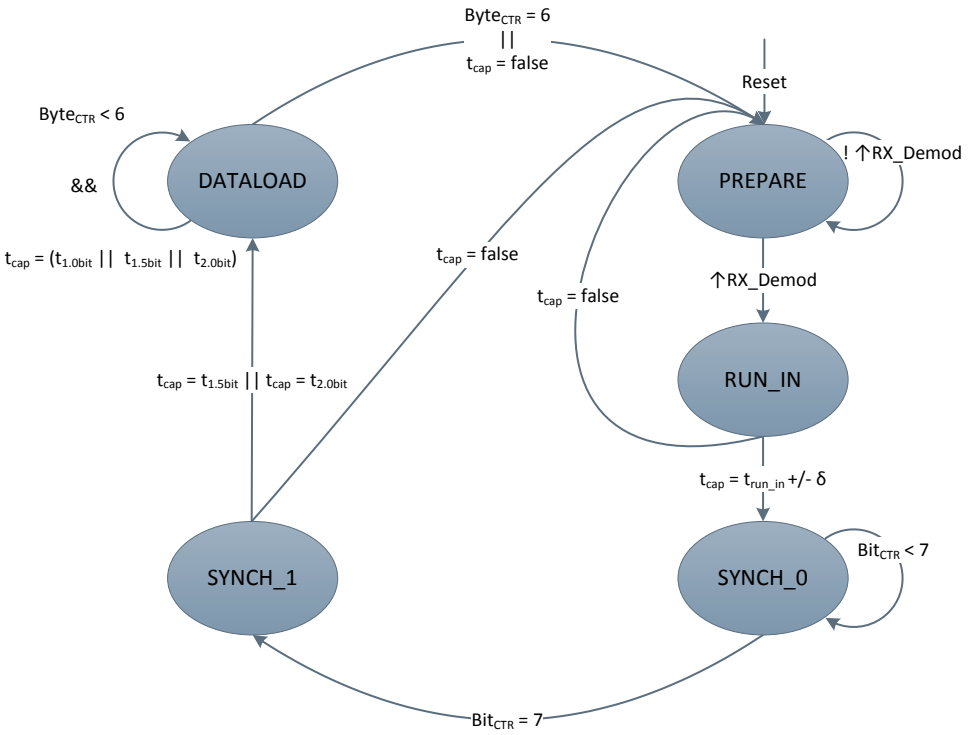


Bild 4.16: *Zustandsfolge des Automaten für die Dekodierung des DL-Empfangssignals auf dem Zellensensor*

Bezieht es sich schließlich um eine an den Zellensensor gerichtete Information, gilt es das empfangene Kommando bzw. die Nachricht anhand des empfangenen Codes zu dekodieren und auszuführen, oder auszuwerten. Ist dies beendet, kann der Datenempfang von Neuem beginnen.
In der Tabelle A.1 im Anhang A befindet sich eine Übersicht über alle implementierten Kommandos bzw. Nachrichten. Weitere Details zu den Kommandos und Nachrichten folgen im Abschnitt 4.3.1.4.

4.3.1.2 Kodierung des Uplink-Sendesignals

Zur grundlegenden Funktionalität des Zellensensors gehört, gemessene Daten, wie primär die Zellenspannung, an die Basisstation zu übertragen. Diese Parameter müssen zunächst, wie im Folgenden dargestellt, in einen geeigneten Frame verpackt werden.

Frameaufbau

Der für den *UL*-Kanal verwendete Frame (siehe Bild 4.17) umfasst die nachfolgend beschriebenen Elemente. Es legt fest in welcher Reihenfolge und mit welcher Länge die einzelnen Informationen zu übertragen sind. Somit kann jeder, dem dieser Frame bekannt ist, die enthaltenen Informationen zumindest separieren. Eine Auswertung der Informationen ist nur möglich, wenn zugleich die Kodierung bekannt ist.

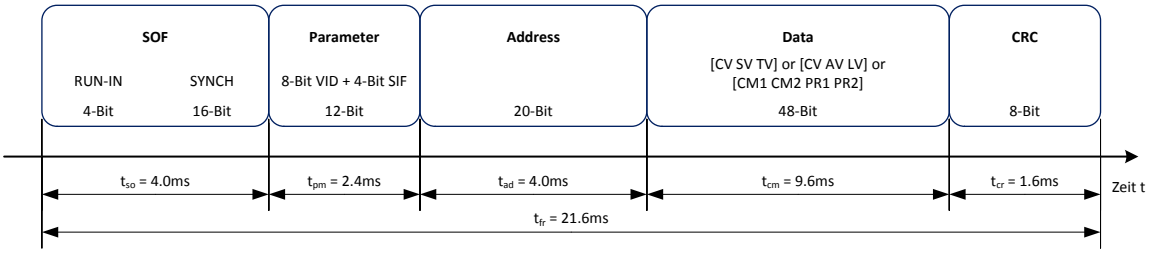


Bild 4.17: *Aufbau des Frames für die Informationsübertragung zwischen den Zellensensoren und der Basisstation (Uplink)*

- Start of Frame (SOF):
  - Dauer: 2,4 ms
  - Zunächst 800 μs (4 Bit) Dauerstrich als Run-In
  - Gefolgt von 16 Bit mit dem dezimalen Wert eins zur Synchronisation der Zeitbasis
- Parameter:
  - Dauer: 2,4 ms
  - 8 Bit *Versions Identifikation (VID)* zur Erkennung des Typs und der Softwareversion
  - 4 Bit *Spezial Informationen Flags (SIF)* ermöglichen Identifikation zur Art der gesendeten Daten
- Adresse:
  - Dauer: 4,0 ms
  - 20 Bit Quelladresse vom Zellensensor
- Daten:
  - Dauer: 9,6 ms
  - 48 Bit
  - Inhalt der Daten ist wie folgt abhängig von den *SIF*
  - $SIF = 0x0$:

    Zellenspannung U_{cell} (CV), Betriebsspannung U_{supply} (SV) und Temperatur T (TV)

- $SIF = 0x1$:

  Zellenspannung U_{cell} (CV), Ausgangsspannung des Tiefpassfilters U_{lp} (LV) sowie der Spannungsvervielfacherschaltung für den *Wake-Up* U_{act_in} (AV)

- $SIF = 0x3$:

  Zuletzt empfangenen zwei Kommandos bzw. Nachrichten (CM1,2) inkl. deren Parameter (PR1,2)

- Paritätsprüfung (CRC):
  - Dauer: 1,6 ms
  - 8 Bit Paritätswert aus einer byteweise Verknüpfung der Nutzdaten

Anhand der in dem Frame enthaltenen *VID* und den *SIF*, ist es dem Empfänger (der Basisstation) möglich die übermittelten Daten auszuwerten, da diesem zugleich die Kodierung bekannt ist. Der beschriebene Frame wird bereits von den vorhandenen Zellensensoren der Klasse 1 verwendet, ursprünglich entwickelt in der Arbeit Plaschke (2008). Durch die Weiterverwendung dieses Frames können die bereits bestehenden Basisstationen unter Kenntnis der Kodierung die transmittierten Daten ebenso auswerten.

Kodierung

Für die Übertragung des Frames dient als Codierungsverfahren, ebenso wie bei dem *DL*-Kanal (vgl. Abschnitt 4.3.1.1), die Manchester-Codierung. Dies ist insbesondere bei den vorhandenen Zellensensoren der Klasse 1 entscheidend für die empfängerseitige Dekodierung, da bei diesen die tatsächliche Übertragungsrate abhängig vom Systemtakt der *MCU* ist. Dieser kann nur durch einen internen RC-Oszillator der *MCU* erzeugt werden und unterliegt dadurch größeren Schwankungen.
Durch die Verwendung des neuen *UHF*-Transmitters „SI4012“ von *SI* ist eine mögliche Abweichung der Übertragungsrate auf ein Minimum reduziert. Die Übertragungsrate wird durch den Transmitter selber bestimmt, welcher einen präzisen integrierten LC-Oszillator aufweist und nicht wie zuvor durch die *MCU*. Somit wäre es möglich auf die Manchester-Codierung zu verzichten und die Datenrate zu erhöhen. Aus Kompatibilitätsgründen wird diese jedoch beibehalten.
Als Datenrate soll ebenfalls aus Kompatibilitätsgründen weiterhin 5 kBaud, entsprechend einer Übertragungsrate von 10 kBaud, verwendet werden. Letztere wäre seitens des Transmitters auf maximal 50 kBaud, bei Verwendung von *FSK* Modulation hingegen bis auf 100 kBaud, zu erhöhen.

Implementierung

Um die gewünschten Daten über den *UL*-Kanal zu transmittieren, bedarf es einer Reihe von notwendigen Implementierungen.

- I^2C- bzw. *SMBus*-Schnittstelle
- Funktionen zur Handhabung des Transmitters
- Initialisierung des Transmitters
- Initialisierung des Frames
- Zusammenführen von Daten in den Frame
- Berechnung des Paritätswerts für den Frame
- Manchester-Codierung des Frame

Primär gilt es, die Kommunikation mit dem Transmitter über die integrierte I^2C-Schnittstelle der *MCU* zu realisieren. Hierfür sind zwei Funktionen (vgl. Anhang C.8) „i2c_bus_write()“ und „i2c_bus_read“ implementiert worden, die eine universelle Kommunikation, ohne die Verwendung von Interrupts, erlauben. Diesen Funktionen muss lediglich die Adresse des anzusprechenden Objektes, die Anzahl der Bytes und ein Zeiger auf die Quell- bzw. Zieldaten übergeben werden. Somit können diese Funktionen gleichsam für die Kommunikation, mit dem ebenfalls an den I^2C-Bus angeschlossen Temperatursensor, dienen. Details zum Handling der I^2C Schnittstelle sind in Texas Instruments (2004) zu finden.

Für die Handhabung des Transmitters bedarf es zum einen der Möglichkeit Einstellungen in einzelnen Registern des Transmitters zu setzen, zum anderen verschiedene Kommandos auszuführen.
Mit Hilfe der implementierten Funktion „tx433_set_prop()“ (siehe Anhang C.18) lassen sich, durch den Übergabewert bestimmt, Einstellungen an den Registern des Transmitters vornehmen. Die einzustellenden Parameter sind allesamt zuvor in der zugehörigen Header-Datei (siehe Anhang C.19) fest definiert worden. Eine Änderung der Parameter zur Laufzeit ist nicht erforderlich. Zum Setzen einer Einstellung sendet die Funktion zunächst ein Byte, welches angibt, dass eine Einstellung gesetzt werden soll, gefolgt von einem Byte, das beschreibt, welche Einstellung vorzunehmen ist. Anschließend folgen bis zu sechs Byte mit dem eigentlichen Wert für die Einstellung. Um zu verifizieren, ob der Vorgang erfolgreich war, nicht unbedingt, ob die Daten fehlerfrei waren, wird anschließend ein Byte mit Statusinformationen von dem Transmitter gelesen.
Weiterhin bedarf es für den Betrieb verschiedene Kommandos auszuführen. Jedes Kommando ist als eine eigene Funktion realisiert (vgl. Anhang C.18), welche mindestens ein Byte mit dem entsprechende Kommando sendet und anschließend ein

Byte mit der Statusinformationen ausließt. Folgende Kommandos sind realisiert worden:

- Starten einer Transmission
- Stoppen einer Transmission
- Initialisieren des FIFO-Speichers
- Senden von Daten an den FIFO-Speicher
- Setzen einer Interruptquelle
- Auslesen des Interrupt Status Registers
- Abfragen des Betriebszustands
- Setzen eines bestimmten Betriebszustands
- Setzen der Status *LED* des Transmitters

Die Initialisierung des Transmitters geschieht mit der Funktion „tx433_init()“ (vgl. Anhang C.18), diese setzt alle notwendigen Einstellungen mit Hilfe der zuvor beschriebenen Funktion „tx433_set_prop()“ entsprechend der definierten Parameter. Die wichtigsten Einstellungen wären:

- Verwendung des internen LC-Oszillators als Taktgeber
- Trägerfrequenz 433,965 MHz
- *OOK*-Modulation
- 10 kBaud Übertragungsrate
- maximale Sendeleistung

Um das Ende einer Transmission zu erkennen, wird zudem die zugehörige Interruptquelle freigeschaltet. Durch die Interrupt-Leitung zu der *MCU* kann die zugehörige *ISR* (vgl. AnhangC.12) hierauf reagieren und über die I^2C-Schnittstelle das Interrupt Status Register auslesen.

Zur Speicherung der Daten, die über den *UL*-Kanal transmittiert werden sollen, wird zwischen zwei Datensätzen unterschieden. Der Vektor „g_tx433.framedata[]“ enthält byteweise die veränderlichen Daten des zu senden Frames, von den Parametern angefangen bis zu dem Paritätswert, insgesamt 88 Bits. In dem Vektor „g_tx433.txdata[]“ befindet sich hingegen der gesamte Frame (108 Bits), inklusive des *SOF*. Diese Daten sind allerdings Manchester-Codiert, da der Transmitter keine eigenständige Manchester-Codierung unterstützt.
Zur Initialisierung der Vektoren dient die Funktion „frame_tx433_init()“ (vgl. Anhang C.18), diese setzt beispielsweise den *SOF*-Block und die Adresse.

Für das Zusammenführen von zu transmittierenden Daten in den Frame sind drei Funktionen implementiert (vgl. Anhang C.18), welche die bereits zuvor beschriebenen Datensätze in den Datenblock des Frame speichern. Anschließend, wenn alle nötigen Informationen in den Frame eingefügt sind, gilt es den Paritätswert der Daten mit der Funktion „frame_tx433_calc_crc()“ zu berechnen. Bevor der Frame nun in den FIFO-Speicher des Transmitters mit der Funktion „tx433_cmd_fifo_set()“

zu laden ist, müssen die in dem Vektor „g_tx433.framedata[]“ enthaltenen Daten mit der Funktion „frame_tx433_code_manchester()“ noch in den Vektor „g_tx433.txdata[]“ kopiert und dabei Manchester-Codiert werden.

Zwar ist die Ansteuerung des Transmitters mit einigen softwaretechnischen Realisierungsaufwänden verbunden. Nach der Initialisierung erhält man dafür den Vorteil, dass die *MCU* keine Prozessorzeit für die Modulation des *UL*-Signals benötigt, verbunden mit höheren möglichen Übertragungsraten und diversen Einstellungsmöglichkeiten seitens des Transmitters. Durch die modulare Art der Implementierung konnte eine hoher Grad an Flexibilität erreicht werden, um das System an veränderliche Aufgaben anzupassen.

4.3.1.3 Messwertaufnahme

Die primäre Funktion des Zellensensors ist es, die Parameter Spannung und Temperatur von der zugehörigen Batteriezelle zu erfassen. Außerdem gilt es weitere Parameter aufzuzeichnen, die einen Rückschluss über den Betriebszustand des Zellensensors selber zulassen.

Zur Erfassung von Parametern der physikalischen Größe „elektrische Spannung“, verfügt die verwendete *MCU* über einen integrierten *ADC* mit einer dazugehörigen Referenzspannungsquelle. Hierbei handelt es sich um einen *ADC*, der nach dem Prinzip der sukzessiven Approximation (SAR) arbeitet und eine Auflösung von 12 Bit besitzt. Weitere Parameter zur Spezifikation des *ADC* siehe (Texas Instruments, 2007a, S. 62). Durch eine einstellbare interne Abtast-Halte-Schaltung und einen Analog Multiplexer lassen sich nacheinander verschiedene Eingangskanäle auswählen und entsprechend mit der nötigen Abtastzeit (Texas Instruments, 2004, S. 571) aufnehmen.

Um die jeweiligen Spannungsmessungen durchzuführen, ist für jede Messgröße eine eigene Funktion (vgl. Anhang C.3), wie im Folgendem dargestellt, implementiert worden. Diese getrennten Funktionen ermöglichen den *ADC* stets entsprechend zu konfigurieren, sowie die Messwerte abhängig der Spannungsteiler zu skalieren und die Ergebnisse in die zugehörigen Variablen zu speichern. Eine Umrechnung der Messergebnisse in die Einheit Volt findet nicht statt. Die Rohdaten des *ADC* sollen vielmehr unverändert an die Basisstation übertragen und erst dort umgerechnet werden. Da es systembedingt keine Vorteile bieten würde, werden für die Messwertaufnahmen keine Interrupts seitens des *ADC* verwendet.

- Zellenspannung U_{cell}:
  - Funktion „sample_ubat()“
  - Abtastzeit mindestens 37,57 μs, gewählt 64 μs

  - Spannungsteiler mit Faktor 0,5 vor *ADC*
  - Auflösung 1,221 mV/LSB

- Betriebsspannung U_{supply}:
  - Funktion „sample_udcdc()“
  - Abtastzeit mindestens 5,13 μs, gewählt 16 μs
  - Spannungsteiler mit Faktor 0,5 vor *ADC*
  - Auflösung 1,221 mV/LSB

- Ausgangsspannung für *Wake-Up* U_{act_in}:
  - Funktion „sample_uact_in()“
  - Abtastzeit mindestens 80,82 μs, gewählt 96 μs
  - Spannungsteiler mit Faktor 0,25 vor *ADC*
  - Auflösung 2,442 mV/LSB

- Ausgangsspannung Tiefpassfilter U_{lp}:
  - Funktion „sample_lp()“
  - Abtastzeit mindestens 271,85 μs, gewählt 384 μs
  - Ohne einen Spannungsteiler vor *ADC*
  - Auflösung 0,610 mV/LSB

Für die Erfassung der physikalischen Größe „Temperatur“ dient ein separater Temperatursensor des Typs „TMP102“ vom Hersteller *TI*. Dieser bietet eine Auflösung von 0,0625 °C/LSB, bei einer Genauigkeit von 0,5 °C, weitere Informationen siehe Texas Instruments (2007b). Die Kommunikation mit dem Sensor erfolgt ebenso wie bei dem Transmitter über den I^2C-Bus, wobei die beschriebenen Funktionen für die Ansteuerung der I^2C-Schnittstelle weiterverwendet werden.
Das Auslesen der Temperaturinformation erfolgt mit Hilfe der Funktion „temp_sensor_get_temp()“ (vgl. Anhang C.16), wobei die Information im Zweierkomplement dargestellt wird. Standardmäßig führt der Sensor im Intervall von 4 Hz eine Temperaturmessung aus, deren Wert dann aus dem entsprechenden Register auszulesen ist. Das Intervall, wie auch andere Einstellungen, lässt sich durch die implementierte Funktionen „temp_sensor_get_control_reg()“ und „temp_sensor_set_control_reg()“ auslesen bzw. konfigurieren. Eine Weiterverarbeitung der Temperaturinformation erfolgt nicht, die Umrechnung findet ebenso auf der Basisstation statt.

Bei allen Messwertaufnahmen gilt es, mögliche Störquellen zu beachten. Zum einen bedingt durch die pulsierende Stromaufnahme und den damit verbundenen Schwankungen der Betriebsspannung, zum anderen durch die Transmission selber, verursacht der *UHF*-Transmitter während einer Transmission erhebliche Störungen.
Eine weitere Störquelle ist der *DC-DC-Wandler*, der ebenso für die Erzeugung der

Betriebsspannung dient. Da es sich hierbei um einen Schaltregler handelt, verursacht dieser Störungen entsprechend der variablen Schaltfrequenz. So ist es unvermeidlich, den *DC-DC-Wandler* während einer Messung der Zellenspannung unbedingt abzuschalten. Selbstverständlich ist darauf zu achten, dass der Transmitter in diesem Moment nicht sendet und die Dauer des Ausschaltens begrenzt ist, sonst bricht die Betriebsspannung zusammen.
Diese beschriebenen Abhängigkeiten müssen von der im nächsten Abschnitt beschriebenen Ablaufsteuerung beachtet werden, welche ebenso für die zeitliche Koordination der Messwertaufnahmen zuständig ist.

Weiterhin ist für eine verlässliche Messwerterfassung eine Kalibration unbedingt erforderlich. Lediglich bei der Temperaturinformation ist dies primär nicht notwendig, da der verwendete Temperatursensor bereits eine absolute Genauigkeit von 0,5 °C aufweist. Bei den Spannungsmessungen hingegen sollte eine Kalibrierung, abhängig von der Betriebsspannung, der Temperatur und der Messgröße selber, durchgeführt werden.
Vorarbeiten diesbezüglich existieren bereits (siehe Ilgin (2011)), die Ergebnisse sind jedoch aus zeitlichen Gründen nicht mehr in diese Arbeit eingeflossen.

4.3.1.4 Kommandos, Messages und Funktionen

Durch die neue Eigenschaft des konzipierten Zellensensors, einen *DL*-Kanal für eine bidirektionale Kommunikation zur Verfügung zu haben, eröffnen sich eine Reihe von Möglichkeiten, die bei den Zellensensoren der Klasse 1 bisher nicht möglich waren.
So können nun theoretisch sämtliche Parameter zur Messwertaufnahme und für die Datentransmission von der Basisstation, durch die Verwendung von Kommandos und Messages, verändert werden. Zuvor waren die Parameter nur einmalig während der Programmierung der Zellensensoren einstellbar.
Ebenso können nun andere Übertragungsverfahren für den *UL*-Kanal angewendet werden. Die Verwendung von pseudozufälligen Sendezeitpunkten ist weitestgehend obsolet.

Zur formalen Beschreibung des Informationsaustauschs zwischen den Zellensensoren und der Basisstation, ebenso wie zwischen der Basisstation und den Steuergeräten des betreffenden Fahrzeugs, ist die Verwendung einer abstrahierenden Sprache, *Battery Monitoring and Control Language (BMCL)* genannt, vorgesehen. Wie in Riemschneider u. Vollmer (2010) dargestellt, wird darin zwischen drei Gruppen wie folgt separiert:

- Kommandos:
  - Aussendung von einem höherem Level, in diesem Fall der Basisstation
  - Gerichtet an ein niedrigeres Level, entsprechend den Zellensensoren
  - Unterscheidung zwischen adressierten und unadressierten Kommandos
  - Können Parameter setzen, Funktionen auslösen oder Nachrichten anfordern
- Messages:
  - Aussendung hingegen von niedrigerem Level ausgehend, hier den Zellensensoren
  - Gerichtet an ein höheres Level, also entsprechend die Basisstation
  - Autonome Aussendung oder kommandierte Aussendung möglich
  - Besitzen stets eine Quelladresse, wenn nötig eine Zieladresse
- Funktionen:
  - Sind auf jedem Level möglich
  - Initiierung durch Kommandos mit und ohne Parameter
  - Autonomes starten möglich
  - Beendigung durch Kommando, vorgegebener Zeitspanne oder selbstständig

Diese Aufteilung ist formal für das erschaffene System anwendbar. Lediglich bei den Messages gibt es eine Anpassung. So können diese ebenso von der Basisstation, dem höherem Level, an die Zellensensoren gesendet werden. Da hierbei eine Trennung zwischen den Kommunikationskanälen vorhanden ist, sowie es zugleich nur einen Sender geben sollte, bedarf es dabei nur der Zieladresse und keiner Quelladresse. Ein Beispiel für solch eine Nachricht wäre die Aussendung eines Frames zur Synchronisation der Zellensensoren, auf einen Taktgeber der Basisstation.
Bei der Gruppe der Funktionen handelt es sich nicht zwangsweise um einzelne ausführbare Funktionen. Vielmehr können damit komplexere Funktionalitäten angestoßen werden, wie beispielsweise die Messwertaufnahme inklusive der Datentransmission, was eine Vielzahl von Funktion zugleich erfordert.

In dem Anhang A befindet sich die Tabelle A.1, mit allen von dem realisiertem Zellensensor unterstützen Kommandos bzw. Messages. Deren Bedeutung wird größtenteils im nächsten Abschnitt erläutert.

4.3.1.5 Ablaufsteuerung

Für den Zellensensor galt es eine Ablaufsteuerung zu entwickeln, welche die zuvor beschriebenen implementierten Funktionen nutzt und zu verschiedenen Funktionalitäten, abhängig von Kommandos und Messages, zusammenfügt. Dabei müssen zugleich die Restriktionen des realisierten Zellensensors beachtet werden. Die wichtigsten sind im Folgendem kurz zusammengefasst.

- keine *DL*-Dekodierung während einer *UL*-Transmission
- keine *DL*-Dekodierung wenn der Transmitter in den Zustand „tune“ wechselt
- keine Messwertaufnahme während Transmitter in den „tune“ Zustand wechselt oder im Zustand „tx“ ist
- keine Messung der Zellenspannung wenn *DC-DC-Wandler* aktiv
- keine *SVS*-Überwachung während einer *UL*-Transmission oder Messung der Zellenspannung

Die ersten beiden Restriktionen beruhen auf der Gegebenheit, dass eine von dem Zellensensor selber ausgehende Übertragung auf den *UL*-Kanal in die Antennenspule des *DL*-Kanals einkoppelt und so ein regulären Empfang in dem Moment nicht möglich ist. Gleiches gilt, wenn der Transmitter in den Zustand „tune“ wechselt und dieser dabei kurzzeitig neben dem internen LC-Oszillator ebenfalls den Antennenkreis abgleicht.
Eine Überwachung der Betriebsspannung während einer *UL*-Transmission durch die *SVS* würde, aufgrund des auftretenden Einbruchs der Betriebsspannung, von den gewähltem Schwellwert abhängig zu einem Neustart des Systems führen. Eine Beschreibung der übrigen Restriktionen ist bereits zuvor erfolgt.

Im Folgenden werden vier verschiedene Funktionalitäten des Zellensensors näher erläutert, an welchen gleichzeitig das verwendete Übertragungsverfahren für den *UL*-Kanal verdeutlicht wird.

Standard-Messbetrieb

Der Standard-Messbetrieb soll, wie der Name bereits ausdrückt, die standardmäßige Messfunktion inklusive der Transmission der Messdaten zu der Basisstation ausführen. Hierzu ist das in dem Bild 4.18 dargestellte zeitliche Verhalten geplant und umgesetzt worden.
Es beginnt mit dem Empfang einer *DL*-Transmission, deren Inhalt dekodiert und das enthaltene Kommando bzw. Message ausgeführt oder ausgewertet wird. Da es

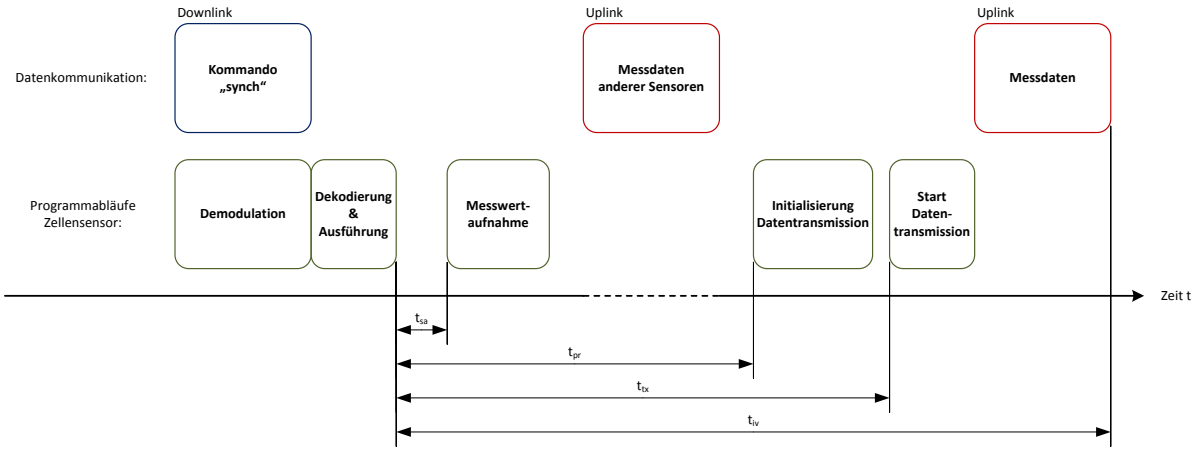


Bild 4.18: *Zeitlichen Abläufe des Zellensensors im Standard-Messbetrieb („Normal-Mode“)*

sich hierbei um das Kommando „synch“ handelt, geht der Zellensensor in den Zustand „Normal-Mode“ über und startet für die eigene Systemzeit einen Zeitgeber. Alternativ, wenn der Zustand bereits vorliegt und der Zeitgeber aktiv ist, erfolgt zur Synchronisation nur ein Rücksetzten des Zeitgebers.
Von diesem Zeitpunkt ausgehend bestimmt sich, wann welche Aktion auszuführen ist. Zunächst ist nach der Zeit t_{sa} die Aufnahme der Messwerte geplant. Anschließend folgt zu dem Zeitpunkt t_{pr} die Übergabe der zu transmittierenden Daten in den FIFO-Speicher des Transmitters, verbunden mit dessen Zustandswechsel von „standby“ in den „tune“ Zustand. Frühstens 6,6 ms nach dem Zustandswechsel ist der Abgleichvorgang des Transmitters beendet und dieser bereit für eine Transmission, welche zu dem definierten Zeitpunkt t_{tx} von der *MCU* initiiert wird.
Die beschriebenen Aktionen wiederholen sich schließlich nach der Zeit t_{iv} der Intervalldauer, mit der die Messwertaufnahme ausgeführt wird. Die drei Parameter t_{sa}, t_{tx} und t_{iv} sind durch entsprechende Kommandos vorab von der Basisstation vorzugeben. Der Parameter t_{pr} wiederum, wird abhängig von t_{tx} selbstständig bestimmt. Sind diese Werte einmal übermittelt, können diese durch das Kommando „save_set“ dauerhaft in den Flash der *MCU* geschrieben werden. Sobald ein *Wake-Up* den Zellensensor aktiviert, lädt dieser diese Werte und ist damit unmittelbar bereit für die Messwertaufnahme.

Alle Aktionen, die zu einem definierten Zeitpunkt auszuführen sind, werden initiiert durch den Zeitgeber für die Systemzeit, den „Timer_B“. Zu den betreffenden Zeitpunkten generiert das jeweilige Vergleichsregister einen *IRQ*, was zur Ausführung der zugehörigen *ISR* (siehe Anhang C.15) führt.
Da der Zeitgeber für die Systemzeit die höchste Priorität besitzt, die Dekodierung des *DL*-Empfangssignals durch den Zeitgeber „Timer_A“ aber ebenfalls einer hohen Priorität bedarf, ist eine minimale Ausführungszeit für die *ISR* des Zeitgebers „Timer_A“ erforderlich. Zur Lösung des Problems erfolgt die eigentliche Ausfüh-

rung der Aktionen durch Software-Interrupts mit niedrigerer Priorität in den *ISR* des Port 2, getriggert durch die *ISR* des Zeitgebers für die Systemzeit. Durch das gleichzeitige Zulassen von verschachtelten Interrupts[1] an ausgewählten Stellen kann der Empfang von Kommandos und Messages nahezu immer sichergestellt werden. Ausgenommen sind natürlich die bereits aufgeführten Restriktionen.
Um diese Vielzahl von auszuführenden *ISR* zugleich im zeitlichen Bezug zueinander darzustellen, dient der in dem Bild 4.19 gezeigt Ablaufplan.
Neben der ersten Spalte für das Hauptprogramm, entspricht jede weitere Spalte einer *ISR*. Der Zeitgeber für die Systemzeit verfügt über drei Vergleichsregister, welche jeweils eine eigene *ISR* auslösen können. Da mehr zeitabhängige Aktionen als Vergleichsregister vorhanden sind, ist die Mehrfachnutzung eines Vergleichsregisters erforderlich. Welche Aktion auszuführen ist, entscheidet sich abhängig von der kommandierten Funktionalität und dem Zustand des globalen Zustandsautomaten (siehe Bild 4.20).
Für den Zeitgeber zur Dekodierung des *DL*-Empfangssignals, sowie der damit verbundenen Ausführung bzw. Auswertung der Kommandos und der Messages, existieren zwei *ISR*, wovon die Zweite nur ausgeführt wird im Falle einer Zeitüberschreitung, ausgelöst durch eine Störung der *DL*-Übertragung.
Die *ISR* vom Port 2, verwendet für die Software-Interrupts, kann von jedem Interruptflag, welches zu den Pins des Ports gehört, ausgelöst werden. Innerhalb der *ISR* (vgl. C.13) erfolgt entsprechend eine Selektion welche Aktion auszuführen ist, abhängig des auslösenden Flags.
In der letzten Spalte ist die *ISR* vom Port 1 eingetragen. Deren einziger *IRQ* kann nur von der Interruptleitung des Transmitters ausgelöst werden. Entsprechend nur, wenn eine *UL*-Transmission vollendet ist, oder der Transmitter einen Reset nach dem Einschaltvorgang meldet.

[1] engl. Nested Interrupts

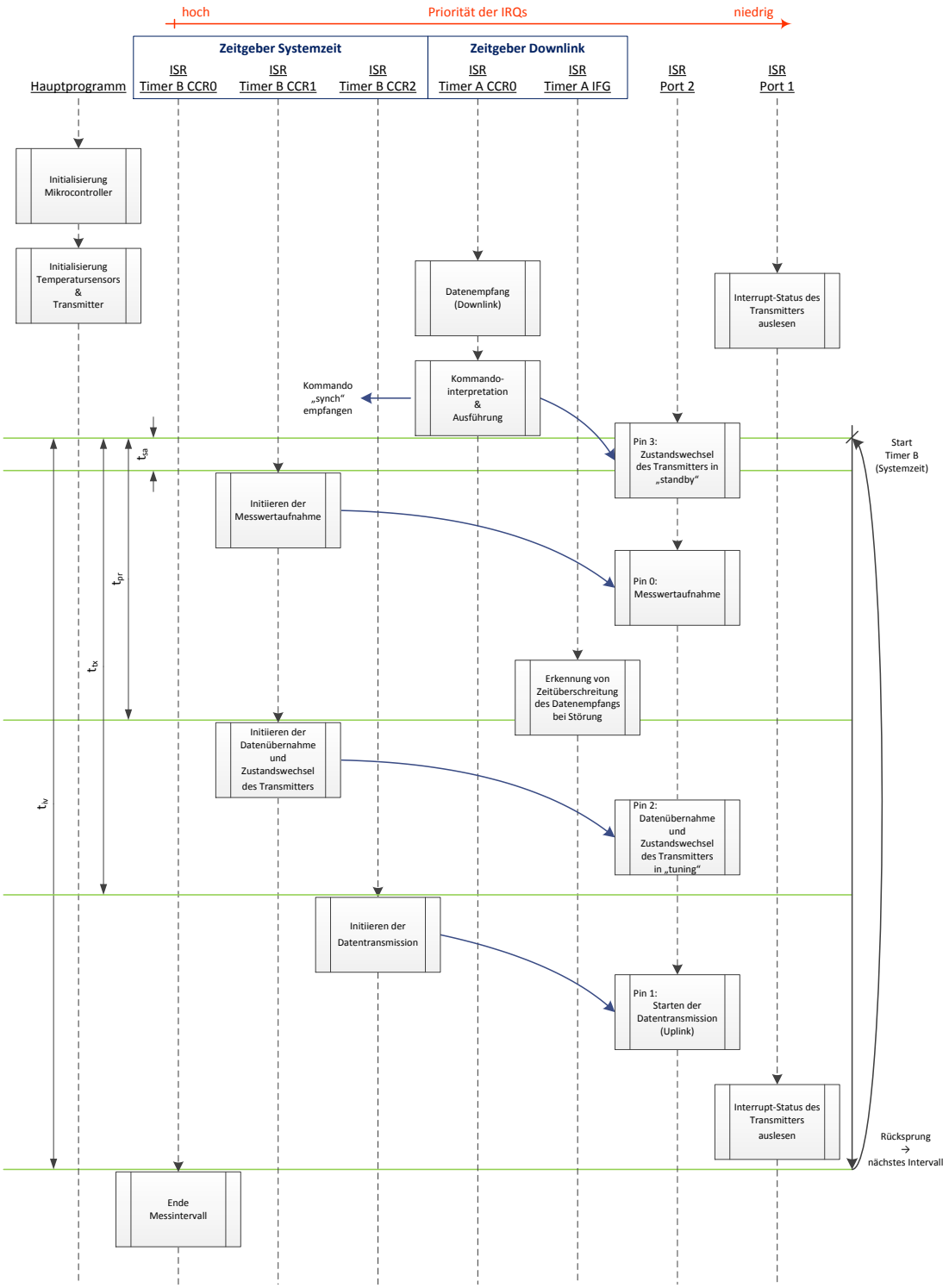


Bild 4.19: *Zeitliche Aufeinanderfolge aller Routinen des Zellensensors im Standard-Messbetrieb („Normal-Mode“)*

In dem Bild 4.20 ist der globale Zustandsautomat des Zellensensors mit den Übergängen für den Standard-Messbetrieb dargestellt.

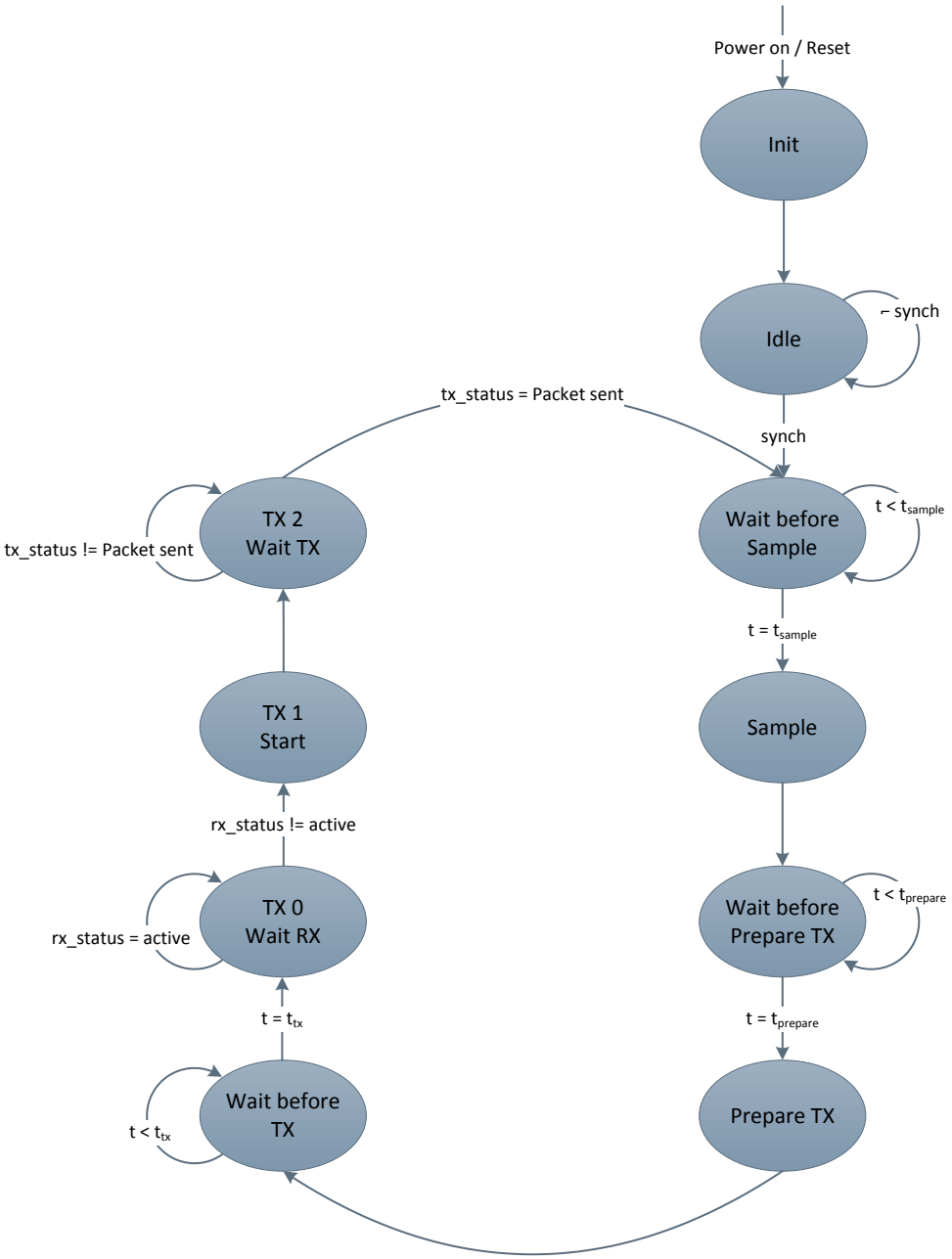


Bild 4.20: *Zustandsfolge des Zellensensors im Standard-Messbetrieb („Normal-Mode“)*

Die Zustandsumschaltung erfolgt üblicherweise durch die Aktionen von ausführenden Routinen. Hingegen bedingen die Routinen des Zeitgebers für die Systemzeit die Auslösung der Software-Interrupts abhängig von dem aktuellen Zustand.

Somit ist zugleich eine Erkennung von Fehlern möglich, wenn nicht der für den jeweiligen Zeitpunkt zu erwartende Zustand vorliegt. Die Folgende Auflistung gibt eine Übersicht über die möglichen Zustände und deren Bedeutung.

- Init:
  - Initialisierung der *MCU*, des Transmitters und Temperatursensors
- Idle:
  - Leerlaufbetrieb, warten auf Kommandos oder Messages von der Basisstation
- Wait before Sample:
  - Warten bis Zeitpunkt zur Messwertaufnahme erreicht
- Sample:
  - Ausführung der Messwertaufnahme und Zusammenführung der Daten in den Frame für den *UL*-Transmission
- Wait before Prepare TX:
  - Warten auf den Zeitpunkt für die Vorbereitung der Transmission
- Prepare TX:
  - Vorbereitung der Transmission, Frame an Transmitter übergeben, Zustandswechsel des Transmitters
- Wait before TX:
  - Warten bis Zeitpunkt zum Initiieren der Transmission erreicht
- TX 0 Wait RX:
  - Zeitpunkt für Transmission erreicht, warten falls *DL*-Kommunikation aktiv, dann *DL*-Dekodierung deaktivieren
- TX 1 Start:
  - Initiierung der Transmission durch Übergabe eines Kommandos an den Transmitter
- TX 2 Wait TX:
  - Warten bis Transmission vollendet, dann *DL*-Dekodierung reaktivieren

Der Zustand „TX 0 Wait RX“ ergibt sich aus einer Restriktion der Basisstation, diese kann analog zu dem Zellensensor nicht gleichzeitig eine *DL*-Transmission aussenden und eine *UL*-Transmission empfangen. So ist es zwingend erforderlich zu warten, bis die Basisstation nicht mehr sendet. Solange die Systemzeit des Zellensensors synchron zur Zeit der Basisstation ist, kann eine *DL*-Transmission zu dem Zeitpunkt üblicherweise ausgeschlossen werden.
Weiteres zur zeitlichen Koordination der Zellensensoren folgt im Abschnitt 4.3.2.3.

Scan-Betrieb

Eine weitere Funktionalität des Zellensensors ist der Scan-Betrieb. Dieser wird verwendet, wenn eine Basisstation nach Zellensensoren in ihrer Umgebung scannt und ermöglicht so die Detektion von verfügbaren Zellensensoren. Eine manuelle Anmeldung der Zellensensoren an der Basisstation durch den Nutzer ist somit nicht erforderlich.
Um den Scan-Betrieb zu starten, sendet die Basisstation das Kommando „scan“ als Broadcast an alle Zellensensoren. Ab diesem Moment transmittiert jeder Zellensensor zu einem pseudozufälligen Zeitpunkt eine durch den Parameter des Kommandos „scan“ festgelegte Anzahl an Frames. Durch die zufälligen Zeitpunkte wird erreicht, dass die Basisstation von jedem Zellensensor einen vollständigen Frame, inklusive der Adresse, mit ausreichender Wahrscheinlichkeit empfängt. Da es sich hierbei um eine Funktion handelt, welche üblicherweise nur einmalig erforderlich ist, z. B. beim Anlernen der Sensoren einer Batterie, wäre eine Mehrfachausführung bei Misserfolg vertretbar.
Der in dem Bild 4.21 gezeigte Zustandsautomat stellt für den beschriebenen Betrieb die Zustandsübergänge des Zellensensors dar.
Zur Berechnung der pseudozufälligen Sendezeitpunkte t_{tx}, innerhalb einer festgelegten Intervalldauer t_{iv}, dient die Funktion „generate_time_tx()“ (vgl. Anhang C.6). Das verwendete Verfahren der pseudozufälligen Sendezeitpunkte entstammt dem bestehenden Zellensensor der Klasse 1 (vgl. Plaschke (2008)) und ist für die Anwendung angepasst worden.

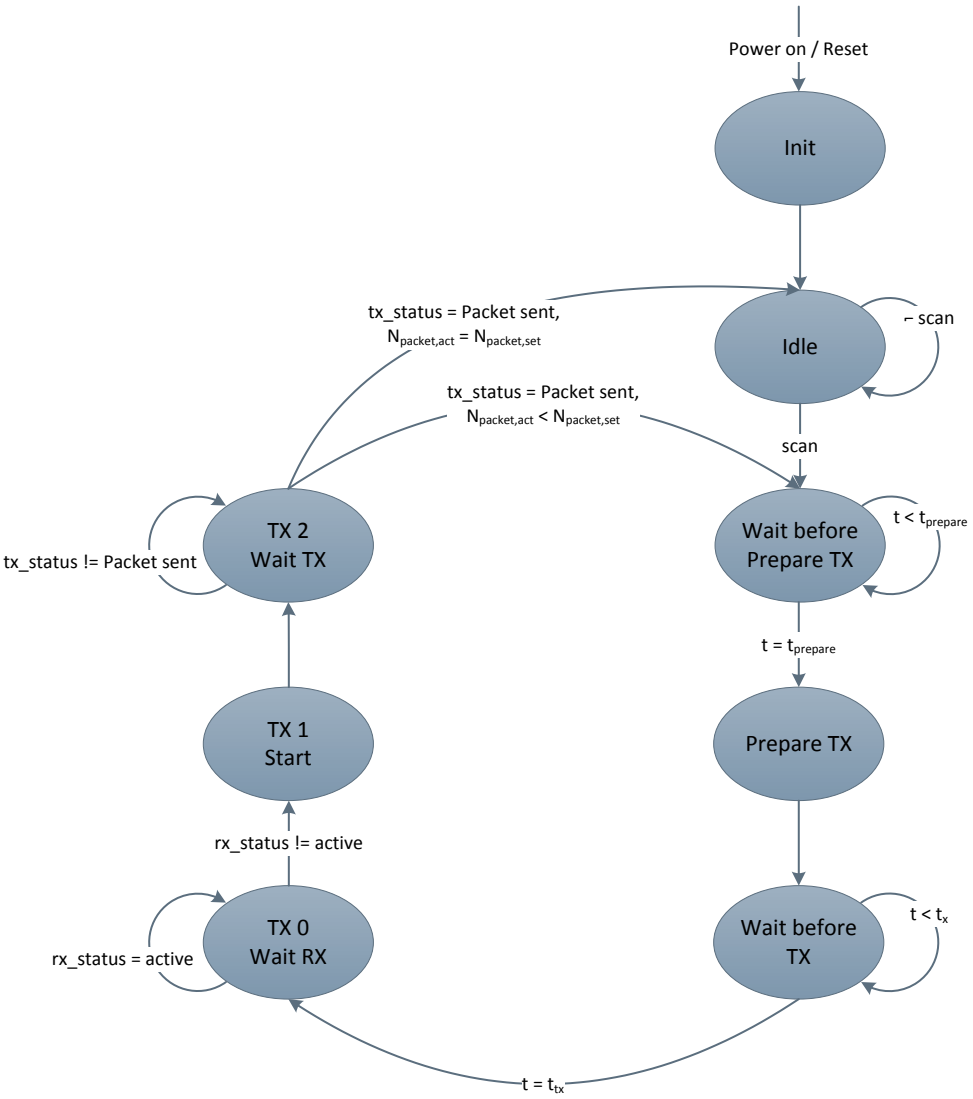


Bild 4.21: *Zustandsfolge des Zellensensors im Scan-Betrieb („Scan-Mode“)*

Reply-Betrieb

Zur Verifikation der *DL*-Kommunikation kann ein Zellensensoren durch das Kommando „reply“ dazu veranlasst werden, die letzten beiden empfangenen Kommandos bzw. Messages inklusive zugehöriger Parameter an die Basisstation zu übertragen.
Der Zellensensor versendet daraufhin, zu einem durch den Parameter des Kommandos vorgegebenem Zeitpunkt, einen Frame (vgl. 4.3.1.2) mit den betreffenden Daten. Die in dieser Betriebsart ablaufenden Zustandsübergänge sind in dem Bild 4.22 dargestellt.

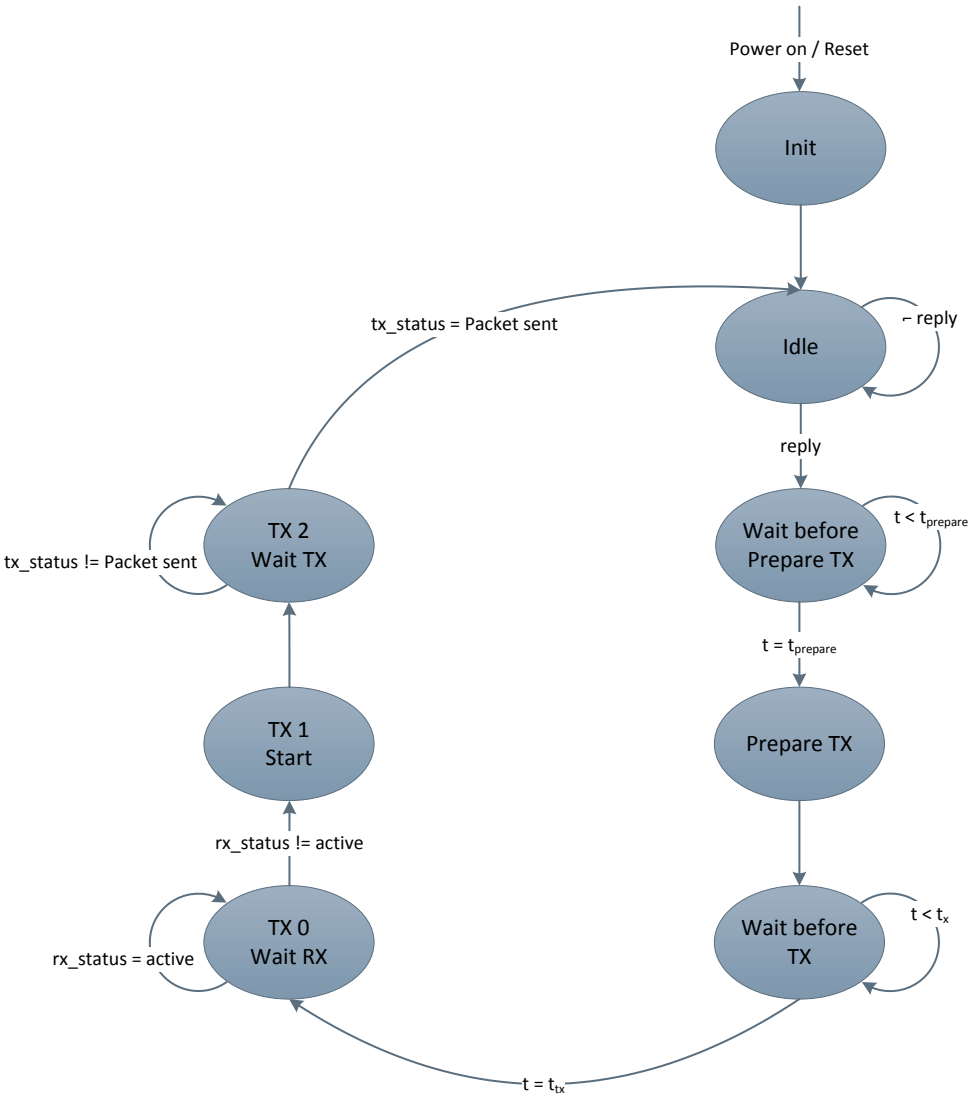


Bild 4.22: *Zustandsfolge des Zellensensors im Reply-Betrieb („Reply-Mode“)*

Hochstrom-Messbetrieb

Weiterhin ist angedacht, eine Funktionalität zur Messwerterfassung von Hochstromereignissen zu implementieren.
Dabei besteht die Notwendigkeit, von einem bestimmtem Zeitpunkt ausgehend, zunächst eine Reihe von Messwerten in einem definierten Abstand aufzuzeichnen und diese erst anschließend zu der Basisstation zu transmittieren. Dies begründet sich darin, dass während eines solchen Hochstromereignisses die notwendige Abtastrate wesentlich größer sein kann, als die Senderate für den *UL*-Kanal (vgl. Püttjer (2011)).

Das Bild 4.23 zeigt hierzu einen möglichen Ablauf des globalen Zustandsautomaten vom Zellensensor. Zunächst würde der Zellensensor in dem Abstand t_{sa} eine

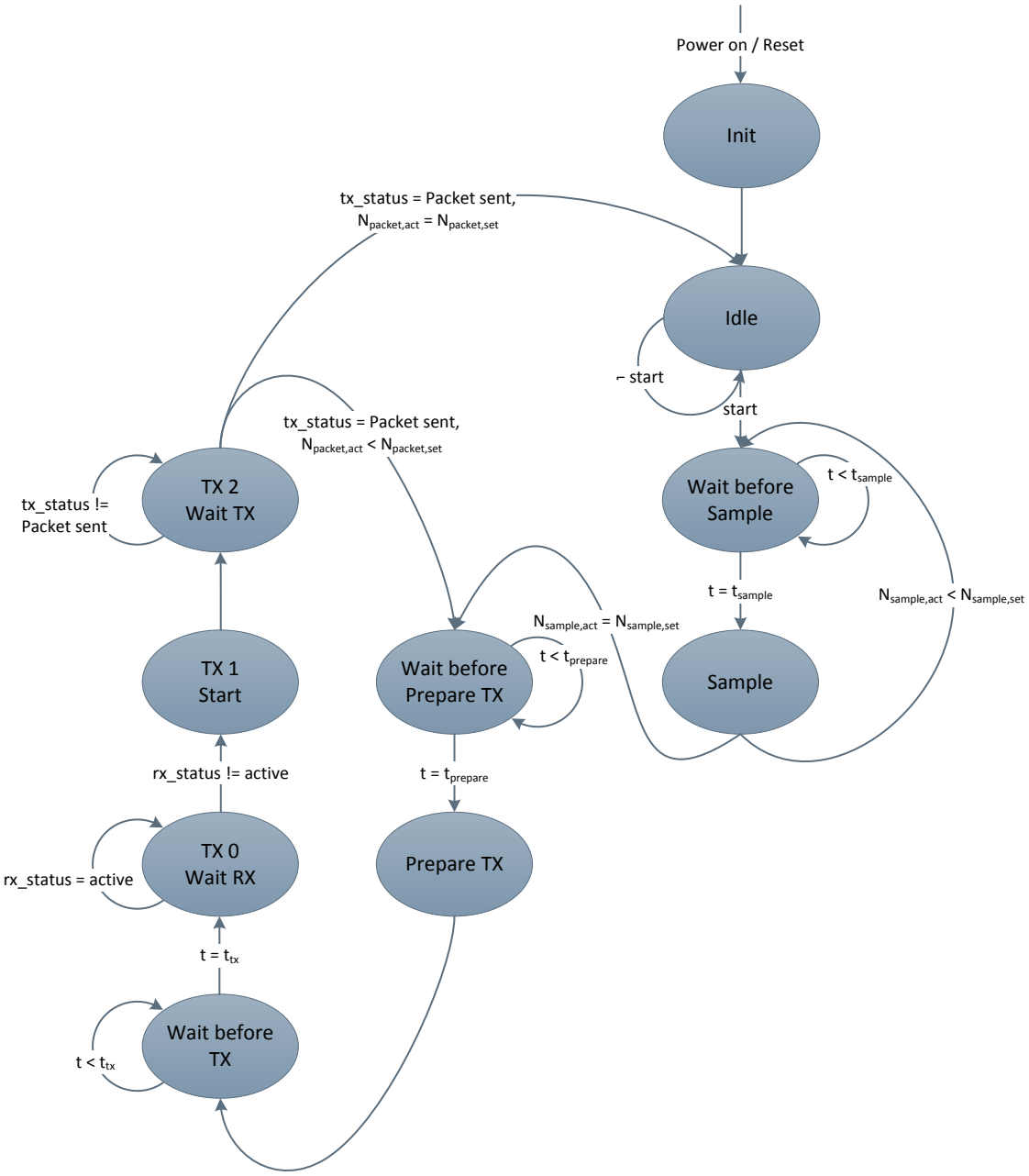


Bild 4.23: *Zustandsfolge des Zellensensors im Hochstrom-Messbetrieb („HC-Mode“)*

Anzahl von $N_{sa,set}$ Abtastwerten aufnehmen, ohne eine *UL*-Transmission vorzunehmen. Anschließend könnten diese Abtastwerte eventuell komprimiert in mehrere Frames eingefügt und versendet werden. Die Messwertaufzeichnung, als auch die

Transmission, sollte in ihren Parametern durch die Basisstation konfigurierbar sein. In der Tabelle A.1 sind hierzu bereits eine Reihe von möglichen Kommandos und Messages vorgesehen.

4.3.1.6 Grundlegender Betrieb des Mikrocontrollers

Sobald die *MCU* eine ausreichende Betriebsspannung von etwa 1,8 V bekommt, beginnt die Ausführung des Programmcodes. Da die Betriebsspannung nur endlich schnell ansteigt, bis sie den Endwert von 3,3 V erreicht hat, verändert sich in dieser Zeit ebenfalls der von der Betriebsspannung abhängige Systemtakt der *MCU*. Weiterhin kann es durch kurzzeitige Einbrüche der Spannung zu Störungen der Programmausführung kommen.
Darum ist es zweckmäßig, dass die *MCU* nach dem Anlaufen zunächst die *SVS*-Einheit konfiguriert (vgl. Anhang C.11) und dann solange wartet bis die Betriebsspannung ihren Endwert erreicht hat, bevor der Programmcode weiter ausgeführt wird. So lässt sich im Anlaufmoment zudem der Strombedarf reduzieren, da der Takt der *MCU* zunächst standardmäßig auf einen minimalen Wert eingestellt ist und der Transmitter für die Initialisierung nicht aktiv sein muss.
Ist der Endwert erreicht, wird der Systemtakt der *MCU* auf eine Frequenz von 1 MHz eingestellt. Dieser Wert erwies sich als ausreichend und führt gegenüber einer Frequenz von 8 MHz für einen um den Faktor 4,6 reduzierten Strombedarf der *MCU*.
Anschließend folgen die übrigen Einstellungen für Ports, Zeitgeber, I^2C-Schnittstelle, *ADC* und den *Watchdog-Timer (WDT)*[2]. Letzterer dient zur Laufzeitüberwachung des von der *MCU* ausgeführten Programmcodes. Wenn der *WDT* nicht alle 65,54 ms eine Meldung bekommt, erfolgt ein Reset der *MCU*. Diese Überwachung ist notwendig, da z. B. Fehler während der I^2C-Kommunikation zu sonst nur schwer detektierbaren Deadlocks[3] führen können.
Ist die grundlegende Initialisierung der *MCU* beendet, folgen die Initialisierungen des Temperatursensors und des *UHF*-Transmitters (siehe Anhang C.1), bevor das Hauptprogramm in eine leere Endlosschleife, zur Ausführung der interruptgesteuerten Sensorfunktionen, übergeht.

[2] engl. Überwachungszeitgeber
[3] engl. Verklemmungen

4.3.2 Software für den Mikrocontroller der Reader-Datenlogger-Einheit

Die Software für den Mikrocontroller der Reader-Datenlogger-Einheit (Basisstation) beruht auf diversen Vorarbeiten[4] aus dem Projekt. So verfügt diese bereits über diverse Funktionalitäten, welche jedoch bisher auf ein unidirektionales Kommunikationssystem zu den Zellensensoren abgestimmt sind.
Durch das neue bidirektionale Kommunikationssystem besteht nun die Möglichkeit einer synchronisierten Messwerterfassung und Kommunikation. In Folge dessen ergibt sich jedoch im Vergleich zu der vorherigen Zellen-Sensorik, eine sehr differenzierte Betriebsweise. Neben zusätzlichen Modulen für die *DL*-Kommunikation sind umfassende Anpassungen an der bestehenden Software erforderlich. Dies gilt insbesondre, da die bisherige Betriebsart keinen besonderen Zeitbezug seitens der Basisstation stellte. Sämtliche veränderten Quellcodedateien befinden sich im Anhang C.1.2.

4.3.2.1 Dekodierung des Uplink-Empfangssignals

Wie bereits in dem Abschnitt 4.2.1 beschrieben, verfügt die Basisstation über ein *UHF*-Empfänger-Modul für den *UL*-Kanal. Dieses Modul verfügt über keine Dekodierungseinheit, sodass es das demodulierte Empfangssignal direkt an einen Eingangs des Zeitgebers „Timer A“ der *MCU* von der Datenlogger-Einheit weitergibt. Somit muss die Dekodierung des Signals in der Software der *MCU* realisiert werden. Das für die *UL*-Übertragung verwendete Frame ist bereits in dem Abschnitt 4.3.1.2 beschrieben worden.

In der ursprünglichen dafür implementierten Routine[5] des Zeitgebers „Timer A“, erfolgte eine Dekodierung nur anhand der steigenden Flanken des Demodulationssignals (vgl. Abschnitt 4.3.1.1). Dieses Verfahren hat jedoch grundsätzliche Nachteile, abhängig von der Bitfolge, bedingt durch die Manchester-Codierung, sodass letzte Bit eines Frames zumeist erst mit dem Beginn des darauf folgenden Frames detektierbar ist. Zur Lösung dieses Problems ist in einer Vorarbeit (Püttjer (2011)) das gesendete Frame um ein zusätzliches Bit erweitert worden, sodass alle notwendigen Bits des Frames stets dekodierbar sind. Aufgrund des in dieser Arbeit erstmals verwendeten *UHF*-Transmitter des Zellensensors, welcher byteweise die zu sendenden Daten erwartet, ist diese Lösung nicht praktikabel.

Durch entsprechende Modifikationen an dem *UHF*-Empfänger-Modul (siehe Anhang B.2.4), welche bereits in einer Vorarbeit (Püttjer (2011)) beschrieben wurden,

[4] u. a. Plaschke (2008), Püttjer (2011)
[5] siehe Plaschke (2008) bzw. Püttjer (2011)

konnte nun eine Dekodierung mit doppelter Flankendetektion des Demodulationssignals verwirklicht werden. Ohne diese Modifikation ist dies zuvor nicht möglich gewesen, da durch ein stark veränderliches Tastverhältnis des Signals eine Auswertung der Puls-Pausenzeit keine zuverlässigen Ergebnisse brachte.
Der Zeitgeber „Timer A“ der *MCU* ist nun derart eingestellt („Capture-Mode“), dass dieser bei steigender und fallender Flanke einen *IRQ* generiert. Auf diesen *IRQ* folgt die zugehörige *ISR* (siehe Anhang C.29), welche anhand der im Bild 4.24 dargestellten Zeiten, die Dekodierung ausführt.

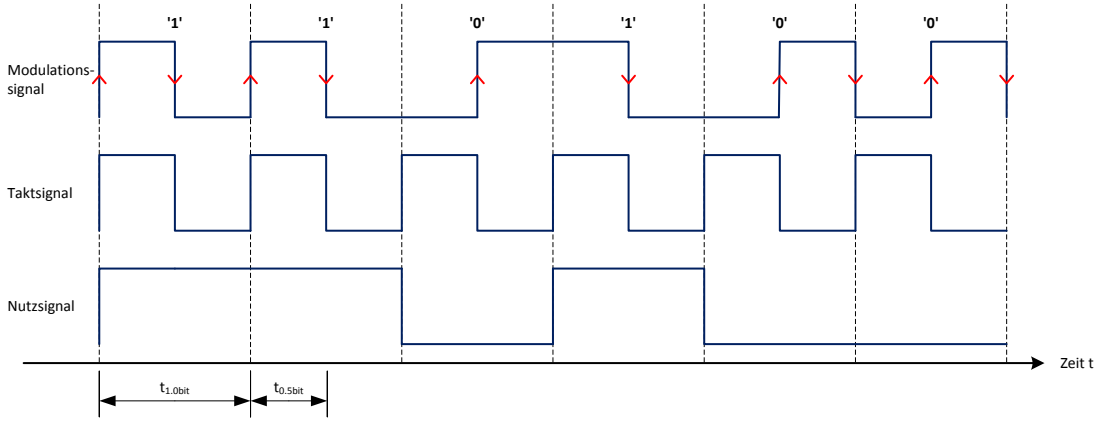


Bild 4.24: *Zustände der Manchester-Codierung, insbesondere bei doppelter Flankendetektion*

Im Gegensatz zu der ursprünglichen Implementierung misst der Zeitgeber nun die Zeit zwischen zwei auftretenden Flankenwechsel. Durch Festlegung einer Obergrenze, anhand der Zeitdauer für den *SOF*, überprüft dieser zugleich („Compare-Mode“), ob eine Zeitüberschreitung in Folge einer Störung aufgetreten ist.
Der in der *ISR* ablaufende Prozess ist durch einen Zustandsautomaten gesteuert. Dessen Zustände ähneln denen des Zustandsautomaten für den *DL*-Empfang vom Zellensensor (siehe Abschnitt 4.3.1.1). Da durch die doppelte Flankendetektion theoretisch nur zwei verschiedene Zeitendauern $t1.0bit$ bzw. $t0.5bit$ auftreten können, vereinfacht sich die Dekodierung des Demodulationssignals erheblich. Weiterhin erfolgt anhand des *SOF* eine Synchronisation, aus der vier Schwellwerte für beiden möglichen Zeitdauern prozentual berechnet werden.
Abhängig von der gemessenen Zeitdauer und des Eingangswertes erfolgt anschließend zunächst die Erfassung des noch nach Manchester codierten Signals. Erst wenn jeweils 16 Bit erfasst sind, erfolgt die byteweise Dekodierung. Bei der Implementierung ist insbesondere Wert auf eine minimale Ausführungszeit der *ISR* gelegt worden, da diese bei einer Übertragungsrate von 10 kBaud alle 100 μs ausgeführt werden kann.
Die weitere Verarbeitung, wenn alle 11 Bytes eines Frames komplett empfangen sind, erfolgt wie auch bereits in der vorherigen Version der Routine, abhängig von der momentanen Betriebsart der Basisstation. In der normalen Betriebsart für die

Aufzeichnung von Messwerten, folgt zunächst eine Überprüfung, ob die Quelladresse mit einem bekanntem Zellensensor übereinstimmt, gefolgt von einem Abgleich des Paritätswerts.
Sind die Daten korrekt und Messwerte von der Quelladresse erwünscht, werden diese mit der neu hinzugefügten Funktion „convert_sensor_data()“ (siehe Anhand C.42) aufbereitet. Dabei unterscheidet die Funktion anhand der im Frame enthaltenen *VID* zwischen dem Typ des Zellensensors und dessen Version, sowie durch die *SIF*, welche Datensätze in dem Daten-Block enthalten sind. Anschließend wird der empfangene Datensatz mit einem Zeitstempel versehen, welcher sich aus dem Zeitpunkt wann das Frame die Basisstation erreichte und der Differenz zwischen Transmission und Messwertaufnahme bildet.
Ist der Vorgang beendet, signalisiert die *ISR* durch ein entsprechendes Flag, sodass die empfangenden Daten in dem Hauptprogramm (siehe Anhang C.21) weiterverarbeitet werden können.

4.3.2.2 Kodierung des Downlink-Sendesignals

Bereits in dem Abschnitt 4.3.1.1 ist das für die Informationsübertragung auf dem *DL*-Kanal verwendete Frame, inklusive der Kodierung, beschrieben worden. Dieses wird mit der Datenlogger-Einheit unter Verwendung der Reader-Einheit erzeugt.

Das verwendete Reader-IC der Reader-Einheit ermöglicht zwar die Verwendung von automatischem Framing (siehe Texas Instruments (2006b)), jedoch dies zu umgehen und vielmehr das Sendesignal für den *DL*-Kanal direkt zu modulieren. Für eine derartige Verwendung muss das Reader-IC, mittels der *SPI*-Schnittstelle, zunächst entsprechend konfiguriert werden.
Da die zu verwendende *SPI*-Schnittstelle für das Reader-Modul die gleiche Hardwareeinheit nutzen muss wie die serielle Schnittstelle (UART), bedarf es einer entsprechenden Konfiguration, je nach Verwendung. Die Funktion „USART0_init_SPI()“ (vgl. Anhang C.38) konfiguriert die Schnittstelle für die Anwendung zum *SPI*-Mode, wohingegen die Funktion „USART0_init()“ (siehe Anhang C.36) diese zum UART-Mode konfiguriert. Mittels der Funktion „reader_set_reg()“ (siehe Anhang C.38) erfolgt schließlich die Umstellung des Reader-IC in den „direct-mode“, sodass eine direkte Modulation des Trägersignals ermöglicht wird.

Für die Transmission eines Frames auf dem *DL*-Kanal sind zwei Funktionen „tx13p56_send_wakeup()“ bzw. „tx13p56_send()“ (siehe Anhang C.38) implementiert worden. Erstere veranlasst die Transmission eines zusätzlichen *Wake-Up*-Blocks (siehe Bild 4.14) zum Aufwecken der Zellensensoren vor dem eigentlichem Frame, wohingegen die zweite Funktion den genannten Block nicht versendet.

Die Übergabeparameter für beide Funktionen sind gleich, sie erhalten jeweils die Zieladresse, das Kommando bzw. die Message mit dem zugehörigen Parameter. Aus diesen Daten wird dann zunächst mit der Funktion „tx13p56_gen_frame()“ (vgl. Anhang C.38) der zu transmittierende Frame inklusive des Paritätswerts gebildet. Die Manchester-Codierung erfolgt hierbei erst direkt bei der Transmission des Frames.
Anschließend erfolgt die Konfiguration des Zeitgebers „Timer B“ für den sogenannten „Compare-Mode“, in dessen *ISR* die eigentliche Modulation des *DL*-Signals abläuft. Vor Beginn der Transmission muss allerdings geprüft werden, ob derzeit eine *UL*-Übertragung aktiv ist. Wenn eine aktiv sein sollte, muss gewartet werden bis diese vollendet ist, da die Zellensensoren, ebenso wie die Basisstation, nicht zugleich empfangen und senden können. Entsprechend ist die *UL*-Dekodierung während einer *DL*-Transmission zu deaktivieren, da das *DL*-Signal in das *UL*-Signal einkoppelt.
Nachdem der Zeitgeber „Timer B“ gestartet ist, führt ein Zustandsautomat in der *ISR* des Zeitgebers (siehe Anhang C.31) die eigentliche Transmission mit einer Übertragungsrate von 10 kBaud aus. Dieser verfügt über insgesamt 10 Zustände, welche angefangen bei dem *Wake-Up*-Block, gefolgt von dem *SOF*-Block, den eigentlichen Nutzdaten, sowie der Zuführung eines definierten Endes (vgl. 4.3.1.1), alle Gegebenheiten des Frames berücksichtigen. In dem letztem Zustand des Automaten erfolgt, verzögert um 100 μs nach Deaktivierung des Trägersignals für den *DL*, die Reaktivierung der *UL*-Dekodierung.

4.3.2.3 Koordination der Zellen-Sensorik

Da die konzipierten Zellensensoren nun aufgrund der bidirektionalen Kommunikation eine koordinierte Betriebsweise gestatten, ist für die zeitliche Abfolge der Zellen-Sensorik im Standard-Messbetrieb ein Verfahren, wie in dem Bild 4.25 gezeigt, vorgesehen.

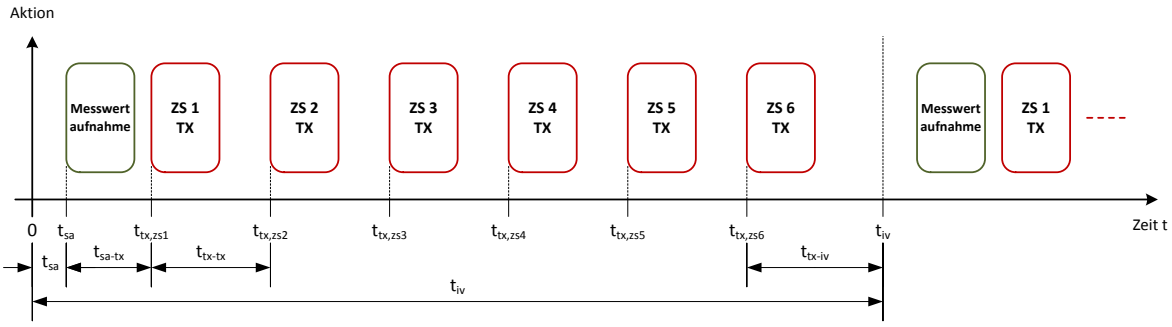


Bild 4.25: *Zeitliche Abfolge der Zellen-Sensorik für den Standard-Messbetrieb, bei einer Anzahl von $n = 6$ Zellensensoren*

Alle Zellensensoren sollen hierbei zu dem gleichen Zeitpunkt t_{sa} ihre Messwertaufnahme starten. Für die anschließende Übertragung der Messwerte über den *UL*-Kanal ist ein synchronisiertes Zeitschlitzverfahren vorgesehen. Jedem Zellensensor wird hierbei von der Basisstation ein Zeitpunkt t_{tx} zugeteilt, bei welchem die Transmission zu starten ist. Unter Annahme idealer Bedingungen kann somit eine Kollision von Frames vermieden werden. Mit einer Intervalldauer t_{iv} wiederholt sich die Messwertaufnahme und die Transmission.

Für die Konfiguration der vorgesehenen zeitlichen Abfolge, gilt es die in dem Bild 4.26 gezeigten Parameter festzulegen, bzw. deren Grenzen zu beachten.

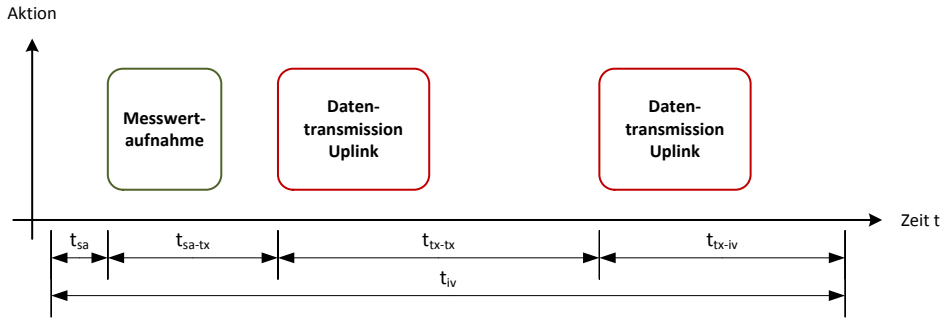


Bild 4.26: *Grundlegende Parameter für die zeitliche Koordination der Zellensensoren im Standard-Messbetrieb*

- t_{sa}:
  - Zeitpunkt der Messwertaufnahme
  - Wert abhängig von dem resultierendem Fehler des Zeitgebers
- t_{sa-tx}:
  - Zeitdauer zwischen Beginn der Messwertaufnahme und erstem möglichen Zeitschlitz für die Transmission
  - Abhängig von der Dauer der Messwertaufnahme und der Vorbereitung für die Transmission
- t_{tx-tx}:
  - Minimale zeitliche Differenz zwischen dem Startzeitpunkt zweier Zeitschlitze
  - Festgelegt anhand der Framedauer $t_{fr,ul}$ von 21,6 ms, der benötigten Verarbeitungszeit t_{proc} der Daten auf der Basisstation und dem Abstand $t_{tx,ab}$ zwischen zwei Zeitschlitzen

- t_{tx-iv}:
  - Zeitdauer zwischen Start des letztem Zeitschlitz und Ende des Intervalls
  - Wert ebenfalls abhängig vom resultierendem Fehler des Zeitgebers, mindestens jedoch so groß wie die Framedauern vom *UL* und *DL* zusammen, entsprechend 33,6 ms
- t_{iv}:
  - Intervalldauer
  - Berechnet sich nach
    $t_{iv} = t_{sa} + t_{sa-tx} + (n-1) \cdot t_{tx-tx} + t_{tx-iv}$ für n Zellensensoren

Beispielhafte Werte für diese Parameter sind in der im Anhang C.45 gezeigten Header-Datei für verschiedene Konfigurationen, zu finden. Eine optimale Auslegung der Parameter bedarf einer empirischen Untersuchung der fehlerbehafteten Zeitgeber von den Zellensensoren, bedingt durch die Verwendung des internen RC-Oszillators. Die momentan ermittelten Werte entstammen praktischen Versuchen mit nur drei verschiedenen Zellensensoren. Eine Aussage über die auftretenden Fehler ist somit nur begrenzt gültig.

Zur Minimierung der Auswirkung des unvermeidbaren Ungleichlaufes der Systemzeit der einzelnen Zellensensoren, wie im Bild 4.27 dargestellt, ist vorgesehen, diese in einem Abstand entsprechend der Intervalldauer t_{iv} zu synchronisieren.

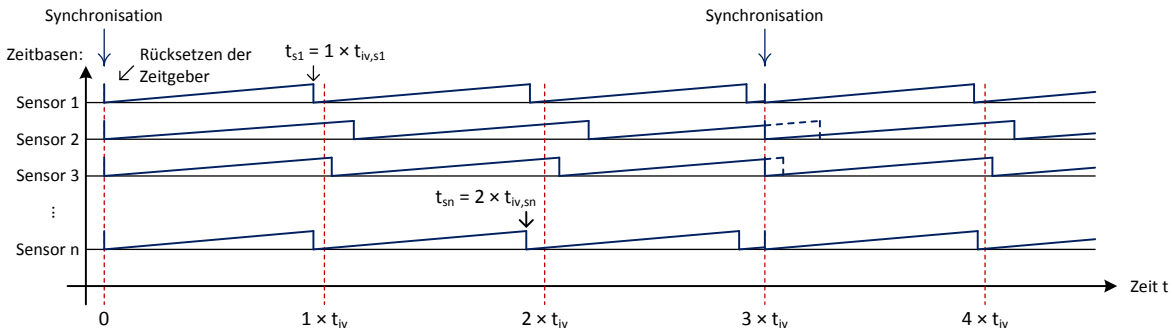


Bild 4.27: *Veranschaulichung der Synchronisation der Zellensensoren auf eine gemeinsame Zeitbasis*

Anderenfalls könnte es, neben eines unzureichend großen Jitters der Messwertaufnahme, zu einer Überlappung der Frames auf dem *UL*-Kanal kommen. Dies begründet auch die Zufügung einer zeitlichen Reserve in den Parametern t_{sa} und t_{tx-iv}, da eine erfolgreiche *DL*-Übertragung zwischen der letzten Transmission und der Messwertaufnahme (siehe Bild 4.28) sichergestellt werden muss.

Wie bereits beschrieben worden ist, benötigt jeder Zellensensor für den Standard-Messbetrieb drei zeitliche Parameter, um die gewünschte Funktionalität ausführen zu können. Diese Parameter können den Zellensensoren, mittels den im Anhang

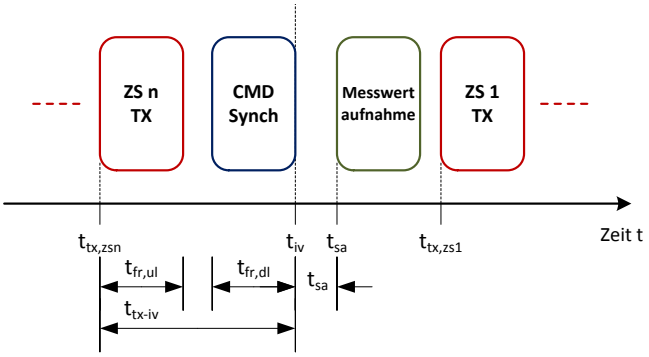


Bild 4.28: *Zeitliche Abfolge der Synchronisation im laufenden Standard-Messbetrieb*

A.1 gezeigten Kommandos, zugeführt werden.
Jedes dargestellte Kommandos bzw. jede Message ist in der Software der Basisstation durch eine separate Funktion (siehe Anhang C.44) realisiert.
Die Funktion „zs_send_time_interval()“ überträgt beispielsweise den Parameter für die Intervalldauer t_{iv} als Broadcast an alle Zellensensoren. Im Gegensatz dazu versendet die Funktion „zs_send_time_tx()“ an jeden einzelnen Zellensensor den zugehörigen Zeitpunkt t_{tx} für den Beginn des Zeitschlitzes. Die Funktion „zs_gen_time_tx()“ berechnet den Zeitpunkt t_{tx} der Transmission, in Abhängigkeit der Anzahl an Zellensensoren und den vorgegebenen zeitlichen Parametern.
Zum Starten des Messbetriebs ist die Funktion „recording_start()“ implementiert worden. Diese sendet zunächst, synchronisiert mit dem Systemzeitgeber, das Kommando „synch“ an die Zellensensoren und aktiviert anschließend die *UL*-Dekodierung. Die während des laufenden Messbetriebs notwendigen Synchronisationen erfolgen wiederum, getriggert durch den Systemzeitgeber (vgl. C.33), in einem Software-Interrupt (siehe Anhang C.28).
Die komplette Abfolge an Kommandos bzw. Messages, welche nötig ist, um die Zellensensoren für den Standard-Messbetrieb zu konfigurieren und anschließend den eigentlichen Messbetrieb durchzuführen, ist in dem Bild A.1 im Anhang A dargestellt.

4.3.2.4 Koordination der Strommessung

In der bisherigen Software der Datenlogger-Einheit erfolgte die zentrale Erfassung der Stromwerte unkoordiniert. Dies stellte kein Problem dar, denn die bisherigen Zellensensoren führten die Messwertaufnahmen ebenso unkoordiniert durch. Da nun aber die Zellensensoren synchronisiert betrieben werden, ist gleiches auch für die zentrale Strommessung vorgesehen.
Wie auch die Synchronisation der Zellensensoren, ist die Strommessung durch die

ISR des Systemzeitgebers gesteuert. Diese startet die Abtastung der beiden analogen Stromsignale des externen Stromsensors (siehe Püttjer (2011)) durch den *ADC*. Erst in der zugehörigen *ISR* des *ADC* (vgl. C.27) erfolgt die Auswertung der Messwerte.
Beachtet wird bei der Strommessung zudem, dass diese nicht während einer Synchronisationssequenz erfolgt, sondern erst danach, da diese sich trotz zusätzlicher Anpassung der Hardware (siehe Anhang B.2.5) störend auswirkt.

Die Festlegung der Parameter von der Strommessung erfolgt in einer Header-Datei (siehe Anhang C.45) zusammen mit den Parametern für die Zellensensoren. Dabei ist darauf zu achten, dass die Strommessung mit dem gleichen bzw. teil- oder vielfachen Intervall wie die Synchronisation durchgeführt wird.
Sowohl bei der zeitlichen Koordination der Strommessung, als auch der Zellen-Sensorik, sind die Einstellungsmöglichkeiten durch den Systemzeitgeber der Datenlogger-Einheit beschränkt. Da der hierfür verwendete *WDT* nur unzureichend einstellbar ist, inkrementiert die Systemzeit in relativ großen Schritten von 62,5 ms. Gegenüber der ursprünglichen Inkrementierung von einer Sekunde, ist dies zwar eine deutliche Verbesserung, aber insbesondere für die Zeitstempel der Messwerte unzureichend. Ein anderer Zeitgeber für die Systemzeit steht nicht zur Verfügung, die beiden übrigen Zeitgeber der *MCU* werden für den *DL*-, sowie *UL*-Kanal, bereits verwendet.

4.3.2.5 Schnittstellen

Die Datenlogger-Einheit verfügt über verschiedene Schnittstellen.
Zum einen ist weiterhin die Ausführung verschiedener Funktionen, auswählbar durch die drei Taster des Entwicklungsboards in Kombination mit dem LC-Display, wie auch bereits bei der alten Software, möglich. Die Funktion „BAT-MON_menu_options()“ (vgl. C.36), welche hierfür das Auswahlmenü erzeugt, ist entsprechend um die neu hinzugefügten Funktionen ergänzt worden.
Andererseits können sämtliche Funktionalitäten der Datenlogger-Einheit durch Klartexteingabe mit einer Konsole, unter Verwendung der seriellen Schnittstelle, ausgeführt werden. Da diese Schnittstelle ebenfalls bereits bestand, war es nur nötig, die zugehörige Datei (siehe Anhand C.47) um die neu hinzugefügten Funktionalitäten zu ergänzen. Eine Übersicht über die verfügbaren Kommandos erhält man durch Eingabe von „help“ auf der Kommandozeile, bzw. in der beschriebenen Datei. Die benötigen Einstellungen für die serielle Schnittstelle sind im Anhang C.48 zu finden.

Neu hinzugefügt ist hingegen eine direkte Ausgabe der Messdaten über die serielle Schnittstelle. Hierbei wird unterschieden zwischen einer dezimalen Darstellung der Messwerte und einer, vorrangig geplant für die Weiterverarbeitung in Matlab,

hexadezimalen Darstellung. Sobald neue Messdaten vorliegen, sei es von einem Zellensensor oder von der zentralen Strommessung, werden diese unmittelbar zeilenweise ausgegeben.
Die dezimale Darstellung dient dazu, auf einer Konsole die Messwerte unmittelbar mitlesen zu können. So zeigt die folgende Zeile eine beispielhafte Ausgabe von einer Strommessung:

$$CUR : |TS : 06 : 48.687|C : 1|V : +0015$$

Der hierbei ausgegebene Zeitstempel TS steht für 6 Minuten, 48 Sekunden und 687 Millisekunden. Anschließend folgt die Ausgabe des gewählten Kanals C, sowie der Messwert V des Stromes von $+1{,}5$ A. Werden hingegen Messdaten von einem Zellensensor ausgegeben, so gestaltet sich die Ausgabe wie folgt:

$$SEN : 00|ADD : 28|VID : 03|CV : 3.0664|SV : 3233|T : 2318$$

Zunächst zeigt die Ausgabe *SEN* die von der Basisstation zugeteilte Nummer des Zellensensors an, diese richtet sich nach der Reihenfolge wie die Zellensensoren in dem „Scan-Mode“ detektiert worden sind. Darauf folgt die hexadezimale Ausgabe der Adresse des Zellensensors und der *VID*. Die Ausgabe der eigentlichen Messwerte ist jeweils gekennzeichnet mit *CV* für die Zellenspannung (3,0664 V), *SV* für die Betriebsspannung (3,233 V) und T für die Temperatur (23,18 °C). Bis auf den Wert für die Zellenspannung *CV* werden, aus praktischen Gründen der Implementierung, die übrigen Messwerte ohne einen Dezimalpunkt ausgegeben.
In der hexadezimalen Darstellung hingegen, erfolgt die Ausgabe mit einer ähnlichen Formatierung, jedoch werden sämtliche Zahlenwerte im hexadezimalen Zahlenformat ausgegeben. Hierdurch lässt sich die Anzahl der zu übertragenen Zeichen reduzieren, sodass weniger Zeit für die Ausgabe benötigt wird. Da somit auch mehr Daten ausgegeben werden können, enthält jede Ausgabe von Messdaten eines Zellensensors ebenso einen Zeitstempel, bestehend aus Tagen, Stunden, Minuten, Sekunden und Millisekunden. Die Ausgabe der Strommessung, welche einen ebenso umfangreichen Zeitstempel nun enthält, besteht nur aus dem vom Offset kompensiertem Ausgabewert des *ADC*, dem Kanal, sowie der Stromrichtung. Eine anschließende Berechnung des Stromwertes, beispielsweise in Matlab, bietet eine höhere Genauigkeit, als es auf der *MCU* effizient zu realisieren wäre.
Die Auswahl der gewünschten Darstellungsform geschieht durch die Eingabe des Kommandos „d_ua_dh“ auf der Kommandozeile. Bei einer hohen Kanalbelegung empfiehlt es sich, zur Reduzierung der Verarbeitungszeit der Messdaten, die Ausgabe der Messwerte auf dem LC-Display mit dem Kommando „d_lcd_sd“ zu deaktivieren.

4.3.3 Messdatenverarbeitung mit Matlab

Zur Analyse von den entstehenden Messwerten der Zellen-Sensorik, erfasst von der zuvor betrachteten Datenlogger-Einheit, ist ein Matlab-Skript erstellt worden. Unter Ausnutzung der beschriebenen seriellen Schnittstelle der Datenlogger-Einheit kann dieses einerseits die Messwerte erfassen, andererseits die Einheit sowie auch die Zellensensoren steuern. Zudem bietet das Skript eine unmittelbare Darstellung der erfassten Messwerte, sodass eine direkte Beobachtung von Messwertverläufen, aufgetragen über die Systemzeit der Basisstation, ermöglicht wird.

Das erstellte Skript (siehe Anhang C.49) verlangt bei der Ausführung zunächst einige Eingaben des Benutzer, wie z. B. die Anzahl der zu erfassenden Zellensensoren. Anschließend öffnet es die serielle Schnittstelle für die Kommunikation mit der Datenlogger-Einheit und bereitet diverse Strukturen für die Erfassung der Messwerte vor sowie Graphen für deren Darstellung. Ist dieser Vorgang abgeschlossen, beginnt das Skript mit der Einstellung der Datenlogger-Einheit und den Zellensensoren. Zunächst wird eine eventuell laufende Messreihe beendet, bevor die Suche nach Zellensensoren („Scan-Mode“), anhand der eingestellten Anzahl, erfolgt. Sollten weniger Sensoren als die angegebene Anzahl vorhanden sein, ist es notwendig die Suche durch Tasteneingabe an der Basisstation zu beenden. Im Anschluss, bevor die Aufzeichnung der Messwerte startet, erhalten die Zellensensoren die Parameter für die Messwerterfassung. Die Ausführung sämtlicher Kommandos zur Steuerung der Datenlogger-Einheit ist durch separate Funktionen realisiert, um die Wiederverwendbarkeit zu erleichtern und die Übersichtlichkeit zu erhöhen.
Die anschließende Erfassung der Messdaten erfolgt gesteuert durch eine Schleife. In dieser wird zeilenweise die serielle Schnittstelle ausgelesen, die erfassten Daten in einer Funktion dekodiert und umgerechnet (vgl. Anhang C.65), in eine Struktur gespeichert (vgl. Anhang C.70) und dargestellt. Die für die Anzeige der Messwerte erstellte Funktion (siehe Anhang C.69) veranlasst die Graphen ihre zuvor festgelegten Datenquellen zu aktualisieren und die Darstellung neu zu zeichnen.
Eine Besonderheit ist hierbei, dass die Darstellung der Messwerte über der Zeit und nicht einfach über der fortlaufenden Nummer der Messwerte erfolgt. Um dies zu erreichen, werden die Messwerte, neben einem direkt lesbarem Zeitstempel, mit sogenannten „serial date numbers“ versehen.
Ist die Anzahl der eingestellten Messwerte erreicht oder die Aufzeichnung von dem Benutzer beendet worden, speichert das Skript automatisch die Messdaten und Graphen mit einem Zeitstempel versehen ab und beendet die Erfassung durch die Datenlogger-Einheit.
Die von dem beschriebenen Matlab-Skript erzeugten Graphen werden in dem nächstem Abschnitt 4.4, im Zusammenhang der Erprobung des Gesamtverfahrens, gezeigt. Sämtliche benötigten Matlab-Dateien befinden sich im Anhang C.2.1.

4.4 Erprobung des Gesamtverfahrens

Nachdem in den vorigen Abschnitten auf die Realisierung der Zellen-Sensorik, inklusive der dazugehörigen Software eingegangen wurde, soll nun eine Erprobung des Gesamtverfahrens zum Nachweis der Funktionsfähigkeit des entwickeltem Systems folgen. Diese wird einerseits anhand von synthetischen Signalen für die Zellenspannung, andererseits mit realen Signalen, gezeigt.

4.4.1 Allgemeine Funktion des Gesamtverfahrens

Vor der Erprobung des Gesamtverfahrens wird zunächst die allgemeine Funktionsfähigkeit des Systems, anhand ausgewählter Messungen veranschaulicht.
Das Bild 4.29 zeigt hierzu beispielhaft einige Signalverläufe des *DL*-Kanals, während der Übertragung des Kommandos „synch“. Wie man erkennen kann, demo-

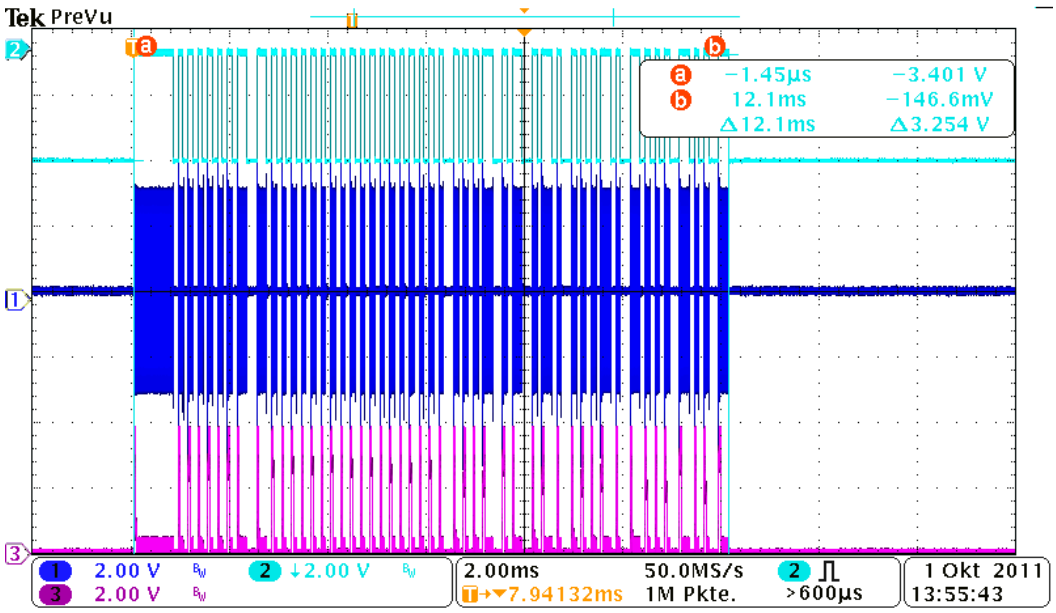


Bild 4.29: *Gemessene Signalverläufe des Downlink-Kanals*
Kanal 1: HF-Sendesignal an der DL-Antenne (TP 1)
Kanal 2: Modulationssignal der Datenlogger-Einheit
Kanal 3: Demodulationssignal des Zellensensors (TP 7)

duliert die entworfene Demodulationsschaltung in Kombination mit dem Hüllkurvendetektor (vgl. Abschnitt 3.3.1.2) wie vorgesehen das von der *DL*-Antenne ausgesendete *HF*-Signal. Die positiven Flanken des Nutzsignals werden allesamt

korrekt rekonstruiert. In Verbindung mit der Manchester-Decodierung für einfache Flankendetektion können somit die übertragenen Kommandos bzw. Messages decodiert werden. In dem Bild 4.29 ist ebenso die Dauer des übertragenen Frames mit 12,1 ms angegeben, dies entspricht der gewünschten Länge von 12 ms, in Verbindung mit dem zusätzlichem 100 μs langem Puls, für ein definiertes Ende (vgl. Abschnitt 4.3.1.1). Experimentell konnte zudem gezeigt werden, dass der Wirkungsbereich des *Wake-Up* nur geringfügig kleiner ist wie der Wirkungsbereich der Informationsübertragung über den *DL*-Kanal. Als maximale Reichweite war eine Distanz von 41 cm, unter Verwendung des Reader-Moduls „BATSEN BS Reader 13.56Mhz v0.3“, mit parallel zur Spulenachse ausgerichteten Antennenspulen, ermittelbar.

In dem Bild 4.30 ist hingegen das von einem Zellensensor über den *UL*-Kanal transmittierte Frame dargestellt, empfangen durch das *UHF*-Empfänger-Modul der Basisstation. Die Dauer des Frames entspricht mit 21,6 ms exakt dem vorgesehenen

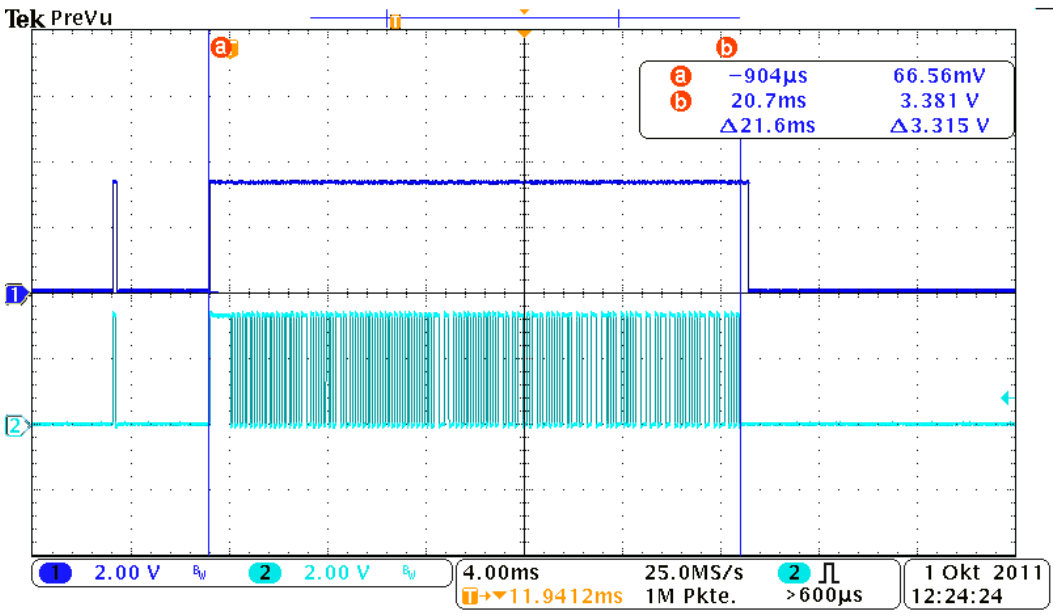


Bild 4.30: *Gemessene Signalverläufe des Uplink-Kanals*
Kanal 1: Dekodierung und Aufbereitung der Informationen durch die Datenlogger-Einheit
Kanal 2: Demodulationssignal des UHF-Empfänger-Modul

Wert (vgl. Abschnitt 4.3.2.1). Etwa 4 ms vor dem eigentlichen Frame ist zusätzlich ein etwa 100 μs dauernder Puls empfangen worden. Dieser Puls resultiert aus dem Abgleichvorgang des Antennenkreises vom *UHF*-Transmitter und ist für die Dekodierung des Frames aufgrund der verwendeten doppelten Flankendetektion nicht relevant. Wie in dem Bild 4.30 gezeigt, bricht unmittelbar nach der fallenden Flanke

des Pulses die Dekodierung ab. In dem zugehörigem Datenblatt des Transmitters (siehe Silicon Laboratories (2010b)) ist das beobachtete Verhalten nicht näher spezifiziert, lediglich ein Hinweis ist vorhanden, dass während eines Abgleichvorgangs der Verstärker eingeschaltet wird. Da nach Ende des Frames die empfangenen Informationen geprüft und aufbereitet werden, endet der Vorgang etwa 200 μs verzögert zu dem Frame.
Nicht ersichtlich ist in diesem Bild die benötigte Zeit für die Übertragung der empfangenen Informationen, durch die serielle Schnittstelle, an eine übergeordnete Anwendung. Dieser Vorgang, zu sehen in dem Bild A.5 im Anhang A, benötigt zusätzlich etwa 8 ms.

Neben der Übertragung von Informationen dient der *DL*-Kanal der Übertragung von Energie, um einen *Wake-Up* der Zellensensoren durchzuführen. Hierzu wird vor dem eigentlichen Frame ein *Wake-Up*-Block der Dauer 5,2 ms (vgl. Abschnitt 4.3.1.1), wie in dem Bild 4.31 mit einer Messung dargestellt, transmittiert. Obwohl

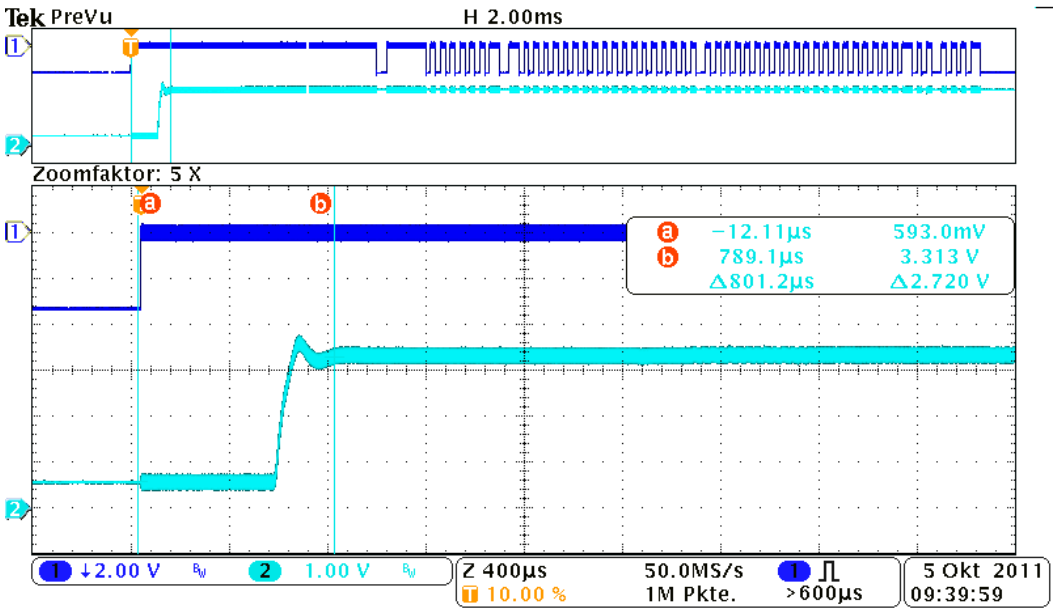


Bild 4.31: *Gemessene Signalverläufe während eines Wake-Up durch den Downlink-Kanal, bei einer Distanz von* 41 cm *zwischen den Antennenspulen*
Kanal 1: Modulationssignal der Datenlogger-Einheit für den DL-Kanal
Kanal 2: Betriebsspannung U_{supply} eines Zellensensors (TP 1)

bei dieser Messung die Distanz zwischen dem Zellensensor und der *DL*-Antenne der Basisstation 41 cm betrug, bedarf es nur einer Zeitdauer von etwa 800 μs, bis die Betriebsspannung des Zellensensors stabil ist. Die verbleibende Zeitdauer bis zum

Anfang des Frames, bedarf es für die Initialisierung der *MCU* sowie der externen Komponenten.

Die zeitlichen Abfolgen der Zellen-Sensorik im Standard-Messbetrieb, unter der Verwendung von drei Zellensensoren, sind in dem Bild 4.32 dargestellt. Ersichtlich

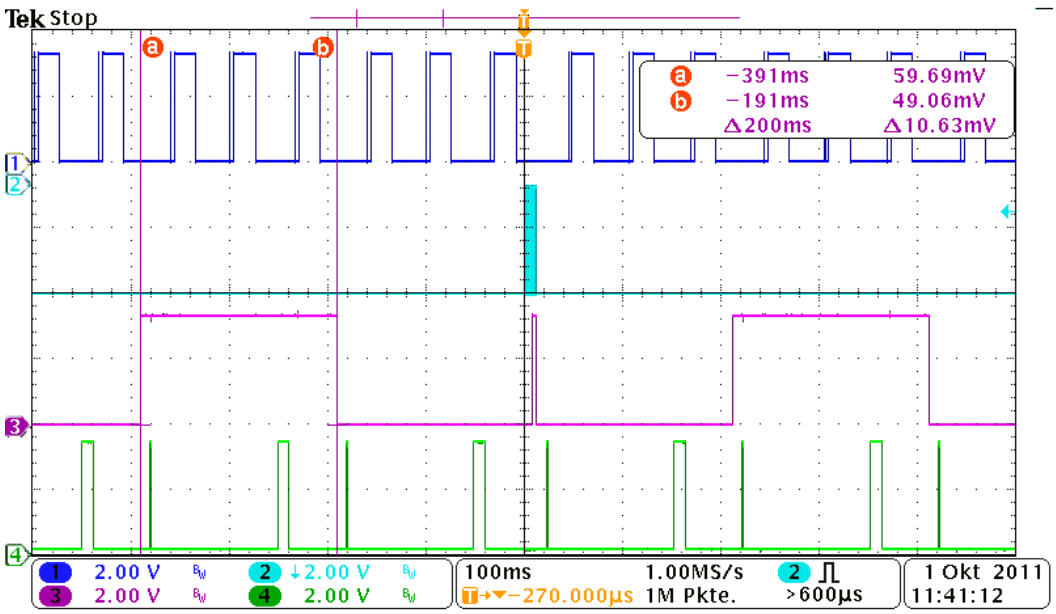


Bild 4.32: *Gemessene Signalverläufe der Zellen-Sensorik bei einer Intervalldauer t_{iv} von* 200 ms *sowie drei Zellensensoren*
Kanal 1: Dekodierung des UL-Signals durch die Datenlogger-Einheit
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Intervalldauer des Systemzeitgebers vom Zellensensor (TP 10)
Kanal 4: Verweildauer des dritten Zellensensors im Zustand „Sample“ und „Prepare TX“ (TP 11)

ist neben der Informationsübertragung über den *DL*- und *UL*-Kanal, die Intervalldauer t_{iv} der Systemzeit eines Zellensensors und die Zeitpunkte für die Ausführung der Messwertaufnahme t_{sa} (kurzer Puls), sowie der Initialisierung Datentransmission t_{pr} (langer Puls).
Zwecks Synchronisation der Zellensensoren wird in diesem Falle alle 2 Sekunden das Kommando bzw. Message „synch“ über den *DL*-Kanal zu den Sensoren übertragen. Aus der zeitlichen Differenz zwischen dem Ende der Intervalldauer t_{iv} und dem Zeitpunkt der Synchronisation, dargestellt anhand einer Messung im Bild A.2 im Anhang A, lässt sich exemplarisch der Fehler vom Zeitgeber eines Zellensensors (in diesem Fall 3,34 ms bzw. 0,167 %) bestimmen.
Die Abstände t_{tx-tx} zwischen den Frames, mit denen die drei Zellensensoren innerhalb des Intervalls t_{iv} die Messdaten an die Basisstation übermitteln, beträgt nach

der Synchronisation wie vorgesehen 62,5 ms (vgl. Bild A.5 im Anhang A). Dieser Wert resultiert aus dem Zeitfenster $t_{tx,pos}$ innerhalb des Intervalls t_{iv}, in dem eine Übertragung $t_{tx,zs1}$ frühstens, bzw. eine Übertragung $t_{tx,zsn}$ spätestens, starten kann. Folgende Parameter (vgl. Abschnitt 4.3.2.3) sind bei den gezeigten Messungen vorgegebenen:

$$\begin{aligned} t_{sa} &= 10\,\text{ms} \\ t_{sa-tx} &= 25\,\text{ms} \\ t_{tx-iv} &= 40\,\text{ms} \\ t_{iv} &= 200\,\text{ms} \end{aligned}$$

Somit ergibt sich das zulässige Zeitfenster $t_{tx,pos}$ zu 125 ms, was bei den verwendeten drei Zellensensoren zu dem Abstand der Transmissionen t_{tx-tx} von 62,5 ms führt. Bei einer alle 2 Sekunden ausgeführten Synchronisation, konnte eine Veränderung des Abstands t_{tx-tx} zwischen den Transmissionen zweier Zellensensoren von 4,2 ms, entsprechend 6,72 % bezogen auf t_{tx-tx}, beobachtet werden. Dieser Fehler beruht auf den Fehlern beider beteiligten Zellensensoren und ist bezogen auf das Synchronisationsintervall von 2 Sekunden mit 0,21 % eher gering.
Der Zeitpunkt t_{sa} für die Ausführung der Messwertaufnahme konnte anhand der Messung im Bild 4.32 ebenso bestätigt werden, wie der Zeitpunkt t_{tx_zs1} für die erste Transmission von 35 ms (vgl. Bild A.4 im Anhang A) und ebenso der letzten Transmission t_{tx_zs3}, entsprechend 160 ms.
Neben den gezeigten Messungen sind ebenso Zeiten von 300 ms, 500 ms, sowie 1000 ms für das Intervall der Messwertaufnahme im Standard-Messbetrieb, in Kombination mit verschiedenen Synchronisationsintervallen, erfolgreich getestet worden.

Der neue Zellensensor ist dahingehend konzipiert, dass dieser möglichst wenig Energie für den Betrieb von der Batteriezelle benötigt. Durch die *Wake-Up*-Funktionalität kann der Zellensensor aus dem Ruhezustand aktiviert werden. In diesem Ruhezustand, in welchem ebenso der *DC-DC-Wandler* deaktiviert ist, konnte eine Stromaufnahme von nur etwa 70 μA, bei einer Zellenspannung U_{cell} von 2 V, gemessen werden. Vergleicht man diesen Wert mit der typischen Selbstentladung einer 80 Ah Bleibatterie (vgl. Abschnitt 2.2), ist die Entladung durch den Zellensensor mit 0,0021 %/Tag etwa um den Faktor 95 kleiner.
Zur Veranschaulichung, wie sich die Stromaufnahme eines Zellensensors im aktivem Betrieb verhält, dient das Bild 4.33. Dieses zeigt die Stromaufnahme bei einer Zellenspannung von ebenfalls 2 V, in Kombination mit den in den zuvor genannten zeitlichen Parametern für die Zellen-Sensorik. Die Messung erfolgte mit einem 1,07 Ω Serienwiderstand, über welchem der Spannungsabfall erfasst worden ist. Aufgrund der Messauflösung des verwendeten Oszilloskops, sowie dem Schaltverhalten des *DC-DC-Wandlers* vom Zellensensor, ist der gemessene Strom-

verlauf mit Rauschen behaftet. Wie man in dem Bild erkennen kann, beträgt die

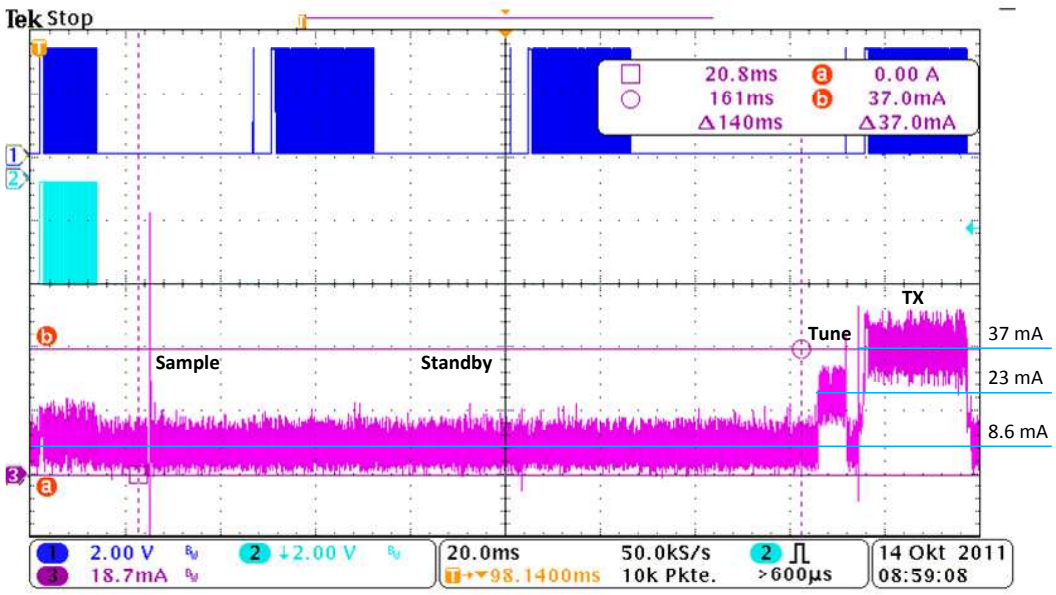
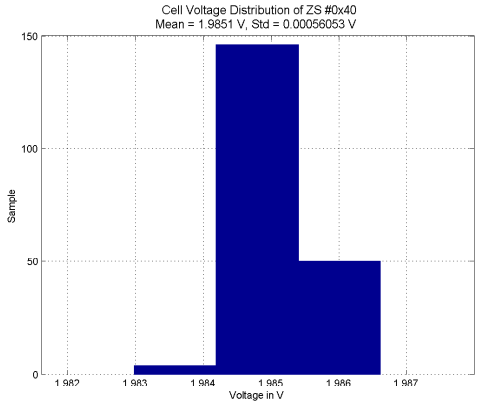


Bild 4.33: *Gemessene Stromaufnahme eines Zellensensors der Zellen-Sensorik bei einer Intervalldauer t_{iv} von* 200 ms *sowie drei Zellensensoren*
Kanal 1: Dekodierung des UL-Signals durch die Datenlogger-Einheit
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Stromaufnahme des dritten Zellensensors bei einer Zellenspannung U_{cell} von 2 V

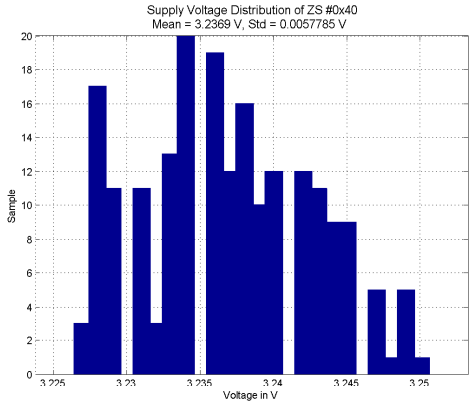
grundlegende Stromaufnahme etwa 8,6 mA. Der Moment der Messwertaufnahme ist ebenfalls ersichtlich, die Reaktivierung des *DC-DC-Wandlers* nach der Messung bewirkt eine kurzzeitige Überhöhung der Stromaufnahme. In dem Moment, wenn der *UHF*-Transmitter in den Zustand „tune“ versetzt wird, steigt für die Verweildauer in diesem Zustand die Stromaufnahme auf etwa 23 mA an. Anschließend, während der Transmission, beträgt die Stromaufnahme nur etwa 37 mA. Somit ergibt sich, mit den verwendeten zeitlichen Parameter, eine mittlere Stromaufnahme von 12,14 mA. Nimmt man für den Vergleich mit der Selbstentladung an, die Zellensensoren wären zu 3,4 % des Tages aktiv[6] , ergibt sich eine mittlere Stromaufnahme von 480 μA. Dieser Wert entspricht einer Entladung von 0,06 %/Tag und ist damit um den Faktor 3,33 kleiner als die Selbstentladung.
Die gemessenen Werte zeigen, dass die sich Stromaufnahme im aktivem Betrieb, in der Größenordnung der des bestehenden Zellensensors der Klasse 1 (vgl. Plaschke (2008)), befindet. Da der neue Zellensensor gegenüber dem alten deaktivierbar ist, kann die Stromaufnahme im Ruhezustand extrem reduziert werden.

[6] entspricht einer Betriebsdauer von 49 min/Tag, Quelle http://www.hvv-futuretour.de

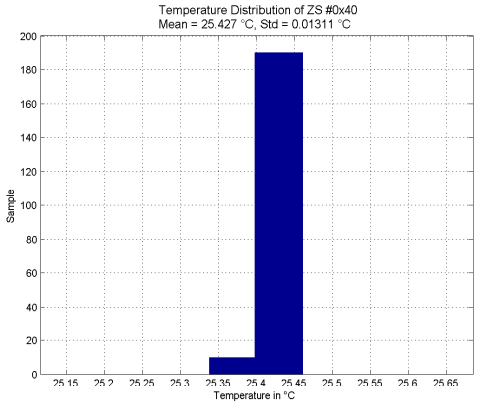
Weiterhin wurde eine Aufnahme von 200 Abtastwerten mit den zuvor gezeigten Parametern der Zellen-Sensorik durchgeführt. Bei gleichbleibender Zellenspannung als auch Temperatur (Raumtemperatur), ergaben sich die in dem folgenden Bild 4.34 exemplarisch dargestellten Verteilungen für die aufgenommenen Messwerte. Insbesondere die Messung der Zellenspannung und der Betriebsspannung zeigen



(a) Zellenspannung U_{cell}



(b) Betriebsspannung U_{supply}



(c) Temperatur T

Bild 4.34: *Verteilung der durch einen Zellensensor gemessenen Parameter bei einer eingestellten Zellenspannung von* 2,0024 V *und* 200 Abtastwerte

im Mittel eine geringfügige Abweichung von $-0{,}86\,\%$ bzw. $-2{,}49\,\%$ zum Sollwert. Da die Zellensensoren unkalibriert betrieben werden, sind diese Fehler zu erwarten. Entscheidender ist, insbesondere bei der Messung der Zellenspannung, die Streuung der Messwerte. Zwischen den auftretenden Messwerten liegt eine Differenz von nur $\pm 1{,}221\,\text{mV}$, entsprechend $\pm 1\,\text{LSB}$. Hingegen ist die Streuung der Messwerte von der Betriebsspannung einer etwa um den Faktor 10 größeren Standardabweichung unterlegen (bzw. $\pm 10\,\text{LSB}$). Diese Streuung beruht vermutlich im

Wesentlichen nicht auf der Messwertaufnahme selber, sondern vielmehr auf der veränderlichen Ausgangsspannung des *DC-DC-Wandlers*.
Die Messung der Temperatur ist ebenso einer sehr geringfügigen Streuung von nur 0,0625 °C bzw. 1 LSB unterlegen. Da der verwendete Temperatursensor bereits auf eine Genauigkeit von 0,5 °C kalibriert ist, stimmen, wie erwartet, die gemessenen Werte mit einer Vergleichsmessung überein.
Mit einer geeigneten Mittlung der erfassten Messwerte, verbunden mit einer Kalibrierung der Spannungsmessungen, ließe sich, wie den Verteilungen zu entnehmen ist, die Genauigkeit der Messungen erhöhen. Im nächsten Abschnitt wird ebenfalls die Messwertaufnahme der Zellensensoren betrachtet, jedoch mit einer veränderlichen Zellenspannung.

4.4.2 Erprobung mittels synthetischen Signalen

Im vorherigen Abschnitt konnte die allgemeine Funktionsfähigkeit des Gesamtverfahrens gezeigt werden. Die Erfassung einer gleichbleibenden Zellenspannung wäre jedoch ebenso mit der bisherigen unkoordinierten Zellen-Sensorik möglich gewesen.
Um hingegen den Vorteil der neuen, durch den *DL*-Kanal koordinierten, Zellen-Sensorik darzustellen, wird als Zellenspannung ein synthetisch erzeugtes Dreiecksignal verwendet. Somit kann gezeigt werden, dass alle Zellensensoren die Messwertaufnahme zum gleichem Zeitpunkt durchführen und nicht, wie bisher, zufällig verteilt.
Da die Zellensensoren nicht direkt von dem 50 Ω Ausgang eines Funktionsgenerators betreibbar sind, muss ein Impedanzwandler, wie bereits in einer vorherigen Arbeit (Püttjer (2011)), realisiert durch einen „Bipolar Operational Power Amplifier“ von Kepco, eingesetzt werden.

Das Bild 4.35 zeigt die von der Zellen-Sensorik aufgenommenen Messwerte eines Dreiecksignals der Frequenz von 0,1 Hz, so wie das bereits in dem Abschnitt 4.3.3 erläuterte Matlab-Skript darstellt. Die Amplitude des Signals ist derart gewählt, dass nahezu der gesamte zulässige Spannungsbereich von kleiner 1 V bis größer 4,5 V ausgenutzt wird. Wie bereits in dieser Darstellung zu erkennen ist, erfolgt die Erfassung der Messwerte simultan durch alle drei verwendeten Zellensensoren.
Um die erfassten Messwerte mit der zugrunde liegenden Zellenspannung vergleichen zu können, ist diese parallel mit einem Oszilloskop erfasst worden. In dem Bild 4.36 ist neben dem Verlauf der aufgezeichneten Spannungen, die Differenz zwischen der von den Zellensensoren und dem Oszilloskop gemessen Spannung dargestellt. Wie man erkennen kann, unterliegen die von den Zellensensoren gemessenen Spannungswerte allesamt annähernd dem gleichen Fehlerverlauf. Die

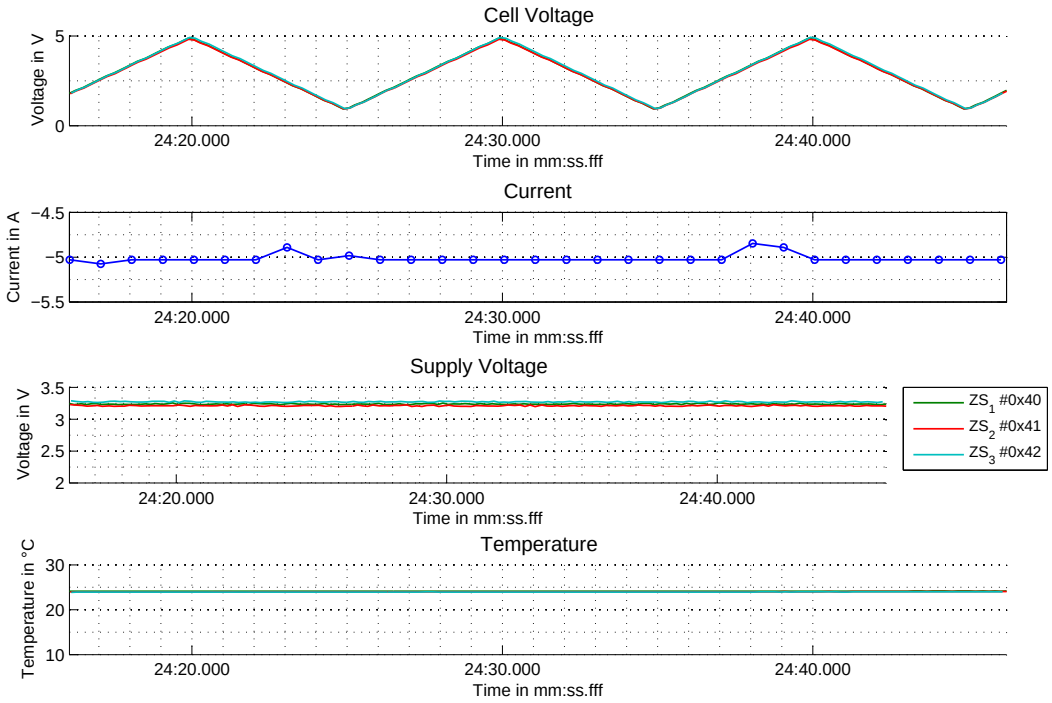


Bild 4.35: *Von der Zellen-Sensorik aufgezeichneter Spannungsverlauf eines Dreiecksignals mit einer Signalfrequenz von* 0,1 Hz

Differenz zwischen den Messwerten entsteht, in Anbetracht der unkalibrierten Zellensensoren, einerseits durch einen individuellen Offsetfehler, andererseits durch einen Linearitätsfehler, größtenteils hervorgerufen durch den Spannungsteiler für die Messwerterfassung. Weiterhin ist zudem die Messung durch das Oszilloskop, trotz entsprechender Einstellungen und anschließender Filterung der Messwerte, mit einem Fehler im Bereich von einigen Millivolt behaftet.
Die wesentliche Aussage dieser Messung ist jedoch, dass der Fehlerverlauf von den Zellensensoren zueinander sehr gleichmäßig und somit von nahezu simultanen Abtastzeitpunkten auszugehen ist. Weiterhin zeigt sich, dass alle Messdaten von den Zellensensoren erfolgreich ohne Kollision über den koordinierten *UL*-Kanal übertragen wurden.

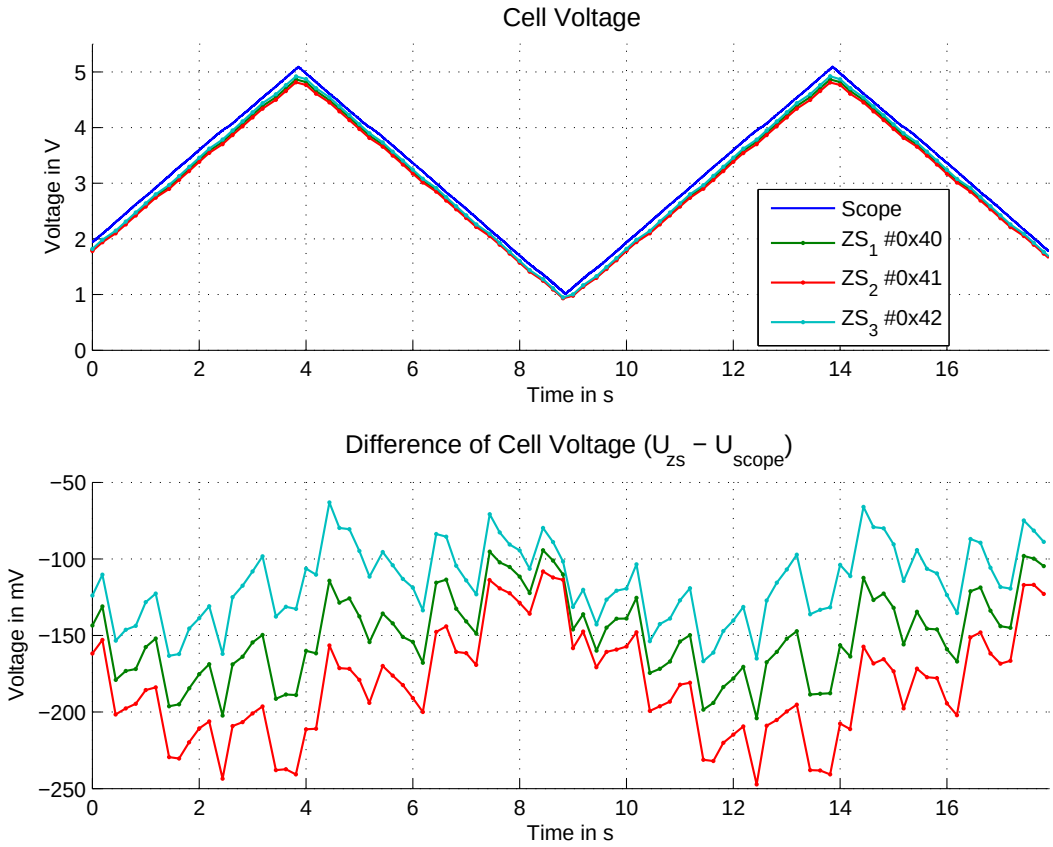


Bild 4.36: *Vergleich zwischen aufgezeichneten Spannungsverläufen der Zellen-Sensorik und eines Oszilloskops, bei einer Signalfrequenz von* 0,1 Hz

4.4.3 Erprobung mittels realen Signalen

Für die Erprobung der Zellen-Sensorik mit realen Signalen, sprich an einem konkretem Messobjekt, soll der dem Projekt zur Verfügung stehenden Gabelstapler verwendet werden.
Der Gabelstapler verfügt über eine 24 V Antriebs- und Traktionsbatterie, aufgebaut aus 12 Batteriezellen, entsprechend der Darstellung im Bild 4.37. Es ist möglich, zur Kontaktierung die Zellensensoren mit einem Adapter direkt auf bereits modifizierte Polanschlüsse der Zellen aufzustecken.

4.4.3.1 Erste Vorversuche zum Funktionstest

Bei der erstmaligen Erprobung der Zellen-Sensorik an der Batterie des Gabelstaplers, zeigte sich unter Verwendung der Reader-Einheit ohne Leistungsverstärker

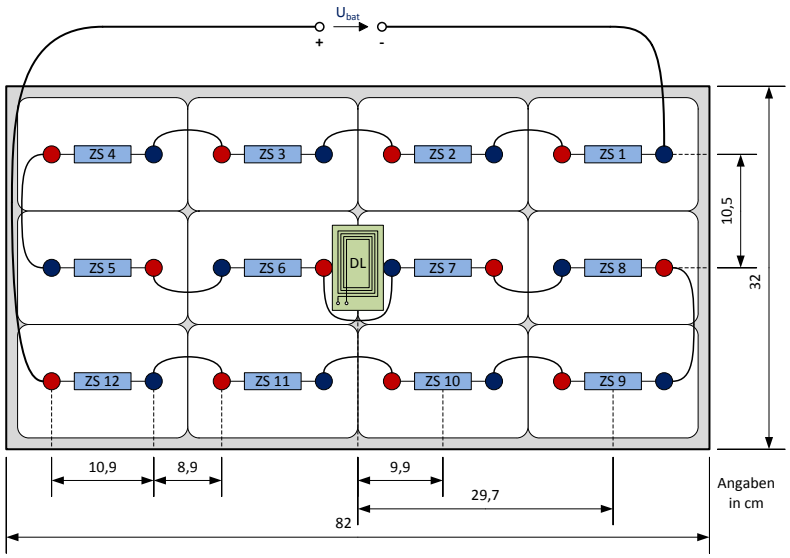


Bild 4.37: *Schematische Darstellung der Batterie des Gabelstaplers*

(„BS Reader 13.56Mhz v0.2“) eine Abhängigkeit der Funktionsfähigkeit vom Zustand der Umgebung.
Hierbei wurde zunächst, wie im Bild 4.38 zu sehen ist, die *DL*-Antenne der Basisstation entsprechend der Darstellung im Bild 4.37 positioniert, in Verbindung mit den Zellensensoren, wahlweise auf den verschiedenen äußeren Batteriezellen.
Bei geöffneten Batteriefachdeckel war ein Funktionstest aller Zellensensoren auf allen möglichen Zellen erfolgreich durchführbar. Durch Schließen des Deckels veränderte sich der Zustand der Funktionsfähigkeit, bis dahin, dass bei geschlossenem Deckel kein Funktionstest erfolgreich war.

Dieses Verhalten ist bedingt durch die metallische Umgebung, welche bei geschlossenem Batteriefachdeckel die Zellensensoren vollkommen umschließt. Sowohl die Seitenwände des Batteriefachs als auch der Batteriefachdeckel mit dem aufmontierten Fahrersitz, sind aus Stahlblech konstruiert. Neben den so zusätzlich im Metall auftretenden Verlusten, führt dies zu einer veränderten Ausbreitung der für die Verkopplung benötigten Magnetfeldlinien im Raum. Gleichzeitig beeinflusst letzteres zudem den Induktivitätswert der Antennenspulen, was eine Verstimmung der Resonanzkreise zur Folge hat.

In einer Messung mit einem Spektrumanalysator[7] in Kombination mit einer VSWR-Brücke[8], konnte diese Verstimmung von der Antennenspule für den *DL*-Kanal der Reader-Einheit nachgewiesen werden.
Wie in der Tabelle 4.3 zu sehen ist, verschiebt sich die Lage der Resonanzfre-

[7] Typ FSC3 von Rohde & Schwarz
[8] Typ HZ547 von Hameg

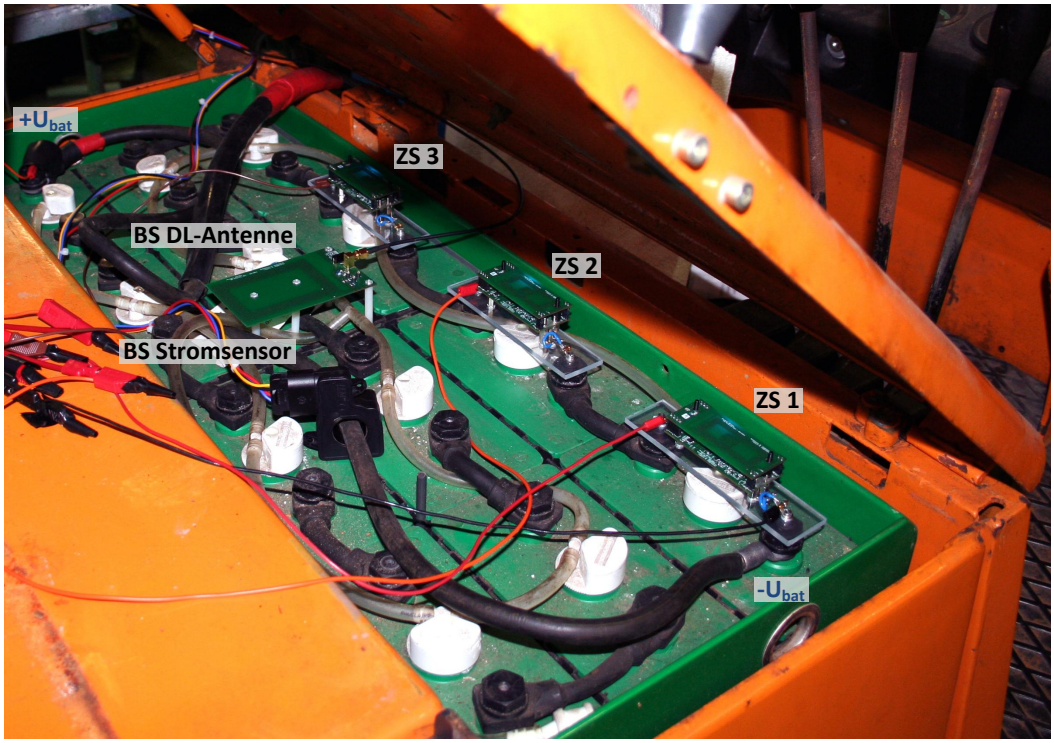


Bild 4.38: *Batterie des Gabelstaplers, bestückt mit drei Zellensensoren $ZS_{1...3}$, sowie dem zentralen Stromsensor*

quenz, deren Sollwert 13,56 MHz ist, abhängig von der Serienkapazität des Resonanzkreises und dem Zustand des Batteriefachdeckels. Aufgrund dessen, dass die

Wert der Serienkapazität	Batteriefachdeckel offen	Batteriefachdeckel geschlossen
23,3 pF	13,607 MHz (+47,62 kHz)	13,674 MHz (+114,28 kHz)
23,7 pF	13,492 MHz (−66,69 kHz)	13,568 MHz (+9,52 kHz)

Tabelle 4.3: *Verstimmung der Resonanzfrequenz von der Antennenspule für den DL-Kanal der Reader-Einheit, in Abhängigkeit von der Umgebung*

Bandbreite B der Resonanzkurve nur etwa 190 kHz beträgt, entsprechend einer Güte Q von 71, bewirkt eine Verschiebung von z. B. +114,28 kHz bereits eine drastische Reduktion der verfügbaren Empfangsenergie.
Zur Probe wurde ein Resonanzkreis mit einer Serienkapazität von 23,7 pF derart vorverstimmt, dass dieser bei geschlossenem Batteriefachdeckel nur eine Abweichung von +9,52 kHz aufweist. In Folge dessen konnte zumindest, bei unmittelbar an der Antenne der Basisstation befindlichen Zellensensoren, eine Reaktion ausgelöst werden.

Als Schlussfolgerung aus dem fehlgeschlagenen Versuch ergab sich, die Sendeleistung der Basisstation für den *DL*-Kanal zu erhöhen. Um im Vorfeld abzuschätzen, wie viel Sendeleistung in etwa benötigt wird, ist daraufhin ein weiterer Versuch mit einem Leistungs-Funktionsgenerator als Sender für den *DL*-Kanal durchgeführt worden. Bei diesem Versuch war es mit einer Sendeleistung von etwa 750 mW möglich, die Zellensensoren auf allen Batteriezellen bei geschlossenem Batteriefachdeckel, zu aktivieren.
Neben der reinen Erhöhung von der Sendeleistung könnte zusätzlich mit einer automatischen Resonanzanpassung die Höhe der zusätzlich benötigten Sendeleistung reduziert werden. Diese Anpassung könnte wahlweise, zur Minimierung des Realisierungsaufwands, entweder nur an dem Resonanzkreis der Basisstation oder ebenso an denen der Zellensensoren erfolgen. Eine derartige Veränderung des Systems bedarf einer umfassenden Analyse und würde über den Rahmen dieser Arbeit hinausgehen.

Aufgrund dieser Erkenntnisse ist schließlich der Leistungsverstärker für die Reader-Einheit, siehe Abschnitt 4.2.2.2, ausgewählt und realisiert worden. Mit der neu aufgebauten Reader-Einheit war es schließlich möglich, einen erfolgreichen Funktionstest der Zellensensoren durchzuführen, wobei ebenfalls eine Sendeleistung von etwa 750 mW genügte. Es bedarf folglich nicht der maximalen Sendeleistung des Leistungsverstärkers von etwa 4,4 W, womit eine Leistungsreserve von 3,65 W zur Verfügung steht.

4.4.3.2 Aufnahme einer Messreihe als Funktionsnachweis

Für einen direkten Funktionsnachweis der Zellen-Sensorik unter nicht idealen Umgebungsbedingungen, wie sie im Bereich der Batterie des Gabelstaplers vorzufinden sind, erfolgt eine Aufzeichnung der von den Zellensensoren gemessenen Spannungsverläufe ausgewählter Batteriezellen. Zur Verifizierung findet parallel eine Erfassung der Zellenspannungen von den betrachteten Batteriezellen mit einem Oszilloskop statt.
Durch diese Erprobung kann einerseits die Funktionsfähigkeit des *UL*-Kanals für die Übertragung der Messdaten von den Sensoren, als auch die des *DL*-Kanals für die Koordination und den *Wake-Up*, unter dem Einfluss einer metallischem Umgebung, nachgewiesen werden.

Die Anordnung der Zellensensoren für die Erprobung ist dem Bild 4.38 zu entnehmen. Die drei verwendeten Sensoren $ZS_{1...3}$ sind auf den Zellen mit den Spannungen $Ucell_{1...3}$ montiert. Das Oszilloskop ist derart verschaltet, dass es die Spannungsverläufe dieser Zellen, sowie die Gesamtspannung U_{bat} der Batterie, gegen den negativem Anschluss $-U_{bat}$ der Batterie gemessen aufzeichnet. Für die zentrale Erfassung des Stromes in und aus der Batterie ist zudem der Stromsensors

der Basisstation, wie ebenfalls auf dem Bild 4.38 zu sehen, mit dem Anschlusskabel der Klemme $-U_{bat}$ verbunden.
Als Parameter für die Koordination der Zellen-Sensorik sind, da nur drei Zellensensoren zur Verfügung standen, die folgenden Größen (vgl. Abschnitt 4.3.2.3) für die Erprobung verwendet worden:

$$\begin{aligned} t_{sa} &= 10\,\text{ms}, \quad t_{sa-tx} = 25\,\text{ms}, \quad t_{tx-iv} = 40\,\text{ms} \\ t_{iv} &= 200\,\text{ms} \\ t_{cur} &= 1000\,\text{ms} \\ t_{synch} &= 2000\,\text{ms} \end{aligned}$$

Würden hingegen alle 12 Zellen der Batterie mit Sensoren bestückt, müsste die Intervalldauer t_{iv} mindestens 460 ms[9] betragen, wenn man von einem minimalen Abstand der *UL*-Transmissionen $ttx - tx$ von 35 ms ausgeht. Durch die gewählte Intervalldauer t_{iv} von 200 ms wird eine vergleichbare Belegung des *UL*-Kanals erzielt.

Zunächst ist für die Erprobung der Zellen-Sensorik die Ruhespannung der betrachteten Batteriezellen im unbelastetem Zustand aufgezeichnet worden. Das Bild 4.39 zeigt die erfassten Messwerte der Zellen-Sensorik.
Die Differenz zu den durch das Oszilloskop erfassten Messwerten für die Zellenspannungen beträgt für $Ucell_{1,2}$ kleiner 25 mV, für $Ucell_3$ hingegen 100 mV. Da die Messung von $Ucell_3$ durch das Oszilloskop, aufgrund der nicht vorhandenen Potentialtrennung, nur indirekt über die Messung von $Ucell_{1,2}$ erfolgen kann, ist hierbei von einem größerem Messfehler auszugehen. Eine potentialfreie Referenzmessung, mit höherer Genauigkeit als es das Oszilloskop bieten kann, wäre hierbei angebracht, ist aber für den Nachweis der Funktionalität zweitrangig.
Den aufgezeichneten Spannungswerten ist zu entnehmen, dass sich einerseits die Batterie mit 23,39 V Gesamtspannung in einem niedrigem Ladezustand befindet, andererseits die Zellen ein ungleiches Ladungsniveau aufweisen.

Aufgrund des Ladezustands der Batterie wird während der Aufzeichnung der zweiten Messreihe das zu dem Gabelstapler gehörende Ladegerät eingeschaltet und somit eine (positive) Belastung der Batterie provoziert. In dem folgendem Bild 4.40 ist der von der Zellen-Sensorik erfasste Spannungs- und Stromverlauf dargestellt.
Betrachtet man die Zellenspannungen, stellt man fest, dass jede Spannung einen differenzierten Verlauf ausweist. Der tatsächliche Spannungswert während des Ladevorgangs lässt sich anhand der erfassten Messwerte nicht eindeutig bestimmen. Gleiches gilt für den aufgezeichneten Strom.

[9] 500 ms seitens der Basisstation einstellbar

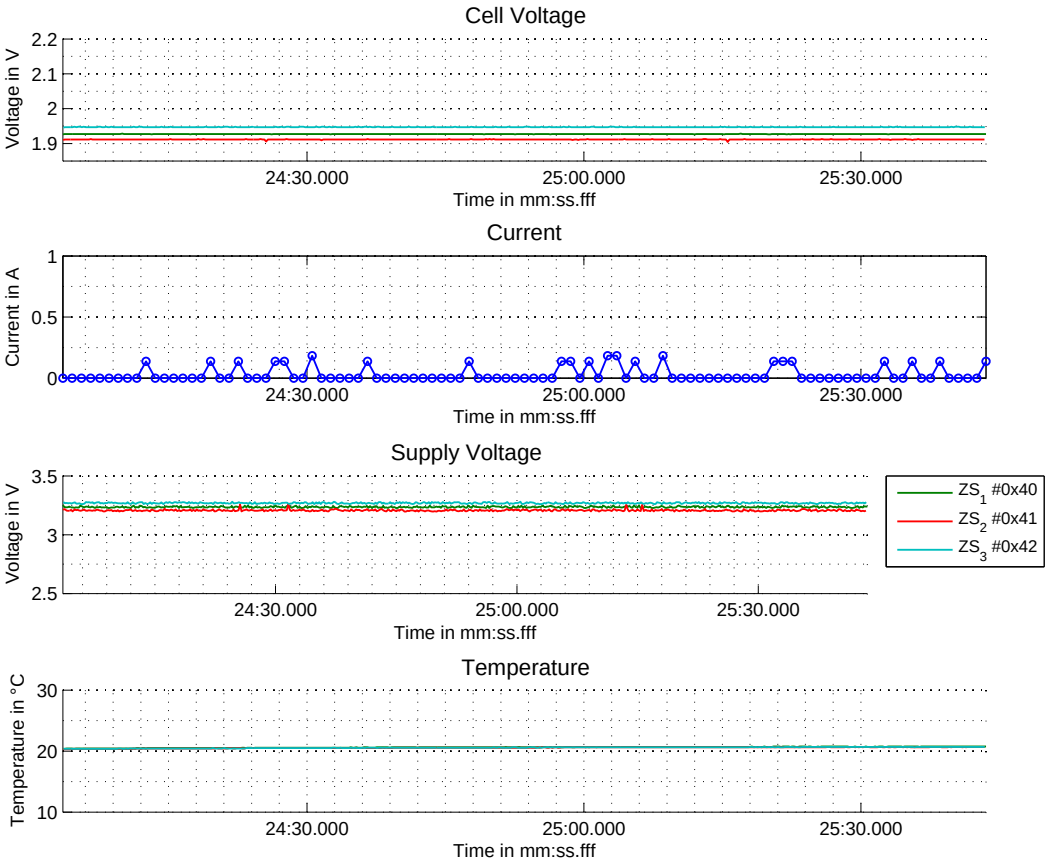


Bild 4.39: *Durch die Zellen-Sensorik im Ruhezustand von der Batterie des Gabelstaplers aufgezeichneter Spannungsverlauf*

Ursächlich für die unzureichende Messwerterfassung ist der tatsächliche Spannungsverlauf der Batteriezellen, in Kombination mit dem gewähltem Verfahren für die Messwerterfassung. Wie in dem Bild 4.41 anhand der durch das Oszilloskop aufgezeichneten Spannungsverläufe deutlich wird, verfügt das Ladegerät vermutlich über eine einfache Zweiweggleichrichtung, welche eine mit 100 Hz pulsierende Ladespannung zur Folge hat. Der Wechselspannungsanteil der betrachteten Zellenspannungen liegt je nach Zelle zwischen 80 mV_{ss} und 150 mV_{ss}, der Gesamtspannung bei 1,29 V_{ss}.
Da die Zellensensoren nur alle 200 ms einen Messwert aufnehmen, was ein Vielfaches der Periodendauer von 10 ms der Signalfrequenz darstellt, können diese den tatsächlichen Spannungsverlauf nicht wiedergeben und es kommt, wie bei U_{cell3} deutlich zu sehen ist, zu Aliasing-Effekten. Dieser Effekt tritt ebenso bei der Strommessung auf, welche mit einer Periodendauer von 1000 ms ebenfalls im Bereich des Vielfachen der Signalfrequenz erfolgt.
Abhilfe für eine bessere Erfassung der Zellenspannungen, beispielsweise während

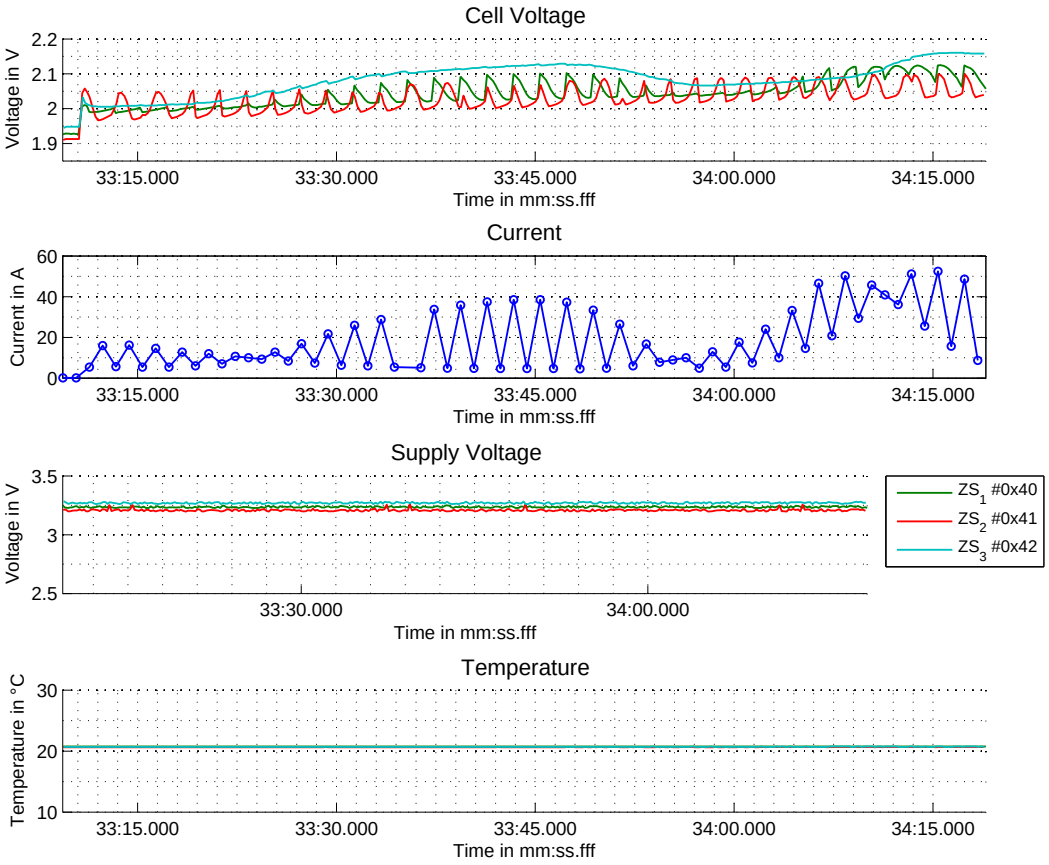


Bild 4.40: *Durch die Zellen-Sensorik während der Aktivierung des Ladevorgangs aufgezeichneter Spannungsverlauf von der Batterie des Gabelstaplers*

des gezeigten Ladevorgangs, könnte ein verändertes Messverfahren darstellten, worauf im Abschnitt 4.5.2 weiter eingegangen wird.

Weiterhin ist eine Messreihe aufgenommen worden, die nochmals den Einfluss der metallischen Umgebung, hier anhand des Batteriefachdeckels vom Gabelstapler auf den *DL*-Kanal veranschaulicht. In dem zugehörigen Bild A.6 im Anhang A ist zu sehen, wie die Ausgangsspannung der Spannungsvervielfacherschaltung für den *Wake-Up*, als auch die Spannung am Tiefpassfilter vom Hüllkurvendetektor für den Datenempfang, sich verändert wenn der Batteriefachdeckel geschlossen und geöffnet wird. Dabei zeigte sich, dass die Veränderung so weitreichend sein kann, ein ohnehin bereits schlecht - aber noch ausreichend - aus dem *HF*-Feld versorgter Zellensensor, bei geschlossenem Deckel nicht mehr funktionsfähig ist.
Der betreffende Sensor ZS_1 befand sich hierbei auf einer der äußeren Zelle von der Batterie, hatte somit folglich die größtmögliche Distanz zur *DL*-Antenne der Basisstation und zudem einen nur ungenügenden Resonanzabgleich für den *DL*-

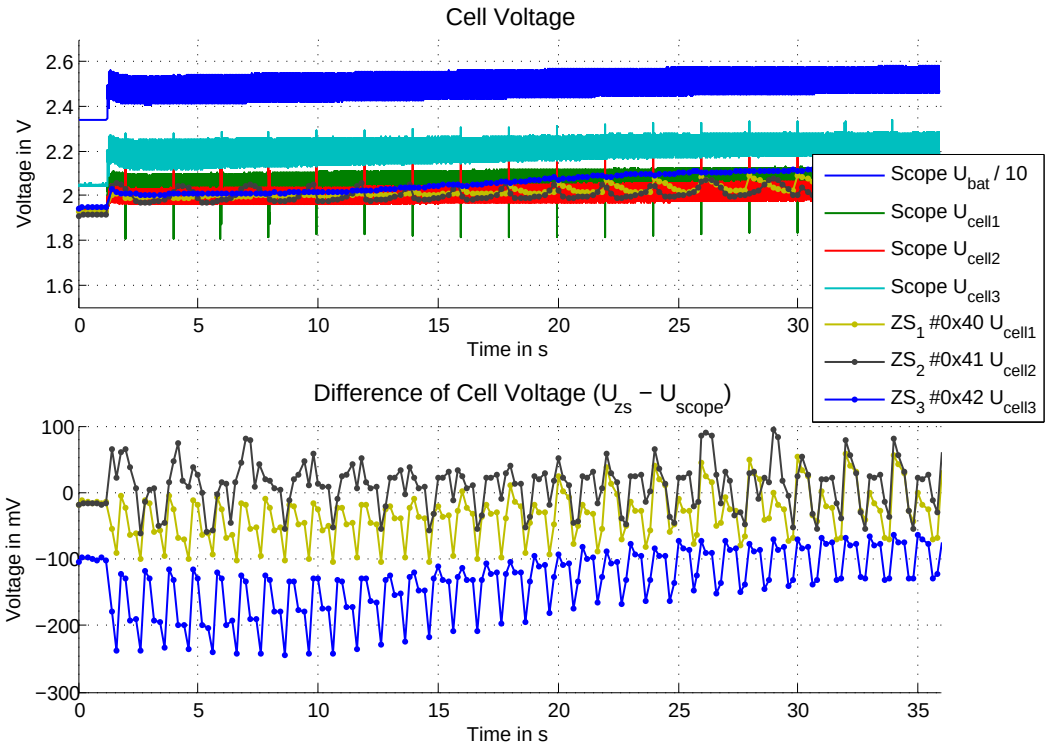


Bild 4.41: *Vergleich der durch die Zellen-Sensorik und durch das Oszilloskop aufgezeichneten Spannungsverläufe während der Aktivierung des Ladevorgangs von der Batterie des Gabelstaplers*

Antennenkreis erhalten. Auf diese Problematik wird ebenso in dem folgendem Abschnitt 4.5.2 eingegangen.

Insgesamt konnte durch die Erprobung gezeigt werden, dass das konzipierte System der Zellen-Sensorik auch unter den realen Bedingungen in der Umgebung von der Batterie des Gabelstaplers grundsätzlich funktionsfähig ist. Für eine umfassendere Bewertung müsste das System nicht nur im Ladebetrieb, sondern insbesondere im Fahr- und Hebebetrieb, näher untersucht werden.

4.5 Wertende Analyse und Optimierung

Nach der Realisierung und Erprobung des neuen Zellensensors, sowie den neuen Komponenten der Basisstation, gilt es das System mit den bestehenden Zellensensoren zu vergleichen. Weiterhin werden Optimierungsmöglichkeiten genannt, sowie eine Einschätzung der durch diese Arbeit für das Projekt gewonnenen relevanten Erkenntnisse.

4.5.1 Variantenvergleich

Bisher stand innerhalb des Projektes „BATSEN“ nur ein Variante von Zellensensoren der Klasse 1 zur Verfügung. Da diese über keinen *DL*-Kanal verfügen, wird notwendigerweise ein unkoordiniertes Mess- und Übertragungsverfahren angewandt.
Mittlerweile ist dieser Zellensensor der Klasse 1 um eine weitere Variante ergänzt worden. Diese ist speziell für die Anwendung in *Unterbrechungsfreien Stromversorgungen (USV)* optimiert, welche hierbei vielmehr die blockweise Überwachung von mehreren Zellen übernehmen soll. Für herkömmliche Starter-, sowie Antriebs- und Traktionsbatterien ist diese Variante ungeeignet.

Der in dieser Arbeit neu entwickelte Zellensensor mit bidirektionaler Kommunikation unterscheidet sich in vielen Punkten grundsätzlich von den beiden anderen Varianten der Klasse 1. In der folgenden Tabelle 4.4 sind die wesentlichen Unterscheidungsmerkmale der drei Varianten hervorgehoben.

Aufgrund des jetzt erstmals vorhandenen *DL*-Kanals, ergibt sich eine deutlich abweichende Betriebsweise zwischen den drei Varianten von Zellensensoren. Die Zellensensoren können nun durch Kommandos der Basisstation in verschiedene Betriebsarten versetzt werden, welche zudem in ihren Parametern flexibel anpassbar sind.

Das für die *UL*-Kommunikation verwendete synchronisierte Zeitschlitzverfahren bietet gegenüber dem Verfahren mit pseudozufälligen Sendezeitpunkten eine erhöhte Übertragungsrate, verbunden mit einer Reduzierung der *FER*. Verwendet man für einen Vergleich mit dem bestehenden Zellensensor (vgl. Abschnitt 2.2) ebenfalls eine Anzahl von sechs Zellensensoren, ergibt sich für das jetzige System eine mögliche mittlere Intervalldauer von 250 ms, in welcher die Messwerte übertragen werden. Daraus folgt eine Datenrate von 432 Bit/s, welche im Vergleich zu den 127,44 Bit/s des alten Zellensensors, um den Faktor 3,39 größer ist.
Dabei ist jedoch zu beachten, dass die genannten Werte nur als eine mögliche Konfiguration zu betrachten sind.
Die Vorteile der durch den *DL*-Kanal synchronisierten *UL*-Übertragung, wirken

sich umso stärker aus, je mehr Zellensensoren in einem System bestehen und je höher die Messdichte wird.

Für die Messwertaufnahme ergibt sich, durch die Synchronisation über den *DL*-Kanal, nun ebenfalls erstmals die Möglichkeit der synchronisierten Messwerterfassung.

Der Hardwareaufwand seitens des Zellensensors und der Basisstation ist zwar gegenüber den anderen beiden Typen deutlich vergrößert, erhöht jedoch gleichzeitig den Umfang der Funktionalitäten erheblich.

	Bestehende Zellensensor „BATSEN BS0406“	Zellensensor für USV-Systeme „BATSEN BSBM20V“	Entwickelter Zellensensor „BATSEN ZS v0.2“
Klasse	Klasse 1	Klasse 1	Klasse 2b
Übertragung über Downlink-Kanal	nicht vorhanden	nicht vorhanden	Broadcast-*Wake-Up* und adressierte Kommandos bzw. Messages (13,56 MHz)
Koordination des Uplink-Kanals (433 MHz)	unkoordiniert, pseudozufällige Sendezeitpunkte	unkoordiniert, pseudozufällige Sendezeitpunkte	Kommandiert, synchronisiertes Zeitschlitzverfahren
Koordination der Messwerterfassung	asynchron, Detektion und Erfassung von Hochstromereignissen	asynchron	Kommandiert und mit Synchronisation
Ladungsbalancierung	nicht vorhanden	nicht vorhanden	kommandierte Ausführung, Anwendung vorbereitet, Entscheidungsmodell noch offen
Zulässige Zellenspannung	0,8 … 3,2 V	3,6 … 20,0 V	0,8 … 5,0 V
Messauflösung Zellenspannung	3,661 mV/LSB	19,530 mV/LSB	1,221 mV/LSB
Hardwareaufwand für Zellensensor	gering	sehr gering	hoch
Hardwareaufwand für Basisstation	gering	gering	hoch

Tabelle 4.4: *Vergleich der im Projekt bestehenden Varianten von Zellensensoren*

4.5.2 Optimierung

Bereits während der Konzipierung des neuen Zellensensors, der nachfolgenden Realisierung und im Laufe der anschließenden Erprobungen, konnten einige Aspekte ausfindig gemacht werden, welche bei einer Weiterführung des Projektes zu berücksichtigen sind.
Die folgende Auflistung gibt nach derzeitigem Kenntnisstand eine Übersicht, über einige Optimierungen des Systems der Zellen-Sensorik.

- Resonanzabgleich der *DL*-Antennenkreise
  - Sowohl für den Schwingkreis der *DL*-Antenne von der Basisstation als auch für den des Zellensensors wäre eine Erweiterung um eine variable Abgleichkapazität zwecks Resonanzanpassung sinnvoll.
    Hiermit könnte einerseits der aufwendige manuelle Resonanzabgleich während der Produktion entfallen, andererseits wäre damit eine Anpassung an veränderliche Umgebungsbedingungen möglich. Bei der Erprobung des Gesamtsystems auf dem Gabelstapler (vgl. Abschnitt 4.4.3) zeigte sich eine Veränderung der Resonanzfrequenz abhängig von der Umgebung. Ein durch Kommandos initiierter halbautomatischer Abgleich wäre seitens der Zellensensoren denkbar. Für den Parallelschwingkreis der *DL*-Antenne von der Basisstation wäre, neben einer Eigenentwicklung, der Einsatz von einem Reader-IC[10] mit bereits integriertem Anpassungsnetzwerk möglich.
- Messwerterfassung der Zellensensoren
  - Bereits in dem Abschnitt 4.3.1.5 ist eine mögliche Zustandsfolge für den Zellensensor im Hochstrom-Messbetrieb, für die Messwertaufnahme beispielsweise während eines Startvorgangs, aufgezeigt worden. Diese Abfolge, in Verbindung mit den vorgeschlagenen Kommandos A.1, gilt es umzusetzen und zu testen.
  - Weiterhin ist insbesondere bei der Erprobung auf dem Gabelstapler deutlich geworden, dass für die Messgröße der Zellenspannung eine Mittelwertbildung seitens des Zellensensors notwendig ist. Hierbei variierte die Amplitude der Zellenspannung mit einer Frequenz von 100 Hz, ausgelöst durch das Ladegerät. Eine Verkürzung des Messintervalls, verbunden mit einer höheren Übertragungsrate der Messgrößen, ist aufgrund der Kanalbelegung des *UL* nicht möglich. So wäre zur Lösung des Problems eine durch Kommandos und Messages steuerbare Mittelwertbildung, über ein einstellbares Zeitfenster mit variabler Abtastrate,

[10] beispielsweise R14AB von IDS Microchip (siehe IDS Microchip (2010))

zweckmäßig. Neben dem Mittelwert könnten zusätzlichen Informationen, wie Extremwerte und Standardabweichung generiert und an die Basisstation übertragen werden.

- Kalibration
  - Bislang übertragen die Zellensensoren ihre Messwerte ohne Fehlerkompensation an die Basisstation. Eine Kalibrierung der Sensoren, sowie die Kompensation der Messwerte bereits auf dem Sensoren anhand der Kalibrierdaten, ist noch vorzunehmen.
- Messwerterfassung der Basisstation
  - Was bereits zuvor für die Messwerterfassung der Zellensensoren beschrieben wurde, gilt gleichsam für die Erfassung des Stromes in und aus der Batterie. Eine Anpassung der Abtastraten an die jeweiligen Gegebenheiten, sowie eine möglichst kontinuierliche Abtastung zwecks Ermittelung der umgesetzten Ladungsmengen, ist noch zu realisieren.
  - Zusätzlich wäre eine Erfassung von der Gesamtspannung der Batterie, ausgeführt durch die Basisstation selber, denkbar. Mit dieser Messgröße könnten, neben der Gewinnung von Informationen für die Parametrisierung der Zellensensoren, bei einer kontinuierlicher Abtastung, in Kombination mit der Strommessung, die umgesetzten Ladungsmengen bestimmt werden.
- Basisstation
  - Für die Datenlogger-Einheit wäre die Verwendung einer *MCU* mit mehr als den jetzigen drei Zeitgebern sinnvoll. Als Systemzeitgeber kann momentan nur der *WDT* verwendet werden, welcher derzeit mit einer Schrittweite von 62,5 ms zählt. Für eine flexiblere Anpassung der Zellen-Sensorik wäre ein kleinere Schrittweite zu bevorzugen.
  - Die Reader-Einheit könnte wiederum, um eine flexibel Anpassung der Sendeleistung durch die Datenlogger-Einheit, ergänzt werden.
  - Anstelle der jetzigen *UHF*-Empfänger-Einheit wäre die Verwendung einer Einheit mit integrierter Dekodierung sinnvoll. Hierdurch ließe sich die für den *UL*-Kanal benötigte Prozessorzeit der Datenlogger-Einheit reduzieren und für andere Anwendungen nutzen.
- Systemdesign
  - Seitens des Zellensensors wäre eine Auslagerung von derzeit softwareseitigen Funktionalitäten in Hardwareblöcke denkbar. Beispielsweise könnte die Dekodierung des *DL*-Signals, inklusive einer Parallelisierung der empfangenen Informationen und Überprüfung der Zieladresse, durch einen Logikbaustein realisiert werden.

- Weiterhin könnte mittels der Systembeschreibungssprache „SystemC“ die Modellierung entweder von Teilen oder der gesamten Digitalschaltung des Zellensensors erfolgen. Die Übertragung der realisierten Funktionalitäten ist durch eine an Automaten angelehnte Software vereinfacht. Unter Beachtung der Synthesefähigkeit der dabei beschriebenen Komponenten ließe sich das entworfene System mittels eines *FPGA* realisieren und verifizieren. Dieses könnte wiederum für die später geplante Integration der Analog- und Digitalschaltungen in einen Chip verwendet werden.

4.5.3 Projektrelevante Erkenntnisse

Weiterhin ist einzuschätzen, welche der erlangten Erkenntnisse aus dieser Arbeit für das Projekt „BATSEN“ von Relevanz sein könnten. Im Folgenden sind einige wesentliche Punkte aufgeführt, welche als relevant zu bewerten sind.

- Frequenzbereiche für den *DL*- und *UL*-Kanal
  - Bei der Konzipierung des neuen Zellensensors ist die Entscheidung für die Verwendung von getrennten Frequenzbereichen für den *DL*- und *UL*-Kanal getroffen worden. Der für den *DL*-Kanal genutzte Frequenzbereich von 13,56 MHz ist erstmals in diesem Projekt angewandt. Ein alternatives Frequenzband von 135 kHz wurde nach Voruntersuchungen (vgl. Abschnitt 3.1) nicht weiter betrachtet. Dabei konnten unter anderem Erkenntnisse im Bezug auf den Wirkungsbereich der induktiv gekoppelten Übertragung, den Einfluss von metallischen Umgebungen, das Antennendesign und das Schaltungsdesign erzielt werden.
- Realisierungsaufwand für *DL*-Kanal
  - Es sind Erkenntnisse im Bereich der Schaltungstechnik für den Aufbau der *Wake-Up*-Funktionalität sowie für die Demodulation des *DL*-Signals geschaffen worden.
  - Von besonderer Relevanz ist der Realisierungsaufwand der Zellensensoren, welchen es zu minimieren gilt. Die Realisierung des *DL*-Kanals bedarf momentan einer Vielzahl von zusätzlichen diskreten Bauelementen. Mittels der geplanten Integration ließen sich vermutlich die Anzahl der diskreten Bauelemente stark reduzieren. Durch den erfolgten Aufbau bestehen erstmals Anhaltspunkte für den beispielhaft benötigten Realisierungsaufwand eines Sensors der Klasse 2.

- Erzielter Nutzen durch den *DL*-Kanal
  - Durch die Anwendung der bidirektionalen Kommunikation konnte zugleich der sich ergebene Nutzen, anhand von deutlich abweichenden Betriebsarten sowie Kommandos und Messages, veranschaulicht werden.
  - Synchronisierte Messwerterfassung, erstmals durch den *DL*-Kanal möglich.
  - Vergrößerung der Übertragungsraten des *UL*-Kanals, durch Verwendung eines synchronisierten Übertragungsverfahrens.
  - Mittels des konzipierten quasi passivem *HF*-Empfänger, der zugleich eine *Wake-Up*-Funktionalität bietet, ließ sich die Realisierungsmöglichkeit für einen sehr energiesparenden Zellensensor aufzeigen.
- Verwendung eines *UHF*-Transmitters ohne Quarzoszillator für den *UL*-Kanal
  - Der Aufbau des Zellensensors soll nach Möglichkeit ohne einen Quarzoszillator erfolgen. Bisher ist für den *UHF*-Transmitter stets ein solcher notwendig gewesen, um die für das genutzte *ISM*-Band geforderten Anforderungen zu erfüllen. Mit dem recherchierten und eingesetzten *UHF*-Transmitter „Si4012“ des Herstellers *SI*, welcher über einen präzisen internen LC-Oszillator verfügt, konnte eine Möglichkeit für die Realisierung eines Zellensensor ohne Quarzoszillator gezeigt werden.
- Anordnung der Antennenkreise auf dem Zellensensor
  - Es konnte gezeigt werden, dass die Antennen der beiden *DL*- und *UL*-Känale auch in unvorteilhafter räumlicher Anordnung (vgl. Abschnitt 4.1.2) keine praktischen Probleme erzeugten.
- Erweiterung des Bereiches der zulässigen Zellenspannung
  - Mit der Verwendung eines *DC-DC-Wandlers*, welcher über eine Step-Up-Down-Funktionalität verfügt war es möglich, den Bereich der zulässigen Zellenspannung gegenüber dem bisherigem Zellensensor zu erweitern und zugleich Kenntnisse über die Verwendung dieses neuen Typs zu erhalten.
- Einsatz einer neuen Serie von *MCU* für den Zellensensor
  - Da der neue Zellensensor eine *MCU* mit abweichender zusätzlicher Peripherie benötigt, ist hierfür eine *MCU* der MSP430 2xx-Serie des Herstellers *TI* ausgewählt worden. Durch dessen Verwendung war es ebenso möglich, zusätzliche Kenntnisse bezüglich der Verwendung zusätzlicher Peripherie zu erlangen.

- Aufbau einer Reader-Einheit mit und ohne Leistungsverstärker
  - Aufgrund des *DL*-Kanals war erstmals eine Reader-Einheit für den gewählten Frequenzbereich erforderlich. Dabei konnte ein kommerzieller Reader-IC, wahlweise in Kombination mit und ohne einen zusätzlichen Leistungsverstärker für einen vergrößerten Wirkungsbereich, erprobt werden.
- Test des Gesamtverfahrens
  - Es konnten erfolgreiche Tests des entwickelten Gesamtverfahrens auch unter suboptimalen Umgebungsbedingungen (siehe Abschnitt 4.4.3) durchgeführt werden. Diese erlauben einen optimistischen Blick auf eine positive Weiterführung dieser Entwicklung im Rahmen des Forschungsvorhabens BATSEN.

5 Fazit

Abschließend gilt zu resümieren, inwieweit die Aufgabenstellung der Masterarbeit erfüllt worden ist. Anhand der erzielten Ergebnisse in dieser Masterarbeit erfolgt daraufhin eine Einschätzung der Relevanz für das Gesamtprojekt.

5.1 Realisierung der Aufgabenstellung

Das primäre Ziel der Aufgabenstellung war die Entwicklung eines Zellensensors mit bidirektionaler drahtloser Kommunikation für den Einsatz in Fahrzeugbatterien. Dabei galt für die Konzeption die Rahmenbedingung, eine möglichst integrierbare Schaltung zwecks Kostenminimierung zu realisieren, die weiterhin zugleich den Anforderungen an die Zellen-Sensorik gerecht wird.

Die Aufgabenstellung der Masterarbeit konnte im vollem Umfang erfolgreich bearbeitet und umgesetzt werden. Es sind verschiedene Konzepte für die Realisierung des Zellensensors entwickelt und vorgestellt worden. Da keine entsprechende kommerziell verfügbaren Lösung recherchiert werden konnte, bedurfte es für den *HF*-Empfänger des *DL*-Kanals zum Zellensensor einer Eigenentwicklung. Dabei dienten energieeffiziente und minimalistische Empfängerkonzepte aus dem *RFID*-Bereich als Grundlage. Die entwickelte *Wake-Up*-Funktionalität des Empfängers ermöglicht einen nahezu passiven Betrieb des deaktivierten Zellensensors. Ungeachtet der Abschaltung der Betriebsspannung des Zellensensors ist dieser dennoch jederzeit empfangsbereit und kann innerhalb kurzer Zeit auf ein Kommando oder eine Message der Basisstation reagieren. Vergleicht man den Strombedarf zwischen dem aktiven und dem nahezu passiven Betrieb des Zellensensors, ergibt sich dabei eine Einsparung[1] um einen Faktor größer 170.
Vergleichbare Systeme versetzten häufig nur die *MCU* in einen Ruhezustand, sowie deaktivieren eventuell den Empfänger für den *DL*-Kanal. Für eine Überprüfung des *DL*-Kanals auf eine aktive Übertragungen, wird dieser periodisch aktiviert. Neben des erhöhten Energiebedarfs ist die Eigenschaft nicht unmittelbar auf eine Nachrichtenübertragung reagieren zu können, für die Anwendung innerhalb der Zellen-Sensorik nachteilig.

[1] bei Verwendung der Parameter aus dem Abschnitt 4.4.1

Die Zielsetzung einen Zellensensor ohne Quarzoszillator zu konzipieren, konnte durch den verwendeten *UHF*-Transmitter, welcher über einen internen LC-Oszillator verfügt, eingehalten werden.

Die für den Zellensensor ausgewählte *MCU* ist bewusst leistungsstärker dimensioniert als für die momentane Anwendung erforderlich. Somit besteht für die Erprobung von softwareseitigen Funktionalitäten eine ausreichende Leistungsreserve. Zusammen mit der Aufspaltung der Hardware auf zwei getrennte Platinen ist so ein flexibles und „erprobungsfreundliches“ System realisiert worden, dass ebenso an die Verwendung anderer Frequenzbereiche anpassbar ist.
Die bestehende Datenlogger-Einheit ist weiter modifiziert und um eine Reader-Einheit für die Realisierung des *DL*-Kanals zu den Zellensensoren ergänzt worden.

Für die *MCU* des Zellensensors ist der Entwurf einer, im Vergleich zu den bisherigen Zellensensoren, vollkommen abweichenden Softwarestruktur erforderlich gewesen. Dies begründet sich einerseits durch die erstmalige Verwendung des *DL*-Kanals und andererseits durch den verwendeten Transmitter für den *UL*-Kanal. Bei der Implementierung ist bewusst auf eine möglichst modulare Programmierung geachtet worden, um eine einfache Anpassung und Ergänzung von Funktionalitäten zu ermöglichen.

Die Software für die *MCU* der Datenlogger-Einheit konnte hingegen zu Teilen aus Vorarbeiten übernommen werden, es bedurfte jedoch umfassender Anpassungen und Ergänzungen aufgrund der deutlich abweichenden Betriebsart. Die Koordination der Zellensensoren, welche bisher nicht erforderlich war, musste zunächst entworfen und anschließend möglichst modular implementiert werden. Des Weiteren ist eine Schnittstelle für die Ausgabe der Messdaten, an eine übergeordnete Anwendung wie z. B. Matlab, hinzugefügt worden.

Darüber hinaus galt es für die Aufnahme von Messreihen ein Matlab-Skript zu erstellen, welches die über die serielle Schnittstelle ausgegebenen Messdaten aufnimmt, aufbereitet, darstellt und archiviert. Zugleich ermöglicht dieses Skript die Steuerung der Datenlogger-Einheit, verbunden mit der Konfiguration der Zellensensoren. Ebenso ist dieses Skript, aus einer Vielzahl von Funktionen bestehend, flexibel und modular aufgebaut.

Eine Erprobung des Gesamtverfahrens konnte wie beschrieben durchgeführt werden. Dabei waren zugleich die Vorteile der durch den *DL*-Kanal gesteuerten Zellen-Sensorik zu veranschaulichen. Erstmals ist nun beispielsweise eine synchronisierte Messwerterfassung möglich, ebenso wie die Verwendung eines synchronisierten Zeitschlitzverfahren für die *UL*-Kommunikation. Die neu erzielten Funktionalitäten des Zellensensors rechtfertigen den zusätzlichen Hardwareaufwand für den *DL*-Kanal. So kann der *UL*-Kanal wesentlich effizienter genutzt werden, was eine

höhere Übertragungsrate mit geringerem *FER* zur Folge hat. Der bereits erstellte Satz an *BMCL*-Kommandos bzw. Messages verdeutlicht zudem die neu hinzugewonnenen Funktionalitäten der Zellen-Sensorik.

5.2 Relevanz der Ergebnisse für das Gesamtprojekt

Da nun erstmals ein *DL*-Kanal für die Kommunikation zu den Zellensensoren verfügbar ist, konnten im Vergleich zu den bisherigen Sensoren, neue Betriebsarten aufgezeigt und erprobt werden.

Eine Anpassung der synchronisierten Messwerterfassung des Zellensensors an die verschiedenen Betriebszustände einer Fahrzeugbatterie ist nun möglich. Abhängig von dem Betriebszustand können spezifische Funktionalitäten ausgeführt werden, was einer Zustandsumschaltung der Zellensensoren entspricht. Es konnten somit einige wesentliche Ziele des Projektes „BATSEN“ getroffen werden:

- synchronisierte Messwerterfassung
- synchronisierte Informationsübertragung über den *UL*-Kanal
- Zustandsumschaltung der Zellensensoren
- Möglichkeit zur Ladungsbalancierung

Die Grundstruktur der Software und insbesondere die Hardware der Zellen-Sensorik ist bereits als Labormuster realisiert und funktionsfähig.

Neben dem realisierten Entwurf, erfolgte in dem Abschnitt 3.2 die Diskussion weiterer Konzepte für den möglichen Aufbau eines Zellensensors. Weiterhin ist in dem Kapitel 2 eine Analyse von möglichen passiven und aktiven Transceiver-Konzepten mit anschließender Beurteilung von kommerziell verfügbaren Lösungen in diesem Bereichen vorgenommen worden. Die hierbei erzielten Erkenntnisse sind ebenso relevant für das Gesamtprojekt, da sie neben der realisierten Lösung die Vor- und Nachteile anderer Lösungswege aufzeigen.

In dem vorherigen Abschnitt 4.5.2 sind einige Optimierungsmöglichkeiten für das System der Zellen-Sensorik genannt worden. Zusammen mit den in dieser Arbeit gewonnenen Erkenntnissen, beschrieben im Abschnitt 4.5.3, ließe sich das bestehende System der Zellen-Sensorik weiter verbessern.
Insgesamt ist somit ein Beitrag für die erfolgreiche Weiterführung des Forschungsvorhabens geleistet worden. Bereits jetzt ist eine erweiterbare Plattform realisiert, die Erprobung von noch zu implementierenden Batteriemodellen ermöglicht.

5.3 Generelle Erkenntnisse

Mit der realisierten Zellen-Sensorik und den dabei erlangten Erkenntnissen konnte ein kleiner Beitrag zu dem aktuellen Thema der Batterietechnik in Fahrzeugen geleistet werden. Durch die Überwachung der einzelnen Batteriezellen ist es möglich, die Betriebssicherheit zu erhöhen und gleichzeitig eine Verfügbarkeitsprognose vorzunehmen. Hierdurch wäre somit eine effizientere und schonendere Ausnutzung der vorhandenen Energieressource, beispielsweise zur Erhöhung der Betriebsdauer, denkbar.

Eine Übertragung von den in dieser Arbeit erlangten Erkenntnissen, ist auch auf andere Themengebiete denkbar. Dazu zählen beispielsweise der Bereich der aktiven *RFID*-Transponder, genauso spezielle Sensornetze. Auch bei diesen ist ein effizienter Umgang mit der zur Verfügung stehenden Energieressource erforderlich.

Es zeigte sich, dass die Verwendung von allgemeinen Lösungen für eine nicht allgemeingültige Problemstellung, unvorteilhaft sein kann. Vielmehr kann mit der spezialisierten Auslegung eines Systems, was die ingenieurmäßige Herangehensweise auszeichnet, eine optimale Lösung für eine spezielle Problemstellung erreicht werden. Jedoch gilt es dabei, stets einen Kompromiss zwischen dem erforderlichem Aufwand und damit erzieltem Nutzen zu finden.

Literaturverzeichnis

[Ansari u. a. 2008] ANSARI, Junaid ; PANKIN, Dmitry ; MÄHÖNEN, Petri: *Radio-Triggered Wake-ups with Addressing Capabilities for Extremely Low Power Sensor Network Applications*. RWTH Aachen, 2008

[Finkenzeller 2008] FINKENZELLER, Klaus: *RFID-Handbuch: Grundlagen und praktische Anwendungen von Transpondern, kontaktlosen Chipkarten und NFC*. 5. Aufl. München : Hanser Fachbuchverlag, 2008. – ISBN 978–3–446–41200–2

[Gebhart 2008] GEBHART, Michael: *RFID Systems - HF-Transponder-Technologie*. Vorlesungsskript TU Graz, 2008

[Hayward u. a. 2003] HAYWARD, Wes ; CAMPBELL, Rick ; LARKIN, Bob: *Experimental Methods in RF Design*. 1. Aufl. USA : American Radio Relay League, 2003. – ISBN 0–87259–879–9

[IDS Microchip 2010] IDS MICROCHIP: *R14AB ISO 14443 RFID Reader IC*. Version: März 2010. `http://www.ids-microchip.com/doc/pf/R14AB_PF.pdf`, Abruf: 10.10.2011

[Ilgin 2011] ILGIN, Sergej: *Drahtlose Sensoren für Batteriemodule - Konzeption und Kalibrierung*. Hamburg, Hochschule für Angewandte Wissenschaften, Bachelorarbeit, August 2011

[Jegenhorst 2009] JEGENHORST, Niels: *Entwicklung eines Controllersystems zur Zustandserkennung von ABS-Sensoren*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, Oktober 2009

[Krannich 2008] KRANNICH, Tobias: *Experimentalsystem für einen Sensor-Controller mit drahtloser Energie- und Datenübertragung*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, Juli 2008

[Krischke 2001] KRISCHKE, Alois: *Rothammels Antennenbuch*. 12. Aufl. Baunatal : DARC Verlag, 2001. – ISBN 3–88692–033–X

[Kube 2011] KUBE, Raik: *Drahtloses Sensornetzwerk für Fahrzeugbatterien - Kanal, Antennen und Fehlerraten*. Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, Dezember 2011

[Kuchling 2004] KUCHLING, Horst: *Taschenbuch der Physik*. 18. Aufl. Leipzig : Fachbuchverlag, 2004. – ISBN 3–446–22883–7

[Linden u. Reddy 2010] LINDEN, David ; REDDY, Thomas: *Linden's Handbook of Batteries*. 4. Aufl. Mcgraw-Hill, 2010. – ISBN 978–0071624213

[Lindner u. a. 2005] LINDNER, H. ; BRAUER, H. ; LEHMANN, C.: *Taschenbuch der Elektrotechnik und Elektronik*. 8. Aufl. Leipzig : Fachbuchverlag, 2005. – ISBN 978–3446210561

[Microchip 2003] MICROCHIP: *Antenna Circuit Design for RFID Applications (AN710)*. Version: Januar 2003. `http://ww1.microchip.com/downloads/en/AppNotes/00710c.pdf`, Abruf: 15.06.2011

[Microchip 2004] MICROCHIP: *Loop Antenna Basics and Regulatory Compliance for Short-Range Radio*. Version: Oktober 2004. `http://ww1.microchip.com/downloads/en/DeviceDoc/RFA1%20parts%205%206%20b.pdf`, Abruf: 13.09.2011

[Plaschke 2008] PLASCHKE, Stephan: *Experimentalsystem für drahtlose Batteriesensorik*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, Juli 2008

[Püttjer 2011] PÜTTJER, Simon: *Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellensensoren*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, Juni 2011

[Reif 2010] REIF, Konrad: *Batterien, Bordnetze und Vernetzung*. 1. Aufl. Wiesbaden : Vieweg + Teubner, 2010. – ISBN 978–3–8348–1310–7

[Riemschneider u. Vollmer 2010] RIEMSCHNEIDER, K.-R. ; VOLLMER, J.: *Drahtlose Zellensensoren für Fahrzeugbatterien BATSEN - Förderantrag*. Hamburg : Hochschule für Angewandte Wissenschaften, 2010

[Seifart 2003] SEIFART, Manfred: *Analoge Schaltungen*. 6. Aufl. Berlin : Verlag Technik, 2003. – ISBN 3–341–01298–2

[Silicon Laboratories 2008] SILICON LABORATORIES: *Advances in Silicon Technology Enables Replacement of Quartz-Based Oscillators*. September 2008

[Silicon Laboratories 2010a] SILICON LABORATORIES: *Si4010 ANTENNA INTERFACE AND MATCHING NETWORK GUIDE (AN369)*. Version: 0.1, Juli 2010. `http://www.silabs.com/pages/DownloadDoc.aspx?FILEURL=Support%20Documents/TechnicalDocs/Si4010.pdf`, Abruf: 16.12.2010

[Silicon Laboratories 2010b] SILICON LABORATORIES: *Si4012 CRYSTAL-LESS FSK/OOK RF TRANSMITTER*. Version: Dezember 2010, Juli 2010. http://www.silabs.com/pages/DownloadDoc.aspx?FILEURL=Support%20Documents/TechnicalDocs/Si4012.pdf, Abruf: 15.06.2011

[Silicon Laboratories 2011] SILICON LABORATORIES: *Si500D DIFFERENTIAL OUTPUT SILICON OSCILLATOR*. Version: 1.0, Mai 2011. http://www.silabs.com/Support%20Documents/TechnicalDocs/Si500D.pdf, Abruf: 19.08.2011

[SiTime 2011] SITIME: *SiT8208 Precision Oscillator 1-80 MHz*. Version: 0.83, Mai 2011. http://www.sitime.com/products/datasheets/sit8208/SiT8208-datasheet.pdf, Abruf: 19.08.2011

[STMicroelectronics 2009] STMICROELECTRONICS: *How to design a 13.56 MHz customized tag antenna (AN2866)*. Version: 1, Januar 2009. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/APPLICATION_NOTE/CD00221490.pdf, Abruf: 12.09.2011

[STMicroelectronics 2010a] STMICROELECTRONICS: *M24LR64-R*. Version: 12, Dezember 2010. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00217247.pdf, Abruf: 15.06.2011

[STMicroelectronics 2010b] STMICROELECTRONICS: *M24LR64-R tool kit user guide*. Version: 1, Januar 2010. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/CD00254727.pdf, Abruf: 14.10.2011

[Texas Instruments 2002] TEXAS INSTRUMENTS: *MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller (SLAS368G)*. Version: März 2011, Oktober 2002. http://focus.ti.com/lit/ds/symlink/msp430f169.pdf, Abruf: 15.06.2011

[Texas Instruments 2004] TEXAS INSTRUMENTS: *MSP430x2xx Family User's Guide (SLAU144H)*. Version: April 2011, Dezember 2004. http://focus.ti.com.cn/cn/lit/ug/slau144h/slau144h.pdf, Abruf: 15.06.2011

[Texas Instruments 2006a] TEXAS INSTRUMENTS: *MSP430x1xx Family User's Guide (SLAU049F)*. Version: 2006. http://focus.ti.com/lit/ug/slau049f/slau049f.pdf, Abruf: 15.06.2011

[Texas Instruments 2006b] TEXAS INSTRUMENTS: *TRF7960 MULTI-STANDARD FULLY INTEGRATED 13.56-MHZ RFID ANALOG FRONT END AND DATA-FRAMING READER SYSTEM (SLOU186F)*. Version: August 2010, August 2006. http://focus.ti.com/lit/ds/symlink/trf7960.pdf, Abruf: 15.06.2011

[Texas Instruments 2007a] TEXAS INSTRUMENTS: *MSP430F23x MSP430F24x(1) MSP430F2410 Mixed Signal Microcontroller (SLAS547F)*. Version: April 2011, Juni 2007. http://focus.ti.com/lit/ds/symlink/msp430f235.pdf, Abruf: 15.06.2011

[Texas Instruments 2007b] TEXAS INSTRUMENTS: *TMP102 - Low Power Digital Temperature Sensor With SMBus/Two-Wire Serial Interface in SOT563 (SBOS397B)*. Version: Oktober 2008, August 2007. http://www.ti.com/lit/ds/symlink/tmp102.pdf, Abruf: 09.09.2011

[Texas Instruments 2007c] TEXAS INSTRUMENTS: *TPS61201 - LOW INPUT VOLTAGE SYNCHRONOUS BOOST CONVERTER WITH 1.3-A SWITCHES (SLVS577B)*. Version: Februar 2008, März 2007. http://www.ti.com/lit/ds/symlink/tps61200.pdf, Abruf: 09.09.2011

[Texas Instruments 2008] TEXAS INSTRUMENTS: *HF Power Amplifier (Reference Design Guide)*. Version: Rev A, September 2008. http://www.ti.com/litv/zip/sloc132, Abruf: 14.09.2011

[Texas Instruments 2009a] TEXAS INSTRUMENTS: *eZ430-TMS37157 Development Tool - User's Guide (SLAU281B)*. Version: März 2010, November 2009. http://www.ti.com/lit/ug/slau281b/slau281b.pdf, Abruf: 25.08.2011

[Texas Instruments 2009b] TEXAS INSTRUMENTS: *TMS37157 PASSIVE LOW FREQUENCY INTERFACE DEVICE WITH EEPROM AND 134.2 kHz TRANSPONDER INTERFACE (SWRS083A)*. Version: November 2009, September 2009. http://focus.ti.com/lit/ds/symlink/tms37157.pdf, Abruf: 15.06.2011

[Tietze u. Schenk 2010] TIETZE, U. ; SCHENK, Ch.: *Halbleiter-Schaltungstechnik*. 13. Aufl. Berlin, Heidelberg : Springer, 2010. – ISBN 978–3–642–01621–9

A Diverses

Im Folgendem sind dargestellt:

- Tabelle A.1: Kommandos und Messages für die Steuerung der Zellensensoren, entsprechend der abstrahierenden Sprache BMCL
- Bild A.1: Abfolge der Kommandos bzw. Messages für die Steuerung der Zellensensoren im Standard-Messbetrieb
- Bild A.2 bis A.6: Gemessene Signalverläufe der Zellen-Sensorik

Name:	Kodierung (8-Bit) in hex:	Funktion:	Parameter (12-Bit):	Imple-mentiert:
ZS_CMD_STAY_ON	0x00	- Aufwecken des Zellensensors - Stoppen der Messungen	-	ja
ZS_CMD_TURN_OFF	0x01	Abschalten des Zellensensors	-	ja
ZS_CMD_BAL_ON	0x02	Aktivierung der Balancierung	-	ja
ZS_CMD_BAL_OFF	0x03	Deaktivierung der Balancierung	-	ja
ZS_CMD_SAMPLE_S1	0x04	Messreihe aufzunehmen: - Ucell - Usupply - T	-	ja
ZS_CMD_SAMPLE_S2	0x05	Messreihe aufzunehmen: - Ucell - Uact_in - Ulp	-	ja
ZS_CMD_SET_SAMPLE_TIME	0x06	- Setzen des Aufnahmezeitpunktes der Messreihe im Intervall - Übergang Normal-Mode	Wert für Zeitgeber	ja
ZS_CMD_SET_TX_TIME	0x07	- Setzen des Zeitpunktes für die Transmission der Messdaten im Intervall - Übergang Normal-Mode	Wert für Zeitgeber	ja
ZS_CMD_SET_INTVAL_TIME	0x08	- Setzen der Intervallzeit - Übergang Normal-Mode	Wert für Zeitgeber	ja
ZS_CMD_SYNCH	0x09	- Synchronisation - Start der Messung - Übergang Normal-Mode	-	ja
ZS_CMD_SCAN	0x0A	Transmission der Zellensensoradresse zu zufälligen Zeitpunkten	Anzahl der Wieder-holungen	ja
ZS_CMD_REPLY	0x0B	Transmission zu einem bestimmten Zeitpunkt von: - Zellensensoradresse - letzten beiden Kommandos inkl. Parameter	Wert für Zeitgeber	ja
ZS_CMD_SAVE_SET	0x0C	Speichern der Einstellung in den Flash des Zellensensors	-	ja
ZS_CMD_HC_SET_S_CNT	0x0D	- Setzen der Anzahl aufzunehmender Abtastwerte einer HC-Messreihe - Übergang HC-Mode	Anzahl der Abtastwerte	nein
ZS_CMD_HC_SET_S_INT	0x0E	- Setzen des Abtastabstands einer HC-Messreihe - Übergang HC-Mode	Wert für Zeitgeber	nein
ZS_CMD_HC_START	0x0F	Start der Aufnahme einer HC-Messreihe	-	nein
ZS_CMD_HC_TX	0x10	Transmission der Messdaten einer HC-Messreihe	-	nein
ZS_CMD_HC_END	0x11	Ende einer HC-Messreihe	-	nein
ZS_CMD_SET_GROUP	0x20	Setzen der Gruppenadresse	Adresse	nein

Tabelle A.1: *Kommandos und Messages für die Steuerung der Zellensensoren, entsprechend der abstrahierenden Sprache BMCL*

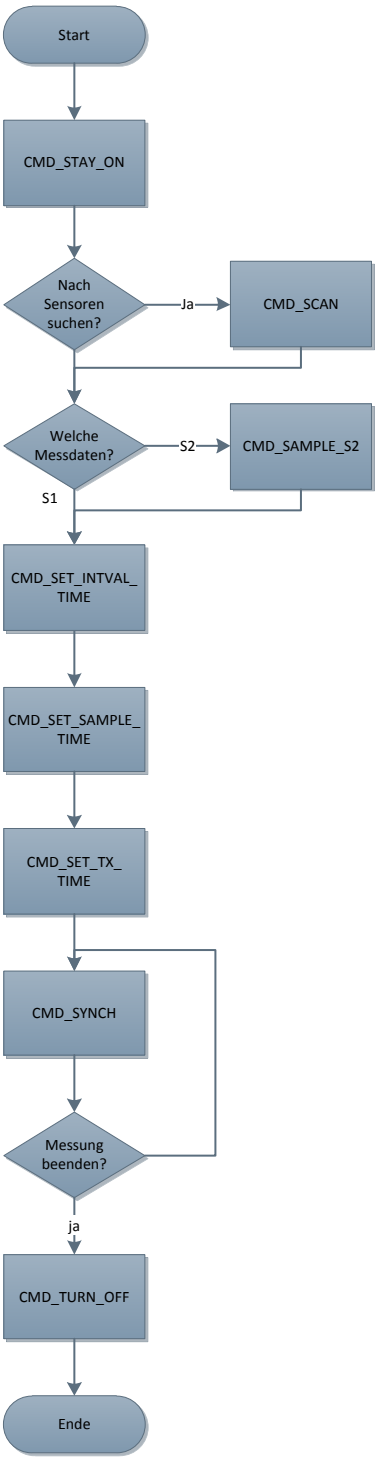


Bild A.1: *Abfolge der Kommandos bzw. Messages für die Steuerung der Zellensensoren im Standard-Messbetrieb*

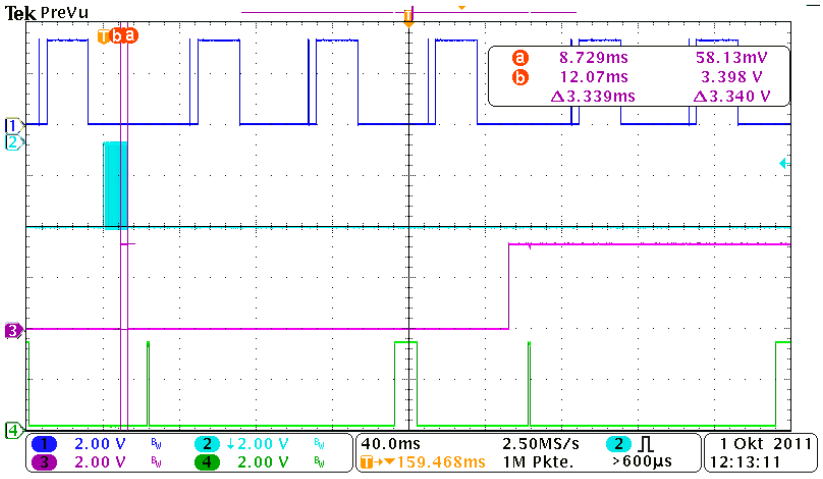


Bild A.2: *Gemessene Signalverläufe der Zellen-Sensorik, Intervalldauer $t_{iv} = 200\,\text{ms}$, $n = 3$ Zellensensoren:*
Differenz des Systemzeitgebers des dritten Zellensensors von $-3{,}34\,\text{ms}$ zur globalen Systemzeit
Kanal 1: Dekodierung des UL-Signals durch die Datenlogger-Einheit
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Intervalldauer des Systemzeitgebers vom Zellensensor (TP 10)
Kanal 4: Verweildauer des Zellensensors im Zustand „Sample“ und „Prepare TX“ (TP 11)

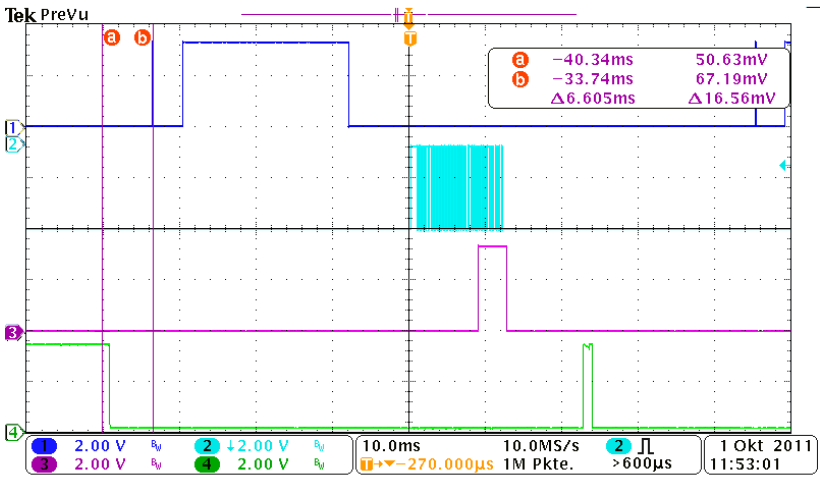


Bild A.3: *Gemessene Signalverläufe der Zellen-Sensorik, Intervalldauer $t_{iv} = 200\,\text{ms}$, $n = 3$ Zellensensoren:*
Abgleich des Oszillators sowie Antennenkreis vom UHF-Transmitter des dritten Zellensensor
Kanal 1: Dekodierung des UL-Signals durch die Datenlogger-Einheit
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Intervalldauer des Systemzeitgebers vom Zellensensor (TP 10)
Kanal 4: Verweildauer des Zellensensors im Zustand „Sample“ und „Prepare TX“ (TP 11)

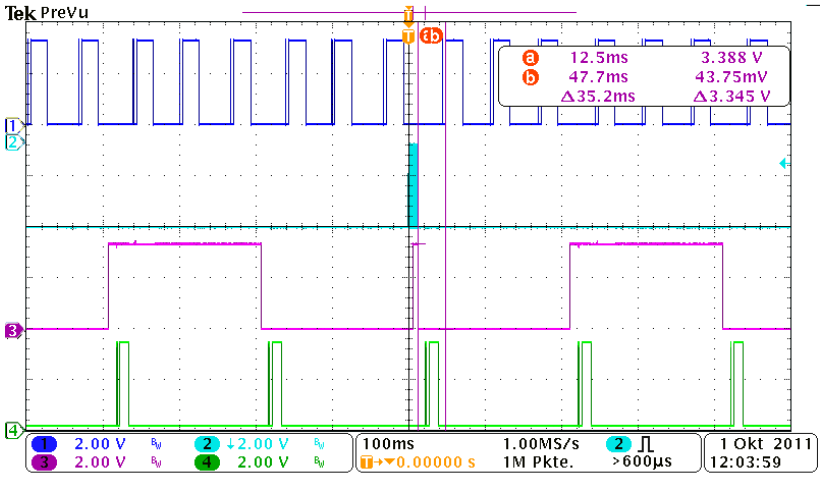


Bild A.4: *Gemessene Signalverläufe der Zellen-Sensorik, Intervalldauer $t_{iv} = 200\,\text{ms}$, $n = 3$ Zellensensoren:*
Zeitpunkt $t_{tx,zs1}$ für die Datentransmission im Intervall t_{iv} des ersten Zellensensor
Kanal 1: Dekodierung des UL-Signals durch die Datenlogger-Einheit
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Intervalldauer des Systemzeitgebers vom Zellensensor (TP 10)
Kanal 4: Verweildauer des Zellensensors im Zustand „Sample“ und „Prepare TX“ (TP 11)

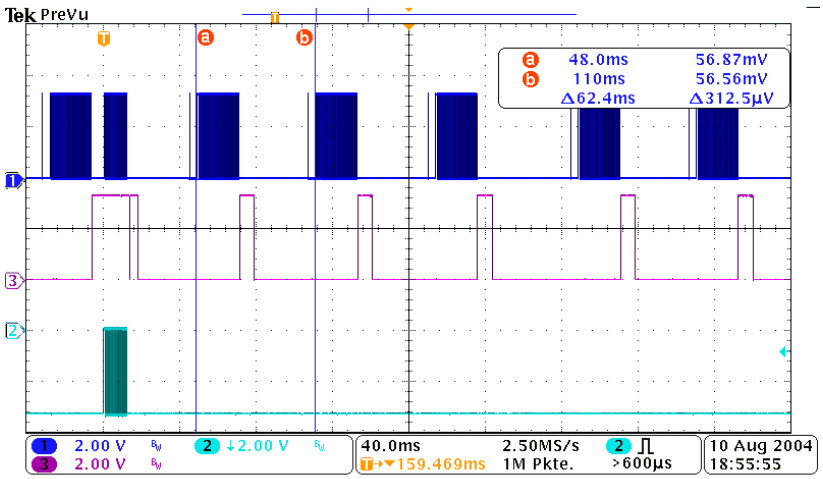


Bild A.5: *Gemessene Signalverläufe der Zellen-Sensorik, Intervalldauer $t_{iv} = 200\,\text{ms}$, $n = 3$ Zellensensoren:*
Zeitdauer t_{tx-tx} von 62,5 ms zwischen zwei Datentransmission im Intervall t_{iv}
Kanal 1: Demodulationssignal des UHF-Empfänger-Modul vom UL-Kanal
Kanal 2: Modulationssignal des DL-Kanals von der Datenlogger-Einheit
Kanal 3: Datenverarbeitung und Strommessung seitens der Datenlogger-Einheit

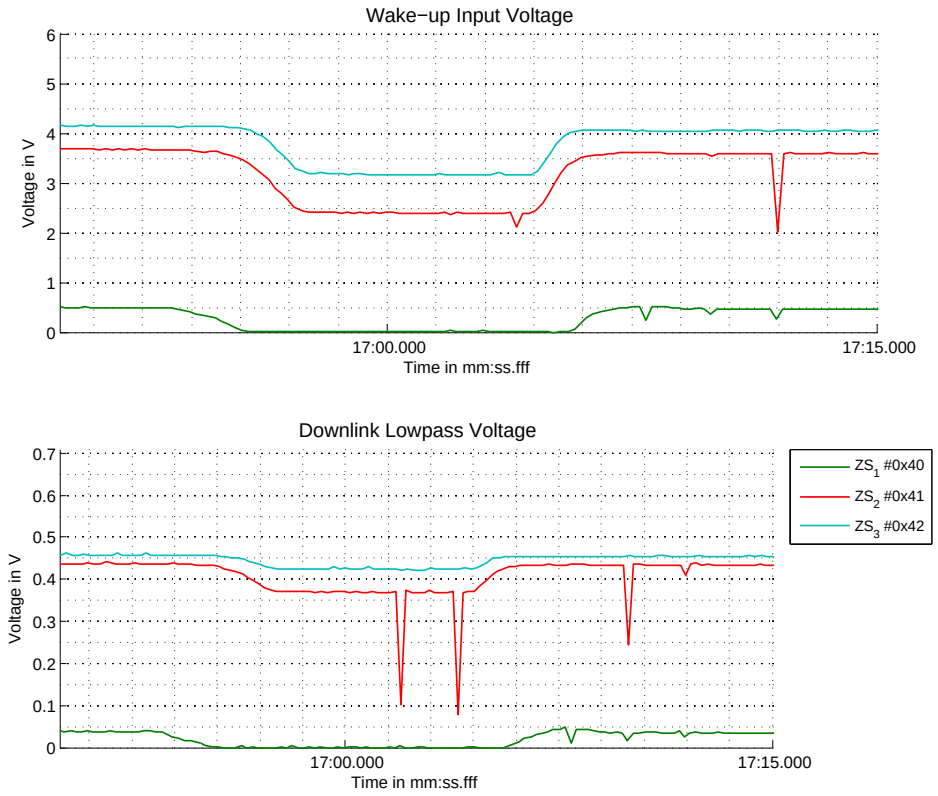
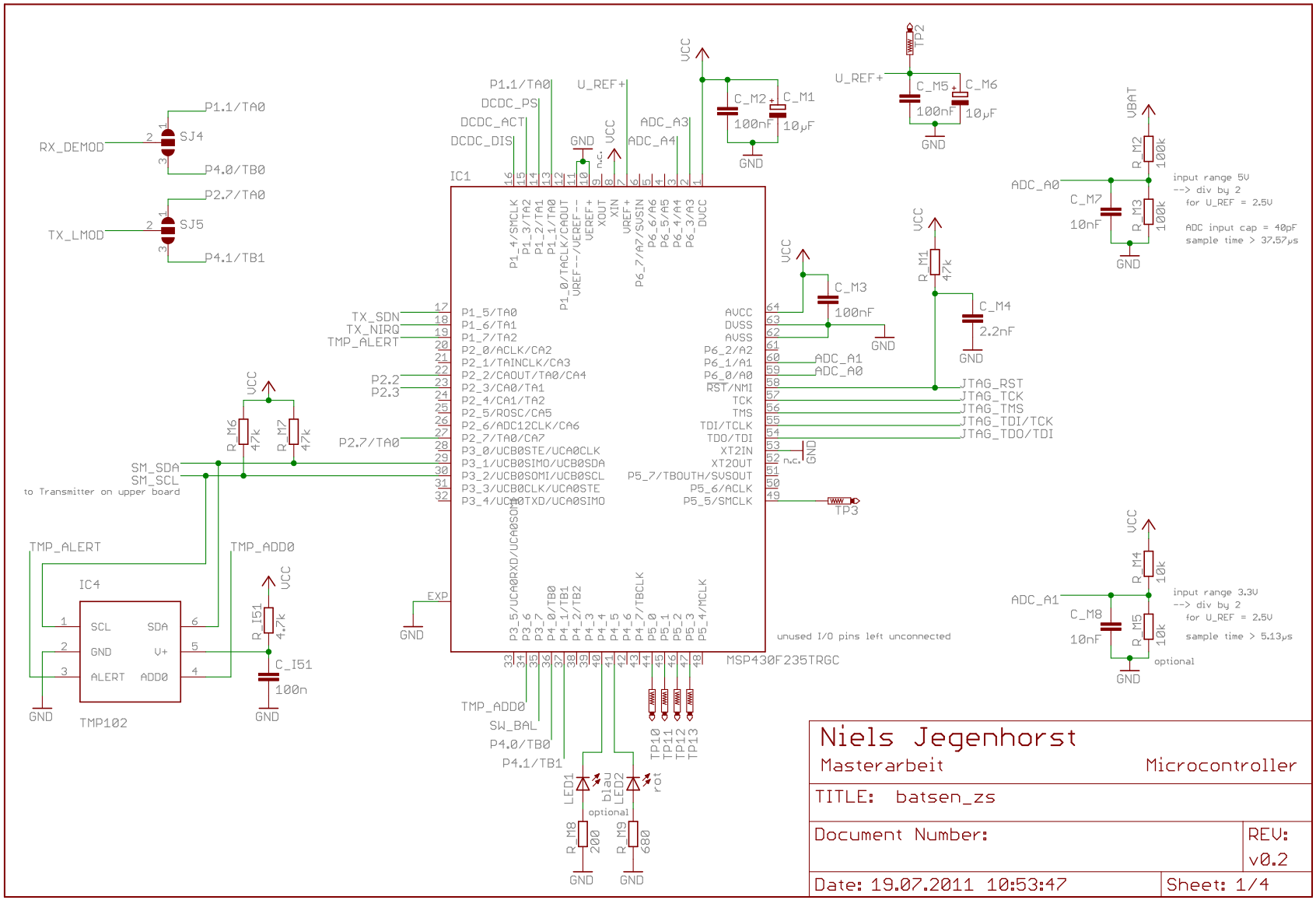


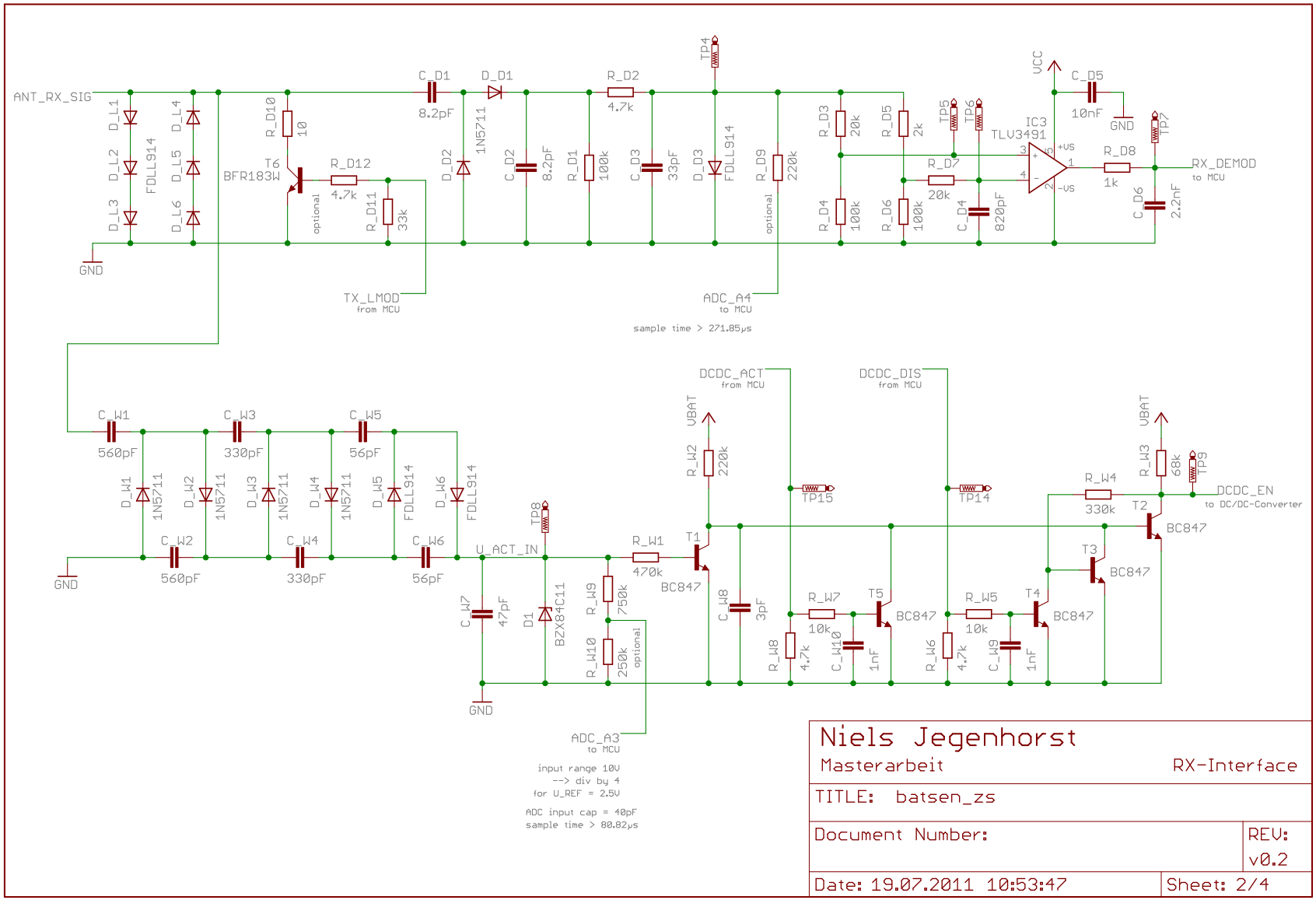
Bild A.6: *Beeinflussung des Downlink-Kanals durch den Batteriefachdeckel vom Gabelstapler, zunächst mit geöffnetem Deckel, dann diesen geschlossen und anschließend wieder geöffnet*

B Schaltpläne

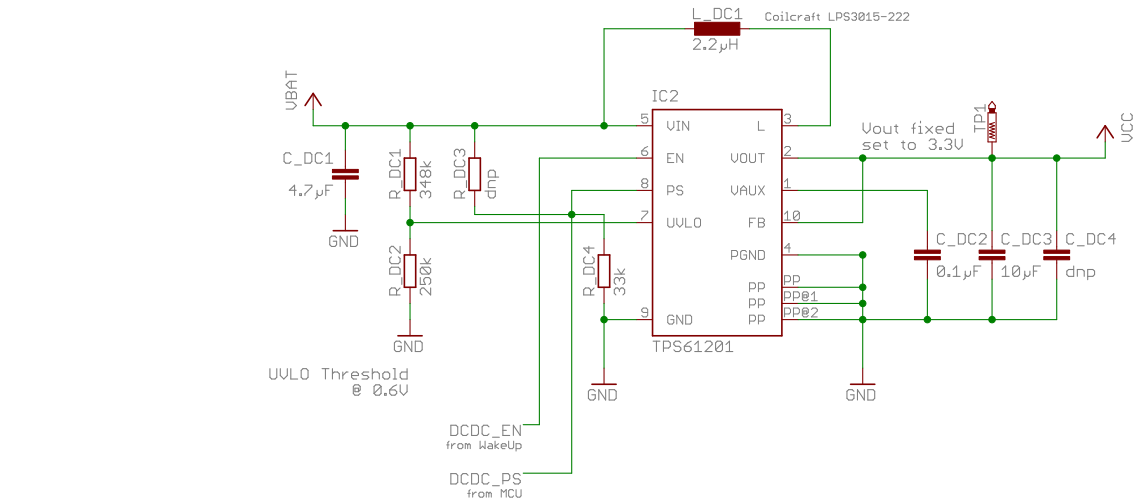
B.1 Zellensensor

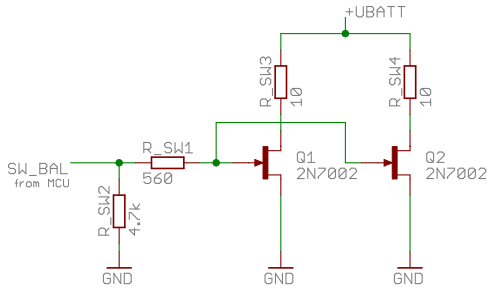
B.1.1 Schaltplan Basisplatine „BATSEN ZS v0.2“


IC1
MSP430F235TRGC
unused I/O pins left unconnected
P1.1/TA0
DCDC_PS
DCDC_ACT
DCDC_DIS
U_REF+
ADC_A3
ADC_A4
VCC
GND
C_M1
C_M2
100nF
10µF
TP2
C_M5
C_M6
RX_DEMOD
SJ4
P4.0/TB0
P2.7/TA0
TX_LMOD
SJ5
P4.1/TB1
TX_SDN
TX_NIRQ
TMP_ALERT
P2.2
P2.3
P1_5/TA0
P1_6/TA1
P1_7/TA2
P2_0/ACLK/CA2
P2_1/TAINCLK/CA3
P2_2/CAOUT/TA0/CA4
P2_3/CA0/TA1
P2_4/CA1/TA2
P2_5/ROSC/CA5
P2_6/ADC12CLK/CA6
P2_7/TA0/CA7
P3_0/UCB0STE/UCA0CLK
P3_1/UCB0SIMO/UCB0SDA
P3_2/UCB0SOMI/UCB0SCL
P3_3/UCB0CLK/UCA0STE
P3_4/UCA0TXD/UCA0SIMO
AVCC
DVSS
AVSS
P6_2/A2
P6_1/A1
P6_0/A0
RST/NMI
TCK
TMS
TDI/TCLK
TDO/TDI
XT2IN
XT2OUT
P5_7/TBOUTH/SVSOUT
P5_6/ACLK
P5_5/SMCLK
C_M3
100nF
C_M4
2.2nF
R_M1
47k
JTAG_RST
JTAG_TCK
JTAG_TMS
JTAG_TDI/TCK
JTAG_TDO/TDI
ADC_A0
ADC_A1
VBAT
R_M2
100k
R_M3
100k
C_M7
10nF
input range 5V
--> div by 2
for U_REF = 2.5V
ADC input cap = 40pF
sample time > 37.57µs
R_M4
10k
R_M5
10k
C_M8
10nF
input range 3.3V
--> div by 2
for U_REF = 2.5V
sample time > 5.13µs
optional
SM_SDA
SM_SCL
to Transmitter on upper board
R_M6
47k
R_M7
47k
TMP_ADD0
IC4
TMP102
SCL
SDA
GND
V+
ALERT
ADD0
R_I51
4.7k
C_I51
100n
SW_BAL
TP3
TP10
TP11
TP12
TP13
LED1
blau
LED2
rot
optional
R_M8
200
R_M9
680
EXP
Niels Jegenhorst
Masterarbeit
Microcontroller
TITLE: batsen_zs
Document Number:
REV: v0.2
Date: 19.07.2011 10:53:47
Sheet: 1/4

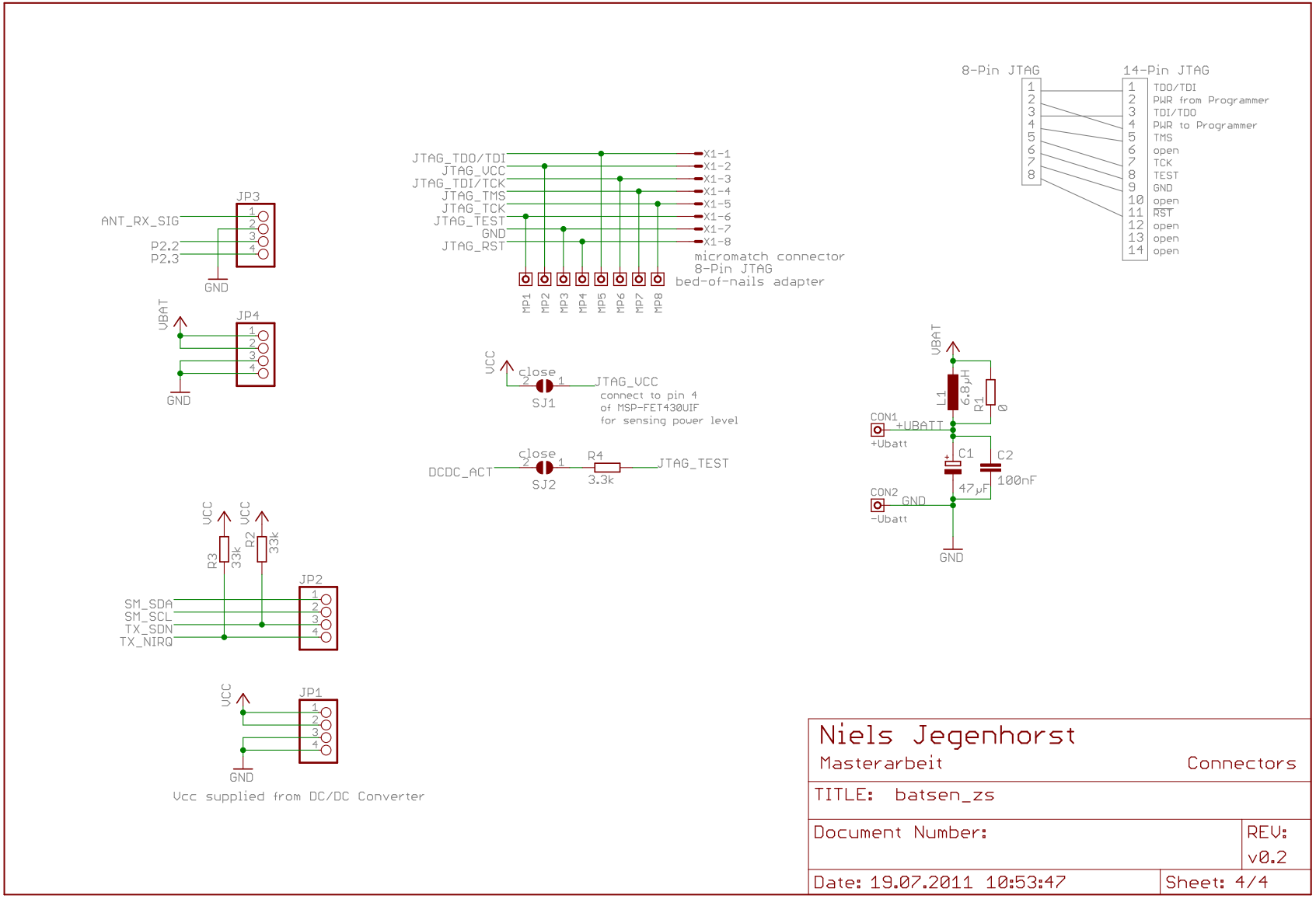


Niels Jegenhorst	
Masterarbeit	RX-Interface
TITLE: batsen_zs	
Document Number:	REV: v0.2
Date: 19.07.2011 10:53:47	Sheet: 2/4



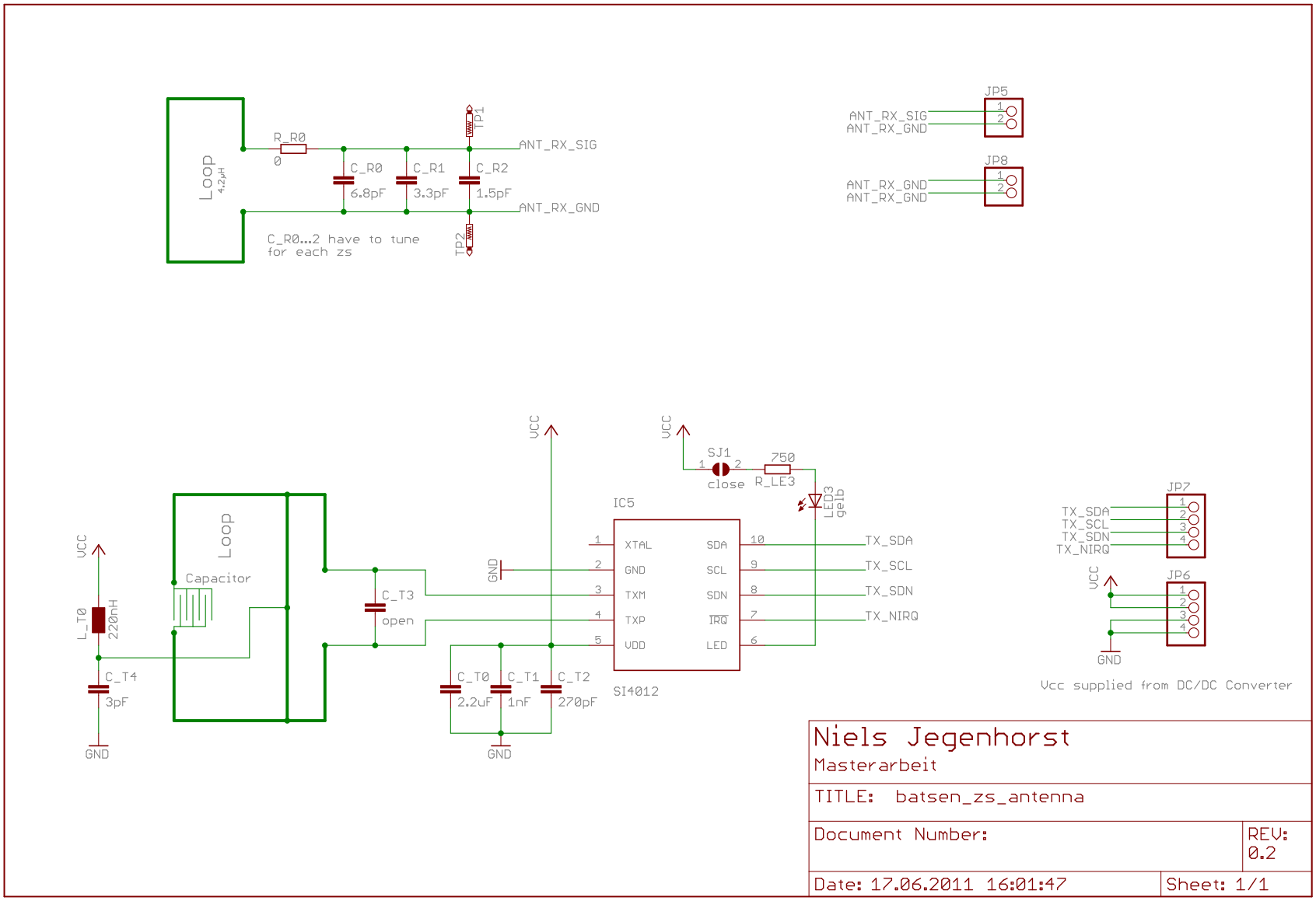


Niels Jegenhorst	
Masterarbeit	Power Supply
TITLE: batsen_zs	
Document Number:	REV: v0.2
Date: 19.07.2011 10:53:47	Sheet: 3/4



Niels Jegenhorst	
Masterarbeit	Connectors
TITLE: batsen_zs	
Document Number:	REV: v0.2
Date: 19.07.2011 10:53:47	Sheet: 4/4

B.1.2 Schaltplan Antennenplatine „BATSEN ZS Antenna v0.2“


Loop
4.2µH
R_R0
0
C_R0
6.8pF
C_R1
3.3pF
C_R2
1.5pF
TP1
TP2
ANT_RX_SIG
ANT_RX_GND
C_R0...2 have to tune
for each zs
JP5
ANT_RX_SIG
ANT_RX_GND
JP8
ANT_RX_GND
ANT_RX_GND
VCC
SJ1
close
750
R_LE3
LED3
gelb
IC5
XTAL
GND
TXM
TXP
VDD
SDA
SCL
SDN
IRQ
LED
TX_SDA
TX_SCL
TX_SDN
TX_NIRQ
SI4012
C_T0
2.2uF
C_T1
1nF
C_T2
270pF
C_T3
open
Loop
Capacitor
L_T0
220nH
C_T4
3pF
JP7
JP6
Vcc supplied from DC/DC Converter
Niels Jegenhorst
Masterarbeit
TITLE: batsen_zs_antenna
Document Number:
REV: 0.2
Date: 17.06.2011 16:01:47
Sheet: 1/1

B.1.3 Layout „BATSEN BS0406“

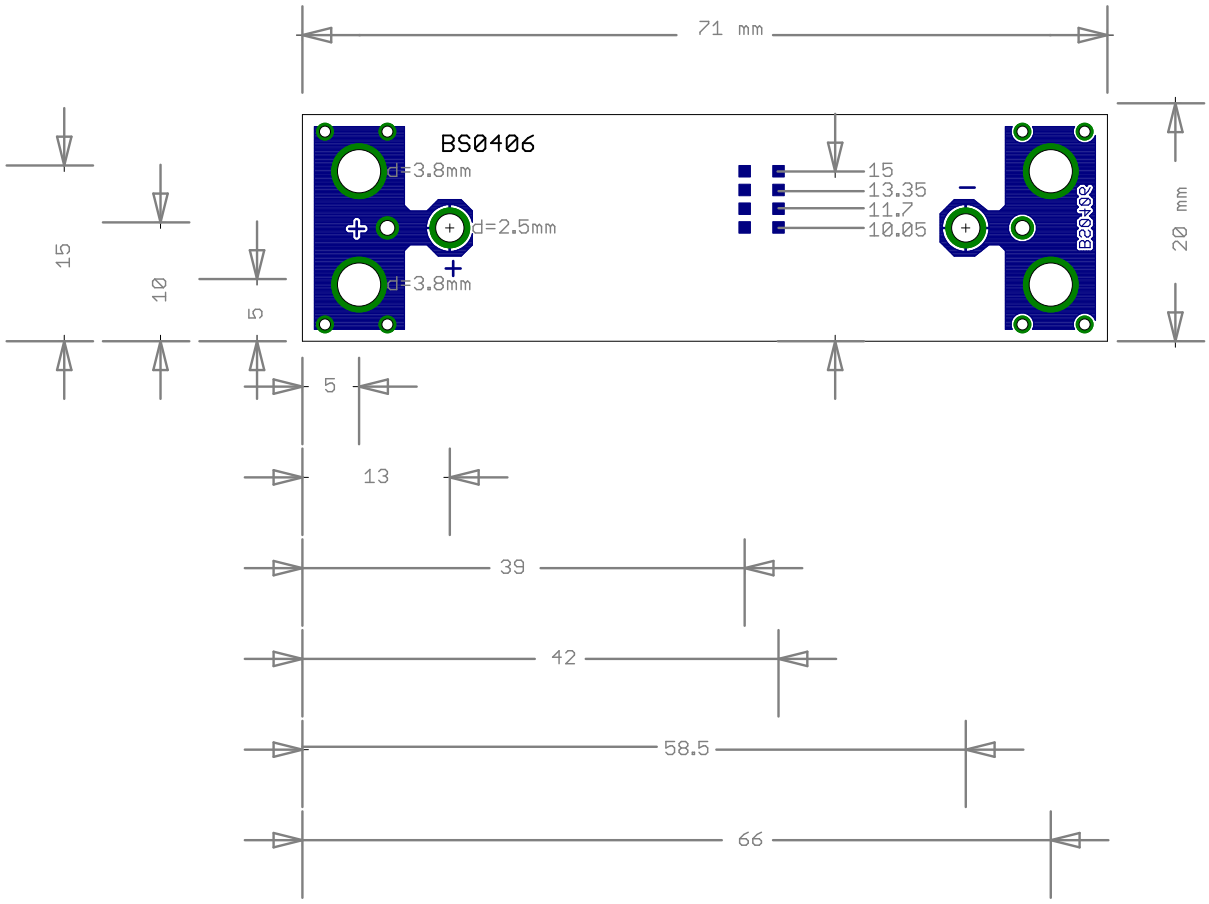
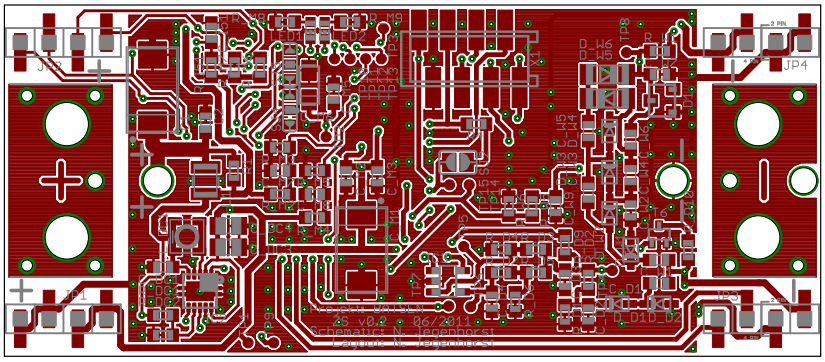
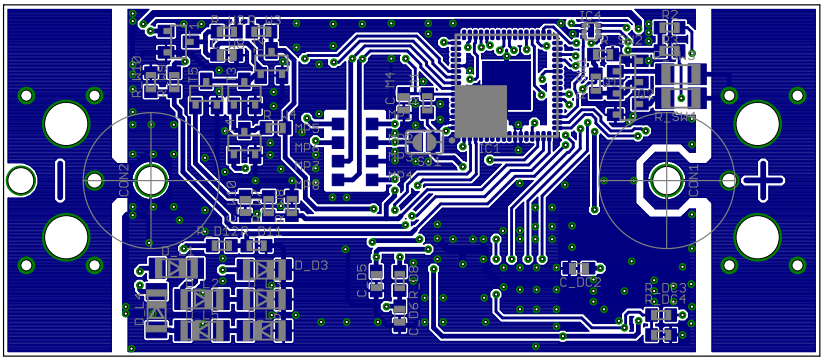


Bild B.1: *Bemaßung des bestehenden Zellensensors „BS0406“ der Klasse 1* ($s = 1.5$) *entnommen aus dem bestehendem Layout*

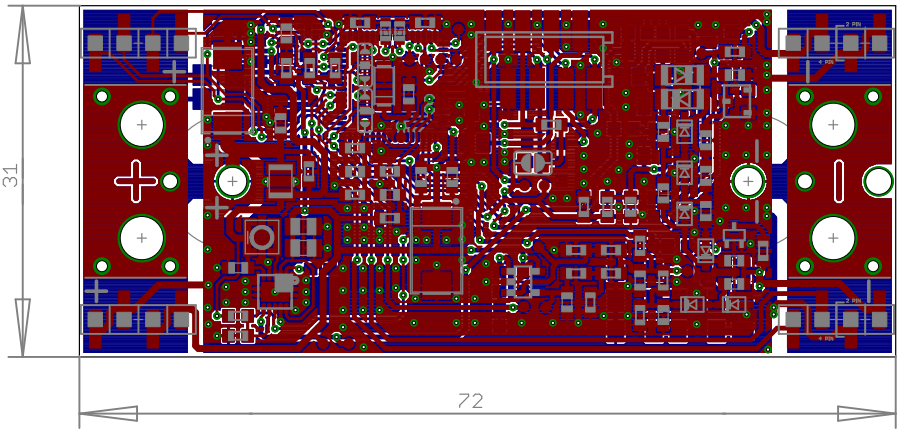
B.1.4 Layout Basisplatine „BATSEN ZS v0.2“



(a) Oberseite



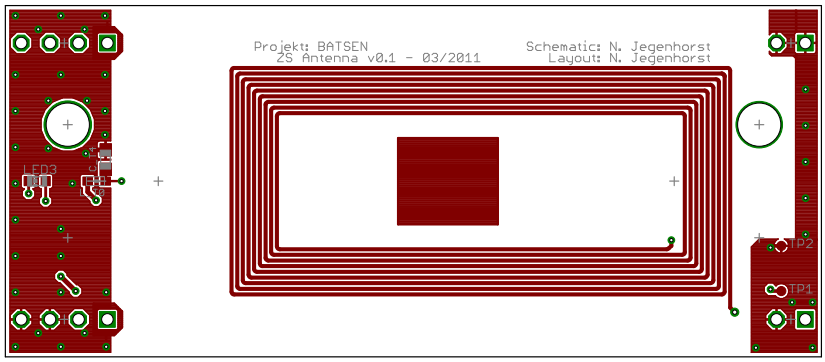
(b) Unterseite



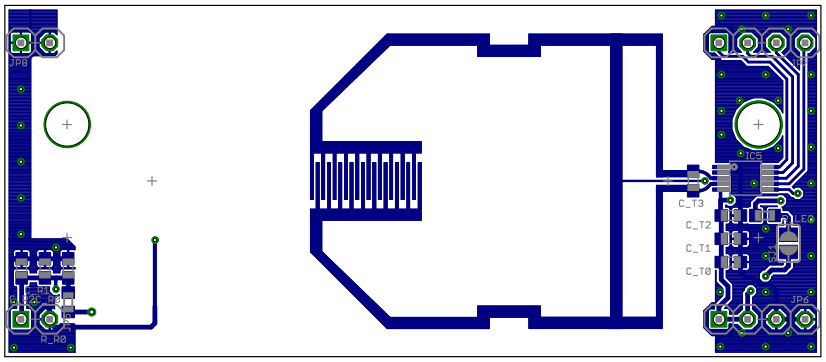
(c) Ober- und Unterseite

Bild B.2: *Layout „BATSEN ZS v0.2“* ($s = 1.5$)

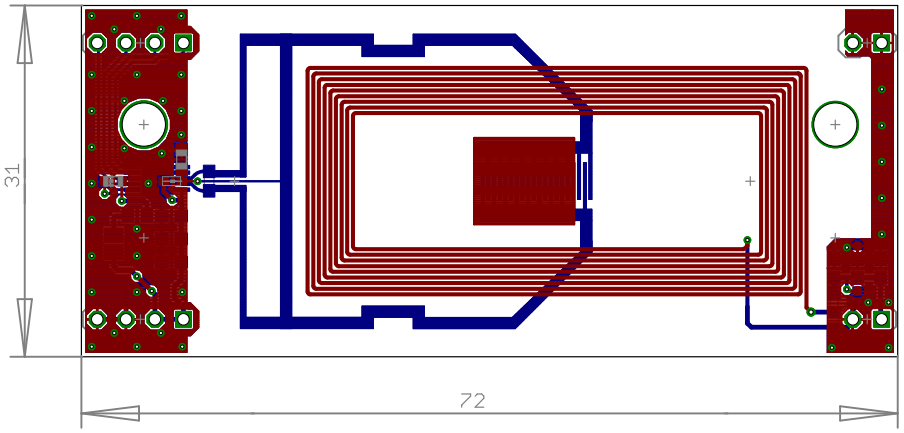
B.1.5 Layout Antennenplatine „BATSEN ZS Antenna v0.1“



(a) Oberseite



(b) Unterseite

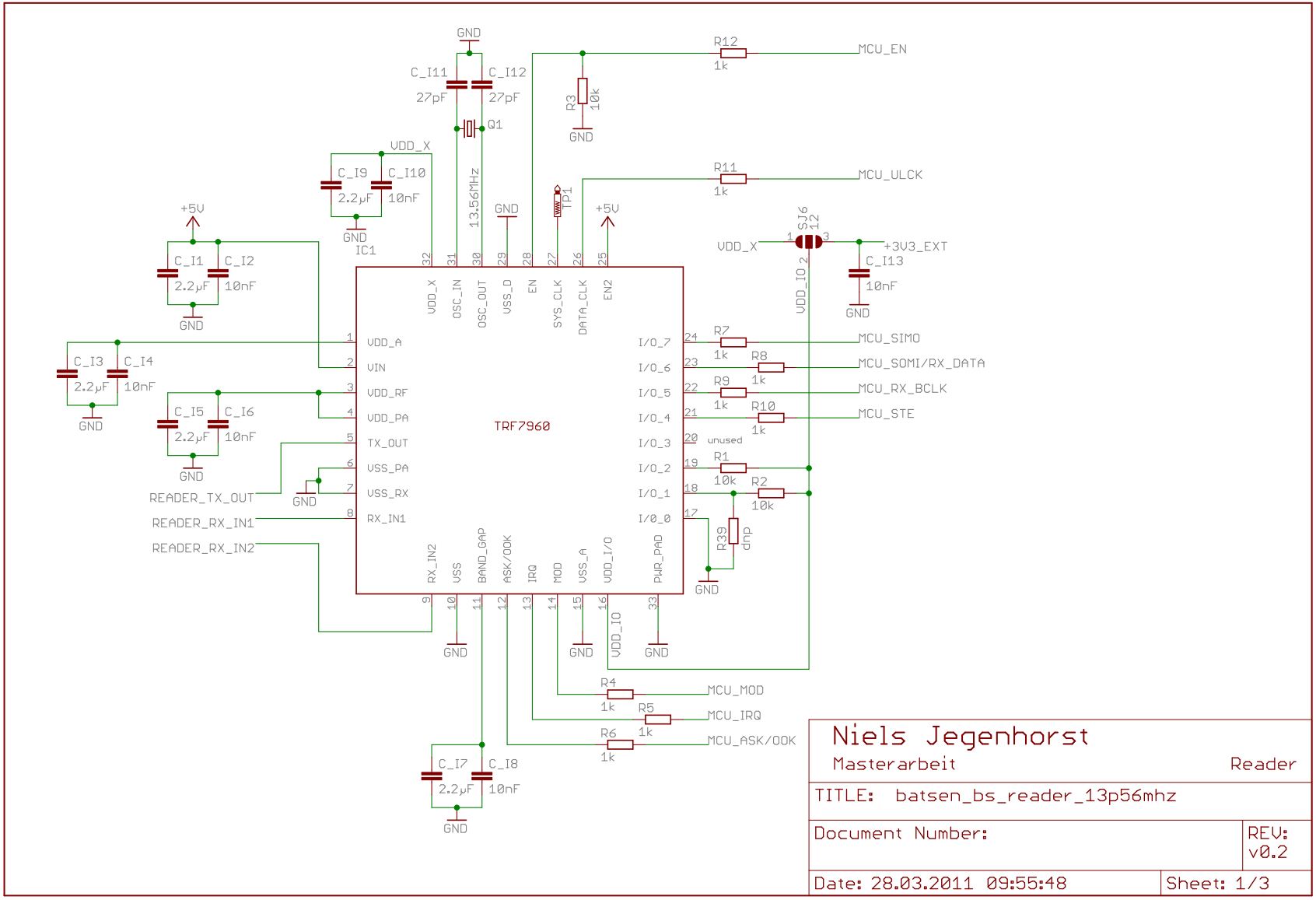


(c) Ober- und Unterseite

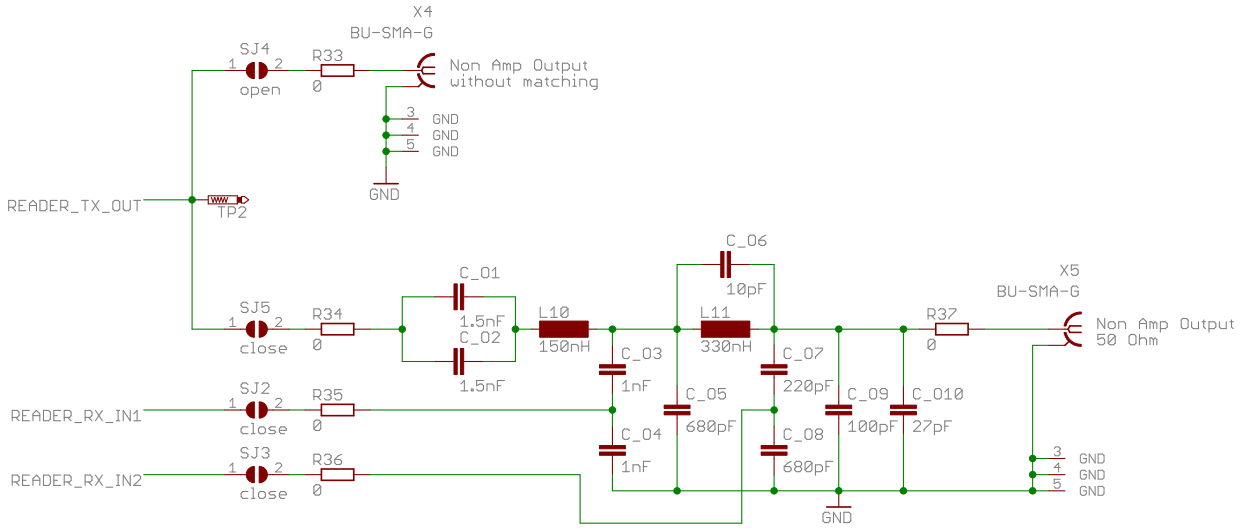
Bild B.3: *Layout „BATSEN ZS Antenna v0.1“* ($s = 1.5$)*, entspricht ebenfalls dem aktuellen Layout von „BATSEN ZS Antenna v0.2“*

B.2 Basisstation

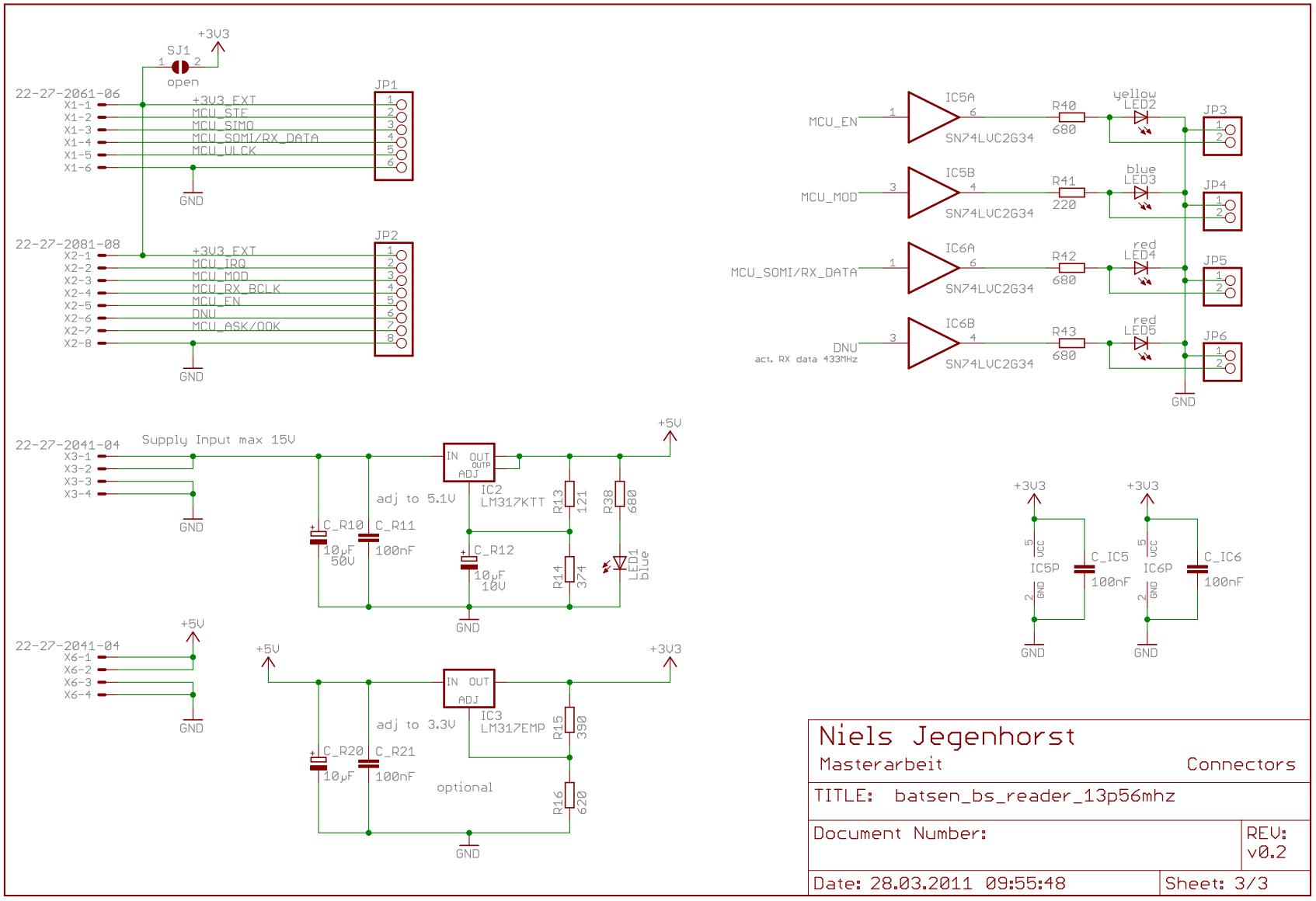
B.2.1 Schaltplan Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.2“



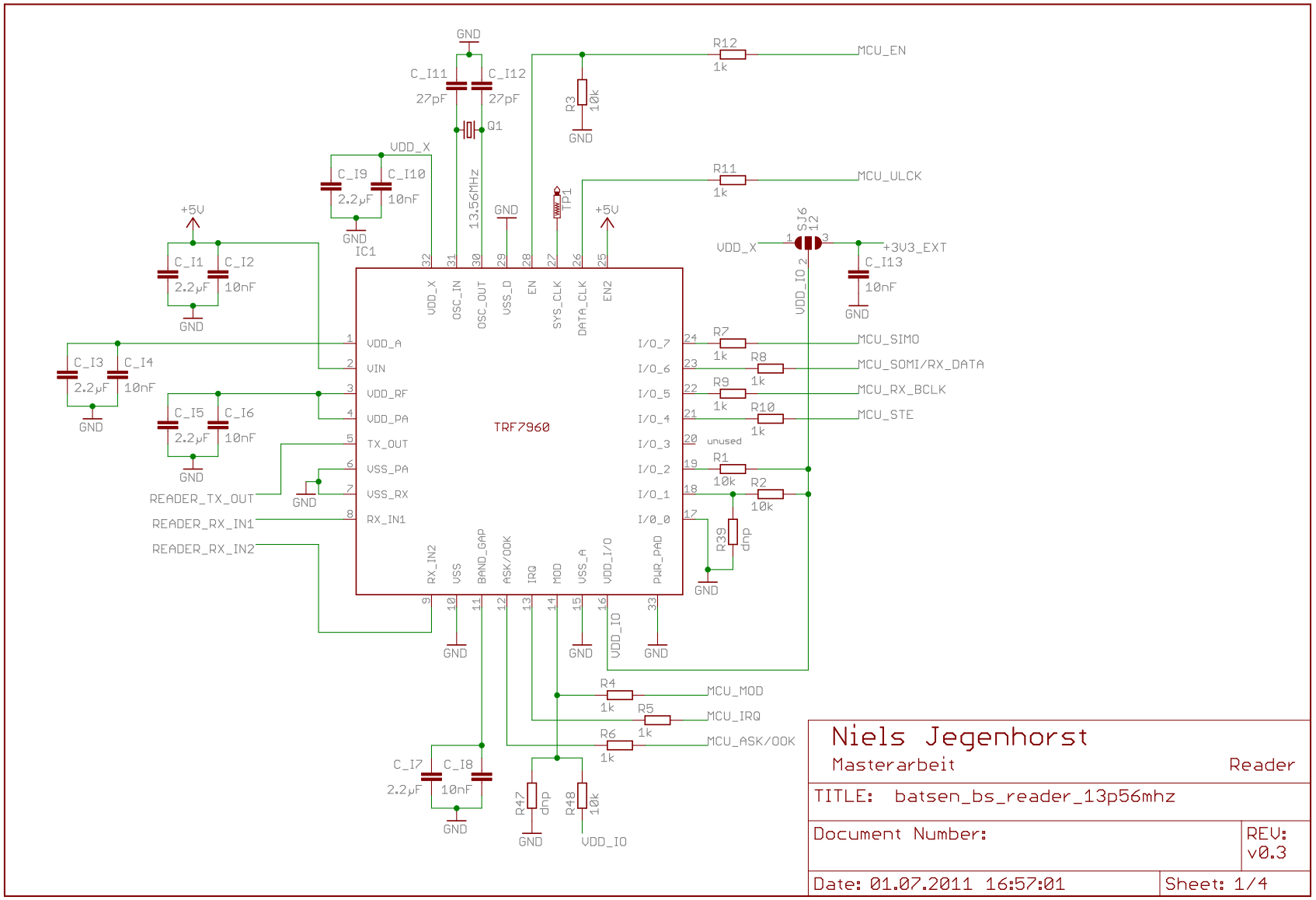
Niels Jegenhorst		
Masterarbeit		Reader
TITLE: batsen_bs_reader_13p56mhz		
Document Number:		REV: v0.2
Date: 28.03.2011 09:55:48	Sheet: 1/3	

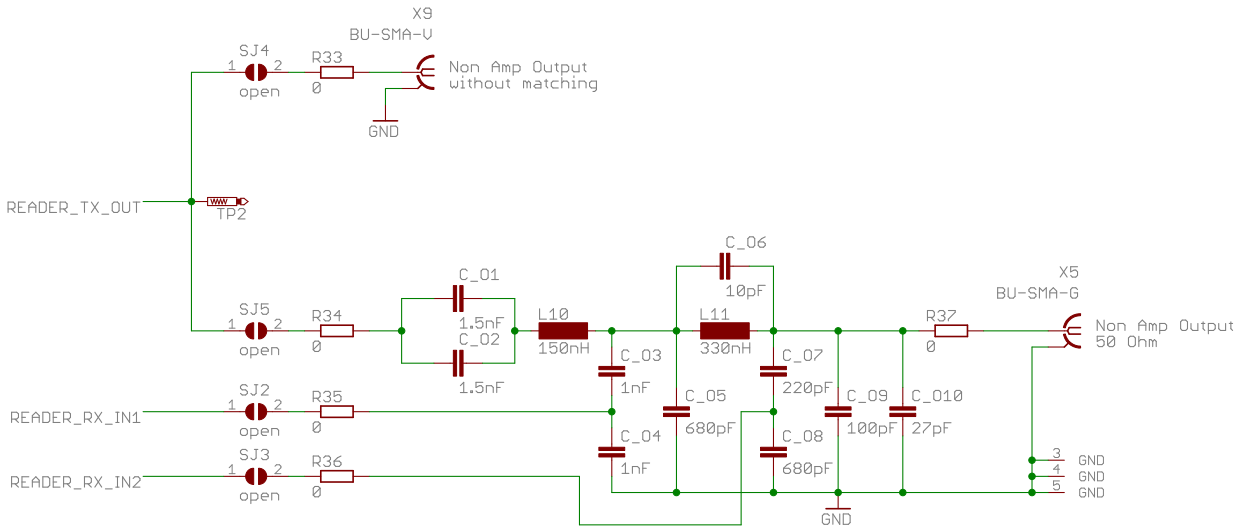


Niels Jegenhorst	
Masterarbeit	Output Circuit
TITLE: batsen_bs_reader_13p56mhz	
Document Number:	REV: v0.2
Date: 28.03.2011 09:55:48	Sheet: 2/3

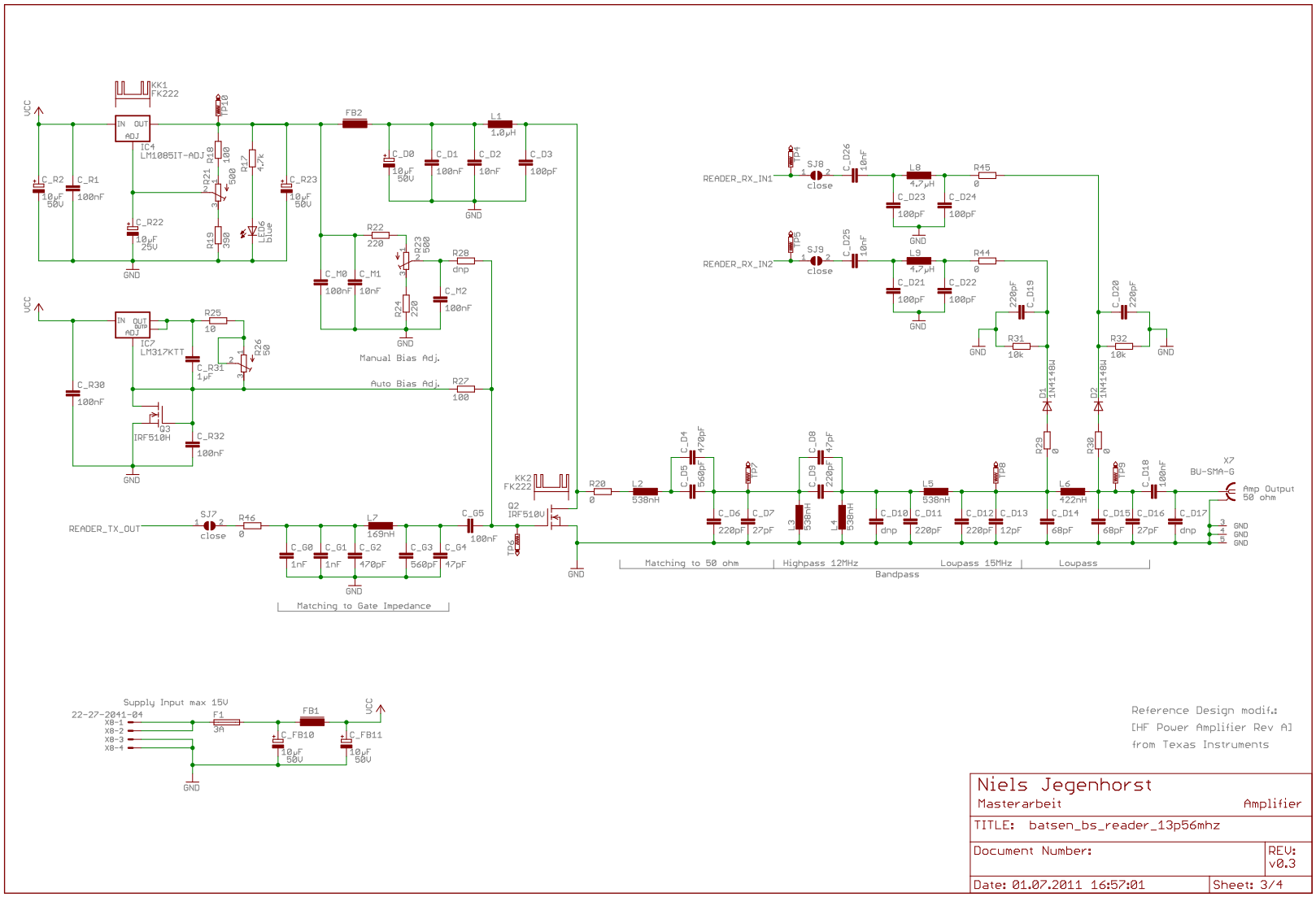

+3V3
SJ1
open
22-27-2061-06
X1-1
X1-2
X1-3
X1-4
X1-5
X1-6
+3V3_EXT
MCU_SIF
MCU_SIMO
MCU_SOMI/RX_DATA
MCU_ULCK
JP1
GND
22-27-2081-08
X2-1
X2-2
X2-3
X2-4
X2-5
X2-6
X2-7
X2-8
+3V3_EXT
MCU_IRQ
MCU_MOD
MCU_RX_BCLK
MCU_EN
DNU
MCU_ASK/OOK
JP2
GND
MCU_EN
MCU_MOD
MCU_SOMI/RX_DATA
DNU
act. RX data 433MHz
IC5A
IC5B
IC6A
IC6B
SN74LVC2G34
R40
R41
R42
R43
680
220
680
680
yellow
LED2
blue
LED3
red
LED4
red
LED5
JP3
JP4
JP5
JP6
GND
22-27-2041-04
X3-1
X3-2
X3-3
X3-4
Supply Input max 15V
GND
IN
OUT
ADJ
IC2
LM317KTT
adj to 5.1V
C_R10
10µF
50V
C_R11
100nF
C_R12
10µF
10V
R13
121
R14
374
R38
680
LED1
blue
+5V
GND
22-27-2041-04
X6-1
X6-2
X6-3
X6-4
+5V
GND
+5V
+3V3
IC3
LM317EMP
adj to 3.3V
C_R20
10µF
C_R21
100nF
optional
R15
390
R16
620
GND
+3V3
+3V3
IC5P
IC6P
VCC
GND
C_IC5
C_IC6
100nF
100nF
GND
GND
Niels Jegenhorst
Masterarbeit
Connectors
TITLE: batsen_bs_reader_13p56mhz
Document Number:
REV: v0.2
Date: 28.03.2011 09:55:48
Sheet: 3/3

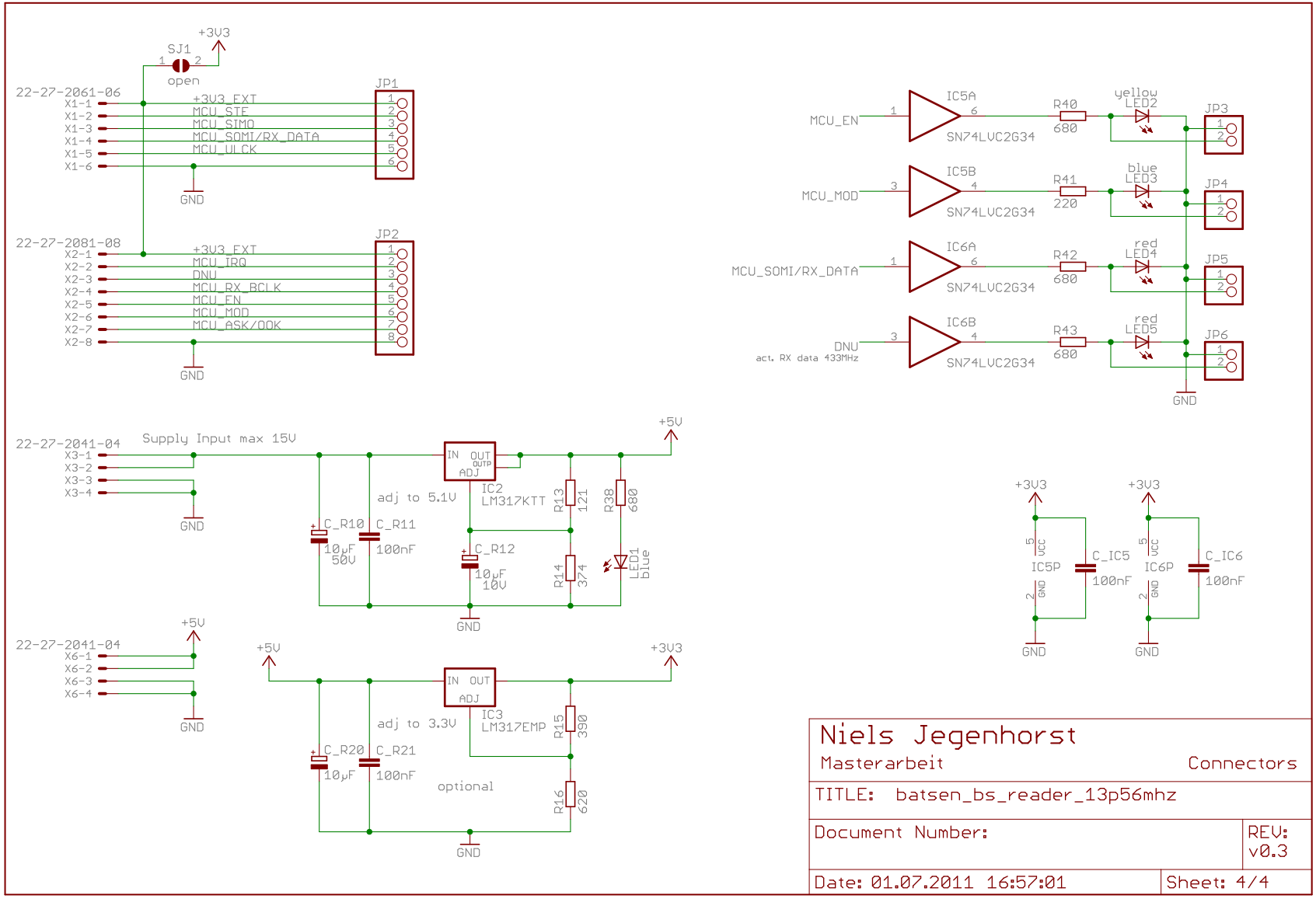
B.2.2 Schaltplan Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.3“


TRF7960
VDD_A
VIN
VDD_RF
VDD_PA
TX_OUT
VSS_PA
VSS_RX
RX_IN1
RX_IN2
VSS
BAND_GAP
ASK/OOK
IRQ
MOD
VSS_A
VDD_I/O
PWR_PAD
VDD_X
OSC_IN
OSC_OUT
VSS_D
EN
SYS_CLK
DATA_CLK
EN2
I/O_7
I/O_6
I/O_5
I/O_4
I/O_3
I/O_2
I/O_1
I/O_0
unused
GND
+5V
VDD_X
VDD_IO
+3V3_EXT
MCU_EN
MCU_ULCK
MCU_SIMO
MCU_SOMI/RX_DATA
MCU_RX_BCLK
MCU_STE
MCU_MOD
MCU_IRQ
MCU_ASK/OOK
READER_TX_OUT
READER_RX_IN1
READER_RX_IN2
C_I1
C_I2
C_I3
C_I4
C_I5
C_I6
C_I7
C_I8
C_I9
C_I10
C_I11
C_I12
C_I13
2.2µF
10nF
27pF
13.56MHz
Q1
IC1
TP1
SJ6
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R39
R47
R48
10k
1k
dnp
Niels Jegenhorst
Masterarbeit
Reader
TITLE: batsen_bs_reader_13p56mhz
Document Number:
REV: v0.3
Date: 01.07.2011 16:57:01
Sheet: 1/4

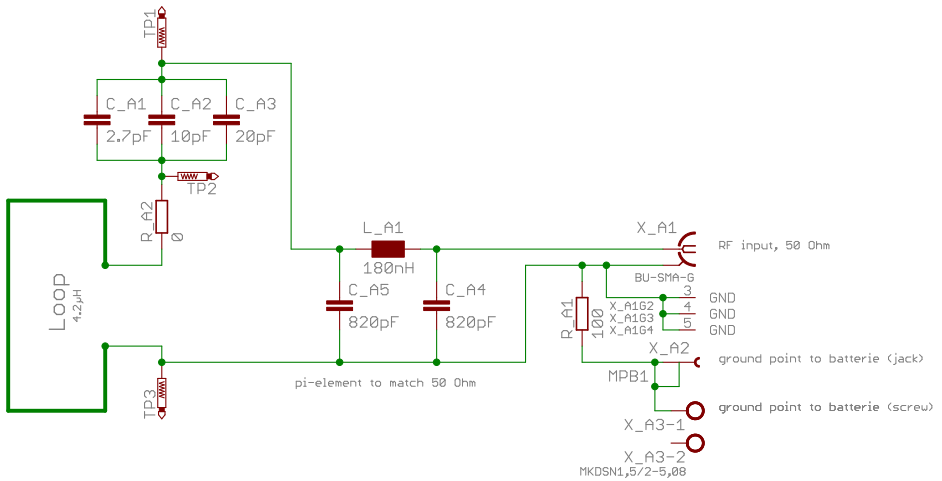


Niels Jegenhorst	
Masterarbeit	Output Circuit
TITLE: batsen_bs_reader_13p56mhz	
Document Number:	REV: v0.3
Date: 01.07.2011 16:57:01	Sheet: 2/4


VCC
KK1
FK222
IC4
LM1085IT-ADJ
IN OUT
ADJ
TP10
FB2
L1
1.0µH
C_R2
10µF
50V
C_R1
100nF
C_R22
10µF
25V
R18
300 100
R21
R17
4.7k
R19
390
LED6
blue
C_R23
10µF
50V
C_D0
10µF
50V
C_D1
100nF
C_D2
10nF
C_D3
100pF
GND
R22
220
R23
500
R28
dnp
C_M0
100nF
C_M1
10nF
C_M2
100nF
R24
220
Manual Bias Adj.
Auto Bias Adj.
R27
100
IC7
LM317KTT
R25
10
R26
50
C_R31
1µF
C_R30
100nF
Q3
IRF510H
C_R32
100nF
KK2
FK222
R20
0
L2
538nH
C_D4
470pF
C_D5
560pF
C_D6
220pF
C_D7
27pF
C_D8
47pF
C_D9
220pF
L3
538nH
L4
538nH
L5
538nH
C_D10
dnp
C_D11
220pF
C_D12
220pF
C_D13
12pF
L6
422nH
C_D14
68pF
C_D15
68pF
C_D16
27pF
C_D17
dnp
C_D18
100nF
X7
BU-SMA-G
Amp Output
50 ohm
GND
GND
GND
Matching to 50 ohm
Highpass 12MHz
Loupass 15MHz
Loupass
Bandpass
READER_RX_IN1
READER_RX_IN2
SJ8
SJ9
close
C_D26
10nF
C_D25
10nF
L8
4.7µH
L9
4.7µH
R45
0
R44
0
C_D23
100pF
C_D24
100pF
C_D21
100pF
C_D22
100pF
C_D19
220pF
C_D20
220pF
R31
10k
R32
10k
D1
1N4148W
D2
1N4148W
R29
0
R30
0
READER_TX_OUT
SJ7
R46
0
L7
169nH
C_G5
100nF
C_G0
1nF
C_G1
1nF
C_G2
470pF
C_G3
560pF
C_G4
47pF
Q2
IRF510U
Matching to Gate Impedance
Supply Input max 15V
22-27-2041-04
X8-1
X8-2
X8-3
X8-4
F1
3A
FB1
C_FB10
10µF
50V
C_FB11
10µF
50V
Reference Design modif.:
[HF Power Amplifier Rev A]
from Texas Instruments
Niels Jegenhorst
Masterarbeit
Amplifier
TITLE: batsen_bs_reader_13p56mhz
Document Number:
REV: v0.3
Date: 01.07.2011 16:57:01
Sheet: 3/4

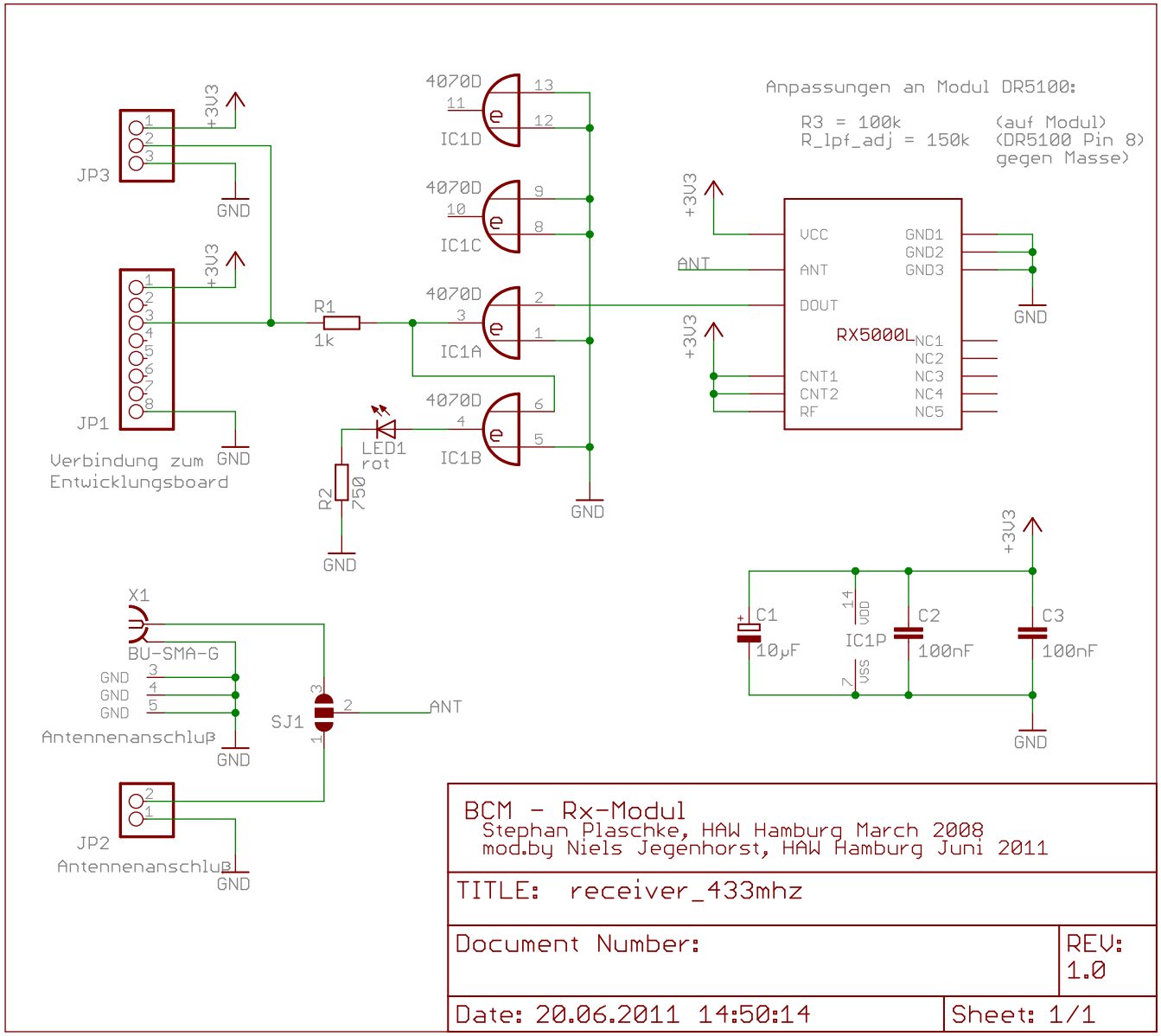

+3V3
SJ1
open
22-27-2061-06
X1-1
X1-2
X1-3
X1-4
X1-5
X1-6
+3V3_EXT
MCU_STE
MCU_SIMO
MCU_SOMI/RX_DATA
MCU_ULCK
JP1
GND
22-27-2081-08
X2-1
X2-2
X2-3
X2-4
X2-5
X2-6
X2-7
X2-8
+3V3_EXT
MCU_IRQ
DNU
MCU_RX_BCLK
MCU_EN
MCU_MOD
MCU_ASK/OOK
JP2
GND
MCU_EN
MCU_MOD
MCU_SOMI/RX_DATA
DNU
act. RX data 433MHz
IC5A
IC5B
IC6A
IC6B
SN74LVC2G34
R40
R41
R42
R43
680
220
680
680
yellow
LED2
blue
LED3
red
LED4
red
LED5
JP3
JP4
JP5
JP6
GND
22-27-2041-04
X3-1
X3-2
X3-3
X3-4
Supply Input max 15V
GND
IN
OUT
ADJ
IC2
LM317KTT
adj to 5.1V
C_R10
C_R11
10µF
50V
100nF
C_R12
10µF
10V
R13
121
R14
374
R38
680
LED1
blue
+5V
GND
22-27-2041-04
X6-1
X6-2
X6-3
X6-4
+5V
GND
+5V
+3V3
IC3
LM317EMP
adj to 3.3V
C_R20
C_R21
10µF
100nF
optional
R15
390
R16
620
GND
+3V3
+3V3
IC5P
IC6P
VCC
GND
C_IC5
C_IC6
100nF
100nF
GND
GND
Niels Jegenhorst
Masterarbeit
Connectors
TITLE: batsen_bs_reader_13p56mhz
Document Number:
REV: v0.3
Date: 01.07.2011 16:57:01
Sheet: 4/4

B.2.3 Schaltplan Antennenplatine „BATSEN BS Antenna v0.1“

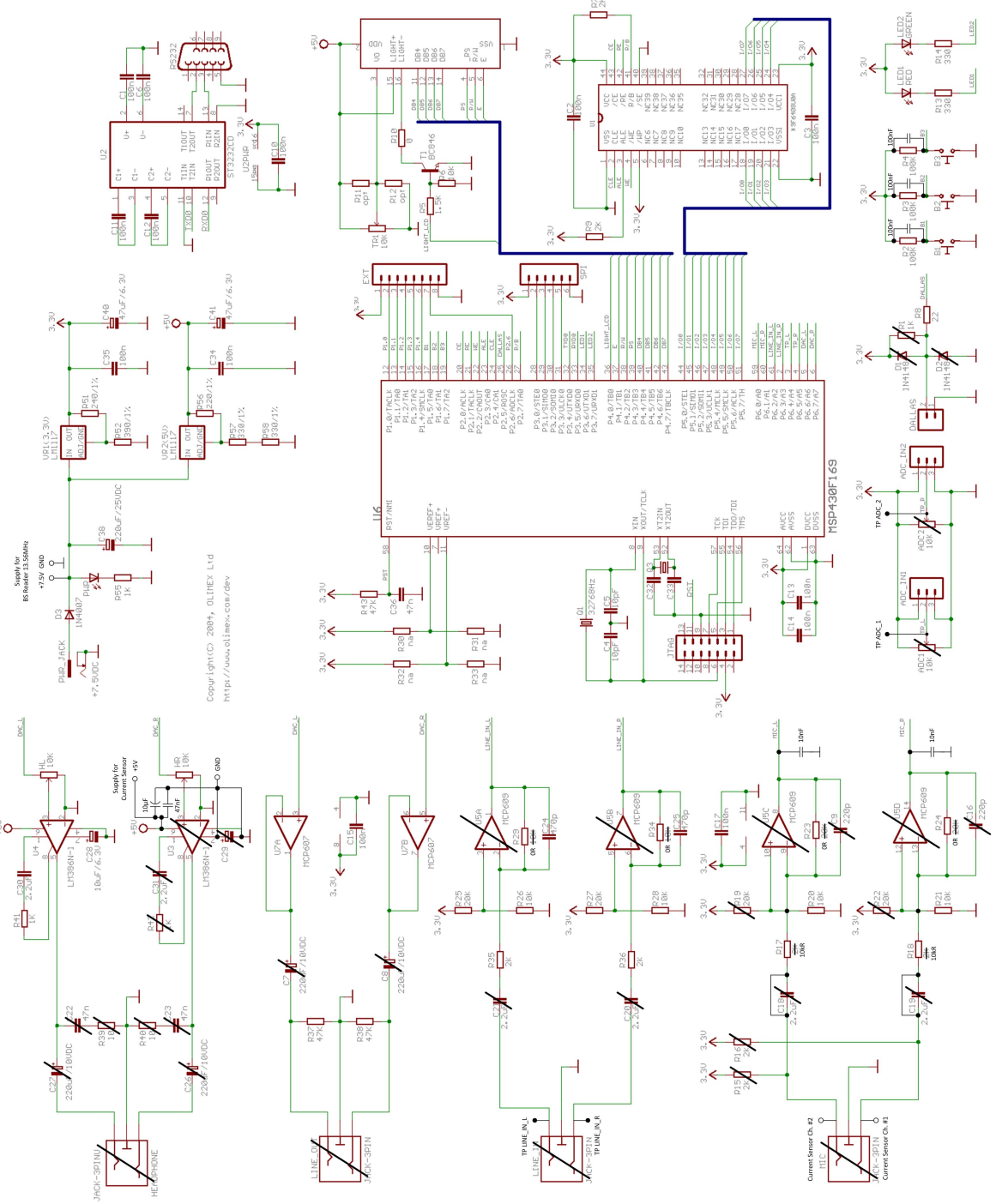


Niels Jegenhorst	
Masterarbeit	Antenne Basisstation
TITLE: batsen_bs_antenna	
Document Number:	REV: v0.1
Date: 18.03.2011 14:23:14	Sheet: 1/1

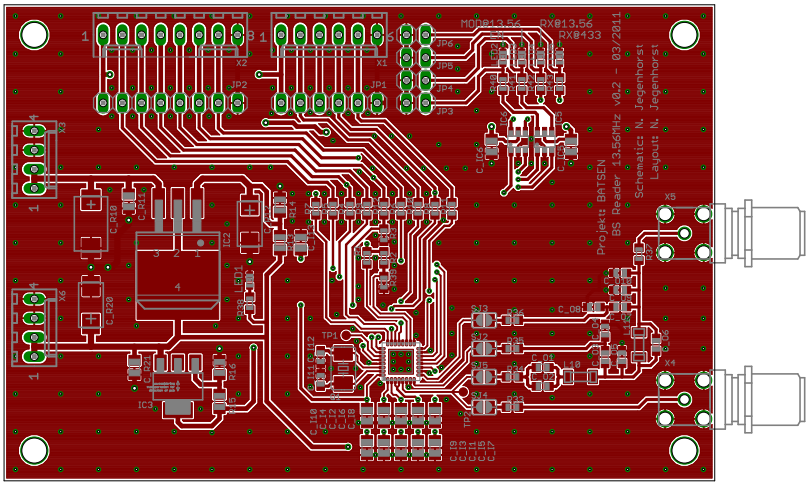
B.2.4 Schaltplan UHF-Empfänger-Modul „BCM Rx-Modul v0.2“


Anpassungen an Modul DR5100:
R3 = 100k (auf Modul)
R_lpf_adj = 150k (DR5100 Pin 8) gegen Masse)
JP3
JP1
Verbindung zum Entwicklungsboard
JP2
Antennenanschluß
R1
1k
R2
750
LED1
rot
4070D
IC1D
IC1C
IC1A
IC1B
RX5000L
VCC
ANT
DOUT
CNT1
CNT2
RF
GND1
GND2
GND3
NC1
NC2
NC3
NC4
NC5
C1
10µF
C2
100nF
C3
100nF
IC1P
VDD
VSS
X1
BU-SMA-G
SJ1
+3V3
GND
ANT
BCM - Rx-Modul
Stephan Plaschke, HAW Hamburg March 2008
mod.by Niels Jegenhorst, HAW Hamburg Juni 2011
TITLE: receiver_433mhz
Document Number:
REV: 1.0
Date: 20.06.2011 14:50:14
Sheet: 1/1

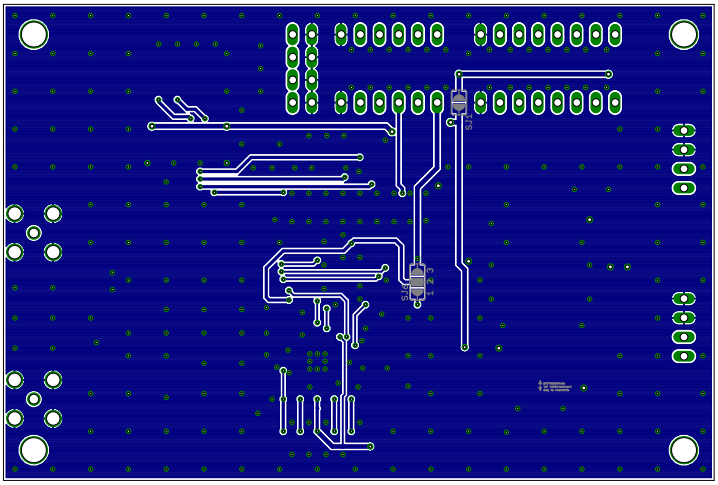
B.2.5 Schaltplan modifiziertes Entwicklungsboard „Olimex MSP430-169STK“



B.2.6 Layout Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.2“



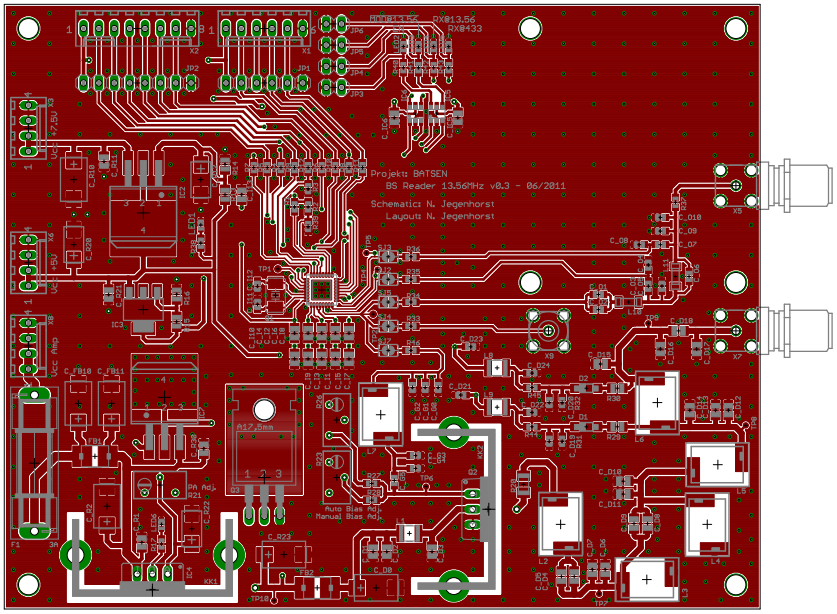
(a) Oberseite



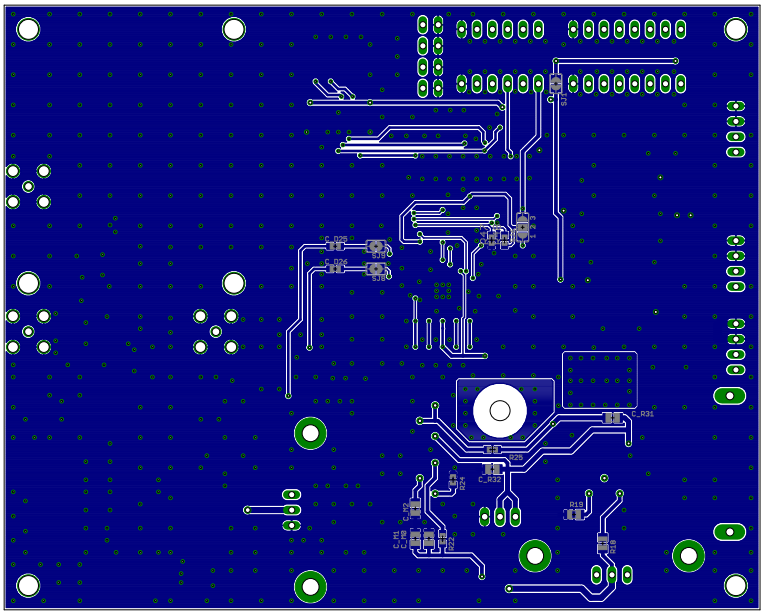
(b) Unterseite

Bild B.4: *Layout „BATSEN BS Reader 13.56Mhz v0.2“* ($s = 1.0$)

B.2.7 Layout Reader-Einheit „BATSEN BS Reader 13.56Mhz v0.3“



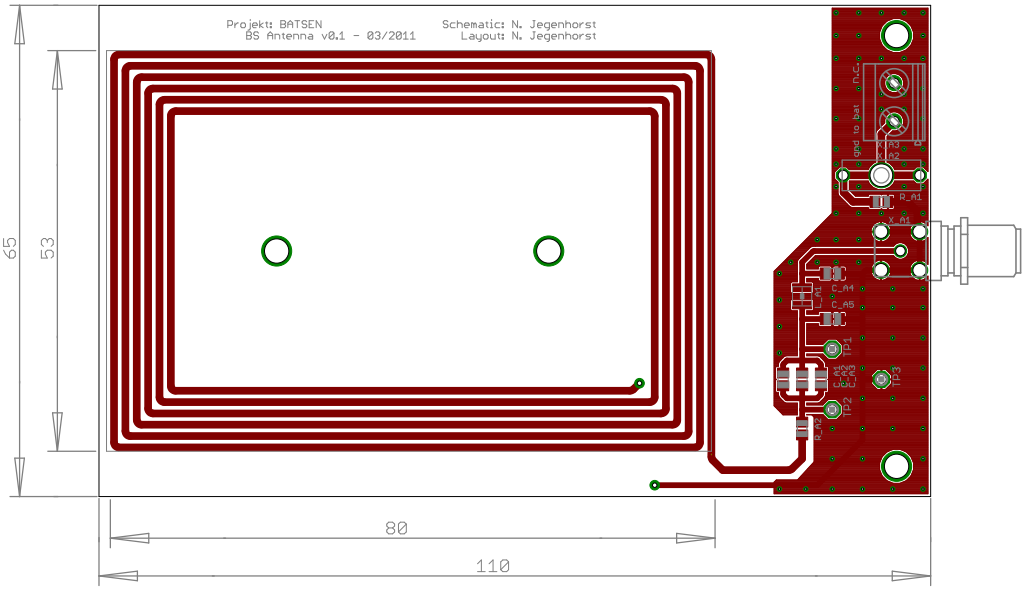
(a) Oberseite



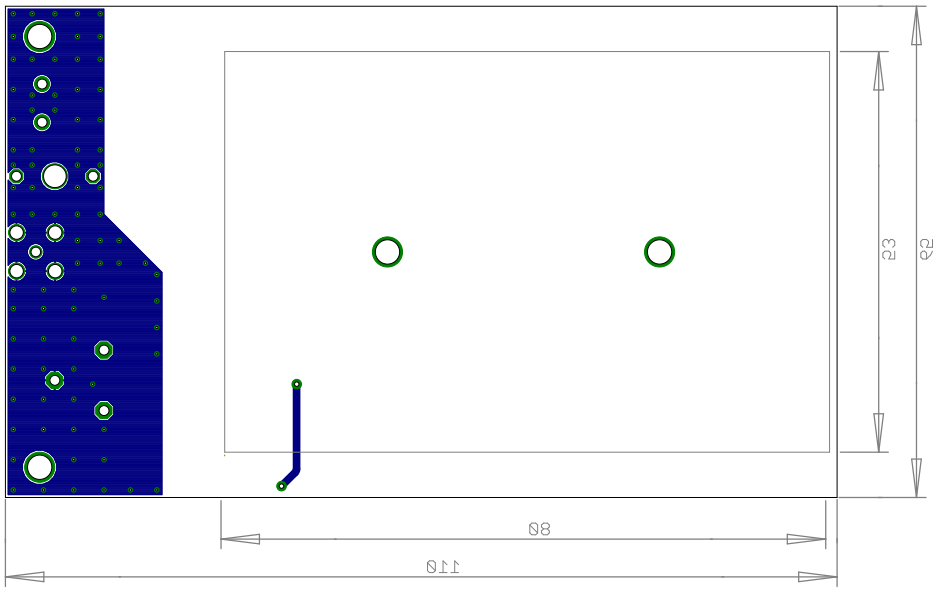
(b) Unterseite

Bild B.5: *Layout „BATSEN BS Reader 13.56Mhz v0.3“* ($s = 0.8$)

B.2.8 Layout Antennenplatine „BATSEN BS Antenna v0.1“



(a) Oberseite



(b) Unterseite

Bild B.6: *Layout „BATSEN BS Antenna v0.1“* ($s = 1.0$)

B.2.9 Layout UHF-Empfänger-Modul „BCM Rx-Modul v0.2“

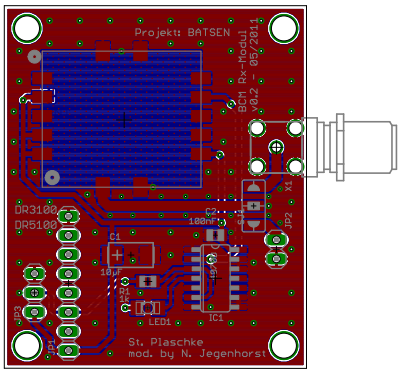


Bild B.7: *Layout „BCM Rx-Modul v0.2“* ($s = 1.0$)

C Quellcode

C.1 Mikrocontroller

C.1.1 Zellensensor

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    21.03.2011
 |  Geändert am:    26.09.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1/v0.2
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Hauptprogramm des Zellensensors
 |
 |  Funktion:       OK
 |  Datei:          main.c
 |  zug. Dateien:   header_main.h, globals.h, cleanup.c, init.c, ...
 ================================================================================================
*/

#define EXTERN                                          // define globals here

#include "header_main.h"
#include "src/globals.h"
#include "src/temp_sensor.h"
#include "src/tx_433.h"
#include "src/adc.h"

//--- Hauptprogramm ------------------------------------------------------------------------------
int main(void) {

    uint8_t check = FALSE;

    init();                                         // Initialize MSP and:
                                                    //  - Power off 433MHz Transmitter
                                                    //  - Keep DC/DC activ
                                                    //  - Set DC/DC not in PS-Mode
                                                    //  - Load-Modulation off
                                                    //  - LED1 & LED2 off
    cleanup();                                      // Globale Variablen rücksetzen
    DCDC_PS_ON;                                     // Set DC/DC in PS-Mode
    infoflash_read();                               // Read Control Data from Flash

    g_rx.state  = PREPARE;
    g_rx.status = WAIT_FIRST_RX;
    g_states    = INIT;
    g_mode      = STARTUP;
```

```
45      eint();                                         // IRQs enable
47      WDTCTL = WDTCTL_CLR;                            // WDT counter clear
48      TACCTL0 |= CCIE;                                // Enable RX
50      TP13_ON;
51      TX433_OFF;                                      // Power off 433MHz Transmitter
52      temp_sensor_init();                             // Initialize Temperatur Sensor first,
53                                                      //  because TX433 needs time to shut down
54      while((P1IN & PIN6) != PIN6);                   // Wait until NIRQ of TX433 goes high
55      TX433_ON;                                       // Power on 433MHz Transmitter
56      WDTCTL = WDTCTL_CLR;                            // WDT counter clear
57                                                      // Wait for power-on-reset of TX433
58      while((g_tx433.irq_status & TX433_IRQ_POR) != TX433_IRQ_POR);
59      WDTCTL = WDTCTL_CLR;                            // WDT counter clear
60      check = tx433_init();                           // Initialize 433MHz Transmitter
61      if(check != EXIT_SUCCESS) {                     // Exit Failure --> Restart
62          WDTCTL = WDTCTL_PUC;
63      }
64      WDTCTL = WDTCTL_CLR;                            // WDT counter clear
65      frame_tx433_init();                             // Initialize 433MHz TX Frame
66      g_states = IDLE;                                // state --> Idle
67      TP13_OFF;
69      LED1_ON;                                        // LED1 on
71      while(TRUE) {                   // === infinite loop ================================
73  //      TXLMOD_TOG;                                 // For Demonstration of Load-Modulation
75          WDTCTL = WDTCTL_CLR;                        // WDT counter clear
76          // --- State machine: ---------------------------------------------------------
77          switch (g_states) {
78              case INIT:                      break;
79              case IDLE:                      break;
80              case WAIT_BEFOR_SAMPLE:         break;
81              case SAMPLE:                    break;
82              case WAIT_BEFOR_PREPARE_TX:     break;
83              case PREPARE_TX:                break;
84              case WAIT_BEFOR_TX:             break;
85              case TX_0_WAIT_RX:              break;
86              case TX_1_START:                break;
87              case TX_2_WAIT_TX:              break;
88              default:                        break;
89          }                           // ___ end switch case _____________________________
90      }                               // === infinite loop end ============================
91      return 0;
92  }
93  //-------------------------------------------------------------------------------------
```

Programmausdruck C.1: *Zellensensor „main.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    21.03.2011
6   |  Geändert am:    26.09.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1/v0.2
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Haupt-Header-Datei
10  |
11  |  Funktion:       OK
12  |  Datei:          header_main.h
13   ================================================================================================
14  */

16  #ifndef HEADER_MAIN_H_
17  #define HEADER_MAIN_H_

19      #define __MSP430_HAS_ADC12__                    // fix da Fehler in msp430x23x.h
20      #define __MSP430_HAS_SVS__                      // fix da Fehler in msp430x23x.h
21      #define __msp430_has_svs_at_0x55                // fix da Fehler in msp430x23x.h

23      #include <msp430x23x.h>
24      #include <msp430/adc12.h>                       // fix da Fehler in msp430x23x.h
25      #include <msp430/svs.h>                         // fix da Fehler in msp430x23x.h
26      #include <signal.h>

28      #define LEDS_ENABLE                             // enable Status LED1 & LED2

30      //___ Sensor Address & Version ID: _______________
31      #define SENSOR_ADDRESS          0x00028         // Sensor-Address in 20-bits = 40
32      #define SENSOR_VID              0x03            // Version ID
33      #define SENSOR_GLOBAL_ADDRESS   0x7FFFF         // global Sensor-Address for all

35      //___ Timings & Clocks: System ___________________
36      #define DCOCLK              1000.0              // DCO Clock [kHz]

38      #define MCLK_DIV            1.0                 // Clock divider for MCLK
39      #define SMCLK_DIV           2.0                 // Clock divider for SMCLK
40      #define BCSCTL2_MCLK_DIV    (0x00)              // Value for MCLK divider setting
41      #define BCSCTL2_SMCLK_DIV   (DIVS0)             // Value for SMCLK divider setting

43      #define MCLK                1000.0              // Main Clock (from DCO) [kHz]
44      #define SMCLK               500.0               // SubMain Clock (from DCO) [kHz]

46      #define TA_DIV_SMCLK        1.0                 // Clock divider for Timer A
47      #define TB_DIV_SMCLK        8.0                 // Clock divider for Timer B
48                                                      // Stepsize Timer A [us]
49      #define TA_STEP             ((TA_DIV_SMCLK * 1000.0 / SMCLK))
50                                                      // Stepsize Timer B [us]
51      #define TB_STEP             ((TB_DIV_SMCLK * 1000.0 / SMCLK))

53      #define UCB0BR_BIT_CLK      100.0               // UCB0 Bit Clock [kHz]
54                                                      // UCB0 Baud Rate Control 0 Setting
55      #define UCB0BR_0_SET        ((uint16_t)( SMCLK / UCB0BR_BIT_CLK))
56      // All time values will be calculated by the Compiler, also by optimisation level 0
57      //___ Timings & Clocks: Operating Sequence _______
58                                              // Time to sample = 10ms (default)
59      #define TIME_TO_SAMPLE      ((uint16_t)( 10000 / TB_STEP))
60                                              // Time to tx     = 200ms (default)
61      #define TIME_TO_TX          ((uint16_t)(200000 / TB_STEP))
62                                              // Time for interval = 500ms (default)
```

```
63      #define TIME_INTERVAL       ((uint16_t)(500000 / TB_STEP))
64                                              // Time between Sample & TX = 25ms
65      #define TIME_SAMPLE_TO_TX   ((uint16_t)( 25000 / TB_STEP))
66                                              // Time between TX and End = 40ms
67      #define TIME_TX_TO_END      ((uint16_t)( 40000 / TB_STEP))
68                                              // Time to start with tune bevor TX  = 20ms
69      #define TIME_FOR_TUNE       ((uint16_t)( 20000 / TB_STEP))
70                                              // Time to tx min = 35ms
71      #define TIME_TO_TX_MIN      ((uint16_t)( 35000 / TB_STEP))
72                                              // Time to tx max = 460ms
73      #define TIME_TO_TX_MAX      ((uint16_t)(460000 / TB_STEP))

75      //___ Pseudo Random Generator: _____________________
76          // This values are designed for: SMCLK = 1MHz / 0.5MHz, div clk Timer B = 8
77          // - maximum random time ~ 500ms
78          //   = (RANDOM_REG_MODULO * 2^RANDOM_REG_SCALE) + TIME_TO_TX_MIN = 58775
79          //      with T_INC Timer B = (SMCLK/8)^-1 = 8us
80          //   = 58775 * 8us  = 470.2ms (SMCLK = 1.0MHz) (RANDOM_REG_SCALE = 6)
81          //   = 29388 * 16us = 470.2ms (SMCLK = 0.5MHz) (RANDOM_REG_SCALE = 5)
82      #define RANDOM_REG_MODULO   850                 // modulo value == max random value
83      #define RANDOM_REG_SCALE    5                   // scale of random value = 2^x

85      //___ TX 433MHz Dataframe: _________________________
86      #define TX433_FRAME_BITS        108             // 4Bit RUN-IN + 16Bit Synch
87                                                      //  + 88Bit Data
88      #define TX433_FRAME_BYTES       11              // TX data bytes = 88 / 8
89      #define TX433_FRAME_BYTES_MCH   27              // Manchester-coded TX bytes
90                                                      //  = (TX433_FRAME_BITS*2) / 8

92      #define TX433_PARA_B8           0x80            // Unused bit 8 in parameters
93      #define TX433_PARA_SCF          0x40            // Supply-voltage calb. factor
94      #define TX433_PARA_TCO          0x20            // Temperature calb. offset
95      #define TX433_PARA_TCF          0x10            // Temperature calb. factor
96                                                      // Data-Alignment in framedata[]:
97      #define TX433_FR_PARA_BYTE0     0
98      #define TX433_FR_ADDR_BYTE0     1               // only upper 4 bits for Address
99      #define TX433_FR_DATA1_BYTE0    4
100     #define TX433_FR_DATA2_BYTE0    6
101     #define TX433_FR_DATA3_BYTE0    8
102     #define TX433_FR_CRC_BYTE0      10
103                                                     // Data-Alignment in txdata[]:
104     #define TX433_TX_SOF_BYTE0      1
105     #define TX433_TX_PARA_BYTE0     6
106     #define TX433_TX_ADDR_BYTE0     9
107     #define TX433_TX_DATA1_BYTE0    14
108     #define TX433_TX_DATA2_BYTE0    18
109     #define TX433_TX_DATA3_BYTE0    22
110     #define TX433_TX_CRC_BYTE0      26

112     //___ RX 13.56MHz Dataframe: _______________________
113     #define RX13P56_ERROR_LOG                       // Error logging enable
114     #define RX13P56_FRAME_BYTES     6               // Number of Frame Bytes
115     #define RX13P56_SYNCH_LENGTH    8               // 8-Bit Synch
116                                                     // Time for RUN IN min: 600us+100us
117     #define RX13P56_RUN_IN_MIN      ((uint16_t)( 700 / TA_STEP))
118                                                     // Time for RUN IN min: 900us+100us
119     #define RX13P56_RUN_IN_MAX      ((uint16_t)(1000 / TA_STEP))
120                                                     // Timeout for RX process: 1100us
121     #define RX13P56_RUN_IN_TIMEOUT  ((uint16_t)(1100 / TA_STEP))

123     //___ SW-IRQ's: ____________________________________
124     #define SW_IRQ_SAMPLE           BIT0
125     #define SW_IRQ_TX               BIT1
```

```
126     #define SW_IRQ_TUNING           BIT2
127     #define SW_IRQ_STANDBY          BIT3

129     //___ I/O: ____________________________________________
130     #define PIN0        0x01
131     #define PIN1        0x02
132     #define PIN2        0x04
133     #define PIN3        0x08
134     #define PIN4        0x10
135     #define PIN5        0x20
136     #define PIN6        0x40
137     #define PIN7        0x80

139     //___ Return-Codes: ___________________________________
140     #define EXIT_SUCCESS    0x01
141     #define EXIT_FAILURE    0x00

143     //--- Typdefinitions: ------------------------------------------
144     typedef enum {
145         INIT,
146         IDLE,
147         WAIT_BEFOR_SAMPLE,
148         SAMPLE,
149         WAIT_BEFOR_PREPARE_TX,
150         PREPARE_TX,
151         WAIT_BEFOR_TX,
152         TX_0_WAIT_RX,
153         TX_1_START,
154         TX_2_WAIT_TX
155     } states;

157     typedef enum {
158         PREPARE,
159         RUN_IN,
160         SYNCH_0,
161         SYNCH_1,
162         DATALOAD,
163         COMPLETE
164     } rx_states;

166     typedef enum {
167         SUCCESS,
168         CRC_FAILURE,
169         TIMEOUT,
170         RUN_IN_ERROR,
171         SYNCH_ERROR,
172         BIT_ERROR,
173         ACTIVE,
174         INACTIVE,
175         WAIT_FIRST_RX
176     } rx_status;

178     typedef enum {
179         SAMPLE_SET1,
180         SAMPLE_SET2
181     } sample_set;

183     typedef enum {
184         STARTUP,
185         SCAN,
186         NORMAL,
187         REPLY,
188         HC
```

```
189     } system_mode;

191     //--- Enums: ------------------------------------------------
192     enum rx_commands {
193         ZS_CMD_STAY_ON          = 0x00,
194         ZS_CMD_TURN_OFF         = 0x01,
195         ZS_CMD_BAL_ON           = 0x02,
196         ZS_CMD_BAL_OFF          = 0x03,
197         ZS_CMD_SAMPLE_S1        = 0x04,
198         ZS_CMD_SAMPLE_S2        = 0x05,
199         ZS_CMD_SET_SAMPLE_TIME  = 0x06,
200         ZS_CMD_SET_TX_TIME      = 0x07,
201         ZS_CMD_SET_INTVAL_TIME  = 0x08,
202         ZS_CMD_SYNCH            = 0x09,
203         ZS_CMD_SCAN             = 0x0A,
204         ZS_CMD_REPLY            = 0x0B,
205         ZS_CMD_SAVE_SET         = 0x0C
206     };

208     enum boolean {
209         FALSE = 0x00,
210         TRUE  = 0x01
211     };

213     enum iostate {
214         LOW   = 0x00,
215         HIGH  = 0x01
216     };

218     //--- Settings for Ports: -----------------------------------
219         // P1 IRQ sources: SW IRQ on Pin 6
220     #define P1IE_SET        (PIN6)
221         // P2 IRQ sources: SW IRQ on Pin 0,1,2,3
222     #define P2IE_SET        (BIT0 | BIT1 | BIT2 | BIT3)

224     // --- Settings for WDT: ----------------------------------------
225         // WDT Counter clear
226     #define WDTCTL_CLR      (WDTPW | WDTCNTCL)
227         // WDT generate PUC
228     #define WDTCTL_PUC      (0x00)

230     //--- Macros: -----------------------------------------------
231     #define TA_STOP_CLR     {   TACTL &= ~(MC0 | MC1);      \
232                                 TACTL |= TACLR;             \
233                             }
234     #define TA_START            (TA0CTL |= (MC1))

236     #define TB_STOP_CLR     {   TBCTL &= ~(MC0 | MC1);      \
237                                 TBCTL |= TBCLR;             \
238                             }
239     #define TB_START            (TBCTL |= (MC0 | ID1 | ID0))

241     //___ LED Handling: _____________________________
242     #ifdef LEDS_ENABLE
243         #define LED1_ON         (P4OUT |=  PIN4)
244         #define LED1_OFF        (P4OUT &= ~PIN4)
245         #define LED1_TOG        (P4OUT ^=  PIN4)
246         #define LED2_ON         (P4OUT |=  PIN5)
247         #define LED2_OFF        (P4OUT &= ~PIN5)
248         #define LED2_TOG        (P4OUT ^=  PIN5)
249     #else
250         #define LED1_ON         (P4OUT &= ~PIN4)
251         #define LED1_OFF        (P4OUT &= ~PIN4)
```

```
252         #define LED1_TOG        (P4OUT &= ~PIN4)
253         #define LED2_ON         (P4OUT &= ~PIN5)
254         #define LED2_OFF        (P4OUT &= ~PIN5)
255         #define LED2_TOG        (P4OUT &= ~PIN5)
256     #endif
257     //___ DC/DC Handling: _____________________
258         // DC/DC turn on:   Signal DCDC_DIS = LOW, Signal DCDC_ACT = HIGH
259     #define DCDC_ON         (P1OUT &= ~PIN4, P1OUT |= PIN3)
260         // DC/DC turn off:  Signal DCDC_DIS = HIGH, Signal DCDC_ACT = LOW
261     #define DCDC_OFF        (P1OUT &= ~PIN3, P1OUT |= PIN4)
262         // DC/DC power save on
263     #define DCDC_PS_ON      (P1OUT &= ~PIN2)
264         // DC/DC power save off
265     #define DCDC_PS_OFF     (P1OUT |=  PIN2)
266     //___ TX 433MHz Transmitter: _______________
267     #define TX433_ON        (P1OUT &= ~PIN5)
268     #define TX433_OFF       (P1OUT |=  PIN5)
269     //___ TX 13.56MHz Load-Modulation: _________
270     #define TXLMOD_ON       (P2OUT |=  PIN7)
271     #define TXLMOD_OFF      (P2OUT &= ~PIN7)
272     #define TXLMOD_TOG      (P2OUT ^=  PIN7)
273     //___ Balancing: ___________________________
274     #define BALANCE_ON      (P3OUT |=  PIN7)
275     #define BALANCE_OFF     (P3OUT &= ~PIN7)

277     //___ Testpads: ____________________________
278     #define TP10_ON         (P5OUT |=  PIN0)        // Intervall Timer
279     #define TP10_OFF        (P5OUT &= ~PIN0)
280     #define TP10_TOG        (P5OUT ^=  PIN0)
281     #define TP11_ON         (P5OUT |=  PIN1)        // Sample, Prepare TX, Set Stby TX
282     #define TP11_OFF        (P5OUT &= ~PIN1)
283     #define TP11_TOG        (P5OUT ^=  PIN1)
284     #define TP12_ON         (P5OUT |=  PIN2)        // TX
285     #define TP12_OFF        (P5OUT &= ~PIN2)
286     #define TP12_TOG        (P5OUT ^=  PIN2)
287     #define TP13_ON         (P5OUT |=  PIN3)        // IRQ from TX
288     #define TP13_OFF        (P5OUT &= ~PIN3)
289     #define TP13_TOG        (P5OUT ^=  PIN3)

291     //--- InfoFlash: ------------------------------------------------
292     #define INFOFLASH_SEG_C     0x1040
293     #define INFOFLASH_VALID     0x11

295     //--- Prototyps: ------------------------------------------------
296     void init(void);
297     void cleanup(void);
298     void frame_tx433_init(void);
299     void frame_tx433_code_manchester(void);
300     void frame_tx433_calc_crc(void);
301     void frame_tx433_load_set1(void);
302     void frame_tx433_load_set2(void);
303     void frame_tx433_load_reply(void);
304     void generate_time_tx(void);
305     void infoflash_read(void);
306     void infoflash_write(void);

308 #endif  // HEADER_MAIN_H_
309 //-------------------------------------------------------------------------------------
```

Programmausdruck C.2: *Zellensensor „header_main.h“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    28.03.2011
6   |  Geändert am:    06.04.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Routinen für die Anwendung des internen ADC12
10  |
11  |  Funktion:       OK
12  |  Datei:          adc.c
13  |  zug. Dateien:   adc.h
14   ================================================================================================
15 */

17 #include "header_main.h"
18 #include "src/globals.h"
19 #include "src/adc.h"

21 //-------------------------------------------------------------------------------
22 //  Channel:    Source:     Division:   S&H-Time for 12-bit:    ADC12MEMx:
23 //      A0      U_BAT       0.5         >37.57us                0
24 //      A1      U_DCDC      0.5         >5.13us                 1
25 //      A3      U_ACT_IN    0.25        >80.82us                2
26 //      A4      U_TP        1           >271.85us               3
27 //-------------------------------------------------------------------------------
28 // ADC-CLK = MCLK = 1.0MHz  --> 1 ADC12CLK cycle = 1.0us

30 //--- Function: sample_ubat() ---------------------------------------------------------
31 void sample_ubat(void) {
32                                                      // conversion start address = 0
33     ADC12CTL0 |= (SHT0_64_CYCLES | ENC | ADC12SC);  // S&H = 64 -> 64us, start ADC,
34                                                      // single channel & conversion-mode

36     while((ADC12IFG & ADC12IFLAG_0) != ADC12IFLAG_0) ;  // wait for result

38     ADC12CTL0 &= ~(SHT0_64_CYCLES | ENC | ADC12SC); // reset for next sampling

40     g_adc.u_bat = ADC12MEM0 << 1;                    // save value, resets IRQ flag

42 }
43 //--- Function: sample_udcdc() --------------------------------------------------------
44 void sample_udcdc(void) {

46     ADC12CTL1 |= CSTARTADD_1;                        // conversion start address = 1
47     ADC12CTL0 |= (SHT0_16_CYCLES | ENC | ADC12SC);  // S&H = 16 -> 16us, start ADC,
48                                                      // single channel & conversion-mode

50     while((ADC12IFG & ADC12IFLAG_1) != ADC12IFLAG_1);   // wait for result

52     ADC12CTL0 &= ~(SHT0_16_CYCLES | ENC | ADC12SC); // reset for next sampling
53     ADC12CTL1 &= ~CSTARTADD_1;

55     g_adc.u_dcdc = ADC12MEM1 << 1;                   // save value, resets IRQ flag

57 }
58 //--- Function: sample_uact_in() ------------------------------------------------------
59 void sample_uact_in(void) {

61     ADC12CTL1 |= CSTARTADD_2;                        // conversion start address = 2
62     ADC12CTL0 |= (SHT0_96_CYCLES | ENC | ADC12SC);  // S&H = 96 -> 96us, start ADC,
```

```
63                                                  // single channel & conversion-mode

65      while((ADC12IFG & ADC12IFLAG_2) != ADC12IFLAG_2);   // wait for result

67      ADC12CTL0 &= ~(SHT0_96_CYCLES | ENC | ADC12SC); // reset for next sampling
68      ADC12CTL1 &= ~CSTARTADD_2;

70      g_adc.u_act_in = ADC12MEM2 << 2;                // save value, resets IRQ flag

72  }
73  //--- Function: sample_ulp() ------------------------------------------------------------
74  void sample_ulp(void) {

76      ADC12CTL1 |= CSTARTADD_3;                       // conversion start address = 3
77      ADC12CTL0 |= (SHT0_384_CYCLES | ENC | ADC12SC); // S&H = 384 -> 384us, start ADC,
78                                                      // single channel & conversion-mode

80      while((ADC12IFG & ADC12IFLAG_3) != ADC12IFLAG_3);   // wait for result

82      ADC12CTL0 &= ~(SHT0_384_CYCLES | ENC | ADC12SC);// reset for next sampling
83      ADC12CTL1 &= ~CSTARTADD_3;

85      g_adc.u_lp = ADC12MEM3;                         // save value, resets IRQ flag

87  }
88  //-----------------------------------------------------------------------------------------
```

Programmausdruck C.3: *Zellensensor „adc.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:         Masterarbeit, Zellensensor
4   |  Erstellt von:    Niels Jegenhorst
5   |  Erstellt am:     06.04.2011
6   |  Geändert am:     06.04.2011
7   |  Hardware:        MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:           MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:    Header-Datei für die Anwendung des internen ADC12
10  |
11  |  Funktion:        OK
12  |  Datei:           adc.h
13   ================================================================================================
14  */

16  #ifndef ADC_H_
17  #define ADC_H_

19      //___ ADC12 Control: _____________________________
20      #define SHT0_4_CYCLES       0x0000
21      #define SHT0_8_CYCLES       0x0100
22      #define SHT0_16_CYCLES      0x0200
23      #define SHT0_32_CYCLES      0x0300
24      #define SHT0_64_CYCLES      0x0400
25      #define SHT0_96_CYCLES      0x0500
26      #define SHT0_128_CYCLES     0x0600
27      #define SHT0_192_CYCLES     0x0700
28      #define SHT0_256_CYCLES     0x0800
29      #define SHT0_384_CYCLES     0x0900
30      #define SHT0_512_CYCLES     0x0A00

32      #define ADC12IFLAG_0        0x0001
33      #define ADC12IFLAG_1        0x0002
34      #define ADC12IFLAG_2        0x0004
35      #define ADC12IFLAG_3        0x0008

37      //--- Prototyps: ------------------------------------------------
38      void sample_ubat(void);
39      void sample_udcdc(void);
40      void sample_uact_in(void);
41      void sample_ulp(void);

43  #endif /* ADC_H_ */
44  //------------------------------------------------------------------------------------------------
```

Programmausdruck C.4: *Zellensensor „adc.h“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    21.03.2011
 |  Geändert am:    08.06.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Rücksetzen der globalen Variablen
 |
 |  Funktion:       OK
 |  Datei:          cleanup.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"

void cleanup(void) {

    #ifdef RX13P56_ERROR_LOG
        g_rx_error.bit_error    = 0;
        g_rx_error.timeout      = 0;
        g_rx_error.synch_error  = 0;
        g_rx_error.wrong_run_in = 0;
    #endif

    g_adc.tosample           = SAMPLE_SET1;
    g_timing.time_to_sample  = FALSE;
    g_timing.time_to_tx      = FALSE;
    g_timing.time_to_tx_tune = FALSE;
    g_timing.t_interval      = 0x0000;
    g_timing.t_tx            = TIME_TO_TX;
    g_timing.t_sample        = TIME_TO_SAMPLE;
    g_timing.t_tx_tune       = TIME_TO_TX - TIME_FOR_TUNE;

    g_rx.cmd[0] = 0x00;
    g_rx.cmd[1] = 0x00;
    g_rx.cmd[2] = 0x00;
    g_rx.parameter[0] = 0x0000;
    g_rx.parameter[1] = 0x0000;
    g_rx.parameter[2] = 0x0000;

    g_tx433.irq_status = 0x00;

    random_reg   = SENSOR_ADDRESS | 0x08000000; // Init register of pseudo-gen.
    random_reg <<= 4;                           // need for different start values
    random_reg  &= 0x0FFFFFFF;                  // only allow 28-bit

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

Programmausdruck C.5: *Zellensensor „cleanup.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    12.04.2011
 |  Geändert am:    14.06.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Funktionen zur Erzeugung des Datenframes für den Uplink
 |
 |  Funktion:       OK
 |  Datei:          frame_tx.c
 ================================================================================================
*/
/*
 ------------------------------------------------------------------------------------------------
    Frame for TX on 433MHz:

    |       SOF        | Parameters |  Address  |   Data    |   CRC     |
    |------------------|------------|-----------|-----------|-----------|
    | RUN-IN    SYNCH  |            |           | TV CV SV  |           |
    | 4-Bit     16-Bit | 12-Bit     | 20-Bit    | 48-Bit    | 8-Bit     | -> 108 Bits
    |------------------|------------|-----------|-----------|-----------|
    |                  | 0       11 | 12     31 | 32     79 | 80     88 | -> in framedata
    |------------------|------------|-----------|-----------|-----------|
    | 0    7    8   39 | 40      63 | 64    103 | 104   199 | 200   215 | -> in txdata
    |------------------|------------|-----------|-----------|-----------|   on 10.0kb/s
    | 800us     3.2ms  |    2.4ms   |   4ms     |   9.6ms   |   1.6ms   | -> 21.6ms

    Datarate = 5.0kb/s
    Transmissionrate = 10.0kb/s (Manchester)

        Parameters: | Bit:
        ------------------------
            VID     | 0-7
            unused  | 8
            SCF     | 9
            TCO     | 10
            TCF     | 11

        CRC-Checksum is generated over Parameters, Address and Data (bits 40 - 199)

 ------------------------------------------------------------------------------------------------
*/

#include "header_main.h"
#include "src/globals.h"

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

//--- Function: frame_tx433_init() -------------------------------------------------------------
void frame_tx433_init(void) {

    uint8_t i;

        // g_tx433.framedata[] contains only the 88 bits of dataload of the frame
        // (Parameters, Address, Data, CRC), without the SOF.
        // The data is not coded in manchester.

    //___ Frame Data: ____________________________________
    g_tx433.framedata[0] = SENSOR_VID;                      // Version ID
    g_tx433.framedata[1] = (uint8_t)((uint32_t)SENSOR_ADDRESS>>16);// Address bits 19-16
```

```
63                                                          // other Parameters left 0
64      g_tx433.framedata[2] = (uint8_t)(SENSOR_ADDRESS >> 8);  // Address bits 15-8
65      g_tx433.framedata[3] = (uint8_t)(SENSOR_ADDRESS);       // Address bits  7-0
66      for(i=4; i < TX433_FRAME_BYTES; i++)
67          g_tx433.framedata[i] = 0x00;                        // Data & CRC = zero


70          // g_tx433.txdata[] contains the complete framedata, inclusive the SOF,
71          // coded in manchester.
72          // Also at the first position of the vector one byte is needed for the
73          // command "set_fifo", which tells to load the data into FIFO.

75      //___ TX Data: ____________________________________
76      g_tx433.txdata[0] = TX433_CMD_SET_FIFO;         // Command = Set_FIFO
77                                                      // SOF: -----------------------------
78      g_tx433.txdata[TX433_TX_SOF_BYTE0]   = 0xFF;    // RUN-IN-SEQ 1111 1111
79      g_tx433.txdata[TX433_TX_SOF_BYTE0+1] = 0x55;    // SYNCH-SEQ  0101 0101 --> "0000"
80      g_tx433.txdata[TX433_TX_SOF_BYTE0+2] = 0x55;    // SYNCH-SEQ  0101 0101 --> "0000"
81      g_tx433.txdata[TX433_TX_SOF_BYTE0+3] = 0x55;    // SYNCH-SEQ  0101 0101 --> "0000"
82      g_tx433.txdata[TX433_TX_SOF_BYTE0+4] = 0x56;    // SYNCH-SEQ  0101 0110 --> "0001"
83                                                      // Other bytes will be set by
84                                                      //  frame_code_manchester()
85  }

87  //--- Function: frame_tx433_code_manchester() ----------------------------------------
88  void frame_tx433_code_manchester(void) {

90      uint8_t byte_fr, byte_tx;
91      uint8_t shift_tx;
92      uint8_t shift_fr;

94      byte_tx = TX433_TX_PARA_BYTE0;                  // set startbyte after SOF bytes
95                                                      // run through bytes of framedata:
96      for(byte_fr = TX433_FR_PARA_BYTE0; byte_fr < TX433_FRAME_BYTES; byte_fr++) {
97          g_tx433.txdata[byte_tx] = 0x00;                 // clear databyte
98          shift_tx = 0x80;                                // shift to MSB first
99                                                          // run through 8-bit dataword:
100         for(shift_fr = 0x80; shift_fr > 0x00; shift_fr >>= 1) {
101             if(g_tx433.framedata[byte_fr] & shift_fr) {     // if value = 1 --> 1/0 trans
102                 g_tx433.txdata[byte_tx] |= shift_tx;        // shift in "10"
103             }
104             else {                                          // if value = 0 --> 0/1 trans
105                 g_tx433.txdata[byte_tx] |= (shift_tx >> 1); // shift in "01"
106             }
107             if(shift_fr == 0x10) {
108                 byte_tx++;                                  // goto next tx databyte
109                 shift_tx = 0x80;                            // shift to MSB first
110                 g_tx433.txdata[byte_tx] = 0x00;             // clear databyte
111             }
112             else {
113                 shift_tx >>= 2;                             // goto next pair
114             }
115         }
116         byte_tx++;                                      // goto next tx databyte
117     }
118 }
119 //--- Function: frame_tx433_calc_crc() -----------------------------------------------
120 void frame_tx433_calc_crc(void) {

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

125     uint8_t i;
```

```
127     g_tx433.framedata[TX433_FR_CRC_BYTE0] = 0x00;   // Start with generator polynom = 0

129     for(i=0; i<TX433_FRAME_BYTES-1; i++) {          // XOR bytewise with framedata
130         g_tx433.framedata[TX433_FR_CRC_BYTE0] ^= g_tx433.framedata[i];
131     }
132 }
133 //--- Function: frame_tx433_load_set1() ------------------------------------------------
134 void frame_tx433_load_set1(void) {

136         // This function loads the measured 16-bit datawords of the
137         //  - Temperature value
138         //  - Cell-voltage value
139         //  - Supply-voltage value
140         // into the frame, each word fragmented in 2 single bytes.
141         // Bit 8 of paramaters = '0' indicates that standard set of data is loaded.

143     g_tx433.framedata[TX433_FR_DATA1_BYTE0]   = (uint8_t)(g_temp.temp_value >> 8);
144     g_tx433.framedata[TX433_FR_DATA1_BYTE0+1] = (uint8_t)(g_temp.temp_value);

146     g_tx433.framedata[TX433_FR_DATA2_BYTE0]   = (uint8_t)(g_adc.u_bat >> 8);
147     g_tx433.framedata[TX433_FR_DATA2_BYTE0+1] = (uint8_t)(g_adc.u_bat);

149     g_tx433.framedata[TX433_FR_DATA3_BYTE0]   = (uint8_t)(g_adc.u_dcdc >> 8);
150     g_tx433.framedata[TX433_FR_DATA3_BYTE0+1] = (uint8_t)(g_adc.u_dcdc);

152     g_tx433.framedata[TX433_FR_PARA_BYTE0+1] &= ~TX433_PARA_B8;

154 }
155 //--- Function: frame_tx433_load_set2() ------------------------------------------------
156 void frame_tx433_load_set2(void) {

158         // This function loads the measured 16-bit datawords of the
159         //  - Lowpass-voltage value
160         //  - Cell-voltage value
161         //  - Activator-voltage value
162         // into the frame, each word fragmented in 2 single bytes.
163         // Bit 8 of paramaters = '1' indicates now that another set of data is loaded.

165     g_tx433.framedata[TX433_FR_DATA1_BYTE0]   = (uint8_t)(g_adc.u_lp >> 8);
166     g_tx433.framedata[TX433_FR_DATA1_BYTE0+1] = (uint8_t)(g_adc.u_lp);

168     g_tx433.framedata[TX433_FR_DATA2_BYTE0]   = (uint8_t)(g_adc.u_bat >> 8);
169     g_tx433.framedata[TX433_FR_DATA2_BYTE0+1] = (uint8_t)(g_adc.u_bat);

171     g_tx433.framedata[TX433_FR_DATA3_BYTE0]   = (uint8_t)(g_adc.u_act_in >> 8);
172     g_tx433.framedata[TX433_FR_DATA3_BYTE0+1] = (uint8_t)(g_adc.u_act_in);

174     g_tx433.framedata[TX433_FR_PARA_BYTE0+1] |= TX433_PARA_B8;

176 }
177 //--- Function: frame_tx433_load_reply() -----------------------------------------------
178 void frame_tx433_load_reply(void) {

180         // This function loads the last two commands and parameters
181         // into the frame, each word fragmented in 2 single bytes.
182         //  [CMD #-1 | CMD #-2 | PARA #-1 | PARA #-2]
183         // Bit 8 of paramaters = '1' indicates now that another set of data is loaded.
184         // Bit 9 of parameters = '1' indicates that this data is form the reply-cmd.

186     g_tx433.framedata[TX433_FR_DATA1_BYTE0]   = g_rx.cmd[1];
187     g_tx433.framedata[TX433_FR_DATA1_BYTE0+1] = g_rx.cmd[2];
```

```
189     g_tx433.framedata[TX433_FR_DATA2_BYTE0]   = (uint8_t)(g_rx.parameter[1] >> 8);
190     g_tx433.framedata[TX433_FR_DATA2_BYTE0+1] = (uint8_t)(g_rx.parameter[1]);

192     g_tx433.framedata[TX433_FR_DATA3_BYTE0]   = (uint8_t)(g_rx.parameter[2] >> 8);
193     g_tx433.framedata[TX433_FR_DATA3_BYTE0+1] = (uint8_t)(g_rx.parameter[2]);

195     g_tx433.framedata[TX433_FR_PARA_BYTE0+1] |= (TX433_PARA_B8 | TX433_PARA_SCF);

197 }
198 //--- Function: generate_time_tx() ------------------------------------------------------
199 void generate_time_tx(void) {

201         // This function generates a pseudo random time for the TX time when the sensor
202         // is in scan-mode. This enables the basestation to detect existing sensors.
203         // - min value for time to TX = TIME_TO_TX_MIN
204         // - max value for time to TX = TIME_TO_TX_MIN +
205         //                              RANDOM_REG_MODULO * RANDOM_REG_SCALE
206         // - max value must be smaller then max compare reg length (16-bit)

208     random_reg <<= 1;                           // preform generating pseudo random time:
209     random_reg  |= (((random_reg >> 28) ^ (random_reg >> 19)) & 0x00000001);
210     random_reg  &= 0x0FFFFFFF;

212     g_timing.t_tx_pseudo  = random_reg % RANDOM_REG_MODULO; // max variation = 850 x 64
213     g_timing.t_tx_pseudo <<= RANDOM_REG_SCALE;              // scale = 2^X
214     g_timing.t_tx_pseudo += TIME_TO_TX_MIN;                 // add minimum time to tx
215     TBCCR2 = g_timing.t_tx_pseudo;                          // set compare reg for tx
216     g_timing.t_tx_tune = g_timing.t_tx_pseudo - TIME_FOR_TUNE;
217     TBCCR1 = g_timing.t_tx_tune;                            // set compare reg for tune
218 }
219 //-----------------------------------------------------------------------------------------
```

Programmausdruck C.6: *Zellensensor „frame_tx.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    21.03.2011
 |  Geändert am:    21.09.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Header-Datei für gloable Variablen
 |
 |  Funktion:       OK
 |  Datei:          globals.h
 ================================================================================================
*/

#ifndef GLOABLS_H_
#define GLOBALS_H_

    #ifndef EXTERN
        #define EXTERN extern
    #endif

    //--- Definitionen: ------------------------------------------
    EXTERN volatile states      g_states;
    EXTERN volatile system_mode g_mode;

    EXTERN volatile uint32_t random_reg;

    struct {
        sample_set  tosample;
        uint16_t    u_bat;
        uint16_t    u_dcdc;
        uint16_t    u_act_in;
        uint16_t    u_lp;
    } EXTERN volatile g_adc;

    struct {
        uint16_t    temp_value;
        uint16_t    config_value;
    } EXTERN volatile g_temp;

    struct {
        uint8_t     return_status;
        uint8_t     irq_status;
        uint8_t     state;
        uint8_t     idlemode;
        uint16_t    acttxpktsize;
        uint8_t     actdatatsize;
        uint8_t     cts;
        uint8_t     error;
        uint8_t     prverror;
        uint8_t     framedata[TX433_FRAME_BYTES];
        uint8_t     txdata[TX433_FRAME_BYTES_MCH+1];    // + 1Byte for I2C Command
    } EXTERN volatile g_tx433;

    struct {
        uint8_t     cnt;
        uint8_t     completed;
        uint8_t     slave;
        uint8_t     rx_byte_ctr;
        uint16_t    rx_halfword;
    } EXTERN volatile g_i2c;
```

```
64      struct {
65          uint8_t   data[RX13P56_FRAME_BYTES];
66          uint8_t   crc;
67          uint8_t   cmd[3];
68          uint16_t  parameter[3];
69          uint32_t  address;
70          rx_states state;
71          rx_status status;
72      } EXTERN volatile g_rx;

74      struct {
75          uint16_t  capture;
76          uint16_t  bit_tavg;
77          uint16_t  bit_10_tmin;
78          uint16_t  bit_10_tmax;
79          uint16_t  bit_20_tmax;
80          uint16_t  bit_15_tmax;
81          uint8_t   bitctr;
82          uint8_t   bytectr;
83          uint8_t   lastbit;
84      } EXTERN volatile g_rx_timing;

86      struct {
87          uint8_t   time_to_sample;
88          uint8_t   time_to_tx;
89          uint8_t   time_to_tx_tune;
90          uint8_t   time_to_tx_stby;
91          uint16_t  t_interval;
92          uint16_t  t_tx;
93          uint16_t  t_tx_pseudo;
94          uint16_t  t_tx_tune;
95          uint16_t  t_sample;
96          uint8_t   tx_packets;
97      } EXTERN volatile g_timing;

99      struct {
100         uint8_t *ptr;
101         uint8_t valid;
102     } EXTERN volatile g_flash;

104     #ifdef RX13P56_ERROR_LOG
105         struct {
106             uint8_t   wrong_run_in;
107             uint8_t   synch_error;
108             uint8_t   bit_error;
109             uint8_t   timeout;
110         } EXTERN volatile g_rx_error;
111     #endif

113 #endif
114 //-------------------------------------------------------------------------------------
```

Programmausdruck C.7: *Zellensensor „globals.h“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    29.03.2011
 |  Geändert am:    11.06.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Funktionen für die Kommunikation auf dem I2C(SM)-BUS
 |
 |  Funktion:       OK
 |  Datei:          i2c_bus.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"
#include "src/i2c_bus.h"

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

    uint8_t i;

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

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

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

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

    uint8_t i;

    if((UCB0CTL1 & UCSWRST) != UCSWRST) {           // when I2C is activ
        return EXIT_FAILURE;                        //  then exit
```

```
63      }
64      UCB0CTL1 &= ~UCTR;                              // reset transmitter-mode
65      UCB0I2CSA = div_address;                        // set slave address of sensor
66      UCB0CTL1 &= ~UCSWRST;                           // unset SWRESET
67      UCB0CTL1 |= UCTXSTT;                            // send START-CON

69      if(data_length > 1) {
70          for(i = 0; i < data_length; i++) {
71              while(TRUE) {                           // wait until ...
72                  if((UCB0STAT & UCNACKIFG) == UCNACKIFG){    // NACK from slave
73                      UCB0CTL1 |= UCTXSTP;                //  then send STOP-CON
74                      UCB0STAT &= ~UCNACKIFG;             //  reset flag
75                      UCB0CTL1 |= UCSWRST;                //  set SWRESET
76                      return EXIT_FAILURE;                //  and exit
77                  }
78                  if((IFG2 & UCB0RXIFG) == UCB0RXIFG) {       // data / start-con was send
79                      data[i] = UCB0RXBUF;            //  readout data
80                      if(i == data_length-2)          //  if next byte will be the last one
81                          UCB0CTL1 |= UCTXSTP;        //  send STOP-CON after next receive
82                      break;
83                  }
84              }
85          }
86      }
87      else {
88          while((UCB0CTL1 & UCTXSTT) == UCTXSTT);     // wait for acknowledge of slave,
89          UCB0CTL1 |= UCTXSTP;                        // then send STOP-COND immediately
90          if((UCB0STAT & UCNACKIFG) == UCNACKIFG) {   // if NACK from slave
91              UCB0CTL1 |= UCSWRST;                    //  set SWRESET
92              return EXIT_FAILURE;                    //  and exit
93          }
94          data[0] = UCB0RXBUF;                        // readout data
95      }

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

100     return EXIT_SUCCESS;
101 }
102 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.8: *Zellensensor „i2c_bus.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    06.04.2011
 |  Geändert am:    06.04.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Header-Datei für die Anwendung des I2C(SM)-Busses
 |
 |  Funktion:       OK
 |  Datei:          i2c_bus.h
 ================================================================================================
*/

#ifndef I2C_BUS_H_
#define I2C_BUS_H_

    //___ I2C Handling: ____________________________________
    #define I2C_TRY_RX      3

    //--- Prototyps: ------------------------------------------------
    uint8_t i2c_bus_write(uint8_t div_address, uint8_t data_length, uint8_t *data);
    uint8_t i2c_bus_read(uint8_t div_address, uint8_t data_length, uint8_t *data);

#endif /* I2C_BUS_H_ */
//------------------------------------------------------------------------------------------------
```

Programmausdruck C.9: *Zellensensor „i2c_bus.h“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    28.06.2011
 |  Geändert am:    02.07.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Funktionen für das Schreiben und Lesen von Einstellungen in den
 |                  Information Memory des MSP
 |
 |  Funktion:       OK
 |  Datei:          infoflash.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"

//--- Function: infoflash_read() ---------------------------------------------------------
void infoflash_read(void) {
                                            // Init flash pointer to flash segment C:
    g_flash.ptr = (uint8_t *)INFOFLASH_SEG_C;
                                            // Read data from flash:
    g_flash.valid = *g_flash.ptr++;
    if(g_flash.valid == INFOFLASH_VALID) {
        g_timing.t_interval  = ((uint16_t)(*g_flash.ptr ++)) << 8;
        g_timing.t_interval |= ((uint16_t)(*g_flash.ptr ++));
        g_timing.t_sample    = ((uint16_t)(*g_flash.ptr ++)) << 8;
        g_timing.t_sample   |= ((uint16_t)(*g_flash.ptr ++));
```

```
31          g_timing.t_tx_tune   = ((uint16_t)(*g_flash.ptr ++)) << 8;
32          g_timing.t_tx_tune  |= ((uint16_t)(*g_flash.ptr ++));
33          g_timing.t_tx        = ((uint16_t)(*g_flash.ptr ++)) << 8;
34          g_timing.t_tx       |= ((uint16_t)(*g_flash.ptr ++));
35          g_adc.tosample       = *g_flash.ptr;
36      }
37  }
38  //--- Function: infoflash_write() -------------------------------------------------------
39  void infoflash_write(void) {
40                                              // Init flash pointer to flash segment C:
41      g_flash.ptr = (uint8_t *)INFOFLASH_SEG_C;
42      FCTL3 = FWKEY;                          // Clear lock bit
43      FCTL1 = (FWKEY | ERASE);                // Set erase bit
44      *g_flash.ptr = 0;                       // Erase flash segment
45      FCTL1 = (FWKEY | WRT);                  // Set write bit
46                                              // Write data to flash:
47      *g_flash.ptr ++ = INFOFLASH_VALID;
48      *g_flash.ptr ++ = (uint8_t)(g_timing.t_interval >> 8);
49      *g_flash.ptr ++ = (uint8_t)(g_timing.t_interval & 0x00FF);
50      *g_flash.ptr ++ = (uint8_t)(g_timing.t_sample >> 8);
51      *g_flash.ptr ++ = (uint8_t)(g_timing.t_sample & 0x00FF);
52      *g_flash.ptr ++ = (uint8_t)(g_timing.t_tx_tune >> 8);
53      *g_flash.ptr ++ = (uint8_t)(g_timing.t_tx_tune & 0x00FF);
54      *g_flash.ptr ++ = (uint8_t)(g_timing.t_tx >> 8);
55      *g_flash.ptr ++ = (uint8_t)(g_timing.t_tx & 0x00FF);
56      *g_flash.ptr    = g_adc.tosample;

58      FCTL1 = FWKEY;                          // Clear write bit
59      FCTL3 = (FWKEY | LOCK);                 // Set lock bit
60  }
61  //-----------------------------------------------------------------------------------------
```

Programmausdruck C.10: *Zellensensor „infoflash.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    21.03.2011
 |  Geändert am:    13.07.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Initialisierung der MCU
 |
 |  Funktion:       OK
 |  Datei:          init.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"

void init(void) {

    //--- Supply Voltage Supervisor ------------------------------------------
        // Voltage level detect = 3.05V, enable POR
    SVSCTL |= (VLD3 | VLD1 | PORON);
        // wait until settling delay time (~50us) of SVS elapsed
    while((SVSCTL & SVSON) != SVSON) nop();

    //--- Watchdogtimer ------------------------------------------------------
        // WDT off, watchdog mode, clock source = SMCLK, reset function
    WDTCTL = WDTPW | WDTHOLD;
        // NMI + WDT Timer Flags clear
    IFG1 &= ~(NMIIFG | WDTIFG);
        // NMI + WDT Timer IRQs off
    IE1  &= ~(NMIIE | WDTIE);
        // WDT on, WDT interval select = clock / 32768
    WDTCTL = (WDTPW);

    //--- DCO ----------------------------------------------------------------
        // DCO = cal value for 1.0MHz
    DCOCTL = CALDCO_1MHZ;
        // XT2 aus, RSEL = cal value for 1.0MHz
    BCSCTL1 = (XT2OFF | CALBC1_1MHZ);
        // MCLK = DCOCLK, div MCLK = X, SMCLK = DCOCLK, div SMCLK = X
    BCSCTL2 = (BCSCTL2_MCLK_DIV | BCSCTL2_SMCLK_DIV);

    //--- I/O Ports ----------------------------------------------------------
    //___ Port 1 _____________________________________
    P1OUT  = (0x04 | 0x08 | 0x20);              // OutputReg. = DC/DC ON, DC/DC PS OFF
                                                //              TX433 OFF
                                                // Input:  Pin 1,6,7
                                                // Output: Pin 0,2,3,4,5
    P1DIR  = (PIN0 | PIN2 | PIN3 | PIN4 | PIN5);
    P1REN  = 0x00;                              // Pullup disable
    P1SEL  = PIN1;                              // I/O Function for Pin 0,2,3,4,5,6,7
                                                // Sec Function for Pin 1
    P1IE   = P1IE_SET;                          // IRQ sources: Pin 6
    P1IFG  = 0x00;                              // IRQ Flags clear
    P1IES  = PIN6;                              // IRQ at rising edge: Pin 1,7
                                                //       falling edge: Pin 6
    //___ Port 2 _____________________________________
    P2OUT  = 0x00;                              // OutputReg. = LOW --> Load-Mod. off
                                                // Input:  Pin -
                                                // Output: Pin 0,1,2,3,4,5,6,7
```

```
63      P2DIR  = (PIN0 | PIN1 | PIN2 | PIN3 | PIN4 | PIN5 | PIN6 | PIN7);
64      P2REN  = 0x00;                              // Pullup disable
65      P2SEL  = 0x00;                              // I/O Function for all pins
66      P2IE   = P2IE_SET;                          // IRQ sources: SW IRQ on Pin 0,1,2,3
67      P2IFG  = 0x00;                              // IRQ Flags clear
68      P2IES  = 0x00;                              // IRQ at rising edge
69      //___ Port 3 _____________________________________
70      P3OUT  = 0x00;                              // OutputReg. = LOW --> Balance off
71                                                  // Input:  Pin -
72                                                  // Output: Pin 0,1,2,3,4,5,6,7
73      P3DIR  = (PIN0 | PIN1 | PIN2 | PIN3 | PIN4 | PIN5 | PIN6 | PIN7);
74      P3REN  = 0x00;                              // Pullup disable
75      P3SEL  = (PIN1 | PIN2);                     // I/O Function for Pin 0,3,4,5,6,7
76                                                  // Sec Function for Pin 1,2
77      //___ Port 4 _____________________________________
78      P4OUT  = 0x00;                              // OutputReg. = LOW --> LED1,2 OFF
79                                                  // Input:  Pin -
80                                                  // Output: Pin 0,1,2,3,4,5,6,7
81      P4DIR  = (PIN0 | PIN1 | PIN2 | PIN3 | PIN4 | PIN5 | PIN6 | PIN7);
82      P4REN  = 0x00;                              // Pullup disable
83      P4SEL  = 0x00;                              // I/O Function for all pins
84      //___ Port 5 _____________________________________
85      P5OUT  = 0x00;                              // OutputReg. = LOW
86                                                  // Input:  Pin -
87                                                  // Output: Pin 0,1,2,3,4,5,6,7
88      P5DIR  = (PIN0 | PIN1 | PIN2 | PIN3 | PIN4 | PIN5 | PIN6 | PIN7);
89      P5REN  = 0x00;                              // Pullup disable
90                              // SMCLK an PIN5
91      P5SEL  = PIN5;                              // I/O Function for all pins
92      //___ Port 6 _____________________________________
93      P6OUT  = 0x00;                              // OutputReg. = LOW
94                                                  // Input:  Pin 0,1,3,4
95                                                  // Output: Pin 2,5,6,7
96      P6DIR  = (PIN2 | PIN5 | PIN6 | PIN7);
97      P6REN  = 0x00;                              // Pullup disable
98      P6SEL  = (PIN0 | PIN1 | PIN3 | PIN4);       // I/O Function for Pin 2,5,6,7
99                                                  // Sec Function for Pin 0,1,3,4

101     //--- Timer A ------------------------------------------------------------
102         // timer clear
103     TACTL    = TACLR;
104         // clock source = SMCLK, div = 1, stop-mode
105     TACTL   |= TASSEL1;
106         // Capture-mode, rising-edge, CCI0A, SCS
107     TACCTL0 |= (CM0 | SCS | CAP);
108         // Compare-mode, IRQ enable
109     TACCTL1 |= CCIE;
110         // Capture/Compare Reg 1 set to RUN IN timeout
111     TACCR1   = RX13P56_RUN_IN_TIMEOUT;
112         // CC2 unused

114     //--- Timer B ------------------------------------------------------------
115         // timer clear
116     TBCTL    = TACLR;
117         // no grouping, counter length = 16bit, clock source = SMCLK, div = 8
118     TBCTL   |= (TBSSEL1 | ID1 | ID0);
119         // Capture/Compare Reg 0 set
120     TBCCR0   = TIME_INTERVAL;
121         // Compare-mode, IRQ enable
122     TBCCTL0 |= CCIE;
123         // Capture/Compare Reg 1 set
124     TBCCR1   = TIME_TO_SAMPLE;
125         // Compare-mode, IRQ enable
```

```
126     TBCCTL1 |= CCIE;
127         // Capture/Compare Reg 2 set
128     TBCCR2   = TIME_TO_TX;
129         // Compare-mode, IRQ enable
130     TBCCTL2 |= CCIE;

132     //--- ADC12 ---------------------------------------------------------------
133         // ref.voltage = 2.5V & on, adc on, no irq's
134     ADC12CTL0 = (REF2_5V | REFON | ADC12ON);
135         // S&H source = adc12SC bit, ADC12CLK div = 1, ADC12CLK = MCLK,
136         // single channel, single conversion
137     ADC12CTL1 |= (ADC12SSEL1 | SHP);

139     ADC12MCTL0 = (SREF_1 | INCH_0);     // VR+ = VREF+, VR- = AVss, channel = A0
140     ADC12MCTL1 = (SREF_1 | INCH_1);     // VR+ = VREF+, VR- = AVss, channel = A1
141     ADC12MCTL2 = (SREF_1 | INCH_3);     // VR+ = VREF+, VR- = AVss, channel = A3
142     ADC12MCTL3 = (SREF_1 | INCH_4);     // VR+ = VREF+, VR- = AVss, channel = A4

144     //--- USCI_A0 -------------------------------------------------------------
145         // not used

147     //--- USCI_B0 -------------------------------------------------------------
148         // first software reset
149     UCB0CTL1 |= UCSWRST;
150         // Own adress 7-bit, slave adress 7-bit, single master, I2C-mode, synch mode
151     UCB0CTL0  = (UCMST | UCMODE1 | UCMODE0 | UCSYNC);
152         // Clock source = SMCLK
153     UCB0CTL1  = (UCSSEL1 | UCSWRST);
154         // Baud rate = 100kbps, f_bitclk = f_brclk / UCB0RX = 100kHz,
155         //  f_brclk = SMCLK = 0.5MHz
156         //  --> UCB0RX = 5
157     UCB0BR0 = UCB0BR_0_SET;     // low byte, UCB0RX = (UCB0R0 + UCB0R1 x 128)
158     UCB0BR1 = 0x00;             // high byte
159         // IRQ Flags UCNACKIFG, UCSTPIFG, UCSTTIFG, UCALIFG clears by sw-reset
160         // I2C own 7-bit address = 0x7B (123)
161     UCB0I2COA |= 0x007B;
162         // no IRQs enabled for NACK, STOP, START, ARB
163     UCB0I2CIE = 0x00;

165     //--- FLASH ---------------------------------------------------------------
166         // flash controller clock source = MCLK,
167         // clock div = 3 --> 333.33kHz (must be 257kHz to 476kHz)
168     FCTL2 = (FWKEY | FSSEL0 | FN1);

170     //--- Comparator A --------------------------------------------------------
171         // not used

173     //--- OP A ----------------------------------------------------------------
174         // not used
175 }
176 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.11: *Zellensensor „init.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    23.03.2011
6   |  Geändert am:    11.06.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Interrupt Service Routine des Port 1
10  |
11  |  Funktion:       OK
12  |  Datei:          isr_port_1.c
13   ================================================================================================
14  */

16  #include "header_main.h"
17  #include "src/tx_433.h"
18  #include "src/globals.h"

20  //--- ISR Port 1 -------------------------------------------------------------------------
21  interrupt ( PORT1_VECTOR ) isr_port1 (void) {

23      uint8_t check;
24      // === if Flag for Pin6 (TX_NIRQ) ========================================================
25      if((P1IFG & PIN6) == PIN6) {
26          TP13_ON;
27          P1IE   = 0x00;                              // disable current IRQ source
28          P1IFG &= ~BIT6;                             // Flag Pin6 clear
29          eint();                                     // Nested IRQ's enable --------------

31          WDTCTL = WDTCTL_CLR;                        // WDT counter clear
32          check = tx433_cmd_get_int_status();         // readout IRQ status
33          if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
34              WDTCTL = WDTCTL_PUC;
35          }                                           // When IRQ for Packet Sent:
36          if((g_tx433.irq_status & TX433_IRQ_PACKET_SENT) == TX433_IRQ_PACKET_SENT) {
37              #ifdef LEDS_ENABLE
38                  tx433_cmd_led_ctrl(FALSE);          // LED3 off
39              #endif
40              switch(g_mode) {
41                  case NORMAL:                        // Normal-Mode: ----------------------
42                      g_states = WAIT_BEFOR_SAMPLE;       // state --> Wait bevor Sample
43                      break;
44                  case SCAN:                          // Scan-Mode: ------------------------
45                      g_timing.tx_packets --;             // decrement packet counter
46                      if(g_timing.tx_packets == 0x00) {   // all allowed packets send?
47                          TB_STOP_CLR;                        // stop & reset timer b
48                          g_mode   = NORMAL;
49                          g_states = IDLE;                    // state --> Idle
50                      }
51                      else {                              // send more packets
52                          g_states = WAIT_BEFOR_PREPARE_TX;   // state --> Wait b. Prepare
53                      }
54                      break;
55                  case REPLY:                         // Reply-Mode: ----------------------
56                      TB_STOP_CLR;                        // stop & reset timer b
57                      g_mode   = NORMAL;
58                      g_states = IDLE;                    // state --> Idle
59                      break;
60                  default:                            // Default: -------------------------
61                      g_states = IDLE;
62                      break;
```

```
63              }
64              DCDC_PS_ON;                             // Set DC/DC PS-Mode on
65              SVSCTL   &= ~SVSFG;                     // SVS clear flag
66              SVSCTL   |= PORON;                      // SVS enable gen POR
67              TA0CCR0  &= ~CCIFG;                     // Clear IRQ flag (RX timer)
68              TA0CCTL0 &= ~TAIFG;                     // Clear IRQ flag (RX timer)
69              TA0CCTL0 |= CCIE;                       // Enable RX
70          }
71          dint();                                     // Nested IRQ's disable -------------
72          P1IE = P1IE_SET;                            // enable current IRQ source
73          TP13_OFF;
74      }
75      // === if Flag for Pin7 (TMP_ALERT) ================================================
76      if((P1IFG & PIN7) == PIN7) {
77          P1IFG &= ~PIN7;                             // Flag Pin7 clear
78      }
79  }
80  //-------------------------------------------------------------------------------------
```

Programmausdruck C.12: *Zellensensor „isr_port_1.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    23.03.2011
6   |  Geändert am:    20.06.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Interrupt Service Routine des Port 2
10  |
11  |  Funktion:       OK
12  |  Datei:          isr_port_2.c
13   ================================================================================================
14  */

16  #include "header_main.h"
17  #include "src/adc.h"
18  #include "src/temp_sensor.h"
19  #include "src/tx_433.h"
20  #include "src/globals.h"

22  //--- ISR Port 2 ---------------------------------------------------------------------------
23  interrupt ( PORT2_VECTOR ) isr_port2 (void) {

25      uint8_t check;

27      switch(P2IFG) {
28      // === if Flag for Pin0 (SW IRQ for Sampling) ==========================================
29      case SW_IRQ_SAMPLE:
30                                      // Runtime with dataset = 1: ~1.1ms @1MHz
31                                      // Runtime with dataset = 2: ~3.6ms @1MHz
32          TP11_ON;
33          P2IE   = 0x00;                              // disable current IRQ source
34          P2IFG &= ~SW_IRQ_SAMPLE;                    // Flag clear
35          g_states = SAMPLE;                          // state --> Sample
36          WDTCTL = WDTCTL_CLR;                        // WDT counter clear
37          LED1_OFF;                                   // LED1 off
38          DCDC_OFF;                                   // Set DC/DC inactiv
39          SVSCTL &= ~PORON;                           // SVS does not gen POR

41          sample_ubat();                              // Get voltage U_BAT
42          if(g_adc.tosample == SAMPLE_SET1) {
43              sample_udcdc();                         // Get voltage U_DCDC
44              DCDC_ON;                                // Set DC/DC activ
45              eint();                                 // Nested IRQ's enable --------------
46              check = temp_sensor_get_temp();         // Get actual temperatur
47              if(check != EXIT_SUCCESS) {             // Exit Failure --> Restart
48                  WDTCTL = WDTCTL_PUC;
49              }
50              frame_tx433_load_set1();                // Load measured values into frame
51          }
52          else {
53              DCDC_ON;                                // Set DC/DC activ
54              sample_uact_in();                       // Get voltage U_ACT_IN
55              sample_ulp();                           // Get voltage U_LP
56              eint();                                 // Nested IRQ's enable --------------
57              frame_tx433_load_set2();                // Load measured values into frame
58          }
59          g_timing.t_tx_tune = g_timing.t_tx - TIME_FOR_TUNE;
60          SVSCTL &= ~SVSFG;                           // SVS clear flag
61          SVSCTL |= PORON;                            // SVS reenable gen POR
62          LED1_ON;                                    // LED1 on
```

```
63          g_states = WAIT_BEFOR_PREPARE_TX;           // state --> Wait b. Prepare TX
64          g_timing.time_to_sample = FALSE;            // Reset flag

66          dint();                                     // Nested IRQ's disable ------------
67          P2IE = P2IE_SET;                            // enable current IRQ source
68          TP11_OFF;
69          break;
70      // === if Flag for Pin1 (SW IRQ for TX) ============================================
71      case SW_IRQ_TX:
72                                      // Runtime to TX on 433MHz ~ 1.6ms @ 1MHz
73          TP12_ON;
74          P2IE   = 0x00;                              // disable current IRQ source
75          P2IFG &= ~SW_IRQ_TX;                        // Flag clear
76          eint();                                     // Nested IRQ's enable --------------

78          g_states = TX_0_WAIT_RX;                    // state --> Wait until RX end
79          WDTCTL = WDTCTL_CLR;                        // WDT counter clear
80          while(TRUE) {                               // Wait until:
81              if(g_rx.status != ACTIVE){              //  RX process is not active
82                  TACCTL0 &= ~CCIE;                   //  then Disable RX
83                  break;
84              }
85          }
86          g_states = TX_1_START;                      // state --> TX 1 Start
87          DCDC_PS_OFF;                                // Set DC/DC PS-Mode off
88          SVSCTL  &= ~PORON;                          // SVS does not gen POR
89          g_tx433.state    = TX433_STATE_IDLE;        // --> idle state
90  //      g_tx433.idlemode = TX433_IDLEMODE_TUNE;     // --> 370us response to TX
91          g_tx433.idlemode = TX433_IDLEMODE_STANDBY;  // --> 6.6ms response to TX
92          check = tx433_cmd_tx_start();               // Start 433MHz TX
93          if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
94              WDTCTL = WDTCTL_PUC;
95          }
96          #ifdef LEDS_ENABLE
97              tx433_cmd_led_ctrl(TRUE);               // LED3 on
98          #endif
99          g_states = TX_2_WAIT_TX;                    // state --> TX 2 wait until TX end
100         g_timing.time_to_tx = FALSE;                // Reset flag

102         dint();                                     // Nested IRQ's disable ------------
103         P2IE = P2IE_SET;                            // enable current IRQ source
104         TP12_OFF;
105         break;
106     // === if Flag for Pin2 (SW IRQ for load txdata & set TX433 in TUNE-State) ==========
107     case SW_IRQ_TUNING:
108                                     // Runtime ~ 12.1ms @ 1MHz (normal-mode)
109                                     // Need to start ~20ms bevor TX
110                                     //  (runtime + 6.6ms time to tune for TX433)
111         TP11_ON;
112         P2IE   = 0x00;                              // disable current IRQ source
113         P2IFG &= ~SW_IRQ_TUNING;                    // Flag clear
114         eint();                                     // Nested IRQ's enable --------------

116         g_states = PREPARE_TX;                      // state --> Prepare TX
117         WDTCTL = WDTCTL_CLR;                        // WDT counter clear
118         switch (g_mode) {
119             case NORMAL:                            // Normal-Mode: ----------------------
120                 break;
121             case SCAN:                              // Scan-Mode: ------------------------
122                 frame_tx433_init();                 // Initialize 433MHz TX Frame
123                 break;
124             case REPLY:                             // Reply-Mode: -----------------------
125                 frame_tx433_init();                 // Initialize 433MHz TX Frame
```

```
126                 frame_tx433_load_reply();           // Load last CMD & Para into TX Frame
127                 break;
128             case HC:                                // High-Current-Mode: ---------------
129                 break;
130             default:                                // Default: -------------------------
131                 break;
132         }
133         frame_tx433_calc_crc();                     // Generate CRC of framedata
134         frame_tx433_code_manchester();              // Code framedata into manchester

136         check = tx433_cmd_fifo_init();              // Init FIFO
137         if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
138             WDTCTL = WDTCTL_PUC;
139         }
140         check = tx433_cmd_fifo_set();               // Load data in FIFO for next TX
141         if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
142             WDTCTL = WDTCTL_PUC;
143         }
144         g_tx433.idlemode = TX433_IDLEMODE_TUNE;     // --> 370us response to TX
145         check = tx433_cmd_change_state();           // Set State on TX433
146         if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
147             WDTCTL = WDTCTL_PUC;
148         }
149         g_states = WAIT_BEFOR_TX;                   // state --> Wait bevor TX
150         g_timing.time_to_tx_tune = FALSE;           // Reset flag

152         dint();                                     // Nested IRQ's disable -------------
153         P2IE = P2IE_SET;                            // enable current IRQ source
154         TP11_OFF;
155         break;
156     // === if Flag for Pin3 (SW IRQ for set TX433 in STBY-State) ========================
157     case SW_IRQ_STANDBY:
158         TP11_ON;
159         P2IE   = 0x00;                              // disable current IRQ source
160         P2IFG &= ~SW_IRQ_STANDBY;                   // Flag clear
161         eint();                                     // Nested IRQ's enable --------------

163         WDTCTL = WDTCTL_CLR;                        // WDT counter clear
164         g_tx433.idlemode = TX433_IDLEMODE_STANDBY;  // --> 6.6ms response to TX
165         check = tx433_cmd_change_state();           // Set State on TX433
166         if(check != EXIT_SUCCESS) {                 // Exit Failure --> Restart
167             WDTCTL = WDTCTL_PUC;
168         }
169         g_timing.time_to_tx_stby = FALSE;           // Reset flag

171         dint();                                     // Nested IRQ's disable -------------
172         P2IE = P2IE_SET;                            // enable current IRQ source
173         TP11_OFF;
174         break;
175     // === default ======================================================================
176     default:
177         break;
178     }
179 }
180 //-------------------------------------------------------------------------------------
```

Programmausdruck C.13: *Zellensensor „isr_port_2.c“*

```
/*
 ===============================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    23.03.2011
 |  Geändert am:    20.06.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Interrupt Service Routinen der Timer A
 |
 |  Funktion:       OK
 |  Datei:          isr_timer_a.c
 ===============================================================================================

 -----------------------------------------------------------------------------------------------
    Frame for RX on 13.56MHz:

    |       SOF        |  Address   |  Command   | Parameter |   CRC     |
    |------------------|------------|------------|-----------|-----------|
    | RUN-IN    SYNCH  |            |            |           |           |
    |           8-Bit  | 20-Bit     | 8-Bit      | 12-Bit    | 8-Bit     |
    |------------------|------------|------------|-----------|-----------|
    |                  | 0       19 | 20     27  | 28     39 | 40     47 | -> 48 Bits
    |                  | 0        2 | 2       3  | 3       4 | 5         | -> 6  Bytes
    |------------------|------------|------------|-----------|-----------|    on 5.0kb/s
    | 800us     1.6ms  |    4ms     |   1.6ms    |   2.4ms   |   1.6ms   | -> 12.0ms

    Datarate = 5.0kb/s
    Transmissionrate = 10.0kb/s (Manchester)

    CRC-Checksum is generated over Address, Command and Data (bits 0 - 39)
 -----------------------------------------------------------------------------------------------*/

#include "header_main.h"
#include "src/globals.h"

//--- ISR Timer A Vector 0 ------------------------------------------------------------------
interrupt ( TIMERA0_VECTOR ) isr_timer_a0 (void) {

        // This Routine is executed every rising edge on CCI0A (RX_DEMOD),
        // to decode an incomming manchester coded datastream.

    TA_STOP_CLR;                                // stop & reset timer A
    TA_START;                                   // start timer A first/again
    g_rx_timing.capture = TA0CCR0;

    switch(g_rx.state) {
        case PREPARE:       // ___ PREPARE ____________________________________________
            g_rx_timing.bytectr  = 0x00;
            g_rx_timing.bitctr   = 0x00;
            g_rx_timing.bit_tavg = 0x0000;
            LED2_ON;
            g_rx.status = ACTIVE;
            g_rx.state  = RUN_IN;               // next state
            break;
        case RUN_IN:        // ___ RUN IN _____________________________________________
            if((g_rx_timing.capture > RX13P56_RUN_IN_MIN) &&
               (g_rx_timing.capture < RX13P56_RUN_IN_MAX)) {
                g_rx.state = SYNCH_0;           // next state -> Synch
            }
            else {
                TA_STOP_CLR;                        // stop & reset timer A
```

```
63                  #ifdef RX13P56_ERROR_LOG
64                      g_rx_error.wrong_run_in ++;
65                  #endif
66                  LED2_OFF;
67                  g_rx.status = RUN_IN_ERROR;
68                  g_rx.state  = PREPARE;          // next state -> back to Prepare
69              }
70              break;
71          case SYNCH_0:       // ___ SYNCH 0 ____________________________________________
72                                                  // calculate floating average:
73              g_rx_timing.bit_tavg += g_rx_timing.capture;
74              g_rx_timing.bit_tavg >>= 1;
75
76              g_rx_timing.bitctr ++;              // wait until last '0' detected
77              if(g_rx_timing.bitctr == (RX13P56_SYNCH_LENGTH-2)) {
78                  g_rx_timing.bitctr = 0x00;
79
80                  g_rx_timing.bit_10_tmin  =  g_rx_timing.bit_tavg -          // 1t
81                                              (g_rx_timing.bit_tavg >> 2);    // -25%
82                  g_rx_timing.bit_10_tmax  =  g_rx_timing.bit_tavg +          // 1t
83                                              (g_rx_timing.bit_tavg >> 2);    // +25%
84
85                  g_rx_timing.bit_20_tmax  = (g_rx_timing.bit_tavg << 1) +    // 1t * 2
86                                              (g_rx_timing.bit_tavg >> 1);    // +25%
87
88                  g_rx_timing.bit_15_tmax  = (g_rx_timing.bit_tavg << 1);     // 1t * 2
89                  g_rx_timing.bit_15_tmax += g_rx_timing.bit_tavg;            // + 1t = 3t
90                  g_rx_timing.bit_15_tmax >>= 1;                              // / 2 = 1.5t
91                  g_rx_timing.bit_15_tmax += (g_rx_timing.bit_15_tmax >> 2);  // +25%
92
93                  g_rx.state = SYNCH_1;           // next state
94              }
95              break;
96          case SYNCH_1:       // ___ SYNCH 1 ____________________________________________
97                                                  // manchester "01...011001" detected:
98              if((g_rx_timing.capture > g_rx_timing.bit_15_tmax) &&
99                 (g_rx_timing.capture < g_rx_timing.bit_20_tmax)) {
100                 g_rx.data[0] = 0x00;                // first bit = 0 detected
101                 g_rx_timing.lastbit = 0x00;
102
103             }                                   // manchester "01...011010" detected:
104             else if((g_rx_timing.capture > g_rx_timing.bit_10_tmax) &&
105                     (g_rx_timing.capture < g_rx_timing.bit_15_tmax)) {
106                 g_rx.data[0] = 0x01;                // first bit = 1 detected
107                 g_rx_timing.lastbit = 0x01;
108             }
109             else {                              // incorrect timing:
110                 TA_STOP_CLR;                        // stop & reset timer A
111                 #ifdef RX13P56_ERROR_LOG
112                     g_rx_error.synch_error ++;
113                 #endif
114                 LED2_OFF;
115                 g_rx.status = SYNCH_ERROR;
116                 g_rx.state  = PREPARE;              // next state -> back to Prepare
117                 break;                              // exit immediatly
118             }
119             g_rx_timing.bitctr = 0x01;          // set bit counter to 1
120             g_rx.state = DATALOAD;              // next state
121             break;
122         case DATALOAD:      // ___ DATALOAD ___________________________________________
123
124             g_rx.data[g_rx_timing.bytectr] <<= 1;   // set bit position
```

```
126                                     // 1.0 bit length detected: -----------------
127             if((g_rx_timing.capture < g_rx_timing.bit_10_tmax) &&
128                (g_rx_timing.capture > g_rx_timing.bit_10_tmin)) {
129                                     // --> sequenze (X 10 10 10 X / X 01 01 01 X)
130                                     // --> bit = last bit
131                 g_rx.data[g_rx_timing.bytectr] |= g_rx_timing.lastbit;
132             }                       // 1.5 bit length detected: -----------------
133             else if((g_rx_timing.capture < g_rx_timing.bit_15_tmax) &&
134                     (g_rx_timing.capture > g_rx_timing.bit_10_tmax)) {
135                                     // --> sequenze (X 10 10 01 X / X 01 10 10 X)
136                                     // --> bits = 10 or 1
137                 if(g_rx_timing.lastbit == 0x01) {
138                     g_rx.data[g_rx_timing.bytectr] |= 0x01;
139                     g_rx_timing.bitctr ++;
140                     g_rx_timing.bytectr = g_rx_timing.bitctr >> 3;
141                     g_rx.data[g_rx_timing.bytectr] <<= 1;
142                     g_rx_timing.lastbit = 0x00;
143                 }
144                 else {
145                     g_rx.data[g_rx_timing.bytectr] |= 0x01;
146                     g_rx_timing.lastbit = 0x01;
147                 }
148             }                       // 2.0 bit length detected: -----------------
149             else if((g_rx_timing.capture < g_rx_timing.bit_20_tmax) &&
150                     (g_rx_timing.capture > g_rx_timing.bit_15_tmax)) {
151                                     // --> sequenze (X 01 10 01 X)
152                                     // --> bits = 10
153                 g_rx.data[g_rx_timing.bytectr] |= 0x01;
154                 g_rx_timing.bitctr ++;
155                 g_rx_timing.bytectr = g_rx_timing.bitctr >> 3;
156                 g_rx.data[g_rx_timing.bytectr] <<= 1;
157                 g_rx_timing.lastbit = 0x00;
158             }                       // incorrect length detected: ---------------
159             else {
160                 TA_STOP_CLR;                    // stop & reset timer A
161                 #ifdef RX13P56_ERROR_LOG
162                     g_rx_error.bit_error ++;
163                 #endif
164                 LED2_OFF;
165                 g_rx.status = BIT_ERROR;
166                 g_rx.state  = PREPARE;          // next state -> back to Prepare
167             }
168             g_rx_timing.bitctr ++;                      // increment bit counter
169             g_rx_timing.bytectr = g_rx_timing.bitctr >> 3;  // set byte counter

171                                     // all bytes complete received: -------------
172             if(g_rx_timing.bytectr == RX13P56_FRAME_BYTES) {
173                 TA_STOP_CLR;                    // stop & reset timer A

175                 g_rx.crc = 0x00;                // Preform CRC check:
176                 for(g_rx_timing.bytectr = 0; g_rx_timing.bytectr < RX13P56_FRAME_BYTES;
177                     g_rx_timing.bytectr ++) {
178                     g_rx.crc ^= g_rx.data[g_rx_timing.bytectr];
179                 }
180                 if(g_rx.crc != 0x00) {          // if CRC fails:
181                     #ifdef RX13P56_ERROR_LOG
182                         g_rx_error.bit_error ++;
183                     #endif
184                     LED2_OFF;
185                     g_rx.status = CRC_FAILURE;
186                     g_rx.state  = PREPARE;          // next state -> back to Prepare
187                     break;                          // exit immediatly
188                 }
```

```
189                                             // if CRC ok, decode address:
190                 g_rx.address   = (uint32_t) g_rx.data[0] << 12;
191                 g_rx.address  |= (uint16_t) g_rx.data[1] << 4;
192                 g_rx.address  |= (g_rx.data[2] >> 4);

194                                             // Check if data is for this Sensor:
195                 if((g_rx.address == SENSOR_ADDRESS) ||
196                    (g_rx.address == SENSOR_GLOBAL_ADDRESS)) {

198                     g_rx.cmd[2] = g_rx.cmd[1];          // Store last commands & param.:
199                     g_rx.cmd[1] = g_rx.cmd[0];
200                     g_rx.parameter[2] = g_rx.parameter[1];
201                     g_rx.parameter[1] = g_rx.parameter[0];
202                                                         // Decode command & parameter:
203                     g_rx.cmd[0]        = (g_rx.data[2] << 4);
204                     g_rx.cmd[0]       |= (g_rx.data[3] >> 4);
205                     g_rx.parameter[0]  = (uint16_t) (g_rx.data[3] & 0x0F) << 8;
206                     g_rx.parameter[0] |= g_rx.data[4];

208                     switch (g_rx.cmd[0]) {              // Decode received Command: -----
209                         case ZS_CMD_STAY_ON:                //--------------------------
210                             DCDC_ON;
211                             TB_STOP_CLR;                    // stop & reset timer b
212                             if(g_timing.time_to_tx_stby == FALSE) {
213                                 g_timing.time_to_tx_stby = TRUE;
214                                 P2IFG |= SW_IRQ_STANDBY;    // Generate SW IRQ
215                             }
216                             g_mode = NORMAL;
217                             break;
218                         case ZS_CMD_TURN_OFF:               //--------------------------
219                             for(g_rx_timing.capture = 0; g_rx_timing.capture < 0x0F;
220                                 g_rx_timing.capture++) nop();
221                             while(TRUE) DCDC_OFF;
222                             break;
223                         case ZS_CMD_BAL_ON:                 //--------------------------
224                             BALANCE_ON;
225                             break;
226                         case ZS_CMD_BAL_OFF:                //--------------------------
227                             BALANCE_OFF;
228                             break;
229                         case ZS_CMD_SAMPLE_S1:              //--------------------------
230                             g_adc.tosample = SAMPLE_SET1;
231                             break;
232                         case ZS_CMD_SAMPLE_S2:              //--------------------------
233                             g_adc.tosample = SAMPLE_SET2;
234                             break;
235                         case ZS_CMD_SET_SAMPLE_TIME:        //--------------------------
236                             TB_STOP_CLR;                    // stop & reset timer b
237                             g_timing.t_sample = g_rx.parameter[0] << 4;
238                             if(g_timing.time_to_tx_stby == FALSE) {
239                                 g_timing.time_to_tx_stby = TRUE;
240                                 P2IFG |= SW_IRQ_STANDBY;    // Generate SW IRQ
241                             }
242                             g_mode = NORMAL;
243                             break;
244                         case ZS_CMD_SET_TX_TIME:            //--------------------------
245                             TB_STOP_CLR;                    // stop & reset timer b
246                             g_timing.t_tx = g_rx.parameter[0] << 4;
247                             if(g_timing.t_tx < (g_timing.t_sample + TIME_SAMPLE_TO_TX)){
248                                 g_timing.t_tx = g_timing.t_sample + TIME_SAMPLE_TO_TX;
249                             }
250                             else if(g_timing.t_tx >(g_timing.t_interval-TIME_TX_TO_END)){
251                                 g_timing.t_tx = g_timing.t_interval - TIME_TX_TO_END;
```

```
252                             }
253                             g_timing.t_tx_tune = g_timing.t_tx - TIME_FOR_TUNE;
254                             if(g_timing.time_to_tx_stby == FALSE) {
255                                 g_timing.time_to_tx_stby = TRUE;
256                                 P2IFG |= SW_IRQ_STANDBY;    // Generate SW IRQ
257                             }
258                             g_mode = NORMAL;
259                             break;
260                         case ZS_CMD_SET_INTVAL_TIME:        //---------------------------
261                             TB_STOP_CLR;                    // stop & reset timer b
262                             g_timing.t_interval = g_rx.parameter[0] << 4;
263                             if(g_timing.time_to_tx_stby == FALSE) {
264                                 g_timing.time_to_tx_stby = TRUE;
265                                 P2IFG |= SW_IRQ_STANDBY;    // Generate SW IRQ
266                             }
267                             g_mode = NORMAL;
268                             break;
269                         case ZS_CMD_SYNCH:                  //---------------------------
270                             TB_STOP_CLR;                    // stop & reset timer b
271                             TBCCR0 = g_timing.t_interval;   // Set Intervall time
272                             TBCCR1 = g_timing.t_sample;     // Set Sample time
273                             TBCCR2 = g_timing.t_tx;         // Set TX time
274                             TB_START;                       // Start timer b (Sample & TX)
275                             TP10_TOG;
276                             g_mode = NORMAL;
277                             break;
278                         case ZS_CMD_SCAN:                   //---------------------------
279                             g_timing.tx_packets = g_rx.parameter[0];
280                             TB_STOP_CLR;                    // stop & reset timer b
281                             TBCCR0   = TIME_INTERVAL;       // Set Interval time
282                             generate_time_tx();             // generate TX time
283                             TB_START;                       // Start timer b (TX)
284                             TP10_TOG;
285                             g_mode = SCAN;
286                             break;
287                         case ZS_CMD_REPLY:                  //---------------------------
288                             TB_STOP_CLR;                    // stop & reset timer b
289                             g_timing.t_tx      = g_rx.parameter[0] << 4;
290                             g_timing.t_tx_tune = g_timing.t_tx - TIME_FOR_TUNE;
291                             TBCCR0 = TIME_INTERVAL;         // Set Interval time
292                             TBCCR1 = g_timing.t_tx_tune;    // Set Tune time
293                             TBCCR2 = g_timing.t_tx;         // Set TX time
294                             TB_START;                       // Start timer b (TX)
295                             TP10_TOG;
296                             g_mode = REPLY;
297                             break;
298                         case ZS_CMD_SAVE_SET:
299                             infoflash_write();              // Save Control Data to Flash
300                             break;
301                         default:
302                             break;
303                     }   // end of "switch (g_rx.cmd)"
304                 }       // end of "if(g_rx.address == SENSOR_ADDRESS)"
305                 LED2_OFF;
306                 g_rx.status = SUCCESS;              // Set status to success
307                 g_rx.state  = PREPARE;              // Allow next RX
308             }           // end of "if(g_rx_timing.bytectr == RX13P56_FRAME_BYTES)"
309             break;
310         case COMPLETE:      // ___ COMPLETE ____________________________________________
311             TA_STOP_CLR;                        // stop & reset timer A
312             break;
313     }
314                                                 // CCR0 CCIFG flag auto reset
```

```
315 }
316 //--- ISR Timer A Vector 1 -------------------------------------------------------------
317 interrupt ( TIMERA1_VECTOR ) isr_timer_a1 (void) {

319         // This Routine is executed after a defined time, above the length of the
320         // RUN IN time, to terminate a failed running data receive.

322     TA_STOP_CLR;                            // stop & reset timer A
323     #ifdef RX13P56_ERROR_LOG
324         g_rx_error.timeout ++;
325     #endif
326     LED2_OFF;
327     g_rx.status = TIMEOUT;
328     g_rx.state  = PREPARE;              // next state -> back to Prepare

330     TA0CCTL1 &= ~CCIFG;                 // reset IRQ flag
331 }
332 //-------------------------------------------------------------------------------------
```

Programmausdruck C.14: *Zellensensor „isr_timer_a.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    23.03.2011
6   |  Geändert am:    22.07.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Interrupt Service Routinen der Timer B
10  |
11  |  Funktion:       OK
12  |  Datei:          isr_timer_a.c
13   ================================================================================================
14  */

16  #include "header_main.h"
17  #include "src/globals.h"


20  //--- ISR Timer B Vector 0 ------------------------------------------------------------------
21  interrupt ( TIMERB0_VECTOR ) isr_timer_b0 (void) {

23          // ISR for Timer b running in up-mode. TCCR0 specifies counter end-value
24          // and time for init data transmission on 433MHz. Timer repeats automatic.

26      TP10_TOG;
27      if(g_mode == SCAN) {                            // Scan-Mode:
28          generate_time_tx();                         // generate pseudo time to tx
29      }
30                                                      // irq flag clears automatically
31  }
32  //--- ISR Timer B Vector 1 ------------------------------------------------------------------
33  interrupt ( TIMERB1_VECTOR ) isr_timer_b1 (void) {

35          // ISR for Timer b running in up-mode.
36          // - TCCR1 specifies time for init sampling of voltages and temperature.
37          // - TCCR1 also specifies time for load txdata into FIFO of the TX433 and
38          //   setting TX433 into Tune-Mode
39          // - TCCR2 specifies time for starting transmisson.

41      switch (TBIV) {
42          case TBIV_CCR1:             // --- Compare Reg.1:---------------------------------
43              switch(g_mode) {
44                  case NORMAL:                        // Normal-Mode: ----------------------
45                      if((g_states == WAIT_BEFOR_PREPARE_TX) &&
46                         (g_timing.time_to_tx_tune == FALSE)) {
47                                                      // Set flag for set Tune-Mode TX433
48                          g_timing.time_to_tx_tune = TRUE;
49                          P2IFG |= SW_IRQ_TUNING;     // Generate SW IRQ for tuning TX433
50                          TBCCR1 = g_timing.t_sample; // Load compare value for Sampling
51                      }
52                      else if((g_states != INIT) && (g_timing.time_to_sample == FALSE)) {
53                                                      // Set flag for taking sample
54                          g_timing.time_to_sample = TRUE;
55                          P2IFG |= SW_IRQ_SAMPLE;     // Generate SW IRQ for Sampling
56                          TBCCR1 = g_timing.t_tx_tune;// Load compare value for Tuning
57                      }
58                      break;
59                  case SCAN:                          // Scan-Mode: ------------------------
60                      if((g_states != TX_2_WAIT_TX) &&
61                         (g_timing.time_to_tx_tune == FALSE)) {
62                                                      // Set flag for set Tune-Mode TX433
```

```
63                          g_timing.time_to_tx_tune = TRUE;
64                          P2IFG |= SW_IRQ_TUNING;     // Generate SW IRQ for tuning TX433
65                      }
66                      break;
67                  case REPLY:                         // Reply-Mode: ----------------------
68                      if((g_states != TX_2_WAIT_TX) &&
69                         (g_timing.time_to_tx_tune == FALSE)) {
70                                                      // Set flag for set Tune-Mode TX433
71                          g_timing.time_to_tx_tune = TRUE;
72                          P2IFG |= SW_IRQ_TUNING;     // Generate SW IRQ for tuning TX433
73                      }
74                      break;
75                  default:                            // Default: -------------------------
76                      break;
77              }
78              break;                                  // irq flag clears automatically

80          case TBIV_CCR2:             // --- Compare Reg.2:--------------------------------
81              if((g_states != INIT) && (g_states == WAIT_BEFOR_TX) &&
82                 (g_timing.time_to_tx == FALSE)){
83                  g_timing.time_to_tx = TRUE;         // Set flag for start transmission
84                  P2IFG |= SW_IRQ_TX;                 // Generate SW IRQ for TX
85              }
86              else if(g_states == TX_2_WAIT_TX) {     // When last TX hasn't finished
87                  WDTCTL = WDTCTL_PUC;                //  Restart
88              }
89              break;                                  // irq flag clears automatically

91          case TBIV_OVERFLOW:         // --- Overflow:-------------------------------------
92              TB_STOP_CLR;                            // stop & reset timer b
93              break;
94      }
95  }
96  //-----------------------------------------------------------------------------------------
```

Programmausdruck C.15: *Zellensensor „isr_timer_b.c“*

```
1  /*
2   ================================================================================================
3   |  Projekt:        Masterarbeit, Zellensensor
4   |  Erstellt von:   Niels Jegenhorst
5   |  Erstellt am:    23.03.2011
6   |  Geändert am:    11.04.2011
7   |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
8   |  Tools:          MSPGCC - 20100218; Eclipse - Helios
9   |  Beschreibung:   Auslesen des externen Temperatursensors TI-TMP102
10  |
11  |  Funktion:       OK
12  |  Datei:          temp_sensor.c
13  |  zug. Dateien:   temp_sensor.h, i2c_bus.c
14   ================================================================================================
15 */

17 #include "header_main.h"
18 #include "src/globals.h"
19 #include "src/temp_sensor.h"
20 #include "src/i2c_bus.h"

22 //--- Function: temp_sensor_get_temp() -------------------------------------------------
23 uint8_t temp_sensor_init(void) {

25     uint8_t ok;

27     ok = temp_sensor_get_control_reg();             // Read current control register
28     if(ok != EXIT_SUCCESS)
29         return EXIT_FAILURE;

31 //  g_temp.config_value |= TS_CONF_CR0;
32 //  g_temp.config_value &= ~TS_CONF_CR1;

34 //  ok = temp_sensor_set_control_reg();             // Write control register
35 //  if(ok != EXIT_SUCCESS)
36 //      return EXIT_FAILURE;

38     return EXIT_SUCCESS;
39 }
40 //--- Function: temp_sensor_get_temp() -------------------------------------------------
41 uint8_t temp_sensor_get_temp(void) {

43         // pointer register of the temperatur sensor must be 0x00 for readout
44         // the temperatur register.

46         // When configurated as 12-bit:
47         //  One LSB equals 0.0625°C, negative numbers are represented in the binary
48         //  twos complement format.
49         //      20°C --> 320 counts
50         //      23°C --> 368 counts

52     uint8_t ok;
53     uint8_t data[2];

55     g_temp.temp_value = 0x0000;

57     ok = i2c_bus_read(ADDR_TEMP, 2, data);
58     if(ok == EXIT_SUCCESS) {
59         g_temp.temp_value  = (uint16_t) data[0] << 4;
60         g_temp.temp_value |= data[1] >> 4;
61         return EXIT_SUCCESS;
62     }
```

```
63      else {
64          g_temp.temp_value  = 0xFFFF;
65          return EXIT_FAILURE;
66      }
67  }
68  //--- Function: temp_sensor_get_control_reg() ------------------------------------------
69  uint8_t temp_sensor_get_control_reg(void) {

71      uint8_t ok;
72      uint8_t data[2];

74      data[0] = TS_CONF_REG;                      // Value for Pointer-Register
75                                                  //  --> Configuration-Register
76      ok = i2c_bus_write(ADDR_TEMP, 1, data);     // Set Pointer-Register
77      if(ok != EXIT_SUCCESS)
78          return EXIT_FAILURE;

80      ok = i2c_bus_read(ADDR_TEMP, 2, data);      // Read Control-Register
81          if(ok == EXIT_SUCCESS) {
82              g_temp.config_value  = (uint16_t) data[0] << 8;
83              g_temp.config_value |= data[1];
84          }
85          else
86              g_temp.config_value  = 0xFFFF;

88      data[0] = TS_TEMP_REG;                      // Value for Pointer-Register
89                                                  //  --> Temperature-Register
90      ok = i2c_bus_write(ADDR_TEMP, 1, data);     // Set Pointer-Register
91      if(ok != EXIT_SUCCESS)
92          return EXIT_FAILURE;


95      return EXIT_SUCCESS;
96  }
97  //--- Function: temp_sensor_set_control_reg() ------------------------------------------
98  uint8_t temp_sensor_set_control_reg(void) {

100     uint8_t ok;
101     uint8_t data[3];

103     data[0] = TS_CONF_REG;                              // Value for Pointer-Register
104     data[1] = (uint8_t) (g_temp.config_value >> 8);     // Byte 1 at first
105     data[2] = (uint8_t) (g_temp.config_value & 0x00FF); // Byte 2 at last

107     ok = i2c_bus_write(ADDR_TEMP, 3, data);     // Write Control-Register
108     if(ok != EXIT_SUCCESS)
109         return EXIT_FAILURE;

111     data[0] = TS_TEMP_REG;                      // Value for Pointer-Register
112                                                 //  --> Temperature-Register
113     ok = i2c_bus_write(ADDR_TEMP, 1, data);     // Set Pointer-Register
114     if(ok != EXIT_SUCCESS)
115         return EXIT_FAILURE;


118     return EXIT_SUCCESS;
119 }
120 //-----------------------------------------------------------------------------------------
```

Programmausdruck C.16: *Zellensensor „temp_sensor.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    06.04.2011
 |  Geändert am:    11.04.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Header-Datei für den externen Temperatursensors TI-TMP102
 |
 |  Funktion:       OK
 |  Datei:          temp_sensor.h
 ================================================================================================
*/

#ifndef TEMP_SENSOR_H_
#define TEMP_SENSOR_H_

    #define ADDR_TEMP       0x48                    // Address of temperatur-sensor
                                                    //  if TMP_ADD0 = 0
    //___ Pointer Register: ________________________
    #define TS_TEMP_REG     0x00                    // Pointer Reg --> Temperatur
    #define TS_CONF_REG     0x01                    // Pointer Reg --> Configuration
    #define TS_TLOW_REG     0x02                    // Pointer Reg --> Temperatur Low
    #define TS_THIGH_REG    0x04                    // Pointer Reg --> Temperatur High

    //___ Configuration Register: ____________________
    #define TS_CONF_OS      0x8000                  // One-Shot/Conversion Ready
    #define TS_CONF_R1      0x4000                  // Conversion Resolution (r)
    #define TS_CONF_R0      0x2000                  // Conversion Resolution (r)
    #define TS_CONF_F1      0x1000                  // Fault Queue
    #define TS_CONF_F0      0x0800                  // Fault Queue
    #define TS_CONF_POL     0x0400                  // Polarity
    #define TS_CONF_TM      0x0200                  // Thermostat Mode
    #define TS_CONF_SD      0x0100                  // Shutdown Mode
    #define TS_CONF_CR1     0x0080                  // Conversion Rate
    #define TS_CONF_CR0     0x0040                  // Conversion Rate
    #define TS_CONF_AL      0x0020                  // Alert
    #define TS_CONF_EM      0x0010                  // Extended Mode

    //--- Prototyps: ------------------------------------------------
    uint8_t temp_sensor_init(void);
    uint8_t temp_sensor_get_temp(void);
    uint8_t temp_sensor_get_control_reg(void);
    uint8_t temp_sensor_set_control_reg(void);

#endif /* TEMP_SENSOR_H_ */
//------------------------------------------------------------------------------------------------
```

Programmausdruck C.17: *Zellensensor „temp_sensor.h“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    06.03.2011
 |  Geändert am:    02.07.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Initialisierung des externen 433MHz Transmitters SI4012
 |
 |  Funktion:       OK
 |  Datei:          tx_433.c
 |  zug. Dateien:   tx_433.h, i2c_bus.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"
#include "src/tx_433.h"
#include "src/i2c_bus.h"

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

    uint8_t ok;

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

    ok = tx433_set_prop(TX433_PROP_CHIP_CONFIG);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_TUNE_INTERVAL);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_MODULATION_FSKDEV);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_TX_FREQUENCY);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_PA_CONFIG);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_BITRATE_CONFIG);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_set_prop(TX433_PROP_LED_INTENSITY);
    if(ok != EXIT_SUCCESS)
        return EXIT_FAILURE;

    ok = tx433_cmd_set_int();
    if(ok != EXIT_SUCCESS)
```

```
63          return EXIT_FAILURE;

65      ok = tx433_cmd_fifo_init();
66      if(ok != EXIT_SUCCESS)
67          return EXIT_FAILURE;

69      ok = tx433_cmd_led_ctrl(FALSE);                     // LED3 off
70      if(ok != EXIT_SUCCESS)
71          return EXIT_FAILURE;

73      return EXIT_SUCCESS;
74  }
75  //--- Function: tx433_set_prop() ------------------------------------------------------
76  uint8_t tx433_get_prop(uint8_t property) {

78      // not needed
79      return EXIT_SUCCESS;
80  }
81  //--- Function: tx433_set_prop() ------------------------------------------------------
82  uint8_t tx433_set_prop(uint8_t property) {

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

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

90                                                  // Spliting of the property data into
91                                                  // single bytes is replaced as constants
92                                                  // by the compiler.
93                                                  // data[2] <-- MSB of propertys data
94                                                  // data[n] <-- LSB, nmax = 7
95      switch(property) {
96          case TX433_PROP_CHIP_CONFIG:
97              bytes   = TX433_PROP_CHIP_CONFIG_BCNT;
98              data[2] = TX433_PROP_CHIP_CONFIG_DATA;
99              break;
100         case TX433_PROP_LED_INTENSITY:
101             bytes   = TX433_PROP_CHIP_CONFIG_BCNT;
102             data[2] = TX433_PROP_LED_INTENSITY_DATA;
103             break;
104         case TX433_PROP_MODULATION_FSKDEV:
105             bytes   = TX433_PROP_MODULATION_FSKDEV_BCNT;
106             data[2] = (uint8_t)(TX433_PROP_MODULATION_FSKDEV_DATA >> 8);
107             data[3] = (uint8_t)(TX433_PROP_MODULATION_FSKDEV_DATA & 0xFF);
108             break;
109         case TX433_PROP_TUNE_INTERVAL:
110             bytes   = TX433_PROP_TUNE_INTERVAL_BCNT;
111             data[2] = (uint8_t)(TX433_PROP_TUNE_INTERVAL_DATA >> 8);
112             data[3] = (uint8_t)(TX433_PROP_TUNE_INTERVAL_DATA & 0xFF);
113             break;
114         case TX433_PROP_FIFO_THRESHOLD:
115             bytes   = TX433_PROP_FIFO_THRESHOLD_BCNT;
116             data[2] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA >> 16);
117             data[3] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA >> 8);
118             data[4] = (uint8_t)(TX433_PROP_FIFO_THRESHOLD_DATA & 0xFF);
119             break;
120         case TX433_PROP_BITRATE_CONFIG:
121             bytes = TX433_PROP_BITRATE_CONFIG_BCNT;
122             data[2] = (uint8_t)((uint32_t)TX433_PROP_BITRATE_CONFIG_DATA >> 16);
123             data[3] = (uint8_t)(TX433_PROP_BITRATE_CONFIG_DATA >> 8);
124             data[4] = (uint8_t)(TX433_PROP_BITRATE_CONFIG_DATA & 0xFF);
125             break;
```

```
126         case TX433_PROP_TX_FREQUENCY:
127             bytes = TX433_PROP_TX_FREQUENCY_BCNT;
128             data[2] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 24);
129             data[3] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 16);
130             data[4] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA >> 8);
131             data[5] = (uint8_t)(TX433_PROP_TX_FREQUENCY_DATA & 0xFF);
132             break;
133         case TX433_PROP_LBD_CONFIG:
134             bytes = TX433_PROP_LBD_CONFIG_BCNT;
135             data[2] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 24);
136             data[3] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 16);
137             data[4] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA >> 8);
138             data[5] = (uint8_t)(TX433_PROP_LBD_CONFIG_DATA & 0xFF);
139             break;
140         case TX433_PROP_PA_CONFIG:
141             bytes = TX433_PROP_PA_CONFIG_BCNT;
142             data[2] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P2 >> 8);
143             data[3] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P2 & 0xFF);
144             data[4] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P1 >> 8);
145             data[5] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P1 & 0xFF);
146             data[6] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P0 >> 8);
147             data[7] = (uint8_t)(TX433_PROP_PA_CONFIG_DATA_P0 & 0xFF);
148             break;
149         default:
150             return EXIT_FAILURE;
151     }
152     bytes = bytes + 2;                              // add 2 for command and property

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

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

163     return EXIT_SUCCESS;
164 }
165 //--- Function: tx433_cmd_tx_start() ------------------------------------------------
166 uint8_t tx433_cmd_tx_start(void) {

168         // Time to transmit the Start Command over SMBUS ~ 1.14ms @ 100kbit/s
169         // Time to TX after received Start Command = 370us @ Idlemode = Tune
170         //                                         = 6.6ms @ Idlemode = Standby

172     uint8_t ok;
173     uint8_t data[6];

175     data[0] = TX433_CMD_TX_START;                       // Command = TX_START
176     data[1] = 0x00;                                     // PacketSize[15:8]
177     data[2] = 0x1B;                                     // PacketSize[ 7:0]
178                                                         //   = TX433_FRAME_BYTES_MCH = 27
179     data[3] = (0x02 & g_tx433.state);                   // Sensor State,
180                                                         //  FIFO Auto-TX disable
181     data[4] = g_tx433.idlemode;                         // Idle Mode
182 //  data[5] = TX433_DTMOD_CW;                           // DataTransmission Mode = CW
183     data[5] = TX433_DTMOD_FIFO;                         // DataTransmission Mode = FIFO

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

```
189     ok = i2c_bus_read(ADDR_TX433, 2, data);             // Read Reply
190     if(ok != EXIT_SUCCESS)
191         return EXIT_FAILURE;
192     g_tx433.return_status = data[0];
193     g_tx433.actdatatsize  = data[1];
194     if(g_tx433.return_status == TX433_RESTATUS_CTS_OK) {
195         return EXIT_SUCCESS;
196     }
197     else {
198         return EXIT_FAILURE;
199     }
200 }
201 //--- Function: tx433_cmd_tx_stop() ------------------------------------------------
202 uint8_t tx433_cmd_tx_stop(void) {

204     uint8_t ok;
205     uint8_t data[3];

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


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

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

221     return EXIT_SUCCESS;
222 }
223 //--- Function: tx433_cmd_led_ctrl() -----------------------------------------------
224 uint8_t tx433_cmd_led_ctrl(const uint8_t ledon) {

226     uint8_t ok;
227     uint8_t data[2];

229     data[0] = TX433_CMD_LED_CTRL;                       // Command = LED_Ctrl
230     data[1] = ledon;

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

236     return EXIT_SUCCESS;
237 }
238 //--- Function: tx433_cmd_fifo_init() ----------------------------------------------
239 uint8_t tx433_cmd_fifo_init(void) {

241     uint8_t ok;
242     uint8_t data[1];

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

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

250     ok = i2c_bus_read(ADDR_TX433, 1, data);             // Read Reply
251     if(ok != EXIT_SUCCESS)
```

```
252         return EXIT_FAILURE;
253     g_tx433.return_status = data[0];

255     return EXIT_SUCCESS;
256 }
257 //--- Function: tx433_cmd_fifo_set() ------------------------------------------------
258 uint8_t tx433_cmd_fifo_set(void) {

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

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

272     return EXIT_SUCCESS;
273 }
274 //--- Function: tx433_cmd_set_int() -------------------------------------------------
275 uint8_t tx433_cmd_set_int(void) {

277     uint8_t ok;
278     uint8_t data[2];

280     data[0] = TX433_CMD_SET_INT;                        // Command = SET_Interrupt
281     data[1] = TX433_IRQ_PACKET_SENT;                    // Set IRQ:
282                                                         //  - Packet Sent

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

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

293     return EXIT_SUCCESS;
294 }
295 //--- Function: tx433_cmd_set_int() -------------------------------------------------
296 uint8_t tx433_cmd_get_int_status(void) {

298     uint8_t ok;
299     uint8_t data[2];

301     data[0] = TX433_CMD_GET_INT_STATUS;                 // Command = GET_Interrupt_Status

303     ok = i2c_bus_write(ADDR_TX433, 1, data);            // Write Command
304     if(ok != EXIT_SUCCESS)
305         return EXIT_FAILURE;

307     ok = i2c_bus_read(ADDR_TX433, 2, data);             // Readout Response
308     if(ok == EXIT_SUCCESS) {
309         g_tx433.return_status = data[0];
310         g_tx433.irq_status = data[1];
311         return EXIT_SUCCESS;
312     }
313     else
314         return EXIT_FAILURE;
```

```
315 }
316 //--- Function: tx433_cmd_get_state() -------------------------------------------------
317 uint8_t tx433_cmd_get_state(void) {

319     uint8_t ok;
320     uint8_t data[6];

322     data[0] = TX433_CMD_GET_STATE;                      // Command = GET_State

324     ok = i2c_bus_write(ADDR_TX433, 1, data);            // Write Command
325     if(ok != EXIT_SUCCESS)
326         return EXIT_FAILURE;

328     ok = i2c_bus_read(ADDR_TX433, 6, data);             // Readout Response
329     if(ok == EXIT_SUCCESS) {
330         g_tx433.error = data[0] & 0x7F;
331         g_tx433.cts   = data[0] & 0x80;
332         g_tx433.state = data[1] & 0x03;
333         g_tx433.idlemode = data[2] & 0x07;              // if state = TX --> DTMode
334         g_tx433.acttxpktsize  = (uint16_t) data[3] << 8;
335         g_tx433.acttxpktsize |= data[4];
336         g_tx433.prverror = data[5];
337         return EXIT_SUCCESS;
338     }
339     else
340         return EXIT_FAILURE;
341 }
342 //--- Function: tx433_change_state() -------------------------------------------------
343 uint8_t tx433_cmd_change_state(void) {

345     uint8_t ok;
346     uint8_t data[3];

348     data[0] = TX433_CMD_CHANGE_STATE;                   // Command = Change_State
349     data[1] = g_tx433.state;
350     data[2] = g_tx433.idlemode;

352     ok = i2c_bus_write(ADDR_TX433, 3, data);            // Write Command
353     if(ok != EXIT_SUCCESS)
354         return EXIT_FAILURE;

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

361     return EXIT_SUCCESS;
362 }
363 //-----------------------------------------------------------------------------------------
```

Programmausdruck C.18: *Zellensensor „tx_433.c“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    06.04.2011
 |  Geändert am:    02.07.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Header-Datei für den externen 433MHz Transmitter SI4012
 |
 |  Funktion:       OK
 |  Datei:          tx_433.h
 ================================================================================================
*/

#ifndef TX_433_H_
#define TX_433_H_

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

    //___ Commands: ____________________________________
    #define TX433_CMD_GET_REV           0x10        // Return product and revision info
    #define TX433_CMD_SET_PROPERTY      0x11        // Set property
    #define TX433_CMD_GET_PROPERTY      0x12        // Get property
    #define TX433_CMD_LED_CTRL          0x13        // Turn LED on/off
    #define TX433_CMD_CHANGE_STATE      0x60        // Change power state
    #define TX433_CMD_GET_STATE         0x61        // Get state
    #define TX433_CMD_TX_START          0x62        // Start transmission
    #define TX433_CMD_TX_STOP           0x67        // Stop transmission
    #define TX433_CMD_SET_INT           0x63        // Interrupt control
    #define TX433_CMD_GET_INT_STATUS    0x64        // Get interrupt status
    #define TX433_CMD_INIT_FIFO         0x65        // Clear TX FIFO
    #define TX433_CMD_SET_FIFO          0x66        // Store data in TX FIFO
    #define TX433_CMD_GET_BAT_STATUS    0x68        // Get battery status (Vdd voltage)

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

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

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

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

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

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

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

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

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

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

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

93      //___ States & IdleModes: _____________________________
94      #define TX433_STATE_IDLE            0x00
95      #define TX433_STATE_SHUTDOWN        0x01
96      #define TX433_STATE_TX              0x10
97      #define TX433_IDLEMODE_STANDBY      0x00
98      #define TX433_IDLEMODE_SENSOR       0x01
99      #define TX433_IDLEMODE_TUNE         0x02

101     //___ DataTransmission Mode: __________________________
102     #define TX433_DTMOD_FIFO            0x00
103     #define TX433_DTMOD_CW              0x01
104     #define TX433_DTMOD_PN90            0x02
105     #define TX433_DTMOD_PN91            0x03

107     //___ Interrupts: _____________________________________
108     #define TX433_IRQ_FIFO_UFLOW        0x80
109     #define TX433_IRQ_FIFO_OFLOW        0x10
110     #define TX433_IRQ_FIFO_AEMTY        0x20
111     #define TX433_IRQ_FIFO_AFULL        0x40
112     #define TX433_IRQ_PACKET_SENT       0x08
113     #define TX433_IRQ_LOW_BAT           0x04
114     #define TX433_IRQ_TUNE              0x02
115     #define TX433_IRQ_POR               0x01

117     //___ Return Status: __________________________________
118     #define TX433_RESTATUS_CTS_OK       0x80

120     //--- Prototyps: ------------------------------------------------
121     uint8_t tx433_init(void);
122     uint8_t tx433_set_prop(const uint8_t);
123     uint8_t tx433_get_prop(const uint8_t);
124     uint8_t tx433_cmd_tx_start(void);
125     uint8_t tx433_cmd_tx_stop(void);
```

```
126       uint8_t tx433_cmd_led_ctrl(const uint8_t);
127       uint8_t tx433_cmd_fifo_init(void);
128       uint8_t tx433_cmd_fifo_set(void);
129       uint8_t tx433_cmd_set_int(void);
130       uint8_t tx433_cmd_get_int_status(void);
131       uint8_t tx433_cmd_get_state(void);
132       uint8_t tx433_cmd_change_state(void);

134   #endif /* TX_433_H_ */
135   //------------------------------------------------------------------------------------------
```

Programmausdruck C.19: *Zellensensor „tx_433.h“*

```
/*
 ================================================================================================
 |  Projekt:        Masterarbeit, Zellensensor
 |  Erstellt von:   Niels Jegenhorst
 |  Erstellt am:    23.03.2011
 |  Geändert am:    23.03.2011
 |  Hardware:       MSP430f235 auf BATSEN - ZS v0.1
 |  Tools:          MSPGCC - 20100218; Eclipse - Helios
 |  Beschreibung:   Interrupt Service Routinen der UCB0
 |
 |  Funktion:       OK
 |  Datei:          isr_ucb0.c
 ================================================================================================
*/

#include "header_main.h"
#include "src/globals.h"

//--- ISR UCB0 TX ------------------------------------------------------------------------------
interrupt ( USCIAB0TX_VECTOR ) isr_ucb0_tx (void) {

}
//--- ISR UCB0 RX ------------------------------------------------------------------------------
interrupt ( USCIAB0RX_VECTOR ) isr_ucb0_rx (void) {

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

Programmausdruck C.20: *Zellensensor „isr_ucb0.c“*

C.1.2 Basisstation

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

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

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

    Modified by:        Niels Jegenhorst
    Last modification:  19/07/2011

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

-------------------------------------------------------------------------------
    Headerfiles
-------------------------------------------------------------------------------*/
#include <main.h>
#include <stdio.h>
#include <signal.h>
#include <math.h>
#include <msp430x16x.h>
#include "headerfiles/rtc.h"
#include "headerfiles/adc.h"
#include "headerfiles/flash.h"
#include "headerfiles/lcd16x2.h"
#include "headerfiles/uart_menu.h"
#include "headerfiles/msp_functions.h"
#include "headerfiles/reader_13p56mhz.h"
#include "headerfiles/sensor_data_proc.h"
#include "headerfiles/system_handling.h"

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

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

    unsigned char OSC_error;
    unsigned char UART_error;
    RTC_MSP430 RTC_tmp = {0,0,0,0};

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

    globals_init();             // Set global variables
    PORTS_init();               // initialize PORT direction/function
    READER_OFF;                 // turn Reader 13.56MHz off
    READER_MOD_OFF;             // switch TX 13.56MHz off
    READER_ON;                  // Turn Reader 13.56MHz on
    LCD_init();                 // initialize 16x2 character LCD
    LED1_OFF;                   // switch LEDS off
```

```
62      LED2_OFF;
64      eint();                     // IRQs enable
66      USART0_init_SPI();          // initialize USART in SPI-Mode
67      reader_set_reg();           // set Reader's Register for direct-mode, etc.
68      USART0_init(RX_INT, RX_TX); // initialize USART, enable RX&TX irq and RX&TX modul
70      init_Flash();               // initialize Flash
71      #ifdef ENABLE_CURRENT_MEASURE
72          ADC_init();             // initialize ADC
73      #endif
74      UART0_send_text("\n System started ----\n");
75      BL_ON;                      // LCD Backlight ON
76      RTC_startup();              // initialize RTC
77      FLASH_read_at_startup();    // read control data from flash?
78      RADIO_init();               // Initialize radio unit
79      LCD_send_time();            // display time on LCD
80      recording_stop();           // stop measurement
82      /*-------------------------------------------------------------------------------
83      ||  Infinite Loop
84      -------------------------------------------------------------------------------*/
85      while(TRUE) {
86                                                      // look if real time clock
87          RTC_compare(&RTC_tmp);                      //  values has changed
89          // string from UART received? ------------------------------------------------
90          if (BATMON_control_reg & GLOBAL_STRING_RECEIVED) {
91              UART_error = UART0_cmp_string();        // compare string with
92                                                      //  valid commands
93              if (UART_error)
94                  UART0_send_text("  NOK\n");
95              else
96                  UART0_send_text("  OK\n");
98              BATMON_control_reg &= !GLOBAL_STRING_RECEIVED;  // clear global value
99          }
100         // data received from sensor? ------------------------------------------------
101         if (BATMON_control_reg & VALID_DATA_RECEIVED) {
102             #ifdef PIN_DEBUG
103                 TP_ADC2_ON;
104             #endif
105             BATMON_control_reg &= !VALID_DATA_RECEIVED;     // clear global value
106             g_rx_status = INACTIVE;                         // set flag
107             if (scan_mode < 1) {
108                 RADIO_store_frame();                        // store valid frame in flash
109             }
110                                                             // show sensor data on LCD:
111             if((show_sensornmb == 0) && (g_show_sensor_lcd == TRUE)) {
112                 LCD_send_sensor_v2(g_sensor_data.id & 0xFF, g_sensor_data.v_cell/10);
113             }                                               // show sensor data on UART
114             if(g_show_sensor_data_uart == DEC) {            //  in dezimal-mode:
115                 switch (g_sensor_data.dataset) {
116                     case SET_1:                                 // dataset 1 received:
117                         printf("SEN:%02x |ADD:%02x |VID:%02x |CV:%01d.%04d |SV:%04d |T:%d\
                                n",
118                             RADIO_frame.sensor_index,
119                             (uint8_t)(g_sensor_data.id & 0xFF),
120                             g_sensor_data.vid,
121                             (uint16_t)g_sensor_data.v_cell / 10000,
122                             (uint16_t)g_sensor_data.v_cell % 10000,
123                             (uint16_t)g_sensor_data.v_supply,
```

```
124                                 (uint16_t)g_sensor_data.temperature);
125                         break;
126                     case SET_2:                                     // dataset 2 received:
127                         printf("SEN:%02x |ADD:%02x |VID:%02x |CV:%01d.%04d |AV:%04d |LV
                                :%04d\n",
128                                 RADIO_frame.sensor_index,
129                                 (uint8_t)(g_sensor_data.id & 0xFF),
130                                 g_sensor_data.vid,
131                                 (uint16_t)g_sensor_data.v_cell / 10000,
132                                 (uint16_t)g_sensor_data.v_cell % 10000,
133                                 (uint16_t)g_sensor_data.v_actin,
134                                 (uint16_t)g_sensor_data.v_lp);
135                         break;
136                     case REPLY_CMD:                                 // reply cmd received:
137                         printf("SEN:%02x |ADD:%02x |VID:%02x |C1:%02x |P1:%04x |C2:%02x |
                                P2:%04x\n",
138                                 RADIO_frame.sensor_index,
139                                 (uint8_t)(g_sensor_data.id & 0xFFFF),
140                                 g_sensor_data.vid,
141                                 g_sensor_data.rx_last_cmd[0],
142                                 g_sensor_data.rx_last_parameter[0],
143                                 g_sensor_data.rx_last_cmd[1],
144                                 g_sensor_data.rx_last_parameter[1]);
145                         break;
146                     default:
147                         break;
148                 }
149             }                                                   // show sensor data on UART
150             else if(g_show_sensor_data_uart == HEX) {           //  in hex-mode:
151                 switch (g_sensor_data.dataset) {
152                     case SET_1:                                     // dataset 1 received:
153                         printf("SEN%02xTS%04x%04xA%04xV%02xC%04xS%04xT%04x\n",
154                                 RADIO_frame.sensor_index,
155                                 (uint16_t)((g_sensor_data.timestamp & 0xFFFF0000) >> 16),
156                                 (uint16_t) (g_sensor_data.timestamp & 0x0000FFFF),
157                                 (uint8_t)(g_sensor_data.id & 0xFFFF),
158                                 g_sensor_data.vid,
159                                 (uint16_t)g_sensor_data.v_cell,
160                                 (uint16_t)g_sensor_data.v_supply,
161                                 (uint16_t)g_sensor_data.temperature);
162                         break;
163                     case SET_2:                                     // dataset 2 received:
164                         printf("SEN%02xTS%04x%04xA%04xV%02xC%04xW%04xL%04x\n",
165                                 RADIO_frame.sensor_index,
166                                 (uint16_t)((g_sensor_data.timestamp & 0xFFFF0000) >> 16),
167                                 (uint16_t) (g_sensor_data.timestamp & 0x0000FFFF),
168                                 (uint8_t)(g_sensor_data.id & 0xFFFF),
169                                 g_sensor_data.vid,
170                                 (uint16_t)g_sensor_data.v_cell,
171                                 (uint16_t)g_sensor_data.v_actin,
172                                 (uint16_t)g_sensor_data.v_lp);
173                         break;
174                     case REPLY_CMD:                                 // reply cmd received:
175                         printf("SEN%02xTS%04x%04xA%04xV%02xC1%02xP1%04xC2%02xP2%04x\n",
176                                 RADIO_frame.sensor_index,
177                                 (uint16_t)((g_sensor_data.timestamp & 0xFFFF0000) >> 16),
178                                 (uint16_t) (g_sensor_data.timestamp & 0x0000FFFF),
179                                 (uint8_t)(g_sensor_data.id & 0xFFFF),
180                                 g_sensor_data.vid,
181                                 g_sensor_data.rx_last_cmd[0],
182                                 g_sensor_data.rx_last_parameter[0],
183                                 g_sensor_data.rx_last_cmd[1],
184                                 g_sensor_data.rx_last_parameter[1]);
```

```
185                         break;
186                     default:
187                         break;
188                 }
189             }
190             #ifdef PIN_DEBUG
191                 TP_ADC2_OFF;
192             #endif
193         }
194         // new current value measured? ---------------------------------------------------
195         if (BATMON_control_reg & NEW_CURRENT_VALUE) {
196             #ifdef PIN_DEBUG
197                 TP_ADC2_ON;
198             #endif                                          // show current data on UART
199             if(g_show_sensor_data_uart == DEC) {            //  in dezimal-mode:
200                 printf("CUR: |TS:%02d:%02d.%03d |C:%01d |V:%c%04d\n",
201                         (uint8_t) ((g_currtiming.timestamp >> 16) & 0x3F),
202                         (uint8_t) ((g_currtiming.timestamp >> 10) & 0x3F),
203                         (uint16_t) (g_currtiming.timestamp & 0x3FF),
204                         g_currmeas.channel,
205                         g_currmeas.current_in_out,
206                         (uint16_t)g_currmeas.last_data);
207             }                                               // show current data on UART
208             else if(g_show_sensor_data_uart == HEX) {       //  in hex-mode:
209                 printf("CURTS%04x%04xV%04x\n",
210                         (uint16_t)((g_currtiming.timestamp & 0xFFFF0000) >> 16),
211                         (uint16_t) (g_currtiming.timestamp & 0x0000FFFF),
212                         g_currmeas.value_signed);
213             }
214             BATMON_control_reg &= ~NEW_CURRENT_VALUE;
215             #ifdef PIN_DEBUG
216                 TP_ADC2_OFF;
217             #endif
218         }
219         // show sensor data in first LCD row? --------------------------------------------
220         if (BATMON_control_reg & CLEAR_FIRST_LCD_ROW) {
221             BATMON_control_reg &= ~CLEAR_FIRST_LCD_ROW;
222             LCD_show_voltage(show_sensornmb);
223         }
224         // Button pressed jump into menu? ------------------------------------------------
225         if((BATMON_control_reg & B1) || (BATMON_control_reg & B2)) {
226             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
227             BATMON_menu_options();
228         }
229         // Button pressed show next info menu --------------------------------------------
230         if(BATMON_control_reg & B3) {
231             BATMON_control_reg &= ~B3;                      // clear button 3 flag
232             show_sensornmb++;
233             BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
234             if (show_sensornmb > (SENSOR_active+6)) {
235                 show_sensornmb = 0;
236             }
237         }
238     }   // End of while(1)
239     return (0);
240 }
241 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.21: *Basisstation „main.c“*

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

    Author:             Stephan Plaschke
    Date:               10/28/2007
    Last update:        04/04/2008

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

    Modified by:        Niels Jegenhorst
    Last modification:  13/07/2011

    File:               main.h
-------------------------------------------------------------------------------------------*/

#ifndef MAIN_H_
#define MAIN_H_

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

#define ENABLE_CURRENT_MEASURE
#define ENABLE_RX_ERROR_LOG

/*-------------------------------------------------------------------------------------------
    Defined values, equal in each project, don't change
-------------------------------------------------------------------------------------------*/

/* Defines for selected frequencies, claculation:   */
/* low power crystal frequency divided by 8 = 4096Khz   */
/* timer uses DCO frequency and counts up in one ACLK wave  */
/* e.g. 1/(2MHz)*488 = 0,000244s = 4098kHz  */
#define MHZ_1                   244
#define MHZ_2                   488
#define MHZ_4                   976
#define MHZ_8                   1952

#define MAX_STRING_LENGTH       15      // max stringlength for UART

#define GLOBAL_STRING_RECEIVED  0x02    // value in BATMON_control_reg for uart data
#define VALID_DATA_RECEIVED     0x04    // value in BATMON_control_reg for rx433 data
#define CLEAR_FIRST_LCD_ROW     0x08    // value in BATMON_control_reg
#define NEW_CURRENT_VALUE       0x10    // value in BATMON_control_reg for current data
#define B1                      0x20    // value in BATMON_control_reg for button 1
#define B2                      0x40    // value in BATMON_control_reg for button 2
#define B3                      0x80    // value in BATMON_control_reg for button 3


#define PRINTF_LCD              (BATMON_control_reg |= 0x01)
#define PRINTF_UART             (BATMON_control_reg &= ~0x01)

enum boolean            {FALSE, TRUE};
enum status             {OFF, ON, STATUS};

enum uart_interrupts    {RX_INT, TX_INT, RX_TX_INT};
enum uart_modules       {RX, TX, RX_TX};
enum uart_speed         {BAUD_4800, BAUD_9600, BAUD_19k2, BAUD_115k2};
```

```
63  enum escapes            {   SPACE = ' ', BACKSPACE = '\b', HTAB = '\t',
64                              RETURN = '\r', NEWLINE = '\n', VTAB = '\v',
65                              NULLTERMINATOR = '\0'   };

67  /*-------------------------------------------------------------------------------------
68      Defined values
69  -------------------------------------------------------------------------------------*/

71  #define MSP430x169      1           // select the used MSP430
72  #define uC_FREQUENCY    MHZ_8       // select the MSP430 frequency  (1,2,4,8 MHZ)

74  #define USART_BAUD      BAUD_115k2  // define USART BAUDRATE(BAUD_4800, BAUD_9600,
75                                      //                       BAUD_19k2,
76                                      //                       BAUD_115k2(only@4&8MHz))
77  #define ACLK_DIVIDER    (DIVA1)             // LFXT1 / 4 = ACLK = 8192Hz
78  //#define ACLK_DIVIDER  (DIVA1 | DIVA0)     // LFXT1 / 8 = ACLK = 4096Hz

80  //--- Global Flags: ----------------------------------------------------------------
81  #define PIN_DEBUG

83  //--- Others: ----------------------------------------------------------------------
84  #define SW_IRQ_ZS_SYNCH (BIT5)

86  /*-------------------------------------------------------------------------------------
87   * PORT definitions: 1->output, 0->input
88   * PORTX_IES 0-> rising edge, 1->falling edge
89   * PORTX_IE  0-> interrupt disabled, 1->interrupt enabled
90   -------------------------------------------------------------------------------------*/
91  //___ Port 1: ____________________________________
92  //  Pin 0:  Reader IRQ          Input
93  //  Pin 1:  TX 433MHz           Input
94  //  Pin 2:  Reader Rx BCLK      Input
95  //  Pin 3:  Reader EN           Output
96  //  Pin 4:  Reader MOD          Output
97  //  Pin 5:  Button 1            Input
98  //  Pin 6:  Button 2            Input
99  //  Pin 7:  Button 3            Input
100 #define PORT1_OUT       (BIT4)                  // Pin 4: Reader Modulation off
101 #define PORT1_DIR       (BIT3 | BIT4)
102 #define PORT1_SEL       BIT1                    // Pin 1: Timer A0 CCI0A
103 #define PORT1_IES       (BIT5 | BIT6 | BIT7)    // rising edge:  Pin 0...4
104                                                 // falling edge: Pin 5,6,7
105 #define PORT1_IE        (BIT5 | BIT6 | BIT7)    // IRQ on:       Pin 5,6,7

107 //___ Port 2: ____________________________________
108 //  Pin 0:  FLASH CE            Output
109 //  Pin 1:  FLASH RE            Output
110 //  Pin 2:  FLASH WE            Output
111 //  Pin 3:  FLASH ALE EN        Output
112 //  Pin 4:  FLASH CLE           Output
113 //  Pin 5:  DALLAS              Output      --> used for software irq
114 //  Pin 6:  Reader ASK/OOK      Output
115 //  Pin 7:  FLASH R/B           Input
116 #define PORT2_DIR       (BIT0 | BIT1 | BIT2 | BIT3 | BIT4 | BIT5 | BIT6)
117 #define PORT2_SEL       0x00
118 #define PORT2_IES       0x00
119 #define PORT2_IE        (SW_IRQ_ZS_SYNCH)

121 //___ Port 3: ____________________________________
122 //  Pin 0:  Reader STE          Output
123 //  Pin 1:  SPI SIMO0           Output
124 //  Pin 2:  SPI SOMI0           Input
125 //  Pin 3:  SPI ULCK0           Output
```

```
126 //  Pin 4:  RS-232 TXD0         Output
127 //  Pin 5:  RS-232 RXD0         Input
128 //  Pin 6:  LED1                Output
129 //  Pin 7:  LED2                Output
130 #define PORT3_DIR       (BIT0 | BIT1 | BIT3 | BIT4 | BIT6 | BIT7)
131 #define PORT3_SEL       0x00

133 //___ Port 4: ____________________________________
134 //  Pin 0:  LCD LIGHT           Output
135 //  Pin 1:  LCD E               Output
136 //  Pin 2:  LCD R/W             Output
137 //  Pin 3:  LCD RS              Output
138 //  Pin 4:  LCD DB4             Output
139 //  Pin 5:  LCD DB5             Output
140 //  Pin 6:  LCD DB6             Output
141 //  Pin 7:  LCD DB7             Output
142 #define PORT4_DIR       0xFF
143 #define PORT4_SEL       0x00

145 //___ Port 5: ____________________________________
146 //  Pin 0:  FLASH I/O 0         Output
147 //  Pin 1:  FLASH I/O 1         Output
148 //  Pin 2:  FLASH I/O 2         Output
149 //  Pin 3:  FLASH I/O 3         Output
150 //  Pin 4:  FLASH I/O 4         Output
151 //  Pin 5:  FLASH I/O 5         Output
152 //  Pin 6:  FLASH I/O 6         Output
153 //  Pin 7:  FLASH I/O 7         Output
154 #define PORT5_DIR       0xFF
155 #define PORT5_SEL       0x00

157 //___ Port 6: ____________________________________
158 //  Pin 0:  ADC A0          Input   --> ADC current measure
159 //  Pin 1:  ADC A1          Input   --> ADC current measure
160 //  Pin 2:  ADC A2          Output  --> TP INL  (WDT)
161 //  Pin 3:  ADC A3          Output  --> TP INR  (RX433 Error / ADC12 ISR)
162 //  Pin 4:  ADC A4          Output  --> TP ADC1 (RX433 Frame)
163 //  Pin 5:  ADC A5          Output  --> TP ADC2 (RX433 / Current in mainloop)
164 //  Pin 6:  ADC A6 / DAC0   Input
165 //  Pin 7:  ADC A7 / DAC1   Input
166 #define PORT6_DIR       (BIT2 | BIT3 | BIT4 | BIT5)
167 #define PORT6_SEL       (BIT0 | BIT1)

169 //--- I/O Pins: ------------------------------------------
170 #define LED1_ON         (P3OUT &= ~BIT6)
171 #define LED1_OFF        (P3OUT |=  BIT6)
172 #define LED1_TOGGLE     (P3OUT ^=  BIT6)

174 #define LED2_ON         (P3OUT &= ~BIT7)
175 #define LED2_OFF        (P3OUT |=  BIT7)
176 #define LED2_TOGGLE     (P3OUT ^=  BIT7)

178 #define BL_ON           (P4OUT |=  BIT0)
179 #define BL_OFF          (P4OUT &= ~BIT0)
180 #define BL_TOGGLE       (P4OUT ^=  BIT0)

182 //--- Debug I/O Pins: ------------------------------------
183 #define TP_INL_ON           (P6OUT |=  BIT2)
184 #define TP_INL_OFF          (P6OUT &= ~BIT2)
185 #define TP_INL_TOGGLE       (P6OUT ^=  BIT2)

187 #define TP_INR_ON           (P6OUT |=  BIT3)
188 #define TP_INR_OFF          (P6OUT &= ~BIT3)
```

```
189 #define TP_INR_TOGGLE       (P6OUT ^=  BIT3)

191 #define TP_ADC1_ON          (P6OUT |=  BIT4)
192 #define TP_ADC1_OFF         (P6OUT &= ~BIT4)
193 #define TP_ADC1_TOGGLE      (P6OUT ^=  BIT4)

195 #define TP_ADC2_ON          (P6OUT |=  BIT5)
196 #define TP_ADC2_OFF         (P6OUT &= ~BIT5)
197 #define TP_ADC2_TOGGLE      (P6OUT ^=  BIT5)

199 /*-------------------------------------------------------------------------------------
200     Prototypes
201 -------------------------------------------------------------------------------------*/
202 void globals_init(void);


205 #endif /*MAIN_H_*/
206 //-------------------------------------------------------------------------------------
```

Programmausdruck C.22: *Basisstation „main.h“*

```
1  /*---------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        ADC functions
4      Used components:    MSP430-169STK
5
6      Author:             Alexander Hoops
7      Date:               11/04/2009
8      Last update:        12/02/2010
9
10     Modified by:        Niels Jegenhorst
11     Last modification:  12/07/2011
12
13     File:               adc.c
14 ---------------------------------------------------------------------------------------*/
15
16 ---------------------------------------------------------------------------------------
17     Headerfiles
18 ---------------------------------------------------------------------------------------*/
19 #include <main.h>
20 #include <signal.h>
21 #include <msp430x16x.h>
22 #include "headerfiles/adc.h"
23 #include "headerfiles/globals.h"
24
25 /*---------------------------------------------------------------------------------------
26     Init ADC
27 ---------------------------------------------------------------------------------------*/
28 void ADC_init(void)
29 /*---------------------------------------------------------------------------------------*/
30 {
31     ADC12CTL0  = 0x0000;
32     ADC12CTL1  = 0x0000;
33     ADC12MCTL2 = 0x0000;
34     ADC12IE    = 0x0000;
35     ADC12IFG   = 0x0000;
36         // SH-Time = 1024 * ADC12CLK, URef+ = 2.5V, Uref ON,
37         // ADC12 ON, Multiple Sample & Conversion
38     ADC12CTL0  = (SHT03 | SHT02 | SHT00 | REF2_5V | REFON | ADC12ON | MSC);
39         // Conv start add = 0, SHS = ADC12SC_BIT, ADC12CLK = SMCLK = 8MHz, CLK div = 4,
40         // SHP, Sequence-of-channels with no Repeat
41     ADC12CTL1  = (CSTARTADD_0 | SHP | ADC12DIV1 | ADC12DIV0 |
42                   ADC12SSEL0 | ADC12SSEL1 | CONSEQ0);
43         // Vr+ = Vref+ and Vr- = AVss, Input Channel = A0
44     ADC12MCTL0 = (SREF_1 | INCH_0);
45         // Vr+ = Vref+ and Vr- = AVss, Input Channel = A1
46         // Vr+ = Vref+ and Vr- = AVss, Input Channel = A1, End of Sequence
47     ADC12MCTL1 = (SREF_1 | INCH_1 | EOS);
48         // enable Interrupt A1
49     ADC12IE    = BIT1;
50 }
51
52 /*---------------------------------------------------------------------------------------
53     Start ADC
54 ---------------------------------------------------------------------------------------*/
55 void ADC_start(void)
56 /*---------------------------------------------------------------------------------------*/
57 {
58         // trigger ADC for current measuring:
59
60     ADC12CTL0 |= (ADC12SC | ENC);   // Enable conversion, Start sample and conversion
61                                     // calculate next init time:
62     g_currtiming.trigger_msec += CURR_MEAS_INTERVAL_MSEC;
```

```
63      if(g_currtiming.trigger_msec >= BS_MSEC) {
64          g_currtiming.trigger_sec += g_currtiming.trigger_msec / BS_MSEC;
65          g_currtiming.trigger_msec = g_currtiming.trigger_msec % BS_MSEC;
66      }
67      g_currtiming.trigger_sec += CURR_MEAS_INTERVAL_SEC;
68      if(g_currtiming.trigger_sec >= 60) {
69          g_currtiming.trigger_sec -= 60;
70      }
71  }

73  /*-------------------------------------------------------------------------------
74      Stop ADC
75  -------------------------------------------------------------------------------*/
76  void ADC_stop(void)
77  /*-------------------------------------------------------------------------------*/
78  {
79      ADC12CTL0 &= ~(ADC12SC | ENC);  // Disable conversion, Start sample and conversion
80  }

82  /*-------------------------------------------------------------------------------
83      Disable ADC
84  -------------------------------------------------------------------------------*/
85  void ADC_disable(void)
86  /*-------------------------------------------------------------------------------*/
87  {
88      ADC12CTL0 &= ~(ADC12ON | REFON);    // ADCOFF + U REF OFF
89  }

91  /*-------------------------------------------------------------------------------
92      Check BAT
93  -------------------------------------------------------------------------------*/
94  void Check_BAT(void)
95  /*-------------------------------------------------------------------------------*/
96  {
97      unsigned char v, t=0;
98      unsigned long test_value, test_value_2;
99      static unsigned char index_last_minutes_low_consumption = 0;
100     static unsigned char Capacity_write = 0;


103     // check consumption of last minute > 0,2 AH
104     if (g_currah.AH_last_minute < 20) {
105         index_last_minutes_low_consumption++;
106         if (index_last_minutes_low_consumption >= 3) {
107             index_last_minutes_low_consumption = 3;
108             Capacity_write = 1;
109             // write voltage based cap. in main cap.
110         }
111     }
112     else {
113         index_last_minutes_low_consumption = 0;
114         Capacity_write = 0;
115     }
116     // find depth cell
117     test_value = (SENSOR_last_data[3][t]<<8);
118     test_value |= (SENSOR_last_data[4][t]);

120     for (v=1;v<SENSOR_active;v++) {
121         test_value_2  = (SENSOR_last_data[3][v]<<8);
122         test_value_2 |= (SENSOR_last_data[4][v]);
123         if (test_value_2 < test_value) {
124             t=v;
125             test_value = test_value_2;
```

```
126             }
127         }

129         // check if lowest is depth discharged => capacity = 0
130         if ((test_value/40.96) < CELL_VOLT_MIN) {
131             g_currah.U_MASTER_CAPACITY = 0;
132         }
133         // voltage value from lowest sensor over max voltage (charging for example)
134         // capacity = 100
135         else if ((test_value/40.96) > CELL_VOLT_MAX) {
136             g_currah.U_MASTER_CAPACITY = 100;
137         }
138         // generate capacity based on voltage value from lowest cell
139         else {
140             g_currah.U_MASTER_CAPACITY = (((test_value/40.96)-CELL_VOLT_MIN)/
141                         (CELL_VOLT_MAX-CELL_VOLT_MIN)*100);
142         }
143         // if low current consumption write voltage based capacity to main cap.
144         if (Capacity_write == 1) {
145             g_currah.AH_MASTER = g_currah.U_MASTER_CAPACITY * BATT_CAPACITY / 44 ;
146             Capacity_write = 0;
147         }
148     }
149     //---------------------------------------------------------------------------------------
```

Programmausdruck C.23: *Basisstation „adc.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        ADC functions
4      Used components:    MSP430-169STK
5
6      Author:             Alexander Hoops
7      Date:               11/04/2009
8      Last update:        12/02/2010
9
10     Modified by:        Niels Jegenhorst
11     Last modification:  07/07/2011
12
13     File:               adc.h
14 -------------------------------------------------------------------------------------*/
15
16 #ifndef ADC_H_
17 #define ADC_H_
18
19 /*-------------------------------------------------------------------------------------
20     Defined values
21 -------------------------------------------------------------------------------------*/
22 #define CURRENT_TARA_STARTVALUE 2047        // TARA_set = 2.5V -> 2.5V/2 = 2047
23 #define CURRENT_DIR_POS         '+'
24 #define CURRENT_DIR_NEG         '-'
25 #define CURRENT_CH1             0
26 #define CURRENT_CH2             1
27
28 /*-------------------------------------------------------------------------------------
29     Prototypes
30 -------------------------------------------------------------------------------------*/
31 void ADC_init(void);
32 void ADC_start(void);
33 void ADC_stop(void);
34 void ADC_disable(void);
35 void Check_BAT(void);
36
37 #endif /*ADC_H_*/
38 //-------------------------------------------------------------------------------------
```

Programmausdruck C.24: *Basisstation „adc.h“*

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

    Author:             Niels Jegenhorst
    Date:               19/04/2011
    Last update:        08/07/2011

    File:               globals.h
-------------------------------------------------------------------------------------*/

#ifndef GLOBALS_H_
#define GLOBALS_H_

    #include "headerfiles/typedefs.h"

    #ifndef EXTERN
        #define EXTERN extern
    #endif

    /*-------------------------------------------------------------------------------------
    ||  Global variables
    -------------------------------------------------------------------------------------*/
    EXTERN volatile unsigned char BATMON_control_reg;   // global control register
    EXTERN volatile unsigned char BUTTON_state;         // actual button state
    EXTERN volatile unsigned char UART_menue_position;  // UART_menue position
    EXTERN unsigned char WDT_function;      // WDT function, BLINKY for RTC SETUP or RTC
    EXTERN unsigned char RTC_menue_position;        // menu position in RTC setup
    EXTERN char global_string[MAX_STRING_LENGTH];   // global command string
    EXTERN unsigned char Enable_clear_lcd;          // clear first LCD row
    EXTERN unsigned char  show_sensornmb;           // choice of LCD screening
    EXTERN unsigned char  g_show_sensor_lcd;
    EXTERN SHOW_DATA_UART g_show_sensor_data_uart;  // choice of UART screening
    EXTERN volatile unsigned char measuring;    // Datalogging activated = 1
    EXTERN volatile unsigned char scan_mode;    // standard_measure =>0 /
                                                // scan mode sensor(1)/temp cal (2)
    EXTERN unsigned long temp_correct_value;    // temp value for calibration

    EXTERN unsigned char SENSOR_addr_req[5];    // array for continuous mode,
                                                // shows required address
    EXTERN unsigned char SENSOR_addr_receiv[5]; // array for checking if all sensors have
                                                // send a value
    EXTERN unsigned char SENSOR_address[NUM_SENSORS_MAX];       // sensor address
    EXTERN unsigned char SENSOR_active;                         // active sensor count
    EXTERN unsigned char SENSOR_last_data[8][NUM_SENSORS_MAX];  // last received data
    EXTERN unsigned char SENSOR_low;            // number of sensors under min voltage
    EXTERN unsigned char fer;

    EXTERN volatile USART_MODE          g_usart_mode;
    EXTERN volatile RTC_MSP430          RTC_values;     // global RTC values
    EXTERN volatile RADIO_FRAME_STRUCT  RADIO_frame;    // received data frame
    EXTERN volatile SE_SEQ_POS          RADIO_rx_state; // data receive state
    EXTERN volatile RX_TIMING           g_rx_timing;
    EXTERN volatile RX_ERROR            g_rx_error;
    EXTERN volatile RX_STATUS           g_rx_status;
    EXTERN volatile uint8_t             g_rx_en;
    EXTERN volatile uint8_t             g_rx_reen;
    EXTERN volatile SENSOR_DATA         g_sensor_data;

    /*----Sensor temperature calibration data----------------------*/
    EXTERN signed char   SENSOR_temp_cal_b[NUM_SENSORS_MAX+1]; // sensor address 0..40
```

```
64      /*-------------Current Measuring-------------------------------*/
65      EXTERN volatile CURR_MEAS   g_currmeas;
66      EXTERN volatile CURR_AH     g_currah;
67      EXTERN volatile CURR_TIMING g_currtiming;

69      /*-------------Battery Parameters------------------------------*/
70      EXTERN unsigned long CELL_VOLT_MAX;         // Cell voltage max 2.10V = 210
71      EXTERN unsigned long CELL_VOLT_MIN;         // Cell voltage min 1.88V = 188
72      EXTERN unsigned long BATT_CAPACITY;         // Battery capacity 500Ah = 500

74      /*-------------Flash-------------------------------------------*/
75      #ifdef FLASH_H_
76          EXTERN unsigned char  WRITE_BUF [BUF_SIZE];
77          EXTERN unsigned char  READ_BUF [BUF_SIZE];
78          EXTERN unsigned char  Ring_Full;
79          EXTERN unsigned short Index_Ring;
80          EXTERN unsigned short Count_Ring;
81          EXTERN unsigned short Index_Last;
82      #endif

84      /*-------------TX 13.56MHz-------------------------------------*/
85      EXTERN volatile READER_TX   g_reader_tx;
86      EXTERN volatile ZS_TIMING   g_zs_timing;
87      EXTERN volatile uint8_t     g_zs_balance;

89      /*-------------------------------------------------------------------------------
90          Constants
91      -------------------------------------------------------------------------------*/
92      #ifdef INIT_GLOBALS                         // define & init global Constants
93          const char *RTC_days[7] = {
94                  "MON ",
95                  "TUE ",
96                  "WED ",
97                  "THU ",
98                  "FRY ",
99                  "SAT ",
100                 "SUN "
101         };
102         const char *RTC_months[12] = {
103                 "Jan",
104                 "Feb",
105                 "Mar",
106                 "Apr",
107                 "May",
108                 "Jun",
109                 "Jul",
110                 "Aug",
111                 "Sep",
112                 "Oct",
113                 "Nov",
114                 "Dec"
115         };
116     #else                                       // declare as extern
117         EXTERN char *RTC_days[];
118         EXTERN char *RTC_months[];
119     #endif

121 #endif /* GLOBALS_H_ */
122 //-------------------------------------------------------------------------------
```

Programmausdruck C.25: *Basisstation „globals.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for init all globals
4      Used components:    MSP430-169STK
5
6      Author:             Niels Jegenhorst
7      Date:               20/04/2011
8      Last update:        26/09/2011
9
10     File:               globals_init.c
11 -------------------------------------------------------------------------------------------
12
13 -------------------------------------------------------------------------------
14     Headerfiles
15 -------------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <msp430x16x.h>
18 #include "headerfiles/msp_functions.h"
19 #include "headerfiles/uart_menu.h"
20 #include "headerfiles/lcd16x2.h"
21 #include "headerfiles/adc.h"
22 #include "headerfiles/flash.h"
23 #include "headerfiles/system_handling.h"
24 #include "headerfiles/globals.h"
25
26 /*-------------------------------------------------------------------------------
27     void globals_init(void)
28 -------------------------------------------------------------------------------*/
29 void globals_init(void) {
30
31     uint8_t i;
32
33     RTC_values.day      = 22;               // initialize clock
34     RTC_values.lastday  = 21;
35     RTC_values.month    = 01;
36     RTC_values.year     = 2009;
37     RTC_values.wday     = TUE;
38     RTC_values.hr       = 23;
39     RTC_values.min      = 59;
40     RTC_values.sec      = 55;
41     RTC_values.msec     = 0;
42
43     Index_Ring = 256;                       // default address in flash
44     Index_Last = 256;
45
46     UART_menu_position = UART_MENU_FIRSTCHAR;
47     Enable_clear_lcd   = STATUS;
48     show_sensornmb     = 0;
49     g_show_sensor_lcd  = TRUE;
50     g_show_sensor_data_uart = HEX;
51
52     measuring = 0;
53     scan_mode = 0;
54
55 /*  SENSOR_temp_cal_b = { 0,27,22,-5,10,15,-14,2,9,-34,
56  *                        81,32,0,-24,38,0,0,0,0,0,
57  *                        0,0,0,0,0,0,0,0,0,0,
58  *                        0,0,0,0,0,0,0,0,0,0,0};
59 */
60     SENSOR_temp_cal_b[0]  = 0;
61     SENSOR_temp_cal_b[1]  = 27;
62     SENSOR_temp_cal_b[2]  = 22;
```

```
63      SENSOR_temp_cal_b[3]  = -5;
64      SENSOR_temp_cal_b[4]  = 10;
65      SENSOR_temp_cal_b[5]  = 15;
66      SENSOR_temp_cal_b[6]  = -14;
67      SENSOR_temp_cal_b[7]  = 2;
68      SENSOR_temp_cal_b[8]  = 9;
69      SENSOR_temp_cal_b[9]  = -34;
70      SENSOR_temp_cal_b[10] = 81;
71      SENSOR_temp_cal_b[11] = 32;
72      SENSOR_temp_cal_b[12] = 0;
73      SENSOR_temp_cal_b[13] = -24;
74      SENSOR_temp_cal_b[14] = 38;
75      for(i=15; i<41; i++) {
76          SENSOR_temp_cal_b[i] = 0x00;
77      }

79      g_currmeas.tara                 = CURRENT_TARA_STARTVALUE;
80      g_currmeas.value                = 1500;
81      g_currmeas.channel              = CURRENT_CH1;
82      g_currmeas.current_invert       = 0;
83      g_currmeas.current_in_out       = CURRENT_DIR_POS;
84      g_currah.AH_last_minute         = 0;
85      g_currah.AH_last_sec            = 0;
86      g_currah.AH_last_minute_IN_OUT  = 0;
87      g_currtiming.trigger_en         = FALSE;

89      CELL_VOLT_MAX   = 214;
90      CELL_VOLT_MIN   = 189;
91      BATT_CAPACITY   = 88;
92      fer = 0;

94      g_rx_error.bit_error   = 0x0000;
95      g_rx_error.wrong_RunIn = 0x0000;
96      g_rx_error.timeout     = 0x0000;
97      for(i=0; i<6; i++) {
98          g_rx_error.crc_error[i] = 0x0000;
99      }

101     g_rx_status                 = INACTIVE;
102     g_rx_en                     = FALSE;
103     g_reader_tx.state           = FINISH;
104     g_zs_timing.t_interval      = ZS_TIME_INTERVAL;
105     g_zs_timing.t_sample        = ZS_TIME_SAMPLE;
106     g_zs_timing.send_synch_en   = FALSE;
107     g_zs_balance                = OFF;

109 }
110 //---------------------------------------------------------------------------------------
```

Programmausdruck C.26: *Basisstation „globals_init.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of ADC12
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               12/15/2007
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Simon Püttjer
14     Last modification:  05/05/2011
15
16     Modified by:        Niels Jegenhorst
17     Last modification:  11/07/2011
18
19     File:               isr_adc12.c
20 -------------------------------------------------------------------------------------*/
21
22 /*-------------------------------------------------------------------------------------
23     Headerfiles
24 -------------------------------------------------------------------------------------*/
25 #include <main.h>
26 #include <signal.h>
27 #include <stdio.h>
28 #include <msp430x16x.h>
29 #include "headerfiles/adc.h"
30 #include "headerfiles/globals.h"
31
32 /*-------------------------------------------------------------------------------------
33     ADC12 interrupt service routine (current measuring)
34 -------------------------------------------------------------------------------------*/
35 interrupt (ADC12_VECTOR) ADC12(void)
36 /*-------------------------------------------------------------------------------------*/
37 {
38     #ifdef PIN_DEBUG
39         TP_INR_ON;
40     #endif
41     ADC12CTL0 &= ~(ADC12SC | ENC);      // Stop ADC
42     PRINTF_UART;                        // set printf to UART
43     //*************************************************************************
44     // measuring range decision
45     //*************************************************************************
46     g_currmeas.channel2 = ADC12MEM0;    // readout ADC Ch.0 --> Sensor Ch.2
47     g_currmeas.channel1 = ADC12MEM1;    // readout ADC Ch.1 --> Sensor Ch.1
48
49                                         // Timestamp = [DAY HR MIN SEC MSEC] (32-bit):
50     g_currtiming.timestamp   = RTC_values.day;          // days:    5-bit used
51     g_currtiming.timestamp <<= 5;                       // shift 5-bit for hours
52     g_currtiming.timestamp  |= RTC_values.hr;           // hours:   5-bit used
53     g_currtiming.timestamp <<= 6;                       // shift 6-bit for minutes
54     g_currtiming.timestamp  |= RTC_values.min;          // minutes: 6-bit used
55     g_currtiming.timestamp <<= 6;                       // shift 6-bit for seconds
56     g_currtiming.timestamp  |= RTC_values.sec;          // seconds: 6-bit used
57     g_currtiming.timestamp <<= 10;                      // shift 10-bit for msec
58     g_currtiming.timestamp  |= RTC_values.msec  / 10;   // milliseconds: 10-bit used
59
60     // switch to channel 2 if channel 1 out of range (channel1:  -70A ... +70 A)
61     //                                               (channel2: -500A ... +500A)
62     if((g_currmeas.channel1 < 3578) && (g_currmeas.channel1 > 1033)) {
```

```
63          if((g_currmeas.value < 3560) && (g_currmeas.value > 1048)) {
64              g_currmeas.value = g_currmeas.channel1;
65              g_currmeas.channel = CURRENT_CH1;       //set channel flag to ch1
66          }
67          else {
68              g_currmeas.value = g_currmeas.channel2;
69              g_currmeas.channel = CURRENT_CH2;       //set channel flag to ch2
70          }
71      }
72      else {
73          g_currmeas.value = g_currmeas.channel2;
74          g_currmeas.channel = CURRENT_CH2;           //set channel flag to ch2
75      }

77      //*****************************************************************************
78      // init Current tara
79      //*****************************************************************************
80      if(scan_mode == 4) {
81          scan_mode = 5;
82          g_currmeas.sum = g_currmeas.value;      // init accumulation
83          g_currmeas.current_sum_counter = 0;
84      }
85      else if(scan_mode == 5) {
86                          // take X-1 values left --> accumulate current value
87          if(g_currmeas.current_sum_counter < (32-1)) {
88              g_currmeas.sum += g_currmeas.value;
89              g_currmeas.current_sum_counter++;
90          }
91          else {          // all data received --> take average of X values as tara value
92              g_currmeas.sum >>= 5;               // calculate average
93              g_currmeas.tara = g_currmeas.sum;   // save tara value
94              scan_mode = 0;                      // reset scan mode to normal
95              printf("Current Tara value: %3li \n", g_currmeas.tara);
96              if (measuring == 0) {
97                  ADC_stop();
98              }
99          }
100     }
101     //*****************************************************************************
102     // normal consumption recording (scan_mode 0)
103     //*****************************************************************************
104     else {
105         // init new tara
106         if (g_currmeas.tara == 0) {
107             g_currmeas.tara = g_currmeas.sum;
108             g_currah.AH_last_minute = 0;
109         }
110         // consumption recording
111         else {
112             // current inverted?
113             if (g_currmeas.current_invert == 0) {
114                 g_currmeas.work = g_currmeas.value;
115             }
116             else {
117                 //invert current value
118                 if(g_currmeas.value <= g_currmeas.tara)
119                     g_currmeas.work = g_currmeas.tara +
120                                                 (g_currmeas.tara - g_currmeas.value);
121                 else
122                     g_currmeas.work = g_currmeas.tara -
123                                                 (g_currmeas.value - g_currmeas.tara);
124             }
125             // Zero Point definition
```

```
126             if ( ((g_currmeas.work) <= (g_currmeas.tara+1)) &&
127                  ((g_currmeas.work) >= (g_currmeas.tara-1))) {
128                 g_currmeas.work = g_currmeas.tara;
129             }
130             // Current calculation:
131             // Attention! last_data is not signed!
132             // value_signed = bit0-11:  Current
133             //                bit14:    Channel
134             //                bit15:    Direction
135             if(g_currmeas.work > g_currmeas.tara) {
136                 g_currmeas.last_data = g_currmeas.work - g_currmeas.tara;
137                 g_currmeas.value_signed = (uint16_t)(g_currmeas.last_data) |
138                                           (uint16_t)(g_currmeas.channel << 14);
139                 g_currmeas.current_in_out = CURRENT_DIR_POS;
140             }
141             else {
142                 g_currmeas.last_data = g_currmeas.tara - g_currmeas.work;
143                 g_currmeas.value_signed = (uint16_t)(g_currmeas.last_data) |
144                                           (uint16_t)(g_currmeas.channel << 14) |
145                                           (uint16_t)(1 << 15) ;
146                 g_currmeas.current_in_out = CURRENT_DIR_NEG;
147             }
148             // Calculation of Current in Ampere for LCD & Console:
149             if(g_currmeas.channel == CURRENT_CH1) {
150                 //  Channel 1:  A = (ADC +- TARA) * 2 * 1/4096 * 2.5 * 1/(26.7*10^-3)
151                 //           10*A = (ADC +- TARA) * 500/1093    (max 11-bit final value)
152                 g_currmeas.last_data *= 500;
153                 g_currmeas.last_data /= 1093;
154             }
155             else {
156                 //  Channel 2:  A = (ADC +- TARA) * 2 * 1/4096 * 2.5 * 1/(4*10^-3)
157                 //           10*A = (ADC +- TARA) * 6250/2048   (max 14-bit final value)
158                 g_currmeas.last_data *= 6250;
159                 g_currmeas.last_data >>= 11;
160             }
161         }       // end else "consumption recording" ------------------------------------
162         BATMON_control_reg |= NEW_CURRENT_VALUE;    // Set flag
163     }       // end of "scan_mode 0" ------------------------------------------------------
164     ADC12IFG &= ~BIT1;                              // Clear IRQ Flags
165     if((scan_mode == 4) || (scan_mode == 5)) {      // If Current-Tara-Mode:
166         ADC12CTL0 |= (ADC12SC | ENC);               // --> Trigger ADC again
167     }
168     #ifdef PIN_DEBUG
169         TP_INR_OFF;
170     #endif
171 }
172 //---------------------------------------------------------------------------------------
```

Programmausdruck C.27: *Basisstation „isr_adc12.c“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of Ports 1 & 2
4      Used components:    MSP430-169STK, BS Reader 13.56MHz
5
6      Author:             Niels Jegenhorst
7      Date:               20/04/2010
8      Last update:        13/07/2010
9
10     File:               isr_ports.c
11 -------------------------------------------------------------------------------------------
12
13 -----------------------------------------------------------------------------
14     Headerfiles
15 -----------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <signal.h>
18 #include <msp430x16x.h>
19 #include <headerfiles/adc.h>
20 #include <headerfiles/reader_13p56mhz.h>
21 #include "headerfiles/globals.h"
22
23 /*-----------------------------------------------------------------------------
24     PORT1 interrupt service routine
25 -----------------------------------------------------------------------------*/
26 interrupt ( PORT1_VECTOR ) isr_port1 (void)
27 /*-----------------------------------------------------------------------------*/
28 {
29         // Attention:   For debouncing of the 3 Buttons a 10nF Capacitor
30         //              must be added in parallel to the Resistors R2, R3, R4
31
32     uint8_t i_deb;
33
34     if(P1IFG & (BIT5 | BIT6 | BIT7)) {          // any Button pressed?
35         for(i_deb=0; i_deb<0x7F; i_deb++) {         // debounce & more buttons @ one time
36             if((P1IN & BIT5) == 0x00)               // Button 1 pressed, low activ
37                 BATMON_control_reg |= B1;
38             if((P1IN & BIT6) == 0x00)               // Button 2 pressed, low activ
39                 BATMON_control_reg |= B2;
40             if((P1IN & BIT7) == 0x00)               // Button 3 pressed, low activ
41                 BATMON_control_reg |= B3;
42         }
43         P1IFG &= ~(BIT5 | BIT6 | BIT7);             // clear IRQ flags
44     }
45 }
46
47 /*-----------------------------------------------------------------------------
48     PORT2 interrupt service routine
49 -----------------------------------------------------------------------------*/
50 interrupt ( PORT2_VECTOR ) isr_port2 (void)
51 /*-----------------------------------------------------------------------------*/
52 {
53         // This ISR is used as an software IRQ for sending the SYNCH CMD to the ZS.
54         // It is triggert by the ISR of WDT every x seconds.
55
56     if((P2IFG & SW_IRQ_ZS_SYNCH) == SW_IRQ_ZS_SYNCH) {
57         P2IE  &= ~SW_IRQ_ZS_SYNCH;                  // disable current IRQ source
58         eint();                                     // Nested IRQ's enable --------------
59                                                     // Send CMD Synch to ZS:
60         tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SYNCH, g_zs_timing.t_interval);
61                                                     // Trigger ADC for current measuring:
62         if(g_currtiming.trigger_en == TRUE) {
```

```
63              ADC_start();
64          }
65                                                  // Calculate next synch time:
66          g_zs_timing.send_synch_msec += SYNCH_INTERVAL_MSEC;
67          if(g_zs_timing.send_synch_msec >= BS_MSEC) {
68              g_zs_timing.send_synch_sec += g_zs_timing.send_synch_msec / BS_MSEC;
69              g_zs_timing.send_synch_msec = g_zs_timing.send_synch_msec % BS_MSEC;
70          }
71          g_zs_timing.send_synch_sec += SYNCH_INTERVAL_SEC;
72          if(g_zs_timing.send_synch_sec >= 60) {
73              g_zs_timing.send_synch_sec -= 60;
74          }

76          dint();                                 // Nested IRQ's disable -------------
77          P2OUT &= ~SW_IRQ_ZS_SYNCH;              // set port pin (dallas)
78          P2IFG &= ~SW_IRQ_ZS_SYNCH;              // clear IRQ flag
79          P2IE  |= SW_IRQ_ZS_SYNCH;               // enable current IRQ source
80      }
81  }
82  //-------------------------------------------------------------------------------------
```

Programmausdruck C.28: *Basisstation „isr_ports.c“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of Timer A
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               12/15/2007
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Simon Püttjer
14     Last modification:  05/05/2011
15
16     Modified by:        Niels Jegenhorst
17     Last modification:  08/07/2011
18
19     File:               isr_timera.c
20 -------------------------------------------------------------------------------------------*/
21
22 /*-----------------------------------------------------------------------------
23     Headerfiles
24 -----------------------------------------------------------------------------*/
25 #include <main.h>
26 #include <signal.h>
27 #include <stdio.h>
28 #include <msp430x16x.h>
29 #include "headerfiles/lcd16x2.h"
30 #include "headerfiles/msp_functions.h"
31 #include "headerfiles/uart_menu.h"
32 #include "headerfiles/system_handling.h"
33 #include "headerfiles/sensor_data_proc.h"
34 #include "headerfiles/isr_timera.h"
35 #include "headerfiles/globals.h"
36
37 //-------------------------------------------------------------------------------------------
38 // calculate current frame error
39 //-------------------------------------------------------------------------------------------
40 void countFER(unsigned char sensor_number, unsigned char crc_error,
41               unsigned short bit_too_long, unsigned short bit_too_short,
42               unsigned short wrong_RunIn) {
43
44     volatile short lost_frames = 0;
45     unsigned short temp_error = 0, temp_error_old = 0;
46
47     switch (fer) {
48
49         //init frame error calc.
50         case 1:
51
52             fer = 2;
53
54         //frame error calc.
55         case 2:
56
57             temp_error = (short)(RADIO_frame.frame_byte[1]);
58             temp_error <<= 8;
59             temp_error |= (short)(RADIO_frame.frame_byte[2]);
60             temp_error_old = (short)(SENSOR_last_data[1][sensor_number]);
61             temp_error_old <<= 8;
62             temp_error_old |= (short)(SENSOR_last_data[2][sensor_number]);
```

```
64              //lost frames is the difference of the current and the last transmission count
65              lost_frames = temp_error - temp_error_old - 1;

67              //matlab binblock protocoll #<N><D><A>
68              UART0_send('#'); //doublecross terminates transmission
69              UART0_send('2'); //number of digits that follows in D
70              UART0_send('1'); //number of bytes that follows in A
71              UART0_send('0');
72              UART0_send(SENSOR_address[sensor_number]);
73              UART0_send((unsigned char) (bit_too_long >> 8));
74              UART0_send((unsigned char) (bit_too_long & 0x00FF));
75              UART0_send((unsigned char) (bit_too_short >> 8));
76              UART0_send((unsigned char) (bit_too_short & 0x00FF));
77              UART0_send((unsigned char) (wrong_RunIn >> 8));
78              UART0_send((unsigned char) (wrong_RunIn & 0x00FF));
79              UART0_send((unsigned char) (lost_frames >> 8));
80              UART0_send((unsigned char) (lost_frames & 0x00FF));
81              UART0_send(crc_error);

83              //send new line
84              UART0_send_text("\n");
85              break;

87          default:
88              fer = 0;
89      }
90  }
91  //-------------------------------------------------------------------------------------------

93  /*-------------------------------------------------------------------------------------
94      Timer A0 interrupt service routine, Capture/compare0 interrupt  (bit recognition)
95  -------------------------------------------------------------------------------------*/
96  interrupt (TIMERA0_VECTOR) TIMER_A0(void)
97  /*-------------------------------------------------------------------------------------*/
98  {
99      static unsigned char Temp_cal_counter[41]=
100         {0,0,0,0,0,0,0,0,0,0,
101          0,0,0,0,0,0,0,0,0,0,
102          0,0,0,0,0,0,0,0,0,0,
103          0,0,0,0,0,0,0,0,0,0,
104          0};
105     signed long temp_diff, real_Temp;
106     // compare values for the received address in global array,
107     // byte = position in array
108     unsigned char address_bit, address_byte;
109     unsigned char i, j, frame_error, ex, sens, sensor_calibrated;

111     //-------------------------------------------------------------
112     //calculate bit width
113     //-------------------------------------------------------------
114     TA0CTL &= ~(MC1 | MC0);                     // stop  timer A
115     TA0CTL |= TACLR;                            // reset timer A
116     TA0CTL |= (ID1 | ID0 | MC1);                // start timer A first/again
117     g_rx_timing.capture = TA0CCR0;              // readout capture time
118     g_rx_timing.rx_in   = (TACCTL0 & CCI) >> 3; // read CCI0A input state

120     //-------------------------------------------------------------
121     //case differentiation - bit recognition
122     //-------------------------------------------------------------
123     switch(RADIO_rx_state)
124     {
125     case SE_SEQ_PREPARE:        //_____________________________________________________
```

```
127         // first interrupt of a new rx frame

129         #ifdef PIN_DEBUG
130             TP_ADC1_ON;
131             TP_INR_OFF;
132         #endif
133         g_rx_timing.shift_in = 0;                   // reset shift counter
134         g_rx_timing.byte_ctr = 0;                   // reset byte counter
135         g_rx_timing.rtc_sof.day  = RTC_values.day;  // Save actual RTC value:
136         g_rx_timing.rtc_sof.hr   = RTC_values.hr;
137         g_rx_timing.rtc_sof.min  = RTC_values.min;
138         g_rx_timing.rtc_sof.sec  = RTC_values.sec;
139         g_rx_timing.rtc_sof.msec = RTC_values.msec / 10;

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

146     case SE_SEQ_START:          //_____________________________________________________________

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

150         if( (g_rx_timing.capture < START_SEQ_LENGTH_MAX) &&
151             (g_rx_timing.capture > START_SEQ_LENGTH_MIN) ) {
152             RADIO_rx_state = SE_SEQ_PRE_WAIT;   // next RX state
153             g_rx_timing.shift_in = (14*2)+1;    // waitcounter set for 15 1/2 bits until
154         }                                       //  second byte received
155         else {                                // when time not equals:
156             TA0CTL &= ~(MC1 | MC0);             // stop timer A
157             TA0CTL |= TACLR;                    // reset timer A
158             g_rx_status    = RUN_IN_ERROR;
159             RADIO_rx_state = SE_SEQ_PREPARE;    // set RX state for next start of frame
160             #ifdef ENABLE_RX_ERROR_LOG
161                 g_rx_error.wrong_RunIn++;
162             #endif
163             #ifdef PIN_DEBUG
164                 TP_ADC1_OFF;
165                 TP_INR_ON;
166             #endif
167         }
168         break;                                      // exit switch-case block

170     case SE_SEQ_PRE_WAIT:       //_____________________________________________________________

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

174         g_rx_timing.shift_in --;

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

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

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

188             RADIO_rx_state = SE_SEQ_DATA_START;     // next RX state
```

```
189         }
190         else {                                      // calc floating average:
191             g_rx_timing.halfbit_tavg += g_rx_timing.capture;
192             g_rx_timing.halfbit_tavg >>= 1;
193         }
194         break;                                      // exit switch-case block

196     case SE_SEQ_DATA_START:     //_____________________________________________________

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

200                                                 // time for long high of transition "01":
201         if( (g_rx_timing.capture < g_rx_timing.bit_tmax) &&
202             (g_rx_timing.capture > g_rx_timing.halfbit_tmax) ){
203             RADIO_rx_state = SE_SEQ_RXDATA;     // next RX state
204         }
205         else {                                  // nothing detected:
206             TA0CTL &= ~(MC1 | MC0);             // stop timer A
207             TA0CTL |= TACLR;                    // reset timer A
208             g_rx_status    = SYNCH_ERROR;
209             RADIO_rx_state = SE_SEQ_PREPARE;    // set RX state for next start of frame
210             #ifdef ENABLE_RX_ERROR_LOG
211                 g_rx_error.wrong_RunIn++;
212             #endif
213             #ifdef PIN_DEBUG
214                 TP_ADC1_OFF;
215                 TP_INR_ON;
216             #endif
217         }
218         break;                                      // exit switch-case block

220     case SE_SEQ_RXDATA:         //_____________________________________________________

222         // catch manchester: ---------------------------------------------------------
223                                                 // time for half bit length detected: -------
224         if( (g_rx_timing.capture < g_rx_timing.halfbit_tmax) &&
225             (g_rx_timing.capture > g_rx_timing.halfbit_tmin) ) {
226                                                 // rising edge  --> shift in "1"
227                                                 // falling edge --> shift in "0"
228             g_rx_timing.rx_seq  <<= 1;
229             g_rx_timing.rx_seq   |= g_rx_timing.rx_in;
230             g_rx_timing.shift_in += 1;
231         }                                       // time for one bit length detected: --------
232         else if( (g_rx_timing.capture < g_rx_timing.bit_tmax) &&
233                  (g_rx_timing.capture > g_rx_timing.bit_tmin) ) {
234                                                 // rising edge  --> shift in "01"
235                                                 // falling edge --> shift in "10"
236             g_rx_timing.rx_seq  <<= 1;
237             g_rx_timing.rx_seq   |= ((!g_rx_timing.rx_in) & 0x01);
238             g_rx_timing.rx_seq  <<= 1;
239             g_rx_timing.rx_seq   |= g_rx_timing.rx_in;
240             g_rx_timing.shift_in += 2;
241         }
242         else {                                  // time to short or long, exit: ------------
243             TA0CTL &= ~(MC1 | MC0);             // stop timer A
244             TA0CTL |= TACLR;                    // reset timer A
245             g_rx_status    = BIT_ERROR;
246             RADIO_rx_state = SE_SEQ_PREPARE;    // set RX state for next start of frame
247             #ifdef ENABLE_RX_ERROR_LOG
248                 g_rx_error.bit_error++;
249             #endif
250             #ifdef PIN_DEBUG
251                 TP_ADC1_OFF;
```

```
252                 TP_INR_ON;
253             #endif
254             break;                              // exit switch-case block
255         }
256         // decode manchester: ------------------------------------------------------------
257         if(g_rx_timing.shift_in == 16) {    // one byte complete: ----------------------
258             RADIO_frame.frame_byte[g_rx_timing.byte_ctr] = 0x00;
259             g_rx_timing.rx_in    = 0x01;        // init for bit setting in frame_byte
260             g_rx_timing.shift_in = 0x0002;      // init for manchester sequenz "10"
261             while(TRUE) {
262                                                 // manchester sequenz "10" --> 1:
263                 if(g_rx_timing.rx_seq & g_rx_timing.shift_in) {
264                                                 //   then set bitvalue in byte
265                     RADIO_frame.frame_byte[g_rx_timing.byte_ctr] |= g_rx_timing.rx_in;
266                 }                               // manchester sequenz "01" --> 0:
267                                                 //   then nothing to set in byte
268                 if(g_rx_timing.shift_in != 0x8000) {
269                     g_rx_timing.rx_in <<= 1;    // shift to next possible bit in byte
270                     g_rx_timing.shift_in <<= 2; // shift to next manchester sequenz
271                 }
272                 else                            // when manchster sequenz decoded:
273                     break;                      //   exit loop while(TRUE)
274             }
275             g_rx_timing.shift_in = 0;           // reset counter for next sequenz
276             g_rx_timing.byte_ctr ++;            // increment byte counter
277         }
278         //--------------------------------------------------------------------------------
279         // end of bitstream: -------------------------------------------------------------
280         if (g_rx_timing.byte_ctr == 11) {
281             TA0CTL &= ~(MC1 | MC0);             // stop timer A
282             TA0CTL |= TACLR;                    // reset timer A
283             i=0;                                // init address counter
284             switch (scan_mode) {
285                 //-----------------------------------------------------------------------
286                 // scan mode started
287                 //-----------------------------------------------------------------------
288                 case 1:
289                     // check if address valid (if it is already existent),skip Address 0
290                     while( (SENSOR_address[i] != 0 ) &&
291                            (SENSOR_address[i] != RADIO_frame.frame_byte[3]) ) {
292                         i++;
293                     }

295                     //Sensor not yet existant - print info to uart
296                     if (SENSOR_address[i] != RADIO_frame.frame_byte[3]) {
297                         PRINTF_UART;
298                         printf("Sensor: %2x => Adresse: %2x\n",
299                                 i+1, RADIO_frame.frame_byte[3]);
300                     }

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

304                     //desired number of sensors found
305                     if (i >= SENSOR_active-1) {
306                         scan_mode = 0;                          // stop scan mode
307                         LCD_send_cmd(LCD_LINE1);
308                         recording_stop();                       // disable radio unit
309                         LCD_send_text(" Scan complete  ");
310                         UART0_send_text("Scan complete\n");
311                         BATMON_control_reg &= ~VALID_DATA_RECEIVED; // set global value
312                         FLASH_Write_Controll_Data();            // write gathered control
313                                                                 //  data to flash
314                         RADIO_disable();                        // stop reveiver
```

```
315                         g_rx_status = INACTIVE;

317                         for (i=0; i<SENSOR_active; i++) {       // set sensors active:
318                             SENSOR_last_data[0][i] = SENSOR_address[i];
319                         }

321                         switch(SENSOR_active) {
322                             case 12:
323                                 SENSOR_addr_req[0] = 0xFF;  // 12 sensors are configured
324                                 SENSOR_addr_req[1] = 0x0F;
325                                 SENSOR_addr_receiv[0] = 0xFF;
326                                 SENSOR_addr_receiv[1] = 0x0F;
327                                 break;
328                             case 24:
329                                 SENSOR_addr_req[0] = 0xFF;  // 24 sensors are configured
330                                 SENSOR_addr_req[1] = 0xFF;
331                                 SENSOR_addr_req[2] = 0xFF;
332                                 SENSOR_addr_req[3] = 0x00;
333                                 SENSOR_addr_req[4] = 0x00;
334                                 SENSOR_addr_receiv[0] = 0xFF;
335                                 SENSOR_addr_receiv[1] = 0xFF;
336                                 SENSOR_addr_receiv[2] = 0xFF;
337                                 SENSOR_addr_receiv[3] = 0x00;
338                                 SENSOR_addr_receiv[4] = 0x00;
339                                 break;
340                             case 40:
341                                 SENSOR_addr_req[0] = 0xFF;  // 40 sensors are configured
342                                 SENSOR_addr_req[1] = 0xFF;
343                                 SENSOR_addr_req[2] = 0xFF;
344                                 SENSOR_addr_req[3] = 0xFF;
345                                 SENSOR_addr_req[4] = 0xFF;
346                                 SENSOR_addr_receiv[0] = 0xFF;
347                                 SENSOR_addr_receiv[1] = 0xFF;
348                                 SENSOR_addr_receiv[2] = 0xFF;
349                                 SENSOR_addr_receiv[3] = 0xFF;
350                                 SENSOR_addr_receiv[4] = 0xFF;
351                                 break;
352                             default:
353                                 SENSOR_addr_req[0] = 0x3F;  // 6 sensors are configured
354                                 SENSOR_addr_receiv[0] = 0x3F;
355                                 break;
356                         }
357                     }           // End if desired number of sensors found
358                     #ifdef PIN_DEBUG
359                         TP_ADC1_OFF;
360                     #endif
361                     break;

363                 //---------------------------------------------------------------------
364                 // Calibration initialization
365                 //---------------------------------------------------------------------
366                 case 2:
367                     for (sens=0;sens<41;sens++) {           // clear temp variables
368                         Temp_cal_counter[sens]=0;
369                     }
370                     scan_mode = 3;
371                     RADIO_enable();
372                     break;
373                 //---------------------------------------------------------------------
374                 // Calibration started
375                 //---------------------------------------------------------------------
376                 case 3:
377                     real_Temp = (RADIO_frame.frame_byte[4]);
```

```
378                     real_Temp = real_Temp << 8;
379                     real_Temp |= RADIO_frame.frame_byte[5];
380                     ex = Temp_cal_counter[RADIO_frame.frame_byte[3]];
381                     if (ex==0) {
382                         SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]] =
383                                 temp_correct_value - real_Temp;
384                     }
385                     else if((ex>=1) && (ex<20)) {
386                         temp_diff = (temp_correct_value - real_Temp) +
387                                     (ex * SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]]);
388                         SENSOR_temp_cal_b[RADIO_frame.frame_byte[3]] = temp_diff /(ex+1);
389                     }
390                     Temp_cal_counter[RADIO_frame.frame_byte[3]]++;
391                     sensor_calibrated = 0;
392                     for (sens=0;sens<SENSOR_active;sens++) {
393                         if (Temp_cal_counter[SENSOR_address[sens]]>5) {
394                             sensor_calibrated++;
395                         }
396                     }
397                     if (sensor_calibrated == SENSOR_active) {
398                         scan_mode = 0;
399                         LCD_send_cmd(LCD_LINE1);
400                         recording_stop();                          // disable radio unit
401                         LCD_send_text("Calibr. complete");
402                         UART0_send_text("Calibration complete\n");
403                         g_rx_status = INACTIVE;
404                         RADIO_init_frame();
405                         FLASH_Write_Controll_Data();
406                     }
407                     else {
408                         BATMON_control_reg &= ~VALID_DATA_RECEIVED; // clear global value
409                     }
410                     break;
411                 //-----------------------------------------------------------------------
412                 // scan mode '0' no certain mode
413                 //-----------------------------------------------------------------------
414                 case 0:
415                     do {                // check if valid sensor:
416                                         // compare if sensor address is allowed
417                         if (SENSOR_address[i] == RADIO_frame.frame_byte[3])
418                             break;              // exit loop, get sensor address index
419                         i++;
420                     } while(i<40);

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

425                         j = 0;
426                         frame_error = 0;
427                         while(j<11) {           // check CRC
428                             frame_error = frame_error ^ RADIO_frame.frame_byte[j];
429                             j++;
430                         }
431                         if (frame_error) {      // CRC not valid, exit
432                             g_rx_status    = CRC_FAILURE;
433                             RADIO_rx_state = SE_SEQ_PREPARE;
434                             #ifdef ENABLE_RX_ERROR_LOG
435                                 g_rx_error.crc_error[RADIO_frame.sensor_index]++;
436                             #endif
437                             #ifdef PIN_DEBUG
438                                 TP_ADC1_OFF;
439                                 TP_INR_ON;
440                             #endif
```

```
441                             break;              // exit switch-case block
442                         }
443                         #ifdef ENABLE_RX_ERROR_LOG
444                             if(fer == 1) {      // reset all error variables for startup
445                                 g_rx_error.bit_error = 0;
446                                 g_rx_error.wrong_RunIn = 0;
447                             }
448                             if(fer > 0) {       // FER calc.
449                                 countFER(RADIO_frame.sensor_index,
450                                         g_rx_error.crc_error[RADIO_frame.sensor_index],
451                                         g_rx_error.bit_error,
452                                         g_rx_error.bit_error,
453                                         g_rx_error.wrong_RunIn);
454                             }
455                                                 // reset error count variables:
456     //                      g_rx_error.crc_error[RADIO_frame.sensor_index] = 0;
457     //                      g_rx_error.wrong_RunIn = 0;
458     //                      g_rx_error.bit_error   = 0;
459                         #endif
460                                                 // calculate position in sensor_addr_req
461                         address_bit  = RADIO_frame.sensor_index % 8;
462                         address_byte = RADIO_frame.sensor_index / 8;
463                                                 // clear sensor request and receive flag
464                         SENSOR_addr_req[address_byte] &= ~(0x01 << address_bit);
465                         SENSOR_addr_receiv[address_byte] &= ~(0x01 << address_bit);

467                         for (j=0; j<8; j++) {   // save last data received for sensor X
468                             SENSOR_last_data[j][RADIO_frame.sensor_index] =
469                                     RADIO_frame.frame_byte[j];
470                         }
471                         convert_sensor_data();              // convert sensor data
472                         g_rx_status = SUCCESS;              // set flag
473                         BATMON_control_reg |= VALID_DATA_RECEIVED;  // set flag
474                         #ifdef PIN_DEBUG
475                             TP_ADC1_OFF;
476                         #endif
477                     }                           // end of "if (i<40) {}"
478                     break;
479                 default:
480                     scan_mode = 0;
481             }           // end of switch case (scan mode)
482                                             // Receiving frame completed (valid or not),
483             RADIO_rx_state = SE_SEQ_PREPARE;//  return to the beginning
484         }                                   // end of "if (.frame_bit_counter == 88) {}"
485         break;                              // exit switch-case block

487     default:                    //_____________________________________________________
488         RADIO_rx_state = SE_SEQ_PREPARE;
489         break;                              // exit switch-case block
490     }
491     TA0CCTL0 &= ~CCIFG;                     // reset IRQ flag, deletes unwanted captures
492                                             // (reset of IRQ flag is done by calling ISR)
493 }
494 /*-------------------------------------------------------------------------------------
495     Timer A1 interrupt service routine for compare register 1 interrupt
496 -------------------------------------------------------------------------------------*/
497 interrupt (TIMERA1_VECTOR) TIMER_A1(void)
498 /*-------------------------------------------------------------------------------------*/
499 {

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

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

507       g_rx_status    = TIMEOUT;
508       RADIO_rx_state = SE_SEQ_PREPARE;        // set RX state for next start of frame
509       #ifdef ENABLE_RX_ERROR_LOG
510           g_rx_error.timeout ++;
511       #endif
512       #ifdef PIN_DEBUG
513           TP_INR_ON;
514           TP_ADC1_OFF;
515       #endif

517       TA0CCTL1 &= ~CCIFG;                     // reset IRQ flag
518   }
519   //-------------------------------------------------------------------------------------
```

Programmausdruck C.29: *Basisstation „isr_timera.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Header-File for ISR of Timer A
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               10/28/2007
8      Last update:        04/04/2008
9
10     Modified by:        Niels Jegenhorst
11     Last modification:  06/05/2011
12
13     File:               isr_timera.h
14 -------------------------------------------------------------------------------------*/
15
16 #ifndef ISR_TIMERA_H
17 #define ISR_TIMERA_H
18
19 /*-------------------------------------------------------------------------------------
20     Defines
21 -------------------------------------------------------------------------------------*/
22
23 /* min length for the start sequence    */
24 #define START_SEQ_LENGTH_MIN    700
25 /* max length for the start sequence */
26 #define START_SEQ_LENGTH_MAX    1200    //equals ~1,2ms @smclk/8
27
28
29 #endif /*ISR_TIMERA_H*/
30 //-------------------------------------------------------------------------------------
```

Programmausdruck C.30: *Basisstation „isr_timera.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of Timer B
4      Used components:    MSP430-169STK
5
6      Author:             Niels Jegenhorst
7      Date:               29/04/2010
8      Last update:        07/06/2010
9
10     File:               isr_timerb.c
11 -------------------------------------------------------------------------------------------
12
13 -----------------------------------------------------------------------------
14     Headerfiles
15 -----------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <signal.h>
18 #include <msp430x16x.h>
19 #include "headerfiles/reader_13p56mhz.h"
20 #include "headerfiles/msp_functions.h"
21 #include "headerfiles/globals.h"
22
23 /*-----------------------------------------------------------------------------
24     Timer B0 interrupt service routine for CC0 (Manchester Coding)
25 -----------------------------------------------------------------------------*/
26 interrupt (TIMERB0_VECTOR) TIMER_B0(void)
27 /*-----------------------------------------------------------------------------*/
28 {
29         // This routine is modulating the output signal "READER_MOD" for transmitting
30         // data on 13.56MHz to the "ZS v0.1".
31         // At first a RUN IN will be transmitted, followed by a 8bit SYNCH.
32         // The payload is modulated in manchester, MSB first. For the correct decoding
33         // with only rising edges a halfbit high phase is added at the end of frame.
34
35     switch(g_reader_tx.state) {
36         case WAKE_UP_END:                   //_____________________________________________
37             READER_MOD_OFF;                     // set 13.56MHz RF off, end of seq
38             TBCCR0 = TBCCR_2BIT;                // compare value for bit handling time
39             g_reader_tx.state = RUN_IN_START;   // next state
40             break;
41         case RUN_IN_START:                  //_____________________________________________
42             READER_MOD_ON;                      // set 13.56MHz RF on, start of seq
43             TBCCR0 = TBCCR_RUN_IN;              // compare value for RUN-IN time
44             g_reader_tx.state = RUN_IN_END;     // next state
45             break;
46         case RUN_IN_END:                    //_____________________________________________
47             READER_MOD_OFF;                     // set 13.56MHz RF off, end of seq
48             TBCCR0 = TBCCR_BIT;                 // compare value for bit handling time
49             g_reader_tx.bitpos = 0x01;          // reset as bit counter
50             g_reader_tx.state  = SYNCH_0;       // next state
51             break;
52         case SYNCH_0:                       //_____________________________________________
53             if(g_reader_tx.bitpos % 2)          // actual transmisson "1" then:
54                 READER_MOD_ON;                              // set 13.56MHz RF on
55             else                                // actual transmission "0" then:
56                 READER_MOD_OFF;                             // set 13.56MHz RF off
57             if(g_reader_tx.bitpos == (TX13P56_SYNCH_LENGTH*2 -3)) { // after X-1 1/2 bit:
58                 g_reader_tx.bitpos = 0x00;                  // reset bit counter
59                 g_reader_tx.state  = SYNCH_1;               // next state
60             }
61             g_reader_tx.bitpos ++;
62             break;
```

```
63          case SYNCH_1:                           //_____________________________________________
64              if(g_reader_tx.bitpos % 2)              // actual transmisson "1" then:
65                  READER_MOD_ON;                          // set 13.56MHz RF on
66              else                                    // actual transmission "0" then:
67                  READER_MOD_OFF;                         // set 13.56MHz RF off
68              if(g_reader_tx.bitpos == 2) {           // after 1 bit goto next state:
69                  g_reader_tx.bitpos  = 0x80;             // set position for shiftout
70                  g_reader_tx.bytectr = 0x00;             // set for first byte
71                  g_reader_tx.manchesterpos = 0x00;       // set first position
72                  g_reader_tx.state = DATALOAD;           // next state
73              }
74              else {
75                  g_reader_tx.bitpos ++;
76              }
77              break;
78          case DATALOAD:                          //_____________________________________________
79              if(g_reader_tx.manchesterpos == 0x00) {     // first position:
80                                                          // actual bit = '1' then:
81                  if(g_reader_tx.txdata[g_reader_tx.bytectr] & g_reader_tx.bitpos)
82                      READER_MOD_ON;                          // set 13.56MHz RF on
83                  else
84                      READER_MOD_OFF;                         // set 13.56MHz RF off
85                  g_reader_tx.manchesterpos = 0x01;
86              }
87              else {                                  // second position:
88                                                          // actual bit = '1' then:
89                  if(g_reader_tx.txdata[g_reader_tx.bytectr] & g_reader_tx.bitpos)
90                      READER_MOD_OFF;                         // set 13.56MHz RF off
91                  else
92                      READER_MOD_ON;                          // set 13.56MHz RF on
93                  g_reader_tx.bitpos >>= 1;                   // increment bit position
94                  g_reader_tx.manchesterpos = 0x00;
95              }                                       // after transmitting 8 bit:
96              if((g_reader_tx.bitpos == 0x00) && (g_reader_tx.manchesterpos == 0x00)) {
97                  g_reader_tx.bytectr ++;                 // increment byte counter
98                  g_reader_tx.bitpos = 0x80;              // set position for shiftout
99                  if(g_reader_tx.bytectr == TX13P56_FRAME_BYTES)  // after tx all bytes:
100                     g_reader_tx.state = END_PULS_1;          // next state
101             }
102             break;
103         case END_PULS_1:                        //_____________________________________________
104             READER_MOD_ON;                          // set 13.56MHz RF on
105             g_reader_tx.state = END_PULS_0;         // next state
106             break;
107         case END_PULS_0:                        //_____________________________________________
108             if(g_reader_tx.cw == ON) {              // Check if CW-mode is on:
109                 READER_MOD_ON;                          // keep 13.56MHz RF on
110             }
111             else {
112                 READER_MOD_OFF;                         // set 13.56MHz RF off
113             }
114             g_reader_tx.state = COMPLETE;           // next state
115             break;
116         case COMPLETE:                          //_____________________________________________
117             TBCTL &= ~(MC1 | MC0);                  // set stop-mode
118             if(g_rx_reen == TRUE)                   // when RX433 was enabled befor TX
119                 RADIO_enable();                     //  enable RX433 again
120             g_reader_tx.state = FINISH;             // next state
121             break;
122         case FINISH:                            //_____________________________________________
123             break;
124     }
125     TBCCTL0 &= ~CCIFG;                          // reset IRQ flag
```

```
126 }

128 /*-------------------------------------------------------------------------------
129     Timer B1 interrupt service routine for CC1-2, TB
130 -------------------------------------------------------------------------------*/
131 interrupt (TIMERB1_VECTOR) TIMER_B1(void)
132 /*-------------------------------------------------------------------------------*/
133 {
134     switch(TBIV) {
135         case 0x02:      // Capture/compare1 interrupt
136                 TBCCTL1 &= ~(CCIFG|COV);            // reset timer occured flag
137                 break;
138         case 0x04:      // Capture/compare2 interrupt
139                 TBCCTL2 &= ~(CCIFG|COV);            // reset timer occured flag
140                 break;
141         case 0x0E:      // Timeroverflow interrupt
142                 TBCTL &= ~TAIFG;                    // reset timer occured flag
143                 break;
144     }
145 }
146 //-------------------------------------------------------------------------------------
```

Programmausdruck C.31: *Basisstation „isr_timerb.c“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of UART0 for RS-232-Interface
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               12/15/2007
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     File:               isr_usart0.c
14 -------------------------------------------------------------------------------------------
15
16 ---------------------------------------------------------------------------
17     Headerfiles
18 ---------------------------------------------------------------------------*/
19 #include <main.h>
20 #include <signal.h>
21 #include <stdio.h>
22 #include <msp430x16x.h>
23 #include "headerfiles/msp_functions.h"
24 #include "headerfiles/uart_menu.h"
25 #include "headerfiles/globals.h"
26
27 /*---------------------------------------------------------------------------
28     UART0 RX interrupt service routine
29 ---------------------------------------------------------------------------*/
30 interrupt (USART0RX_VECTOR) USART0_RX(void)
31 /*---------------------------------------------------------------------------*/
32 {
33     static unsigned char rx_string[MAX_STRING_LENGTH];
34     static unsigned char i;
35
36     switch (RXBUF0) {
37 //      case (RETURN):
38         case (NEWLINE):
39             // if carriage return jump to next line first position and
40             // store string global
41             UART0_send(NEWLINE);
42 //          UART0_send(RETURN);
43
44             rx_string[i] = NULLTERMINATOR;      // terminate string
45             i++;
46
47             for (; i>0; i--)
48             // copy string
49             {
50                 global_string[i-1] = rx_string[i-1];
51             };
52             BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
53             // set control register to global string received
54             break;
55
56         case ('%'):
57             // if '%' jump to next line first position and store string global
58             UART0_send(NEWLINE);
59 //          UART0_send(RETURN);
60
61             rx_string[i] = NULLTERMINATOR;
62             // terminate string
```

```
63              i++;
64
65              for (; i>0; i--)
66              // copy string
67              {
68                  global_string[i-1] = rx_string[i-1];
69              };
70              BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
71              // set control register to global string received
72              break;
73
74          case (BACKSPACE):
75              // backspace, delete last char
76              UART0_send(BACKSPACE);
77              UART0_send(SPACE);
78              UART0_send(BACKSPACE);
79              i--;
80              break;
81
82          default:
83              // echo
84              if(i<(MAX_STRING_LENGTH-1))
85              // string length reached?
86              {
87                  UART0_send(RXBUF0);
88                  // string length not reached store received char
89                  rx_string[i] = RXBUF0;
90                  i++;
91              }
92              else
93                  // string length reached, save string in global variable
94              {
95                  rx_string[i] = NULLTERMINATOR;
96                  // terminate string
97                  for (; i>0; i--)
98                  // copy string
99                  {
100                     global_string[i-1] = rx_string[i-1];
101                 };
102                 BATMON_control_reg |= GLOBAL_STRING_RECEIVED;
103                 // set control register to global string received
104             }
105             break;
106     }
107 }
108
109 /*-------------------------------------------------------------------------------
110     UART0 TX interrupt service routine
111 -------------------------------------------------------------------------------*/
112 interrupt (USART0TX_VECTOR) USART0_TX(void)
113 /*-------------------------------------------------------------------------------*/
114 {
115
116 }
117 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.32: *Basisstation „isr_usart0.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for ISR of Watch-Dog-Timer
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               02/08/2008
8      Last update:        09/05/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  13/07/2011
15
16     File:               isr_wdt.c
17 -------------------------------------------------------------------------------------
18
19 -------------------------------------------------------------------------------------
20     Headerfiles
21 -------------------------------------------------------------------------------------*/
22 #include <main.h>
23 #include <signal.h>
24 #include <msp430x16x.h>
25 #include "headerfiles/rtc.h"
26 #include "headerfiles/adc.h"
27 #include "headerfiles/lcd16x2.h"
28 #include "headerfiles/msp_functions.h"
29 #include "headerfiles/reader_13p56mhz.h"
30 #include "headerfiles/globals.h"
31
32 /*-------------------------------------------------------------------------------------
33     Watchdog interrupt service routine
34 -------------------------------------------------------------------------------------*/
35 interrupt (WDT_VECTOR) WATCH_DOG(void)
36 /*-------------------------------------------------------------------------------------*/
37 {
38     static uint8_t WDT_tmp;
39     static uint8_t BLINKY_state;
40     unsigned char time_char;
41
42     #ifdef PIN_DEBUG
43         TP_INL_ON;
44     #endif
45     //-------------------------------------------------------------------------------------
46     switch (WDT_function) {         // timer for the real time clock
47         case RTC:
48             RTC_values.msec += 625;                         // WDT counts with 1/16 sec
49             if(RTC_values.msec == 10000) {                  // 1sec reached?
50                 RTC_values.msec = 0x0000;                   // reset milliseconds
51                 RTC_values.sec ++;                          // increment seconds
52             }
53             if(RTC_values.sec == 60) {                      // 60 seconds reached?
54                 RTC_values.sec = 0;
55                 if(++RTC_values.min == 60)  {               // 60 minutes reached?
56                     RTC_values.min = 0;
57                     if(++RTC_values.hr == 24) {             // 0 o'clock reached?
58                         RTC_values.hr = 0;
59                         RTC_values.lastday = RTC_values.day;
60                         if ((RTC_values.year % 4) == 0) {   // leap year?
61                             RTC_values.day++;
62                             if((RTC_values.month == 1)&&(RTC_values.day==30)) { // feb.?
```

```
63                              RTC_values.month++;
64                              RTC_values.day = 1;
65                          }
66                      }
67                      else {                              // normal year
68                          RTC_values.day++;
69                          if((RTC_values.month == 1)&&(RTC_values.day==29)) {
70                              RTC_values.month++;
71                              RTC_values.day = 1;
72                          }
73                      }

75                      // 30 day update (April, June, August, October, December)
76                      if ( ((RTC_values.month == 3) || (RTC_values.month == 5) ||
77                            (RTC_values.month == 7) || (RTC_values.month == 9) ||
78                            (RTC_values.month == 11))
79                           && (RTC_values.day == 31))
80                      {
81                          RTC_values.day = 1;
82                          RTC_values.month++;
83                      }

85                      // 31 day update (january, March, Mai, July, September, November)
86                      if (RTC_values.day == 32) {
87                          RTC_values.day = 1;
88                          RTC_values.month++;
89                      }

91                      // year update
92                      if(RTC_values.month == 12) {
93                          RTC_values.month = 0;
94                          RTC_values.year++;
95                      }

97                      // weekday update
98                      if(++RTC_values.wday == (SUN+1)) {
99                          RTC_values.wday = MON;
100                     }
101                 }
102             }
103         }                           // if radio enabled clear LCD/reset rx process:
104         if(Enable_clear_lcd & ON) {     // Clear LCD active message every 4 seconds:
105             if( (!((RTC_values.sec+1) % 4))) {
106                 BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
107             }
108         }                           // if CMD SYNCH to zs must be send:
109         if(g_zs_timing.send_synch_en == TRUE) {
110                                         // every X seconds, init for sending:
111             if((RTC_values.sec  == g_zs_timing.send_synch_sec) &&
112                (RTC_values.msec == g_zs_timing.send_synch_msec)) {
113                                             // trigger SW IRQ for sending SYNCH CMD
114                 P2OUT |= SW_IRQ_ZS_SYNCH;   // and ADC current measuring when needed
115             }
116             else if((g_currtiming.trigger_en == TRUE) &&
117                     (RTC_values.sec  == g_currtiming.trigger_sec) &&
118                     (RTC_values.msec == g_currtiming.trigger_msec)) {
119                                         // every X seconds init for current measure:
120                 ADC_start();            // trigger ADC for current measuring
121             }
122         }
123         break;
124     //-------------------------------------------------------------------------------
125     /* switch every 0,5 sec the editable value on or off, only in RTC_EDIT_MODE */
```

```
126         //-------------------------------------------------------------------------
127         case BLINKY:
128             if(++WDT_tmp == 2) {
129                 WDT_tmp = 0;
130                 LED1_TOGGLE;
131                 if(BLINKY_state ^= 0x01) {
132                     switch (RTC_menue_position) {
133                         case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
134                             LCD_send_text(RTC_days[RTC_values.wday]);
135                             break;
136                         case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
137                             time_char = (RTC_values.hr / 10) + 48;
138                             LCD_send_data(time_char);
139                             time_char = (RTC_values.hr % 10) + 48;
140                             LCD_send_data(time_char);
141                             break;
142                         case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
143                             time_char = (RTC_values.min / 10) + 48;
144                             LCD_send_data(time_char);
145                             time_char = (RTC_values.min % 10) + 48;
146                             LCD_send_data(time_char);
147                             break;
148                         case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
149                             time_char = (RTC_values.sec / 10) + 48;
150                             LCD_send_data(time_char);
151                             time_char = (RTC_values.sec % 10) + 48;
152                             LCD_send_data(time_char);
153                             break;
154                         case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
155                             time_char = (RTC_values.day / 10) + 48;
156                             LCD_send_data(time_char);
157                             time_char = (RTC_values.day % 10) + 48;
158                             LCD_send_data(time_char);
159                             break;
160                         case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
161                             LCD_send_text(RTC_months[RTC_values.month]);
162                             break;
163                         case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
164                             time_char = RTC_values.year / 1000;
165                             LCD_send_data(time_char+48);
166                             time_char = (RTC_values.year / 100) -
167                                         ((RTC_values.year/1000) * 10);
168                             LCD_send_data(time_char+48);
169                             time_char = (RTC_values.year / 10) -
170                                         ((RTC_values.year/100) * 10);
171                             LCD_send_data(time_char+48);
172                             time_char = RTC_values.year % 10;
173                             LCD_send_data(time_char+48);
174                             break;
175                     }
176                 }
177                 else {
178                     // update screen
179                     BATMON_control_reg |= CLEAR_FIRST_LCD_ROW;
180                     switch (RTC_menue_position) {
181                         case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
182                             LCD_send_text("   ");
183                             break;
184                         case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
185                             LCD_send_text("  ");
186                             break;
187                         case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
188                             LCD_send_text("  ");
```

```
189                             break;
190                         case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
191                             LCD_send_text("  ");
192                             break;
193                         case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
194                             LCD_send_text("  ");
195                             break;
196                         case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
197                             LCD_send_text("   ");
198                             break;
199                         case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
200                             LCD_send_text("    ");
201                             break;
202                     }
203                 }
204             }
205             break;
206         //-------------------------------------------------------------------------------
207         default: break;
208     }
209     #ifdef PIN_DEBUG
210         TP_INL_OFF;
211     #endif
212     IFG1 &= ~WDTIFG;                            // clear watchdog interrupt flag
213 }
214 //-------------------------------------------------------------------------------------
```

Programmausdruck C.33: *Basisstation „isr_wdt.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        LCD16x2 functions
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               10/28/2007
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  22/07/2011
15
16     File:               lcd16x2.h
17 -------------------------------------------------------------------------------------*/
18
19 /*-------------------------------------------------------------------------------------
20     Headerfiles
21 -------------------------------------------------------------------------------------*/
22 #include <main.h>
23 #include <msp430x16x.h>
24 #include "headerfiles/adc.h"
25 #include "headerfiles/lcd16x2.h"
26 #include "headerfiles/globals.h"
27
28 /*-------------------------------------------------------------------------------------
29     x ms delay loop
30 -------------------------------------------------------------------------------------*/
31 void LCD_delay_ms(unsigned int LCD_delay_ms)
32 /*-------------------------------------------------------------------------------------*/
33 {
34     unsigned int LCD_delay_loop;
35
36     switch (uC_FREQUENCY)
37     {
38         case MHZ_1: LCD_delay_loop = 73;    //(88*b) cycles (for 1MHz)
39                     break;
40         case MHZ_2: LCD_delay_loop = 150;   //(175*b) cycles (for 2MHz)
41                     break;
42         case MHZ_4: LCD_delay_loop = 304;   //(350*b) cycles (for 4MHz)
43                     break;
44         case MHZ_8: LCD_delay_loop = 610;   //(610*b) cycles (for 8MHz)
45                     break;
46         default: break;
47     }
48
49     for (; LCD_delay_ms>0; LCD_delay_ms--)
50     {
51         LCD_delay(LCD_delay_loop);
52     }
53 }
54
55 /*-------------------------------------------------------------------------------------
56     delay loop
57     counts x times
58 -------------------------------------------------------------------------------------*/
59 void LCD_delay(unsigned int LCD_delay_loop)
60 /*-------------------------------------------------------------------------------------*/
61 {
62     for(; LCD_delay_loop>0; LCD_delay_loop--) _NOP();
```

```
63  }
64
65  /*---------------------------------------------------------------------------
66      Init 16x2LCD
67  ---------------------------------------------------------------------------*/
68  void LCD_init(void)
69  /*---------------------------------------------------------------------------*/
70  {
71      LCD_DATA_PORT &= ~LCD_RS_PIN;       // sets LCD in COMMAND MODE
72      LCD_delay_ms(15);                   // Delay ca.15ms
73
74      LCD_DATA_PORT |= BIT4 | BIT5;       // D7-D4 = 0011
75      LCD_DATA_PORT &= ~BIT6 & ~BIT7;
76      LCD_enable();
77      LCD_delay_ms(5);
78      LCD_enable();
79      LCD_delay_ms(1);
80      LCD_enable();
81      LCD_delay_ms(1);
82      LCD_DATA_PORT &= ~BIT4;             // D7-D4 = 0010
83      LCD_enable();
84
85      LCD_send_cmd(LCD_ON);               //  switch LCD on
86      LCD_send_cmd(LCD_CLR);              //  clear LCD
87      LCD_send_cmd(LCD_LINE1);            //  set cursor to x=0, y=0
88  }
89
90
91  /*---------------------------------------------------------------------------
92      Toggle LCD_E pin
93  ---------------------------------------------------------------------------*/
94  void LCD_enable(void)
95  /*---------------------------------------------------------------------------*/
96  {
97      LCD_DATA_PORT |= LCD_E_PIN;
98      _NOP();
99      _NOP();
100     LCD_DATA_PORT &= ~LCD_E_PIN;
101 }
102
103
104 /*---------------------------------------------------------------------------
105     Send command to LCD
106 ---------------------------------------------------------------------------*/
107 void LCD_send_cmd (unsigned char LCD_out_byte)
108 /*---------------------------------------------------------------------------*/
109 {
110     unsigned char LCD_tmp;
111
112     LCD_delay_ms(1);
113     LCD_tmp = LCD_out_byte & 0xf0;      // get upper nibble
114     LCD_DATA_PORT &= 0x0f;
115     LCD_DATA_PORT |= LCD_tmp;           // send CMD to LCD
116     LCD_DATA_PORT &= ~LCD_RS_PIN;       // sets LCD in COMMAND MODE
117     LCD_enable();
118     LCD_tmp = LCD_out_byte & 0x0f;
119     LCD_tmp = LCD_tmp << 4;             // get down nibble
120     LCD_DATA_PORT &= 0x0f;
121     LCD_DATA_PORT |= LCD_tmp;
122     LCD_DATA_PORT &= ~LCD_RS_PIN;       // sets LCD in COMMAND MODE
123     LCD_enable();
124 }
```

```
127 /*-------------------------------------------------------------------------------------
128     Send DATA to LCD
129 -------------------------------------------------------------------------------------*/
130 void LCD_send_data (unsigned char LCD_out_byte)
131 /*-------------------------------------------------------------------------------------*/
132 {
133     unsigned char LCD_tmp;

135     LCD_delay_ms(1);
136     LCD_tmp = LCD_out_byte & 0xf0;      // get upper nibble
137     LCD_DATA_PORT &= 0x0f;
138     LCD_DATA_PORT |= LCD_tmp;           // send DATA to LCD
139     LCD_DATA_PORT |= LCD_RS_PIN;        // sets LCD in DATA MODE
140     LCD_enable();
141     LCD_tmp = LCD_out_byte & 0x0f;
142     LCD_tmp = LCD_tmp << 4;             // get lower nibble
143     LCD_DATA_PORT &= 0x0f;
144     LCD_DATA_PORT |= LCD_tmp;
145     LCD_DATA_PORT |= LCD_RS_PIN;        // sets LCD in DATA MODE
146     LCD_enable();
147 }

149 /*-------------------------------------------------------------------------------------
150     Send string to LCD
151 -------------------------------------------------------------------------------------*/
152 void LCD_send_text(char *out_string)
153 /*-------------------------------------------------------------------------------------*/
154 {
155     while (*out_string)                 // *out_string != '\0'
156     {
157         LCD_send_data(*out_string);     // send byte to LCD
158         out_string++;
159     }
160 }


163 /*-------------------------------------------------------------------------------------
164     send date
165 -------------------------------------------------------------------------------------*/
166 void LCD_send_time(void)
167 /*-------------------------------------------------------------------------------------*/
168 {
169     unsigned char time_char;

171     LCD_send_cmd(LCD_LINE2);
172     LCD_send_text("              >>");

174     LCD_send_cmd(LCD_RTC_WDAY);
175     LCD_send_text(RTC_days[RTC_values.wday]);

177     time_char = (RTC_values.hr / 10) + 48;
178     LCD_send_data(time_char);
179     time_char = (RTC_values.hr % 10) + 48;
180     LCD_send_data(time_char);
181     LCD_send_data(':');

183     time_char = (RTC_values.min / 10) + 48;
184     LCD_send_data(time_char);
185     time_char = (RTC_values.min % 10) + 48;
186     LCD_send_data(time_char);
187     LCD_send_data(':');
```

```
189     time_char = (RTC_values.sec / 10) + 48;
190     LCD_send_data(time_char);
191     time_char = (RTC_values.sec % 10) + 48;
192     LCD_send_data(time_char);
193 }


196 /*---------------------------------------------------------------------------
197     send date
198 ---------------------------------------------------------------------------*/
199 void LCD_send_date(void)
200 /*---------------------------------------------------------------------------*/
201 {
202     unsigned char time_char;

204     LCD_send_cmd(LCD_LINE2);
205     LCD_send_text("                ");

207     LCD_send_cmd(LCD_RTC_DAY);
208     time_char = (RTC_values.day / 10) + 48;
209     LCD_send_data(time_char);
210     time_char = (RTC_values.day % 10) + 48;
211     LCD_send_data(time_char);

213     LCD_send_data('-');
214     LCD_send_text(RTC_months[RTC_values.month]);

216     LCD_send_data('-');
217     time_char = RTC_values.year / 1000;
218     LCD_send_data(time_char+48);
219     time_char = (RTC_values.year / 100) - ((RTC_values.year/1000) * 10);
220     LCD_send_data(time_char+48);
221     time_char = (RTC_values.year / 10) - ((RTC_values.year/100) * 10);
222     LCD_send_data(time_char+48);
223     time_char = RTC_values.year % 10;
224     LCD_send_data(time_char+48);
225 }


228 /*---------------------------------------------------------------------------
229     send sensor address + voltage modified by A.H. and S.P.
230 ---------------------------------------------------------------------------*/
231 void LCD_send_sensor(unsigned char address_value, unsigned char voltage_value_high,
232                      unsigned char voltage_value_low)
233 /*---------------------------------------------------------------------------*/
234 {
235     unsigned char address_hundred, address_tenner, address_ones;
236     unsigned char voltage_ones, voltage_dezi, voltage_cent;
237     unsigned long voltage_work;
238     unsigned long voltage_complete;
239     voltage_work = voltage_value_high;
240     voltage_work = voltage_work << 8;
241     voltage_work |= voltage_value_low;

243     address_hundred = address_value / 100;
244     address_value %= 100;
245     address_tenner = address_value / 10;
246     address_value %= 10;
247     address_ones = address_value / 1;
248     voltage_work = voltage_work / 4.096;
249     voltage_complete = voltage_work;
250     if ((voltage_work % 10) < 5){
251             voltage_work = (voltage_work + 10);
```

```
252       }
253
254       voltage_ones = (unsigned char)(voltage_work / 1000);
255       voltage_work %= 1000;
256       voltage_dezi = (unsigned char)(voltage_work / 100);
257       voltage_work %= 100;
258       voltage_cent = (unsigned char)(voltage_work / 10);
259
260       LCD_send_cmd(LCD_LINE1);
261
262       LCD_send_text("Sens ");
263       LCD_send_data(address_hundred+48);
264       LCD_send_data(address_tenner+48);
265       LCD_send_data(address_ones+48);
266       LCD_send_text(" ");
267       if ((voltage_complete) < ((CELL_VOLT_MIN*10)+100))
268       {
269           if ((voltage_complete) > (CELL_VOLT_MIN*10))
270           {
271               LCD_send_text("lowvolt");
272           }
273           else
274           {
275               LCD_send_text("depth d");
276           }
277       }
278       else
279       {
280           LCD_send_data(voltage_ones+48);
281           LCD_send_text(",");
282           LCD_send_data(voltage_dezi+48);
283           LCD_send_data(voltage_cent+48);
284           LCD_send_text("V  ");
285       }
286
287   }
288   /*-------------------------------------------------------------------------------------
289       send sensor address + voltage modified by N.J.
290   -------------------------------------------------------------------------------------*/
291   void LCD_send_sensor_v2(uint8_t address_value, uint16_t voltage_value)
292   /*-------------------------------------------------------------------------------------*/
293   {
294       LCD_send_cmd(LCD_LINE1);
295
296       LCD_send_text("Sens ");
297       LCD_send_data((address_value / 100) + 48);
298       address_value  %= 100;
299       LCD_send_data((address_value / 10) + 48);
300       address_value  %= 10;
301       LCD_send_data((address_value) + 48);
302       LCD_send_text(" ");
303
304       LCD_send_data((uint8_t)((voltage_value / 1000) + 48));
305       LCD_send_text(",");
306       voltage_value %= 1000;
307       LCD_send_data((uint8_t)((voltage_value / 100) + 48));
308       voltage_value %= 100;
309       LCD_send_data((uint8_t)((voltage_value / 10) + 48));
310       voltage_value %= 10;
311       LCD_send_data((uint8_t)((voltage_value) + 48));
312       LCD_send_text("V ");
313   }
314   /*-------------------------------------------------------------------------------------
```

```
315     send sensor address/voltage from senosornmb by A.H. modified by N.J.
316 ------------------------------------------------------------------------------*/
317 void LCD_show_voltage(unsigned char sensornmb)
318 /*------------------------------------------------------------------------------*/
319 {
320     unsigned long voltage_work, address_work, sensornmb_work, voltage_sum;
321     static unsigned long voltage_old, sensornmb_old, AH_last_minute_old;
322     static unsigned long current_old;
323     unsigned long current_work;
324     unsigned char o, Battery_octa, U_CAP;

326     voltage_work  = SENSOR_last_data[3][(sensornmb-1)];
327     voltage_work <<= 8;
328     voltage_work |= SENSOR_last_data[4][(sensornmb-1)];

330     if((voltage_work != voltage_old) ||
331        (sensornmb    != sensornmb_old) ||
332        (current_old  != g_currmeas.last_data)) {
333         // show active sensor address + voltage ----------------------------------------
334         if((sensornmb > 0) && (sensornmb <= SENSOR_active)) {
335             if((sensornmb != sensornmb_old) || (voltage_work != voltage_old)) {
336                 voltage_old  = voltage_work;
337                 address_work = SENSOR_address[sensornmb-1];
338                 sensornmb_work = sensornmb;

340                 voltage_work = voltage_work / 10;       //scale voltage value

342                 LCD_send_cmd(LCD_LINE1);

344                 LCD_send_text("Sens");
345                 LCD_send_data((sensornmb_work / 10) + 48);
346                 sensornmb_work %= 10;
347                 LCD_send_data((sensornmb_work) + 48);

349                 LCD_send_text("/");
350                 LCD_send_data((address_work / 100) + 48);
351                 address_work  %= 100;
352                 LCD_send_data((address_work / 10) + 48);
353                 address_work  %= 10;
354                 LCD_send_data((address_work) + 48);
355                 LCD_send_text(" ");

357                 LCD_send_data((uint8_t)((voltage_work / 1000) + 48));
358                 LCD_send_text(",");
359                 voltage_work %= 1000;
360                 LCD_send_data((uint8_t)((voltage_work / 100) + 48));
361                 voltage_work %= 100;
362                 LCD_send_data((uint8_t)((voltage_work / 10) + 48));
363                 LCD_send_text("V");
364             }
365         }
366         // show overall voltage ---------------------------------------------------------
367         else if(sensornmb > SENSOR_active) {
368             if(sensornmb == SENSOR_active+1) {
369                 if((sensornmb != sensornmb_old)||(voltage_work != voltage_old)) {
370                     voltage_old = voltage_work;
371                     voltage_sum = 0;

373                     for(o=0; o<SENSOR_active; o++) {    // summerize cell voltages
374                         voltage_work  = SENSOR_last_data[3][o];
375                         voltage_work <<= 8;
376                         voltage_work |= SENSOR_last_data[4][o];
377                         voltage_sum  += voltage_work;
```

```
378                     }
379                     voltage_sum = voltage_sum / 10;     // scale voltage value

381                     LCD_send_cmd(LCD_LINE1);
382                     LCD_send_text("U ges: ");

384                     LCD_send_data((uint8_t)((voltage_sum / 10000) + 48));
385                     voltage_sum %= 10000;
386                     LCD_send_data((uint8_t)((voltage_sum / 1000) + 48));
387                     LCD_send_text(",");
388                     voltage_sum %= 1000;
389                     LCD_send_data((uint8_t)((voltage_sum / 100) + 48));
390                     voltage_sum %= 100;
391                     LCD_send_data((uint8_t)((voltage_sum / 10) + 48));
392                     voltage_sum %= 10;
393                     LCD_send_data((uint8_t)((voltage_sum) + 48));
394                     LCD_send_text("V  ");
395                 }
396             }
397             // show consumption last minute ---------------------------------------------
398             else if(sensornmb == SENSOR_active+2) {
399                 if((sensornmb != sensornmb_old) ||
400                    (g_currah.AH_last_minute != AH_last_minute_old)) {

402                     current_old = g_currmeas.last_data;
403                     AH_last_minute_old = g_currah.AH_last_minute;

405                     LCD_send_cmd(LCD_LINE1);
406                     LCD_send_text("Co: ");
407                     if(g_currah.AH_last_minute_IN_OUT == 1) {
408                         current_work = g_currah.AH_last_minute;
409                         LCD_send_text(" ");
410                     }
411                     else {
412                         current_work = g_currah.AH_last_minute;
413                         LCD_send_text("-");
414                     }

416                     LCD_send_data((uint8_t)((current_work / 100000) + 48));
417                     current_work  %= 100000;
418                     LCD_send_data((uint8_t)((current_work / 10000) + 48));
419                     current_work  %= 10000;
420                     LCD_send_data((uint8_t)((current_work / 1000) + 48));
421                     current_work  %= 1000;
422                     LCD_send_text(",");
423                     LCD_send_data((uint8_t)((current_work / 100) + 48));
424                     current_work  %= 100;
425 //                  LCD_send_data((uint8_t)((current_work / 10) + 48));
426 //                  current_work  %= 10;
427 //                  LCD_send_data((uint8_t)((current_work) + 48));
428                     LCD_send_text("Ah/min");
429                 }
430             }
431             // show actual batterie current ---------------------------------------------
432             else if(sensornmb == SENSOR_active+3) {
433                 if((sensornmb != sensornmb_old) ||
434                    (current_old != g_currmeas.last_data)) {

436                     current_old = g_currmeas.last_data;

438                     LCD_send_cmd(LCD_LINE1);
439                     LCD_send_text(" Ibat :");
440                     if(g_currmeas.current_in_out == CURRENT_DIR_POS) {
```

```
441                         LCD_send_text(" ");
442                     }
443                     else {
444                         LCD_send_text("-");
445                     }
446                     current_work = g_currmeas.last_data;
447                     if((current_work % 10) >= 5) {
448                         current_work += 5;
449                     }

451                     LCD_send_data((uint8_t)((current_work / 1000) + 48));
452                     current_work  %= 1000;
453                     LCD_send_data((uint8_t)((current_work / 100) + 48));
454                     current_work  %= 100;
455                     LCD_send_data((uint8_t)((current_work / 10) + 48));
456                     LCD_send_text(",");
457                     current_work  %= 10;
458                     LCD_send_data((uint8_t)((current_work) + 48));
459                     LCD_send_text("A  ");
460                 }
461             }
462             // show batterie capacity ---------------------------------------------------
463             else if(sensornmb == SENSOR_active+4) {
464                 if((sensornmb != sensornmb_old) ||
465                    (AH_last_minute_old != g_currah.AH_last_minute)) {

467                     LCD_send_cmd(LCD_LINE1);
468                     LCD_send_text(" Cbat: ");
469                     AH_last_minute_old = g_currah.AH_last_minute;
470                     current_work = g_currah.AH_MASTER*44;
471                     LCD_send_text(" ");

473                     AH_last_minute_old = current_work;
474                     if(current_work%10>=5) {
475                         current_work += 5;
476                     }

478                     LCD_send_data((current_work / 10000) + 48);
479                     current_work %= 10000;
480                     LCD_send_data((current_work / 1000) + 48);
481                     current_work %= 1000;
482                     LCD_send_data((current_work / 100) + 48);
483                     LCD_send_text(",");
484                     current_work %= 100;
485                     LCD_send_data((current_work / 10) + 48);
486                     LCD_send_text("Ah ");
487                 }
488             }
489             // show batterie capacity segment -------------------------------------------
490             else if(sensornmb == SENSOR_active+5) {
491                 if((sensornmb != sensornmb_old) ||
492                    (AH_last_minute_old != g_currah.AH_last_minute)) {

494                     AH_last_minute_old = g_currah.AH_last_minute;
495                     Battery_octa = (g_currah.AH_MASTER*0.44 / BATT_CAPACITY)*9;
496                     LCD_send_cmd(LCD_LINE1);
497                     LCD_send_text(" Bat :");
498                     LCD_send_data(91);
499                     if(Battery_octa==0) {
500                         LCD_send_text(" !LOW!  ");
501                     }
502                     else {
503                         for(o=0; o<Battery_octa && o<8; o++) {
```

```
504                             LCD_send_data(1);
505                         }
506                         for(o=Battery_octa; o<8; o++) {
507                             LCD_send_text(" ");
508                         }
509                     }
510                     LCD_send_data(93);
511                 }
512             }
513             // show battery capacity voltage based in percent --------------------------
514             else {
515                 if((sensornmb != sensornmb_old) ||
516                    (AH_last_minute_old != g_currah.AH_last_minute)) {

518                     AH_last_minute_old = g_currah.AH_last_minute;
519                     U_CAP = g_currah.U_MASTER_CAPACITY;

521                     LCD_send_cmd(LCD_LINE1);
522                     LCD_send_text(" Cap_U:");
523                     LCD_send_data((U_CAP / 100) + 48);
524                     U_CAP %= 100;
525                     LCD_send_data((U_CAP / 10) + 48);
526                     U_CAP %= 10;
527                     LCD_send_data((U_CAP) + 48);
528                     LCD_send_text("% ");
529                     LCD_send_text("    ");
530                 }
531             }
532         }
533         // battery meter percent ---------------------------------------------------
534         else {
535             if((sensornmb != sensornmb_old) ||
536                (AH_last_minute_old != g_currah.AH_last_minute)) {

538                 AH_last_minute_old = g_currah.AH_last_minute;
539                 Battery_octa = (g_currah.AH_MASTER*0.44/BATT_CAPACITY/1)*100;

541                 LCD_send_cmd(LCD_LINE1);
542                 LCD_send_text(" Bat%  ");
543                 if((Battery_octa / 100) == 1) {
544                     LCD_send_text("100");
545                 }
546                 else {
547                     LCD_send_data((Battery_octa / 100) + 48);
548                     Battery_octa %= 100;
549                     LCD_send_data((Battery_octa / 10) + 48);
550                     Battery_octa %= 10;
551                     LCD_send_data((Battery_octa) + 48);
552                 }
553                 LCD_send_text("%");
554                 LCD_send_text("     ");
555             }
556         }
557         // ------------------------------------------------------------------------------
558         sensornmb_old = sensornmb;
559     }
560 }
561 //-----------------------------------------------------------------------------------
```

Programmausdruck C.34: *Basisstation „lcd16x2.c“*

```
/*-------------------------------------------------------------------------------------
    Project:            BATSEN
    Discription:        LCD16x2 module header
    Used components:    MSP430-169STK

    Author:             Stephan Plaschke
    Date:               10/28/2007
    Last update:        10/28/2007

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

    Modified by:        Niels Jegenhorst
    Last modification:  02/05/2011

    File:               lcd16x2.h
-------------------------------------------------------------------------------------*/

#ifndef LCD16X2_H_
#define LCD16X2_H_

/*-------------------------------------------------------------------------------------
    Headerfiles
-------------------------------------------------------------------------------------*/
#include <headerfiles/rtc.h>

/*-------------------------------------------------------------------------------------
    Defined values, change for each project
-------------------------------------------------------------------------------------*/

#define   LCD_E_PIN         BIT1
#define   LCD_RS_PIN        BIT3
#define   LCD_LIGHT_PIN     BIT0
#define   LCD_DATA_PORT     P4OUT

/*-------------------------------------------------------------------------------------
    Defined values, equal in each project, don't change
-------------------------------------------------------------------------------------*/

#define   LCD_LIGHT_ON      (LCD_DATA |= LCD_LIGHT_PIN) // sets pin4.0 HIGH
#define   LCD_LIGHT_OFF     (LCD_DATA &= ~LCD_LIGHT_PIN)    // clears pin4.0

#define   LCD_ON            0x0c    //  LCD control command, ON
#define   LCD_OFF           0x08    //  LCD control command, OFF
#define   LCD_CLR           0x01    //  LCD control command, CLEARSCREEN
#define   LCD_LINE1         0x80
    //  LCD control command, sets cursor on y=0,x=0
#define   LCD_LINE2         0xc0
    //  LCD control command, sets cursor on y=1,x=0
#define   LCD_ENDLINE1      0x8f
    //  LCD control command, sets cursor on y=0,x=15
#define   LCD_ENDLINE2      0xcf
    //  LCD control command, sets cursor on y=1,x=15

/*-------------------------------------------------------------------------------------
    Prototypes
-------------------------------------------------------------------------------------*/

void LCD_delay_ms(unsigned int);
void LCD_delay(unsigned int);
void LCD_send_cmd (unsigned char);
void LCD_send_data (unsigned char);
```

```
63  void LCD_send_text(char *);
64  void LCD_enable(void);
65  void LCD_init(void);
66  void LCD_send_time(void);
67  void LCD_send_date(void);
68  void LCD_send_sensor(unsigned char, unsigned char, unsigned char);
69  void LCD_send_sensor_v2(uint8_t address_value, uint16_t voltage_value);
70  void LCD_show_voltage(unsigned char);

72  #endif /*16X2LCD_H_*/
73  //-----------------------------------------------------------------------------------------
```

Programmausdruck C.35: *Basisstation „lcd16x2.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Specialfunctions for the MSP430x1232 and MSP430x169
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               10/28/2007
8      Last update:        09/05/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  11/07/2011
15
16     File:               msp_functions.h
17 -------------------------------------------------------------------------------------------*/
18
19 /*-------------------------------------------------------------------------------------------
20     Headerfiles
21 -------------------------------------------------------------------------------------------*/
22 #include <main.h>
23 #include <stdio.h>
24 #include <flash.h>
25 #include <stdlib.h>
26 #include <string.h>
27 #include <signal.h>
28 #include <msp430x16x.h>
29 #include "headerfiles/lcd16x2.h"
30 #include "headerfiles/uart_menu.h"
31 #include "headerfiles/msp_functions.h"
32 #include "headerfiles/adc.h"
33 #include "headerfiles/isr_timera.h"
34 #include "headerfiles/reader_13p56mhz.h"
35 #include "headerfiles/system_handling.h"
36 #include "headerfiles/globals.h"
37
38 /*-------------------------------------------------------------------------------------------
39     Statics
40 -------------------------------------------------------------------------------------------*/
41 #define BATMON_MENUE_MSG_MAX            21
42 #define BATMON_MENUE_MSG_LINE2_MAX      8
43
44 static char *BATMON_menu_msg[BATMON_MENUE_MSG_MAX] = {
45         " Show sensors   ",     // [0]
46         " Start recording",
47         " Stop recording ",
48         " Load contr.data",
49         " Save contr.data",
50         " Set date/time  ",     // [5]
51         " Backlight      ",
52         " Current tara   ",
53         " Current invert ",
54         " Scan sensors   ",
55         " Calibrate temp ",     // [10]
56         " ZS Wakeup cw   ",
57         " ZS Wakeup pulse",
58         " ZS Shutdown all",
59         " ZS Synch       ",
60         " ZS Set time IV ",     // [15]
61         " ZS Set time SA ",
62         " ZS Set time TX ",
```

```
63          " ZS Send set 1  ",
64          " ZS Send set 2  ",
65          " ZS Balance     "      // [20]
66  };
67  static char *BATMON_menu_msg_line2[BATMON_MENUE_MSG_LINE2_MAX] = {
68          "NEXT   OK   EXIT",     // [0]
69          "YES    NO       ",
70          "OFF    ON       ",
71          "6      12   MORE",
72          "24     40   EXIT",
73          "+      OK   EXIT",     // [5]
74          "OFF    ON   EXIT",
75          "SEND        EXIT"      // [7]
76  };
77  /*---------------------------------------------------------------------------
78      Functions
79  ---------------------------------------------------------------------------*/
80
81  /*---------------------------------------------------------------------------
82      Set XT2 to selected frequency
83      clk_multi is the multiplier of the  A_clk (32768KHz)
84      Error_code:
85      0   ->  no error
86      1   ->  external oscillator error
87      2   ->  clock wrong, set to 4MHZ
88  ---------------------------------------------------------------------------*/
89  unsigned char XT2_set (unsigned int clk_multi)
90  /*---------------------------------------------------------------------------*/
91  {
92      unsigned int Max_turns = 1000;
93      unsigned char i, Error_code = 0;
94
95      _BIC_SR(OSCOFF);                // switch LFXT on
96      BCSCTL1 &= ~XT2OFF;             // XT2on
97
98      do {
99          IFG1 &= ~OFIFG;                 // Clear OSCFault flag
100         for(i = 0xFF; i > 0; i--) {};   // Time for flag to set
101
102         if(!(--Max_turns))  {           // stop after 1k tries, error with ext osc!
103             Error_code = 1;
104             break;
105         }
106     } while((IFG1 & OFIFG));        // OSCFault flag still set?
107
108     BCSCTL2 |= SELM_2 | SELS;       // MCLK = SMCLK = XT2 (safe)
109
110     switch (clk_multi) {
111         case MHZ_8:
112             BCSCTL2 &= ~(DIVS_3 | DIVM_3);  // SMCLK&MCLK = XT2 / 1 (now used)
113             break;
114         case MHZ_4:
115             BCSCTL2 |= DIVS_1 | DIVM_1;     // SMCLK&MCLK = XT2 / 2
116             break;
117         case MHZ_2:
118             BCSCTL2 |= DIVS_2 | DIVM_2;     // SMCLK&MCLK = XT2 / 4
119             break;
120         case MHZ_1:
121             BCSCTL2 |= DIVS_3 | DIVM_3;     // SMCLK&MCLK = XT2 / 8
122             break;
123         default:
124             Error_code = 2;
125             BCSCTL2 |= DIVS_1 | DIVM_1;     // SMCLK&MCLK = XT2 / 2
```

```
126             break;
127     }
128     return (Error_code);
129 }

131 /*-------------------------------------------------------------------------------
132     putchar() defined for use with printf
133     PRINTF_LCD :    output on LCD of the MSP430x169STK board
134     PRINTF_UART:    output through RS232
135 -------------------------------------------------------------------------------*/
136 int putchar(int c)
137 //-------------------------------------------------------------------------------
138 {
139     if(g_usart_mode == MODE_UART) {
140         UART0_send((unsigned char)c);
141     }
142     return(c);
143 }

145 //*******************************************************************************/
146 /*-------------------------------------------------------------------------------
147     MSP430x169 specific functions
148 -------------------------------------------------------------------------------*/

150 /*-------------------------------------------------------------------------------
151     PORTS init
152 -------------------------------------------------------------------------------*/
153 void PORTS_init(void)
154 /*-------------------------------------------------------------------------------*/
155 {
156     P1OUT = PORT1_OUT;
157     P1IFG = 0x00;
158     P1SEL = PORT1_SEL;      // Port1 I/O Funktion
159     P1DIR = PORT1_DIR;      // Port1 Ausgang
160     P1IES = PORT1_IES;      // Port1 interrupt edge select
161     P1IE  = PORT1_IE;       // Port1 interrupt enable

163     P2OUT = 0x00;
164     P2IFG = 0x00;
165     P2SEL = PORT2_SEL;      // Port2 I/O Funktion
166     P2DIR = PORT2_DIR;      // Port2 Ausgang
167     P2IES = PORT2_IES;      // Port2 interrupt edge select
168     P2IE  = PORT2_IE;       // Port2 interrupt enable

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

173     P4OUT = PORT4_SEL;      // LCD ini, set PORT4 as output
174     P4DIR = PORT4_DIR;

176     P5SEL = PORT5_SEL;      // Port5 I/O Funktion
177     P5DIR = PORT5_DIR;      // Port5 Ausgang

179     P6SEL = PORT6_SEL;      // P6.x ADC option select
180     P6DIR = PORT6_DIR;
181 }

183 /*-------------------------------------------------------------------------------
184     Init USART0, automatic baud rate generation for selected uC frequency
185 -------------------------------------------------------------------------------*/
186 void USART0_init(unsigned char USART0_IRQ, unsigned char USART0_MODE)
187 /*-------------------------------------------------------------------------------*/
188 {
```

```
189       g_usart_mode = MODE_UART;   // Indicate USART in UART-Mode
190       P3OUT &= ~BIT0;                 // clear STE
191       P3SEL &= ~(BIT1 | BIT2 | BIT3); // clear SPI I/O
192       P3SEL  = (BIT4 | BIT5);         // set  UART I/O

194       UCTL0  = (SWRST | CHAR);    // set 8bit, none parity and 1 stoppbit
195       UTCTL0 = SSEL_2;            // set SMCLK as USART0_CLK

197       switch (uC_FREQUENCY) {     // set UART to selected mode with selected
198                                   // uC frequency
199           case MHZ_1:
200               switch (USART_BAUD) {
201                   case BAUD_19k2: UBR00 = 0x34;           // see manual
202                       UBR10 = 0x00;
203                       UMCTL0 = 0x20;
204                       break;
205                   case BAUD_9600: UBR00 = 0x68;           // see manual
206                       UBR10 = 0x00;
207                       UMCTL0 = 0x04;
208                       break;
209                   case BAUD_4800: UBR00 = 0xd0;           // see manual
210                       UBR10 = 0x00;
211                       UMCTL0 = 0x92;
212                       break;
213                   default: UBR00 = 0x68;                  // see manual
214                       UBR10 = 0x00;
215                       UMCTL0 = 0x04;
216                   break;
217               };
218               break;
219           case MHZ_2:
220               switch (USART_BAUD) {
221                   case BAUD_115k2: UBR00 = 0x11;          // see manual
222                       UBR10 = 0x00;
223                       UMCTL0 = 0x52;
224                       break;
225                   case BAUD_19k2: UBR00 = 0x68;           // see manual
226                       UBR10 = 0x00;
227                       UMCTL0 = 0x04;
228                       break;
229                   case BAUD_9600: UBR00 = 0xD0;           // see manual
230                       UBR10 = 0x00;
231                       UMCTL0 = 0x92;
232                       break;
233                   case BAUD_4800: UBR00 = 0xA0;           // see manual
234                       UBR10 = 0x01;
235                       UMCTL0 = 0x6D;
236                       break;
237                   default: UBR00 = 0x68;                  // see manual
238                       UBR10 = 0x00;
239                       UMCTL0 = 0x04;
240                   break;
241               };
242               break;
243           case MHZ_4:
244               switch (USART_BAUD) {
245                   case BAUD_115k2: UBR00 = 0x22;          // see manual
246                       UBR10 = 0x00;
247                       UMCTL0 = 0xDD;
248                       break;
249                   case BAUD_19k2: UBR00 = 0xD0;           // see manual
250                       UBR10 = 0x00;
251                       UMCTL0 = 0x92;
```

```
252                     break;
253                 case BAUD_9600: UBR00 = 0xA0;           // see manual
254                     UBR10 = 0x01;
255                     UMCTL0 = 0x6D;
256                     break;
257                 case BAUD_4800: UBR00 = 0x41;           // see manual
258                     UBR10 = 0x03;
259                     UMCTL0 = 0x92;
260                     break;
261                 default: UBR00 = 0x68;              // see manual
262                     UBR10 = 0x00;
263                     UMCTL0 = 0x04;
264                 break;
265             };
266             break;
267         case MHZ_8:
268             switch (USART_BAUD) {
269                 case BAUD_115k2: UBR00 = 0x45;          // see manual
270                     UBR10 = 0x00;
271                     UMCTL0 = 0xAA;
272                     break;
273                 case BAUD_19k2: UBR00 = 0xa0;           // see manual
274                     UBR10 = 0x01;
275                     UMCTL0 = 0x6D;
276                     break;
277                 case BAUD_9600: UBR00 = 0x41;           // see manual
278                     UBR10 = 0x03;
279                     UMCTL0 = 0x92;
280                     break;
281                 case BAUD_4800: UBR00 = 0x82;           // see manual
282                     UBR10 = 0x6;
283                     UMCTL0 = 0x6D;
284                     break;
285                 default: UBR00 = 0x68;              // see manual
286                     UBR10 = 0x00;
287                     UMCTL0 = 0x04;
288                 break;
289             };
290             break;
291     }

293     ME1 &= ~(UTXE0 | URXE0 | USPIE0);       // reset ME1 register
294     switch (USART0_MODE) {
295         case RX:
296                 ME1 |= URXE0;               // enable UART0 RX
297                 break;
298         case TX:
299                 ME1 |= UTXE0;               // enable UART0 TX
300                 break;
301         case RX_TX:
302                 ME1 |= (URXE0 | UTXE0);     // enable UART0 TX&RX
303                 break;
304         default:
305                 ME1 &= ~(UTXE0 | URXE0);    // disable UART0 TX&RX
306                 break;
307     }
308     IE1   &= ~(UTXIE0 | URXIE0);            // reset IE1 register
309     UCTL0 &= ~SWRST;                        // clear SWRST
310     switch (USART0_IRQ) {
311         case RX_INT:
312                 IE1 |= URXIE0;              // enable UART0 RX ISR
313                 break;
314         case TX_INT:
```

```
315                 IE1 |= UTXIE0;              // enable UART0 TX ISR
316                 break;
317         case RX_TX_INT:
318                 IE1 |= UTXIE0+URXIE0;       // enable UART0 TX&RX ISR
319                 break;
320         default:
321                 IE1 &= ~UTXIE0+URXIE0;      // disable UART0 TX&RX ISR
322                 break;
323     }
324 }

326 /*-------------------------------------------------------------------------------------
327     send character
328 -------------------------------------------------------------------------------------*/
329 void UART0_send(unsigned char out_char)
330 /*-------------------------------------------------------------------------------------*/
331 {
332     if(g_usart_mode == MODE_UART) {
333         while(!(UTCTL0&TXEPT));     // look if USART0 still busy
334         TXBUF0 = out_char;          // write char in transmit buffer
335     }
336 }

338 /*-------------------------------------------------------------------------------------
339     Send string to USART
340 -------------------------------------------------------------------------------------*/
341 void UART0_send_text(char *out_string)
342 /*-------------------------------------------------------------------------------------*/
343 {
344     if(g_usart_mode == MODE_UART) {
345         while (*out_string) {           // *out_string != '\0'
346             UART0_send(*out_string);        // send byte to LCD
347             out_string++;
348         }
349     }
350 }

352 /*-------------------------------------------------------------------------------------
353     store receive frame
354 -------------------------------------------------------------------------------------*/
355 void RADIO_store_frame(void)
356 /*-------------------------------------------------------------------------------------*/
357 {
358     static DATA_STRUCT Data;
359     DATA_STRUCT *pData = &Data;

361     pData->C0  = RTC_values.day;
362     pData->C1  = RTC_values.month;
363     pData->C2  = (unsigned char)(RTC_values.year >> 8);
364     pData->C3  = (unsigned char)RTC_values.year;
365     pData->C4  = RTC_values.wday;
366     pData->C5  = RTC_values.hr;
367     pData->C6  = RTC_values.min;
368     pData->C7  = RTC_values.sec;
369     pData->C8  = RADIO_frame.frame_byte[3]; //Address
370     pData->C9  = RADIO_frame.frame_byte[4]; //Temp
371     pData->C10 = RADIO_frame.frame_byte[5]; //Temp
372     pData->C11 = RADIO_frame.frame_byte[6]; //Supp
373     pData->C12 = RADIO_frame.frame_byte[7]; //Supp
374     pData->C13 = RADIO_frame.frame_byte[8]; //Cell
375     pData->C14 = RADIO_frame.frame_byte[9]; //Cell
376     pData->C15 = RADIO_frame.frame_byte[10];//CRC
```

```
378     Write_Ring_Flash(pData);
379 }

381 /*-------------------------------------------------------------------------------------
382     init receive frame
383 -------------------------------------------------------------------------------------*/
384 void RADIO_init_frame(void)
385 /*-------------------------------------------------------------------------------------*/
386 {
387     static DATA_STRUCT Data;
388     DATA_STRUCT *pData;

390     pData = &Data;

392     // store time and data
393     pData->C0  = RTC_values.day;
394     pData->C1  = RTC_values.month;
395     pData->C2  = (unsigned char)(RTC_values.year >> 8);
396     pData->C3  = (unsigned char)RTC_values.year;
397     pData->C4  = RTC_values.wday;
398     pData->C5  = RTC_values.hr;
399     pData->C6  = RTC_values.min;
400     pData->C7  = RTC_values.sec;
401     pData->C8  = 0x00;
402     pData->C9  = 0x00;
403     pData->C10 = 0x00;
404     pData->C11 = 0x00;
405     pData->C12 = 0x00;
406     pData->C13 = 0x00;
407     pData->C14 = 0x00;
408     pData->C15 = 0x00;

410     // write data to flash
411     Write_Ring_Flash(pData);
412 }

414 /*-------------------------------------------------------------------------------------
415     init radio unit             by N.J.
416 -------------------------------------------------------------------------------------*/
417 void RADIO_init(void)
418 /*-------------------------------------------------------------------------------------*/
419 {
420     // initialise TIMERA0:
421         // set control register 0:
422         //  clear clock divider, clear TAR, clear count direction
423     TACTL   = TACLR;
424         // set control register 0:
425         //  clock source = SMCLK, clock divider = 8
426     TACTL  |= (TASSEL_2 | ID_3);
427         // set capture/compare control register 0:
428         //  capture on both edges, synchron capture
429     TACCTL0 = (CM1 | CM0 | CAP | SCS);
430         // set capture/compare register 1:
431     TACCR1  = START_SEQ_LENGTH_MAX + 50;
432         // set capture/compare control register 1:
433         //  compare-mode, IRQ enable
434     TACCTL1 = CCIE;

436     // starting Timer_A in continous mode is done by itself in the isr "TIMER_A0"
437 }

439 /*-------------------------------------------------------------------------------------
440     enable radio unit
```

```
441 ------------------------------------------------------------------------------*/
442 void RADIO_enable(void)
443 /*------------------------------------------------------------------------------*/
444 {
445     //  begin with new frame
446     global_string[0] = NULLTERMINATOR;
447     BATMON_control_reg &= ~VALID_DATA_RECEIVED; // clear global value
448     RADIO_rx_state = SE_SEQ_PREPARE;
449     g_rx_status    = INACTIVE;
450     g_rx_en        = TRUE;                      // indicate RX enable

452     TACTL   &= ~TAIFG;                  // clear IRQ flag
453     TACCTL0 &= ~CCIFG;                  // clear IRQ flag
454     TACCTL0 |= (CCIE | CAP);            // enable capture interrupt

456     // starting Timer_A in continous mode is done by itself in the isr "TIMER_A0"
457 }

459 /*------------------------------------------------------------------------------
460     disable radio unit
461 ------------------------------------------------------------------------------*/
462 void RADIO_disable(void)
463 /*------------------------------------------------------------------------------*/
464 {
465     TACTL   &= ~(MC1 | MC0);            // stop Timer_A
466     TACCTL0 &= ~(CCIE | CAP);           // disable interrupt
467     TACTL   |= TACLR;                   // reset timer A

469     BATMON_control_reg &= ~VALID_DATA_RECEIVED; // clear global value
470     g_rx_status    = INACTIVE;
471     RADIO_rx_state = SE_SEQ_PREPARE;
472     g_rx_en        = FALSE;                     // indicate RX disable
473     #ifdef PIN_DEBUG
474         TP_INR_OFF;
475         TP_ADC1_OFF;
476     #endif
477 }

479 /*------------------------------------------------------------------------------
480     batmon main menu
481 ------------------------------------------------------------------------------*/
482 void BATMON_menu_options(void)
483 /*------------------------------------------------------------------------------*/
484 {
485     BMON_MENUE BATMON_menu_position;
486     unsigned char BATMON_menu_counter;
487     unsigned char time_tmp, i;

489     BATMON_menu_position = BATMON_MENU_MAIN;
490     BATMON_menu_counter  = 0;

492     while(BATMON_menu_position != BATMON_MENU_EXIT) {
493         switch(BATMON_menu_position) {
494         // menue start ____________________________________________________________
495         case BATMON_MENU_START:
496             LCD_send_cmd(LCD_LINE1);
497             LCD_send_text(BATMON_menu_msg[1]);
498             LCD_send_cmd(LCD_LINE2);
499             LCD_send_text(BATMON_menu_msg_line2[1]);

501             while(1) {
502                 if(BATMON_control_reg & B1) {               // Button 1 = YES,
503                     recording_start();
```

```
504                     break;                                  // exit while loop
505                 }
506                 if(BATMON_control_reg & B2) {               // Button 2 = NO,
507                     break;                                  // exit while loop
508                 }
509             }
510             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
511             BATMON_menu_position = BATMON_MENU_EXIT;
512             break;                                          // exit switch case

514         // menue stop _____________________________________________________________________
515         case BATMON_MENU_STOP:
516             LCD_send_cmd(LCD_LINE1);
517             LCD_send_text(BATMON_menu_msg[2]);
518             LCD_send_cmd(LCD_LINE2);
519             LCD_send_text(BATMON_menu_msg_line2[1]);

521             while(1) {
522                 if(BATMON_control_reg & B1) {               // Button 1 = YES,
523                     recording_stop();
524                     break;                                  // exit while loop
525                 }
526                 if(BATMON_control_reg & B2) {               // Button 2 = NO,
527                     break;                                  // exit while loop
528                 }
529             }
530             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
531             BATMON_menu_position = BATMON_MENU_EXIT;
532             break;                                          // exit switch case

534         // menue save _____________________________________________________________________
535         case BATMON_MENU_SAVE:
536             LCD_send_cmd(LCD_LINE1);
537             LCD_send_text(BATMON_menu_msg[4]);
538             LCD_send_cmd(LCD_LINE2);
539             LCD_send_text(BATMON_menu_msg_line2[1]);

541             while(1) {
542                 if(BATMON_control_reg & B1) {               // Button 1 = YES
543                     UART_menu_write_controll();
544                     break;                                  // exit while loop
545                 }
546                 if(BATMON_control_reg & B2) {               // Button 2 = NO
547                     break;                                  // exit while loop
548                 }
549             }
550             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
551             BATMON_menu_position = BATMON_MENU_EXIT;
552             break;                                          // exit switch case

554         // menue load _____________________________________________________________________
555         case BATMON_MENU_LOAD:
556             LCD_send_cmd(LCD_LINE1);
557             LCD_send_text(BATMON_menu_msg[3]);
558             LCD_send_cmd(LCD_LINE2);
559             LCD_send_text(BATMON_menu_msg_line2[1]);

561             while(1) {
562                 if(BATMON_control_reg & B1) {               // Button 1 = YES
563                     UART_menu_read_controll();
564                     break;                                  // exit while loop
565                 }
566                 if(BATMON_control_reg & B2) {               // Button 2 = NO,
```

```
567                     break;                                  // exit while loop
568                 }
569             }
570             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
571             BATMON_menu_position = BATMON_MENU_EXIT;
572             break;                                          // exit switch case

574         // menue rtc ___________________________________________________________________
575         case BATMON_MENU_RTC:
576             LCD_send_cmd(LCD_LINE1);                        // clear first LCD row
577             LCD_send_text("                ");
578             RTC_init();                                     // enter RTC init mode

580             BATMON_menu_position = BATMON_MENU_EXIT;
581             break;                                          // exit switch case

583         // menue show __________________________________________________________________
584         case BATMON_MENU_SHOW:
585             time_tmp = RTC_values.sec + 5;

587             if (time_tmp > 59)
588                 time_tmp -= 60;

590             LCD_send_cmd(LCD_LINE1);
591             LCD_send_text("                ");
592             LCD_send_cmd(LCD_LINE2);
593             LCD_send_text("                ");
594             LCD_send_cmd(LCD_LINE1);

596             while(time_tmp != RTC_values.sec) {
597                 for(i=0; i<SENSOR_active; i++) {
598                     // jump to second line if maximum of line one reached
599                     if(((i%16) == 0) && (i!=0))
600                         LCD_send_cmd(LCD_LINE2);
601                     // i/8 = byte-, i%8 = bit position in array
602                     if((SENSOR_addr_req[(i/8)] >> (i%8)) & 0x01)
603                         LCD_send_data('.');
604                     else
605                         LCD_send_data('*');
606                 }
607                 LCD_send_cmd(LCD_LINE1);
608             }

610             LCD_send_cmd(LCD_LINE1);
611             LCD_send_text("                ");

613             BATMON_menu_position = BATMON_MENU_EXIT;
614             break;                                          // exit switch case

616         // menue main __________________________________________________________________
617         case BATMON_MENU_MAIN:
618             LCD_send_cmd(LCD_LINE1);
619             LCD_send_text(BATMON_menu_msg[0]);
620             LCD_send_cmd(LCD_LINE2);
621             LCD_send_text(BATMON_menu_msg_line2[0]);

623             while(1) {
624                 // Button B1 pressed, scroll through menu
625                 if(BATMON_control_reg & B1) {
626                     BATMON_menu_counter++;
627                     if (BATMON_menu_counter > (BATMON_MENUE_MSG_MAX-1)) {
628                         BATMON_menu_counter = 0;
629                     }
```

```
630                     LCD_send_cmd(LCD_LINE1);
631                     LCD_send_text(BATMON_menu_msg[BATMON_menu_counter]);
632                     BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
633                 }
634                 // Button B2 pressed, accept menu item
635                 else if(BATMON_control_reg & B2) {
636                     switch (BATMON_menu_counter) {
637                         case 0:
638                             BATMON_menu_position = BATMON_MENU_SHOW;
639                             break;                          // exit switch case
640                         case 1:
641                             BATMON_menu_position = BATMON_MENU_START;
642                             break;
643                         case 2:
644                             BATMON_menu_position = BATMON_MENU_STOP;
645                             break;
646                         case 3:
647                             BATMON_menu_position = BATMON_MENU_LOAD;
648                             break;
649                         case 4:
650                             BATMON_menu_position = BATMON_MENU_SAVE;
651                             break;
652                         case 5:
653                             BATMON_menu_position = BATMON_MENU_RTC;
654                             break;
655                         case 6:
656                             BATMON_menu_position = BATMON_MENU_BL;
657                             break;
658                         case 7:
659                             BATMON_menu_position = BATMON_MENU_CTAR;
660                             break;
661                         case 8:
662                             BATMON_menu_position = BATMON_MENU_CINV;
663                             break;
664                         case 9:
665                             BATMON_menu_position = BATMON_MENU_SCAN;
666                             break;
667                         case 10:
668                             BATMON_menu_position = BATMON_MENU_CALI;
669                             break;
670                         case 11:
671                             BATMON_menu_position = BATMON_MENU_ZS_WAKEUP_CW;
672                             break;
673                         case 12:
674                             BATMON_menu_position = BATMON_MENU_ZS_WAKEUP_PULSE;
675                             break;
676                         case 13:
677                             BATMON_menu_position = BATMON_MENU_ZS_SHUTDOWN_ALL;
678                             break;
679                         case 14:
680                             BATMON_menu_position = BATMON_MENU_ZS_SYNCH;
681                             break;
682                         case 15:
683                             BATMON_menu_position = BATMON_MENU_ZS_SET_IV_TIME;
684                             break;
685                         case 16:
686                             BATMON_menu_position = BATMON_MENU_ZS_SET_SAMPLE_TIME;
687                             break;
688                         case 17:
689                             BATMON_menu_position = BATMON_MENU_ZS_SET_TX_TIME;
690                             break;
691                         case 18:
692                             BATMON_menu_position = BATMON_MENU_ZS_SEND_SET1;
```

```
693                             break;
694                         case 19:
695                             BATMON_menu_position = BATMON_MENU_ZS_SEND_SET2;
696                             break;
697                         case 20:
698                             BATMON_menu_position = BATMON_MENU_ZS_BALANCE;
699                             break;
700                     }
701                     break;                                  // exit while loop
702                 }
703                 else if(BATMON_control_reg & B3) {  // Button B3 pressed, exit menu
704                     if (Enable_clear_lcd == ON) {
705                         LCD_send_cmd(LCD_LINE1);
706                         LCD_send_text("                ");
707                     }
708                     else if (Enable_clear_lcd == STATUS) {
709                         LCD_send_cmd(LCD_LINE1);
710                         LCD_send_text(" Radio disabled ");
711                     }
712                     BATMON_menu_position = BATMON_MENU_EXIT;
713                     break;                                  // exit while loop
714                 }
715             }
716             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
717             break;                                          // exit switch case

719         // switch Backlight ON/OFF ____________________________________________________
720         case BATMON_MENU_BL:
721             LCD_send_cmd(LCD_LINE1);
722             LCD_send_text(BATMON_menu_msg[6]);
723             LCD_send_cmd(LCD_LINE2);
724             LCD_send_text(BATMON_menu_msg_line2[2]);

726             while(1) {
727                 if(BATMON_control_reg & B1) {       // Button B1 pressed, Backlight OFF
728                     BL_OFF;
729                     UART0_send_text("Backlight OFF\n");
730                     break;                                  // exit while loop
731                 }
732                 else if(BATMON_control_reg & B2) {  // Button B2 pressed, Backlight ON
733                     BL_ON;
734                     UART0_send_text("Backlight ON\n");
735                     break;                                  // exit while loop
736                 }
737             }
738             BATMON_menu_position = BATMON_MENU_EXIT;
739             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
740             break;                                          // exit switch case

742         // Set actual current value at zero  Current TARA ____________________________
743         case BATMON_MENU_CTAR:
744             LCD_send_cmd(LCD_LINE1);
745             LCD_send_text(BATMON_menu_msg[7]);
746             LCD_send_cmd(LCD_LINE2);
747             LCD_send_text(BATMON_menu_msg_line2[1]);

749             while(1) {
750                 if(BATMON_control_reg & B1) {       // Button B1 pressed, set zero
751                     scan_mode = 4;
752                     ADC_start();
753                     break;                                  // exit while loop
754                 }
755                 else if(BATMON_control_reg & B2) {  // Button B2 pressed, exit menu
```

```
756                     break;                                  // exit while loop
757                 }
758             }
759             BATMON_menu_position = BATMON_MENU_EXIT;
760             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
761             break;                                          // exit switch case

763         // invert current values "sensor type specific" ____________________________________
764         case BATMON_MENU_CINV:
765             LCD_send_cmd(LCD_LINE1);
766             LCD_send_text(BATMON_menu_msg[8]);
767             LCD_send_cmd(LCD_LINE2);
768             LCD_send_text(BATMON_menu_msg_line2[1]);

770             while(1) {
771                 if(BATMON_control_reg & B1) { // Button B1 pressed, invert current values
772                     if (g_currmeas.current_invert == 0) {
773                         g_currmeas.current_invert = 1;
774                     }
775                     else {
776                         g_currmeas.current_invert = 0;
777                     }
778                     break;                                  // exit while loop
779                 }
780                 else if(BATMON_control_reg & B2) {  // Button B2 pressed, exit menu
781                     break;                                  // exit while loop
782                 }
783             }
784             BATMON_menu_position = BATMON_MENU_EXIT;
785             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
786             break;                                          // exit switch case
787         // scan for sensors ________________________________________________________________
788         case BATMON_MENU_SCAN:
789             LCD_send_cmd(LCD_LINE1);
790             LCD_send_text(BATMON_menu_msg[9]);
791             LCD_send_cmd(LCD_LINE2);
792             LCD_send_text(BATMON_menu_msg_line2[3]);

794             while(TRUE) {

796                 if(BATMON_control_reg & B1) {       // Button B1 pressed, scan 6 sensors
797                     BATMON_control_reg  &= ~(B1|B2|B3);         // clear button state
798                     SCAN(6);
799                     break;                                      // exit while loop
800                 }
801                 else if(BATMON_control_reg & B2) {  // Button B2 pressed, scan 12 sensors
802                     BATMON_control_reg  &= ~(B1|B2|B3);         // clear button state
803                     SCAN(12);
804                     break;                                      // exit while loop
805                 }
806                 else if(BATMON_control_reg & B3) {  // Button B3 pressed, more submenu
807                     LCD_send_cmd(LCD_LINE2);
808                     LCD_send_text(BATMON_menu_msg_line2[4]);
809                     BATMON_control_reg &= ~(B1|B2|B3);          // clear button state
810                     while(TRUE) {
811                                                         // Button B1 pressed, scan 24 sensors
812                         if(BATMON_control_reg & B1) {
813                             BATMON_control_reg  &= ~(B1|B2|B3); // clear button state
814                             SCAN(24);
815                             break;                              // exit inner while loop
816                         }
817                                                         // Button B2 pressed, scan 40 sensors
818                         else if(BATMON_control_reg & B2) {
```

```
819                             BATMON_control_reg  &= ~(B1|B2|B3); // clear button state
820                             SCAN(40);
821                             break;                              // exit inner while loop
822                         }
823                                                     // Button B3 pressed, exit menu
824                         else if(BATMON_control_reg & B3) {
825                             BATMON_control_reg  &= ~(B1|B2|B3); // clear button state
826                             break;                              // exit inner while loop
827                         }
828                     }
829                     break;                                      // exit while loop
830                 }
831             }
832             BATMON_menu_position = BATMON_MENU_EXIT;
833             break;                                              // exit switch case

835         // calibrate temperature sensors ____________________________________________________
836         case BATMON_MENU_CALI:
837             LCD_send_cmd(LCD_LINE1);
838             LCD_send_text(BATMON_menu_msg[10]);
839             LCD_send_cmd(LCD_LINE2);
840             LCD_send_text(BATMON_menu_msg_line2[1]);
841             unsigned long Temperature=200;
842             unsigned char tenner,ones,dezi;

844             while(1) {
845                 if(BATMON_control_reg & B1) {           // Button B1 pressed, submenu
846                     LCD_send_cmd(LCD_LINE1);
847                     LCD_send_text(" Temp:");
848                     tenner = (Temperature%1000)/100;
849                     ones = (Temperature%100)/10;
850                     dezi = Temperature %10;
851                     LCD_send_data(tenner+48);
852                     LCD_send_data(ones+48);
853                     LCD_send_text(".");
854                     LCD_send_data(dezi+48);
855                     LCD_send_data(223);
856                     LCD_send_text("C    ");
857                     LCD_send_cmd(LCD_LINE2);
858                     LCD_send_text(BATMON_menu_msg_line2[5]);
859                     BATMON_control_reg &= ~(B1|B2|B3);          // clear button state
860                     while(1) {
861                         // Button B1 pressed, increment start value
862                         if(BATMON_control_reg & B1) {
863                             Temperature+=5;
864                             if (Temperature > 500) {
865                                 Temperature = 100;
866                             }
867                             LCD_send_cmd(LCD_LINE1);
868                             LCD_send_text(" Temp:");
869                             tenner = (Temperature%1000)/100;
870                             ones = (Temperature%100)/10;
871                             dezi = Temperature %10;
872                             LCD_send_data(tenner+48);
873                             LCD_send_data(ones+48);
874                             LCD_send_text(".");
875                             LCD_send_data(dezi+48);
876                             LCD_send_data(223);
877                             LCD_send_text("C    ");
878                             BATMON_control_reg &= ~(B1|B2|B3);  // clear button state
879                         }
880                         // Button B2 pressed, calibrate
881                         else if(BATMON_control_reg & B2) {
```

```
882                             BATMON_menu_position = BATMON_MENU_EXIT;
883                             BATMON_control_reg  &= ~(B1|B2|B3); // clear button state
884                             CALIBRATE(Temperature);
885                             break;                              // exit while loop
886                         }
887                         // Button B3 pressed, exit
888                         else if(BATMON_control_reg & B3) {
889                             BATMON_menu_position = BATMON_MENU_EXIT;
890                             BATMON_control_reg  &= ~(B1|B2|B3); // clear button state
891                             break;                              // exit while loop
892                         }
893                     }
894                 }
895                 else if(BATMON_control_reg & B2) {      // Button B2 pressed, EXIT
896                     BATMON_menu_position = BATMON_MENU_EXIT;
897                     BATMON_control_reg &= ~(B1|B2|B3);  // clear button state
898                     break;                              // exit while loop
899                 }
900                 else if (BATMON_menu_position == BATMON_MENU_EXIT) {
901                     break;                              // exit while loop
902                 }
903             }
904         // ZS WAKEUP CW ____________________________________________________________
905         case BATMON_MENU_ZS_WAKEUP_CW:
906             LCD_send_cmd(LCD_LINE1);
907             LCD_send_text(BATMON_menu_msg[11]);
908             LCD_send_cmd(LCD_LINE2);
909             LCD_send_text(BATMON_menu_msg_line2[6]);

911             while(1) {
912                 if(BATMON_control_reg & B1) {       // Button B1 pressed --> OFF
913                     zs_wakeup_cw(FALSE);
914                     BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
915                 }
916                 else if(BATMON_control_reg & B2) {  // Button B2 pressed --> ON
917                     zs_wakeup_cw(TRUE);
918                     BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
919                 }
920                 else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
921                     break;                                  // exit while loop
922                 }
923             }
924             BATMON_menu_position = BATMON_MENU_EXIT;
925             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
926             break;                                          // exit switch case
927         // ZS WAKEUP PULSE _________________________________________________________
928         case BATMON_MENU_ZS_WAKEUP_PULSE:
929                 LCD_send_cmd(LCD_LINE1);
930                 LCD_send_text(BATMON_menu_msg[12]);
931                 LCD_send_cmd(LCD_LINE2);
932                 LCD_send_text(BATMON_menu_msg_line2[7]);

934                 while(1) {
935                     if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
936                         zs_wakeup_pulse();
937                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
938                     }
939                     else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
940                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
941                     }
942                     else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
943                         break;                                  // exit while loop
944                     }
```

```
945                 }
946                 BATMON_menu_position = BATMON_MENU_EXIT;
947                 BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
948                 break;                                          // exit switch case
949         // ZS SHUTDOWN ALL _____________________________________________________________
950         case BATMON_MENU_ZS_SHUTDOWN_ALL:
951                 LCD_send_cmd(LCD_LINE1);
952                 LCD_send_text(BATMON_menu_msg[13]);
953                 LCD_send_cmd(LCD_LINE2);
954                 LCD_send_text(BATMON_menu_msg_line2[7]);

956                 while(1) {
957                     if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
958                         zs_shutdown_all();
959                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
960                     }
961                     else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
962                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
963                     }
964                     else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
965                         break;                                  // exit while loop
966                     }
967                 }
968                 BATMON_menu_position = BATMON_MENU_EXIT;
969                 BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
970                 break;
971         // ZS SYNCH ____________________________________________________________________
972         case BATMON_MENU_ZS_SYNCH:
973                 LCD_send_cmd(LCD_LINE1);
974                 LCD_send_text(BATMON_menu_msg[14]);
975                 LCD_send_cmd(LCD_LINE2);
976                 LCD_send_text(BATMON_menu_msg_line2[7]);

978                 while(1) {
979                     if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
980                         zs_send_synch();
981                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
982                     }
983                     else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
984                         BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
985                     }
986                     else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
987                         break;                                  // exit while loop
988                     }
989                 }
990                 BATMON_menu_position = BATMON_MENU_EXIT;
991                 BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
992                 break;                                          // exit switch case
993         // ZS SET TX TIME ______________________________________________________________
994         case BATMON_MENU_ZS_SET_TX_TIME:
995                 LCD_send_cmd(LCD_LINE1);
996                 LCD_send_text(BATMON_menu_msg[17]);
997                 LCD_send_cmd(LCD_LINE2);
998                 LCD_send_text(BATMON_menu_msg_line2[7]);

1000                while(1) {
1001                    if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
1002                        zs_send_time_tx();
1003                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1004                    }
1005                    else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
1006                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1007                    }
```

```
1008                    else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1009                        break;                          // exit while loop
1010                    }
1011                }
1012                BATMON_menu_position = BATMON_MENU_EXIT;
1013                BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1014                break;                                          // exit switch case
1015        // ZS SET SAMPLE TIME ____________________________________________________________
1016        case BATMON_MENU_ZS_SET_SAMPLE_TIME:
1017                LCD_send_cmd(LCD_LINE1);
1018                LCD_send_text(BATMON_menu_msg[16]);
1019                LCD_send_cmd(LCD_LINE2);
1020                LCD_send_text(BATMON_menu_msg_line2[7]);

1022                while(1) {
1023                    if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
1024                        zs_send_time_sample();
1025                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1026                    }
1027                    else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
1028                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1029                    }
1030                    else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1031                        break;                          // exit while loop
1032                    }
1033                }
1034                BATMON_menu_position = BATMON_MENU_EXIT;
1035                BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1036                break;                                          // exit switch case
1037        // ZS SET INTERVAL TIME __________________________________________________________
1038        case BATMON_MENU_ZS_SET_IV_TIME:
1039                LCD_send_cmd(LCD_LINE1);
1040                LCD_send_text(BATMON_menu_msg[15]);
1041                LCD_send_cmd(LCD_LINE2);
1042                LCD_send_text(BATMON_menu_msg_line2[7]);

1044                while(1) {
1045                    if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
1046                        zs_send_time_interval();
1047                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1048                    }
1049                    else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
1050                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1051                    }
1052                    else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1053                        break;                          // exit while loop
1054                    }
1055                }
1056                BATMON_menu_position = BATMON_MENU_EXIT;
1057                BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1058                break;                                          // exit switch case
1059        // ZS SEND SET1 __________________________________________________________________
1060        case BATMON_MENU_ZS_SEND_SET1:
1061                LCD_send_cmd(LCD_LINE1);
1062                LCD_send_text(BATMON_menu_msg[18]);
1063                LCD_send_cmd(LCD_LINE2);
1064                LCD_send_text(BATMON_menu_msg_line2[7]);

1066                while(1) {
1067                    if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
1068                        zs_send_set1();
1069                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1070                    }
```

```
1071                    else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
1072                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1073                    }
1074                    else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1075                        break;                                  // exit while loop
1076                    }
1077                }
1078                BATMON_menu_position = BATMON_MENU_EXIT;
1079                BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1080                break;
1081        // ZS SEND SET2 ____________________________________________________________________
1082        case BATMON_MENU_ZS_SEND_SET2:
1083                LCD_send_cmd(LCD_LINE1);
1084                LCD_send_text(BATMON_menu_msg[19]);
1085                LCD_send_cmd(LCD_LINE2);
1086                LCD_send_text(BATMON_menu_msg_line2[7]);

1088                while(1) {
1089                    if(BATMON_control_reg & B1) {       // Button B1 pressed --> SEND
1090                        zs_send_set2();
1091                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1092                    }
1093                    else if(BATMON_control_reg & B2) {  // Button B2 pressed --> unused
1094                        BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1095                    }
1096                    else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1097                        break;                                  // exit while loop
1098                    }
1099                }
1100                BATMON_menu_position = BATMON_MENU_EXIT;
1101                BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1102                break;
1103        // ZS BALANCE ______________________________________________________________________
1104        case BATMON_MENU_ZS_BALANCE:
1105            LCD_send_cmd(LCD_LINE1);
1106            LCD_send_text(BATMON_menu_msg[20]);
1107            LCD_send_cmd(LCD_LINE2);
1108            LCD_send_text(BATMON_menu_msg_line2[6]);

1110            while(1) {
1111                if(BATMON_control_reg & B1) {       // Button B1 pressed --> OFF
1112                    zs_send_balance(FALSE);
1113                    BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1114                }
1115                else if(BATMON_control_reg & B2) {  // Button B2 pressed --> ON
1116                    zs_send_balance(TRUE);
1117                    BATMON_control_reg &= ~(B1|B2|B3);      // clear button state
1118                }
1119                else if(BATMON_control_reg & B3) {  // Button B3 pressed --> EXIT
1120                    break;                                  // exit while loop
1121                }
1122            }
1123            BATMON_menu_position = BATMON_MENU_EXIT;
1124            BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
1125            break;                                          // exit switch case
1126        // default _________________________________________________________________________
1127        default:
1128            BATMON_menu_position = BATMON_MENU_EXIT;
1129        //_________________________________________________________________________________
1130        }   // end switch(BATMON_menu_position)
1131    }   // end while(BATMON_menu_position != BATMON_MENU_EXIT)

1133    if (Enable_clear_lcd == STATUS) {                   // set text in first LCD line
```

```
1134            LCD_send_cmd(LCD_LINE1);
1135            LCD_send_text(" Radio disabled ");
1136        }
1137        else if (Enable_clear_lcd == ON) {
1138            LCD_send_cmd(LCD_LINE1);
1139            LCD_send_text("                ");
1140        }
1141        LCD_send_time();                                    // set time on second LCD line
1142    }
1143    /*-------------------------------------------------------------------------------------
1144        Calibration function
1145    -------------------------------------------------------------------------------------*/
1146    void CALIBRATE(unsigned long Temperature)
1147    /*-------------------------------------------------------------------------------------*/
1148    {
1149        recording_stop();
1150        temp_correct_value = Temperature*8/10;
1151        UART0_send_text("Calibrating...\n");
1152        LCD_send_cmd(LCD_LINE1);
1153        LCD_send_text(" Calibrating... ");
1154        LCD_send_cmd(LCD_LINE2);
1155        LCD_send_text("Stop            ");
1156        Enable_clear_lcd = OFF;
1157        scan_mode = 2;
1158        RADIO_enable();
1159        while (scan_mode > 1)
1160        {
1161            if(BATMON_control_reg & B1)
1162            {
1163                BATMON_control_reg &= ~(B1|B2|B3);
1164                // clear button state
1165                scan_mode = 0;
1166                RADIO_disable();
1167                Enable_clear_lcd = ON;
1168                LCD_send_cmd(LCD_LINE1);
1169                LCD_send_text(" Cali. halted   ");
1170                UART0_send_text("Calibration halted\n");
1171            }
1172        }
1173        LCD_send_time();
1174    }
1175    //-------------------------------------------------------------------------------------
```

Programmausdruck C.36: *Basisstation „msp_functions.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        MSP430 MSP_functions header
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               02/14/2008
8      Last update:        09/05/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     File:               msp_functions.h
14 -------------------------------------------------------------------------------------*/
15
16 #ifndef MSP_FUNCIONS_H_
17 #define MSP_FUNCIONS_H_
18
19 /*-------------------------------------------------------------------------------------
20     Includes
21 -------------------------------------------------------------------------------------*/
22 #include <headerfiles/flash.h>
23
24 /*-------------------------------------------------------------------------------------
25     Prototypes
26 -------------------------------------------------------------------------------------*/
27
28 unsigned char XT2_set(unsigned int);
29
30 void PORTS_init(void);
31 void USART0_init (unsigned char, unsigned char);
32
33 void UART0_send (unsigned char);
34 void UART0_send_text(char *);
35 unsigned char UART0_cmp_string(void);
36
37 void RADIO_init_frame(void);
38 void RADIO_init(void);
39 void RADIO_show_received_data(void);
40 void RADIO_store_frame(void);
41 void RADIO_enable(void);
42 void RADIO_disable(void);
43
44 void BATMON_menu_options(void);
45
46 void CALIBRATE(unsigned long Temperature);
47
48 unsigned char* DEZ2HEX(unsigned char);
49
50
51
52 #endif /*MSP_FUNCIONS_H_*/
53 //-------------------------------------------------------------------------------------
```

Programmausdruck C.37: *Basisstation „msp_functions.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for Reader TRF7960
4      Used components:    MSP430-169STK, BS Reader 13.56MHz
5
6      Author:             Niels Jegenhorst
7      Date:               20/04/2011
8      Last update:        07/07/2011
9
10     File:               reader_13p56mhz.c
11 -------------------------------------------------------------------------------------------
12
13 -------------------------------------------------------------------------------
14     Headerfiles
15 -------------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <msp430x16x.h>
18 #include "headerfiles/globals.h"
19 #include "headerfiles/msp_functions.h"
20 #include "headerfiles/reader_13p56mhz.h"
21
22 /*-------------------------------------------------------------------------------
23     void USART0_init_SPI(void)
24 -------------------------------------------------------------------------------*/
25 void USART0_init_SPI(void) {
26
27         // This function sets the USART0 for communication via SPI.
28         // The SPI-Interface is used for the reader communication.
29
30     g_usart_mode = MODE_SPI;                // Indicate USART in SPI-Mode
31     P3SEL &= ~(BIT4 | BIT5);
32     P3SEL  = (BIT1 | BIT2 | BIT3);
33     READER_STE_OFF;
34
35     // Control Reg.: SW Reset on
36     U0CTL  = SWRST;
37     // Control Reg.: SPI-Mode, 8-Bit, Master-Mode
38     U0CTL |= (CHAR | SYNC | MM);
39     // Transmit Control Reg.: CKPH = 1, BRCLK source = SMCLK, 3-Pin SPI-Mode
40     U0TCTL = (CKPH | SSEL1 | STC);
41     // Baud rate = UBRCLK / [U0BR1, U0BR0] = 100kBit/s
42     // UBRCLK = SMCLK = MCLK = 8MHz
43     //  --> U0BR = 80 = 0x50
44     U0BR0  = 0x50;                          // lower byte
45     U0BR1  = 0x00;                          // higher byte
46     // Modulation Control Reg.: = 0x00
47     U0MCTL = 0x00;
48     // Interrupt Enable Reg.: disable UTXIFG0 & URXIFG0 IRQ
49     IE1   &= ~(UTXIE0 | URXIE0);
50     // Modul Enable Reg.: enable USART0 SPI
51     ME1   &= ~(UTXE0 | URXE0);
52     ME1   |= USPIE0;
53     // Control Reg.: SW Reset off
54     U0CTL &= ~SWRST;
55 }
56 /*-------------------------------------------------------------------------------
57     void reader_set_reg(void)
58 -------------------------------------------------------------------------------*/
59 void reader_set_reg(void) {
60
61         // This function sets the reader "BS Reader 13.56MHz v0.2" in direct-mode
62         // via the SPI-Interface.
```

```
64      READER_STE_ON;                              // slave select for Reader

66      while((U0TCTL & TXEPT) != TXEPT);           // wait until TX shift register is empty
67      U0TXBUF = READER_REG_SYS_CLK;               // send data for register selection
68      while((IFG1 & UTXIFG0) != UTXIFG0);         // wait until U0TXBUF is empty
69      U0TXBUF = READER_REG_SYS_CLK_DATA;          // send data for register value

71      while((U0TCTL & TXEPT) != TXEPT);           // wait until TX shift register is empty
72      READER_STE_OFF;                             // slave unselect for Reader

74      READER_STE_ON;                              // slave select for Reader

76      while((U0TCTL & TXEPT) != TXEPT);           // wait until TX shift register is empty
77      U0TXBUF = READER_REG_CHIP_STATUS;           // send data for register selection
78      while((IFG1 & UTXIFG0) != UTXIFG0);         // wait until U0TXBUF is empty
79      U0TXBUF = READER_REG_CHIP_STATUS_DATA;      // send data for register value

81      while((U0TCTL & TXEPT) != TXEPT);           // wait until TX shift register is empty
82      READER_STE_OFF;                             // slave unselect for Reader
83  }
84  /*---------------------------------------------------------------------------
85      void tx13p56_send_wakeup(void)
86  ---------------------------------------------------------------------------*/
87  void tx13p56_send_wakeup(uint32_t address, uint8_t cmd, uint16_t data) {

89          // This function initiates a data transmission to the "ZS v0.1" on 13.56MHz.
90          // At first a long RUN IN will be transmitted, this is needed to
91          // give the Sensor time for wakeup (power on DC/DC and MCU).

93      g_reader_tx.address  = address;
94      g_reader_tx.cmd      = cmd;
95      g_reader_tx.dataload = data;
96      tx13p56_gen_frame();            // generate TX frame

98      g_reader_tx.state = WAKE_UP_END;// set state for TX

100     TBCTL &= ~(MC1 | MC0);          // set stop-mode
101     TBCTL |= TBCLR;                 // resets TBR, clockdiv, count dir

103     TBCCTL0 &= ~(COV | CCIFG);      // clear IRQ flags
104     TBCCR0   = TBCCR_WAKE_UP;       // compare value for WAKE-UP time
105     TBCCTL0 |= CCIE;                // compare-mode, IRQ enable

107     TBCTL &= ~TBIFG;                // clear IRQ flag
108     TBCTL |= (TBSSEL1 | TBIE);      // clock source = SMCLK, div = 1, IRQ enable

110     if(g_rx_en == TRUE) {           // Check if RX433 is enabled
111         g_rx_reen = TRUE;               // indicate RX433 should be reenabled
112         while(TRUE) {                   // wait until RX433 complete
113             if(g_rx_status != ACTIVE)
114                 break;
115         }
116         RADIO_disable();                // disable RX433
117     }
118     else {
119         g_rx_reen = FALSE;              // indicate RX should not be reenabled
120     }
121     TBCTL |= MC0;                   // set up-mode --> start RX process
122     READER_MOD_ON;                  // set 13.56MHz RF on
123     while(TRUE) {                   // wait until TX complete
124         if(g_reader_tx.state == FINISH)
125             break;
```

```
126     }
127                                     // enable RX433 is done by ISR

129 }
130 /*-------------------------------------------------------------------------------------
131     void tx13p56_send(void)
132 -------------------------------------------------------------------------------------*/
133 void tx13p56_send(uint32_t address, uint8_t cmd, uint16_t data) {

135         // This function initiates a data transmission to the "ZS v0.1" on 13.56MHz.

137     g_reader_tx.address  = address;
138     g_reader_tx.cmd      = cmd;
139     g_reader_tx.dataload = data;
140     tx13p56_gen_frame();            // generate TX frame

142     TBCTL &= ~(MC1 | MC0);          // set stop-mode
143     TBCTL |= TBCLR;                 // resets TBR, clockdiv, count dir

145     TBCCTL0 &= ~(COV | CCIFG);      // clear IRQ flags
146     if(g_reader_tx.cw == ON) {      // Check if CW-mode is on:
147         g_reader_tx.state = RUN_IN_START;   // set state for TX
148         TBCCR0 = TBCCR_BIT;                 // compare value for bit handling time
149     }
150     else {
151         g_reader_tx.state = RUN_IN_END;     // set state for TX
152         TBCCR0   = TBCCR_RUN_IN;            // compare value for RUN IN time
153     }
154     TBCCTL0 |= CCIE;                // compare-mode, IRQ enable

156     TBCTL &= ~TBIFG;                // clear IRQ flag
157     TBCTL |= (TBSSEL1 | TBIE);      // clock source = SMCLK, div = 1, IRQ enable

159     if(g_rx_en == TRUE) {           // Check if RX433 is enabled:
160         g_rx_reen = TRUE;               // indicate RX433 should be reenabled
161         while(TRUE) {                   // wait until RX433 complete
162             if(g_rx_status != ACTIVE)
163                 break;
164         }
165         RADIO_disable();                // disable RX433
166     }
167     else {
168         g_rx_reen = FALSE;              // indicate RX should not be reenabled
169     }
170     if(g_reader_tx.cw == ON) {      // Check if CW-mode is on:
171         READER_MOD_OFF;                 // set 13.56MHz RF off
172     }
173     else {
174         READER_MOD_ON;                  // set 13.56MHz RF on
175     }
176     TBCTL |= MC0;                   // set up-mode --> start RX process
177     while(TRUE) {                   // wait until TX complete
178         if(g_reader_tx.state == FINISH)
179             break;
180     }
181                                     // enable RX433 is done by ISR
182 }
183 /*-------------------------------------------------------------------------------------
184     void tx13p56_gen_frame(void)
185  * -------------------------------------------------------------------------------------*/
186 void tx13p56_gen_frame(void) {

188         // This functions generates the data frame with only the payload and the crc,
```

```
189         // for transmitting data on 13.56MHz to the "ZS v0.1".

191                                     // Address 20bit, MSB first:
192     g_reader_tx.txdata[0]  = (g_reader_tx.address >> 12);   // upper 8bit to byte
193     g_reader_tx.txdata[1]  = (g_reader_tx.address >> 4);    // middle 8bit to byte
194     g_reader_tx.txdata[2]  = (g_reader_tx.address << 4);    // lower 4bit to upper 4bit
195                                     // Command 8bit, MSB first:
196     g_reader_tx.txdata[2] |= (g_reader_tx.cmd >> 4);        // upper 4bit to lower 4bit
197     g_reader_tx.txdata[3]  = (g_reader_tx.cmd << 4);        // lower 4bit to upper 4bit
198                                     // Data 12bit, MSB first:
199     g_reader_tx.txdata[3] |= (g_reader_tx.dataload >> 8);   // upper 4bit to lower 4bit
200     g_reader_tx.txdata[4]  = (g_reader_tx.dataload & 0xFF); // lower 8bit to byte

202     tx13p56_calc_crc();                         // calculate CRC
203     g_reader_tx.txdata[5]  = g_reader_tx.crc;   // CRC 8bit, MSB first:

205 }
206 /*---------------------------------------------------------------------------
207     void tx13p56_calc_crc(void)
208  * ------------------------------------------------------------------------*/
209 void tx13p56_calc_crc(void) {

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

214     uint8_t i;

216     g_reader_tx.crc = 0x00;                         // Start with generator polynom = 0

218     for(i=0; i<TX13P56_FRAME_BYTES-1; i++) {        // XOR bytewise with txdata
219         g_reader_tx.crc ^= g_reader_tx.txdata[i];
220     }
221 }
222 //---------------------------------------------------------------------------------------
```

Programmausdruck C.38: *Basisstation „reader_13p56mhz.c“*

```
/*---------------------------------------------------------------------------------------
    Project:            BATSEN
    Discription:        Header for Reader TRF7960
    Used components:    MSP430-169STK, BS Reader 13.56MHz

    Author:             Niels Jegenhorst
    Date:               20/04/2011
    Last update:        20/06/2011

    File:               reader_13p56mhz.h
 ---------------------------------------------------------------------------------------
 ---------------------------------------------------------------------------------------
    Frame for TX on 13.56MHz:

    |       SOF        |  Address   |  Command  | Parameter |   CRC     |
    |------------------|------------|-----------|-----------|-----------|
    | RUN-IN    SYNCH  |            |           |           |           |
    |           8-Bit  | 20-Bit     | 8-Bit     | 12-Bit    | 8-Bit     |
    |------------------|------------|-----------|-----------|-----------|
    |                  | 0       19 | 20     27 | 28     39 | 40     47 | -> 48 Bits
    |                  | 0        2 | 2       3 | 3       4 | 5         | -> 6  Bytes
    |------------------|------------|-----------|-----------|-----------|    on 5.0kb/s
    | 800us     1.6ms  |    4ms     |   1.6ms   |   2.4ms   |   1.6ms   | -> 12.0ms

    Datarate = 5.0kb/s
    Transmissionrate = 10.0kb/s (Manchester)

    CRC-Checksum is generated over Address, Command and Data (bits 0 - 39)
---------------------------------------------------------------------------------------*/

#ifndef READER_13P56MHZ_H_
#define READER_13P56MHZ_H_

    //___ Port I/O: ____________________________________________
    #define READER_MOD_OFF  P1OUT |= BIT4
    #define READER_MOD_ON   P1OUT &= ~BIT4

    #define READER_ON       P1OUT |= BIT3
    #define READER_OFF      P1OUT &= ~BIT3

    #define READER_ASK      P2OUT &= ~BIT6;
    #define READER_OOK      P2OUT |= BIT6;

    #define READER_STE_OFF  P3OUT |= BIT0;
    #define READER_STE_ON   P3OUT &= ~BIT0;

    //___ TRF7960: Register Address Space _________________
    #define READER_REG_CHIP_STATUS      0x00
    #define READER_REG_ISO_CONTROL      0x01
    #define READER_REG_SYS_CLK          0x09

    //___ TRF7960: Register Settings ____________________
                                            // VDD_RF = 5V, RF output activ, Direct-mode
    #define READER_REG_CHIP_STATUS_DATA (BIT0 | BIT5 | BIT6)
    #define READER_REG_ISO_CONTROL_DATA 0x00   // Direct-mode: output is sub-carrier data
    #define READER_REG_SYS_CLK_DATA     (BIT0) // OOK-Modulation, no SYSCLK output

    //___ TX Data: ____________________________________________
    #define ZS_ADDRESS_GLOB     0x7FFFF

    enum rx_commands {
        ZS_CMD_STAY_ON          = 0x00,
```

```
63          ZS_CMD_TURN_OFF         = 0x01,
64          ZS_CMD_BAL_ON           = 0x02,
65          ZS_CMD_BAL_OFF          = 0x03,
66          ZS_CMD_SAMPLE_S1        = 0x04,
67          ZS_CMD_SAMPLE_S2        = 0x05,
68          ZS_CMD_SET_SAMPLE_TIME  = 0x06,
69          ZS_CMD_SET_TX_TIME      = 0x07,
70          ZS_CMD_SET_INTVAL_TIME  = 0x08,
71          ZS_CMD_SYNCH            = 0x09,
72          ZS_CMD_SCAN             = 0x0A,
73          ZS_CMD_REPLY            = 0x0B,
74          ZS_CMD_SAVE_SET         = 0x0C
75      };

77      #define TBCCR_WAKE_UP       40000-1     // Compare value for: 5.0ms (SMCLK = 8MHz)
78      #define TBCCR_RUN_IN        6400-1      // Compare value for: 800us (SMCLK = 8MHz)
79      #define TBCCR_PP_SHORT      400-1       // Compare value for: 50us  (SMCLK = 8MHz)
80      #define TBCCR_PP_LONG       800-1       // Compare value for: 100us (SMCLK = 8MHz)
81      #define TBCCR_BIT           800-1       // Compare value for: 100us (SMCLK = 8MHz)
82      #define TBCCR_2BIT          1600-1      // Compare value for: 200us (SMCLK = 8MHz)

84      #define TX13P56_FRAME_BYTES     6
85      #define TX13P56_SYNCH_LENGTH    8       // length of SYNCH in bit

87      /*-------------------------------------------------------------------------------
88          Prototypes
89      -------------------------------------------------------------------------------*/
90      void USART0_init_SPI(void);
91      void reader_set_reg(void);
92      void tx13p56_send_wakeup(uint32_t address, uint8_t cmd, uint16_t data);
93      void tx13p56_send(uint32_t address, uint8_t cmd, uint16_t data);
94      void tx13p56_gen_frame(void);
95      void tx13p56_calc_crc(void);

97  #endif /* READER_13P56MHZ_H_ */
98  //-------------------------------------------------------------------------------------
```

Programmausdruck C.39: *Basisstation „reader_13p56mhz.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        MSP430 RTC
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               02/08/2008
8      Last update:        09/05/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  08/07/2011
15
16     File:               rtc.h
17 -------------------------------------------------------------------------------------------
18
19 -----------------------------------------------------------------------------
20     Headerfiles
21 -----------------------------------------------------------------------------*/
22 #include <main.h>
23 #include <stdio.h>
24 #include <msp430x16x.h>
25 #include "headerfiles/rtc.h"
26 #include "headerfiles/lcd16x2.h"
27 #include "headerfiles/msp_functions.h"
28 #include "headerfiles/globals.h"
29
30 /*-----------------------------------------------------------------------------
31     RTC init (initialze watchdog timer)
32 -----------------------------------------------------------------------------*/
33 void RTC_init(void)
34 /*-----------------------------------------------------------------------------*/
35 {
36     unsigned char time_char;
37
38     RTC_menue_position = RTC_MENUE_DAY;
39     BATMON_control_reg &= ~(B1 | B2 | B3);
40
41     // initialise watchdog timer used for BLINKY, intervall mode ------------------------
42     WDTCTL   = WDTCTL_BLK_MODE;
43     BCSCTL1 &= ~(DIVA1 | DIVA0);                        // LFXT1 / 1 = ACLK = 32768Hz
44     WDT_function = BLINKY;
45     IE1 |= WDTIE;                                       // enable Watchdog interrupt
46
47     LCD_send_date();                                    // display real time and weekday
48     /*-------------------------------------------------------------------------------
49      Edit the date
50     -------------------------------------------------------------------------------*/
51     do
52     {
53         /* if B1 and B2 are pressed exit the RTC_EDIT_MODE  */
54         if ((BATMON_control_reg & B1) && (BATMON_control_reg & B2))
55         {
56             BATMON_control_reg &= ~(B1 | B2);
57             RTC_menue_position = RTC_MENUE_EXIT;
58         }
59         /* only B1 is pressed decrement the editable value(day,sec,min,hr)  */
60         else if(BATMON_control_reg & B1)
61         {
62                 BATMON_control_reg &= ~B1;      // clear button state variable
```

```
63                  LED2_TOGGLE;
64                  switch(RTC_menue_position)
65                  {
66                  case RTC_MENUE_DAY: if(RTC_values.day == 1) RTC_values.day = 31;
67                      else RTC_values.day--;
68                      break;
69                  case RTC_MENUE_MONTH: if(RTC_values.month == 0) RTC_values.month = 11;
70                      else RTC_values.month--;
71                      break;
72                  case RTC_MENUE_YEAR: if(RTC_values.year == 0) RTC_values.year = 4000;
73                      else RTC_values.year--;
74                      break;
75                  default: break;
76                  }
77          }
78          /* only B2 is pressed increment the editable value(day,sec,min,hr)  */
79          else if(BATMON_control_reg & B2)
80          {
81                  BATMON_control_reg &= ~B2;      // clear button state variable
82                  LED2_TOGGLE;
83                  switch(RTC_menue_position)
84                  {
85                  case RTC_MENUE_DAY: if(RTC_values.day == 31) RTC_values.day = 1;
86                      else RTC_values.day++;
87                      break;
88                  case RTC_MENUE_MONTH: if(RTC_values.month == 11) RTC_values.month = 0;
89                      else RTC_values.month++;
90                      break;
91                  case RTC_MENUE_YEAR: if(RTC_values.year == 4000) RTC_values.year = 0;
92                      else RTC_values.year++;
93                      break;
94                  default: break;
95                  }
96          }
97          /* only B3 is pressed change the editable value */
98          else if(BATMON_control_reg & B3)
99          {
100             BATMON_control_reg &= ~B3;  // clear button state variable
101             LED2_TOGGLE;
102             /* display previews editable value */
103             switch(RTC_menue_position)
104             {
105                 case RTC_MENUE_DAY: LCD_send_cmd(LCD_RTC_DAY);
106                     time_char = (RTC_values.day / 10) + 48;
107                     LCD_send_data(time_char);
108                     time_char = (RTC_values.day % 10) + 48;
109                     LCD_send_data(time_char);
110                     break;
111                 case RTC_MENUE_MONTH: LCD_send_cmd(LCD_RTC_MONTH);
112                     LCD_send_text(RTC_months[RTC_values.month]);
113                     break;
114                 case RTC_MENUE_YEAR: LCD_send_cmd(LCD_RTC_YEAR);
115                     time_char = RTC_values.year / 1000;
116                     LCD_send_data(time_char+48);
117                     time_char = (RTC_values.year / 100) - ((RTC_values.year/1000) * 10);
118                     LCD_send_data(time_char+48);
119                     time_char = (RTC_values.year / 10) - ((RTC_values.year/100) * 10);
120                     LCD_send_data(time_char+48);
121                     time_char = RTC_values.year % 10;
122                     LCD_send_data(time_char+48);
123                     break;
124                 default: break;
125             }
```

```
126             /* change editable value */
127             if(RTC_menue_position == RTC_MENUE_YEAR)
128                 RTC_menue_position = RTC_MENUE_DAY;
129             else
130                 RTC_menue_position++;
131         }
132     }
133     while(RTC_menue_position != RTC_MENUE_EXIT);

135 /*-------------------------------------------------------------------------------
136  Edit the weekday and time
137 -------------------------------------------------------------------------------*/
138     /* display real time and weekday */
139     IE1 &= ~WDTIE;                          // disable Watchdog interrupt
140     RTC_menue_position = RTC_MENUE_WDAY;
141     LCD_send_time();
142     IE1 |= WDTIE;           // enable Watchdog interrupt

144     do
145     {
146         /* if B1 and B2 are pressed exit the RTC_EDIT_MODE  */
147         if ((BATMON_control_reg & B1) && (BATMON_control_reg &B2))
148         {
149             BATMON_control_reg &= ~(B1 | B2);
150             RTC_menue_position = RTC_MENUE_EXIT;
151         }
152         /* only B1 is pressed decrement the editable value(day,sec,min,hr)  */
153         else if(BATMON_control_reg & B1)
154         {
155                 BATMON_control_reg &= ~B1;      // clear button state variable
156                 LED2_TOGGLE;
157                 switch(RTC_menue_position)
158                 {
159                 case RTC_MENUE_WDAY: if(RTC_values.wday == MON) RTC_values.wday = SUN;
160                     else RTC_values.wday--;
161                     break;
162                 case RTC_MENUE_HR: if(RTC_values.hr == 0) RTC_values.hr = 23;
163                     else RTC_values.hr--;
164                     break;
165                 case RTC_MENUE_MIN: if(RTC_values.min == 0) RTC_values.min = 59;
166                     else RTC_values.min--;
167                     break;
168                 case RTC_MENUE_SEC: if(RTC_values.sec == 0) RTC_values.sec = 59;
169                     else RTC_values.sec--;
170                     break;
171                 default: break;
172                 }
173         }
174         /* only B2 is pressed increment the editable value(day,sec,min,hr)  */
175         else if(BATMON_control_reg & B2)
176         {
177                 BATMON_control_reg &= ~B2;      // clear button state variable
178                 LED2_TOGGLE;
179                 switch(RTC_menue_position)
180                 {
181                 case RTC_MENUE_WDAY: if(RTC_values.wday == SUN) RTC_values.wday = MON;
182                     else RTC_values.wday++;
183                     break;
184                 case RTC_MENUE_HR: if(RTC_values.hr == 23) RTC_values.hr = 0;
185                     else RTC_values.hr++;
186                     break;
187                 case RTC_MENUE_MIN: if(RTC_values.min == 59) RTC_values.min = 0;
188                     else RTC_values.min++;
```

```
189                     break;
190                 case RTC_MENUE_SEC: if(RTC_values.sec == 59) RTC_values.sec = 0;
191                     else RTC_values.sec++;
192                     break;
193                 default: break;
194                 }
195         }
196         /* only B3 is pressed change the editable value */
197         else if(BATMON_control_reg & B3)
198         {
199             BATMON_control_reg &= ~B3;  // clear button state variable
200             LED2_TOGGLE;
201             /* display previews editable value */
202             switch(RTC_menue_position)
203             {
204                 case RTC_MENUE_WDAY: LCD_send_cmd(LCD_RTC_WDAY);
205                     LCD_send_text(RTC_days[RTC_values.wday]);
206                     break;
207                 case RTC_MENUE_HR: LCD_send_cmd(LCD_RTC_HR);
208                     time_char = (RTC_values.hr / 10) + 48;
209                     LCD_send_data(time_char);
210                     time_char = (RTC_values.hr % 10) + 48;
211                     LCD_send_data(time_char);
212                     break;
213                 case RTC_MENUE_MIN: LCD_send_cmd(LCD_RTC_MIN);
214                     time_char = (RTC_values.min / 10) + 48;
215                     LCD_send_data(time_char);
216                     time_char = (RTC_values.min % 10) + 48;
217                     LCD_send_data(time_char);
218                     break;
219                 case RTC_MENUE_SEC: LCD_send_cmd(LCD_RTC_SEC);
220                     time_char = (RTC_values.sec / 10) + 48;
221                     LCD_send_data(time_char);
222                     time_char = (RTC_values.sec % 10) + 48;
223                     LCD_send_data(time_char);
224                     break;
225                 default: break;
226             }
227             /* change editable value */
228             if(RTC_menue_position == RTC_MENUE_SEC)
229                 RTC_menue_position = RTC_MENUE_WDAY;
230             else
231                 RTC_menue_position++;
232         }
233     }
234     while(RTC_menue_position != RTC_MENUE_EXIT);

236     // initialise watchdog timer used for RTC, intervall mode, --------------------------
237     WDTCTL   = WDTCTL_RTC_MODE;
238     BCSCTL1 |= ACLK_DIVIDER;                            // LFXT1 / DIV = ACLK
239     WDT_function = RTC;
240     IE1 |= WDTIE;                                       // enable Watchdog interrupt

242     LED1_OFF;                                           // switch LEDS off
243     LED2_OFF;

245     LCD_send_time();                                    // display real time and day

247     if (Enable_clear_lcd == STATUS) {
248         LCD_send_cmd(LCD_LINE1);
249         LCD_send_text(" Radio disabled ");
250     }
251     else if(Enable_clear_lcd == ON) {
```

```
252             LCD_send_cmd(LCD_LINE1);
253             LCD_send_text("                ");
254         }
255     }

257     /*-------------------------------------------------------------------------------
258         RTC compare tmp and real time
259     -------------------------------------------------------------------------------*/
260     void RTC_compare(RTC_MSP430 *RTC_tmp)
261     /*-------------------------------------------------------------------------------*/
262     {
263         unsigned char time_char;

265         /* compare old RTC values with real values */
266         if(RTC_values.wday != RTC_tmp->wday)
267         {
268             RTC_tmp->wday = RTC_values.wday;
269             LCD_send_cmd(LCD_RTC_WDAY);
270             LCD_send_text(RTC_days[RTC_values.wday]);
271         }
272         if(RTC_values.hr != RTC_tmp->hr)
273         {
274             RTC_tmp->hr = RTC_values.hr;
275             LCD_send_cmd(LCD_RTC_HR);
276             time_char = (RTC_values.hr / 10) + 48;
277             LCD_send_data(time_char);
278             time_char = (RTC_values.hr % 10) + 48;
279             LCD_send_data(time_char);
280         }
281         if(RTC_values.min != RTC_tmp->min)
282         {
283             RTC_tmp->min = RTC_values.min;
284             LCD_send_cmd(LCD_RTC_MIN);
285             time_char = (RTC_values.min / 10) + 48;
286             LCD_send_data(time_char);
287             time_char = (RTC_values.min % 10) + 48;
288             LCD_send_data(time_char);
289         }
290         if(RTC_values.sec != RTC_tmp->sec)
291         {
292             RTC_tmp->sec = RTC_values.sec;
293             LCD_send_cmd(LCD_RTC_SEC);
294             time_char = (RTC_values.sec / 10) + 48;
295             LCD_send_data(time_char);
296             time_char = (RTC_values.sec % 10) + 48;
297             LCD_send_data(time_char);
298         }
299     }

301     /*-------------------------------------------------------------------------------
302         RTC startup, set time/date or init timer only
303     -------------------------------------------------------------------------------*/
304     void RTC_startup(void)
305     //-------------------------------------------------------------------------------
306     {
307         LCD_send_cmd(LCD_LINE1);
308         LCD_send_text(" Set date/time  ");
309         LCD_send_cmd(LCD_LINE2);
310         LCD_send_text("YES    NO       ");

312         while(1) {
313             if(BATMON_control_reg & B1) {                   // Button 1 = YES,
314                 RTC_init();                                     // Function for adjust RTC
```

```
315             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
316             break;                                          // exit while loop
317         }
318         if(BATMON_control_reg & B2) {                    // Button 2 = NO,
319             // initialise watchdog timer used for RTC, intervall mode --------------------
320             WDTCTL   = WDTCTL_RTC_MODE;
321             BCSCTL1 |= ACLK_DIVIDER;                        // LFXT1 / DIV = ACLK
322             WDT_function = RTC;
323             IE1 |= WDTIE;                                   // enable Watchdog interrupt
324             LCD_send_time();                                // display real time and day
325             BATMON_control_reg &= ~(B1|B2|B3);              // clear button state
326             break;                                          // exit while loop
327         }
328     }
329 }
330 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.40: *Basisstation „rtc.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        RTC Header-File
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               02/08/2008
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  08/07/2011
15
16     File:               rtc.h
17 -------------------------------------------------------------------------------------*/
18
19 #ifndef RTC_H_
20 #define RTC_H_
21
22 #include "headerfiles/typedefs.h"
23
24 /*-------------------------------------------------------------------------------------
25     Defines
26 -------------------------------------------------------------------------------------*/
27                                     // for RTC-mode:
28                                     // 1 sec intervall mode, div = 32768, div ACLK = 1
29 //#define WDTCTL_RTC_MODE (WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL)
30                                     // 1/4 sec intervall mode, div = 8192, div ACLK = 1
31 //#define WDTCTL_RTC_MODE (WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS_1)
32                                     // 1/8 sec intervall mode, div = 512, div ACLK = 8
33 //#define WDTCTL_RTC_MODE   (WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS1)
34                                     // 1/16 sec intervall mode, div = 512, div ACLK = 4
35 #define WDTCTL_RTC_MODE (WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS1)
36                                     // for BLINKY-mode:
37                                     // 1/4 sec intervall mode, div = 8192, div ACLK = 1
38 #define WDTCTL_BLK_MODE (WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS_1)
39
40 /*  clock position on LCD   */
41 #define LCD_RTC_SEC     (LCD_LINE2+10)
42 #define LCD_RTC_MIN     (LCD_LINE2+7)
43 #define LCD_RTC_HR      (LCD_LINE2+4)
44 #define LCD_RTC_WDAY     LCD_LINE2
45 #define LCD_RTC_YEAR    (LCD_LINE2+7)
46 #define LCD_RTC_MONTH   (LCD_LINE2+3)
47 #define LCD_RTC_DAY     LCD_LINE2
48
49 /*-------------------------------------------------------------------------------------
50     Prototypes
51 -------------------------------------------------------------------------------------*/
52 void RTC_init(void);
53 void RTC_compare(RTC_MSP430 *);
54 void RTC_startup(void);
55
56 #endif /*RTC_H_*/
57 //-------------------------------------------------------------------------------------
```

Programmausdruck C.41: *Basisstation „rtc.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for sensor data processing
4      Used components:    MSP430-169STK
5
6      Author:             Niels Jegenhorst
7      Date:               20/04/2011
8      Last update:        26/09/2011
9
10     File:               sensor_data_proc.c
11 -------------------------------------------------------------------------------------------
12
13 ---------------------------------------------------------------------------
14     Headerfiles
15 ---------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <msp430x16x.h>
18 #include "headerfiles/globals.h"
19 #include "reader_13p56mhz.h"
20
21 /*---------------------------------------------------------------------------
22     void check_depth_discharge(uint8_t index)
23 ---------------------------------------------------------------------------*/
24 void check_depth_discharge(uint8_t index) {
25
26         // Author:              Alexander Hoops
27         // Last modification:   12/02/2010
28
29     unsigned long check_voltage;
30
31     check_voltage = (SENSOR_last_data[3][index]<<8);
32     check_voltage |= SENSOR_last_data[4][index];
33     if ((check_voltage/4.096) <= ((CELL_VOLT_MIN*10)+100)) {
34         if ((check_voltage/4.096) <= (CELL_VOLT_MIN*10)) {
35             SENSOR_low++;
36         }
37     }
38 }
39 /*---------------------------------------------------------------------------
40     void convert_sensor_data(void)
41 ---------------------------------------------------------------------------*/
42 void convert_sensor_data(void) {
43
44     g_sensor_data.vid  = RADIO_frame.frame_byte[0];
45     g_sensor_data.id   = RADIO_frame.frame_byte[3];
46     g_sensor_data.id  |= (uint32_t) RADIO_frame.frame_byte[2] << 8;
47     g_sensor_data.id  |= ((uint32_t) RADIO_frame.frame_byte[1] << 16) & 0xF0;
48     g_sensor_data.dataset = 0x00;
49                                     // Select the correct conversion on the
50     switch (g_sensor_data.vid) {    //  basis of the sensor VID:
51         case 0x00:
52             break;
53
54         case 0x01:              // VID of BS0406 ____________________________________
55                                     // Conversion of temperature value:
56             g_sensor_data.temperature  = (uint16_t) RADIO_frame.frame_byte[4] << 8;
57             g_sensor_data.temperature |= RADIO_frame.frame_byte[5];
58                                     // no Info?!
59             g_sensor_data.temperature = g_sensor_data.temperature;
60             RADIO_frame.frame_byte[4] = g_sensor_data.temperature >> 8;
61             RADIO_frame.frame_byte[5] = g_sensor_data.temperature & 0xFF;
```

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

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

80          case 0x03:              // VID of ZS v0.1/0.2 ____________________________________

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

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

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

109                     g_sensor_data.v_actin = 0;
110                     g_sensor_data.v_lp    = 0;
111                     break;
112                 case SET_2:             // When alternative sensor data received: -------
113                                         // Conversion of lowpass-voltage:
114                     g_sensor_data.v_lp  = (uint16_t) RADIO_frame.frame_byte[4] << 8;
115                     g_sensor_data.v_lp |= RADIO_frame.frame_byte[5];
116                                         // data / 2^12 * 2500 --> [mV]
117                     g_sensor_data.v_lp  = (g_sensor_data.v_lp * 2500) >> 12;
118                     RADIO_frame.frame_byte[4] = g_sensor_data.v_lp >> 8;
119                     RADIO_frame.frame_byte[5] = g_sensor_data.v_lp & 0xFF;

121                                         // Conversion of Activator-voltage:
122                     g_sensor_data.v_actin  = (uint16_t) RADIO_frame.frame_byte[8] << 8;
123                     g_sensor_data.v_actin |= RADIO_frame.frame_byte[9];
124                                         // data / 2^12 * 2500 --> [mV]
125                     g_sensor_data.v_actin  = (g_sensor_data.v_actin * 2500) >> 12;
```

```
126                     RADIO_frame.frame_byte[8] = g_sensor_data.v_actin >> 8;
127                     RADIO_frame.frame_byte[9] = g_sensor_data.v_actin & 0xFF;

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

137                     g_sensor_data.v_supply    = 0;
138                     g_sensor_data.temperature = 0;
139                     break;
140                 case REPLY_CMD:
141                     g_sensor_data.rx_last_cmd[0] = RADIO_frame.frame_byte[4];
142                     g_sensor_data.rx_last_cmd[1] = RADIO_frame.frame_byte[5];
143                     g_sensor_data.rx_last_parameter[0]  =
144                                             (uint16_t) RADIO_frame.frame_byte[6] << 8;
145                     g_sensor_data.rx_last_parameter[0] |= RADIO_frame.frame_byte[7];
146                     g_sensor_data.rx_last_parameter[1]  =
147                                             (uint16_t) RADIO_frame.frame_byte[8] << 8;
148                     g_sensor_data.rx_last_parameter[1] |= RADIO_frame.frame_byte[9];
149                     break;
150                 default:
151                     break;
152             }
153             break;

155         default:                // ____________________________________________________
156             g_sensor_data.v_cell      = 0;
157             g_sensor_data.v_supply    = 0;
158             g_sensor_data.v_actin     = 0;
159             g_sensor_data.v_lp        = 0;
160             g_sensor_data.temperature = 0;
161             break;
162     }                           // end of switch case
163                                             // store converted cell voltage value:
164     SENSOR_last_data[3][RADIO_frame.sensor_index] = g_sensor_data.v_cell >> 8;
165     SENSOR_last_data[4][RADIO_frame.sensor_index] = g_sensor_data.v_cell & 0xFF;

167     // --- Generate Timestamp: ------------------------------------------------------
168         // RTC_t_sample = RTC_t_sof - t_offset
169         //     t_offset = (t_tx - t_sample)/10 * scale / 10 in ms
170     g_rx_timing.t_offset   = (g_zs_timing.t_tx[RADIO_frame.sensor_index] -
171                               g_zs_timing.t_sample) / 10;
172     g_rx_timing.t_offset <<= ZS_TIME_SCALE;
173     g_rx_timing.t_offset  /= 100;
174     g_rx_timing.rtc_sof.msec -= g_rx_timing.t_offset;
175         // check if milliseconds are negative --> then fix other time units:
176     if(g_rx_timing.rtc_sof.msec < 0) {
177         g_rx_timing.rtc_sof.sec --;
178         g_rx_timing.rtc_sof.msec += 1000;
179         if(g_rx_timing.rtc_sof.sec < 0) {
180             g_rx_timing.rtc_sof.min --;
181             g_rx_timing.rtc_sof.sec += 60;
182             if(g_rx_timing.rtc_sof.min < 0) {
183                 g_rx_timing.rtc_sof.hr --;
184                 g_rx_timing.rtc_sof.min += 60;
185                 if(g_rx_timing.rtc_sof.hr < 0) {
186                     g_rx_timing.rtc_sof.day --;
187                     g_rx_timing.rtc_sof.hr += 24;
188                     if(g_rx_timing.rtc_sof.day < 1) {
```

```
189                         g_rx_timing.rtc_sof.day = RTC_values.lastday;
190                     }
191                 }
192             }
193         }
194     }
195         // Timestamp = [DAY HR MIN SEC MSEC] (32-bit):
196     g_sensor_data.timestamp   = g_rx_timing.rtc_sof.day;    // days:    5-bit used
197     g_sensor_data.timestamp <<= 5;                          // shift 5-bit for hours
198     g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.hr;     // hours:   5-bit used
199     g_sensor_data.timestamp <<= 6;                          // shift 6-bit for minutes
200     g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.min;    // minutes: 6-bit used
201     g_sensor_data.timestamp <<= 6;                          // shift 6-bit for seconds
202     g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.sec;    // seconds: 6-bit used
203     g_sensor_data.timestamp <<= 10;                         // shift 10-bit for msec
204     g_sensor_data.timestamp  |= g_rx_timing.rtc_sof.msec;   // milliseconds: 10-bit used

206 }
207 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.42: *Basisstation „sensor_data_proc.c“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Header-File for sensor data processing
4      Used components:    MSP430-169STK

6      Author:             Niels Jegenhorst
7      Date:               20/04/2011
8      Last update:        20/04/2011

10     File:               sensor_sig_proc.h
11 -------------------------------------------------------------------------------------------*/

13 #ifndef SENSOR_DATA_PROC_H_
14 #define SENSOR_DATA_PROC_H_

16     /*---------------------------------------------------------------------------------------
17         Prototypes
18     ---------------------------------------------------------------------------------------*/
19     void check_depth_discharge(uint8_t);
20     void convert_sensor_data(void);

22 #endif /* SENSOR_DATA_PROC_H_ */
23 //-------------------------------------------------------------------------------------------
```

Programmausdruck C.43: *Basisstation „sensor_data_proc.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for handling of the sensor system
4      Used components:    MSP430-169STK, BS Reader 13.56MHz
5
6      Author:             Niels Jegenhorst
7      Date:               20/05/2011
8      Last update:        13/07/2011
9
10     File:               system_handling.c
11 -------------------------------------------------------------------------------------------
12
13 ---------------------------------------------------------------------------
14     Headerfiles
15 ---------------------------------------------------------------------------*/
16 #include <main.h>
17 #include <stdio.h>
18 #include <msp430x16x.h>
19 #include "headerfiles/adc.h"
20 #include "headerfiles/lcd16x2.h"
21 #include "headerfiles/msp_functions.h"
22 #include "headerfiles/reader_13p56mhz.h"
23 #include "headerfiles/system_handling.h"
24 #include "headerfiles/globals.h"
25
26 /*---------------------------------------------------------------------------
27     Function to start the measurement (Author Stephan Plaschke, Mod by Niels Jegenhorst)
28 ---------------------------------------------------------------------------*/
29 void recording_start(void)
30 /*---------------------------------------------------------------------------*/
31 {
32     uint8_t i;
33     LCD_send_cmd(LCD_LINE1);
34     LCD_send_text(" Radio enabled  ");
35     UART0_send_text("Radio enabled\n");
36     LED2_ON;
37     Enable_clear_lcd = ON;
38     measuring = 1;
39     #ifdef PIN_DEBUG
40         TP_INR_OFF;
41         TP_ADC1_OFF;
42         TP_ADC2_OFF;
43     #endif
44     #ifdef ENABLE_RX_ERROR_LOG
45         g_rx_error.bit_error   = 0x0000;
46         g_rx_error.wrong_RunIn = 0x0000;
47         g_rx_error.timeout     = 0x0000;
48         for(i=0; i<6; i++) {
49             g_rx_error.crc_error[i] = 0x0000;
50         }
51     #endif
52                                                 // wait until RTC change befor send
53     g_zs_timing.send_synch_msec = RTC_values.msec;  //  synch CMD, for better repeats
54     while(RTC_values.msec == g_zs_timing.send_synch_msec);
55     g_currtiming.trigger_msec  = RTC_values.msec;
56     g_currtiming.trigger_sec   = RTC_values.sec;
57     g_zs_timing.send_synch_msec = g_currtiming.trigger_msec;
58     g_zs_timing.send_synch_sec  = g_currtiming.trigger_sec;
59
60     g_zs_timing.send_synch_en = TRUE;           // enable sending CMD Synch every x sec
61     #ifdef ENABLE_CURRENT_MEASURE
62         g_currtiming.trigger_en = TRUE;         // enable triggering ADC every x sec
```

```
63      #endif
64                                              // trigger SW IRQ for sending SYNCH CMD
65      P2OUT |= SW_IRQ_ZS_SYNCH;               // and ADC current measuring when needed
66                                              // wait until CMD Synch to ZS send
67      while((P2OUT & SW_IRQ_ZS_SYNCH) == SW_IRQ_ZS_SYNCH);
68      RADIO_enable();                         // enable RX
69  }
70  /*-------------------------------------------------------------------------------
71      Function to stop the measurement (Author Stephan Plaschke, Mod by Niels Jegenhorst)
72  -------------------------------------------------------------------------------*/
73  void recording_stop(void)
74  /*-------------------------------------------------------------------------------*/
75  {
76      LCD_send_cmd(LCD_LINE1);
77      LCD_send_text(" Radio disabled ");
78      UART0_send_text("Radio disabled\n");
79      LED2_OFF;
80      Enable_clear_lcd = STATUS;
81      measuring = 0;
82      g_zs_timing.send_synch_en = FALSE;      // disable sending CMD Synch every x sec
83      g_currtiming.trigger_en = FALSE;        // disable triggering ADC every x sec
84      RADIO_disable();                        // disable RX
85      #ifdef ENABLE_CURRENT_MEASURE
86          ADC_stop();
87      #endif
88      LED1_OFF;
89  }
90  /*-------------------------------------------------------------------------------
91      Function to wake up the ZS with cw
92  -------------------------------------------------------------------------------*/
93  void zs_wakeup_cw(uint8_t mod)
94  /*-------------------------------------------------------------------------------*/
95  {
96      if(mod == TRUE) {
97          READER_MOD_ON;
98          g_reader_tx.cw = ON;
99          UART0_send_text("ZS Wakeup cw ON\n");
100     }
101     else {
102         READER_MOD_OFF;
103         g_reader_tx.cw = OFF;
104         UART0_send_text("ZS Wakeup cw OFF\n");
105     }
106 }
107 /*-------------------------------------------------------------------------------
108     Function to wake up the ZS with data transmisson
109 -------------------------------------------------------------------------------*/
110 void zs_wakeup_pulse(void)
111 /*-------------------------------------------------------------------------------*/
112 {
113     tx13p56_send_wakeup(ZS_ADDRESS_GLOB, ZS_CMD_STAY_ON, 0x0000);
114     UART0_send_text("ZS Wakeup pulse sent\n");
115 }
116 /*-------------------------------------------------------------------------------
117     Function to shut down all ZS
118 -------------------------------------------------------------------------------*/
119 void zs_shutdown_all(void)
120 /*-------------------------------------------------------------------------------*/
121 {
122     tx13p56_send_wakeup(ZS_ADDRESS_GLOB, ZS_CMD_TURN_OFF, 0x0000);
123     UART0_send_text("ZS Shutdown all sent\n");
124 }
125 /*-------------------------------------------------------------------------------
```

```
126     Function to send an synch command to all ZS
127 ------------------------------------------------------------------------------*/
128 void zs_send_synch(void)
129 /*------------------------------------------------------------------------------*/
130 {
131     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SYNCH, 0x0000);
132     UART0_send_text("ZS Synch sent\n");
133 }
134 /*------------------------------------------------------------------------------
135     Function to send an command to any ZS separatly, to set time for TX
136 ------------------------------------------------------------------------------*/
137 void zs_send_time_tx(void)
138 /*------------------------------------------------------------------------------*/
139 {
140     uint8_t i, delay;
141     if(zs_gen_time_tx() == FALSE) {         // generate time to TX for each found sensor
142         UART0_send_text("ZS Set TX time spacing failure\n");
143     }
144     else {                                  // Send CMD "Set TX Time" to each sensor:
145         for(i=0; i < g_sensor_data.num_sensors_detected; i++) {
146             tx13p56_send(SENSOR_address[i], ZS_CMD_SET_TX_TIME, g_zs_timing.t_tx[i]);
147             for(delay=0; delay<0xFF; delay++) nop();
148         }
149         UART0_send_text("ZS Set TX time sent\n");
150     }
151 }
152 /*------------------------------------------------------------------------------
153     Function to send an command to all ZS which sets time for Sampling
154 ------------------------------------------------------------------------------*/
155 void zs_send_time_sample(void)
156 /*------------------------------------------------------------------------------*/
157 {
158     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SET_SAMPLE_TIME, ZS_TIME_SAMPLE);
159     UART0_send_text("ZS Set Sample time sent\n");
160 }
161 /*------------------------------------------------------------------------------
162     Function to send an command to all ZS which sets time for Interval
163 ------------------------------------------------------------------------------*/
164 void zs_send_time_interval(void)
165 /*------------------------------------------------------------------------------*/
166 {
167     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SET_INTVAL_TIME, ZS_TIME_INTERVAL);
168     UART0_send_text("ZS Set Interval time sent\n");
169 }
170 /*------------------------------------------------------------------------------
171     Function to send an command to all ZS, for set sample set 1
172 ------------------------------------------------------------------------------*/
173 void zs_send_set1(void)
174 /*------------------------------------------------------------------------------*/
175 {
176     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SAMPLE_S1, 0x0000);
177     UART0_send_text("ZS Sample Set 1 sent\n");
178 }
179 /*------------------------------------------------------------------------------
180     Function to send an command to all ZS, for set sample set 2
181 ------------------------------------------------------------------------------*/
182 void zs_send_set2(void)
183 /*------------------------------------------------------------------------------*/
184 {
185     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SAMPLE_S2, 0x0000);
186     UART0_send_text("ZS Sample Set 2 sent\n");
187 }
188 /*------------------------------------------------------------------------------
```

```
189     Function to send an command to all ZS, for set balance or unset balance
190 ------------------------------------------------------------------------------*/
191 void zs_send_balance(uint8_t balance)
192 /*------------------------------------------------------------------------------*/
193 {
194     if(balance == TRUE) {
195         tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_BAL_ON, 0x0000);
196         g_zs_balance = ON;
197         UART0_send_text("ZS Balance ON sent\n");
198     }
199     else {
200         tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_BAL_OFF, 0x0000);
201         g_zs_balance = OFF;
202         UART0_send_text("ZS Balance OFF sent\n");
203     }
204 }
205 /*------------------------------------------------------------------------------
206     Function to generate separete values for time to tx for each found sensor
207 ------------------------------------------------------------------------------*/
208 uint8_t zs_gen_time_tx(void)
209 /*------------------------------------------------------------------------------*/
210 {
211     uint8_t i;
212                                             // calc possible time for transmissions:
213     g_zs_timing.t_tx_possible_range = (g_zs_timing.t_interval - ZS_TIME_TX_TO_END) -
214                                       (g_zs_timing.t_sample + ZS_TIME_SAMPLE_TO_TX);
215                                             // calc spacing between transmissons:
216     if(g_sensor_data.num_sensors_detected > 1) {
217         g_zs_timing.t_tx_spacing = g_zs_timing.t_tx_possible_range /
218                                    (g_sensor_data.num_sensors_detected - 1);
219     }
220     else {                                  // if only one sensor detected
221         g_zs_timing.t_tx_spacing = 0;
222     }                                       // check spacing between transmissions:
223     if( (g_zs_timing.t_tx_spacing < ZS_TIME_TX_SPACE) &&
224         (g_sensor_data.num_sensors_detected > 1) ) {
225         return FALSE;
226     }                                       // calc time to tx for each found sensor:
227     g_zs_timing.t_tx[0] = g_zs_timing.t_sample + ZS_TIME_SAMPLE_TO_TX;
228     for(i=1; i < g_sensor_data.num_sensors_detected; i++) {
229         g_zs_timing.t_tx[i] = g_zs_timing.t_tx[i-1] + g_zs_timing.t_tx_spacing;
230     }
231     return TRUE;
232 }
233 /*------------------------------------------------------------------------------
234     Function to send an command to one an separatly, to get the last two
235     received commands and parameters
236 ------------------------------------------------------------------------------*/
237 void zs_send_reply(void)
238 /*------------------------------------------------------------------------------*/
239 {
240     uint8_t i, delay;
241     if(zs_gen_time_tx() == FALSE) {         // generate time to TX for each found sensor
242         UART0_send_text("ZS Reply TX time spacing failure\n");
243     }
244     else {
245         for(i=0; i < g_sensor_data.num_sensors_detected; i++) {
246             tx13p56_send(SENSOR_address[i], ZS_CMD_REPLY, g_zs_timing.t_tx[i]);
247             for(delay=0; delay<0xFF; delay++) nop();
248         }
249         UART0_send_text("ZS Reply sent\n");
250     }
251 }
```

```
252 /*-------------------------------------------------------------------------------------
253     Function to send an command to all ZS, to save the settings into flash
254 -------------------------------------------------------------------------------------*/
255 void zs_send_save_set(void)
256 /*-------------------------------------------------------------------------------------*/
257 {
258     tx13p56_send(ZS_ADDRESS_GLOB, ZS_CMD_SAVE_SET, 0x0000);
259     UART0_send_text("ZS Save Settings sent\n");
260 }
261 /*-------------------------------------------------------------------------------------
262     Scan function - scans for sensors  (Author Stephan Plaschke, Mod by Niels Jegenhorst)
263 -------------------------------------------------------------------------------------*/
264 void SCAN(unsigned char sensors)
265 /*-------------------------------------------------------------------------------------*/
266 {
267     unsigned char i;
268     if ((sensors == 6) || (sensors == 12) || (sensors == 24) || (sensors == 40)) {
269         for (i=0; i < NUM_SENSORS_MAX; i++) {   // first clear all addresses
270             SENSOR_address[i] = 0;
271         }
272         SENSOR_active = sensors;            // change numer of active sensors
273         recording_stop();                   // stop so scanning cannot be interrupted
274         LCD_send_cmd(LCD_LINE1);            // set cursor to first line of display
275         LCD_send_text("  Scanning...   ");  // write text to display
276         UART0_send_text("Scan enabled\n");  // write to uart
277         LCD_send_cmd(LCD_LINE2);            // set cursor to second line of display
278         LCD_send_text("Stop            ");  // write to display
279         Enable_clear_lcd = OFF;             // disable clear lcd flag
280         scan_mode = 1;                      // enable scanmode in interrupt
281                                             // TX CMD SCAN to sensors, 16 repeats:
282         tx13p56_send_wakeup(ZS_ADDRESS_GLOB, ZS_CMD_SCAN, 0x000F);
283         RADIO_enable();                     // enable receiver
284         while (scan_mode == 1) {            // wait until numbers of sensors found
285             if(BATMON_control_reg & B1) {   // or Button 1 stopps scan
286                 BATMON_control_reg &= ~(B1|B2|B3);  // clear button state
287                 scan_mode = 0;
288                 RADIO_disable();
289                 Enable_clear_lcd = ON;
290                 LCD_send_cmd(LCD_LINE1);
291                 LCD_send_text(" Scan halted    ");
292                 UART0_send_text("Scan halted\n");
293             }
294         }
295         for (i=0; i < NUM_SENSORS_MAX; i++) {   // check how many sensors were found:
296             if(SENSOR_address[i] == 0) {
297                 g_sensor_data.num_sensors_detected = i;
298                 printf("Sensors detected = %d\n", g_sensor_data.num_sensors_detected);
299                 break;
300             }
301         }
302         LCD_send_time();
303     }
304     else {                                  //wrong number of sensors --> error try again
305         LCD_send_cmd(LCD_LINE1);
306         LCD_send_text(" Scan impossible");
307         UART0_send_text("Scan impossible\n");
308         Enable_clear_lcd = ON;
309     }
310 }
311 //-------------------------------------------------------------------------------------
```

Programmausdruck C.44: *Basisstation „system_handling.c“*

```
1  /*-------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Header-File for handling of the sensor system
4      Used components:    MSP430-169STK, BS Reader 13.56MHz
5
6      Author:             Niels Jegenhorst
7      Date:               20/05/2011
8      Last update:        13/07/2011
9
10     File:               system_handling.c
11 -------------------------------------------------------------------------------------*/
12
13 #ifndef SYSTEM_HANDLING_H_
14 #define SYSTEM_HANDLING_H_
15
16     /*-------------------------------------------------------------------------------
17         Definition
18     -------------------------------------------------------------------------------*/
19     #define BS_MSEC_SCALE           10.0        // scaler for milliseconds on BS
20     #define BS_MSEC                 ((uint16_t) (1000 * BS_MSEC_SCALE))
21
22     //___ Timing for Sensors ___________________________________
23         // - T_INC of Timer on Sensor = 8us
24         // - values for timing must be scale down by 2^4, because there are only 12-bit
25         //   provided in the RX protocol for dataload.
26         // - maximum value for timing is 2^12
27     #define SMCLK_ZS                500.0       // SubMainCLK in kHz
28     #define TIMER_DIV_ZS            8           // Timer clock divider on ZS
29     #define T_STEP_ZS               (TIMER_DIV_ZS * 1000.0 / SMCLK_ZS)  // Step ZS [us]
30     #define T_TX_SCALE_DOWN_ZS      16          // Scale down for tx to ZS by 2^4 = 16
31     #define ZS_TIME_SCALE           7           // Total scale down on ZS = log2(8*16)
32
33     #define TEST_VALUES_1                       // Choose some test values for timing!
34     // --- Test values 1 for 3 sensors ---------------------------------------------
35     #ifdef TEST_VALUES_1
36                                                 // Time for intervalt = 200ms
37     #define ZS_TIME_INTERVAL      ((uint16_t)((200000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
38                                                 // Time to Sample = 10ms
39     #define ZS_TIME_SAMPLE        ((uint16_t)(( 10000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
40                                                 // Time between Sample & TX = 25ms
41     #define ZS_TIME_SAMPLE_TO_TX  ((uint16_t)(( 25000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
42                                                 // Time between TX and End = 40ms
43     #define ZS_TIME_TX_TO_END     ((uint16_t)(( 40000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
44                                                 // Time between TX and TX = 35ms
45     #define ZS_TIME_TX_SPACE      ((uint16_t)(( 35000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
46
47                                                 // Time between synch-commands in sec
48     #define SYNCH_INTERVAL_SEC      ((uint16_t)(2))
49                                                 // Time between synch-commands in milli
50     #define SYNCH_INTERVAL_MSEC     ((uint16_t)(0 * BS_MSEC_SCALE))
51                                                 // Timing for Current Measuring in sec
52     #define CURR_MEAS_INTERVAL_SEC  ((uint16_t)(1))
53                                                 // Timing for Current Measuring in milli
54     #define CURR_MEAS_INTERVAL_MSEC ((uint16_t)(0 * BS_MSEC_SCALE))
55     #endif
56     // --- Test values 2 for 6 sensors ---------------------------------------------
57     #ifdef TEST_VALUES_2
58                                                 // Time for intervalt = 300ms
59     #define ZS_TIME_INTERVAL      ((uint16_t)((300000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
60                                                 // Time to Sample = 10ms
61     #define ZS_TIME_SAMPLE        ((uint16_t)(( 10000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
62                                                 // Time between Sample & TX = 25ms
```

```
63      #define ZS_TIME_SAMPLE_TO_TX  ((uint16_t)(( 25000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
64                                                  // Time between TX and End = 40ms
65      #define ZS_TIME_TX_TO_END     ((uint16_t)(( 40000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
66                                                  // Time between TX and TX = 35ms
67      #define ZS_TIME_TX_SPACE      ((uint16_t)(( 35000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))

69                                                  // Time between synch-commands in sec
70      #define SYNCH_INTERVAL_SEC      ((uint16_t)(1))
71                                                  // Time between synch-commands in milli
72      #define SYNCH_INTERVAL_MSEC     ((uint16_t)(500 * BS_MSEC_SCALE))
73                                                  // Timing for Current Measuring in sec
74      #define CURR_MEAS_INTERVAL_SEC  ((uint16_t)(1))
75                                                  // Timing for Current Measuring in milli
76      #define CURR_MEAS_INTERVAL_MSEC ((uint16_t)(500 * BS_MSEC_SCALE))
77      #endif
78      // --- Test values 3 for 12 sensors -------------------------------------------------
79      #ifdef TEST_VALUES_3
80                                                  // Time for intervalt = 500ms
81      #define ZS_TIME_INTERVAL      ((uint16_t)((500000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
82                                                  // Time to Sample = 10ms
83      #define ZS_TIME_SAMPLE        ((uint16_t)(( 10000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
84                                                  // Time between Sample & TX = 25ms
85      #define ZS_TIME_SAMPLE_TO_TX  ((uint16_t)(( 25000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
86                                                  // Time between TX and End = 40ms
87      #define ZS_TIME_TX_TO_END     ((uint16_t)(( 40000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
88                                                  // Time between TX and TX = 35ms
89      #define ZS_TIME_TX_SPACE      ((uint16_t)(( 35000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))

91                                                  // Time between synch-commands in sec
92      #define SYNCH_INTERVAL_SEC      ((uint16_t)(2))
93                                                  // Time between synch-commands in milli
94      #define SYNCH_INTERVAL_MSEC     ((uint16_t)(0 * BS_MSEC_SCALE))
95                                                  // Timing for Current Measuring in sec
96      #define CURR_MEAS_INTERVAL_SEC  ((uint16_t)(0))
97                                                  // Timing for Current Measuring in milli
98      #define CURR_MEAS_INTERVAL_MSEC ((uint16_t)(500 * BS_MSEC_SCALE))
99      #endif
100     // --- Test values 4 for 12 sensors -------------------------------------------------
101     #ifdef TEST_VALUES_4
102                                                 // Time for intervalt = 1000ms
103     #define ZS_TIME_INTERVAL      ((uint16_t)((1000000/ T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
104                                                 // Time to Sample = 10ms
105     #define ZS_TIME_SAMPLE        ((uint16_t)(( 10000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
106                                                 // Time between Sample & TX = 25ms
107     #define ZS_TIME_SAMPLE_TO_TX  ((uint16_t)(( 25000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
108                                                 // Time between TX and End = 40ms
109     #define ZS_TIME_TX_TO_END     ((uint16_t)(( 40000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))
110                                                 // Time between TX and TX = 35ms
111     #define ZS_TIME_TX_SPACE      ((uint16_t)(( 35000 / T_STEP_ZS) / T_TX_SCALE_DOWN_ZS))

113                                                 // Time between synch-commands in sec
114     #define SYNCH_INTERVAL_SEC      ((uint16_t)(2))
115                                                 // Time between synch-commands in milli
116     #define SYNCH_INTERVAL_MSEC     ((uint16_t)(0 * BS_MSEC_SCALE))
117                                                 // Timing for Current Measuring in sec
118     #define CURR_MEAS_INTERVAL_SEC  ((uint16_t)(1))
119                                                 // Timing for Current Measuring in milli
120     #define CURR_MEAS_INTERVAL_MSEC ((uint16_t)(0 * BS_MSEC_SCALE))
121     #endif
122     // ------------------------------------------------------------------------------------

124     #define NUM_SENSORS_MAX         40          // Maximum of sensors for BS
```

```
126     /*---------------------------------------------------------------------------
127         Prototypes
128     ---------------------------------------------------------------------------*/
129     void recording_start(void);
130     void recording_stop(void);
131     void zs_wakeup_cw(uint8_t);
132     void zs_wakeup_pulse(void);
133     void zs_shutdown_all(void);
134     void zs_send_synch(void);
135     void zs_send_time_tx(void);
136     void zs_send_time_sample(void);
137     void zs_send_time_interval(void);
138     void zs_send_set1(void);
139     void zs_send_set2(void);
140     void zs_send_balance(uint8_t);
141     uint8_t zs_gen_time_tx(void);
142     void zs_send_reply(void);
143     void zs_send_save_set(void);
144     void SCAN(unsigned char);


147 #endif /* SYSTEM_HANDLING_H_ */
148 //---------------------------------------------------------------------------
```

Programmausdruck C.45: *Basisstation „system_handling.h“*

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

    Author:             Niels Jegenhorst
    Date:               20/04/2011
    Last update:        08/07/2011

    File:               typedefs.h
-------------------------------------------------------------------------------------*/

#ifndef TYPEDEFS_H_
#define TYPEDEFS_H_

    #include "headerfiles/reader_13p56mhz.h"
    #include "headerfiles/system_handling.h"

    /*-------------------------------------------------------------------------------------
    ||  Enums
    -------------------------------------------------------------------------------------*/
    enum week {MON, TUE, WED, THU, FRY, SAT, SUN};
    enum wdt_state {RTC, BLINKY};
    enum rtc_edit_menue {RTC_MENUE_WDAY, RTC_MENUE_HR, RTC_MENUE_MIN, RTC_MENUE_SEC,
                         RTC_MENUE_DAY, RTC_MENUE_MONTH, RTC_MENUE_YEAR, RTC_MENUE_EXIT};

    enum uart_menu_positions {  UART_MENU_FIRSTCHAR, UART_MENU_CLOCK,
                                UART_MENU_DATE, UART_MENU_SENSORADDR,
                                UART_MENU_SCAN, UART_MENU_CAL};

    typedef enum {                  //  Receive sequences in interrupt
        SE_SEQ_PREPARE,                 // 0 -> nothing to do, first interrupt
        SE_SEQ_START,                   // 1 -> first measurement completed,
                                        //  start sequence detected?
        SE_SEQ_PRE_WAIT,                // 2 -> complete start sequence detected
        SE_SEQ_DATA_START,              // 3 -> check for first 1
        SE_SEQ_RXDATA                   // 4 -> first zero detected, receive data
    } SE_SEQ_POS;
    typedef enum {
        ACTIVE,
        INACTIVE,
        SUCCESS,
        CRC_FAILURE,
        TIMEOUT,
        RUN_IN_ERROR,
        SYNCH_ERROR,
        BIT_ERROR,
    } RX_STATUS;
    typedef enum {
        BATMON_MENU_START,
        BATMON_MENU_STOP,
        BATMON_MENU_LOAD,
        BATMON_MENU_SAVE,
        BATMON_MENU_RTC,
        BATMON_MENU_EXIT,
        BATMON_MENU_SHOW,
        BATMON_MENU_MAIN,
        BATMON_MENU_BL,
        BATMON_MENU_SCAN,
        BATMON_MENU_CTAR,
        BATMON_MENU_CINV,
        BATMON_MENU_CALI,
```

```
63          BATMON_MENU_ZS_WAKEUP_CW,
64          BATMON_MENU_ZS_WAKEUP_PULSE,
65          BATMON_MENU_ZS_SHUTDOWN_ALL,
66          BATMON_MENU_ZS_SYNCH,
67          BATMON_MENU_ZS_SET_IV_TIME,
68          BATMON_MENU_ZS_SET_SAMPLE_TIME,
69          BATMON_MENU_ZS_SET_TX_TIME,
70          BATMON_MENU_ZS_SEND_SET1,
71          BATMON_MENU_ZS_SEND_SET2,
72          BATMON_MENU_ZS_BALANCE,
73      } BMON_MENUE;
74      typedef enum {
75          WAKE_UP_END,
76          RUN_IN_START,
77          RUN_IN_END,
78          SYNCH_0,
79          SYNCH_1,
80          DATALOAD,
81          END_PULS_1,
82          END_PULS_0,
83          COMPLETE,
84          FINISH
85      } TX_SEQ_POS;
86      typedef enum {
87          MODE_UART,
88          MODE_SPI
89      } USART_MODE;
90      typedef enum {
91          NONE,
92          DEC,
93          HEX
94      } SHOW_DATA_UART;
95      typedef enum {
96          SET_1       = 0x0,
97          SET_2       = 0x8,
98          REPLY_CMD   = 0xC
99      } RXDATA_SET;

101     /*-------------------------------------------------------------------------------
102     ||  Structures
103     -------------------------------------------------------------------------------*/
104     typedef struct {
105             uint8_t     hr;
106             uint8_t     min;
107             uint8_t     sec;
108             uint16_t    msec;
109             uint8_t     wday;
110             uint8_t     day;
111             uint8_t     lastday;
112             uint8_t     month;
113             uint16_t    year;
114     } RTC_MSP430;
115     typedef struct {
116             int8_t      day;
117             int8_t      hr;
118             int8_t      min;
119             int8_t      sec;
120             int16_t     msec;
121     } RTC_SHORT;
122     typedef struct {
123             uint8_t frame_byte[13];
124             uint8_t sensor_index;
125     } RADIO_FRAME_STRUCT;
```

```
126     typedef struct {
127             uint16_t    capture;
128             uint16_t    halfbit_tavg;
129             uint16_t    halfbit_tmin;
130             uint16_t    halfbit_tmax;
131             uint16_t    bit_tmin;
132             uint16_t    bit_tmax;
133             uint8_t     rx_in;
134             uint16_t    rx_seq;
135             uint16_t    shift_in;
136             uint8_t     byte_ctr;
137             RTC_SHORT   rtc_sof;
138             uint32_t    t_offset;
139     } RX_TIMING;
140     typedef struct {
141             uint16_t bit_error;
142             uint16_t wrong_RunIn;
143             uint16_t timeout;
144             uint16_t crc_error[6];  //crc_error count for each sensor address
145     } RX_ERROR;
146     typedef struct {
147             unsigned char C0;
148             unsigned char C1;
149             unsigned char C2;
150             unsigned char C3;
151             unsigned char C4;
152             unsigned char C5;
153             unsigned char C6;
154             unsigned char C7;
155             unsigned char C8;
156             unsigned char C9;
157             unsigned char C10;
158             unsigned char C11;
159             unsigned char C12;
160             unsigned char C13;
161             unsigned char C14;
162             unsigned char C15;
163             unsigned char C16;
164             unsigned char C17;
165     } DATA_STRUCT;
166     typedef struct {                    // structure for UART menu
167             char *cmd;
168             void (*fct)(void);              // pointer on function with no return- and
169                                             // no parameter value
170     } UART_MEN;
171     typedef struct {
172             uint32_t    address;            // only 20 bits used
173             uint8_t     cmd;
174             uint8_t     crc;
175             uint16_t    dataload;           // only 12 bits used
176             uint8_t     txdata[TX13P56_FRAME_BYTES];
177             uint8_t     bitpos;
178             uint8_t     bytectr;
179             uint8_t     manchesterpos;
180             TX_SEQ_POS  state;
181             uint8_t     cw;
182     } READER_TX;
183     typedef struct {
184             uint8_t     send_synch_en;
185             uint16_t    send_synch_msec;
186             uint8_t     send_synch_sec;
187             uint16_t    t_interval;
188             uint16_t    t_tx_possible_range;
```

```
189             uint16_t    t_tx_spacing;
190             uint16_t    t_tx[NUM_SENSORS_MAX];
191             uint16_t    t_sample;
192     } ZS_TIMING;
193     typedef struct {
194             uint32_t    v_cell;
195             uint32_t    v_supply;
196             uint32_t    v_lp;
197             uint32_t    v_actin;
198             int32_t     temperature;
199             uint8_t     vid;                    // Version ID,  8-bit
200             uint32_t    id;                     // Sensor ID,   28-bit
201             RXDATA_SET  dataset;
202             uint32_t    timestamp;              // Timestamp = [DAY HR MIN SEC], 32-bit
203             uint8_t     rx_last_cmd[2];
204             uint16_t    rx_last_parameter[2];
205             uint8_t     num_sensors_detected;
206     } SENSOR_DATA;
207     typedef struct {
208             uint16_t    channel1, channel2;
209             uint32_t    value;
210             uint16_t    value_signed;
211             uint8_t     channel;                // used sensor channel (1: ch1(low),
212                                                 //                      2: ch2(high))
213             uint32_t    sum;                    // current sum for tara
214             int8_t      current_sum_counter;    // counter for tara
215             uint32_t    tara;                   // tara for current value reset = 0
216             uint32_t    last_data;              // stored current value
217             uint32_t    work;                   // actual tared current value
218             uint8_t     current_in_out;         // 1 = charge 0 = discharge
219             uint8_t     current_invert;         // variable to setup sensor
220     } CURR_MEAS;
221     typedef struct {
222             uint32_t    AH_last_minute;         // AH last minute
223             uint32_t    AH_last_sec;            // AH last second
224             uint8_t     AH_last_minute_IN_OUT;  // 1 = charge 0 = discharge
225             uint32_t    AH_MASTER;              // AH picker *0.44 = AH
226             uint8_t     U_MASTER_CAPACITY;      // Capacity created by voltage
227                                                 // values in percent DOC
228     } CURR_AH;
229     typedef struct {
230             uint8_t     trigger_en;
231             uint16_t    trigger_msec;
232             uint8_t     trigger_sec;
233             uint32_t    timestamp;              // Timestamp = [DAY HR MIN SEC], 32-bit
234     } CURR_TIMING;

236 #endif /* TYPEDEFS_H_ */
237 //-----------------------------------------------------------------------------------------
```

Programmausdruck C.46: *Basisstation „typedefs.h“*

```
1  /*-------------------------------------------------------------------------------------------
2      Project:            BATSEN
3      Discription:        Sourcecode for handling of RS-232-Interface
4      Used components:    MSP430-169STK
5
6      Author:             Stephan Plaschke
7      Date:               02/14/2008
8      Last update:        04/04/2008
9
10     Modified by:        Alexander Hoops
11     Last modification:  12/02/2010
12
13     Modified by:        Niels Jegenhorst
14     Last modification:  08/07/2011
15
16     File:               uart_menue.c
17 -------------------------------------------------------------------------------------------*/
18
19 /*-------------------------------------------------------------------------------------------
20     Headerfiles
21 -------------------------------------------------------------------------------------------*/
22 #include <main.h>
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <string.h>
26 #include <msp430x16x.h>
27 #include "headerfiles/adc.h"
28 #include "headerfiles/lcd16x2.h"
29 #include "headerfiles/uart_menu.h"
30 #include "headerfiles/msp_functions.h"
31 #include "headerfiles/system_handling.h"
32 #include "headerfiles/globals.h"
33
34 /*-------------------------------------------------------------------------------------------
35     Static
36 -------------------------------------------------------------------------------------------*/
37 //  helpscreen and UART menu information
38 static char *UART_menu_cmd[10]={
39     "\n--- Battery monitor help screen --------------\n"
40     " | Enter commands via the Hyperterminal\n"
41     " | Shortcuts are displayed in brackets\n"
42     " | Capital and small letters are allowed\n"
43     " | Jump back from each submenu via EXIT (ex)\n"
44     "-----------------------------------------\n"
45     "Help     (he) : Display this screen\n"
46     "Realtime (rtc): Clock time adjustment via UART\n"
47     "Sensors  (se) : Set the address of each used sensor\n"
48     "Cal      (cal): Calibrate temperature from Sensors\n"
49     "Memory   (mem): Read the complete flash-memory of the receiver\n"
50     "Clear    (clr): Clear the flash-memory\n"
51     "Read     (rd) : Read controll data from flash\n"
52     "Write    (wr) : Write controll data to flash\n"
53     "Blon     (bn) : Switch backlight on\n"
54     "Bloff    (bf) : Switch backlight off\n"
55     "-----------------------------------------\n"
56     "Scan     (sc) : Scan for Sensors\n"
57     "Start    (st) : Start measurement\n"
58     "Stop     (stp): Stop measuremnt\n"
59     "-----------------------------------------\n"
60     "RXon          : Radio enable\n"
61     "RXoff         : Radio disable\n"
62     "-----------------------------------------\n"
```

```
63      "D_lcd_sd       : Show RX Sensor Data on LCD  (ON/OFF)\n"
64      "D_ua_dh        : Show RX Sensor Data on UART (HEX/OFF/DEC)\n"
65      "-------------------------------------------\n"
66      "Ctara          : Current Tara\n"
67      "Curinv         : Invert Current\n"
68      "-------------------------------------------\n"
69      "ZS_wk_a        : ZS Wakeup all Sensors\n"
70      "ZS_wk_cw       : ZS Wakeup CW (off/on)\n"
71      "ZS_sht_a       : ZS Shutdown all Sensors\n"
72      "ZS_syn         : ZS Send Synch\n"
73      "ZS_ds_1        : ZS Send Dataset 1\n"
74      "ZS_ds_2        : ZS Send Dataset 2\n"
75      "ZS_st_iv       : ZS Set Interval time\n"
76      "ZS_st_sa       : ZS Set Sample time\n"
77      "ZS_st_tx       : ZS Set TX time\n"
78      "ZS_bal         : ZS Balance (off/on)\n"
79      "ZS_reply       : ZS Send Reply\n"
80      "ZS_save        : ZS Save Settings\n"
81      "-------------------------------------------\n\n",

83      "Enter time in following format:\n"
84      "  WDAY,HR,MIN,SEC (d,hh,mm,ss: 0->Monday, 6->Sunday)\n"
85      "  Jump back via >EXIT< or jump further with >DATE<\n",

87      "Enter date in following format:\n"
88      "  DAY,MONTH,YEAR (dd,mm,yyyy: 01->January, 12->December)\n",

90      "Enter number of Sensors (6,12,24,40) and confirm with >ENTER<\n",

92      "Enter address of one sensor and confirm with >ENTER<\n",

94      "Read memory\n",

96      "Clear memory\n",

98      "Illegal command, type >help< for more information\n",

100     "Jump back to main menu\n",

102     "Enter real temperature value (example 20,5°C = 205) and confirm with enter\n"

104 };

106 //  first char in an UART command
107 static char *UART_menu_cmp_1st_char = "bcfhrsmwzd" ;

109 //  structure arrays for each first char, 1st element compare string,
110 //  2nd element pointer to function
111 //  if 1st char found compare string with received string,
112 //  if this matches execute function where pointer points at
113 //  each string can contain capital and small letters
114 static UART_MEN UART_menu_B[]={     // B...: ------------------------
115         {"lon", UART_menu_bl_on},       // Backlight on
116         {"n", UART_menu_bl_on},         // Backlight on
117         {"loff", UART_menu_bl_off},     // backlight off
118         {"f", UART_menu_bl_off},        // backlight off
119         {"\0", 0}                       // dummy to see the end of
120                                         // array of structs
121 };

123 static UART_MEN UART_menu_C[]={     // C...: ------------------------
124         {"al", UART_menu_cal},          // calibrate sensor temp
125         {"lr", UART_menu_flash_clr},    // flash clear command
```

```
126         {"lear", UART_menu_flash_clr},  // flash clear command
127         {"urinv", UART_menu_invert_cur},// invert current
128         {"tara", UART_menu_tara_cur},   // get current tara
129         {"\0", 0}                       // dummy to see the end of
130                                         // array of structs
131 };

133 static UART_MEN UART_menu_F[]={     // F...: ------------------------
134         {"er", UART_menu_FER_calc},     // start FER calc.
135         {"\0", 0}                       // dummy to see the end of
136                                         // array of structs
137 };

139 static UART_MEN UART_menu_H[]={     // H...: ------------------------
140         {"e", UART_menu_show_help},     // show help screen
141         {"elp", UART_menu_show_help},   // show help screen
142         {"\0", 0}                       // dummy to see the end of
143                                         // array of structs
144 };

146 static UART_MEN UART_menu_R[]={     // R...: -------------------------
147         {"tc", UART_menu_setrtc},               // set rtc, jump in submenu
148         {"ealtime", UART_menu_setrtc},          // set rtc, jump in submenu
149         {"d",       UART_menu_read_controll},   // read controll data from flash
150         {"ead",     UART_menu_read_controll},   // read controll data from flash
151         {"xon",     UART_menu_rx_on},           // enable Radio 433MHz
152         {"xoff",    UART_menu_rx_off},          // disbale Radio 433MHz
153         {"\0", 0}                               // dummy to see the end of
154                                                 // array of structs
155 };

157 static UART_MEN UART_menu_S[]={     // S...: ------------------------
158         {"c", UART_menu_scan},          // scan for sensors, jump in submenu
159         {"can", UART_menu_scan},        // scan for sensors, jump in submenu
160         {"e", UART_menu_setsensor},     // set sensor address, jump in submenu
161         {"ensor", UART_menu_setsensor}, // set sensor address, jump in submenu
162         {"t", recording_start},         // start measurement
163         {"tart", recording_start},      // start measurement
164         {"tp", recording_stop},         // stop measurement
165         {"top", recording_stop},        // stop measurement
166         {"\0", 0}                       // dummy to see the end of
167                                         // array of structs
168 };

170 static UART_MEN UART_menu_M[]={     // M...: ------------------------
171         {"em", UART_menu_read_data},    // set sensor address, jump in
172                                         // submenu
173         {"emory", UART_menu_read_data}, // set sensor address, jump in
174                                         // submenu
175         {"\0", 0}                       // dummy to see the end of
176                                         // array of structs
177 };

179 static UART_MEN UART_menu_W[]={     // W...: ------------------------
180         {"r", UART_menu_write_controll},    // set sensor address, jump in
181                                             // submenu
182         {"rite", UART_menu_write_controll}, // set sensor address, jump in
183                                             // submenu
184         {"\0", 0}                           // dummy to see the end of
185                                             // array of structs
186 };

188 static UART_MEN UART_menu_Z[]={     // ZS Commands: -----------------
```

```
189         {"s_wk_a",      UART_menu_zs_wakeup_pulse},     // wakeup all sensors with cmd
190         {"s_wk_cw",     UART_menu_zs_wakeup_cw},        // wakeup sensors with cw
191         {"s_sht_a",     UART_menu_zs_shutdown_all},     // shutdown all sensors
192         {"s_syn",       UART_menu_zs_synch},            // send synch cmd
193         {"s_ds_1",      UART_menu_zs_ds_1},             // send cmd for dataset 1
194         {"s_ds_2",      UART_menu_zs_ds_2},             // send cmd for dataset 2
195         {"s_st_tx",     UART_menu_zs_set_time_tx},      // send cmd set tx time
196         {"s_st_sa",     UART_menu_zs_set_time_sample},  // send cmd set sample time
197         {"s_st_iv",     UART_menu_zs_set_time_interval},// send cmd set interval time
198         {"s_bal",       UART_menu_zs_balance},          // send cmd balance
199         {"s_reply",     UART_menu_zs_reply},            // send cmd reply
200         {"s_save",      UART_menu_zs_save_set},         // send cmd save settings
201         {"\0", 0}                                       // dummy to see the end of
202                                                         // array of structs
203 };

205 static UART_MEN UART_menu_D[]={     // D...: ------------------------
206         {"_lcd_sd",     UART_menu_lcd_sensor_data},     // choice of show data on lcd
207         {"_ua_dh",      UART_menue_uart_show_dechex},   // choice of show on uart
208         {"\0", 0}                                       // dummy to see the end of
209                                                         // array of structs
210 };

212 static UART_MEN UART_menu_err[]={   // E...: ------------------------
213         {"\0", UART_menu_error},        // display error message,
214                                         //  wrong command...
215         {"\0", 0}                       // dummy to see the end of
216                                         // array of structs
217 };

219 //  array of pointer on array of structs
220 //  used to find a valid command, compare each command in array of structs
221 //  until dummy element is found
222 static UART_MEN *pUART_menu[NUM_UART_MENUE_PTRS] = {
223         UART_menu_B,
224         UART_menu_C,
225         UART_menu_F,
226         UART_menu_H,
227         UART_menu_R,
228         UART_menu_S,
229         UART_menu_M,
230         UART_menu_W,
231         UART_menu_Z,
232         UART_menu_D,
233         UART_menu_err
234 };

236 void binblock(int long datatowrite);

238 /*---------------------------------------------------------------------------
239     compare received string from UART
240 ---------------------------------------------------------------------------*/
241 unsigned char UART0_cmp_string(void)
242 /*---------------------------------------------------------------------------*/
243 {
244     static unsigned char sensor_address_index;

246     UART_MEN *pUART_menu_sub;

248     char *pglobal_string = global_string;
249     char *pUART_menu_cmp_1st_char = UART_menu_cmp_1st_char;
250     unsigned char UART_cmd_counter=0;
251     char convert_string[4];
```

```
252     unsigned int UART0_tmp;
253     unsigned char i;

255     switch (UART_menu_position)
256     {
257 //-------------------------------------------------------------------------------
258 //-------------------------------------------------------------------------------
259     case  UART_MENU_FIRSTCHAR:
260         // compare first char
261         sensor_address_index = 0;
262                                             // check if character is allowed:
263         if ((*pglobal_string >= 48) & (*pglobal_string <= 122)) {
264             // look for first command character in small or capital letters
265             // used is the chararray with 6 letters, if all 6 letters aren't
266             // found exit with error
267             do {
268                 if((*pglobal_string == *pUART_menu_cmp_1st_char) |
269                    (*pglobal_string == *pUART_menu_cmp_1st_char-32)) {
270                     break;                  // exit do while() loop
271                 }
272                 pUART_menu_cmp_1st_char++;  // counts compare char one up
273                 UART_cmd_counter++;
274             } while(UART_cmd_counter < NUM_UART_MENUE_PTRS);
275                                             // set pointer on control structur:
276             pUART_menu_sub = pUART_menu[UART_cmd_counter];

278             // if no valid character detected return with error
279             if (pUART_menu_sub[0].cmd == '\0') {
280                 pUART_menu_sub[0].fct();
281                 break;                      // exit switch case
282             }
283             pglobal_string++;               // set global pointer on second char

285             // compare global string with command string.
286             // strcasecmp() compares all characters in command string
287             // with all characters in control structure.
288             // (control structure is the structure with the valid first char)
289             // exit with valid command or when dummy in strucutre found (error).
290             do {
291                 if( strcasecmp( pUART_menu_sub[0].cmd, pglobal_string ) == 0 ) {
292                     if( pUART_menu_sub[0].fct != NULL ) {
293                         (*(pUART_menu_sub[0].fct))();   // execute function
294                         return (0);                     // exit & return with no error
295                     }
296                 }
297                 pUART_menu_sub++;
298             } while(*pUART_menu_sub[0].cmd != '\0');    // until end of commands

300             UART_menu_error();                  // display error if no commands compares
301         }
302         else {
303             UART_menu_error();                  // display error if character not allowed
304         }
305         break;                                  // exit switch case
306 //-------------------------------------------------------------------------------
307 //-------------------------------------------------------------------------------
308     case UART_MENU_CLOCK:           // set rtc via UART
309         UART_cmd_counter = strlen(pglobal_string);

311         // check string for "EXIT"
312         // if exit jump back to main menu
313         if (strcasecmp(pglobal_string, "EXIT")==0)
314         {
```

```
315             UART0_send_text(UART_menu_cmd[8]);
316             UART_menu_position = UART_MENU_FIRSTCHAR;
317             return(0);
318         }
319         // check string for "DATE"
320         // if exit jump back to main menu
321         else if (strcasecmp(pglobal_string, "DATE")==0)
322         {
323             UART0_send_text(UART_menu_cmd[2]);
324             UART_menu_position = UART_MENU_DATE;
325             return(0);
326         }
327         // string to short, error
328         else if (UART_cmd_counter<10)
329         {
330             UART_menu_setrtc();     // display error
331             break;
332         }
333         // check each char
334         while(((*pglobal_string > 43) && (*pglobal_string < 58)))
335         {
336             UART_cmd_counter--;
337             pglobal_string++;   // set global pointer on next char
338         }
339         // if only one char isn't ok, error
340         if(UART_cmd_counter != 0)
341         {
342             UART_menu_setrtc();     // display error
343             break;
344         }

346         // convert weekday string to int, if value valid store else exit
347         // with error
348         convert_string[0] = '0';
349         convert_string[1] = global_string[0];
350         UART0_tmp = atoi(convert_string);
351         if ((UART0_tmp >= 0)&&(UART0_tmp < 7))
352                 RTC_values.wday = (unsigned char)UART0_tmp;
353         else
354         {
355             UART_menu_setrtc();     // display error
356             break;
357         }

359         // convert hour string to int, if value valid store else exit with
360         // error
361         convert_string[0] = global_string[2];
362         convert_string[1] = global_string[3];
363         UART0_tmp = atoi(convert_string);
364         if ((UART0_tmp >= 0)&&(UART0_tmp < 24))
365             RTC_values.hr = (unsigned char)UART0_tmp;
366         else
367         {
368             UART_menu_setrtc();     // display error
369             break;
370         }

372         // convert minute string to int, if value valid store else exit with
373         // error
374         convert_string[0] = global_string[5];
375         convert_string[1] = global_string[6];
376         UART0_tmp = atoi(convert_string);
377         if ((UART0_tmp >= 0)&&(UART0_tmp < 60))
```

```
378             RTC_values.min = (unsigned char)UART0_tmp;
379         else
380         {
381             UART_menu_setrtc();     // display error
382             break;
383         }

385         // convert seconds string to int, if value valid store else exit
386         // with error
387         convert_string[0] = global_string[8];
388         convert_string[1] = global_string[9];
389         UART0_tmp = atoi(convert_string);
390         if ((UART0_tmp >= 0)&&(UART0_tmp < 60))
391             RTC_values.sec = (unsigned char)UART0_tmp;
392         else
393         {
394             UART_menu_setrtc();     // display error
395             break;
396         }

398         UART0_send_text(UART_menu_cmd[2]);// print next command, enter date
399         UART_menu_position = UART_MENU_DATE;
400         return(0);                                  // return with no error

402         break;
403 //-------------------------------------------------------------------------------
404 //-------------------------------------------------------------------------------
405     case UART_MENU_DATE:            // set rtc via UART
406         UART_cmd_counter = strlen(pglobal_string);

408         // compare string if "EXIT" send and if the length and information
409         // is OK if exit jump back to main menu
410         if (strcasecmp(pglobal_string, "EXIT")==0)
411         {
412             UART0_send_text(UART_menu_cmd[8]);
413             UART_menu_position = UART_MENU_FIRSTCHAR;
414             return(0);
415         }
416         // string to short, error
417         else if (UART_cmd_counter<10)
418         {
419             UART0_send_text(UART_menu_cmd[2]);  // display error
420             break;
421         }
422         // check each char
423         while(((*pglobal_string > 43) && (*pglobal_string < 58)))
424         {
425             UART_cmd_counter--;
426             pglobal_string++;   // set global pointer on next char
427         }
428         // if only one char isn't ok, error
429         if(UART_cmd_counter != 0)
430         {
431             UART0_send_text(UART_menu_cmd[2]);  // display error
432             break;
433         }

435         // convert day string to int, if value valid store else exit with
436         // error
437         convert_string[0] = global_string[0];
438         convert_string[1] = global_string[1];
439         UART0_tmp = atoi(convert_string);
440         if ((UART0_tmp >= 0)&&(UART0_tmp < 32))
```

```
441                 RTC_values.day = (unsigned char)UART0_tmp;
442         else
443         {
444             UART0_send_text(UART_menu_cmd[2]);  // display error
445             break;
446         }
447
448         // convert month string to int, if value valid store else exit with
449         // error
450         convert_string[0] = global_string[3];
451         convert_string[1] = global_string[4];
452         UART0_tmp = atoi(convert_string);
453         if ((UART0_tmp >= 1)&&(UART0_tmp < 13))
454             RTC_values.month = (unsigned char)(UART0_tmp - 1);
455         else
456         {
457             UART0_send_text(UART_menu_cmd[2]);  // display error
458             break;
459         }
460
461         // convert year string to int, if value valid store else exit with
462         // error
463         convert_string[0] = global_string[6];
464         convert_string[1] = global_string[7];
465         convert_string[2] = global_string[8];
466         convert_string[3] = global_string[9];
467         UART0_tmp = atoi(convert_string);
468         if ((UART0_tmp >= 0)&&(UART0_tmp < 4000))
469             RTC_values.year = UART0_tmp;
470         else
471         {
472             UART0_send_text(UART_menu_cmd[2]);  // display error
473             break;
474         }
475
476         UART0_send_text(UART_menu_cmd[8]);
477         // print next command, enter date
478         UART_menu_position = UART_MENU_FIRSTCHAR;
479         return(0);                                  // return with no error
480
481         break;
482 //-------------------------------------------------------------------------------
483 //-------------------------------------------------------------------------------
484     case UART_MENU_SENSORADDR:      // set sensor addresses via UART
485         UART_cmd_counter = strlen(pglobal_string);
486         // compare string if "EXIT" send and if the length and information
487         // iss OK if exit jump back to main menu
488         if (strcasecmp(pglobal_string, "EXIT")==0)
489         {
490             UART0_send_text(UART_menu_cmd[8]);
491             UART_menu_position = UART_MENU_FIRSTCHAR;
492             return(0);
493         }
494
495         // check each char
496         while(((*pglobal_string > 47) && (*pglobal_string < 58)))
497         {
498             UART_cmd_counter--;
499             pglobal_string++;   // set global pointer on next char
500         }
501
502         // Enter the maximum number of sensors
503         if(sensor_address_index == 0)
```

```
504         {
505             // if only one char isn't ok, error
506             if(UART_cmd_counter != 0)
507             {
508                 UART0_send_text(UART_menu_cmd[3]);// display error
509                 break;
510             }
511
512             // convert string to integer
513             UART0_tmp = atoi(global_string);
514
515             // only 4 options: 6,12,24,40 sensors configured
516             if ((UART0_tmp == 6) || (UART0_tmp == 12) ||
517                 (UART0_tmp == 24) || (UART0_tmp == 40))
518             {
519                 if (UART0_tmp < SENSOR_active)
520                 {
521                     for (i=0; i < SENSOR_active; i++)
522                         SENSOR_address[i] = 0;
523                 }
524                 SENSOR_active = (unsigned char)UART0_tmp;
525
526             }
527             else    // invalid value, exit
528             {
529                 UART0_send_text(UART_menu_cmd[3]);
530                 // display error
531                 break;
532                 // exit switch case
533             }
534
535             sensor_address_index = 1;
536             UART0_send_text(UART_menu_cmd[4]);
537             return(0);
538         }
539         // Enter the address of each sensors
540         else
541         {
542             // if only one char isn't ok, error
543             if(UART_cmd_counter != 0)
544             {
545                 UART0_send_text(UART_menu_cmd[4]);// display error
546                 break;
547             }
548
549             UART0_tmp = atoi(global_string);
550             SENSOR_address[(sensor_address_index-1)] =
551             (unsigned char)UART0_tmp;   // store sensor address
552             sensor_address_index++;
553
554             // all sensors configured
555             if (sensor_address_index > SENSOR_active)
556             {
557                 UART0_send_text(UART_menu_cmd[8]);
558                 UART_menu_position = UART_MENU_FIRSTCHAR;
559                 sensor_address_index = 0;
560                 FLASH_Write_Controll_Data();
561                 for (i=0; i<SENSOR_active; i++)
562                 {
563                     SENSOR_last_data[0][i] = SENSOR_address[i];
564                 }
565
566                 // set sensors active
```

```
567                 switch(SENSOR_active)
568                 {
569                 case 12:
570                     SENSOR_addr_req[0] = 0xFF;
571                     // 12 sensors are configured
572                     SENSOR_addr_req[1] = 0x0F;
573                     SENSOR_addr_receiv[0] = 0xFF;
574                     SENSOR_addr_receiv[1] = 0x0F;
575                     break;
576                 case 24:
577                     SENSOR_addr_req[0] = 0xFF;
578                     // 24 sensors are configured
579                     SENSOR_addr_req[1] = 0xFF;
580                     SENSOR_addr_req[2] = 0xFF;
581                     SENSOR_addr_receiv[0] = 0xFF;
582                     SENSOR_addr_receiv[1] = 0xFF;
583                     SENSOR_addr_receiv[2] = 0xFF;
584                     break;
585                 case 40:
586                     SENSOR_addr_req[0] = 0xFF;
587                     // 40 sensors are configured
588                     SENSOR_addr_req[1] = 0xFF;
589                     SENSOR_addr_req[2] = 0xFF;
590                     SENSOR_addr_req[3] = 0xFF;
591                     SENSOR_addr_req[4] = 0xFF;
592                     SENSOR_addr_receiv[0] = 0xFF;
593                     SENSOR_addr_receiv[1] = 0xFF;
594                     SENSOR_addr_receiv[2] = 0xFF;
595                     SENSOR_addr_receiv[3] = 0xFF;
596                     SENSOR_addr_receiv[4] = 0xFF;
597                     break;
598                 default:
599                     SENSOR_addr_req[0] = 0x3F;
600                     // 6 sensors are configured
601                     SENSOR_addr_receiv[0] = 0x3F;
602                     break;
603                 }
604             }
605             return(0);
606         }
607         break;
608 //-------------------------------------------------------------------------------------
609 //-------------------------------------------------------------------------------------
610     case UART_MENU_SCAN:                        // scan sensors via UART
611         UART_cmd_counter = strlen(pglobal_string);

613         // compare string if "EXIT" send and if the length and information
614         // iss OK if exit jump back to main menu
615         if (strcasecmp(pglobal_string, "EXIT")==0) {
616             UART0_send_text(UART_menu_cmd[8]);
617             UART_menu_position = UART_MENU_FIRSTCHAR;
618             return(0);
619         }
620                                                 // check each char:
621         while(((*pglobal_string > 47) && (*pglobal_string < 58))) {
622             UART_cmd_counter--;
623             pglobal_string++;                       // set global pointer on next char
624         }

626         // Enter the maximum number of sensors

628         if(UART_cmd_counter != 0) {             // if only one char isn't ok, error
629             UART0_send_text(UART_menu_cmd[3]);      // display error
```

```
630             break;
631         }

633         UART0_tmp = atoi(global_string);        // convert string to integer

635         // only 4 options: 6,12,24,40 sensors configured
636         if ((UART0_tmp == 6) || (UART0_tmp == 12) ||
637             (UART0_tmp == 24) || (UART0_tmp == 40))
638         {
639             if (UART0_tmp < SENSOR_active) {
640                 for (i=0; i < SENSOR_active; i++)
641                     SENSOR_address[i] = 0;
642             }
643             SCAN((unsigned char)UART0_tmp);
644         }
645         else {                                  // invalid value, exit
646             UART0_send_text(UART_menu_cmd[3]);      // display error
647             break;                                  // exit switch case
648         }
649         UART_menu_position = UART_MENU_FIRSTCHAR;
650         return(0);

652 //-------------------------------------------------------------------------------
653 //-------------------------------------------------------------------------------
654     case UART_MENU_CAL:
655         UART_cmd_counter = strlen(pglobal_string);
656         // compare string if "EXIT" send and if the length and information
657         // iss OK if exit jump back to main menu
658         if (strcasecmp(pglobal_string, "EXIT")==0)
659         {
660             UART0_send_text(UART_menu_cmd[8]);
661             UART_menu_position = UART_MENU_FIRSTCHAR;
662             return(0);
663         }

665         // check each char
666         while(((*pglobal_string > 47) && (*pglobal_string < 58)))
667         {
668             UART_cmd_counter--;
669             pglobal_string++;   // set global pointer on next char
670         }

672         // Enter the maximum number of sensors

674         // if only one char isn't ok, error
675         if(UART_cmd_counter != 0)
676         {
677             UART0_send_text(UART_menu_cmd[3]);  // display error
678             break;
679         }

681         // convert string to integer
682         UART0_tmp = atoi(global_string);

684         // only between 0 and 80°C
685         if ((UART0_tmp >= 0) && (UART0_tmp <= 800))
686         {
687             if (UART0_tmp < SENSOR_active)
688             {
689                 for (i=0; i < SENSOR_active; i++)
690                     SENSOR_address[i] = 0;
691             }
692             CALIBRATE((unsigned long)UART0_tmp);
```

```
694             }
695             else
696             // invalid value, exit
697             {
698                 UART0_send_text(UART_menu_cmd[3]);
699                 // display error
700                 break;
701                 // exit switch case
702             }
703             UART_menu_position = UART_MENU_FIRSTCHAR;
704             return(0);
705         }
706         return(1);                      // return with error
707     }
708
709     /*---------------------------------------------------------------------------
710         UART menu function to switch backlight
711     ---------------------------------------------------------------------------*/
712     void UART_menu_bl_on(void)
713     /*---------------------------------------------------------------------------*/
714     {
715         BL_ON;
716         UART0_send_text("Backlight ON\n");
717     }
718
719     /*---------------------------------------------------------------------------
720         UART menu function to clear the flash
721     ---------------------------------------------------------------------------*/
722     void UART_menu_bl_off(void)
723     /*---------------------------------------------------------------------------*/
724     {
725         BL_OFF;
726         UART0_send_text("Backlight OFF\n");
727     }
728
729     /*---------------------------------------------------------------------------
730         UART menu function to clear the flash
731     ---------------------------------------------------------------------------*/
732     void UART_menu_flash_clr(void)
733     /*---------------------------------------------------------------------------*/
734     {
735         UART0_send_text(UART_menu_cmd[6]);
736         Erase_Flash_All();
737     }
738
739     /*---------------------------------------------------------------------------
740         UART menu function to send error
741     ---------------------------------------------------------------------------*/
742     void UART_menu_error(void)
743     /*---------------------------------------------------------------------------*/
744     {
745         UART0_send_text(UART_menu_cmd[7]);
746     }
747
748     /*---------------------------------------------------------------------------
749         UART menu function to show help screen
750     ---------------------------------------------------------------------------*/
751     void UART_menu_show_help(void)
752     /*---------------------------------------------------------------------------*/
753     {
754         UART0_send_text(UART_menu_cmd[0]);
755     }
```

```
757 /*---------------------------------------------------------------------------------------
758     UART menu function to show help screen
759 ---------------------------------------------------------------------------------------*/
760 void UART_menu_setrtc(void)
761 /*---------------------------------------------------------------------------------------*/
762 {
763     UART0_send_text(UART_menu_cmd[1]);
764     UART_menu_position = UART_MENU_CLOCK;
765 }

767 /*---------------------------------------------------------------------------------------
768     UART menu function to scan sensors
769 ---------------------------------------------------------------------------------------*/
770 void UART_menu_scan(void)
771 /*---------------------------------------------------------------------------------------*/
772 {
773     UART0_send_text(UART_menu_cmd[3]);
774     UART_menu_position = UART_MENU_SCAN;
775 }

777 /*---------------------------------------------------------------------------------------
778     UART menu function to cal temperature value from sensors
779 ---------------------------------------------------------------------------------------*/
780 void UART_menu_cal(void)
781 /*---------------------------------------------------------------------------------------*/
782 {
783     UART0_send_text(UART_menu_cmd[9]);
784     UART_menu_position = UART_MENU_CAL;
785 }

787 /*---------------------------------------------------------------------------------------
788     UART menu function to show help screen
789 ---------------------------------------------------------------------------------------*/
790 void UART_menu_setsensor(void)
791 /*---------------------------------------------------------------------------------------*/
792 {
793     UART0_send_text(UART_menu_cmd[3]);
794     UART_menu_position = UART_MENU_SENSORADDR;
795 }

797 /*---------------------------------------------------------------------------------------
798     UART menu function to read the whole measurement from flash
799 ---------------------------------------------------------------------------------------*/
800 void UART_menu_read_data(void)
801 /*---------------------------------------------------------------------------------------*/
802 {
803     DATA_STRUCT *pData;
804     unsigned int year_temp;
805     unsigned char temp, count;
806     unsigned char control;
807     // flash temp variables
808     unsigned char  tmp_Ring_Full;
809     unsigned short tmp_Index_Ring;
810     unsigned short tmp_Count_Ring;
811     unsigned short tmp_Index_Last;

813     recording_stop();   // disable radio modul/ADC
814     control = 1;

816     tmp_Ring_Full = Ring_Full;
817     tmp_Index_Ring = Index_Ring;
818     tmp_Count_Ring = Count_Ring;
```

```
819     tmp_Index_Last = Index_Last;


822     PRINTF_UART;        // set printf to UART

824     while(control)
825     {
826         printf("Control Data\r\n");
827         printf("Capacity: %4li Ah\r\n", BATT_CAPACITY);
828         printf("Min cell voltage: %1li.%2li V \r\n",
829             CELL_VOLT_MIN/100, CELL_VOLT_MIN%100);
830         printf("Max cell voltage: %1li.%2li V \r\n",
831             CELL_VOLT_MAX/100, CELL_VOLT_MAX%100);
832         printf("SOC (end of record): %3li %%\r\n",
833             (g_currah.AH_MASTER*44 / BATT_CAPACITY ));
834         printf("Current Tara value: %4li \r\n", g_currmeas.tara);
835         for(count=0;count<SENSOR_active;count++)
836         {
837             printf("Nummer: %2x ", count+1);
838             if(SENSOR_temp_cal_b[SENSOR_address[count]]<0)
839             {
840                 printf("Adress: %2x -", SENSOR_address[count]);
841                 UART0_send((SENSOR_temp_cal_b[SENSOR_address
842                     [count]]/100*-1)+48);
843                 UART0_send(((SENSOR_temp_cal_b[SENSOR_address
844                     [count]]%100)/10*-1)+48);
845                 UART0_send((SENSOR_temp_cal_b[SENSOR_address
846                     [count]]%10*-1)+48);
847             }
848             else
849             {
850                 printf("Adress: %2x  ", SENSOR_address[count]);
851                 UART0_send((SENSOR_temp_cal_b[SENSOR_address
852                     [count]]/100)+48);
853                 UART0_send(((SENSOR_temp_cal_b[SENSOR_address
854                     [count]]%100)/10)+48);
855                 UART0_send((SENSOR_temp_cal_b[SENSOR_address
856                     [count]]%10)+48);
857             }
858             printf("\r\n");

860         }

862         pData = Read_Ring_Flash();
863         while(pData != NULL)
864         {
865             // send weekday
866             UART0_send_text(RTC_days[pData->C4]);

868             // send date
869             UART0_send(pData->C0/10+48);
870             UART0_send(pData->C0%10+48);
871             UART0_send('-');
872             UART0_send_text(RTC_months[pData->C1]);
873             UART0_send('-');
874             // get year integer value
875             year_temp = (pData->C2 << 8) | pData->C3;
876             temp = year_temp / 1000;
877             UART0_send(temp+48);
878             temp = (year_temp / 100) - ((year_temp/1000) * 10);
879             UART0_send(temp+48);
880             temp = (year_temp / 10) - ((year_temp/100) * 10);
881             UART0_send(temp+48);
```

```
882             temp = year_temp % 10;
883             UART0_send(temp+48);
884             UART0_send(' ');

886             // send time
887             UART0_send(pData->C5/10+48);
888             UART0_send(pData->C5%10+48);
889             UART0_send(':');
890             UART0_send(pData->C6/10+48);
891             UART0_send(pData->C6%10+48);
892             UART0_send_text(":");
893             UART0_send(pData->C7/10+48);
894             UART0_send(pData->C7%10+48);

896             // send temperature, cell-voltage and supply-voltage
897             // values in hex
898             printf(" %2x %2x %2x %2x %2x %2x %2x %2x\r\n",
899                     pData->C8, pData->C9,pData->C10, pData->C11,
900                     pData->C12, pData->C13, pData->C14,
901                     pData->C15); //, pData->C16, pData->C17);

903             pData = Read_Ring_Flash();
904         };

906         UART0_send_text("OK\r\n");
907         LCD_send_cmd(LCD_LINE1);
908         LCD_send_text(" Data received? ");
909         LCD_send_cmd(LCD_LINE2);
910         LCD_send_text("YES    NO       ");

912         while(1)
913         {
914             if(BATMON_control_reg & B1) // Button 1 = YES,
915             {
916                 control = 0;
917                 BATMON_control_reg &= ~(B1|B2|B3);
918                 // clear button state

920                 LCD_send_cmd(LCD_LINE1);
921                 if (Enable_clear_lcd == ON)
922                     LCD_send_text("                ");
923                 else if (Enable_clear_lcd == STATUS)
924                     LCD_send_text(" Radio disabled ");

926                 // set time on second LCD line
927                 LCD_send_time();

929                 RADIO_init_frame();
930                 // write first frame with actual time and 0 as data
931                 FLASH_Write_Controll_Data();

933                 break;
934                     // exit while loop
935             }
936             if(BATMON_control_reg & B2) // Button 2 = NO,
937             {
938                 Ring_Full  = tmp_Ring_Full;
939                 Index_Ring = tmp_Index_Ring;
940                 Count_Ring = tmp_Count_Ring;
941                 Index_Last = tmp_Index_Last;
942                 BATMON_control_reg &= ~(B1|B2|B3);
943                 // clear button state
944                 break;
```

```
945                 // exit while loop
946             }
947         }
948     }
949     /*  begin with new frame    */
950     //RADIO_enable();       // enable radio modul
951 }

953 /*---------------------------------------------------------------------------
954     UART menu function to read controll data from flash
955 ---------------------------------------------------------------------------*/
956 void UART_menu_read_controll(void)
957 /*---------------------------------------------------------------------------*/
958 {
959     UART0_send_text("Read data\n");
960     FLASH_Read_Controll_Data();
961 }

963 /*---------------------------------------------------------------------------
964     UART menu function to write controll data to flash
965 ---------------------------------------------------------------------------*/
966 void UART_menu_write_controll(void)
967 /*---------------------------------------------------------------------------*/
968 {
969     UART0_send_text("Write data\n");
970     FLASH_Write_Controll_Data();
971 }
972 /*---------------------------------------------------------------------------
973     UART menu start FER calc.
974 ---------------------------------------------------------------------------*/
975 void UART_menu_FER_calc(void)
976 /*---------------------------------------------------------------------------*/
977 {
978     //start if a measuring is running and fer is off
979     if ((fer < 1) && (measuring > 0)) {
980         fer = 1;
981     }
982     //stop fer if it is already on
983     else if ((fer > 0) && (measuring > 0)) {
984         fer = 0;
985         UART0_send_text("fer count off!\n");
986     } else {
987         UART0_send_text("Please start the measurement first\n");
988     }
989 }
990 /*---------------------------------------------------------------------------
991     UART menu invert current
992 ---------------------------------------------------------------------------*/
993 void UART_menu_invert_cur(void)
994 /*---------------------------------------------------------------------------*/
995 {
996     g_currmeas.current_invert ^= 1;
997     if(g_currmeas.current_invert)
998         UART0_send_text("Current invert: ON\n");
999     else
1000        UART0_send_text("Current invert: OFF\n");
1001 }

1003 /*---------------------------------------------------------------------------
1004     UART menu tara current
1005 ---------------------------------------------------------------------------*/
1006 void UART_menu_tara_cur(void)
1007 /*---------------------------------------------------------------------------*/
```

```
1008  {
1009      #ifdef ENABLE_CURRENT_MEASURE
1010          scan_mode = 4;
1011          ADC_start();
1012      #endif
1013  }
1014  /*---------------------------------------------------------------------------
1015      UART menu RX 433MHz on
1016  ---------------------------------------------------------------------------*/
1017  void UART_menu_rx_on(void)
1018  /*---------------------------------------------------------------------------*/
1019  {
1020      RADIO_enable();
1021      UART0_send_text("RX 433MHz: ON\n");
1022  }
1023  /*---------------------------------------------------------------------------
1024      UART menu RX 433MHz off
1025  ---------------------------------------------------------------------------*/
1026  void UART_menu_rx_off(void)
1027  /*---------------------------------------------------------------------------*/
1028  {
1029      RADIO_disable();
1030      UART0_send_text("RX 433MHz: OFF\n");
1031  }
1032  /*---------------------------------------------------------------------------
1033      UART menu function for enable/disable sensor data screening on lcd
1034  ---------------------------------------------------------------------------*/
1035  void UART_menu_lcd_sensor_data(void)
1036  /*---------------------------------------------------------------------------*/
1037  {
1038      if(g_show_sensor_lcd == TRUE) {
1039          g_show_sensor_lcd = FALSE;
1040          UART0_send_text("Show sensor data on LCD: OFF\n");
1041      }
1042      else {
1043          g_show_sensor_lcd = TRUE;
1044          UART0_send_text("Show sensor data on LCD: ON\n");
1045      }
1046  }
1047  /*---------------------------------------------------------------------------
1048      UART menu function for choice of sensor data screening on uart
1049  ---------------------------------------------------------------------------*/
1050  void UART_menue_uart_show_dechex(void)
1051  /*---------------------------------------------------------------------------*/
1052  {
1053      if(g_show_sensor_data_uart == NONE) {
1054          g_show_sensor_data_uart = DEC;
1055          UART0_send_text("Show sensor data on UART: DEC\n");
1056      }
1057      else if(g_show_sensor_data_uart == DEC) {
1058          g_show_sensor_data_uart = HEX;
1059          UART0_send_text("Show sensor data on UART: HEX\n");
1060      }
1061      else {
1062          g_show_sensor_data_uart = NONE;
1063          UART0_send_text("Show sensor data on UART: OFF\n");
1064      }
1065  }
1066  /*---------------------------------------------------------------------------
1067      UART menu function for ZS CMD wakeup-pulse
1068  ---------------------------------------------------------------------------*/
1069  void UART_menu_zs_wakeup_pulse(void)
1070  /*---------------------------------------------------------------------------*/
```

```
1071  {
1072      zs_wakeup_pulse();
1073  }
1074  /*---------------------------------------------------------------------------------------
1075      UART menu function for wakeup with cw
1076  ---------------------------------------------------------------------------------------*/
1077  void UART_menu_zs_wakeup_cw(void)
1078  /*---------------------------------------------------------------------------------------*/
1079  {
1080      if(g_reader_tx.cw == OFF) {
1081          zs_wakeup_cw(TRUE);
1082      }
1083      else {
1084          zs_wakeup_cw(FALSE);
1085      }
1086  }
1087  /*---------------------------------------------------------------------------------------
1088      UART menu function for ZS CMD shut down all
1089  ---------------------------------------------------------------------------------------*/
1090  void UART_menu_zs_shutdown_all(void)
1091  /*---------------------------------------------------------------------------------------*/
1092  {
1093      zs_shutdown_all();
1094  }
1095  /*---------------------------------------------------------------------------------------
1096      UART menu function for ZS CMD send synch
1097  ---------------------------------------------------------------------------------------*/
1098  void UART_menu_zs_synch(void)
1099  /*---------------------------------------------------------------------------------------*/
1100  {
1101      zs_send_synch();
1102  }
1103  /*---------------------------------------------------------------------------------------
1104      UART menu function for ZS CMD send dataset 1
1105  ---------------------------------------------------------------------------------------*/
1106  void UART_menu_zs_ds_1(void)
1107  /*---------------------------------------------------------------------------------------*/
1108  {
1109      zs_send_set1();
1110  }
1111  /*---------------------------------------------------------------------------------------
1112      UART menu function for ZS CMD send dataset 2
1113  ---------------------------------------------------------------------------------------*/
1114  void UART_menu_zs_ds_2(void)
1115  /*---------------------------------------------------------------------------------------*/
1116  {
1117      zs_send_set2();
1118  }
1119  /*---------------------------------------------------------------------------------------
1120      UART menu function for ZS CMD set tx time
1121  ---------------------------------------------------------------------------------------*/
1122  void UART_menu_zs_set_time_tx(void)
1123  /*---------------------------------------------------------------------------------------*/
1124  {
1125      zs_send_time_tx();
1126  }
1127  /*---------------------------------------------------------------------------------------
1128      UART menu function for ZS CMD set sample time
1129  ---------------------------------------------------------------------------------------*/
1130  void UART_menu_zs_set_time_sample(void)
1131  /*---------------------------------------------------------------------------------------*/
1132  {
1133      zs_send_time_sample();
```

```
1134  }
1135  /*---------------------------------------------------------------------------------------
1136      UART menu function for ZS CMD set interval time
1137  ---------------------------------------------------------------------------------------*/
1138  void UART_menu_zs_set_time_interval(void)
1139  /*---------------------------------------------------------------------------------------*/
1140  {
1141      zs_send_time_interval();
1142  }
1143  /*---------------------------------------------------------------------------------------
1144      UART menu function for ZS CMD balance
1145  ---------------------------------------------------------------------------------------*/
1146  void UART_menu_zs_balance(void)
1147  /*---------------------------------------------------------------------------------------*/
1148  {
1149      if(g_zs_balance == OFF) {
1150          zs_send_balance(TRUE);
1151      }
1152      else {
1153          zs_send_balance(FALSE);
1154      }
1155  }
1156  /*---------------------------------------------------------------------------------------
1157      UART menu function for ZS CMD sent reply
1158  ---------------------------------------------------------------------------------------*/
1159  void UART_menu_zs_reply(void)
1160  /*---------------------------------------------------------------------------------------*/
1161  {
1162      zs_send_reply();
1163  }
1164  /*---------------------------------------------------------------------------------------
1165      UART menu function for ZS CMD Save Settings
1166  ---------------------------------------------------------------------------------------*/
1167  void UART_menu_zs_save_set(void)
1168  /*---------------------------------------------------------------------------------------*/
1169  {
1170      zs_send_save_set();
1171  }
1172  //-------------------------------------------------------------------------------------------
```

Programmausdruck C.47: *Basisstation „uart_menu.c“*

```
/*-------------------------------------------------------------------------------------------
    Project:            BATSEN
    Discription:        Header-File for handling of RS-232-Interface
    Used components:    MSP430-169STK

    Author:             Stephan Plaschke
    Date:               02/14/2008
    Last update:        04/04/2008

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

    Modified by:        Niels Jegenhorst
    Last modification:  08/07/2011

    File:               uart_menue.h
-------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------
 Config for Serial Terminal: ( HTerm - http://www.der-hammer.info/terminal )

    Baudrate            = 115200
    Databits            = 8
    Stopbits            = 1
    Parity              = None
    CTS flow control    = Off

    RX-Side:
        ASCII Dataformat,
        Newline @ LF,

    TX-Side:
        ASCII Dataformat,
        Newline @ LF,
-------------------------------------------------------------------------------------------*/

#ifndef UART_MENUE_H_
#define UART_MENUE_H_

#include "headerfiles/typedefs.h"

/*-------------------------------------------------------------------------------------------
    Definition
-------------------------------------------------------------------------------------------*/
#define NUM_UART_MENUE_PTRS     11              // Number of Pointers to structure arrays
                                                // for command decoding

/*-------------------------------------------------------------------------------------------
    Prototypes
-------------------------------------------------------------------------------------------*/

void UART_menu_bl_on(void);
void UART_menu_bl_off(void);
void UART_menu_cal(void);
void UART_menu_scan(void);
void UART_menu_flash_clr(void);
void UART_menu_error(void);
void UART_menu_show_help(void);
void UART_menu_read_data(void);
void UART_menu_setrtc(void);
void UART_menu_setsensor(void);
void UART_menu_read_controll(void);
void UART_menu_write_controll(void);
```

```
64  void UART_menu_FER_calc(void);
65  void UART_menu_invert_cur(void);
66  void UART_menu_tara_cur(void);

68  void UART_menu_rx_on(void);
69  void UART_menu_rx_off(void);
70  void UART_menu_lcd_sensor_data(void);
71  void UART_menue_uart_show_dechex(void);

73  void UART_menu_zs_wakeup_pulse(void);
74  void UART_menu_zs_wakeup_cw(void);
75  void UART_menu_zs_shutdown_all(void);
76  void UART_menu_zs_synch(void);
77  void UART_menu_zs_ds_1(void);
78  void UART_menu_zs_ds_2(void);
79  void UART_menu_zs_set_time_tx(void);
80  void UART_menu_zs_set_time_sample(void);
81  void UART_menu_zs_set_time_interval(void);
82  void UART_menu_zs_balance(void);
83  void UART_menu_zs_reply(void);
84  void UART_menu_zs_save_set(void);

86  #endif /*UART_MENUE_H_*/
```

Programmausdruck C.48: *Basisstation „uart_menu.h“*

C.2 Matlab

C.2.1 Datenlogger

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      21.10.2011
% Beschreibung:     Skript zum Aufnehmen einer Messreihe, mit Fernsteuerung der
%                   Basisstation sowie der Zellensensoren
%
% Funktion:         OK
% Datei:            datalogger.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

close all;
clear all;

disp('--- Start datalogger ------------');

%% --- User Adjustments ------------------------------------------------------------------
[user_input, stop] = make_gui();
if(stop == true)
    error('break by user');
end

%% --- Open COM Port ---------------------------------------------------------------------
display('Opening COM Port');
try
    fclose(serial_obj);
    display('COM Port was open!');
catch E
end
serial_obj = comport_open('COM1');

%% --- Prepare Graphs & Data -------------------------------------------------------------
display('Prepare Graphs & Data');
                                            % create struct timestamp:
ts = struct('year', 0, 'month', 0, 'day', 0,...
            'hour', 0, 'min', 0, 'sec', 0.0, 'msec', 0,...
            'str', 'dd-mmm-yyyy HH:MM:SS.FFF', 'num', 0.0);
                                            % create array of timestamps
ts(1:user_input.num_sensors_rec+1, 1:user_input.num_sample+1, 1) = ts;
                                            % create struct for saving all sensordate:
sensordata = struct('address', zeros(user_input.num_sensors_rec+1, 1, 'uint32'),...
                    'vid', zeros(user_input.num_sensors_rec+1, 1, 'uint16'),...
                    'sample', zeros(user_input.num_sensors_rec+1, 1),...
                    'timestamp', ts,...
                    'cv', zeros(user_input.num_sensors_rec+1, user_input.num_sample+1, '
                        double'),...
                    'sv', zeros(user_input.num_sensors_rec+1, user_input.num_sample+1, '
                        double'),...
                    'av', zeros(user_input.num_sensors_rec+1, user_input.num_sample+1, '
                        double'),...
                    'lv', zeros(user_input.num_sensors_rec+1, user_input.num_sample+1, '
                        double'),...
                    'temperature', zeros(user_input.num_sensors_rec+1, user_input.
                        num_sample+1, 'double'),...
                    'current', zeros(1, user_input.num_sample+1, 'double'),...
```

```
52                      'current_channel', zeros(1, user_input.num_sample+1, 'uint16'),...
53                      'current_dir',     zeros(1, user_input.num_sample+1, 'uint16'));
54                                                  % create struct for rx data from comport:
55  rxdata = struct('timestamp', ts(1,1),...
56                  'sensor', 0, 'address', uint32(0), 'vid', uint16(0),...
57                  'cv', double(0), 'sv', double(0), 'av', double(0), 'lv', double(0),...
58                  'temperature', double(0), ...
59                  'current_ch', uint16(0), 'current_dir', uint16(0), 'current', double(0));

61  wait_first_sensor   = true;
62  last_max_sample     = 1;
63  MARKER              = '-o';
64                                                  % prepare graphs
65  [hf, ha, hp, hl, xaxis_start, sensor_display] = prepare_graphs(sensordata, ...
66                                                                 user_input.num_sensors_rec
                                                                       ,...
67                                                                 MARKER);
68  set(hf, 'Visible', 'off');

70  %% --- Stop Recording on Basestation ------------------------------------------------
71  display('Adjust datalogger...');
72  display('Stop recording');
73  bs_stop_recording(serial_obj);

75  %% --- Execute Current Tara CMD -----------------------------------------------------
76  if(user_input.tara == true)
77      display('Current Tara');
78      bs_current_tara(serial_obj);
79  end

81  %% --- Wakeup Sensors ---------------------------------------------------------------
82  ui = questdlg('Now wake-up all sensors?');drawnow;
83  if strcmp(ui, 'Yes')
84      bs_send_wakeup(serial_obj);
85      while (true)
86          ui = questdlg('All Sensors waken?', 'Wake-Up','Yes','No','Yes');drawnow;
87          if strcmp(ui, 'Yes')
88             break;
89          else
90              ui = questdlg('Wake-up again?', 'Wake-Up','Yes','No','Yes');drawnow;
91              if strcmp(ui, 'Yes')
92                  bs_send_wakeup(serial_obj);
93              else
94                  comport_close(serial_obj);
95                  error('Aborted by user: Not all sensors waken!');
96              end
97          end
98      end
99  elseif strcmp(ui, 'Cancel')
100     comport_close(serial_obj);
101     error('break by user');
102 end

104 %% --- Scan for Sensors -------------------------------------------------------------
105 if(user_input.scan_sensors == true)
106     display('Scan for sensors');
107     bs_scan_sensors(serial_obj, user_input.num_sensors_scan);
108 end
109 %% --- Set Dataset on Sensors -------------------------------------------------------
110 if(user_input.send_dataset_sen == true)
111     display('Set Dataset on sensors');
112     bs_send_dataset(serial_obj, user_input.dataset);
113 end
```

```
114 %% --- Set Interval Time on Sensors ------------------------------------------------------
115 if(user_input.send_time_sen == true)
116     display('Set Interval time on sensors');
117     bs_send_time_interval(serial_obj);
118 end
119 %% --- Set Sample Time on Sensors -------------------------------------------------------
120 if(user_input.send_time_sen == true)
121     display('Set Sample time on sensors');
122     bs_send_time_sample(serial_obj);
123 end
124 %% --- Set TX Time on Sensors -----------------------------------------------------------
125 if(user_input.send_time_sen == true)
126     display('Set TX time on sensors');
127     bs_send_time_tx(serial_obj);
128 end
129 %% --- Activate CW for Downlink ---------------------------------------------------------
130 if(user_input.wakeup_cw == true)
131     display('Activate CW for downlink');
132     bs_send_wakeup_cw(serial_obj, true);
133 end
134 %% --- Start Recording ------------------------------------------------------------------
135 ui = questdlg('Press YES key to start recording!','Start','Yes','No','Yes');
136 drawnow;
137 if strcmp(ui, 'Yes')
138     display('Start recording...');
139     bs_start_recording(serial_obj);
140 else
141     comport_close(serial_obj);
142     error('break by user');
143 end

145 %% --- Readout Measured Data ------------------------------------------------------------
146 set(hf, 'Visible', 'on');
147                                                 % create waitbar for status & user break
148 hwaitbar = waitbar(0, '0', 'Name', 'Processing ...',...
149                       'CreateCancelBtn', 'setappdata(gcbf,''canceling'',1)');
150 setappdata(hwaitbar, 'canceling', 0);

152 while(true)                  % infinite loop ------------------------------------------
153                              % --> breaks by sample count or user input on waitbar

155     try                                         % read one line from COM-Port ----------
156         [string_in] = fscanf(serial_obj, '%s\n');
157     catch E2
158         comport_close(serial_obj);
159         delete(hwaitbar);
160         error('Error: fscanf()');
161     end
162     pause(0);                                   % dummy pause to help break execution
163     if isempty(string_in)
164         comport_close(serial_obj);
165         delete(hwaitbar);
166         error('Error: fscanf() isempty');
167     else                                        % decode received datastring: ----------
168         [rxdata, dataok] = decode_datastring(string_in, rxdata);
169         if(dataok == true)                      % when received contains measure data:
170                                                 % wait for first sensor to begin: -------
171             if(((rxdata.sensor==1)||(rxdata.sensor==2)) && (wait_first_sensor == true))
172                 wait_first_sensor = false;
173                 tic;
174                 t_record_start = rxdata.timestamp.str;
175                 display(['  Timestamp start: ' t_record_start]);
176             end
```

```
177             if(wait_first_sensor == false)      % if first sensor was found:
178                                                 % store received data in struct: --------
179                 sensordata = store_rxdata(sensordata, rxdata);
180                                                 % refresh graphs: ----------------------
181                 [xaxis_start, last_max_sample] = refresh_graphs(hp,ha,hl, xaxis_start,...
182                                                                 user_input.samples_graph
                                                                        ,...
183                                                                 last_max_sample,...
184                                                                 user_input.num_sample,...
185                                                                 false,...
186                                                                 sensordata,...
187                                                                 hwaitbar);
188                                                 % show sensor number & address: ---------
189                 if(sensor_display(rxdata.sensor) == true)
190                     sensor_display(rxdata.sensor) = false;
191                     display(['  Sensor #' num2str(rxdata.sensor-1,'%03d') ,...
192                              ': address = 0x' num2str(rxdata.address,'%05x'), ...
193                              ', vid = 0x' num2str(rxdata.vid, '%02x')]);
194                 end
195             end                                 % when specified samples recorded
196                                                 % or by user input on waitbar: ----------
197             if((sensordata.sample(rxdata.sensor) == user_input.num_sample+1) || ...
198                (getappdata(hwaitbar, 'canceling') > 0) )
199                 delete(hwaitbar);                       % close waitbar
200                                                         % refresh graphs:
201                 [xaxis_start, last_max_sample] = refresh_graphs(hp,ha,hl, xaxis_start,...
202                                                                 user_input.samples_graph
                                                                        ,...
203                                                                 last_max_sample,...
204                                                                 user_input.num_sample,...
205                                                                 true,...
206                                                                 sensordata,...
207                                                                 hwaitbar);
208                 t_process = toc;
209                 t_record_stop = rxdata.timestamp.str;
210                 t_record = datestr((datenum(t_record_stop) - datenum(t_record_start)),...
211                                    'dd-mmm-yyyy HH:MM:SS.FFF');
212                 display(['  Timestamp stop:  ' t_record_stop]);
213                 display(['  Logged time:     ' t_record]);
214                 display(['  Elapsed processing time: ' num2str(t_process) 'sec']);
215                 break;
216             end
217         else                                % when received data is not measure data: ---
218             display(['  datalogger: ' string_in]);
219         end
220     end
221 end

223 %% --- Save Data ---------------------------------------------------------------------
224 display('Save Data');

226 dirname = 'recordings';
227 if(exist(dirname, 'dir') == 0)                  % create folder:
228     mkdir(dirname);
229 end

231 filedate = datestr(now, 'yyyymmdd_HH-MM-SS');   % generate timestamp for files
232                                                 % save variables:
233 save([dirname '\' filedate '_sensordata.mat'], 'sensordata', 'user_input');
234                                                 % save figures as pdf and matlab figure:
235 saveas(hf(1), [dirname '\' filedate '_data_1.pdf']);
236 saveas(hf(1), [dirname '\' filedate '_data_1.fig']);
237 saveas(hf(2), [dirname '\' filedate '_data_2.pdf']);
```

```
238  saveas(hf(2), [dirname '\' filedate '_data_2.fig']);

240  %% --- Stop Recording ------------------------------------------------------------
241  display('Stop recording');
242  bs_stop_recording(serial_obj);

244  %% --- Shutdown Sensors ----------------------------------------------------------
245  ui = questdlg('Now shutdown all sensors?','Shutdown Sensors','Yes','No','Yes');
246  drawnow;
247  if strcmp(ui, 'Yes')
248      bs_send_shutdown(serial_obj);
249  end
250  %% --- Deactivate CW for Downlink ------------------------------------------------
251  if(user_input.wakeup_cw == true)
252      display('Deactivate CW for downlink');
253      bs_send_wakeup_cw(serial_obj, false);
254  end
255  %% --- Close COM Port ------------------------------------------------------------
256  display('Close COM Port');
257  comport_close(serial_obj);

259  disp('--- End   datalogger ------------');
260  %-------------------------------------------------------------------------
```

Programmausdruck C.49: *Datenlogger Matlab-Hauptprogramm „datalogger.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      06.10.2011
% Geändert am:      06.10.2011
% Beschreibung:     Funktion zum Ausführen des Ctara-Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_current_tara.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_current_tara( serial_obj )

    fprintf(serial_obj, 'ctara');    % send cmd "ctara" ================================
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'CurrentTaravalue:')) % when CMD executed
            display(['  datalogger: ' string_in]);
            break;
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.50: *Datenlogger Matlab-Funktion „bs_current_tara.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      25.05.2011
% Beschreibung:     Funktion zum Ausführen des Scan-Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_scan_sensors.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_scan_sensors( serial_obj, NUM_SENSORS_SCAN )

    display('  User has to stop scan-mode via button 1 if needed!');
    fprintf(serial_obj, 'scan');            % send cmd "scan for sensors" ===============
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'OK'))                % when CMD finished:
            break;                                  % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.2);
    check = false;
    fprintf(serial_obj, num2str(NUM_SENSORS_SCAN)); % send cmd "X sensors to scan" ======
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'Scanhalted'))       % when scan-mode is halted
            check = true;
        elseif(strfind(string_in, 'Scancomplete')) % when scan-mode found desired sensors
            check = true;
        elseif(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(5);                                       % wait until all sensors finished
end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.51: *Datenlogger Matlab-Funktion „bs_scan_sensors.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      07.06.2011
% Geändert am:      07.06.2011
% Beschreibung:     Funktion zum Ausführen des "Send Dataset" - Kommandos
%                   auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_dataset.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_send_dataset( serial_obj, dataset )

    if(dataset == 1)
        fprintf(serial_obj, 'zs_ds_1');     % send cmd "send dataset 1" =================
    else
        fprintf(serial_obj, 'zs_ds_2');     % send cmd "send dataset 2" =================
    end
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(dataset == 1)
            if(strfind(string_in, 'ZSSampleSet1sent'))% when CMD executed
                check = true;
            end
        else
            if(strfind(string_in, 'ZSSampleSet2sent'))% when CMD executed
                check = true;
            end
        end
        if(strfind(string_in, 'OK'))                % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.5);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.52: *Datenlogger Matlab-Funktion „bs_send_dataset.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      20.10.2011
% Beschreibung:     Funktion zum Ausführen des Shutdown-Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_shutdown.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_send_shutdown( serial_obj )

    for i=1:1:2
        fprintf(serial_obj, 'zs_sht_a');    % send cmd "shutdown all sensors" ===========
        check = false;
        while(true)
            try
                string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
            catch E2
                fclose(serial_obj);
                delete(serial_obj);
                error('Error: fscanf()');
            end
            if(strfind(string_in, 'ZSShutdownallsent'))% when CMD executed
                check = true;
            elseif(strfind(string_in, 'OK'))            % and CMD finished:
                if(check == true) ,break; end           % break while loop
            elseif(isempty(string_in))                  % when an timeout occured:
                fclose(serial_obj);
                delete(serial_obj);
                error('Error: fscanf()');
            end
            display(['  datalogger: ' string_in]);
        end
        pause(0.5);
    end
end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.53: *Datenlogger Matlab-Funktion „bs_send_shutdown.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      14.06.2011
% Geändert am:      14.06.2011
% Beschreibung:     Funktion zum Ausführen des Kommandos "Send Time Interval"
%                   auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_time_interval.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [  ] = bs_send_time_interval( serial_obj )

    fprintf(serial_obj, 'zs_st_iv');     % send cmd "send time interval" ================
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'ZSSetIntervaltimesent'))   % when CMD executed
            check = true;
        elseif(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.5);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.54: *Datenlogger Matlab-Funktion „bs_send_time_interval.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      26.05.2011
% Geändert am:      26.05.2011
% Beschreibung:     Funktion zum Ausführen des Kommandos "Send Time Sample"
%                   auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_time_sample.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [  ] = bs_send_time_sample( serial_obj )

    fprintf(serial_obj, 'zs_st_sa');     % send cmd "send time sample" ==================
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'ZSSetSampletimesent'))   % when CMD executed
            check = true;
        elseif(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.5);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.55: *Datenlogger Matlab-Funktion „bs_send_time_sample.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      26.05.2011
% Geändert am:      26.05.2011
% Beschreibung:     Funktion zum Ausführen des Kommandos "Send Time TX"
%                   auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_time_tx.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [  ] = bs_send_time_tx( serial_obj )

    fprintf(serial_obj, 'zs_st_tx');     % send cmd "send time tx" ======================
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'ZSSetTXtimesent'))    % when CMD executed
            check = true;
        elseif(strfind(string_in, 'ZSSetTXtimespacingfailure')) % when CMD fails
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: ZS Set TX time spacing failure');
        elseif(strfind(string_in, 'OK'))             % and CMD finished:
            if(check == true) ,break; end            % break while loop
        elseif(isempty(string_in))                   % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.5);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.56: *Datenlogger Matlab-Funktion „bs_send_time_tx.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      25.05.2011
% Beschreibung:     Funktion zum Ausführen des "Wake-up" Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_wakeup.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_send_wakeup( serial_obj )

    fprintf(serial_obj, 'zs_wk_a');     % send cmd "wakeup all sensors" =================
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'ZSWakeuppulsesent'))% when CMD executed
            check = true;
        elseif(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(1);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.57: *Datenlogger Matlab-Funktion „bs_send_wakeup.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      08.06.2011
% Geändert am:      08.06.2011
% Beschreibung:     Funktion zum Ausführen des "Wake-up CW" Kommandos
%                   auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_send_wakeup_cw.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_send_wakeup_cw( serial_obj, set)

    fprintf(serial_obj, 'zs_wk_cw');     % send cmd "wakeup cw" ========================
    actual = -1;
    check  = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'ZSWakeupcwON'))          % when CMD executed and cw is on
            actual = true;
            check  = true;
        elseif(strfind(string_in, 'ZSWakeupcwOFF'))     % when CMD executed and cw is off
            actual = false;
            check  = true;
        end
        if(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) && (actual == set)    % set value = actual value then:
                break;                                  % break while loop and finish
            elseif(check == true) && (actual ~= set)% set value != actual value then:
                fprintf(serial_obj, 'zs_wk_cw');        % send cmd "wakeup cw" again
                check = false;
            end
        elseif(isempty(string_in))              % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end
    pause(0.5);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.58: *Datenlogger Matlab-Funktion „bs_send_wakeup_cw.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      12.07.2011
% Beschreibung:     Funktion zum Ausführen des Start-Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_start_recording.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_start_recording( serial_obj )

    comport_flush(serial_obj);              % empty serial port buffer:

    fprintf(serial_obj, 'start');           % send cmd "radio enable" ====================
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'Radioenabled'))     % when CMD Radio enable executed
            check = true;
        elseif(strfind(string_in, 'OK'))            % and CMD finished:
            if(check == true) ,break; end           % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        display(['  datalogger: ' string_in]);
    end

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.59: *Datenlogger Matlab-Funktion „bs_start_recording.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      25.05.2011
% Beschreibung:     Funktion zum Ausführen des Stop-Kommandos auf der Basisstation
%
% Funktion:         OK
% Datei:            bs_stop_recording.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = bs_stop_recording( serial_obj )

    comport_flush(serial_obj);              % empty serial port buffer:

    fprintf(serial_obj, 'stop');            % send cmd "radio disable" ==================
    cnt = 1;
    check = false;
    while(true)
        try
            string_in = fscanf(serial_obj, '%s\n'); % read one line from COM-Port
        catch E2
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        end
        if(strfind(string_in, 'Radiodisabled'))    % when CMD executed
            check = true;
        elseif(strfind(string_in, 'OK'))            % when CMD finished:
             if(check == true) ,break; end          % break while loop
        elseif(isempty(string_in))                  % when an timeout occured:
            fclose(serial_obj);
            delete(serial_obj);
            error('Error: fscanf()');
        else
            cnt = cnt + 1;
        end
        if(strfind(string_in, 'SEN:'))
        else
            display(['  datalogger: ' string_in]);
        end
        if(cnt == 10)
            fprintf(serial_obj, 'stop');            % send cmd "radio disable" again
            cnt = 1;
        end
    end
    pause(1);

end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.60: *Datenlogger Matlab-Funktion „bs_stop_recording.m“*

```
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      25.05.2011
% Beschreibung:     Funktion zum Schließen einer seriellen Schnittstelle zur Basisstation
%
% Funktion:         OK
% Datei:            comport_open.m
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = comport_close( serial_obj )

    fclose(serial_obj);
    delete(serial_obj);

end
%----------------------------------------------------------------------------------------
```

Programmausdruck C.61: *Datenlogger Matlab-Funktion „comport_close.m“*

```
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      12.07.2011
% Geändert am:      12.07.2011
% Beschreibung:     Funktion zum Leeren des Puffers der seriellen Schnittstelle
%
% Funktion:         OK
% Datei:            comport_flush.m
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ ] = comport_flush( serial_obj )

    try                                     % empty serial port buffer:
        fread(serial_obj, serial_obj.BytesAvailable);
    catch E2    %#ok<NASGU>
    end
end
%----------------------------------------------------------------------------------------
```

Programmausdruck C.62: *Datenlogger Matlab-Funktion „comport_flush.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      25.05.2011
% Geändert am:      25.05.2011
% Beschreibung:     Funktion zum Öffnen einer seriellen Schnittstelle zur Basisstation
%
% Funktion:         OK
% Datei:            comport_open.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ serial_obj ] = comport_open( serial_port )

    % Standard Serial COM Port Settings for Basestation:
    %   Port                = COM1
    %   Baudrate            = 115200
    %   Databits            = 8
    %   Stopbits            = 1
    %   Parity              = None
    %   CTS flow control    = Off
    %   InputBufferSize     = 1024 * 1000
    %   RX-Side:
    %       ASCII Dataformat,
    %       Newline @ LF,
    %   TX-Side:
    %       ASCII Dataformat,
    %       Newline @ LF

%   serial_port = 'COM1';
    serial_obj = serial(serial_port, 'BaudRate', 115200, 'DataBits', 8, ...
                        'Parity', 'none', 'StopBits', 1, ...
                        'InputBufferSize', 2^10*1000, 'Timeout', 30);
    try
        fopen(serial_obj);
    catch E1
        fclose(serial_obj);
        delete(serial_obj);
        error('Error: fopen()');
    end

end
%-----------------------------------------------------------------------------------------
```

Programmausdruck C.63: *Datenlogger Matlab-Funktion „comport_open.m“*

```
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      30.05.2011
% Geändert am:      30.05.2011
% Beschreibung:     Funktion zur veränderten Darstellung der Cursor-Daten Anzeige
%                   im Plot (Konvertierung datenum -> datestr).
%
% Funktion:         OK
% Datei:            cursor_display_datestr.m
%++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function output_txt = cursor_display_datestr(~,event_obj)
% Display the position of the data cursor
% obj          Currently not used (empty)
% event_obj    Handle to event object
% output_txt   Data cursor text string (string or cell array of strings).

pos = get(event_obj,'Position');
output_txt = {['t: ',num2str(datestr(pos(1), 'dd-mmm-yyyy HH:MM:SS.FFF'),4)],...
    ['y: ',num2str(pos(2),'%1.6f')]};

% If there is a Z-coordinate in the position, display it as well
if length(pos) > 2
    output_txt{end+1} = ['Z: ',num2str(pos(3),4)];
end
%----------------------------------------------------------------------------------------
```

Programmausdruck C.64: *Datenlogger Matlab-Funktion „cursor_display_datestr.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      27.05.2011
% Geändert am:      11.07.2011
% Beschreibung:     Funktion zum Dekodieren der übermittelten Daten
%
% Funktion:         OK
% Datei:            decode_datastring.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ rxdata, dataok ] = decode_datastring( string, rxdata )

    dataok = false;
    if(strfind(string, 'SEN'))              % Sensor Data received: -----------------
        [rxvec, count] = sscanf(string,...              % decode string for dataset 1
                        'SEN%02xTS%08xA%04xV%02xC%04xS%04xT%04x\n');
        if(count == 7)                                  % Dataset 1 received:
            rxdata.cv        = rxvec(5) / 10000.0;      % convert to V
            rxdata.sv        = rxvec(6) / 1000.0;       % convert to V
            rxdata.av        = -1;                      % value not received
            rxdata.lv        = -1;                      % value not received
            rxdata.temperature = rxvec(7) / 100.0;      % convert to °C
            dataok = true;                              % set flag
        else
            [rxvec, count] = sscanf(string,...      % decode string for dataset 2
                            'SEN%02xTS%08xA%04xV%02xC%04xW%04xL%04x\n');
            if(count == 7)                          % Dataset 2 received:
                rxdata.cv        = rxvec(5) / 10000.0;  % convert to V
                rxdata.sv        = -1;                  % value not received
                rxdata.av        = rxvec(6) / 1000.0;   % convert to V
                rxdata.lv        = rxvec(7) / 1000.0;   % convert to V
                rxdata.temperature = -1;                % value not received
                dataok = true;                          % set flag
            end
        end
        if(dataok == true)
            rxdata.sensor    = rxvec(1)+2;
            rxdata.address   = rxvec(3);
            rxdata.vid       = rxvec(4);
            rxdata.timestamp = decode_timestamp(rxdata.timestamp, rxvec(2));
            rxdata.current_ch  = 0;
            rxdata.current_dir = 0;
            rxdata.current     = 0;
        end
    elseif(strfind(string, 'CUR'))              % Current Data received: ----------------
        [rxvec, count] = sscanf(string,...      % decode string for current data
                        'CURTS%08xV%04x\n');
        if(count == 2)
            rxdata.sensor      = 1;             % current sensor
            rxdata.address     = 0;
            rxdata.vid         = 0;
            rxdata.timestamp   = decode_timestamp(rxdata.timestamp, rxvec(1));
                                                % bit 15: current channel 0/1
            rxdata.current_ch  = bitand(bitsrl(uint16(rxvec(2)), 14), hex2dec('01'));
                                                % bit 16: current direction 0=pos / 1=neg
            rxdata.current_dir = bitand(bitsrl(uint16(rxvec(2)), 15), hex2dec('01'));
                                                % bit 1-14: current value
            rxdata.current(:)  = bitand(uint16(rxvec(2)), hex2dec('3FFF'));
            if(rxdata.current_ch == 0)
                rxdata.current = rxdata.current * 2*2.5*1/4096 * 1/(26.7*10^-3);
            else
```

```
63                  rxdata.current = rxdata.current * 2*2.5*1/4096 * 1/(4*10^-3);
64              end
65              if(rxdata.current_dir == 1)         % set direcion
66                  rxdata.current = rxdata.current * -1.0;
67              end
68              rxdata.cv = 0;
69              rxdata.sv = 0;
70              rxdata.av = 0;
71              rxdata.lv = 0;
72              rxdata.temperature = 0;
73              dataok = true;                      % set flag
74          end
75      end
76  end
77  %--------------------------------------------------------------------------------------
```

Programmausdruck C.65: *Datenlogger Matlab-Funktion „decode_datastring.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      27.05.2011
% Geändert am:      11.07.2011
% Beschreibung:     Funktion zum Dekodieren des übermitteltem Zeitstempels
%
% Funktion:         OK
% Datei:            decode_timestamp.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [ timestamp ] = decode_timestamp( timestamp, rx_timestamp )

    rx_timestamp_u32  = uint32(rx_timestamp);
                                            % 5-bit for day
    timestamp.day(:)  = bitand( bitsrl(rx_timestamp_u32, 27),...
                                uint32(hex2dec('0000001F')) );
                                            % 5-bit for hour
    timestamp.hour(:) = bitand( bitsrl(rx_timestamp_u32, 22),...
                                uint32(hex2dec('0000001F')) );
                                            % 6-bit for minute
    timestamp.min(:)  = bitand( bitsrl(rx_timestamp_u32, 16),...
                                uint32(hex2dec('0000003F')) );
                                            % 6-bit for second
    timestamp.sec(:)  = bitand( bitsrl(rx_timestamp_u32, 10),...
                                uint32(hex2dec('0000003F')) );
                                            % 10-bit for milli
    timestamp.msec(:) = bitand( rx_timestamp_u32, uint32(hex2dec('000003FF')));
                                            % second inclusive milli for datestr
    timestamp.sec(:)  = timestamp.sec + timestamp.msec/1000;

    timenow = clock;                        % get actual time
    timestamp.year  = timenow(1);
    timestamp.month = timenow(2);

    timestamp.str = datestr([ timestamp.year, timestamp.month, timestamp.day, ...
                              timestamp.hour, timestamp.min, timestamp.sec ], ...
                            'dd-mmm-yyyy HH:MM:SS.FFF');

    timestamp.num = datenum(timestamp.str);

end
%-----------------------------------------------------------------------------------------
```

Programmausdruck C.66: *Datenlogger Matlab-Funktion „decode_timestamp.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      20.10.2011
% Geändert am:      21.10.2011
% Beschreibung:     Funktion zum Erstellen einer GUI für die User-Eingaben
%
% Funktion:         OK
% Datei:            make_gui.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [user_input, stop] = make_gui( ~ )

    user_input.num_sample       = 10;
    user_input.num_sensors_rec  = 6;
    user_input.num_sensors_scan = 6;
    user_input.scan_sensors     = false;
    user_input.dataset          = 1;
    user_input.send_time_sen    = false;
    user_input.send_dataset_sen = false;
    user_input.wakeup_cw        = false;
    user_input.samples_graph    = 10000;
    user_input.tara             = false;
    stop = false;
                                            % create figure for gui elements
    hf = figure('Resize', 'off', 'Visible', 'off', 'Position', [360,500,615,400], ...
                'Color', 'w', 'Name', 'Datalogger User Input');

    hp(1) = uipanel('Parent',hf,'Title','General','FontSize',10,...
                    'Units', 'pixels', 'Position',[5 250 305 150]);
    hp(2) = uipanel('Parent',hf,'Title','Scan for Sensors','FontSize',10,...
                    'Units', 'pixels', 'Position',[5 140 305 100]);
    hp(3) = uipanel('Parent',hf,'Title','Current Measure','FontSize',10,...
                    'Units', 'pixels', 'Position',[5 75 305 55]);
    hp(4) = uipanel('Parent',hf,'Title','Cell Sensor Dataset','FontSize',10,...
                    'Units', 'pixels', 'Position',[315 250 305 150]);
    hp(5) = uipanel('Parent',hf,'Title','Cell Sensor Time','FontSize',10,...
                    'Units', 'pixels', 'Position',[315 185 305 55]);

                                            % create text and buttons
    htext(1)  = uicontrol('Style','text','String','Number of Samples to Record',...
                          'Parent',hp(1),'Position',[10,95,110,30]);
    hedit(1)  = uicontrol('Style','edit','String', '10', 'Min', 0,...
                          'Parent',hp(1),'Position',[130,95,110,30],...
                          'Callback',{@edit_numsample_Callback});
    htext(2)  = uicontrol('Style','text','String','How many Samples to show in Graph',...
                          'Parent',hp(1),'Position',[10,50,110,30]);
    hedit(2)  = uicontrol('Style','edit','String', '10000', 'Min', 0,...
                          'Parent',hp(1),'Position',[130,50,110,30],...
                          'Callback',{@edit_samplegraph_Callback});
    htext(3)  = uicontrol('Style','text','String','Number of Sensors to Record',...
                          'Parent',hp(1),'Position',[10,5,110,30]);
    hpopup(1) = uicontrol('Style','popupmenu','String',{'6','12','18'},...
                          'Parent',hp(1),'Position',[130,5,110,30],...
                          'Callback',{@popup_menu_numrec_Callback});

    hcheck(1) = uicontrol('Style','checkbox','String','Scan for Sensors',...
                          'Parent',hp(2),'Position',[10,50,110,30],...
                          'Callback',{@scanbutton_Callback});
    htext(4)  = uicontrol('Style','text','String','Number of Sensors to Scan',...
                          'Parent',hp(2),'Position',[10,5,110,30]);
    hpopup(2) = uicontrol('Style','popupmenu','String',{'6','12','18'},...
```

```
63                          'Parent',hp(2),'Position',[130,5,110,30],...
64                          'Enable', 'off', ...
65                          'Callback',{@popup_menu_numscan_Callback});

67      hcheck(2) = uicontrol('Style','checkbox','String','Current Tara',...
68                          'Parent',hp(3),'Position',[10,5,110,30],...
69                          'Callback',{@tarabutton_Callback});

71      htext(5)  = uicontrol('Style','text','String','Select a Dataset to Record',...
72                          'Parent',hp(4),'Position',[10,95,110,30]);
73      hpopup(3) = uicontrol('Style','popupmenu',...
74                          'String',{'Ucell, Usupply, Temperatur','Ucell, Uact_in, Ulp'
                                },...
75                          'Parent',hp(4),'Position',[120,95,180,30],...
76                          'Callback',{@popup_menu_dataset_Callback});
77      hcheck(3) = uicontrol('Style','checkbox','String','Send selected Dataset to Sensors'
            ,...
78                          'Parent',hp(4),'Position',[10,50,200,30],...
79                          'Callback',{@datasetbutton_Callback});
80      hcheck(4) = uicontrol('Style','checkbox','String','Activate CW for downlink',...
81                          'Parent',hp(4),'Position',[10,5,150,30],...
82                          'Callback',{@cwbutton_Callback});

84      hcheck(5) = uicontrol('Style','checkbox','String','Send Time Settings to Sensors',...
85                          'Parent',hp(5),'Position',[10,5,180,30],...
86                          'Callback',{@timesetbutton_Callback});

88      hpush(1)  = uicontrol('Style', 'pushbutton', 'String', 'Close',...
89                          'Position', [615/2-70 20 60 30],...
90                          'Callback', {@CloseButton_Callback});
91      hpush(2)  = uicontrol('Style', 'pushbutton', 'String', 'Start',...
92                          'Position', [615/2+10 20 60 30],...
93                          'Callback', {@StartButton_Callback});

95      set([hf,hcheck,hpush,htext,hpopup,hedit], 'Units','normalized');
96      movegui(hf,'center');                   % move gui to center of screen
97      set(hf,'Visible','on');                 % make gui visible

99      uiwait;             % wait with end of function until close or start button pressed

101     function popup_menu_numrec_Callback(source,~)
102         str = get(source, 'String');
103         val = get(source, 'Value');
104         user_input.num_sensors_rec = str2double(str(val));
105     end
106     function popup_menu_numscan_Callback(source,~)
107         str = get(source, 'String');
108         val = get(source, 'Value');
109         user_input.num_sensors_scan = str2double(str(val));
110     end
111     function popup_menu_dataset_Callback(source,~)
112         val = get(source, 'Value');
113         user_input.dataset = val;
114     end
115     function scanbutton_Callback(source,~)
116         if(get(source, 'Value') == 1)
117             user_input.scan_sensors = true;
118             set(hcheck(5), 'Value', 1, 'Enable', 'off');
119             user_input.send_time_sen = true;
120             set(hpopup(2), 'Enable', 'on');
121         else
122             user_input.scan_sensors = false;
123             set(hcheck(5), 'Value', 0, 'Enable', 'on');
```

```
124             user_input.send_time_sen = false;
125             set(hpopup(2), 'Enable', 'off');
126         end
127     end
128     function cwbutton_Callback(source,~)
129         if(get(source, 'Value') == 1)
130             user_input.wakeup_cw = true;
131         else
132             user_input.wakeup_cw = false;
133         end
134     end
135     function datasetbutton_Callback(source,~)
136         if(get(source, 'Value') == 1)
137             user_input.send_dataset_sen = true;
138         else
139             user_input.send_dataset_sen = false;
140         end
141     end
142     function timesetbutton_Callback(source,~)
143         if(get(source, 'Value') == 1)
144             user_input.send_time_sen = true;
145         else
146             user_input.send_time_sen = false;
147         end
148     end
149     function tarabutton_Callback(source,~)
150         if(get(source, 'Value') == 1)
151             user_input.tara = true;
152         else
153             user_input.tara = false;
154         end
155     end
156     function edit_numsample_Callback(source,~)
157         str = get(source, 'String');
158         user_input.num_sample = str2double(str);
159     end
160     function edit_samplegraph_Callback(source,~)
161         str = get(source, 'String');
162         user_input.samples_graph = str2double(str);
163     end
164     function CloseButton_Callback(~,~)
165         uiresume;
166         stop = true;
167         close(hf);
168     end
169     function StartButton_Callback(~,~)
170         uiresume;
171         close(hf);
172     end

174 end
175 %--------------------------------------------------------------------------------------
```

Programmausdruck C.67: *Datenlogger Matlab-Funktion „make_gui.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      06.10.2011
% Geändert am:      06.10.2011
% Beschreibung:     Funktion zum Erstellen der Figures für die Messwertanzeigen
%
% Funktion:         OK
% Datei:            prepare_graphs.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [hf, ha, hp, hl, xaxis_start, sensor_display] = prepare_graphs(sensordata, ...
                                                                        NUM_SENSORS,...
                                                                        MARKER)

hp                  = zeros(6, NUM_SENSORS+1);              % handler for plots
ha                  = zeros(6, 1);                          % handler for axis
hf                  = zeros(2, 1);                          % handler for figures
xaxis_start         = 1;
legendtext          = cell(NUM_SENSORS, 1);
sensor_display      = zeros(NUM_SENSORS+1, 1);
scrsz               = get(0,'ScreenSize');                  % get screensize
for i=1 :1: NUM_SENSORS                                     % set legendtext
    legendtext(i,:)    = {['ZS_{' num2str(i) '}']};
    sensor_display(i) = true;
end
    % ___ Figure 1 _____________________________________________________
hf(1) = figure('Name', 'Measured Data 1',...
               'OuterPosition', [1 scrsz(4)/3 scrsz(3)/2 scrsz(4)*2/3]);
linecolor = colormap(lines(NUM_SENSORS+1));                 % get colors for lines
ha(1) = subplot(4,1,1);
    hold on;
    for i=2 :1: NUM_SENSORS+1
        hp(1,i) = plot(0:0, sensordata.cv(i,1:1), MARKER, 'MarkerSize', 4);
        set(hp(1,i),'YDataSource', ['sensordata.cv('        num2str(i) ', xaxis_start:
            sensordata.sample(' num2str(i) '))']);
        set(hp(1,i),'XDataSource', ['[sensordata.timestamp(' num2str(i) ', xaxis_start:
            sensordata.sample(' num2str(i) ')).num]']);
        set(hp(1,i),'Color', linecolor(i,:),'LineWidth', 1);
    end
    hold off;
    grid on;
    grid minor;
    ylim([0 5]);
    datetick('x','MM:SS.FFF');
    xlabel('Time in mm:ss.fff', 'FontSize', 10);
    ylabel('Voltage in V', 'FontSize', 10);
    title('Cell Voltage', 'FontSize', 12);
ha(6) = subplot(4,1,2);
    hp(6,1) = plot(0:0, sensordata.current(1,1:1), '-o', 'MarkerSize', 4);
    set(hp(6,1),'YDataSource', 'sensordata.current(1,   xaxis_start:sensordata.sample(1))'
        );
    set(hp(6,1),'XDataSource', '[sensordata.timestamp(1, xaxis_start:sensordata.sample(1))
        .num]');
    set(hp(6,1),'Color', linecolor(1,:),'LineWidth', 1);
    grid on;
    grid minor;
    %ylim([0 5]);
    datetick('x','MM:SS.FFF');
    xlabel('Time in mm:ss.fff', 'FontSize', 10);
    ylabel('Current in A', 'FontSize', 10);
    title('Current', 'FontSize', 12);
```

```
59  ha(2) = subplot(4,1,3);
60      hold on;
61      for i=2 :1: NUM_SENSORS+1
62          hp(2,i) = plot(0:0, sensordata.sv(i,1:1), MARKER, 'MarkerSize', 4);
63          set(hp(2,i),'YDataSource', ['sensordata.sv('        num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) '))']);
64          set(hp(2,i),'XDataSource', ['[sensordata.timestamp(' num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) ')).num]']);
65          set(hp(2,i),'Color', linecolor(i,:),'LineWidth', 1);
66      end
67      hold off;
68      grid on;
69      grid minor;
70      ylim([2 3.5]);
71      datetick('x','MM:SS.FFF');
72      xlabel('Time in mm:ss.fff', 'FontSize', 10);
73      ylabel('Voltage in V', 'FontSize', 10);
74      title('Supply Voltage', 'FontSize', 12);
75  ha(3) = subplot(4,1,4);
76      hold on;
77      for i=2 :1: NUM_SENSORS+1
78          hp(3,i) = plot(0:0, sensordata.temperature(i,1:1), MARKER, 'MarkerSize', 4);
79          set(hp(3,i),'YDataSource', ['sensordata.temperature(' num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) '))']);
80          set(hp(3,i),'XDataSource', ['[sensordata.timestamp('   num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) ')).num]']);
81          set(hp(3,i),'Color', linecolor(i,:),'LineWidth', 1);
82      end
83      hold off;
84      grid on;
85      grid minor;
86      ylim([10 30]);
87      datetick('x','MM:SS.FFF');
88      xlabel('Time in mm:ss.fff', 'FontSize', 10);
89      ylabel('Temperature in °C', 'FontSize', 10);
90      title('Temperature', 'FontSize', 12);
91      % ___ Figure 2 _____________________________________________________
92  hf(2) = figure('Name', 'Measured Data 2',...
93                 'OuterPosition', [scrsz(3)/2 scrsz(4)/3 scrsz(3)/2 scrsz(4)*2/3]);
94  ha(4) = subplot(2,1,1);
95      hold on;
96      for i=2 :1: NUM_SENSORS+1
97          hp(4,i) = plot(0:0, sensordata.av(i,1:1), MARKER, 'MarkerSize', 4);
98          set(hp(4,i),'YDataSource', ['sensordata.av('        num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) '))']);
99          set(hp(4,i),'XDataSource', ['[sensordata.timestamp(' num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) ')).num]']);
100         set(hp(4,i),'Color', linecolor(i,:),'LineWidth', 1);
101     end
102     hold off;
103     grid on;
104     grid minor;
105     ylim([0 10]);
106     datetick('x','MM:SS.FFF');
107     xlabel('Time in mm:ss.fff', 'FontSize', 10);
108     ylabel('Voltage in V', 'FontSize', 10);
109     title('Wake-up Input Voltage', 'FontSize', 12);
110 ha(5) = subplot(2,1,2);
111     hold on;
112     for i=2 :1: NUM_SENSORS+1
113         hp(5,i) = plot(0:0, sensordata.lv(i,1:1), MARKER, 'MarkerSize', 4);
114         set(hp(5,i),'YDataSource', ['sensordata.lv('        num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) '))']);
```

```
115         set(hp(5,i),'XDataSource', ['[sensordata.timestamp(' num2str(i) ', xaxis_start:
                sensordata.sample(' num2str(i) ')).num]']);
116         set(hp(5,i),'Color', linecolor(i,:),'LineWidth', 1);
117     end
118     hold off;
119     grid on;
120     grid minor;
121     ylim([0 1]);
122     datetick('x','MM:SS.FFF');
123     xlabel('Time in mm:ss.fff', 'FontSize', 10);
124     ylabel('Voltage in V', 'FontSize', 10);
125     title('Downlink Lowpass Voltage', 'FontSize', 12);

127 hl(1) = legend(ha(2), legendtext,'Location', 'NorthEastOutside', 'FontSize', 9);
128 hl(2) = legend(ha(5), legendtext,'Location', 'NorthEastOutside', 'FontSize', 9);

130 dcm_obj(1) = datacursormode(hf(1));                          % set cursor data display:
131 dcm_obj(2) = datacursormode(hf(2));
132 set(dcm_obj(1),'UpdateFcn',@cursor_display_datestr);
133 set(dcm_obj(2),'UpdateFcn',@cursor_display_datestr);
134                                                             % settings for printing:
135 set(hf(1:2), 'PaperType', 'A4');
136 set(hf(1:2), 'PaperOrientation', 'landscape');
137 set(hf(1:2), 'PaperPositionMode', 'auto');


140 end
141 %-------------------------------------------------------------------------------------
```

Programmausdruck C.68: *Datenlogger Matlab-Funktion „prepare_graphs.m“*

```
1  %+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2  % Projekt:          Masterarbeit
3  % Erstellt von:     Niels Jegenhorst
4  % Erstellt am:      08.06.2011
5  % Geändert am:      06.10.2011
6  % Beschreibung:     Funktion zum Aktualisieren der Messwertanzeigen
7  %
8  % Funktion:         OK
9  % Datei:            refresh_graphs.m
10 %+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

12 function [ xaxis_start, last_max_sample] = refresh_graphs(hp, ha, hl, xaxis_start,...
13                                                           SAMPLES_IN_GRAPH, ...
14                                                           last_max_sample,...
15                                                           NUM_SAMPLES, last_sample,...
16                                                           sensordata,...
17                                                           hwaitbar)

19                                                  % get max sample count, without current:
20     [max_sample sensor] = max(sensordata.sample(2:end));
21                                                  % refresh graph only once per 5 periodes
22                                                  %   or when last sample recorded,
23                                                  %   and min 2 samples recorded:
24     if( ((max_sample > last_max_sample+5) || (last_sample == true)) && (max_sample > 1) )
25         last_max_sample = max_sample;                % save actual sample count
26         sensor = sensor + 1;                         % don't show current sensor

28         if(max_sample > SAMPLES_IN_GRAPH)            % running x-axis:
29             xaxis_start = max_sample - SAMPLES_IN_GRAPH;
30         end
```

```
32          if((max_sample == 7) || (last_sample == true))  % refresh text for legend
33              for test=2 :1: length(sensordata.sample)    % get number of activ sensors
34                  if(sensordata.sample(test) == 0.0)
35                      num_sensors = test-1;
36                      break;
37                  else
38                      num_sensors = test;
39                  end
40              end
41              legendtext = cell(num_sensors, 1);
42              for s=2 :1: num_sensors
43                  legendtext(s,1) = {['ZS_{' num2str(s-1) '} #0x' ...
44                                     num2str(num2str(sensordata.address(s)))]};
45              end
46              pos_ha(1,:) = get(ha(2), 'Position');       % get position of plot
47              pos_ha(2,:) = get(ha(5), 'Position');
48              set(hl(1), 'String', legendtext(2:end,:));  % refresh text in legend of plot
49              set(hl(2), 'String', legendtext(2:end,:));
50              set(ha(2), 'Position', pos_ha(1,:));        % reset old position of plot
51              set(ha(5), 'Position', pos_ha(2,:));
52          end

54          refreshdata(hp(1:end,:),'caller');          % refresh datasources:

56          xrange = [sensordata.timestamp(sensor, xaxis_start).num ...
57                    sensordata.timestamp(sensor, max_sample).num];
58          for i=1 :1: 6
59              xlim(ha(i), xrange);                            % set x-axis limits
60              datetick(ha(i), 'x','MM:SS.FFF', 'keeplimits'); % set x-axis datetick:
61          end
62          drawnow;                                    % redraw graphs
63      end
64      if(last_sample == false)                        % update the waitbar:
65          waitbar(max_sample / NUM_SAMPLES, hwaitbar, ...
66                  [num2str(max_sample) ' / ' num2str(NUM_SAMPLES)]);
67      end
68  end
69  %--------------------------------------------------------------------------------------------
```

Programmausdruck C.69: *Datenlogger Matlab-Funktion „refresh_graphs.m“*

```
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
% Projekt:          Masterarbeit
% Erstellt von:     Niels Jegenhorst
% Erstellt am:      27.05.2011
% Geändert am:      11.07.2011
% Beschreibung:     Funktion zum Abspeichern der empfangenden Daten in eine Struktur
%
% Funktion:         OK
% Datei:            store_rxdata.m
%+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

function [sensordata] = store_rxdata(sensordata, rxdata)

    sample_pos = sensordata.sample(rxdata.sensor) + 1;  % actual sample position

    if(rxdata.sensor > 1)                               % Cell-Sensor:
        sensordata.sample(rxdata.sensor)                = sample_pos;
        sensordata.address(rxdata.sensor)               = rxdata.address;
        sensordata.vid(rxdata.sensor)                   = rxdata.vid;
        sensordata.cv(rxdata.sensor, sample_pos)        = rxdata.cv;
        sensordata.sv(rxdata.sensor, sample_pos)        = rxdata.sv;
        sensordata.av(rxdata.sensor, sample_pos)        = rxdata.av;
        sensordata.lv(rxdata.sensor, sample_pos)        = rxdata.lv;
        sensordata.temperature(rxdata.sensor, sample_pos) = rxdata.temperature;
        sensordata.timestamp(rxdata.sensor, sample_pos) = rxdata.timestamp;
    else                                                % Current-Sensor:
        sensordata.sample(rxdata.sensor)                = sample_pos;
        sensordata.current(sample_pos)                  = rxdata.current;
        sensordata.current_channel(sample_pos)          = rxdata.current_ch;
        sensordata.current_dir(sample_pos)              = rxdata.current_dir;
        sensordata.timestamp(rxdata.sensor, sample_pos) = rxdata.timestamp;
    end
end
%---------------------------------------------------------------------------------------
```

Programmausdruck C.70: *Datenlogger Matlab-Funktion „store_rxdata.m“*

Tabellenverzeichnis

Bildverzeichnis

Programmausdruck-Verzeichnis

Glossar

Back-End
An das Front-End angeschlossener Datenträger oder logische Schaltung, als Teils eines Transponders

Basisstation
Zentrale Gegenstelle (Reader-Datenlogger-Einheit)

Close coupling Systeme
Klassifizierung von RFID-Systemen mit Reichweiten $\approx 1\,\mathrm{cm}$

Downlink
Übertragungen von der Basisstation zum Zellensensor

FPGA
engl. Field Programmable Gate Array

Front-End
Schnittstelle zum Antennenkreis, als Teil eines Transponder

Jitter
Zeitliche Genauigkeit bzw. Abweichung, z. B. einer Messung

Long-range Systeme
Klassifizierung von RFID-Systemen mit Reichweiten $> 1\,\mathrm{cm}$

Reader
Schreib- / Leseeinheit, Bestandteil eines RFID-Systems

Remote-coupling Systeme
Klassifizierung von RFID-Systemen mit Reichweiten $< 1\,\mathrm{m}$

Transponder
kontaktloser Datenträger, Bestandteil eines RFID-Systems

Uplink
Übertragungen vom Zellensensor zur Basisstation

Wake-Up
Bezeichnet das Aufwecken bzw. Aktivieren einer Einheit

Abkürzungsverzeichnis

I^2C	Inter-Integrated Circuit
ADC	Analog Digital Konverter
ASK	Amplitudenumtastung
BMCL	Battery Monitoring and Control Language
DAC	Digital Analog Konverter
DC-DC-Wandler	Gleichspannungswandler
DL	*Downlink*
EEPROM	Electrically Erasable Programmable Read-Only Memory
FDX	Vollduplex
FER	Frame-Fehlerrate
FLL	Frequency Locked Loop
FSK	Frequenzumtastung
HDX	Halbduplex
HF	Kurzwellen (engl. High Frequency)
IDT	Integrated Device Technology
IRQ	Interrupt Request
ISM	Industrial Scientific and Medical
ISR	Interrupt Service Routine
LED	Lichtemittierende Diode
LF	Langwellen (engl. Low Frequency)

MCU	Mikrocontroller
OOK	On-Off Keying
RF	Radio Frequenz
RFID	Radio-Frequency Identification
SEQ	sequentielle
SI	Silicon Laboratories
SIF	Spezial Informationen Flags
SMBus	System Management Bus
SNR	Störabstand
SOF	Start of Frame
SPI	Serial Peripheral Interface
SRD	Short Range Devices
ST	STMicroelectronics
SVS	Supply Voltage Supervisor
TI	Texas Instruments
UHF	Mikrowellen (engl. Ultra High Frequency)
UL	*Uplink*
USV	Unterbrechungsfreien Stromversorgungen
VID	Versions Identifikation
WDT	Watchdog-Timer

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 27. Oktober 2011
Ort, Datum

Unterschrift