Diplomarbeit

Tobias Krannich

Experimentalsystem für einen Sensor-Controller mit drahtloser Energie- und Datenübertragung

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Tobias Krannich

Experimentalsystem für einen Sensor-Controller mit drahtloser Energie- und Datenübertragung

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

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

Abgegeben am 21. Juli 2008

Tobias Krannich

Thema der Diplomarbeit
Experimentalsystem für einen Sensor-Controller mit drahtloser Energie- und Datenübertragung

Stichworte
Drahtlos, Funkübertragung, Energieübertragung, Sensorik, Mikrocontroller

Kurzzusammenfassung
Zur Zeit befindet sich die Entwicklung von Sensorik mit drahtloser Energie- und Datenübertragung in einem frühen Entwicklungsstadium. Die Protokolle, die sich in der RFID-Technik bewährt haben, müssen für diese Systeme zunächst untersucht werden. Zusätzlich ergeben sich durch den höheren Energiebedarf weitere Anforderungen an die analoge Energieerzeugung im Transponder.
Damit die neu auftretenden Aspekte untersucht werden können, wird ein funktionsfähiges System benötigt, in dem die Komponenten erweitert und getestet werden können.
Im Rahmen dieser Diplomarbeit wird ein System für drahtlose Sensorik, bestehend aus einem passiven Transponder, einem Reader und einer Steuersoftware für den PC entwickelt.

Tobias Krannich

Title of the paper
Experimental system for a sensor-controller with wireless energy and data transmission

Keywords
Wireless, radio transmission, electrical transmission, sensors, microcontroller

Abstract
Presently the development of sensor system with wireless energy and data communication is in an early development stage. Protocols, which worked satisfactorily in the RFID technology, have to be examined for these systems. In addition further requirements arise to the analog energy production in the transponder as a result of the higher power requirement. To analyze these aspects a functional system is necessary, in which the components can be tested.
The thesis concerns itself with the development of a passive transponder, a reader and an application for the PC to control the system.

Danksagungen

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Entstehen dieser Diplomarbeit beigetragen haben.

Für die Betreuung durch die Hochschule für Angewandte Wissenschaften bedanke ich mich bei meinem Erstprüfer Herrn Prof. Dr.-Ing. Karl-Ragmar Riemschneider für die fachliche und persönliche Unterstützung.

Zudem gilt mein Dank Herrn Prof. Dr.-Ing. Jürgen Vollmer, der mir während der Diplomarbeit mit fachlichem Rat zur Seite stand und die Zweitprüfung übernommen hat.

Ein besonderer Dank geht an Herrn Gerhard Wolff und Herrn Jörg Pflüger, die mir bei der Planung und der Erstellung der Hardware mit fachlichem Rat zur Seite standen.

Außerdem gilt mein Dank allen Studienkollegen, die ihre Diplomarbeit parallel zu meiner in der Fachhochschule angefertigt haben und mir hilfreich bei Fragen zur Seite standen. Besonderer Dank gilt Stephan Plaschke und Thorsten Eger, mit denen ich viele technische Details diskutieren konnte.

Herzlich danken möchte ich auch meiner Freundin und meiner Familie für ihre Unterstützung.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Die RF-ID-Technologie hat bereits eine Reihe von Anwendungen. Es werden gegenwärtig wachsende Märkte gesehen. Dafür sind die Übertragungsverfahren und Protokolle standardisiert, die jeweils spezifisch in den Transponderschaltkreisen realisiert sind. Hingegen ist die drahtlose Sensorik noch in frühen Entwicklungsstadien. Drahtlose Sensoren nutzen viele mit der RF-ID verwandte Verfahren. Bisher sind Übertragungsprotokolle für drahtlose Sensoren kaum etabliert, weil nur wenige Produkte kommerziell verfügbar sind.
Daher ist es wünschenswert, über ein offenes, programmierbares Sensor-Transponder-System zu verfügen, um neue und vorhandene Protokolle implementieren zu können. Mit diesem System wird es außerdem möglich, die Sensorfunktion anwendungsspezifisch gestalten zu können. Im Gegensatz zur denkbaren Neuentwicklung oder der Erweiterung von RF-ID-Transponder-Chips erscheint ein derartiges Konzept viel einfacher umsetzbar. Für Kleinserien-Anwendungen könnte dieser Ansatz wirtschaftlich sein. Durch Softwarevarianten wären auch bestehende RF-ID-Protokolle kompatibel realisierbar. In der Diplomarbeit soll zunächst ein experimentelles Grundsystem dieser Art entstehen. Es soll die Möglichkeiten von passiven Transpondern in Kombination mit einer Sensorfunktionen aufzeigen.
Anforderungen aus einer exemplarischen Anwendung sollen für die Systemauslegung betrachtet werden, z.B. für die drahtlose Überwachung von Batterien oder für die Temperaturbestimmung der Sterilisation. Die Arbeiten sollen umfassen:

1. **Konzeption**, Recherche und Vorstudien zur Komponentenauswahl und zu Entwurfsentscheidungen, ggfs. Berechnungen und Simulationen

2. **Sensortransponder**, bestehend aus passivem Frontend für den 125kHz-Bereich (Energieversorgung, analoge Sende- und Empfangsschaltung) sowie einem Backend aus einem Controller aus der MSP430 Familie und steuernder Software (Kommunikationsprotokoll, Identifikations- und Sensorfunktion u.a.).

3. **Reader**, zur Erzeugung des elektromagnetischen Feldes für die Transponderversorgung sowie das Senden von Daten an den Transponder und Empfangen von Transponderdaten. Im Reader sollen auf einem steuernden Mikrocontroller die Codierung, Decodierung und Verifizierung von Sende- und Empfangsdaten sowie die Abfrage und Zuordnung einzelner Sensortransponder realisiert werden.

4. **Funktionsdemonstration**, bestehend aus der Auswahl und Beschreibung einer Beispielapplikation, der Erfassung einiger Messreihen und deren geeignete Auswer-

tung, sowie der Übergabe an einen PC zur Datenverarbeitung oder Visualisierung

5. **Bewertung und Diskussion** des verfolgten Lösungsansatzes „frei programmierbarer Sensor-Controller“ und der gefundenen Detaillösungen

1.2 Radio Frequency Identification

RFID steht für Radio Frequency Identification und bedeutet im deutschen Identifizierung mit Hilfe von elektromagnetischen Wellen. Erstmalig wurde die Technologie zum Ende des zweiten Weltkrieges für die Freund-Feinderkennung eingesetzt. Die prinzipielle Funktionsweise von RFID kann folgendermaßen beschrieben werden:
Der Reader erzeugt ein hochfrequentes elektromagnetisches Wechselfeld, welches den Transponder mit Energie versorgt. Die Energieerzeugung im passiven Transponder übernimmt ein auf die Trägerfrequenz abgestimmter Schwingkreis. Mit Hilfe der am Schwingkreis auftretenden Spannungsüberhöhung wird eine kleine Kapazität geladen, die die Spannungsversorgung für den Digitalteil darstellt. Zusätzlich zu den passiven Transpondern gibt es auch semi-passive Transponder, bei denen die Versorgung der Digitalschaltung durch eine Batterie gestützt wird. Die Datenübertragung zum Reader erfolgt bei beiden Varianten über das vom Reader erzeugte elektromagnetische Feld. Bei aktiven Transpondern wird sowohl die Energie für die Versorgung der Digitalschaltung als auch für die Datenübertragung aus Batterien oder anderen Energiespeichern genutzt. Im Rahmen dieser Diplomarbeit wird, falls nicht anders erwähnt, grundsätzlich das passive System beschrieben.
Die Digitalschaltung im Transponder kann im einfachsten Fall ein Frequenzteiler sein, der mit einer 1/n-fachen Frequenz den eigenen Eingangsschwingkreis belastet. Diese Modulation wird im Reader erkannt. Da bei dieser Variante nur erkannt werden kann, ob der Transponder im Feld ist oder nicht, wird dieser Transponder als 1-Bit-Transponder bezeichnet. Diese Technologie wird bereits seit ca. 1970 im Bereich der Warensicherung eingesetzt. Eine komplexere Digitalschaltung im Transponder kann zusätzlich Automatenstrukturen ausführen und serielle Datenströme an den Reader senden. Bei der Digitalschaltung, die im Rahmen dieser Diplomarbeit eingesetzt wird, handelt es sich um einen frei programmierbaren Mikrocontroller. Zusätzlich können noch weitere Peripherieelemente wie Sensoren an den Mikrocontroller angeschlossen und von diesem ausgewertet werden.
Die Datenübertragung geschieht entweder unidirektional in Richtung vom Transponder zum Reader. Bei dieser Art der Datenübertragung sendet der Transponder unaufgefordert die Daten zum Reader. Bei mehreren Transpondern im Feld müssen daher spezielle Antikollisionsverfahren eingesetzt werden, damit sichergestellt werden kann, dass die gesendeten Daten von einem Transponder in einem gewissen Zeitraum vom Reader empfangen werden. Oder es kann Zusätzlich zu der unidirektionalen Datenübertragung

auch eine bidirektionale Datenübertragung implementiert werden, bei der die Datenübertragung sowohl in Richtung zum Transponder, als auch in Richtung Reader möglich ist. Bei dieser Übertragungsart ergibt sich bei mehreren Transpondern eine sternförmige Netzwerktopologie, bei der die Kommunikation mithilfe geeigneter Handshakeverfahren abläuft. Mit größerem schaltungstechnischen Aufwand im Transponder ist auch die Kommunikation unter den einzelnen Transponder möglich. Bei dieser Methode lässt sich jede beliebige Netzwerktopologie implementieren.

1.2.1 Anwendungsbereiche

Die Anwendung von RFID-Systemen kann nicht einer speziellen Branche zugeordnet werden, sondern ist in nahezu allen Bereichen vertreten. Grundsätzlich wird RFID angewendet, um Kosten zu sparen und die Effizienz von Produktion oder Warenfluss zu steigern. Zunehmend werden die Systeme auch zur Komfortsteigerung und zur Überwachung bei sicherheitsrelevanten Prozessen eingesetzt. Um den weiten Anwendungsbereich von RFID-Systemen aufzuzeigen, werden nachfolgend einige Bereiche aufgezählt:

- Gesundheitswesen

  Operationsbesteck wird mit Transpondern bestückt, so dass mit einem Reader eventuell vergessene Teile im Körper des Patienten bemerkt werden.

  Medikamentenausgabe kann genauestens kontrolliert und dokumentiert werden

  Sterilisationsprozesse können nachgewiesen werden

  Transponder am Patienten enthalten Information über den Krankheitsverlauf

- Identifikationssysteme

  Biometrische und andere Informationen werden in Reisepässen gespeichert

  Bei Zugangskontrollen als spezifischer Besitz

  Tieridentifikation

  Elektronische Wegfahrsperre

- Materialflusssysteme

- Erkennung von Produktfälschungen

- Mauterfassung

In den Jahren zwischen 1944 und 2005 wurden weltweit 2,397 Milliarden RFID Chips verkauft. Die Verteilung auf die verschiedenen Branchen zeigt die Tabelle 1 auf Seite 6.

1.2.2 Frequenzbereiche

Für die verschiedenen Einsatzbereiche von RFID-Lösungen wird nahezu das komplette Frequenzspektrum genutzt, beginnend im langwelligen Bereich um 125kHz bis in den Mikrowellenbereich um 5,8GHz. Trotz der relativ geringen Entfernungen von RFID-Systemen muss bei der Frequenzwahl für die Energieversorgung und die Kommunikation zwischen Reader und Transponder der Einfluss auf andere Funksysteme berücksichtigt werden, da die vom Reader ausgehenden elektromagnetischen Wellen gerade im langwelligen Bereich noch in großer Entfernung zu messen sind. Auf Grund der speziellen Energiekopplung bei RFID-Systemen müssen diese untereinander nicht von einander beeinflusst werden, jedoch können herkömmliche Funksysteme beeinträchtig oder ganz außer Funktion gesetzt werden. Infolgedessen darf bei der Frequenzauswahl nur auf bestimmte freigegebene Bereiche im Frequenzspektrum zurückgegriffen werden. In erster Linie werden für RFID-Systeme die ISM-Bänder genutzt, die für industrielle, wissenschaftliche und medizinische Anwendungen reserviert wurden. Die Tabelle 2 auf Seite 6 zeigt die Frequenzbereiche für die ISM-Bänder.

Langwellenbereiche bis 135kHz Der Langwellenbereich bis 135kHz ist nicht als ISM-Band ausgewiesen. Trotzdem nutzt ein großer Teil der RFID-Systeme den Bereich von 125kHz bis 135kHz als Arbeitsfrequenz, da dieser Frequenzbereich zusätzlich international nutzbar ist. RFID-Systeme im Langwellenbereich erreichen eine Reichweite bis ca. 1,5m. Die Kopplung zwischen Reader und Transponder ist in diesem Bereich induktiv. Ein großer Nachteil bei der niedrigen Arbeitsfrequenz ist, dass im Frequenzbereich bis 135kHz keine große Bandbreite für die Übertragung von Daten zur Verfügung steht. Dies ist insbesondere kritisch, wenn mehrere Transponder im Feld ausgelesen werden müssen. Auf Grund der geringen Übertragungsrate der Daten eignen sich Systeme im Langwellenbereich nur für Anwendungen, bei denen die Datenmenge aus dem Transponder relativ klein ist oder genügend Zeit für das Auslesen der Transponder zur Verfügung steht. Ein weiterer Nachteil ist die große Empfindlichkeit gegenüber elektromagnetischen Störfeldern, die zum Beispiel von starken Elektromotoren erzeugt werden. Dieser Nachteil macht sie für viele industrielle Anwendungen ungeeignet. Ein entscheidender Vorteil der Langwellen-Systeme ist die niedrige spezifische Apsorptionsrate (Dämpfung) für Wasser und nichtleitende Stoffe. Zusätzlich sind die Systeme im Langwellenbereich störungsunanfälliger auf metallische Umgebungen. Infolgedessen werden diese Systeme in Umgebungen mit großem Wasser- oder Metallanteil eingesetzt.

Branche	Ver. (Anz. in Mio.)
Transport/Automotiv	1000
Finanzen/Sicherheit	670
Handel/Finanzen	230
Freizeit	100
Wäschereien	75
Bibliotheken	70
Fertigung	50
Tiere/Landwirtschaft	45
Gesundheitswesen	40
Flugverkehr	25
Logistik/Post	10
Militär	2
Sonstige	80
Total	2397

Tabelle 1: Verbreitung von RFID nach Anwendung

Band	untere Frequenz	obere Frequenz
1	6,765MHz	6,795MHz
2	**13,553MHz**	**13,567MHz**
3	26,957MHz	27,283MHz
4	40,66MHz	40,70MHz
5	433,05MHz	434,79MHz
6	**868MHz**	**870MHz**
7	902MHz	928MHz
8	**2,4GHz**	**2,5MHz**
9	5,725GHz	5,875GHz
10	24GHz	24,25GHz
11	61GHz	61,5GHz
12	122GHz	123GHz
13	244GHz	246GHz

Tabelle 2: ISM-Bänder

Kurzwellenbereiche bei 13,56MHz RFID-Systeme, die den Kurzwellenbereich nutzen, arbeiten zumeist in dem Mid Range-Bereich. Die Reichweiten, die Systeme in diesem Bereich erreichen, liegen in der Regel zwischen 1m und 1,5m. Zusätzlich zu der erhöhten Reichweite zeichnen sich RFID-Systeme im Kurzwellenbereich durch die höhere Bandbreite und die damit verbundene höhere Übertragungsrate von Daten aus. Auf Grund der höheren Datenrate sind diese Systeme auch für Anwendungsbereiche geeignet, bei denen größere Datenmengen verarbeitet werden müssen. Insbesondere ist die höhere Datenrate wichtig, wenn eine große Anzahl von Transpondern im Feld ausgelesen werden soll.

Ultra-Kurzwellenbereich Die Frequenzen für den Ultra-Kurzwellenbereich liegen in Europa bei 868MHz und in den USA bei 915MHz. RFID-Systeme in diesem Bereich zeichnen sich durch noch höhere Übertragungsraten als bei den Systemen im Kurzwellenbereich aus. Jedoch gewinnt die Stellung von Reader und Transponder bei den höheren Frequenzen mehr an Bedeutung, so dass zusätzliche Anstrengungen in die Positionierung von Reader und Transponder gesteckt werden müssen, um ein sicheres Auslesen zu gewährleisten. Ein weiterer Nachteil, insbesondere für die Logistikbranche, sind die verschiedenen Arbeitsfrequenzen für Europa und die USA.

Mikrowellenbereich 2,45GHz und 5,8GHz Systeme im Mikrowellenbereich zeichnen sich durch die hohe Reichweite von ca. 10m bis 15m aus. Für die Frequenzen 2,45GHz und 5,8GHz stehen Frequenzbänder des ISM weltweit zur Verfügung. Auf Grund der hohen Reichweite gestaltet sich die Energieversorgung der Transponder schwierig und es werden überwiegend semi-passive Transponder eingesetzt, bei denen die Energieversorgung des Digitalteils von einer Batterie übernommen wird. Für die Kommunikation zwischen Reader und Transponder nutzen Mikrowellen-Systeme in diesem Frequenzbereich elektromagnetische Felder zur Kopplung. Durch die große spezifische Apsorptionsrate gestaltet sich das zuverlässige Auslesen der Transponder in feuchter Umgebung als äußerst schwierig. Als Beispiel für die große spezifische Absorptionsrate durch Wasser kann die Mikrowelle genannt werden, die genau diesen Effekt nutzt, um wasserhaltige Speisen und Getränke zu erwärmen.

Die wichtigsten Vor- und Nachteile für RFID-Systeme in den verschiedenen Frequenzbereichen sind in der Tabelle 3 auf Seite 8 zusammengefasst.

1.2.3 Bestandteile

In diesem Abschnitt werden die Grundelemente für ein vollständiges Systeme für die drahtlose Energie- und Datenübertragung beschrieben. Es wird gezeigt, welche Bau-

Langwellenbereich	
Vorteile	**Nachteile**
• Verwendung von günstigen passiven Transpondern • Gute Durchdringung von nicht-metallischen Gegenständen, Wasser und organischem Gewebe • Relativ unempfindlich gegen metallische Umgebungseinflüsse • Frequenzband weltweit verfügbar • Hohe erlaubte Sendeleistung	• Große Transponder-Bauformen (hohe Antennenwindungszahl) • Geringe Übertragungsgeschwindigkeit
Kurzwellenbereich	
Vorteile	**Nachteile**
• Verwendung von günstigen passiven Transpondern • Mittlere Datenübertragungsgeschwindigkeit (26 kBit/s) • Frequenzband weltweit verfügbar	• Hohe Dämpfung durch metallische Umgebung • Große Reichweiten erfordern große Antennenbauformen
Ultrakurzwellenbereich	
Vorteile	**Nachteile**
• Große Reichweite • Einfaches Antennendesign • Kostengünstig	• Schlechte Durchdringung von Wasser und organischem Gewebe
Mikrowellenbereich	
Vorteile	**Nachteile**
• Hohe Datenübertragungsgeschwindigkeiten • Hohe Reichweiten	• Große Bauform • Preis • Lebensdauer • Batterie

Tabelle 3: Vor- und Nachteile der Frequenzbänder

gruppen notwendig sind, und welche Aufgaben die einzelnen Elemente im System haben. Nach dem Lesen dieses Abschnittes soll der Leser grundsätzlich in der Lage sein, die einzelnen Komponenten und deren Zusammenspiel nachzuvollziehen. Außerdem sollen die Erkenntnisse als Verständnisgrundlage für das weitere Vorgehen in den folgenden Kapiteln dienen.
Ein vollständiges RFID System besteht aus einem Transpondern einer Readerstation und optional einem PC. Die Abbildung 1 auf Seite 9 zeigt ein mögliches System.

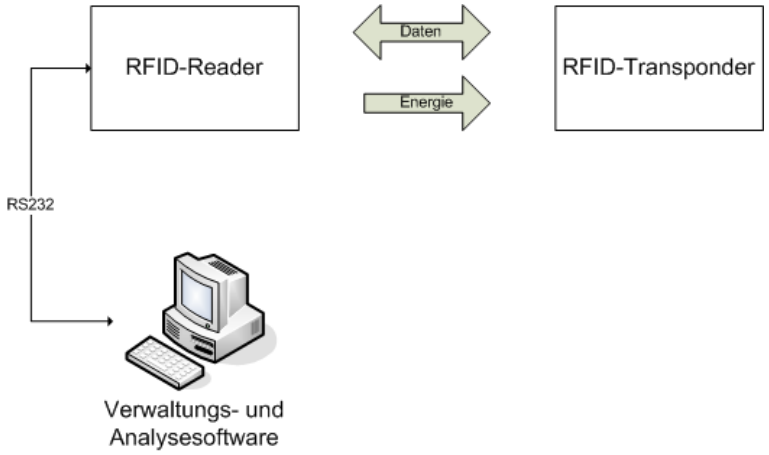

Bild 1: Beispielhaftes RFID-System

Bei der Planung und Entwicklung der Komponenten muss berücksichtigt werden in welchem Verhältnis die Anzahl der Reader und Transponder in der Anwendung zueinander stehen. Die Tabelle 4 zeigt vier Anwendungsfälle mit je einem möglichen Beispiel.

	1 Transponder	**mehrere Transponder**
1 Reader	Fall 1 Wegfahrsperre	Fall 2 Zeiterfassung
mehrere Reader	Fall 3 private Raumzugangskontrolle	Fall 4 Ticket-System bei öffentlichen Verkehrsmitteln

Tabelle 4: Anwendungsfälle von RFID-Systemen im Bezug auf die Reader- und Transponderanzahl in der Anwendung

Bei der Planung der Herstellungskosten für den Reader und den Transponder muss berücksichtigt werden für welchen Fall aus der Tabelle 4 die Komponenten eingesetzt werden. Für die Fälle, dass nur ein Transponder oder ein Reader verwendet werden und die andere Komponente in größerer Anzahl hergestellt wird (Fall 2, Fall 3), können die Herstellungskosten für die einzelne Komponente höher sein. Hingegen müssen die Herstellungskosten für die Komponente mit der großen Stückzahl gering gehalten werden. Für den Fall, dass es nicht möglich ist die Komponente mit der hohen Stückzahl kostengünstig zu produzieren muss die Komponente flexibel in der Anwendung sein, so dass sie

für unterschiedliche Anwendungen konfiguriert werden kann und eventuell auch wiederverwertbar ist. Für die Fälle 1 und 4 muss die Kostenplanung auf den Anwendungsfall und auf das Einsatzgebiet angepasst werden.

Transponder Der Begriff Transponder setzt sich aus Transmitter und Responder zusammen. Grundsätzlich wird unter dem Sammelbergriff Transponder ein System beschrieben, welches über eine Antenne und einen digitalen Speicher verfügt und sich berührungslos beschreiben und auslesen lässt. Die Abbildung 2 auf Seite 10 zeigt den schematischen Aufbau eines Transponders.

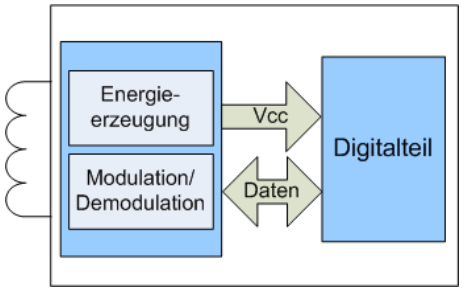

Bild 2: Schematischer Aufbau eines Transponders

Die Aufgabe des analogen Frontends ist es, die Versorgungsspannung für den Digitalteil aus dem elektromagnetischen Feld zu erzeugen und Baugruppen für die Kommunikation zwischen Transponder und Reader zur Verfügung zu stellen, die sich digital ansprechen, beziehungsweise auslesen lassen. Die Gruppe der Transponder lässt sich grob in die Gruppe der aktiven und die Gruppe der passiven Transponder einteilen. Die aktiven Transponder verfügen über eine Energiequelle, die sowohl für den digitalen Schaltungsteil als auch für das Senden der Daten genutzt wird. Passive Transponder beziehen ihre Energie aus dem elektromagnetischen Feld vom Reader. Auch die Kommunikation läuft über verschiedene Verfahren über das vom Reader erzeugte Feld ab. Eine Untergruppe der passiven Transponder stellen die semipassiven Transponder dar. Bei diesen wird die Energieversorgung des Digitalteils von einer Batterie gestützt. Dies hat den Vorteil, dass höhere Reichweiten erzielt werden und kontinuierliche Abläufe in der Digitalschaltung ablaufen können, unabhängig davon, ob der Transponder im Feld ist oder nicht. Durch die Verwendung einer zusätzlichen Batterie steigt der Preis gegenüber den passiven Transpondern und ist daher nur in Systemen zu finden, die eine hohe Lebensdauer besitzen oder wiederverwendet werden können. Zusätzlich zur Kategorisierung entsprechend der Energieversorgung können Transponder auch auf Grund ihrer Bauform unterschieden werden. Typische Bauformen sind in der folgenden Liste zusammengefasst.

- Disks und Münzen
- Schlüssel und Schlüsselanhänger
- Smart Label
- Glasgehäuse
- Uhren
- Smart-Ticket
- Plastikgehäuse
- Chipkarten
- Coil on Chip

Reader Der Reader verfügt über eine Empfangseinheit und kann zusätzlich trotz der Namensgebung auch ein Sendemodul enthalten. Viele Lesegeräte sind zusätzlich mit einer Schnittstelle zur Kommunikation mit weiteren Systemen ausgestattet, auf denen die eigentliche Applikation läuft. Häufig ist schon ein gewisser Teil der Applikation im Reader integriert, so dass zum Beispiel Daten aus Transpondern zwischengespeichert werden oder gewisse Abfrageroutinen in der Readersoftware integriert sind. Zusätzlich zur Kommunikation hat der Reader die Aufgabe, ein elektromagnetisches Feld für die Energieversorgung des Transponders zu erzeugen. Für die Erzeugung des elektromagnetischen Feldes wird ein RC-Schwingkreis mit seiner Resonanzfrequenz angeregt. Die Abbildung 3 zeigt den schematischen Aufbau eines Readers.

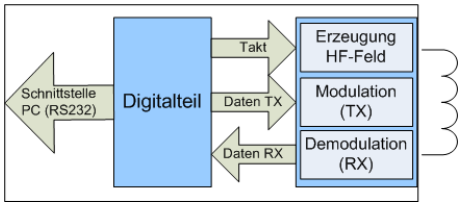

Bild 3: Schematischer Aufbau eines Readers

Applikation Die Verwaltungs- und Analysesoftware kann in Abhängigkeit der Anwendung sehr unterschiedlich ausgeführt sein. Sie muss nicht als PC ausgeführt sein, sondern kann je nach Anwendungsfall auch eine Automatensteuerung oder ein mobiles Gerät sein, in dem der Reader integriert ist. Da die Ausführungsmöglichkeiten der Applikation so vielfältig sind, werden an dieser Stelle nur einige Möglichkeiten für Applikationen aufgezeigt. Die im Rahmen dieser Diplomarbeit entwickelte Applikation wird in den späteren Abschnitten beschrieben.

- Im einfachsten Fall kann die Applikation aus einer Automatensteuerung bestehen. Als Beispiel hierfür könnte sich eine Zugangskontrolle zu einem Gelände vorgestellt werden, dass nur von Personen mit einer Zugangsberechtigung in Form eines Transponders betreten werden darf. Die Automatensteuerung hätte in diesem Fall nur die Aufgabe ein Tor oder eine Schranke zu öffnen, wenn ein Transponder detektiert wurde.

- Für mobile Geräte, wie sie zum Beispiel bei der Tieridentifikation eingesetzt werden oder bei Embedded Systemen für die Zeiterfassung, kann die Applikation aus einem leistungsstarken Mikrocontroller und weiterer Peripherie wie zum Beispiel

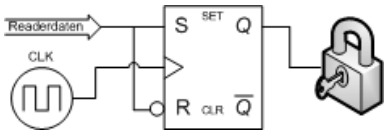

Bild 4: Automatensteuerung für Zutrittskontrolle

externem Speicher oder einem Display bestehen. Die Information aus den augelesenen Transpondern kann direkt auf dem Display angezeigt oder gespeichert werden und zu einem späteren Zeitpunkt über eine Schnittstelle zu einem anderen System übertragen werden. Für komplexere Anwendungen könnten auch embedded Betriebssysteme eingesetzt werden, um die Daten zu verwalten und über Schnittstellen mit höherem Protokollaufwand zur Verfügung zu stellen. So könnten die Daten zum Beispiel in einer SQL-Datenbank gespeichert und über einen Webserver angezeigt werden.

Bild 5: Embedded System von PointSync

- Für den leistungstärksten Fall wäre der Reader direkt mit einem PC verbunden. Ein solches System wäre für die Verwaltung und Überwachung des Materialflusses in einem industriellen Lagersystem geeignet.

1.3 Sensorik

Das Gebiet der Sensorik beschäftigt sich mit der Ermittlung von physikalischen Größen. Für die Erfassung wird ein Sensor benötigt, der die physikalische Größe in eine elektrisch messbare Größe umwandelt. Mit einer auf den Sensor angepassten Schaltung wird ein elektrisches Signal erzeugt, dass der zu messenden Größe entspricht. Die Schaltung für die Erzeugung des elektrischen Signals kann im einfachsten Fall aus einer Spannungsmessung bestehen. In den meisten Fällen wird aber eine umfangreiche Analogschaltung für die Aufbereitung des Signals benötigt. Für die Aufbereitung und die Verstärkung werden vielfach Operationsverstärker eingesetzt. Typische physikalische Größen, die mit Sensoren aufgenommen werden, werden in der folgenden Liste aufgezeigt.

- Kraft
- Position
- Geschwindigkeit
- Temperatur
- Drehzahl
- Beschleunigung
- Druck
- Drehmoment

Für die Erfassung der oben aufgezählten Größen werden verschiedene Verfahren angewandt, die unterschiedliche, elektrisch messbare Parameter verändern oder erzeugen. Nachfolgend werden die gängigsten Verfahren aufgezählt.

- resistive Sensoren
- reaktive Sensoren
- spannungsabgebende Sensoren
- stromsteuernde Sensoren
- ladungstrennende Sensoren
- oszillatorische Sensoren

Für das im Rahmen dieser Diplomarbeit entwickelte drahtlose Sensorsystem wird die Verwendbarkeit von allen aufgezählten Sensorverfahren angestrebt. Jedoch gilt bei allen für dieses System verwendeten Sensoren der Grundsatz des geringen Energiebedarfs. Die Umsetzung der einzelnen Messverfahren auf dem Transponder wird nicht im Fokus dieser Arbeit liegen, jedoch wird bei der Planung der Hardware die Verwendbarkeit der Sensortypen beachtet.

1.4 Energiebetrachtung für drahtlos versorgte Sensorik

Das Haupkreterium für die Auslegung des drahtlos versorgten Sensorsystems ist der Energiebedarf der einzelnen Komponenten im System. Für die Planung des Systems muss daher zunächst eine Gegenüberstellung der zur Verfügung stehenden Energie und der für die Komponenten benötigten Energie erfolgen.
Für das im Rahmen dieser Diplomarbeit geplanten System wird ein verfügbare Leistung von ca. $P_{ver} = 150\mu W$ für den Digital- und Sensorikteil angestrebt. Zur Veranschaulichung dieses Energieengpasses wird am Beispiel des bekannten Temperatursensors PT100 eine Gegenüberstellung der aufgenommenen Leistung vom PT100 bei $U = 2V$ und der zur Verfügung stehenden Leistung für das geplante System gezeigt. Die aufgenommene Leistung berechnet sich nach $P_{auf} = (2V)^2/100\Omega$ zu $P_{auf} = 40mW$. Die Abbildung 6 zeigt zur Verdeutlichung des Energieengpasses den grafischen Vergleich der zur Verfügung stehenden Leistung und der vom PT100 aufgenommenen Leistung.

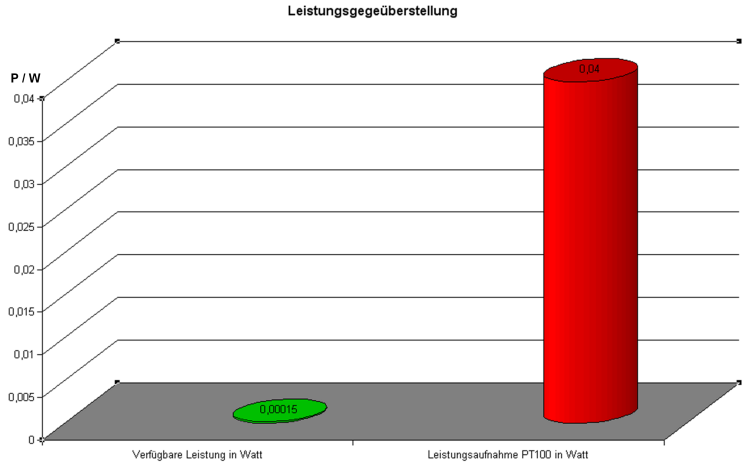

Bild 6: Gegenüberstellung der vom PT100 aufgenommenen und der zur Verfügung stehenden Leistung

Dieses einfache Beispiel zeigt die Problematik bei der Auslegung der verwendeten Sensorik. Die folgende Tabelle 5 gibt einen Überblick über die Energieaufnahme einiger elektronischer Baugruppen.

Funktion	**Energieaufnahme**	**Typ**	**Quelle**
RFID-Chip [Low Power]	$10\mu W$	T5557	www.atmel.com
Operationsverstärker	$34\mu W$	TLC52L4	www.focus.ti.com
Mikrocontroller [$\approx 80kHz$]	$40\mu W$	**MSP430F1232**	www.focus.ti.com
Temperatursensor	$94,5\mu W$	TMP121	www.focus.ti.com
Beschleunigungssensor	$540\mu W$	ADXL311	www.analog.com
RFID-Chip [256x4 Bit RAM]	$600\mu W$	U9280M-H	www.atmel.com

Tabelle 5: Energieaufnahme elektronischer Baugruppen

Bei der Auswahl der Komponenten für das drahtlose Sensorsystem muss darauf geachtet werden, dass die Summe der Leistung von den ausgewählten Baugruppen nicht den Betrag der zur Verfügung stehenden Leistung überschreitet. Die folgende Abbildung 7 zeigt die Leistungszusammensetzung von zwei möglichen Systemen.

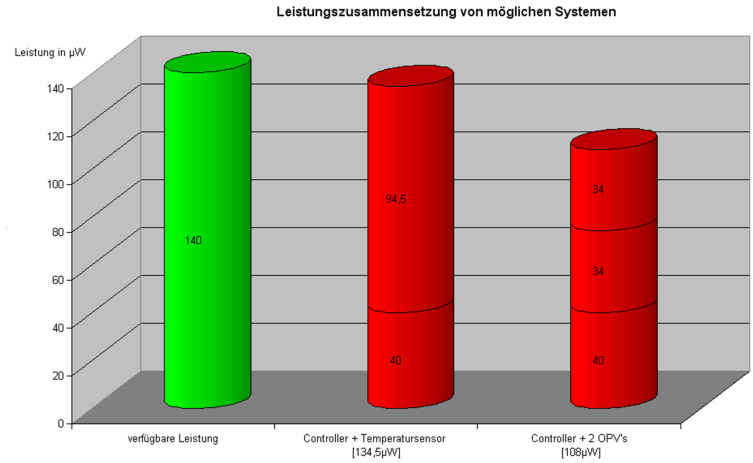

Bild 7: Gegenüberstellung der zur Verfügung stehenden Leistung und der Leistungsaufnahme möglicher Systeme

2 Planung der Komponenten

In diesem Abschnitt werden die Spezifikationen sowohl für das Gesamtsystem, wie auch für die einzelnen Baugruppen festgelegt. Auf Grund dieser Spezifikationen werden Voruntersuchungen in Form von Berechnungen und Simulationen durchgeführt, um mit diesen Erkenntnissen die Bauteilwahl und die Bauteildimensionierung für die spätere Implementierung abzuschätzen. Zudem werden die physikalischen und datentechnischen Hintergründe erläutert. Insbesondere werden die Punkte der Energieversorgung passiver Transponder, die Modulations- und Demodulationssverfahren und die Codierung für die serielle Datenübertragung ins Augenmerk genommen.

2.1 Anforderungen an das Gesamtsystem

Das im Rahmen dieser Diplomarbeit entwickelte drahtlose Sensorsystem soll in erster Linie als experimentelles Grundsystem für weitere Entwicklungen im Bereich der kontaktlosen Sensortechnik dienen. Es soll die Möglichkeiten von passiven Transpondern in Kombination mit Sensorik aufzeigen. Zudem wurden die Anforderungen auf eine mögliche Anwendung im Bereich der drahtlosen Temperaturermittlung in der Medizintechnik ausgelegt. In den folgenden Abschnitten sind die genauen Anforderungen an das drahtlose Sensorsystem stichpunktartig zusammengefasst:

Gesamtsystem

- Reichweite im Zentimeter-Bereich
- Leicht modifizier- und erweiterbar
- Frei in der Hochsprache C programmierbar
- Ermittlung der Umgebungstemperatur

Transponder

- passiv (drahtlose Energieversorgung)
- analoge Sende- und Empfangsschaltung
- Digitalteil in einer Hochsprache frei programmierbar
- Anschlussmöglichkeiten für analoge Sensoren

- Sicherstellen von Datenintegrität
- Codierung der Daten
- Einlesen und Verarbeiten von seriellen Daten

Reader

- Erzeugung eines elektromagnetischen Feldes
- Upload von Daten an den Transponder
- Empfangen von Transponderdaten
- Decodierung und Verifizierung empfangener Daten
- Zwischenspeicherung der empfangenen Transponderdaten
- Bereitstellung diverser Funktionsmodi
- Schnittstelle zum PC

Applikation

- Steuerbefehle an den Reader senden
- Aktuellen Status des Systems anzeigen
- Daten aus dem Reader auslesen und speichern
- Darstellung der Messwerte
- Grafische Messprotokolle erzeugen

Frequenzauswahl Bei der Auswahl eines Frequenzbereiches für das drahtlose Sensorsystem ist die Wahl auf den Langwellenbereich gefallen. Der Reader erzeugt ein elektromagnetisches Feld mit der Frequenz von 125kHz, über das die Kommunikation abläuft und das die Energieversorgung des Transponders darstellt. Die Gründe für diese Entscheidung sind nachfolgend zusammengefasst.

- Relativ unempfindlich gegen metallische Umgebungseinflüsse
- Gute Durchdringung von Wasser und organischem Gewebe

- Hohe erlaubte Sendeleistung
- Leichtere Handhabung als Kurz- und Mikrowellensysteme

Überblick Gesamtsystem Damit ein besseres Verständnis für die Planung der einzelnen Komponenten im weiteren Verlauf diese Kapitels gegeben ist, wird an dieser Stelle vorgegriffen und eine Übersicht über das geplante Gesamtsystem gegeben. Die Abbildung 8 zeigt den Aufbau des Gesamtsystems mit den einzelnen Komponenten.

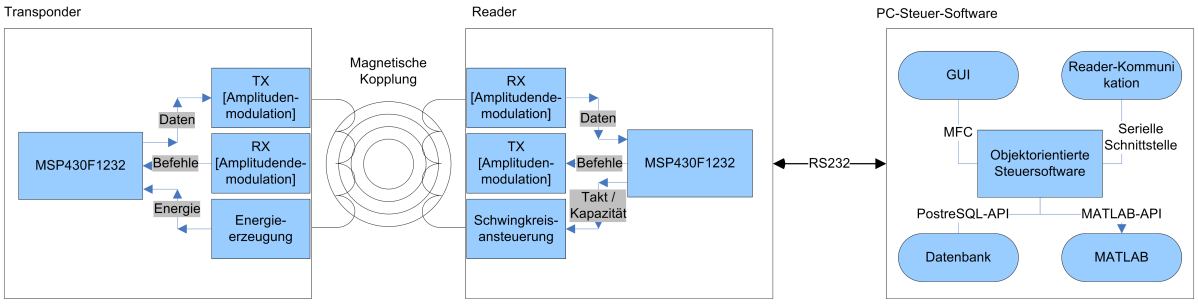

Bild 8: Überblick des Gesamtsystems

2.2 Transponder

Im Abschnitt Transponder wird die Funktionsweise der 4 wesentlichen Baugruppen des Transponders beschrieben, zu denen die Digitalschaltung, Energieversorgung, Modulationsschaltung und die Demodulationsschaltung gehören. Die Abbildung 9 zeigt das Blockschaltbild des Transponders. Zusätzlich werden anhand von Simulationen und Rechnungen die Einflüsse von Schaltungsparametern auf das Transpondersystem aufgezeigt. Die Rechnungen und Simulationen dienen bei der späteren Implementierung als Richtwerte für die Bauteildimensionierung. Der gesamte Abschnitt verzichtet bewusst auf schaltungstechnisch aufgenommene Messwerte oder Spektren, sondern nutz die vielfältigen Berechnungs- und Simulationsmöglichkeiten mit Analysetools, wie zum Beispiel MATLAB oder PSpice. Mit Hilfe der verschiedenen Simulationswerkzeuge soll die Planungsphase einen möglichst genauen Überblick über das System verschaffen, um so einerseits die Entscheidungsauswahl für die Implementierung einzugrenzen und andererseits die Entwicklungszeit zu reduzieren.

Eine Ausnahme bildet die Messung des Ersatzwiderstandes für die Digitalschaltung. Auf Grund der enormen Wichtigkeit dieses Parameters für das weitere Vorgehen in der Planung und Entwicklung wurde dieser messtechnisch bestätigt.

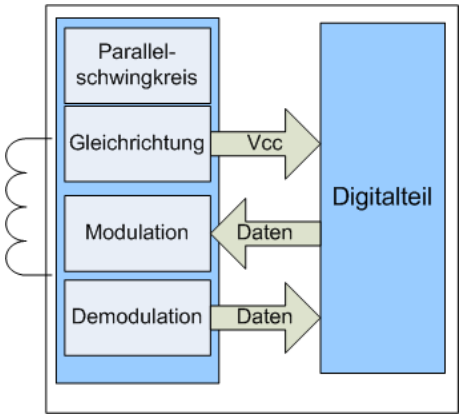

Bild 9: Blockschaltbild des Transponders

2.2.1 Auswahl der digitalen Hardware

Bei der Auswahl der digitalen Hardware stehen die Aspekte des geringen Energiebedarfs auf der Transponderseite, programmierbar in einer Hochsprache und die gleiche Familienzugehörigkeit der Bauelemente (im Gesamtsystem) im Fokus. Infolge dieser Auswahlkriterien wird ein Mikrocontroller der Firma Texas Instruments aus der MSP430 Familie verwendet. Die Entscheidungsbegründung und weitere Eigenschaften des Mikrocontrollers sind in dem folgenden Abschnitt dargestellt.

Texas Instruments MSP430F1232 Die MSP430-Mikrocontrollerfamilie von Texas Instruments verfügt über eine große Controllerauswahl, die in C und/oder Assembler programmierbar sind und mit leistungsfähiger Peripherie ausgestattet sind. Das Hauptaugenmerk liegt bei der niedrigen Stromaufnahme, die sich im μA-Bereich bewegt. Auf Grund der folgenden stichpunktartigen Eigenschaften wird der MSP430F1232 aus der Controller-Familie für den Transponder verwendet.

- Stromaufnahme im μA-Bereich
- Verschiedene Low-Power-Modi
- Interne über Software steuerbare Takterzeugung
- 10 bit ADC mit interner Referenzspannung
- Timer (seriellen Datenstrom einlesen)

Der MSP430F1232 ist ein 16 Bit Mixed Signal Mikrocontroller mit Reduced Instruction Set Computing (RISC), bei dem 27 hardverdrahtete Grundfunktionen und 24 emulierte Befehle implementiert sind. Er besitzt eine klassische Von-Neumann-Architektur, bei der Steuerbefehle und Daten auf einem Bus laufen.

Eine genaue Beschreibung des Mikrocontrollers erfolgt an dieser Stelle nicht, sondern es wird auf das Datenblatt [1] verwiesen. Die einzelnen Peripheriegruppen, die für die Implementierung des System benötigt werden, werden bei den jeweiligen Implementierungsschritten beschrieben.

Entwicklungsumgebung Als Übersetzungstool wurde sich für den freien C-Compiler mspgcc entschieden. Dieser kann kostenlos von der Internetseite sourceforge.net heruntergeladen werden. Der mspgcc bietet verschiedene Möglichkeiten der Optimierung und ermöglicht die Inlineassembler-Programmierung. Zur leichteren Handhabung wurde der Compiler in die grafische Oberfläche von Eclipse eingefügt. Hierüber ist es auch möglich, den Quellcode auf dem Device zu debuggen. Hierbei stehen Funtionen wie Breakpoints, Singlestep und die Anzeige von Variablen zur Verfügung. Zum Flashen und Debuggen des Controllers wird ein an den Parallelport angeschlossener JTAG-Programmieradapter über einen Proxy angesprochen. Weitere Informationen zur Installation und der genauen Handhabung der Entwicklungsumgebung sind auf den Internetseiten der Hersteller zu finden.

Stromaufnahme MSP430F1232 Damit für weitere Berechnungen die Stromaufnahme der Digitalschaltung genutzt werden kann, ist es notwendig, einen Ersatzwiderstand für den MSP430F1232 zu ermitteln. Da der Ersatzwiderstand eine ausschlaggebende Größe für die Dimensionierung der restlichen Bauelemente ist, wurde er zunächst berechnet und anschließend messtechnisch noch einmal überprüft.
Für die Berechnung wurde näherungsweise ein proportionaler Zusammenhang zwischen Taktfrequenz und Stromaufnahme des MSP430F1232 angenommen. Im Datenblatt von Texas Instruments [1] ist die Stromaufnahme mit 200μA bei einer Taktfrequenz von 1MHz angegeben. Die Betriebstaktfrequenz im späteren Betrieb ist bei ca. 80kHz festgelegt. Damit ergibt sich durch folgende Rechnung (1) ein Ersatzwiderstand von 137,5kΩ.

$$
\begin{aligned}
R(80kHz) &\approx \frac{R(1MHz) * 1MHz}{80kHz} \qquad (1)\\
R(1MHz) &= \frac{2,2V}{200\mu A}\\
R(1MHz) &= 11k\Omega\\
\\
R(80kHz) &\approx \frac{11k\Omega * 1MHz}{80kHz}\\
R(80kHz) &\approx 137,5k\Omega
\end{aligned}
$$

Auf Grund der hohen Bedeutung des Ersatzwiderstandes und der Tatsache, dass der Takt im MSP430F1232 großen Schwankungen unterliegt, die auf Einflüsse wie Temperaturänderung und Spannungsunterschiede zurückzuführen sind, wird der Ersatzwiderstand anhand der folgenden Schaltung in der Abbildung 10 nochmals überprüft.

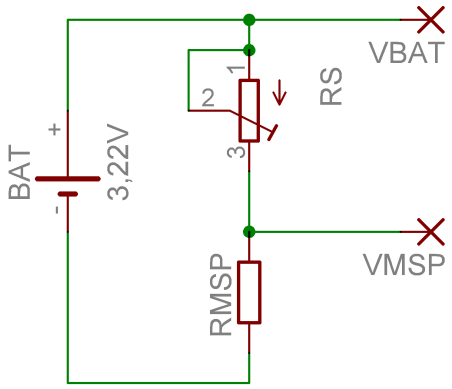

Bild 10: Messschaltung zur Bestimmung des MSP430F1232-Ersatzwiderstandes

Die Tabelle 6 stellt die ermittelten Messwerte dar. Für die weiteren Berechnungen wird der Ersatzwiderstand des MSP430F1232 auf 100kΩ festgelegt. Dies führt bei einer Betriebsspannung von 2V zu einer Leistungsaufnahme von $40\mu W$.

Rs/kΩ	83	56	39
VMSP/ V	1,78	2,04	2,3
VRS/ V	1,44	1,18	0,92
RMSP/ kΩ	101,36	96,81	97,5

Tabelle 6: Bestimmung des Ersatzwiderstandes für MSP430F1232

2.2.2 Energieversorgung

Der im Rahmen dieser Diplomarbeit entstehende Transponder wird wie schon erwähnt als passiver Transponder ohne zusätzliche Spannungsversorgung ausgelegt. Die gesamte Energie zur Versorgung des Transponders wird mit Hilfe eines Parallelschwingkreises und einer Gleichrichterschaltung aus dem vom Reader erzeugten elektromagnetischen Feld gewonnen. Im weiteren Verlauf dieses Abschnitts werden die theoretischen Grundlagen, Berechnungen und Simulationen zu diesen beiden analogen Komponenten zur Spannungserzeugung des Transponders beschrieben.

Parallelschwingkreis Der Schwingkreis zur Energieerzeugung auf dem Transponder besteht aus einer Luftspule und einem parallel geschalteten Kondensator. Die Abbildung 11 zeigt die Schaltung. Wobei sich das Ersatzschaltbild der Spule durch eine ideale Spule und einen in Reihe geschalteten Widerstand, der auf die Kupferverluste der Spule zurückzuführen ist, darstellen lässt. Der Kondensator setzt sich aus dem Kondensator C_1 und der parasitären Kapazität C_P der Gleichrichter-Schaltung zusammen. Bei der Betrachtung des Parallelschwingkreises werden die Kapazitäten C_P und C_1 zum Vereinfachen der Schaltung zu C zusammengefasst. Die Aufteilung der Kapazität C erfolgt im nachfolgenden Kapitel bei der Berechnung der Gleichrichter-Schaltung. Der Kondensator C wird so ausgelegt, dass die Resonanzfrequenz f_0 bei $125kHz$ liegt. Für die Transponderspule wird zunächst eine für RFID-Transponder ausgelegte Luftspule verwendet, für die folgende Daten bekannt sind:

$L = 2,8mH$
$R = 56\Omega$
$r = 12,875mm$
$d = 0,09mm$
$N = 240$

Die restlichen Komponenten im Transponder und im Reader werden bezüglich dieser Werte dimensioniert. Ziel ist es, die Reichweite für die Energieversorgung und den Datentransport zu maximieren.
Damit eine möglichst hohe Spannung am Ausgang des Parallelschwingkreises entsteht, wird der Kondensator nach der folgenden Formel (2) für die Resonanzfrequenz berechnet:

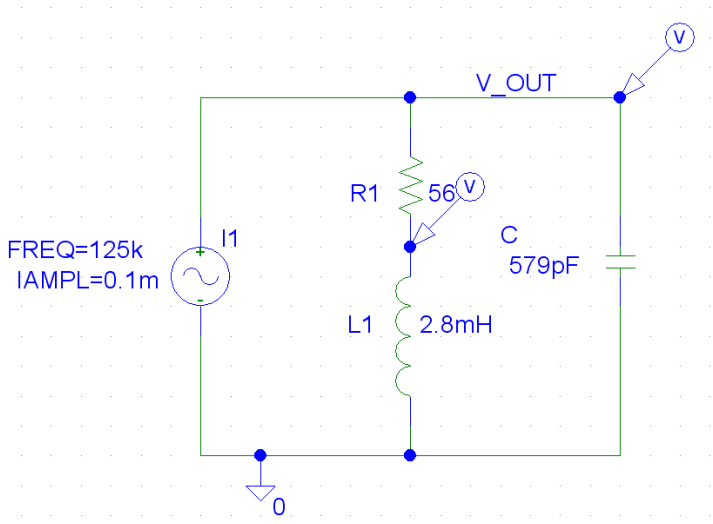

Bild 11: Unbelasteter Transponder Parallelschwingkreis

$$
\begin{aligned}
f_0 &= \frac{1}{2 * \pi * \sqrt{C * L}} \\
C &= \frac{1}{4 * \pi^2 * f_0^2 * L} \\
C &= \frac{1}{4 * \pi^2 * (125kHz)^2 * 2,8mH} \\
C &= 579pF
\end{aligned}
\tag{2}
$$

Die Abbildung 12 zeigt den mit pSpice simulierten unbelasteten Amplitudengang des Parallelschwingkreises. Es kann deutlich erkannt werden, dass die Spannung bei der Resonanzfrequenz ihr Maximum erreicht.

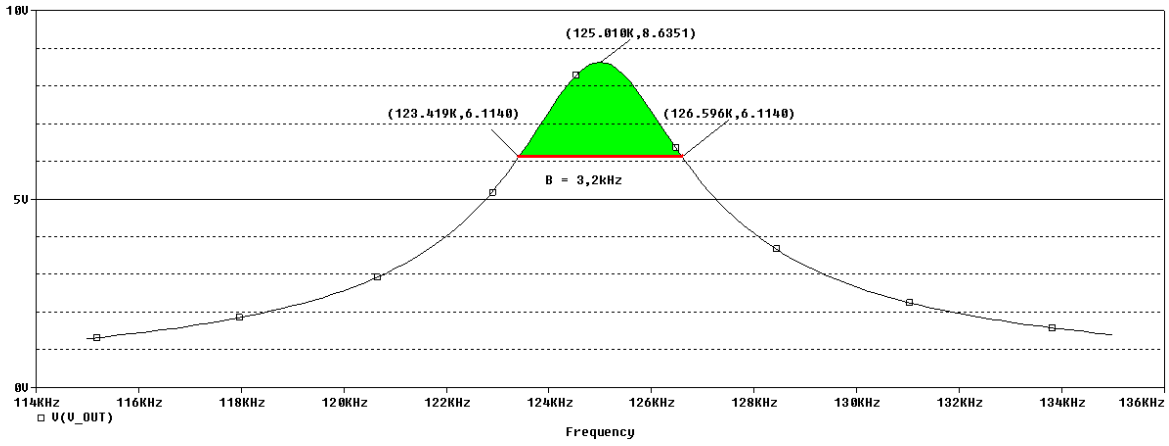

Bild 12: Amplitudengang des Parallelschwingkreises (unbelastet)

Aus der Simulation kann die Bandbreite des Systems direkt bestimmt werden. Hierfür wird die Differenz aus oberer und unterer Grenzfrequenz gebildet. Die Grenzfrequenzen

sind diejenigen Frequenzen bei denen die Spannung um 3dB gegenüber der Resonanzspannung gesunken ist. An den Punkten der Grenzfrequenzen ist die Leistung halb so groß wie die Leistung im Resonanzpunkt. Aus der folgenden Rechnung (3) ergibt sich eine Bandbreite von ca. $3,2kHz$.

$$\begin{aligned} B &= f_2 - f_1 \\ B &= 126,6kHz - 123,4kHz \\ B &= 3,2kHz \end{aligned} \tag{3}$$

Aus der bestimmten Bandbreite B und der bekannten Resonanzfrequenz f_0 kann die Güte Q des unbelasteten Schwingkreises mit der folgenden Gleichung (4) für einen unbelasteten Schwingkreis bestimmt werden.

$$\begin{aligned} Q &= \frac{f_0}{B} \\ Q &= \frac{125kHz}{3,2kHz} \\ Q &= 39 \end{aligned} \tag{4}$$

Zur Überprüfung der simulierten Werte wird die Güte zusätzlich über die Schaltungsparameter berechnet. Hierfür ergibt sich für die Güte Q nach der Gleichung(5) auch ein Wert von $Q = 39$

$$\begin{aligned} Q &= \frac{1}{R} * \sqrt{\frac{L}{C}} \\ Q &= \frac{1}{56\Omega} * \sqrt{\frac{2,8mH}{579pF}} \\ Q &= 39 \end{aligned} \tag{5}$$

Auf Grund der Belastung des Schwingkreises durch die Digitalschaltung und die Modulations- und Demodulationsschaltungen muss für eine genauere Abschätzung der Parameter B und Q noch eine zusätzliche Last in den Schwingkreis hinzugefügt werden. Die Last wird auf Grund der vorigen Berechnungen und Messungen mit $100k\Omega$ angenommen. Die Abbildung 14 zeigt den veränderten Aufbau der Simulationsschaltung. Aus dem Verlauf des Amplitudengangs werden die Parameter B und Q für den belasteten Fall neu bestimmt. Wie erwartet erhöht sich die Bandbreite auf $B = 5,9kHz$ auf Grund der zusätzlichen Belastung. Aus der Bandbreite wird die Güte auf $Q = 21,2$ bestimmt.

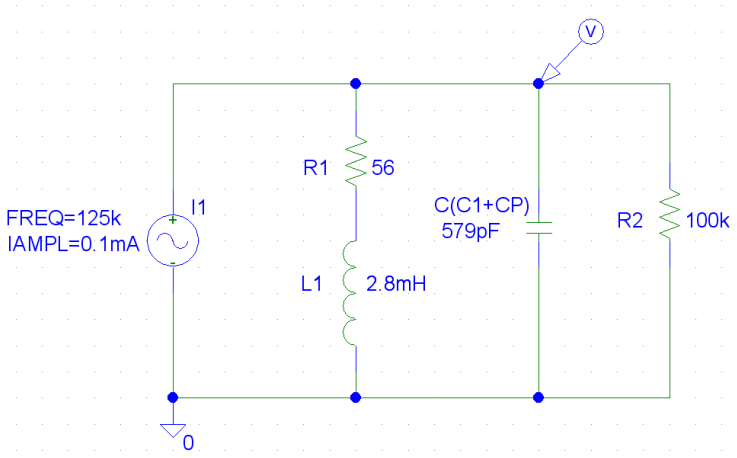

Bild 13: Belasteter Transponder Parallelschwingkreis

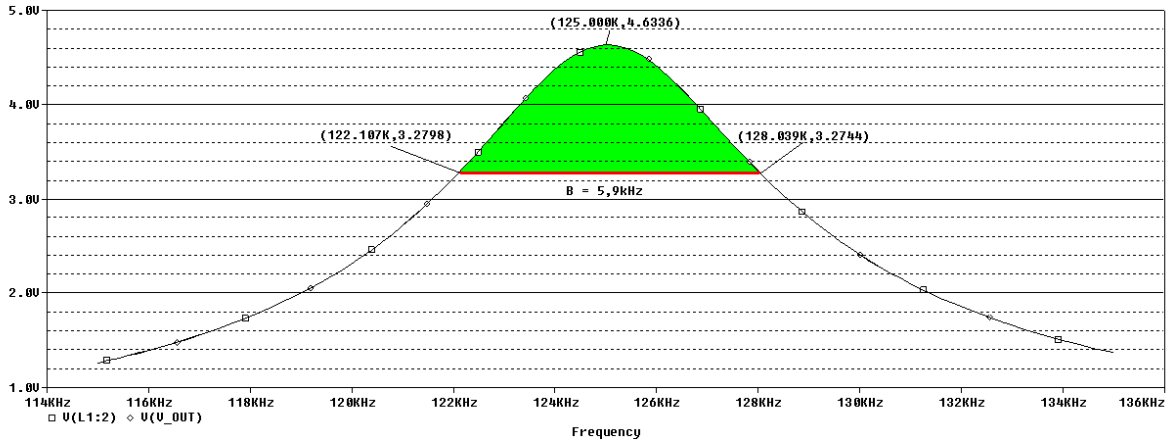

Bild 14: Amplitudengang des Parallelschwingkreises (belastet)

Für die folgenden Berechnungen und Simulationen in den anschließenden Kapiteln werden die Bandbreite und die Güte aus dem belasteten Parallelschwingkreis verwendet, da sie den realen Werten näher entsprechen. Die Einflüsse der Parameter auf das System werden in dem Abschnitt der Readerschwingkreisberechnung näher erläutert, da die Betrachtung der Parameter aus dem Transponder- und Readerschwingkreis zum besseren Verständnis gemeinsam durchgeführt werden muss.

Gleichrichterschaltung Für die Gleichrichtung der am Parallelschwingkreis anliegenden Wechselspannung wird eine Spannungsverdopplerschaltung nach Villard wie in Abbildung 15 zu sehen ist, eingesetzt. Der Vorteil gegenüber einer gewöhnlichen Gleichrichterschaltung ist, dass im Idealfall die gleichgerichtete Spannung doppelt so hoch ist, wie die Eingangswechselspannung. In der Abbildung 16 sind die Ausgangsspannungen für Kapazitäten (C_1) im Bereich von $100pF$ bis $1300pF$ dargestellt.

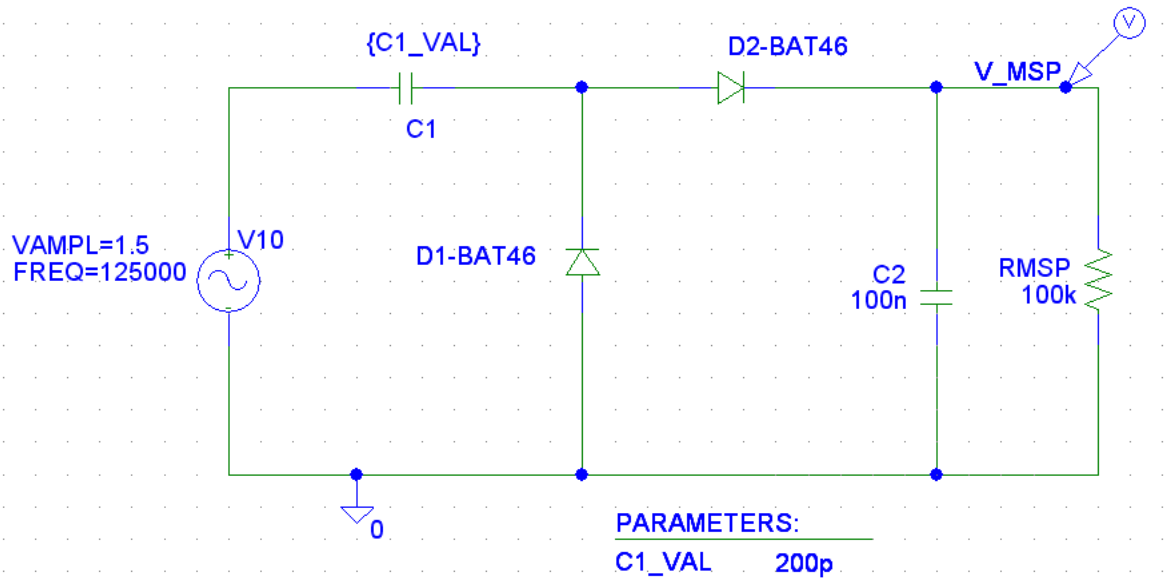

Bild 15: Spannungsverdoppler-Gleichrichter-Schaltung

An den Spannungsverläufen aus der Abbildung 16 kann erkannt werden, dass die größte Ausgangsamplitude bei der größten Kapazität C_1 entsteht. Die maximale Spannung erreicht nicht den doppelten Wert der Eingangsspannung. Die Spannung liegt ca. $0,6V$ unter dem erwarteten Wert. Diese Differenz ist auf die Flussspannungen der beiden Dioden zurückzuführen. Dies ist aber nur im Idealfall, wenn die Spannungsquelle unbegrenzt Strom zur Verfügung stellen kann, der Fall. Für die Simulation der Spannungsverdopplergleichrichtung im Transponder muss zu der Wechselspannungsquelle noch ein Widerstand in Reihe geschaltet werden. Dieser Widerstand stellt die Leistungsbeschränkung der Spannungsquelle im Transponder dar. Für die Simulation wurden Kapazitäten im Bereich von $10pF$ bis $130pF$ eingesetzt. Der Reihenwiderstand und somit der Innenwiderstand der Quelle wurde auf $10k\Omega$ gesetzt. Die Abbildung 17 zeigt, dass die

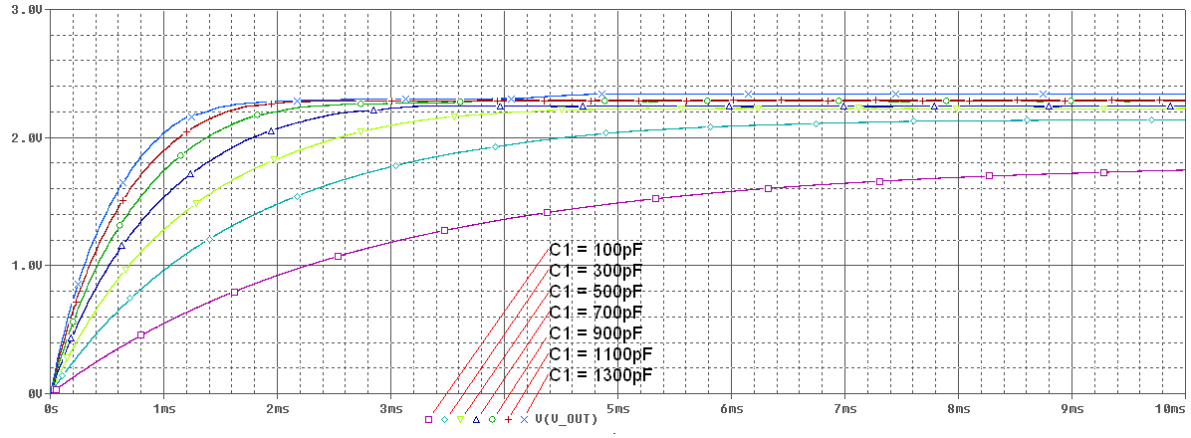

Bild 16: Ausgangsspannung am Gleichrichter mit verschiedenen Kondensatoren

Ausgangsamplitude ab einer Kapazität von ca. $110pF$ nicht mehr nennenswert steigt. Auf Grund dessen wird der Kondensator für die weiteren Simulationen auf $115pF$ festgelegt.

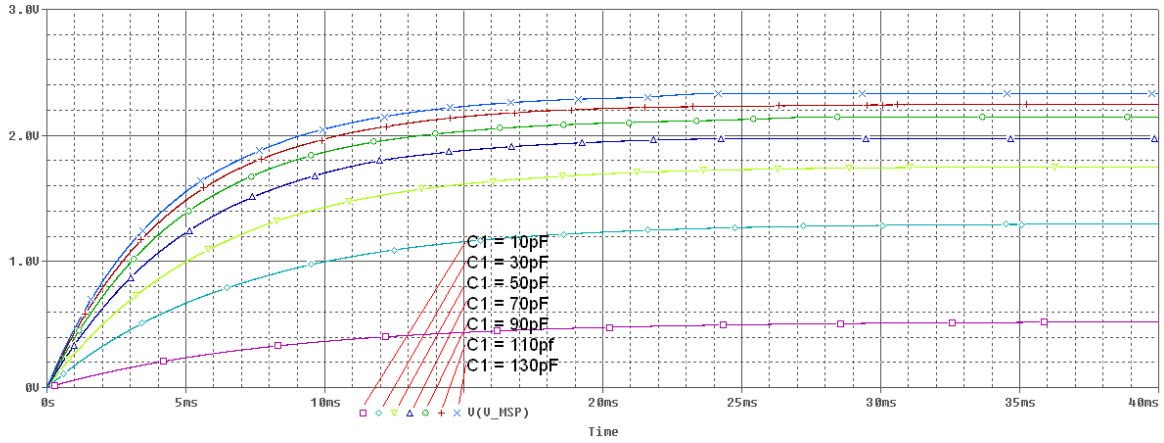

Bild 17: Ausgangsspannung am Gleichrichter mit verschiedenen Kondensatoren und Innenwiderstand

Da der Gleichrichter durch die Kondensatoren und die Dioden auch eine Kapazität darstellt, die bei der Auslegung des Parallelschwingkreises berücksichtigt werden muss, wird die Kapazität mit Hilfe einer Aufladekurve abgeschätzt. Dazu wird die gesamte Gleichrichterschaltung zu C_P zusammengefasst und bildet mit dem Widerstand R ein RC-Glied. Die Kapazität kann jetzt ermittelt werden, indem ein Rechtecksignal an das RC-Glied angelegt wird und zwei Spannungs-Zeit-Punkte von der Auf- oder Endladekurve festgehalten werden. Durch Gleichsetzen der Gleichung (6) und anschließendes Umstellen nach τ kann die Zeitkonstante des RC-Gliedes festgestellt werden. Die Abbildung 18 zeigt den Schaltungsaufbau für die Simulation. Die Aufladekurve und die Messwerte sind aus der Abbildung 19 zu entnehmen.

Die folgende Rechnung (6) zeigt, wie mit Hilfe von zwei Messwerten die unbekannte

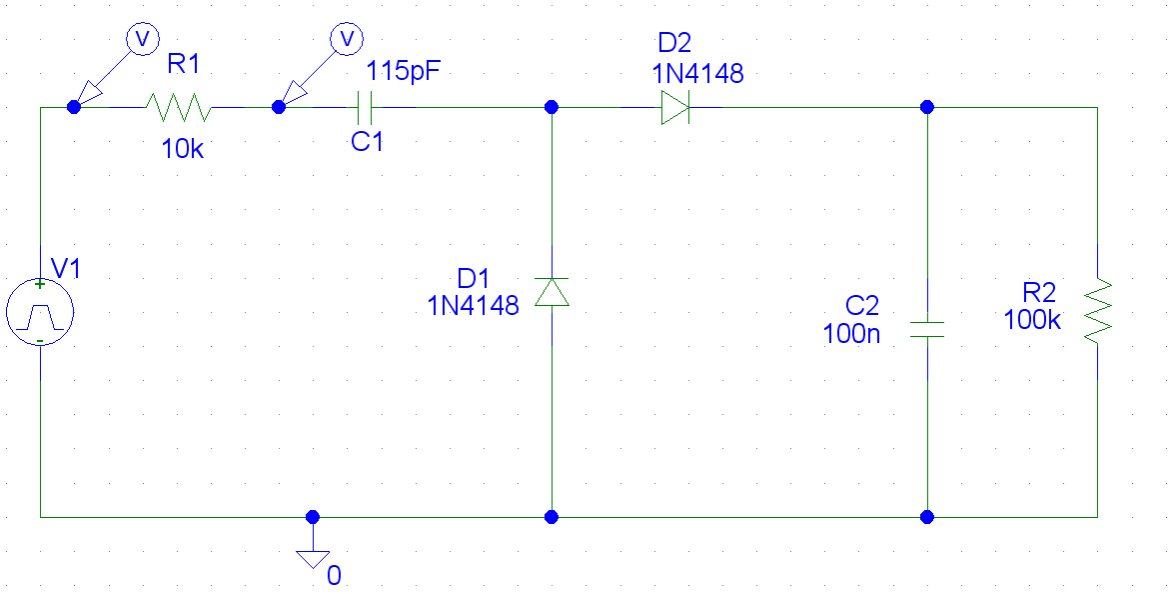

Bild 18: Schaltung zur Bestimmung der parasitären Kapazität des Gleichrichters

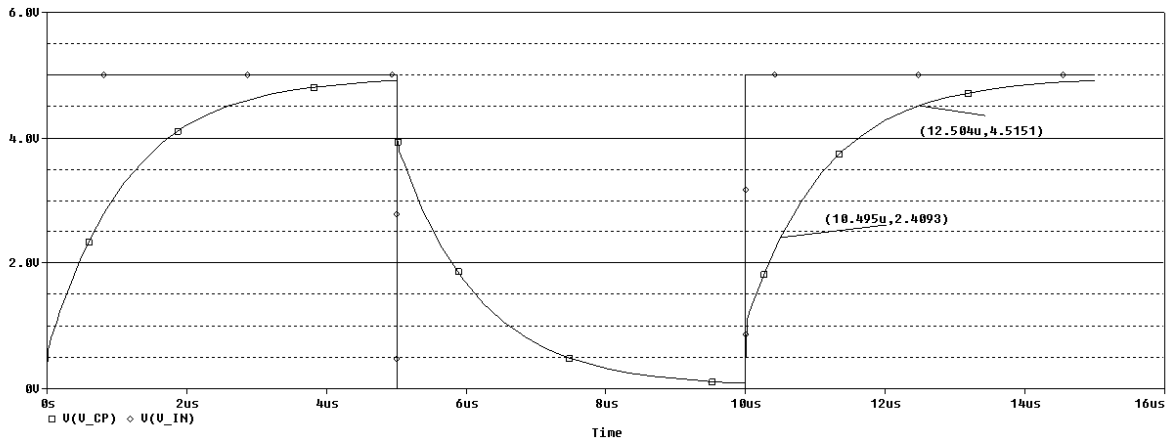

Bild 19: Aufladekurve der Gleichrichterschaltung

Kapazität bestimmt werden kann. Die ermittelten Messwerte aus der Aufladekurve sind $(0,5\mu s,\ 2,41V)$ und $(2,5\mu s,\ 4,52V)$.

$$\begin{aligned} U_C &= U_0 * (1 - e^{(-t/\tau)}) \\ U_0 &= \frac{U_C}{1 - e^{(-t/\tau)}} \\ \frac{U_{C1}}{1 - e^{(-t1/\tau)}} &= \frac{U_{C2}}{1 - e^{(-t2/\tau)}} \\ \tau &= 682,1ns \\ C_P &= \frac{\tau}{R} \\ C_P &= 68,21pF \end{aligned} \tag{6}$$

Um das Zusammenspiel zwischen Parallelschwingkreis und Gleichrichter zu überprüfen, werden die beiden Schaltungteile in einer Simulation zusammengefügt. Die Abbildung 20 zeigt die Gesamtschaltung für die Energieerzeugung im Transponder. Um die zuvor ermittelte parasitäre Kapazität von 68pF zu prüfen, werden die Werte für den Kondensator C im Bereich von 450pF bis 550pF simuliert. Die Abbildung 21 zeigt, dass bei einer Kapazität von $C = 510pF$ die maximale Ausgangsspannung erreicht wird. Dies entspricht dem erwarteten Wert, der wie folgt berechnet wird: $C = C_{f0} - C_P = 579pF - 68pF = 511pF$

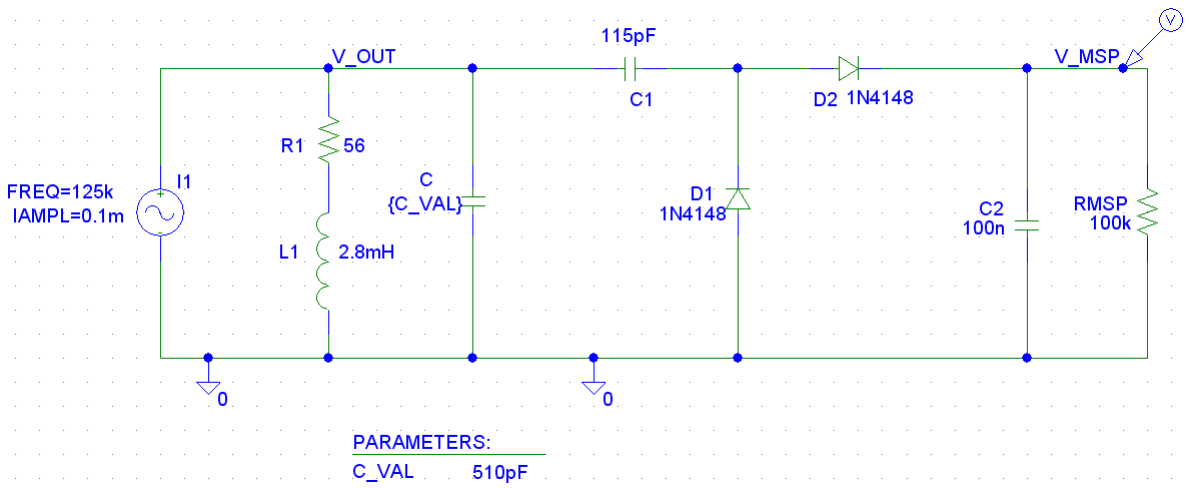

Bild 20: Schaltung für die Energieerzeugung im Transponder

Minimale Ansprechfeldstärke Mit Hilfe der zuvor bestimmten Parameter lässt sich die minimale Ansprechfeldstärke H_{min} berechnen. Die minimale Ansprechfeldstärke gibt an, welche Feldstärke am Transponder herrschen muss, damit er ausreichend mit Energie versorgt ist. Laut [5] berechnet sich die minimale Ansprechfeldstärke nach der Gleichung(7). Mit den zuvor bestimmten Parametern ergibt sich eine minimale Ansprechfeldstärke $H_{min} = 0,577A/m$.

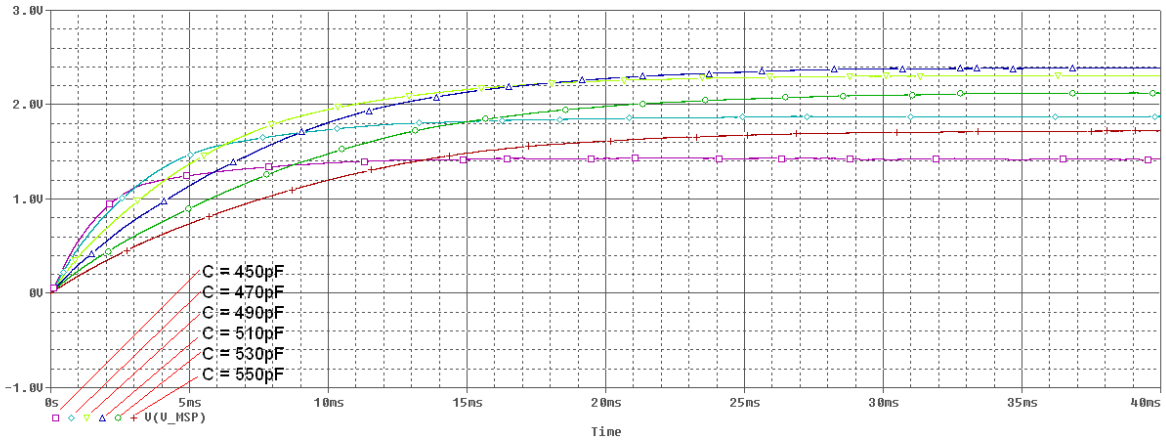

Bild 21: Ausgangsspannung der gesamten Ergieversorgungsschaltung

$u_2 = 1,5V$
$\omega = 2 * \pi * 125kHz$
$L_2 = 2,8mH$
$R_L = 100k\Omega$
$R_2 = 56\Omega$
$C_2 = 579pF$
$A = \pi \cdot (12,875mm)^2 = 521\mu m^2$
$N = 240$

$$H_{min} = \frac{u_2 \cdot \sqrt{(\frac{\omega \cdot L_2}{R_L} + \omega \cdot R_2 \cdot C_2)^2 + (1 - \omega^2 \cdot L_2 \cdot C_2 + \frac{R_2}{R_L})^2}}{\omega \cdot \mu_0 \cdot A \cdot N} \quad (7)$$

Die zuvor berechnete Ansprechfeldstärke ergibt sich nur bei einem komplett abgeglichenen System, bei dem die Resonanzfrequenzen des Transponders und des Readers exakt übereinstimmen. Um eine realistischere Einschätzung der minimalen Ansprechfeldstärke zu gewinnen, wird diese erneut berechnet. Bei der Gleichung (8) [5] wird eine Abweichung von den Resonanzfrequenzen berücksichtigt. Es wird davon ausgegangen, dass die Resonanzfrequenzen bei $f_{0T} = 127kHz$ und $f_{0R} = 122kHz$ liegen. Auf Grund dieser Rechnung ergibt sich eine minimale Feldstärke von $H_{min} = 1,12A/m$. Dieser Wert wird für spätere Entfernungsabschätzungen weiterverwendet.

$$H_{min} = \frac{u_2 \cdot \sqrt{\omega_R^2 \cdot (\frac{L_2}{R_L} + \frac{R2}{\omega_T^2 \cdot L_2})^2 + (\frac{\omega_T^2 - \omega_R^2}{\omega_T^2} + \frac{R_2}{R_L})^2}}{\omega_R \cdot \mu_0 \cdot A \cdot N} \quad (8)$$

2.2.3 Modulation

Damit die im Transponder aufgenommenen Daten an den Reader übertragen werden können, ist ein drahtloser Übertragungskanal zum Reader notwendig. Dieser wird mit

Hilfe des vom Reader ausgesendeten elektromagnetischen Feldes realisiert. Diese Sinusschwingung stellt den Träger bei der Modulation dar. Die digitalen Daten werden auf diesen Träger aufmoduliert. Grundsätzlich gibt es 3 Verfahren zur Modulation eines Trägers. Die Verfahren unterscheiden sich durch den Signalparameter, den sie beeinflussen. Es kann die Leistung, die Frequenz oder die Phase des Trägers verändert werden. Bei digitaler Modulation nennen sich diese Verfahren entsprechend ASK, FSK und PSK. Für den Transponder wird die Amplitudenmodulation (ASK) gewählt. Der Grund für diese Entscheidung liegt in der einfachen Modulation und Demodulation des Signals. Bei der digitalen Amplitudenmodulation wird ein Träger entsprechend des digitalen Signals zwischen zwei Amplituden umgeschaltet. Die Abbildung 22 zeigt den grundsätzlichen Ablauf einer Amplitudenmodulation. In der Abbildung 23 sind die Signalverläufe an den 2 Eingängen und dem Ausgang des Multiplizierers aus der Abbildung 22 zu sehen. Zusätzlich ist der Modulationsgrad zu erkennen, der in diesem Beispiel bei 40% liegt.

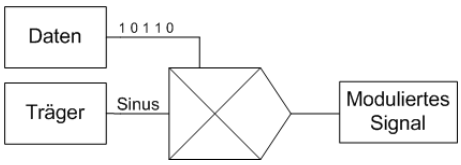

Bild 22: Blockschaltbild der Amplitudenmodulation

Der Quellcode für die MATLAB-Simulation der Amplitudengänge ist im Anhang unter A.1.1 zu finden.
Da der Transponder keinen eigenen Träger für die Datenübertragung generiert, wird das von Reader generierte Feld als Träger verwendet und mit Hilfe eines Lastwiderstandes moduliert. Die Schaltung 24 zeigt den Aufbau der Simulationsschaltung. In diesem Beispiel wird der Lastwiderstand periodisch mit einer Frequenz von 1kHz zugeschaltet. In der späteren Transponderschaltung stellen die digitalen Daten den seriellen Datenstrom dar, der die Last zuschaltet. Diese Aufgabe wird vom Mikrocontroller übernommen.

In der Abbildung 25 ist der Amplitudenverlauf der Amplitudenmodulation zu sehen. Das dazugehörige Spektrum ist in der Abbildung 26 dargestellt. In dem Spektrum sind die Seitenbänder gut zu erkennen. Sie liegen bei $f_0 \pm f_{mod} = 124kHz, 126kHz$. Die weiteren spektralen Anteile treten durch die Modulation mit einem Rechteck-Signal auf.

2.2.4 Demodulation

Damit die vom Reader erzeugten Steuerdaten vom Transponder gelesen werden können, müssen diese zunächst demoduliert werden. Für den Daten-Upload zum Transponder wird ein Spezialfall der Amplitudenmodulation verwendet, das On-Off-Keying. Beim On-Off-Keying wird die Amplitude des Trägers zwischen voller Aplitude und $0V$ im Takt des Datenstromes umgeschaltet. Ein Nachteil dieses Verfahrens ist, dass während der Zeit,

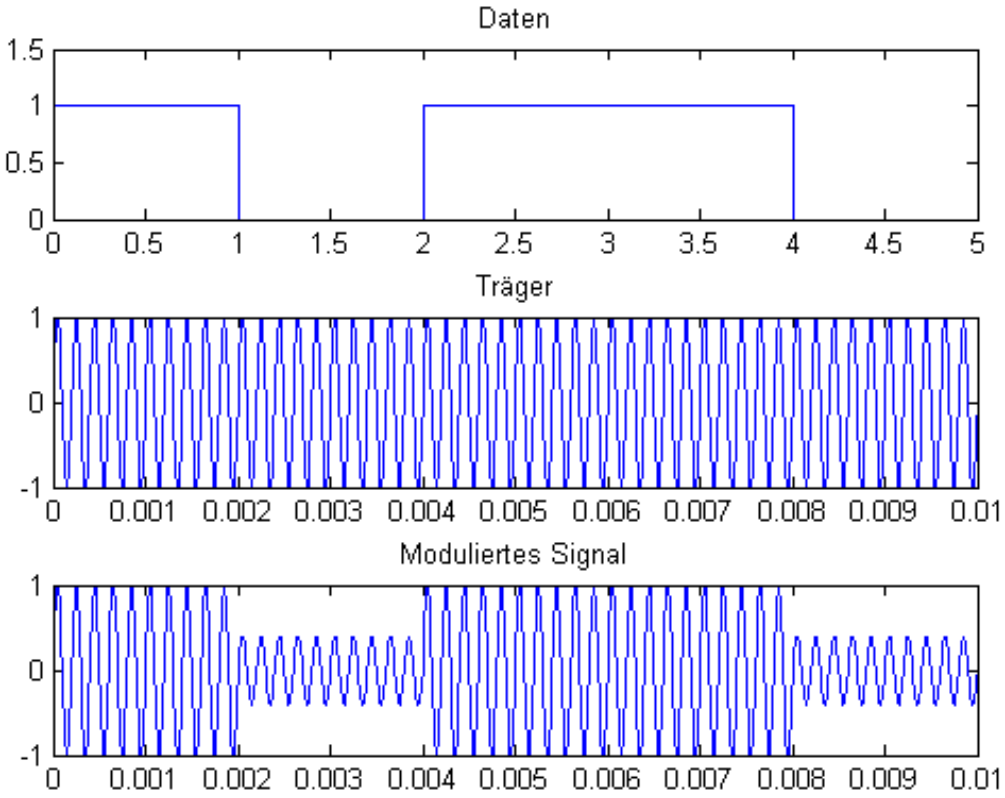

Bild 23: Signalverläufe bei der Amplitudenmodulation

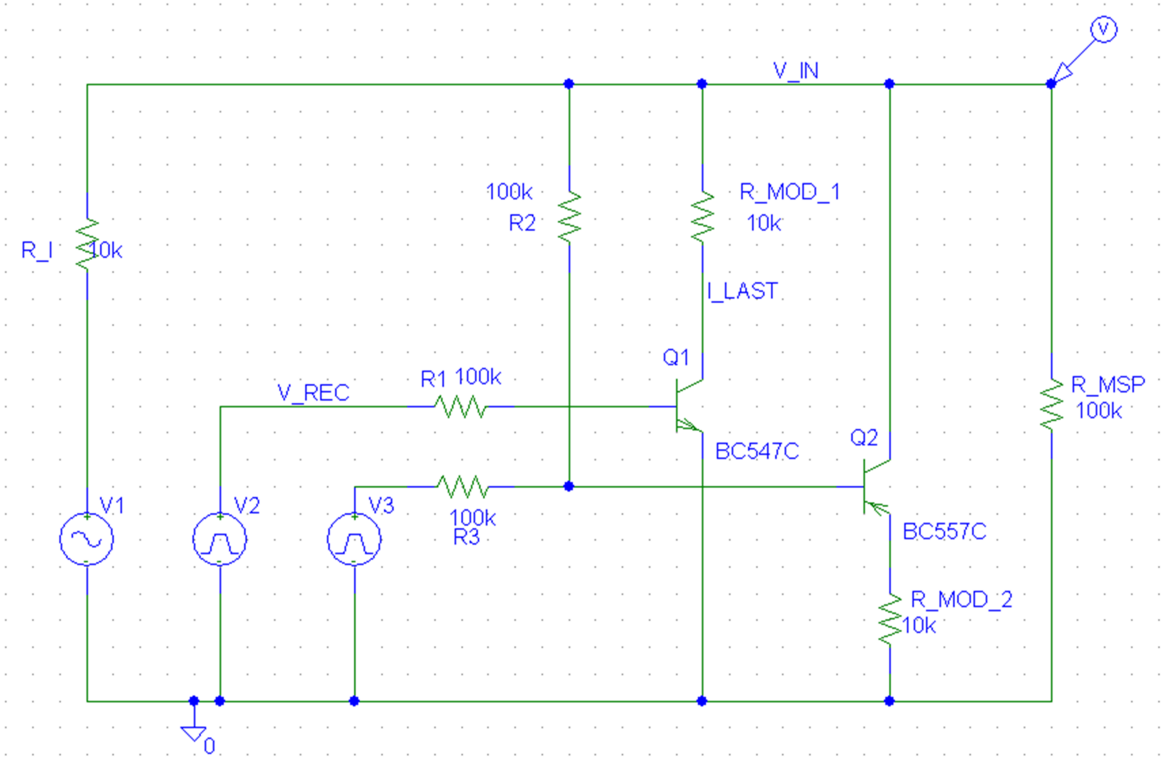

Bild 24: Simulationsschaltbild der Amplitudenmodulation

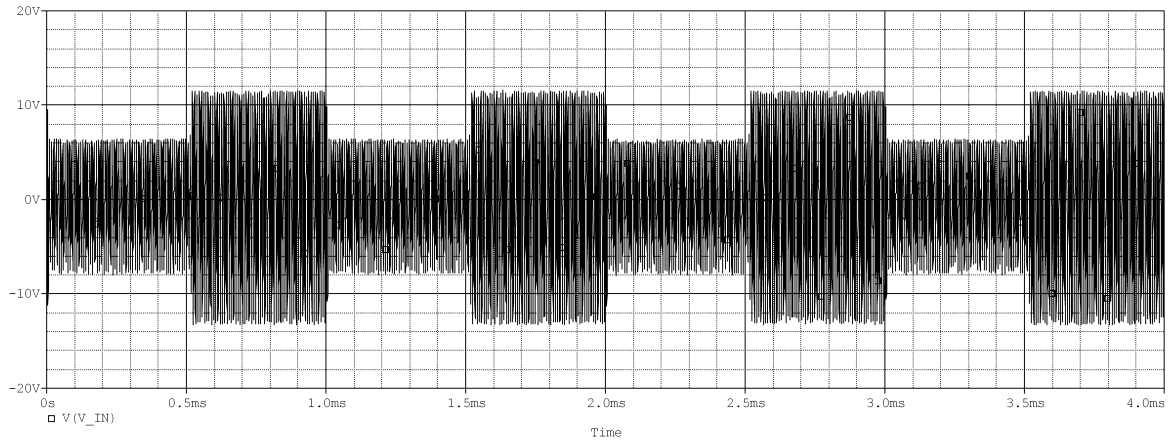

Bild 25: Amplitudengang der Amplitudenmodulation

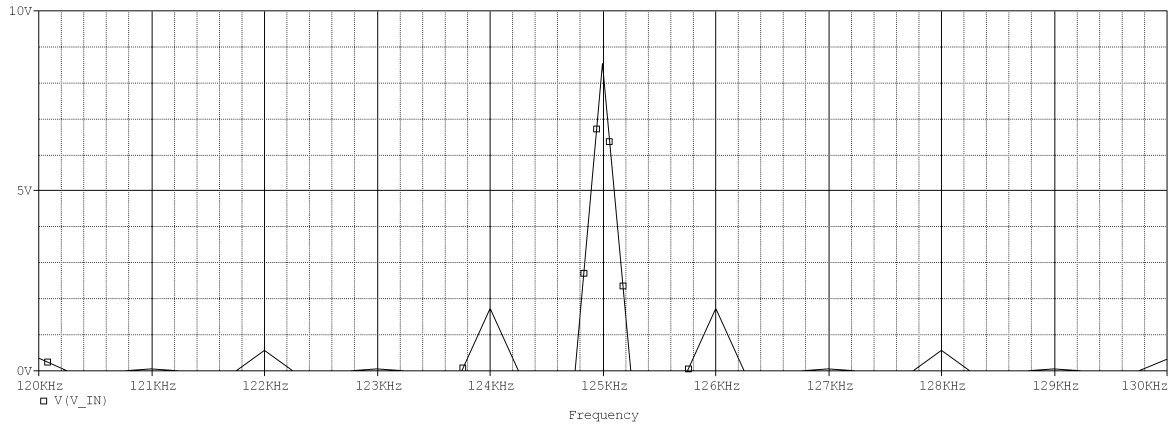

Bild 26: Spektrum der Amplitudenmodulation

in der der Träger abgeschaltet ist, kein Feld für die Energieversorgung des Transponders zur Verfügung steht. Damit der Transponder während dieser Zeit mit ausreichend Energie versorgt ist, wird zuvor ein Kondensator aufgeladen, der die Energieversorgung während der Off-Phase darstellt. Der entscheidende Vorteil und damit der Grund für die Wahl des On-Off-Keying ist die sehr einfache Demodulation des Signals. Für die Demodulation reicht eine einfache Gleichrichtung mit nachgeschaltetem RC-Tiefpass und einer Amplitudenbegrenzung in Form einer Diodenkette aus. Die Modulationsfrequenz für den Daten-Upload ist für die leichtere Demodulation und Decodierung im Transponder auf $100Hz$ gelegt. Der Filter hat somit die Aufgabe, die $100Hz$ von den $125kHz$ des Trägers zu trennen. Als Grenzfrequenz wurde 15kHz gewählt und mit der folgenden pSpice-Simulation aus der Abbildung 27 bestätigt. Die Signalverläufe sind in der Abbildung 28 dargestellt.

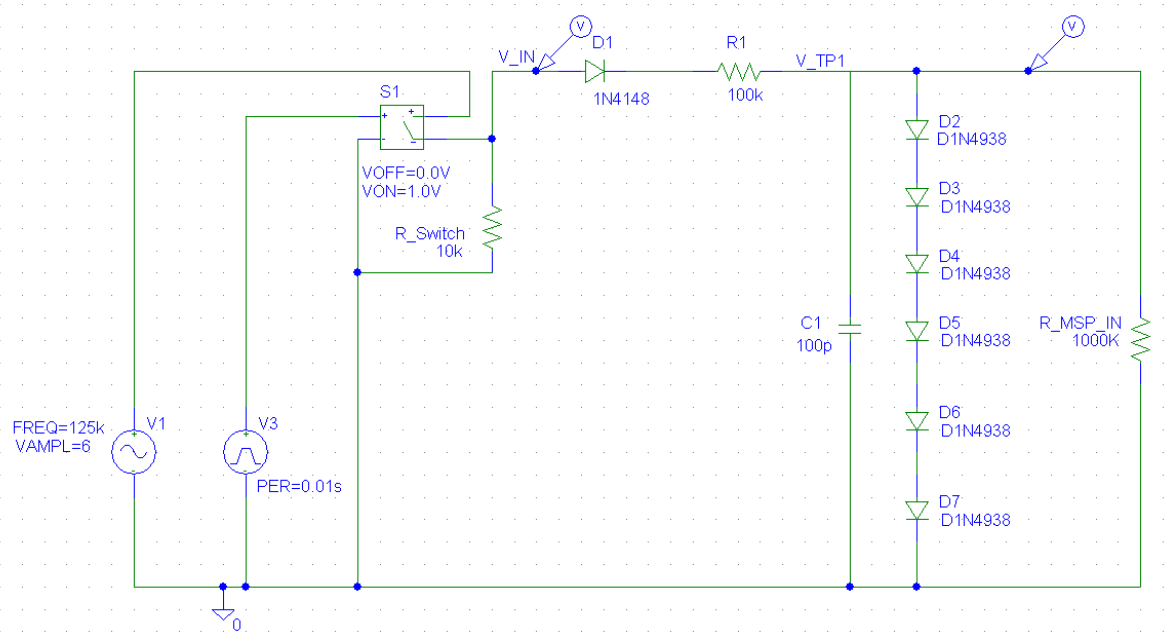

Bild 27: Schaltung der Amplitudendemodulation

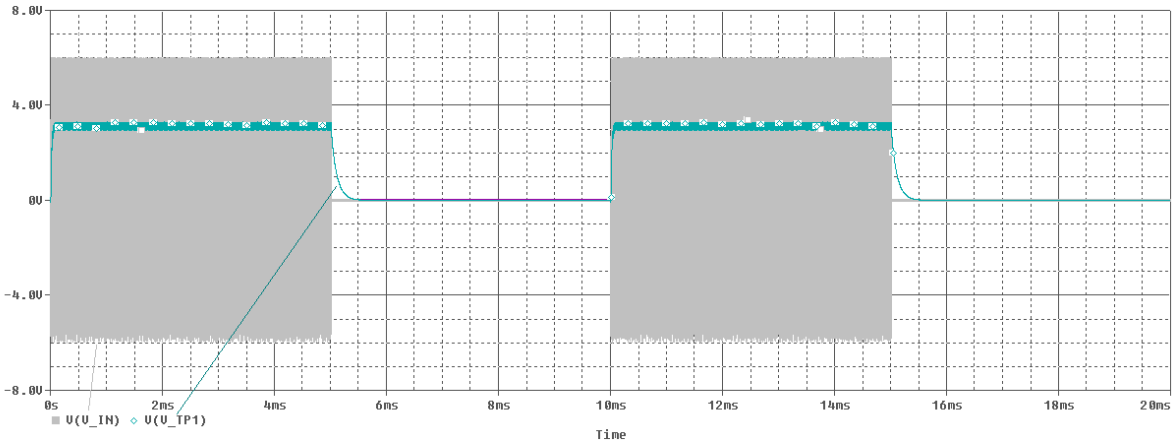

Bild 28: Amplitudengang der Amplitudendemodulation

2.3 Reader

Der Reader stellt die Basisstation des Systems dar. Er besteht aus mehreren Komponenten, die selbstständig ihre Teilaufgaben erfüllen. Zu den grundlegensten Funktionen des Readers gehört die Erzeugung des elektromagnetischen Feldes und dessen Modulation. Durch die Modulation des Feldes werden Daten vom Reader zum Transponder übertragen. Die Modulation erfolgt wie in dem vorigen Kapitel angedeutet durch On-Off-Keying. Die Besonderheit bei dem im Rahmen dieser Diplomarbeit entwickelten Reader ist der automatische Schwingkreisabgleich, der durch die Überwachung verschiedener Parameter von einem Mikrocontroller gesteuert wird. Die Schaltungen für die Aufnahme der Systemparameter, sowie das genaue Verfahren zum Abgleichen des Schwingkreises wird in den folgenden Unterabschnitten beschrieben.
Als zweite entscheidene Aufgabe ist die Demodulation der vom Transponder gesendeten Daten zu nennen. Das Abtrennen der Nutzdaten vom Träger ist in diesem Fall erheblich schwieriger als die Demodulation im Transponder, da der Modulationsgrad im Bereich von $0,1\%$ $(-60dB)$ liegt. Auf Grund des niedrigen Modulationsgrades reicht ein passiver Amplitudendemodulator, wie er im Transponder genutzt wird, nicht aus. Für die Filterung muss in diesem Fall entweder ein Signalprozessor eingesetzt werden, der das analoge Signal digitalisiert und anschließend die Filterung vornimmt, oder es muss eine aktive Filterung mit Operationsverstärkern durchgeführt werden. Zur besseren Analyse des Datenflusses und auf Grund der schnelleren Implementierung und des niedrigeren Preises wird für dieses drahtlose Sensorsystem ein aktives Filtersystem entworfen. Das Blockschaltbild und die Berechnung der einzelnen Komponenten sind in dem dazugehörigen Unterabschnitt beschrieben.
Zusätzlich zu der Felderzeugung und der bidirektionalen Kommunikation steuert ein weiterer Mikrocontroller das Auslesen der einzelnen Transponder und verwaltet die empfangenen Daten in einem externen Flash-Speicher. Der Mikrocontroller ist zusätzlich mit einer Schnittstelle zu einem PC ausgestattet. Ist kein PC an den Mikrocontroller angeschlossen, läuft der Mikrocontroller mit zur Programmierzeit festgelegten Standardparametern. In diesem Fall fragt der Mikroprozessor alle Transponder aus dem Standard-Adressbereich ab und speichert die erhaltenen Daten im Flash ab. Bei dem Betrieb mit angeschlossenem PC ist es möglich, über eine selbstentwickelte Applikation die Standardparameter zu ändern und weitere Steuerfunktionen auszuführen. Die Beschreibung der Firmware und der Applikation folgt in den späteren Abschnitten, die sich mit der Implementierung der Komponenten beschäftigen. Für den Fall, dass der Mikrocontroller ohne einen PC läuft, ist ein Display in dem Reader integriert, das die wichtigsten Systemparameter und den aktuellen Bearbeitungsstatus anzeigt. Das folgende Blockschaltbild 29 zeigt den schematischen Aufbau des Readers.

In den folgenden 4 Unterabschnitten wird der Aufbau und die Funktionsweise der einzelnen Blöcke des Analogteils beschrieben. Die Blöcke Resonanzabgleich, Amplitudenmes-

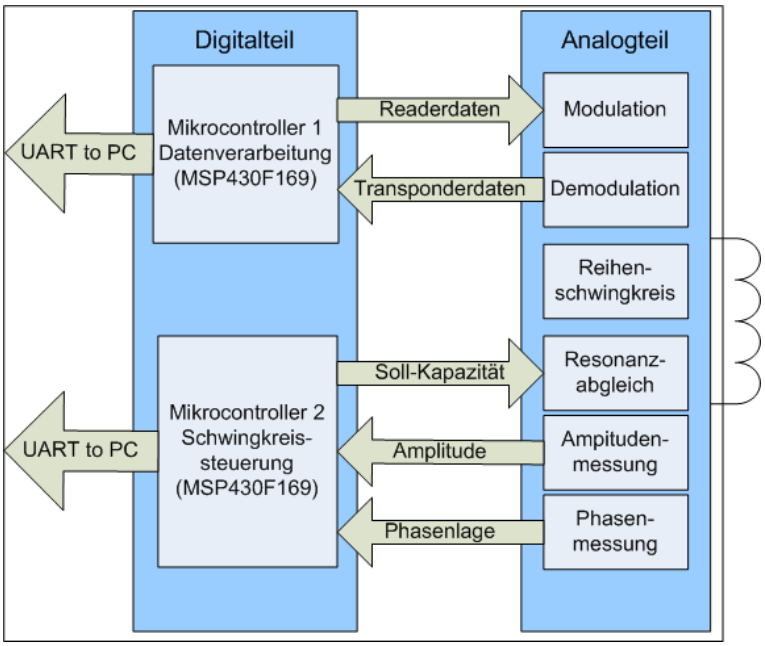

Bild 29: Blockschaltbild des Readers im Experimentiersystem

sung und Phasenmessung werden zusammen in einem Abschnitt behandelt. Im Anschluss werden der Aufbau und die Funktionsanforderungen des Digitalteils beschrieben.

2.3.1 Reihenschwingkreis

Für die Erzeugung des elektromagnetischen Readerfeldes wird ein Reihenschwingkreis eingesetzt. Der Schwingkreis besteht aus einem Widerstand, der die Güte und Bandbreite des Systems mitbestimmt, einer Luftspule, deren Berechnung folgt und parallel geschalteten Kondensatoren, mit denen es möglich ist, den Schwingkreis im laufenden Betrieb auf Resonanz abzustimmen. In diesem Abschnitt werden zur leichteren Darstellung die parallelen Kondensatoren zu einer Gesamtkapazität zusammengefasst. Die genaue Schaltung und das Vorgehen beim Resonanzabgleich werden in den folgenden drei Unterabschnitten beschrieben. Damit ein möglichst starkes elektromagnetisches Feld vom Schwingkreis erzeugt wird, wird dieser in Resonanz bei $125kHz$ betrieben. Die Spulenberechnung und die übrigen Systemparameter werden für eine Distanz von $10cm$ zwischen Readerspule und Tansponder ausgelegt.

Berechnung der Readerspule Für die Readerspule wird eine selbstberechnete Luftspule eingesetzt. Die Induktivität wird mit $L = 1mH$ festgelegt. Die restlichen Spulenparameter wie Radius und Wicklungsanzahl werden auf der Grundlage der gegebenen Induktivität berechnet.
Zunächst wird der optimale Spulenradius für die gegebene Distanz von $10cm$ berechnet.

Dazu wird die Gleichung (9) zur Berechnung der Feldstärke nach dem Radius abgeleitet und gleich 0 gesetzt, um das Maximum der Feldstärke zu bestimmen. Die Gleichung für die Bestimmung der Feldstärke ist [5] entnommen.

$$\begin{aligned} H &= \frac{I \cdot N \cdot R^2}{2 \cdot \sqrt{(r^2 + x^2)^3}} \\ \frac{\partial H}{\partial r} &= \frac{-0.5 \cdot I \cdot N \cdot r \cdot (r^2 - 2 \cdot x^2)}{(r^2 + x^2)^{5/2}} \\ 0 &= \frac{-0.5 \cdot I \cdot N \cdot r \cdot (r^2 - 2 \cdot x^2)}{(r^2 + x^2)^{5/2}} \\ 0 &= r^2 - 2 \cdot x^2 \\ r &= \sqrt{2} \cdot x \end{aligned} \tag{9}$$

Bei einer Distanz von $10cm$ zwischen Readerspule und Transponder ergibt sich somit ein Spulenradius von $14,1cm$.
Für die Berechnung der Wicklungsanzahl wird die Gleichung (10) für die Induktivität aus dem Datenblatt [4] von Microchip als Berechnungsgrunglage verwendet. Durch anschließendes Umstellen der Gleichung ergibt sich die Wicklungszahl $N = 45$. Die folgende Rechnung zeigt die Vorgehensweise.

Induktivität: $L = 1mH$
durchschnittlicher Spulenradius: $r = 0,1414m$
Wicklungshöhe: $h = 0,04m$
Wicklungsbreite: $b = 0,0004m$
Wicklungszahl: gesucht

$$\begin{aligned} L &= \frac{3,1 \cdot 10^{-5} \cdot (r \cdot N)^2}{6 \cdot r + 9 \cdot h + 10 \cdot b} \\ N &= \sqrt{\frac{L \cdot (6 \cdot r + 9 \cdot h + 10 \cdot b)}{3,1 \cdot 10^{-5} \cdot r^2}} \end{aligned} \tag{10}$$

Für die weiteren Berechnungen wird der Kupferwiderstand der Spule benötigt. Dieser wird mit Hilfe des spezifischen Widerstandes von Kupfer bestimmt. In der folgenden Rechnung (11) wird der Kupferwiderstand auf $R_K = 5,66\Omega$ bestimmt.

$d = 0,4 \cdot 10^{-4}$
$r = 0,1414m$
$N = 45$
$\rho = 0,0178 \frac{\Omega \cdot mm^2}{m}$

$$
\begin{aligned}
l &= N \cdot 2 \cdot \pi \cdot r \\
R_K &= \frac{l \cdot \rho}{A} \\
R_K &= \frac{45 \cdot 2 \cdot \pi \cdot 0,1414m \cdot 0,0178 \frac{\Omega \cdot mm^2}{m}}{\pi \cdot (\frac{0,4mm}{2})^2} \\
R_K &= 5,66\Omega
\end{aligned} \tag{11}
$$

Resonanz Um die größtmögliche Feldstärke zu erreichen, wird der Readerschwingkreis in Resonanz bei $125kHz$ betrieben. Bei einer Induktivität der Readerspule von $L = 1mH$ ergibt sich nach $f_0 = \frac{1}{2 \cdot \pi \cdot \sqrt{L \cdot C}}$ eine Kapazität von $C = 1,62nF$. Für die Erhöhung der Bandbreite wird ein Widerstand in Reihe zu der Spule und der Kapazität geschaltet. Zusätzlich zu der erhöhten Datenübertragungsrate ergibt sich eine höhere Toleranz in Bezug auf die Transponderresonanzfrequenz durch den in Reihe geschalteten Widerstand. Die PSpice Simulation in Abbildung 30 zeigt die Amplitudengänge für verschiedene Reihenwiderstände im Bereich von 5Ω bis 25Ω der Simulationsschaltung aus Abbildung 31. Für einen Reihenwiderstand von $R = 15\Omega$ (gelbe Kurve) ergibt sich wie in der Abbildung eingezeichnet eine Bandbreite von $B = 3,3kHz$. Die Güte des Reihenschwingkreises ist aus dem Quotienten aus Resonanzspannung und Anregespannung zu bilden. Damit ergibt sich für die Gte $Q = 190V/5V = 38$. Um die simulierte Bandbreite und Güte zu prüfen, werden die Güte Q und die Bandbreite B rechnerisch über die Parameter des Schwingkreises erneut bestimmt.

$$
\begin{aligned}
Q &= \frac{Q_L}{R_V} = \frac{2 \cdot \pi \ f_0 \cdot L}{R_V} \\
mit & \\
f_0 &= \frac{1}{2 \cdot \pi \cdot \sqrt{L \cdot C}} \\
Q &= \frac{2 \cdot \pi \cdot L}{R_V \cdot 2 \cdot \pi \cdot \sqrt{L \cdot C}} \\
Q &= \frac{1}{R_V} \cdot \sqrt{\frac{L}{C}} \\
Q &= \frac{1}{20,66\Omega} \cdot \sqrt{\frac{1mH}{1,62nf}} = 37,6 \\
B &= \frac{f_0}{Q} = \frac{125kHz}{37,6} = 3,32kHz
\end{aligned} \tag{12}
$$

Die berechneten Werte stimmen mit den simulierten Werten für die Güte und die Bandbreite überein. Die beiden Rechnungen zeigen den Konflikt auf, der bei der Dimensionierung des Schwingkreises entsteht, da die Güte direkt abhängig von der Bandbreite ist und

umgekehrt. Wird eine Größe durch die Veränderung des Reihenwiderstands vergrößert, verkleinert sich die andere. Die Güte kann in diesem Fall als Maß für die Energieversorgung genommen werden. Die Bandbreite beschreibt das Maß für die Datenübertragung. Die Auslegung des Reihenwiderstands muss so erfolgen, dass die Reichweiten für die Datenübertragung und die Energieversorgung gleich sind. So wird die größtmögliche Arbeitsreichweite für das System erzielt.
Für die spätere Berechnung für die vom Reader erzeugte Feldstärke wird die Stromstärke im Schwingkreis benötigt. Die Stromstärke berechnet sich nach $I = \frac{U_0}{Z}$. Die folgende Rechnung zeigt, dass sich bei der Resonanzfrequenz der Widerstand allein aus dem Verlustwiderstand der Spule R_K und dem Serienwiderstand R_S zusammensetzt. Somit ergibt sich ein Strom von $I = 0,242A$.

$$
\begin{aligned}
Z &= R_K + R_S + abs(X_L + X_C) \qquad (13)\\
Z &= R_K + R_S + abs((2 \cdot \pi \cdot f \cdot L) - \frac{1}{2 \cdot \pi \cdot f \cdot C})\\
Z &= 5,66\Omega + 15\Omega + abs((2 \cdot \pi \cdot 125kHz \cdot 1mH) - \frac{1}{2 \cdot \pi \cdot 125kHz \cdot 1,62nF})\\
Z &= 20,66\Omega + abs(785\Omega - 785\Omega)\\
Z &= 20,66\Omega
\end{aligned}
$$

$$
\begin{aligned}
I &= \frac{U_0}{Z} = \frac{5V}{20,66\Omega}\\
I &= 0,242A
\end{aligned}
$$

Feldstärke im Bezug auf Transponderabstand und Spulenradius Für die Abschätzung für die vom Readerschwingkreis erzeugte Feldstärke wird mit Hilfe einer MATLAB-Simulation die Feldstärke in Abhängigkeit vom Spulenradius und der Distanz zwischen Readerspule und Transponder berechnet. Die Abbildung 32 auf Seite 41 zeigt das mit MATLAB erzeugte Diagramm, in dem die Feldstärke in Abhängigkeit zum Spulenradius dargestellt wird. Für die Distanz zwischen Readerspule und Transponder werden $7,5cm$, $10cm$ und $12,5cm$ als Berechnungsgrundlagen angenommen. Die Abbildung 33 auf Seite 42 zeigt das MATLAB-Diagramm, dass die Feldstärke in Abhängigkeit von der Entfernung zwischen Readerspule und Transponder zeigt. Die Radien für die Berechnungen betragen $10,6cm$, $14,1cm$ und $17,7cm$. Zusätzlich ist eine horizontale Linie bei der Feldstärke $H_{min} = 1,12A/m$ eingezeichnet, die der im vorigen Abschnitt berechneten minimalen Ansprechfeldstärke des Transponders entspricht. In der Abbildung 34 auf Seite 42 ist der Schnittpunkt mit der minimalen Ansprechfeldstärke des Transponders und der Feldstärke in Abhängigkeit zur Distanz zwischen Readerspule und Transponder vergrößert dargestellt. Auf Grund des Schnittpunktes zwischen der

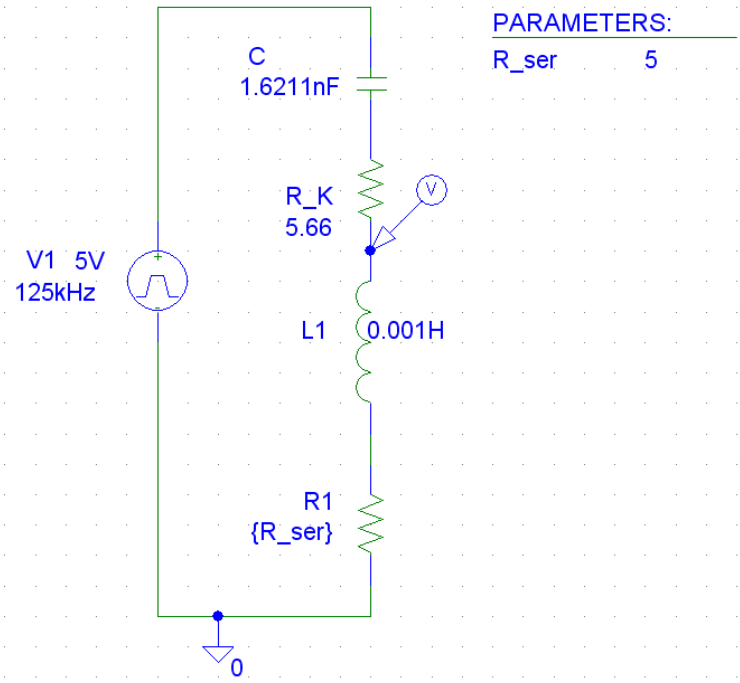

Bild 30: Readerschwingkreis Schaltbild

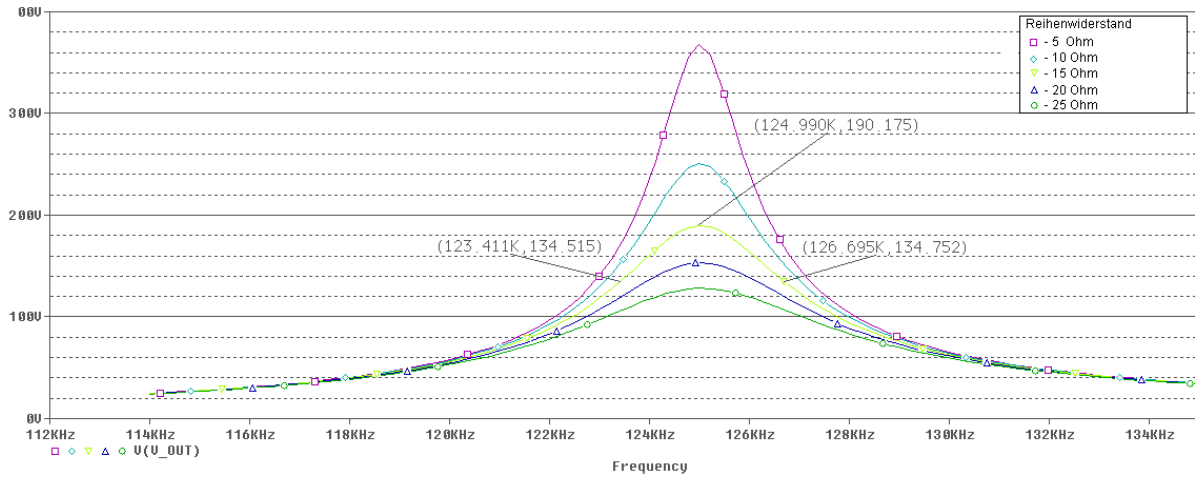

Bild 31: Readerschwingkreis Amplitudengang

minimalen Ansprechfeldstärke und der herrschenden Feldstärke ergibt sich für einen Spulenradius von $r = 14,1cm$ eine theoretische Distanz von $44cm$ zwischen Readerspule und Transponder. Innerhalb von dieser Entfernung ist der Transponder mit ausreichend Energie versorgt, wenn die Transponderspule und die Readerspule auf einer gemeinsamen Achse, die durch die Mittelpunkte der Spulen verläuft parallel zueinander ausgerichtet sind. Da dieser Fall in der Praxis nur in den seltensten Fällen eintritt, ist mit einer niedrigeren maximalen Entfernung zwischen Readerspule und Transponder zu rechnen. Der MATLAB-Code für die Berechnung und Darstellung der Diagramme ist im Anhang unter A.1.2 zu finden.

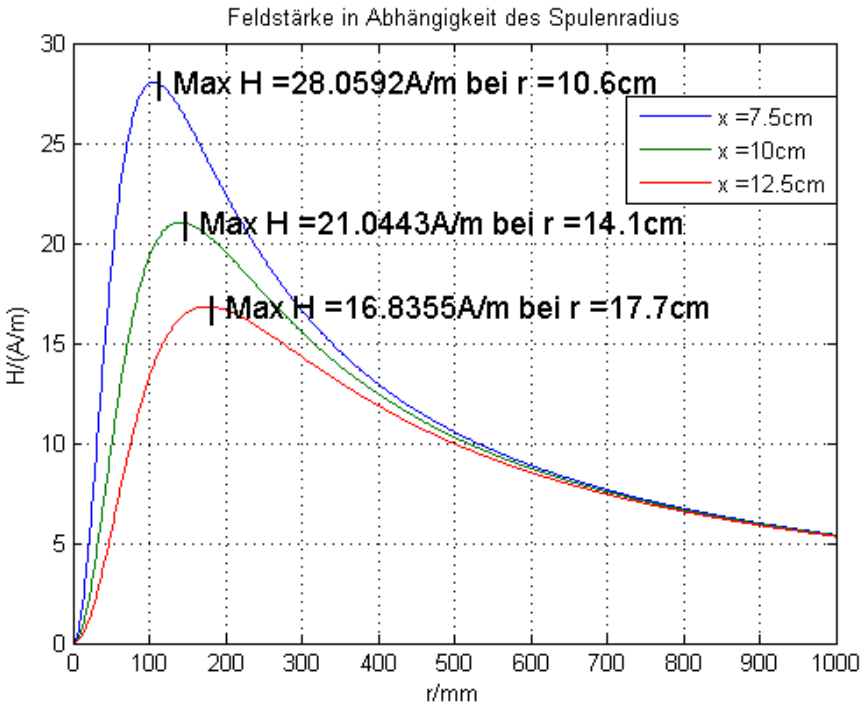

Bild 32: Feldstärke in Abhängigkeit vom Spulenradius für drei verschiedene Entfernungen zwischen Reader und Transponder

2.3.2 Modulation

Für den Datentransfer vom Reader zum Transponder wird das Readerfeld wie in den zuvor beschriebenen Abschnitten auf Basis der Amplitudenmodulation moduliert. Da die Amplitude beim On-Off-Keying zwischen voller und ausgeschalteter Amplitude umgeschaltet wird, kann auf eine analoge Modulationsschaltung verzichtet werden. Die Modulation wird mit Hilfe von zwei UND-Gattern erreicht, auf deren Eingänge das Taktsignal und das Modulationssignal gelegt werden. Am Ausgang des UND-Gatters ensteht so das modulierte Signal.
In den vorigen Unterabschnitten wurde der Reihenschwingkreis zur Vereinfachung mit einem idealen Rechteckgenerator angesteuert. Die Aufgabe der Schwingkreisansteuerung

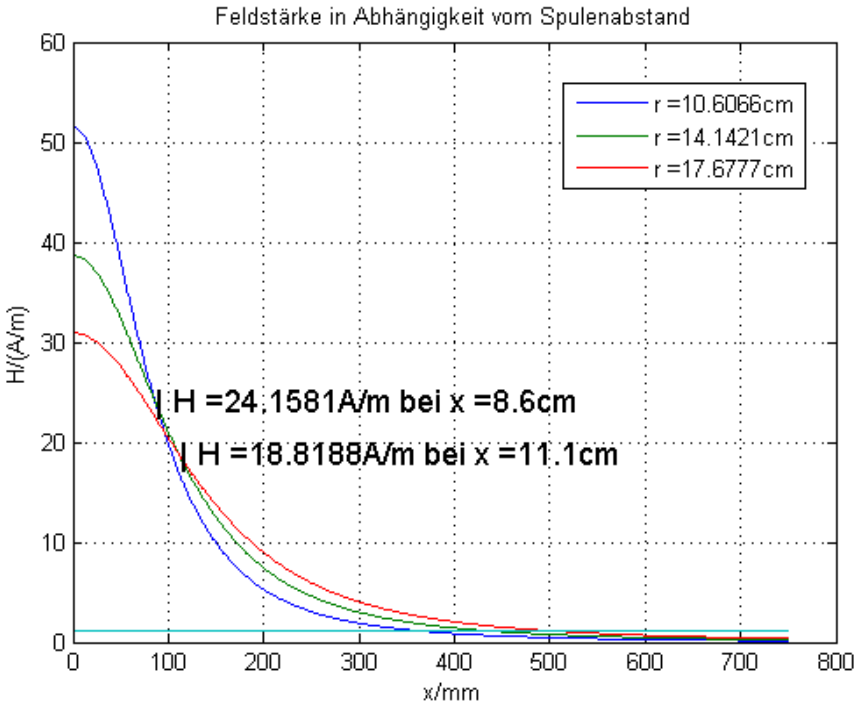

Bild 33: Feldstärke in Abhängigkeit vom Spulenabstand für drei verschiedene Readerspulenradien

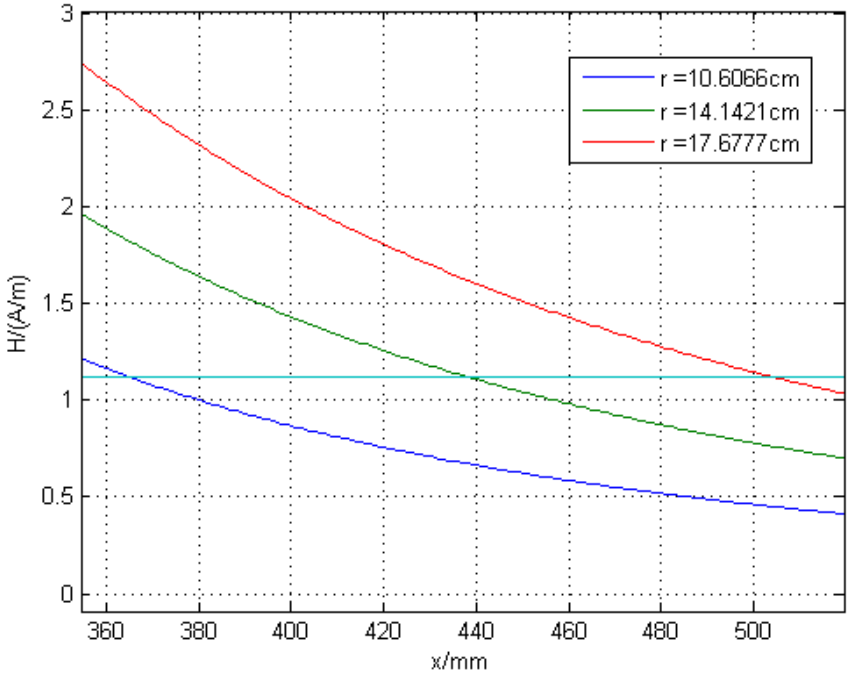

Bild 34: Feldstärke in Abhängigkeit vom Spulenabstand für drei verschiedene Readerspulenradien (vergrößert)

wird auf der Readerplatine von einem Mikrocontroller (*Mikrocontroller 2 Schwingkreisansteuerung*) übernommen. Dies bietet den Vorteil, dass die Ansteuerfrequenz leicht und sogar im laufenden Betrieb geändert werden kann. Ein Nachteil besteht in den leistungsschwachen Portausgängen des Mikrocontrollers, die nicht genügend Strom zur Verfügung stellen, um den Schwingkreis zu treiben. Auf Grund dieser Tatsache werden Treiberbausteine zwischen die Ausgänge vom Mikrocontroller und den Schwingkreisanschlüssen geschaltet. Damit eine höhere Leistung erzielt wird und für die Abstimmung des Schwingkreises im laufenden Betrieb, werden mehrere Treiberbausteine, die unterschiedliche Kapazitäten ansteuern, parallel eingesetzt. Für die Darstellung der Modulation werden die parallelen Treiberbausteine für das einfachere Verständnis zu einem Treiber und einer Kapazität zusammengefasst. Die Aufteilung in die einzelnen Treiberbausteine und Kapazitäten wird in dem Abschnitt zum Resonanzabgleich beschrieben. Das Prinzipschaltbild aus Abbildung 35 zeigt den grundlegenden Aufbau und die Signalverläufe der Modulationsschaltung. Das m-File zur Berechnung der Signalverläufe ist im Anhang A.1.3 zu finden.

Für die Treiber aus dem Block „Treiber-Baustein 1“ aus Abbildung 35 wird das Treiber-IC 74HC125N eingesetzt. Das IC besteht aus 4 Treibern mit 4 separaten Enable-Eingängen, so dass jeder Treiber einzeln gesteuert werden kann. Für die folgende PSpice-Simulation wird das IC 74HC125 eingesetzt, da das oben genannte IC nicht in der PSpice-Libary verfügbar ist. Die IC's 74HC125N und 74HC125 unterscheiden sich nur in dem invertierten Enable-Eingang. Für die Treiber aus dem Block „Treiber-Baustein 2“ wird das Treiber-IC 74HCT365N verwendet. Dieses IC hat einen gemeinsamen Enable-Eingang für 6 interne Treiber. Da bei diesem Treiber alle Stufen gemeinsam geschaltet werden, vermindert dies den Verdrahtungsaufwand. Die Abbildung 36 zeigt die Schaltung zur Simulation der Amplitudenmodulation im Reader. Die Abbildung 37 zeigt den Spannungsverlauf an der Schwingkreis-Spule. Die Spannung kann als Maß für die erzeugte Feldstärke des Readers genommen werden.

2.3.3 Demodulation

Die Demodulation im Reader zum Aufbereiten der vom Transponder gesendeten Daten geschieht über mehrere in Serie geschaltete Baugruppen. Das Blockschaltbild aus Abbildung 38 zeigt den prinzipiellen Ablauf der Demodulation.

Für die Auslegung der einzelnen Komponenten und für das bessere Verständnis der Amplitudenverläufe wurde die gesamte Amplitudendemodulation mit Hilfe von MATLAB Simulink nachgebildet. Zusätzlich dienen die Amplitudenverläufe und Spektren als Referenzen für die spätere Implementation der Schaltung. Die Trägerfrequenz ist in der Simulink-Simulation auf $f = 10 rad/s$ festgelegt. Die Modulation erfolgt mit einer Si-

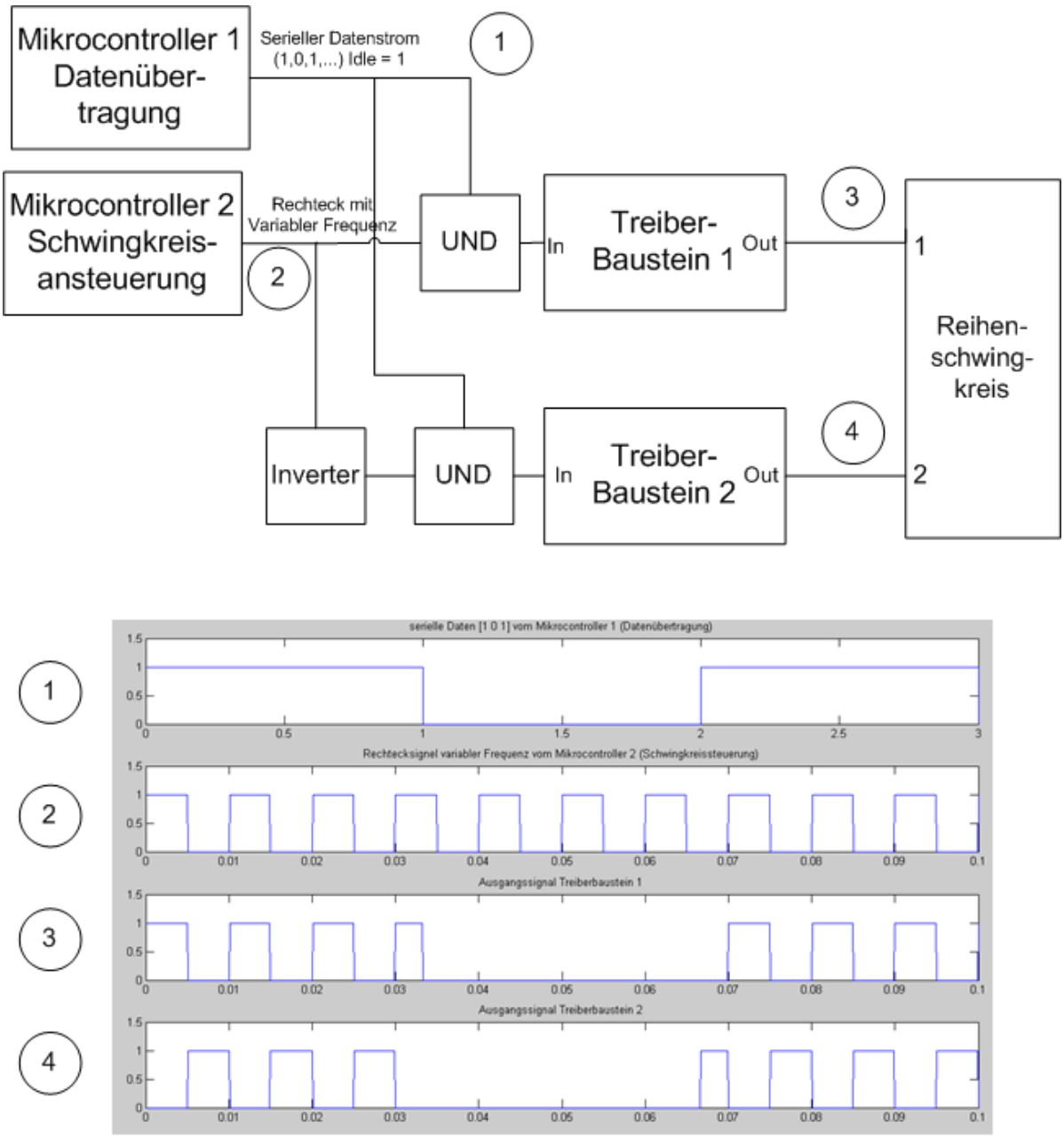

Bild 35: Prinzipschaltbild der Modulation des Readerfeldes

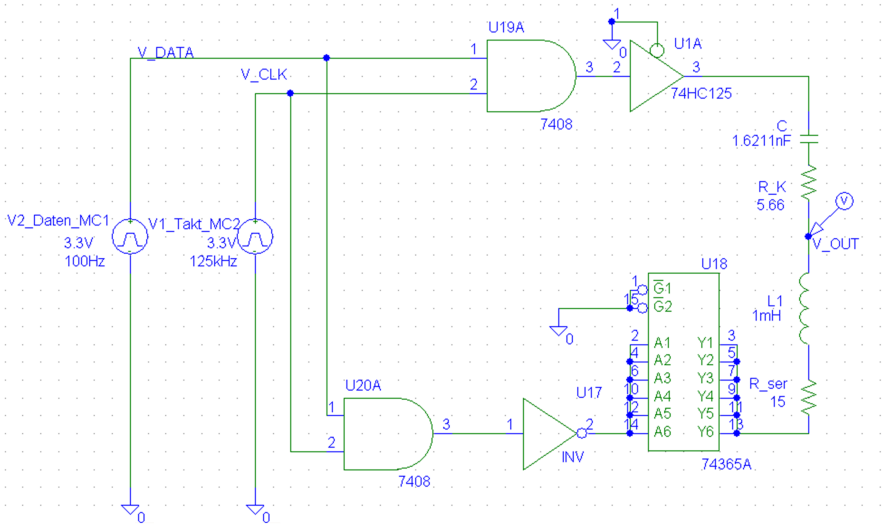

Bild 36: Simulationsschaltung für die Modulation des Readerfeldes

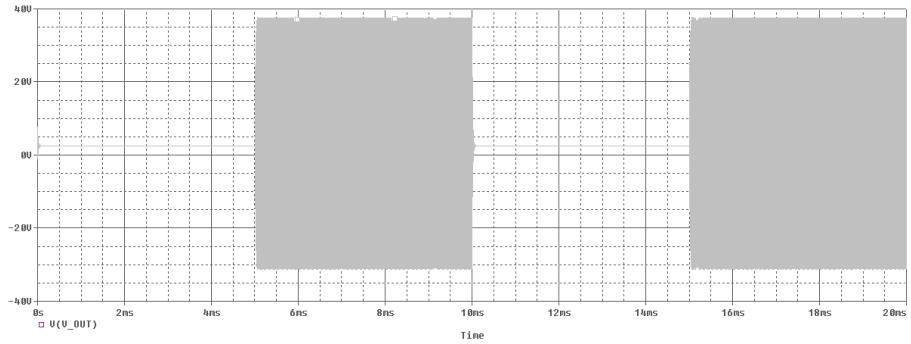

Bild 37: Spannungsverlauf an der Readerspule beim On-Off-Keying

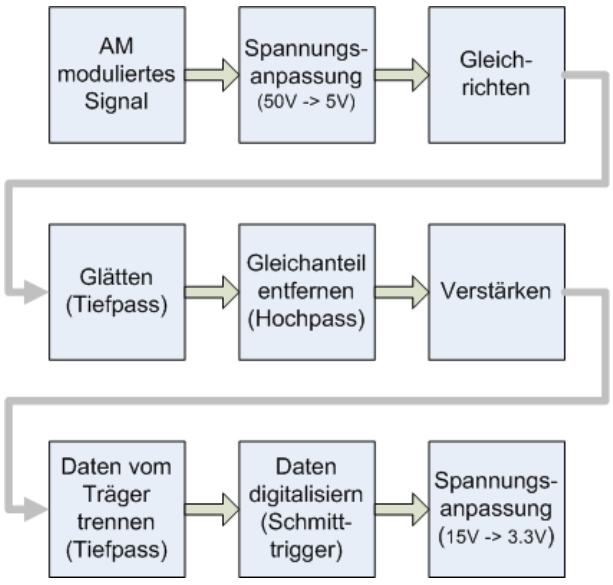

Bild 38: Blockschaltbild der Amplitudendemodulation im Reader

nusschwingung mit der Frequenz $f = 2rad/s$ anstatt mit einem Rechtecksignal. Diese Vereinfachung wurde auf Grund der leichteren Implementierung gewählt und hat grundsätzlich keine Auswirkung auf die Auslegung der Komponenten. Die Frequenzen für die Träger- und Modulationsschwingungen weichen auf Grund der besseren Darstellbarkeit von den tatsächlichen Frequenzen ab. Die Abweichung der Frequenzen hat keinen Einfluß auf die prinzipiellen Amplitudenverläufe. Die Simulinkschaltung ist in Abbildung 39 dargestellt.

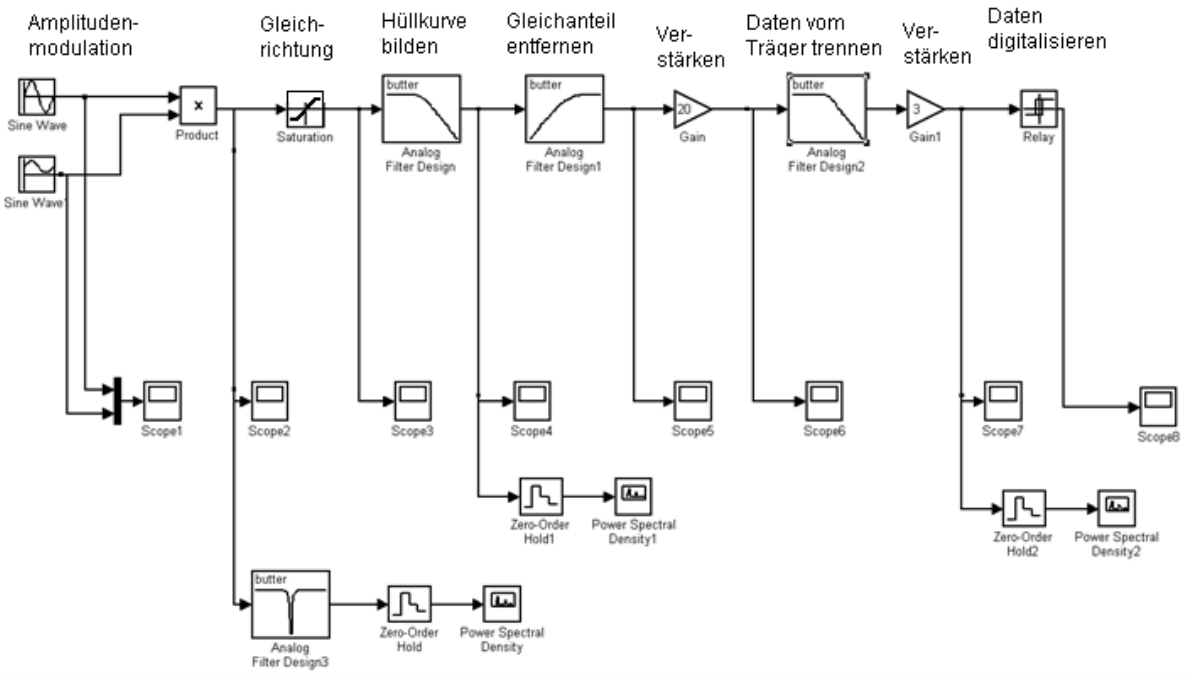

Bild 39: Simulink Schaltung zur Amplituden-Demodulation

Die Amplitudenverläufe der Oszilloskope 1 bis 8 sind in der Abbildung 40 dargestellt.

Scope 1:
Zeigt die sinusförmigen Amplitudenverläufe der Trägerschwingung (gelb) und der Modulationsspannung (lila).

Scope 2:
Zeigt das mit Hilfe des Multiplizierers erzeugte amplitudenmodulierte Signal. Der Modulationsgrad wurde mit 5% festgelegt. Bei dem Signal, dass im real aufgebauten Reader demoduliert wird, ist mit einem Modulationsgrad von ca. 0,1% zu rechnen. In der Simulation wurde der größere Modulationsgrad zur besseren Darstellung der Amplitudenverläufe gewählt. Im Verlauf der Simulation wurde aber auch ein Signal mit einem Modulationsgrad von 0.1% erfolgreich mit der Simulationsschaltung demoduliert.

Scope 3:
Zeigt das gleichgerichtete Signal. Die Gleichrichtung ist erforderlich, damit die anschließende Glättung der Spannung der Hüllkurve des Signals entspricht. Ohne die Gleich-

richtung würde sich die anschließende Glättung über den RC-Tiefpass zu 0 ergeben.

Scope 4:
Zeigt die obere Hüllkurve des amplitudenmodulierten Signals. Die Hüllkurve ist zusätzlich von der hochfrequenteren Trägerschwingung überlagert. Die Gleichrichtung gekoppelt mit dem nachgeschalteten RC-Tiefpass stellt einen allgemeinen Hüllkurvendetektor dar, der nach dem Prinzip des Spitzenwertmessers arbeitet. Für die Demodulation von amplitudenmodulierten Signalen mit einem größeren Modulationsgrad könnte die Demodulation hinter diesem Schaltungsteil abgeschlossen sein.

Scope 5:
Zeigt die obere Hüllkurve des amplitudenmodulierten Signals ohne Gleichanteil. Das heißt, dass die Modulationsschwingung (vom hochfrequenten Träger überlagert) um den Nullpunkt schwingt. Der Gleichanteil des Signals wird entfernt, damit das Signal verstärkt werden kann. Würde der Gleichanteil vor der Verstärkung nich rausgefiltert werden, würde hauptsächlich der Gleichanteil, der aus dem Träger entsteht (keine Information über die Daten) verstärkt werden. Zusätzlich wäre der Verstärkungsfaktor stark begrenzt, da die Spannung schnell in die Sättigung des Verstärkers gehen würde. Deshalb ist es zwingend notwendig, vor der Verstärkung den Gleichanteil mit Hilfe eines Hochpasses zu entfernen. Die Verzögerungszeit zu Beginn der Simulation, die der Hochpass benötigt, bis der Gleichanteil herausgefiltert ist, ist für den Betrieb nicht störend, da die Trägerschwingung schon lange Zeit anliegt, bevor Daten demoduliert werden müssen. So hat der Hochpass genügend Zeit, sich einzuschwingen.

Scope 6:
Zeigt das verstärkte Modulationssignal. Nach der Verstärkung ist die hochfrequente Überlagerung durch den Träger wieder stark zu erkennen.

Scope 7:
Zeigt das Signal hinter dem aktiven Tiefpass 2. Ordnung. Bis auf den Offset zu Beginn der Simulation entspricht es von der Frequenz dem Modulationssignal, dass die Daten darstellt. Wird als Modulationssignal an Stelle der Sinusschwingung ein Rechtecksignal verwendet, wie es bei der Datenübertragung der Fall ist, ist das Signal hinter dem Tiefpass trotzdem sinusförmig. Dies ist darauf zurückzuführen, dass die Tiefpässe die Vielfachen der Grundfrequnz von der Rechteckspannung herausfiltern und somit wieder ein Sinus entsteht.

Scope 8:
Zeigt das digitalisierte Signal, das dem rechteckförmigen Datenfluss entspricht. Die Umwandlung vom Sinus in das Rechtecksignal wird mit Hilfe eines Schmitttriggers vorgenommen. Die Schaltschwellen für den Schmitttrigger werden ober- und unterhalb von $0V$ gelegt. Der Grund dafür ist, dass der Schmitttrigger durchgehend schalten würde, wenn

keine Modulation des Feldes stattfinden würde und die Schaltschwelle auf $0V$ gelegt wäre.

Damit die Auswahl der Grenzfrequenzen der Filter besser nachvollzogen werden kann, werden die wichtigsten Signale im Frequenzbereich dargestellt. Zunächst soll aber anhand der theoretischen Rechnung gezeigt werden, wie sich die Spektralen Anteile bei der Amplitudenmodulation ergeben.
Für die folgende Rechnung werden die beiden Cosinusschwingungen als Träger- und Modulationssignal betrachtet.

$u_T(t) = 1V \cdot cos(10 \cdot \omega \cdot t)$
$u_M(t) = 0,95V + 0,05V \cdot cos(2 \cdot \omega \cdot t)$

$$\begin{aligned} u_{AM}(t) &= u_T(t) \cdot u_M(t) \\ u_{AM}(t) &= (1V \cdot cos(10 \cdot \omega \cdot t)) \cdot (0,95V + 0,05V \cdot cos(2 \cdot \omega \cdot t)) \\ u_{AM}(t) &= 0,95V \cdot cos(10 \cdot \omega \cdot t) + 0,05V \cdot cos(10 \cdot \omega \cdot t) \cdot cos(2 \cdot \omega \cdot t) \end{aligned} \tag{14}$$

Nach dem Additionstheorem: $cos(\alpha) \cdot cos(\beta) = \frac{1}{2} \cdot cos(\alpha - \beta) + \frac{1}{2} \cdot cos(\alpha + \beta)$ ergibt sich

$u_{AM}(t) = 0,95V \cdot cos(10 \cdot \omega \cdot t) + 0,025V \cdot cos(8 \cdot \omega \cdot t) + 0,025V \cdot cos(12 \cdot \omega \cdot t)$

In der Abbildung 41 ist das Spektrum zu dem amplitudenmodulierten Signal aus der MATLAB Simulink Simulation dargestellt. Es zeigt den durch die Rechnung erwarteten Verlauf. Der Träger liegt bei $10 \cdot \omega \cdot t$ und die beiden Seitenbänder liegen mit entsprechend niedrigerer Amplitude bei $8 \cdot \omega \cdot t$ und $12 \cdot \omega \cdot t$. Die Höhe der Amplituden entsprechen durch die vorgeschaltete Bandsperre nicht den realen Werten. Die Bandsperre wird in diesem Fall für die bessere Darstellung des Spektrums eingesetzt, da bei dieser einfachen Art der Spektrendarstellung keine logarithmische Einteilung der Spannungsachse möglich ist. Für das Verdeutlichen der spektralen Anteile ist dies aber nicht weiter störend.

Durch die Gleichrichtung teilen sich die spektralen Anteile in die Trägerfrequenz von $10 \cdot \omega \cdot t$ und die Modulationsfrequenz $2 \cdot \omega \cdot t$ auf. Die spektralen Anteile werden in der Abbildung 42 dargestellt. Die rechnerische Darstellung, die auf die Fourier-Reihenentwicklung zurückzuführen ist, wird an dieser Stelle nicht beschrieben. Stattdessen wird auf die gängige Literatur für Nachrichtentechnik verwiesen.
Anschließend kann die Modulationsschwingung durch den Hochpass und Tiefpass vom Träger getrennt werden. Die Abbildung 43 zeigt das Spektrum hinter dem 2. Tiefpass. Es kann erkannt werden, dass sich das Spektrum nur noch aus dem Anteil des Modulationssignals bei $2 \cdot \omega \cdot t$ zusammensetzt, und somit die Filterung abgeschlossen ist.

Die genauen Ausführungen und Berechnungen der Filter, des Gleichrichters und des

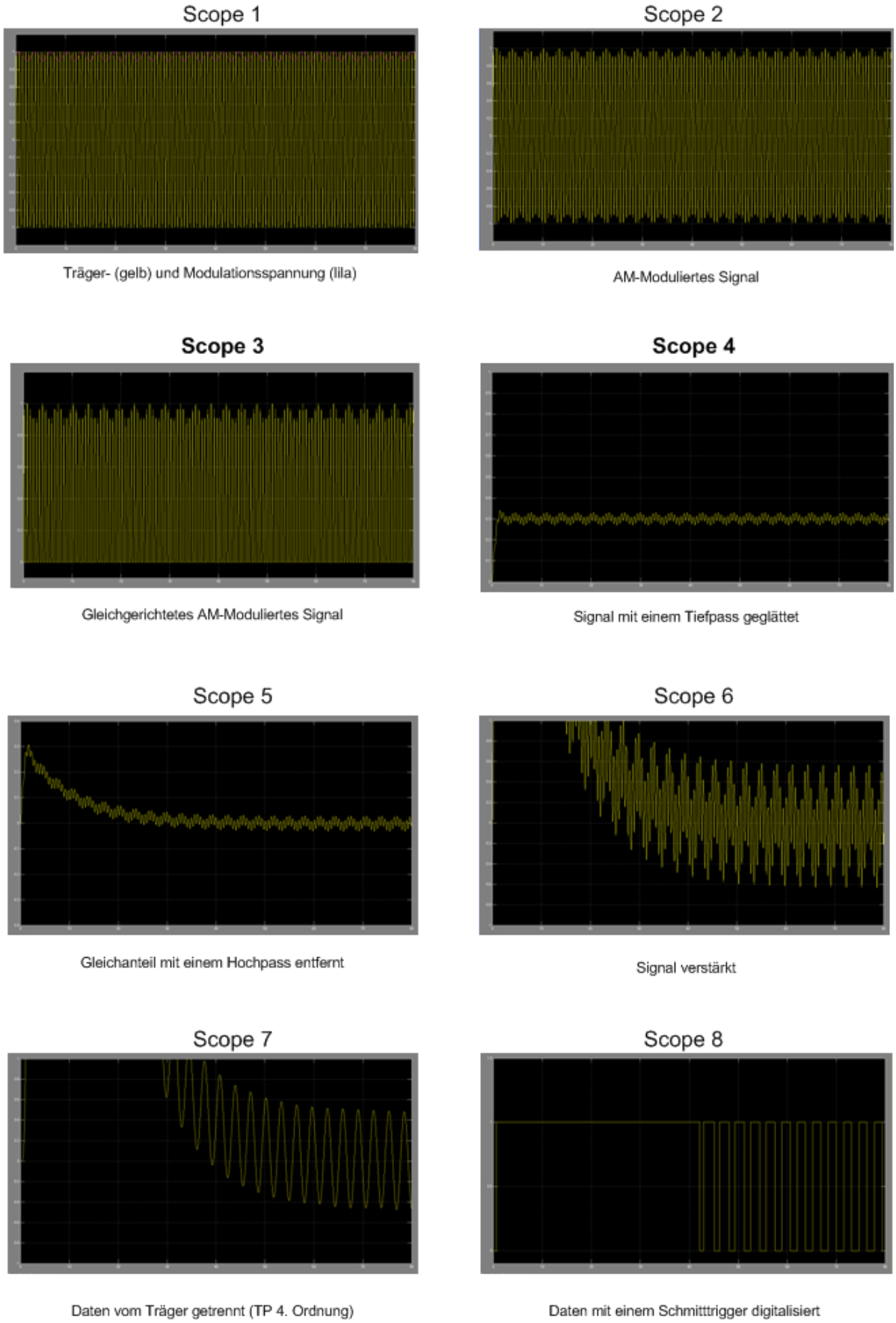

Bild 40: Scopebilder zur Simulink Amplituden-Demodulation

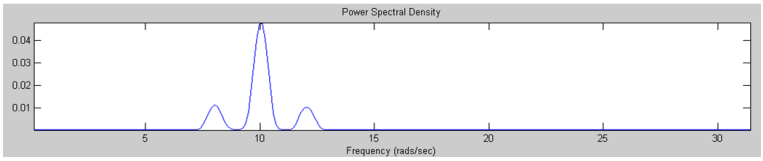

Bild 41: Spektrum der amplitudenmodulierten Spannung

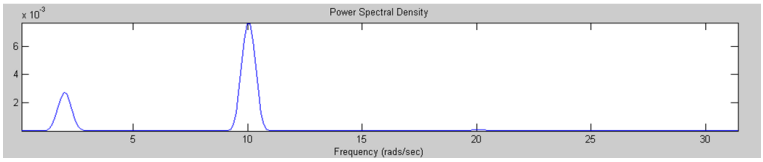

Bild 42: Spektrum der gleichgerichteten, amplitudenmodulierten Spannung

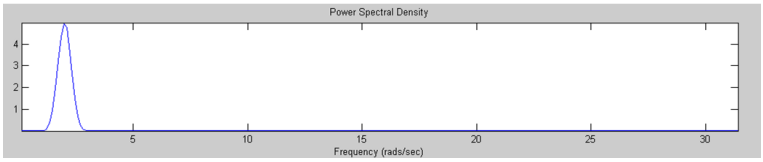

Bild 43: Spektrum des demodulierten Signals

Schmitttriggers werden in dem Kapitel zur Implementation des Readers vorgenommen. Dort werden die tatsächlichen Spektren und Amplitudenverläufe mit den hier simulierten verglichen und eventuelle Abweichungen diskutiert.

2.3.4 Automatischer Schwingkreisabgleich

Der automatische Schwingkreisabgleich wird vom Mikrocontroller (*Mikrocontroller 2: Schwingkreisabgleich*) durch Variation der Kapazität im Schwingkreis durchgeführt. Als Steuergrößen dienen dabei entweder die Spannung, die an der Spule anliegt, oder die Phasenlage an der Spule. Zusätzlich kann der Schwingkreis durch Veränderung der Anregefrequenz in Resonanz gebracht werden.

Abgleich durch Amplitudenmessung Für das erste Verfahren, bei dem die Spannungshöhe gemessen wird, wird die Messspannung an der Spule über einen geeigneten Spannungteiler heruntergeteilt und anschließend gleichgerichtet und geglättet. Die geglättete Gleichspannung wird über einen ADC-Port vom Mikrocontroller erfasst und bewertet. Beim Einschalten der Readerschaltung wird die Schwingkreiskapazität vom Mikrocontroller vom Minimalwert bis zum Maximalwert durchgeschaltet. Dabei wird für jede Kapazität die Messspannung aufgenommen und mit dem aktuellen Maximum verglichen. Anschließend wird die Kapazität eingestellt, bei der das Spannungsmaximum erreicht wurde. Der Abgleich ist auch im laufenden Betrieb möglich, jedoch muss wäh-

rend der Abgleichphase auf die Kommunikation verzichtet werden. Da diese Pausen im laufenden Betrieb zu Lücken in den Messreihen führen, wird für den Abgleich im laufenden Betrieb das oben genannte Verfahren mit Hilfe der Phasenmessung genutzt. Es ist mit dem Verfahren der Spannungsmessung zwar auch möglich, ohne Pausen in der Kommunikation die Kapazitätsanpassung zu steuern, indem mit kleinen Schritten um das aktuelle Spannungsmaximum ein neues Maximum gesucht wird. Dies führt aber zu ständigen Kapazitätsumschaltungen im Schwingkreis. Diese ständigen Kapazitätsänderungen würden wiederum Störungen in der Datendemodulation verursachen, weshalb dieses Verfahren nur zum Abgleich während des Einschaltvorgangs verwendet wird.
Die Abbildung 44 zeigt den Schaltplan der PSpice-Simulation für den Kapazitätsabgleich. Die Rechteckgeneratoren, die die Enable-Eingänge der Treiber ansteuern, stellen den Mikrocontroller dar, der die Kapazitäten durchschaltet. Der Amplitudenverlauf aus Abbildung 45 zeigt die Spulenspannung in Abhängigkeit der Schwingkreiskapazität, die als Maß für den Schwingkreisabgleich herangezogen wird.

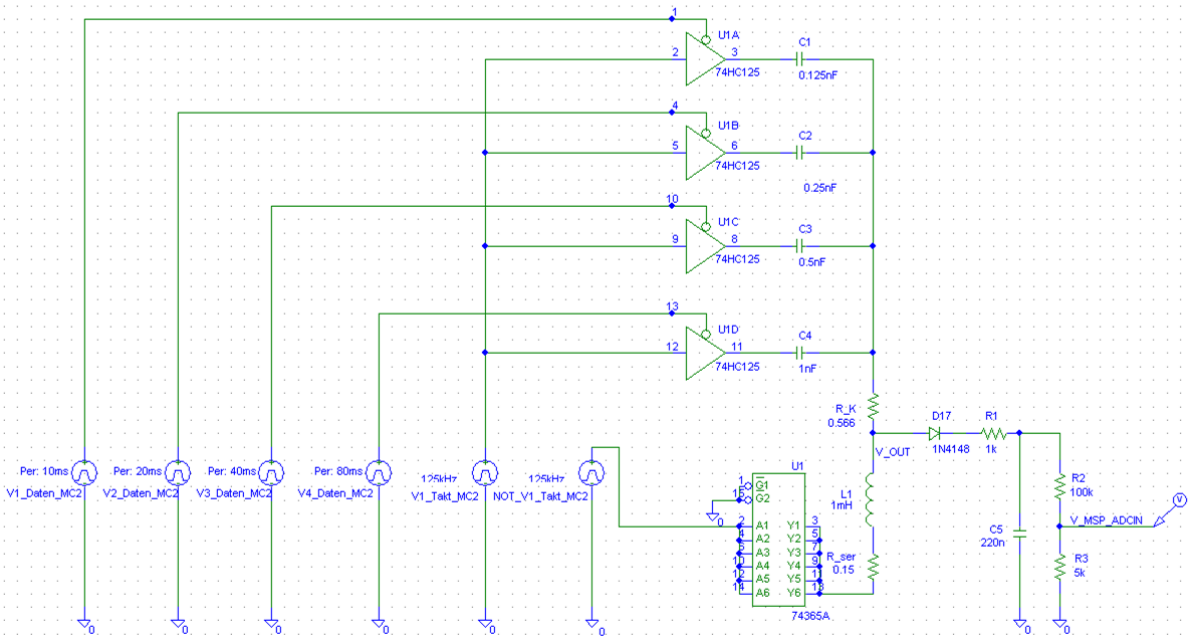

Bild 44: Simulationsschaltung des Schwingkreisabgleiches (amplitudengesteuert)

Die Größe der Kondensatoren ist so ausgelegt, dass sich der Wert zur vorigen Stufe verdoppelt. Der Kondensator aus der Stufe 0 hat die kleinste Kapazität $C_0 = 0.125nF$ und gibt somit die minimal mögliche Kapazitätsänderung vor. Die weiteren Kondensatoren berechnen sich nach $C_n = C_0 \cdot 2^n$. Dies hat den Vorteil, dass sich bei binärer Zählweise eine konstante Schrittweite ergibt. Somit kann von dem Zählstand des Portausganges direkt auf die Kapazität geschlossen werden $C = Z \cdot C_0$. In der Simulation werden vier Kapazitäten angesteuert. Damit ergeben sich 16 Kapazitäten im Bereich von $0nF$ bis $1.825nF$. Auf der Readerplatine wird der Abgleich mit 10 Stufen realisiert und somit ergeben sich 1024 mögliche Kapazitäten. Damit die einzelnen Umschaltungen der Kon-

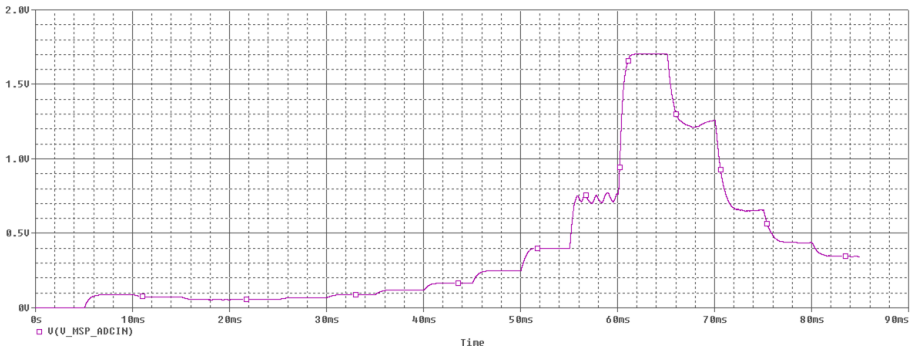

Bild 45: Spannungsverlauf beim amplitudengesteuerten Schwingkreisabgleich

C3	**C2**	**C1**	**C0**	C_{ges}
aus	aus	aus	aus	$0pF$
aus	aus	aus	ein	$125pF$
aus	aus	ein	aus	$250pF$
aus	aus	ein	ein	$375pF$
aus	ein	aus	aus	$500pF$
aus	ein	aus	ein	$625pF$
aus	ein	ein	aus	$750pF$
aus	ein	ein	ein	$875pF$
ein	aus	aus	aus	$1000pF$
ein	aus	aus	ein	$1125pF$
ein	aus	ein	aus	$1250pF$
ein	aus	ein	ein	$1375pF$
ein	ein	aus	aus	$1500pF$
ein	**ein**	**aus**	**ein**	**1625pF**
ein	ein	ein	aus	$1750pF$
ein	ein	ein	ein	$1875pF$

Tabelle 7: Kapazitätsbeschaltung mit resultierender Gesamtkapazität

densatoren genau zu erkennen sind, ist die Simulation mit vier Stufen besser geeignet und zeigt das Prinzip des Schwingkreisabgleiches. Die Tabelle 7 stellt die Ansteuerung der Kapazitäten und die daraus resultierende Gesamtkapazität dar.

Anhand des Spannungsverlaufes können die einzelnen Stufen der Kapazitätsumschaltung gesehen werden. Im unteren Bereich bis $30ms$ können die Kapazitätsveränderungen noch nicht erkannt werden, da der Schwingkreis zu weit von der Resonanz entfernt ist. Ab einer Zeit von $30ms$ werden die Stufen im $5ms$ Abstand gut sichtbar.
Die schlechte Stufenform ist auf den Gleichrichter in der Schaltung zurückzuführen. Es müssten größere Stufenbreiten für die Simulation gewählt werden, damit die Kapazität der Gleichrichterschaltung erhöht werden könnte. Dies ist auf Grund des beschränkten Arbeitsspeicher auf dem PC nicht möglich. Für die spätere Implementierung werden die Stufenbreiten verzehnfacht. Damit ist eine gut Gleichrichtung und somit eine gute Stufenform gewährleistet.
Die Amplitude steigt im Bereich bis $65ms$ an. Zu diesem Zeitpunkt ist der Zählstand, der die Kapazität darstellt, 13. Und somit berechnet sich die aktuelle Kapazität zu $C_{13} = 13 \cdot 0.125nF = 1,625nF$. Dieser Wert entspricht in etwa dem in den vorigen Kapiteln berechneten Kapazitätswert von $C_{Resonanz} = 1,6211nF$ für den auf Resonanz abgeglichenen Schwingkreis. Nachdem das Maximum durchschritten wurde, fällt die Amplitude für steigende Kapazitäten wie erwartet ab.

Abgleich durch Phasenmessung Für den Schwingkreisabgleich im laufenden Betrieb wird ein deutlich eleganteres Verfahren eingesetzt, welches den Phasenwinkel der Spulenspannung misst und bewertet. Hierzu wird die Tatsache genutzt, dass der Phasenwinkel bei einem abgeglichenen Schwinkreis genau $\varphi = 90°$ beträgt. Für zu große Kapazitäten in Bezug auf Resonanz ist der Phasenwinkel im Bereich von $0° < \varphi < 90°$ und für zu kleine Kapazitäten im Bereich $90° < \varphi < 180°$. Auf Grund dieser Tatsache ist es nicht nur möglich zu erkennen, ob der Schwingkreis abgeglichen ist, sondern auch noch zu wissen, ob die aktuelle Kapazität zu klein oder zu groß gewählt ist. Die Abbildung 46 zeigt die Phasengänge für die folgenden drei Kapazitäten:

$C_1 = 1,57nF$
$C_2 = 1,62nF$
$C_3 = 1,67nF.$

Damit der Phasenwinkel vom Mikrocontroller ausgewertet werden kann, wird die Spulenspannung über die Schaltung aus Abbildung 47 so aufbereitet, dass sie an einen Komparatoreingang des Mikrocontrollers angelegt werden kann. Dazu wird die Spannung zunächst über einen Spannungsteiler heruntergeteilt und anschließend mit Hilfe eines Summierers in den Arbeitspunkt $U_{3,3V}/2$ gehoben. Dadurch, dass der Summierer die

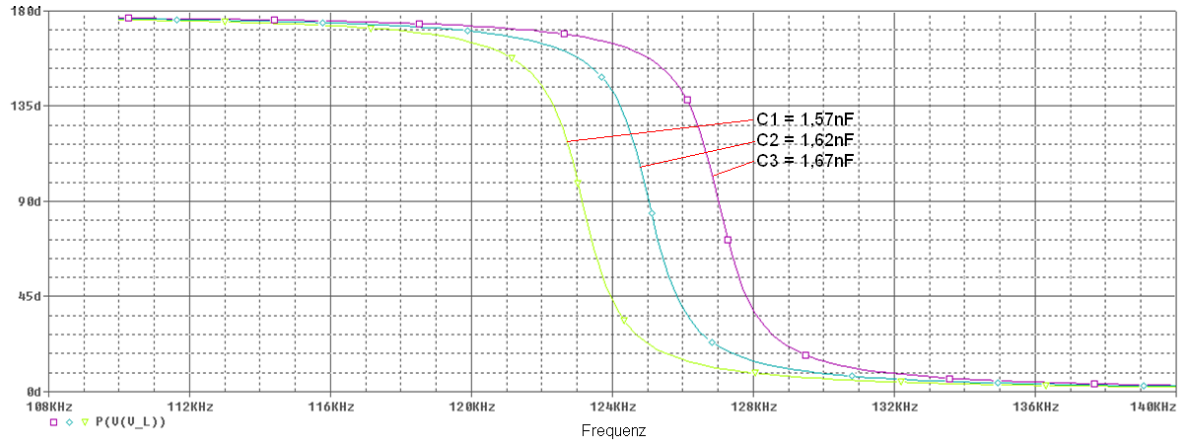

Bild 46: Phasengang vom Reihenschwingkreis

gleiche Spannung zum Anheben der Sinusschwingung benutzt, die auch als Versorgung für den Mikrocontroller dient und somit die Schaltschwelle des Komparators festlegt, ist die Schaltung störunanfällig gegenüber Schwankungen der Versorgungsspannung. Die Abbildung 48 zeigt die Amplitudenverläufe der Simulation.
Damit ein ständiges Umschalten der Kapazitäten um den Resonanzpunkt verhindert wird, ist es mit dem Abgleichverfahren über den Phasenwinkel möglich, einen Bereich festzulegen, in dem die Kapazität nicht weiter geändert wird und der Schwingkreis als abgeglichen gilt. Beispielsweise könnte der Mikrocontroller keine Kapatzitätsänderungen vornehmen, wenn der Phasenwinkel im Bereich $85° < \varphi < 95°$ liegt.

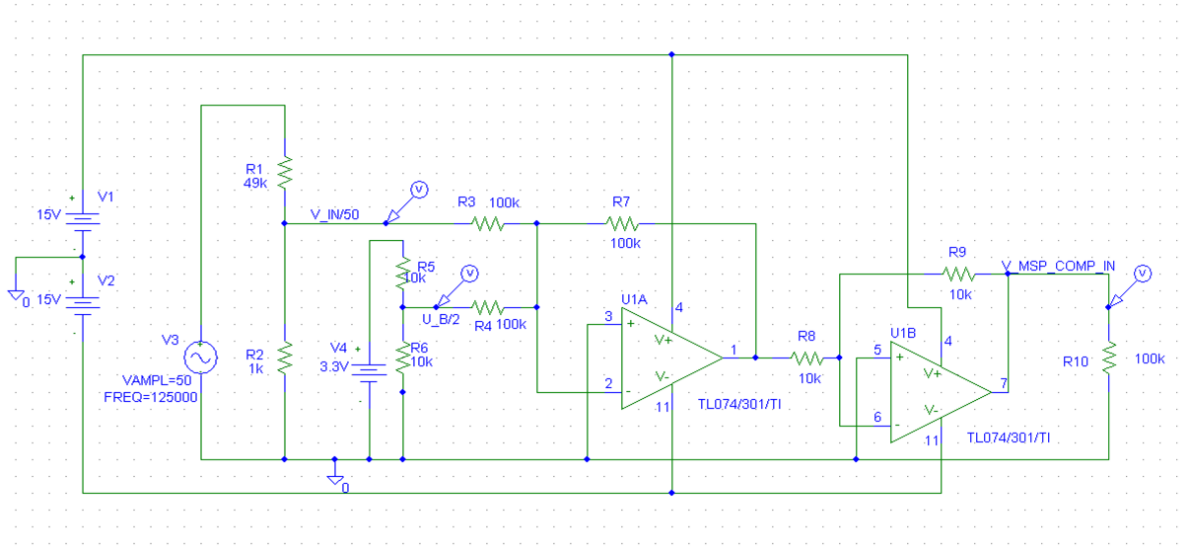

Bild 47: Schaltung zur Signalaufbereitung für die Messung des Phasenwinkels

Abgleich durch Frequenzanpassung Zusätzlich ist es möglich, den Schwingkreis über Veränderung der Anregefrequenz in Resonanz zu bringen. Dazu wird die vom Mikrocontroller erzeugte Rechteckspannung in der Frequenz variiert. Bei diesem Verfahren

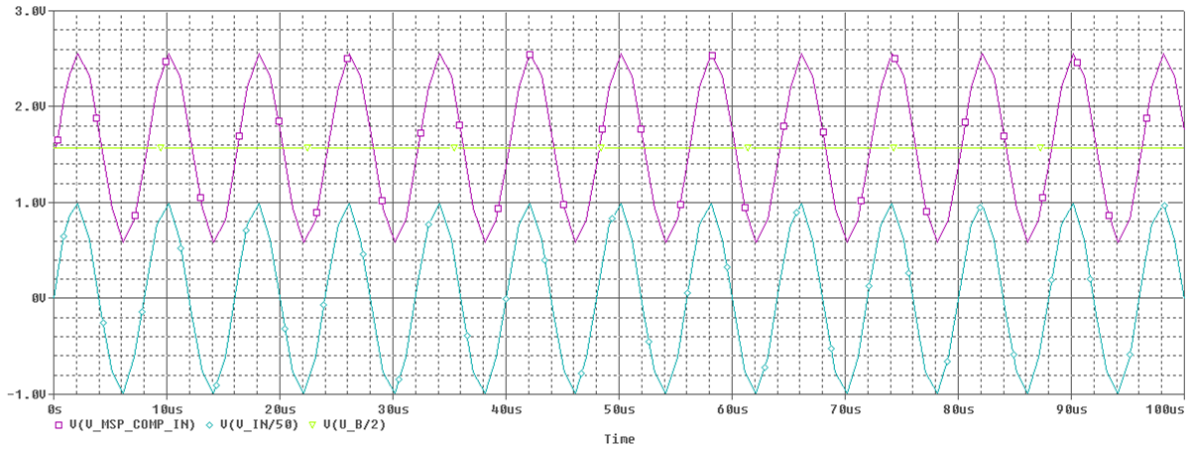

Bild 48: Aufbereitete Spulenspannung für den Komparatoreingang

verschiebt sich die Resonanzfrequenz. Dabei muss beachtet werden, dass die Resonanzfrequenz im Transponder gleich bleibt und eventuell nicht mehr mit der des Readers übereinstimmt.
Aber gerade dieser Effekt kann genutzt werden, um Transponder anzusprechen, deren Schwingkreis nicht auf die festgelegten $125kHz$ abgeglichen sind. Wird die Resonanzfrequenz in einem zuvor bestimmten Bereich variiert und zusätzlich das zuvor beschriebene Verfahren zum Schwingkreisabgleich durch die Bestimmung des Phasenwinkels eingesetzt, ist es möglich, Transponder mit abweichenden Resonanzfrequenzen anzusprechen. Es besteht zudem die Möglichkeit, die vom Transponder gesendete Versorgungsspannung auszuwerten und somit die Frequenz an die Resonanzfrequenz vom Transponder anzupassen. Durch den Einsatz dieses Verfahrens werden größere Fertigungstoleranzen für die Transponderherstellung erreicht, so dass ein Abgleich der Kapazität auf dem Transponder nicht mehr nötig ist und somit der Preis für die Herstellung sinkt.
Besonders deutlich wird das Problem der Frequenzabweichung bei einem schmalbandigen System. Die übertragene Energie, die am Transponder entsteht, bildet sich aus dem Integral aus dem Produkt zwischen dem Reader- und Transponderamplitudengang. Weichen die Resonanzfrequenzen von Transponder und Reader voneinander ab, verkleinert sich dieses Integral und somit verringert sich die übertragene Energie. Um dieses Problem zu verdeutlichen, wurden die folgenden mit MATLAB erzeugten Grafiken erstellt. In Abbildung 49 sind die Amplitudengänge und das dazugehörige Produkt für ein breitbandiges System dargestellt. Das linke Diagramm zeigt ein abgeglichenes System, bei dem die Resonanzfrequenzen vom Transponder und Reader übereinstimmen. Auf der rechten Seite ist ein System dargestellt, bei dem die Resonanzfrequenzen um $5kHz$ abweichen. Das gebildete Integral kann als Maß für die Energieübertragung herangezogen werden. Der absolute Wert hat keine Aussagekraft für die Energieübertragung. Jedoch kann aus dem Verhältnis zwischen dem Wert für das nicht abgeglichene System und dem für das abgeglichene System der relative Energieverlust berechnet werden. Für das breitbandige System ergibt sich hierfür ein Faktor von $\Delta E = \frac{158}{323} \approx 0,49$.

In der Abbildung 50 sind die entsprechenden Diagramme für das schmalbandige System dargestellt. Es ist gut zu sehen, dass die Frequenzabweichung einen größeren Einfluss auf das System hat. Für den Faktor, der die relative Energieübertragung zum abgeglichenen System darstellt, ergibt sich $\Delta E \approx 0,19$. Dies bedeutet, dass die übertragene Energie nur ca. $\frac{1}{5}$ der möglichen übertragenen Energie beträgt.

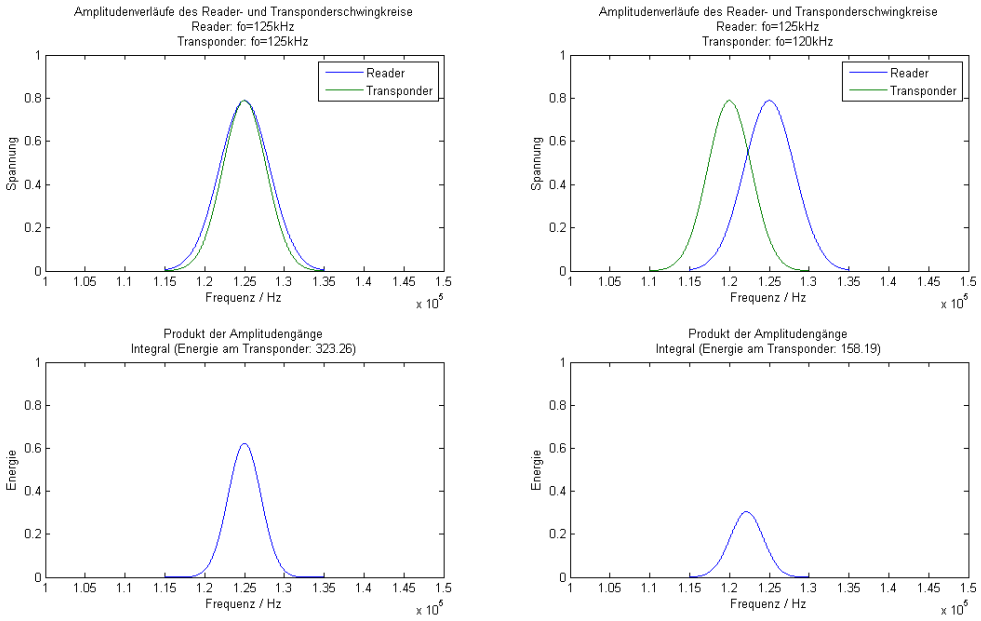

Bild 49: Amplitudenverläufe des breitbandigen Systems

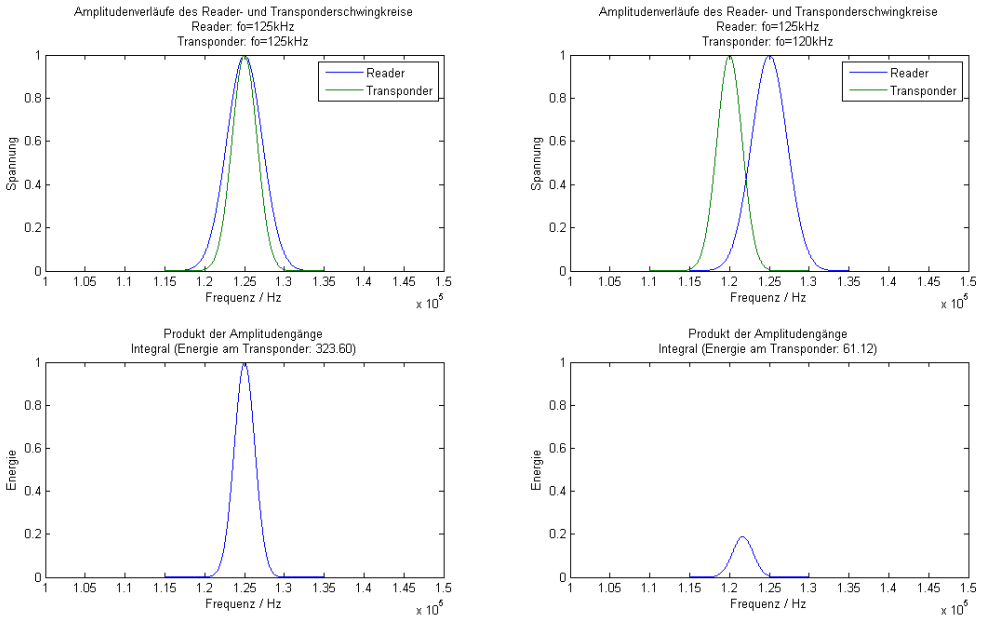

Bild 50: Amplitudenveräufe des schmalbandigen Systems

In der folgenden Abbildung 51 sind die Amplitudenverläufe und die entsprechenden Produkte für ein System dargestellt, bei dem die Readerresonanzfrequenz im Bereich von

$125kHz$ bis $130kHz$ variiert wird und die Transponderresonanzfrequenz bei $129kHz$ liegt. Die Berechnungen verdeutlichen den gewünschten Effekt, nicht abgeglichene Transponder im Feld anzusprechen. Für die drei Readerfrequenzen ergeben sich die Energiefaktoren $\Delta E = \frac{1}{200}$, $\Delta E = \frac{1}{2,9}$, $\Delta E = \frac{1}{1,07}$. Auf Grund der veränderten Frequenz, ist es in diesem Fall möglich, den Transponder mit der Resonanzfrequenz von $f_0 = 129kHz$ anzusprechen. Bei einer festen Readerfrequenz von $125kHz$ würde nur ca. $\frac{1}{3}$ der möglichen Energie zum Transponder übertragen werden. Bei diesem schlechten Übertragungsfaktor würde der Transponder wenn überhaupt nur für sehr kurzen Distanzen mit ausreichend Energie versorgt sein. Die mFiles für die Berechnungen sind im Anhang A.1.4 und A.1.5 zu finden

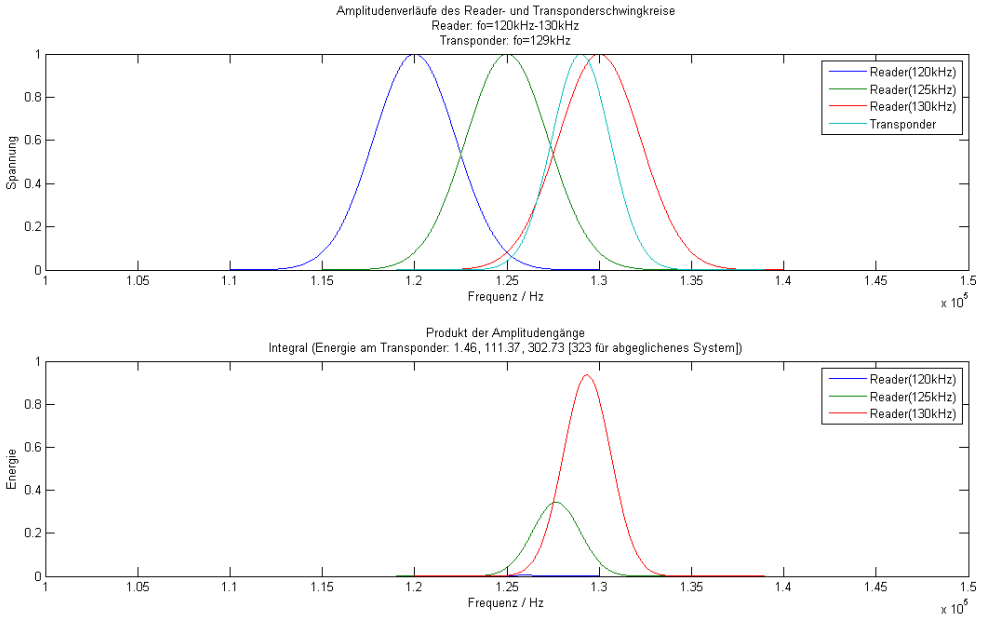

Bild 51: Amplitudenveräufe und Energieintegrale für variierte Readerfrequenzen

2.3.5 Digitalteil

Der Digitalteil des Readers besteht im wesentlichen aus zwei Mikrocontrollern. Es werden wie für den Transponder Controller aus der MSP430-Familie verwendet, die in der gleichen Entwicklungsumgebung programmiert werden wie der MSPF1232. Für die Kommunikationssteuerung und Datenverwaltung wird ein MSP430F169 auf einem Evaluierungsboard von Olimex eingesetzt. Der Schaltplan des Boards ist im Anhang B.3 dargestellt. Neben dem Mikrocontroller werden auf dem Board noch der externe Flash, die Pegelwandlung und das Display benutzt. Die Beschreibungen der einzelnen Baugruppen folgen in den Kapiteln zur Implementierung oder im kommentierten Quellcode.
Für die Feldsteuerung wird ein MSP430F169 eingesetzt, der auf einem Headerboard ohne weitere Baugruppen aufgelötet ist. Für die Kommunikation über die RS232 Schnittstelle

ist eine Pegelwandlung auf der Readerplatine vorhanden.
Die beiden Mikrocontroller laufen im Gegensatz zu dem Transpondercontroller mit einem externen Quartz mit einer Taktfrequenz von 8 MHz. Der MSP430F169 besitzt die gleiche Prozessorarchitektur wie der bereits beschriebene MSP430F1232. Der MSP430F169 ist aber mit deutlich mehr Peripherie ausgestattet.

Die für dieses Projekt eingesetzten Perephirien werden in dem Kapiteln zur Implementierung ausführlich beschrieben. Für das übrige System wird an dieser Stelle auf das Datenblatt [2] und den User Guide [3] verwiesen.

2.4 Datenübertragung Reader-Transponder

Die Datenübertragung zwischen Reader und Transponder geschieht bidirektional im Halbduplex-Betrieb. Das heißt, dass der Reader und der Transponder einen gemeinsamen Kanal für die Datenübertragung nutzen, so dass die Kommunikation wechselseitig stattfindet. Damit ein beidseitiges Senden und somit ein Fehler in beiden Datenübertragungen vermieden wird, ist das System in eine feste Netztopologie eingeteilt, die die Kommunikationsabläufe festlegt. Die Beschreibung vom Reader und den Transpondern in diesem Aufbau folgt in dem Unterabschnitt zur Netzwerktopologie.

2.4.1 Aufbau der Kommunikations-Protokolle

Für die Übertragung der Daten zwischen Transponder und Reader wurden zwei unterschiedliche Protokolle entworfen, da sich der Datenaufbau und die Datengröße stark unterscheiden. Die Daten vom Reader zum Transponder bestehen aus einer 4 Bit langen Adresse und einem 2 Bit breiten Befehl. Somit ergeben sich insgesamt 6 zu übertragende Bits. Die Datenmenge, die vom Reader zum Transponder gesendet wird, ist mit 48 Bits deutlich größer. Die 48 Bits setzten sich aus zwei 16 Bit langen Datenwörtern und einer 16 Bit langen CRC16-Checksumme zusammen. Zusätzlich zu den Nutzdaten sind noch weitere Bits für die Übertragung notwendig. Die Abbildung 52 zeigt die beiden beschriebenen Protokolle. Die grünen Blöcke stellen dabei die Nutzdaten da. Die Bits in den grauen Blöcken enthalten keine Information über die Nutzdaten. Diese Bits werden für die Verifizierung und den Kommunikationsablauf benötigt. Durch die Manchester-Codierung verdoppelt sich die Anzahl der benötigten Sendebits in Bezug auf die Datenbits. Die Beschreibung der Manchester-Codierung erfolgt im nächsten Unterabschnitt.
Das Protokoll für den Daten-Download besteht nach der Manchester-Codierung aus 121 Bits. Die Tabelle 8 zeigt die Funktionen der einzelnen Bitgruppen. Das Protokoll für den Daten-Upload besteht nach der Manchester-Codierung aus 18 Bits. Die Tabelle 9 zeigt

die Funktionen der einzelnen Bitgruppen.

Bild 52: Übertragungsprotokolle zwischen Reader und Transponder

2.4.2 Manchester Codierung

Damit die Daten auf den Träger aufmoduliert werden können, müssen sie zunächst codiert werden, um lange 1/0-Folgen zu verhindern. Würden die Daten direkt auf den Träger moduliert werden, müsste die untere Grenzfrequenz für die Datenübertragung sehr klein sein, um zu gewährleisten, dass auch lange 1/0-Folgen übertragen werden. Dies ist bei der Funkübertragung nicht realisierbar.
Für die Codierung wird das Manchester Verfahren eingesetzt, bei dem die Informationen über die Daten in den Flanken enthalten sind. Die Manchester-Codierung stellt damit eine Form der digitalen Phasenmodulation dar. Um ein Datenbit zu codieren werden 2 Bits benötigt. Infolgedessen sinkt die Daten-Bitrate auf die halbe Baudrate. Die Abbildung 53 zeigt die Signalverläufe von den codierten und nicht codierten Signalen. Dabei werden für die Manchester-Codierung die beiden möglichen Varianten gezeigt. Für dieses Projekt wird die Codierung nach G.E. Thomas vorgenommen, bei dem eine logische 1 durch die Bitreihenfolge 1 0 dargestellt wird. Die Beschreibung der Vorgehensweise bei der Codierung und der Decodierung erfolgt in den Kapiteln zur Software-Implementierung.

Bits	Bedeutung	Inhalt (Manchester-Codiert)
0 - 12	Run In	0x0ABD
13 - 20	Preamble	0x55
21 - 52	Daten 1	Temperaturwert
		(0 -> 0°C; 0xFFFF -> 200°C)
53 - 84	Daten 2	Betriebsspannung
		(0 -> 0V; 0xFFFF -> 5V)
85 - 116	Checksumme	CRC16 datenabhängig
117 - 120	Run Out	0x5

Tabelle 8: Bitgruppen des Daten-Download-Protokolls

Bits	Bedeutung	Inhalt (Manchester-Codiert)
0 - 1	Run In	0x1
2 - 9	Adresse	Transponder-Adresse
10 - 13	Befehl	Transponder-Befehl
14 - 17	Run Out	0x5

Tabelle 9: Bitgruppen des Daten-Upload-Protokolls

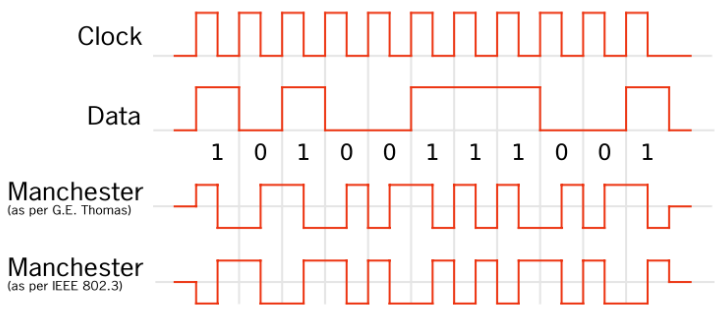

Bild 53: Signalverlauf bei der Manchester-Codierung

2.4.3 Netzwerktopologie

Für das Netzwerk aus mehreren Transpondern und einem Reader muss die Kommunikation nach einem festen Ablauf erfolgen, da die Kommunikation bidirektional über einen Kanal stattfindet. Da für dieses System keine Kommunikation zwischen den einzelnen Transpondern implementiert wird, sondern nur eine Kommunikation zwischen Reader und Transpondern möglich ist, wird ein sternförmiges Netzwerk gebildet, bei dem der Reader den Master und die Transponder die Slaves darstellen. Die Abbildung 54 zeigt den Aufbau des Netzwerkes.
Für dieses Projekt werden zwei Betriebsarten innerhalb des Netzwerks zur Verfügung gestellt. Bei der einen Betriebsart durchläuft der Reader einen festen Adressbereich und sendet die Adressen über den Kommunikationskanal an alle sich im Feld befindenden Transponder. Anschließend wird für eine zuvor festgelegte Antwortzeit auf das Antwortpaket vom angesprochenen Transponder gewartet. Sendet dieser keine Daten innerhalb dieser Zeit, wird das Anfragepaket verworfen und die nächste Adresse auf den Kanal gelegt. Dieses Verfahren entspricht dem Prinzip des zyklischen DPM1 Zugriffs beim Profibus. Zusätzlich zu dem zyklischen Abfragen können durch einen an dem Reader angeschlossenen PC über eine Steuersoftware azyklische Anfragebefehle mit einer vorgegebenen Adresse an den Reader gesendet werden. Diese Befehle unterbrechen den zyklischen Ablauf und legen die festgelegten Adressen auf den Bus. Anschließend läuft die zyklische Abfrage weiter. Dieses Verfahren entspricht dem Prinzip des DPM2 Zugriffs beim Profibus. Eine ausführliche Beschreibung der Zuzgriffsverfahren beim Profibus kann

dem Dokument [11] entnommen werden.

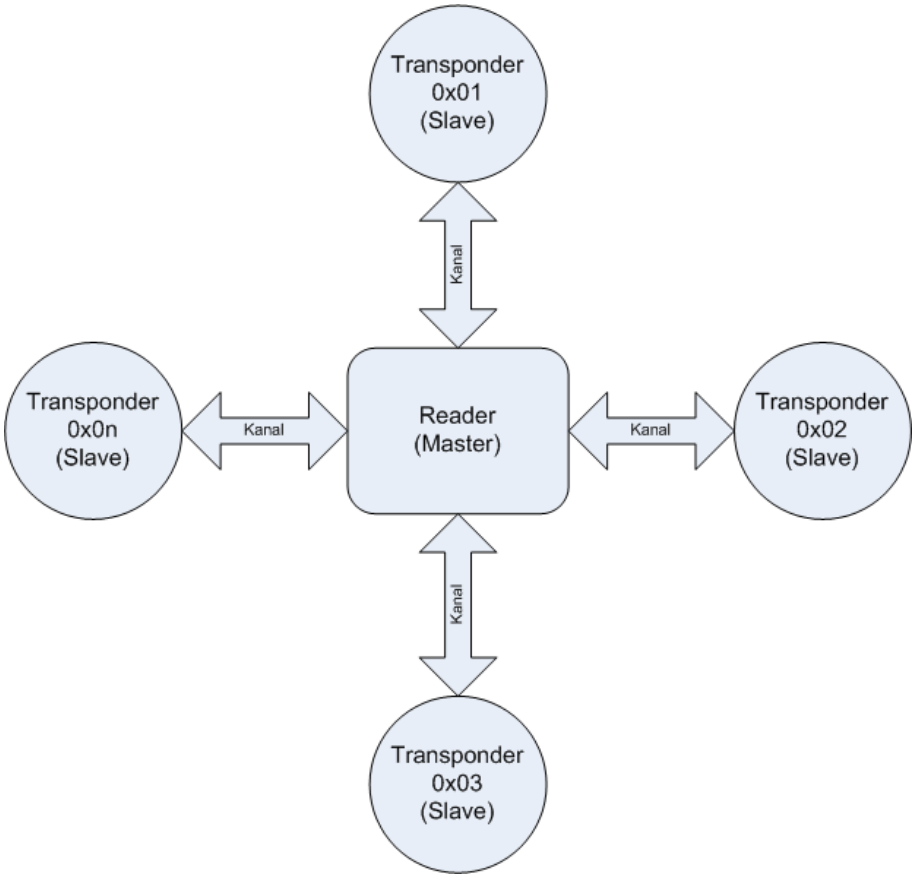

Bild 54: Sternförmiges Netzwerk zwischen Reader und Transpondern

3 Implementierung Transponder

In diesem Kapitel wird mit den Erkenntnissen aus dem vorigen Kapitel eine Transponderplatine mit zugehöriger Firmware erstellt. Bevor die endgültige Transponderplatine mit Hilfe des Cad-Programms Eagel erstellt wird, werden mehrere Prototypen auf Lochrasterplatinen entworfen, um die einzelnen Schaltungsbaugruppen zu testen. In diesem Zusammenhang werden nur die Schaltungen beschrieben, die auf der aktuellen Transponderplatine enthalten sind. Parallel zu der Entwicklung des Transponders werden einige Baugruppen des Readers auf Lochrasterplatinen aufgebaut, damit der Transponder getestet werden kann. Eine Beschreibung der Readerelemente folgt an diesen Stellen nicht, sondern es wird auf die Dokumentation zur Implementierung des Readers verwiesen.
Das Kapitel zum Transponder wird in zwei Unterkapitel eingeteilt. Im ersten Unterkapitel wird die Entwicklung der Hardware beschrieben und die Funktionalität anhand geeigneter Messung nachgewiesen und beschrieben. Das zweite Kapitel beschäftigt sich mit der Softwareimplementierung für den MSP430F1232 Mikrocontroller. Damit der Überblick für das Gesamtsystem nicht verloren geht, wird die Beschreibung der Firmware auf einem hohen Abstraktionslevel gehalten.

3.1 Hardware

Für die Entwicklung der Transponderhardware werden die in den vorigen Kapitel berechneten und simulierten Schaltungsteile in Hardware aufgebaut. Zunächst werden die Baugruppen separat getestet, bevor sie zusammengefügt werden. Da die Prinzipien und die Funktionsweisen im Kapitel zur Planung der Komponenten ausführlich beschrieben werden, wird an dieser Stelle das Hauptaugenmerk auf den Vergleich der simulierten Ergebnisse und der realen Ergebnisse gelegt. In diesem Zusammenhang werden Abweichungen diskutiert und für die weitere Einsatzmöglichkeit bewertet. Der aktuelle Schaltplan für den Transponder ist im Anhang B.1 dargestellt. Nachfolgend werden Messungen an den drei Hauptkomponenten im Transponder durchgeführt. Zum Ende des Abschnitts Hardware werden Messungen an der endgültigen Transponderplatine durchgeführt.

3.1.1 Spannungserzeugung

Um die im Parallelschwingkreis erzeugte Leistung in Bezug auf die Entfernung zur Readerspule zu ermitteln, werden die Spannungen für Entfernungen im Bereich von $0cm$ bis $50cm$ mit einem Lastwiderstand von $R_L = 100k\Omega$ aufgenommen. Die beiden Spulen sind bei der Messung parallel angeordnet und die Mittelpunkte der Spulen liegen auf einer gemeinsamen Achse. Die Messwerte und die daraus ermittelte Leistung sind in der

Tabelle 10 dargestellt. Zur Veranschaulichung sind die Messwerte in der Abbildung 55 grafisch dargestellt.
Aus der grafischen Darstellung der Messwerte kann erkannt werden, dass eine Last von $100k\Omega$ bis zu einer Entfernung von bis zu $18cm$ ausreichend mit Energie versorgt ist. In der Simulation ergab sich eine maximale Entfernung von $44cm$.
Zusätzlich zeigt das Diagramm den mehr als quadratischen Abfall der Leistung in Bezug auf den Spulenabstand. Die Kurve für die abfallende Leistung leitet sich aus dem Verlauf der Feldstärke in Abhängigkeit zum Spulenabstand her. Dieser wurde im Kapitel zur Planung des Readers simuliert und kann als Referenz verwendet werden.

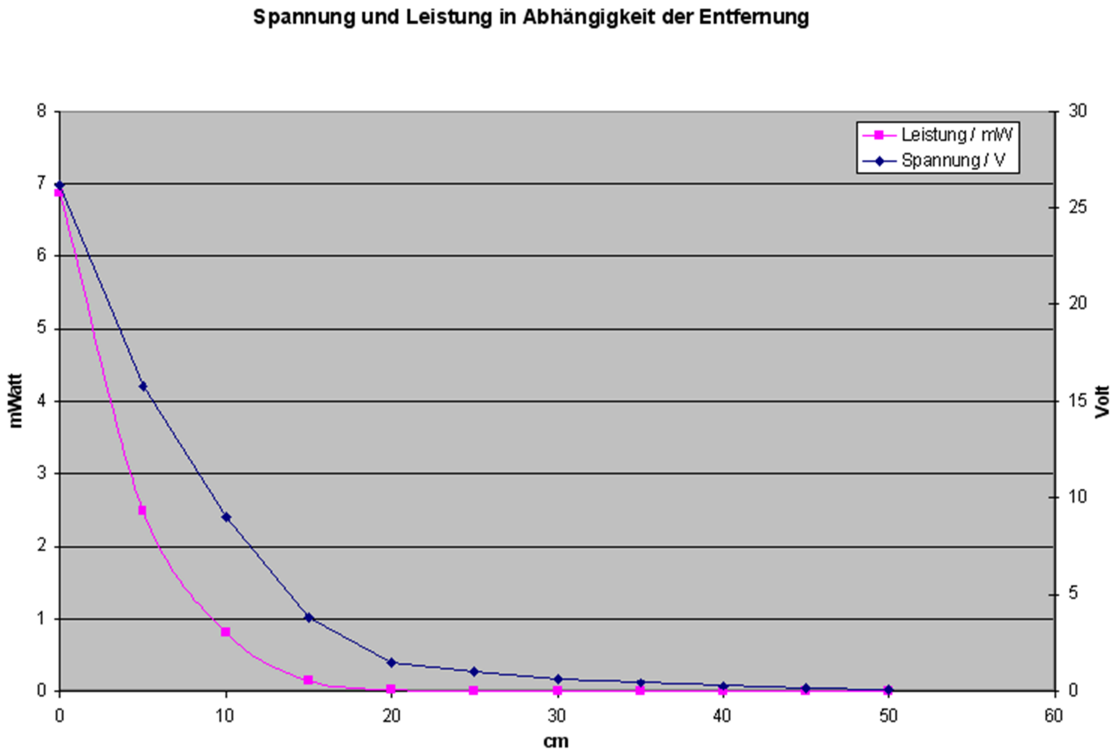

Bild 55: Darstellung der Spannung und der Leistung im Transponder-Frontend in Abhängigkeit zum Spulenabstand mit einer $100k\Omega$ Last

Damit die Funktionsweise des Gleichrichter gezeigt wird, werden für eine Entfernung von $10cm$ zwischen Reader- und Transponderspule die Spannungen am Schwingkreis und hinter dem Gleichrichter in dem folgenden Oszilloskopbild 56 gezeigt.

Entfernung/cm	0	5	10	15	20	25	30	35	40	45	50
Spannung/V	26,2	15,8	9	3,8	1,5	1	0,6	0,44	0,28	0,14	0,08
Leistung/ μ W	6900	2500	810	140	23	10	3,6	1,9	0,78	0,2	0,06

Tabelle 10: Spannung und Leistung am Transponder-Frontend in Abhängigkeit zur Entfernung mit einer $100k\Omega$ Last

An den Spannungsverläufen kann erkannt werden, dass die Ausgangsgleichspannung $U_A = 2,2V$ beträgt. Im optimalen Zustand würde die Ausgangsspannung des Gleichrichters auf $U_{Aopt} = 4,4V - 2 * 0,35V = 3,7V$ steigen. Auf Grund dieser Spannungen berechnet sich der Wirkungsgrad vom Gleichrichter zu $\eta = \frac{2,2V}{3,7V} = 0,6$.

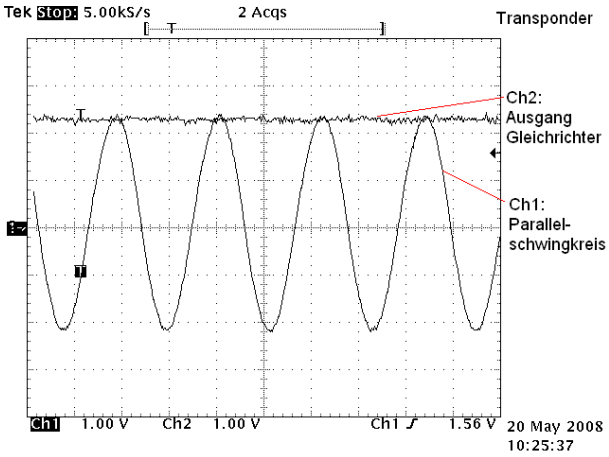

Bild 56: Spannung vor und hinter dem Gleichrichter im Transponder

3.1.2 Modulation

Die Modulation erfolgt wie beschrieben nach dem Prinzip der Amplitudenmodulation. Die Amplitude wird durch Lastmodulation moduliert. Dabei wird ein Lastwiderstand mit Hilfe eines Transistors im Takt des Datenstromes ein- und ausgeschaltet. Durch die Modulation entstehen zwei Seitenbänder zum Träger im Abstand von $\pm f_{mod}$. Die Theorie zur Entstehung der Seitenbänder ist im Abschnitt zur Planung des Transponders beschrieben. Die Größe des Lastwiderstandes, der zur Modulation verwendet wird, wird empirisch ermittelt. Dafür wird der Lastwiderstand verändert und die dazugehörigen Spektren bewertet. Die größten Seitenbänder ergeben sich bei einem Modulationswiderstand von $R_{mod} = 10k\Omega$. Das dazugehörige Spektrum, das am Transponderschwingkreis aufgenommen wurde, ist in Abbildung 57 dargestellt. Aus dem Verhältnis zwischen der Höhe der Seitenbänder und des Trägers wird der Modulationsgrad ermittelt. Für einen Modulationswiderstand von $Rmod = 10k\Omega$ ergibt sich ein Modulationsgrad von $m = \frac{1}{1000} = 0,1\%$. Dies entspricht einer Dämpfung von $60dB$. Der serielle Datenstrom, mit dem der Transponder den Lastwiderstand steuert, wird bei der Demodulation im Reader vom Träger getrennt und im Mikrocontroller in die einzelnen Datenblöcke unterteilt.

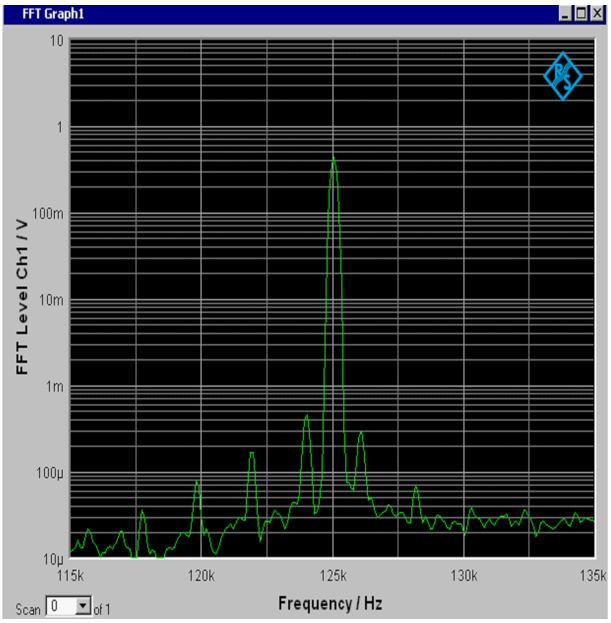

Bild 57: Spektrum am Transponderschwingkreis bei Modulation

3.1.3 Demodulation

Die Demodulation der vom Reader gesendeten Daten erfolgt mit einem RC-Tiefpass erster Ordnung mit einer Grenzfrequenz von ca. $f = 16kHz$. Dadurch werden die Daten ($100Hz$) gut vom Träger ($125kHz$) getrennt. Am Ausgang des Tiefpasses werden 6 Dioden in Reihe gegen Masse geschaltet, damit die Ausgangsspannung am Tiefpass nicht über die zulässige Eingangsspannung des Mikrocontrollers steigen kann. Da die Modulation im Reader mit On-Off-Keying vorgenommen wird, reicht für die Demodulation der RC-Tiefpass aus, so dass der Ausgang des Tiefpasses ohne weitere Signalaufbereitung direkt mit dem Dateneingang des Mikrocontrollers verbunden werden kann. Das Oszilloskopbild aus Abbildung 58 zeigt den Amplitudenverlauf der Spannung nach der Demodulation. Da der Idle-Zustand 1 ist, müssen die Pegel für die Interpretation der Daten negiert werden. Zusätzlich ist in dem Oszilloskopbild eine Beschreibung der Datenbits vorgenommen. Vor der gesendeten Adresse kann die Start 1 (10) erkannt werden. Am Ende des Protokolls werden 2 Stop Einsen (1010) gesendet. Die Länge eines gesamten Protokolls beträgt $86,4ms$.

3.1.4 Transponder

Damit die Funktionsweise des Transponders gezeigt wird, werden abschließend zum Abschnitt der Hardwarebeschreibung Messungen an der endgültigen Transponderplatine durchgeführt. Die Abbildungen 59 und 60 zeigen die Fotos der bestückten Vorder- und Rückseite Transponderplatine. Bei der aktuellen Platine ist es möglich, die Größe auf

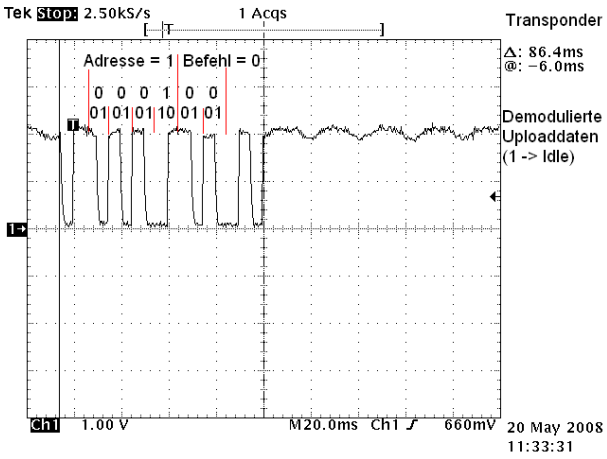

Bild 58: Amplitudenverläufe bei der Demodulation im Transponder

ein Drittel zu reduzieren. Dadurch würden sich die Abmaße auf ca. $2cm * 2,5cm$ reduzieren. Die erstellte Transponderplatine wird zunächst so groß entworfen, damit eventuelle Umbauten leichter zu handhaben sind.

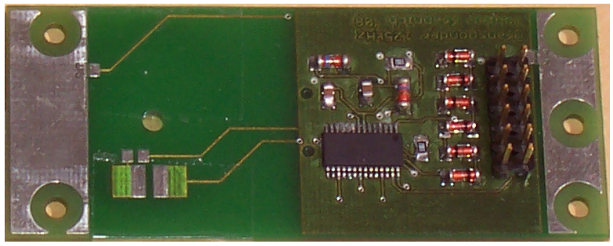

Bild 59: Foto von der bestückten Transponderplatine (Vorderseite)

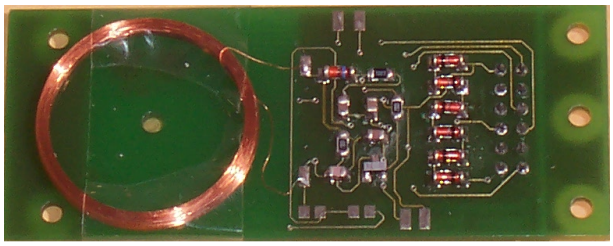

Bild 60: Foto von der bestückten Transponderplatine (Rückseite)

Damit eine Aussage über die mögliche Entfernung zwischen Readerspule und Transponderspule in Bezug auf die ausreichende Energieversorgung getroffen werden kann, wird die Messung aus dem Unterabschnitt zur Spannungserzeugung, bei der der Abstand zwischen Reader und Transponder variiert, mit der Transponderplatine wiederholt. Damit die Verluste in der Gleichrichtung berücksichtigt werden, wird die Spannung hinter

dem Gleichrichter gemessen, also die Versorgungsspannung des MSP430F1232. Zusätzlich wird notiert, ob der Mikrocontroller für die eingestellte Entfernung ordnungsgemäß arbeitet.
In der Tabelle 11 sind die Spannungen in Bezug auf die Entfernungen zwischen Readerspule und Transponderspule aufgenommen. Zusätzlich ist vermerkt, ob der Controller richtig arbeitet, also mit ausreichend Energie versorgt ist und ob die Datenübertragung in Richtung Reader fehlerfrei funktioniert.

Entfernung/cm	0	5	10	15	20	25	30	35	40	45	50
Spannung/V	3,46	3,26	2,82	2,18	1,5	1,34	1,06	0,78	0,54	0,26	0,22
μC arbeitet	ja	ja	ja	ja	wechselnd	nein	nein	nein	nein	nein	nein
Datenübetr. erfolgreich	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein

Tabelle 11: Transponderparameter in Abhängigkeit zur Entfernung zur Readerspule

Bei der Auswertung der Daten aus Tabelle 11 wird deutlich, dass die maximal erreichbare Arbeitsentfernung durch die Datenübertragung beeinträchtigt wird.
Bei geringen Entfernungen bis ca. $10cm$ wird die Spannung durch die in Reihe geschalteten Dioden zum Schutz des Digitalteils begrenzt. Ab einer Entfernung von $20cm$ nimmt die Spannung nur noch langsam ab. Dies liegt daran, dass der Mikrocontroller nicht mehr läuft und sich somit der Ersatzwiderstand für diesen erhöht.

3.2 Software

Die Software für den Transponder ist modular aufgebaut. Bei der Softwareentwicklung werden die Module zunächst einzeln erstellt und getestet. Teile der Software, bei denen die Hardwareplattform keine Rolle spielt, werden zunächst im Visual Studio geschrieben und getestet. Bei der Erstellung der Softwareroutinen muss darauf geachtet werden, dass der Mikrocontroller nur sehr begrenzten Speicherplatz zur Verfügung stellt. Trotz der geringen Speicherkapazität wird bei der Entwicklung der Transponder-Software für das Experimentalsystem Wert darauf gelegt, dass der Funktionsablauf klar und einfach nachzuvollziehen ist, damit ein leichter Einstieg in die Funktionsweise des Transponders gegeben ist. Für die Weiterführung des Projektes gibt es viel Spielraum zur Komprimierung des Codes.
An die Software für das Experimentalsystem werden die folgenden stichpunktartig zusammengefassten Anforderungen gestellt.

- Gesamte Software muss mit geringem Systemtakt arbeiten
- Empfang und Decodierung von seriellen Datenströmen

- Umgebungstemperatur ermitteln
- Betriebsspannung ermitteln
- Externe Sensorspannung bestimmen
- Verifizierungsdaten für die Sendedaten erstellen
- Codierung der Sendedaten
- Serielle Ausgabe der Daten

Die Abbildung 61 zeigt den Programmablaufplan für die Transpondersoftware. Das Programm besitzt nur zwei Abzweigungen. Ansonsten läuft das Programm in einem festen Ablauf innerhalb der Hauptschleife ab. Für zukünftige Weiterentwicklungen können weitere Betriebsmodi erstellt werden, wie sie, später zu sehen, in der Readersoftware implementiert sind. Zur Zeit wird ein 2 Bit Kommando vom Reader an den Transponder gesendet und vom Transponder empfangen, aber nicht weiterverwendet. Dieses Kommando könnte zum Umschalten der Betriebsmodi genutzt werden.
Im weiteren Verlauf der Softwarebeschreibung wird das Hauptaugenmerk auf den Datenempfang gerichtet, da dieser den kompliziertesten Teil der Transpondersoftware darstellt. Für die Beschreibung der übrigen Softwarekomponenten wird auf den kommentierten Quellcode im Anhang A.2 verwiesen.

Datenempfang Die vom Reader gesendeten Daten werden analog demoduliert. Anschließend werden sie über einen Spannungsüberhöhungsschutz an einen Port des Mikrocontrollers gelegt. Als Eingang sollte ein Timer Input Capture Eingang verwendet werden. Aber auf Grund eines Fehlers beim Leiterplattendesign ist die Datenleitung an einen normalen Port gelegt worden, so dass die Daten über einen Portinterrupthandler eingelesen werden müssen. Auf die Funktionalität des Einlesevorganges hat dies keinen Einfluss. Jedoch wird mit der Timer Input Capture Methode der Quellcode vereinfacht und die Genauigkeit erhöht. Deshalb sollte bei der nächsten Transponderversion die Datenleitung auf einen Timereingang umgelegt werden.
Das Einlesen und Decodieren der Daten wird komplett im Interrupthandler des Portinterrupts abgehandelt. Der Interrupt wird bei einer steigenden Flanke aktiviert. Die lange Ausführungszeit des Interrupthandlers stört in diesem Fall nicht, da keine weiteren Aufgaben parallel verarbeitet werden müssen und keine weiteren asynchronen Ereignisse auftreten. Nach dem Empfang eines kompletten Datensatzes wird das Empfangen gesperrt, indem der Interrupt ausgeschaltet wird. Anschließend läuft die Datenverarbeitung, Messwertaufnahme, Codierung und das Senden der Daten. Erst nach dem Senden der Daten wird das Empfangen wieder freigegeben. Daraufhin wartet das Hauptprogramm, bis ein weiterer Datensatz empfangen wird.

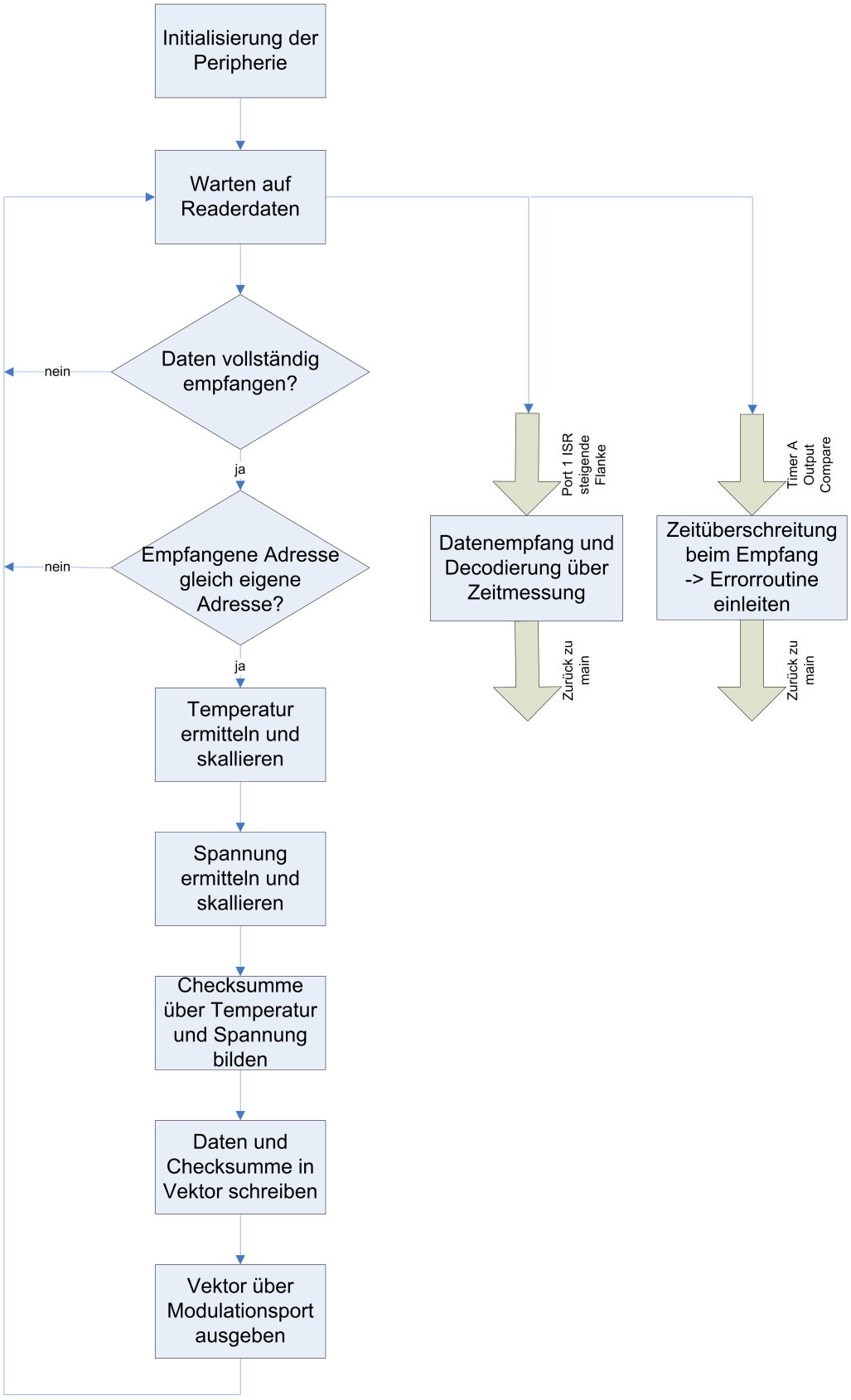

Bild 61: Programmablaufplan der Transpondersoftware

Für die Decodierung beim Einlesen der Daten wird die Zeit zwischen zwei steigenden Flanken bestimmt. Da der Reader die Daten mit einem festgelegten genauen Takt moduliert, kann auf ein Preamble verzichtet werden. Die Run-In-Sequenz wird zu einer Starteins zusammengefasst, da keine Störungen auf dem Übertragungskanal in Richtung Transponder auftreten. Die Starteins ist notwendig, um den Empfang zu starten. Zusätzlich wird bei der hier entwickelten Decodierungsmethode die Kenntnis benötigt, ob das erste Zeichen eine 0 oder eine 1 ist, um die weiteren Daten zu decodieren.
Nachdem die Starteins erkannt wurde, werden die Nutzdaten eingelesen. Dafür wird mit Hilfe eines Timers die Zeit zwischen der letzten und der aktuellen steigenden Flanke gemessen und einem von fünf Zeitintervallen zugeordnet. Die Zeitintervalle werden vor der Laufzeit mit der Erkenntnis der Modualtionsfrequenz im Reader ($100Hz \rightarrow 10ms$) festgelegt. Zusätzlich werden die Zeitintervalle an die Manchastercodierung angepasst, bei der drei verschieden Zeiten zwischen zwei steigenden Flanken möglich sind. Die Abbildung 62 zeigt die fünf Zeitintervalle und deren Bedeutung. Zusätzlich wird die Demodulation an einem Beispielframe, bei dem die Transponderadresse 11 und das Kommando 0 ist, gezeigt.

Intervall	Zeit	Beschreibung	Reaktion
I	t < 7ms	zu kurz	Fehlersequenz einleiten
II	7ms < t < 12,5ms	kurze Zeit	Bit[i] = H_Bit; i++
III	12,5ms < t < 17,5ms	mittlere Zeit	**H_Bit == 0** -> Bit[i] = H_Bit; i++; H_Bit^1
			H_Bit == 1 -> Bit[i] = H_Bit; i++; H_Bit^1; Bit[i] = H_Bit; i++
IV	17,5ms < t < 22,5ms	lange Zeit	Bit[i] = H_Bit; i++; H_Bit^1; Bit[i] = H_Bit; i++; H_Bit^1
V	t > 22,5ms	zu lang	Fehlersequenz einleiten

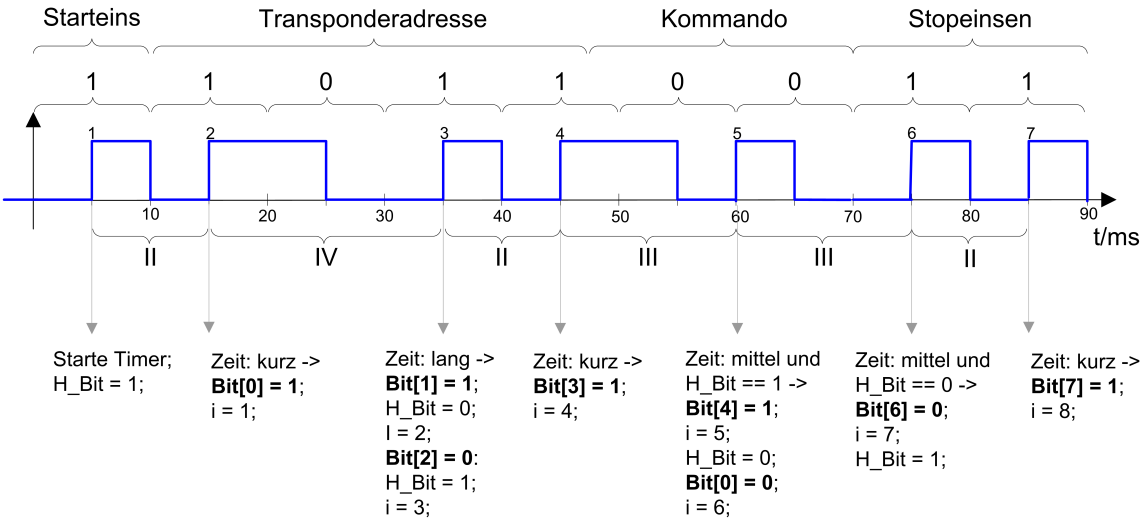

Bild 62: Datendemodulation im Transponder

4 Implementierung Reader

Bei der Entwicklung des Readers wird zunächst eine Baugruppe einzeln entworfen und auf Funktion getestet, bevor die nächste Baugruppe entwickelt wird und letztendlich die einzelnen Komponenten zusammengeführt werden und auf korrekte Funktion geprüft werden. Für den Leiterplattenentwurf wird wie schon für den Transponder das Cad-Programm Eagle verwendet. Während der Entwicklungszeit wurden diverse Prototypen auf Lochrasterplatinen aufgebaut, deren Schaltungen als Grundlage für die endgültige Readerplatine dienen. Zusätzlich zu der selbsterstellten Readerplatine wird ein Evaluierungsboard von Olimex für den Reader verwendet. Im Kapitel zur Hardwarebeschreibung wird ein genauer Aufbau des Readers dargestellt.
Der Aufbau dieses Kapitels lehnt sich an den Aufbau des Kapitels zur Implementierung des Transponders an. Es wird in die Entwicklung der Hardware und Software aufgeteilt.

4.1 Hardware

In dem Unterkapitel zur Hardwarebeschreibung des Readers wird zunächst ein Überblick der entwickelten Komponenten im System gegeben. Die Abbildung 63 zeigt die einzelnen Komponenten und deren Schnittstellen zueinander. In der Komponentenübersicht werden nicht alle Baugruppen des Evaluierungsboards dargestellt, da dies das Blockschaltbild unübersichtlich gestalten würde, sondern es werden nur die Komponenten dargestellt, die verwendet werden. Für eine weitere Version der Readerplatine wäre es erstrebenswert, auf das Evaluierungsboard zu verzichten und die benötigten Komponenten separat auf der Readerplatine zu integrieren. Der komplette Schaltplan der Readerplatine ist im Anhang B.2 zu finden. Die Abbildung 64 zeigt ein Foto von dem entwickelten Reader.
In den folgenden Unterabschnitten werden Signalflüsse, Amplitudengänge und Spektren an geeigneten Schnittstellen im System aufgenommen und diskutiert.

4.1.1 Erzeugung des elektromagnetischen Feldes

Für die Erzeugung des elektromagnetischen Feldes wird, wie in dem Kapitel zur Planung des Readers beschrieben, ein Reihenschwingkreis über Treiberbausteine mit einem Rechtecksignal in seiner Resonanzfrequenz angeregt. Damit eine höhere Leistung am Schwingkreis zur Verfügung steht und damit der Schwingkreis feiner abgeglichen werden kann, werden mehrere Treiberbausteine parallel geschaltet. Die genauen Anordnungen der Bauelemente sind dem Schaltplan aus dem Anhang B.2 zu entnehmen.
Damit ein Optimum in Bezug auf die Bandbreite und die Feldstärke für das System ge-

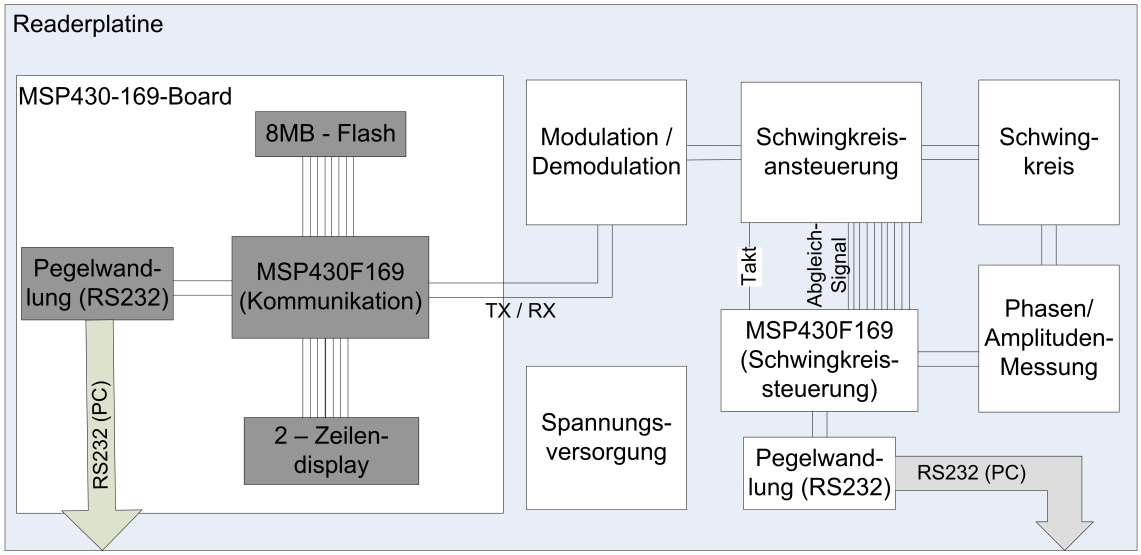

Bild 63: Komponentenübersicht im Reader

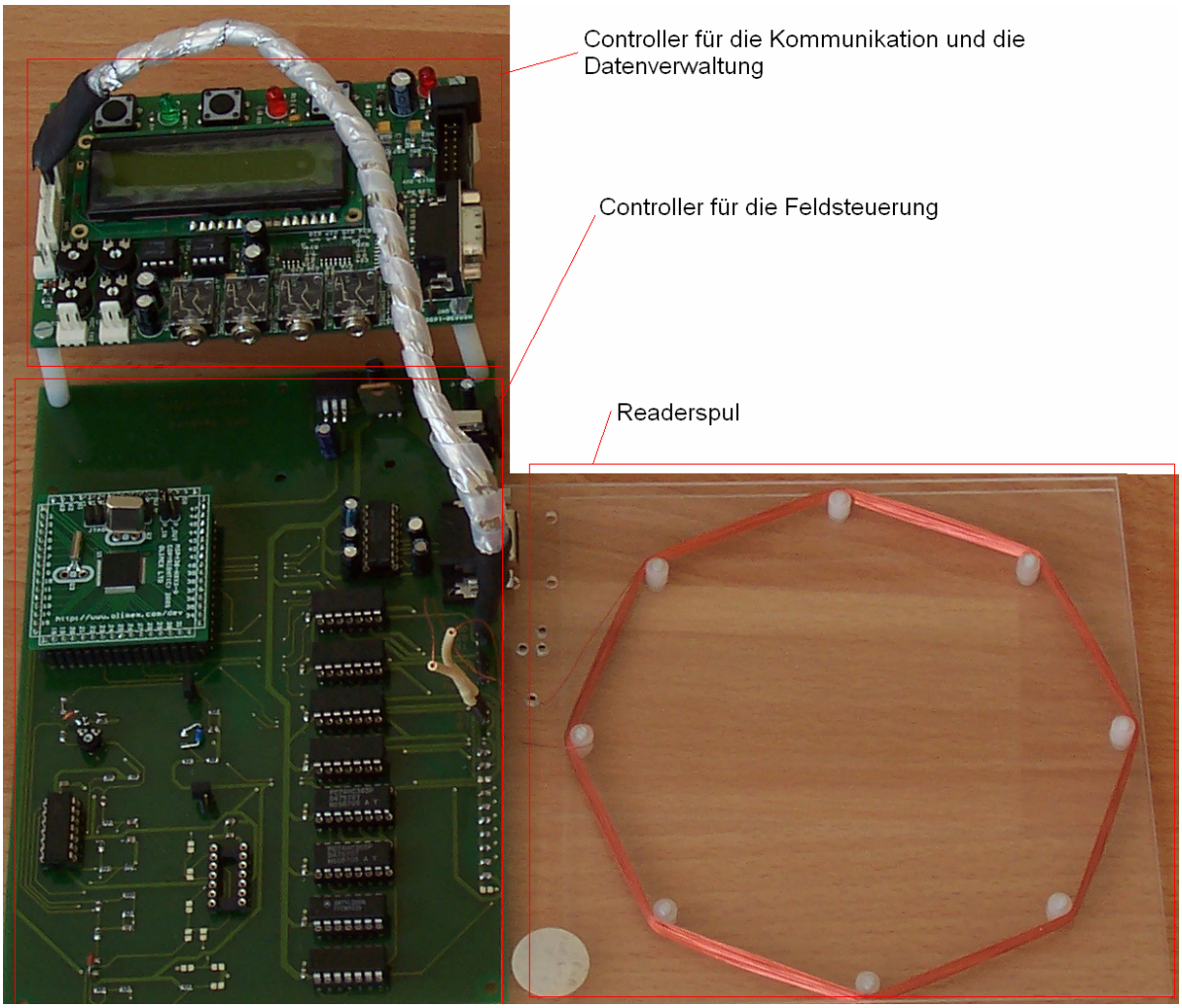

Bild 64: Foto vom Reader

funden werden kann, wird für den Serienwiderstand ein variabler Widerstand im Bereich von 0Ω bis 50Ω eingesetzt. Durch die Variation des Widerstandswertes können Feinabstimmungen vorgenommen werden.
Die Abbildung 65 zeigt das Oszilloskopbild mit den Amplitudenverläufen von der Spannung im Resonanzpunkt des Reihenschwingkreises und den beiden gegensätzlichen Anregespannungen am oberen und unteren Anschluss des Schwingkreises.

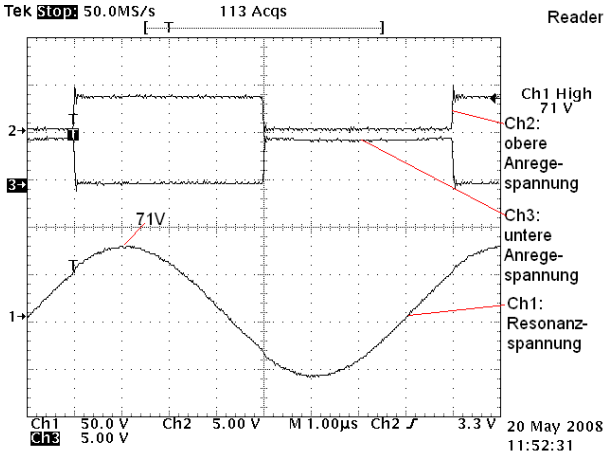

Bild 65: Resonanz- und Anregespannung im Reader

4.1.2 Schwingkreisabgleich

Für den Resonanzabgleich werden in der Theorie zwei Methoden beschrieben. Einerseits ist es möglich, die Anregefrequenz auf die Schwingkreisparameter anzupassen und somit den Schwingkreis in Resonanz zu bringen, oder es werden die Schwingkreisparameter auf die Taktfrequenz angepasst. Für die Bestimmung auf Resonanz können die Amplitudenhöhe und der Phasenwinkel herangezogen werden. Die Schaltungen zur Bestimmung der Parameter werden nachfolgend beschrieben.
Bei der ersten Methode, bei der die Frequenz des Anregetaktes geändert wird, ist es zusätzlich möglich, die Kapazität mit zu verändern. So kann die Anregefrequenz im Bereich von $120kHz$ bis $130kHz$ durchgeschaltet werden und mit Hilfe der Kapazitätsanpassung in Resonanz gebracht werden. So kann die Readerresonanzfrequenz an unabgeglichenen Transponderresonanzfrequenzen angepasst werden. Für die Realisierung ist es allerdings notwendig, dass der Mikrocontroller für die Felderzeugung eine Rückmeldung über die Energieversorgung des zur Zeit angesprochenen Transponders benötigt. Dazu müsste der Mikrocontroller für die Kommunikation (der diese Information besitzt) und der für die Felderzeugung Daten miteinander austauschen können. Diese Verbindung kann über die serielle Schnittstelle der Controller hergestellt werden. Dies wurde im Rahmen dieser Diplomarbeit nicht mehr realisiert, so dass auch das eben beschriebene Verfahren noch nicht implementiert werden konnte. Für eine Weiterentwicklung wäre diese Ergänzung

erstrebenswert, da so ein aufwändiger Abgleich der Transponder wegfallen würde.
Auf der aktuellen Readerplatine sind die Schaltung zur Bestimmung des Phasenwinkels, die Schaltung zur Bestimmung der Amplitudenhöhe und die Beschaltung der Treiber zum Kapazitätsabgleich implementiert.
Die Einstellung der Schwingkreiskapazität wird wie in der Planung beschrieben mit der Beschaltung der Enable-Eingänge der Treiberbausteine erreicht. Es werden insgesamt 11 Leitungen aus dem Controller für die Felderzeugung zu den Enable-Eingängen der Treiberbausteine gelegt, mit denen es möglich ist, 2048 verschiedene Kapazitäten im Bereich von $C_{min} = 0pF$ bis $C_{max} = 1876,9pF$ mit einer Schrittweite von $1pF$ einzustellen. Die Kapazität der Kondensatoren C_1 bis C_{11} wurde so ausgelegt, dass sie nach einer Zweierpotenz steigen. Da es in der Standard-Reihe für Kondensatoren nicht die benötigten Werte gibt, wurden die Kapazitäten so gewählt, dass sie den berechneten möglichst genau entsprechen. Die Tabelle 12 zeigt die berechneten und die verwendeten Kapazitäten für die Kondensatoren.

Die Auswahl und Auswertung der Kapazitäten wird in dem Unterabschnitt zur Implementierung der Software für den Feldsteuerungungs-Controller erläutert.
In den folgenden Oszilloskopbildern werden die Amplitudengänge an den Schaltungen für die Amplitude- und Phasenwinkelmessung dargestellt. Die Abbildung 66 zeigt die gleichgerichtete, geglättete und heruntergeteilte Spannung im Resonanzpunkt des Reihenschwingkreises, die an einen ADC-Eingang vom Controller für die Feldsteuerung angelegt ist. Mit Hilfe des Spannungswerts im Vergleich zu anderen, kann eine Aussage über den Resonanzabgleich getroffen werden.

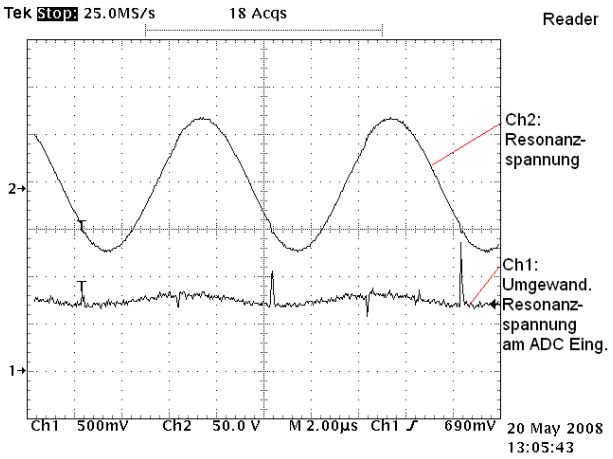

Bild 66: Gleichgerichtete, geglättete und runtergeteilte Spannung im Resonanzpunkt des Reihenschwingkreises des Readers

Für den Resonanzabgleich über die Bestimmung des Phasenwinkels wird die Spannung im Resonanzpunkt des Reihenschwingkreises zunächst über einen Spannungsteiler heruntergeteilt und mit einer Summierschaltung um $\frac{V_{CC}}{2}$ angehoben. Der Ausgang der Summierschaltung wird an einen Komparatoreingang des Feldsteuerungs-Controllers gelegt.

Kondensator	Berechnete Kapazität	Gewählte Kapazität
C_1	$1000pF$	$1000pf$
C_2	$500pF$	$470pf$
C_3	$250pF$	$220pf$
C_4	$125pF$	$100pf$
C_5	$62,5pF$	$47pf$
C_6	$31,25pF$	$22pf$
C_7	$15,625pF$	$10pf$
C_8	$7,8125pF$	$4,7pf$
C_9	$3,90625pF$	$2,2pf$
C_{10}	$1,953125pF$	$1pf$
C_{11}	$0,9765625pF$	$0pf$

Tabelle 12: Kapazitäten des Readerschwingkreises

Die interne Schwellspannung des Komparators im Mikrocontroller wird auf $\frac{V_{CC}}{2}$ gelegt. Diese Einstellung und Beschaltung hat zur Folge, dass der Komparatorinterrupt im Controller ausgelöst wird, wenn die Spannung im Nulldurchgang ist. Zusätzlich ist eine Kompensation gegenüber Schwankungen in der Betriebsspannung gegeben, da die Schwellspannung und die Anhebespannung im Summierer von V_{CC} abhängig sind und sich somit Änderungen gleichwertig auf beide Komponenten auswirken. Die Abbildung 67 zeigt den Spannungsverlauf der Anregung und den der Spannung im Resonanzpunkt des Reihenschwingkreises. In der gezeigten Abbildung ergibt sich ein Phasenwinkel von $\varphi = 135°$. Da der Phasenwinkel über 90° beträgt, weist dies auf eine zu kleine Schwingkreiskapazität hin. Infolge dessen müsste die Kapazität vom Controller erhöht werden, damit der Schwingkreis abgeglichen wird.

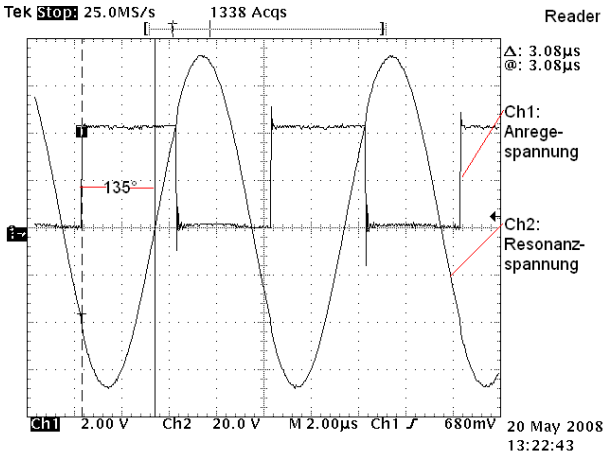

Bild 67: Bestimmung des Phasenwinkels im Readerschwingkreis

4.1.3 Modulation

Die Modulation des Schwingkreises erfolgt durch das Ab- und das Zuschalten des Anregetaktes an die Treiberbausteine. Der Anregetakt wird von dem Mikrocontroller für die Feldsteuerung im Bereich von $125kHz$ erzeugt. Das Modulationssignal wird vom Mikrocontroller für die Kommunikation im Bereich von $1kHz$ erzeugt. Zur Modulation werden diese beiden Signale auf die Eingänge eines UND-Gatters gelegt. Der Ausgang des UND-Gatters wird mit den Eingängen der Treiberbausteine verbunden. Diese Beschaltung sorgt dafür, dass der Anregetakt von $125kHz$ nur am Treiber ankommt, wenn das Modulationssignal 1 ist. Bei dem Modulationssignal ist darauf zu achten, dass der Idle-Zustand 1 ist, da das elektromagnetische Feld während Kommunikationspausen sonst abgeschaltet wäre und somit die Energieversorgung der Transponder fehlen würde.
In der Abbildung 68 wird die Modulation anhand des Oszilloskopbildes deutlich. Auf Grund der großen Frequenzunterschiede ist das Taktsignal von $125kHz$ nicht mehr deutlich darzustellen.

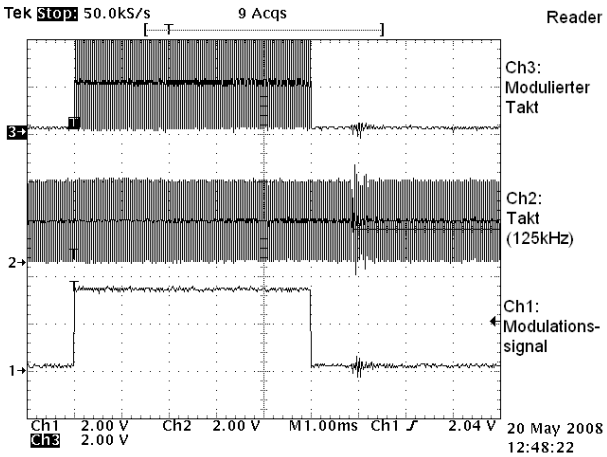

Bild 68: Modulation des Taktsignals der Treiberbausteine

Um die Ein- und Abschwingdauer des Schwingkreises bei der Modulation zu bestimmen, wird zusätzlich eine Oszilloskopbild von der Modulationsspannung und der Spannung im Resonanzpunkt des Reihenschwingkreises bei der steigenden und bei der fallenden Flanke des Modulationssignals aufgenommen. In den Abbildungen 69 und 70 wird deutlich, dass der Schwingkreis schnell genug anschwingt und wieder abklingt, so dass keine weiteren schaltungstechnischen Änderungen wie zum Beispiel das Kurzschließen des Schwingkreises bei der Modulation getroffen werden müssen.

4.1.4 Demodulationsschaltung

Die Demodulationsschaltung entspricht der Schaltung aus der Simulink-Simulation im Kapitel zur Planung des Readers. Die Realisierung der Pässe und der weiteren Bau-

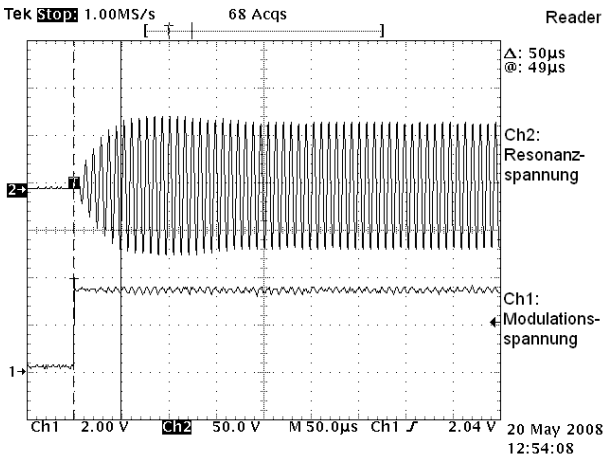

Bild 69: Anschwingvorgang des Readerschwingkreises bei der Modulation

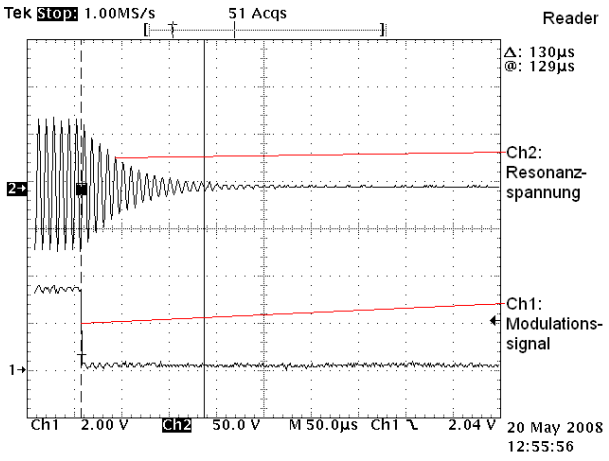

Bild 70: Abschwingvorgang des Readerschwingkreises bei der Modulation

gruppen sind dem Schaltplan aus dem Anhang B.2 zu entnehmen. Die Festlegung der Grenzfrequenzen erfolgte zunächst theoretisch, indem die Grenzfrequenz zwischen die zu trennenden Frequenzen gelegt wurde und mit pSpice optimiert wurde. Beim ersten Aufbau des Filters wurden Widerstandsdekaden und verstellbare Kondensatoren verwendet. Durch Variation der Werte wurde die Filterschaltung weiter optimiert und führte zu den jetzt verwendeten Grenzfrequenzen. Die Grenzfrequenzen sind im Schaltplan notiert.
Für die Auslegung des aktiven Tiefpasses wurden die Werte für die Widerstände und Kondensatoren zunächst berechnet. Da sich die Grenzfrequenz im Laufe der Entwicklung häufiger veränderte, um die Schaltung weiter zu optimieren, wurde dazu übergegangen, den Filter mit einem Design-Tool von Analog Devices zu entwerfen. Diese Art der Filterentwicklung hat sich als sehr vorteilhaft erwiesen, da mit Hilfe des Design-Tools Zeit gespart wird und keine Rechenfehler auftreten können. Die Abbildung 71 zeigt das auf Java basierende Design-Tool mit dem für die Demodulierung entworfenen Filterparametern. Die Abbildung 72 zeigt den Amplituden- und Phasenverlauf des entworfenen Filters. Die Werte für die Widerstände und Kondensatoren in dem endgültigen Schaltplan weichen wieder auf Grund von Optimierungen in der Entwicklungsphase von den berechneten Werten ab.
Das Ausgangssignal, welches am Filterausgang anliegt, entspricht von der Frequenz her den Daten. Aber da es noch sinusförmig ist, muss es mit einem Schmitttrigger digitalisiert werden. Für die Digitalisierung wird ein nichtinvertierender Schmitttrigger mit vom Nullpunkt abweichenden Schaltschwellen eingesetzt. Die Schaltschwellen können für die Optimierung über das Potenziometer am Schmitttrigger eingestellt werden. Um eine phasengleiche Digitalisierung zu erreichen, müssten die Schaltschwellen auf $0V$ gelegt werden. Dies würde aber dazu führen, dass es in den Sendepausen ständig zu Flankenwechseln am Ausgang des Schmitttriggers kommen würde. Die ständigen Flankenwechsel würden zwar zu keiner Fehlinterpretation der Daten führen, würden aber den Mikrocontroller, der den seriellen Datenstrom einliest, stark belasten, da dieser bei jeder steigenden Flanke über einen Interrupthandler prüft, ob es sich bei dem ankommenden Signal um die festgelegte Run-In-Sequenz handelt. Mit einem Schmitttrigger, bei dem die Schaltschwelle auf $0V$ liegt, wird die größte Leseentfernung erreicht.
Andererseits dürfen die Schaltschwellen auch nicht zu weit vom Nullpunkt gelegt werden, da dadurch die Empfangsreichweite geringer wird. Bei größeren Schaltschwellen ist es möglich und wurde im Rahmen dieser Diplomarbeit auch zunächst implementiert, auf die Run-In-Sequenz zu verzichten, da keine Störungen mit übertragen werden. Dies führt zu einer leichteren Softwareimplementierung im Empfangs-Mikrocontroller.
Für die endgültige Implementierung wird ein Kompromiss aus den beiden beschriebenen Lösungen eingesetzt. Die Schaltschwellen werden im laufenden Betrieb so verändert, dass eine große Empfangsreichweite erzielt wird und der Schmitttrigger in den Sendepausen nur wenige Flankenwechsel erzeugt. In der Software für den Empfangsmikrocontroller wird eine Prüfung der Run-In-Sequenz implementiert.

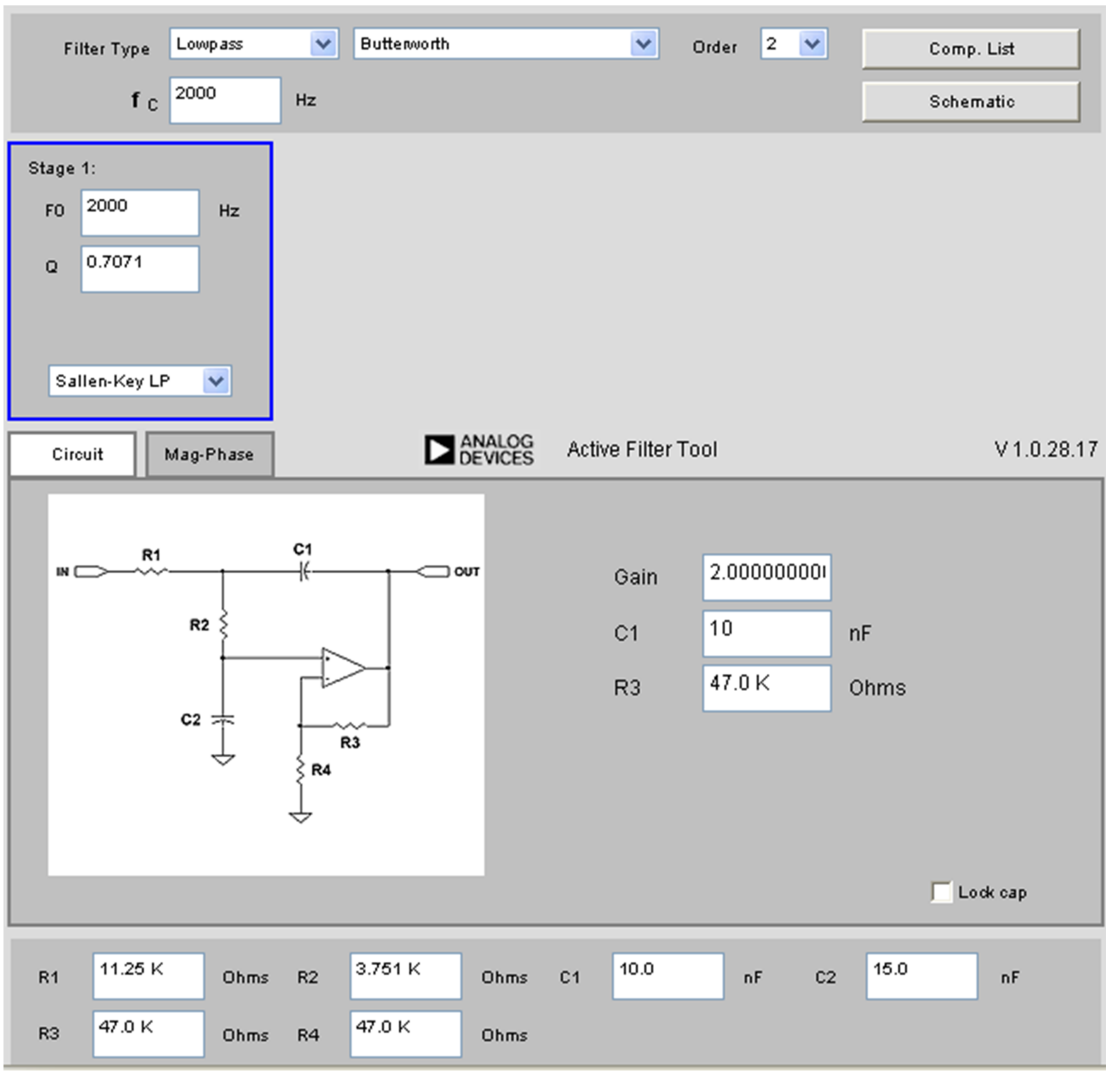

Bild 71: Active Filter Tool V1.0.28.17 von Analog Devices

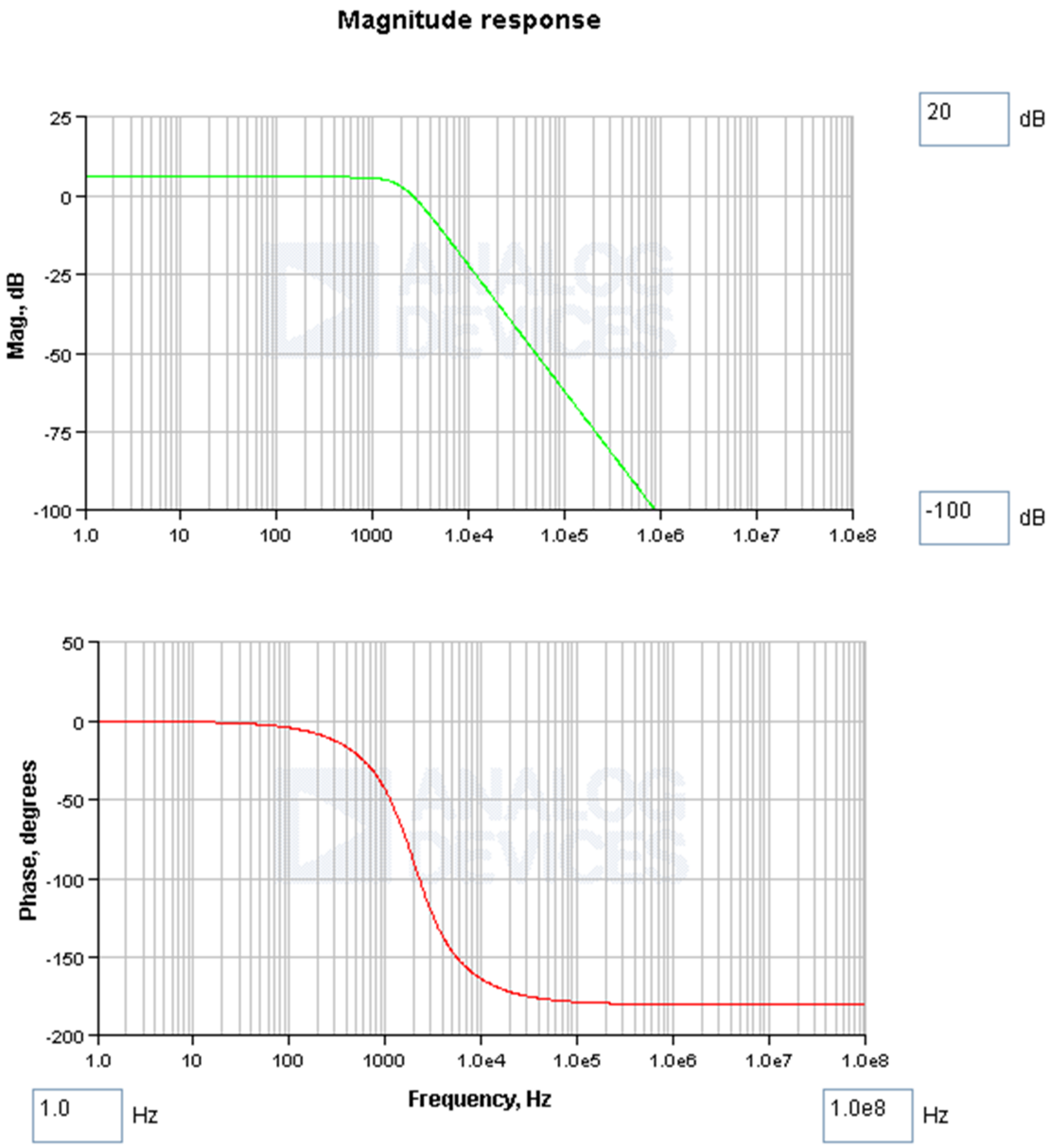

Bild 72: Amplituden- und Phasengang des aktiven Filters

4.1.5 Datenübertragung

Für die Ermittlung der Zeit, die für die Abfrage eines Transponders benötigt wird, wird ein kompletter Abfragevorgang auf dem Übertragungskanal aufgenommen. Da die Übertragung nach dem Halbduplex-Prinzip abläuft, benutzen der Reader und der Transponder den gleichen Kanal zum Senden. Deshalb ist es möglich, einen kompletten Abfragevorgang auf dem Oszilloskop darzustellen. In Abbildung 73 ist der Signalverlauf am Ausgang der Demodulationsschaltung im Reader dargestellt. Zu Beginn der Aufzeichnung ist das Anfragepaket, dass der Reader auf dem Kanal sendet zu sehen. Es besteht aus 18 Bits und wird im Bereich von $50Hz$ bis $100Hz$ auf dem Kanal gesendet. Anschließend kommt eine Pause. In dieser Zeit ermittelt der angesprochene Transponder die Temperatur und Betriebsspannung und codiert diese. Anschließend werden die 121 Bits vom Transponder mit einer Frequenz im Bereich von $500Hz$ bis $1000Hz$ auf dem Kanal gesendet. Aus dem Oszilloskopbild wird die Zeit für eine komplette Transponderabfrage auf $273ms$ festgelegt. Diese Zeiten werden für die Festlegung von Timeoutzeiten bei der Kommunikation in der Readersoftware verwendet. Zusätzlich werden die Timeoutzeiten der PC-Steuersoftware beim Senden von Abfragebefehlen an den Reader auf diese Zeiten angepasst.

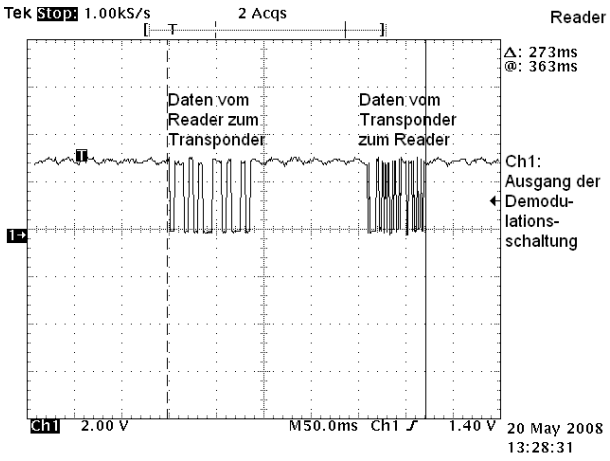

Bild 73: Signalverlauf von einem kompletten Kommunikationsvorgang

4.2 Software Kommunikations-Controller

Die Software, die für den Reader geschrieben wird, ist deutlich komplexer als die Transpondersoftware. Infolge der großen Codemenge wurde der Quellcode in Funktionsblöcke aufgeteilt, die jeweils in einer eigenen Quelldatei mit zugehörigen Headerfile implementiert werden. Mit diesem modularen Aufbau sind Erweiterungen leicht möglich. Und es können projektspezifisch ausgewählte Funktionsblöcke eingebunden oder entfernt werden. Das main-File ist auf den modularen Aufbau der Funktionsblöcke angepasst, so

dass jederzeit weitere Funktionen integriert oder entfernt werden können.
Grundsätzlich werden für das Hauptprogramm mehrere Funktionsmodi zur Verfügung gestellt, die spezifische Aufgaben erledigen. Zur Zeit sind dri verschiedene Modi von 255 möglichen implementiert. Die Steuerung der Modi übernimmt die für den PC entwickelte Steuerapplikation, die in einer grafischen Oberfläche läuft. Der Reader und der PC sind dafür über die RS232-Schnittstelle miteinander verbunden. Für den Betrieb ohne PC ist es möglich, die Funktionsmodi über die Taster am Reader umzustellen. So können aktuelle Temperaturen und Spannungen von Transpondern ermittelt werden oder die zyklische Abfrage der Transponder eingeleitet werden, ohne dass ein PC angeschlossen ist. Die Abbildung 74 zeigt den Programmablaufplan des Programms auf der höchsten Abstraktionsebene. Die Realisierung der einzelnen Blöcke wird in den folgenden Abschnitten erläutert. Im Programmablaufplan wird die einfach und modular gehaltene Struktur aufgezeigt, die mögliche Erweiterungen erleichtert.
Zu Beginn des Programmes werden Initialisierung für die interne Hardware des MSP430F-169 und für externe Hardware auf dem Evaluierungsboard vorgenommen. Anschließend wird in der Initialisierung der Defaultzustand hergestellt. Dies beinhaltet das Einstellen der Parameter für den Adressraum für abzufragende Transponder, Abfrageraten und weitere Programmparameter. Anschließend läuft das Programm in der Hauptschleife. Zu Beginn der Hauptschleife wird geprüft, ob eine Anfrage auf einen Modewechsel ansteht. Ist dies der Fall, wird die Anfrage bearbeitet und ein neuer Betriebsmode entgegengenommen. Ansonsten wird der aktuelle Mode beibehalten. Anschließend wird einer der zur Zeit drei implementierten Betriebsmodi ausgeführt. Zusätzlich zu den im Folgenden beschriebenen Funktionen werden modusabhängige Informationen auf dem Display angezeigt.
Im Betriebsmodus 1 [zyklischer Modus] werden periodisch im Abstand der eingestellten Abfragerate die Transponder aus dem eingestellten Adressbereich abgefragt. Die Transponderdaten werden in dem ringspeicher-organisierten Flash geschrieben.
Im Betriebsmodus 2 [azyklischer Modus] wird auf die Datenanfrage der Steuersoftware vom PC gewartet. Nachdem eine Datenanforderung mit einer Transponderadresse vom PC empfangen wurde, wird der entsprechende Transponder ausgelesen und die ermittelten Daten werden über die serielle Schnittstelle an die PC-Steuersoftware gesendet.
Im Betriebsmodus 3 [Abfrage-Modus] wird auf die Anfrage von Systemparametern der Steuersoftware vom PC gewartet. Bei einer empfangenen Anfrage wird der angefragte Systemparameter über die serielle Schnittstelle an die PC-Steuersoftware gesendet. Dies können zum Beispiel Informationen über die Anzahl der Datensätze im Ringspeicher oder die Systemzeit im Mikrocontroller sein.

Für die Realisierung der Funktionen im gezeigten Programmablaufplan werden die folgenden Quellcodedateien erstellt und erweitert. Wie schon bei der Vorgehensweise bei der Softwareimplementierung für den Transponder werden die einzelnen Module zunächst seperat geschrieben und getestet, bevor sie in das Gesamtprojekt integriert werden. Auf

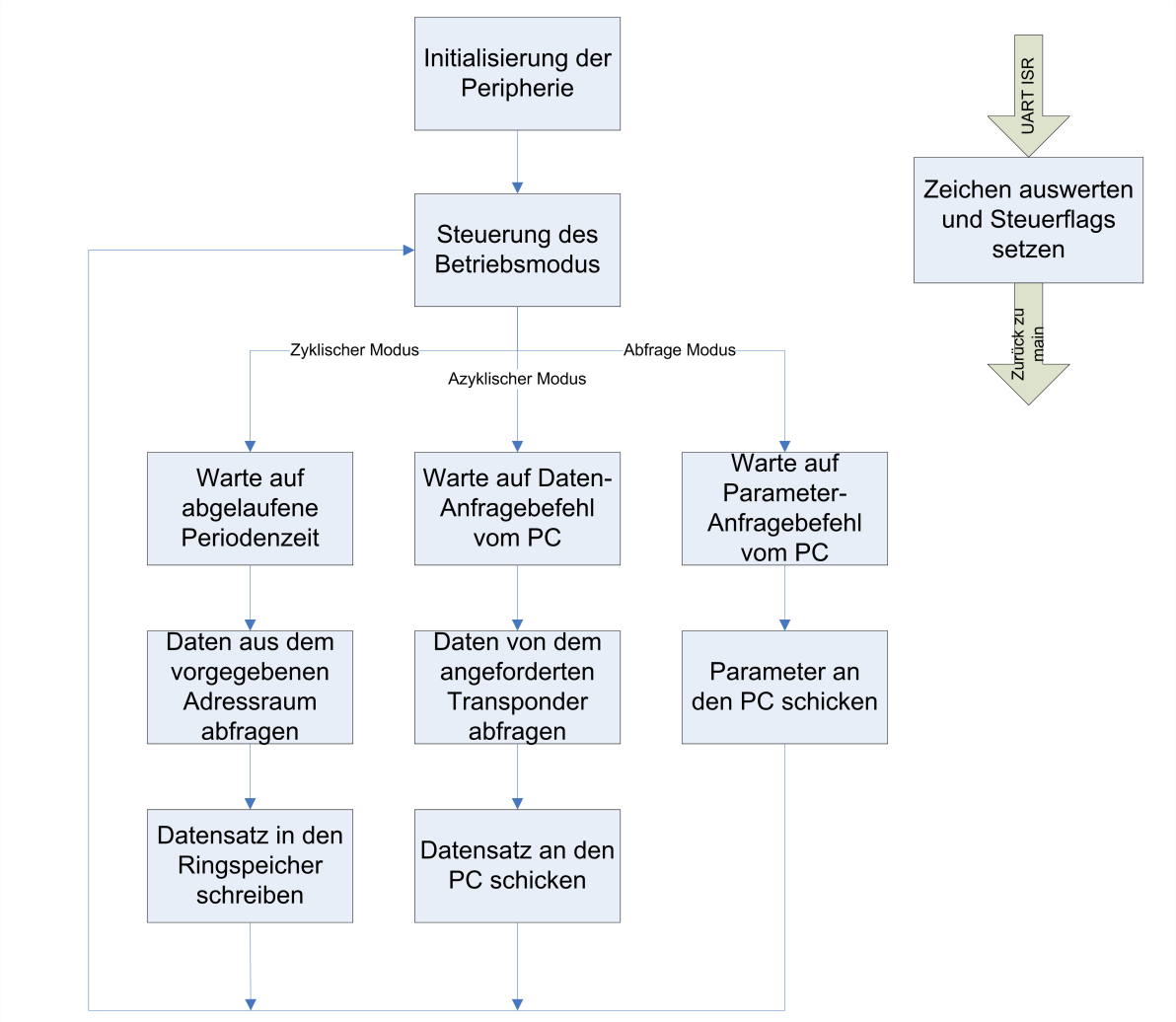

Bild 74: Programmablaufplan des Reader-Controller-Datenkommunikation

Grund der Komplexität der gesamten Software für den Kommunikationscontroller wird an dieser Stelle nicht auf die Implementierung der einzelnen Funktionen eingegangen, sondern es wird ein Überblick über die Firmware für den Kommunikations-Controller gegeben. Dazu werden die Funktionen der Quellcodedateien kurz beschrieben. Für die Beschreibung der genauen Implementierung wird auf den Quellcode im Anhang A.3 verwiesen.

main.c In der main-Funktion werden keine Funktionen implementiert, sondern es werden Funktionen aus den weiteren Quellcodedateien, wie im Programmablaufplan dargestellt, aufgerufen.

ComPC.h/c Die Dateien ComPC.h und ComPC.c stellen Funktionen für die Kommunikation mit der PC-Steuersoftware zur Verfügung. Das Empfangen von Steuerdaten, die vom PC gesendet werden, wird interruptgesteuert ausgeführt. Im Interrupthandler läuft der Zustandsautomat für die Modisteuerung, ein Echoservice, und zusätzlich werden Steuerdaten an die Betriebsmodi weitergegeben. Weitere Funktionen steuern das Handshakeverfahren bei der Kommunikation zwischen Reader und PC.
Zusätzlich zu den Funktionen zur Kommunikation mit dem PC werden Funktionen zur Ansteuerung des Diplays zur Verfügung gestellt.

Flash.h/c Die Dateien Flash.h und Flash.c steuern das Schreiben und das Lesen des 8MB großen, externen Flashspeichers. Für das Speichern von Transponderdaten steht eine 16 Byte große Struktur zur Verfügung, in der Spannung, Temperatur, Transponderadresse und Systemzeit gespeichert werden. Die Datensätze werden in einer Ringstruktur im Speicher abgelegt. Die Adressen des Speichers werden von den Schreib- und Lesefunktionen selbstständig berechnet und verwaltet. Bei der Implementierung der Funktion zum Beschreiben des Speichers muss darauf geachtet werden, dass der Inhalt einer Speicheradresse zunächst gelöscht werden muss, bevor neue Daten in die Adresse geschrieben werden können. Für ein zusätzliches Problem beim Löschen sorgt die Tatsache, dass der Flash nicht byteweise gelöscht werden kann, sondern nur blockweise. Außerdem ist die Adressstruktur des Speichers 3-dimensional angeordent. Diese Eigenschaften führen zu einer komplexen Adressberechnung bei der Schreibfunktion.
Für den Fall, dass die 8MB vollgeschrieben sind, wird der älteste Datenblock gelöscht, damit ein neuer Datensatz gespeichert werden kann. Die Übergänge der Blöcke stellen die größte Herausforderung bei der Implementierung der Schreib- und Lesefunktion dar.
Zusätzlich zu den Schreib- und Lesefunktionen und den dafür benötigten Hilfsfunktionen steht eine Funktion zur Verfügung, mit der die Funktion des Speichers getestet werden kann. Auf Grund der langen Ausführzeit von ca. 1/2 Stunde sollte sie nicht bei jedem Programmstart ausgeführt werden, sondern nur bei der ersten in Betriebnahme.

RadioTXRX.h/c Die Dateien steuern die Kommunikation zwischen Reader und Transponder. Das Senden und das Empfangen von Daten findet interruptgesteuert statt. Für das Empfangen wird ein Input Capture Interrupt des Timers verwendet. In dem Interrupthandler findet der Empfang und die Decodierung der vom Transponder gesendeten Daten statt. Bevor die Daten im Interruphandler aufgenommen und decodiert werden, muss zunächst eine Run-In-Sequenz erkannt werden. Die Steuerung läuft in einem Zustandsautomaten ab, der die Verhältnisse zwischen den Zeiten zwischen zwei steigenden Flanken auswertet. Bei dem gesamten Datenempfang werden nicht die absoluten Zeiten, sondern die Verhältnisse zwischen den einzelnen Zeiten ausgewertet. Diese Art des Dateneinlesens ermöglicht den Empfang von fast beliebigen Frequenzen. Für diese Art der Datenauswertung ist es notwendig, nach der Run-In-Sequenz eine Preamble-Folge aus zuvor festgelegten Daten zu senden, damit die Sendefrequenz bestimmt werden kann.
Mit der automatischen Frequenzbestimmung beim Einlesen der Daten werden die Systemtaktungenauigkeiten der einzelnen Transponder kompensiert. Zusätzlich ist es möglich, die Datenrate vom Transponder zu ändern, ohne das die Kommunikationssoftware im Reader geändert werden muss.
Die Datendecodierung der empfangenen Daten läuft nach dem gleichen Prinzip wie die Decodierung im Transponder ab.
Für das Senden von Steuerdaten zum Transponder wird zunächst ein Sendevektor beschrieben und anschließend selbstständig über einen Output Compare Interrupt des Timers gesendet.
Zusätzlich zu der eigentlichen Kommunikation werden die Daten mit Hilfe einer Checksumme verifiziert.

System.h/c Steuern die Einstellungen der Ports und des Watchdogtimers. Der Watchdogtimer wird periodisch im 1/4 Sekundentakt aufgerufen und inkrementiert die Variable Tick, die für die Zeitsteuerung im System benötigt wird. Mit Hilfe der Tick-Variable ist es möglich, Zeitabstände zwischen den Datensätzen zu erfassen. Die absolute Zeit der Datensätze wird nach der Übertragung im PC ermittelt und als Zeitstempel zum Datensatz in die Datenbank gespeichert.

CLK.h/c Die Dateien sind von Stephan Plaschke übernommen. Sie stellen eine Funktion zur Steuerung des internen Taktgenerators zur Verfügung.

4.3 Software Feldsteuerungs-Controller

Die Software, die für den Mikrocontroller geschrieben wird, der für die Feldsteuerung zuständig ist, ist zur Zeit in einem frühen Entwicklungsstadium. Sie ist im Anhang unter

A.4 dargestellt. Mit der Erweiterung der Software können die in der Planung beschriebenen Verfahren zum Schwingkreisabgleich durch Bewertung des Phasenwinkels oder die Readerresonanzfrequenzanpassung an die Transponderresonanzfrequenz umgesetzt werden. Die Abbildung 75 zeigt den Programmablaufplan für die zur Zeit implementierte Software.
Nachdem der Controller initialisiert ist, wird ein Timer gestartet, der automatisch über seine Hardware einen 125kHz-Takt erzeugt und an einem Portausgang ausgibt. Anschließend wird eine Routine zum Schwingkreisabgleich ausgeführt. In der Routine wird die Kapazität vom minimalen Wert bis zum maximalen Wert durchgeschaltet. Die dazugehörige Spannung im Resonanzpunkt des Schwingkreises wird mit dem ADC aufgenommen und mit dem aktuellen Maximum verglichen. Nachdem die Maximalkapazität erreicht ist, wird die Kapazität eingestellt, bei der die maximale Ausgangsspannung im Schwingkreis ermittelt wurde.
Parallel ist es möglich, dem Controller eine Kapazitätsvorgabe über die serielle Schnittstelle zu schicken. Der Controller stellt die vorgegebene Kapazität ein und ermittelt anschließend die Ausgangsspannung im Schwingkreis und sendet diese über die serielle Schnittstelle zurück.

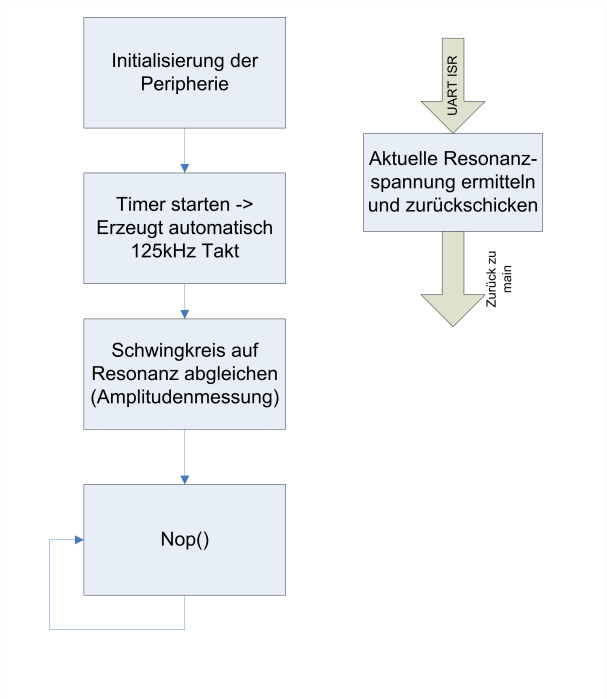

Bild 75: Ablaufplan der Reader-Controller-Feldansteuerung

5 Implementierung Applikation

Für die Steuerung des Systems und für die Darstellung und Auswertung der Transponderdaten wird eine grafische Oberfläche in C++ mit der MFC erstellt. Die Steuersoftware kommuniziert einerseits mit dem Reader über die serielle Schnittstelle und andererseits über verschiedene APIs mit weiteren Programmen, die auf dem PC laufen und für die bessere Darstellung und Verwaltung der Daten benötigt werden. Die Abbildung 76 zeigt den Aufbau und das Zusammenspiel der einzelnen Elemente der PC-Applikation. Die selbsterstellten Quellcode-Dateien sind im Anhang A.5 dargestellt. Die weiteren benötigten Klassen sind im Anhang nur aufgezählt. Die Quellcode-Dateien zu diesen Klassen befinden sich auf der CD.

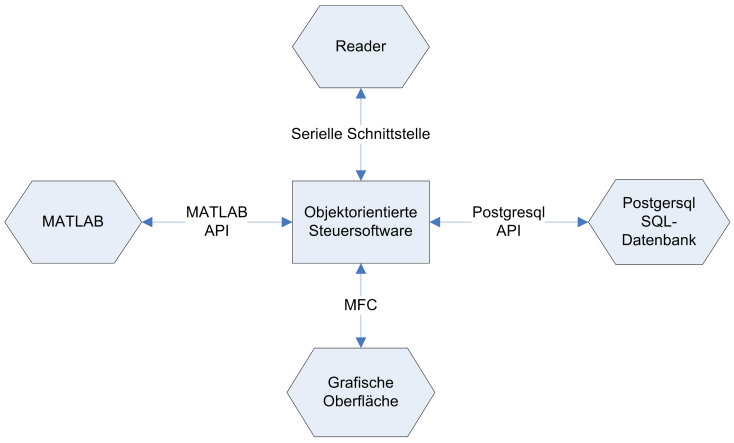

Bild 76: Aufbau der PC-Applikation

Auf Grund der sehr komplexen Softwarestruktur werden innerhalb dieses Kapitels nur die Möglichkeiten, die die Applikation bietet, beschrieben, da eine ausführliche Beschreibung der einzelnen Komponenten den Rahmen sprengen würde. Deshalb wird für eine ausführliche Beschreibung auf den Quellcode und die sich darin befindenden Kommentare verwiesen.
Grundsätzlich stellt der mittlere Block Steuersoftware aus der Abbildung 76 den Master bei der Kommunikation im Gesamtsystem dar. Die Steuersoftware initiiert die Kommunikationsvorgänge mit den angeschlossenen Komponenten und steuert die Handshakeverfahren und Timeouts bei fehlerhaften Kommunikationsabläufen. Nachfolgend werden die Funktionalitäten der einzelnen Elemente aus der Steuersoftware beschrieben. Die grafische Oberfläche ist in der Abbildung 77 dargestellt.

1. Datentransfer vom Reader-Speicher in die Datenbank Der Datentransfer wird von der Steuersoftware initiiert. Dafür sendet die Steuersoftware zunächst eine Verfügbarkeits-

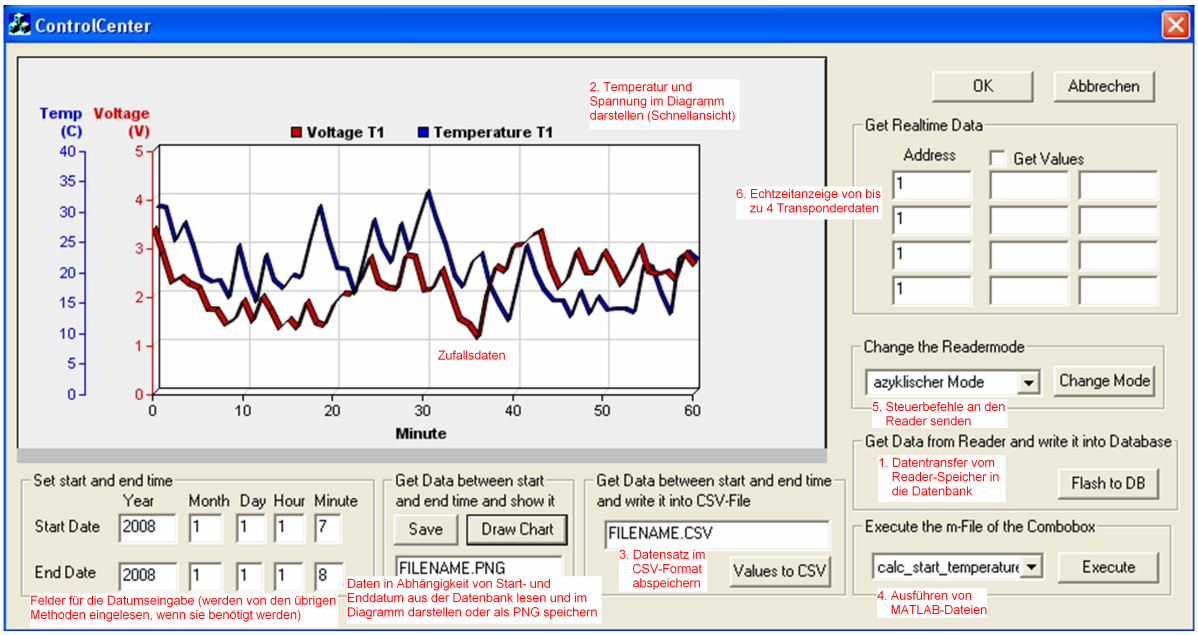

Bild 77: Grafische Oberfläche der PC-Applikation

Anfrage an den Reader. Zusätzlich wird die aktuelle Systemzeit vom Reader abgefragt. Bei positiver Rückmeldung vom Reader werden die Daten aus dem Ringspeicher vom Reader über die serielle Schnittstelle zur Steuersoftware übertragen. Es werden beginnend mit dem ältesten Datensatz alle Datensätze aus dem Ringspeicher übertragen. Die Datensätze beinhalten die Transponderadresse, Temperatur, Spannung und die Systemzeit vom Reader.

Aus der zu Beginn der Übertragung ermittelten aktuellen Systemzeit des Readers und dem Eintrag der Systemzeit des jeweiligen Datensatzes wird ein Zeitstempel mit Datum für den jeweiligen Datensatz erstellt. Um den Zeitstempel zu erstellen, wird auf die Systemzeit des Betriebssystems zurückgegriffen.

Die Transponderadresse, die Temperatur, die Spannung und der Zeitstempel werden über eine API in eine SQL Datenbank geschrieben. Für die Implementierung der Datenbank wurde PostgreSQL verwendet, da diese Datenbank frei verfügbar und weit verbreitet ist. Die Verbindung zwischen der Steuerapplikation und der Datenbank wird über die IP-Adresse und den Port der Datenbank realisiert. Die Kommunikation findet mit Hilfe von TCP Paketen statt.

Beim Tabellendesign zur Speicherung der Transponderdaten wurde die 3. Normalform nach Boyce Codd angestrebt. Diese Form erleichtert zukünftige Erweiterungen oder Änderungen der Tabellen. Die Tabellen sind im Anhang A.5.12 dargestellt. Für die Kontrolle und administrative Eingriffe wird das Datenbanktool pgAdmin verwendet, mit dem die Struktur der Datenbank visualisiert werden kann und Daten eingesehen und verändert werden können.

2. Temperatur und Spannung im Diagramm darstellen Damit die ermittelten Datensätze eingesehen werden können, ist es möglich die Temperatur und die Spannung für einen Transponder in einem Diagramm darzustellen. Die Daten können über einen beliebigen Zeitraum, der über die Oberfläche einzugeben ist, dargestellt werden. Die Datensätze werden mit Hilfe einer SQL-Abfrage aus der Dantenbank ermittelt. Damit ist die Erweiterbarkeit für spätere Weiterentwicklungen gegeben. So können zum Beispiel SQL-Abfragen integriert werden, die nur bestimmte Werte aus der Datenbank extrahieren. Zusätzlich können Minima und Maxima über einen bestimmten Zeitraum über SQL-Abfragen ermittelt werden.
Die erstellten Diagramme können für die Dokumentation von Messreihen im png-Format abgespeichert werden. Die Abbildung 78 zeigt ein Beispiel für ein gespeichertes Diagramm.

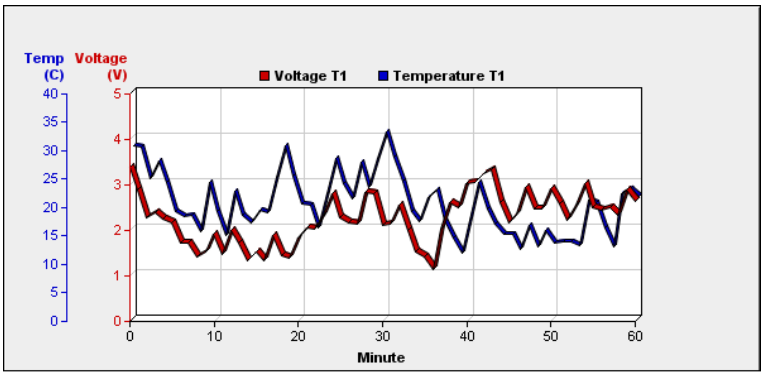

Bild 78: Temperatur-Spannungs-Diagramm mit Zufallsdaten

3. Datensatz im CSV Format abspeichern Für die Auswertung der Datensätze mit anderen Programmen können die Daten im CSV-Format in eine Datei geschrieben werden. Dafür muss zunächst das Start- und das Enddatum der Messreihe über die Oberfläche eingegeben werden. Anschließend wird der Dateiname festgelegt und die Daten in diese Datei geschrieben.

4. Ausführen von MATLAB-Dateien Damit die Daten noch besser dargestellt und ausgewertet werden können ist es möglich, beliebige m-Files auszuführen. Die m-Files werden über die Oberfläche ausgewählt und gestartet. Für das Ausführen der m-Files wurde eine Klasse erstellt, die über die engine-API mit MATLAB kommuniziert. Die Datensätze können entweder über zuvor erstellte Dateien in den m-Files eingelesen werden oder direkt über die API zu MATLAB übertragen werden.
Mit der Möglichkeit, die Datensätze mit MATLAB-Funktionalitäten zu bearbeiten, sind der Auswertung und Darstellung kaum noch Grenzen gesetzt. Für die Entwicklung einer neuen MATLAB-Routine kann diese zunächst in der MATLAB-Umgebung entwickelt

und geprüft werden. Anschließend kann diese MATLAB-Routine in die Steuersoftware-Umgebung kopiert und bei der Oberfläche angemeldet werden und steht dem Anwender ab diesem Zeitpunkt zur Verfügung.
In der aktuellen Version der Steuersoftware ist bisher nur eine MATLAB-Routine integriert. Mit Hilfe dieser Routine wird die Starttemperatur von abkühlenden Stoffen ermittelt. Die Ermittlung der Starttemperatur über die Abkühlfunktion ist notwendig, wenn die Starttemperatur über der spezifizierten maximalen Temperatur des MSP430F1232 ($85°C$) liegt, oder wenn eine Messung an dem Ort der Erwärmung nicht möglich ist. Für die Auswertung muss über die Oberfläche die Startzeit des Abkühlvorgangs eingegeben werden. Anschließend zieht sich die Steuersoftware Temperaturproben zu Zeitpunkten nach der Startzeit aus der Datenbank, zu der der Mikrocontroller wieder im spezifizierten Bereich gearbeitet hat. Mit Hilfe dieser Temperaturproben und der MATLAB-Routine wird die Starttemperatur ermittelt und die Abkühlkurve graphisch dargestellt. Das mFile für die Berechnung ist im Anhang A.5.11 dargestellt. Die Abbildung 79 zeigt ein Beispiel für die erstellte Abkühlkurve.

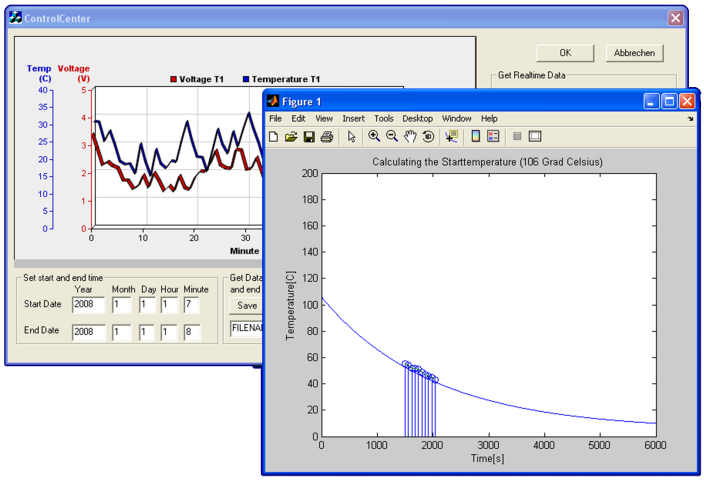

Bild 79: Abkühlkurve zur Bestimmung der Starttemperatur mit Testdaten

5. Steuerbefehle an den Reader senden Bei den meisten Aktionen werden Steuerbefehle an den Reader gesendet. Diese veranlassen den Reader, in einen bestimmten Betriebsmodus zu wechseln, oder bestimmte Parameter im System umzustellen.
Die einzige Möglichkeit, einen direkten Steuerbefehl an der Reader zu senden, besteht mit der Aktion, den Betriebsmodus im Reader umzustellen. Dabei kann über die Oberfläche betimmt werden, in welchem Modus der Reader laufen soll. Mit Hilfe dieser Funktion kann der Reader nach einer Echtzeitanzeige von Daten wieder in den zyklischen Modus versetzt werden, bei dem periodisch Datensätze in den Speicher des Readers geschrieben werden.

6. Echtzeitanzeige von bis zu 4 Transponderdaten Bei der Echtzeitanzeige von Transponderdaten sendet die Steuersoftware zunächst eine Anfrage zum Reader, die diesen dazu veranlassen soll, in den azyklischen Modus zu wechseln. Bei positiver Rückmeldung werden periodisch Anfragepakete mit der über die Oberfläche einzugebenden Transponderadresse an den Reader geschickt. Der Reader liest auf Grund dieser Anfrage den Transponder mit der übergebenen Adresse aus und sendet die empfangenen Daten an die Steuersoftware auf dem PC. Diese wandelt die skalierten Daten in Gleitkommazahlen um und stellt sie auf der Oberfläche dar.
Zusätzlich ist es möglich, die aktuell empfangenen Daten grafisch in dem Diagramm darzustellen. Das Diagramm wird periodisch aktualisiert und zeigt die Daten der letzten 60 Sekunden an.

6 Funktionsnachweis des Gesamtsystems

Damit die Funktion des Gesamtsystems nachgewiesen wird, wird ein Messaufbau zur Bestimmung der Starttemperatur von Kunststoffplatten mit Hilfe der Abkühlkurve realisiert.
In diesem Fall könnte die Starttemperatur auch direkt ermittelt werden, da die Starttemperatur der Kunststoffplatten unter der spezifizierten Höchsttemperatur des Controllers liegt. Jedoch zeigt dieser relativ einfache Messaufbau die Funktionsweise für die Berechnung der Starttemperatur anhand von nachträglich aufgenommenen Temperaturproben.
Für die Messung werden die Kunststoffplatten in einem Wasserbad auf $56°C$ erwärmt und in einer vor Zugluft geschützten Kunststoffbox abgekühlt. Ein Transponder wird in die Box zwischen den Kunststoffplatten platziert. Die Kunststoffbox wird auf die Readerspule gestellt, so dass der sich zwischen den erwärmten Kunststoffplatten befindliche Transponder ausgelesen werden kann.
Der Reader wird mit der Steuersoftware in den zyklischen Abfragemode versetzt. Innerhalb der nächsten 30 Minuten fragt der Reader im Sekundentakt die Temperatur und Betriebsspannung des Transponders ab und schreibt die erhaltenen Daten in den Flash. Anschließend initiiert die Steuersoftware auf dem PC den Datenaustausch. Die 1800 Datensätze werden über die serielle Schnittstelle an den PC übertragen und dort mit einem wie beschrieben berechneten Zeitstempel in die SQL-Datenbank geschrieben. Anschließend wird das m-File für die Berechnung der Starttemperatur ausgeführt. Die 10 Temperaturproben und die Zeit zwischen Abkühlbeginn und der ersten Temperaturprobe (20 Minuten) werden über die MATLAB-API in den MATLAB-Workspace übertragen. Nachdem die Berechnung abgeschlossen ist, wird der berechnete Temperaturverlauf grafisch dargestellt. Die berechnete Starttemperatur wird im Diagrammtitel notiert.

Die Abbildung 80 zeigt einen Auszug aus der SQL-Datenbank. In der Spalte „temperature“ kann die Temperatur zur Zeit des Abkühlbeginns am 2008-05-27 um 14:54:00 Uhr abgelesen werden. Sie beträgt 55.97°C. Mit einem Thermometer, dessen Sensor am Transponder befestigt ist, wurde eine Temperatur von 56°C ermittelt. Zusätzlich werden während des gesamten Abkühlvorgangs die Temperaturen im Minutentakt mit dem Thermometer aufgenommen. Die Temperaturen, die vom Transponder ermittelt werden und die, die mit dem Thermometer aufgenommen werden, weisen eine maximale Differenz von einem Grad Celsius auf.

Die über die Abkühlkurve berechnete Starttemperatur beträgt 57°C. In der Abbildung 81 ist die PC-Steuersoftware mit der über MATLAB berechneten Abkühlkurve dargestellt.

Für den Vergleich des berechneten und des mit dem Transponder aufgenommenen Temperaturverlaufs werden die Temperaturdaten für den Abkühlvorgang in eine CSV-Datei

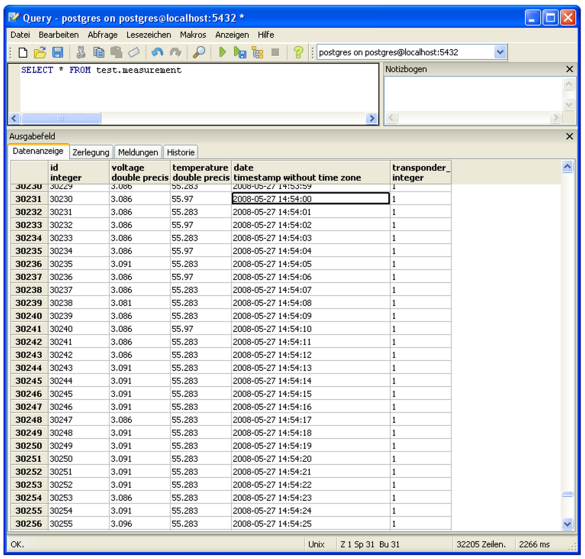

Bild 80: Auszug aus der SQL-Datenbank zum Beginn des Abkühlvorgangs

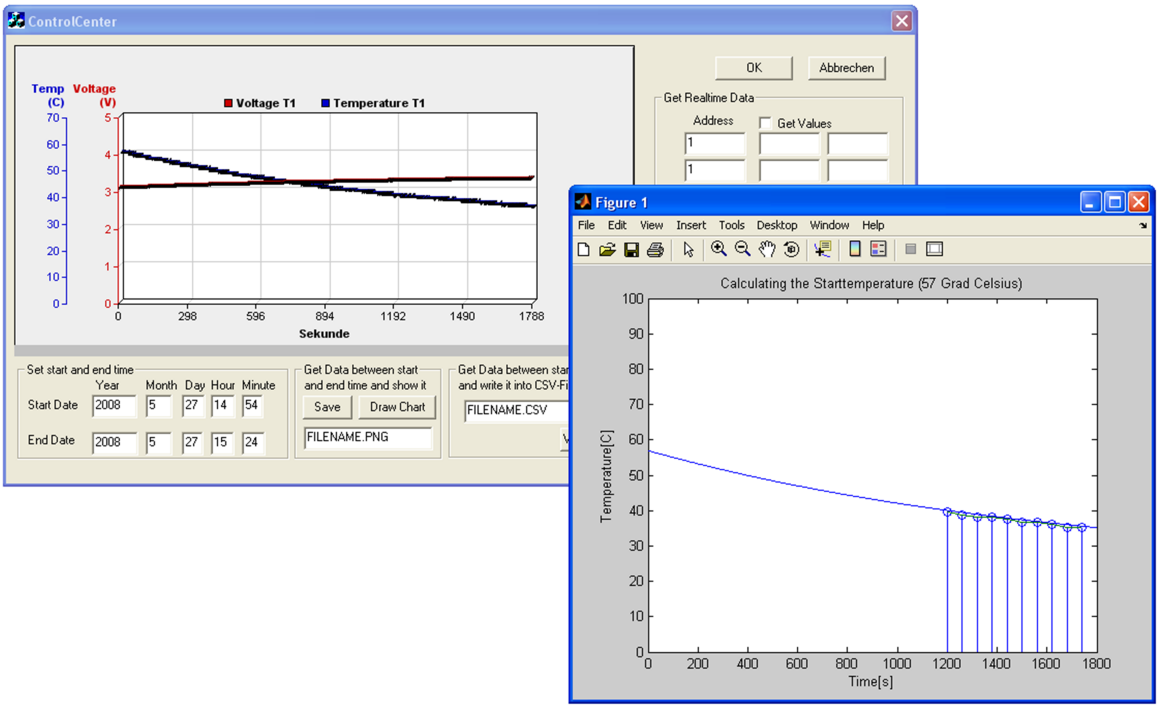

Bild 81: PC-Steuersoftware mit der Abkühlkuve aus MATLAB

gespeichert. Die CSV-Datei wird mit Hilfe der PC-Steuersoftware erstellt. Zum Darstellen der Temperaturverläufe wird die CSV-Datei mit MATLAB eingelesen und die eingelesenen Temperaturdaten werden zusammen mit dem berechneten Verlauf in einem Diagramm dargestellt. Das m-File für die Darstellung ist im Anhang A.1.6 dargestellt. In der Abbildung 82 ist der berechnete Temperaturverlauf blau dargestellt und der vom Transponder aufgenommene rot. Auf Grund der fast identischen Kurvenverläufe ist die Funktionalität des gesamten Systems nachgewiesen, da für die korrekte Berechnung alle Baugruppen im System funktionieren müssen.

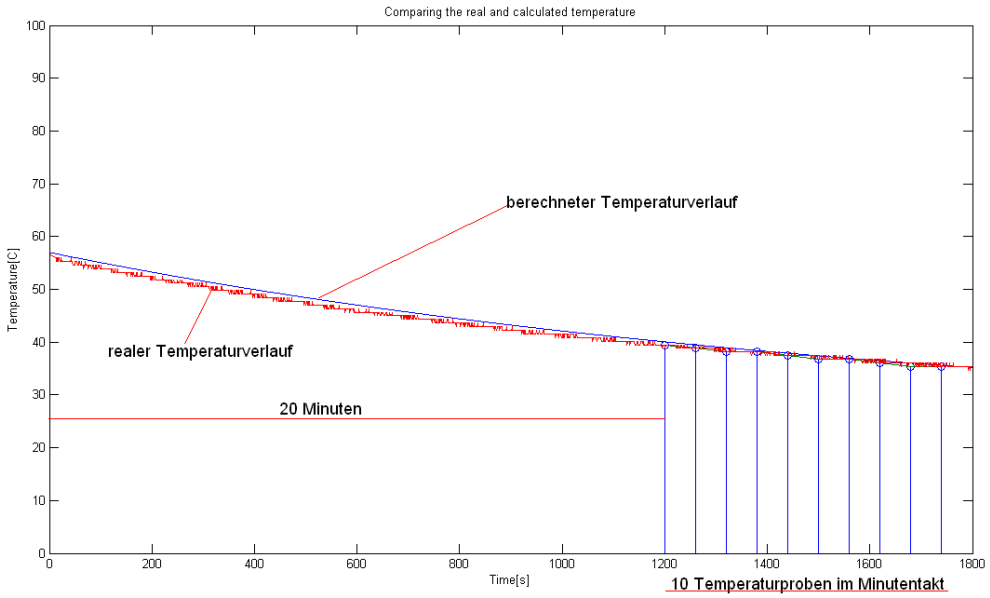

Bild 82: Vergleich des berechneten und des realen Temperaturverlaufs

7 Fazit

Zusammenfassend kann über das im Rahmen dieser Diplomarbeit entwickelte System für drahtlose Sensortechnik gesagt werden, dass alle vor Projektbeginn festgelegten Anforderungen an das System erfüllt wurden. Wie schon in den einzelnen Kapiteln vermerkt gibt es an fast jeder Komponente im System die Möglichkeit, Verbesserungen vorzunehmen. Aber auf Grund der zeitlichen Einschränkung und des komplexen Themas konnten diese Verbesserungen oder Erweiterungen zur Entwicklungszeit nicht durchgeführt oder implementiert werden. Infolge der vielen Erweiterungsmöglichkeiten wäre die Weiterführung dieses Projektes sehr wünschenswert. Insbesondere, da sich die Sensorik mit drahtloser Energie- und Datenübertragung noch im frühen Entwicklungsstadium befindet und somit die Möglichkeit gegeben ist, die Entwicklung in diesem Nieschengebiet voranzutreiben. Die Abbildung 83 zeigt die festgelegten Anforderungen und deren Umsetzung.

Da für die Implementierung dieses Systems Wissen aus den verschiedensten Fachgebieten, die von der analogen Schaltungstechnik mit Modulation und Funkübertragung über die digitale Schaltungstechnik bis zur Programmierung in verschiedenen Sprachen benötigt wird, konnte das erlernte Wissen aus dem Studium gut angewendet und erweitert werden.
Eine große Motivation stellte die eigenständige Entwicklung eines kompletten, funktionsfähigen Produktes dar, da somit die eigene Leistungsfähigkeit getestet werden konnte. Zusätzlich ist die Fertigstellung ein großes Erfolgserlebnis.
Nachfolgend werden die Verbesserungen und Erweiterungen, die im Rahmen dieser Diplomarbeit nicht durchgeführt werden konnten, aber aufgefallen sind, stichpunktartig dargestellt.

- Evaluierungsboard auf dem Reader durch eigene Entwicklung ersetzen.
- Kommunikation zwischen den Mikrocontrollern auf dem Reader ermöglichen, damit die Frequenz vom Readerfeld an die Resonanzfrequenzen von den Transpondern angepasst werden kann.
- Eine externe Hardware für die Messung des Phasenwinkels aufbauen, da auf Grund der niedrigen Taktfrequenz des Controllers für die Feldsteuerung die Auflösung bei der Messung des Phasenwinkels zu ungenau ist.
- Software für den Controller für die Feldsteuerung erweitern. (Frequenz an Resonanzfrequenz vom Transponder anpassen, extern ermittelten Phasenwinkel auswerten)
- Versorgungsspannung für die Operationsverstärker auf der Readerplatine erzeugen.

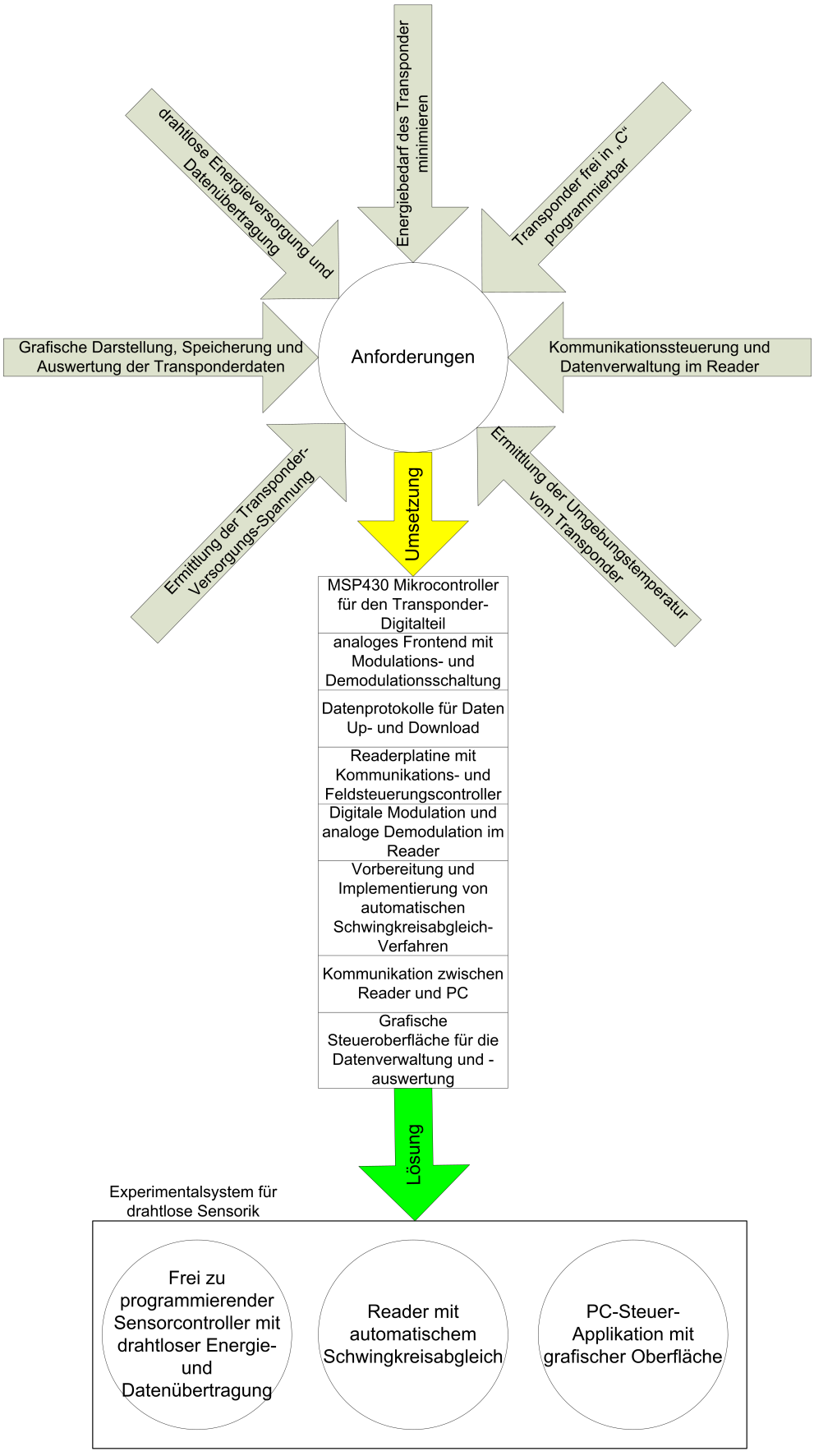

Bild 83: Anforderungen an das System und deren Umsetzungen

Literatur

[1] Texas Instruments: MSP430x12x2 Datasheet

[2] Texas Instruments: MSP430x16x Datasheet

[3] Texas Instruments: MSP430x16x User Guide

[4] Microchip: RFID Coil Design Datasheet

[5] Finkenzeller: RFID-Handbuch, 3. Auflage, Hansa-Verlag, 2002

[6] Tietze, U., Schenk, Ch.: Halbleiter - Schaltungstechnik, 12. Auflage, 2002

[7] Patrick J. Sweeney: RFID für Dummies, 1. Auflage, 2005

[8] Robert Schoblick und Gabriele Schoblick: RFID Radio Frequency Identification, 1. Auflage, 2005

[9] Lutz Bierl: Das große MSP430 Praxisbuch , 1. Auflage, 2004

[10] Matthias Sturm: Mikrocontrollertechnik. Am Beispiel der MSP 430-Familie , 2. Auflage, 2007

[11] Fieldbus Foundation: Profibus Technik

[12] A. Angermann, M. Beuschel, M. Rau, U. Wohlfarth: Matlab - Simulink - Stateflow , 5. Auflage, 2007

[13] Robert Heinemann: PSpice, Einführung in die Elektroniksimulation, 5. Auflage, 2007

Abkürzungsverzeichnis

AM ..... Amplituden Modulation

API ..... Application Programming Interface

ASK ..... Amplitude Shift Keying

FSK ..... Frequency Shift Keying

IC ..... Integrated Circuit

OOK ..... On-Off-Keying

PSK ..... Phase Shift Keying

RFID ..... Radio Frequency Identification

RISC ..... Reduced Instruction Set Computing

VCO ..... Voltage Controlled Oscillator

Bildverzeichnis

Tabellenverzeichnis

Anhang

A Quellcode

A.1 MATLAB Code

A.1.1 Darstellung der Amplitudengänge bei der AM

Listing 1: M-File für die Darstellung der Amplitudenmodulation

```
clear('t', 'Traeger', 'Data', 'Signal', 'Index');
Samples_per_Period = 20;
F_Traeger = 5000;
F_Mod = 1000;
Modulation = 0.4;

Data = [1 0 1 1 0];

F_Abtast = F_Traeger * Samples_per_Period;


%Berechnung der Abtastzeitschritte
t = 0:1/F_Abtast:50*Samples_per_Period/F_Abtast;
[M,N] = size(t);    %N ist die Länge

%Berechnung des Trägers
Traeger = sin(2*pi*F_Traeger*t);

%Berechnung des modulierten Signals
Sinal = zeros(N);
Index = 0;
for i=1:N
    Index = ceil((i / (N/5)));

    if (Data(Index) == 0)
        Signal(i) = Traeger(i) * Modulation;
    elseif(Data(Index) == 1)
        Signal(i) = Traeger(i);
    end;
end;


figure
subplot(3,1,1);
stairs(0:4,Data);
title('Daten');
axis([0 5 0 1.5])

subplot(3,1,2);
plot(t,Traeger);
title('Träger');

subplot(3,1,3);
plot(t,Signal);
title('Moduliertes Signal');
```

A.1.2 Darstellung der Feldstärke in Abhängigkeit von Radius und Distanz

Listing 2: M-File für die Darstellung der Feldstaerke

```
clear('H1', 'H2', 'H3', 'x', 'XMAX', 'RMAX', 'I', 'N', 'r', 'x_opt', 'H1Max', 'H2Max',
    'H3Max', 'H', 'text_figure1_1', 'text_figure1_2', 'text_figure1_3', 'text_figure2_1
    ', 'text_figure2_2', 'hline');

RMAX = 1000;
XMAX = 750;
x_opt_1 = 0.075;
x_opt_2 = 0.10;
x_opt_3 = 0.125;
I = 0.243;
N = 45;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Figure 1 Feldstaerke in Abhaengigkeit zum Radius
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i=1:RMAX
    r=i/1000;
    H1(i)= (I*N*power(r,2)) / (2*(sqrt( power((power(r,2) + power(x_opt_1,2)),3) )));
    H2(i)= (I*N*power(r,2)) / (2*(sqrt( power((power(r,2) + power(x_opt_2,2)),3) )));
    H3(i)= (I*N*power(r,2)) / (2*(sqrt( power((power(r,2) + power(x_opt_3,2)),3) )));
end

% Maximum ermitteln
H1Max = find(H1 == max(H1));
H1Max = H1Max(1);

H2Max = find(H2 == max(H2));
H2Max = H2Max(1);

H3Max = find(H3 == max(H3));
H3Max = H3Max(1);


clear('r');
r = 1:1:RMAX;
plot(r,H1,r,H2,r,H3), grid on
title('Feldstärke in Abhängigkeit des Spulenradius')
ylabel('H/(A/m)')
xlabel('r/mm')
l1 = strcat('x =', num2str(x_opt_1*100), 'cm' );
l2 = strcat('x =', num2str(x_opt_2*100), 'cm' );
l3 = strcat('x =', num2str(x_opt_3*100), 'cm' );
legend(l1,l2,l3,0);
text_figure1_1 = strcat('| Max H = ', num2str(H1(H1Max)), 'A/m bei r = ', num2str(r(
    H1Max)/10.0), 'cm')
text(r(H1Max), H1(H1Max),text_figure1_1,'FontSize',13)
text_figure1_2 = strcat('| Max H = ', num2str(H2(H2Max)), 'A/m bei r = ', num2str(r(
    H2Max)/10.0), 'cm')
text(r(H2Max), H2(H2Max),text_figure1_2,'FontSize',13)
text_figure1_3 = strcat('| Max H = ', num2str(H3(H3Max)), 'A/m bei r = ', num2str(r(
    H3Max)/10.0), 'cm')
text(r(H3Max), H3(H3Max),text_figure1_3,'FontSize',13)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Figure 2 Feldstaerke in Abhaengigkeit zur Distanz zwischen Readerspule
```

```
53 %% und Transponder
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55 r1 = x_opt_1 * sqrt(2);
56 r2 = x_opt_2 * sqrt(2);
57 r3 = x_opt_3 * sqrt(2);
58
59 clear('x', 'i', 'H1', 'H2', 'H3', 'H1Max', 'H2Max', 'H3Max');
60 for i=1:XMAX
61     hline(i) = 1.12;
62     x=i/1000;
63     H1(i)= (I*N*power(r1,2)) / (2*(sqrt( power((power(r1,2) + power(x,2)),3) )));
64     H2(i)= (I*N*power(r2,2)) / (2*(sqrt( power((power(r2,2) + power(x,2)),3) )));
65     H3(i)= (I*N*power(r3,2)) / (2*(sqrt( power((power(r3,2) + power(x,2)),3) )));
66     if(H1(i) > H2(i))
67         SP_H1_H2 = i;
68     end
69     if(H2(i) > H3(i))
70         SP_H2_H3 = i;
71     end
72 end
73
74 clear('x');
75 x = 1:1:XMAX;
76 figure(2);
77 plot(x,H1,x,H2,x,H3, x, hline), grid on
78 zoom on;
79 title('Feldstärke in Abhängigkeit vom Spulenabstand')
80 ylabel('H/(A/m)')
81 xlabel('x/mm')
82 l1 = strcat('r =', num2str(r1*100), 'cm' );
83 l2 = strcat('r =', num2str(r2*100), 'cm' );
84 l3 = strcat('r =', num2str(r3*100), 'cm' );
85 legend(l1,l2,l3,0);
86
87 text_figure2_1 = '';
88 text_figure2_1 = strcat('| H = ', num2str(H1(SP_H1_H2)), 'A/m bei x = ', num2str(x(
       SP_H1_H2)/10.0), 'cm')
89 text(x(SP_H1_H2), H1(SP_H1_H2),text_figure2_1,'FontSize',13)
90
91 text_figure2_2 = '';
92 text_figure2_2 = strcat('| H = ', num2str(H2(SP_H2_H3)), 'A/m bei x = ', num2str(x(
       SP_H2_H3)/10.0), 'cm')
93 text(x(SP_H2_H3), H2(SP_H2_H3),text_figure2_2,'FontSize',13)
94
95 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
96 %% Figure 3 Feldstaerke in Abhaengigkeit zum Radius und Distanz zwischen
97 %% Readerspule und Transponder
98 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
99
100 clear('x', 'r', 'H','i', 'n');
101
102 [x,r] = meshgrid([0:1:10]);
103
104 for i=10:50
105     for n=10:50
106         r = (60-i) / 100;
107         x = (60-n) / 100;
108         H(i,n) = (I*N*power(r,2)) / (2*sqrt( power((power(r,2)+power(x,2)),3) ));
109     end
110 end
111
```

```
figure(3);
surf(H)
grid on
```

A.1.3 Darstellung der Amplitudengänge beim OOK des Readerfeldes

Listing 3: M-File für die Darstellung der Signalverläufe beim OOK des Readerfeldes

```
% m-File zur Darstellung der Signale beim On-Off-Keying im Reader

clear('t', 'Traeger', 'Data', 'Signal', 'Signal_inv', 'Index');
Samples_per_Period = 100;
F_Traeger = 100;
F_Abtast = 10000;
Anz_Perioden = 10;
Modulation = 0; % entspricht 100%

Data = [1 0 1 0];




%Berechnung der Abtastzeitschritte
t = 0:1/F_Abtast:Anz_Perioden*Samples_per_Period/F_Abtast;
[M,N] = size(t)    %N ist die Länge

%Berechnung des Trägers
Traeger = ((square(sin(2*pi*F_Traeger*t)) + 1) / 2);

%Berechnung der invertierten Trägers
for i=1:N
    if(Traeger(i) == 1)
        Traeger_inv(i) = 0;
    else
        Traeger_inv(i) = 1;
    end;
end;

%Berechnung des modulierten Signals
Sinal = zeros(N);
Index = 0;
for i=1:N
    Index = ceil((i / (N/3)));

    if (Data(Index) == 0)
        Signal(i) = Traeger(i) * Modulation;
        Signal_inv(i) = Traeger_inv(i) * Modulation;
    elseif(Data(Index) == 1)
        Signal(i) = Traeger(i);
        Signal_inv(i) = Traeger_inv(i);
    end;

end;

% Ausgabe der Diagramme
figure
subplot(4,1,1);
stairs(0:3,Data);
title('serielle Daten [1 0 1] vom Mikrocontroller 1 (Datenübertragung)');
```

```
52 axis([0 3 0 1.5])
53
54 subplot(4,1,2);
55 plot(t,Traeger);
56 title('Rechtecksignel variabler Frequenz vom Mikrocontroller 2 (Schwingkreissteuerung)'
       );
57 axis([0 0.1 0 1.5])
58
59 subplot(4,1,3);
60 plot(t,Signal);
61 title('Ausgangssignal Treiberbaustein 1');
62 axis([0 0.1 0 1.5])
63
64 subplot(4,1,4);
65 plot(t,Signal_inv);
66 title('Ausgangssignal Treiberbaustein 2');
67 axis([0 0.1 0 1.5])
```

A.1.4 Darstellung der Amplitudengänge mit Resonanzabweichung

Listing 4: M-File für die Darstellung der Amplitudengänge mit Resonanzabweichung

```
1  % mFile zur Darstellung der Amplitudengänge für abgeglichene und
2  % nichtabgeglichen Systeme (schmal-/breitbandig)
3
4  %Grundkurve berechnen
5  x1 = -10:0.01:9.99;
6  size(x1)
7  amplitude_reader = exp(-x1.*x1/20)/1.267;       %schmal 10)/1
8  amplitude_transponder = exp(-x1.*x1/15)/1.267;  %schmal  5)/1
9
10 % Berechnung der Amplitudenverläufe für abgeglichene Schwingkreise
11 x = 100000:10:149990;
12 y1 = zeros(1,5000);
13 y2 = zeros(1,5000);
14
15 start_index_y1 = (125000 - 99990) /10 -1000;
16 start_index_y2 = (125000 - 99990) /10 -1000;
17
18 for i=1:2000
19     y1(i+start_index_y1) = amplitude_reader(i);
20     y2(i+start_index_y2) = amplitude_transponder(i);
21 end;
22
23 figure(1)
24
25 subplot(2,2,1);
26 plot(x, y1, x, y2);
27 title({'Amplitudenverläufe des Reader- und Transponderschwingkreise' ,'Reader: fo=125
       kHz','Transponder: fo=125kHz'})
28 legend('Reader', 'Transponder');
29 xlabel('Frequenz / Hz');
30 ylabel('Spannung');
31 axis([100000 150000 0 1])
32
33 % Produkt aus Readerspannung und Transponderspannung bilden und Integral
34 % aufnehmen
35
36 integral = zeros(1,1);
```

```
37 produkt = zeros(1,5000);
38 for i=1:5000
39     produkt(i) = y1(i) * y2(i);
40     integral = integral + produkt(i);
41 end;
42 integral_text = sprintf('Integral (Energie am Transponder: %.2f)', integral);
43 subplot(2,2,3);
44 plot(x, produkt);
45 title({'Produkt der Amplitudengänge', integral_text});
46 xlabel('Frequenz / Hz');
47 ylabel('Energie');
48 axis([100000 150000 0 1])
49
50 % Berechnung der Amplitudenverläufe für NICHT abgeglichene Schwingkreise (Transponder:
       fo=120kHz)
51 start_index_y1 = (125000 - 99990) /10 -1000;
52 start_index_y2 = (120000 - 99990) /10 -1000;
53
54 y1 = zeros(1,5000);
55 y2 = zeros(1,5000);
56
57 for i=1:2000
58     y1(i+start_index_y1) = amplitude_reader(i);
59     y2(i+start_index_y2) = amplitude_transponder(i);
60 end;
61
62 subplot(2,2,2);
63 plot(x, y1, x, y2);
64 title({'Amplitudenverläufe des Reader- und Transponderschwingkreise' ,'Reader: fo=125
       kHz','Transponder: fo=120kHz'})
65 legend('Reader', 'Transponder');
66 xlabel('Frequenz / Hz');
67 ylabel('Spannung');
68 axis([100000 150000 0 1])
69
70
71
72 % Produkt aus Readerspannung und Transponderspannung bilden und Integral
73 % aufnehmen
74
75 integral = 0;
76 produkt = zeros(1,5000);
77 for i=1:5000
78     produkt(i) = y1(i) * y2(i);
79     integral = integral + produkt(i);
80 end;
81
82 integral_text = sprintf('Integral (Energie am Transponder: %.2f)', integral);
83
84 subplot(2,2,4);
85 plot(x, produkt);
86 title({'Produkt der Amplitudengänge', integral_text});
87 xlabel('Frequenz / Hz');
88 ylabel('Energie');
89 axis([100000 150000 0 1])
```

A.1.5 Darstellung der Amplitudengänge bei unterschiedlichen Readerfrequenzen

Listing 5: M-File für die Darstellung der Amplitudengänge bei unterschiedlichen Readerfrequenzen

```
% mFile zur Darstellung der Energieübertragung für verschiedene
% Readerfrequenzen
x1 = -10:0.01:9.99;
size(x1)
amplitude_reader = exp(-x1.*x1/10);
amplitude_transponder = exp(-x1.*x1/5);
%figure(1)
%plot(x1, amplitude_reader);

x = 100000:10:149990;

y1_1 = zeros(1,5000);
y1_2 = zeros(1,5000);
y1_3 = zeros(1,5000);
y2 = zeros(1,5000);


% Berechnung der Amplitudenverläufe für unterschiedliche Frequenzen
start_index_y1_1 = (120000 - 99990) /10 -1000;
start_index_y1_2 = (125000 - 99990) /10 -1000;
start_index_y1_3 = (130000 - 99990) /10 -1000;
start_index_y2 = (129000 - 99990) /10 -1000;

for i=1:2000
    y1_1(i+start_index_y1_1) = amplitude_reader(i);
    y1_2(i+start_index_y1_2) = amplitude_reader(i);
    y1_3(i+start_index_y1_3) = amplitude_reader(i);
    y2(i+start_index_y2) = amplitude_transponder(i);
end;

figure(1)

subplot(2,1,1);
plot(x, y1_1, x, y1_2, x, y1_3, x, y2);
title({'Amplitudenverläufe des Reader- und Transponderschwingkreise' ,'Reader: fo=120
    kHz-130kHz','Transponder: fo=129kHz'})
legend('Reader(120kHz)', 'Reader(125kHz)', 'Reader(130kHz)', 'Transponder');
xlabel('Frequenz / Hz');
ylabel('Spannung');
axis([100000 150000 0 1])

% Produkt aus Readerspannung und Transponderspannung bilden und Integral
% aufnehmen
integral_1 = zeros(1,1);
integral_2 = zeros(1,1);
integral_3 = zeros(1,1);

produkt_1 = zeros(1,5000);
produkt_2 = zeros(1,5000);
produkt_3 = zeros(1,5000);

for i=1:5000
    produkt_1(i) = y1_1(i) * y2(i);
    produkt_2(i) = y1_2(i) * y2(i);
    produkt_3(i) = y1_3(i) * y2(i);
```

```
55
56      integral_1 = integral_1 + produkt_1(i);
57      integral_2 = integral_2 + produkt_2(i);
58      integral_3 = integral_3 + produkt_3(i);
59  end;
60  integral_text = sprintf('Integral (Energie am Transponder: %.2f, %.2f, %.2f [323 für
        abgeglichenes System])', integral_1, integral_2, integral_3);
61  subplot(2,1,2);
62  plot(x, produkt_1, x, produkt_2, x, produkt_3);
63  title({'Produkt der Amplitudengänge', integral_text});
64  xlabel('Frequenz / Hz');
65  ylabel('Energie');
66  legend('Reader(120kHz)', 'Reader(125kHz)', 'Reader(130kHz)');
67  axis([100000 150000 0 1])
```

A.1.6 Vergleich vom realen und berechneten Temperaturverlauf

Listing 6: M-File für den Vergleich vom realen und berechneten Temperaturverlauf Readerfrequenzen

```
1   clear 'all'
2
3   %Variablen für die Berechnung anlegen
4
5   t = 0:1:(60*30);     % 30 Minuten
6   c= 0.000541525;       % wird in der Schleife variiert
7   RT=20;                % wird in der Schleife variiert
8   % Quadratischen Fehler mit großer Zahl initialisieren
9   Smalest_Error = 99999999999;
10
11  % Reale Temperaturdaten einlesen
12  real_data_read = textread('Val_abkuehl.CSV', '', 'delimiter', ',', 'emptyvalue', NaN);
13  real_data = zeros(1,1801);
14  for i=1:1801
15     real_data(i) = real_data_read(i);
16  end;
17
18  % zum einlesen von Temperaturdaten aus einer CSV-Datei
19  data = zeros(1,11);
20  data = textread('temperature_data.csv', '', 'delimiter', ',','emptyvalue', NaN);
21
22  % Temperaturdaten Kopieren
23  Temperature_Samples = zeros(1,10);
24  for i=1:10
25     Temperature_Samples(i) = data(i);
26  end;
27
28  % Startindex wird momentan fest auf 22 Minuten gelegt
29  %Start_Index = 60 * data(11);
30
31  % Indexe für die Temperaturproben berechnen
32  Start_Index = 20 * 60;
33  Index_Measurements = Start_Index:60:(Start_Index + 540);
34
35
36  % In den folgenden 3 Schleifen werden die 3 Parameter
37  % Starttemperatur, RT und c variiert und mit diesen Parametern
38  % ein Temperaturverlauf nach der e-Funktion berechnet.
```

```
39 % Der Temperaturverlauf wird mit den Temperaturproben verglichen
40 % und der quadratische Fehler berechnet. Für die Parameter bei denen
41 % der quadratische Fehler am geringsten ist wird der Temperaturverlauf
42 % und die Starttemperatur berechnet.
43
44 for j = 1:10
45     c = j/10000;
46     for n = 1:50
47         RT = n;
48         for i = 20:200
49             StartTemperature_switch = i;
50             Temperature = (StartTemperature_switch-RT)*exp(-c*t)+RT;
51             Error = 0;
52             for n=1:10
53                 Error = Error + power((Temperature(Start_Index + 60*n-1) -
                       Temperature_Samples(n)),2);
54             end;
55             if(Error < Smalest_Error)
56                 Smalest_Error = Error;
57                 Start_Temperature = i;
58                 RT_min = RT;
59                 c_min = c;
60             end;
61         end;
62     end;
63 end;
64
65
66 % Darstellung des berechneten Temperaturverlaufes und der Temperatur-
67 % proben.
68 Temperature = (Start_Temperature-RT_min)*exp(-c_min*t)+RT_min;
69 figure(1)
70 stem(Index_Measurements, Temperature_Samples);
71 hold on
72 plot(t, Temperature, Index_Measurements, Temperature_Samples, t, real_data);
73 hold off
74 axis([0 1800 0 100]);
75 Title_Text = sprintf('Comparing the real and calculated temperature');
76 title(Title_Text)
77 xlabel('Time[s]');
78 ylabel('Temperature[C]');
```

A.2 Firmware Transponder

A.2.1 main.c

Listing 7: main.c

```
1 /*****************************************************************************/
2 /* Projekt: Transponderquellcode                                             */
3 /* Dateiname: main.c                                                         */
4 /* Autor: Tobias Krannich                                                    */
5 /* Kurzbeschreibung:      Die komplette Firmeware vom Transponder.           */
6 /*                              Der Code beinhaltet die folgenden Funktionen */
7 /*                              1. Steuerdaten mit Adresse einlesen          */
8 /*                              2. Temperatur und Sensorspannung einlesen    */
9 /*                              3. Daten codieren und Chacksumme anhängen    */
```

```
10 /*                              4. Seriellen Datenstrom ausgeben                */
11 /*                                                                              */
12 /********************************************************************************/
13
14
15
16
17 // Defines
18 #define CRC32POLY 0x1021 // CRC-16 Polynom
19
20 // Wartezeit zwischen den einzelnen Bits für das serielle Senden der Daten
21 // Wartezeit + Setzen des Ausgangs entspricht 500us
22 #define MY_WAIT   nop();      nop();      nop();      nop();
23                   nop();      nop();      nop();      nop();
24                   nop();      nop();      nop();      nop();
25                   nop();      nop();      nop();      nop();
26                   nop();      nop();      nop();      nop();
27                   nop();      nop();      nop();      nop();
28                   nop();      nop();      nop();      nop();
29                   nop();      nop();      nop();      nop();
30                   nop();      nop();      nop();
31
32 // Wartezeit zwischen den einzelnen Bits, wenn die seriellen Daten in
33 // einer Schleife ausgegeben werden.
34 // Wartezeit + Schleifenaufruf + Setzen des Ausgangs = 500us
35 #define MY_WAIT_SHORT    nop();      nop();      nop();      nop();
36                          nop();      nop();      nop();      nop();
37                          nop();      nop();      nop();      nop();
38                          nop();      nop();      nop();      nop();
39                          nop();      nop();      nop();
40 #define MY_ADDRESS (0x02)
41
42
43 // Includes
44 #include <msp430x12x2.h>      // Register, ... für den MSP430F1232
45 #include <signal.h>           // Für die Interrupts
46
47
48 // Prototypen (die Beschreibung der Funktionen erfolgt bei den einzelen
49 // Implementierungen)
50 void init_System(void);
51 void init_SendVector(void);
52 void get_Address(void);
53 unsigned short get_Temperature(void);
54 unsigned short get_Vss(void);
55 void build_DataVector((void);
56 void calc_crc16(unsigned char);
57 unsigned short calc_crc16Vector(unsigned char[], unsigned char);
58 void build_CRC16Vector(void);
59 void build_SendVector(void);
60 void send_SendcVector(void);
61
62
63 // Globale Variablen (Daten die im Interrupt gestzt werden als volatile deklarieren)
64 volatile unsigned char Command;           // Vom Reader gesendetes Steuerkommando wird
       in
65                                           // Command gespeichert
66 unsigned short Temperatur = 0;            // Aufgenommene Temperatur
67 unsigned short AD_Wert    = 0;            // Aufgenommene Sensorspannung
68 unsigned char DatenVektor[32];            // Temperatur und Spannung in Bits zerlegt
69                                           // (MSB bei Index = [0])
```

```
70  unsigned char CRC16Vektor[16];              // Ckecksumme in Bits zerlegt
71  unsigned char SendeVektor[108];             // Kompletter Sendevektor in Bits zerlegt
72                                              // (16 + 32)*2 + 8 + 4 = 108
73  unsigned short SendeCRC16;                  // CRC16 Checksumme
74  unsigned short crc16;                       // CRC16 Schieberegister
75  volatile unsigned char ReceiveReady;        // Flag gibt an, dass Empfang beendet ist
76  volatile unsigned char Address;             // Vom Reader gesendete Adresse
77  volatile unsigned char AddressRight;        // Flag, dass angibt, ob die eigene Adresse
78                                              // mit der vom Reader gesendeten Adresse
79                                              // übereinstimmt (bin ich angesprochen?)
80  volatile unsigned char DataIndex;           // Feldindex für den Datenempfang
81
82
83  // Die Interrupt Service Routine wird aktiviert, wenn der Transponder bereit
84  // ist Steuerdaten vom Reader zu empfangen. Es wird die Zeit zwischen zwei
85  // steigenden Flanken gemessen und auf Grund dieser Zeiten und den vorherig
86  // gemessenen Zeiten werden die manchester Codierten Daten empfangen und dekodiert.
87  // Die empfangenen Daten beinhalten die vom Reader angesprochene Adresse und
88  // ein Kommando. Diese Daten werden in die globalen Variablen Address und Command
89  // gespeichert.
90  interrupt(PORT1_VECTOR) isr_Port1_1(void)
91  {
92        unsigned short Time = 0;                  // Speichert Zeit zwischen 2 steigenden
93                                                  // Flanken
94        static unsigned short Data[9] = {0};      // Zwischenspeicher für die empfangenen
95                                                  // Zeichen
96        static unsigned char Bit = 0;             // Für die Ermittlung der Datenbts
97                                                  // (speichert Information über das
98                                                  // vorige Datenbit)
99        unsigned char i = 0;                      // Für for()
100
101       Time = TAR;                               // Zeit wird aufgenommen
102       TAR = 0;                                  // Zählregister zurücksetzen
103
104
105       if(DataIndex == 0)                        // Erste steigende Flanke?
106       {
107             TACTL |=  MC1 ;                     // Starte Timer im upMode
108             Bit = 1;                            // Für die Manchesterdecodierung
109                                                 // auf 1 setzen
110             DataIndex++;                        // Datenindex erhöhen
111             DataIndex++;                        // Datenindex erhöhen
112       }
113       else                                      // Einleseforgang ist gestartet?
114       {
115             if(Time < 70)                       // Zeit < 7ms? Keine gültigen Daten ->
116             {                                   // Errorbehandlung einleiten
117                   TACTL &=~ (MC0 + MC1);        // Stoppe Timer
118                   TAR = 0;                      // Zählregister = 0
119                   P1IE  &=~  BIT0;              // Interrupt sperren
120                   DataIndex = 0;                // Datenindex zurücksetzen
121                   ReceiveReady = 2;             // Empfangen mit Fehler abbrechen
122                   Address = 0;                  // Keine gültige Adresse empfangen
123             }
124             else if(Time < 125)                 // 2 kurze Bits?
125             {
126                   Data[DataIndex-1] = Bit;      // Voriges Datenbit übernehmen
127                   DataIndex++;                  // Datenindex hochsetzen
128             }
129             else if(Time < 175)                 // 1 kurzes 1 langes Bit?
130             {
```

```
131                 if(Bit == 1)                    // Voriges Datenbit ist 1
132                 {
133                       Data[DataIndex - 1] = Bit; // Voriges Datenbit übernehmen
134                       DataIndex++;               // Datenindex hochsetzen
135                       Bit ^= 0x01;               // Bit für die Berechnung des
136                                                  // nächsten Datenbits vorbereiten
137                 }
138                 else
139                 {
140                       Data[DataIndex - 1] = Bit; // Voriges Datenbit übernehmen
141                       DataIndex++;               // Datenindex hochsetzen
142                       Bit ^= 0x01;               // Bit für die Berechnung des
143                                                  // nächsten Datenbits vorbereiten
144                       Data[DataIndex - 1] = Bit; // Datenbit übernehmen
145                       DataIndex++;               // Datenindex hochsetzen
146                 }
147           }
148           else if(Time < 225)                    // 2 lange Bits
149           {                                      // Datenbits einlesen und das
150                                                  // Bit für die Berechnung anpassen
151                       Data[DataIndex - 1] = Bit;
152                       DataIndex++;
153                       Bit ^= 0x01;
154                       Data[DataIndex - 1] = Bit;
155                       DataIndex++;
156                       Bit ^= 0x01;
157           }
158           else                                   // Zeit > 22.5ms? Keine gültigen Daten
159           {                                      // Errorbehandlung einleiten
160                 TACTL &=~ (MC0 + MC1);           // Stoppe Timer
161                 TAR = 0;                         // Zähler zurücksetzen
162                 P1IE  &=~  BIT0;                 // Interrupt sperren
163                 DataIndex = 0;                   // Datenindex zurücksetzen
164                 ReceiveReady = 3;                // Empfangen mit Fehler abbrechen
165                 Address = 0;                     // Keine gültige Adresse empfangen
166           }
167     }
168
169     if(DataIndex >= 8)                  // Daten vollständig empfangen?
170     {
171           TACTL &=~ (MC0 + MC1);        // Stoppe Timer
172           TAR = 0;                      // Zähler zurücksetzen
173           P1IE  &=~  BIT0;              // Interrupt sperren
174           DataIndex = 0;                // Datenindex zurücksetzen
175           ReceiveReady = 1;             // Empfangen erfolgreich abgeschlossen
176           Address = 0;                  // Address auf 0 setzen, damit die
177                                         // Datenbits, die die Adresse darstellen
178                                         // in die Variable geschoben werden können
179           for(i=0; i<4; i++)            // Datenbits 1-4 in die Adressvariable
180                                         // schieben
181           {
182                 Address |= (Data[i+1]) << (3-i);
183                 Data[i+1] = 0;
184           }
185           Command = 0;              // Command auf 0 setzen, damit die
186                                     // Datenbits, die das Kommando darstellen
187                                     // in die Variable geschoben werden können
188
189           // Datenbits 5-6 in die Kommandovariable schieben
190           Command = (Data[6] << 1) | Data[5];
191     }
```

```
192       P1IFG &=~ BIT0;                           // reset ISR-Flag
193
194 }
195
196
197 // Dieser Interrupthandler wird ausgeführt, wenn ein Empfang gestartet wurde, aber
198 // innerhalb von 50ms keine weitere steigende Flanke erkannt wird. Dies ist notwendig,
199 // damit bei falsch übertragenden Signalen die Errorroutine eingeleitet werden kann
200 // und auf das nächste gültige Signal gewartet wird. Die ISR stellt den Grundzustand
201 // zum empangen von Daten her und bricht das aktuelle Empfangen mit einen Fehler ab.
202 interrupt(TIMERA0_VECTOR) isr_TimerA0(void)
203 {
204       TACTL &=~ (MC0 + MC1);    // Stoppe Timer
205       TAR = 0;                  // Zähler zurücksetzen
206       P1IE  &=~  BIT0;          // Interrupt sperren
207       DataIndex = 0;            // Datenindex zurücksetzen
208       ReceiveReady = 4;         // Empfang mit Fehler 4 abbrechen
209       Address = 0;              // Keine gültige Adresse empfangen
210 }
211
212
213 // In der Hauptfunktion werden zunächst das System und die Variable
214 // initialisiert. Anschließend läuft das Programm in der Endlos-
215 // Hauptschleife. Zu Beginn der Schleife wartet das Programm auf den
216 // Empfang von Readerdaten. Ist der Empfang erfolgreich abgeschlossen
217 // und die Empfangene Adresse stimmt mit der eigenen überein werden
218 // die folgenden Punkte abgearbeitet:
219 //       - Temperatur über die interne Diode bestimmen
220 //       - Betriebsspannung über den ADC einlesen
221 //       - Datenvektor aus Temperatur und Spannung erstellen
222 //       - Checksumme über den Datenvektor bilden
223 //       - Sendevektor erstellen
224 //            + [(Preamble + Daten + CRC16) Manchester codieren]
225 //       - Daten seriell über einen Port ausgeben
226 int main(void)
227 {
228       DataIndex = 0;                          // Datenindex für das erste
229                                               // Empfangen auf 0 setzen
230       init_System();                          // System intialisieren (Register, ...)
231       init_SendVector();                      // Datenvektor vorbereiten
232                                               // (Preamble, Stopbits)
233       while(1)                                // Haupt-Endlos-Schleife
234       {
235             get_Address();                    // Empfangen der Daten Starten
236                                               // Adresse und Kommando in der
237                                               // isr_Port1_1 decodieren
238                                               // Empfangende Adresse mit eigener
239                                               // vergleichen
240             if(AddressRight == 1)             // Eigene Adresse empfangen?
241             {
242                   Temperatur = get_Temperature();  // Temperatur ermitteln
243                   AD_Wert = get_Vss();             // Betriebsspannung ermitteln
244                   build_DataVector();              // Datenvektor aus Spannung und
245                                                    // Temperatur bilden
246
247                   // Checksumme über den Datenvektor bilden;
248                   SendeCRC16 = calc_crc16Vector(DatenVektor, 32);
249                   build_CRC16Vector();       // CRC16 Checksumme in Bits zerlegen
250                   build_SendVector();        // Sendevektor erstellen
251                                              // [(Preamble + Daten + CRC16)
252                                              // Manchester codieren]
```

```
253                     send_SendVector();          // Sendevektor über Port ausgeben
254                 }
255         }
256         return 0;
257 }
258
259
260 ////////////////////////////////////////////////////////////
261 //////////////Funktionen////////////////////////////////////
262 ////////////////////////////////////////////////////////////
263
264 // Die Funktion init_system nimmt die Initialisierung des
265 // Systems über das setzen der entsprechende Register vor.
266 void init_System(void)
267 {
268
269         // Watchdogtimer ausschalten
270         WDTCTL = WDTPW + WDTHOLD;
271
272         // Den internen Taktgenerator (DCO) auf 80kHz setzen
273         // RSEL=0 DCO=0 MOD=0 DCOR=0
274         DCOCTL = 0;
275         BCSCTL1 = 0;
276         BCSCTL2 = 0;
277
278         // Modulationsausgang (TX)
279         P3SEL &=~ 0x40;
280         P3DIR |=  0x40;
281         P3OUT  =  0x00;
282
283         // Dateneingang (RX)
284         P1SEL &=~ BIT0;
285         P1DIR &=~ BIT0;
286
287
288         // Initialisierung des Timers
289         TACTL = TASSEL1 + TAIE;             // SMCLK,  clear TAR
290         TACCTL0  = CCIE;                    // Interrupt enabled
291         TACTL |= ID0 | ID1;                 // Divider = 8 -> 10kHz
292         TACCR0 = 500;                       // Vergleichsregister setzen (50ms)
293 }
294
295
296 // Die Funktion get_Temperature ermittelt die Temperatur über eine
297 // interne Diode und gibt sie als 16 Bit Rückgabewert zurück, wobei
298 // 0xFFFF 200 Grad Celsius entspricht.
299 // T = Temperatur * 200/0xFFFF
300 unsigned short get_Temperature(void)
301 {
302         unsigned short AD_Wert = 0;         // Spannung über der Diode
303         unsigned short Temperatur = 0;      // Rückgabewert für die
304                                             // Temperatur
305
306         // Diodenspannung am ADC anlegen
307         ADC10CTL1 = INCH_10;
308
309         // /*VREF = 2.5V*/ 1,5V, ADC anschalten, Sample- and Holdtime = 640 ADC_CLK's,
310         // Interne Referenzspannung benutzen
311         ADC10CTL0 = /*REF2_5V*/ + REFON + ADC10ON + ADC10SHT_3 + SREF_1;
312
313         // ADC freigeben
```

```
314         ADC10CTL0 |= ENC;
315
316         // Starte AD Umsetzung
317         ADC10CTL0 |= ADC10SC;
318
319         // Warten bis Umsetzung abeschlossen ist und Spannung übernehmen
320         while ((ADC10CTL1 & ADC10BUSY) == 1);
321         AD_Wert = ADC10MEM;
322
323         ADC10CTL0 &=~ ENC;          // ADC ausschalten. ENC muss zuerst und
324                                     // alleine gelöscht werden
325         ADC10CTL0 = 0;              // sonst schaltet der ADC nicht richtig
326                                     // ab ("verbraucht" Strom)
327         ADC10CTL1 = 0;
328
329         //                            1.5V
330         //Temperatur_f = ((AD_Wert * 2.5 / 1024.00) - 0.986 ) / 0.00355;
331         //Temperatur = (unsigned short)(65535.00 * Temperatur_f / 200.00);
332
333
334         // die obere Rechnung ausmultipliziert
335         // sonst braucht der so viel Speicher und
336         // überschreibt globale Variablen
337         // Temperatur = 225*AD_Wert - 91010;  // V_REF = 2,5V
338             Temperatur = 135*AD_Wert - 91010; // V_REF = 1,5V
339
340
341         // Temperatur auf 200 Grad Celsius skaliert zurückgeben
342         return Temperatur;
343 }
344
345
346 // Die Funktion init_SendVector setzt 4 Einsen als Preamble zu Beginn
347 // des SendVector und fügt am Ende des SendVector 2 Stopbits ein
348 void init_SendVector(void)
349 {
350         SendeVektor[0]=0x40;
351         SendeVektor[1]=0x00;
352         SendeVektor[2]=0x40;
353         SendeVektor[3]=0x00;
354         SendeVektor[4]=0x40;
355         SendeVektor[5]=0x00;
356         SendeVektor[6]=0x40;
357         SendeVektor[7]=0x00;
358
359
360         SendeVektor[104]=0x40;
361         SendeVektor[105]=0x00;
362         SendeVektor[106]=0x40;
363         SendeVektor[107]=0x00;
364 }
365
366
367 // Die Funktion get_Vss ermittelt die Betriebsspannung des
368 // MSP430F1232 und gibt sie als 16 Bit Rückgabewert zurück.
369 // V = V_ss * 2.5/0xFFFF
370 // Durch das Umschalten des ADC Kanals kann auch die externe
371 // Sensorspannung ermittelt werden. Es muss evt. noch das
372 // Spannungsteilerverhältnis berücksichtigt werden
373 unsigned short get_Vss(void)
374 {
```

```
375         unsigned short AD_Wert = 0;         // ADC Registerwert
376         unsigned short Vss = 0;             // Betriebsspannung auf 2.5V skaliert
377
378         // Betriebsspannung an den ADC anlegen
379         ADC10CTL1 = INCH_11;      // INCH_11: VCC INCH_2 V_RC
380
381         // /*VREF = 2.5V*/ 1,5V, ADC anschalten, Sample- and Holdtime = 640 ADC_CLK's,
382         // Interne Referenzspannung benutzen
383         ADC10CTL0 = /*REF2_5V*/ + REFON + ADC10ON + ADC10SHT_3 + SREF_1;
384
385         // ADC freigeben
386         ADC10CTL0 |= ENC;
387
388         // Starte AD Umsetzung
389         ADC10CTL0 |= ADC10SC;
390
391         // Warten bis Umsetzung abeschlossen ist und Spannung übernehmen
392         while ((ADC10CTL1 & ADC10BUSY) == 1);
393         AD_Wert = ADC10MEM;
394
395         ADC10CTL0 &=~ ENC;          // ADC ausschalten. ENC muss zuerst und
396                                     // alleine gelöscht werden
397         ADC10CTL0 = 0;              // sonst schaltet der ADC nicht richtig
398                                     // ab ("verbraucht" Strom)
399         ADC10CTL1 = 0;
400
401         //                  1.5V
402         //Vss_f = AD_Wert * 2.5 * 2 / 1024;
403         //Vss = (unsigned short)(65535.00 * Vss_f / 5);
404
405         Vss = AD_Wert * 38;
406         //Vss = AD_Wert * 64;         // die obere Rechnung ausmultipliziert
407                                     // sonst braucht der so viel Speicher und
408                                     // überschreibt globale Variablen
409
410         // Betriebsspannung auf 5V skaliert zurückgeben
411         return Vss;
412 }
413
414
415 // Die Funktion build_DataVector erzeugt ein 32 Byte langes
416 // Feld, bei dem jedes Byte einem Bit von der Temperatur oder
417 // der Spannung entspricht. In den unteren 16 Byte liegt der
418 // Temperaturwert und in den oberen 16 Byte der Spannungswert
419 // Das MSB vom Temperaturwert liegt bei Index = 0
420 // Das MSB vom Spannungwert liegt bei Index = 16
421 void build_DataVector(void)
422 {
423         int i = 0;
424
425         for(i=0; i<16; i++)
426         {
427                 DatenVektor[i] = (Temperatur >> (15-i)) & 0x01;
428         }
429         for(i=0; i<16; i++)
430         {
431                 DatenVektor[i+16] = (AD_Wert >> (15-i)) & 0x01;
432         }
433
434 }
435
```

```
436
437 // Die Funktion calc_crc16 berechnet die Ckecksumme für die
438 // übergebenen Bits. Das Ergebnis der CRC16 bildet sich
439 // in dem Schieberegister crc16
440 void calc_crc16(unsigned char bit)
441 {
442       int hbit;
443
444     hbit=(crc16 & 0x8000) ? 1 : 0;
445     if (hbit != bit)
446         crc16=(crc16<<1) ^ CRC32POLY;
447     else
448         crc16=crc16<<1;
449 }
450
451 // Die Funktion calc_crc16Vector ruft die Funktion calc_crc16 in einer
452 // Schleife auf und übergiebt ihr die 32 Datenbits. Anschließend
453 // wird die Chacksumme invertiert und zurückgegeben
454 unsigned short calc_crc16Vector(unsigned char Data[], unsigned char lenght)
455 {
456       int i;
457       crc16 = 0xFFFF;
458       for (i=0; i<lenght; ++i)
459       {
460         calc_crc16(Data[i]);
461       }
462       crc16 ^= 0xFFFF;
463
464       return crc16;
465 }
466
467 // Die Funktion build_CRC16Vector teilt die 16 Bit Checksumme
468 // auf ein 16 Byte großes Feld auf
469 void build_CRC16Vector(void)
470 {
471       int i = 0;
472
473       for (i=0; i<16; ++i)
474       {
475             CRC16Vektor[i] = (SendeCRC16 >> (15-i)) & 0x01;
476       }
477 }
478
479 // Die Funktion build_SendVector codiert die Daten (Temperatur, Spannung)
480 // und die Checksumme in Manchestercode und Speichert sie im SendVector
481 void build_SendVector(void)
482 {
483       int i = 0;
484
485       for(i=0; i<32; i++)                    // Daten (Temperatur, Spannung)
486       {
487             if(DatenVektor[i] == 1)
488             {
489                   SendeVektor[(2*i) + 8]=0x40;
490                   SendeVektor[(2*i+1) + 8]=0x00;
491             }
492             else
493             {
494                   SendeVektor[(2*i) + 8]=0x00;
495                   SendeVektor[(2*i+1) + 8]=0x40;
496             }
```

```
497         }
498
499         for(i=0; i<16; i++)                    // CRC16
500         {
501               if(CRC16Vektor[i] == 1)
502               {
503                     SendeVektor[(2*i) + 72]=0x40;
504                     SendeVektor[(2*i+1) + 72]=0x00;
505               }
506               else
507               {
508                     SendeVektor[(2*i) + 72]=0x00;
509                     SendeVektor[(2*i+1) + 72]=0x40;
510               }
511         }
512   }
513
514   // Die Funktion send_SendVector sendet zunächst ein Startbit
515   // und anschließend eine Runinsequenz. Am Ende folgen die
516   // manchester codierten Nutzdaten mit der Checksumme.
517   void send_SendVector(void)
518   {
519         int i = 0;
520
521         // Die folgenden Bits gehören noch zum Sendevektor
522         // -> noch in die Vektorinitialisierung einbringen
523
524         // Startbit (wird für das RunIn benötigt, wenn nicht
525         // genügend Störungen (steigende Flanken) auf dem
526         // Kanal sind.
527         P3OUT = 0x40;
528         MY_WAIT
529         P3OUT = 0x00;
530         MY_WAIT
531
532         // RunIn Senden
533             // Die folgenden Komentare dienen dem Verständnis
534             // der Run In Sequenz im Reader und müssen daher
535             // mit der Readersoftwere zusammen betrachtet
536             // werden
537         P3OUT = 0x40; //
538         MY_WAIT       //
539         P3OUT = 0x00; //
540         MY_WAIT       //
541         P3OUT = 0x40;    //  kurze Zeit (1) wird bei dieser steigenden erkannt
542         MY_WAIT          //
543         P3OUT = 0x40;    // case 0
544         MY_WAIT          //
545         P3OUT = 0x00;    //
546         MY_WAIT          //
547         P3OUT = 0x00;    //
548         MY_WAIT          //
549         P3OUT = 0x40;  //  lange Zeit (2) wird bei dieser steigenden erkannt
550         MY_WAIT        // -> case 1
551         P3OUT = 0x40;  //
552         MY_WAIT        //
553         P3OUT = 0x00;  //
554         MY_WAIT        //
555         P3OUT = 0x40;    // mittlere Zeit (1,5) wird bei dieser steigenden erkannt
556         MY_WAIT          // -> case 2
557         P3OUT = 0x00;    //
```

```
558       MY_WAIT          //
559       P3OUT = 0x40;  // RunInOK wird bei diese steigenden gesetzt
560       MY_WAIT        // -> case 0
561       P3OUT = 0x00;  //
562       MY_WAIT        //
563
564       // Vektor Senden
565       for(i=0; i<108; i++)
566       {
567             P3OUT = SendeVektor[i];
568             MY_WAIT_SHORT
569       }
570 }
571
572 // Die Funktion get_Address leitet die Empfangsroutine ein und
573 // wartet bie das Empfangen abgeschlossen ist. Anschließend wird
574 // geprüft, ob das Empfangen Fehlerhaft war und welcher Fehler
575 // aufgetreten ist
576 void get_Address(void)
577 {
578       ReceiveReady = 0;             // Vor dem empfangen auf 0 setzen, damit
579                                     // in der while-Schleife gewartet wird bis
580                                     // dasempfangen abgeschlossen ist
581       AddressRight = 0;             // Zurücksetzen, da es vom vorigen Empfang
582                                     // noch gesetzt sein kann
583
584       P1IE  |=  BIT0;               // Port Interrupt freigeben
585       P1IES |=  BIT0;               // Interrupt auf fallende Flanke
586       _EINT();                      // Interrupts global freigeben
587       // Warten bis Empfangen beendet
588       while(ReceiveReady == 0)
589       {
590             nop();
591       }
592       // Prüfen, ob die empfangene Adresse mit der eigenen übereinstimmt
593       if(Address == MY_ADDRESS)
594       {
595             AddressRight = 1;
596       }
597       // Hier die Fehlerauswertung einfügen (switch(ReceiveReady))
598       _DINT();                        // Interrupt global sperren
599
600 }
```

A.3 Firmware Reader Controller-Kommunikation

A.3.1 main.c

Listing 8: main.c

```
1 /*****************************************************************************/
2 /* Projekt: Readerquellcode                                                  */
3 /* Dateiname: main.c                                                         */
4 /* Autor: Tobias Krannich                                                    */
5 /* Kurzbeschreibung: In der main-Funktion des Projektes wird zunächst die    */
6 /*                   Initialisierung des Controlles vorgenommen.             */
7 /*                   Anschließend läuft das Programm in der Hauptendlos-     */
```

```
8  /*                  schleife. In der Endlosschleife wird geprüft, ob in      */
9  /*                  einem neuen Betriebsmodus gewechselt werden soll.        */
10 /*                  Anschließend wird in einen von den implementierten       */
11 /*                  Betriebsmodi gewechselt. Zur Zeit sind die folgenden     */
12 /*                  3 Betriebsarten umgesetzt. Durch den modularen Aufbau    */
13 /*                  können weitere Modi in die switch case integriert        */
14 /*                  werden.                                                  */
15 /*   Betriebsmode 1: (zyklische Abfrage)                                     */
16 /*                  Es wird zklisch ein festgelegter Adressraum von Trans-   */
17 /*                  pondern abgefragt und die zurückempfangenen Daten        */
18 /*                  werden in den externen Flash gespeichert.                */
19 /*   Betriebsmode 2: (azyklische Abfrage)                                    */
20 /*                  Es wird auf Datenanfragen von der PC-Steuersoftware      */
21 /*                  gewartet. Sendet die Stewuersoftware eine Datenanfrage   */
22 /*                  mit einer Adresse, werden die Daten von diesem Trans-    */
23 /*                  ponder abgehohlt und über die serielle Schnittstelle     */
24 /*                  an den PC geschickt.                                     */
25 /*   Betriebsmode 3: (Statusabfrage)                                         */
26 /*                  Über zuvor festgelegte Kommandos können von der Steuer-  */
27 /*                  software Daten und Information abgefragt werden.         */
28 /***************************************************************************/
29
30 // Includes
31 #include <msp430x16x.h>
32 #include <signal.h>
33 #include <stdio.h>
34 #include "CLK.h"
35 #include "System.h"
36 #include "Com_PC.h"
37 #include "Radio_TX_RX.h"
38 #include "Flash.h"
39
40 // Globale Variabeln
41 volatile unsigned char State;              // Aktueller Betriebsmodi
42 volatile unsigned char get_new_State;      // 1: Zustandswechsel vom PC empfangen
43                                            // 0: kein Zustandswechsel
44 extern volatile unsigned long Tick;        // Wird in der Watchdog-ISR (250ms)
45                                            // inkrementiert
46 extern volatile unsigned short Count_Ring; // Anzahl der im Flash gespeicherten
47                                            // Datensätze
48 extern volatile unsigned char Order_Code;  // Anfrage Code wird von der Steuer-
49                                            // software vom PC gesendet
50
51
52 int main(void)
53 {
54       int i = 0;                           // for-Schleife
55       unsigned long Sample_Rate = 4;       // Zeit (x*250ms) zwischen den
56                                            // zyklischen Transponderabfragen
57       unsigned long Last_Tick = 0;         // für die Prüfung der zyklischen
58                                            // Wartezeit
59       int Number_Transponders = 1;         // Anzahl der Transponder, die zyklisch
60                                            // abgefragt werden sollen
61       int Index_Number_Transponders;       // Für das durchlaufen der Transponder-
62                                            // adressen im zyklischen Modus
63       DATA_STRUCT Data_to_Flash;           // Zwischenpuffer zur Speicherung(Flash)
64                                            // der Transponderdaten
65       DATA_STRUCT *pRead;                  // Für den Zeigerzugriff aufs struct
66
67       // Adressraum für die zyklische Abfrage
68       unsigned char Adresses[16] = {0x01, 0x02};
```

```
69        unsigned short Temperature;          // Zwischenpuffer für die vom Trans-
70                                             // ponder empfangene Temperatur
71        unsigned short Voltage;              // Zwischenpuffer für die vom Trans-
72                                             // ponder empfangene Spannung
73        unsigned char Address;               // Adresse wird von der Steuersoft-
74                                             // ware bei der azyklischen Abfrage
75                                             // gesetzt
76
77        XT2_set(MHZ_8);                      // Internen CLK auf 8 MHz
78        init_Radio_TX_RX();                  // Vorbereitungen für die
79                                             // Kommunikation zwischen Reader und
80                                             // Transponder
81        init_Com_PC();                       // Initialisierung der seriellen
82                                             // Schnittstelle
83        init_System();                       // Restlichen System Initialisierungen
84                                             // (Ports, WDT)
85        InitLCD();                           // Display Initialisieren (Startausgabe)
86        init_Flash ();                       // Ports die an den Flash angeschlossen
87                                             // sind setzen
88
89        _EINT();                             // enable Interrupts
90        use_Display(1,0);                    // printf-Ausgabe auf die 1. Zeile im
91                                             // Diplay umleiten
92        printf("Mode waehlen-> 0");          // Ausgabe auf dem Diplay erzeugen
93        use_Display(0,0);                    // printf-Ausgabe auf die serielle
94                                             // Schnitstelle (putchar) umleiten
95
96
97        while(1)                             // Hauprendlosschleife
98        {
99              get_State();                   // Prüfen, ob ein Befehl zum Zustands-
100                                            // wechsel von der Steuersoftware
101                                            // empfangen wurde. J: neuen Zustand
102                                            // anfordern und quittieren.
103                                            // Informationen zum Zustandswechsel
104                                            // auf dem Display ausgeben
105             switch(State)                  // Betriebsmodus prüfen
106             {
107                   case 0:                  // Modus 0 wird nicht vergeben, da 0
108                                            // das Sonderzeichen für das Kommando
109                                            // zum Zustandswechsel ist
110                   break;
111
112                   case 1:                  // Zyklischer Modus
113                         // Warten bis die Zeit zwischen 2 zyklischen
114                         // Abfragen abgelaufen ist
115                         while( (Tick - Last_Tick) < Sample_Rate)
116                         {
117                               if(get_new_State == 1) // Zustandswechsel empfangen?
118                               {
119                                     break;           // Modus abbrechen und Zustand
120                                                      // wechseln
121                               }
122                         }
123                         Last_Tick = Tick;            // Für die Messung der Wartezeit
124
125                         // Für den festgelegten Adressraum Daten von den Transpondern
126                         // abholen und in den Flash schreiben
127                         Index_Number_Transponders = 0;
128                         while(Index_Number_Transponders < Number_Transponders)
129                         {
```

```
130                         if(get_new_State == 1) // Zustandswechsel empfangen?
131                         {
132                               break;           // Modus abbrechen und in den
133                                                // Folgezustand wechseln
134                         }
135                         // Adresse und Kommando senden
136                         Send_Address(Adresses[Index_Number_Transponders]);
137
138                         // Auf die Antwort vom angesprochenen Transponder
139                         // warten (750ms).
140                         // Falls eine Antwort kommt werden die Daten decodiert
141                         // und verifiziert und anschließend in die
142                         // übergebenen Variablen gespeichert.
143                         ReceiveData(&Temperature, &Voltage);
144
145                         // Empfangene Daten in das Datenformat für die
146                         // Speicherung im Flash umwandeln
147                         Vals_To_Data(&Data_to_Flash, Tick, Temperature, Voltage,
                                Adresses[Index_Number_Transponders]);
148
149                         // Datensatz in den Ringspeicher einfügen
150                         Write_Ring_Flash(&Data_to_Flash);
151                         Index_Number_Transponders++;
152                   }
153
154                   use_Display(1,2);
155                   printf("Anzahl We.: %u", Count_Ring);
156                   use_Display(0,2);
157
158             break;
159
160             // Auf Anfrage des PC's Werte vom Transponder anfordern
161             // und zum PC senden
162             case 2:
163
164                   // Funktion wartet, dass über die serielle Schnittstelle
165                   // eine Adresse von der PC-Steuersoftware empfangen wird.
166                   // -> Schreibt die empfangene Adresse in die Variable.
167                   Address = get_Address__from_PC();
168
169                   // Sendet die zuvor empfangene Adresse und ein Steuer-
170                   // befehl über den Kanal an alle sich im Feld befindenen
171                   // Transponder. Der Steuerbefehl wird noch nicht genutzt,
172                   // da zur Zeit nur eine Betriebsart auf den Transpondern
173                   // Implementiert ist. Später soll der Steuerbefehl zum
174                   // Umschalten der Betriebsmodi verwendet werden. Dann
175                   // sollte die Funktion in Send_Data() umbenannt werden.
176                   Send_Address(Address);
177
178                   // Zustandswechsel empfangen?
179                   if(get_new_State == 1)
180                   {
181                         // Modus abbrechen und in den Folgezustand wechseln
182                         break;
183                   }
184
185                   // Daten vom angesprochenen Transponder
186                   // innerhalb von 750ms empfangen?
187                   if(ReceiveData(&Temperature, &Voltage) == 1)
188                   {
189                         // Ja: Empfangene Daten an die PC-Steuersoftware
```

```
190                         //     senden
191                         Send_Data_to_PC(Temperature, Voltage);
192                   }
193                   else
194                   {
195                   // nein: gar nichts an die PC-Steuersoftware
196                   //       senden. Zuvor wurde ein Error-Zeichen
197                   //       (0xFFFF, 0xFFFF) an die PC-Steuersoftware
198                   //       gesendet. -> PC-Steuersoftware ist so um-
199                   //       geschrieben, dass sie über ein Timeout,
200                   //       dass an das Timeout vom Reader (750ms)
201                   //       angepasst ist, den Empfang fehlerhaft
202                   //       (keine Transponderdaten erhalten -> noch-
203                   //        mal Anfragen) abbricht.
204
205                         //Send_Data_to_PC(0xFFFF, 0xFFFF);
206
207
208                   }
209
210                   // Empfangenen Werte auf dem Display ausgeben
211                   Display_Vals(Address, Temperature, Voltage);
212             break;
213
214             case 3:  // Auf Anfrage der PC-Steuersoftware Systemparameter
215                      // zurücksenden. Kann um beliebige Parameter
216                      // erweitert werden.
217
218                   // Auf eine Anfrage von der PC-Steursoftware warten
219                   // Order_Code wird in der USART-ISR auf 1 gesetzt,
220                   // wenn ein Anfrage empfangen wurde.
221                   while(Order_Code==0)
222                   {
223                         // Zustandswechsel empfangen?
224                         if(get_new_State == 1)
225                         {
226                               // Warten abbrechen und in den
227                               // Folgezustand wechseln.
228                               break;
229                         }
230                   }
231
232                   // Zustandswechsel empfangen?
233                   if(get_new_State == 1)
234                   {
235                         // Modus abbrechen und in den
236                         // Folgezustand wechseln.
237                         break;
238                   }
239
240                   // Anfrage auf die Anzahl der im Flash gespeicherten
241                   // Datensätze.
242                   else if(Order_Code == 1)
243                   {
244                         putchar( (Count_Ring >> 8) & 0xFF);
245                         putchar( (Count_Ring) & 0xFF);
246                   }
247
248                   // Aktueller Tick-Wert wird zurückgegeben. Wird beim Auslesen
249                   // des Flashspeicher benötigt, um die Uhrzeit und das
250                   // Datum des jeweiligen Datensatzes zu ermitteln.
```

```
251                         else if(Order_Code == 2)
252                         {
253                               putchar( (Tick >> 24) & 0xFF);
254                               putchar( (Tick >> 16) & 0xFF);
255                               putchar( (Tick >> 8) & 0xFF);
256                               putchar( (Tick >> 0) & 0xFF);
257                         }
258
259                         // Sendet den "ältesten" Datensatz aus dem Ringspeicher des
260                         // Flash und dessen Position im Ringspeicher zurück.
261                         else if(Order_Code == 3)
262                         {
263                               pRead = Read_Ring_Flash();
264
265                               putchar( (Count_Ring >> 8) & 0xFF);
266                               putchar( (Count_Ring) & 0xFF);
267
268                               // Weiter Datensätze im Ringspeicher?
269                               if(pRead != NULL)
270                               {
271                                     putchar(pRead->C0);
272                                     putchar(pRead->C1);
273                                     putchar(pRead->C2);
274                                     putchar(pRead->C3);
275                                     putchar(pRead->C4);
276                                     putchar(pRead->C5);
277                                     putchar(pRead->C6);
278                                     putchar(pRead->C7);
279                                     putchar(pRead->C8);
280                                     putchar(pRead->C9);
281                                     putchar(pRead->C10);
282                                     putchar(pRead->C11);
283                                     putchar(pRead->C12);
284                                     putchar(pRead->C13);
285                                     putchar(pRead->C14);
286                                     putchar(pRead->C15);
287                               }
288                               else      // Ringspeicher ist leer.
289                               {
290                                     for (i=0; i<16;i++)
291                                       putchar(0x00);
292                               }
293                               nop();
294
295                         }
296                         Order_Code = 0;  // Order_Code zurücksetzten, damit die
297                                          // Anfrage nicht mehrmals ausgeführt wird.
298                   break;
299                   default:
300                   break;
301             }                            // switch(State)
302       }                                  // while(1)
303       return 0;
304 }
```

A.3.2 System.h

Listing 9: System.h

```
/*****************************************************************************/
/* Projekt: Readerquellcode                                                  */
/* Dateiname: System.h                                                       */
/* Autor: Tobias Krannich                                                    */
/* Kurzbeschreibung: Heder für System.c                                      */
/*****************************************************************************/

#ifndef SYSTEM_H_
#define SYSTEM_H_

// Für die LED-Ansteuerung
#define LED1_AN (P3OUT &=~ BIT6);
#define LED1_TOGGLE (P3OUT ^= BIT6);
#define LED1_AUS (P3OUT |= BIT6);
#define LED2_AN (P3OUT &=~ BIT7);
#define LED2_AUS (P3OUT |= BIT7);
#define LED2_TOGGLE (P3OUT ^= BIT7);


// Prototypen
void init_System (void);

// Globale Variablen
extern volatile unsigned long Tick;

#endif /*SYSTEM_H_*/
```

A.3.3 System.c

Listing 10: System.c

```
/*****************************************************************************/
/* Projekt: Readerquellcode                                                  */
/* Dateiname: System.c                                                       */
/* Autor: Tobias Krannich                                                    */
/* Kurzbeschreibung: Initialisierung für das System. WDT-ISR, die Tick alle  */
/*                   250ms inkrementiert                                     */
/*****************************************************************************/

#include <msp430x16x.h>
#include <signal.h>
#include "System.h"


void init_System (void)
{

      // WatchdogTimer auf 250ms stellen und Interrupt freigeben
      WDTCTL = WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDTIS_1;
      IE1 |= WDTIE;



      ME1 |= UTXE0 + URXE0;     // enable UART0 RX&TX
      UCTL0 &= ~SWRST;          // clear SWRST
      IE1 |= URXIE0;            // enable RX interrupt

```

```
27
28        P3DIR |= BIT4 | BIT6 | BIT7;     // TX USART0, LED1, LED2
29        P3DIR &=~ BIT5;                  // RX USART0
30        P3SEL |= BIT4 | BIT5;            // TX, RX USART0
31        P3SEL &=~ (BIT6 | BIT7);
32        LED1_AUS
33        LED2_AUS
34  }
35
36  // Inkrementiert die Variable Tick alle 250ms. Tick wird im System
37  // Für Zeitmessungen und Timeouts benötigt.
38  interrupt(WDT_VECTOR) isr_WDT(void)
39  {
40        Tick++;
41  }
```

A.3.4 RadioTXRX.h

Listing 11: RadioTXRX.h

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: RadioTXRX.h                                                    */
4  /* Autor: Tobias Krannich                                                    */
5  /* Kurzbeschreibung: Heder für RadioTXRX.c                                   */
6  /*****************************************************************************/
7
8
9  #ifndef RADIO_TX_RX_H_
10 #define RADIO_TX_RX_H_
11
12 #define CRC32POLY 0x1021 // CRC-16 Polynom
13
14 // Prototypen
15 void init_Radio_TX_RX(void);
16 void Send_Address (unsigned char);
17 unsigned char ReceiveData (unsigned short*, unsigned short*);
18 void calc_crc16(unsigned char bit);
19 unsigned short calc_crc16_vector(unsigned char Data[], unsigned char lenght);
20
21 // Globale Variablen
22 extern volatile unsigned long Tick;
23 #endif /*RADIO_TX_RX_H_*/
```

A.3.5 RadioTXRX.c

Listing 12: RadioTXRX.c

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: RadioTXRX.c                                                    */
4  /* Autor: Tobias Krannich                                                    */
5  /* Kurzbeschreibung: Stellt Funktion für die Kommunikation zwischem Reader   */
6  /*                   und Transponder zur Verfügung                           */
7  /*****************************************************************************/
```

```
8
9  #include <msp430x16x.h>
10 #include <signal.h>
11 #include "Radio_TX_RX.h"
12 #include "System.h"
13
14 volatile unsigned short Time_Average;  // Durchschittszeit zwischen 2
15                                        // steigenden Flanken der Preambles
16 volatile unsigned char DataIndex;      // Index der empfangenen Bits
17 volatile unsigned char SendData[18];   // 18 Bit Übertragungsprotokoll
18                                        // für den Daten-Up-Load.
19                                        // (Reder -> Transponder)
20 volatile unsigned short CRC;           // Variable für die empfangene Check-
21                                        // summe aus den Transponderdaten.
22                                        // Checksumme
23 volatile unsigned char Data[49];       // Feld für die empfabgenen Daten,
24                                        // die vom Transponder gesendet wurden.
25 volatile unsigned char SendReady;      // Flag, das angibt, ob das Senden
26                                        // zum Transponder abgeschlossen ist.
27 volatile unsigned char ReceiveReady;   // Flag, das angibt, ob der Empfang
28                                        // von Transponderdarten abgeschlossen
29                                        // ist.
30 volatile unsigned short crc16;         // Schieberegister für die CRC16
31 volatile unsigned short CRC_calc;      // Variable für die auf Basis der
32                                        // empfangenen Daten berechneten
33                                        // Chaecksumme.
34 volatile unsigned char Run_In_State;   // Speichert den Status des Run-In-
35                                        // Zustandsautomaten.
36 volatile unsigned char RunInOK;        // Gibt an, ob eine gültige Run-In-
37                                        // Sequenz empfangen wurde.
38 volatile unsigned char Command;        // In Command kann ein Kommando
39                                        // an den Transponder gesendet
40                                        // werden. Das Kommando wird zur Zeit
41                                        // im Transponder noch nicht aus-
42                                        // gewährtet.
43
44 void init_Radio_TX_RX(void)
45 {
46
47       // Initilisierung Timer A (Daten empfangen)
48       TACTL = TASSEL1 + TACLR;      // SMCLK, lösche TAR
49       TACCTL0  = CM0 + CAP;         // steigende Flanken, Capture
50       TACTL |= ID_3;                // Divider = 8
51
52       // Initialisierung Timer B (Daten senden)
53       TBCTL = TBSSEL1 + TBCLR;      // SMCLK, lösche TAR
54       TBCCTL0 = CCIE;               // CCR0 Interrupt freigeben, Compare
55       TBCTL |= ID_3;                // Divider = 8
56       TBCCR0 = 0x1388;              // Compare Register setzen
57
58
59
60       // Initialisierung der Ports
61
62       P1SEL |= BIT1;        // Port 1.1 TimerA input Capture
63       P1DIR &=~ BIT1;
64
65       P1SEL &=~ BIT2;       // Port 1.2 Daten zum Transponder
66       P1DIR |= BIT2;
67 }
68
```

```
69
70  // Die Funktion Send_Address sendet die übergebene Adresse und ein Kommando
71  // an die Transponder. Das senden des Kommandos wurde nachträglich implementiert,
72  // so dass der Name der Funktion in Send_Data geändert werden muss, wenn für
73  // Weiterentwicklungen das Kommando genutzt wird.
74  // Für die Kommunikation in Richtung Transponder wird keine Run-In-Sequenz
75  // benötigt, da durch das On-Off-Keying keine Störungen am Transponder
76  // ankommen.
77  // Die Preambles können durch ein Startbit (1 -> [10]) ersetzt werden, da
78  // die Taktfrequenz im Readercontroller konstant 8Mhz beträgt, so dass für
79  // die Bitbreiten mit einer festen Frequenz (100Hz) gearbeitet werden kann.
80  void Send_Address (unsigned char Adress)
81  {
82        unsigned char i = 0;
83        SendData[0] = 1;
84        SendData[1] = 0;
85
86        for(i=0; i<4; i++)  // Adresse Manchester codieren
87        {
88              if( (Adress >> (3-i)) & 0x01 )
89              {
90                    SendData[2*i + 2] = 1;
91                    SendData[2*i + 1 + 2] = 0;
92              }
93              else
94              {
95                    SendData[2*i + 2] = 0;
96                    SendData[2*i + 1 + 2] = 1;
97              }
98        }
99
100
101       if(Command == 0) // Kommando Manchester codieren
102       {
103             SendData[10] = 0;//LSB
104             SendData[11] = 1;
105             SendData[12] = 0;//MSB
106             SendData[13] = 1;
107       }
108       else if (Command == 1)
109       {
110             SendData[10] = 1;//LSB
111             SendData[11] = 0;
112             SendData[12] = 0;//MSB
113             SendData[13] = 1;
114       }
115       else if (Command == 2)
116       {
117             SendData[10] = 0;//LSB
118             SendData[11] = 1;
119             SendData[12] = 1;//MSB
120             SendData[13] = 0;
121       }
122       else if (Command == 3)
123       {
124             SendData[10] = 1;//LSB
125             SendData[11] = 0;
126             SendData[12] = 1;//MSB
127             SendData[13] = 0;
128       }
129
```

```
130
131         SendData[14] = 1; Run-Out-Sequenz
132         SendData[15] = 0;
133         SendData[16] = 1;
134         SendData[17] = 0;
135
136         SendReady = 0;    // Senden der Daten starten
137         TBCTL |= MC1;     // Start Timer_A in continuous mode
138
139         while(SendReady == 0);  // Warten bis der Daten-Upload beendet ist
140
141         TBCTL &=~ (MC1 | MC0);  // TimerB stop
142
143 }
144
145 // Die Funktion ReceiveDate steurt den Datenempfang. Nähere Information
146 // in der Funktion.
147 unsigned char ReceiveData (unsigned short *Temperature_Ret, unsigned short *Voltage_Ret
        )
148 {
149
150         unsigned short  j = 0;          // for
151         unsigned long Time_Start = 0;   // Für die Messung der Timeout Zeit
152         unsigned short Temperature = 0;  // Empfagene Temperatur
153         unsigned Voltage= 0;           // Empfangene Spannung
154
155         for(j=0;j<200;j++)              // Kurz warten, damit keine Signale vom
156                                         // zuvor gesendeten Up-Load-Paket mehr
157                                         // auf der Leitung sind.
158         {
159               nop();
160         }
161
162         // Datenempfang vorbereiten
163         ReceiveReady = 0;
164         DataIndex = 0;
165         Run_In_State = 0;
166         RunInOK = 0;
167         TACCTL0 &=~ CCIFG;
168         TACCTL0  |= CCIE;
169
170
171         Time_Start = Tick;        // Zeit für die Messung des Time Outs speichern
172         while(ReceiveReady == 0) // Warten bis der Empfang abgeschlosseen ist
173         {
174               if( (Tick - Time_Start) > 3)  // Time Out?
175               {
176                     ReceiveReady = 3;       // Empfang mit Fehler 3 abbrechen
177               }
178         }
179         TACCTL0  &=~ CCIE;      // Capture ISR sperren
180
181         // Nächsten Datenempfang vorbereiten
182         DataIndex = 0;
183         RunInOK = 0;
184         Run_In_State = 0;
185
186         if(ReceiveReady == 1) // Daten vollständig ohne Fehler empfangen?
187         {
188               // Daten decodieren und verifizieren
189               Temperature = 0;
```

```
190             Voltage= 0;
191             CRC = 0;
192             for(j=1; j<49; j++)
193             {
194                   if(j<17)  // Wert für die Temperatur extrahieren
195                   {
196                         Temperature |= Data[j] << (16 - j);
197                   }
198                   else if(j<33)  // Wert für die Spannung extrahieren
199                   {
200                         Voltage|= Data[j] << (32 - j);
201                   }
202                   else     // Wert für die Checksumme extrahieren
203                   {
204                         CRC |= Data[j] << (48 - j);
205                   }
206             }
207
208             // CRC16-Checksumme über die empfangenen Daten bilden
209             CRC_calc = calc_crc16_vector(&Data[1], 32);
210             if(CRC_calc != CRC) // Berechnete Checksumme != empfangener?
211             {
212                   ReceiveReady = 4; // Empfang mit Fehler 4 abbrechen
213             }
214       }
215
216       *Temperature_Ret = Temperature;
217       *Voltage_Ret = Voltage;
218
219
220       return ReceiveReady; // 1: Alles OK
221                            // 2: Daten unvollständig empfangen
222                            // 3: Keine Antwort
223                            // 4: Checksummen Fehler
224 }
225
226 // Die Funktion calc_crc16Vector ruft die Funktion calc_crc16 in einer
227 // Schleife auf und übergiebt ihr die 32 Datenbits. Anschließend
228 // wird die Chacksumme invertiert und zurückgegeben
229 void calc_crc16(unsigned char bit)
230 {   int hbit;
231
232     hbit=(crc16 & 0x8000) ? 1 : 0;
233     if (hbit != bit)
234         crc16=(crc16<<1) ^ CRC32POLY;
235     else
236         crc16=crc16<<1;
237 }
238
239 // Die Funktion build_CRC16Vector teilt die 16 Bit Checksumme
240 // auf ein 16 Byte großes Feld auf
241 unsigned short calc_crc16_vector(unsigned char Data[], unsigned char lenght)
242 {
243       int i;
244       crc16 = 0xFFFF;
245       for (i=0; i<lenght; i++)
246       {
247         calc_crc16(Data[i]);
248       }
249       crc16 ^= 0xFFFF;
250
```

```
251       return crc16;
252
253 }
254
255
256 // !!!!!!!!!! Bitte erst die Beschreibung in der Dokumentation Lesen !!!!!!!!!!
257 // !!!!!!!!!! bevor der folgende Quelltext analysiert wird !!!!!!!!!!!!!!!!!!!!
258 // 8000000Mhz / (1/1000us) = 8000000/1000 = 8000 CLK's Zeit
259
260 // Input Capture Interrupt des Timer A. Mist und bewertet Zeiten für den
261 // Empfang der seriellen Transponderdaten.
262 // Für diese Art der Datedecodierung wird nur die zeit zwischen den steigenden
263 // Flanken ausgewährtet. Die genaue Beschreibung des Decodierungsablaufs kann
264 // der Dokumentation entnommen werden, da eine ausführliche Beschreibung an
265 // dieser Stelle ohne Grafiken nicht möglich ist.
266 //
267 // Für eine Datenfrequenz von 1kHz ergibt sich für die maximale Ausführzeit
268 // der ISR ohne das sie sich selbst unterbricht (oder Pending entsteht /
269 // ISR- Anfragen verloren gehen) eine Zyklenzahl von 8000 Takten. Die ISR
270 // Fall einen großen Zeitumfang (bis 8000 CLK's) haben, da keine weitere
271 // kann in diesem Funktionalität während der Empfangsphase im Programm abläuft.
272 // Der Interrupt ist in die folgenden 5 Hauptabschnitte geteilt:
273 // 1. Erkennung der Run-In-Sequenz mit Hilfe eines Zustandsautomaten
274 interrupt(TIMERA0_VECTOR) isr_TimerA0(void)
275 {
276       static unsigned char n = 0;              // for()
277       static unsigned short Time;              // Zeitdifferenz zur letzten
278                                                // steigenden Flanke
279       static unsigned short Time_Preamble[3]; // Feld für die 3 Preamble
280       static unsigned short Time_1;            // Entspricht der Durchschnitts-
281                                                // zeit der 3 Preamble
282
283       // Die folgenden 4 Zeiten werden für die Decodierung benötigt.
284       // Mit Hilfe dieser Zeitabschnitte kann die Zeit zwischen
285       // 2 steigenden Flanken in 3 Abschnitte eingeteilt werden
286       // Mit Hilfe dieser Einteilung und der Kenntnis des vorigen
287       // Datenbits ist eine Decodierung möglich.
288       static unsigned short Time_0_8;
289       static unsigned short Time_1_3;
290       static unsigned short Time_1_8;
291       static unsigned short Time_2_3;
292
293       static unsigned char Bit = 0;          // Enthält Information über den
294                                              // vorigen Empfang (siehe Code)
295       static unsigned short LastTime = 0;    // Für die Zeitmessung bei
296                                              // der Run-In-Sequenz
297
298
299       Time = TACCR0;            // Zeit aus dem Register auslesen
300       TACTL &=~ (MC1 | MC0);    // TimerA stop
301       TAR = 0;                  // Zähler löschen
302       TACTL |= MC1;             // Starte Timer_A in continuous mode
303
304       // RunIn Zustandsautomat
305       // Für die Prüfung auf die Run-In-Sequenz werden auf Grund der
306       // Taktschwankungen im Transponder keine ablosluten Zeiten geprüft,
307       // sondern die Verhältnisse zwischen den gemessenen Zeiten.
308       // Hierfür muss nur gewährleistet werden, dass der Transpondertakt
309       // innerhalb eines Frames konstant bleibt.
310       switch(Run_In_State)
311       {
```

```
312             case 0: // Prüfe ob die aktuelle Zeit ca. 3/4 so lang ist
313                     // wie die letzte Zeit.
314
315                   // Zeit entspricht 3/4 der vorigen Zeit?
316                   if( (Time<LastTime*17) && (Time>LastTime*13) )
317                   {
318                         // Ersten beiden Zeiten der Run-In-Sequenz erkannt!
319                         // -> Übergang in den Folgezustand.
320                         Run_In_State = 1;
321
322                         // Zeit auf gemeinsamen Nenner bringen
323                         LastTime = (unsigned short)(Time / 15);
324
325                   }
326                   else // Zeit enspricht nicht 3/4 der vorigen Zeit
327                   {
328                         // Run-In-Suche neu starten
329
330                         // Zeit auf gemeinsamen Nenner bringen
331                         LastTime = (unsigned short)(Time / 20);
332                   }
333             break;
334
335             case 1: // Prüfe ob die aktuelle Zeit ca. 3/4* so lang ist
336                     // wie die letzte Zeit.
337
338                   // Zeit entspricht 3/4 der vorigen Zeit?
339                   if( (Time<LastTime*12) && (Time>LastTime*8) )
340                   {
341                         // Komplette (3) Run-In-Sequenz erkannt!
342                         // -> Übergang in den Folgezustand.
343                         Run_In_State = 2;
344                         LastTime = 0;  // Löschen zur Vorbreitung für die
345                                        // nächste Run-In-Suche
346
347                   }
348                   else // Zeit enspricht nicht 3/4 der vorigen Zeit
349                   {
350                         // Run-In-Suche neu starten
351                         Run_In_State = 0;
352
353                         // Zeit auf gemeinsamen Nenner bringen
354                         LastTime = (unsigned short)(Time / 20);
355                   }
356             break;
357                   case 2:  // Flag für die erfolgreich abgeschlossene
358                            // setzenRun-In-Suche
359                   RunInOK = 1;
360                   Run_In_State = 3; // Verhindert weiters Suchen der Run-
361                                     // In-Sequenz während des Datenempfangs
362
363             break;
364       }
365
366
367       if(RunInOK == 1)  // Run-In-Suche erfolgreich abgeschlossen?
368       {
369             // Mit dem Einlesen der Daten beginnen.
370
371             if(DataIndex<1)            // Erste steignede Flanke nach
372                                        // der Run-In-Sequenz?
```

```
373             {
374                   TACTL |= MC1;         // Starte Timer_A in continuous mode
375                   Bit = 1;              // Startbit = 1 (wissen wir) -> Bit = 1
376                   LED2_AN               // Zeigt die Dauer der Datenübertragung an
377                   DataIndex++;
378             }
379             else if(DataIndex<4)      // Preamles einlesen (3+1 = 4)
380             {
381                   Time_Preamble[DataIndex-1] = Time;
382                   if(DataIndex==3)  // Alle Preambles empfangen?
383                   {
384                         // Durchschnittszeit pro Bit berechnen
385                         Time_1 = 0;
386                         for(n=0; n<3; n++)
387                         {
388                               Time_1 += Time_Preamble[n];
389                         }
390                         Time_1 /= 3;
391
392                         // Zeitabschnitte für die folgende Decodierung berechnen.
393                         // Auch hier wird wie bei der Run-In-Suche nur mit Zeit-
394                         // verhältnisen gerechnet. (ungenauer Transpondertakt)
395                         Time_0_8 = (unsigned short)((Time_1 * 8) / 10 );
396                         Time_1_3 = (unsigned short)((Time_1 * 13) / 10 );
397                         Time_1_8 = (unsigned short)((Time_1 * 18) / 10 );
398                         Time_2_3 = (unsigned short)((Time_1 * 23) / 10 );
399                   }
400                   DataIndex++;
401             }
402
403
404             else if(DataIndex<53)      // Daten einlesen (1+3+48+1 = 53)
405             {
406                   // In diesem Abschitt werden die Datenbits eingelesen
407                   // unmittelbar decodiert. Das Funktionsprinzip ist
408                   // der Dokumentation zu entnehmen.
409
410
411                   if(Time < Time_0_8)      // Gemessene Zeit zu kurz
412                   {                        // Empfang mit Fehler 2 abbrechen
413                         ReceiveReady = 2;
414                         TACTL &=~ (MC1 | MC0);  // TimerA stop
415                         TACCTL0  &=~ CCIE;      // Capture ISR sperren
416                         TAR = 0;                // Zähler löschen
417                         DataIndex = 0;          // DatenIndex für das
418                                                 // nächste Empfangen
419                                                 // zurücksetzen
420                   }
421                   else if(Time < Time_1_3)      // ca. 1 * Durchschnittszeit?
422                   {
423                         Data[DataIndex-4] = Bit; // Bit übernehmen
424                         DataIndex++;
425                   }
426                   else if(Time < Time_1_8)      // ca. 1,5 * Durchschnittszeit?
427                   {
428                         if(Bit == 1)  // Bit == 1?
429                         {
430                               // 1 Datenbit empfangen!
431
432                               Data[DataIndex-4] = Bit; // Bit übernehmen
433                               DataIndex++;
```

```
434                                 Bit ^= 0x01;      // Bitwechsel einleiten
435                           }
436                           else // Bit == 0;
437                           {
438                                 // 2 Datenbits empfangen!
439
440                                 Data[DataIndex-4] = Bit; // Bit übernehmen
441                                 DataIndex++;
442                                 Bit ^= 0x01;      // Bitwechsel einleiten
443                                 Data[DataIndex-4] = Bit;  // Bit übernehmen
444                                 DataIndex++;
445                           }
446                     }
447                     else if(Time < Time_2_3)      // ca. 2 * Durchschnittszeit?
448                     {
449                           // 2 Datenbits empfangen!
450
451                           Data[DataIndex-4] = Bit;  // Bit übernehmen
452                           DataIndex++;
453                           Bit ^= 0x01;              // Bitwechsel einleiten
454                           Data[DataIndex-4] = Bit;  // Bit übernehmen
455                           DataIndex++;
456                           Bit ^= 0x01;              // Bitwechsel einleiten
457                     }
458                     else      // Gemessene Zeit zu lang
459                     {         // Empfang mit Fehler 2 abrechen
460                           ReceiveReady = 2;
461                           TACTL &=~ (MC1 | MC0);      // TimerA stop
462                           TACCTL0  &=~ CCIE;          // Capture ISR sperren
463                           TAR = 0;                    // Zähler löschen
464                           DataIndex = 0;              // Datenindex für das nächste
465                                                       // Empfangen zurücksetzen
466                     }
467               }
468               else      // alles Empfangen
469               {
470                     TACTL &=~ (MC1 | MC0);      // TimerA stop
471                     TAR = 0;                    // Zähler löschen
472                     DataIndex = 0;              // Datenindex für das nächste
473                                                 // Empfangen zurücksetzen
474                     ReceiveReady = 1;           // Empfang ohne Fehle beendet!
475                     RunInOK = 0;                // Run-In vorbereiten
476                     Run_In_State = 0;           // Starten der Suche nach der
477                                                 // nächsten Run-In-Sequenz vorber.
478                     TACCTL0  &=~ CCIE;          // Capture ISR sperren
479               }
480         }
481
482         TACCTL0 &=~ CCIFG;  // Flag löschen
483   }
484
485
486
487
488
489   // Output Compare ISR des Timer B. Wird für die Ausgabe der Uploaddaten
490   // benötigt. Der Timer wird gestartet, wenn Daten an den Transponder
491   // gesendet werden sollen. Sendet die 18 Bits aus SendData[] mit einer
492   // Frequenz von 100Hz über Port 1 an die Modulationsschaltung.
493   interrupt (TIMERB0_VECTOR) Timer_B(void)
494   {
```

```
495         static unsigned char OutDataIndex = 0;
496         TBCCR0 += 0x1388;                   // Compareregister um "0,01s" erhöhen
497         if(SendData[OutDataIndex] == 0)     // == 0 statt == 1 um das Signal zu
498                                             // invertieren (1 entspricht dem
499                                             // Idlezustand auf Transponderseite.
500                                             // 0 kann nicht als Idlezustand gewählt
501                                             // werden, da dem Transponder sonst
502                                             // die Verorgung Fehlen würde.)
503                                             // [OOK -> siehe Schaltplan]
504         {
505               P1OUT |= BIT2;
506         }
507         else
508         {
509               P1OUT &=~ BIT2;
510         }
511
512         OutDataIndex++;
513         if(OutDataIndex >= 18)     // Alle Bits (18) gesendet?
514         {
515               // Datenindex für das nächste Senden vorbereiten
516               OutDataIndex = 0;
517
518               // Senden erfolgreich abgeschlossen
519               SendReady = 1;
520         }
521
522 }
```

A.3.6 ComPC.h

Listing 13: ComPC.h

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: ComPC.h                                                        */
4  /* Autor: Tobias Krannich                                                    */
5  /* Kurzbeschreibung: Heder für ComPC.c                                       */
6  /*****************************************************************************/
7
8
9  #ifndef COM_PC_H_
10 #define COM_PC_H_
11
12 // Konstanten für die Kommunikation mit dem Diplay
13 #define            DISP_ON             0x0c
14 #define            DISP_OFF            0x08
15 #define            CLR_DISP            0x01
16 #define            CUR_HOME            0x02
17 #define            ENTRY_INC           0x06
18 #define            DD_RAM_ADDR         0x80
19 #define            DD_RAM_ADDR2        0xc0
20 #define            DD_RAM_ADDR3        0x28
21 #define            CG_RAM_ADDR         0x40
22
23 #define   E_HIGH          P4OUT |= BIT1
24 #define   E_LOW           P4OUT &= ~BIT1
25 #define   RS_HIGH         P4OUT |= BIT3
26 #define   RS_LOW          P4OUT &= ~BIT3
```

```
27 #define   LCD_Data        P4OUT
28 #define   LCD_LIGHT_ON    P4OUT |= BIT0
29 #define   LCD_LIGHT_OFF   P4OUT &= ~BIT0
30 #define   _100us     7
31
32 // Baudrate für die PC-Kommunikation
33 #define BAUDRATE 115200
34
35 // Prototypen
36 void Delay (unsigned int a);
37 void Delayx100us(unsigned char b);
38 void _E(void);
39 void SEND_CHAR (unsigned char d);
40 void SEND_CMD (unsigned char e);
41 void InitLCD(void);
42 void use_Display(unsigned char, unsigned char);
43 void init_Com_PC(void);
44 int putchar (int);
45 void get_State(void);
46 unsigned char get_Address__from_PC(void);
47 void Send_Data_to_PC(unsigned short, unsigned short);
48 void Display_Vals(unsigned char, unsigned short, unsigned short);
49
50 // Globale Variablen
51 extern volatile unsigned char State;
52 extern volatile unsigned short Temperatur;
53 extern volatile unsigned short Spannung;
54 extern volatile unsigned char get_new_State;
55
56
57 #endif /*COM_PC_H_*/
```

A.3.7 ComPC.c

Listing 14: ComPC.c

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: ComPC.c                                                        */
4  /* Autor: Tobias Krannich                                                    */
5  /* Kurzbeschreibung: Stellt Funktionen für die Kommunikation mit der PC-     */
6  /*                   Steursoftware zur Verfügung. Funktionen zum Ansprechen  */
7  /*                   des Displays                                            */
8  /*****************************************************************************/
9
10 #include <msp430x16x.h>
11 #include <signal.h>
12 #include <stdio.h>
13 #include "Com_PC.h"
14 #include "CLK.h"
15
16
17 // Diplay
18 volatile unsigned char Display; // 1: printf-Ausgabe auf dem Display
19                                 // 0: printf-Ausgabe über die serielle
20                                 //    Schnittstelle
21
22 volatile unsigned char temp;    // Zwischenpuffer bei der Ausgabe
23                                 // auf dem Display
```

```
24
25 // Zustandsautomat
26 volatile unsigned char new_State;  // Zeigt an, ob ein Zustandswechselbefehl
27                                    // von der PC-Steuersoftware empfangen wurde.
28
29 // Mode 2: azyklische Abfrage
30 volatile unsigned char Address;         // Adresse für die azyklische Abfrage
31 volatile unsigned char get_new_Address; // Zeigt an, ob eine neue Adresse für
32                                         // die azyklische Abfrage von det PC-
33                                         // Steursoftware empfangen wurde.
34
35 // Mode 3: Informationsanfrage
36 volatile unsigned char Order_Code;      // Order_Code gibt an, welche Daten die
37                                         // PC-Steuersoftware angeforedert hat:
38                                         // 1: Anzahl der im Flash gespeicherten
39                                         //    Datensätze.
40                                         // 2: Tick. Wird für die Ermittlung
41                                         //    der Zeit und des Datums der Daten-
42                                         //    sätze beim Auslesen des Flashs
43                                         //    benötigt.
44                                         // 3: "Ältesten" Datensatz im Ring-
45                                         //    speicher.
46
47
48 // Initialisierung der seriellen Schnittstelle, des Zustandsautomaten
49 // und des Displays
50 void init_Com_PC(void)
51 {
52       unsigned short BaudRegister = 0;
53
54       // Initialisierung UART (auf das Modulationsregister
55       // kann bei 8MHz verzichtet werden)
56       UCTL0 = SWRST + CHAR;              // 8N1
57       UTCTL0 |= SSEL1;                   // SMCLK ist USART0_CLK
58       BaudRegister = CLK/BAUDRATE;       // Wert für das Bautratenregister
59                                          // berechnen
60       UBR00 = BaudRegister & 0xFF;       // Wert in das Register schreiben
61       UBR10 = (BaudRegister >> 8) & 0xFF;
62       UMCTL0 = 0x00;
63
64       // Initialiesierung des Zustandsautomats und des Displays
65       State = 0;           // Mode 0 -> fordert die Modeeingabe auf
66       get_new_State = 0;   // Anfrage auf Zustandsänderung zurück setzen
67       Display = 1;         // printf-Ausgabe auf das Display weiterleiten
68 }
69
70
71 int putchar (int out_char)
72 {
73       if(Display == 0) // Ausgabe über die serielle Schnitstelle
74       {
75             while((IFG1 & UTXIFG0) != UTXIFG0); // Letztes Zeichen fertig gesendet
76             TXBUF0 = out_char;                  // Neus Zeichen zum Senden
77                                                 // übernehmen
78             return 0;
79       }
80       else             // Ausgabe über das Display
81       {
82           SEND_CHAR(out_char); // Buchstaben auf das Display schreiben
83           return 0;
84
```

```
85        }
86  }
87
88
89  // ISR für das Empfangen von Zeichen über die serielle Schnittstelle.
90  // Zeichen werden eingelesen und bewertet. In Folge dessen werden Flags
91  // für die Steuerung des Zustandsautomaten oder dem Datenaustausch
92  // mit der PC-Steuersoftware gesetzt.
93  interrupt(USART0RX_VECTOR) isr_UART_RX(void)
94  {
95        unsigned char Zeichen;  // Puffer für das empfangene Zeichen
96        Zeichen = RXBUF0;       // Empfangsregister auslesen
97        RXBUF0 = 0x00;          // Empfangsregister löschen
98
99        if(Zeichen == 0xFF)     // Sendet die PC-Steuersoftware 0xFF wird
100                              // das Zeichen 0xAA zurückgesendet. Diese
101                              // Antwort benötigt die PC-Steuersoftware zum
102                              // automatischen Verbindungsaufbau (COM-Suche)
103       {
104             putchar(0xAA);
105       }
106       else  // Steurzeichen empfangen
107       {
108             /////////// Steuerung des Zustandsautomaten ///////////////////////
109             if(get_new_State == 0)  // Programm läuft in einem Betriebs-
110                                     // Modus (keine aktuelle Zustandsänderung)
111             {
112                   if(Zeichen == 0)  // Anfrage auf Zustandsänderung
113                   {
114                         get_new_State = 1; // Funktion get_State wird auf
115                                            // eine folgende Zustandsänderung
116                                            // vorbereitet.
117                   }
118             }
119             else if(get_new_State == 2)     // get_State hat die Zustands-
120                                             // änderung vorbereitet und kann
121                                             // neuen Zustand entgegennehmen.
122             {
123                   new_State = Zeichen;      // Folgezustand, der von der PC-
124                                             // Steuersoftware gesendet wurde
125                                             // wird global gespeichert.
126                   get_new_State = 3;        // Funktion get_State wird mitgeteilt,
127                                             // dass ein neuer Folgezustand
128                                             // empfangen wurde
129             }
130
131             ////////////// Steuerung der Informationsabfrage /////////////////
132
133             // Prüfen, ob keine laufende Zustandsänderung aktiv ist.
134             // Anschließend das Zeichen auswerten. In Folge der angefragten
135             // Information Flags setzen, die das zurücksenden der
136             // angeforderten Information einleiten.
137             if(Zeichen != 0 && get_new_State != 3)
138             {
139                   switch(State)
140                   {
141                         case 0:  // Initialisierungsmode nimmt keine Anfragen
142                                  // entgegen
143                         break;
144
145                         case 1:  // Zyklischer Mode nimmt keine Anfragen
```

```
146                                     // entgegen
147                         break;
148
149                         case 2:  // Azyklischer Mode nimmt die Transponderdresse
150                                  // entgegen, für die Daten angefordert werden.
151                               Address = Zeichen;
152                               get_new_Address = 1;
153                         break;
154
155                         case 3: // Informationsmodus nimmt Order_Code entgegen.
156                                 // Der Order_Code gibt an, welche Informatiom
157                                 // angefragt wird.(siehe deklaration Order_Code)
158                               Order_Code = Zeichen;
159                         break;
160
161                         default:
162                         break;
163                   }
164             }
165       }
166
167 }
168
169 // Die Funktion get_State steuert die Modeänderung im Zustandsautomat.
170 void get_State(void)
171 {
172       if(get_new_State == 0)      // Keine Modeänderung empfangen
173       {
174             State = State;        // Mode beibehalten
175       }
176       else if(get_new_State == 1) // Anfrage auf Modeänderung von der PC-
177                                   // Steuersoftware empfangen.
178       {
179             use_Display(1,0);     // Modeanfrage auf dem Diplay ausgeben.
180                                   // Die Ausgabe auf dem Diplay wird gemacht,
181                                   // Damit die Modeänderung auch ihne die
182                                   // PC-Steuersoftware über ein Terminal-
183                                   // Programm vorgenommen werden kann.
184             printf("Mode eingeben!");
185             use_Display(0,0);     // Untere Zeile des Displays löschen
186             use_Display(1,1);     // (alte Ausschriften)
187             printf(" ");
188             use_Display(0,0);     // printf-Ausgabe auf die serielle Schitt-
189                                   // stelle umleiten.
190             get_new_State = 2;
191             putchar(0x00);        // Bestätigung für die Zustandwechelanfrage
192                                   // an die PC-Steuersoftware senden.
193             while(get_new_State == 2); // Warten bis der neue Zustand von der PC-
194                                        // Steuersoftware gesendet und in der
195                                        // UART-ISR empfangen wurde.
196
197             if(new_State == 0)      // 0 ist kein gültiger neuer Zustand.
198                                     // Neue Zustandsanfrage wird eingeleitet
199                                     // und mit dem Zeichen 0xFF quittiert.
200             {
201                   putchar(0xFF);
202                   get_new_State = 1;
203             }
204             else  // Gültigen Folgezustand empfangen. (hier könnte
205                   // auch eine Bereichsprüfung hinsichtlich der implemen-
206                   // tierten Modi stattfinden, ist aber nicht nötig, der bei
```

```
207                     // der Wahl eines nichimplematierten Modi der default-
208                     // Mode eingeschaltet wird, der das Programm in den
209                     // Leerlauf versetzt)
210               {
211                     putchar(new_State);  // Empfangenen Folgezustand quittieren
212                     get_new_State = 0;   // Zustandwechsel als beendet makieren
213                     State = new_State;   // Folgezustand übernehmen
214                     use_Display(1,0);    // Zustand auf dem Diplay ausgeben
215                     printf("Mode: %d", State);
216                     use_Display(0,0);
217               }
218         }
219
220
221   }
222
223   // Die Funktion wartet auf dem Empfang einer von der PC-Steuersoftware
224   // gesendeten Adresse. Funktion wird entweder mit Rückgabe der empfangenen
225   // Adresse beendet. Oder durch die unterbrechung auf Grund einer
226   // Zustandsänderunganfrage mit 0 beendet.
227   unsigned char get_Address__from_PC(void)
228   {
229         get_new_Address = 0;          // Flag zurücksetzen
230         while(get_new_Address == 0)  // Warten bis neue Adresse empfangen ist
231         {
232               if(get_new_State == 1) // Zustandsänderunganfrage empfangen?
233               {
234                     break;           // Funtion abbrechen
235               }
236         }
237         return Address;
238   }
239
240
241   // Die Funktion sendet die zuletzt empfangene Temperatur und Spannung
242   // an die PC-Steuersoftware
243   void Send_Data_to_PC(unsigned short Temperatur, unsigned short Spannung)
244   {
245         putchar( (unsigned char)((Temperatur >> 8) & 0xFF) );
246         putchar( (unsigned char)((Temperatur >> 0) & 0xFF) );
247         putchar( (unsigned char)((Spannung >> 8) & 0xFF) );
248         putchar( (unsigned char)((Spannung >> 0) & 0xFF) );
249   }
250
251
252   // Die Funktion use_Display steuert die Ausgabe auf dem Display mit printf.
253   void use_Display(unsigned char use_Display, unsigned char Zeile)
254   {
255         Display = use_Display;      // 0: RS232, 1: Display
256
257         if(use_Display == 1)        // Ausgabe über das Diplay?
258         {
259               if(Zeile == 0)        // Obere Zeile des Diplays löschen
260               {
261                     SEND_CMD (DD_RAM_ADDR);
262                     printf("                ");
263                     SEND_CMD (DD_RAM_ADDR);
264
265               }
266               else                  // Untere Zeile des Diplays löschen
267               {
```

```
268                     SEND_CMD (DD_RAM_ADDR2);
269                     printf("                ");
270                     SEND_CMD (DD_RAM_ADDR2);
271                 }
272         }
273 }
274
275
276 // Die Funktion Display_Vals stellt die gerundeten Werte für die zuletzt
277 // empfangene Temperatur und Spannung mit der Jeweiligen Transponderadresse
278 // auf dem Diplay dar.
279 void Display_Vals(unsigned char Address, unsigned short Temperatur, unsigned short
        Spannung)
280 {
281       unsigned char Temperature_High = 0;
282       unsigned char Voltage_High = 0;
283
284       // Berechnung der auf 200 Grad und 16 Bit skalierten Temperatur
285       // 0xFFFF -> 200 Grad
286       // 0x0000 ->   0 Grad
287       Temperature_High = (unsigned char)(((unsigned long)200 * (unsigned long)
              Temperatur) / (unsigned long)65535);
288
289       // Berechnung der auf 5 Volt und 16 Bit skalierten Spannung
290       // 0xFFFF -> 5 V
291       // 0x0000 -> 0 V
292       Voltage_High = (unsigned char)(((unsigned long)5 * (unsigned long)Spannung) / (
              unsigned long)65535);
293
294       // Ausgabe der Transponderadresse, Transpndertemperatur
295       // und der Transponderbertiebsspannung
296       use_Display(1,2);
297       printf("%u T: %uGC V: %uV", Address ,Temperature_High , Voltage_High);
298       use_Display(0,2);
299 }
300
301
302
303 // Die folgenden Funktionen für die Ansteuerung des Displays sind aus dem
304 // Beispielcode von Olimex for das Bord MSP430-169STK entnommen. Die
305 // Kommentare wurden aus dem Beispielcode beibehalten.
306 void Delay (unsigned int a)
307 {
308   unsigned char k;
309   for (k=0 ; k != a; ++k);
310 }
311
312 void Delayx100us(unsigned char b)
313 {
314   unsigned char j;
315   for (j=0; j!=b; ++j) Delay (_100us);
316 }
317
318 void _E(void)
319 {
320     E_HIGH;                 //toggle E for LCD
321       _NOP();
322       _NOP();
323       E_LOW;
324 }
325 void SEND_CHAR (unsigned char d)
```

```
326 {
327       Delayx100us(5);       //.5ms
328       temp = d & 0xf0;      //get upper nibble
329       LCD_Data &= 0x0f;
330       LCD_Data |= temp;
331       RS_HIGH;              //set LCD to data mode
332       _E();                 //toggle E for LCD
333       temp = d & 0x0f;
334       temp = temp << 4;     //get down nibble
335       LCD_Data &= 0x0f;
336       LCD_Data |= temp;
337       RS_HIGH;              //set LCD to data mode
338       _E();                 //toggle E for LCD
339 }
340
341 void SEND_CMD (unsigned char e)
342 {
343       Delayx100us(10);       //10ms
344       temp = e & 0xf0;       //get upper nibble
345       LCD_Data &= 0x0f;
346       LCD_Data |= temp;      //send CMD to LCD
347       RS_LOW;                //set LCD to CMD mode
348       _E();                  //toggle E for LCD
349       temp = e & 0x0f;
350       temp = temp << 4;      //get down nibble
351       LCD_Data &= 0x0f;
352       LCD_Data |= temp;
353       RS_LOW;                //set LCD to CMD mode
354       _E();                  //toggle E for LCD
355 }
356
357 void InitLCD(void)
358 {
359       P4OUT = 0;             //LCD ini
360         P4DIR = 0xff;
361     RS_LOW;
362     Delayx100us(250);        //Delay 100ms
363     Delayx100us(250);
364     Delayx100us(250);
365     Delayx100us(250);
366     LCD_Data |= BIT4 | BIT5;            //D7-D4 = 0011
367     LCD_Data &= ~BIT6 & ~BIT7;
368     _E();                               //toggle E for LCD
369     Delayx100us(100);                   //10ms
370     _E();                               //toggle E for LCD
371     Delayx100us(100);                   //10ms
372     _E();                               //toggle E for LCD
373     Delayx100us(100);                   //10ms
374     LCD_Data &= ~BIT4;                  //D7-D4 = 0010
375     _E();                               //toggle E for LCD
376
377     SEND_CMD(DISP_ON);
378     SEND_CMD(CLR_DISP);
379     SEND_CMD(DD_RAM_ADDR);
380 }
```

A.3.8 Flash.h

Listing 15: Flash.h

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: Flash.h                                                        */
4  /* Autor: Tobias Krannich                                                    */
5  /* Kurzbeschreibung: Heder für Flash.c                                       */
6  /*****************************************************************************/
7
8
9
10 #ifndef FLASH_H_
11 #define FLASH_H_
12
13 // Defines zum Ansprechen des Flashes
14 #define MAX_BLOCK_NUMB      1024
15
16 #define TRANS_LDY           50
17 #define WRITE_DLY           400
18 #define ERASE_DLY           4000
19
20 #define OUT_PORT            P5OUT
21 #define IN_PORT             P5IN
22 #define IO_DIR              P5DIR
23
24 #define _CE_ON              P2OUT &= ~BIT0
25 #define _CE_OFF             P2OUT |= BIT0
26 #define _RE_ON              P2OUT &= ~BIT1
27 #define _RE_OFF             P2OUT |= BIT1
28 #define _WE_ON              P2OUT &= ~BIT2
29 #define _WE_OFF             P2OUT |= BIT2
30
31 #define ALE_ON              P2OUT |= BIT3
32 #define ALE_OFF             P2OUT &= ~BIT3
33 #define CLE_ON              P2OUT |= BIT4
34 #define CLE_OFF             P2OUT &= ~BIT4
35
36 #define R_B                 P2IN & BIT7
37 #define DALLAS              P2IN & BIT5
38
39 #define READ_SPARE          0x50
40 #define READ_0              0x00
41 #define READ_1              0x01
42 #define READ_STATUS         0x70
43
44 #define WRITE_PAGE          0x80
45 #define WRITE_AKN           0x10
46
47 #define ERASE_BLOCK         0x60
48 #define ERASE_AKN           0xD0
49
50 #define DEV_ID              0x90
51
52 #define SAMSUNG_ID          0xECE6
53
54 #define   INPUT           0
55 #define   OUTPUT          0xff
56 #define   ON              1
57 #define   OFF             0
58 #define   BUF_SIZE        16
59
60 #define   ZEICHEN          0xAA
```

```
61
62
63  // Struktur für die zu speichernden Datensätze
64  typedef struct{
65        unsigned char C0;
66        unsigned char C1;
67        unsigned char C2;
68        unsigned char C3;
69        unsigned char C4;
70        unsigned char C5;
71        unsigned char C6;
72        unsigned char C7;
73        unsigned char C8;
74        unsigned char C9;
75        unsigned char C10;
76        unsigned char C11;
77        unsigned char C12;
78        unsigned char C13;
79        unsigned char C14;
80        unsigned char C15;
81        } DATA_STRUCT;
82
83  // Prototypen
84  void init_Flash (void);
85  unsigned char Read_Data(void);
86  void Erase_Flash_All(void);
87  void Write_Ring_Flash(DATA_STRUCT *);
88  void Data_Struct_To_Write_Buffer(DATA_STRUCT *);
89  void Val_To_Data(DATA_STRUCT *, unsigned long);
90  void Vals_To_Data(DATA_STRUCT *, unsigned long, unsigned short, unsigned short,
        unsigned char);
91  DATA_STRUCT * Read_Ring_Flash(void);
92  void MemTest(void);
93
94  // Prototypen zu übernommenen Funktionen (Beispielapplikation Olimex MSP430-169STK)
95  unsigned char Erase_Flash (unsigned char BLOCK_ADDL, unsigned char BLOCK_ADDH);
96  void Write_Data(unsigned char a);
97  void Inactive_Flash(void);
98  unsigned char Programm_Bytes(unsigned char COL_ADD,
99                               unsigned char ROW_ADDL,
100                              unsigned char ROW_ADDH,
101                              unsigned char NUMBER);
102 void Read_Bytes (unsigned char COL_ADD,
103                  unsigned char ROW_ADDL,
104                  unsigned char ROW_ADDH,
105                  unsigned char NUMBER);
106
107
108
109
110 #endif /*FLASH_H_*/
```

A.3.9 Flash.c

Listing 16: Flash.c

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: Flash.c                                                        */
```

```
4  /* Autor: Tobias Krannich                                                  */
5  /* Kurzbeschreibung: Stellt Funktion für das Speichern und Auslesen des im  */
6  /*                   externen Flash realisierten Ringspeichers zur Verfügung */
7  /*************************************************************************/
8
9  #include "Flash.h"
10 #include <msp430x16x.h>
11 #include <stdio.h>
12
13
14 volatile unsigned char WRITE_BUF [BUF_SIZE]; // Buffer, der für das Schreiben
15                                              // des Speichers notwendig ist
16 volatile unsigned char READ_BUF [BUF_SIZE];  // Buffer, der für das lesen
17                                              // des Speichers notwendig ist
18 volatile unsigned short Index_Ring;         // Gibt die aktuelle Speicher-
19                                              // position im Ringspeicher an
20 volatile unsigned short Count_Ring;         // Gibt die Anzahl der gespeich-
21                                              // erten Datensätze an
22 volatile unsigned char Ring_Full;           // Gibt an, ob der Ringspeicher
23                                              // schon einmal komplett voll-
24                                              // geschrieben wurde
25 volatile unsigned short Index_Last;         // Zeigt auf den ältesten Daten-
26                                              // satz im Ringspeicher
27
28
29 // Die Funktion init_Flash initialisiert den Port 2 für die
30 // Kommunikation mit dem Speicherbaustein.
31 void init_Flash (void)
32 {
33       P2OUT=0x07;
34       P2DIR=0x1F;
35 }
36
37
38
39 // Die Funktion Erase_Flash_All löscht den gesamten Flashspeicher
40 void Erase_Flash_All(void)
41 {
42       int i = 0;           // for()
43       unsigned char ADDL;  // unteres Adressbyte
44       unsigned char ADDH;  // oberes Adressbyte
45
46       for(i=0; i<0x1000; i++)
47       {
48             ADDL = (unsigned char)(i & 0xFF);
49             ADDH = (unsigned char)((i>>8) & 0xFF);
50             Erase_Flash(ADDL,ADDH);
51       }
52 }
53
54 // Die Funktion Write_Ring_flash ermittelt automatisch die aktuelle
55 // Speicherposition im Speicher und Schreibt den übergebenen Datensatz
56 // in den Speicher. Zusätzlich werden die Adressen und Speicherfüllstände
57 // aktualisiert.
58 // Zudem wird wenn nötig der nächste Block im Speicher gelöscht.
59 // [Flash kann nur Blockweise nicht Byteweise gelöscht werden! 1Block = 4Kb]
60 // Der genaue Speicheraufbau ist mit Hilfe einer Abbildung in der Dokumentation
61 // dargestellt.
62 void Write_Ring_Flash(DATA_STRUCT *pData)
63 {
64       unsigned char ADDL;  // unteres Adressbyte
```

```
65        unsigned char ADDH;  // oberes Adressbyte
66        unsigned short ADD;  // Adresse 16 Bit (Zeile)
67        unsigned char COL;   // Spalte (siehe Abbildung in der Dokumentation)
68
69
70        ADD = Index_Ring / 16;  // Berechnung der Adresse (Zeile) [siehe Doku]
71        COL = Index_Ring % 16;  // Berechnung der Spalte [siehe Doku]
72
73        // Aufteilung in oberes und unteres Adressbyte
74        ADDL = (unsigned char)(ADD & 0xFF);
75        ADDH = (unsigned char)((ADD>>8) & 0xFF);
76
77
78        if( (Index_Ring%256) == 0) Letze Adresse im Block erreicht?
79        {
80              // Lösche den nächsten Block
81              Erase_Flash(ADDL,ADDH);
82        }
83
84        // Übergebenen Datensatz in den Schreibpuffer kopieren
85        Data_Struct_To_Write_Buffer(pData);
86
87        // Datensatz aus dem Schreibpuffer in die zuvor berechnete Speicher-
88        // position schreiben.
89        Programm_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);
90
91        if(Count_Ring < 65535) // Maximale Speicheranzahl noch NICHT erreicht?
92        {
93              // Anzahl der gespeicherten Datensätze erhöhen
94              Count_Ring++;
95        }
96        else  // Speicher voll?
97        {
98              // Flag setzen, das angibt, dass der Speicher voll ist.
99              // Das Flag wird nie zurück gesetzt! (siehe Read_Ring_Flash)
100             Ring_Full = 1;
101       }
102
103       Index_Ring++;
104       if(Index_Ring >= 65536)  // Letze Speicheradresse überschritten?
105       {
106             // Index_Ring auf die erste Speicheradresse setzen
107             Index_Ring = 0;
108       }
109
110
111 }
112
113
114 // Die Funktion Data_Struct_To_Write_Buffer kopiert den übergebenen
115 // Datensatz in den Schreibpuffer, damit dieser in den Speicher
116 // geschrieben weren kann.
117 void Data_Struct_To_Write_Buffer(DATA_STRUCT *pData)
118 {
119       WRITE_BUF[0] = pData->C0;
120       WRITE_BUF[1] = pData->C1;
121       WRITE_BUF[2] = pData->C2;
122       WRITE_BUF[3] = pData->C3;
123       WRITE_BUF[4] = pData->C4;
124       WRITE_BUF[5] = pData->C5;
125       WRITE_BUF[6] = pData->C6;
```

```
126         WRITE_BUF[7] = pData->C7;
127         WRITE_BUF[8] = pData->C8;
128         WRITE_BUF[9] = pData->C9;
129         WRITE_BUF[10] = pData->C10;
130         WRITE_BUF[11] = pData->C11;
131         WRITE_BUF[12] = pData->C12;
132         WRITE_BUF[13] = pData->C13;
133         WRITE_BUF[14] = pData->C14;
134         WRITE_BUF[15] = pData->C15;
135 }
136
137
138 // Die Funktion Val_To_Data schreibt einen 32 Bit langen Wert in die
139 // ersten 4 Datenbytes der Datenstruktur. Die Funktion wird für den
140 // Speichertest benötigt, der den Speicher komplett vollschreibt und
141 // wieder ausließt.
142 void Val_To_Data(DATA_STRUCT *pData, unsigned long i)
143 {
144         pData->C0 = (unsigned char)((i>>0) & 0xFF);
145         pData->C1 = (unsigned char)((i>>8) & 0xFF);
146         pData->C2 = (unsigned char)((i>>16) & 0xFF);
147         pData->C3 = (unsigned char)((i>>24) & 0xFF);
148 }
149
150 // Die Funktion Vals_To_Data kopiert die übergebenen Werte an die
151 // vorgesehenen Stellen in der Datenstruktur.
152 void Vals_To_Data(DATA_STRUCT *pData, unsigned long Tick_Index, unsigned short
        Temperature, unsigned short Voltage, unsigned char Adress)
153 {
154         pData->C0 = (unsigned char)((Adress>>0) & 0xFF);
155         pData->C1 = (unsigned char)((Adress>>8) & 0xFF);
156
157         pData->C2 = (unsigned char)((Tick_Index>>0) & 0xFF);
158         pData->C3 = (unsigned char)((Tick_Index>>8) & 0xFF);
159         pData->C4 = (unsigned char)((Tick_Index>>16) & 0xFF);
160         pData->C5 = (unsigned char)((Tick_Index>>24) & 0xFF);
161
162         pData->C6 = (unsigned char)((Temperature>>0) & 0xFF);
163         pData->C7 = (unsigned char)((Temperature>>8) & 0xFF);
164
165         pData->C8 = (unsigned char)((Voltage>>0) & 0xFF);
166         pData->C9 = (unsigned char)((Voltage>>8) & 0xFF);
167
168         // Nicht benötigten Datenbytes 0 setzen
169         pData->C10 = 0x00;
170         pData->C11 = 0x00;
171         pData->C12 = 0x00;
172         pData->C13 = 0x00;
173         pData->C14 = 0x00;
174         pData->C15 = 0x00;
175 }
176
177
178 // Die Funktion Read_Ring_Flash ermittelt selbstständig die Adresse des
179 // ältesten Datensatzes im Speicher und ließt diesen aus. Die Daten
180 // werden in der Struktur von der Funktion zurückgegeben. Zusätzlich
181 // werden die Adressen und Speicherfüllstände aktualisiert.
182 DATA_STRUCT * Read_Ring_Flash(void)
183 {
184         unsigned short ADD = 0;  // 16 Bit breite Adresse (Zeile)
185         unsigned char ADDH = 0;  // oberes Adressbyte
```

```
186         unsigned char ADDL = 0;  // unteres Adressbyte
187         unsigned char COL = 0;   // Spalte (siehe Abbildung in der Dokumentation)
188 
189         // Struktur für die gelesenen Daten
190         static DATA_STRUCT Data = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
191 
192         // Für den Zeigerzugriff auf die Struktur
193         DATA_STRUCT *pData = &Data;
194 
195 
196 
197         // Letztes Element im Ringbuffer ermitteln
198         if(Ring_Full == 0)      // Ringbuffer hatte noch keinen Überlauf?
199         {
200               // Keine Daten im Ringspeicher???
201               if(Count_Ring == 0)
202               {
203                     // Lesen abbrechen -> NULL-Zeiger zurückgeben
204                     return NULL;
205               }
206 
207               // Index für die auszulesende Speicherposition berechenen
208               Index_Last = Index_Ring - Count_Ring;
209         }
210         else // Es gab schon einen Überlauf -> lese ab dem ersten Struct
211         {
212               // Keine Daten im Ringspeicher???
213               if((Count_Ring - 255) == 0)
214               {
215                     // Lesen abbrechen -> NULL-Zeiger zurückgeben
216                     return NULL;
217               }
218 
219               // Index für die auszulesende Speicherposition berechenen
220               Index_Last = Index_Ring - (Count_Ring + 1 - 256);
221         }
222 
223         // Anzahl der gespeicherten Datensätze --1
224         Count_Ring--;
225 
226         // Adresse aus dem Index berechnen
227         ADD = Index_Last / 16;
228         COL = Index_Last % 16;
229         ADDL = (unsigned char)(ADD & 0xFF);
230         ADDH = (unsigned char)((ADD>>8) & 0xFF);
231 
232         // Speicher auslesen
233         Read_Bytes(COL*16,ADDL,ADDH,BUF_SIZE);
234 
235         // Read_BUF in das Datenstruct schreiben
236         pData->C0  = READ_BUF[0];
237         pData->C1  = READ_BUF[1];
238         pData->C2  = READ_BUF[2];
239         pData->C3  = READ_BUF[3];
240         pData->C4  = READ_BUF[4];
241         pData->C5  = READ_BUF[5];
242         pData->C6  = READ_BUF[6];
243         pData->C7  = READ_BUF[7];
244         pData->C8  = READ_BUF[8];
245         pData->C9  = READ_BUF[9];
246         pData->C10 = READ_BUF[10];
```

```
247         pData->C11 = READ_BUF[11];
248         pData->C12 = READ_BUF[12];
249         pData->C13 = READ_BUF[13];
250         pData->C14 = READ_BUF[14];
251         pData->C15 = READ_BUF[15];
252 
253         return pData; // Datenstruktur zurück geben
254 
255 
256 }
257 
258 
259 // Die funktion MemTest wurde zur Überprüfung der geschriebenen
260 // Speicherfunktionen erstellt. Auf Grund der langen Laufzeit (ca. 1 Stunde)
261 // wird die Funktion im laufenden Betrieb nicht eingesetzt. Für
262 // die Erstinbtriebnahme sollte sie aber einmal ausgeführt werden,
263 // um zu sehen ob der Speicher in Ordnung ist.
264 void MemTest(void)
265 {
266         unsigned long i = 0;  // 4 Byte
267 
268         DATA_STRUCT Data = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
269         DATA_STRUCT *pData = &Data;
270         DATA_STRUCT *p_ReceiveData;
271 
272         // Init Ring_Flash
273         Ring_Full = 0;
274         for(i=0; i<16;i++)
275         {
276               READ_BUF[i] = 0;
277               WRITE_BUF[i] = 0;
278         }
279         Index_Ring = 0;
280         Count_Ring = 0;
281 
282         pData->C0 = ZEICHEN;
283         pData->C1 = ZEICHEN;
284         pData->C2 = ZEICHEN;
285         pData->C3 = ZEICHEN;
286         pData->C4 = ZEICHEN;
287         pData->C5 = ZEICHEN;
288         pData->C6 = ZEICHEN;
289         pData->C7 = ZEICHEN;
290         pData->C8 = ZEICHEN;
291         pData->C9 = ZEICHEN;
292         pData->C10 = ZEICHEN;
293         pData->C11 = ZEICHEN;
294         pData->C12 = ZEICHEN;
295         pData->C13 = ZEICHEN;
296         pData->C14 = ZEICHEN;
297         pData->C15 = ZEICHEN;
298 
299         printf("\n\r\n\r\n\rSTART\n\r");
300 
301         printf("Schreiben\n\r");
302         for(i=0; i<5; i++)
303         {
304               Val_To_Data(pData, i);
305               Write_Ring_Flash(pData);
306         }
307 
```

```
308         printf("Lesen\n\r");
309         for(i=0; i<6; i++)
310         {
311               p_ReceiveData = Read_Ring_Flash();
312               if(p_ReceiveData == NULL)
313               {
314                     printf("RingBuffer leer!\n\r");
315               }
316               else
317               {
318                     printf("%ld %u %u %u ||||  ", i, Index_Ring, Count_Ring, Index_Last);
319                     printf("%X %X %X %X %X %X %X %X %X %X %X %X (%ld) \n\r",
                          p_ReceiveData->C15, p_ReceiveData->C14, p_ReceiveData->C13,
                          p_ReceiveData->C12, p_ReceiveData->C11, p_ReceiveData->C10,
                          p_ReceiveData->C9, p_ReceiveData->C8, p_ReceiveData->C7,
                          p_ReceiveData->C6, p_ReceiveData->C5, p_ReceiveData->C4, (
                          unsigned long)((unsigned long)p_ReceiveData->C3 * (unsigned long)
                          16777216 + (unsigned long)p_ReceiveData->C2 * (unsigned long)
                          65536 + (unsigned long)p_ReceiveData->C1 * (unsigned long)256 + (
                          unsigned long)p_ReceiveData->C0) );
320               }
321         }
322
323         pData->C15 = 0xBB;
324         printf("Schreiben\n\r");
325         for(i=0; i<7; i++)
326         {
327               Val_To_Data(pData, i);
328               Write_Ring_Flash(pData);
329         }
330
331         printf("Lesen\n\r");
332         for(i=0; i<4; i++)
333         {
334               p_ReceiveData = Read_Ring_Flash();
335               if(p_ReceiveData == NULL)
336               {
337                     printf("RingBuffer leer!\n\r");
338               }
339               else
340               {
341                     printf("%ld %u %u %u ||||  ", i, Index_Ring, Count_Ring, Index_Last);
342                     printf("%X %X %X %X %X %X %X %X %X %X %X %X (%ld) \n\r",
                          p_ReceiveData->C15, p_ReceiveData->C14, p_ReceiveData->C13,
                          p_ReceiveData->C12, p_ReceiveData->C11, p_ReceiveData->C10,
                          p_ReceiveData->C9, p_ReceiveData->C8, p_ReceiveData->C7,
                          p_ReceiveData->C6, p_ReceiveData->C5, p_ReceiveData->C4, (
                          unsigned long)((unsigned long)p_ReceiveData->C3 * (unsigned long)
                          16777216 + (unsigned long)p_ReceiveData->C2 * (unsigned long)
                          65536 + (unsigned long)p_ReceiveData->C1 * (unsigned long)256 + (
                          unsigned long)p_ReceiveData->C0) );
343               }
344         }
345
346         pData->C15 = 0xCC;
347         printf("Schreiben\n\r");
348         for(i=0; i<65546; i++)
349         {
350               Val_To_Data(pData, i);
351               Write_Ring_Flash(pData);
352         }
```

```
353
354         printf("Lesen\n\r");
355         for(i=0; i<65536; i++)
356         {
357               p_ReceiveData = Read_Ring_Flash();
358               if(p_ReceiveData == NULL)
359               {
360                     printf("RingBuffer leer!\n\r");
361               }
362               else
363               {
364                     printf("%ld %u %u %u ||||  ", i, Index_Ring, Count_Ring, Index_Last);
365                     printf("%X %X %X %X %X %X %X %X %X %X %X %X (%ld) \n\r",
                          p_ReceiveData->C15, p_ReceiveData->C14, p_ReceiveData->C13,
                          p_ReceiveData->C12, p_ReceiveData->C11, p_ReceiveData->C10,
                          p_ReceiveData->C9, p_ReceiveData->C8, p_ReceiveData->C7,
                          p_ReceiveData->C6, p_ReceiveData->C5, p_ReceiveData->C4, (
                          unsigned long)((unsigned long)p_ReceiveData->C3 * (unsigned long)
                          16777216 + (unsigned long)p_ReceiveData->C2 * (unsigned long)
                          65536 + (unsigned long)p_ReceiveData->C1 * (unsigned long)256 + (
                          unsigned long)p_ReceiveData->C0) );
366               }
367         }
368
369         pData->C15 = 0xDD;
370         printf("Schreiben\n\r");
371         for(i=0; i<280; i++)
372         {
373               Val_To_Data(pData, i);
374               Write_Ring_Flash(pData);
375         }
376
377         printf("Lesen\n\r");
378         for(i=0; i<300; i++)
379         {
380               p_ReceiveData = Read_Ring_Flash();
381               if(p_ReceiveData == NULL)
382               {
383                     printf("RingBuffer leer!\n\r");
384               }
385               else
386               {
387                     printf("%ld %u %u %u ||||  ", i, Index_Ring, Count_Ring, Index_Last);
388                     printf("%X %X %X %X %X %X %X %X %X %X %X %X (%ld) \n\r",
                          p_ReceiveData->C15, p_ReceiveData->C14, p_ReceiveData->C13,
                          p_ReceiveData->C12, p_ReceiveData->C11, p_ReceiveData->C10,
                          p_ReceiveData->C9, p_ReceiveData->C8, p_ReceiveData->C7,
                          p_ReceiveData->C6, p_ReceiveData->C5, p_ReceiveData->C4, (
                          unsigned long)((unsigned long)p_ReceiveData->C3 * (unsigned long)
                          16777216 + (unsigned long)p_ReceiveData->C2 * (unsigned long)
                          65536 + (unsigned long)p_ReceiveData->C1 * (unsigned long)256 + (
                          unsigned long)p_ReceiveData->C0) );
389               }
390         }
391   }
392
393
394
395   // Übernommene Funktionen (Beispielapplikation Olimex MSP430-169STK)
396   void Inactive_Flash(void)  //pull flash pins to inactive condition
397   {
```

```
398   IO_DIR=INPUT;          //IO is inputs
399   _CE_OFF;               //=1
400   _RE_OFF;               //=1
401   _WE_OFF;               //=1
402   ALE_OFF;               //=0
403   CLE_OFF;               //=0
404 }
405
406 void Write_Data(unsigned char a)
407 {
408   IO_DIR=OUTPUT;         //IO is outputs
409   _WE_ON;
410   OUT_PORT=a;
411   _WE_OFF;               //latch data
412 }
413
414 unsigned char Read_Data(void)
415 {
416   unsigned char f;
417   IO_DIR=INPUT;          //IO is inputs
418   _RE_ON;
419   f=IN_PORT;
420   _RE_OFF;               //read data
421   return (f);
422 }
423
424 unsigned char Programm_Bytes(unsigned char COL_ADD,
425                              unsigned char ROW_ADDL,
426                              unsigned char ROW_ADDH,
427                              unsigned char NUMBER)
428 {
429         unsigned char k, l;
430         Inactive_Flash();
431         CLE_ON;
432         _CE_ON;
433         Write_Data(WRITE_PAGE);
434         CLE_OFF;
435         ALE_ON;
436         Write_Data(COL_ADD);
437         Write_Data(ROW_ADDL);
438         Write_Data(ROW_ADDH);
439         ALE_OFF;
440         for (k=0; k != NUMBER; k++)
441           {
442              l=WRITE_BUF[k];
443              Write_Data(l);
444           }
445         CLE_ON;
446         Write_Data(WRITE_AKN);
447         while ((R_B) == 0);
448         Write_Data(READ_STATUS);
449         CLE_OFF;
450         l = Read_Data();
451         Inactive_Flash();
452         return(l);
453 }
454
455 void Read_Bytes (unsigned char COL_ADD,
456                  unsigned char ROW_ADDL,
457                  unsigned char ROW_ADDH,
458                  unsigned char NUMBER)
```

```
459 {
460         unsigned char n, r;
461         Inactive_Flash();
462         CLE_ON;
463         _CE_ON;
464         Write_Data(READ_0);
465         CLE_OFF;
466         ALE_ON;
467         Write_Data(COL_ADD);
468         Write_Data(ROW_ADDL);
469         Write_Data(ROW_ADDH);
470         ALE_OFF;
471         while ((R_B) == 0);
472         for (n=0; n != NUMBER; n++)
473           {
474              r=Read_Data();
475              READ_BUF[n] = r;
476           }
477         Inactive_Flash();
478 }
479
480 // Es wird der Block gelöscht, in dem die Adresse liegt
481 unsigned char Erase_Flash (unsigned char BLOCK_ADDL, unsigned char BLOCK_ADDH)
482 {
483         unsigned char m;
484         Inactive_Flash();
485         CLE_ON;
486         _CE_ON;
487         Write_Data(ERASE_BLOCK);
488         CLE_OFF;
489         ALE_ON;
490         Write_Data(BLOCK_ADDL);
491         Write_Data(BLOCK_ADDH);
492         ALE_OFF;
493         CLE_ON;
494         Write_Data(ERASE_AKN);
495         while ((R_B) == 0);
496         Write_Data(READ_STATUS);
497         CLE_OFF;
498         m = Read_Data();
499         Inactive_Flash();
500         return (m);
501 }
```

A.3.10 CLK.h

Listing 17: CLK.h

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: CLK.h                                                          */
4  /* Autor: !!!Stephan Plaschke!!!                                             */
5  /* Kurzbeschreibung: Heder für CLK.c                                         */
6  /*****************************************************************************/
7
8  #ifndef CLK_H_
9  #define CLK_H_
10
11
```

```
12 #define CLK_DIVIDER 1
13 #define CLK 8000000/CLK_DIVIDER
14
15 /* e.g. 1/(2MHz)*488 = 0,000244s = 4098kHz      */
16 #define MHZ_1       244
17 #define MHZ_2       488
18 #define MHZ_4       976
19 #define MHZ_8      1952
20 #endif /*CLK_H_*/
21
22
23 unsigned char XT2_set (unsigned int);
```

A.3.11 CLK.c

Listing 18: CLK.c

```
1  /*****************************************************************************/
2  /* Projekt: Readerquellcode                                                  */
3  /* Dateiname: CLK.c                                                          */
4  /* Autor: !!!Stephan Plaschke!!!                                             */
5  /* Kurzbeschreibung: Steuerung des internen Taktgenerators                   */
6  /*****************************************************************************/
7
8  #include "CLK.h"  // Header enthält verschiedene Defines für mögliche
9                    // einzustellbare Taktraten
10 #include <msp430x16x.h>
11
12
13
14 // Die Funktion XT2_set setzt den internen Taktgenerator auf die
15 // übergebene Frequenz. Der Rückgabewert der Funktion gibt an,
16 // ob die Taktfrequenz erfolgreich gesetzt wurde.
17 unsigned char XT2_set (unsigned int clk_multi)
18 {
19       unsigned int Max_turns = 1000;
20       unsigned char i, Error_code = 0;
21
22       BCSCTL1 &= ~XT2OFF;                  /* XT2on */
23
24       do
25       {
26               IFG1 &= ~OFIFG;             /* Clear OSCFault flag */
27           for (i = 0xFF; i > 0; i--);     /* Time for flag to set */
28
29           if (!(--Max_turns))           /* stop after 1k tries, error with ext osc! */
30           {
31                 Error_code = 1;
32                 break;
33           }
34       }
35       while ((IFG1 & OFIFG));             /* OSCFault flag still set? */
36
37       BCSCTL2 |= SELM_2 | SELS;           /* MCLK = SMCLK = XT2 (safe) */
38
39       switch (clk_multi)
40       {
41       case MHZ_8:       BCSCTL2 &= ~(DIVS_3 | DIVM_3);      /* SMCLK&MCLK = XT2 / 1 */
42                          BCSCTL1 &= ~DIVA_3;                /* ACLK = XT2 / 1 */
```

```
43                          break;
44          case MHZ_4:     BCSCTL2 |= DIVS_1 | DIVM_1;         /* SMCLK&MCLK = XT2 / 2 */
45                          BCSCTL1 |= DIVA_1;                  /* ACLK = XT2 / 2 */
46                          break;
47          case MHZ_2:     BCSCTL2 |= DIVS_2 | DIVM_2;         /* SMCLK&MCLK = XT2 / 4 */
48                          BCSCTL1 |= DIVA_2;                  /* ACLK = XT2 / 4 */
49                          break;
50          case MHZ_1:     BCSCTL2 |= DIVS_3 | DIVM_3;         /* SMCLK&MCLK = XT2 / 8 */
51                          BCSCTL1 |= DIVA_3;                  /* ACLK = XT2 / 8 */
52                          break;
53          default:        Error_code = 2;
54                          BCSCTL2 |= DIVS_1 | DIVM_1;         /* SMCLK&MCLK = XT2 / 2 */
55                          BCSCTL1 |= DIVA_1;                  /* ACLK = XT2 / 2 */
56                          break;
57          }
58
59          return (Error_code);
60  }
```

A.4 Firmware Reader Controller-Feldsteuerung

A.4.1 main.c

Listing 19: main.c

```
1   /*****************************************************************************/
2   /* Projekt: Readerquellcode (Feldansteuerung                                 */
3   /* Dateiname: main.c                                                         */
4   /* Autor: Tobias Krannich                                                    */
5   /* Kurzbeschreibung: Steuert die Anregung des Reihenschwingkreises           */
6   /*****************************************************************************/
7
8   #define BAUDRATE 9600
9
10  #include <msp430x16x.h>
11  #include <signal.h>
12  #include "CLK.h"
13  #include "stdio.h"
14
15  int putchar (int);
16  void set_enable(unsigned char);
17  unsigned short sampling_ADC(void);
18  void resonanz_match(void);
19
20  int main(void)
21  {
22
23          int BaudRegister = 0;
24
25          WDTCTL = WDTPW + WDTHOLD;   // Stop WDT
26          XT2_set(MHZ_8);             // CLK = externer 8MHz Takt
27                                      // -> nur 4MHz, da 4MHz Quartz angeschloessen ist
28
29          // Timer
30          TACTL = TASSEL1 + TACLR;    // SMCLK, clear TAR, Divider = 1
31          CCTL0 = OUTMOD_4;           // Toggle Pin bei Outputcompare
32          TACCR0 = 15;                // 4000/250 - 1 = 15
```

```
33          TACTL |= MC0;               // Starte Timer im UpMode
34
35
36
37          // Init UART0
38          UCTL0 = SWRST + CHAR;   // set 8bit, none parity and 1 stoppbit
39          UTCTL0 |= SSEL1;                // set SMCLK as USART0_CLK
40          BaudRegister = 4000000/BAUDRATE;
41          UBR00 = BaudRegister & 0xFF;    // 4000000/9600 = 0x01A0
42          UBR10 = (BaudRegister >> 8) & 0xFF;
43          UMCTL0 = 0x00;
44          ME1 |= UTXE0 + URXE0;   // enable UART0 RX&TX
45          UCTL0 &= ~SWRST;                // clear SWRST
46          IE1 |= URXIE0;          // enable RX ISR
47
48          // ADC
49          ADC12CTL0 = ADC12ON;    // ADC12ON / reference on Avcc
50
51          // Timer-Ausgang
52          P1DIR |= BIT1;              // Ausgang für Outputcompare
53          P1SEL |= BIT1;              // Timerfunktion für P1.1
54
55
56          // Ausgänge für Enable
57          P3DIR |= BIT0 + BIT1 + BIT2;
58          P3OUT = 0;
59          P4DIR |= 0xFF;
60          P4OUT = 0x1B;
61
62          // Serielle Schnitstelle
63          P3DIR |= BIT4;              // TX USART0
64          P3DIR &=~ BIT5;             // RX USART0
65          P3SEL |= BIT4 | BIT5;       // TX, RX USART0
66
67          // ADC Eingang
68          P6SEL |= 0x01;
69
70
71          resonanz_match();
72
73
74          _EINT();        // enable Interrupts
75
76          while(1)
77          {
78                  nop();
79          }
80
81          return 0;
82  }
83
84
85  // Funktion sendet einzelndes Zeichen über die serielle
86  // Schnittstelle
87  int putchar (int out_char)
88  {
89          while((IFG1 & UTXIFG0) != UTXIFG0); // Warte auf die Fertigstellung
90                                              // des vorigen Sendevorgangs
91          TXBUF0 = out_char;  // Zu sendenes Zeichen in Ausgangsbuffer schreiben
92          return 0;
93  }
```

```
94
95
96  // ISR für den Empfang von Zeichen über die serielle Schnittstelle
97  interrupt(USART0RX_VECTOR) isr_UART_RX(void)
98  {
99          unsigned char Zeichen;
100         unsigned short ADC_Value = 0;
101
102         // Zeichen einlesen
103         Zeichen = RXBUF0;
104
105         // Empfangene Kapazität einstellen
106         set_enable(Zeichen);
107
108         // Resonanzspannung ermitteln
109         ADC_Value = sampling_ADC();
110
111         // Ermittelte Spannung zurücksenden
112         putchar( (unsigned char)((ADC_Value >> 8) & 0xFF));
113         putchar( (unsigned char)((ADC_Value >> 0) & 0xFF));
114
115 }
116
117 // Die Funktion setzt die Ausgänge für die Treiberansteuerung
118 // Da die Port-Indexe nicht zu den Kondensator-Indexen passen,
119 // müssen die Buts zunächst gespiegelt werden, bevor sie auf
120 // dem Port ausgegeben werden.
121 // (Port[0] steuert C[0]) und C[0] ist die größte Kapazität...
122 void set_enable(unsigned char Val)
123 {
124         unsigned char Reverse = 0;
125         unsigned char i = 0;
126
127         for(i=0;i<8;i++)
128         {
129                 Reverse |= ( (Val >> i)& 0x01) << (7 -i);
130         }
131
132         P4OUT = Reverse;
133 }
134
135 // Funktion ermittelt die Resonanzspannung
136 unsigned short sampling_ADC(void)
137 {
138   ADC12CTL0 |= ADC12SC + ENC;
139   ADC12CTL0 &= ~ADC12SC;
140   while ((ADC12CTL1 & ADC12BUSY) == 1);
141   return(ADC12MEM0);
142 }
143
144 // Die Funktion stellt verschiedene Kapazitäten für
145 // den Schwingkreis ein. Anschließend wird die
146 // Resonanzspannung über den ADC ermittelt und bewertet.
147 // Nachdem alle Kapazitäten eingestellt wurden, wird die
148 // Kapazität eingestellt, bei der die größte
149 // Resonanzspannung festgestellt wurde
150 void resonanz_match(void)
151 {
152         unsigned short i = 0;
153         unsigned short ADC_Voltage = 0;
154         unsigned short ADC_MAX_Voltage = 0;
```

```
155         unsigned short MAX_Index;
156
157         for(i=0; i<0xFF; i++)
158         {
159                 set_enable(i);
160                 ADC_Voltage = sampling_ADC();
161                 if(ADC_Voltage > ADC_MAX_Voltage)
162                 {
163                         ADC_MAX_Voltage = ADC_Voltage;
164                         MAX_Index = i;
165                 }
166         }
167         set_enable(MAX_Index);
168 }
```

A.4.2 CLK.h

Listing 20: CLK.h

```
1  #ifndef CLK_H_
2  #define CLK_H_
3
4
5  #define CLK_DIVIDER 1
6  #define CLK 8000000/CLK_DIVIDER
7
8  /* e.g. 1/(2MHz)*488 = 0,000244s = 4098kHz      */
9  #define MHZ_1   244
10 #define MHZ_2   488
11 #define MHZ_4   976
12 #define MHZ_8   1952
13 #endif /*CLK_H_*/
14
15
16 unsigned char XT2_set (unsigned int);
```

A.4.3 CLK.c

Listing 21: CLK.c

```
1
2  #include "CLK.h"
3  #include <msp430x16x.h>
4
5  unsigned char XT2_set (unsigned int clk_multi)
6  /*-------------------------------------------------------------------------*/
7  {
8         unsigned int Max_turns = 1000;
9         unsigned char i, Error_code = 0;
10
11        BCSCTL1 &= ~XT2OFF;             /* XT2on */
12
13        do
14        {
15                IFG1 &= ~OFIFG;             /* Clear OSCFault flag      */
```

```
16              for (i = 0xFF; i > 0; i--); /* Time for flag to set */
17
18              if (!(--Max_turns))         /* stop after 1k tries, error with ext osc!  */
19              {
20                  Error_code = 1;
21                  break;
22              }
23          }
24          while ((IFG1 & OFIFG));         /* OSCFault flag still set?             */
25
26          BCSCTL2 |= SELM_2 | SELS;               /* MCLK = SMCLK = XT2 (safe)    */
27
28          switch (clk_multi)
29          {
30          case MHZ_8:     BCSCTL2 &= ~(DIVS_3 | DIVM_3);  /* SMCLK&MCLK = XT2 / 1
                                */
31                              BCSCTL1 &= ~DIVA_3;                             /* ACLK
                                    = XT2 / 1                               */
32                              break;
33          case MHZ_4:     BCSCTL2 |= DIVS_1 | DIVM_1;             /* SMCLK&MCLK = XT2 / 2
                                */
34                              BCSCTL1 |= DIVA_1;                              /* ACLK
                                    = XT2 / 2                               */
35                              break;
36          case MHZ_2:     BCSCTL2 |= DIVS_2 | DIVM_2;             /* SMCLK&MCLK = XT2 / 4
                                */
37                              BCSCTL1 |= DIVA_2;                              /* ACLK
                                    = XT2 / 4                               */
38                              break;
39          case MHZ_1:     BCSCTL2 |= DIVS_3 | DIVM_3;             /* SMCLK&MCLK = XT2 / 8
                                */
40                              BCSCTL1 |= DIVA_3;                              /* ACLK
                                    = XT2 / 8                               */
41                              break;
42          default:        Error_code = 2;
43                              BCSCTL2 |= DIVS_1 | DIVM_1;             /* SMCLK&MCLK =
                                    XT2 / 2                 */
44                              BCSCTL1 |= DIVA_1;                              /* ACLK
                                    = XT2 / 2                               */
45                              break;
46          }
47
48          return (Error_code);
49  }
```

A.5 PC-Applikation

A.5.1 ControlCenterDlg.h

Listing 22: ControlCenterDlg.h

```
1  // Projekt: ControlCenter
2  // Datei: ControlCenterDlg.h
3  // Autor: Tobias Krannich
4  // Beschreiung: Haeder für ControlCenterDlg.cpp
5
6  #include <iostream>
```

```
7
8  #if !defined(AFX_CONTROLCENTERDLG_H__8EB382AA_1F1B_456C_B3D6_D38C207630DA__INCLUDED_)
9  #define AFX_CONTROLCENTERDLG_H__8EB382AA_1F1B_456C_B3D6_D38C207630DA__INCLUDED_
10
11 #if _MSC_VER > 1000
12 #pragma once
13 #endif // _MSC_VER > 1000
14
15 /////////////////////////////////////////////////////////////////////////////
16 // CControlCenterDlg Dialogfeld
17 class CControlCenterDlg : public CDialog
18 {
19 // Konstruktion
20 public:
21         CControlCenterDlg(CWnd* pParent = NULL);  // Standard-Konstruktor
22
23 // Dialogfelddaten
24         //{{AFX_DATA(CControlCenterDlg)
25         enum { IDD = IDD_CONTROLCENTER_DIALOG };
26         CChartViewer    m_CHART;
27         CString m_EDIT_Temperature;
28         CString m_EDIT_Voltage;
29         CString m_EDIT_Temperature_T2;
30         CString m_EDIT_Temperature_T3;
31         CString m_EDIT_Voltage_T2;
32         CString m_EDIT_Temperature_T4;
33         CString m_EDIT_Voltage_T4;
34         CString m_EDIT_Voltage_T3;
35         BOOL    m_GET_VALUES;
36         //}}AFX_DATA
37
38         // Vom Klassenassistenten generierte Überladungen virtueller Funktionen
39         //{{AFX_VIRTUAL(CControlCenterDlg)
40         protected:
41         virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV-Unterstützung
42         //}}AFX_VIRTUAL
43
44 // Implementierung
45 protected:
46         HICON m_hIcon;
47
48         // Generierte Message-Map-Funktionen
49         //{{AFX_MSG(CControlCenterDlg)
50         virtual BOOL OnInitDialog();
51         afx_msg void OnPaint();
52         afx_msg HCURSOR OnQueryDragIcon();
53         afx_msg void OnTimer(UINT nIDEvent);
54         //}}AFX_MSG
55         DECLARE_MESSAGE_MAP()
56 public:
57         afx_msg void OnBnClickedButtonFlashToDb();
58 public:
59         afx_msg void OnBnClickedButtonDrawChart();
60         int m_START_YEAR;
61         int m_START_MONTH;
62         int m_START_DAY;
63         int m_START_HOUR;
64         int m_START_MINUTE;
65         int m_END_YEAR;
66         int m_END_MONTH;
67         int m_END_DAY;
```

```
68          int m_END_HOUR;
69          int m_END_MINUTE;
70          afx_msg void OnBnClickedButtonValuesToCsv();
71          CString CSV_FILENAME;
72          CString m_CSV_FILENAME;
73          int m_M_FILE_LIST;
74          afx_msg void OnBnClickedButtonExecute();
75          int m_TRANSPONDER_ADD_1;
76          int m_TRANSPONDER_ADD_2;
77          int m_TRANSPONDER_ADD_3;
78          int m_TRANSPONDER_ADD_4;
79  };
80
81  #endif // !defined(
        AFX_CONTROLCENTERDLG_H__8EB382AA_1F1B_456C_B3D6_D38C207630DA__INCLUDED_)
```

A.5.2 ControlCenterDlg.cpp

Listing 23: ControlCenterDlg.cpp

```
1   // Projekt: ControlCenter
2   // Datei: ControlCenterDlg.cpp
3   // Autor: Tobias Krannich
4   // Beschreiung: Klasse für die Kommunikation mit der
5   //              Oberfläche. Eventhandler der einzelnen
6   //              Objekte sind in dieser Klasse implementiert.
7   //              Initialisierungen und Aufbau der Oberfläche.
8
9   // Bindet alle weitern Headerdateien ein
10  #include "stdafx.h"
11
12  #include "ControlCenter.h"
13  #include "ControlCenterDlg.h"
14
15  #ifdef _DEBUG
16  #define new DEBUG_NEW
17  #undef THIS_FILE
18  static char THIS_FILE[] = __FILE__;
19  #endif
20
21  // Globale Variablen
22  UART my_uart1;      // Objekt für die Kommunikation über
23                      // die serielle Schnittstelle mit dem Reader
24  Database db1;       // Objekt für Datenbankzugriffe (PostgreSQL)
25  CMatlabEng matlab;  // Objekt für die Kommunikation mit MATLAB
26
27  // Anzahl von Transpondern, die vom Timer ausgelesen werden sollen
28  int NumberTransponderRead = 4;
29
30  /////////////////////////////////////////////////////////////////////////////
31  // CControlCenterDlg Dialogfeld
32
33  // Initialisierung der Membervariblen
34  CControlCenterDlg::CControlCenterDlg(CWnd* pParent /*=NULL*/)
35          : CDialog(CControlCenterDlg::IDD, pParent)
36          , m_START_YEAR(2008)
37          , m_START_MONTH(4)
38          , m_START_DAY(24)
39          , m_START_HOUR(16)
```

```
40          , m_START_MINUTE(7)
41          , m_END_YEAR(2008)
42          , m_END_MONTH(4)
43          , m_END_DAY(24)
44          , m_END_HOUR(16)
45          , m_END_MINUTE(8)
46          , CSV_FILENAME(_T(""))
47          , m_CSV_FILENAME(_T(""))
48          , m_M_FILE_LIST(0)
49          , m_TRANSPONDER_ADD_1(0)
50          , m_TRANSPONDER_ADD_2(0)
51          , m_TRANSPONDER_ADD_3(0)
52          , m_TRANSPONDER_ADD_4(0)
53  {
54          //{{AFX_DATA_INIT(CControlCenterDlg)
55          m_EDIT_Temperature = _T("");
56          m_EDIT_Voltage = _T("");
57          m_EDIT_Temperature_T2 = _T("");
58          m_EDIT_Temperature_T3 = _T("");
59          m_EDIT_Voltage_T2 = _T("");
60          m_EDIT_Temperature_T4 = _T("");
61          m_EDIT_Voltage_T4 = _T("");
62          m_EDIT_Voltage_T3 = _T("");
63          m_GET_VALUES = FALSE;
64          //}}AFX_DATA_INIT
65          // Beachten Sie, dass LoadIcon unter Win32 keinen
66          // nachfolgenden DestroyIcon-Aufruf benötigt
67          m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
68  }
69
70  // Methode für den Datenaustausch zwischem Objekt
71  // und Membervariable.
72  void CControlCenterDlg::DoDataExchange(CDataExchange* pDX)
73  {
74          CDialog::DoDataExchange(pDX);
75          //{{AFX_DATA_MAP(CControlCenterDlg)
76          DDX_Control(pDX, IDC_STATIC_CHART, m_CHART);
77          DDX_Text(pDX, IDC_EDIT_Temperature, m_EDIT_Temperature);
78          DDX_Text(pDX, IDC_EDIT_Voltage, m_EDIT_Voltage);
79          DDX_Text(pDX, IDC_EDIT_Temperature_T2, m_EDIT_Temperature_T2);
80          DDX_Text(pDX, IDC_EDIT_Temperature_T3, m_EDIT_Temperature_T3);
81          DDX_Text(pDX, IDC_EDIT_Voltage_T2, m_EDIT_Voltage_T2);
82          DDX_Text(pDX, IDC_EDIT_Temperature_T4, m_EDIT_Temperature_T4);
83          DDX_Text(pDX, IDC_EDIT_Voltage_T4, m_EDIT_Voltage_T4);
84          DDX_Text(pDX, IDC_EDIT_Voltage_T3, m_EDIT_Voltage_T3);
85          DDX_Check(pDX, IDC_CHECK_GET_VALS, m_GET_VALUES);
86          //}}AFX_DATA_MAP
87          DDX_Text(pDX, IDC_EDIT_START_YEAR, m_START_YEAR);
88          DDX_Text(pDX, IDC_EDIT_START_MONTH, m_START_MONTH);
89          DDX_Text(pDX, IDC_EDIT_START_DAY, m_START_DAY);
90          DDX_Text(pDX, IDC_EDIT_START_HOUR, m_START_HOUR);
91          DDX_Text(pDX, IDC_EDIT_START_MINUTE, m_START_MINUTE);
92          DDX_Text(pDX, IDC_EDIT_END_YEAR, m_END_YEAR);
93          DDX_Text(pDX, IDC_EDIT_END_MONTH, m_END_MONTH);
94          DDX_Text(pDX, IDC_EDIT_END_DAY, m_END_DAY);
95          DDX_Text(pDX, IDC_EDIT_END_HOUR, m_END_HOUR);
96          DDX_Text(pDX, IDC_EDIT_END_MINUTE, m_END_MINUTE);
97          DDX_Text(pDX, IDC_EDIT_CSV_NAME, CSV_FILENAME);
98          DDX_Text(pDX, IDC_EDIT_CSV_FILENAME, m_CSV_FILENAME);
99          DDX_CBIndex(pDX, IDC_COMBO_M_FILES, m_M_FILE_LIST);
100         DDX_Text(pDX, IDC_EDIT2_TRANSPODER_ADD_1, m_TRANSPONDER_ADD_1);
```

```
101         DDX_Text(pDX, IDC_EDIT3_TRANSPONDER_ADD_2, m_TRANSPONDER_ADD_2);
102         DDX_Text(pDX, IDC_EDIT4_TRANSPONDER_ADD_3, m_TRANSPONDER_ADD_3);
103         DDX_Text(pDX, IDC_EDIT5_TRANSPONDER_ADD_, m_TRANSPONDER_ADD_4);
104 }
105
106 // CControlCenterDlg Nachrichten-Handler
107 BEGIN_MESSAGE_MAP(CControlCenterDlg, CDialog)
108         //{{AFX_MSG_MAP(CControlCenterDlg)
109         ON_WM_PAINT()
110         ON_WM_QUERYDRAGICON()
111         ON_WM_TIMER()
112         //}}AFX_MSG_MAP
113         ON_BN_CLICKED(IDC_BUTTON_FLASH_TO_DB, &CControlCenterDlg::
                OnBnClickedButtonFlashToDb)
114         ON_BN_CLICKED(IDC_BUTTON_DRAW_CHART, &CControlCenterDlg::
                OnBnClickedButtonDrawChart)
115         ON_BN_CLICKED(IDC_BUTTON_VALUES_TO_CSV, &CControlCenterDlg::
                OnBnClickedButtonValuesToCsv)
116         ON_BN_CLICKED(IDC_BUTTON_EXECUTE, &CControlCenterDlg::OnBnClickedButtonExecute)
117 END_MESSAGE_MAP()
118
119
120
121 // Methode die beim Fensteraufbau aufgrufen wird.
122 // Hier können Initialisierungen vorgenommen werden.
123 BOOL CControlCenterDlg::OnInitDialog()
124 {
125         CDialog::OnInitDialog();
126
127     // Symbol für dieses Dialogfeld festlegen. Wird automatisch erledigt
128     //  wenn das Hauptfenster der Anwendung kein Dialogfeld ist
129         SetIcon(m_hIcon, TRUE);     // Großes Symbol verwenden
130         SetIcon(m_hIcon, FALSE);    // Kleines Symbol verwenden
131
132     // ZU ERLEDIGEN: Hier zusätzliche Initialisierung einfügen
133
134         // Parameter für die serielle Schnittstelle festlegen.
135         // Handle auf die serielle Schnittstelle öffnen
136         // COM-Port an dem der Reader angeschlossen ist automatisch suchen
137         my_uart1.Connect();
138
139         // Verbindung zur lokalen Datenbank aufbauen:
140         // IP-Adresse: 127.0.0.1
141         // Port: 5432
142         // dbname: postgres
143         // user: postgres
144         // password: Saturn06
145         db1.Connect_to_Database();
146
147         // Interval der Timer Methode auf 1 Sekunde stellen
148         SetTimer(1, 1000, NULL);
149
150         // Verbindung zu MATLAB der API MatlabEng herstellen
151         // Instanz öffnen und im Hintergrund laufen lassen
152         matlab.Open(NULL);
153         matlab.SetVisible(FALSE);
154
155         // TRUE zurück geben, außer ein Steuerelement soll den Fokus erhalten
156         return TRUE;
157 }
158
```

```
159 // Der folgende Code wird benötigt um die Minimieren Schaltfläche zu erstellen.
160 // (Noch nicht zuende geschrieben -> vielleicht später, wenn Zeit da ist)
161 void CControlCenterDlg::OnPaint()
162 {
163         if (IsIconic())
164         {
165                 CPaintDC dc(this); // Gerätekontext für Zeichnen
166
167                 SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);
168
169                 // Symbol in Client-Rechteck zentrieren
170                 int cxIcon = GetSystemMetrics(SM_CXICON);
171                 int cyIcon = GetSystemMetrics(SM_CYICON);
172                 CRect rect;
173                 GetClientRect(&rect);
174                 int x = (rect.Width() - cxIcon + 1) / 2;
175                 int y = (rect.Height() - cyIcon + 1) / 2;
176
177                 // Symbol zeichnen
178                 dc.DrawIcon(x, y, m_hIcon);
179         }
180         else
181         {
182                 CDialog::OnPaint();
183         }
184 }
185
186 // Die Systemaufrufe fragen die Cursorform ab, die angezeigt werden soll, während der
    Benutzer
187 // das zum Symbol verkleinerte Fenster mit der Maus zieht.
188 HCURSOR CControlCenterDlg::OnQueryDragIcon()
189 {
190         return (HCURSOR) m_hIcon;
191 }
192
193 // Eventhandler vom erstellten Timer. Wird jede Sekunde periodisch
194 // aufgerufen.
195 void CControlCenterDlg::OnTimer(UINT nIDEvent)
196 {
197         double Temperature_D = 0.00, Voltage_D = 0.00;
198
199         // Damit nur ein Transponder pro Durchgang abgefragt wird
200         static unsigned char Choice = 1;
201         char Temperature_Str[128];
202         char Voltage_Str[128];
203
204         // Dateninhalt aus den Objekten in die Membervariablen kopieren
205         UpdateData(TRUE);
206
207         // Datenabruf aktiviert?
208         if(m_GET_VALUES == TRUE)
209         {
210                 // Nachfolgend werden Daten-Anfragen zu den angegebenen
211                 // Transponderadressen an den Reader geschickt. Die
212                 // empfangenen Daten werden auf die Obefläche geschrieben.
213                 if(NumberTransponderRead >= 1 && Choice == 1)
214                 {
215                         // Temperatur und Spannung zu der in der Oberfläche
216                         // angegebenen Transponderadresse vom Reader abholen
217                         my_uart1.Get_TemperatureVoltage(m_TRANSPONDER_ADD_1,
                            Temperature_D, Voltage_D);
```

```
218
219                     // Werte in Strings umwandeln. Es werden CString
                            Membervariablen
220                     // für die Objekte verwendet, damit die Werte noch weiter
                            formatiert werden
221                     // können.
222                     sprintf(Temperature_Str, "%f", Temperature_D);
223                     sprintf(Voltage_Str, "%f", Voltage_D);
224
225                     // Datenstrings in die Membervariablen der Objekte kopieren
226                     m_EDIT_Temperature = Temperature_Str;
227                     m_EDIT_Voltage = Voltage_Str;
228
229                     // Inhalt der Membervariablen an die Objekte übergeben.
230                     UpdateData(FALSE);
231             }
232             if(NumberTransponderRead >= 2 && Choice == 2)
233             {
234
235                     // Temperatur und Spannung zu der in der Oberfläche
236                     // angegebenen Transponderadresse vom Reader abholen
237                     my_uart1.Get_TemperatureVoltage(m_TRANSPONDER_ADD_2,
                            Temperature_D, Voltage_D);
238
239                     // Werte in Strings umwandeln. Es werden CString
                            Membervariablen
240                     // für die Objekte verwendet, damit die Werte noch weiter
                            formatiert werden
241                     // können.
242                     sprintf(Temperature_Str, "%f", Temperature_D);
243                     sprintf(Voltage_Str, "%f", Voltage_D);
244
245                     // Datenstrings in die Membervariablen der Objekte kopieren
246                     m_EDIT_Temperature_T2 = Temperature_Str;
247                     m_EDIT_Voltage_T2 = Voltage_Str;
248
249                     // Inhalt der Membervariablen an die Objekte übergeben.
250                     UpdateData(FALSE);
251             }
252             if(NumberTransponderRead >= 3 && Choice == 3)
253             {
254                     // Temperatur und Spannung zu der in der Oberfläche
255                     // angegebenen Transponderadresse vom Reader abholen
256                     my_uart1.Get_TemperatureVoltage(m_TRANSPONDER_ADD_3,
                            Temperature_D, Voltage_D);
257
258                     // Werte in Strings umwandeln. Es werden CString
                            Membervariablen
259                     // für die Objekte verwendet, damit die Werte noch weiter
                            formatiert werden
260                     // können.
261                     sprintf(Temperature_Str, "%f", Temperature_D);
262                     sprintf(Voltage_Str, "%f", Voltage_D);
263
264                     // Datenstrings in die Membervariablen der Objekte kopieren
265                     m_EDIT_Temperature_T3 = Temperature_Str;
266                     m_EDIT_Voltage_T3 = Voltage_Str;
267
268                     // Inhalt der Membervariablen an die Objekte übergeben.
269                     UpdateData(FALSE);
270             }
```

```
271                 if(NumberTransponderRead >= 4 && Choice == 4)
272                 {
273                         // Temperatur und Spannung zu der in der Oberfläche
274                         // angegebenen Transponderadresse vom Reader abholen
275                         my_uart1.Get_TemperatureVoltage(m_TRANSPONDER_ADD_4,
                                Temperature_D, Voltage_D);
276
277                         // Werte in Strings umwandeln. Es werden CString
                                Membervariablen
278                         // für die Objekte verwendet, damit die Werte noch weiter
                                formatiert werden
279                         // können.
280                         sprintf(Temperature_Str, "%f", Temperature_D);
281                         sprintf(Voltage_Str, "%f", Voltage_D);
282
283                         // Datenstrings in die Membervariablen der Objekte kopieren
284                         m_EDIT_Temperature_T4 = Temperature_Str;
285                         m_EDIT_Voltage_T4 = Voltage_Str;
286
287                         // Inhalt der Membervariablen an die Objekte übergeben.
288                         UpdateData(FALSE);
289                 }
290
291                 // Transponder-Tooken noch nicht beim letzten Platz?
292                 if(Choice < NumberTransponderRead)
293                 {
294                         // Tooken an den nächsten Platz übergeben
295                         Choice++;
296                 }
297                 else // Tooken am letzten Platz?
298                 {
299                         // Tooken an den ersten Platz übergeben
300                         Choice = 0;
301                 }
302         }
303         CDialog::OnTimer(nIDEvent);
304 }
305
306 // Eventhandler für den Button, der das Auslesen des Readerflashes startet.
307 // Die vom Reader empfangenen Daten werden mit einen Zeitstempel in die Datenbank
        geschrieben
308 void CControlCenterDlg::OnBnClickedButtonFlashToDb()
309 {
310
311         unsigned short Numbers = 0;       // Anzahl der im Flash(Reader)
312                                           // gespeicherten Datensätze
313         int Numbers_of_Ticks = 0;         // Aktueller Tickstand im Reader
314         unsigned long TickIndex = 0;      // Tickstand des Datensatzes
315         unsigned char Address = 0x0000;   // Adresse des Transponders zum
316                                           // zugehörigen Datensatz
317         double Temperature = 0.00;        // Empfangene Temperatur
318         double Voltage = 0.00;            // Empfangene Spannung
319
320
321         // Dateninhalt aus den Objekten in die Membervariablen kopieren
322         UpdateData(TRUE);
323
324         // Für den Fall, dass keine Daten im Reader-Flash sind,
325         // können die folgenden beiden Zeilen zum Testen eingefügt
326         // werden. Sie veranlassen den Reader 10 Minuten lang Daten vom
327         // Transponder abzuholen und in den Flash zu schreiben.
```

```
328 //      my_uart1.ChangeMode(1);
329 //      Sleep(60000 * 10);
330
331         // Aktuellen Tickstand aus dem Reader abfragen.
332         // Wird für die Ermittlung der Zeit und des Datums
333         // desDatensatzes benötigt.
334         Numbers_of_Ticks = my_uart1.GetTick();
335
336         // Anzahl der im Flash gespeicherten Datensätze abfragen.
337         // Falls 0 Werte im Speicher sind vor der Schleife die
338         // Anzahl ermitteln
339         Numbers = my_uart1.GetNumberFlashValues();
340
341
342         // Alle Datensätze aus dem Reader-Speicher abfragen und
343         // mit Zeitstempel In die Datenbank schreiben.
344         while(Numbers > 0)
345         {
346
347                 my_uart1.GetFlashValue(Numbers, Address, TickIndex, Temperature,
                        Voltage);
348                 db1.Write_to_Database(Address,TickIndex, Numbers_of_Ticks, 1,
                        Temperature, Voltage);
349
350         }
351
352 }
353
354
355 // Der Eventhandler ließt das Start- und das Enddatum aus den Feldern ein.
356 // Anschließend werden die Datensätze für diesen Zeitraum aus der Datenbank
357 // ausgelesen und in einem XY-Diagram dargestellt.
358 void CControlCenterDlg::OnBnClickedButtonDrawChart()
359 {
360         int LabelStep = 0;  // Inkrement für die X-Achse
361         int Number = 0;     // Anzahl der Werte für den angegebenen Zeitraum
362         int i = 0;          // for()
363
364         // Dateninhalt aus den Objekten in die Membervariablen kopieren
365         UpdateData(TRUE);
366
367         // CTime Objekte mit dem eingestellten Start- und Enddatum erstellen
368         CTime Start(m_START_YEAR, m_START_MONTH, m_START_DAY, m_START_HOUR,
                m_START_MINUTE, 0);
369         CTime End(m_END_YEAR, m_END_MONTH, m_END_DAY, m_END_HOUR, m_END_MINUTE, 0);
370
371
372         // Anzahl der Datensätze für den angegebenen Zeitraum ermitteln.
373         // Und Abfrage der Werte starten.
374         Number = db1.Query(Start, End, 1);
375
376         // Dynamischen Speicher für die Anzahl der Werte kaufen
377         double *Temperature = new double[Number];
378         double *Voltage = new double[Number];
379
380         // Abfrageergebnis in den zuvor gekauften Felder (Temperatur, Spannung)
381         // schreiben.
382         db1.Get_Data(Temperature, Voltage, Number);
383
384         // Zeigerfeld für die Labels auf der Y-Achse kaufen
385         char **labels = new char*[Number];
```

```
386
387         // 10 Byte breite Speicher zu dem Zeigerfeld dazu kaufen.
388         // So das ein dynamisches 2-Dimensionales Feld entsteht.
389         for(i=0; i<Number; i++)
390         {
391                 labels[i] = new char[10];
392                 sprintf(labels[i], "%u",i);
393         }
394
395         // X-Achsen Inkrement so berechnen, dass 6 Label gesetzt werden.
396         LabelStep = (int)(Number/6);
397
398         // Neues Objekt vom Typ XYChart kaufen und über den Konstruktor
399         // Initialisieren.
400         // 600 * 300 Pixel
401         // Grauer Hintergrund (0xeeeeee )
402         // Schwarzer Rahmen
403         // 1 Pixel 3D-Effekt
404     XYChart *c = new XYChart(600, 300, 0xeeeeee, 0x000000, 1);
405
406         // 100mm * 70mm große Zeichenfläche mit weißem Hintergund im Diagramm erstellen
407         // Vertikale und horizontales Grid einstellen
408     c->setPlotArea(100, 70, 400, 180, 0xffffff)->setGridColor(0xcccccc, 0xcccccc);
409
410         // Legende oben in der Mitte erstellen. Schriftgöße 8 Arial Bold.
411     LegendBox *legendBox = c->addLegend(300, 70, false, "arialbd.ttf", 8);
412         legendBox->setAlignment(Chart::BottomCenter);
413
414         // Hintergrund und Rahmen der Legende Transparent setzen-
415     legendBox->setBackground(Chart::Transparent, Chart::Transparent);
416
417         // Labels auf die X-Achse setzen
418     c->xAxis()->setLabels(StringArray(labels, Number));
419
420         // Bei der X-Achse die Werte um LabelStep erhöhen.
421         // (LabelStep = 3 -> 0, 3, 6, 9, ...)
422     c->xAxis()->setLabelStep(LabelStep);
423
424         // X-Achse beschriften
425     c->xAxis()->setTitle("Minute");
426
427     // Y-Achse für die Temperatur beschriften
428     c->yAxis()->setTitle("Voltage\n(V)")->setAlignment(Chart::TopLeft2);
429
430         // Spannungsachse, Kurve und Label rot darstellen.
431     c->yAxis()->setColors(0xcc0000, 0xcc0000, 0xcc0000);
432
433         // Einteilung der Spannungsachse. Von 0 bis 5 mit
434         // dem Inkrement 1.
435         c->yAxis()->setLinearScale(0, 5, 1);
436
437
438
439     // Weiter Y-Achse für die Temperatur hinzufügen.
440     Axis *leftAxis = c->addAxis(Chart::Left, 50);
441
442         // Temperaturachse beschriften
443     leftAxis->setTitle("Temp\n(C)")->setAlignment(Chart::TopLeft2);
444
445     //Spannungsachse, Kurve, Label blau darstellen
446     leftAxis->setColors(0x0000cc, 0x0000cc, 0x0000cc);
```

```
447
448     // Werte für die Temperatur einfügen. Kurve 2 Pixel breit darstellen.
449     LineLayer *layer0 = c->addLineLayer(DoubleArray(Voltage,
450         Number ), 0xcc0000, "Voltage T1",3);
451     layer0->setLineWidth(2);
452
453         // Werte für die Spannung einfügen. Kurve 2 Pixel breit darstellen.
454     LineLayer *layer2 = c->addLineLayer(DoubleArray(Temperature,
455         Number), 0x0000cc, "Temperature T1",2);
456     layer2->setLineWidth(2);
457     layer2->setUseYAxis(leftAxis);
458
459
460     // Diagram über die Membervariable des Bildobjektes darstellen
461     m_CHART.setChart(c);
462
463     // Tool tip für das Diagram einfügen
464     m_CHART.setImageMap(
465         c->getHTMLImageMap("", "", "title='{xLabel}: US${value}K'"));
466
467         // Dynamisch gekauftes Objekt löschen.
468     delete c;
469
470         CClientDC dc(this);     // Für die bessere Darstellung...
471         dc.FillSolidRect(8,300,600,11, RGB(192, 192, 192));
472
473         // Speicher wieder freigeben
474         delete[] Temperature;
475         delete[] Voltage;
476
477         for(i=0; i<Number; i++)
478         {
479                 delete[] labels[i];
480         }
481         delete[] labels;
482
483 }
484
485 // Speichert Daten für den angegebenen Zeitraum im CSV-Format ab.
486 void CControlCenterDlg::OnBnClickedButtonValuesToCsv()
487 {
488         // Dateninhalt aus den Objekten in die Membervariablen kopieren
489         UpdateData(TRUE);
490
491         // CTime Objekte mit dem eingestellten Start- und Enddatum erstellen
492         CTime Start(m_START_YEAR, m_START_MONTH, m_START_DAY, m_START_HOUR,
                m_START_MINUTE, 0);
493         CTime End(m_END_YEAR, m_END_MONTH, m_END_DAY, m_END_HOUR, m_END_MINUTE, 0);
494
495         // Daten für den angegebenen Zeitraum in eine CSV-Datei mit dem
496         // eingegebenen Namen schreiben.
497         db1.CSV_Export((char *)(LPCTSTR)m_CSV_FILENAME, Start, End, 1);
498 }
499
500 // Eventhandler führt über die MatlabEng API MATLAB Befehle aus
501 // dem C++ Code aus. Mit der Methode EvalString werden die vorgegebenen
502 // m-Files Zeile für Zeile ausgeführt.
503 void CControlCenterDlg::OnBnClickedButtonExecute()
504 {
505         ifstream FileIn;  // Für das Einlesen der m-Files
506         string Line;      // Zwischenpuffer für die einzelnen
```

```
507                         // Zeilen aus dem m-File
508
509         // Dateninhalt aus den Objekten in die Membervariablen kopieren
510         UpdateData(TRUE);
511
512         // Erstes m_File aus der Liste ausführen
513         if(m_M_FILE_LIST == 0)  // Testfile sin.n
514         {
515                 // m-File öffnen
516                 FileIn.open("mFiles/sin.m");
517
518                 // Zeile für Zeile lesen und mittels der Methode
519                 // EvalString ausführen
520                 while (! FileIn.eof() )
521                 {
522                         getline (FileIn,Line);
523                         matlab.EvalString(Line.c_str());
524                 }
525                 FileIn.close();
526         }
527
528         // Zweites m_File aus der Liste ausführen
529         else if(m_M_FILE_LIST == 1)  // m-File: calc_start_temperature
530         {
531                 double Temperature[10];   // 10 Temperaturproben
532                 char Query_String[512];   // Für den Datenbankzugriff
533
534                 // Für die CSV-Datei mit der die Temperaturwerte
535                 // Aus der C++ Umgebung in das m-File kopiert werden
536                 ofstream Temperature_Value_File;
537
538                 int i = 0;      // for()
539
540                 // Startwert für den Abkühlungvorgang einlesen
541                 CTime Start(m_START_YEAR, m_START_MONTH, m_START_DAY, m_START_HOUR,
                        m_START_MINUTE, 0);
542
543                 // 22 Minuten Addieren (als Testbeispiel)
544                 Start += 22 * 60;
545
546                 // Endzeit 10 Minuten nach der Startzeit -> 10 Temperaturproben
547                 CTime End = Start + 10 * 60;    // 10 Minuten nach dem Start 16:29
548
549                 // String für die Datenbankabfrage erstellen
550                 sprintf(Query_String, "SELECT temperature FROM test.measurement WHERE
                        TO_CHAR(date, 'YYYY-MM-DD HH24:MI:SS') BETWEEN '%s' AND '%s' AND
                        transponder_id = (SELECT DISTINCT id FROM test.transponder WHERE
                        address = 1) AND TO_CHAR(date, 'SS') = '00' ORDER BY date",Start.
                        Format( "%Y-%m-%d %H:%M:%S" ), End.Format( "%Y-%m-%d %H:%M:%S"));
551
552                 // Abfrage ausführen
553                 db1.Query(Query_String);
554
555                 // Abfrageergebnis (10 Temperaturen) in das Feld für die Temperatur-
556                 // proben schreiben.
557                 db1.Get_Data(Temperature, 10);
558
559                 // 10 Temperaturwerte in Datei Speichern
560                 Temperature_Value_File.open("mFiles/temperature_data.csv");
561                 for(i=0; i<10; i++)
562                 {
```

```
563                         Temperature_Value_File << Temperature[i] << ",";
564                 }
565
566                 // Zeitdifferenz in Minuten zischen dem Start des Abkühl-
567                 // vorgangs und der ersten Temperaturprobe
568                 Temperature_Value_File << "22";
569                 Temperature_Value_File.close();
570
571
572                 // mFile calc_start_temperature ausführen
573                 FileIn.open("mFiles/calc_start_temperature.m");
574                 while (! FileIn.eof() )
575                 {
576                         getline (FileIn,Line);
577                         matlab.EvalString(Line.c_str());
578                 }
579                 FileIn.close();
580         }
581
582 }
```

A.5.3 UART.h

Listing 24: UART.h

```
1  // Projekt: ControlCenter
2  // Datei: Class_DB.h
3  // Autor: Tobias Krannich
4  // Beschreiung: Headerdatei für Class_DB.cpp
5
6  #include "stdafx.h"
7
8  class UART {
9
10 private:
11         HANDLE hCom;
12         DCB dcb;
13         COMMTIMEOUTS timeouts;
14         int ComIndex;
15         TCHAR COM[5];
16         int Global_Mode;
17
18 public:
19         UART();                                 // S-CTOR
20         ~UART();                                // DTOR
21         void Connect(void);
22         void Disconnect(void);
23         void SendReceive(unsigned char, unsigned char[], int);
24         void Send(unsigned char);
25         int Receive(unsigned char[], int);
26         int ChangeMode(unsigned char);
27         int Get_TemperatureVoltage(unsigned char, unsigned short&, unsigned short&);
28         int Get_TemperatureVoltage(unsigned char, double&, double&);    //überladen
29         unsigned short GetNumberFlashValues(void);
30         int GetTick(void);
31         int GetFlashValue(unsigned short&, unsigned char&, unsigned long&, double&,
               double&);
32         int GetCom(void){return ComIndex;};
33 };
```

A.5.4 UART.cpp

Listing 25: UART.cpp

```
1  // Projekt: ControlCenter
2  // Datei: Class_DB.cpp
3  // Autor: Tobias Krannich
4  // Beschreiung: Die Klasse stellt Methoden für die
5  // Kommunikation mit dem Reader zur Verfügung.
6
7  #include "stdafx.h"
8  using namespace std;
9
10
11 // Konstruktor der Klasse UART initialisiert die Variablen.
12 // Sucht automatisch den COM-Port an dem der Reader angeschlossen
13 // ist.
14 UART::UART()
15 {
16         char *ComPort[] = {"COM0", "COM1", "COM2", "COM3", "COM4", "COM5", "COM6", "
               COM7", "COM8", "COM9", "COM10"};
17         int i = 0;
18         unsigned char Receive[1];
19
20         // Initialisierung von Variablen
21         Global_Mode = -1;
22
23         // Automatische COM-Port-Suche
24         for(i=0; i<11; i++)
25         {
26                         wsprintf (COM, ComPort[i]);
27                         this->Connect();
28                         Receive[0] = 0x00;
29                         this->SendReceive(0xFF, Receive, 1);
30                         if( (Receive[0] == 0xAA) )
31                         {
32                                 this->Disconnect();
33                                 ComIndex = i;
34                                 break;
35                         }
36         }
37 }
38
39 // Destruktor trennt die Verbindung zur seriellen Schnittstelle
40 // (Reader)
41 UART::~UART()
42 {
43         this->Disconnect();     // falls es vergessen wird
44 }
45
46
47 // Stellt eine Verbindung über die serielle Schnittstelle zum Reader her.
48 // Die Einstellparameter sind fest eingestellt. Für eien Erweiterung
49 // könnten diese in eine Propertydatei ausgelagert werden.
50 void UART::Connect(void)
51 {
52         FillMemory(&dcb, sizeof(dcb), 0);
```

```
53
54          // Handle auf den aus dem Konstruktor ermittelten
55          // Com-Port öffnen.
56          hCom = CreateFile(COM,
57                                          GENERIC_READ | GENERIC_WRITE,
58                                          0,
59                                          NULL,
60                                          OPEN_EXISTING,
61                                          0,
62                                          NULL);
63
64          // Parameter für die serielle Schnittstelle
65          // festlegen.
66          dcb.BaudRate = 115200;
67          dcb.ByteSize = 8;
68          dcb.Parity = NOPARITY;
69          dcb.StopBits = ONESTOPBIT;
70          dcb.fDtrControl = DTR_CONTROL_DISABLE;
71          dcb.fInX = FALSE;
72
73          // Parameter einstellen
74          SetCommState(hCom, &dcb);
75
76          // Timeouts festlegen
77          timeouts.ReadIntervalTimeout=MAXDWORD;
78          timeouts.ReadTotalTimeoutMultiplier=2;
79          timeouts.ReadTotalTimeoutConstant=0;
80
81          // Timeouts einestellen
82          (SetCommTimeouts(hCom, &timeouts));
83  }
84
85  // Handel zum COM-Port schließen. (Verbindungaabbau)
86  void UART::Disconnect(void)
87  {
88          CloseHandle(hCom);
89  }
90
91  // Die Methode SendReceive Sendet das übergebene Zeichen
92  // über die serielle Schnittstelle an den Reader.
93  // Anschließend werden versucht "NumberReceive" Zeichen über
94  // die serielle Schnittstelle einzulesen (Antwort vom Reader).
95  // Kann der Reader nicht (vollständig) Antworten innerhalb der
96  // vorgegebenen Timeoutzeiten Antworten, wird der Empfang abgebrochen.
97  void UART::SendReceive(unsigned char SendChar, unsigned char Receive[], int
        NumberReceive)
98  {
99          DWORD Schreiben = 1;
100         DWORD Geschrieben;
101         DWORD BytesRead;
102         int i=0;
103         int Watchdog = 0;
104
105         // Verbindung hergestellt?
106         if(hCom != NULL)
107         {
108                 // Zeichen an den Reader Senden
109                 WriteFile( hCom, &SendChar, Schreiben, &Geschrieben, NULL);
110                 Sleep(50);
111
112                 // Auf die Antwort vom Reader warten und empangene
```

```
113                 // Zeichen in das übergebene Feld schreiben
114                 // Zeichen von der seriellen Schnittsatelle lesen.
115                 // Die Timeout Zeit wird mit der Variablen Watchdog
116                 // realisiert. So ergibt sich für das Empfangen
117                 // eines Zeichen in des Schleife eine Timeoutzeit
118                 // von 32 * TimeOutUART + Schleifenzeit.
119                 i=0;
120                 while( (i<NumberReceive) & (Watchdog<0x20) )
121                 {
122                         Watchdog++;
123                         ReadFile ( hCom, &Receive[i], 1, &BytesRead, NULL);
124                         if (BytesRead>0)
125                         {
126                                 i++;
127                                 Watchdog = 0;
128                         }
129                 }
130         }
131         // Rüchgabewert der empfangenen Zeichen einfügen.
132 }
133
134 // Sendet ein einzelnes Zeichen über die serielle Schnittstelle zum Reader
135 void UART::Send(unsigned char SendChar)
136 {
137         DWORD Schreiben = 1;
138         DWORD Geschrieben;
139
140         // Verbindung herhestellt?
141         if(hCom != NULL)
142         {
143                 // Sende Zeichen
144                 WriteFile( hCom, &SendChar, Schreiben, &Geschrieben, NULL);
145         }
146 }
147
148 // Die Methode ließt "NumberReceive" Zeichen über die serielle
149 // Schnittstelle ein. Können diese nicht vollständig innerhalb
150 // der Timeoutzeit empfangen werden, wird die Methode mit dem
151 // Rückgabewert -1 beendet. Werden die Zeichen vollständig empfangen,
152 // wird die Methode mit dem Rückgabewert 0 beendet.
153 int UART::Receive(unsigned char Receive[], int NumberReceive)
154 {
155         DWORD BytesRead;
156         int i=0;
157         int Watchdog = 0;
158
159         // Verbindung hergestellt?
160         if(hCom != NULL)
161         {
162                 i=0;
163                 // Zeichen von der seriellen Schnittsatelle lesen.
164                 // Die Timeout Zeit wird mit der Variablen Watchdog
165                 // realisiert. So ergibt sich für das Empfangen
166                 // eines Zeichen in des Schleife eine Timeoutzeit
167                 // von 32 * TimeOutUART + Schleifenzeit.
168                 while( (i<NumberReceive) & (Watchdog<0x20) )
169                 {
170                         Watchdog++;
171                         ReadFile ( hCom, &Receive[i], 1, &BytesRead, NULL);
172                         if (BytesRead>0)
173                         {
```

```
174                                 i++;
175                                 Watchdog = 0;
176                         }
177
178                 }
179         }
180
181         // Auswerten, ob alle Zeichen empfangen wurden.
182         if(i != NumberReceive)
183         {
184                 return -1;
185         }
186         else
187         {
188                 return 0;
189         }
190 }
191
192 // Mit der Methode ChangeMode wird eine Steuersequenz
193 // aus mehreren Zeichen an den Reader gesendet, die diesen
194 // veranlassen, in den übergebenen Betriebsmodus zu wechseln.
195 // Zur Sicherstellung der Modänderung wird diese über ein
196 // Handshake-Verfahren gesichert.
197 int UART::ChangeMode(unsigned char Mode)
198 {
199         unsigned char Receive[1];
200
201         // Anfrage für eine Modeänderung an den Reader senden.
202         this->Send(0x00);
203
204         // Befindet sich der Reader zur Zeit in Mode 1?
205         if(Global_Mode == 0x01)
206         {
207                 // Zusätzliche Wartezeit einfügen, da der Reader aus diesem
208                 // Mode nicht so schnell in die Betriebsmodeänderungsroutine
209                 // springen kann.
210                 Sleep(1000);
211         }
212         Sleep(200);
213
214         // Keine Antwort vom Reader erhalten?
215         if( this->Receive(Receive, 1) == -1 )
216                 return -1; // Methode mit Fehler abbrechen
217
218         // Antwort vom Reader erhalten, das er bereit ist
219         // einen neuen Betriebsmodus entgegen zu nehmen.
220
221         // Übergebenen Mode an den Reader senden.
222         this->Send(Mode);
223         Sleep(100);
224
225         // Keine Antwort vom Reader erhalten?
226         if( this->Receive(Receive, 1) == -1 )
227                 return -1; // Methode mit Fehler abbrechen
228
229         // Antwort vom Reader (Betriebsmodus in den er gewechselt ist)
230         // prüfen.
231         // Modus in dem der Reader jetzt läuft == übergebener Modus?
232         if(Receive[0] == Mode)
233         {
234                 // Modus vom Reader global speichern
```

```
235                 // (für die Steuerungen der nächstem Modeänderung)
236                 Global_Mode = Mode;
237
238                 // Methode erfolgreich beenden und den Modus zurückgeben.
239                 return Mode;
240         }
241         // Die Modi stimmen nicht überein?
242         else
243         {
244                 // Globalen Mode auf -1 setzen (Reader Betriebsmodus unbekannt)
245                 Global_Mode = -1;
246
247                 // Mit Fehler abbrechen
248                 return -1;
249         }
250 }
251
252 // Die überladene Methode
253 // Get_TemperatureVoltage(unsigned char, unsigned short&, unsigned short&)
254 // ermittelt die Aktuelle Temperatur und Betriebsspannung von dem Transponder
255 // mit der übergebenen Adresse und übergibt die ermittelten Werte per
256 // Referenz an die übergebenen Adressen für die Temperatur und Spannung.
257 // Für die weitere Verwndung müssen die 16Bit skallierten Werte noch
258 // umgerechnet werden.
259 int UART::Get_TemperatureVoltage(unsigned char Address, unsigned short& Temperature,
        unsigned short& Voltage)
260 {
261         unsigned char Receive[4];       // Temperature_highByte / lowByte;
                Voltage_highByte / llowByte
262
263         // Prüfen ob der Reader im Mode 2 laüft.
264         if(Global_Mode != 0x02)
265         {
266                 // Nein: -> Reader in den Mode 2 (azyklischer Mode) setzen
267                 if(this->ChangeMode(0x02) == -1)
268                         return -1;
269         }
270
271         // Adresse von dem Transponder an den Reader senden,
272         // für den die Werte ermittelt werden sollen.
273         this->Send(Address);
274         Sleep(500);
275
276         // Antwort vom Reader in Receive Speichern.
277         if(this->Receive(Receive, 4) == -1)
278                 return -1;
279
280         // Empfangene Daten in den Speicherplatz für die übergebenen
281         // Adressen schreiben.
282         Temperature = ((Receive[0] << 8) & 0xFF00) | (Receive[1] & 0xFF);
283         Voltage = ((Receive[2] << 8) & 0xFF00) | (Receive[3] & 0xFF);
284
285         return 0;
286 }
287
288 // Die überladene Methode
289 // Get_TemperatureVoltage(unsigned char, unsigned double&, unsigned double&)
290 // ermittelt die Aktuelle Temperatur und Betriebsspannung von dem Transponder
291 // mit der übergebenen Adresse und übergibt die ermittelten Werte per
292 // Referenz an die übergebenen Adressen für die Temperatur und Spannung.
```

```
293 int UART::Get_TemperatureVoltage(unsigned char Address, double& Temperature_D, double&
        Voltage_D)
294 {
295         unsigned short Temperature_S;
296         unsigned short Voltage_S;
297 
298         Temperature_D = 0;
299         Voltage_D = 0;
300 
301         // Temperatur und Spannung zu dem Transponder mit der übergebenen
302         // Adresse vom Reader abfragen.
303         if(this->Get_TemperatureVoltage(Address, Temperature_S, Voltage_S) == -1)
304                 return -1;
305 
306         // Werte in darstellbare Kommazahlen umrechnen
307         Temperature_D = (Temperature_S * 200.00) / 0xFFFF;
308         Voltage_D = (Voltage_S * 5.0) / 0xFFFF;
309 
310         return 0;
311 }
312 
313 
314 // Die Methode ermittelt die Anzahl von Datensätzen,
315 // die im Reader-Flash gespeichert sind.
316 unsigned short UART::GetNumberFlashValues(void)
317 {
318         unsigned char Receive[2];
319         unsigned short Number = 0;
320 
321         // Prüfen ob der Reader im Mode 3 laüft.
322         if(Global_Mode != 0x03)
323         {
324                 // Nein: -> Reader in den Mode 3 (Informationsabfrage Mode) setzen
325                 if(this->ChangeMode(0x03) == -1)
326                         return -1;
327         }
328 
329         // Steuerzeichen senden (1 -> Anfrage auf Anzahl Datenzätze)
330         this->Send(0x01);
331 
332         // Antwort vom Reader in Receive Speichern.
333         if(this->Receive(Receive, 2) == -1)
334                 return -1;
335 
336         // H_Byte und L_Byte zusammenfügen
337         Number = (Receive[0] << 8) | Receive[1] & 0xFF;
338 
339         // Anzahl zurückgeben
340         return Number;
341 
342 }
343 
344 // Ermittelt die den aktuellen Tick-Wert im Reader.
345 int UART::GetTick(void)
346 {
347         unsigned char Receive[4];
348         unsigned long Tick = 0;
349 
350         // Prüfen ob der Reader im Mode 3 laüft.
351         if(Global_Mode != 0x03)
352         {
```

```
353                 // Nein: -> Reader in den Mode 3 (Informationsabfrage Mode) setzen
354                 if(this->ChangeMode(0x03) == -1)
355                         return -1;
356         }
357
358         // Steuerzeichen senden (2 -> Anfrage auf aktuellen Tick-Wert)
359         this->Send(0x02);
360
361         // Antwort vom Reader in Receive Speichern.
362         if(this->Receive(Receive, 4) == -1)
363                 return -1;
364
365         // Bytes zu unsigned long zusammensetzen
366         Tick = Receive[0] << 24 | Receive[1] << 16 | Receive[2] << 8 | Receive[3] << 0;
367
368         // Aktuellen Tick-Wert zurückgeben
369         return Tick;
370 }
371
372 // Die Methode fragt den ältesten Datensatz aus dem Readerspeicher ab.
373 // Die vom Reader empfangenen Daten werden per Referenz an die übergebenen
374 // Paramter übergeben.
375 int UART::GetFlashValue(unsigned short& Numbers, unsigned char& Address, unsigned long&
        TickIndex, double& Temperature, double& Voltage)
376 {
377         unsigned char Receive[18];
378
379         // Prüfen ob der Reader im Mode 3 laüft.
380         if(Global_Mode != 0x03)
381         {
382                 // Nein: -> Reader in den Mode 3 (Informationsabfrage Mode) setzen
383                 if(this->ChangeMode(0x03) == -1)
384                         return -1;
385         }
386
387         // Steuerzeichen senden (3 -> Anfrage auf kompletten Datensatz (ältester))
388         this->Send(0x03);
389
390         // Antwort vom Reader in Receive Speichern.
391         if(this->Receive(Receive, 18) == -1)
392                 return -1;
393
394         // Empfangenen Bytes auf die Variablen aufteilen
395         Numbers = 0;
396         Address = 0;
397         TickIndex = 0;
398         TickIndex = 0;
399         Voltage = 0;
400
401         Numbers = Receive[1] << 0 | Receive[0] << 8;
402         Address = Receive[2] << 0 | Receive[3] << 8;
403         TickIndex = Receive[4] << 0 | Receive[5] << 8 | Receive[6] << 16 | Receive[7]
              << 24;
404         Temperature = (Receive[8] << 0 | Receive[9] << 8) * 200.0/65535;
405         Voltage = (Receive[10] << 0 | Receive[11] << 8) * 5.0/65535;
406
407
408         return 0;
409 }
```

A.5.5 ClassDB.h

Listing 26: ClassDB.h

```
1  // Projekt: ControlCenter
2  // Datei: Class_DB.h
3  // Autor: Tobias Krannich
4  // Beschreiung: Haeder für Class_DB.cpp
5
6  // C:\Programme\PostgreSQL\8.3\include
7  // C:\Programme\PostgreSQL\8.3\lib\libpq.lib
8
9  #ifndef DATABASE_H
10 #define DATABASE_H
11
12 #include "libpq-fe.h" // Datenbank API
13 #include <iostream>   // cout
14 #include <fstream>    // Dateizugriff
15 #include <string>     // Zeichenketten
16 #include <iomanip>    // cout formatieren
17 #include <time.h>     // Datum
18 #include <afx.h>      // CTime
19
20 using namespace std;
21
22
23 // Beschreibung der Klasse in Class_DB.cpp
24 class Database{
25 private:
26         PGconn *conn;
27         PGresult *Result;
28 public:
29         Database();     //CTOR
30         int Connect_to_Database();
31         void Display_Data();
32         void Query(char*);
33         int Query(CTime, CTime, int);
34         void Generate_Test_Data(int, int);
35         void Query_from_File(char*);
36         int Get_Data(double[], double[], int);
37         int Get_Data(double[],  int);
38         void Write_to_Database(unsigned char, unsigned long, unsigned long, int, double
               , double);
39         void CSV_Export(char*, CTime, CTime, int);
40
41 };
42
43 #endif
```

A.5.6 ClassDB.cpp

Listing 27: ClassDB.cpp

```
1  // Projekt: ControlCenter
2  // Datei: Class_DB.cpp
3  // Autor: Tobias Krannich
4  // Beschreiung: Die Klasse stellt Methoden für den Zugriff auf eine
5  //              Datenbank im Netzwerk zur Verfügung. Es gibt Methoden
```

```
6  //              mit vorgegebenen Abfragestrings, die für die Standart-
7  //              abfragen genutzt werden können. Zusätzlich besteht die
8  //              Möglichkeit zur Laufzeit generierte Abfragestrings aus-
9  //              zuführen. Zu dem gibt es Methoden, die den Datenaustausch
10 //              zwischen der Datenbank und der C++ Umgebung oder dem File-
11 //              system steuern.
12
13
14 #include "Class_DB.h"
15 using namespace std;
16
17 // Der Konstruktor hat noch keine Aufgaben. Es wäre möglich
18 // den Verbindungsaufbau in den Konstruktor einzufügen.
19 Database::Database()
20 {
21
22 }
23
24 // Methode baut eine Verbindung zu einer Datenbank im Netzwerk auf.
25 // Zur Vereinfachung des Programmes sind die Verbindungsparameter
26 // (IP-Adresse, Port, Datenkankname, user, Passwort) fest eingestellt.
27 // Für spätere Weiterentwicklung könnte es Sinvoll sein, diese Werte
28 // als Übergabeparameter zu setzen.
29 int Database::Connect_to_Database()
30 {
31
32         conn = PQconnectdb("hostaddr = '127.0.0.1' port = '5432' dbname = 'postgres'
               user = 'postgres' password = 'Saturn06' connect_timeout = '10'");
33
34         if (!conn)
35         {
36                 return -1;
37         }
38         else
39         {
40                 return 0;
41         }
42
43 }
44
45 // Die Methode Diplay Data gibt mit cout das Abfrageergebnis der
46 // vorigen Abfrage aus.
47 void Database::Display_Data()
48 {
49
50         int i = 0;
51         int j = 0;
52         int tuples = 0;
53         int column = 0;
54
55         // Anzahl der Datenzätze bestimmen
56         tuples = PQntuples(Result);
57
58         // Anzahl der Spalten bestimmen
59         column = PQnfields(Result);
60
61         // Atributnamen über der Tabelle ausgeben
62         for(i=0; i<column; i++)
63         {
64                 cout  << setw(14) << PQfname(Result, i) << " ";
65         }
```

```
66
67          cout << endl << "
                ------------------------------------------------------------------------
                " << endl;
68
69          // Werte in einer Tabelle darstellen
70          for(i=0; i<tuples; i++)
71          {
72                  for(j=0; j<column;j++)
73                  {
74                          cout << setiosflags(ios::right) << setw(10) << PQgetvalue(
                                Result, i, j) << "   " << resetiosflags(ios::right);
75                  }
76                  cout << endl;
77          }
78          cout << endl;
79
80  }
81
82  // Die überladene Methode Query(char *) führt eine belibige Datenbank-
83  // abfrage aus, die sie aus dem übergebenen String bekommt.
84  // Das Ergebnis der Abfrage wird in Result gespeichert.
85  void Database::Query(char *Query_String)
86  {
87          Result = PQexec(conn, Query_String);
88  }
89
90  // Die überladene Methode Query(CTime, CTime, int) startet eine Datenbank-
91  // abfrage, in der die Temperaturen und die Spannungen für den übergebenen
92  // Zeitraum und der übergebenen Transponderadresse aus der Datenbank geholt werden.
93  // Das Ergebnis der Abfrage wird in Result gespeichert. Die Anzahl der gelesenen
94  // Datensätze wird zurückgegeben.
95  int Database::Query(CTime T_Start, CTime T_End, int Address)
96  {
97          char Query_String[1024];
98          int Tuples;
99          sprintf(Query_String, "SELECT temperature, voltage FROM test.measurement WHERE
                TO_CHAR(date, 'YYYY-MM-DD HH24:MI:SS') BETWEEN '%s' AND '%s' AND
                transponder_id = (SELECT DISTINCT id FROM test.transponder WHERE address =
                %u) ORDER BY date",T_Start.Format( "%Y-%m-%d %H:%M:%S" ), T_End.Format( "%Y
                -%m-%d %H:%M:%S" ), Address);
100         Result = PQexec(conn, Query_String);
101         Tuples = PQntuples(Result);
102
103         return Tuples;
104 }
105
106 // Die Methode Generate_Test_Data löscht die komplette
107 // Datenbank und setzt sie neu auf. Zusätzlich werden Pseudo-
108 // daten in die Datenbank geschrieben. Der übergabeparameter
109 // Number stellt die Anzahl der zu erzeugenden Datensätze dar.
110 // Interval gibt das Zeitinkrement in Sekunden für die zu
111 // erstellenden Datensätze an.
112 void Database::Generate_Test_Data(int Number, int Interval)
113 {
114         int i = 0;
115         char Query_Sample[1024] = {"INSERT INTO test.measurement (id, voltage,
                temperature, transponder_id, date) VALUES (<ID>, <VOLTAGE>, <TEMPERATURE>,
                <TRANSPONDER_ID>, '<DATE>'"};
116         char Query[1024];
117         double Temperature;
```

```
118         double Voltage;
119         double Last_Temperature1 = 0;
120         double Last_Voltage1 = 0;
121         double Last_Temperature2 = 0;
122         double Last_Voltage2 = 0;
123         time_t Now;
124     tm *T;
125         char Timestamp[128];
126         CTime ct (2008, 1, 1, 0, 0, 0);
127 
128         // Aktuelle Zeit vom Betriebssystem holen
129         Now = ct.GetTime();
130 
131 
132         // Datenbank neu aufsetzen
133         this->Query_from_File("New_Database.sql");
134 
135         // Transponder hinzufügen
136         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (1, 1, '
                Transponder_1')");
137         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (2, 2, '
                Transponder_2')");
138         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (3, 3, '
                Transponder_3')");
139         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (4, 4, '
                Transponder_4')");
140         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (5, 5, '
                Transponder_5')");
141         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (6, 6, '
                Transponder_6')");
142         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (7, 7, '
                Transponder_7')");
143         this->Query("INSERT INTO test.transponder (id, address, name) VALUES (8, 8, '
                Transponder_8')");
144 
145         // Zufallsmesswerte erzeugen
146         srand ( time(NULL) );   // init rand()
147 
148         // T wird für den Zeitstempel auf die aktuelle Uhrzeit gesetzt.
149         T = localtime(&Now);
150 
151         // Daten für die Temperatur und die Spannung werden erzeugt. Damit die Werte
152         // nicht so stark springen werden die Werte durch eine gewichtete
                Mittelwertbildung
153         // zwischen dem letzen Wert und dem aktuellen Wert bestimmt.
154         for(i=0; i<Number*2; i = i + 2)
155         {
156                 // Dastensätze für die Transponderadresse 1 erzeugen
157                 Temperature = ( ((double)((rand() * 2)  % 50000) / 1000.00) + (3 *
                        Last_Temperature1) ) / 4;   // [* 2] weil rand() MAX default bei
                        32768 liegt
158                 Last_Temperature1 = Temperature;
159                 Voltage = ( ((double)(rand() % 5000) / 1000.00) + (3 * Last_Voltage1)
                        ) / 4;
160                 Last_Voltage1 = Voltage;
161                 T = localtime(&Now);
162                 Now += Interval;
163                 strftime(Timestamp, sizeof(Timestamp), "%Y-%m-%d %H:%M:%S", T);
164 
165                 // Die für Transponder 1 erzeugten Datensätze in die Datenbank
                        schreiben.
```

```
166                 sprintf(Query, "INSERT INTO test.measurement (id, voltage, temperature
                        , transponder_id, date) VALUES (%d, %.3lf, %.3lf, 1, '%s' )" , i,
                        Voltage, Temperature, Timestamp);
167                 this->Query(Query);
168
169                 // Dastensätze für die Transponderadresse 1 erzeugen
170                 Temperature = ( ((double)((rand() * 2)  % 50000) / 1000.00) + (3 *
                        Last_Temperature2) ) / 4;   // [* 2] weil rand() MAX default bei
                        32768 liegt
171                 Last_Temperature2 = Temperature;
172                 Voltage = ( ((double)(rand() % 5000) / 1000.00) + (3 * Last_Voltage2)
                        ) / 4;
173                 Last_Voltage2 = Voltage;
174
175                 // Die für Transponder 1 erzeugten Datensätze in die Datenbank
                        schreiben.
176                 sprintf(Query, "INSERT INTO test.measurement (id, voltage, temperature
                        , transponder_id, date) VALUES (%d, %.3lf, %.3lf, 2, '%s' )" , (i
                        +1), Voltage, Temperature, Timestamp);
177                 this->Query(Query);
178         }
179 }
180
181 // Die Methode Query_from_File führt die Abfragen aus einer SQL-Datei
182 // aus. Das Ergebnis der Abfrage wird in Result gespeichert.
183 void Database::Query_from_File(char *SQL_File)
184 {
185         int i = 0;
186         string Line_str;
187         char Line[1024];
188
189         // SQL-Datei öffnen.
190         ifstream pFile (SQL_File);
191         if (pFile.is_open())
192         {
193                 // Zeile für Zeile aus der Datei auslesen
194                 // und mit der Datenbank-API über die
195                 // Memberfunktion Query(char *) ausführen.
196                 while (! pFile.eof() )
197                 {
198                         getline (pFile,Line_str, ';');
199                         strcpy(Line, (Line_str + ';').c_str());
200                         this->Query(Line);
201                 }
202
203     pFile.close();
204         }
205         else cout << "Unable to open file";
206 }
207
208 // Die überladene Memberfunktion Get_Data(double[], double[], int)
209 // schreibt die Temperatur und die Spannung als Kommawerte in die
210 // übergebenen Felder. Lenght stellt die größe der Felder da, so dass
211 // für den Fall, dass die Datenmenge größer als die Felddimension ist
212 // keine Überschreibung über die Feldgrenzen auftritt. Damit Daten
213 // in dem Abfrageergebnis Result sind, muss zuvor mit
214 // Query(CTime, CTime, int) eine Abfrage ausgeführt werden.
215 int Database::Get_Data(double Voltage[], double Temperature[], int Lenght)
216 {
217         int i;
218         int Tuples;
```

```
219
220         Tuples = PQntuples(Result);
221
222         for(i=0; i<Tuples; i++)
223         {
224                 if(i >= Lenght) // Buffergröße überschritten?
225                         return i;
226
227                 Voltage [i]    = atof(PQgetvalue(Result, i, 0));
228                 Temperature[i] = atof(PQgetvalue(Result, i, 1));
229         }
230
231         return i;
232
233
234 }
235
236 // Die überladene Memberfunktion Get_Data(double[], int)
237 // schreibt die Temperaturen als Kommawerte in das
238 // übergebene Feld. Lenght stellt die größe des Feldes da, so dass
239 // für den Fall, dass die Datenmenge größer als die Felddimension ist
240 // keine Überschreibung über die Feldgrenzen auftritt. Damit Daten
241 // in dem Abfrageergebnis Result sind, muss zuvor mit
242 // Query(CTime, CTime, int) eine Abfrage ausgeführt werden.
243 int Database::Get_Data(double Temperature[], int Lenght)
244 {
245         int i;
246         int Tuples;
247
248         Tuples = PQntuples(Result);
249
250         for(i=0; i<Tuples; i++)
251         {
252                 if(i >= Lenght) // Buffergröße überschritten?
253                         return i;
254                 Temperature[i] = atof(PQgetvalue(Result, i, 0));
255         }
256
257         return i;
258 }
259
260 // Die Methode Write_to Database schreibt den übergebenen
261 // Datensatz in die Datenbank. Die Methode errechnet die
262 // Zeit für den jeweiligen datensatz aus den 3 Parametern:
263 // Tick_Index:       Tickwert für den jeweiligen Datensatz
264 // Number_of_Ticks:  Anzahl der Ticks zum Zeitpunkt der
265 //                   Datenübertragung vom Reader zum Transponder
266 // Samplerate:       Zeit in Sekunden zwischen zwei Ticks
267 void Database::Write_to_Database(unsigned char Address, unsigned long Tick_Index,
        unsigned long Numbers_of_Ticks, int SampleRate, double Temperature, double Voltage)
268 {
269         char Timestamp[128];
270         tm *T;
271         time_t Measurement;
272         char Query[1024];
273         unsigned long  id;
274
275         // Aktuelle Zeit holen
276         time_t Get_Time;
277         CTime now = CTime::GetCurrentTime();
278         Get_Time = now.GetTime();
```

```
279
280         // Zeit für den übergebenen Datensatz berechnen
281         // Ticksize Zur Zeit immer 1 Sekunde -> für die Weiterentwicklung
282         // variabel setzen
283         Measurement = Get_Time + (int)((Numbers_of_Ticks - Tick_Index)/4);
284
285         // Zeitstempel für den jeweiligen Datensatz erstellen.
286         T = localtime(&Measurement);
287         strftime(Timestamp, sizeof(Timestamp), "%Y-%m-%d %H:%M:%S", T);
288
289         // Aktuellen Wert des Primary Key erfragen.
290         this->Query("SELECT MAX(id) FROM test.measurement");
291         id = atoi(PQgetvalue(Result, 0, 0)) + 1;
292
293         // Datensatz mit Zeitstempel in die Datenbank schreiben.
294         sprintf(Query, "INSERT INTO test.measurement (id, voltage, temperature,
                transponder_id, date) VALUES (%d, %.3lf, %.3lf, %u, '%s' )" , id, Voltage,
                Temperature, Address, Timestamp);
295         this->Query(Query);
296 }
297
298 // Die Methode CSV_Export erstellt eine CSV-Datei mit den Datensätzen
299 // für den übergebenen Zeitraum. Zusätzlich wird eien zweite Datei mit
300 // dem gleichen Namen (Übergabeparameter: Name) und den Zusatz Info_...
301 // erzeugt, in dem die Information über die CSV-Datei wie folgt stehen.
302 //   Erstellungsdatum: 2008-04-26 14:48:03
303 //   Spalten: temperature voltage
304 //   Anzahl Messwerte: 63
305 //   Startzeit: 2008-04-24 16:07:00
306 //   Endzeit: 2008-04-24 16:08:00
307 //   Transponder Adresse: 1
308 void Database::CSV_Export(char *Name, CTime T_Start, CTime T_End, int Address)
309 {
310         int i = 0;
311         int j = 0;
312         int tuples = 0;
313         int column = 0;
314         ofstream Value_File;
315         ofstream Info_File;
316         char CSV_Value_Name[128] = "Matlab_Files/a.txt";
317         char CSV_Info_Name[128];
318         CTime now = CTime::GetCurrentTime();
319
320
321         //sprintf(CSV_Value_Name, "Matlab_Files/%s_Val_%s.txt", now.Format( "%Y%m%d%H%M
                %S" ), Name);
322         //sprintf(CSV_Info_Name, "Matlab_Files/%s_Info_%s.txt", now.Format( "%Y%m%d%H%M
                %S" ), Name);
323
324         // Dateinamen aus dem Übergabeparameter Name und den Zusätzen
325         // Val/Info bilden.
326         sprintf(CSV_Value_Name, "Files/Val_%s.txt", Name);
327         sprintf(CSV_Info_Name, "Files/Info_%s.txt", Name);
328
329         // Dateien erstellen
330         Value_File.open (CSV_Value_Name);
331         Info_File.open (CSV_Info_Name);
332
333         // Datenbankabfrage für den übergebenen Zeitraum und Transponder-
334         // adresse ausführen.
335         this->Query(T_Start, T_End, Address);
```

```
336
337         // Anzahl der Zeilen und Spalten ermitteln
338         tuples = PQntuples(Result);
339         column = PQnfields(Result);
340
341         // Informationsdatei beschreiben
342         Info_File << "Erstellungsdatum: " << (string)now.Format( "%Y-%m-%d %H:%M:%S" )
                << endl;
343         Info_File << "Spalten: ";
344         for(i=0; i<column; i++)
345         {
346                 Info_File << PQfname(Result, i) << " ";
347         }
348         Info_File << endl;
349         Info_File << "Anzahl Messwerte: " << tuples << endl;
350         Info_File << "Startzeit: " << (string)T_Start.Format( "%Y-%m-%d %H:%M:%S" ) <<
                endl;
351         Info_File << "Endzeit: " << (string)T_End.Format( "%Y-%m-%d %H:%M:%S" ) << endl
                ;
352         Info_File << "Transponder Adresse: " << Address << endl;
353
354         // Datensätze in die Wertedatei schreiben.
355         for(i=0; i<tuples; i++)
356         {
357                 for(j=0; j<column;j++)
358                 {
359                         Value_File <<  PQgetvalue(Result, i, j);
360                         if(j<column-1) Value_File << ";";
361                 }
362                 Value_File << endl;
363         }
364         Value_File << endl;
365         Value_File.close();
366         Info_File.close();
367 }
```

A.5.7 StdAfx.h

Listing 28: StdAfx.h

```
1  // stdafx.h : Include-Datei für Standard-System-Include-Dateien,
2  //  oder projektspezifische Include-Dateien, die häufig benutzt, aber
3  //      in unregelmäßigen Abständen geändert werden.
4  //
5
6  #if !defined(AFX_STDAFX_H__D8AE169A_C87D_4DEC_B195_89FB3364ADA3__INCLUDED_)
7  #define AFX_STDAFX_H__D8AE169A_C87D_4DEC_B195_89FB3364ADA3__INCLUDED_
8
9  #if _MSC_VER > 1000
10 #pragma once
11 #endif // _MSC_VER > 1000
12
13 #define VC_EXTRALEAN            // Selten verwendete Teile der Windows-Header nicht
       einbinden
14
15 #include <afxwin.h>         // MFC-Kern- und -Standardkomponenten
16 #include <afxext.h>         // MFC-Erweiterungen
17 #include <afxdisp.h>        // MFC Automatisierungsklassen
```

```
18 #include <afxdtctl.h>           // MFC-Unterstützung für allgemeine Steuerelemente von
       Internet Explorer 4
19 #ifndef _AFX_NO_AFXCMN_SUPPORT
20 #include <afxcmn.h>                    // MFC-Unterstützung für gängige Windows-
       Steuerelemente
21 #endif // _AFX_NO_AFXCMN_SUPPORT
22
23 #include "ChartViewer.h"
24 #include "UART.h"
25 #include "Class_DB.h"
26 #include "engine.h"
27 #include "mat.h"
28 #include "MatlabEng.h"
29
30
31 #include <winbase.h> // sleep()
32 #include<iostream>
33 #include <stdio.h>
34 #include <windows.h>    // HANDLE, ...
35
36 //{{AFX_INSERT_LOCATION}}
37 // Microsoft Visual C++ fügt unmittelbar vor der vorhergehenden Zeile zusätzliche
       Deklarationen ein.
38
39 #endif // !defined(AFX_STDAFX_H__D8AE169A_C87D_4DEC_B195_89FB3364ADA3__INCLUDED_)
```

A.5.8 StdAfx.cpp

Listing 29: StdAfx.cpp

```
1 // stdafx.cpp : Quelltextdatei, die nur die Standard-Includes einbindet
2 //      ControlCenter.pch ist die vorcompilierte Header-Datei
3 //      stdafx.obj enthält die vorcompilierte Typinformation
4
5 #include "stdafx.h"
```

A.5.9 ControlCenter.h

Listing 30: ControlCenter.h

```
1  // ControlCenter.h : Haupt-Header-Datei für die Anwendung CONTROLCENTER
2  //
3
4  #if !defined(AFX_CONTROLCENTER_H__46BACAA9_11CC_4C47_9D08_1C7D86306553__INCLUDED_)
5  #define AFX_CONTROLCENTER_H__46BACAA9_11CC_4C47_9D08_1C7D86306553__INCLUDED_
6
7  #if _MSC_VER > 1000
8  #pragma once
9  #endif // _MSC_VER > 1000
10
11 #ifndef __AFXWIN_H__
12         #error include 'stdafx.h' before including this file for PCH
13 #endif
14
15 #include "resource.h"           // Hauptsymbole
```

```
16
17 /////////////////////////////////////////////////////////////////////////////
18 // CControlCenterApp:
19 // Siehe ControlCenter.cpp für die Implementierung dieser Klasse
20 //
21
22 class CControlCenterApp : public CWinApp
23 {
24 public:
25         CControlCenterApp();
26
27 // Überladungen
28         // Vom Klassenassistenten generierte Überladungen virtueller Funktionen
29         //{{AFX_VIRTUAL(CControlCenterApp)
30         public:
31         virtual BOOL InitInstance();
32         //}}AFX_VIRTUAL
33
34 // Implementierung
35
36         //{{AFX_MSG(CControlCenterApp)
37                 // HINWEIS - An dieser Stelle werden Member-Funktionen vom Klassen-
                        Assistenten eingefügt und entfernt.
38                 //    Innerhalb dieser generierten Quelltextabschnitte NICHTS VERÄNDERN
                        !
39         //}}AFX_MSG
40         DECLARE_MESSAGE_MAP()
41 };
42
43
44 /////////////////////////////////////////////////////////////////////////////
45
46 //{{AFX_INSERT_LOCATION}}
47 // Microsoft Visual C++ fügt unmittelbar vor der vorhergehenden Zeile zusätzliche
        Deklarationen ein.
48
49 #endif // !defined(AFX_CONTROLCENTER_H__46BACAA9_11CC_4C47_9D08_1C7D86306553__INCLUDED_
        )
```

A.5.10 ControlCenter.cpp

Listing 31: ControlCenter.cpp

```
1  // ControlCenter.cpp : Legt das Klassenverhalten für die Anwendung fest.
2  //
3
4  #include "stdafx.h"
5  #include "ControlCenter.h"
6  #include "ControlCenterDlg.h"
7
8  #ifdef _DEBUG
9  #define new DEBUG_NEW
10 #undef THIS_FILE
11 static char THIS_FILE[] = __FILE__;
12 #endif
13
14 /////////////////////////////////////////////////////////////////////////////
15 // CControlCenterApp
16
```

```
17 BEGIN_MESSAGE_MAP(CControlCenterApp, CWinApp)
18         //{{AFX_MSG_MAP(CControlCenterApp)
19                 // HINWEIS - Hier werden Mapping-Makros vom Klassen-Assistenten
                       eingefügt und entfernt.
20                 //    Innerhalb dieser generierten Quelltextabschnitte NICHTS VERÄNDERN
                       !
21         //}}AFX_MSG
22         ON_COMMAND(ID_HELP, CWinApp::OnHelp)
23 END_MESSAGE_MAP()
24
25 /////////////////////////////////////////////////////////////////////////////
26 // CControlCenterApp Konstruktion
27
28 CControlCenterApp::CControlCenterApp()
29 {
30         // ZU ERLEDIGEN: Hier Code zur Konstruktion einfügen
31         // Alle wichtigen Initialisierungen in InitInstance platzieren
32 }
33
34 /////////////////////////////////////////////////////////////////////////////
35 // Das einzige CControlCenterApp-Objekt
36
37 CControlCenterApp theApp;
38
39 /////////////////////////////////////////////////////////////////////////////
40 // CControlCenterApp Initialisierung
41
42 BOOL CControlCenterApp::InitInstance()
43 {
44         AfxEnableControlContainer();
45
46         // Standardinitialisierung
47         // Wenn Sie diese Funktionen nicht nutzen und die Größe Ihrer fertigen
48         //  ausführbaren Datei reduzieren wollen, sollten Sie die nachfolgenden
49         //  spezifischen Initialisierungsroutinen, die Sie nicht benötigen, entfernen.
50
51 #ifdef _AFXDLL
52         Enable3dControls();                     // Diese Funktion bei Verwendung von
                   MFC in gemeinsam genutzten DLLs aufrufen
53 #else
54         Enable3dControlsStatic();       // Diese Funktion bei statischen MFC-
                   Anbindungen aufrufen
55 #endif
56
57         CControlCenterDlg dlg;
58         m_pMainWnd = &dlg;
59         int nResponse = dlg.DoModal();
60         if (nResponse == IDOK)
61         {
62                 // ZU ERLEDIGEN: Fügen Sie hier Code ein, um ein Schließen des
63                 //  Dialogfelds über OK zu steuern
64         }
65         else if (nResponse == IDCANCEL)
66         {
67                 // ZU ERLEDIGEN: Fügen Sie hier Code ein, um ein Schließen des
68                 //  Dialogfelds über "Abbrechen" zu steuern
69         }
70
71         // Da das Dialogfeld geschlossen wurde, FALSE zurückliefern, so dass wir die
72         //  Anwendung verlassen, anstatt das Nachrichtensystem der Anwendung zu starten
                   .
```

```
73          return FALSE;
74  }
```

A.5.11 calcstarttemperature.m

Listing 32: calcstarttemperature.m

```
1   %Variablen für die Berechnung anlegen
2
3   t = 0:1:(60*100);      % 60 Minuten
4   c= 0.000541525;        % wird in der Schleife variiert
5   RT=20;                 % wird in der Schleife variiert
6   % Quadratischen Fehler mit großer Zahl initialisieren
7   Smalest_Error = 99999999999;
8
9
10  % zum einlesen von Temperaturdaten aus einer CSV-Datei
11  %data = zeros(1,11);
12  %data = textread('temperature_data.csv', '', 'delimiter', ',','emptyvalue', NaN);
13
14  % Temperaturdaten Kopieren
15  Temperature_Samples = zeros(1,10);
16  for i=1:10
17     Temperature_Samples(i) = data(i);
18  end;
19
20  % Startindex wird momentan fest auf 22 Minuten gelegt
21  %Start_Index = 60 * data(11);
22
23  % Indexe für die Temperaturproben berechnen
24  Start_Index = 22 * 60;
25  Index_Measurements = Start_Index:60:(Start_Index + 540);
26
27
28  % In den folgenden 3 Schleifen werden die 3 Parameter
29  % Starttemperatur, RT und c variiert und mit diesen Parametern
30  % ein Temperaturverlauf nach der e-Funktion berechnet.
31  % Der Temperaturverlauf wird mit den Temperaturproben verglichen
32  % und der quadratische Fehler berechnet. Für die Parameter bei denen
33  % der quadratische Fehler am geringsten ist wird der Temperaturverlauf
34  % und die Starttemperatur berechnet.
35  for j = 1:10
36      c = j/10000;
37      for n = 1:50
38          RT = n;
39          for i = 20:200
40              StartTemperature_switch = i;
41              Temperature = (StartTemperature_switch-RT)*exp(-c*t)+RT;
42              Error = 0;
43              for n=1:10
44                  Error = Error + power((Temperature(Start_Index + 60*n-1) -
                        Temperature_Samples(n)),2);
45              end;
46              if(Error < Smalest_Error)
47                  Smalest_Error = Error;
48                  Start_Temperature = i;
49                  RT_min = RT;
50                  c_min = c;
51              end;
```

```
52          end;
53      end;
54  end;
55
56
57  % Darstellung des berechneten Temperaturverlaufes und der Temperatur-
58  % proben.
59  Temperature = (Start_Temperature-RT_min)*exp(-c_min*t)+RT_min;
60  figure(1)
61  stem(Index_Measurements, Temperature_Samples);
62  hold on
63  plot(t, Temperature, Index_Measurements, Temperature_Samples);
64  hold off
65  axis([0 6000 0 200]);
66  Title_Text = sprintf('Calculating the Starttemperature (%d Grad Celsius)',
        Start_Temperature);
67  title(Title_Text)
68  xlabel('Time[s]');
69  ylabel('Temperature[C]');
```

A.5.12 Tabellen.sql

Listing 33: calcstarttemperature.m

```
1   -- Table: test.measurement
2   -- DROP TABLE test.measurement;
3   CREATE TABLE test.measurement
4   (
5     id integer NOT NULL,
6     voltage double precision,
7     temperature double precision,
8     date timestamp without time zone,
9     transponder_id integer NOT NULL,
10    CONSTRAINT id_measurement_primary PRIMARY KEY (id),
11    CONSTRAINT transponder_id_foreign FOREIGN KEY (transponder_id)
12        REFERENCES test.transponder (id) MATCH SIMPLE
13        ON UPDATE NO ACTION ON DELETE NO ACTION
14  )
15  WITH (OIDS=FALSE);
16  ALTER TABLE test.measurement OWNER TO postgres;
17
18
19  -- Table: test.transponder
20  -- DROP TABLE test.transponder;
21  CREATE TABLE test.transponder
22  (
23    id integer NOT NULL,
24    address integer,
25    "name" character varying(32),
26    CONSTRAINT id_transponder_primary PRIMARY KEY (id)
27  )
28  WITH (OIDS=FALSE);
29  ALTER TABLE test.transponder OWNER TO postgres;
```

Weitere Quellcodedateien, die für die PC-Applikation notwendig sind, aber nicht selbst geschrieben wurden, sind in der nachfolgenden Liste dargestellt und auf der beiliegenden

CD enthalten:

- MatlabEng.h
- MatlabEng.cpp
- ChartViewer.h
- ChartViewer.cpp

B Schaltpläne

B.1 Schaltplan Transponder

B.2 Schaltplan Reader

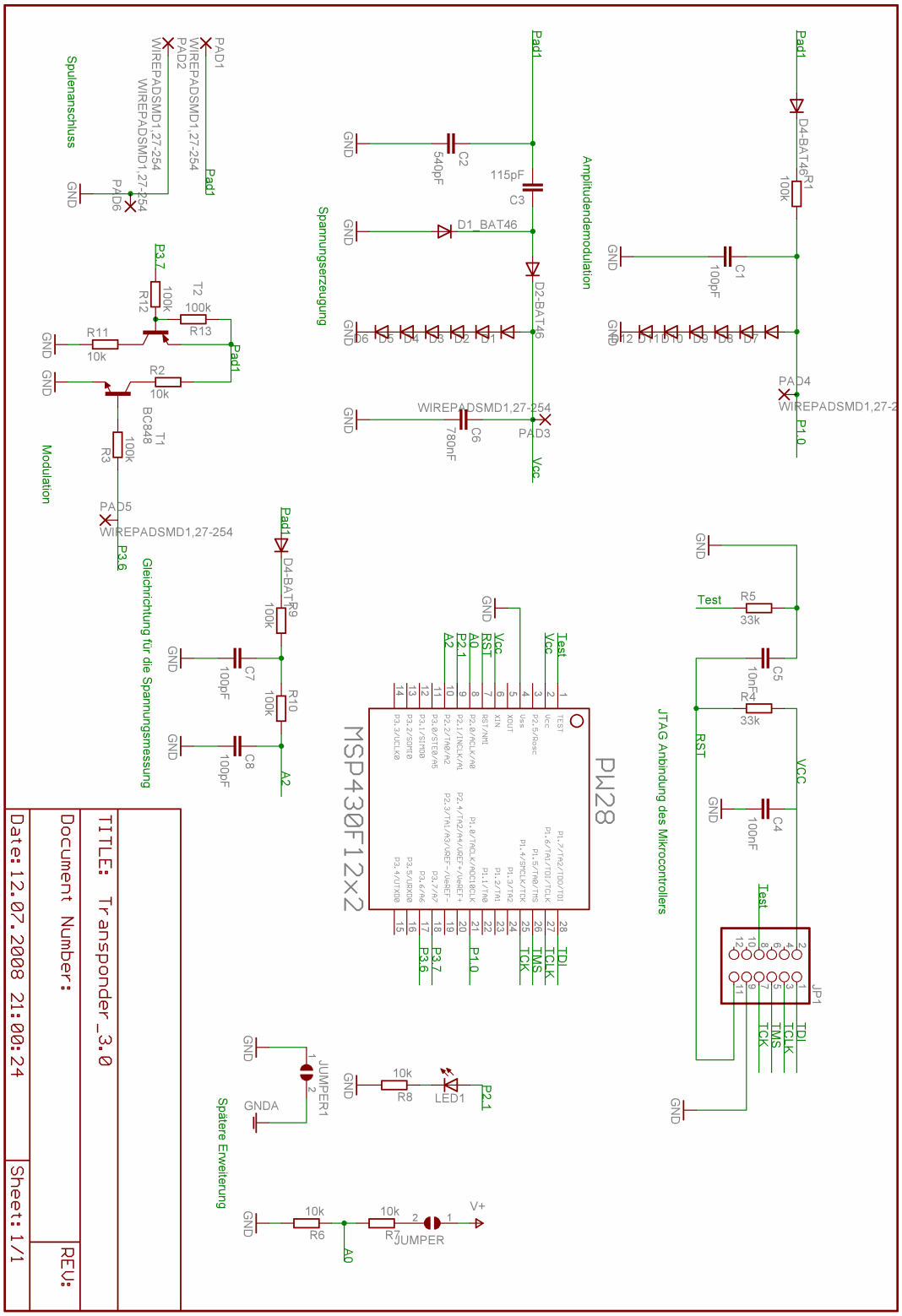

Bild 84: Schaltplan Transponder

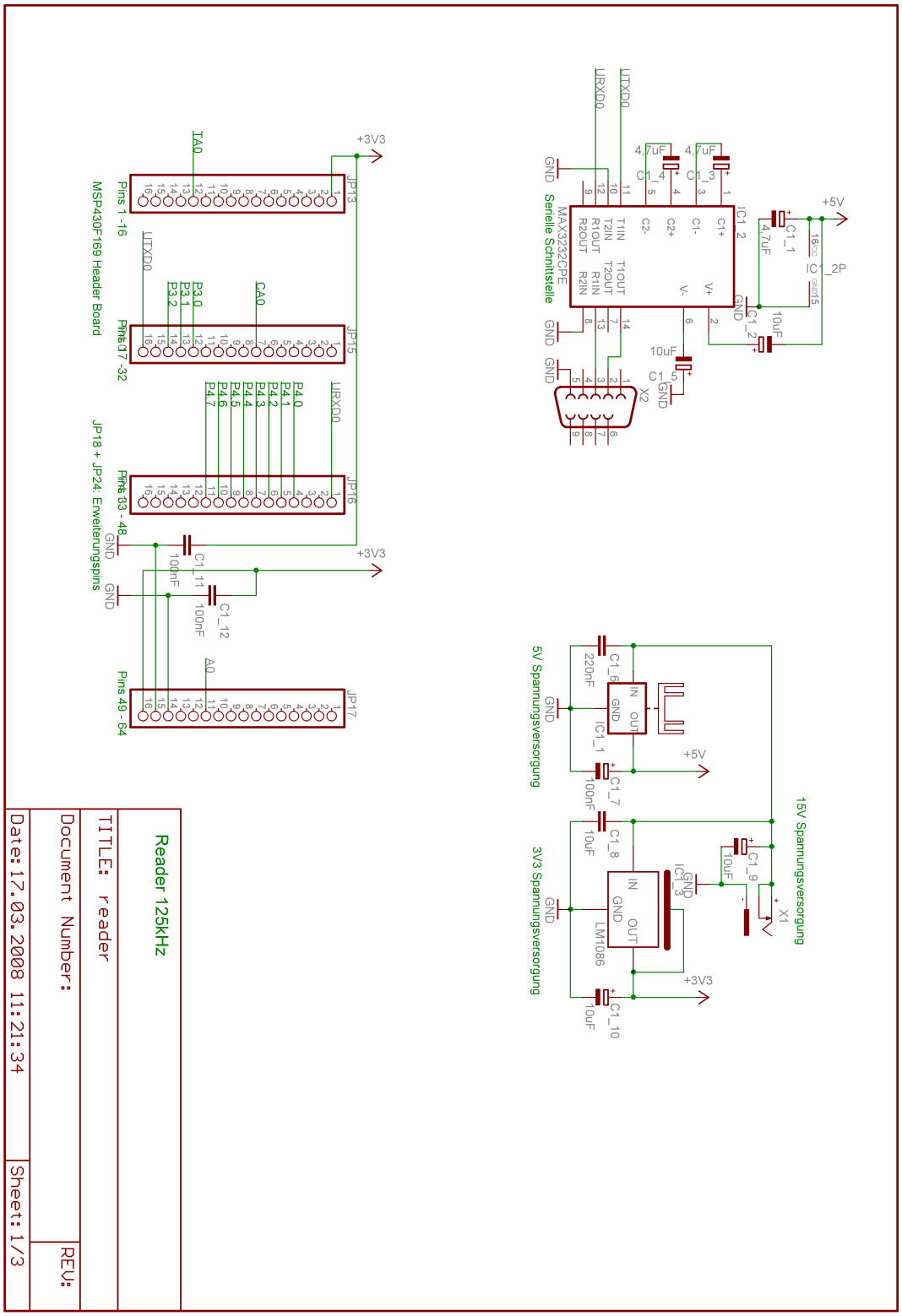

Bild 85: Schaltplan Reader 1

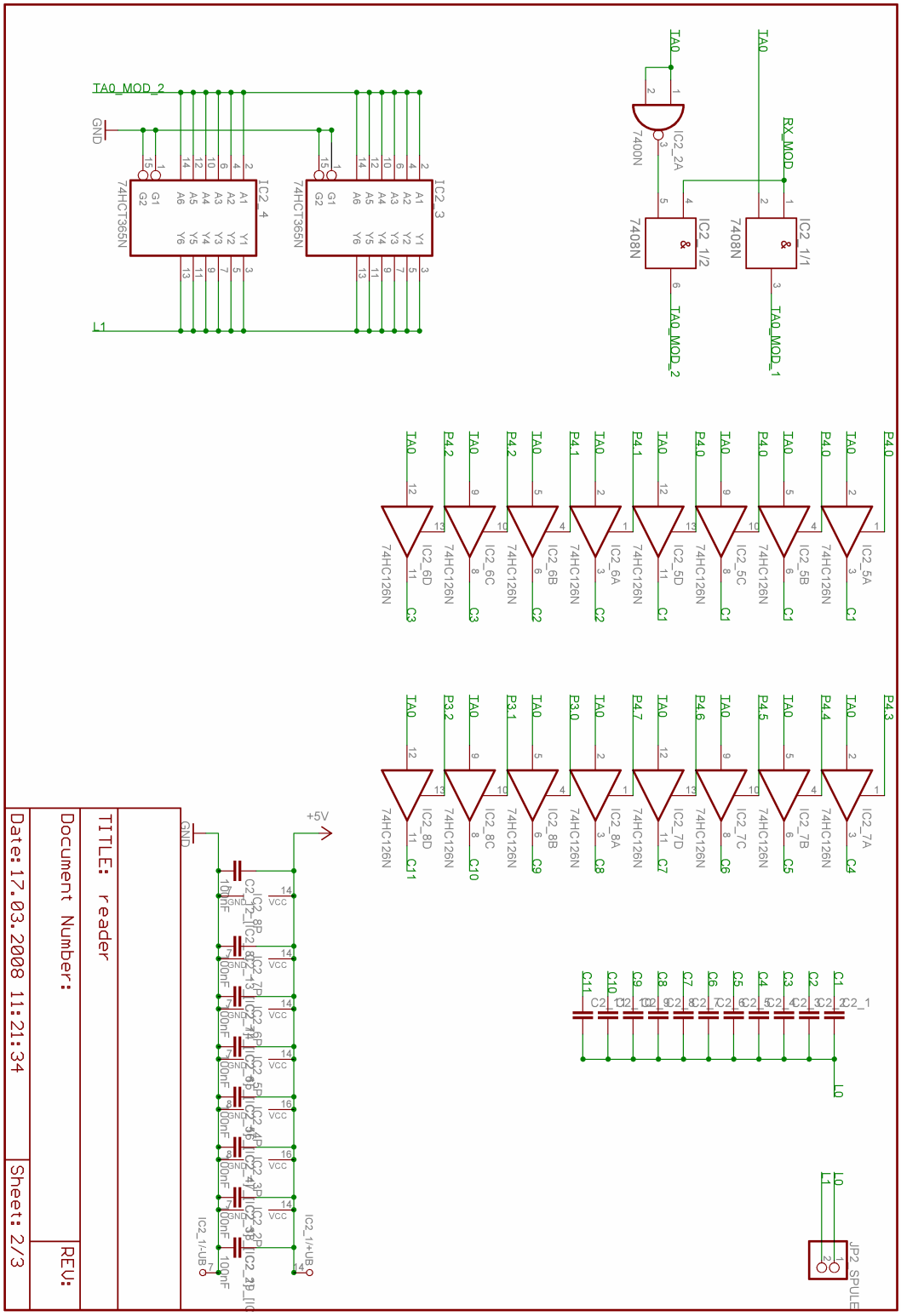

Bild 86: Schaltplan Reader 2

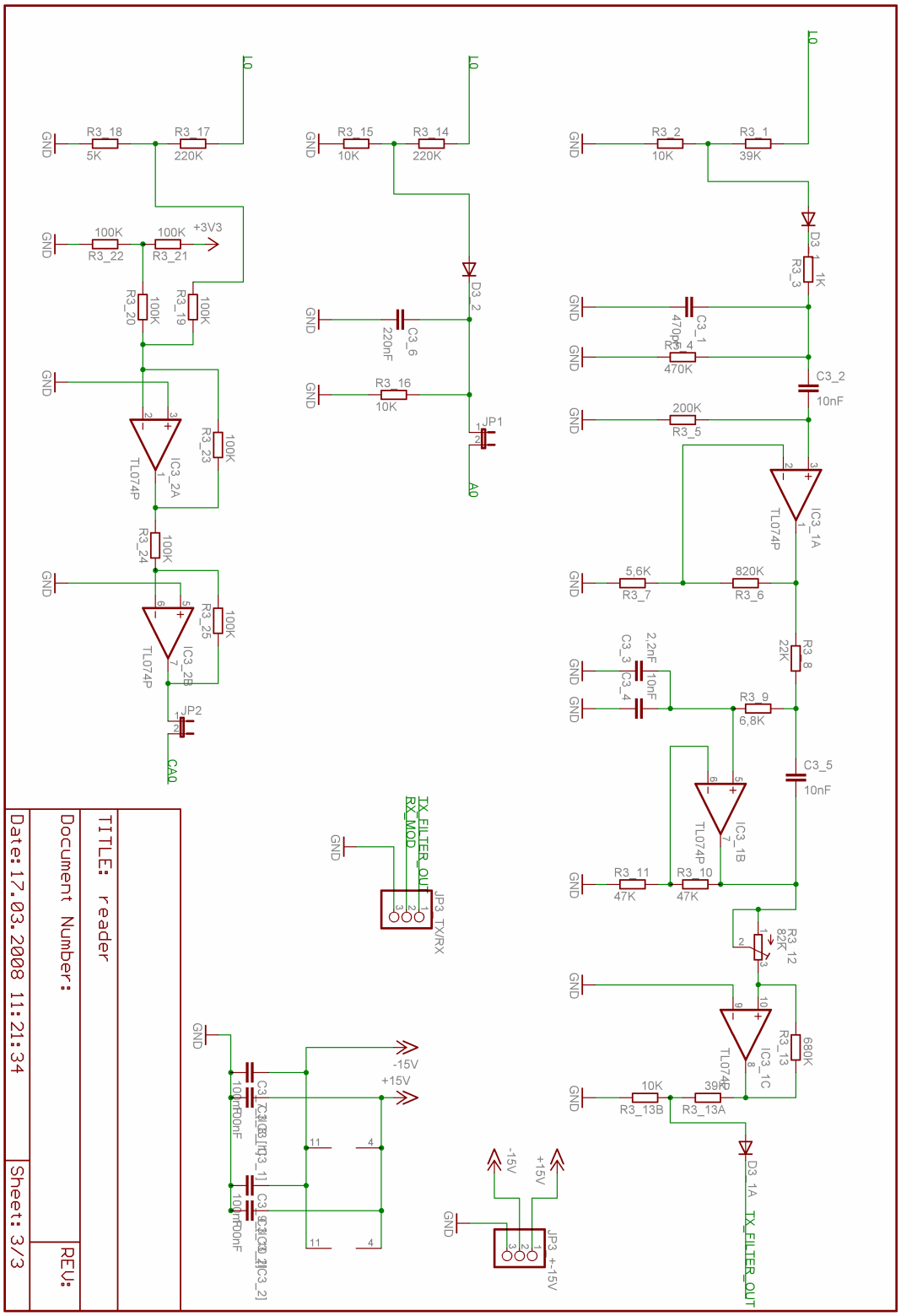

Bild 87: Schaltplan Reader 3

B.3 Olimex-MSP430F169-Board

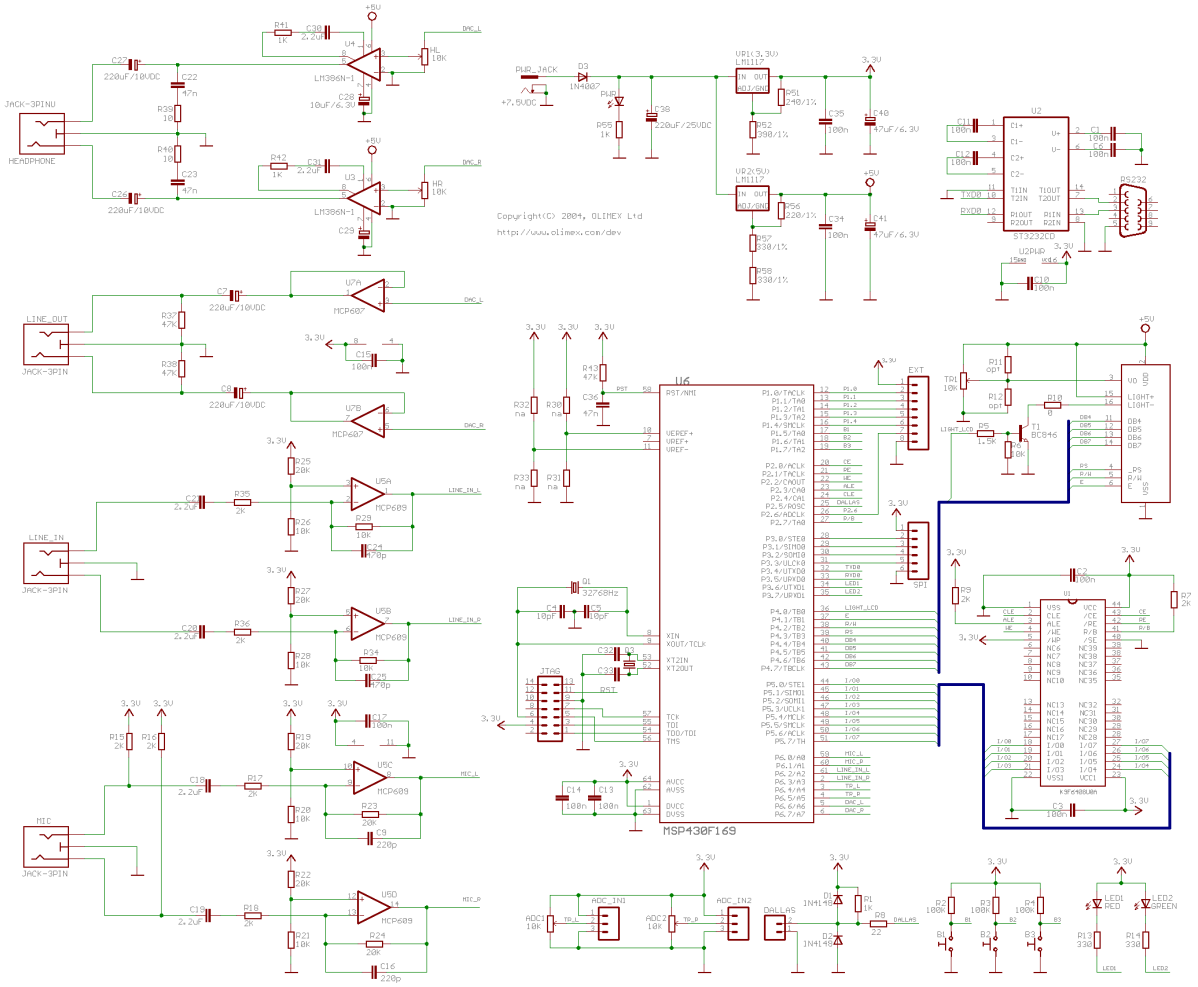

Bild 88: Olimex-MSP430F169-Board

VERSICHERUNG ÜBER DIE SELBSTSTÄNDIGKEIT

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

_____ _____

Ort, Datum Unterschrift