Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Tobias Steinmann

Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Tobias Steinmann

Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator

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

Betreuender Prüfer : Prof. Dr. -Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr. rer. nat. Jochen Schneider

Abgegeben am 24. August 2012

Tobias Steinmann

Thema der Bachelorthesis

Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator

Stichworte

ARM Mikrocontroller, Ethernet, SDRAM, Power-OP, Kalibrierung

Kurzzusammenfassung

Diese Arbeit beschreibt die Hard- und Software-Entwicklung für einen Zellspannungsgenerator. Mit dem Generator können aufgezeichnete und synthetisch erzeugte Spannungssequenzen wiedergegeben werden, um das Verhalten von Zellspannungssensoren, die der Überwachung von Batteriezellen dienen, zu untersuchen und zu optimieren. Zur Steuerung des Systems kommt ein ARM Cortex M3 Mikrocontroller zum Einsatz. Die Spannungssequenzen können aus Matlab per Ethernet an den Generator übertragen werden. Als weitere Betriebsart unterstützt der Generator einen hochgenauen Modus, in dem die genannten Sensoren kalibriert werden können.

Tobias Steinmann

Title of the paper

Hard- and software development of a networked Microprocessor controlled cell voltage generator

Keywords

ARM Mikrocontroller, Ethernet, SDRAM, Power Operational Amplifier, Calibration

Abstract

Inside this report the development of a generator to replay voltage sequences is described. One goal of the development is to analyse and optimize the performance of battery sensors. Theses sensors are used to monitor battery cells. To control the system, an ARM Cortex M3 microcontroller is used. The voltage sequences can be transfered to the generator via ethernet using Matlab. Another goal of the generator is the sensor calibration so high accuracy is required.

Inhaltsverzeichnis

1. Einführung

1.1. Allgemeines

Diese Bachelorthesis ist im Rahmen des Forschungsprojekts *Drahtlose Zellensensoren für Fahrzeugbatterien (BATSEN)* an der *Hochschule für Angewandte Wissenschaften Hamburg (HAW Hamburg)* entstanden. Hauptsächliche Ziele des Forschungsvorhabens sind es, mithilfe eines Sensornetzwerkes jede einzelne Zelle einer Fahrzeugbatterie zu überwachen und aus den gewonnenen Daten Kenntnisse über den genauen Ladezustand, den so genannten *State of Charge (SOC)*, sowie den Gesundheitszustand der Batterie, den so genannten *State of Health (SOH)* zu erhalten. Diese Erkenntnisse sind insbesondere in der Elektromobilität von großer Bedeutung, wenn es um die Abschätzung der noch zur Verfügung stehenden Reichweite oder um die frühzeitige Erkennung eines eintretenden Defektes einer einzelnen Batteriezelle geht.

Im Rahmen des Projekts BATSEN sind bereits mehrere Veröffentlichungen [15, 20, 33, 34, 32, 27] sowie Abschlussarbeiten [11, 35, 17, 14, 30, 21, 10, 19, 29] zu unterschiedlichen Teilbereichen entstanden. In den Arbeiten nach Plaschke [29], Ilgin [14] und Jegenhorst [17] wurden drahtlose Sensoren entwickelt, welche jeweils die Spannung einer einzelnen Batteriezelle messen und per Funk an eine Empfangseinheit senden. Die Erfassung der Messwerte erfolgt kontinuierlich, dabei wird jedoch die Abtastrate dynamisch an das Lastverhalten des von der Batterie gespeisten Verbrauchers angepasst. Damit ist gewährleistet, dass schnelle Hochstromereignisse, wie der Startvorgang eines Kraftfahrzeugs, detaillierter erfasst werden als Ruhephasen, wie beispielsweise der Zeitraum zwischen dem Abstellen des Fahrzeugs und dem nächsten Startvorgang.

1.2. Sensortypen

Bei den bisher entwickelten Sensoren gibt es zwei unterschiedliche Ansätze. Die erste Generation von Sensoren nach Ilgin [14] wird als *Klasse 1* bezeichnet und besitzt nur einen Uplink-Kanal im 433 MHz *Industrial, Scientific and Medical Band (ISM-Band)*. Sobald die Sensoren an eine Batteriezelle angeschlossen werden, beginnen diese mit der Erfassung und Aussendung der Messwerte. Die Aussendung erfolgt prinzipiell zyklisch, jedoch mit

einer Zufallsverzögerung versehen, damit beim Einsatz mehrerer Sensoren möglichst wenig Überlagerungen auf dem Funkkanal entstehen. Die Messwerte paralleler Aussendungen würden ansonsten verloren gehen, eine weitere Koordinierung des Sendeverhaltens ist nicht möglich.

In einem zweiten Ansatz nach Jegenhorst [17] wurden neue, als *Klasse 2* bezeichnete Sensoren entwickelt, welche zusätzlich einen Downlink-Kanal besitzen, der per *Radio Frequency Identification (RFID)* realisiert wurde. Der Vorteil dieser Lösung mit passivem Empfänger ist, dass die Sensoren die zwischengespeicherten Messwerte auf Anfrage senden können, wodurch keine Werte durch Überlagerung von zwei sendenden Sensoren verloren gehen können. Nachteilig ist jedoch die geringe Bandbreite des Downlink-Kanals und der hohe Hardware-Aufwand, welcher durch die beiden unterschiedlichen verwendeten Funktechniken entsteht.

Aktuell findet die Entwicklung eines neuen Sensortyps statt, welcher die Vorteile der Klasse 1 und 2 Sensoren kombinieren soll und dessen Uplink und Downlink im ISM-Band arbeiten soll. Dieser Typ wird als Klasse 3 bezeichnet.

1.3. Motivation

Besonders in den Phasen von schnellen Lastwechseln und den damit verbundenen Spannungseinbrüchen werden viele Messwerte erfasst, die zeitnah an die Empfangseinheit gesendet werden müssen. Hier ist ein effektiver Umgang mit der Bandbreite des Funkkanals gefragt, da alle Sensoren auf der gleichen Frequenz im 433 MHz *ISM-Band* senden. An dieser Stelle tritt das Problem auf, dass bisherige Praxistests mit den Sensoren nur am realen Kraftfahrzeug durchgeführt werden konnten. Allerdings ist ein Startvorgang und die damit verbundenen Spannungseinbrüche an einer realen Batterie nicht hundertprozentig reproduzierbar, wodurch die Analyse sowie Optimierung des Sensorverhaltens erschwert wird.

Um das genannte Problem zu umgehen, wurde in der Arbeit nach Schwartau [35] ein erster Prototyp eines Zellspannungsgenerators zur reproduzierbaren Wiedergabe von aufgezeichneten oder synthetisch erzeugten Spannungssequenzen entwickelt, welcher nachfolgend als Zellspannungsgenerator V01 bezeichnet wird. Dieser Prototyp zeigt die generelle Funktionsweise, wirft aber verschiedene Probleme und Einschränkungen auf, welche im Rahmen dieser Arbeit aufgegriffen werden.

1.4. Ziele

Nachfolgende Hauptziele dieser Abschlussarbeit sind sinngemäß aus der Aufgabenstellung (Anhang A) entnommen, Details sind dort dargestellt. Ziel dieser Abschlussarbeit ist die Weiterentwicklung eines modularen Systems von Zellspannungsgeneratoren. Dabei soll das Konzept vom Zellspannungsgenerator V01 untersucht und Ansatzpunkte zur Fortführung aufgegriffen werden. Zum einen soll eine schnelle Transienten-Wiedergabe möglich sein, zum anderen wird eine hochgenaue Betriebsart zur Kalibrierung der Sensoren benötigt.

Weitere Teilziele in Bezug auf die Hardware:

- Realisierung einer störungsarmen Spannungsversorgung
- Minimierung des Einflusses der Step-Up/Down Regler der Sensoren auf den Generator
- Integration des Mikrocontrollers in das Redesign ohne separates Evaluierungsboard
- Anbindung von externem RAM an den Mikrocontroller
- Aufbau eines Komplettgerätes mit Rack, Vernetzung, Stromversorgung, Frontplatten, Kühlung auf Basis von 19-Zoll-Gehäusen

Weitere Teilziele in Bezug auf die Software:

- Umstellung und Beschleunigung der Übertragung von Spannungswerten von ASCII-Kodierung auf binäre Übertragung (inkl. Prüfsumme)
- Implementierung der Strommessung und der Aufzeichnung in der Controller-Software
- Einbeziehung des Temperatursensors in die Controller-Software
- Entwicklung eines Telnet/Command-Interfaces mittels Command-Table und Funktionszeigern

Die genannten Ziele sollen durch ein komplettes Redesign von Hard- und Software umgesetzt werden.

2. Analyse und Voruntersuchungen

Zunächst wurde das Konzept vom Zellspannungsgenerator V01 untersucht. Weiterhin wurden Analysen in Verbindung mit den Zellspannungssensoren durchgeführt sowie ein Testaufbau für das anstehende Redesign errichtet.

Für nachfolgende Messungen standen ein Oszilloskop vom Typ MSO3034 der Firma Tektronix [40] sowie ein Tischmultimeter vom Typ PM2519 der Firma Philips [28] zur Verfügung. Die Messgenauigkeiten von Tischmultimeter und Oszilloskop wurden bereits in [35, Abschn. 6.2/6.3.3] untersucht. Es wurden je nach Messbereich Abweichungen bis zu 20 mV zwischen Oszilloskop und Tischmultimeter festgestellt. Weil jedoch das Tischmultimeter aufgrund seiner höheren Genauigkeit als Referenz bei der Kalibrierung dient, ist in den folgenden aus den Oszilloskopdaten generierten Plots ein Offsetfehler vorhanden. Dieser hat jedoch keinen Einfluss auf die durchgeführten Bewertungen, etwa ob Ripple oder Noise in dem jeweiligen Signal vorhanden sind. Weiterhin wurden unterschiedliche Amplituden von steilflankigen Signalen bei verschiedenen Zeitauflösungen beobachtet.

2.1. Sensorverhalten

Zur Verwendung der Sensoren mit unterschiedlichen Batterie-Technologien wurden diese für einen weiten Eingangsspannungsbereich ausgelegt. Zur Erzeugung einer konstanten Betriebsspannung für den auf dem Sensor befindlichen Controller sowie für den RF-Transmitter dient ein Step-Up/Down Converter, welcher sich aus der Zellspannung versorgt.

Auf den Sensoren der Klasse 1 befindet sich ein Step-Up Converter vom Typ L6920 der Firma STMicroelectronics [38] mit einem Eingangsspannungsbereich von 0,6 V bis 5,5 V, welcher ab 1 V anläuft. Die Beschaltung des Converters ist in Abbildung 2.1 dargestellt. Die Sensoren der Klasse 2 nutzen einen Step-Up/Down Converter vom Typ TPS61201 der Firma Texas Instruments [42], welcher mit einem Eingangsspannungsbereich von 0,3 V bis 5,5 V arbeitet und ab 0,5 V anläuft.

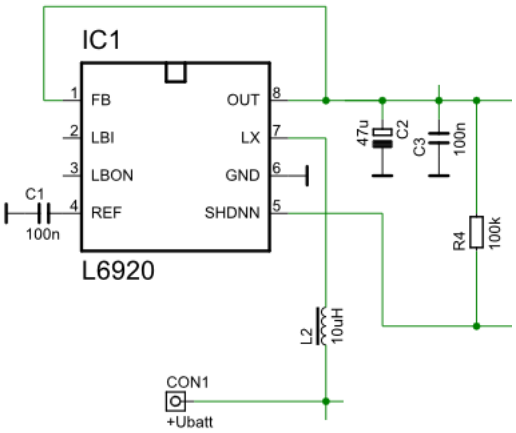


Abbildung 2.1.: Beschaltung des Step-Up Converters L6920 eines Klasse 1 Sensors nach [29, Kap. 2.1]

Da beide Schaltregler zur Erhöhung der Ausgangsspannung mit einer Spule arbeiten, entstehen beim Ein- und Ausschalten der Induktivität auf der Eingangsseite des Schaltreglers und damit an den Anschlussklemmen der Sensoren Spannungsspitzen, welche in die Zellspannung bzw. in den Zellspannungsgenerator einkoppeln können. Weil dies ebenfalls die vom Analog Digital Converter (ADC) des Sensors gemessene Spannung beeinflusst, schalten die Sensoren vor der Werteerfassung den Schaltregler ab und werden währenddessen über einen Kondensator gespeist.

Nachfolgend wurde die Einkopplung der Sensoren in die Zellspannung einer Bleibatterie, in den Zellspannungsgenerator V01 sowie in den in Unterkapitel 2.4 beschriebenen Testaufbau für das Redesign untersucht. Die Messungen wurden mit den Sensoren der Klasse 1 durchgeführt, da diese auch im Wesentlichen bei Tests und Experimenten des BATSEN-Projektes eingesetzt werden. Auf Tests mit den Klasse 2 Sensoren wurde deshalb verzichtet, auch weil sich die Inbetriebnahme dieser Sensoren aufgrund des RFID-Downlinks, welcher überhaupt erst die Aktivierung des Sensors ermöglicht, als sehr aufwendig gestaltet.

Von den Klasse 1 Sensoren existieren weiterhin verschiedene Bestückungsvarianten, die etwa mit verschiedenen Spulen unterschiedlicher Güte für die Spannungserhöhung des Schaltreglers ausgerüstet sind. Bei Tests verschiedener Sensoren konnten auch bei anscheinend gleicher Bestückung sowie identischer Softwareversion unterschiedlich starke Einkopplungen in die Zellspannung beobachtet werden. Weil die Messwerterfassung aber bei deaktiviertem Schaltregler erfolgt, wurde insbesondere dieses bei den Klasse 1 Sensoren etwa 90 μs lange Zeitfenster für Vergleiche herangezogen. Die Vergleiche wurden bei den 2 V Nennspannung einer Bleibatterie durchgeführt.

2.1.1. Einkopplung in Batterie

Abbildung 2.2 zeigt die Einkopplung des Klasse 1 Sensors in die Zelle einer 100 Ah Bleibatterie. Bemerkenswert sind die auftretenden Spannungsspitzen mit bis zu 40 mV Amplitude an einer so starken Quelle. Abbildung 2.3 zeigt den Spannungsverlauf bei hoher Zeitauflösung. Sobald das Shutdown-Signal des Schaltreglers auf Low gezogen wird, wird dieser deaktiviert. Es folgt ein etwa 15 μs langer Unter-/Überschwinger mit einer Amplitude von 55 mV, welcher vermutlich durch den Wegfall der Last hinter der Induktivität verursacht wird. Anschließend hat sich die Zellspannung stabilisiert und die Messwerterfassung durch den ADC des Sensor-Controllers kann erfolgen.

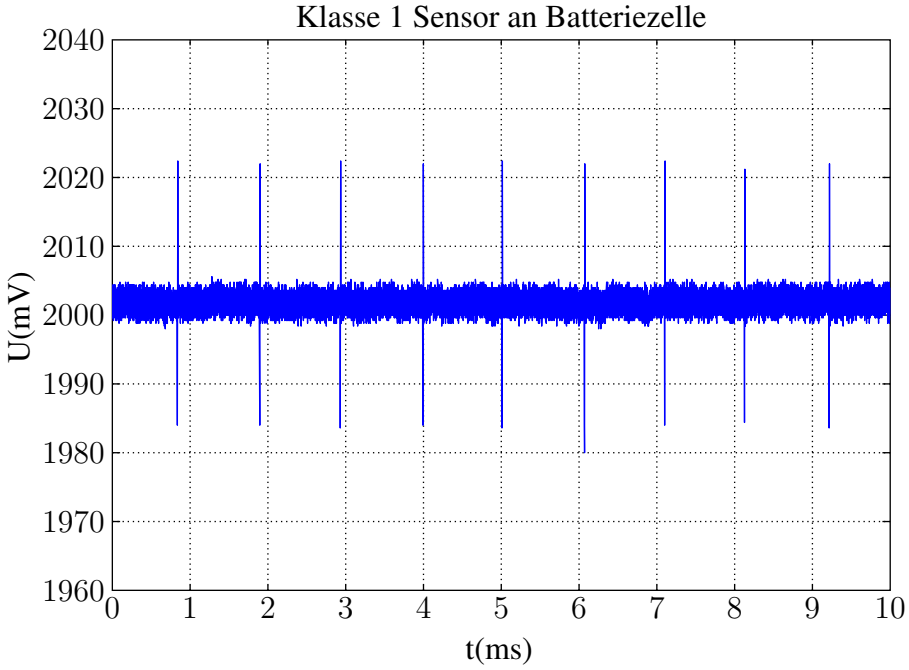


Abbildung 2.2.: Einkopplung in die Zellspannung einer Bleibatterie durch angeschlossenen Klasse 1 Sensor

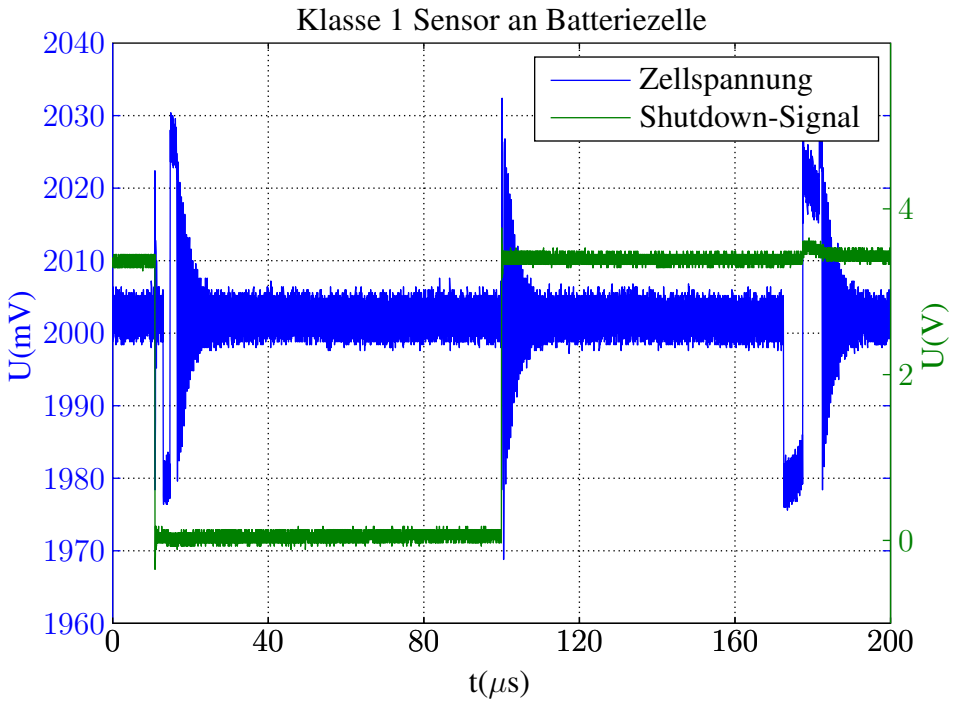


Abbildung 2.3.: Einkopplung eines Klasse 1 Sensors in die Zellspannung einer Bleibatterie (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals)

2.1.2. Einkopplung in Zellspannungsgenerator V01

Die Abbildungen 2.4 und 2.5 zeigen das Ausgangssignal vom Zellspannungsgenerator V01 bei einer eingestellten Spannung von 2 V. Die Spannung ist einerseits stark verrauscht, weiterhin treten die Spikes mit einer maximalen Amplitude von 80 mV auf. Die Unter-/Überschwinger nach Abschaltung des Schaltreglers in Abbildung 2.5 haben eine sehr hohe Amplitude größer 200 mV. Weiterhin dauert es mit etwa 45 μs dreimal so lange wie an der Bleibatterie, bis sich die Spannung wieder weitestgehend stabilisiert hat. Dennoch treten in dieser Ruhephase Spannungsspitzen mit bis zu 30 mV Amplitude auf, welche jedoch vermutlich durch den DC/DC-Konverter auf der Baugruppe und nicht durch den Sensor verursacht werden (siehe Abschnitt 2.2.2).

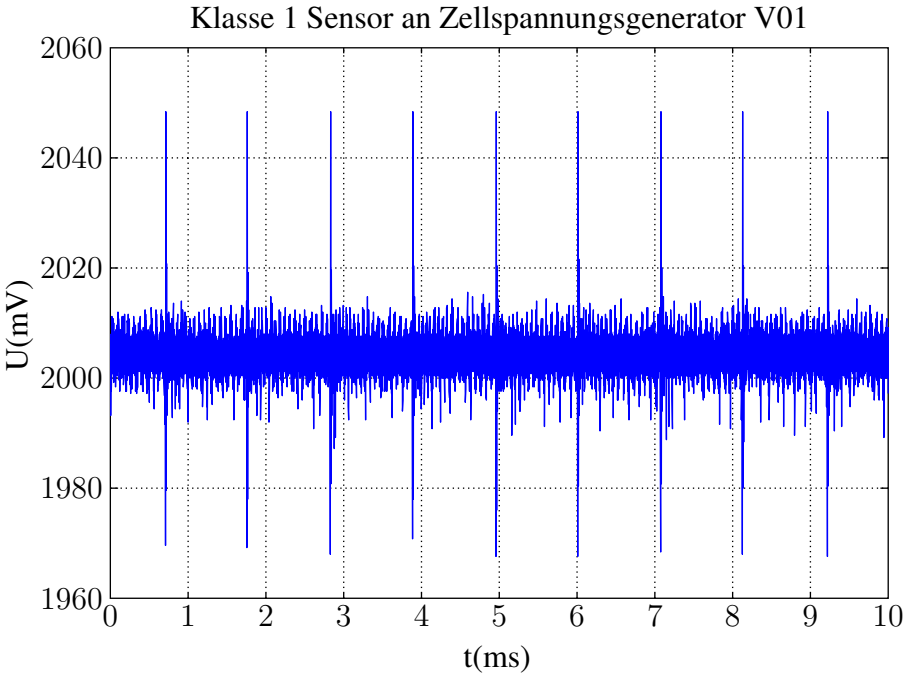


Abbildung 2.4.: Einkopplung in die Ausgangsspannung vom Zellspannungsgenerator V01 durch angeschlossenen Klasse 1 Sensor

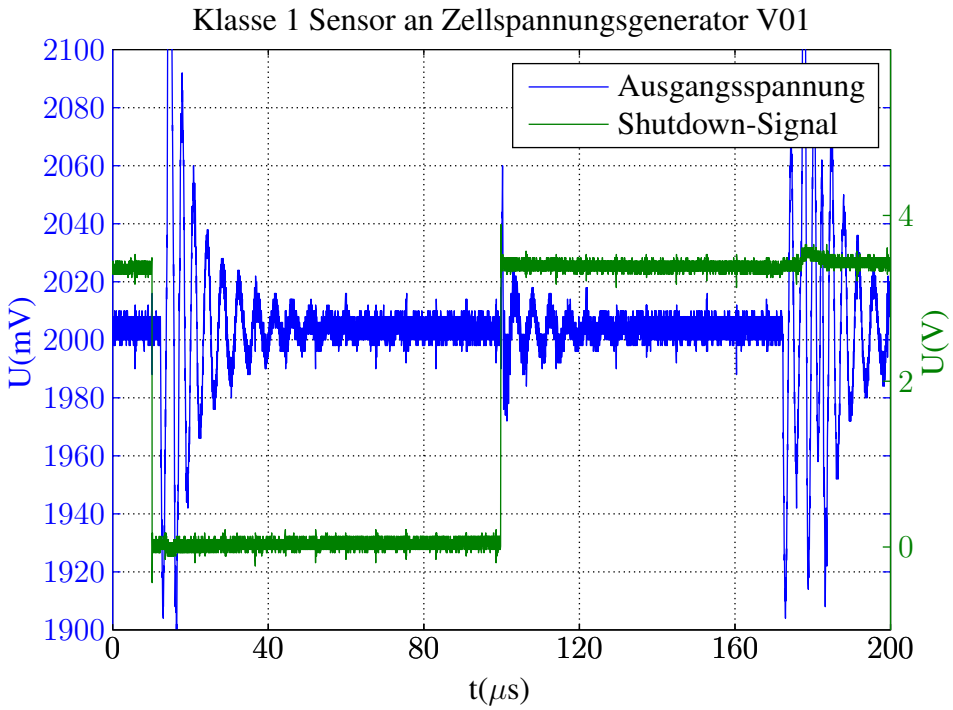


Abbildung 2.5.: Einkopplung eines Klasse 1 Sensors in die Ausgangsspannung vom Zellspannungsgenerator V01 (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals)

2.1.3. Einkopplung in Testaufbau

In den Abbildungen 2.6 und 2.7 ist das Ausgangssignal des in Unterkapitel 2.4 erläuterten Testaufbaus ersichtlich. Die zyklischen Einkopplungen haben eine maximale Amplitude von 40 mV. Damit liegen diese auf gleichem Level mit den Einkopplungen in die Batterie. Im Moment der Werteerfassung nach Abschaltung des Schaltreglers gibt es Unter-/Überschwinger mit bis zu 60 mV Amplitude, welche leicht über dem Level der Batterie liegen. Es dauert etwa 45 μs, bis sich die Spannung wieder in einem Noise Level kleiner 10 mV stabilisiert hat.

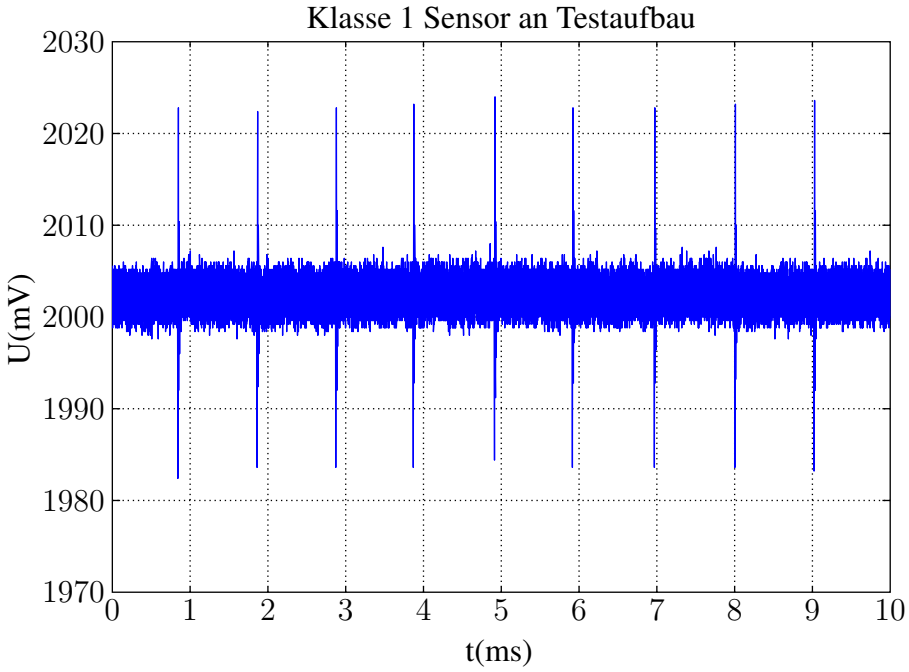


Abbildung 2.6.: Einkopplung in die Ausgangsspannung vom Testaufbau durch angeschlossenen Klasse 1 Sensor

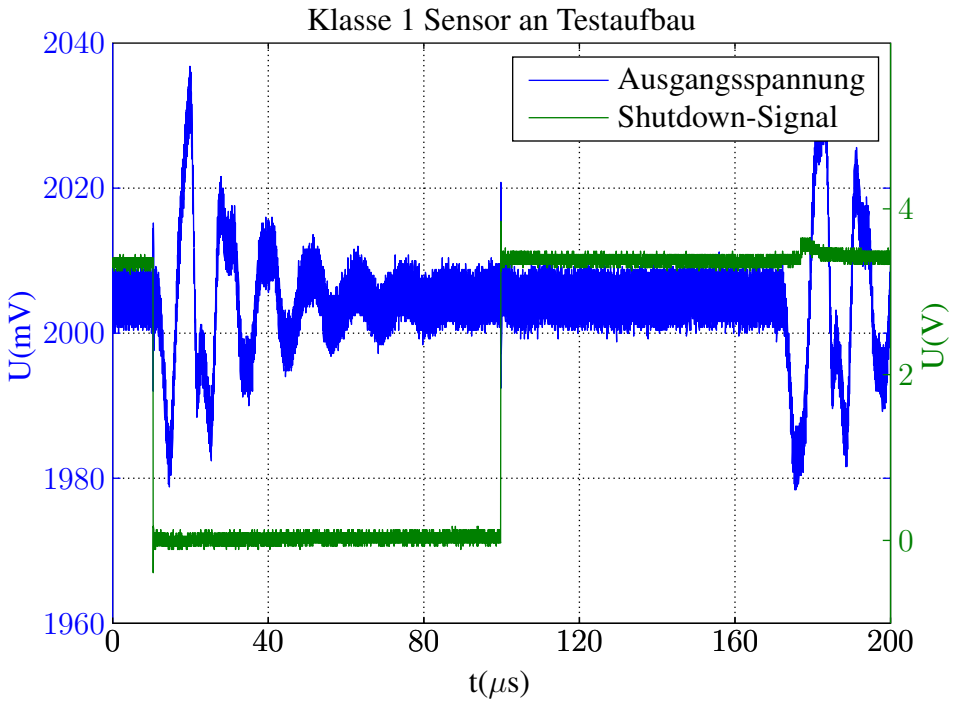


Abbildung 2.7.: Einkopplung eines Klasse 1 Sensors in die Ausgangsspannung vom Testaufbau (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals)

Die Messungen haben gezeigt, dass der Schaltregler auf den Sensoren selbst in die Zellspannung einer starken Bleibatterie einkoppeln kann. Umso wichtiger ist die Abschaltung des Schaltreglers, bevor die Zellspannung gemessen wird. Weiterhin sollte die Messwerterfassung durch den ADC im hinteren Teil des $90\,\mu s$ langen Zeitfensters erfolgen, nachdem die durch die Induktivität verursachten Unter-/Überschwinger abgeklungen sind. Die Amplituden der Einkopplungen während der Messwerterfassung liegen beim Testaufbau in Abbildung 2.7 um Faktor 3 bis 4 unter denen vom Zellspannungsgenerator V01 aus Abbildung 2.5. Weiterhin ist die Ruhespannung nach Abklingen der Unter-/Überschwinger beim Testaufbau mit keinen Spannungsspitzen belastet, wohingegen beim Zellspannungsgenerator V01 im Messzeitraum deutlicher Noise vorhanden ist. Für das finale Redesign sollte versucht werden, eine noch schnellere Stabilisierung der Ausgangsspannung nach Abschaltung des Schaltreglers zu erreichen.

2.2. Zellspannungsgenerator V01

Bei der Untersuchung des Konzeptes vom Zellspannungsgenerator V01 wurden Hardware, Software sowie die Bedienung des Systems beleuchtet. Ebenfalls wurden die offenen Punkte und Probleme nach [35, Kap. 7.2] aufgegriffen.

2.2.1. Modularer Aufbau und galvanische Trennung

Der Generator besteht aus einzelnen Modulen, welche jeweils im Eurokarten-Format als 19“ Einschub realisiert sind. Alle Module sind von der Hard- und Software her identisch, so dass sich diese sowohl separat, als auch im Verbund betreiben lassen.

Um die Nachbildung des Verhaltens einer Fahrzeugbatterie zu ermöglichen, ist eine Reihenschaltung mehrerer Modulausgänge notwendig. Diese erfordert eine galvanische Trennung aller sonstigen Verbindungen zwischen den einzelnen Modulen. Dazu zählen Spannungsversorgung, Ethernetanbindung sowie ein Synchronisationssignal zur samplegenauen synchronen Wiedergabe von Spannungssequenzen. Die galvanische Trennung der Spannungsversorgung ist durch einen DC/DC-Konverter gewährleistet, die Ethernetanbindung ist prinzipiell durch die Übertrager in einer Ethernetbuchse ebenfalls galvanisch getrennt. Das Synchronisationssignal wird über einen induktiven Koppler galvanisch getrennt, der aber eine zusätzliche gemeinsame 3,3 V Versorgung für alle Module erfordert.

Der beschriebene Aufbau ist prinzipiell sinnvoll, eine Änderung des mechanischen Aufbaus oder der Modulabmessungen ist nicht notwendig. Probleme bzw. Einschränkungen gibt es allerdings in Verbindung mit dem DC/DC-Konverter, der Ethernetbuchse sowie dem Synchronisationssignal. Diese werden in den Abschnitten 2.2.2, 2.2.4 und 2.2.7 im Detail betrachtet.

2.2.2. Störungen durch den DC/DC-Konverter

Zur galvanischen Trennung der Betriebsspannung wird auf den Modulen vom Zellspannungsgenerator V01 eingangsseitig ein DC/DC-Konverter vom Typ NMXS1215UC der Firma Murata [25] eingesetzt. Es wurden jedoch Störungen im Ausgangssignal festgestellt, welche durch den DC/DC-Konverter verursacht werden. Nachfolgend wurden Messungen an verschiedenen relevanten Punkten der Baugruppe durchgeführt, um das Problem zu verifizieren.

In Abbildung 2.8 wurde die Ausgangsspannung des DC/DC-Konverters gemessen. In diesem Fall musste für die Messung der 15 V Ausgangsspannung des Konverters zusätzlich ein Labornetzteil zur Generierung einer 5 V Offsetspannung in Reihe geschaltet werden, da das

verwendete Oszilloskop lediglich Spannungen bis 10 V mit einer Auflösung kleiner 1 V pro Division darstellen und aufzeichnen kann.

Die Abbildung 2.8 zeigt alle 8 μs eine Spannungsspitze mit bis zu 250 mV Amplitude. Aufgrund der notwendigen Offsetspannung für diese Messung lässt sich die Amplitude schwer qualitativ beurteilen.Die Spikes treten alle 125 kHz auf und damit zweimal pro Schaltzyklus des DC/DC-Konverters, dessen typische Schaltfrequenz bei 70 kHz liegt. Es lässt sich deshalb vermuten, dass die Spikes einmal beim Ein- und beim Ausschaltvorgang des Konverters entstehen.

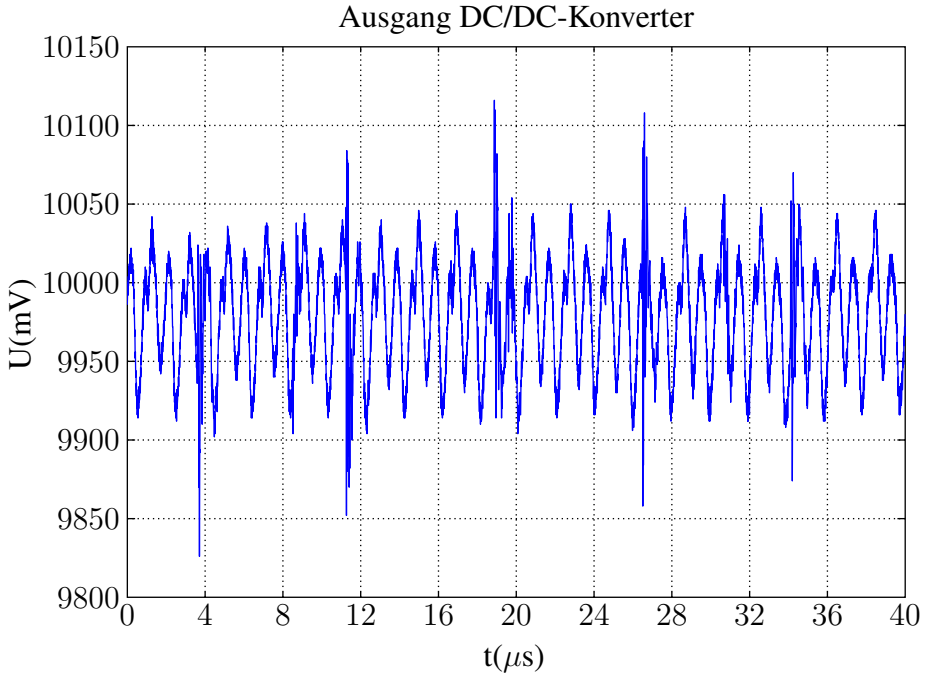


Abbildung 2.8.: Ausgangsspannung des DC/DC-Konverters mit 5 V Offsetspannung

In den Abbildungen 2.9 und 2.10 sind die Einkopplungen in die Ausgangsspannung des Linearreglers zur Versorgung der Analogkomponenten sowie in den Ausgang der Referenzspannungsquelle ersichtlich. Es zeigt sich, dass der Linearregler zur Ausregelung der extrem kurzen Spikes zu langsam ist. Die Spikes treten noch mit einer Amplitude von maximal 25 mV auf und sind damit um Faktor 10 abgeschwächt gegenüber der vorhergehenden Messung am Ausgang des DC/DC-Konverters.

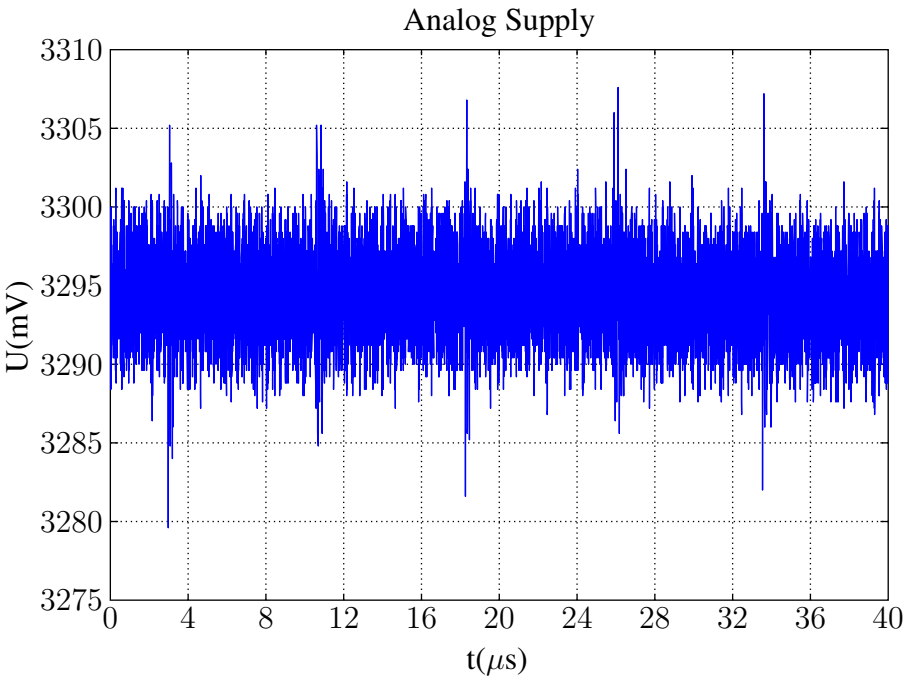


Abbildung 2.9.: Einkopplung des DC/DC-Konverters in die Ausgangsspannung des Linearreglers TPS76133 zur Analogversorgung

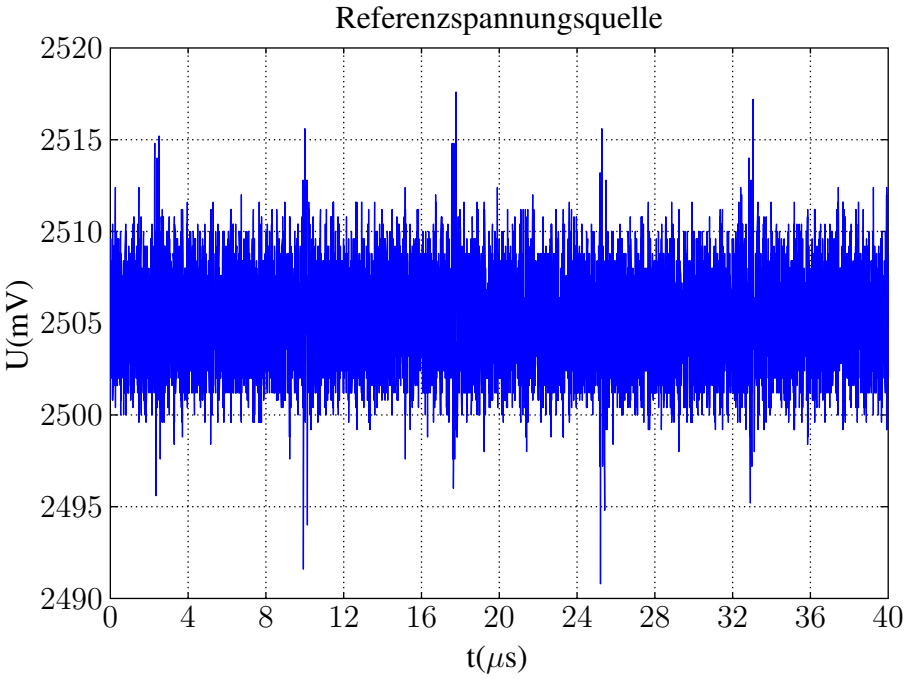


Abbildung 2.10.: Einkopplung des DC/DC-Konverters in die Ausgangsspannung der Referenzspannungsquelle REF5025

Die Abbildungen 2.11 und 2.12 zeigen das Ausgangssignal des Generators bei 2 V sowie bei 5 V Ausgangsspannung ohne eine angeschlossene Last. Es treten Spannungsspitzen mit Amplituden bis zu 38 mV auf. Insbesondere aufgrund der Störungen in der Referenzspannung sowie in der Analogversorgung wirken sich die Spannungsspitzen über den Digital Analog Converter (DAC) bis in den Leistungsteil und dessen Ausgangssignal aus.

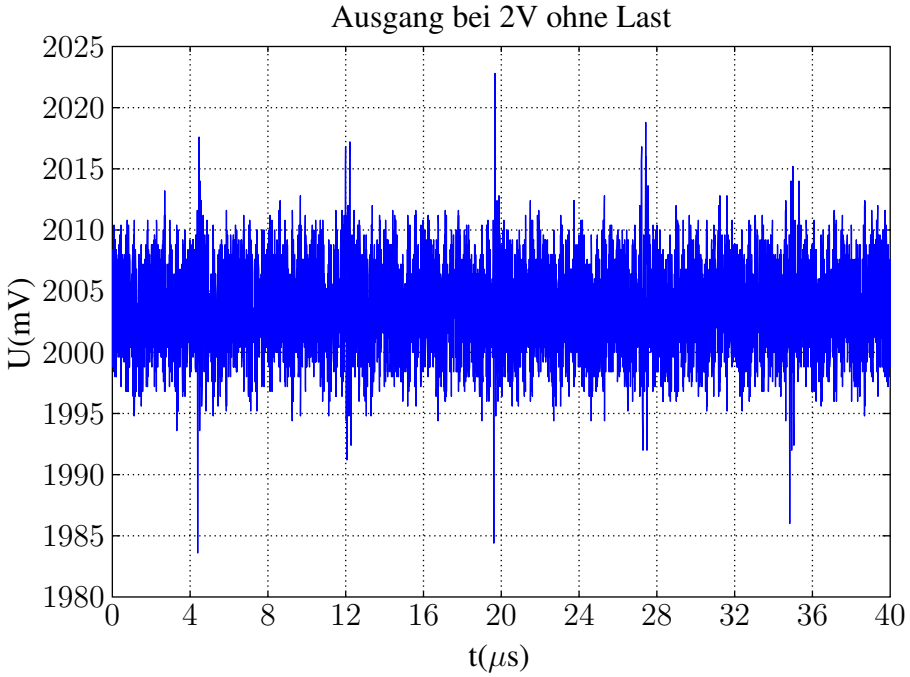


Abbildung 2.11.: Einkopplung des DC/DC-Konverters in eine Ausgangsspannung von 2 V ohne eine angeschlossene Last

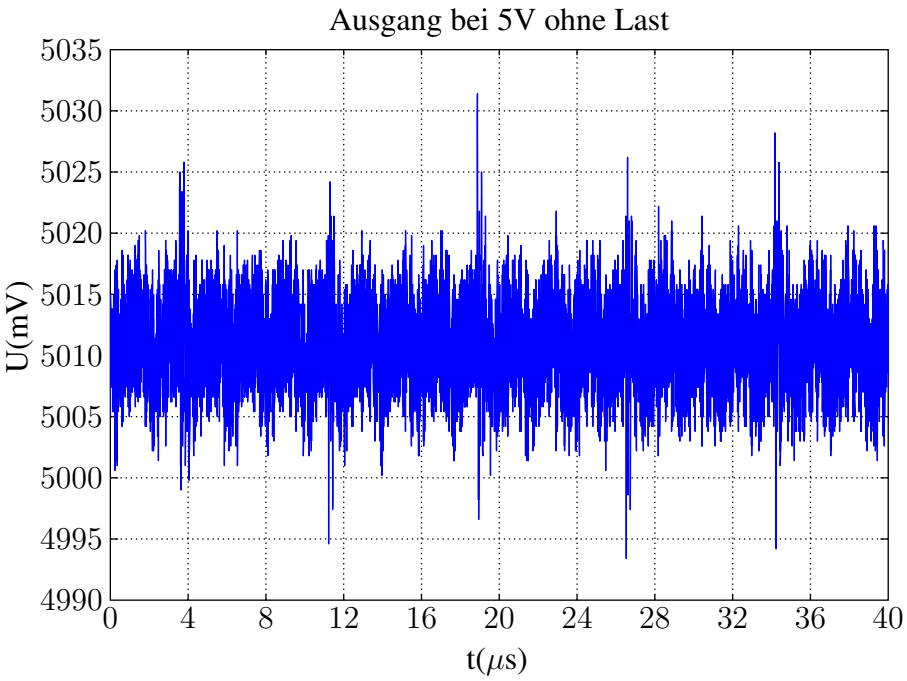


Abbildung 2.12.: Einkopplung des DC/DC-Konverters in eine Ausgangsspannung von 5 V ohne eine angeschlossene Last

2.2.3. Störungen durch den Sensor

Bei Speisung eines Sensors durch den Generator wurden Rückwirkungen des auf dem Sensor befindlichen Schaltreglers in das Generatorsignal festgestellt. Diese wurden in Abschnitt 2.1.2 untersucht.

2.2.4. Ethernetverbindung

Die Verwendung von Ethernet als Schnittstelle zur Übertragung von Spannungssequenzen sowie zur Steuerung der Baugruppen wurde in [35, Kap. 2.5] bereits untersucht und erscheint aus folgenden Gründen als sinnvoll:

- Hohe Übertragungsraten bis zu 100 MBit/s
- Weite Verbreitung der Schnittstelle im PC-Bereich
- Ethernet MAC und PHY in Mikrocontroller vorhanden
- Galvanische Trennung durch Übertrager in Ethernetbuchse

Der Vorteil der galvanischen Trennung durch die sich in der Ethernetbuchse befindlichen Übertrager wird beim Zellspannungsgenerator V01 durch das eingesetzte Stellaris Evaluations-Board wieder aufgehoben, da hier die Abschirmung der Buchse fest mit dem Ground der 4-lagigen Platine verbunden ist. Bei Verwendung von geschirmten Ethernet-Kabeln wäre die galvanische Trennung der Module aufgehoben. Verschiedene Versuche, die GND-Verbindung zu trennen, waren nicht erfolgreich, da die Lötfähnchen der GND-Verbindung gleichzeitig der Befestigung der Buchse dienen.

2.2.5. Übertragung von Spannungssequenzen

Der Generator besitzt eine Samplerate von 20 kHz. Zur Wiedergabe einer 10-sekündigen Spannungssequenz müssen deshalb 200000 Werte aus Matlab per Ethernet an den Mikrocontroller übertragen werden und in den per Serial Peripheral Interface (SPI) angebundenen Flash-Speicher geschrieben werden. Diese Übertragung dauert beim Zellspannungsgenerator V01 mit seiner Ursprungssoftware etwa 32 - 35 Sekunden. Die Spannungswerte wurden bisher als ASCII kodiert jeweils mit einem Trennzeichen übertragen, wodurch pro Wert 8 Byte erforderlich sind. Im Controller erfolgt dann eine Umwandlung ins Binärformat. Eine anschließende Umrechnung der Werte mithilfe der durch die Kalibrierung erzeugten Korrekturdaten ergibt dann die 16 Bit DAC-Werte, welche in den Flash-Speicher geschrieben werden.

Der im Zellspannungsgenerator V01 verwendete Flash Speicher vom Typ SST25VF032B der Firma SST [37] wird von der Controller-Software mit einem SPI-Takt von 1 MHz betrieben. Bei einer zu übertragenden Datenmenge von

$$200000 * 16\,\mathrm{Bit} = 3,2\,\mathrm{MBit}$$

würde alleine die serielle Übertragung unter idealen Umständen lediglich 3,2 s benötigen. Real kann der Flash-Speicher jedoch immer nur blockweise beschrieben werden, der Controller muss also immer erst 4 KByte empfangene Daten zwischenspeichern und diesen Block dann in den Flash schreiben, was wiederum etwas Zeit kostet. Dennoch fällt auf, dass anscheinend die Verarbeitung der Daten im Controller einen Großteil der Zeit in Anspruch nimmt.

Zunächst wurde die Übertragung von ASCII auf Binär umgestellt, um die Umwandlung jedes Wertes im Controller einzusparen. Weil mit dem Matlab TCP/IP-Stack keine Binärdaten gesendet werden können, wurden die 24 Bit Spannungswerte in Matlab zunächst ins Binärformat umgerechnet und dann jeweils mit 3 Byte in den TCP-Socket geschrieben. Weiterhin wurde angenommen, dass durch die Fehlerkorrektur im TCP-Protokoll alle gesendeten Bytes auch ohne Verfälschung oder Verluste ankommen. Deshalb wurde bei Umstellung der Übertragung auf das Trennzeichen nach jedem Wert verzichtet. Als zusätzliche Sicherheit wurde im Controller sowie im Matlab-Script noch eine Checksumme implementiert, welche

alle übertragenen Werte aufaddiert. Durch einen Vergleich der beiden errechneten Summen am Ende der Übertragung wird die Fehlerfreiheit sichergestellt.

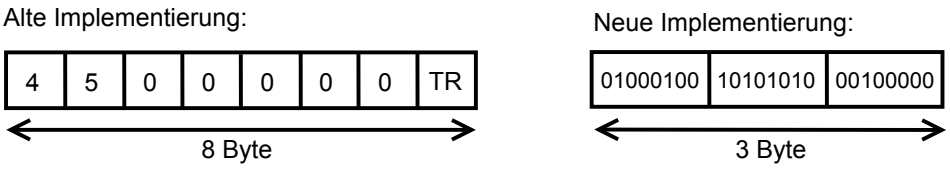


Abbildung 2.13.: Vergleich der alten und neuen Implementierung zur Übertragung eines Spannungswertes am Beispiel von 4500000 μV

Auch wenn nur ein 16 Bit DAC zum Einsatz kommt, fiel die Entscheidung, die in Microvolt vorliegenden Spannungswerte mit 24 Bit Genauigkeit zu übertragen. Dadurch ist gewährleistet, dass die Werte erst nach Einbeziehung der Kalibrierungsdaten im Controller auf 16 Bit gerundet werden und vorher keine Genauigkeit verloren geht.

Nach Umstellung der Übertragungsart sank die Übertragungszeit von 200000 Werten auf etwa 28 s. Weil die Optimierung nur wenige Sekunden Übertragungszeit einsparte, wurde die Verarbeitung der Werte zwischen Empfang und Speicherung in der Controller-Software genauer untersucht. Dabei stellte sich die Implementation des Empfangs-FIFOs als nicht so effizient heraus, weiterhin wurden die Werte mehrmals zwischengespeichert, bevor die 4 KByte-Datenblöcke erstellt und in den Flash-Speicher geschrieben wurden.

Durch Vereinfachung der Werteverarbeitung sowie durch eine effizientere FIFO-Implementierung konnte die Übertragungszeit von 200000 Werten mit 13 s mehr als halbiert werden. Als letzte Maßnahme wurde der SPI-Takt erhöht, da der Flash-Speicher in einem High-Speed Modus mit einem maximalen Takt von 80 MHz arbeiten kann und der Mikrocontroller einen maximalen SPI-Takt von 25 MHz generieren kann. Versuche mit Taktraten höher der bisherigen 1 MHz führten jedoch zu Datenverlusten und Fehlfunktionen des Speichers. Ein Grund dafür könnte die Signalführung über die Stiftleiste zwischen Evaluations-Board und Mainboard sein.

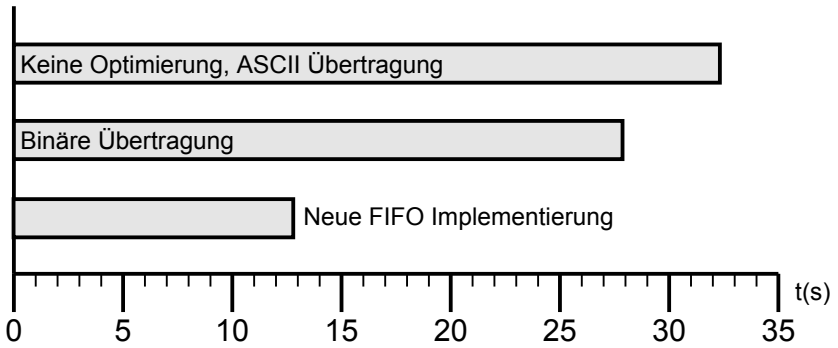


Abbildung 2.14.: Vergleich der Übertragungsdauer einer Spannungssequenz mit 200000 Werten bei den verschiedenen Implementierungen

2.2.6. Kalibrierung

Die Kalibrierung des Generators erfolgt nach [35, Abschn. 5.2.6] in zwei Schritten. Im ersten Schritt werden durch Ausgabe von zwei Spannungen der obere sowie der untere Punkt des benötigten Wertebereichs vom ADC (500 mV sowie 5,5 V) durch Abgleich mit einem externen Messgerät bestimmt, um daraus den Offset- und Verstärkungsfehler zu bestimmen. Im zweiten Schritt pegelt das System über die Rückführung automatisch verschiedene Ausgangsspannungen im benötigten Spannungsbereich ein und ermittelt dazu jeweils den 16 Bit DAC-Wert. Per linearer Interpolation können nun Zwischenwerte, die zwischen zwei kalibrierten Punkten liegen, bestimmt werden.

Diese Vorgehensweise ist bei Systemen mit Rückführung nicht unüblich und auch die erzielte Genauigkeit konnte bei Vergleichen der Soll-Spannung mit dem Tischmultimeter überzeugen. Verbesserungswürdig ist nur die Tatsache, dass die Kalibrierung der einzelnen Spannungsstufen im Interrupt-Handler des Ethernet-Stacks erfolgt. Die Zeitdauer für das Einstellen und Korrigieren der ausgegebenen Spannungen führt häufig dazu, dass es einen Timeout des Ethernet-Stacks gibt, der zum Abbruch der TCP-Verbindung führt.

2.2.7. Synchronisation

Die Synchronisation der einzelnen Module ist notwendig, um eine Spannungssequenz beim Einsatz mehrerer Module samplegenau wiederzugeben. Da Ethernet nicht echtzeitfähig ist, erfolgt die Synchronisation beim Zellspannungsgenerator V01 über ein einfaches Logiksignal. Dabei stellt ein als Master gejumpertes Modul den Ausgabetakt für die restlichen als Slave gejumperten Module zur Verfügung. Zur Sicherstellung der galvanischen Trennung

sitzt auf jedem Modul ein induktiver Koppler vom Typ ADUM1201 der Firma Analog Devies [7].

Die Lösung funktioniert mit wenigen Modulen, nachteilig sind jedoch die Hardware-Jumper auf den Modulen, denn bei falscher Jumperstellung (zwei Master) treiben zwei Ausgänge gegeneinander. Weiterhin besteht die Gefahr, dass das Synchronisationssignal bei größerer Ausbaustufe des Zellspannungsgenerators von bis zu 40 Modulen und damit längeren Leitungswegen anfällig für Einstreuungen wird. Weiterhin erfordert die galvanisch getrennte Seite vom ADUM1201 eine zusätzliche 3,3 V-Versorgung, welche extern für alle Module zur Verfügung gestellt werden muss.

2.2.8. Steuerung

Um die Steuerung der Generatoren über die TCP-Verbindung sowohl über einen aus Matlab geöffneten TCP-Socket, als auch über ein Terminal zu ermöglichen, wurden verschiedene ASCII-Kommandos implementiert, welche in einem größeren if-else-Konstrukt ausgewertet und verarbeitet werden. Die Wahl, hier ASCII-Befehle anstatt bspw. kryptischer Hexadezimal-Werte zu verwenden, ist sinnvoll, da der Generator so auch einfach über das Terminal bedient werden kann. Der geringfügige Overhead der ASCII-Übertragung ist hier im Gegensatz zur Übertragung von Spannungssequenzen zu vernachlässigen.

Bei Tests mit dem Zellspannungsgenerator V01 fiel auf, dass die Controller-Software bei aktiver Terminal-Verbindung manchmal nach längerer Zeit einfriert. Möglicherweise hängt dies mit der eingestellten Stack-Size des Controllers zusammen. Eine weitere Analyse des Problems wurde nicht durchgeführt, da die Controller-Software für das Redesign ohnehin neu implementiert werden muss.

2.3. Schaltregler vs. Linearregler

Ein wesentliches Ziel ist es, mit dem Redesign des Zellspannungsgenerators eine störungsarme, stabile Ausgangsspannung insbesondere für die Kalibrierung der Zellspannungssensoren erzeugen zu können. Aufgrund der in Abschnitt 2.2.2 genannten Störungen, welche beim Zellspannungsgenerator V01 durch den DC/DC-Konverter verursacht werden, wurde auf die Spannungsversorgung besonderes Augenmerk gelegt. Wichtig ist eine saubere Versorgung ohne Ripple und mit wenig Noise für alle analogen Komponenten des Systems. Dazu zählen DAC, ADC sowie die Operationsverstärker der Ausgangsstufe. Insbesondere die OPs, welche die Ausgangsspannung des DACs verstärken, würden auch die Störungen verstärken, was das Problem noch verschärft. Für die Versorgung der rein digitalen Komponenten spielen die in Abschnitt 2.2.2 genannten Störungen keine Rolle.

Bei dem im Zellspannungsgenerator V01 eingesetzten DC/DC-Konverter vom Typ NMXS1215UC der Firma Murata [25] handelt es sich um einen galvanisch getrennten Schaltregler, welcher zur Gruppe der Flyback-Converter gehört. Die prinzipielle Funktionsweise eines Flyback-Converters lässt sich nach [53] sowie nach [18, Kap. 3] in zwei Schritten beschreiben:

1. Ein periodisch arbeitender elektronischer Schalter ist zunächst geschlossen (Abbildung 2.15 links). Die Primärspule ist direkt mit der Eingangspannung verbunden und speichert Energie im Luftspalt des Transformators. Die Last auf der Sekundärseite wird durch den Kondensator gespeist.
2. Im zweiten Zyklus ist der elektronische Schalter geöffnet (Abbildung 2.15 rechts). Die im Transformator gespeicherte Energie lädt den Kondensator wieder auf und versorgt die Last.

Durch schlagartiges Öffnen und Schließen des elektronischen Schalters muss sich das Magnetfeld in der Primärspule sehr schnell auf- und abbauen, wodurch eine Spannungsspitze in die Sekundärspule und damit in die Ausgangsspannung des Converters induziert wird.

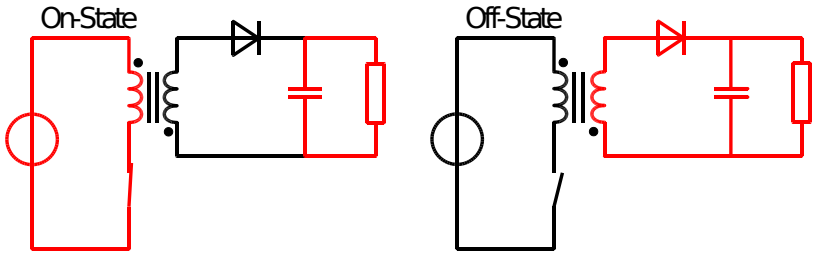


Abbildung 2.15.: Funktionsweise eines Flyback-Converters nach [53]. Im On-State wird Energie im Luftspalt des Transformators gespeichert, im Off-State lädt die dort gespeicherte Energie den Kondensator.

Der beim Zellspannungsgenerator V01 eingesetzte DC/DC-Konverter vom Typ NMXS1215UC der Firma Murata [25] arbeitet mit einer typischen Schaltfrequenz von 70 kHz, der elektronische Schalter wird also 70000 mal pro Sekunde geschlossen und wieder geöffnet, wodurch alle 8 μs eine Spannungsspitze entsteht. Die Amplitude der Störungen lässt sich durch einen Kondensator mit niedrigem Equivalent Series Resistance (ESR) am Ausgang des Converters minimieren, die Spikes lassen sich damit aber nicht komplett glätten.

Nach der Application Note 101 von Linear Technology [24] ist es gängige Praxis, für eine saubere Versorgungsspannung einem Schaltregler, der die Umsetzung von einem hohen auf ein niedrigeres Spannungspotenzial übernimmt, noch einen Linearregler zur Minimierung

der durch die Schaltregler verursachten Störungen nachzuschalten. Während der Linearregler den Ripple des Schaltreglers leicht ausregulieren kann, werden die extrem kurzen Spikes nur leicht gedämpft. Eine Filterung des Ripples ist nach der genannten Application Note [24] nicht unmöglich, aufgrund diverser parasitärer Elemente in den Bauteilen sowie im Platinenlayout aber sehr aufwendig und würde diverse Tests mit unterschiedlichen Layoutvarianten erfordern. Aus den genannten Gründen wurde beschlossen, für das Redesign des Zellspannungsgenerators ein Trafo-Netzteil mit Linearreglern zu verwenden, obwohl hier eine höhere Verlustleistung entsteht. In Tabelle 2.1 sind noch einmal die Vor- und Nachteile zwischen DC/DC-Konverter und Linear-Netzteil aufgelistet.

DC/DC-Konverter	**Linear-Netzteil**
hoher Wirkungsgrad ~80%	niedriger Wirkungsgrad der Linearregler bei großer Differenz von U_e zu U_a
geringe Verlustleistung/Abwärme	hohe Verlustleistung bei großer Differenz von U_e zu U_a
Ripple/Spikes am Ausgang	saubere Ausgangsspannung
aufwendige Filterung notwendig	geringer Filteraufwand
hohe Anforderungen an angepasstes Layout	relativ unabhängig vom Layout
gemeinsames Netzteil für alle Baugruppen	Netztrafo je Baugruppe erforderlich
teuer bei entpr. Ausgangsleistung	kostengünstig hohe Ausgangsleistung
Über-/Unterschwinger bei Laständerungen	sehr stabil/robust bei Lastschwankungen

Tabelle 2.1.: Vor- und Nachteile von DC/DC-Konverter und Linear-Netzteil

Die Entscheidung für ein Trafo-Netzteil mit Linearreglern für die verschiedenen Versorgungsschienen führt dazu, dass aufgrund der notwendigen galvanischen Trennung zwischen den einzelnen Modulen auch jede Baugruppe einen eigenen Netztrafo benötigt. Weiterhin entsteht bei der 3,3 V-Schiene zur Versorgung der digitalen Komponenten eine gewisse Verlustleistung, welche in Wärme abgegeben wird. Im Gegenzug entfällt eine insbesondere für den Leistungsteil sehr aufwendige Filterung von unerwünschten Spannungsspitzen. Desweiteren kann bei Verwendung eines Trafos kostengünstig eine höhere Ausgangsleistung zur Verfügung gestellt werden, während entsprechende DC/DC-Konverter sehr teuer sind.

2.4. Testaufbau

Bevor mit der Hardware-Planung für das Redesign begonnen wurde, wurde die prinzipielle Funktionalität sowie Stabilität von Netzteil und Leistungsverstärker untersucht. Dazu wurde gemäß Abbildung 2.16 ein Testaufbau bestehend aus folgenden Komponenten errichtet.

- Netztrafo
- 3,3 V und 8 V Linearregler
- Ausgangsstufe mit Power-OP
- Pufferung für Kalibriermodus

Zur Dimensionierung des Netzteils wurde zunächst die zu erwartende maximale Stromaufnahme abgeschätzt. Dazu wurde beim Zellspannungsgenerator V01 die Stromaufnahme der digitalen Versorgungsschiene gemessen, welche bei etwa 200 mA liegt. Weiterhin liegt die maximale Stromaufnahme der Sensoren nach [35, Abschn. 2.1.2] ebenfalls bei etwa 200 mA. Als Leistungsverstärker soll ebenfalls der OPA564 der Firma Texas Instruments [47] verwendet werden, der sich bei Voruntersuchungen nach [35, Kap. 3.2] sowie beim Einsatz vom Zellspannungsgenerator V01 bewährt hat.

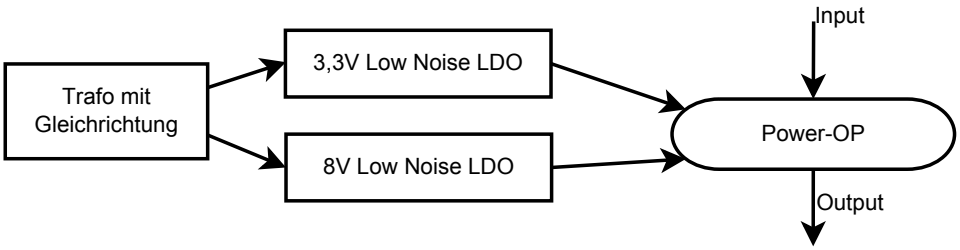


Abbildung 2.16.: Schematische Darstellung des Testaufbaus

Die Step-Up/Down Regler der Sensoren arbeiten, wie in Unterkapitel 2.1 beschrieben, mit einer maximalen Eingangsspannung von 5,5 V. Der Leistungs-Operationsverstärker OPA564 kann an einer unsymmetrischen Versorgung betrieben werden. Damit dieser die 5,5 V am Ausgang zur Verfügung stellen kann, benötigt dieser bei $I_{Out} = 0,5\,\mathrm{A}$ eine lediglich um 0,4 V höhere Versorgungsspannung. Um eine ausreichende Regelreserve auch bei Spitzenströmen zur Verfügung zu haben, soll der OPA564 mit 8 V für den Leistungsteil versorgt werden. Hier wurde ein Low-Noise *Low-dropout linear voltage regulator (LDO)* vom Typ LT1963A der Firma Linear Technology [23] vorgesehen, welcher laut Datenblatt speziell für schnelle Lastwechsel optimiert ist, welche bei Speisung eines Zellspannungssensors durch das zyklische Abschalten des Schaltreglers auftreten. Den LT1963A gibt es sowohl als Festspannungsregler für gängige Ausgangsspannungen von 1,5 - 3,3 V sowie in einer Variante für variable Ausgangsspannungen, welche hier zum Einsatz kommt. Nach der im Datenblatt [23] angegebenen Formel

$$V_{OUT} = 1,21V \cdot \left(1 + \frac{R2}{R1}\right) + I_{ADJ} \cdot R2$$

mit $I_{ADJ} = 3\,\mu\text{A}$ bei $25\,^{\circ}\text{C}$ wurden die beiden Widerstände zur Konfiguration von 8 V Ausgangsspannung mit $R1 = 2,2\,\text{k}\Omega$ und $R2 = 12,4\,\text{k}\Omega$ bestimmt. Es ist nicht erforderlich, exakt 8,0 V einzustellen, da das System ohnehin kalibriert werden muss.

Zusätzlich zu der Versorgungsspannung für den Leistungstreiber benötigt der OPA564 eine 3,3 V Versorgung für seine digitalen Funktionen. Dazu zählen die De-/Aktivierung des Ausgangs sowie ein *Over Temperature*- und *Current Limit*-Flag. Zur Versorgung der digitalen Komponenten wurde ein Low Noise LDO vom Typ LT1760 der Firma Linear Technology [22] vorgesehen.

Als Netztrafo kam zunächst ein 16 VA Printtrafo mit 12 V Nennspannung zum Einsatz. Dieser lieferte hinter den Glättungskondensatoren jedoch eine Leerlaufspannung von 19 V, was bei ersten Tests zu starker Abwärme der Linearregler führte. Daher wurde der Trafo später durch einen 10 VA Flachtrafo der Firma Myrra [26] mit einer Nennspannung von 9 V ersetzt. Dieser besitzt ebenfalls ausreichend Leistung zur Versorgung der Komponenten sowie des Sensors, es entsteht aber durch die niedrigere Nenn- und Leerlaufspannung weniger Verlustwärme an den LDOs. Wie erwartet und wie in Abbildung 2.17 ersichtlich, lieferten die Low Noise LDOs eine sehr saubere Ausgangsspannung.

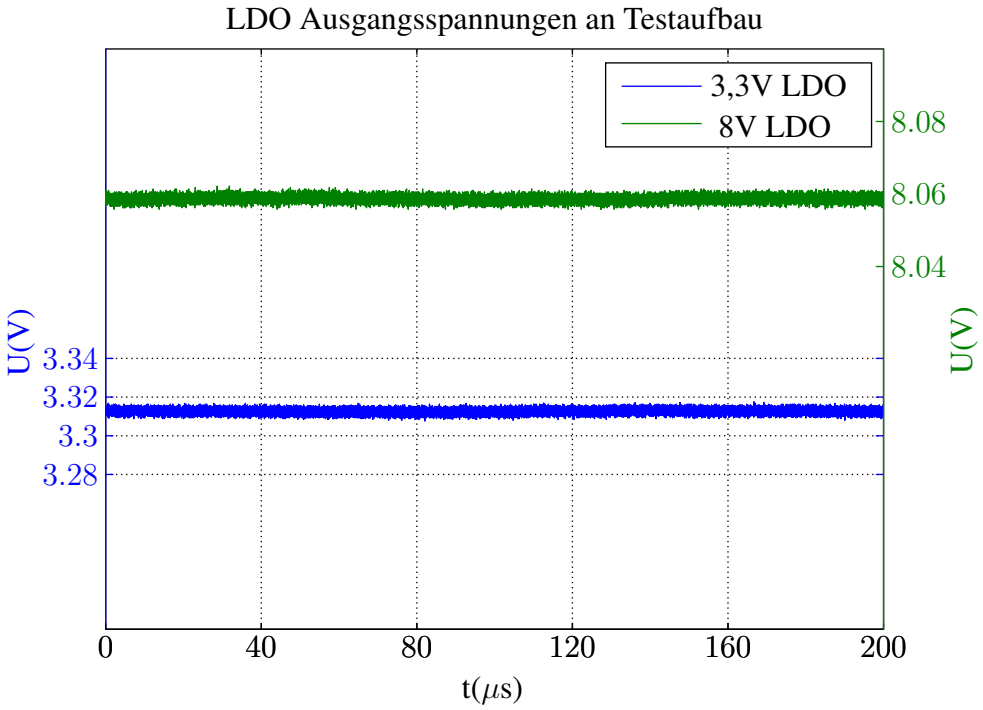


Abbildung 2.17.: Ausgangsspannungen vom 3,3 V LDO (blau) sowie vom 8 V LDO (grün) am Testaufbau

Um die Einkopplung eines Zellspannungssensors zu untersuchen, wurde der Eingang des Power-OPs (Abbildung 2.16) mit 1 V Eingangsspannung von einem Labornetzteil beschaltet. Bei einem eingestellten Verstärkungsfaktor von 2 lieferte der Power-OP am Ausgang nun saubere 2 V mit knapp 10 mV Noise, wie Abbildung 2.18 zeigt. Die Einkopplung eines Zellspannungssensors wurde in Abschnitt 2.1.3 näher untersucht. Es zeigte sich, dass die Störamplituden gegenüber dem Ausgangssignal vom Zellspannungsgenerator V01 deutlich geringer ausfielen.

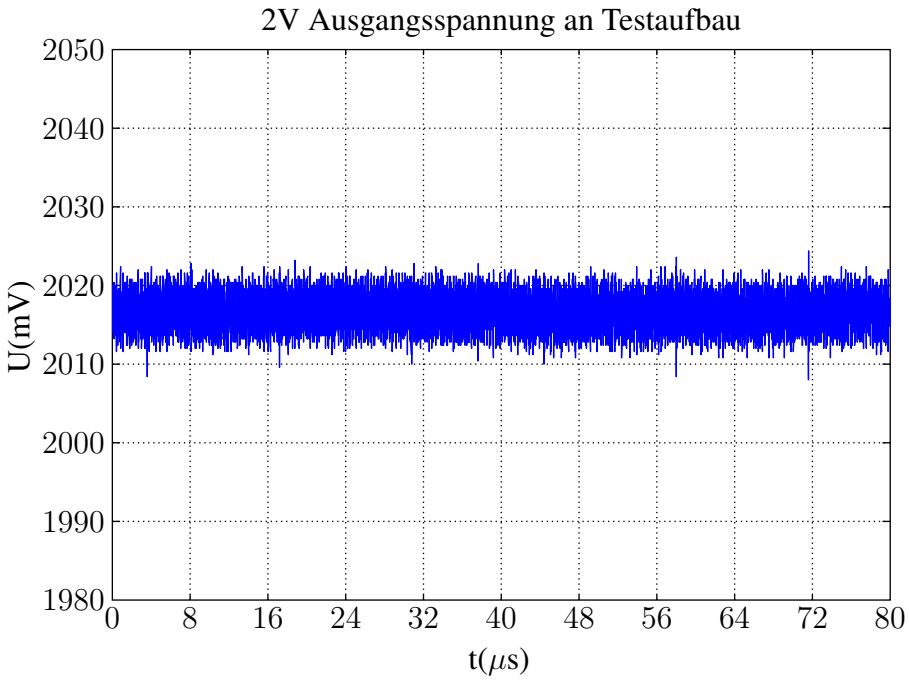


Abbildung 2.18.: 2 V Ausgangsspannung des Power-OPs am Testaufbau

Bei Belastung der Ausgangsspannung durch einen Sensor mit permanent laufendem Step-Up/Down Converter wurde bei hoher Zeitauflösung des Oszilloskops festgestellt, dass die Spannung um mehr als 10 mV einbricht. Es stellte sich heraus, dass sich der Ground des Testaufbaus bei Belastung um wenige mV anhebt. Durch den Verstärkungsfaktor 2 des Power-OPs verdoppelt sich der Fehler dann an der Ausgangsspannung. Abbildung 2.19 zeigt im unteren Teil die einbrechende Ausgangsspannung. Im oberen Teil ist der Potenzialunterschied zwischen dem Ground-Potential am Netzteil sowie dem Ground-Potential des Power-OPs ersichtlich. Als Grund für diesen Effekt wurde der nicht optimale Aufbau sowie die Kabelverbindung zwischen Netzteilaufbau und Verstärker verantwortlich gemacht. Beim anstehenden Platinenlayout für das Redesign soll dieses Problem durch entsprechende Leiterbahnbreite sowie Massefläche verhindert werden.

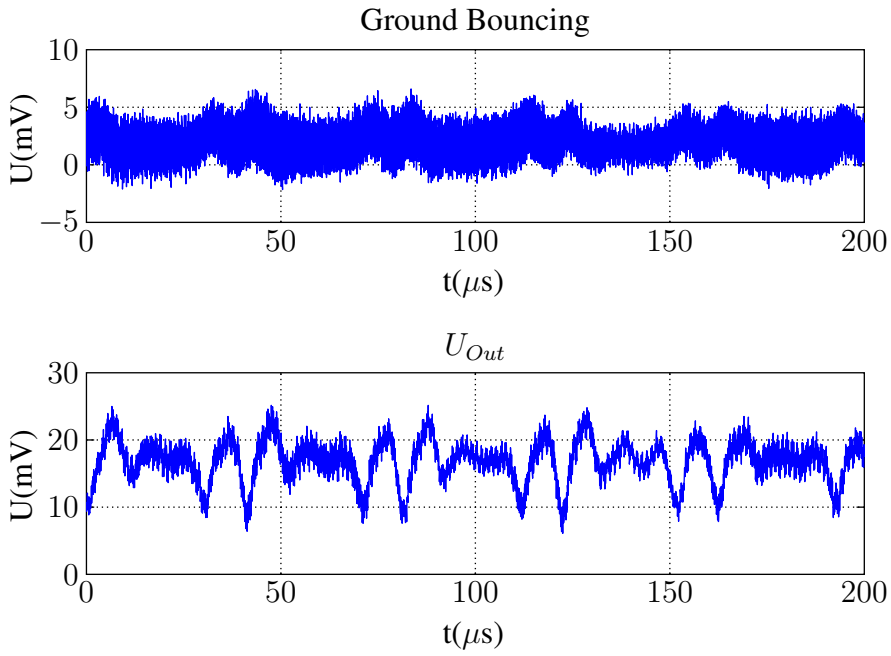


Abbildung 2.19.: Ground Bouncing bei Last durch einen Klasse 1 Sensor - Der obere Plot zeigt das schwankende Ground-Potential, der untere Plot zeigt die schwankende Ausgangsspannung des Power-OPs

3. Hardware-Entwurf

Dieses Kapitel beschreibt zunächst die Anforderungen, welche an die Hardware gestellt werden und die daraus resultierende Auswahl an Bauelementen. In den darauf folgenden Schritten wurde der Schaltplan erstellt und anschließend in das Platinenlayout umgesetzt. Das entstehende Redesign wird zur Vereinfachung nachfolgend als Zellspannungsgenerator V02 bezeichnet.

3.1. Anforderungen

Die grundsätzlichen Anforderungen an die Hardware des Zellspannungsgenerators wurden in [35, Kap. 2.1-2.4] bereits aufgestellt und wie folgt definiert:

Anforderung	**Spezifikation**
Spannungsbereich	$0,5\,\mathrm{V} - 5,5\,\mathrm{V}$
Strombedarf	$ca.\ 250\,\mathrm{mA}$
Genauigkeit	$\approx 1,22\,\mathrm{mV}$
Auflösung	$< 1,22\,\mathrm{mV}$
Grenzfrequenz	$\geq 2,5\,\mathrm{kHz}$
Samplerate	$\geq 20\,\mathrm{kHz}$

Tabelle 3.1.: Grundsätzliche Anforderungen an den Zellspannungsgenerator nach [35, Kap. 2.4]

Zusätzliche Anforderungen ergeben sich durch die notwendige galvanische Trennung sowie durch die Notwendigkeit der Synchronisation der einzelnen Module. Diese werden in den nächsten Unterkapiteln im Zuge der Beschreibung des Hardware-Konzeptes erläutert.

3.2. Konzept

Im Folgenden wurde ein Konzept für das Redesign des Zellspannungsgenerators erarbeitet, in dem zunächst geeignete Bauelemente ausgewählt wurden. Das Grobkonzept sieht aufgrund der Fortführung vom Zellspannungsgenerator V01 diesem sehr ähnlich und unterscheidet sich vor allem in Details. Alle Komponenten, welche sich bewährt haben, wurden auch im Redesign wieder eingesetzt. Diese werden in den folgenden Abschnitten erläutert.

Die Module des Zellspannungsgenerators sollen weiterhin im Eurokarten-Format aufgebaut werden und anschließend als Einschübe in einem 19“-Rack Platz finden. Alle Module, sowohl Master als auch Slaves, sollen identisch aufgebaut werden. Dadurch können die Module sowohl einzeln zur Nachbildung einer Batteriezelle als auch im Mehrkanalbetrieb zur Nachbildung einer kompletten Batterie eingesetzt werden.

Die Abbildung 3.2 gibt einen Überblick über das Gesamtsystem, Abbildung 3.1 zeigt das Spannungsversorgungskonzept.

3.2.1. Spannungsversorgung

Die Spannungsversorgung ist ein wichtiger Bestandteil des Zellspannungsgenerators. Bestätigt durch die Voruntersuchungen in Unterkapitel 2.3 soll im Redesign ein herkömmliches Linear-Netzteil zum Einsatz kommen. Dadurch entfällt eine insbesondere für den Leistungsteil aufwändige Filterung der einzelnen Versorgungsspannungen, welche beim Einsatz eines DC/DC-Konverters notwendig wäre.

Die Spannungsversorgung setzt auf dem Netzteil des Testaufbaus aus Unterkapitel 2.4 auf, welches einen Flachtrafo der Firma Myrra [26] mit 9 V Nennspannung und 10 VA Nennleistung verwendet. Der Netztrafo kann damit die doppelte Leistung des zuvor verwendeten DC/DC-Konverters zur Verfügung stellen und sollte damit eine stabile Versorgung auch bei Lastschwankungen eines angeschlossenen Zellspannungssensors ermöglichen. Nachgeschaltet werden drei Linearregler sowie eine Referenzspannungsquelle. Zunächst werden 3,3 V für den Digitalteil der meisten Komponenten benötigt, dafür wurde ein LDO vom Typ LT1763 der Firma Linear Technology [22] vorgesehen.

Die Bauelemente, welche ebenfalls analoge Komponenten beinhalten, wie der ADC und DAC, werden von einem separaten Linearregler gespeist, um nicht von den Schaltvorgängen der Digitaltechnik und daraus resultierenden Spannungsschwankungen beeinträchtigt zu werden. Hier wurde ein Low Noise LDO vom Typ LT1763 der Firma Linear Technology [22] vorgesehen. Zusätzlich wird vom ADC sowie dem DAC noch eine 2,5 V Referenzspannungsquelle benötigt, welche mit dem Typ REF5025 der Firma Texas Instruments [45]

bereitgestellt wird. Für die Ausgangsstufe soll ein Linearregler zum Einsatz kommen, welcher sich auf 8 V Ausgangsspannung einstellen lässt. Hier wurde ein Low Noise LDO vom Typ LT1963A der Firma Linear Technology [23] vorgesehen, welcher laut Datenblatt speziell für schnelle Lastwechsel optimiert ist, welche bei Speisung ein Zellspannungssensors auftreten. Die Abbildung 3.1 zeigt das beschriebene Spannungsversorgungskonzept.

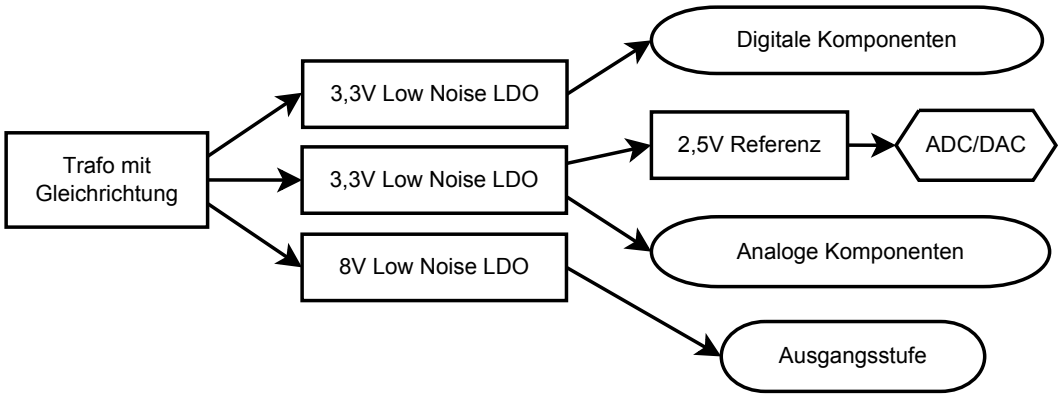


Abbildung 3.1.: Darstellung des Spannungsversorgungskonzepts für das Redesign

3.2.2. Mikrocontroller

Beim Zellspannungsgenerator V01 wurde ein Evaluations-Board vom Typ EK-LM3S9B92 der Firma Texas Instruments [48] eingesetzt. Dieses enthält einen 32-Bit ARM Cortex M3 vom Typ LM3S9B92 [49] mit 256 kB Flash und 96 kB SRAM. Diverse benötigte Peripherie wie SPI, I2C, drei ADC Kanäle, Ethernet MAC und PHY sind ebenfalls enthalten. Der Controller hat sich sowohl von der Hardware als auch von der herstellereigenen, als StellarisWare bezeichneten Treiberbibliothek [51] bewährt und soll auch in anderen Bereichen des BATSEN Forschungsprojekts zum Einsatz kommen. Als Nachteil haben sich jedoch verschiedene Einschränkungen des Evaluations-Boards erwiesen, wie zum Teil schon in den Abschnitten 2.2.1 und 2.2.4 erwähnt.

Darunter fällt unter anderem die Ethernetbuchse, deren Abschirmung fest mit dem Ground-Potential des 4-lagigen Evaluations-Boards verbunden ist und damit die galvanische Trennung der einzelnen Module verhindert (siehe Abschnitt 2.2.4). Weiterhin sind nicht alle IOs des Controllers über die Stiftleisten von dem Evaluations-Board nach außen geführt, weshalb einige Komponenten, allen voran das External Peripheral Interface (EPI), notwendig zur parallelen Anbindung von externem Speicher, nicht genutzt werden können. Ebenso kann die Traffic-LED der Ethernetbuchse nicht zur Signalisierung von Datenverkehr genutzt werden, da es eine Doppelbelegung mit der SPI-Schnittstelle gibt.

Aus diesen Gründen wurde für das Redesign entschieden, trotz der deutlich höheren Komplexität, den Controller direkt auf dem Mainboard zu platzieren. Wegen Lieferschwierigkeiten des genannten Typs LM3S9B92 [49] wurde auf den pinkompatiblen Typ LM3S9B96 [50] ausgewichen, welcher bis auf zwei zusätzliche, nicht benötigte Features (Zeitprotokoll IEEE 1588 sowie SafeRTOS Unterstützung) komplett identisch ist. Der Controller ist die zentrale Komponente jedes Zellspannungsgenerator-Moduls, wie Abbildung 3.2 zeigt. Dieser kann mit einem maximalen Systemtakt von 80 MHz betrieben werden, allerdings treten dabei mehrere Einschränkungen auf, da verschiedene Peripherie-Komponenten nur mit maximal 50 MHz Takt arbeiten können. Das hat zur Folge, dass etwa die EPI-Schnittstelle bei einem Systemtakt größer 50 MHz nur noch mit halbem Systemtakt arbeitet. Weiterhin benötigten verschiedene Operationen dann zwei Clock Cycles und arbeiten effektiv langsamer als bei 50 MHz Systemtakt.

Aus diesen Gründen wurde entschieden, den Controller mit 50 MHz Systemtakt zu betreiben, da ein höherer Takt bei der gewünschten Anwendung mehr Nach- als Vorteile mit sich bringt. Der Systemtakt wird aus einer internen PLL generiert, welche stets einen 200 MHz Takt erzeugt, der Systemtakt leitet sich dann über einen konfigurierbaren Teiler davon ab. Die Application Note AN01240 [43] gibt einen Überblick über den internen Clock Tree und dessen Konfigurationsmöglichkeiten. Als Taktquelle für die PLL wurde ein 16 MHz Quarz verwendet.

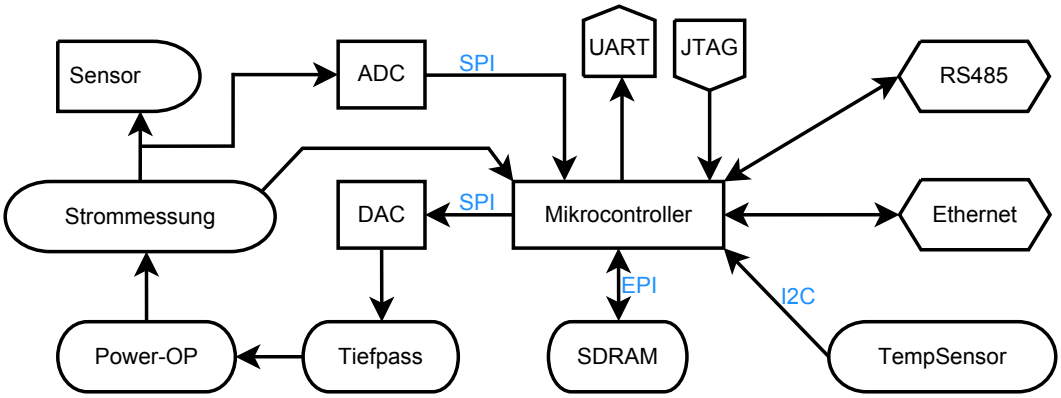


Abbildung 3.2.: Darstellung der am zentralen Controller angebundenen Komponenten und Schnittstellen

3.2.3. Ethernetverbindung

Die Steuerung des Zellspannungsgenerators sowie die Übertragung von Spannungssequenzen soll aufgrund der in Abschnitt 2.2.4 beschriebenen Vorteile weiterhin per Ethernet erfolgen. Dazu soll der integrierte Ethernet-Controller des LM3S9B96 [50] genutzt werden. Dieser enthält bereits den Media Access Control (MAC), welcher die Zugriffssteuerung auf

Layer 2 im OSI-Modell übernimmt. Auch der Ethernet Physical Layer (PHY) ist im Controller enthalten, welcher die Anbindung an die physikalische Schnittstelle übernimmt, welche den Layer 1 im OSI-Modell darstellt. Der Ethernet-Controller ist nicht an den internen Clock Tree des Mikrocontrollers angebunden und benötigt deshalb einen eigenen 25 MHz Quarz.

Eine Ethernetbuchse, wie sie bisher auf dem Evaluations-Board vorhanden war, wird nun direkt auf dem Mainboard integriert. Dazu wurde eine Buchse mit integrierten Übertragern und zwei LEDs zur generellen Anzeige einer Ethernetverbindung sowie zur Signalisierung von Datenverkehr vom Typ SI-60062-F der Firma Bel Components [8] verwendet. Die Abschirmung der Buchse soll zur Wahrung der galvanischen Trennung im Layout nicht angebunden werden, dennoch wurden Lötpads zur optionalen Bestückung eines Abblockkondensators vorgesehen werden.

3.2.4. Speicher

Um mit dem Zellspannungsgenerator eine Spannungssequenz wiedergeben zu können, muss diese zuerst aus Matlab per Ethernet an den Mikrocontroller übertragen werden. Der Controller besitzt allerdings nur 256 kB Flash und 96 kB SRAM. Darin könnten nur wenige Werte und damit sehr kurze Spannungssequenzen gespeichert werden. Zusätzlich sollen auch noch, sofern aktiviert, die während der Transienten-Wiedergabe gemessenen Strom-Werte gespeichert werden, um diese anschließend per Ethernet abfragen zu können. Bei einer 60 s langen Sequenz würden bei einer Samplerate von 20 kHz und jeweils 16 Bit für das Spannungs- und Strom-Sample

$$20\,\mathrm{kHz} * 60\,\mathrm{s} * 2 * 16\,\mathrm{Bit} = 38,4\,\mathrm{MBit}$$

Speicherplatz benötigt werden. Bisher wurde als externer Speicher ein SPI-Flash verwendet. Dieser hat den Nachteil der geringen Datenübertragungsrate über die serielle SPI-Schnittstelle und stellt damit gewissermaßen einen Flaschenhals im System dar. Weiterhin ist nachteilig, dass auf den SPI-Flash nur blockweise zugegriffen werden kann, wodurch immer nur 4 KByte-Blöcke gelesen und geschrieben werden können.

Für das Redesign soll deshalb ein externer RAM parallel an den Mikrocontroller angebunden werden. Dafür bietet der Controller die EPI-Schnittstelle. Diese kann einen bis zu 64 MByte externen Speicher 16 Bit breit adressieren und im Adressraum des Controllers abbilden. Ebenso übernimmt die Hardware der EPI-Schnittstelle das komplette Handling aller notwendigen Signale wie *Bank Select*, *Row/Column Address Strobe Command*, *Write Enable* sowie *Lower/Upper Byte Mask* und arbeitet bei vollem Systemtakt von 50 MHz.

Dieses, als auch die parallele Anbindung bringt deutliche Vorteile in der Datenübertragungsrate gegenüber dem zuvor eingesetzten SPI-Flash. Ebenso ist ein wesentlich einfacherer und schnellerer Zugriff auf jedes einzelne Datenwort per Adress-Pointer möglich.

Als Speicherchip wurde ein SDRAM mit 32 MB bzw. 256 MBit Kapazität vom Typ IS42S83200D der Firma Issi [16] ausgewählt. Damit stehen genügend Reserven zur Verfügung, um auch längere Spannungssequenzen zwischenspeichern zu können. Weil SDRAM allerdings flüchtig ist, können keine Konfigurationsdaten mehr darin gespeichert werdern, wie zuvor im SPI-Flash. Diese sollen nun im Flash des Controllers gespeichert werden, wie Unterkapitel 4.10 im Einzelnen beschreibt.

3.2.5. Ausgangsstufe

Die Ausgangsstufe für das Redesign wurde analog zu [35, Kap. 4.1-4.3] aufgebaut und besteht aus dem DAC mit anschließender Vorverstärkung, Tiefpass-Filter, Impedanzwandler sowie dem Leistungs-Operationsverstärker. Abbildung 3.3 zeigt den prinzipiellen Aufbau.

Abbildung 3.3.: Komponenten der Ausgangsstufe

Laut Anforderung soll eine maximale Ausgangsspannung von 5,5 V am Ausgang des Zellspannungsgenerators eingestellt werden können, mit entsprechender Reserve wurde die maximale Ausgangsspannung auf 6 V festgelegt. Aufgrund der Referenzspannung von 2,5 V muss die Ausgangsspannung vom DAC um den Faktor $v_{Out} = \frac{6\,\mathrm{V}}{2{,}5\,\mathrm{V}} = 2,4$ verstärkt werden, woraus sich bei den 16 Bit Auflösung etwa 91 μV pro LSB ergeben, was um Faktor 13 kleiner der geforderten Auflösung von 1,22 mV ist und damit auch innerhalb der aufgestellten Spezifikation liegt.

Als DAC wird weiterhin der AD5541 von Analog Devices [6] verwendet. Dieser hat 16 Bit Auflösung und benötigt eine Referenzspannung von 2,5 V. Die Ansteuerung erfolgt per SPI. Der bisher eingesetzte und nach [35, Kap. 3.2] evaluierte Leistungs-Operationsverstärker vom Typ OPA564 der Firma Texas Instruments [47] hat sich beim Zellspannungsgenerator V01 sowie im zuvor durchgeführten Testaufbau ebenfalls bewährt und wird weiterhin eingesetzt. Der Power-OP kann einen maximalen Ausgangsstrom von 1,5 A bereitstellen und kommt ohne symmetrische Versorgung aus. Weiterhin hat der OPA564 ein Übertemperatur- und ein Überstrom-Signal, welches von der Software im Mikrocontroller als Flag ausgewertet werden kann.

Beim Zellspannungsgenerator V01 wurde die Verstärkerschaltung auf einer Zusatzplatine realisiert und auf das Mainboard gesteckt. Für das Redesign soll der Power-OP direkt auf der Modulplatine integriert werden. Vorteile sind kurze Leiterbahnwege für die Versorgung des Leistungsteils sowie zu den in Abschnitt 3.2.7 erläuterten Instrumentenverstärkern der Strommessung und damit verbunden zu den ADCs des Mikrocontrollers. Auch der Verzicht auf Steckverbindungen sollte zu geringen Leitungswiderständen beitragen.

Für die Vorverstärkung sowie für die Impedanzwandlung wurde ein Operationsverstärker vom Typ LM7301 der Firma Texas Instruments [46] verwendet, wobei hier als Verstärkungsfaktor $v = 1,2$ gewählt wurde. Um auf die Gesamtverstärkung von $v_{Out} = 2,4$ zu kommen, soll der Power-OP mit $v = 2$ betrieben werden. Die Verstärkung für den Power-OP wurde größer $v = 1$ gewählt, um eine bessere Stabilität zu gewährleisten.

Die erforderliche Grenzfrequenz des Tiefpasses liegt nach [35, Kap. 2.3] bei 2,5 kHz. Dieser wird als Filter 2. Ordnung basierend auf [35, Kap. 4.2] realisiert und besteht aus zwei RC-Gliedern vor und hinter dem Impedanzwandler. Der Impedanzwandler ist notwendig, um die gegenseitige Einflussnahme beider Tiefpässe 1. Ordnung zu vermeiden.

3.2.6. Ausgangspuffer

Ein weiterer Schwerpunkt des Redesigns liegt in der Realisierung von zwei Betriebsarten, zum einen der Transienten-Wiedergabe und zum anderen der Kalibrierung der Zellspannungssensoren. Um für den Kalibriermodus eine maximale Stabilität der eingestellten Ausgangsspannung zu gewährleisten, wurde nach Abbildung 3.4 eine zuschaltbare Pufferung bestehend aus zwei zum Ausgang parallel geschalteten Kondensatoren und einer in Reihe geschalteten Induktivität vorgesehen. Die Kondensatoren C1 und C2 dienen als Puffer bei schnellen Lastwechseln, wohingegen C1 als Elektrolyt-Kondensator zur Dämpfung langsamer Einkopplungen und C2 als Keramik-Kondensator zur Dämpfung hochfrequenter Störungen dienen soll. Die Induktivität soll Einkopplungen in den Verstärker des Zellspannungsgenerators minimieren.

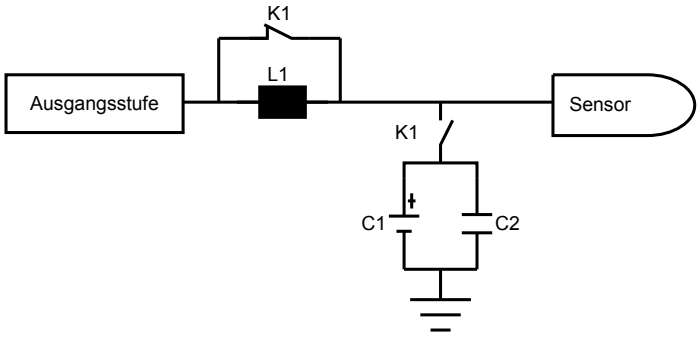


Abbildung 3.4.: Zuschaltbare Pufferung der Ausgangsspannung

Es wurde entschieden, die Umschaltung zwischen beiden Betriebsmodi per Relais zu realisieren, wozu kleine SMD Relais mit 2 A Schaltleistung und 3 V Nennspannung der Firma TE Connectivity [39] verwendet wurden. Diese lassen sich mit Hilfe eines Transistors zur Verstärkung des Ausgangsstroms durch einen GPIO des Mikrocontrollers ansteuern. Die Vorteile eines Relais gegenüber einem FET liegen in der galvanischen Trennung des Schaltkreises gegenüber dem Steuerkreis, einem nahezu unendlichen Widerstand bei geöffnetem Kontakt sowie einem sehr niederohmigen Durchgangswiderstand bei geschlossenem Kontakt. Ein FET hätte im durchgeschaltetem Zustand stets noch ein $R_{DSon} = 10 - 50\,\mathrm{m\Omega}$. Weiterhin kommt es nicht auf schnelle Schaltvorgänge an, weil das Relais nur einmal zur Auswahl der Betriebsart geschaltet werden muss.

3.2.7. Strommessung

Wichtig zur Analyse des Verhaltens der Zellspannungssensoren ist die Kenntnis der Stromaufnahme bei konstanter Ausgangsspannung und während der Transienten-Wiedergabe. Dazu muss nach der Ausgabe eines Samples über den DAC der aktuelle Stromfluss zwischen Ausgangsstufe und Sensor gemessen und aufgezeichnet werden. Beim Zellspannungsgenerator V01 war bereits eine Strommessung vorgesehen. Diese ist mit Hilfe eines 1 Ω Shunt-Widerstands in der High-Side zwischen Power-OP und Anschlussbuchse für den Sensor realisiert. Der Shunt bringt allerdings Probleme mit sich, da der Spannungsabfall je nach Stromaufnahme variiert und dadurch die Ausgangsspannung unterschiedlich stark einbricht, was vor allem die Genauigkeit im Kalibriermodus beeinträchtigt.

Allen Erkenntnissen nach scheint eine Strommessung speziell von sehr kleinen Strömen im μA-Bereich ohne jegliche Beeinträchtigung der Ausgangsspannung nicht möglich zu sein. Deshalb wurde entschieden, diese genau wie den Ausgangspuffer abschaltbar zu realisieren, indem die Shunt-Widerstände per Relais gebrückt werden. Somit kann die Strommessung sowie deren Einfluss auf die Ausgangsspannung wahlweise aktiviert werden. Abbildung 3.5 zeigt den prinzipiellen Aufbau.

Bei der alten Realisierung wurde der Spannungsabfall über dem 1 Ω Shunt durch zwei Instrumentenverstärker gemessen und mit unterschiedlichem Gain verstärkt und dann über zwei ADC-Kanäle des Mikrocontrollers gemessen. Der *kleine* Messbereich wies durch die hohe Verstärkung mit einem Gain von $G = 838$ allerdings ein starkes Rauschen auf. Im Redesign wurden ebenfalls zwei Messbereiche vorgesehen, allerdings mit einem eigenen Shunt-Widerstand je Instrumentenverstärker. Für den *großen* Messbereich wurde der 1 Ω Shunt beibehalten, für den *kleinen* Messbereich hingegen wurde ein 5 Ω Shunt vorgesehen, um einen höheren Spannungsabfall zu erzeugen. Damit kann der Instrumentenverstärker mit niedrigerem Gain und somit geringerem Rauschen betrieben werden.

Als Instrumentenverstärker wurden weiterhin die AD8226 von Analog Devices [4] verwendet. Dieser Rail-to-Rail Verstärker kann die Spannung zwischen zwei Punkten unabhängig

vom Ground-Potential hochohmig messen und mit einem durch einen Widerstand konfigurierbaren Gain verstärken. Mit einer angelegten Offset-Spannung ist auch die Messung negativer Potentialdifferenzen möglich. Das Ausgangssignal wird an jeweils einen ADC-Kanal des Mikrocontrollers mit einer Referenzspannung von $V_{REF} = 3,3\,\mathrm{V}$ angeschlossen. Diese haben zwar nur eine Auflösung von 10 Bit, können aber mit einer maximalen Samplerate von 1 MSPS betrieben werden. Die anschließende Bestimmung der Messbereiche zeigt, dass die vorhandene Auflösung ausreichend ist und auf den Einsatz externer AD-Converter verzichtet werden kann.

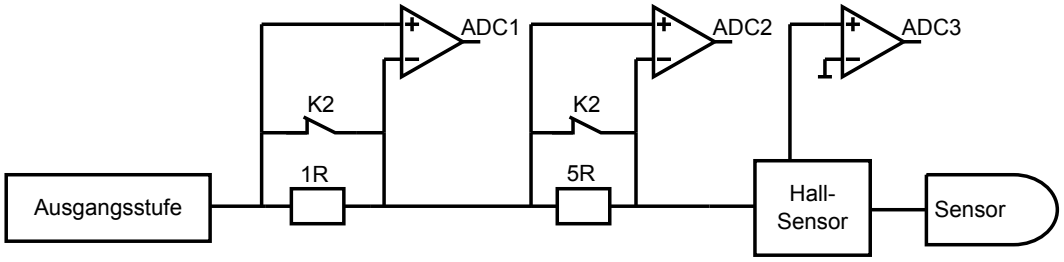


Abbildung 3.5.: Zuschaltbare Strommessung mit zwei Shunt-Widerständen für zwei Messbereiche sowie experimentellem Hall-Sensor als dritten Messbereich

Laut den gestellten Anforderungen soll ein maximaler Strom von $I_{max} = 250\,\mathrm{mA}$ und nach [35, Kap. 4.4] ein minimaler Strom von $I_{min} = 100\,\mu\mathrm{A}$ gemessen werden können. Die Bestimmung der Messbereiche erfolgt in den folgenden Unterkapiteln. Als weitere experimentelle Messmethode wird ein integrierter Hall-Sensor beschrieben.

Bestimmung von Messbereich 1

Der *große* Messbereich wurde wie folgt definiert:

$$U_{Shunt,1} = R_{Shunt,1} \cdot I_{max} \tag{3.1}$$

$$U_{Shunt,1} = 1\,\Omega \cdot 250\,\mathrm{mA} = 250\,\mathrm{mV} \tag{3.2}$$

Um eine maximale Aussteuerung des ADC mit $V_{REF} = 3,3\,\mathrm{V}$ zu erreichen, muss zunächst der Gain des zugehörigen Instrumentenverstärkers bestimmt werden:

$$G_1 = \frac{V_{REF}}{U_{Shunt,1}} = \frac{3,3\,\mathrm{V}}{250\,\mathrm{mV}} = 13,2 \tag{3.3}$$

Der Widerstand zur Konfiguration der Verstärkung berechnet sich laut Datenblatt des Instrumentenverstärkers [4] folgendermaßen:

$$R_{G,1} = \frac{49,4\,\mathrm{k\Omega}}{G-1} = \frac{49,4\,\mathrm{k\Omega}}{13,2-1} \approx 4\,\mathrm{k\Omega} \tag{3.4}$$

Um mit einem Widerstand der E24-Reihe auszukommen, sollten hier $4,3\,\mathrm{k\Omega}$ gewählt werden. Nach Umstellung von Gleichung 3.4 ergibt sich dann eine Verstärkung von $G_1 \approx 12,5$. Die maximale Auflösung des 10 Bit ADC in Verbindung mit der 3,3 V Referenz beträgt:

$$U_{LSB} = \frac{3,3\,\mathrm{V}}{1024} = 3,22\,\mathrm{mV} \tag{3.5}$$

Unter Einbeziehung der zuvor berechneten Verstärkung ergibt sich:

$$U_{LSB,1} = \frac{U_{LSB}}{G_1} = \frac{3,22\,\mathrm{mV}}{12,5} = 258\,\mu\mathrm{V} \tag{3.6}$$

Dies führt an $R_{Shunt,1} = 1\Omega$ zu einer Strom-Auflösung von $I_{LSB,1} = 258\,\mu\mathrm{A}$ für den *großen* Messbereich.

Bestimmung von Messbereich 2

Im *kleinen* Messbereich sollen Ströme von etwa $100\,\mu\mathrm{A}$ gemessen werden können. Es wurde eine Auflösung von $I_{LSB,2} = 10\,\mu\mathrm{A}$ anvisiert, dabei ergibt sich am 5Ω Shunt folgender Spannungsabfall:

$$U_{Shunt,2} = R_{Shunt,2} \cdot I_{LSB,2} \tag{3.7}$$

$$U_{Shunt,2} = 5\,\Omega \cdot 10\,\mu\mathrm{A} = 50\,\mu\mathrm{V} \tag{3.8}$$

Um die definierte Strom-Auflösung von $I_{LSB,2} = 10\,\mu\mathrm{A}$ zu erreichen, muss $U_{Shunt,2}$ auf die minimale ADC-Auflösung von $U_{LSB} = 3,22\,\mathrm{mV}$ verstärkt werden:

$$G_2 = \frac{U_{LSB}}{U_{Shunt,2}} = \frac{3,22\,\mathrm{mV}}{50\,\mu\mathrm{V}} = 64,4 \tag{3.9}$$

Nach Gleichung 3.4 ergibt sich zur Konfiguration der Verstärkung ein Widerstand von $R_{G,2} \approx 780\,\Omega$. Nach der E24-Reihe sollten hier $820\,\Omega$ gewählt werden, dies führt nach Gleichung 3.4 zu einem Gain von $G_2 \approx 61$. Daraus resultiert folgende Strom-Auflösung:

$$I_{LSB,2} = \frac{1}{R_{Shunt,2}} \cdot \frac{U_{LSB}}{G_2} \tag{3.10}$$

$$I_{LSB,2} = \frac{1}{5\Omega} \cdot \frac{3,22\,\mathrm{mV}}{61} \approx 10\,\mu\mathrm{A} \tag{3.11}$$

Integrierter Hall-Sensor

Zusätzlich zur Strommessung per Shunt-Widerstand wurde experimentell eine Strommessung mit einem integrierten Hall-Sensor vorgesehen. Ein Hall-Sensor bestimmt den zu messenden Strom über das Magnetfeld, welches vom stromdurchflossenen Leiter induziert wird und liefert eine dazu proportionale Ausgangsspannung. Prinzipiell ist also eine kontaktlose Strommessung möglich, allerdings eignen sich die meisten Hall-Sensoren eher zur Bestimmung großer Ströme, wo der Einsatz eines Shunts nicht mehr sinnvoll ist. Der neuartige und hier eingesetzte Hall-Sensor vom Typ ACS710 der Firma Allegro [2] soll sich auch zur Messung relativ kleiner Ströme eignen. Laut Datenblatt besitzt der Sensor ein ausgangsseitiges Rauschen von $V_{NOISE} = 4,5\,\mathrm{mV}$ bei einer Sensitivität von $V_{SENSE} = 151\,\mathrm{mV/A}$. Damit sollte der Sensor brauchbare Ergebnisse ab etwa $30\,\mathrm{mA}$ Stromdurchfluss liefern.

Zur Anpassung der Ausgangsspannung des Hall-Sensors an $I_{max} = 250\,\mathrm{mA}$ wird diese ebenfalls per Instrumentenverstärker verstärkt. Bei $V_{SENSE} = 151\,\mathrm{mV/A}$ liefert der Sensor bei I_{max} eine Ausgangsspannung von $U_{Hall} = 37,75\,\mathrm{mV}$. Um den ADC voll auszusteuern, muss diese auf V_{REF} verstärkt werden:

$$G_3 = \frac{V_{REF}}{U_{Hall}} = \frac{3,3\,\mathrm{V}}{37,75\,\mathrm{mV}} = 87,4 \tag{3.12}$$

Nach Gleichung 3.4 errechnet sich $R_{G,3} \approx 560\,\Omega$. Damit ergibt sich eine maximale Strom-Auflösung von:

$$I_{LSB,3} = \frac{U_{Hall}}{10\,\mathrm{Bit}} = \frac{37,75\,\mathrm{mV}}{1024} \approx 37\,\mu\mathrm{A} \tag{3.13}$$

Zusammenfassung der Messbereiche

In Tabelle 3.2 sind zusammenfassend alle drei Messbereiche aufgeführt.

	Messbereich 1	Messbereich 2	Hall-Sensor
Auflösung	$258\,\mu\text{A}$	$\approx 10\,\mu\text{A}$	$\approx 37\,\mu\text{A}$
Gain	$12,5$	61	$87,4$
I_{min}	$258\,\mu\text{A}$	$\approx 10\,\mu\text{A}$	$\approx 30\,\text{mA}$
I_{max}	$250\,\text{mA}$	$\approx 10\,\text{mA}$	$250\,\text{mA}$

Tabelle 3.2.: Zusammenfassung der drei Messbereiche der Strommessung

Der Instrumentenverstärker für den Hall-Sensor muss mit dem höchsten Gain von $G = 87,4$ betrieben werden. Dieses liegt dennoch nur bei $\approx 10\%$ des Gains vom *kleinen* Messbereich beim Zellspannungsgenerator V01. Wie genau die Strommessung mit den drei geplanten Messbereichen in der Praxis tatsächlich funktioniert, kann nur durch eine Erprobung an der realen Hardware festgestellt werden.

3.2.8. Rückführung

Zur Kalibrierung des Systems ist die Messung der tatsächlich ausgegebenen Spannung über eine Rückführung notwendig. Dazu wird der 16-Bit ADC vom Typ AD7798 der Firma Analog Devices [3] verwendet. Der AD7798 wird per SPI abgefragt und besitzt drei differentielle Eingänge, wodurch unabhängig vom Ground-Potential gemessen werden kann. Die Messung erfolgt direkt an den Anschlussbuchsen, um alle Leiterverluste mit einzubeziehen. Der ADC wird ebenfalls mit der 2,5 V Referenzspannung betrieben, aus diesem Grund ist ein Spannungsteiler am Eingang notwendig, um die maximalen 6 V Ausgangsspannung messen zu können. Analog zur Verstärkung des DAC-Ausgangs aus Abschnitt 3.2.5 benötigt der Spannungsteiler ein Verhältnis $\frac{6}{2,5}$, womit sich auch hier eine Auflösung von $U_{LSB} = 91\,\mu\text{V}$ ergibt. Aufgrund der Bauteiltoleranzen, insbesondere des Spannungsteilers, ist eine Bestimmung von Offset- und Verstärkungs-Fehler bei jedem Modul erforderlich. Dieses kann jedoch softwareseitig realisiert werden, wie in Unterkapitel 4.4 beschrieben. Der eingesetzte ADC kann lediglich mit einer maximalen Samplerate von 470 Hz betrieben werden. Dieses genügt für eine Kalibrierung, für ein permanentes Nachregeln der Ausgangsspannung ist dies jedoch zu langsam. Ein dafür notwendiger Highspeed-ADC wäre jedoch sehr teuer und die Voruntersuchungen am Testaufbau lassen erwarten, dass der Power-OP durch seine Rückkopplung die Ausgangsspannung bei entsprechend stabiler Versorgung je nach Last nachregelt.

3.2.9. Synchronisation

Bei der gleichzeitigen Transienten-Wiedergabe von mehreren Modulen ist eine Synchronisation der vom DAC ausgegebenen Samples notwendig. Die Synchronisation erfolgte beim Zellspannungsgenerator V01 über ein einfaches Logiksignal. Die Nachteile dieser Lösung wurden bereits in Abschnitt 2.2.7 beschrieben.

Das neue Konzept sieht als alternative Lösung die Synchronisation per RS485-Bus vor. Dabei handelt es sich um ein differentielles Bus-System, welches durch die damit einhergehende Störsicherheit vor allem im industriellen Umfeld eingesetzt wird. Mit dieser Lösung ergeben sich eine Reihe von Vorteilen. Die Umschaltung zwischen Sende- und Empfangs-Modus ist per Software steuerbar, Jumpereinstellungen entfallen. Weiterhin sind die Treiberstufen eines RS485-Transceivers sehr hochohmig, diese können im Fehlerfall ohne Auswirkungen auf die Hardware gegeneinander senden.

Für die Anwendung der Synchronisation muss ein Modul als Master und damit softwareseitig als RS485-Transmitter und die Slave-Module als Receiver konfiguriert werden. Die Slave-Module müssen bei der Transienten-Wiedergabe nur auf die „SYNC“-Nachricht warten und anschließend per DAC ein Sample ausgeben. Für spätere Anwendungen ließen sich über den RS485-Bus aber auch problemlos weitere Daten zwischen den Modulen austauschen.

Als RS485-Transceiver wird der ADM2682E von Analog Devices [5] verwendet. Dabei handelt es sich um den eigentlichen Transceiver inklusive induktivem Koppler sowie DC/DC-Konverter in einem Gehäuse. Damit ist bei minimalem Hardware-Aufwand die galvanische Trennung weiterhin sichergestellt. Durch den integrierten DC/DC-Konverter entfällt ebenfalls eine zusätzliche Spannungsversorgung für den galvanisch getrennten Teil des Transceivers. Der Transceiver ist über einen UART mit dem Mikrocontroller verbunden, über zwei weitere GPIOs wird der Receive- und Transmit-Modus des Transceivers gesteuert.

3.2.10. Temperatursensor

Zur Messung der Umgebungstemperatur wurde weiterhin der Temperatursensor vom Typ TMP102 von Texas Instruments [41] integriert. Dieser kann per I2C angesteuert und abgefragt werden und hat eine Genauigkeit von $0,5°C$ sowie eine Auflösung von $0.0625°C$ pro LSB. Mit Hilfe des Temperatursensors kann einerseits festgestellt werden, wann der Zellspannungsgenerator seine Betriebstemperatur erreicht hat, bei der etwa eine Kalibrierung durchgeführt wurde. Andererseits können in einem Temperatur-Schrank Abhängigkeiten zwischen Temperatur und Ausgangsspannung evaluiert werden, welche sich dann softwareseitig kompensieren lassen.

3.2.11. Weitere Komponenten

Auf dem Board wurden weiterhin 6 LEDs vorgesehen, welche in erster Linie zum Debugging während der Entwicklung dienen sollen. Genau wie beim Zellspannungsgenerator V01 wurden davon zwei LEDs an die Frontplatte geführt, welche den Normalbetrieb und aufgetretene Störungen signalisieren sollen. Zusätzlich ist der zweite I2C-Kanal des Mikrocontrollers zusammen mit 3,3 V Versorgungsspannung auf eine Stiftleiste geführt, um eventuell hinzukommende Peripherie ansteuern zu können.

Zur Programmierung des Controllers wurde der beim Stellaris Evaluations-Board [48] mitgelieferte Programmer vom Typ BD-ICDI-B verwendet. Dieser enthält neben dem JTAG-Port ebenfalls einen UART-Port, über den Debug-Meldungen ausgegeben werden können. Aufgrund der verwendeten Wannenstecker, welche schwer zu beschaffen sind, wurde auf dem Board zusätzlich eine 10-polige Micromatch-Buchse vorgesehen, um einen zukünftigen eigenen Programmer mit einem gängigen Steckverbinder anschließen zu können.

3.3. Umsetzung

3.3.1. Schaltplan

Der Schaltplan und das Platinenlayout wurden in Eagle 5.11.0 umgesetzt. Dabei mussten verschiedene Komponenten erstellt werden, weil diese nicht in der Eagle Bibliothek vorhanden waren. Darunter fiel auch das Symbol sowie Pin-Mapping für den LM3S9B96 in seinem LQFP-100 Gehäuse. Der Schaltplan aus Anhang B wurde aufgrund der besseren Übersicht in 8 Unterpläne aufgeteilt, welche jeweils einen Bereich des Gesamtsystems widerspiegeln:

1. Spannungsversorgung
2. Synchronisation
3. Ethernetanbindung
4. SDRAM-Speicher
5. ADC & Temperatursensor
6. DAC & Filter
7. Power-OP & Strommessung
8. Instrumentenverstärker

3.3.2. Platinenlayout

Das Layout der Platine aus Anhang C wurde auf einer doppelseitigen Eurokarte mit den Abmessungen $160\,\mathrm{mm} * 100\,\mathrm{mm}$ erstellt, wobei alle Bauteile auf dem Top-Layer untergebracht werden konnten. Das Routing wurde komplett von Hand durchgeführt, auf den Einsatz eines Autorouters wurde verzichtet. Das manuelle Routing gestaltete sich insbesondere durch den Mikrocontroller mit seinen 100 Pins, welche auch fast alle belegt sind, sowie durch den SDRAM mit seinen 54 Pins als recht zeitaufwändig. Dennoch konnten so zuerst alle kritischen Pfade mit Bedacht geroutet werden, wozu unter anderem die Signale zur Ethernetbuchse, zum SDRAM sowie der komplette Analogteil gehört. Als Standard-Leiterbahnbreite für sämtliche Signale wurden 8 mil verwendet, was etwa 0,2 mm entspricht. Die Leiterbahnen des Leistungsteils wurden mit 80 mil (2 mm) Breite geroutet, um einen möglichst niedrigen Spannungsabfall zu erzielen. Nach [12, S. 841] tritt bei einer Breite von 80 mil und der hier verwendeten Kupferauflage von 35 μm nur noch ein Leiterwiderstand von $2,54\,\frac{\mathrm{m\Omega}}{\mathrm{cm}}$ auf. Es wurden getrennte Masseflächen unter dem Digitalteil sowie unter den analogen Komponenten erzeugt, um Einkopplungen der rein digitalen Komponenten in den Analogteil zu vermeiden.

3.3.3. Frontblende

Für die einzelnen Module wurden Frontblenden mit einer Breite von 8 Längeneinheiten (LE, etwa 40,6 mm) vorgesehen. In der Frontblende sollen die Bananenbuchsen zum Anschluss des Zellspannungssensors sowie zwei LEDs Platz finden. Die LEDs dienen der Signalisierung vom Normalbetrieb des Generators sowie einer aufgetretenen Störung.

3.3.4. 19“ Sub-Rack

Die einzelnen Module sollen hochkant in einem 19“ Sub-Rack Platz finden. Es wurde ein Rack ähnlich dem vom Zellspannungsgenerator V01 ausgewählt, allerdings eine um 6 cm tiefere Variante. Durch die zusätzlichen Tiefe hinter der Backplane bleibt mehr Platz für Isolationsmaßnahmen, welche durch die Versorgung mit 230 V Netzspannung erforderlich sind.

Das ausgewählte Rack der Firma Schroff bietet Platz für 84 Längeneinheiten (LE). Bei einer Breite von jeweils 8 LE pro Modul passen 10 Module in ein Sub-Rack, in den verbleibenden 4 LE kann ein Ein-/Ausschalter und eine Sicherung untergebracht werden. Für den geplanten größtmöglichen Ausbau von 40 Modulen zur Nachbildung einer Gabelstaplerbatterie würden damit 4 Sub-Racks benötigt werden.

Die Backplane enthält einen 48-poligen Verbinder mit halber Bauhöhe, im oberen Bereich sitzt die Ethernetbuchse. Über den Backplane-Anschluss muss nur die 230 V Netzspannung zugeführt werden und der RS485-Bus zwischen den Modulen verbunden werden.

4. Software-Implementierung

Der Hauptteil der Softwareentwicklung steckt in der Controller-Software für den Stellaris Mikrokontroller LM3S9B96 [50]. Als Entwicklungsumgebung wurde das Code Composer Studio v5 von Texas Instruments eingesetzt. Bei der Entwicklung wurde auf die herstellereigene Software-Bibliothek namens StellarisWare [51] zurückgegriffen, welche die Verwendung der Controller-Peripherie vereinfacht. Die entstandene Software basiert nur in wenigen Teilen auf der Software vom Zellspannungsgenerator V01 und wurde aufgrund der geänderten Rahmenbedingungen größtenteils neu implementiert.

Die Steuererung des Zellspannungsgenerators soll per Matlab erfolgen. Dafür wurden einige Matlab-Scripts geschrieben, welche die Übertragung sowie Wiedergabe der Spannungssequenzen ermöglichen.

4.1. Strukturierung

Der Mikrocontroller startet zunächst mit seinem internen RC-Oszillator, im Hauptprogramm wird deshalb zuerst die Taktquelle auf die PLL umgestellt und der Systemtakt auf 50 MHz gesetzt. Anschließend werden sämtliche *Init()*-Routinen zur Initialisierung der benötigten Komponenten aufgerufen. Die eigentliche Funktionalität läuft komplett interruptbasiert, ausgehend von den entsprechenden Interrupt-Handlern.

Das Controller-Programm wurde in einzelne Komponenten aufgeteilt, zu denen jeweils ein Source- und ein Header-File angelegt wurde. Sofern die jeweilige Komponente Zugriff auf Ports oder andere Peripherie benötigt, existiert im jeweiligen Source-File eine dafür zuständige *nameInit()*-Routine, wobei der *name* der Bezeichnung der jeweiligen Komponente entspricht. Beispielsweise existiert zur Initialisierung des SDRAMs eine Routine namens *sdramInit()*. Alle Funktionsnamen wurden nach dem camel-case Schema benannt und beginnen jeweils mit dem Namen der Komponente. Einzige Ausnahme sind die Command-Funktionen, welche auf einen Befehl hin ausgeführt werden, wie in Unterkapitel 4.3 erläutert. Diesen Funktionen ist der Prefix *cmd* vorangestellt.

Zu vielen Funktionen der StellarisWare-Bibliothek existiert eine weitere Variante mit dem Prefix *ROM_* mit identischer Funktionalität. Diese Funktionen greifen jedoch auf die on-chip StellarisWare zurück, welche sich im ROM des Controllers befindet. Die Nutzung der

on-chip Funktionen wirkt sich positiv auf die Programmgröße aus, wodurch Platz im Flash des Controllers eingespart wird. Wenn möglich, wurde deshalb von den *ROM*-Funktionen Gebrauch gemacht.

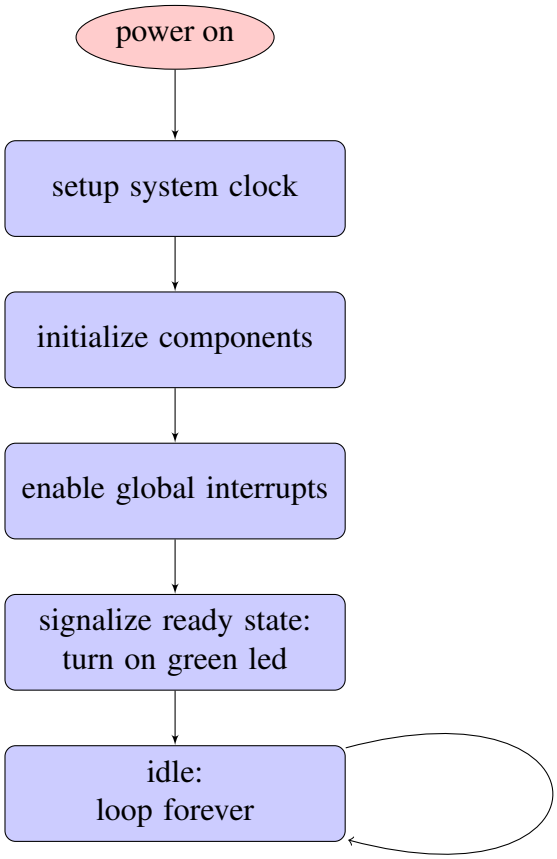


Abbildung 4.1.: Ablauf des Hauptprogramms

4.2. Ethernet-Kommunikation

Über die Ethernetverbindung erfolgt die Kommunikation mit dem Zellspannungsgenerator sowie die Übertragung von Spannungssequenzen. Im Controller sind mit PHY und MAC bereits die Layer 1 und 2 des OSI-Modells vorhanden. Die Kommunikation soll per Transmission Control Protocol (TCP) stattfinden, welches eine sichere Übertragung sowie ein Eintreffen der Datenpakete in der richtigen Reihenfolge gewährleistet. TCP befindet sich auf Layer 4 des OSI-Modells, darunter auf Layer 3 liegt das Internet Protocol (IP), welches die Adressierung der Module per IP-Adresse übernimmt.

Layer 3 und 4 wurden mit Hilfe des Lightweight TCP/IP stack (lwIP) [1] implementiert. Der Software Stack übernimmt das komplette Handling der Verbindung inklusive des Auf- und Abbaus. Dafür ist es notwendig, in regelmäßigen Abständen die Funktion *lwIPTimer()* aufrufen zu lassen. Dieses wird über den *SysTickIntHandler* realisiert, welcher periodisch vom Nested Vectored Interrupt Controller (NVIC) des Controllers aufgerufen wird. Beim SysTick handelt es sich um einen speziellen Timer-Interrupt, dessen Flag im Interrupt-Handler nicht zurückgesetzt werden muss. Der SysTick eignet sich beispielsweise zum Aufruf des Schedulers eines Real-Time Operating System (RTOS), zum Refresh des Controllers einer SD-Karte oder wie hier zum periodischen Aufruf eines Protokoll-Stacks.

Auf Layer 2 eines Netzwerks werden die einzelnen Geräte über ihre MAC-Adresse angesprochen. Jedes Ethernet-Interface besitzt daher eine 48 Bit lange MAC-Adresse, welche normalerweise weltweit eindeutig ist. Der Netzwerk-Controller des LM3S9B92 auf dem bisher verwendeten Stellaris Evaluations-Board hat ebenfalls eine offiziell von der Institute of Electrical and Electronics Engineers (IEEE) vergebene MAC-Adresse. Wie sich herausstellte, werden jedoch nur die Controller, welche auf einem Evaluations-Board ausgeliefert werden, mit einer offiziellen MAC-Adresse versehen. Alle einzeln verkauften Controller, wie der hier eingesetzte LM3S9B96 besitzen standardmäßig keine MAC-Adresse. Diese kann mit einem Software-Tool namens LMFlashProgrammer von Texas Instruments einmalig programmiert werden und wird dabei in einem speziellen USER-Register im Read-Only Memory (ROM) des Controllers gespeichert. Aufgrund der Kosten und des Zeitaufwands wurde auf eine Beantragung offizieller MAC-Adressen bei der IEEE verzichtet. Da der Zellspannungsgenerator nicht verkauft und nur im eigenen Labornetz betrieben wird, wurden für alle Module so genannte *Locally Administered Addresses* nach [13] vergeben, welche nicht anderweitig von der IEEE vergeben werden. Es muss hier lediglich beachtet werden, dass die MAC-Adresse im Labornetz eindeutig ist. Weil das für die MAC-Adresse gedachte USER-Register standardmäßig nur einmalig beschrieben werden kann, wurde darauf verzichtet, die MAC-Adressen dort zu programmieren, um diese später doch noch ändern zu können. Stattdessen wurde die MAC-Adresse als Konstante im Header-File der Board-Settings definiert, wie in Unterkapitel 4.10 erläutert.

Weiterhin notwendig zum Aufbau der Kommunikation ist eine IP-Adresse. Diese kann der lwIP-Stack zum einen dynamisch beziehen, sofern im Netzwerk ein DHCP-Server vorhanden ist. Weil dies im lokalen Labornetz nicht der Fall ist, wurden die IP-Adressen statisch vergeben und ebenfalls in den Board-Settings definiert.

Das Diagramm 4.2 zeigt den Ablauf nach Eintreffen eines TCP-Requests beim lwIP-Stack.

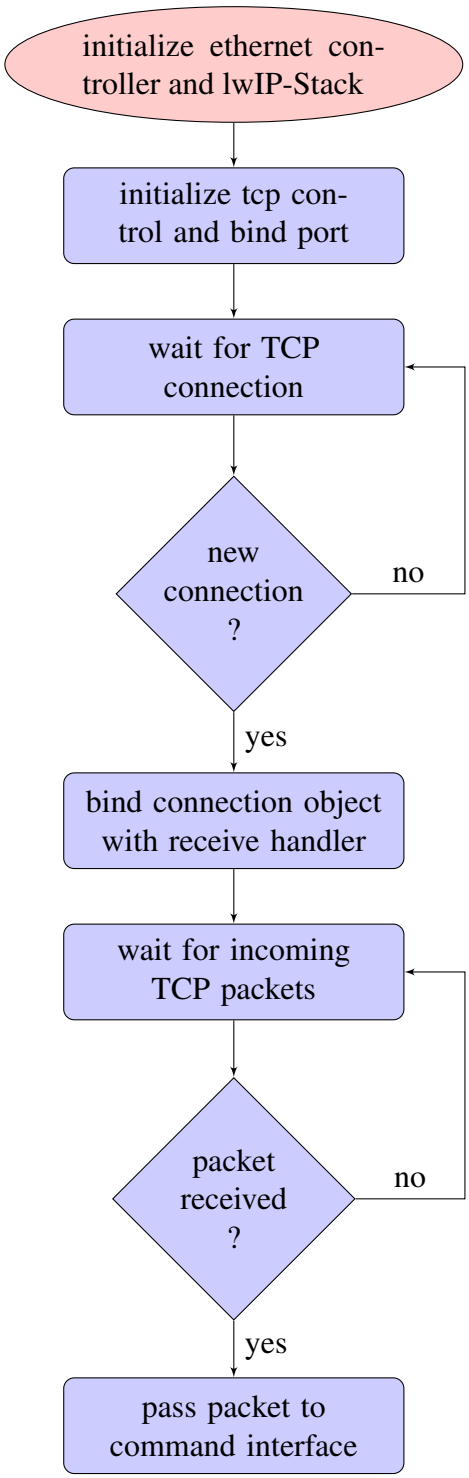


Abbildung 4.2.: Handling einer TCP-Verbindung

4.3. Command-Interface

Eine zentrale Komponente der Controller-Software ist das Command-Interface, welches die Kommunikation mit dem Zellspannungsgenerator steuert. Eine Verbindung kann mit einem beliebigen Telnet-Client hergestellt werden. Während beim Zellspannungsgenerator V01 alle eingehenden ASCII Befehle in einem größeren if-else-Konstrukt ausgewertet und ausgeführt wurden, wurde im Redesign ein Command-Interface mit einer Command-Table und Funktionszeigern zu den ausführenden Funktionen implementiert. Die Funktionen, welche die Ausführung eines Befehls übernehmen, wurden der Übersichtlichkeit halber mit dem Prefix *cmd* versehen und können sich in jedem Source File befinden. Als Vorteil der neuen Implementierung ergibt sich eine bessere Übersichtlichkeit, was zu einer leichteren Wartbarkeit und Erweiterbarkeit des Quellcodes führt.

Während viele Befehle nur aus einem einzelnen Wort bestehen, gibt es auch Kommandos, welche einen oder mehrere Parameter benötigen. Beispielsweise benötigt der Befehl *voltage* zum Setzen der Ausgangsspannung einen Parameter, welcher die zu setzende Spannung in Mikrovolt wiederspiegelt. Um eine beliebige Anzahl an Parametern hinter einem Befehl zu ermöglichen, zerlegt die Routine *commandExecute()*, welche die Befehlsverarbeitung übernimmt, den übergebenen *cmdString* in den eigentlichen Befehl und die dahinter folgenden Parameter. Sofern Parameter vorhanden sind, werden diese in das in der Programmiersprache C übliche Format *argument count(argc)* und *argument vector(argv)* überführt. Dabei enthält *argc* die Anzahl der Parameter und das Array *argv* die Parameter selbst. Die Variablen *argc* und *argv* werden dann der befehlsausführenden Funktion übergeben, welche daraufhin ganz einfach auf die benötigten Parameter zugreifen kann.

Die Routine zur Initialisierung des Command-Interfaces *commandInit()* fügt zunächst alle definierten Kommandos der Command-Table hinzu. Die dazu implementierte Funktion *commandAddCmd()* erwartet als ersten Parameter den Befehlsnamen als String und als zweiten Parameter einen Pointer auf die bei diesem Befehl auszuführende Funktion. Befehl und Funktionszeiger werden dann einem Array bestehend aus Elementen vom Typ *command_table_t* hinzugefügt.

Wird nun nach Abbildung 4.2 ein TCP-Paket empfangen, wird dieses dem Command-Interface mittels der Routine *commandReceive()* übergeben. Diese Routine kennt zwei Empfangs-Modi, zum einen *RECEIVE_BINARY_VALUES* und zum anderen *RECEIVE_CMD*. Standardmäßig befindet sich die Routine im Modus *RECEIVE_CMD*, in dem Befehle samt Parameter ausgewertet werden. In diesem Fall wird das empfangene Datenpaket an die Routine *commandRecvCmd()* übergeben, welche einen Empfangs-FIFO enthält. Dieser ist notwendig, da die Länge eines Datenpaketes beim TCP-Protokoll durch den Sender bestimmt wird. Dies ist in diesem Fall ein PC, welcher anhand seiner Konfiguration sowie Auslastung der Netzwerk-Schnittstelle die Paketgröße bestimmt. Es kann daher nicht vorausgesagt werden, ob ein Befehl samt Parametern in einem oder in mehreren Datenpaketen empfangen wird. Die Funktion *commandRecvCmd()* speichert die empfangenen Daten

daher zuerst im FIFO und übergibt einen Befehls-String erst nach einem erkannten Linebreak an den zuvor beschriebenen Befehlsinterpreter namens *commandExecute()*. Abbildung 4.3 zeigt den Ablauf der Befehlsauswertung. In Tabelle 4.1 sind alle implementierten Kommandos mit einer kurzen Beschreibung aufgeführt.

Der spezielle Befehl *voltagesequence* wechselt in den Binärübertragungsmodus *RECEIVE_BINARY_VALUES*. Alle nun empfangenen Daten werden an die Routine *commandReceiveBinaryValues()* übergeben, welche die Daten binär interpretiert und die Spannungs-Werte mit Hilfe der Routine *sdramWriteVoltage()* im SDRAM speichert. Dieser Modus dient dem Empfang einer Spannungssequenz, welche nun vom PC aus Matlab im Binärformat gesendet werden kann. Das Format der Übertragung wurde in Abschnitt 2.2.5 beschrieben. Um den Binärübertragungsmodus nach Übertragung der Spannungssequenz wieder zu verlassen, muss der Hexadezimalwert *FFFFFF* gesendet werden. Diese Escape-Sequenz kann nicht in der übertragenen Spannungssequenz vorkommen, da der Hexadezimalwert *FFFFFF* einer Spannung von 16,7... V entspricht, was außerhalb des spezifizierten Spannungsbereiches liegt.

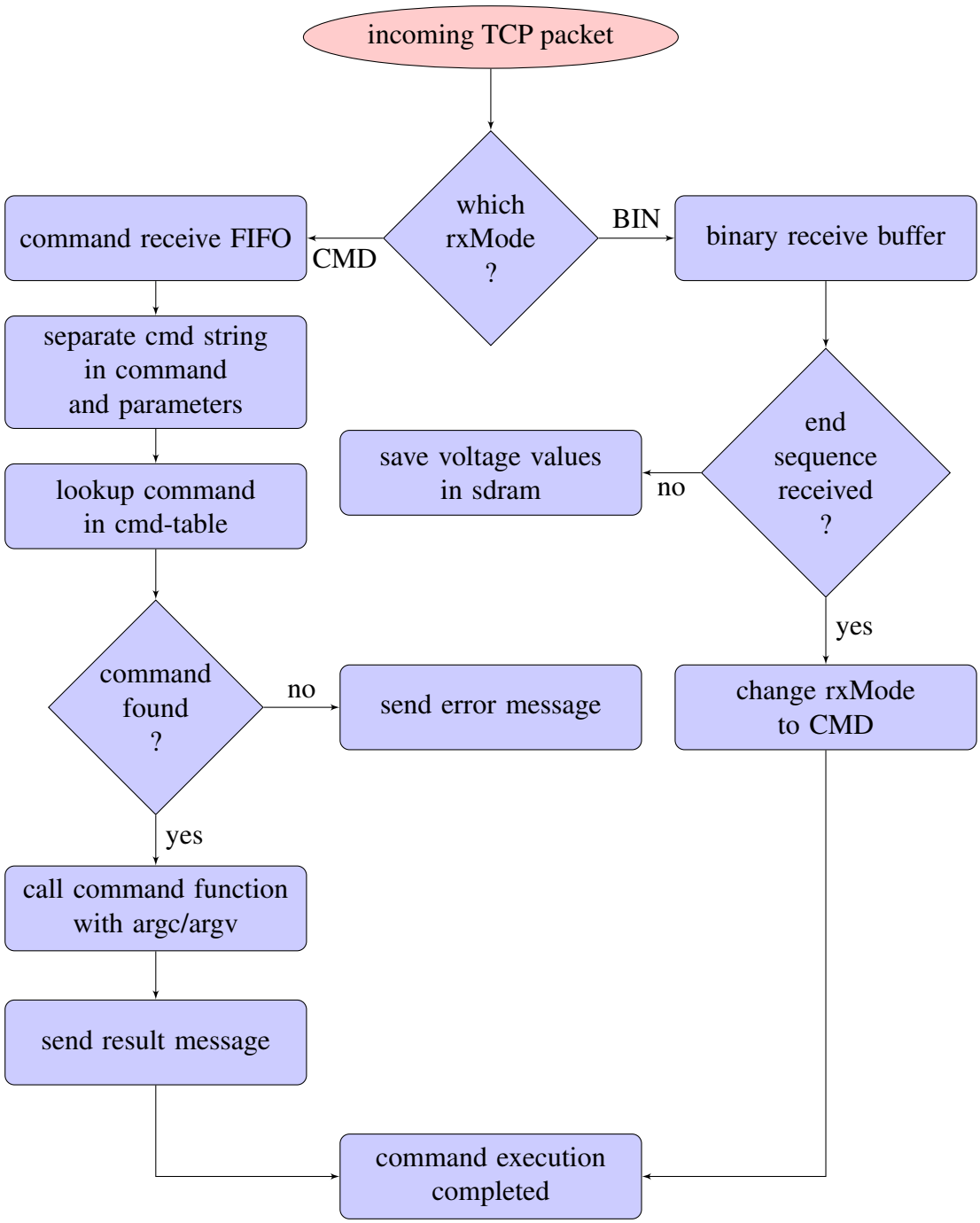


Abbildung 4.3.: Befehlsauswertung des neuen Command-Interfaces

Befehl	Aktion
temp	Abfrage der Board-Temperatur
adcvolt	Gemessene Spannung des ADC abfragen
adcval	16 Bit Wert vom ADC abfragen
sdtest	SDRAM Test durchführen
readpos	Einzelnen Spannungswert abfragen (Parameter: index)
voltagesequence	In den Binärübertragungsmodus wecheln
voltagechecksum	Checksumme der übertragenen Spannungssequenz abfragen
dac	DAC auf 16 Bit Wert setzen (Parameter: dacval)
voltage	DAC auf Ausgangsspannung setzen (Parameter: microvolt)
opon	Ausgang vom Power-OP aktivieren
opoff	Ausgang vom Power-OP deaktivieren
start	Transienten-Wiedergabe starten
stop	Transienten-Wiedergabe stoppen
curon	Shunt-Widerstände aktivieren
curoff	Shunt-Widerstände deaktivieren
bufon	Puffer-Kondensatoren aktivieren
bufoff	Puffer-Kondensatoren deaktvieren
settings	Board-Einstellungen abfragen (optionale Parameter: load, save)
syncmode	Synchronisations-Modus ändern (Parameter: standalone, master, slave)
outputmode	Ausgabe-Modus ändern (Parameter: static, single, repeat)
calibrate	ADC und DAC Kalibrierung starten (Parameter: start, lower, upper)
help	Befehlsübersicht ausgeben
exit	Steuerverbindung beenden

Tabelle 4.1.: Implementierte Befehle des neuen Command-Interfaces

Die Ausführung eines Kommandos vom Terminal wird nach Eingabe des Befehls mit abschließender Enter-Taste gestartet. Befehlsparameter müssen durch ein Leerzeichen getrennt hinter dem Befehl angegeben werden. Die Befehlsfolge zum Setzen der Ausgangsspannung auf 2 V lautet beispielsweise:

voltage 2000000 Enter

4.4. Kalibrierung

Aufgrund der Toleranzen von Referenzspannungsquelle, Spannungsteilern und anderen Bauteilen muss das System kalibriert werden. Für Systeme mit Rückführung, wie auch beim Zellspannungsgenerator, ist es üblich, die Kalibrierung in zwei Schritten durchzuführen. Zuerst werden zwei Punkte des ADC gegen ein externes Referenz-Messgerät abgeglichen, daraus können dann Offset- und Verstärkungsfehler des ADC bestimmt werden. Ab dem Zeitpunkt, wo der ADC kalibriert ist, kann das System selbst die vom DAC erzeugte Ausgangsspannung messen und die Differenz zwischen Soll- und Ist-Wert feststellen. Dieser Prozess kann nun automatisiert werden, indem der spezifizierte Ausgangsspannungsbereich in mehrere Abschnitte unterteilt wird, an dessen unterem und oberem Punkt nun jeweils ein Soll-/Ist-Vergleich zwischen der vom DAC ausgegebenen und vom ADC gemessenen Spannung durchgeführt wird.

Die Spannung vom DAC kann dann entsprechend korrigiert werden, bis der Ist- dem Soll-Wert entspricht. Der tatsächliche 16 Bit DAC-Wert kann dann in Verbindung mit der Ist-Spannung in einer Wertetabelle abgelegt werden. Über das als lineare Interpolation bezeichnete Verfahren kann nun mithilfe von zwei Wertepaaren der Wertetabelle die genaue Ausgangsspannung eingestellt werden. Beim Zellspannungsgenerator V01 wurde das beschriebene Verfahren bereits so umgesetzt, allerdings wurde der komplette Kalibrier-Prozess im Interrupt-Handler des Ethernet-Stacks ausgeführt. Weil das einzelne Anfahren der Spannungsstufen jeweils einige 100 ms dauert, kommt es jedoch regelmäßig zum Timeout der Ethernetverbindung.

Für das Redesign wurde der Kalibrierprozess interruptgesteuert mittels einer State-Machine realisiert. Die Kalibrierung wird in vier Stufen durchgeführt:

1. Rücksetzen der Kalibrierungswerte
2. Abgleich des ADC bei 0,5 V gegen Referenz-Messgerät
3. Abgleich des ADC bei 5,5 V gegen Referenz-Messgerät
4. Automatische DAC Kalibrierung

Die Kalibrierungswerte Offset und Gain vom ADC sowie die Wertetabelle werden in den Board-Settings [Kap. 4.10] persistent gespeichert, damit die Kalibrierungsdaten auch noch nach einem Reset der Baugruppe oder einer Unterbrechung der Spannungsversorgung erhalten bleiben.

In Tabelle 4.2 sind die sechs Zustände der State-Machine zur automatischen DAC Kalibrierung aufgeführt. Abbildung 4.4 zeigt die Transitionen zwischen den einzelnen Zuständen.

Zustand	Aktionen
CALSTART	Startmeldung an Terminal-Client senden
SETVOLTAGE	DAC auf Ausgangsspannung der zu kalibrierenden Stufe setzen
CALIBRATEDAC	Ausgangsspannung um Differenz zwischen Soll- und Ist-Wert anpassen und Fortschrittsmeldung an Terminal-Client senden
SAVEDATA	Ausgangsspannung und DAC-Wert in Wertetabelle speichern
CHECKNEXT	Prüfung durchführen, ob alle Stufen kalibriert sind
CALFINISH	Temporäre Wertetabelle persistent speichern und Fertigmeldung an Terminal Client senden

Tabelle 4.2.: Zustände der State-Machine zur DAC Kalibrierung

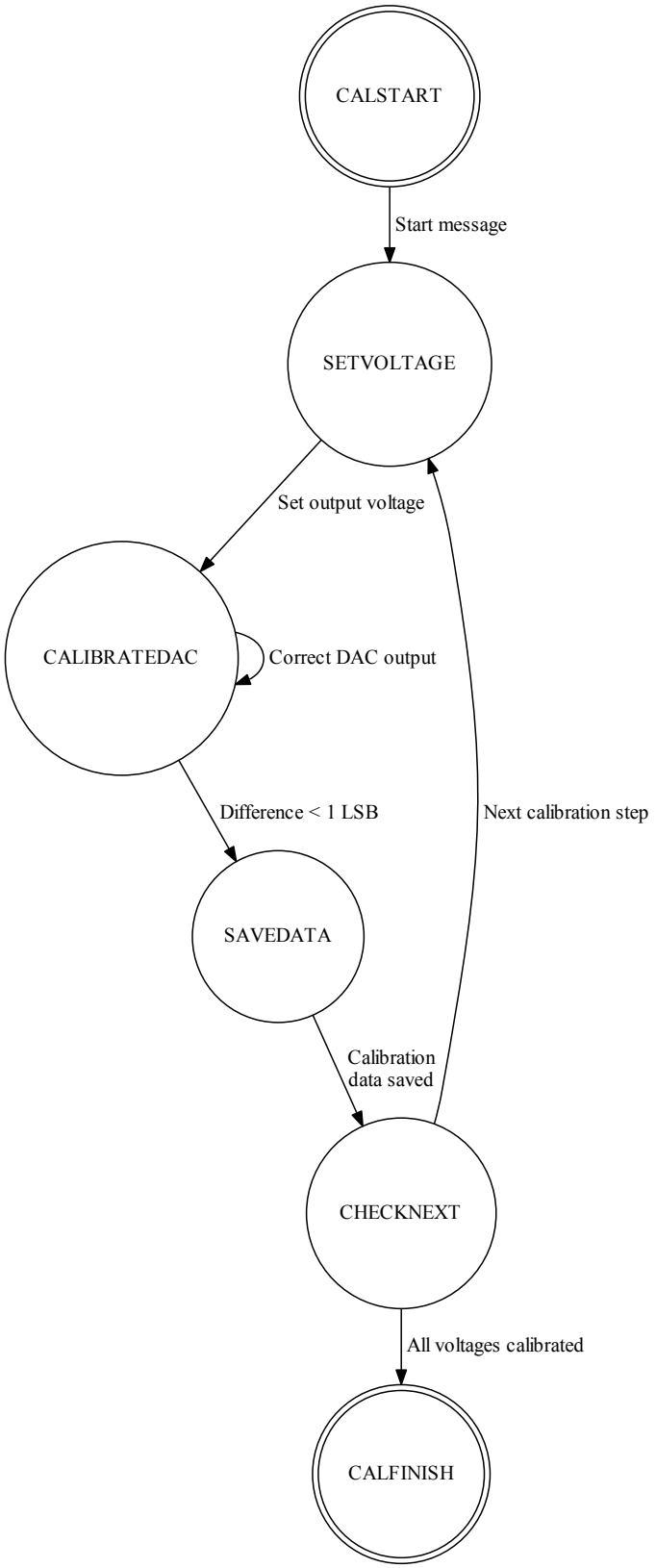


Abbildung 4.4.: State-Machine zur interruptgesteuerten, automatischen DAC Kalibrierung

4.5. SDRAM

Der in Abschnitt 3.2.4 beschriebene SDRAM zur temporären Speicherung der übertragenen Spannungssequenzen wird über die EPI-Schnittstelle des Controllers angebunden. Die Initialisierungs-Routine *sdramInit()* konfiguriert zunächst die benötigten Ports und aktiviert den EPI-Mode. Die StellarisWare-Bibliothek stellt entsprechende Funktionen zur Initialisierung des SDRAMs bereit. Der Speicher wird mit vollem Systemtakt von 50 MHz betrieben. Weiterhin wird der Adressbereich des Speichers in den Adressraum des Controllers gemappt. Das Auftreten von so genanntem *Mirroring*, dem mehrfachen Mapping eines Speicherbereichs an unterschiedliche Adressen, wird automatisch verhindert.

Der Zugriff auf den SDRAM erfolgt mittels Pointer auf eine Adresse im zuvor gemappten Bereich. Auf diese Weise kann selektiv auf jedes 16 Bit-breite Datum zum Lesen und Schreiben zugegriffen werden. Wenn nun eine Spannungssequenz an den Controller gesendet werden soll, wird zunächst mit dem Kommando *voltagesequence* in den Binärübertragungsmodus gewechselt. Die anschließend empfangenen Binärdaten spiegeln die Spannungswerte wieder und werden an die Routine *sdramWriteVoltage()* übergeben. Die empfangenen Werte sind 24 Bit breit, wie in Abschnitt 2.2.5 beschrieben. Es wurde entschieden, die Werte auch mit vollen 24 Bit Genauigkeit zu speichern, auch wenn diese vom DAC nur mit 16 Bit ausgegeben werden können. Dieser Overhead fällt aufgrund der SDRAM-Kapazität von 32 MByte nicht stark ins Gewicht und es besteht auch nach dem Speichervorgang die Möglichkeit, auf die Ursprungswerte und nicht nur auf die gerundeten 16 Bit Werte zuzugreifen. Anwendung finden die 24 Bit Werte derzeit bei der Kalkulation der Checksumme, welche bei der derzeitigen Implementierung nicht nur die Übertragung sondern auch den Schreib- und Lesevorgang des SDRAMs mit einbezieht. Die Abbildung 4.5 zeigt den Ablauf beim Empfang der Spannungswerte.

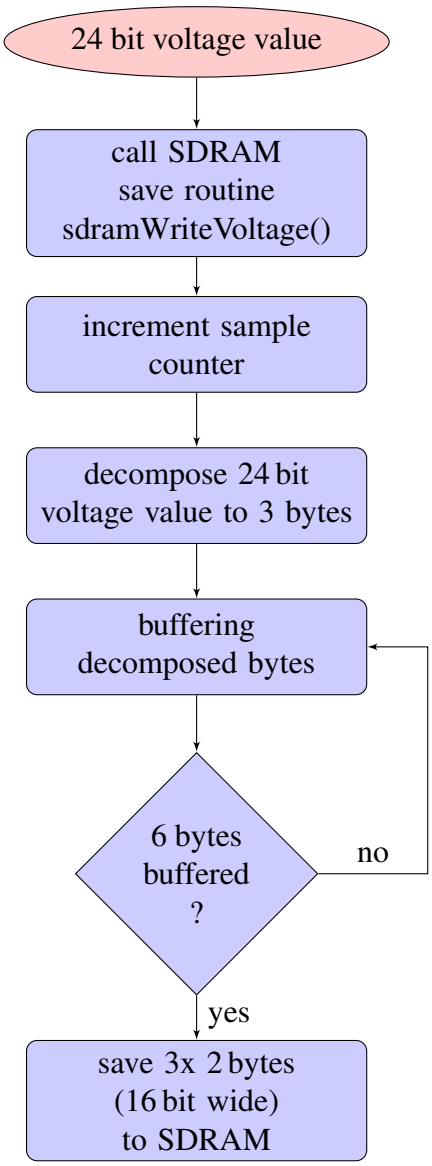


Abbildung 4.5.: Speichern der 24 Bit Spannungswerte im SDRAM

4.6. Synchronisation

Die in Abschnitt 2.2.7 beschriebene Synchronisation ist notwendig, wenn verschiedene oder auch identische Spannungssequenzen parallel über mehrere Module wiedergegeben werden sollen. Dazu wurden drei Synchronisations-Modi implementiert:

1. Master
2. Slave
3. Standalone

Der Modus lässt sich mit dem Kommando *syncmode* und dem entsprechenden Namen des Modus *(master, slave, standalone)* als Parameter einstellen. Um den Synchronbetrieb einzuleiten, muss zunächst ein Modul als Master konfiguriert werden. Wenn die Transienten-Wiedergabe mit dem *start*-Kommando gestartet wird, gibt das Master-Modul den Takt vor und sendet periodisch einen Sync-Befehl auf den RS485-Bus. Die UART-Schnittstelle, über die der RS485-Transceiver kommuniziert, wird mit einer Baudrate von 1 MBaud betrieben. Damit liegt die Übertragungsdauer des 10 Bit langen Sync-Befehls deutlich unterhalb der Wiedergabezeit eines Samples von 50 μs.

Für den Synchronbetrieb müssen die anderen Module zuvor als Slave konfiguriert und ebenfalls mit dem *start*-Kommando auf Empfangsbereitschaft gesetzt worden sein. Anschließend warten die Slave-Module auf einen über den RS485-Bus empfangenen Sync-Befehl und senden daraufhin ein Sample per SPI an den DAC. Um keine Zeitdifferenz aufgrund der Laufzeit des Sync-Befehls über den RS485-Bus zwischen der Sample-Ausgabe des Masters und den Slave-Modulen entstehen zu lassen, empfängt das Master-Modul den selbst gesendeten Sync-Befehl ebenfalls und sendet erst daraufhin ein Sample an den DAC. Die Abbildung 4.6 gibt einen Überblick über den Ablauf nach dem Start der Sequenzwiedergabe. Im Modus *standalone* wird der RS485-Transceiver deaktiviert, woraufhin keine Synchronisation mehr stattfindet. In diesem Modus können die Module komplett unabhängig voneinander betrieben werden.

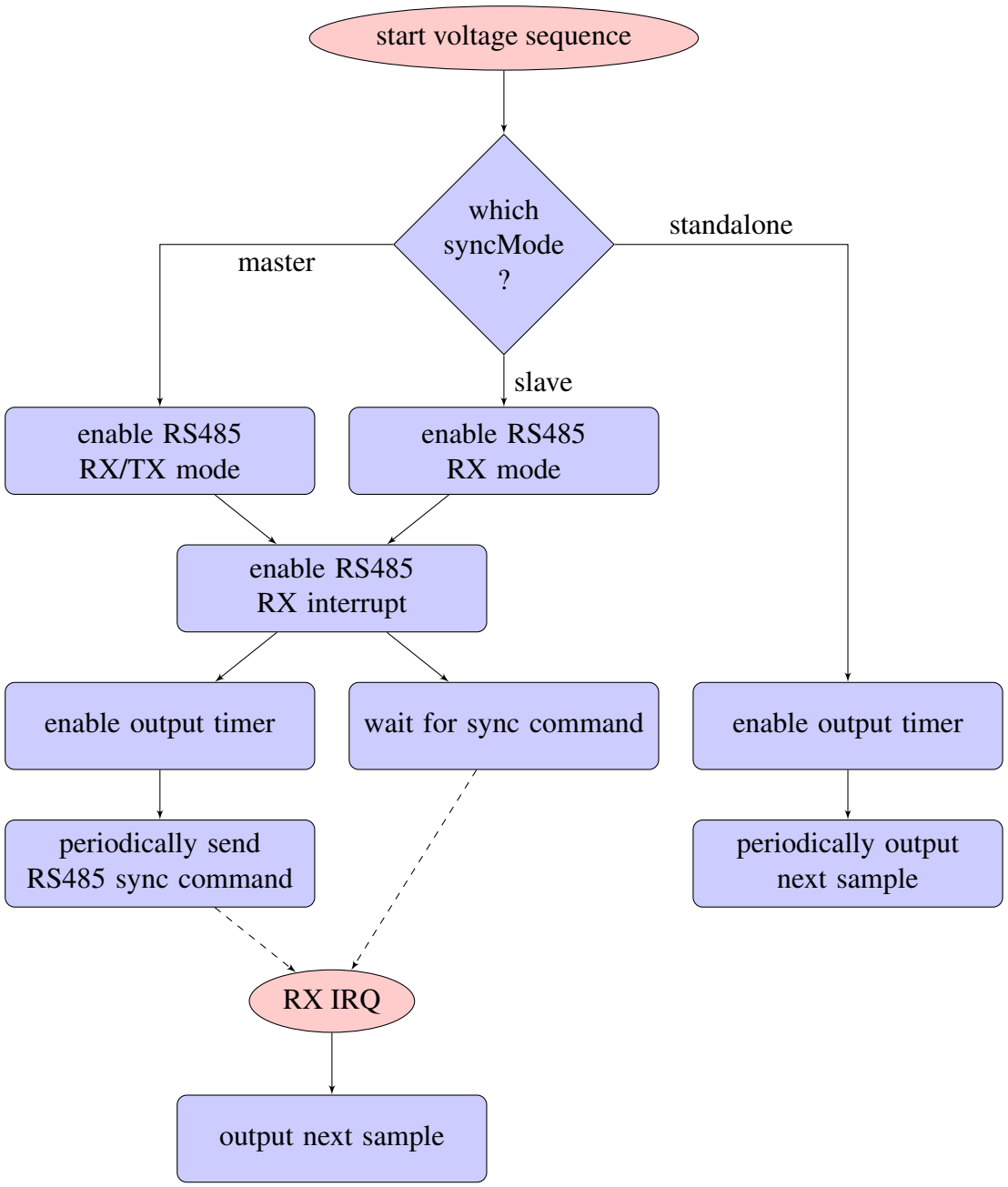


Abbildung 4.6.: Ablauf nach dem Start der Sequenzwiedergabe

4.7. Sequenzwiedergabe

Der Zellspannungsgenerator V02 verfügt über drei Output-Modi, welche mit dem Kommando *outputmode* und dem Namen des Modus als Parameter konfiguriert werden können:

1. Single
2. Repeat
3. Static

Eine zuvor in den SDRAM übertragene Spannungssequenz kann im *Single*-Mode nach Absetzen des *start*-Kommandos einmalig wiedergegeben werden. Im *Repeat*-Mode wird die Wiedergabe solange wiederholt, bis diese mit dem *stop*-Kommando angehalten wird. Die Ausgabe der Samples geschieht mit der in den Anforderungen festgelegten Samplerate von 20 kHz. Die Abbildung 4.6 zeigt den Programmablauf einer Sequenzwiedergabe. In Abbildung 4.7 ist der Ablauf einer Sample-Ausgabe dargestellt.

Im *Static*-Mode ist keine Transienten-Wiedergabe möglich, hier kann mit dem Befehl *voltage* eine feste Ausgangsspannung eingestellt werden. Weiterhin werden automatisch die Puffer-Kondensatoren per Relais parallel zum Ausgang zugeschaltet. Diese lassen sich auch manuell mit den Befehlen *bufon* und *bufoff* aktivieren bzw. deaktivieren.

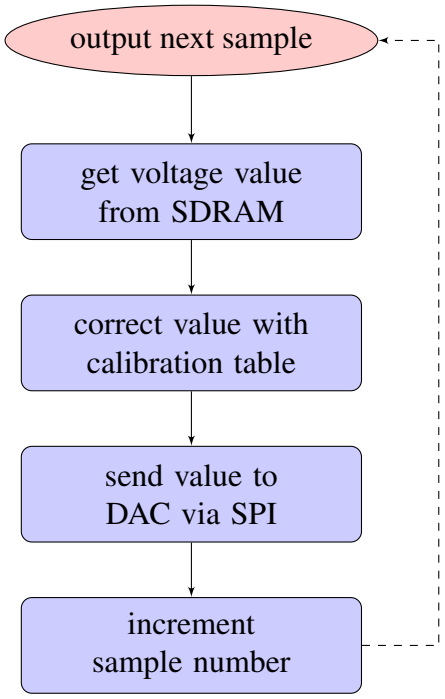


Abbildung 4.7.: Ablauf einer Sample-Ausgabe per SPI an den DAC

4.8. Interrupt-Prioritäten

Die Controller-Software arbeitet komplett interruptgesteuert, im Hauptprogramm findet daher nur die Initialisierung aller Komponenten statt. Die gesamte Befehlsverarbeitung durch das Command-Interface wird vom *SysTick-Interrupt* ausgelöst, welcher durch den lwIP-Stack periodisch den Empfangspuffer der Ethernet-Schnittstelle auf empfangene Datenpakete überprüft.

Das Interrupt Handling wird vom Nested Vectored Interrupt Controller (NVIC) des Mikrocontrollers übernommen, welcher durch acht Prioritäts-Level die unterschiedliche Priorisierung der einzelnen Interrupts erlaubt. Damit der *SysTick-Interrupt* jedoch nicht für Verzögerungen und damit für Jitter des 20 kHz Ausgabetaktes für die Transienten-Wiedergabe sorgt, wurde dieser am niedrigsten priorisiert. Der NVIC erlaubt das sogenannte *Nesting*, worunter eine verschachtelte Ausführung von mehreren Interrupts zu verstehen ist. Tritt ein Interrupt höherer Priorität während der Ausführung eines niedriger priorisierten Interrupts auf, wird dieser unterbrochen und zunächst der höher priorisierte Interrupt-Handler ausgeführt. Anschließend wird die Ausführung des zuvor unterbrochenen Interrupt-Handlers fortgeführt.

Nachfolgend sind die vier konfigurierten Interrupt-Quellen mit ihrer jeweiligen Priorität in absteigender Rangfolge aufgelistet:

1. Output Timer Interrupt

1. RS485 UART Receive Interrupt

2. Calibration Timer Interrupt

3. Ethernet lwIP SysTick Interrupt

Der *Output Timer Interrupt* sowie der *RS485 UART Receive Interrupt* besitzen beide die höchste Priorität. Der einzige Fall, wo beide Interrupts gleichzeitig aktiviert sind, tritt bei dem als Master konfigurierten Modul auf. Hier tritt jedoch immer zuerst der *Output Timer Interrupt* auf, welcher wie in Unterkapitel 4.6 beschrieben, den Sync-Befehl in den UART-Transmit-FIFO schreibt. Anschließend sendet der UART das im Transmit-FIFO enthaltene Zeichen und empfängt es gleichzeitig selbst, woraufhin der *RS485 UART Receive Interrupt* ausgelöst wird.

Der *Calibration Timer Interrupt* ist nur während der automatischen DAC-Kalibrierung aktiv, um periodisch die einzelnen Spannungsstufen anzufahren und anschließend über die Rückführung die Ausgangsspannung zu messen. Die genaue Vorgehensweise wurde in Unterkapitel 4.4 beschrieben.

4.9. ADC/DAC Ansteuerung

Der ADC sowie der DAC werden von der gleichen SPI-Schnittstelle des Mikrocontrollers angesteuert. Da diese Schnittstelle hardwaremäßig nur ein Chip-Select Signal steuern kann, wurde die Chip-Select-Steuerung in Software mittels zwei GPIOs realisiert. Um sicherzustellen, dass die Kommunikation immer nur mit einem der beiden Teilnehmer vom SPI-Bus stattfindet, muss zuvor immer mit der Funktion *spiChipSelect()* der gewünschte Teilnehmer ausgewählt werden. Die Funktion erwartet einen Parameter, welcher der Übersichtlichkeit halber vom Datentyp *Enum* mit den drei Möglichkeiten *DAC*, *ADC* und *NONE* implementiert wurde. Bei der Auswahl eines Bus-Teilnehmers wird automatisch der jeweils andere deaktiviert, bei Auswahl von *NONE* werden beide Teilnehmer deaktiviert.

Das Setzen des DACs auf eine bestimmte Ausgangsspannung sowie die Abfrage der vom ADC gemessenen Spannung ist in Unterkapitel 4.3 beschrieben.

4.10. Board-Settings

Jedes Modul des Zellspannungsgenerators benötigt zum Betrieb verschiedene Einstellungen und Parameter, welche permanent gespeichert werden können und auch ohne Versorgungsspannung erhalten bleiben. Bei dem extern angebundenen SDRAM handelt es sich um einen flüchtigen Speicher, welcher daher für diese Aufgabe nicht in Frage kommt. Weil die Board-Settings nur wenige Bytes in Anspruch nehmen, werden diese daher im Flash des Controllers gespeichert. Der Zugriff auf den 256 kB großen Flash ist normalerweise nur blockweise mit einer Blockgröße von 1 kB möglich. Zum vereinfachten Handling wurde eine *Soft-EEPROM Emulation* nach der Application Note AN01267 [44] von Texas Instruments implementiert. Dadurch ist ein einfacher Lese- und Schreibzugriff auf die als Struct implementierten Board-Settings möglich.

Die Abbildung 4.8 zeigt alle definierten Parameter der globalen Settings-Struktur. Diese beinhaltet die IP-Adresse, MAC-Adresse, die zuvor beschrieben Modi für Synchronbetrieb und des Ausgabeverhaltens sowie die Kalibrierungsdaten. Mit dem Befehl *settings* können die Einstellungen ausgegeben werden, mit den Befehlsparametern *load* und *save* können diese jederzeit nochmals aus dem Flash geladen bzw. in diesen geschrieben werden. Während der Initialisierungsphase des Controllers wird die Settings-Struktur jedoch ohnehin automatisch geladen.

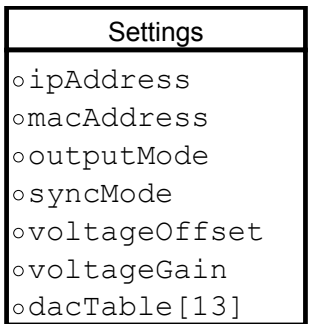


Abbildung 4.8.: Attribute der Board-Settings Struktur

4.11. Strommessung

Die Implementierung der Strommessung konnte aus Zeitgründen nicht mehr realisiert werden. Das angedachte Konzept soll dennoch beschrieben werden.

Zunächst müssen für die drei ADC-Kanäle, welche in Abschnitt 3.2.7 beschrieben wurden, Offset und Gain bestimmt werden. Hier kann analog zur in Unterkapitel 4.4 beschriebenen Kalibrierung des ADCs für die Rückführung vorgegangen werden. Die errechneten Offset- und Gain-Werte können dann ebenfalls in den Board-Settings persistent gespeichert werden. Zur Durchführung einer Strommessung müssen die Shunt-Widerstände durch Öffnen des entsprechenden Relais in den Strompfad eingeschleift werden. Zur Strommessung im stationären Betrieb kann dann ein entsprechender *current*-Befehl dem Command-Interface hinzugefügt werden, welcher eine Werteerfassung der drei ADC-Kanäle auslöst. Anhand der drei Ergebnisse kann der am besten geeignete Messbereich mit der höchsten Genauigkeit ausgewählt werden, das Ergebnis kann dann, um Gain- und Offset-Fehler bereinigt, ausgegeben werden.

Zur Messung des Stromes während der Transienten-Wiedergabe kann auf den beschriebenen Ablauf aufgesetzt werden. Die Werteerfassung der drei ADC-Kanäle und Auswahl des geeigneten Messbereiches muss dazu jedoch im Interrupt-Handler, welcher die Samples per SPI an den DAC sendet, erfolgen. Die Ergebnisse des ausgewählten Messbereiches müssen dann im SDRAM fortlaufend gespeichert werden. Die Implementierung einer entsprechenden *sdramWriteCurrent()*-Routine kann analog zu der in Unterkapitel 4.5 beschriebenen Routine *sdramWriteVoltage()* erfolgen.

Abschließend muss noch sowohl auf Seite des Controllers sowie in Matlab eine Prozedur zur Übertragung der Strom-Werte über die TCP-Verbindung implementiert werden. Hier kann analog zur binären Übertragung der Spannungs-Werte vorgegangen werden, nur dass Sender und Empfänger in diesem Fall vertauscht sind.

4.12. Signalisierung

Auf jedem Modul sind sechs LEDs vorhanden, welche in erster Linie zu Debugging-Zwecken während der Entwicklung dienten. Die Ports zu LED 1 und LED 2 sind auch auf die Frontblende zu einer grünen und einer roten LED geführt. Sofern nach dem Einschalten eines Moduls die Initialisierung der Hardware abgeschlossen ist, leuchtet die grüne LED und signalisiert die Bereitschaft des Moduls. Während der Übertragung einer Spannungssequenz als auch während einer Sequenzwiedergabe blinkt diese LED.

Die rote LED signalisiert analog zum Zellspannungsgenerator V01 einen aufgetretenen Fehler im Programmablauf. Diese LED wird unter anderem beim Auftreten eines nicht gewünschten Interrupts eingeschaltet. Der weitere Programmablauf wird dann gestoppt. Die Tabelle 4.3 gibt einen Überblick über die Belegung der onboard LEDs/Signale.

	Signalisierung
LED 1	Betriebsbereitschaft, blinkt bei Transfer/Wiedergabe
LED 2	Aufgetretener Fehler im Programmablauf
LED 3	Ausgangspuffer aktiv/inaktiv
LED 4	Shunt-Widerstände aktiv/gebrückt
LED 5	RS485 Sync-Interrupt
LED 6	Takt des Output-Timers

Tabelle 4.3.: Signalisierung der onboard LEDs

4.13. Matlab Steuerung

Zur Steuerung des Zellspannungsgenerators per Matlab wurden einige Scripts entwickelt, welche sich in Anhang D befinden. Die Scripts haben das Prefix *vb*, welches als Abkürzung für *Virtual Battery* steht, einem Synonym für den Zellspannungsgenerator. Zur einfachen Auswahl der gewünschten Module wurde eine Klasse namens *vbBoards* mit Konstanten angelegt, denen jeweils die IP-Adresse eines Moduls zugeordnet ist. In allen weiteren Scripts können die Module dann über ihren Namen, beispielsweise als *vbBoards.CELL2*, angesprochen werden.

Um nun eine zuvor aufgezeichnete oder eine synthetisch erzeugte Spannungssequenz an ein Modul des Zellspannungsgenerators zu übertragen, muss im ersten Schritt eine TCP-Verbindung zum gewünschten Modul hergestellt werden und ein TCP-Socket für die weiteren Operationen erzeugt werden. Dazu dient die Funktion *vbConnect()*, welche als Parame-

ter einen oder mehrere der zuvor erläuterten Konstanten erwartet. Nach Öffnen der Verbindungen gibt die Funktion ein Array mit dem Handle des jeweiligen Sockets zurück.

Die zu übertragende Spannungssequenz muss auf Mikrovolt normiert sein. Falls diese noch auf Volt normiert ist, kann eine Umwandlung mit der Funktion *vbDouble2Microvolt()* erfolgen. Anschließend muss das Array der Funktion *vbVoltageTransfer()* zusammen mit dem Connection-Handle übergeben werden. Diese Routine wechselt dann mit dem Befehl *voltagesequence* in den Binärübertragungsmodus, führt die binäre Übertragung der Spannungs-Werte aus dem übergebenen Array durch und beendet die Übertragung durch die $endSequence = 0xFFFFFF$. Matlab sieht standardmäßig die Übertragung eines Bytes als kleinstmögliche Einheit über den TCP-Socket vor. Deshalb musste zur Realisierung der binären Übertragung eine manuelle Zerlegung der in Mikrovolt vorliegenden Spannungs-Werte in jeweils 24 Bit bzw. 3 Byte implementiert werden.

Im letzten Schritt wird mit dem Befehl *voltagechecksum* die Checksumme der übertragenen Werte vom Controller abgefragt und im Matlab Script mit der lokalen Checksumme verglichen. Die Checksumme wird im Controller ermittelt, indem die zuvor im SDRAM gespeicherten Werte wieder ausgelesen werden. Dadurch wird nicht nur die eigentliche TCP-Übertragung auf Fehler überprüft, auch die Fehlerfreiheit des Schreibvorgangs in den SDRAM ist damit gewährleistet.

5. Erprobung

In der Erprobungsphase wurde das Redesign des Zellspannungsgenerators zunächst im stationären Betrieb untersucht. Anschließend wurden die eigentlichen Funktionen der Baugruppe sowie die Wiedergabe von Spannungssequenzen getestet und die dabei aufgetretenen Auffälligkeiten erläutert. Das Redesign wird nachfolgend als Zellspannungsgenerator V02 bezeichnet.

5.1. Stationärer Betrieb

5.1.1. Ripple/Noise des Ausgangssignals

Die durch den DC/DC-Konverter verursachten Spannungsspitzen waren ein zentrales Problem beim Zellspannungsgenerator V01. Durch die Umstellung auf ein Linear-Netzteil sind diese Störungen beim Zellspannungsgenerator V02 nun nicht mehr vorhanden. Die Abbildungen 5.1 und 5.2 zeigen das Ausgangssignal jeweils bei 2 V und bei 5 V ohne eine angeschlossene Last. Das Signal ist frei von Spannungsspitzen, das Rauschlevel liegt bei etwa 5 mV.

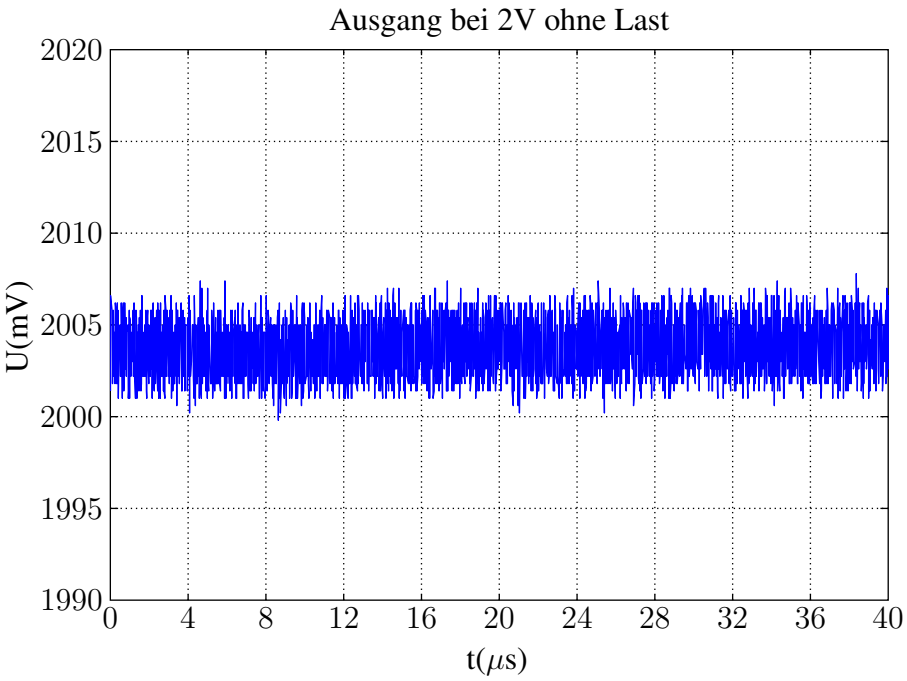


Abbildung 5.1.: 2 V Ausgangsspannung ohne eine angeschlossene Last

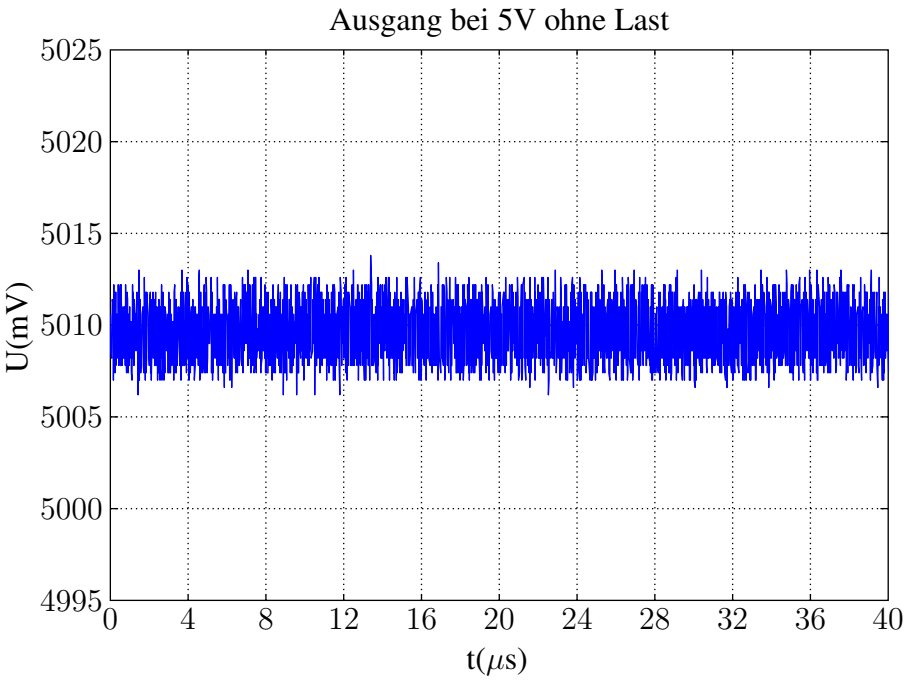


Abbildung 5.2.: 5 V Ausgangsspannung ohne eine angeschlossene Last

Durch die neue Spannungsversorgung konnte das Rauschlevel des Ausgangssignals im Gegensatz zum Zellspannungsgenerator V01 aus Abbildung 2.11 um etwa Faktor 3 gesenkt werden, wie Abbildung 5.3 zeigt. Die Spannungsspitzen konnten durch den Wegfall des DC/DC-Konverters vermieden werden.

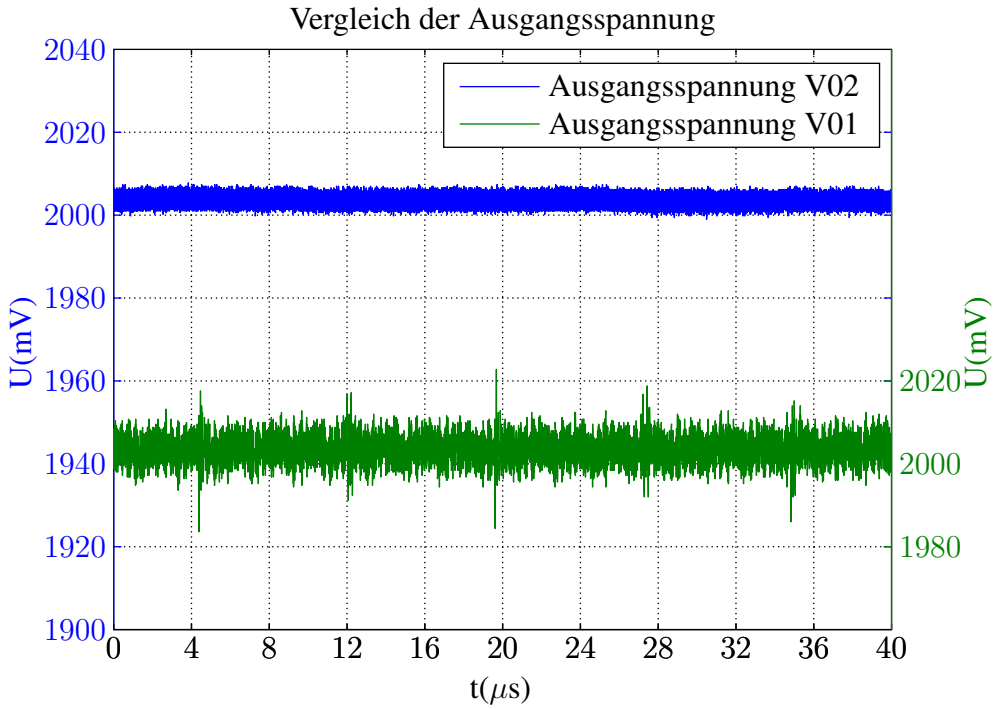


Abbildung 5.3.: Vergleich der Ausgangsspannung vom Zellspannungsgenerator V01 (grün) sowie vom Zellspannungsgenerator V02 (blau) bei 2 V ohne Last

5.1.2. Leiterwiderstand der Strommessung

Es stellte sich heraus, dass der in Abschnitt 3.2.7 beschriebene Leitungspfad der Strommessung einen Leiterwiderstand von einigen Milliohm ausweist, weshalb die Ausgangsspannung bei Belastung einbricht. Bei Belastung des Ausgangs mit einem 10 Ω Lastwiderstand bei einer Ausgangsspannung von 2 V bricht diese um 22 mV ein. Dies deutet auf einen Leiterwiderstand von $R = {}^{22\,\mathrm{mV}}/_{200\,\mathrm{mA}} = 110\,\mathrm{m\Omega}$ hin. Die Leiterbahn durch alle Komponenten der Strommessung hat eine Länge von etwa 80 mm und eine Breite von 60-80 mil. Nach [12, S. 841] entsteht nur durch die Leiterbahn ein Widerstand von $\approx 20\,\mathrm{m\Omega}$. Die verbleibenden $90\,\mathrm{m\Omega}$ werden durch die Relaiskontakte sowie den Hall-Sensor verursacht, wobei der Hall-Sensor laut Datenblatt [2] nur einen Innenwiderstand von $1\,\mathrm{m\Omega}$ zwischen Ein- und Ausgang aufweist.

Eine genauere Analyse war mit dem verwendeten Philips Tischmultimeter vom Typ PM2519 nicht möglich, da dessen minimale Auflösung bei $100\,\mathrm{m\Omega}$ liegt. Es wurde noch

der Versuch unternommen, den Spannungseinbruch durch den Power-OP ausregeln zu lassen, indem dessen Rückführung hinter der Strommessung angeschlossen wurde. Es wurde im Platinenlayout bereits vorgesehen, die Rückführung mit unterschiedlichen Punkten der Ausgangsstufe zu verbinden. Beim Test mit geänderter Rückführung stellte sich jedoch heraus, dass der Power-OP sofort sehr heiss wurde und der Übertemperatur-Schutz aktiv wurde. Dies deutet darauf hin, dass die Rückkopplung ebenfalls sehr niederohmig angebunden sein muss und der hier vorliegende Spannungsabfall deshalb nicht ausgeregelt werden kann.

Um das Problem des Leiterwiderstands anderweitig zu umgehen, wurde beschlossen, ein zweites Paar Anschlussbuchsen in die Frontblende der Module zu integrieren. Die Zellspannungssensoren können dann einerseits, wie ursprünglich vorgesehen, hinter den Komponenten der Strommessung angeschlossen werden. Sofern für die geplante Anwendung eine maximale Genauigkeit der Ausgangsspannung erforderlich ist, kann der Sensor alternativ über das zweite Buchsenpaar unmittelbar hinter dem Power-OP angeschlossen werden.

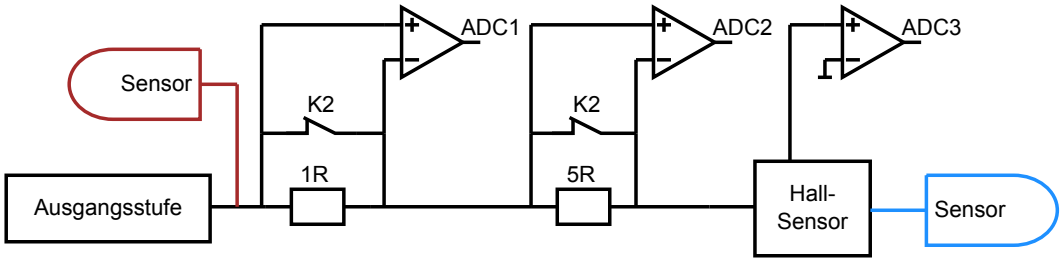


Abbildung 5.4.: Zwei Ausgänge des Zellspannungsgenerators V02: Ausgang 1 (rot) direkt hinter der Ausgangsstufe, Ausgang 2 (blau) hinter der Strommessung

5.1.3. Einkopplung in Zellspannungsgenerator V02

Wie zuvor in Abschnitt 5.1.2 erläutert, besitzt der Zellspannungsgenerator V02 nun zwei Ausgangspunkte, um einen Sensor anzuschließen. Im Folgenden werden die Einkopplungen durch einen Klasse 1 Sensor an beiden Ausgängen untersucht.

Klasse 1 Sensor hinter der Strommessung (Ausgang 2)

Die Abbildung 5.5 zeigt die Einkopplung des Schaltreglers eines Klasse 1 Sensors in den Ausgang hinter der Strommessung. Die Spikes treten mit einer maximalen Amplitude von 20 mV auf und liegen damit um Faktor 4 geringer als beim Zellspannungsgenerator V01 in Abbildung 2.4. Das Rauschen liegt bei etwa 8 mV im Gegensatz zu den 30 mV in Abbildung 2.4. Die Abbildung 5.6 zeigt den Zeitpunkt der Messwerterfassung. Nach dem Shutdown des Schaltreglers gibt es Unter-/Überschwinger mit knapp 40 mV Amplitude, welche im

Vergleich zu Abbildung 2.5 um Faktor 5 geringer ausfallen. Es dauert etwa 30 μs, bis sich die Spannung wieder stabilisiert hat und der ADC des Sensors einen korrekten Messwert erfassen kann.

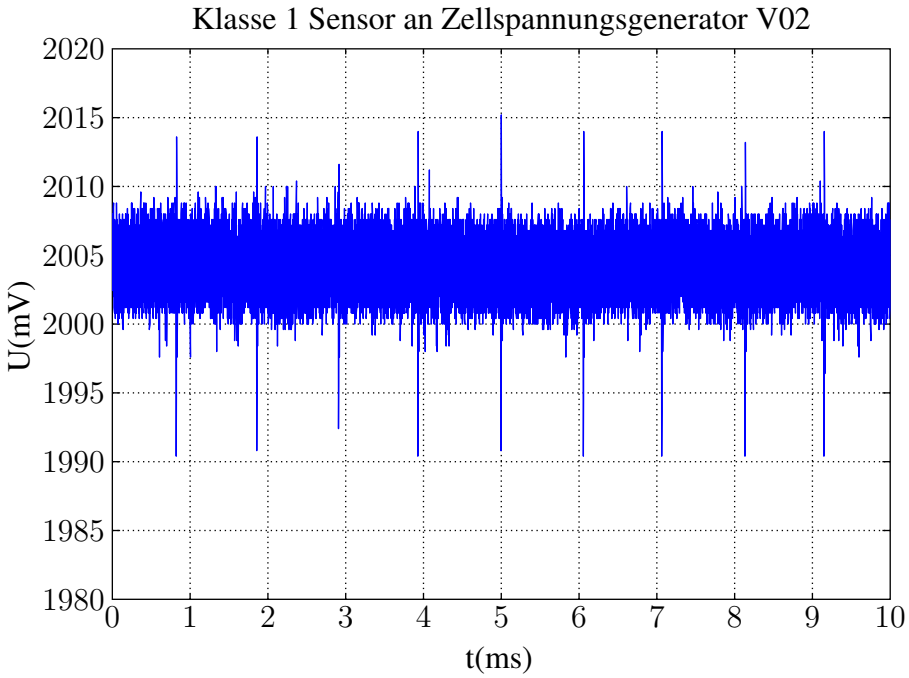


Abbildung 5.5.: Einkopplung in die Ausgangsspannung vom Zellspannungsgenerator V02 durch angeschlossenen Klasse 1 Sensor an Ausgang 2 hinter der Strommessung

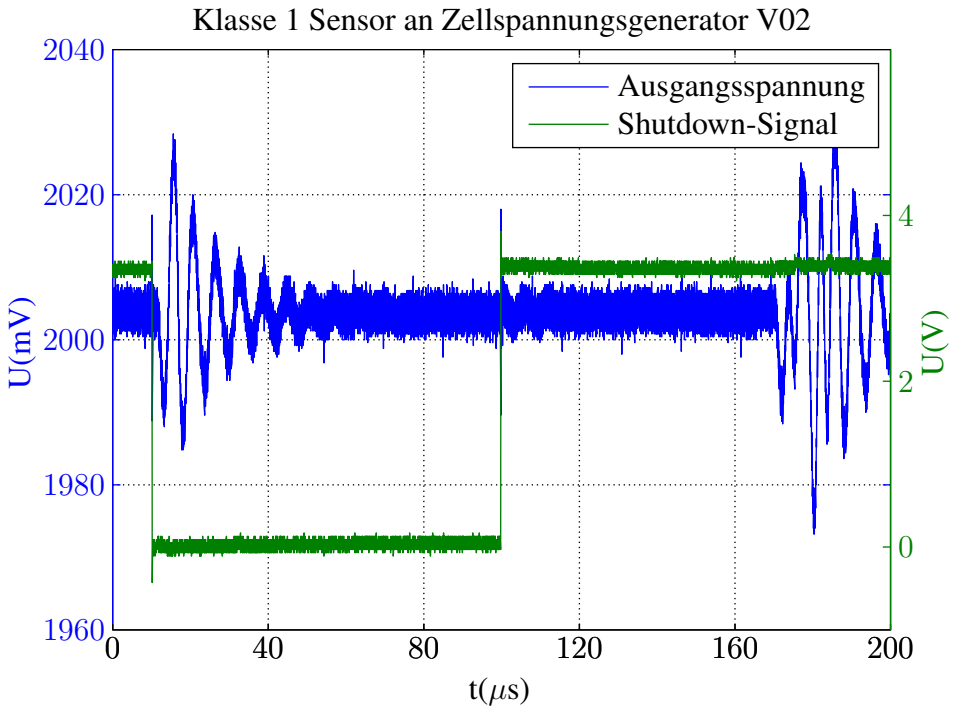


Abbildung 5.6.: Einkopplung eines Klasse 1 Sensors in die Ausgangsspannung vom Zellspannungsgenerator V02 (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals), angeschlossen an Ausgang 2 hinter der Strommessung

Klasse 1 Sensor hinter Power-OP (Ausgang 1)

Für nachfolgende Messungen wurde der Klasse 1 Sensor am zusätzlichen Ausgang des Generators und damit direkt hinter dem Power-OP angeschlossen. Die Messung in Abbildung 5.7 zeigt mit knapp 20 μs eine schnellere Stabilisierung der Ausgangsspannung nach dem Abschaltvorgang des Schaltreglers. Weiterhin gibt es nur einen großen Unter-/Überschwinger mit knapp 20 mV Amplitude.

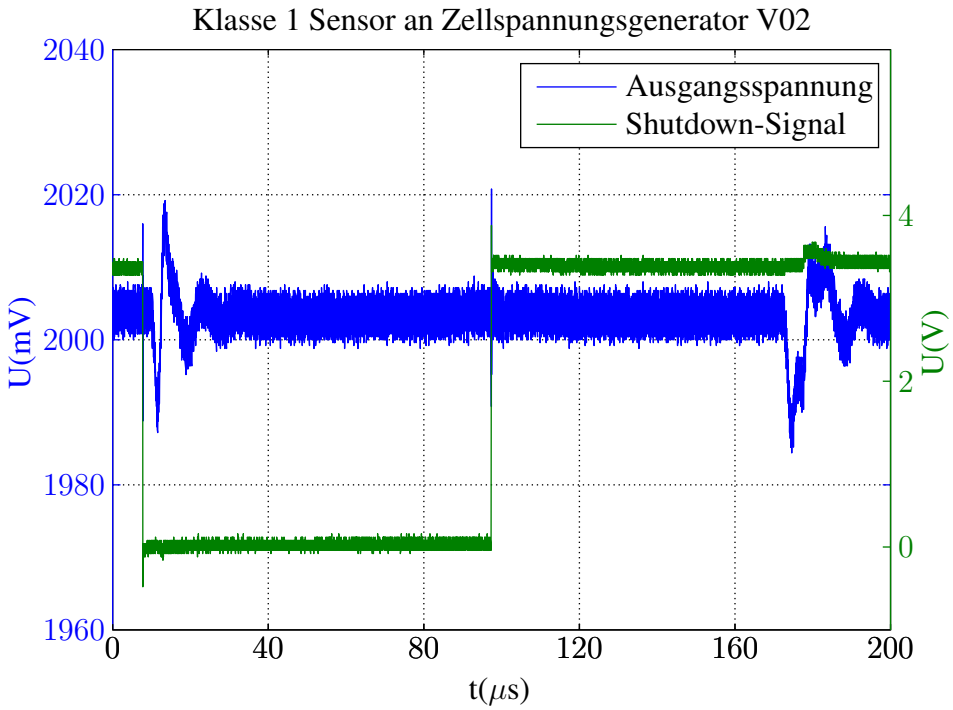


Abbildung 5.7.: Einkopplung eines Klasse 1 Sensors in die Ausgangsspannung vom Zellspannungsgenerator V02 (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals), angeschlossen an Ausgang 1 hinter dem Power-OP mit deaktivierter Pufferung

In Abbildung 5.8 ist nochmals der Zeitpunkt der Messwerterfassung dargestellt, jedoch mit aktivierter Pufferung, wie in Abschnitt 3.2.6 beschrieben. Es zeigt sich eine nochmals geringere Amplitude der Unter-/Überschwinger mit nur noch etwa 15 mV und eine noch kürzere Dauer von nur noch knapp 15 μs bis zur Stabilisierung der Ausgangsspannung.

In Abbildung 5.9 ist ein Vergleich der Ausgangsspannungen vom Zellspannungsgenerator V01 und vom Zellspannungsgenerator V02 am Ausgang hinter dem Power-OP mit aktivierter Pufferung dargestellt. Das Ausgangssignal vom Zellspannungsgenerator V02 enthält keine zyklischen Spannungspitzen, stabilisiert sich wesentlich schneller nach dem Schaltvorgang des Schaltreglers auf dem Sensor und die Amplituden der Einkopplungen fallen deutlich geringer aus.

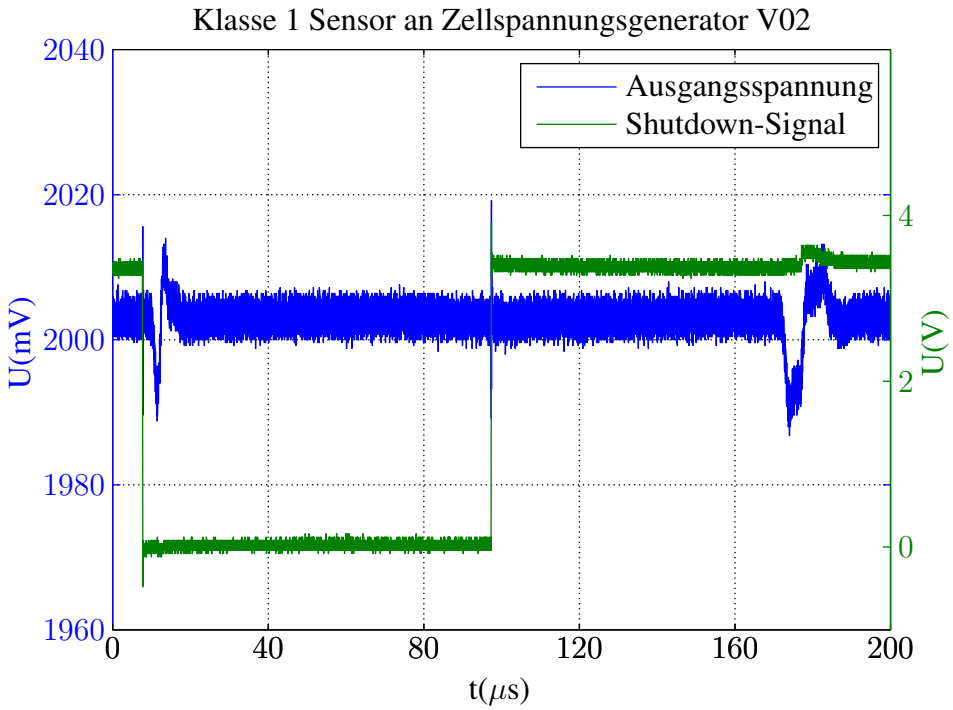


Abbildung 5.8.: Einkopplung eines Klasse 1 Sensors in die Ausgangsspannung vom Zellspannungsgenerator V02 (blau) im Moment der Messwerterfassung (gekennzeichnet durch Low-Pegel des grünen Shutdown-Signals), angeschlossen an Ausgang 1 hinter dem Power-OP mit aktivierter Pufferung

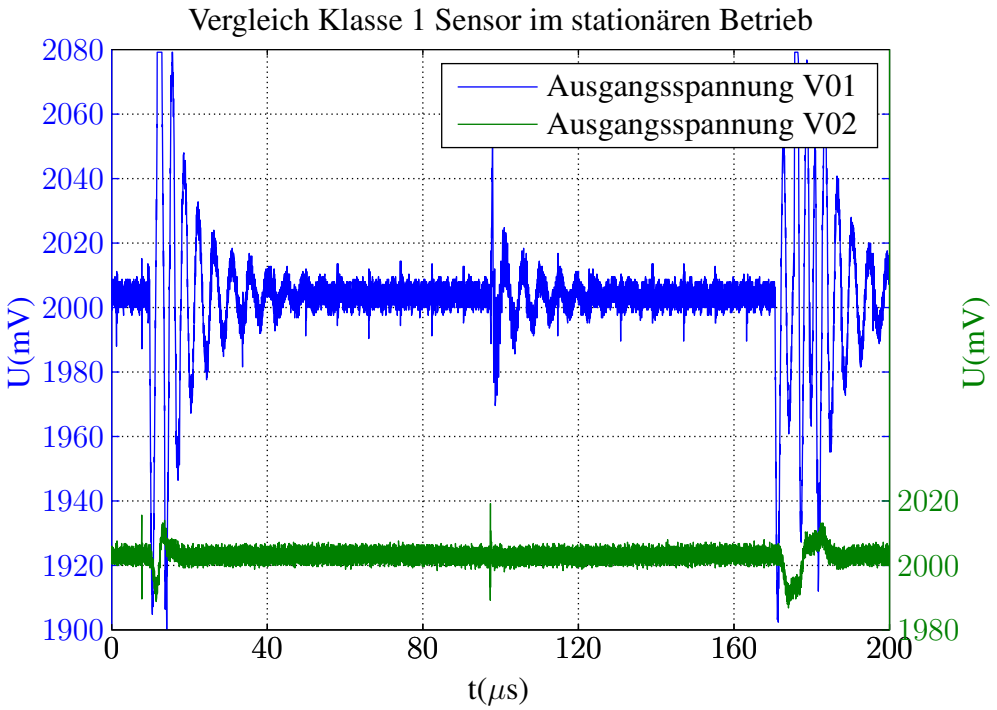


Abbildung 5.9.: Vergleich der Einkopplungen eines Klasse 1 Sensors in die Ausgangsspannung vom Zellspannungsgenerator V01 (blau) sowie vom Zellspannungsgenerator V02 mit aktiviertem Ausgangspuffer (grün)

5.2. Kalibrierung und Linearität

Analog zu [35, Kap. 6.2] wurde die Kalibrierung sowie die Linearität eines Moduls vom Zellspannungsgenerator V02 bei $RT = 23\,^{\circ}\mathrm{C}$ nach mehrstündiger Betriebszeit überprüft. Zunächst wurde der in Abschnitt 4.4 beschriebene Kalibrierprozess durchgeführt und der untere Punkt (500 mV) sowie der obere Punkt (5,5 V) des spezifizierten Spannungsbereiches gegen das Tischmultimeter PM2519 abgeglichen. Daraus konnte das System den Offset und Gain vom ADC bestimmen und anschließend selbsttätig die Kalibrierung des DACs durchführen.

Zur Prüfung des Ergebnisses vom Kalibrierprozess wurden über den *voltage*-Befehl Ausgangspannungen im spezifizierten Bereich von 500 mV bis 5,5 V in Schritten von 250 mV eingestellt. Die Ausgangspannungen wurden mit dem Tischmultimeter PM2519 gemessen und in Tabelle 5.1 dargestellt. Es zeigt sich ein sehr lineares Verhalten trotz der Kalibrierung des ADCs in nur zwei Punkten. In den Tabellenzellen mit zwei Werten zeigte das Tischmultimeter, welches im Messbereich ab 1 V nur noch über eine Auflösung von 1 mV verfügt,

abwechselnd einen der Werte, der reale Wert liegt demnach dazwischen. Die Abweichungen liegen alle bei maximal 1 mV und damit unterhalb der spezifizierten 1,22 mV.

Soll-Wert (in mV)	Ist-Wert (in mV)
500	500
750	750
1000	1000.1
1250	1250
1500	1500
1750	1750
2000	2000
2250	2250
2500	2500
2750	2750
3000	3000
3250	3250
3500	3500
3750	3750
4000	4000/4001
4250	4250/4251
4500	4500
4750	4750
5000	5000/5001
5250	5250/5251
5550	5550/5551

Tabelle 5.1.: Vergleich zwischen Soll- und Ist-Spannung nach Kalibrierprozess mit Philips Tischmultimeter PM2519

5.3. Ground Bouncing bei Last

Das Einbrechen der Ausgangsspannung bei Belastung des Ausgangs hinter der Strommessung wurde in Abschnitt 5.1.2 bereits beschrieben. Es wurde ebenfalls die Stabilität des zweiten Ausgangs hinter dem Power-OP getestet. Die eingestellte Ausgangsspannung von 2 V wurde auch hier mit einem 10 Ω Lastwiderstand und dadurch mit 200 mA belastet. Dabei wurde festgestellt, dass die Ausgangsspannung um etwa 2-3 mV einbricht. Weiterhin konnte festgestellt werden, dass sich das Ground-Potential der Analog-Versorgung an

ADC und DAC gegenüber dem Ground-Potential hinter den Linearreglern um wenige Millivolt anhebt. Einige Versuche, das Ground-Potential testweise auf kürzerem Weg zu verbinden, konnten diesen Effekt jedoch nicht minimieren. Hierzu können noch Versuche unternommen werden, die verschiedenen Masseflächen der Platine testweise an verschiedenen Punkten niederohmig miteinander zu verbinden. Versuche dieser Art konnten jedoch aus Zeitgründen nicht mehr durchgeführt werden. Dennoch fällt der Spannungsabfall bei der genannten Belastung geringer aus, als beim Zellspannungsgenerator V01, dessen Ausgang nach [35, Abschn. 6.2.2] bei identischer Belastung um 10 mV einbricht.

5.4. Sequenzwiedergabe

Neben dem Kalibriermodus sollen mit dem Zellspannungsgenerator aufgezeichnete sowie synthetisch erzeugte Spannungssequenzen wiedergegeben werden können. Der Test einer Transienten-Wiedergabe wurde mit einem aufgezeichneten Spannungsverlauf durchgeführt, gemessen an einer Lithiumzelle während des Startvorgangs eines Audi A4 [11]. Die Spannungssequenz lag mit einer Abtastrate von 100 kHz vor und musste zunächst auf 20 kHz downgesampled werden, bevor diese zum Zellspannungsgenerator V02 übertragen werden konnte. Bei der Übertragung wurde der Vorteil des eingesetzten SDRAMs deutlich sichtbar. Die Übertragungsdauer der Spannungssequenz mit 200000 Werten dauert durch den schnellen Speicher in Verbindung mit den in Abschnitt 2.2.5 durchgeführten Optimierungen nur noch etwa 3 s und damit nur noch etwa 10 % der ursprünglichen Zeit vom Zellspannungsgenerator V01, wie Abbildung 5.10 zeigt.

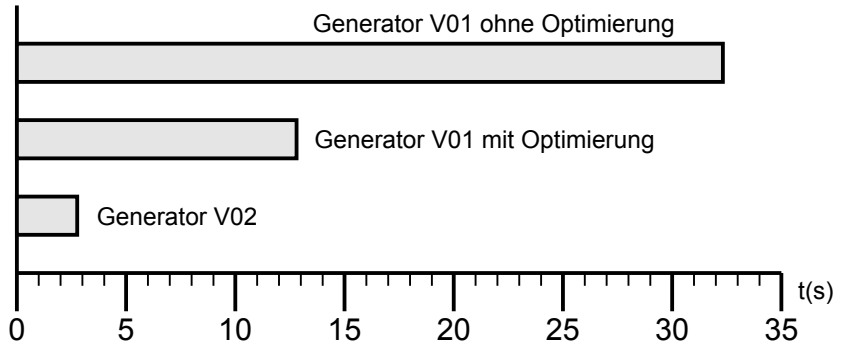


Abbildung 5.10.: Vergleich der Übertragungsdauer einer Spannungssequenz mit 200000 Werten zwischen dem Zellspannungsgenerator V01 und V02

Während der Transienten-Wiedergabe wurde die Ausgangsspannung des Generatos wieder mit dem Oszilloskop vom Typ MSO3034 mit einer Abtastrate von 100 kHz aufgezeichnet. Die Abbildung 5.11 zeigt die ausgegebene, und die gemessene Spannungssequenz sowie

deren Differenz. Die Darstellung der gemessenen Sequenz wurde dabei um ein zeitliches Offset von wenigen Samples verschoben, welcher durch die Triggerung des Oszilloskops entstanden ist. Es zeigt sich eine sehr gute zeitliche Übereinstimmung, welche insbesondere beim Start des Hochstromereignisses sichtbar wird. Auffällig ist jedoch ein nahezu konstanter Offset von etwa 25 mV, welcher sich durch die Messungenauigkeiten zwischen Oszilloskop und Tischmultimeter erklären lässt, wie anfangs in Kapitel 2 sowie in [35, Abschnitt 6.3.3] erläutert.

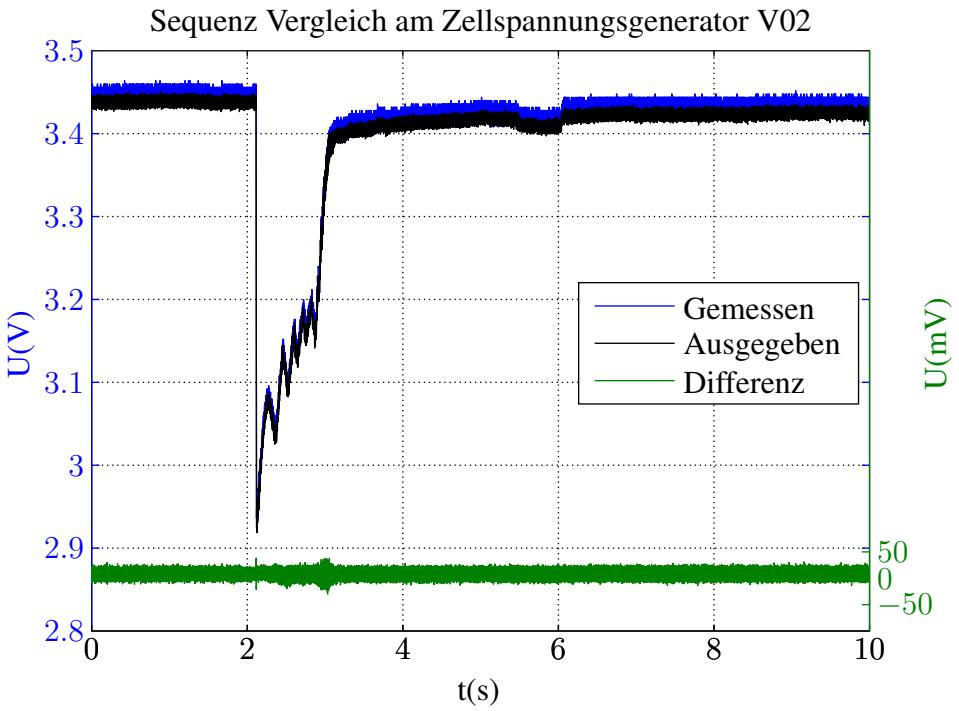


Abbildung 5.11.: Vergleich zwischen wiedergegebener Spannungssequenz (schwarz) und mit Oszilloskop gemessener Sequenz (blau). Im unteren Bereich (grün) ist die Differenz dargestellt.

5.5. Synchronisation

Die in den Abschnitten 3.2.9 und 4.6 beschriebene Synchronisation wurde anhand eines als Master konfigurierten Moduls getestet. Bei der Transienten-Wiedergabe wird der Sync-Befehl mit einer Übertragungrate von 1 MBaud auf den RS485-Bus gesendet. Das Master Modul empfängt ebenfalls den eigens ausgesendeten Sync-Befehl und sendet daraufhin ein Sample per SPI an den DAC. In Abbildung 5.12 ist der empfangene Sync-Befehl sowie der daraufhin ausgelöste Interrupt durch ein Rechtecksignal dargestellt. In diesem alle 50 μs auftretenden Interrupt wird jeweils ein Sample an den DAC gesendet.

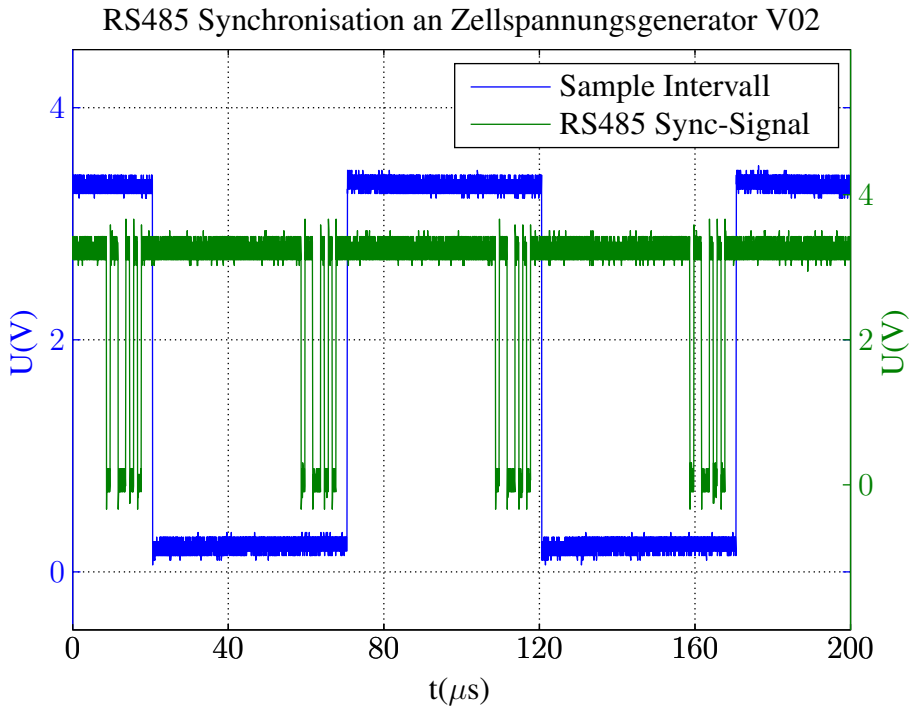


Abbildung 5.12.: Der grüne Spannungsverlauf zeigt das RS485 Sync-Signal, das blaue Rechteck-Signal wurde in jedem auftretenden Sync-Interrupt getoggelt.

5.6. Temperaturmessung

In Abbildung 5.13 sind die Ergebnisse einer Temperaturmessung mittels des in Abschnitt 3.2.10 beschriebenen Temperatursensors dargestellt. Die Messungen fanden bei einer Raumtemperatur von $RT = 23\,^{\circ}\mathrm{C}$ statt. Alle fünf Minuten wurde eine Messung mit dem *temp*-Befehl durchgeführt. Es zeigt sich, dass sich die Temperatur nach 65 Minuten bei $\approx 31,5\,^{\circ}\mathrm{C}$ stabilisiert und das Modul dann seine Betriebstemperatur erreicht hat. Während dieser Zeit erwärmen sich verschiedene Komponenten wie Trafo, Linearregler, Controller und geben ihre Wärme an die Platine ab, welche sich dann vor allem über die Masseflächen ausbreitet. Die beiden nach 20 und nach 40 Minuten durchgeführten Messungen zeigen einen Ausreißer von $\approx 0,5\,^{\circ}\mathrm{C}$, was damit der Genauigkeitsgrenze des Temperatursensors entspricht.

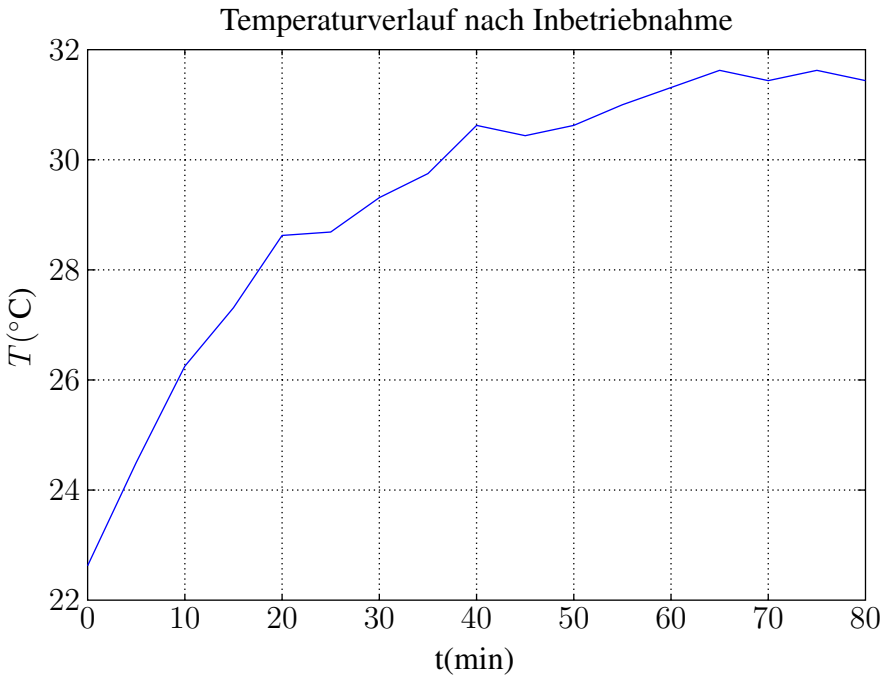


Abbildung 5.13.: Temperaturverlauf nach dem Einschalten bei $RT = 23\,^{\circ}\mathrm{C}$

5.7. Hohe Abwärme eines LDOs

Während des Betriebs stellte sich heraus, dass der für die digitale Versorgungsschiene vorgesehene LDO vom Typ LT1760 seine Verlustleistung von etwa 2 W nicht ausreichend an die Platine abgeben kann. Dies führte zu einer Überhitzung des Reglers, woraufhin dieser seine Ausgangsspannung absenkt. Aufgrund des kleinen Gehäuses vom Typ SOIC-8 und der dichten Bestückung gestaltete sich die Montage eines Kühlkörpers als schwierig.

Aufgrund dessen wurde auf einen anderen LDO vom Typ LM1084 der Firma National Semiconductor [36] im TO-220 Gehäuse ausgewichen. Dieser wurde auf die vorhandenen Pads gelötet und mit einem Kühlkörper im U-Profil versehen.

5.8. Layoutfehler

Bei der Bestückung stellten sich zwei kleine Fehler im Layout bzw. im zuvor erstellten Schaltplan heraus. Zum einen existiert eine Verbindung zwischen dem negativen Eingang der ADCs und einem Masse-Potential des Boards, welche eine differentielle Messung verhindert. Zum anderen wurden bei dem Relais zum Zuschalten des Ausgangspuffers zwei Kontakte vertauscht. Beide Fehler konnten jedoch durch Auftrennen einer Leiterbahn behoben werden.

6. Fazit

Im Folgenden soll das Ergebnis dieser Bachelorthesis betrachtet werden. Ebenso soll ein Ausblick gegeben werden, welche Punkte dieser Arbeit noch einmal zur Fortführung oder Optimierung aufgegriffen werden können.

6.1. Bewertung

Ein wichtiges Ziel war es, mit dem Redesign des Zellspannungsgenerators eine störungsarme Spannungsversorgung zu realisieren. Diese Anforderung konnte mit dem jetzt eingesetzten Linear-Netzteil erfüllt werden, wie die Erprobung im Unterkapitel 5.1 gezeigt hat. Das Ausgangssignal enthält weniger Noise und keine Spikes mehr, wie durch den zuvor eingesetzten DC/DC-Konverter verursacht. Die Vorteile des deutlich stabileren Linear-Netzteils, welches weiterhin die doppelte Leistung gegenüber dem zuvor eingesetzten DC/DC-Konverter bereitstellen kann, wiegen den Nachteil der höheren Verlustleistung wieder auf.

Mit dem Redesign wurde ein komplett neues Platinenlayout entwickelt. In dieses wurde der Mikrocontroller mit seiner notwendigen Peripherie direkt integriert, wodurch sämtliche in Abschnitt 3.2.2 beschriebenen Nachteile des zuvor eingesetzten Evaluations-Boards entfallen sind. Die galvanische Trennung der Ethernet-Buchse konnte damit direkt im Layout vorgesehen werden. Ebenso konnte der in Abschnitt 3.2.4 erläuterte SDRAM direkt über die EPI-Schnittstelle an den Mikrocontroller angebunden werden. Durch den neuen parallel angebundenen Speicher wurde die Übertragungszeit einer Spannungssequenz deutlich reduziert.

Die hohe Abwärme des LDOs zur Versorgung der digitalen Komponenten konnte durch den Tausch gegen einen anderen Typ und anderes Gehäuse sowie durch Montage eines Kühlkörpers gelöst werden. Die zwei kleinen Layoutfehler konnten durch Auftrennen zweier Leiterbahnen ebenfalls behoben werden. Sollten weitere Module des Zellspannungsgenerators gefertigt werden, sollten die beiden Leiterbahnen im Layout zunächst korrigiert und der Footprint vom LDO angepasst werden.

Die alternative Lösung zur Synchronisation der einzelnen Module per RS485-Bus konnte mit dem in Abschnitt 3.2.9 beschriebenen RS485-Transceiver mit integrierter galvanischer

Trennung umgesetzt werden. Aufgrund der besseren Robustheit der differentiellen Übertragung der Sync-Befehle ist diese Lösung zu bevorzugen.

Mit der Auswahl und Beschaffung des 19“ Sub-Racks inklusive Zubehör (Frontplatten, Anschlussbuchsen, etc.) steht der Aufbau eines Komplettgerätes kurz bevor.

Das Schaltungskonzept der Strommessung wurde überarbeitet und zusätzlich um einen Hall-Sensor erweitert, wie in Abschnitt 3.2.7 erläutert. Mit diesem Konzept ist eine rauscharme Strommessung durch die niedrigeren Gain-Faktoren der Instrumentenverstärker zu erwarten. Die in Unterabschnitt 4.11 erläuterte Implementierung der Strommessung in der Controller-Software konnte aus Zeitgründen nicht mehr umgesetzt werden. Das in Abschnitt 5.1.2 erläuterte Problem des Leiterwiderstands der Strommessung wurde durch den Einbau eines zweiten Ausgangs mit Abgriff des Signals direkt hinter dem Power-OP gelöst.

Mit dem zusätzlichen Ausgang existieren nun verschiedene Betriebsmodi. Zur Kalibrierung im stationären Betrieb empfiehlt sich der Anschluss des Sensors an den zusätzlichen Ausgang direkt hinter dem Power-OP. Zur Untersuchung der Stromaufnahme muss der Sensor an den Ausgang hinter der Strommessung angeschlossen werden. Im Transienten-Betrieb kann der Sensor je nach Ziel des Versuchs, ob eine hohe Genauigkeit oder die Erfassung der Stromaufnahme gefordert ist, an den entsprechenden Ausgang angeschlossen werden. Es hat sich gezeigt, dass die gleichzeitige Strommessung im Hochgenauigkeitsmodus mit dem entwickelten Hardware-Konzept nicht möglich ist.

Die Übermittlung der Spannungssequenzen wurde von ASCII-Kodierung auf eine binäre Übertragung umgestellt, wie in Abschnitt 2.2.5 erläutert. Zusammen mit einer verbesserten Implementierung des Empfangs-FIFOs sowie dem Einsatz eines SDRAMs konnte die Übertragungsdauer einer Spannungssequenz auf etwa 10 % reduziert werden. Die implementierte Prüfsumme sichert sowohl die binäre Übertragung als auch den Schreibvorgang in den SDRAM ab.

Das in Unterkapitel 4.3 beschriebene Command-Interface in der Controller-Software erlaubt das Hinzufügen beliebiger Befehle mit Verweis eines Funktionszeigers auf die auszuführende Funktion. Auf diese Weise konnte der Quellcode übersichtlich strukturiert werden, da es zu jedem Befehl eine zugehörige Routine gibt, welche die Ausführung des Befehls übernimmt und welche sich in jedem Source-File befinden kann. Ebenso unterstützt das Command-Interface eine variable Anzahl an Parametern, welche der Befehls-Routine zur weiteren Verarbeitung übergeben werden.

Der neu entwickelte Zellspannungsgenerator kann zum einen über eine Telnet-Verbindung über die im Command-Interface implementierten Befehle gesteuert werden. Zum anderen wurden verschiedene Matlab-Scripts zur Steuerung sowie zum Test des Generators entwickelt, welche über einen TCP-Socket ebenfalls mit dem Command-Interface interagieren.

Mit dem im Rahmen dieser Thesis entwickelten Zellspannungsgenerator V02 ist eine optimierte Fortführung vom Zellspannungsgenerator V01 gelungen. Die noch offenen Punkte sowie Aspekte mit Optimierungspotential werden im nachfolgenden Ausblick betrachtet.

6.2. Ausblick

Noch durchzuführen wäre die softwareseitige Implementierung der Strommessung, welche bisher nur hardwareseitig besteht. Diese kann unter Berücksichtigung des Konzeptes aus Abschnitt 3.2.7 sowie mit Hilfe der genannten Ansätze in Unterkapitel 4.11 implementiert werden.

Weiterhin offen sind Untersuchungen der Abhängigkeit zwischen der Ausgangsspannung der Module sowie der Umgebungstemperatur. Dazu können Messreihen in einem Temperatur-Schrank bei unterschiedlichen Temperaturen durchgeführt werden. Abhängigkeiten zwischen Ausgangsspannung und Temperatur können dann mit dem in Abschnitt 3.2.10 beschriebenen Temperatur-Sensor softwareseitig kompensiert werden. Sofern sich bei diesen Messreihen ein Einfluss auf die Linearität des Generators abzeichnet, kann zur Berechnung der Zwischenwerte ein anderes Interpolationsverfahren als die bisher implementierte lineare Interpolation herangezogen werden.

Das in Unterkapitel 5.3 beschriebene geringfügige Einbrechen der Ausgangsspannung sowie das Ground-Bouncing kann eventuell durch Verbindungen der verschiedenen Masseflächen minimiert oder behoben werden.

Das Problem der hohen Verlustleistung des LDOs der digitalen Versorgung, welches in Unterkapitel 5.7 beschrieben wurde, kann eventuell durch Tausch gegen einen Schaltregler minimiert werden. Es ist jedoch zu beobachten, ob durch den Schaltregler nicht wieder Störungen in die Versorgung der analogen Komponenten und damit in den Ausgang des Generators einkoppeln.

Sofern das 19“ Sub-Rack in ein Gehäuse eingebaut wird, ist die Abwärme zu beobachten, die eventuell den Einsatz mehrerer Lüfter erforderlich machen. Eine Lüftersteuerung könnte ebenfalls von einem der Module übernommen werden. Dazu steht einerseits der zusätzliche I2C-Bus zur Verfügung, welcher auf den Modulen auf eine Stiftleiste geführt wurde. Andernfalls könnte die Ansteuerung auch über den RS485-Bus erfolgen.

Insbesondere die sich in der Entwicklung befindlichen Klasse 3 Sensoren können durch den Downlink-Kanal über ihre Basisstation veranlasst werden, Messwerte zu erfassen und diese zu senden. Über den RS485-Bus wäre eine Synchronisation zwischen Zellspannungsgenerator und Basisstation denkbar, wodurch automatisierte Testreihen möglich wären.

Als weiterer Ausbau wäre ein Display vorstellbar, welches permanent den Status aller Module des Zellspannungsgenerators anzeigt. Dieses könnte ebenfalls über den I2C- oder RS485-Bus angesteuert werden.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] ADAM DUNKELS: *Lightweight TCP/IP stack*. Version: 2012. http://savannah.nongnu.org/projects/lwip/, Abruf: 13.08.2012

[2] ALLEGRO: *ACS710 - 120 kHz Bandwidth Current Sensor*. http://www.allegromicro.com/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/~/media/Files/Datasheets/ACS710-Datasheet.ashx, Abruf: 10.08.2012

[3] ANALOG DEVICES: *AD7798 - Low Noise, Low Power, 16-Bit ADC*. Version: 2007. http://www.analog.com/static/imported-files/data_sheets/AD7798_7799.pdf, Abruf: 10.08.2012

[4] ANALOG DEVICES: *AD8226 - Rail-to-Rail Output Instrumentation Amplifier*. Version: 2011. http://www.analog.com/static/imported-files/data_sheets/AD8226.pdf, Abruf: 10.08.2012

[5] ANALOG DEVICES: *ADM2682E - Signal & Power Isolated RS-485 Transceiver*. Version: 2011. http://www.analog.com/static/imported-files/data_sheets/ADM2682E_2687E.pdf, Abruf: 10.08.2012

[6] ANALOG DEVICES: *AD5541 - Serial-Input, Voltage-Output, 16-Bit DAC*. Version: 2012. http://www.analog.com/static/imported-files/data_sheets/AD5541_5542.pdf, Abruf: 06.08.2012

[7] ANALOG DEVICES: *ADUM1201 - Dual-Channel Digital Isolator*. Version: 2012. http://www.analog.com/static/imported-files/data_sheets/ADuM1200_1201.pdf, Abruf: 02.07.2012

[8] BEL COMPONENTS: *SI-60062-F - MagJack 10/100BaseT*. Version: 2008. http://belfuse.com/pdfs/SI-60062-F.pdf, Abruf: 07.08.2012

[9] BOGART, Theodore F.: *Electronic devices and circuits*. 5, Aufl. New Jersey : Prentice Hall, 2001. – ISBN 0–13–085178–7

[10] EGER, T.: *Entwicklung von Hard- und Software eines Readers für drahtlose Sensorik mit Resonanzabgleich*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, 2008

[11] HILLERMANN, L.: *Starterbatterie in Lithium-Eisen-Phosphat-Technologie – parallele Zellenmodule mit Überwachungs- und Leistungselektronik*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, 2012

[12] HOROWITZ, Paul ; HILL, Winfield: *The Art of Electronics*. 2, Aufl. Cambridge : Cambridge Univ. Press, 1995. – ISBN 0–521–37095–7

[13] IEEE: *Standard Group MAC Addresses: A Tutorial Guide*. http://standards.ieee.org/develop/regauth/tut/macgrp.pdf, Abruf: 13.08.2012

[14] ILGIN, S.: *Drahtlose Sensoren für Batteriemodule - Konzeption, Kalibrierung, Hard- und Softwareentwicklung*. Hamburg, Hochschule für Angewandte Wissenschaften, Bachelorthesis, 2011

[15] ILGIN, S. ; JEGENHOST, N. ; KUBE, R. ; PÜTTJER, S. ; RIEMSCHNEIDER, K.-R. ; SCHNEIDER, M. ; VOLLMER, J.: Zellenweiser Messbetrieb, Vorverarbeitung und drahtlose Kommunikation bei Fahrzeugbatterien. In: *10. GI/ITG KuVS Fachgespräch Sensornetze* (2011)

[16] ISSI: *IS42S83200D - 256-MBit Synchronous DRAM*. Version: 2011. www.issi.com/pdf/42S83200D-16160D.pdf, Abruf: 07.08.2012

[17] JEGENHORST, N.: *Entwicklung eines Zellensensors für Fahrzeugbatterien mit bidirektionaler drahtloser Kommunikation*. Hamburg, Hochschule für Angewandte Wissenschaften, Masterarbeit, 2011

[18] KILGENSTEIN, Otmar: *Schaltnetzteile in der Praxis*. 1, Aufl. Würzburg : Vogel-Fachbuch, 1986. – ISBN 3–8023–0727–5

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

[20] KRANNICH, T. ; PLASCHKE, S. ; RIEMSCHNEIDER, K.-R. ; VOLLMER, J.: Drahtlose Sensoren für Batterie-Zellen - ein Diskussionsbeitrag aus Sicht einer Anwendung. In: *8. GI/ITG KuVS Fachgespräch Sensornetze* (2009)

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

[22] LINEAR TECHNOLOGY: *LT1763CS8-3.3 - 500mA 3,3V Low Noise LDO*. http://cds.linear.com/docs/Datasheet/1763fg.pdf, Abruf: 22.07.2012

[23] LINEAR TECHNOLOGY: *LT1963A - 1,5A Fast Transient Low Noise LDO*. http://cds.linear.com/docs/Datasheet/1963afe.pdf, Abruf: 22.07.2012

[24] LINEAR TECHNOLOGY: *Minimizing Switching Regulator Residue in Linear Regulator Outputs*. Version: 2005. http://cds.linear.com/docs/Application%20Note/an101f.pdf, Abruf: 01.08.2012

[25] MURATA: *NMXS1215UC - Isolated 5W Single & Dual Output DC/DC Converter*. http://www.murata-ps.com/data/power/ncl/kdc_nmxu.pdf, Abruf: 30.07.2012

[26] MYRRA: *Flachtrafo ,10VA, 115-230V, 2x9V*. http://http://www.myrra.com, Abruf: 22.07.2012

[27] NOTTEN, P. ; HETZENDORF, G. ; RIEMSCHNEIDER, K.-R.: Arrangement and Method for Monitoring Pressure within an Battery Cell. In: *EP1856760B1, US2008097704A1, WO2006077519A1, CN100533845C* (2006)

[28] PHILIPS: *Automatik-Multimeter PM2519*. http://www.helmut-singer.de/pdf/phpm2519.pdf, Abruf: 13.08.2012

[29] PLASCHKE, S.: *Experimentalsystem für drahtlose Batteriesensorik*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, 2008

[30] PÜTTCHER, S.: *Diagnosefunktion für Automobil-Starterbatterien mit drahtlosen Zellsensoren*. Hamburg, Hochschule für Angewandte Wissenschaften, Diplomarbeit, 2011

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

[32] RIEMSCHNEIDER, K.-R.: Wireless Battery Management System. In: *Pat.appl. WO2004/047215A1, US020060152190A1, EP000001573851A1* (2004)

[33] RIEMSCHNEIDER, K.-R. ; SCHNEIDER, M.: Drahtlose Sensoren in den Zellen von Fahrzeug-Batterien. In: *IWKM21 Scientific Report* (2011)

[34] SCHNEIDER, M. ; ILGIN, S. ; JEGENHOST, N. ; KUBE, R. ; PÜTTJER, S. ; RIEMSCHNEIDER, K.-R. ; VOLLMER, J.: Automotive Battery Monitoring by Wireless Cell Sensors. In: *Instrumentation and Measurement Technology Conference* (2012)

[35] SCHWARTAU, F.: *Vielkanaliger controller-gesteuerter Zellspannungsgenerator zur Kalibrierung und zum Test von Batteriesensoren*. Hamburg, Hochschule für Angewandte Wissenschaften, Bachelorthesis, 2012

[36] SEMICONDUCTOR, National: *LM1084IT-3.3 - Linear Spannungsregler 3,3V*. Version: 2005. http://cache.national.com/ds/LM/LM1084.pdf, Abruf: 22.07.2012

[37] SST: *SST25VF032B - 32 Mbit SPI Serial Flash*. Version: 2011. http://www.sst.com/dotAsset/40373.pdf, Abruf: 02.07.2012

[38] STMICROELECTRONICS: *L6920 - 1V high efficiency syncronous step up converter*. Version: 2005. http://www.stm32circle.com/resources/Datasheets/L6920.pdf, Abruf: 02.07.2012

[39] TE CONNECTIVITY: *3V SMD Relais*. Version: 1999. http://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=108-98001, Abruf: 07.08.2012

[40] TEKTRONIX: *MSO3034 Mixed Signal Oscilloscope*. http://www.tek.com/sites/tek.com/files/media/media/resources/MSO-DPO3000_Series_Oscilloscopes_Datasheet_3GW-21364-9_1.pdf, Abruf: 13.08.2012

[41] TEXAS INSTRUMENTS: *TMP102 - Low Power Digital Temp Sensor with SMBus/Two-Wire Serial Interface*. Version: 2008. http://www.ti.com/lit/gpn/tmp102, Abruf: 10.08.2012

[42] TEXAS INSTRUMENTS: *TPS61201 - Low input voltage synchronous boost converter*. Version: 2008. http://www.ti.com/lit/gpn/tps61201, Abruf: 02.07.2012

[43] TEXAS INSTRUMENTS: *Clocking Options for Stellaris Family Microcontrollers (AN01240)*. Version: 2009. http://www.ti.com/lit/an/spma004/spma004.pdf, Abruf: 07.08.2012

[44] TEXAS INSTRUMENTS: *Using Stellaris MCUs Internal Flash Memory to Emulate EEPROM (AN01267)*. Version: 2009. http://www.ti.com/litv/pdf/spma030, Abruf: 08.08.2012

[45] TEXAS INSTRUMENTS: *REF5025 - Low Noise, Very Low Drift, Precision Voltage Reference*. Version: 2010. http://www.ti.com/lit/gpn/ref5025, Abruf: 22.07.2012

[46] TEXAS INSTRUMENTS: *LM7301 - 4 MHz GBW, Rail-to-Rail Input-Output Operational Amplifier*. Version: 2011. http://www.ti.com/lit/gpn/lm7301, Abruf: 07.08.2012

[47] TEXAS INSTRUMENTS: *OPA564 - 1.5A, 24V, 17MHz Power Operational Amplifier*. Version: 2011. http://www.ti.com/lit/gpn/opa564, Abruf: 06.08.2012

[48] TEXAS INSTRUMENTS: *Stellaris LM3S9B92 Evaluation Kit*. Version: 2011. http://www.ti.com/lit/ug/spmu035c/spmu035c.pdf, Abruf: 06.08.2012

[49] TEXAS INSTRUMENTS: *Stellaris LM3S9B92 Microcontroller*. Version: 2012. http://www.ti.com/lit/gpn/lm3s9b92, Abruf: 06.08.2012

[50] TEXAS INSTRUMENTS: *Stellaris LM3S9B96 Microcontroller*. Version: 2012. http://www.ti.com/lit/gpn/lm3s9b96, Abruf: 06.08.2012

[51] TEXAS INSTRUMENTS: *Stellaris Peripheral Driver Library*. Version: 2012. http://www.ti.com/tool/sw-lm3s, Abruf: 07.08.2012

[52] TIETZE, U. ; SCHENK, Ch. ; GAMM, E.: *Halbleiter-Schaltungstechnik*. 13, Aufl. Heidelberg : Springer, 2010. – ISBN 978–3–642–01621–9

[53] WIKIPEDIA: *Flyback converter*. Version: 2012. http://en.wikipedia.org/wiki/Flyback_converter, Abruf: 01.08.2012

Abkürzungsverzeichnis

HAW Hamburg Hochschule für Angewandte Wissenschaften Hamburg

BATSEN Drahtlose Zellensensoren für Fahrzeugbatterien

Zellspannungsgenerator V01 Zellspannungsgenerator Prototyp nach [35]

Zellspannungsgenerator V02 Im Rahmen dieser Arbeit entstandenes Redesign

SOC State of Charge

SOH State of Health

ISM-Band Industrial, Scientific and Medical Band

RFID Radio Frequency Identification

ADC Analog Digital Converter

DAC Digital Analog Converter

MAC Media Access Control

TCP Transmission Control Protocol

IP Internet Protocol

PHY Ethernet Physical Layer

OSI-Modell Schichtenmodell zur Kommunikation in Rechnernetzen

lwIP Lightweight TCP/IP stack

FIFO First In - First Out

ASCII American Standard Code for Information Interchange

LDO Low-dropout linear voltage regulator

ESR Equivalent Series Resistance

EPI External Peripheral Interface

SPI Serial Peripheral Interface

I2C Inter-Integrated Circuit

GPIO General Purpose Input/Output Port

PCB Printed Circuit Board

SMD Surface-Mounted Device

PLL Phase-Locked Loop

JTAG Joint Test Action Group - Programmier-/Debug-Schnittstelle

UART Universal Asynchronous Receiver Transmitter

LE Längeneinheit

ROM Read-Only Memory

NVIC Nested Vectored Interrupt Controller

RTOS Real-Time Operating System

IEEE Institute of Electrical and Electronics Engineers

DHCP Dynamic Host Configuration Protocol

Enum Datentyp für Aufzählungen

Gain Verstärkungsfaktor

Anhang

A. Aufgabenstellung

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Hochschule für Angewandte Wissenschaften Hamburg
Department Informations- und Elektrotechnik
Prof. Dr.-Ing. Karl-Ragmar Riemschneider

11. Mai 2012

Bachelorthesis Tobias Steinmann

Hard- und Softwareentwicklung für einen Controller-gesteuerten, vernetzten Zellspannungsgenerator

Motivation

Fahrzeugbatterien sind in der Regel aus in Reihe geschalteten Batteriezellen aufgebaut. Für den optimierten Betrieb mit dem Ziel der Lebensdauersicherung dieser Batterien soll der messtechnischer Zugang zu den Zellen mit drahtlosen Sensoren erfolgen. An der HAW Hamburg werden Konzepte dafür im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Forschungsvorhabens BATSEN (drahtlose Zellensensoren für Fahrzeugbatterien) untersucht.
Für die Sensorentwicklung ist die Kalibrierung und der Testbetrieb mit reproduzierbaren Messwerten eine wichtige Teilaufgabe.

Aufgabe

Die Aufgabe der Abschlussarbeit ist ein modulares System von Zellspannungsgeneratoren weiterzuentwicklen. Zunächst hat Herr Steinmann die Aufgabe, bisherige Konzepte und Realisierungsansätze zu untersuchen und Ansatzpunkte der Fortführung aufzugreifen. Eine Besonderheit besteht darin das hochgenaue und schnelle Betriebsarten möglich sein sollen, obwohl die Anforderungen schwer zu vereinen sind. Nach einer Reihe von Untersuchungen und Analysen soll ein Redesign und eine Erweiterung der Hard- und Software erfolgen. Eine Erprobung soll die Arbeiten begleiten und den Erfolg der Maßnahmen sichern.

1) **Analyse der Rahmenbedingungen und des Standes der bisherigen Lösung**

   - Einarbeitung durch Recherche und Erfassung der Vorarbeiten
   - Identifikation und Analyse der Anschlußarbeiten und Problembereiche der Vorarbeiten
   - Voruntersuchungen zu passenden Bauelementen und Schaltungsvarianten
   - Erarbeitung und Gegenüberstellung von Lösungskonzepten und weitere Variantenbetrachtungen

2) **Untersuchungen und Redesign zur Hardware**

   a) Realisierung einer störungsarmen Spannungsversorgung, Erprobung und Vergleich
   - Verhinderung von Einstreuungen/Ripple der Schaltregler
   - Minimierung des Einflusses der Step-Up/Down Regler der Sensoren auf den den Generator

   b) Entwurf und Realisierung von Mainboards mit Mikrocontroller, insbesondere um Einschränkungen des bisherigen Evaluierungsboards zu umgehen
   - dabei Sicherstellung der galvanische Trennung inkl. der Ethernetbuchse
   - direkte RAM-Anbindung per External Peripheral Interface des Controllers

   c) Alternative Synchronisation der Generatoren über einen Industriebus-Chip (RS-485) mit galvanisicher Trennung

d) konzeptielle Untersuchung und Entwurfsentscheidung ob das Verstärkerboard in das Mainboard oder getrenntes Modul gestaltet werden soll

e) Aufbau eines Komplettgerätes mit Rack, Vernetzung, Stromversorgung, Frontplatten, Kühlung usw. auf der Basis von 19-Zoll-Gehäusen

f) Untersuchung der Störeinflüsse und Optimierung des Schaltungskonzeptes im Zusammenhang mit der Strommessung

3) **Weiterentwicklung und Strukturierung der Software**

a) Umstellung und Beschleunigung der Übertragung von Spannungswerten von ASCII-Kodierung auf binäre Übertragung, Absicherung der Übertragung
mittels Prüfsummenverfahren

b) Strommessung und Aufzeichnung in der Controller-SW implementieren

c) Einbeziehung des Temperatursensors in die Controller-SW, insbesondere zur Werterfassung

d) Aufbau Telnet/Command Interfaces mittels Command-Table und Funktionszeigern auf der Controllerseite

e) Anpassung der Matlab-Steuerskripten an neues Command-Interface und an die binäre Werte-Übertragung

f) sowie Erweiterung der Test-, Kalibrier- und Mess-Skripten in Matlab für die Erprobungen

4) **Optimierung von Hard- und Software für verschiedene Betriebmodi**

- Analyse und ggf. Realsierung einer Umschaltung von Stationär- und Transienten-Betrieb
- Untersuchung von der Störeinflüsse von Open- und Closed-Loop-Ausgabe

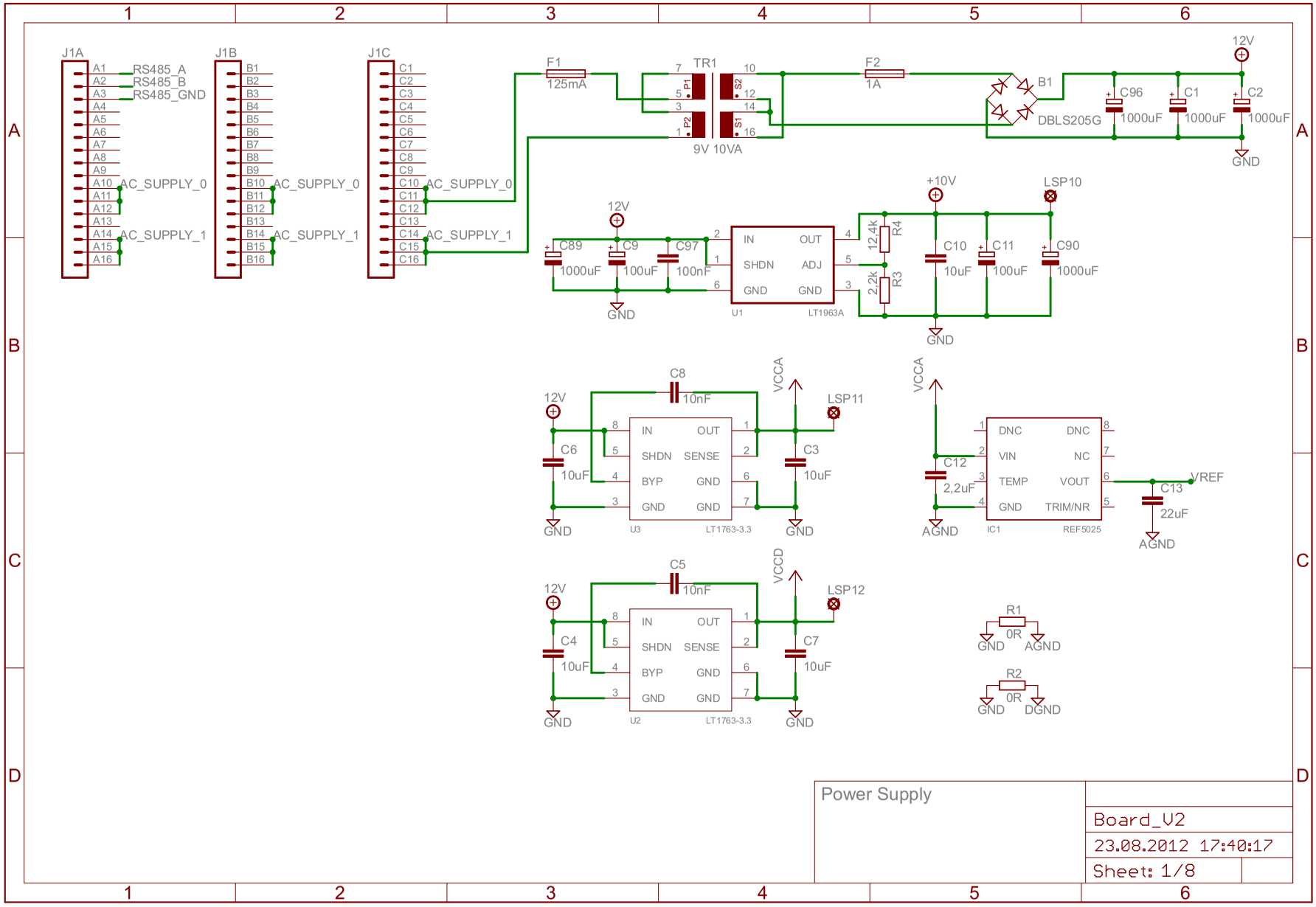
5) **Erprobung von Teilsystemen und des Gesamtsystems sowie Gesamtbewertung**

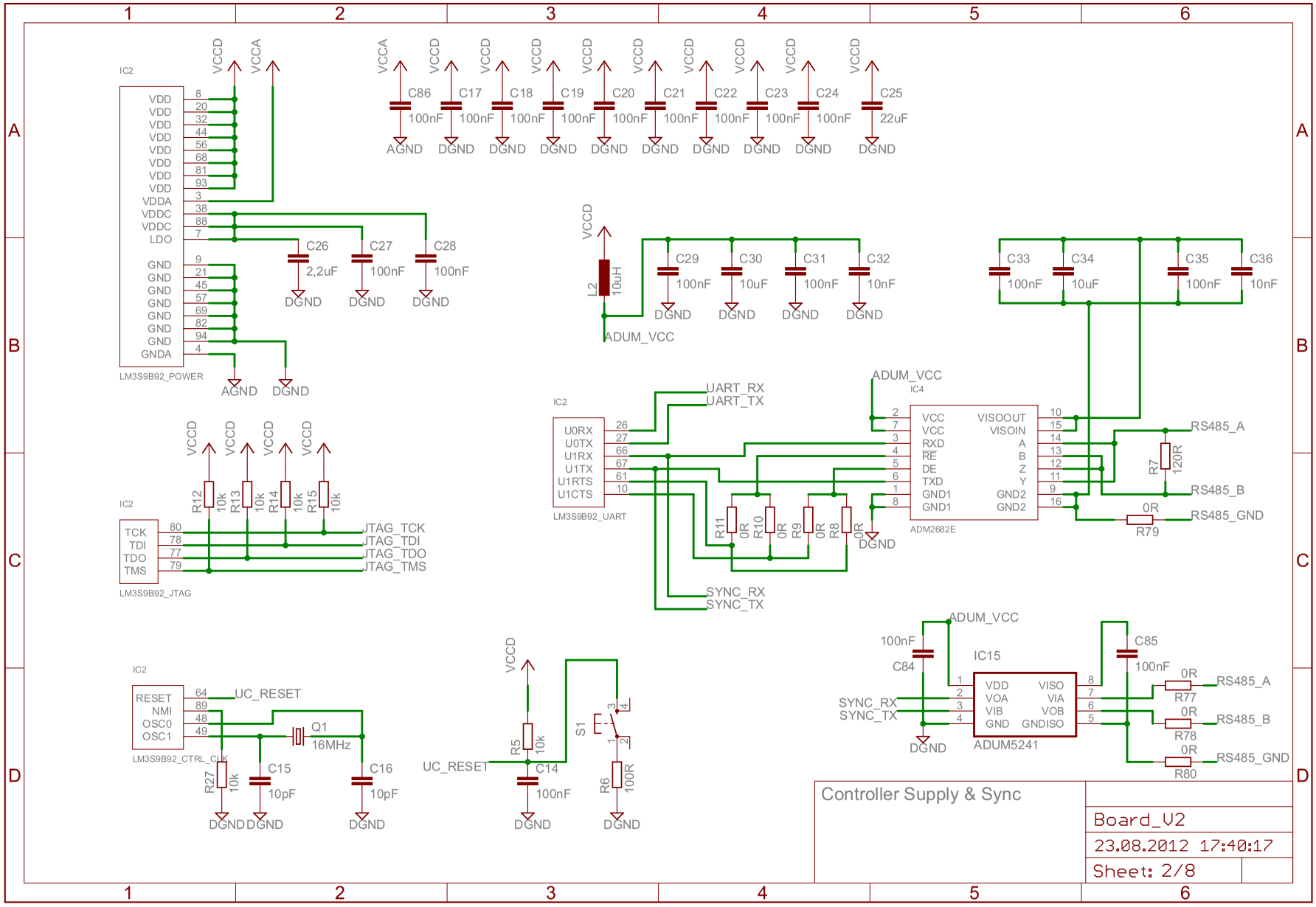
- Funktionsnachweis mit Messmitteln und durch ausgewählte Messreihen
- optional: Funktionserprobung durch Kalibrierlauf unter Einbeziehung des Sensors
- Darstellung und Dokumentation der Messergebnisse
- Diskussion und Bewertung der Ergebnisse

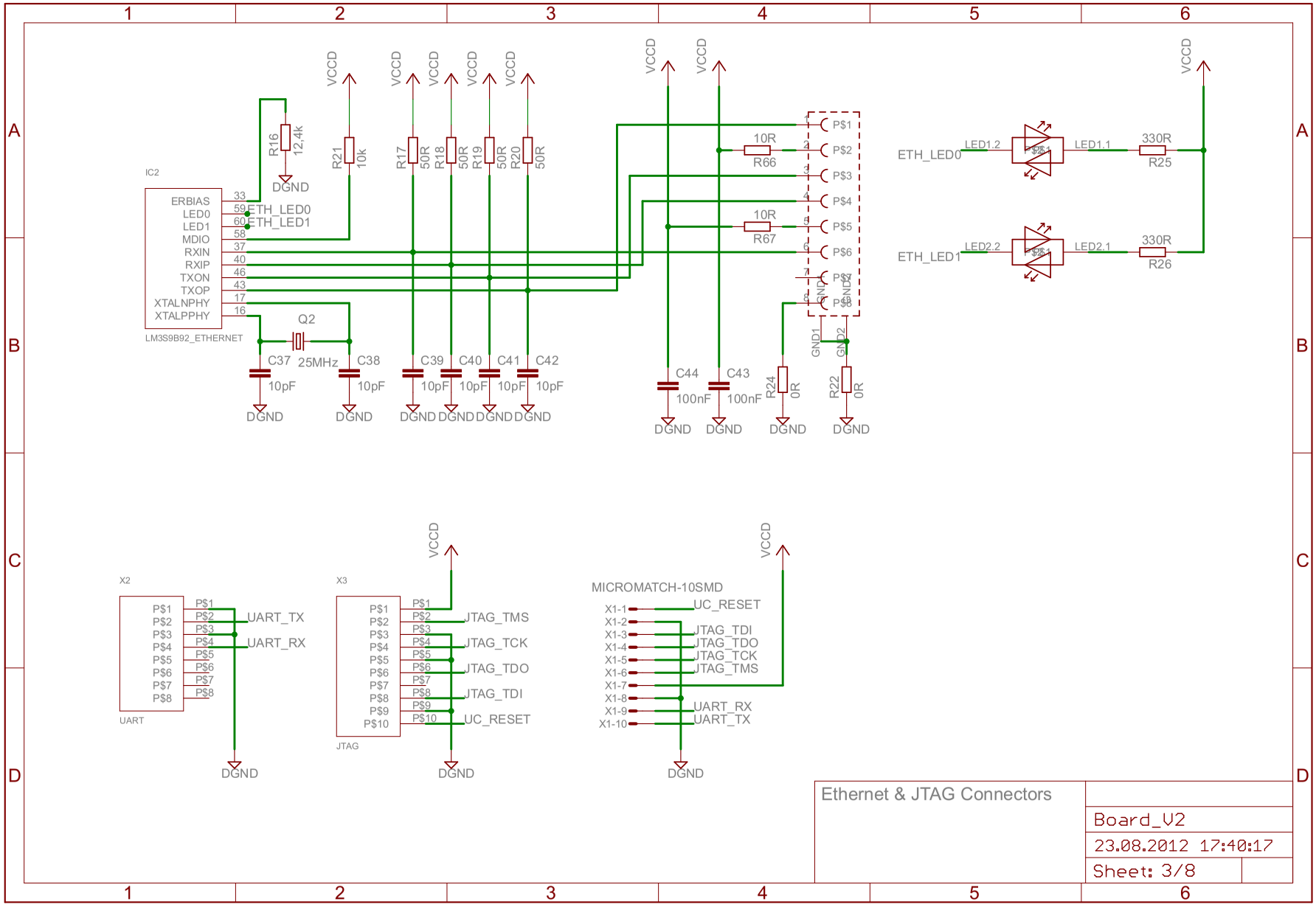
Dokumentation

Die Fachliteratur, die Vorarbeiten und die kommerziellen Unterlagen sind zielgerichtet zu recherchieren. Dabei sind insbesondere aktuelle Bauelemente und Beispiellösungen näher zu betrachten. Die gewählte Lösung, die Funktionsweise der Hardware und die Struktur der Software-Module sind gut nachvollziehbar zu dokumentieren. Die Rahmenbedingungen, die Konzeptionen, auftretende Probleme und wesentliche Folgerungen sollen beschrieben werden. Die Messergebnisse sind in exemplarischem Umfang zu erfassen und auszuwerten. Die realisierten Lösungen und die Ergebnisse sind kritisch einordnend zu bewerten. Ansätze für Verbesserungen und weitere Arbeiten sind zu nennen.

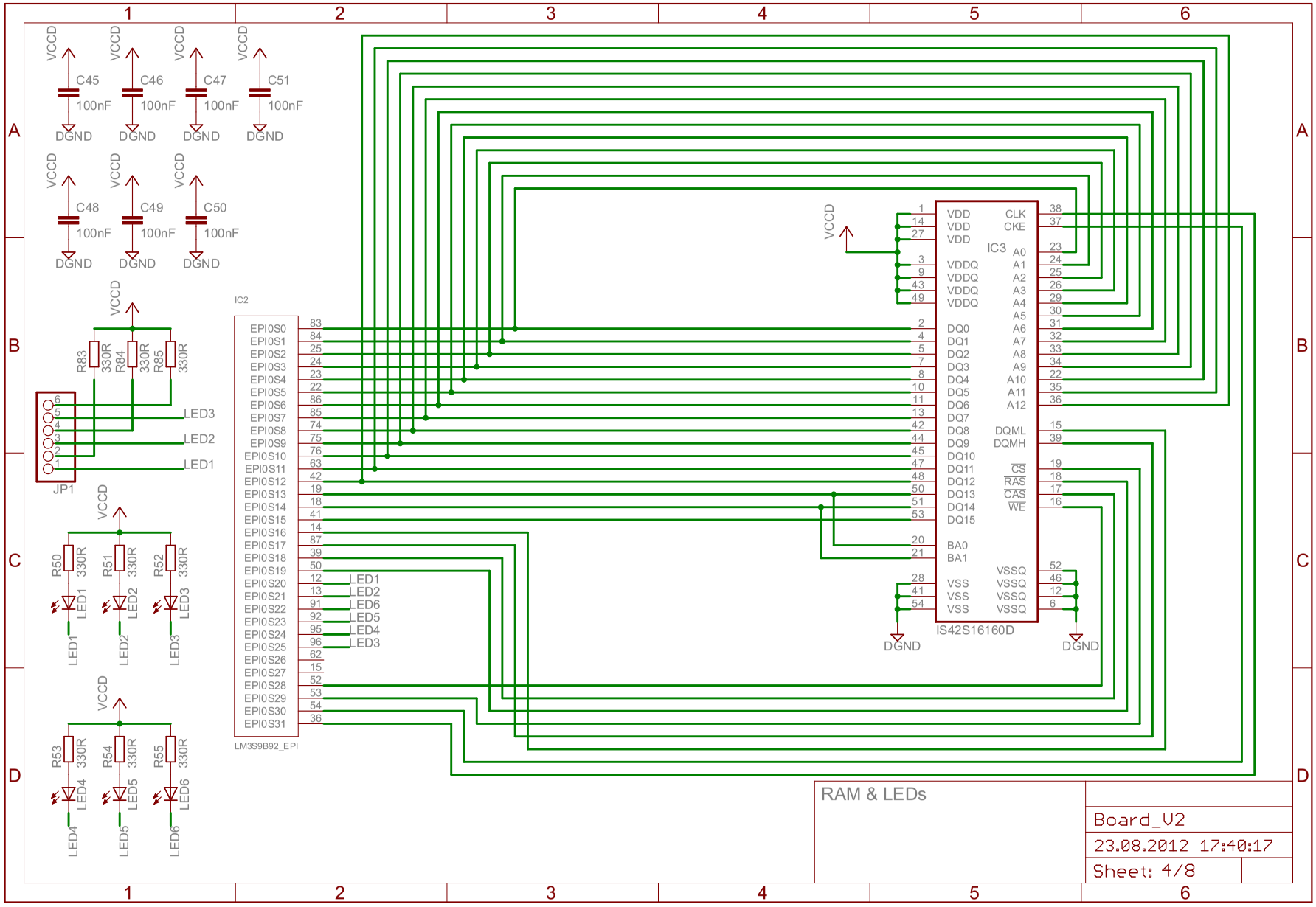
B. Schaltplan

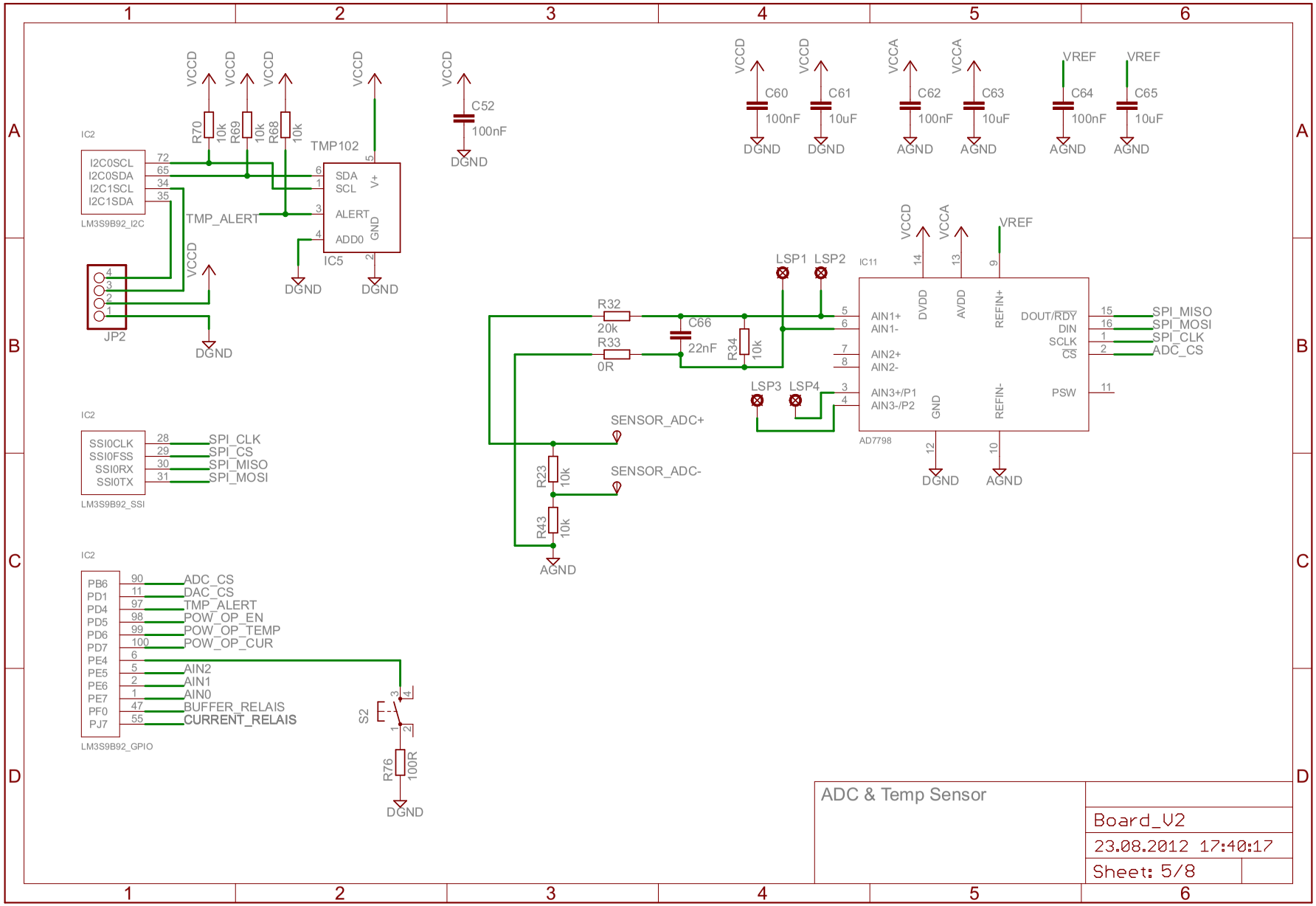

J1A
A1 RS485_A
A2 RS485_B
A3 RS485_GND
A10 AC_SUPPLY_0
A11
A12
A14 AC_SUPPLY_1
A15
A16
J1B
B10 AC_SUPPLY_0
B11
B12
B14 AC_SUPPLY_1
B15
B16
J1C
C10 AC_SUPPLY_0
C11
C12
C14 AC_SUPPLY_1
C15
C16
F1
125mA
TR1
9V 10VA
F2
1A
B1
DBLS205G
C96
1000uF
C1
1000uF
C2
1000uF
12V
GND
C89
1000uF
C9
100uF
C97
100nF
IN
SHDN
GND
OUT
ADJ
GND
U1
LT1963A
12.4k
R4
2.2k
R3
+10V
C10
10uF
C11
100uF
LSP10
C90
1000uF
C8
10nF
VCCA
LSP11
C6
10uF
C3
10uF
U3
LT1763-3.3
DNC
VIN
TEMP
GND
DNC
NC
VOUT
TRIM/NR
C12
2,2uF
AGND
IC1
REF5025
VREF
C13
22uF
C5
10nF
VCCD
LSP12
C4
10uF
C7
10uF
U2
LT1763-3.3
R1
0R
R2
0R
DGND
Power Supply
Board_V2
23.08.2012 17:40:17
Sheet: 1/8

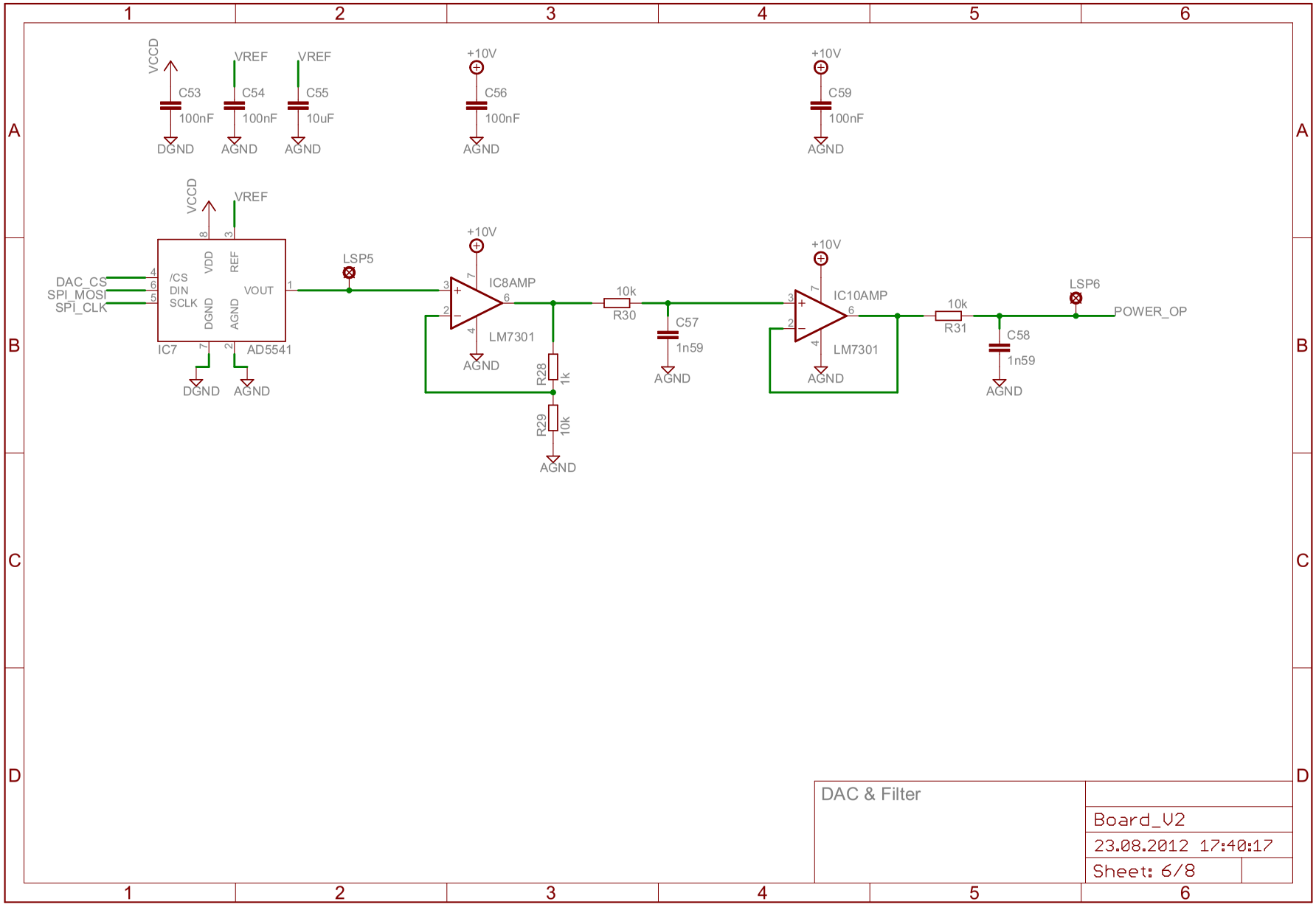

Controller Supply & Sync
Board_V2
23.08.2012 17:40:17
Sheet: 2/8
LM3S9B92_POWER
LM3S9B92_JTAG
LM3S9B92_UART
LM3S9B92_CTRL_CLK
ADM2682E
ADUM5241
VCCD
VCCA
AGND
DGND
ADUM_VCC
UART_RX
UART_TX
SYNC_RX
SYNC_TX
JTAG_TCK
JTAG_TDI
JTAG_TDO
JTAG_TMS
UC_RESET
RS485_A
RS485_B
RS485_GND

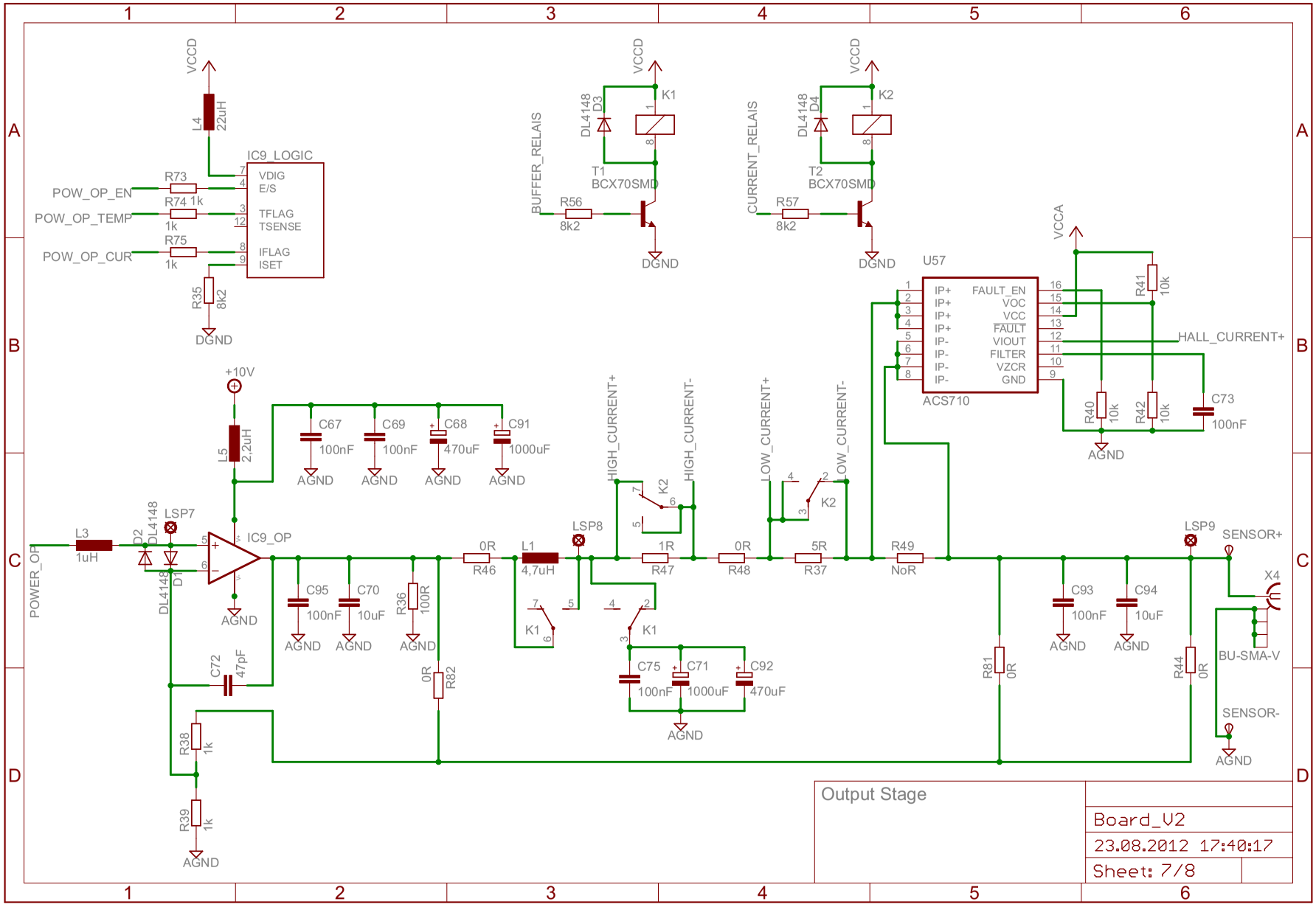


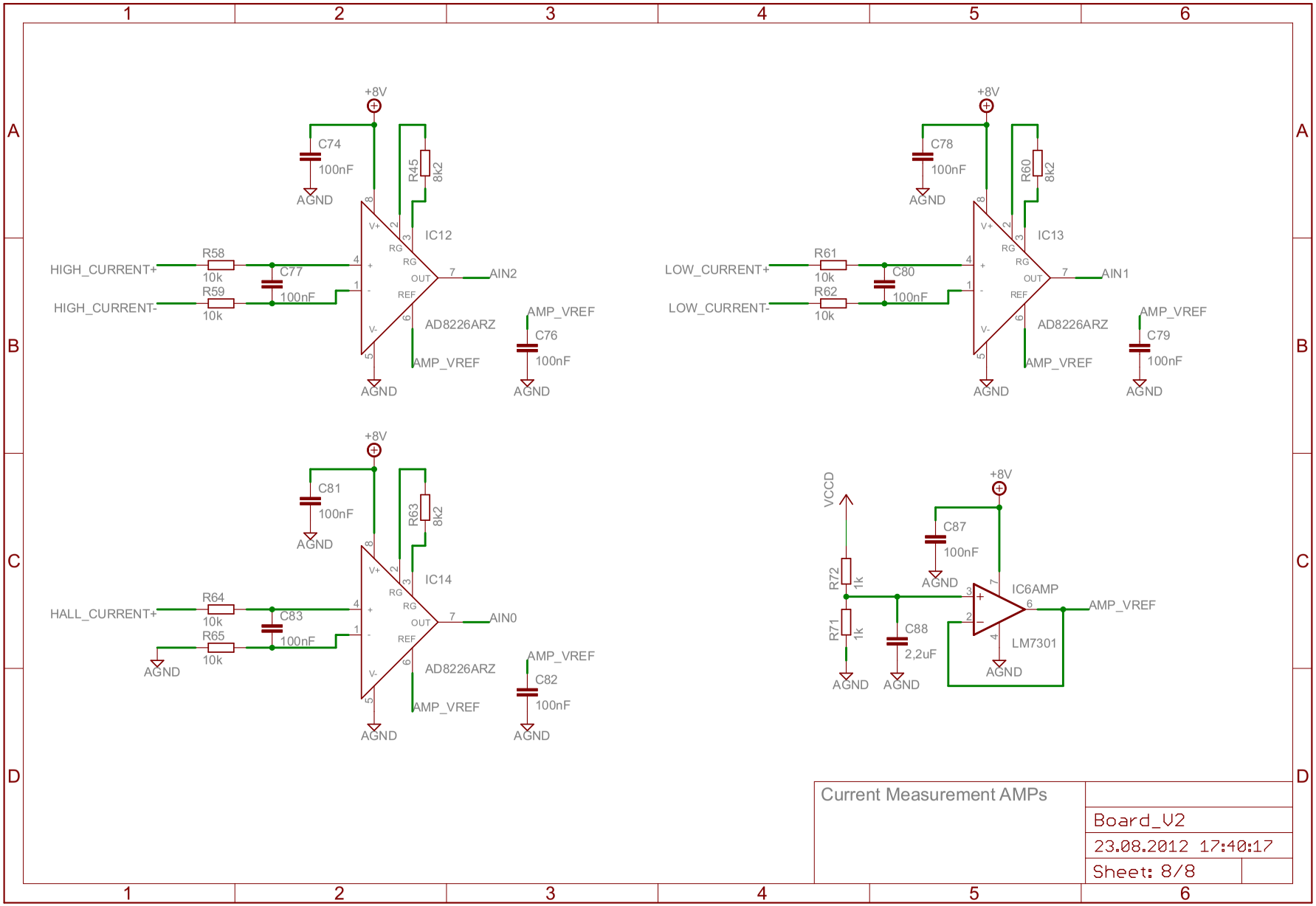
Ethernet & JTAG Connectors	
	Board_V2
	23.08.2012 17:40:17
	Sheet: 3/8


C45 100nF
C46 100nF
C47 100nF
C51 100nF
C48 100nF
C49 100nF
C50 100nF
VCCD
DGND
R83 330R
R84 330R
R85 330R
JP1
LED3
LED2
LED1
R50 330R
R51 330R
R52 330R
R53 330R
R54 330R
R55 330R
LED4
LED5
LED6
IC2
EPI0S0
EPI0S1
EPI0S2
EPI0S3
EPI0S4
EPI0S5
EPI0S6
EPI0S7
EPI0S8
EPI0S9
EPI0S10
EPI0S11
EPI0S12
EPI0S13
EPI0S14
EPI0S15
EPI0S16
EPI0S17
EPI0S18
EPI0S19
EPI0S20
EPI0S21
EPI0S22
EPI0S23
EPI0S24
EPI0S25
EPI0S26
EPI0S27
EPI0S28
EPI0S29
EPI0S30
EPI0S31
LM3S9B92_EPI
IC3
VDD
VDDQ
DQ0
DQ1
DQ2
DQ3
DQ4
DQ5
DQ6
DQ7
DQ8
DQ9
DQ10
DQ11
DQ12
DQ13
DQ14
DQ15
BA0
BA1
VSS
VSSQ
CLK
CKE
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
DQML
DQMH
CS
RAS
CAS
WE
IS42S16160D
RAM & LEDs
Board_V2
23.08.2012 17:40:17
Sheet: 4/8

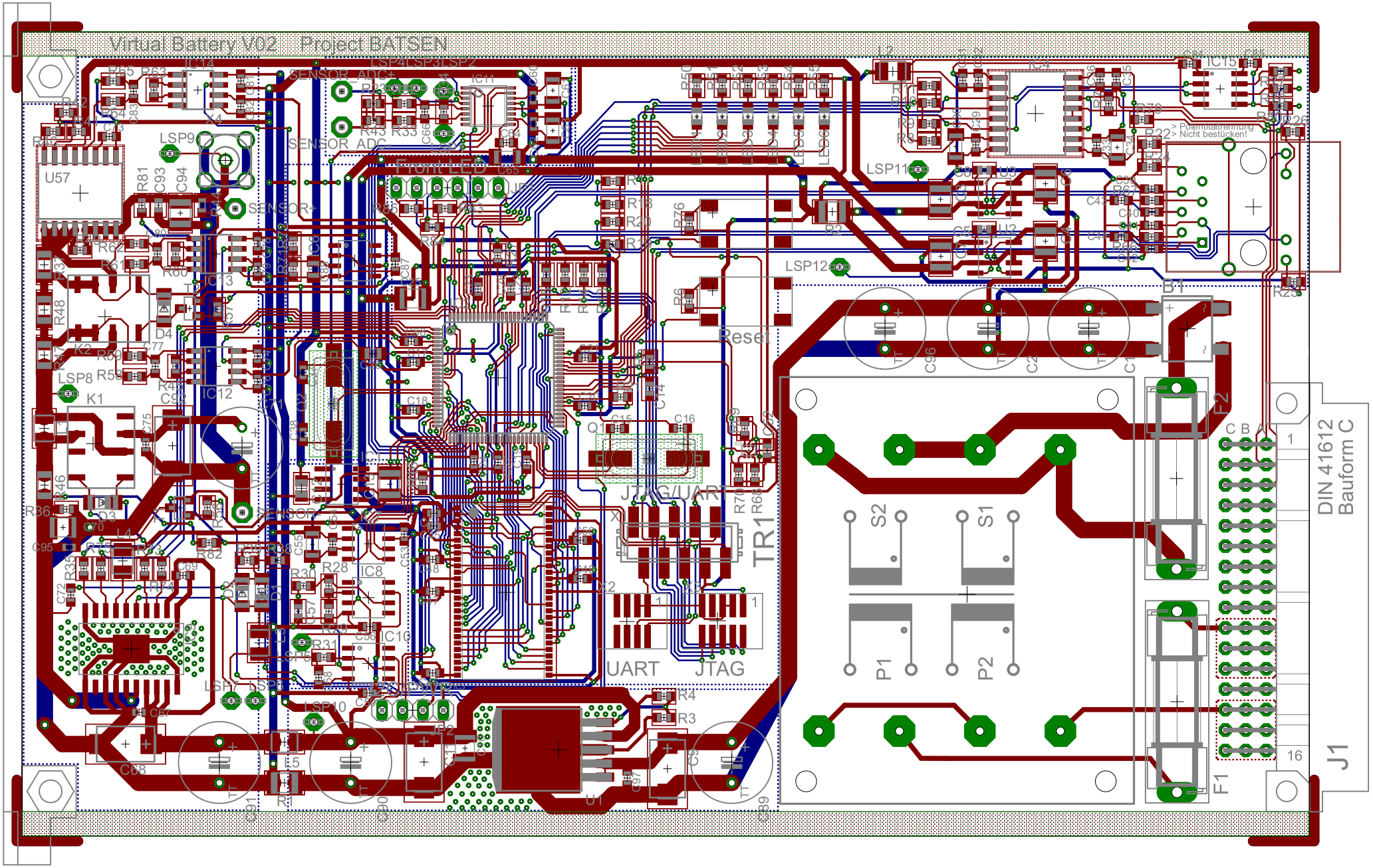

1
2
3
4
5
6
A
B
C
D
VCCD
VCCA
VREF
DGND
AGND
IC2
I2C0SCL
I2C0SDA
I2C1SCL
I2C1SDA
LM3S9B92_I2C
R70 10k
R69 10k
R68 10k
TMP102
SDA
SCL
ALERT
ADD0
V+
GND
IC5
TMP_ALERT
JP2
C52 100nF
C60 100nF
C61 10uF
C62 100nF
C63 10uF
C64 100nF
C65 10uF
IC11
AIN1+
AIN1-
AIN2+
AIN2-
AIN3+/P1
AIN3-/P2
DVDD
AVDD
REFIN+
REFIN-
DOUT/RDY
DIN
SCLK
CS
PSW
AD7798
LSP1
LSP2
LSP3
LSP4
R32 20k
R33 0R
C66 22nF
R34 10k
R23 10k
R43 10k
SENSOR_ADC+
SENSOR_ADC-
SPI_MISO
SPI_MOSI
SPI_CLK
ADC_CS
SSI0CLK
SSI0FSS
SSI0RX
SSI0TX
LM3S9B92_SSI
PB6
PD1
PD4
PD5
PD6
PD7
PE4
PE5
PE6
PE7
PF0
PJ7
LM3S9B92_GPIO
DAC_CS
POW_OP_EN
POW_OP_TEMP
POW_OP_CUR
AIN2
AIN1
AIN0
BUFFER_RELAIS
CURRENT_RELAIS
S2
R76 100R
ADC & Temp Sensor
Board_V2
23.08.2012 17:40:17
Sheet: 5/8


VCCD
VREF
VREF
+10V
+10V
C53
100nF
C54
100nF
C55
10uF
C56
100nF
C59
100nF
DGND
AGND
AGND
AGND
AGND
VCCD
VREF
DAC_CS
SPI_MOSI
SPI_CLK
/CS
DIN
SCLK
VDD
REF
VOUT
DGND
AGND
IC7
AD5541
LSP5
+10V
IC8AMP
LM7301
AGND
R28
1k
R29
10k
AGND
10k
R30
C57
1n59
AGND
+10V
IC10AMP
LM7301
AGND
10k
R31
C58
1n59
AGND
LSP6
POWER_OP
DAC & Filter
Board_V2
23.08.2012 17:40:17
Sheet: 6/8


VCCD
L4
22uH
IC9_LOGIC
VDIG
E/S
TFLAG
TSENSE
IFLAG
ISET
R73
R74 1k
R75
1k
POW_OP_EN
POW_OP_TEMP
POW_OP_CUR
R35
8k2
DGND
BUFFER_RELAIS
DL4148
D3
K1
T1
BCX70SMD
R56
8k2
CURRENT_RELAIS
D4
K2
T2
R57
VCCA
U57
IP+
IP-
FAULT_EN
VOC
VCC
FAULT
VIOUT
FILTER
VZCR
GND
ACS710
R41
10k
HALL_CURRENT+
R40
R42
C73
100nF
AGND
+10V
L5
2,2uH
C67
C69
C68
C91
470uF
1000uF
HIGH_CURRENT+
HIGH_CURRENT-
LOW_CURRENT+
LOW_CURRENT-
LSP7
LSP8
LSP9
L3
1uH
D2
D1
IC9_OP
POWER_OP
0R
R46
L1
4,7uH
1R
R47
R48
5R
R37
R49
NoR
SENSOR+
X4
BU-SMA-V
SENSOR-
C95
C70
10uF
R36
100R
C72
47pF
R82
R38
R39
C75
C71
C92
R81
C93
C94
R44
Output Stage
Board_V2
23.08.2012 17:40:17
Sheet: 7/8


1
2
3
4
5
6
A
B
C
D
+8V
C74
100nF
AGND
R45
8k2
IC12
HIGH_CURRENT+
HIGH_CURRENT-
R58
10k
R59
10k
C77
100nF
AIN2
AD8226ARZ
AMP_VREF
C76
100nF
C78
R60
IC13
LOW_CURRENT+
LOW_CURRENT-
R61
R62
C80
AIN1
C79
C81
R63
IC14
HALL_CURRENT+
R64
R65
C83
AIN0
C82
VCCD
R72
1k
R71
C87
C88
2,2uF
IC6AMP
LM7301
Current Measurement AMPs
Board_V2
23.08.2012 17:40:17
Sheet: 8/8

C. Platinenlayout


Virtual Battery V02 Project BATSEN
SENSOR_ADC+
SENSOR_ADC-
SENSOR+
SENSOR-
Front LED
Reset
JTAG/UART
UART
JTAG
TR1
B1
S1
S2
P1
P2
F1
F2
DIN 41612
Bauform C
J1
U57
U1
U2
U3
LSP8
LSP9
LSP10
LSP11
LSP12
> Polemittelanordnung
> Nicht bestücken!

D. Matlab-Files

vbBoards.m

```
% Zuordnung Boardname <-> IP-Adresse
classdef vbBoards

    properties (Constant)
        CELL0 = '192.168.0.130';
        CELL1 = '192.168.0.133';
        CELL2 = '192.168.0.132';
        CELL3 = '192.168.0.200';
    end

end
```

vbConnect.m

```
% TCP/IP Verbindung herstellen
function [ con ] = vbConnect( addr )

    % Bei einzelner Adresse Array erstellen
    if(ischar(addr))
        addr = {addr};
    end

    % Verbindungen mit Parametern herstellen
    port = 56936;
    for i = 1:length(addr)
        con(i) = tcpip(addr{i}, port); %#ok<*AGROW,TNMLP>
        con(i).OutputBufferSize = 15000;
        con(i).Timeout = 1;

        fopen(con(i));

        % Pruefen, ob Verbindung geoeffnet werden konnte
        if(strcmp(con(i).Status, 'open') == false)
            fprintf('Unable to connect to %s\n', con(i).RemoteHost);
        end

        % Puffer leeren
        fread(con(i), 1);
    end

end
```

vbCmd.m

```
% Befehl an Board senden
function vbCmd( con, cmd )
    fwrite(con, sprintf([cmd '\n']));
end
```

vbDisconnect.m

```
% TCP/IP Verbindungen trennen
function vbDisconnect( con )

    for i = 1:length(con)
        fclose(con(i));
    end

end
```

vbDouble2Microvolt.m

```
function [ result ] = vbDouble2Microvolt( voltageArray )

    valueCnt = length(voltageArray);

    % Werte in Microvolt konvertieren
    result = zeros(1, valueCnt);

    for i = 1:valueCnt
        result(i) = uint32(voltageArray(i) * 1000000);
    end

end
```

vbVoltageTransfer.m

```
% Spannungswerte im Binaerformat an Board senden
function vbVoltageTransfer( con, voltageArray )

    valueCnt = length(voltageArray);
    localChecksum = 0;

    % Buffer anlegen
    bsize = 900;                    % Puffergroesse in Byte (durch drei teilbar)
    buffer = zeros(bsize, 1);
    bufIndex = 1;
    restCnt = mod(valueCnt, bsize); % Restwerte kleiner Puffergroesse

    % In Binaeruebertragungsmodus wechseln
    vbCmd(con, 'voltagearray');

    % Zeitmessung starten
    tic;

    for i = 1:valueCnt
        %value = data(i);
        value = voltageArray(i);
        localChecksum = localChecksum + value;

        % 24 Bit Binaerwert in 3 Oktette zerlegen und an Buffer anhaengen
        ubyte = floor(value / 65536);
        mbyte = floor(mod(value,  65536) / 256);
        lbyte = (mod(value,  256));
```

```
        buffer(bufIndex : bufIndex+2) = [ubyte, mbyte, lbyte];
        bufIndex = bufIndex+3;

        % Buffer senden
        if(bufIndex >= bsize)
            fwrite(con, buffer, 'uint8');
            bufIndex = 1;

            % Bestaetigungsbyte einlesen
            fread(con, 1);

            % Status ausgeben
            fprintf('%2.1f%%\n', i/valueCnt*100);
        end

        % Buffer an Restgroesse anpassen
        if(valueCnt-i == restCnt)
            bsize = restCnt*3;
            buffer = zeros(bsize, 1);
            bufIndex = 1;
        end

        %daten = fread(con, 1);
    end

    % Uebertragung beenden
    endSequence = [255 255 255];
    fwrite(con, endSequence, 'uint8');

    % Zeitmessung beenden
    tStop = toc;
    fprintf('Uebertragungszeit: %f\n\n', tStop);

    % Checksumme abfragen
    fprintf('Pruefe Checksumme... ');
    pause(0.5);
    fwrite(con, sprintf('voltagechecksum\n'));

    rxdata = fread(con, 15);
    remoteChecksum = char(rxdata(3:15));
    remoteChecksum = uint64(str2double(remoteChecksum));

    % Checksumme auswerten
    if(localChecksum == remoteChecksum)
        fprintf('Uebertragung erfolgreich, checksum: %13.0f\n', localChecksum);
    else
        fprintf('Uebertragung fehlgeschlagen !!!\n');
        fprintf('localChecksum:  %13.0f\n', localChecksum);
        fprintf('remoteChecksum: %13.0f\n', remoteChecksum);
    end

end
```

wiedergabe.m

```
% ------------------------------------------------------------------------
% Sequenz Wiedergabe an Zellspannungsgenerator V02
% ------------------------------------------------------------------------

load('hillermann_transient.mat');

% Verbindung herstellen
con = vbConnect(vbBoards.CELL3);

% Werte in Mikrovolt umwandeln
voltageArray = vbDouble2Microvolt(tr1_ds);

% Sequenz uebertragen
vbVoltageTransfer(con, voltageArray);

% Wiedergabe starten
vbCmd(con, 'start');

% Verbinden trennen
vbDisconnect(con);
```

auswertung.m

```
% ------------------------------------------------------------------------
% Sequenz Auswertung an Zellspannungsgenerator V02
% ------------------------------------------------------------------------

% CSV File importieren
scopeImport('scope_messung.csv');
plotData(:,1) = data(:,2);

% Originaldaten laden
load('hillermann_transient.mat');

% Zeitintervall
interval = 1 : recordLength;
time = interval' * sampleInterval;

% Differenz bilden
diff = plotData - tr1_org;

% Plot erstellen
figure(1);
[AX,H1,H2] = plotyy(time, plotData, time, diff) ;
hold all;

plot(AX(1), time, tr1_org, 'k');
legend({'Gemessen{~~~}', 'Ausgegeben', 'Differenz'}, 'Location', 'East');

set(AX(1), 'ylim', [2.800 3.500], 'ytick', 2.8:0.1:3.5);
set(AX(2), 'ylim', [-0.1 1]     , 'ytick', -0.05:0.05:0.05, 'YTickLabel', -50 : 50 :
    50);

title('Sequenz Vergleich am Zellspannungsgenerator V02');

grid on;
xlabel('t(s)');
```

```
34 ylabel(AX(1), 'U(V)');
   ylabel(AX(2), 'U(mV)');
36
   % PGF generieren
38 pgfFileName = 'sequenz_vergleich.pgf';
   matfig2pgf('filename', pgfFileName, 'figwidth', 12);
```

E. Controller Source-Files

adc_external.h

```
/*
 * adc_external.h
 *
 *  Created on: 16.07.2012
 *      Author: Tobias
 */

#ifndef ADC_EXTERNAL_H_
#define ADC_EXTERNAL_H_

int adcExternalInit();
unsigned char adcExternalGetStatus();
void adcExternalSetConfig(unsigned short value);
unsigned short adcExternalGetConfigRegister();
void adcExternalSetModeRegister(unsigned short value);
unsigned short adcExternalGetOffsetRegister();
void adcExternalSetOffsetRegister(unsigned short value);
unsigned short adcExternalGetFullscaleRegister();
void adcExternalSetFullscaleRegister(unsigned short value);
unsigned short adcExternalGetData();
long adcExternalGetSample();
unsigned int adcExternalGetVoltage();
unsigned short adcExternalGetValue();
void cmdAdcExternalGetValue();
void cmdAdcExternalGetVoltage();

#endif /* ADC_EXTERNAL_H_ */
```

adc_external.c

```
/*
 * adc_external.c
 * Routinen zur Abfrage des ADC basieren auf denen vom
 * Zellspannungsgenerator V01
 *
 *  Created on: 16.07.2012
 *      Author: Tobias
 */

#include "adc_external.h"
#include "spi.h"
#include "settings.h"
#include "ethernet.h"
#include <stdio.h>
#include <utils/uartstdio.h>

// ADC Wert aus letztem Aufruf von adcExternalGetSample()
volatile unsigned short adcExternalResult;

// Letzter ADC wert umgerechnet in Mikrovolt
volatile unsigned long adcExternalVoltage;

/**
```

```
 * Externen ADC initialisieren
 */
int adcExternalInit()
{
    UARTprintf("Initializing external ADC...");

    // Reset durchfuehren (32 Einsen senden)
    spiChipSelect(ADC);
    spiReadWrite(0xFF);
    spiReadWrite(0xFF);
    spiReadWrite(0xFF);
    spiReadWrite(0xFF);
    spiChipSelect(NONE);

    // Dummy Read des Statusregisters durchfuehren
    adcExternalGetStatus();

    // Setzen des "configuration register": unipolar mode, gain = 1, buffered, noref =
        0x1010
    adcExternalSetConfig(0x1010);

    // In den "power down" Modus gehen, um das "mode register" zu aendern
    adcExternalSetModeRegister(0x600A);

    // Initialisierung des "offset registers" mit 0
    adcExternalSetOffsetRegister(0);

    // Initialisierung des "full scale registers" mit 0
    adcExternalSetFullscaleRegister(0);

    // Pruefen, ob die Werte korrekt geschrieben wurden
    if(adcExternalGetOffsetRegister() != 0) {
        UARTprintf("failed: offset calibration reset failed!\n");
        return 0;
    }
    if(adcExternalGetFullscaleRegister() != 0) {
        UARTprintf("failed: full scale calibration reset failed!\n");
        return 0;
    }

    // Durchfuehrung der internen Kalibrierung des ADCs
    adcExternalSetModeRegister(0x800A); // Internal zero offset calibration
    while((adcExternalGetStatus() & (1<<7)));
    adcExternalSetModeRegister(0xA00A); // Internal full scale calibration
    while((adcExternalGetStatus() & (1<<7)));

    // Pruefen, ob die Kalibierung erfolgreich war (sich die Werte geaendert haben)
    if(adcExternalGetOffsetRegister() == 0) {
        UARTprintf("failed: offset calibration failed!\n");
        return 0;
    }
    if(adcExternalGetFullscaleRegister() == 0)  {
        UARTprintf("failed: full scale calibration failed!\n");
        return 0;
    }

    // Kontinuirliches lesen des Kanals 0
    adcExternalSetModeRegister(0x000A);

    // "config register" zur Ueberpruefung lesen
    if(adcExternalGetConfigRegister() != 0x1010)    {
        UARTprintf("failed: config register check failed!\n");
        return 0;
```

```
    }

    UARTprintf("done\n");
    return 1;
}

/*
 * Lesen des ADC Status Registers
 */
unsigned char adcExternalGetStatus()
{
    spiChipSelect(ADC);
    spiClearFifo();
    spiReadWrite(0x40);

    unsigned char statusReg = spiReadWrite(0x00);
    spiChipSelect(NONE);

    return statusReg;
}

/*
 * Schreiben des Konfigurations-Registers
 */
void adcExternalSetConfig(unsigned short value)
{
    spiChipSelect(ADC);
    spiReadWrite(0x10);
    spiReadWrite((unsigned char)(value>>8));
    spiReadWrite((unsigned char)(value&0xff));
    spiChipSelect(NONE);
}

/*
 * Auslesen des Konfigurations-Registers
 */
unsigned short adcExternalGetConfigRegister()
{
    unsigned short data;
    spiChipSelect(ADC);
    spiReadWrite(0x50);
    data = (spiReadWrite(0x00))<<8;
    data |= spiReadWrite(0x00);
    spiChipSelect(NONE);
    return data;
}

/*
* Schreiben des Mode-Registers
 */
void adcExternalSetModeRegister(unsigned short value)
{
    spiChipSelect(ADC);
    spiReadWrite(0x08);
    spiReadWrite((unsigned char)(value>>8));
    spiReadWrite((unsigned char)(value&0xff));
    spiChipSelect(NONE);
}

/*
 * Auslesen des Offset-Registers
 */
unsigned short adcExternalGetOffsetRegister()
```

```
149 {
        unsigned short data;
151     spiChipSelect(ADC);

153     spiReadWrite(0x70);
        data = (spiReadWrite(0x00))<<8;
155     data |= spiReadWrite(0x00);

157     spiChipSelect(NONE);
        return data;
159 }

161 /*
     * Setzen des Offset-Registers
163  */
    void adcExternalSetOffsetRegister(unsigned short value)
165 {
        spiChipSelect(ADC);
167     spiReadWrite(0x30);
        spiReadWrite((unsigned char)(value>>8));
169     spiReadWrite((unsigned char)(value));
        spiChipSelect(NONE);
171 }

173 /*
     * Auslesen des Fullscale-Registers
175  */
    unsigned short adcExternalGetFullscaleRegister()
177 {
        unsigned short data;
179     spiChipSelect(ADC);
        spiReadWrite(0x78);
181     data = (spiReadWrite(0x00))<<8;
        data |= spiReadWrite(0x00);
183     spiChipSelect(NONE);
        return data;
185 }

187 /*
     * Setzen des Fullscale-Registers
189  */
    void adcExternalSetFullscaleRegister(unsigned short value)
191 {
        spiChipSelect(ADC);
193     spiReadWrite(0x38);
        spiReadWrite((unsigned char)(value>>8));
195     spiReadWrite((unsigned char)(value));
        spiChipSelect(NONE);
197 }

199 /*
     * Auslesen der Daten
201  */
    unsigned short adcExternalGetData()
203 {
        unsigned short data;
205     spiChipSelect(ADC);
        spiReadWrite(0x58);
207     data = (spiReadWrite(0x00))<<8;
        data |= spiReadWrite(0x00);
209     spiChipSelect(NONE);
        return data;
211 }
```

```
213 /**
     * ADC Wert auslesen
215  */
unsigned short adcExternalGetValue()
217 {
    adcExternalGetData();                       // make shure last data is read
219     while(adcExternalGetStatus() & (1<<7));     // wait for new sample
    adcExternalResult = adcExternalGetData();
221
    return adcExternalResult;
223 }

225 /**
     * ADC Wert auslesen und in Spannung inkl. Gain- und Offsetkorrektur umrechnen
227  */
unsigned int adcExternalGetVoltage()
229 {
    unsigned int voltage = (unsigned long long)adcExternalGetValue() * settings.
        voltageGain / 1000000 + settings.voltageOffset;
231
    return voltage;
233 }

235 /**
     * Sample ausgeben
237  */
void cmdAdcExternalGetValue()
239 {
    char buffer[100];
241     sprintf(buffer, "ADC Value: %d\n", adcExternalGetValue());
    telnetWrite(buffer);
243 }

245 /**
     * Gemessene Spannung ausgeben
247  */
void cmdAdcExternalGetVoltage()
249 {
    char buffer[100];
251     unsigned int voltage = adcExternalGetVoltage();
    sprintf(buffer, "Measured voltage: %d\n", voltage);
253     telnetWrite(buffer);
}
```

calibrate.h

```
/*
2  * calibrate.h
   *
4  *  Created on: 21.07.2012
   *      Author: Tobias
6  */

8 #ifndef CALIBRATE_H_
#define CALIBRATE_H_
10
// Modi der automatischen Kalibrierung
12 typedef enum
```

```
{
    CALSTART,
    SETVOLTAGE,
    CALIBRATEDAC,
    SAVEDATA,
    CHECKNEXT,
    CALFINISH
} calibrate_handler_t;

void calibrateInit();
void cmdCalibrateAdcExternal(int argc, char *argv[]);
void Timer1AIntHandler();

#endif /* CALIBRATE_H_ */
```

calibrate.c

```
/*
 * calibrate.c
 *
 *  Created on: 21.07.2012
 *      Author: Tobias
 */

#include "calibrate.h"
#include "adc_external.h"
#include "dac.h"
#include "settings.h"
#include "ethernet.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <inc/hw_memmap.h>
#include <inc/hw_types.h>
#include <inc/hw_ints.h>
#include <driverlib/interrupt.h>
#include <driverlib/sysctl.h>
#include <driverlib/timer.h>

unsigned int dacCalibrateVoltage[13];
unsigned int voltageValue1;
unsigned int voltageValue2;
unsigned short adcValue1;
unsigned short adcValue2;

/**
 * Kalibrierung initialisieren
 */
void calibrateInit()
{
    // Enable Timer1
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

    // Configure Timer1 as a 32-bit periodic timer
    //TimerConfigure(TIMER1_BASE, TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_PERIODIC);
    TimerConfigure(TIMER1_BASE, TIMER_CFG_32_BIT_PER_UP);

    // Set the Timer1A load value to 0.8sec
    //TimerLoadSet(TIMER1_BASE, TIMER_A, SysCtlClockGet() * 8/10);
```

```
    TimerLoadSet(TIMER1_BASE, TIMER_A, SysCtlClockGet());

    // Configure the Timer1A interrupt for timer timeout
    TimerIntEnable(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

    // Enable the Timer1A interrupt on the processor
    IntEnable(INT_TIMER1A);

    // Set Timer1A to second highest priority
    IntPrioritySet(INT_TIMER1A, 0x20);

    // Define calibration voltages
    dacCalibrateVoltage[0]  =  100000;
    dacCalibrateVoltage[1]  =  500000;
    dacCalibrateVoltage[2]  = 1000000;
    dacCalibrateVoltage[3]  = 1500000;
    dacCalibrateVoltage[4]  = 2000000;
    dacCalibrateVoltage[5]  = 2500000;
    dacCalibrateVoltage[6]  = 3000000;
    dacCalibrateVoltage[7]  = 3500000;
    dacCalibrateVoltage[8]  = 4000000;
    dacCalibrateVoltage[9]  = 4500000;
    dacCalibrateVoltage[10] = 5000000;
    dacCalibrateVoltage[11] = 5500000;
    dacCalibrateVoltage[12] = 5600000;
}

/**
 * Befehls-Routine zum Durchfuehren der Kalibrierung
 */
void cmdCalibrateAdcExternal(int argc, char *argv[])
{
    if(argc == 0) {
        telnetWrite("Parameter missing\n");
        return;
    }

    // Start calibration
    if(strcmp(argv[0], "start") == 0) {
        telnetWrite("Starting calibration process...\n");

        // Reset calibration values
        int i;
        for(i=0; i<13; i++) {
            settings.dacTable[i].microvolt = dacCalibrateVoltage[i];
            settings.dacTable[i].dacValue = 0;
        }
        settings.voltageGain = 91000000;
        settings.voltageOffset = 0;

        // Set output voltage to 500mV
        telnetWrite("Setting output voltage to 500mV\n");
        dacSetVoltage(500000);

        telnetWrite("Please enter measured voltage in microvolt\n");
        telnetWrite("Syntax: calibrate lower <value>\n");

    // Set lower value
    } else if(strcmp(argv[0], "lower") == 0) {

        // Get entered voltage and read ADC value
        voltageValue1 = atoi(argv[1]);
        adcValue1 = adcExternalGetValue();
```

```
106
        // Set output voltage to 5.5V
108         telnetWrite("Setting output voltage to 5.5V\n");
            dacSetVoltage(5500000);
110
            telnetWrite("Please enter measured voltage in microvolt\n");
112         telnetWrite("Syntax: calibrate upper <value>\n");

114     // Set upper value
        } else if(strcmp(argv[0], "upper") == 0) {
116
            // Get entered voltage and read ADC value
118         voltageValue2 = atoi(argv[1]);
            adcValue2 = adcExternalGetValue();
120
            // Calculate offset
122         int offset = (float)(adcValue2 * voltageValue1 - adcValue1 * voltageValue2) /
                (float)(adcValue2 - adcValue1);
            settings.voltageOffset = offset;
124
            // Calculate gain
126         int gain = (((float)voltageValue1 - offset) / (float)adcValue1) * 1000000;
            settings.voltageGain = gain;
128
            UARTprintf("ADC value for %d microvolt: %d\n", voltageValue1, adcValue1);
130         UARTprintf("ADC value for %d microvolt: %d\n", voltageValue2, adcValue2);
            UARTprintf("offset: %d\n", offset);
132         UARTprintf("gain: %d\n", gain);

134         // Start timer for automatic DAC calibration
            TimerEnable(TIMER1_BASE, TIMER_A);
136
        } else {
138         telnetWrite("Unknown parameter\n");
        }
140 }

142 /**
     * Interrupt Handler fuer automatische DAC Kalibrierung
144  */
    void Timer1AIntHandler()
146 {
        static volatile calibrate_handler_t handlerState = CALSTART;
148     static unsigned char step = 0;
        static int diff = 0;
150     dac_calibration_t valuePair;

152     switch(handlerState) {
            case CALSTART:
154             telnetWrite("Running automatic DAC calibration");
                handlerState = SETVOLTAGE;
156             break;

158         case SETVOLTAGE:
                dacSetVoltage(dacCalibrateVoltage[step]);
160             handlerState = CALIBRATEDAC;
                break;
162
            case CALIBRATEDAC:
164             telnetWrite(".");
                diff = dacCalibrateVoltage[step] - adcExternalGetVoltage();
166
                if(diff < 90) {
```

```
                handlerState = SAVEDATA;
            } else if(diff > 0) {
                dacCorrectValue(diff/90);
            } else {
                dacCorrectValue(-diff/90);
            }
            break;

        case SAVEDATA:
            valuePair.microvolt = dacCalibrateVoltage[step];
            valuePair.dacValue = dacGetValue();
            settings.dacTable[step] = valuePair;
            step++;
            handlerState = CHECKNEXT;
            break;

        case CHECKNEXT:
            if(step < 13) {
                handlerState = SETVOLTAGE;
            } else {
                handlerState = CALFINISH;
            }
            break;

        case CALFINISH:
            TimerDisable(TIMER1_BASE, TIMER_A);
            dacSetValue(0);
            step = 0;
            settingsSave();
            telnetWrite("done\n$");
            break;
    }

    // Clear the timer interrupt
    TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);
}
```

cmd_help.h

```
/*
 * cmd_help.h
 *
 *  Created on: 23.07.2012
 *      Author: Tobias
 */

#ifndef CMD_HELP_H_
#define CMD_HELP_H_

const char help_string[] = "Help: \n"
        "  temp                    Abfrage der Board-Temperatur\n"
        "  adcvolt                 Gemessene Spannung des ADC abfragen\n"
        "  adcval                  16Bit Wert vom ADC abfragen\n"
        "  sdtest                  SDRAM Test durchfuehren\n"
        "  readpos                 Einzelnen Spannungswert abfragen (Parameter: index
            )\n"
        "  voltagesequence         In den Binaeruebertragungsmodus wecheln\n"
        "  voltagechecksum         Checksumme der uebertragenen Spannungssequenz
            abfragen\n"
```

```
        "  dac                     DAC auf 16Bit Wert setzen (Parameter: dacval)\n"
        "  voltage                 DAC auf Ausgangsspannung setzen (Parameter:
             microvolt)\n"
        "  opon                    Ausgang vom Power-OP aktivieren\n"
        "  opoff                   Ausgang vom Power-OP deaktivieren\n"
        "  start                   Transienten-Wiedergabe starten\n"
        "  stop                    Transienten-Wiedergabe stoppen\n"
        "  curon                   Shunt-Widerstaende aktivieren\n"
        "  curoff                  Shunt-Widerstaende deaktivieren\n"
        "  bufon                   Puffer-Kondensatoren aktivieren\n"
        "  bufoff                  Puffer-Kondensatoren deaktvieren\n"
        "  settings                Board-Einstellungen abfragen (optionale Parameter:
             load, save)\n"
        "  syncmode                Synchronisations-Modus aendern (Parameter:
             standalone, master, slave)\n"
        "  outputmode              Ausgabe-Modus aendern (Parameter: static, single,
             repeat)\n"
        "  calibrate               ADC und DAC Kalibrierung starten (Parameter: start
             , lower, upper)\n"
        "  help                    Diese Befehlsuebersicht ausgeben\n"
        "  exit                    Steuerverbindung beenden\n";

#endif /* CMD_HELP_H_ */
```

command.h

```
/*
 * command.h
 *
 *  Created on: 11.07.2012
 *      Author: Tobias
 */

#ifndef COMMAND_H_
#define COMMAND_H_

#include <lwip/tcp.h>

#define ASCII_SPACE 32

/**
 * Eintrag in Command Table mit Befehl und Funktionspointer
 */
typedef struct {
    char *cmd;
    void (*cmdFuncPtr)(int argc, char *argv[]);
} command_table_t;

/**
 * Empfangsmodi der Telnet Schnittstelle
 */
typedef enum
{
    RECEIVE_CMD,            // Commands empfangen
    RECEIVE_ASCII_VALUES,   // Spannungswerte im ASCII-Format empfangen
    RECEIVE_BINARY_VALUES   // Spannungswerte im Binaer-Format empfangen
} rx_mode_t;

void commandInit();
```

```
void commandAddCmd(char *cmdName, void (*cmdFuncPtr)(int argc, char *argv[]));
void commandReceive(struct pbuf *packet);
void commandRecvCmd(struct pbuf *packet);
void commandExecute(char *cmdString);
void commandReceiveBinaryValues(struct pbuf *packet);
void commandSetReceiveMode(rx_mode_t mode);

void cmdVoltagesequence();
void cmdVoltagechecksum();
void cmdHelp();
void cmdTest();

#endif /* COMMAND_H_ */
```

command.c

```
/*
 * command.c
 *
 *  Created on: 11.07.2012
 *      Author: Tobias
 */

#include "command.h"

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <utils/uartstdio.h>

#include "helper.h"
#include "cmd_help.h"
#include "ethernet.h"
#include "sdram.h"
#include "settings.h"
#include "led.h"
#include "dac.h"
#include "relais.h"
#include "rs485.h"
#include "output.h"
#include "adc_external.h"
#include "calibrate.h"
#include "temp.h"


command_table_t *cmdTable;
int cmdTableCnt = 0;
volatile rx_mode_t rxMode;

char buffer[100];
char *bufString;

/**
 * Alle erlaubten Befehle werden zur cmd_table hinzugefuegt
 */
void commandInit()
{
    commandAddCmd("test", &cmdTest);
```

```
    commandAddCmd("sdtest", &sdramTest);
    commandAddCmd("voltagesequence", &cmdVoltagesequence);
    commandAddCmd("exit", &telnetExit);
    commandAddCmd("voltagechecksum", &cmdVoltagechecksum);
    commandAddCmd("read", &sdramReadVoltage);
    commandAddCmd("readpos", &sdramReadPos);
    commandAddCmd("dac", &cmd_dac);
    commandAddCmd("voltage", &cmd_voltage);
    commandAddCmd("opon", &cmdOutputEnable);
    commandAddCmd("opoff", &cmdOutputDisable);
    commandAddCmd("rs", &rs485SendSyncCommand);
    commandAddCmd("start", &cmdOutputStart);
    commandAddCmd("stop", &cmdOutputStop);
    commandAddCmd("curon", &cmdShuntEnable);
    commandAddCmd("curoff", &cmdShuntDisable);
    commandAddCmd("bufon", &cmdBufferEnable);
    commandAddCmd("bufoff", &cmdBufferDisable);
    commandAddCmd("adcval", &cmdAdcExternalGetValue);
    commandAddCmd("adcvolt", &cmdAdcExternalGetVoltage);
    commandAddCmd("settings", &cmdSettings);
    commandAddCmd("syncmode", &cmdSetSyncMode);
    commandAddCmd("outputmode", &cmdSetOutputMode);
    commandAddCmd("calibrate", &cmdCalibrateAdcExternal);
//  commandAddCmd("adjon", &outputEnableAutoAdjustment);        // TESTING
//  commandAddCmd("adjoff", &outputDisableAutoAdjustment);      // TESTING
    commandAddCmd("temp", &cmdGetTemperature);
    commandAddCmd("help", &cmdHelp);

    commandSetReceiveMode(RECEIVE_CMD);
}

/**
 * Hinzufuegen von Befehlen sowie ihrem Funktionspointer zur cmdTable
 */
void commandAddCmd(char *cmdName, void (*cmdFuncPtr)(int argc, char *argv[]))
{
    void *tmp = realloc(cmdTable, (cmdTableCnt+1) * sizeof(command_table_t));
    cmdTable = (command_table_t*)tmp;

    cmdTable[cmdTableCnt].cmd = cmdName;
    cmdTable[cmdTableCnt].cmdFuncPtr = cmdFuncPtr;
    cmdTableCnt++;
}

/**
 * Empfangsroutine fuer TCP Pakete
 * Pakete je nach Empfangsmodus an entpr. Routine uebergeben
 */
void commandReceive(struct pbuf *packet)
{
    // Empfangene Daten je nach Empfangsmodus an entspr. Funktion uebergegben
    switch(rxMode) {
        case RECEIVE_CMD:
            commandRecvCmd(packet);
            break;

        case RECEIVE_ASCII_VALUES:
            //control_recv_ascii_values(p);
            break;

        case RECEIVE_BINARY_VALUES:
            //control_recv_binary_values(p);
            commandReceiveBinaryValues(packet);
```

```
            break;
    }
}

/**
 * Commands empfangen
 */
void commandRecvCmd(struct pbuf *packet)
{
    struct pbuf *current_packet = packet;
    static char fifo[200];
    static unsigned char rd = 0, wr = 0;
    static char cmd_buffer[100];
    static unsigned char cmd_len = 0;
    short i = 0;

    // copy received payload to rx fifo
    do {
        for(i=0; i<current_packet->len; i++) {
            fifo[(wr+i) % 200] = ((char*)current_packet->payload)[i];
        }
        wr = (wr + current_packet->len) % 200;

    } while((current_packet=current_packet->next) != NULL);

    // parse fifo buffer for line breaks and give cmd_buffer to cmd_interface
    while (rd != wr)
    {
        if(fifo[rd] != '\n') {

            // put fifo content (without carriage returns) to cmd_buffer
            if(fifo[rd] != '\r') {
                cmd_buffer[cmd_len+1] = cmd_buffer[cmd_len];
                cmd_buffer[cmd_len] = fifo[rd];
                cmd_len++;
            }

        } else {

            cmd_buffer[cmd_len] = 0;
            commandExecute(cmd_buffer);
            cmd_len = 0;
            telnetWrite("$");
        }
        rd = (rd + 1) % 200;
    }
}

/**
 * Befehlsverarbeitung inkl. Parametern
 * Der cmdString wird in Command sowie eventuelle Parameter zerlegt
 * Wenn Command in cmdTable vorhanden, wird zugehoeriger Funktionspointer
 * mit uebergebenen Parametern aufgerufen
 */
void commandExecute(char *cmdString)
{
    int i;
    int str_len = strlen(cmdString);
    char *buffer;
    char *cmd;
    int argc = 0;
    char *argv[10];
```

```
170     //UARTprintf("string execute start\n");
        UARTprintf("Command String: %s\n", cmdString);
172
        //char *buf_string = (char*)malloc(str_len*sizeof(char));
174     bufString = (char*)malloc(str_len*sizeof(char));
        strcpy(bufString, cmdString);
176
        // initialize strtok() and extract command
178     buffer = strtok(bufString, " ");
        cmd = buffer;
180
        // split arguments on whitespace
182     while(buffer != NULL) {
            buffer = strtok(NULL, " ");
184         argv[argc] = buffer;
            argc++;
186     }
        argc--;
188
        // lookup command in cmd_table
190     bool cmdFound = false;
        for(i=0; i<cmdTableCnt; i++) {
192         if(strcmp(cmd, cmdTable[i].cmd) == 0) {
                cmdTable[i].cmdFuncPtr(argc, argv);
194             cmdFound = true;
                break;
196         }
        }
198     free(bufString);

200     if(!cmdFound) {
            UARTprintf("cmd not found\n");
202         telnetWrite("Command not found\n");
        }
204
        //UARTprintf("string execute end\n");
206 }

208 /**
     * Spannungswerte im Binaerformat empfangen
210  */
    void commandReceiveBinaryValues(struct pbuf *packet)
212 {
        struct pbuf *current_packet = packet;
214     static char value_buffer[3];
        static char value_part = 0;
216     short i = 0;

218     ledToggle(LED1);    // blink while transmitting

220     // Walk through payload while receiving packages
        do {
222         while(i < current_packet->len) {

224             // Write rerceived bytes to temporary buffer
                value_buffer[value_part++] = ((char*)current_packet->payload)[i++];
226
                // If temp buffer holds three bytes, assign them to 24 bit value
228             if(value_part == 3) {
                    long value = (value_buffer[0] << 16) | (value_buffer[1] << 8) |
                        value_buffer[2];
230
```

```
                //UARTprintf("voltage bytes: %d %d %d\n", value_buffer[0],
                    value_buffer[1], value_buffer[2]);
232             //UARTprintf("voltage: %d\n", value);

234             // If no end sequence received, save voltages in sdram
                if(value != 16777215) {
236                 sdramWriteVoltage(value);
                } else {
238                 ////voltage_finish();
                    commandSetReceiveMode(RECEIVE_CMD);
240                 ledEnable(LED1);
                }
242             value_part = 0;
            }
244     }
        i = 0;
246
    } while((current_packet=current_packet->next) != NULL);
248
    telnetWrite(">");       // send acknowledgement byte
250 }

252 /**
 * Empfangsmodus setzen
254  */
void commandSetReceiveMode(rx_mode_t mode)
256 {
    rxMode = mode;
258 }

260 /**
 * In Binaeruebertragungsmodus wechseln
262  */
void cmdVoltagesequence()
264 {
    UARTprintf("Switching in BINARY mode\n");
266     sdramResetPointer();
    commandSetReceiveMode(RECEIVE_BINARY_VALUES);
268 }

270 /**
 * Ausgabe der Checksumme der empfangenen Spannungen nach dem Befehl voltagechecksum.
272  * Der Wert wird ueben den TCP-Socket mit einer Laenge von 13 Zeichen ausgegeben.
 */
274 void cmdVoltagechecksum()
{
276     UARTprintf("voltagechecksum called\n");
    unsigned long long checksum = sdramGetVoltageChecksum();
278
    UARTprintf("voltage checksum: %s\n", itoa(checksum));
280
    char buffer[15];
282     sprintf(buffer, "%s\n", itoa(checksum));

284     char vchecksum[15];
    vchecksum[13] = '\n';
286     vchecksum[14] = 0;
    unsigned char zeros = 13-strlen(buffer);
288     unsigned char i;

290     for(i=0; i<13; i++) {

292         if(i <= zeros) {
```

```
            vchecksum[i] = '0';
        } else {
            vchecksum[i] = buffer[i-zeros-1];
        }
    }

    telnetWrite(vchecksum);
}

// Kommandouebersicht ausgeben
void cmdHelp()
{
    telnetWrite(help_string);
}

void cmdTest(int argc, char *argv[])
{
    UARTprintf("command test\n");

    ledToggle(LED1);
    ledToggle(LED2);
    ledToggle(LED3);
    ledToggle(LED4);
    ledToggle(LED5);
    ledToggle(LED6);
}
```

dac.h

```
/*
 * dac.h
 *
 *  Created on: 17.07.2012
 *      Author: Tobias
 */

#ifndef DAC_H_
#define DAC_H_

void dacSetValue(unsigned short value);
unsigned short dacGetValue();
void dacCorrectValue(short value);
void cmd_dac(int argc, char *argv[]);
void dacSetVoltage(unsigned int microvolt);
unsigned int dacGetVoltage();
void cmd_voltage(int argc, char *argv[]);

#endif /* DAC_H_ */
```

dac.c

```
/*
 * dac.c
 *
 *  Created on: 17.07.2012
```

```
 *      Author: Tobias
 */

#include "dac.h"
#include "spi.h"
#include "settings.h"

#include <stdlib.h>
#include <inc/hw_types.h>
#include <utils/uartstdio.h>

unsigned short dacValue;
unsigned int dacVoltage;

/**
 * Wert an DAC senden
 */
void dacSetValue(unsigned short value)
{
    spiChipSelect(DAC);
    spiReadWrite(value >> 8);
    spiReadWrite(value & 0xFF);
    spiChipSelect(NONE);
}

/**
 * Zuletzt gesetzten DAC Wert abfragen
 */
unsigned short dacGetValue()
{
    return dacValue;
}

/**
 * Spannung per DAC inkl. Korrektur durch Wertetabelle (wenn vorhanden) ausgeben
 * Full Scale: 6000000 > 65535
 */
void dacSetVoltage(unsigned int microvolt)
{
    dacVoltage = microvolt;
    tBoolean calValue;
    unsigned int regionVolt;
    int i;

    // Check if voltage is out of calibration range
    if(microvolt < settings.dacTable[0].microvolt || microvolt > settings.dacTable
        [12].microvolt) {
        calValue = false;

    // Check if calibration values exist
    } else {
        for(i=0; i<13; i++) {
            if(settings.dacTable[i].microvolt > microvolt) {

                if(settings.dacTable[i].dacValue == 0) {
                    calValue = false;
                } else {
                    calValue = true;
                }
                break;
            }
        }
    }
```

```
67
    // Output with calibration corrections
69  if(calValue) {
        regionVolt = microvolt - settings.dacTable[i-1].microvolt;
71      dacValue = (settings.dacTable[i].dacValue - settings.dacTable[i-1].dacValue) *
             regionVolt / 500000 + settings.dacTable[i-1].dacValue;

73  // Output without calibration corrections
    } else {
75      dacValue = (microvolt * (unsigned long long)65535) / (unsigned long long)
            6000000;
    }
77
    dacSetValue(dacValue);
79  //UARTprintf("dac value: %d\n", dacValue);
}
81
/**
83 * Zuletzt gesetzte DAC Spannung abfragen
 */
85 unsigned int dacGetVoltage()
{
87      return dacVoltage;
}
89
/**
91 * DAC Value um ganzzahlen Wert anpassen
 */
93 void dacCorrectValue(short value)
{
95      dacValue += value;
    dacSetValue(dacValue);
97  UARTprintf("new dac value: %d\n", dacValue);
}
99
/**
101 * Wert per DAC ausgeben
 */
103 void cmd_dac(int argc, char *argv[])
{
105     unsigned short value = atoi(argv[0]);
    UARTprintf("dac value: %d\n", value);
107
    dacSetValue(value);
109 }

111 /**
    * Spannung per DAC ausgeben
113  */
    void cmd_voltage(int argc, char *argv[])
115 {
    unsigned int value = atoi(argv[0]);
117 UARTprintf("cmd voltage: %d\n", value);

119     dacSetVoltage(value);
    }
```

ethernet.h

```
/*
 * ethernet.h
 *
 *  Created on: 09.07.2012
 *      Author: Tobias
 */

#ifndef ETHERNET_H_
#define ETHERNET_H_

#include <utils/uartstdio.h>
#include <lwip/tcp.h>

#define SYSTICKHZ               100
#define SYSTICKMS               (1000 / SYSTICKHZ)

int ethernetInit();
void lwIPHostTimerHandler(void);
void SysTickIntHandler(void);

void ethernetInitTcpControl();
err_t ethernetTcpNewConnection(void* arg, struct tcp_pcb* newpcb, err_t err);
err_t ethernetTcpRx(void* arg, struct tcp_pcb* tpcb, struct pbuf* p, err_t err);
void telnetWrite(const void *data);
void telnetExit();

#endif /* ETHERNET_H_ */
```

ethernet.c

```
/*
 * ethernet.c
 *
 *  Created on: 09.07.2012
 *      Author: Tobias
 */

#include "ethernet.h"

#include "inc/lm3s9b96.h"
#include <inc/hw_types.h>
#include <driverlib/rom.h>
#include <driverlib/sysctl.h>
#include <driverlib/gpio.h>
#include <inc/hw_memmap.h>
#include <utils/locator.h>
#include <utils/lwiplib.h>
#include <inc/hw_ints.h>
#include <driverlib/interrupt.h>
#include <lwip/ip_addr.h>
#include <stdlib.h>
#include <string.h>
#include "command.h"
#include "settings.h"
#include "led.h"

// MAC Adresse
unsigned long long macAddress;
unsigned char pucMACArray[8];
```

```
31  // IP Adresse
    unsigned long ipAddress;
33
    // Globale Variablen fuer lwIPHostTimerHandler
35  static char g_pcTwirl[4] = { '\\', '|', '/', '-' };
    static unsigned long g_ulTwirlPos = 0;
37  static unsigned long g_ulLastIPAddr = 0;

39  // TCP control block
    struct tcp_pcb* controlTpcb;
41  struct tcp_pcb* controlConnection;

43  /**
     * Initialize ethernet controller
45   */
    int ethernetInit()
47  {
        UARTprintf("Initializing ethernet controller:\n");
49
        macAddress = settings.macAddress;
51      ipAddress = settings.ipAddress;

53      // Enable and reset ethernet controller
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ETH);
55      ROM_SysCtlPeripheralReset(SYSCTL_PERIPH_ETH);

57      // Enable port F for link status/activity leds
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
59      GPIOPinConfigure(GPIO_PF2_LED1);
        GPIOPinConfigure(GPIO_PF3_LED0);
61      GPIOPinTypeEthernetLED(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);

63      // Configure SysTick for a periodic interrupt
        ROM_SysTickPeriodSet(ROM_SysCtlClockGet() / SYSTICKHZ);
65      ROM_SysTickEnable();
        ROM_SysTickIntEnable();
67
        // lower priority
69      IntPrioritySet(INT_ETH, 0xE0);
        IntPrioritySet(INT_SYSCTL, 0xE0);
71
        // MAC Adresse zuweisen
73      pucMACArray[0] = ((macAddress >> 40) & 0xff);
        pucMACArray[1] = ((macAddress >> 32) & 0xff);
75      pucMACArray[2] = ((macAddress >> 24) & 0xff);
        pucMACArray[3] = ((macAddress >> 16) & 0xff);
77      pucMACArray[4] = ((macAddress >>  8) & 0xff);
        pucMACArray[5] = ((macAddress >>  0) & 0xff);
79
        // Initialize lwIP stack with DHCP
81      lwIPInit(pucMACArray, ipAddress, 0xFFFFFF00, 0, IPADDR_USE_STATIC);

83      return 0;
    }
85
    /**
87   * Required by lwIP library to support any host-related timer functions
     */
89  void lwIPHostTimerHandler(void)
    {
91      unsigned long ulIPAddress;
```

```
93      // Get the local IP address.
        ulIPAddress = lwIPLocalIPAddrGet();
95
        // See if an IP address has been assigned.
97      if(ulIPAddress == 0)
        {
99          // Draw a spinning line to indicate that the IP address is being
            // discoverd.
101         UARTprintf("\b%c", g_pcTwirl[g_ulTwirlPos]);

103         // Update the index into the twirl.
            g_ulTwirlPos = (g_ulTwirlPos + 1) & 3;
105     }

107     // Check if IP address has changed, and display if it has
        else if(ulIPAddress != g_ulLastIPAddr)
109     {
            // Display the new MAC address
111         UARTprintf("\n  MAC: %x:%x:%x:%x:%x:%x\n", pucMACArray[0],
                    pucMACArray[1], pucMACArray[2], pucMACArray[3],
113                 pucMACArray[4], pucMACArray[5]);

115         // Display the new IP address
            UARTprintf("  IP: %d.%d.%d.%d\n", ulIPAddress & 0xff,
117                 (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
                    (ulIPAddress >> 24) & 0xff);
119
            //  Save the new IP address
121         g_ulLastIPAddr = ulIPAddress;

123         // Display the new network mask
            ulIPAddress = lwIPLocalNetMaskGet();
125         UARTprintf("  Netmask: %d.%d.%d.%d\n", ulIPAddress & 0xff,
                    (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
127                 (ulIPAddress >> 24) & 0xff);

129         // Display the new gateway address
            ulIPAddress = lwIPLocalGWAddrGet();
131         UARTprintf("  Gateway: %d.%d.%d.%d\n", ulIPAddress & 0xff,
                    (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
133                 (ulIPAddress >> 24) & 0xff);

135         ethernetInitTcpControl();

137         UARTprintf("Ethernet initialization complete...\n\n");
        }
139 }

141 /**
     * SysTick Handler fuehrt regelmaessige lwIP Aktionen aus
143  */
    void SysTickIntHandler(void)
145 {
        lwIPTimer(SYSTICKMS);
147 }

149 /**
     * TCP Server initialisieren
151  */
    void ethernetInitTcpControl()
153 {
        UARTprintf("\nInitializing TCP control...");
155
```

```
    controlTpcb = tcp_new();
157 tcp_bind(controlTpcb, IP_ADDR_ANY, 56936);
    controlTpcb = tcp_listen(controlTpcb);
159 tcp_accept(controlTpcb, &ethernetTcpNewConnection);

161 UARTprintf("done\n");
}
163
/**
165  * Funktion wird bei TCP-Verbindungsaufbau aufgerufen
 */
167 err_t ethernetTcpNewConnection(void* arg, struct tcp_pcb* newpcb, err_t err)
{
169     if(controlConnection)
        tcp_close(controlConnection);
171     controlConnection = newpcb;
    tcp_accepted(newpcb);
173     tcp_recv(newpcb, &ethernetTcpRx);

175     UARTprintf("New Telnet connection established!\n");

177     commandSetReceiveMode(RECEIVE_CMD);
    telnetWrite("$");       // send prompt
179     return ERR_OK;
}
181
/**
183  * Empfang und Bestaetigung der TCP-Pakete
 */
185 err_t ethernetTcpRx(void* arg, struct tcp_pcb* tpcb, struct pbuf* p, err_t err)
{
187     if(p == NULL)
    {
189         UARTprintf("Telnet connection closed!\n");
        controlConnection = NULL;
191         return ERR_OK;
    }
193
    commandReceive(p);
195
    tcp_recved(tpcb, p->tot_len);
197     pbuf_free(p);

199     return ERR_OK;
}
201
/**
203  * Daten ueber Telnet Verbindung ausgeben
 */
205 void telnetWrite(const void *data)
{
207     tcp_write(controlConnection, data, strlen(data), TCP_WRITE_FLAG_COPY |
        TCP_WRITE_FLAG_MORE);
}
209
/**
211  * Telnet Verbindung beenden
 */
213 void telnetExit()
{
215     tcp_close(controlConnection);
}
```

helper.h

```
/*
 * helper.h
 *
 *  Created on: 21.07.2012
 *      Author: Tobias
 */

#ifndef HELPER_H_
#define HELPER_H_

char* itoa(unsigned long long z);

#endif /* HELPER_H_ */
```

helper.c

```
/*
 * helper.c
 *
 *  Created on: 21.07.2012
 *      Author: Tobias
 */

#include "helper.h"

#include <stdlib.h>

/**
 * Rueckgabe eines 64 Bit Integers als String
 * Quelle: http://www.mikrocontroller.net/articles/FAQ#Eigene_Umwandlungsfunktionen
 */
char* itoa(unsigned long long z)
{
    int i = 0;
    int j;
    char tmp;
    char *buffer;

    buffer = malloc(21*sizeof(char));

    // die einzelnen Stellen der Zahl berechnen
    do {
        buffer[i++] = '0' + z % 10;
        z /= 10;
    } while( z > 0 );

    // den String in sich spiegeln
    for( j = 0; j < i / 2; ++j ) {
        tmp = buffer[j];
        buffer[j] = buffer[i-j-1];
        buffer[i-j-1] = tmp;
    }
    buffer[i] = '\0';

    return buffer;
}
```

led.h

```
/*
 * led.h
 *
 *  Created on: 13.07.2012
 *      Author: Tobias
 */

#ifndef LED_H_
#define LED_H_

typedef enum
{
    LED1,
    LED2,
    LED3,
    LED4,
    LED5,
    LED6
} led_num_t;

void ledInit();
void ledEnable(led_num_t led);
void ledDisable(led_num_t led);
void ledToggle(led_num_t led);

#endif /* LED_H_ */
```

led.c

```
/*
 * led.c
 *
 *  Created on: 13.07.2012
 *      Author: Tobias
 */

#include "led.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_epi.h"
#include "inc/hw_gpio.h"
#include <inc/hw_ints.h>
#include <driverlib/interrupt.h>
#include <driverlib/timer.h>
#include <driverlib/gpio.h>
#include <driverlib/sysctl.h>
#include <driverlib/rom.h>
#include <utils/uartstdio.h>
#include <stdlib.h>

/**
 * LED Ports initialisieren
 */
void ledInit()
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
```

```
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_2);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_3);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_4);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_5);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_2);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_3);

    ledDisable(LED1);
    ledDisable(LED2);
    ledDisable(LED3);
    ledDisable(LED4);
    ledDisable(LED5);
    ledDisable(LED6);
}

/**
 * LEDs aktivieren
 */
void ledEnable(led_num_t led)
{
    switch (led) {
        case LED1:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_2, 0);
            break;
        case LED2:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_3, 0);
            break;
        case LED3:
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_3, 0);
            break;
        case LED4:
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
            break;
        case LED5:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0);
            break;
        case LED6:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0);
            break;
        default:
            break;
    }
}

/**
 * LEDs deaktivieren
 */
void ledDisable(led_num_t led)
{
    switch (led) {
        case LED1:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_2, GPIO_PIN_2);
            break;
        case LED2:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_3, GPIO_PIN_3);
            break;
        case LED3:
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_3, GPIO_PIN_3);
            break;
        case LED4:
```

```
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, GPIO_PIN_2);
            break;
        case LED5:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, GPIO_PIN_4);
            break;
        case LED6:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, GPIO_PIN_5);
            break;
        default:
            break;
    }
}

/**
 * LEDs togglen
 */
void ledToggle(led_num_t led)
{
    switch (led) {
        case LED1:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_2, ~ROM_GPIOPinRead(
                GPIO_PORTD_BASE, GPIO_PIN_2));
            break;
        case LED2:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_3, ~ROM_GPIOPinRead(
                GPIO_PORTD_BASE, GPIO_PIN_3));
            break;
        case LED3:
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_3, ~ROM_GPIOPinRead(
                GPIO_PORTE_BASE, GPIO_PIN_3));
            break;
        case LED4:
            ROM_GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, ~ROM_GPIOPinRead(
                GPIO_PORTE_BASE, GPIO_PIN_2));
            break;
        case LED5:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, ~ROM_GPIOPinRead(
                GPIO_PORTB_BASE, GPIO_PIN_4));
            break;
        case LED6:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, ~ROM_GPIOPinRead(
                GPIO_PORTB_BASE, GPIO_PIN_5));
            break;
        default:
            break;
    }
}
```

main.c

```
/*
 * main.c
 *
 *  Created on: 25.06.2012
 *      Author: Tobias
 */

#include "inc/lm3s9b96.h"
#include "inc/hw_ints.h"
```

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"

#include "uart.h"
#include "utils/uartstdio.h"

#include "settings.h"
#include "led.h"
#include "sdram.h"
#include "ethernet.h"
#include "spi.h"
#include "dac.h"
#include "command.h"
#include "adc_external.h"
#include "relais.h"
#include "rs485.h"
#include "calibrate.h"
#include "output.h"
#include "temp.h"


// main program
int main(void)
{
    // Set clock to 50 MHz
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
        SYSCTL_XTAL_16MHZ);

    // Initialize components
    ledInit();
    uartInit();
    settingsInit();
    sdramInit();
    ethernetInit();
    commandInit();
    spiInit();
    adcExternalInit();
    dacSetValue(0);
    relaisInit();
    rs485Init();
    calibrateInit();
    outputInit();
    tempInit();

    ledEnable(LED1);


    // Enable interrupts
    ROM_IntMasterEnable();

    // Loop forever
    while(1)
    {
    }
}
```

output.h

```
/*
 * output.h
 *
 *  Created on: 17.07.2012
 *      Author: Tobias
 */

#ifndef OUTPUT_H_
#define OUTPUT_H_

void outputInit();
void cmdOutputEnable();
void cmdOutputDisable();
void outputEnableAutoAdjustment();
void outputDisableAutoAdjustment();
void cmdOutputStart();
void cmdOutputStop();
void outputNextSample();
void Timer0BIntHandler();
void Timer2AIntHandler();
void cmdBufferEnable();
void cmdBufferDisable();
void cmdShuntEnable();
void cmdShuntDisable();

#endif /* OUTPUT_H_ */
```

output.c

```
/*
 * output.c
 *
 *  Created on: 17.07.2012
 *      Author: Tobias
 */

#include "output.h"
#include "settings.h"
#include "ethernet.h"
#include "led.h"
#include "rs485.h"
#include "sdram.h"
#include "dac.h"
#include "adc_external.h"
#include "relais.h"

#include <inc/hw_memmap.h>
#include <inc/hw_types.h>
#include <inc/hw_ints.h>
#include <driverlib/interrupt.h>
#include <driverlib/gpio.h>
#include <driverlib/sysctl.h>
#include <driverlib/rom.h>
#include <utils/uartstdio.h>
#include <inc/lm3s9b92.h>
#include <driverlib/timer.h>

```

```
// Index des auszugebenden Samples
volatile unsigned int outputSampleNum;


/**
 * Ausgangsstufe initialisieren
 */
void outputInit()
{
    UARTprintf("Initializing output stage...");

    // Setup Power-OP control pins
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_5);
    ROM_GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_6);
    ROM_GPIOPinTypeGPIOInput(GPIO_PORTD_BASE, GPIO_PIN_7);

    /**
     * Output timer settings
     */

    // Enable Timer0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

    // Configure Timer0B as a 16-bit periodic timer
    TimerConfigure(TIMER0_BASE, TIMER_CFG_16_BIT_PAIR | TIMER_CFG_B_PERIODIC);

    // Set the Timer0B load value to 20kHz
    TimerLoadSet(TIMER0_BASE, TIMER_B, SysCtlClockGet() / 20000);

    // Configure the Timer0B interrupt for timer timeout
    TimerIntEnable(TIMER0_BASE, TIMER_TIMB_TIMEOUT);

    // Enable the Timer0B interrupt on the processor
    IntEnable(INT_TIMER0B);

    // Set Timer0B to highest priority
    IntPrioritySet(INT_TIMER0B, 0x00);

    /**
     * Adjustment timer settings
     */

    // Enable Timer2
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);

    // Configure Timer2 as a 32-bit periodic timer
    TimerConfigure(TIMER2_BASE, TIMER_CFG_32_BIT_PER_UP);

    // Set the Timer2A load value to 1sec
    TimerLoadSet(TIMER2_BASE, TIMER_A, SysCtlClockGet() / 1);

    // Configure the Timer2A interrupt for timer timeout
    TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

    // Enable the Timer2A interrupt on the processor
    IntEnable(INT_TIMER2A);

    // Enable buffer in static mode
    if(settings.outputMode == STATIC) {
        cmdBufferEnable();
    }
```

```
92
    cmdOutputEnable();
94
    UARTprintf("done\n");
96  }

98  /**
     * Power-OP Output aktivieren
100  */
    void cmdOutputEnable()
102 {
        ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_5, GPIO_PIN_5);
104 }

106 /**
     * Power-OP Output deaktivieren
108  */
    void cmdOutputDisable()
110 {
        ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_5, 0);
112 }

114 /**
     * Ausgangskondensator aktivieren
116  */
    void cmdBufferEnable()
118 {
        relaisClose(BUFFER);
120     ledDisable(LED3);
    }
122
    /**
124  * Ausgangskondensator deaktivieren
     */
126 void cmdBufferDisable()
    {
128     relaisOpen(BUFFER);
        ledEnable(LED3);
130 }

132 /**
     * Strom Shunts aktivieren
134  */
    void cmdShuntEnable()
136 {
        ledEnable(LED4);
138     relaisClose(CURRENT);
    }
140
    /**
142  * Strom Shunts deaktivieren
     */
144 void cmdShuntDisable()
    {
146     ledDisable(LED4);
        relaisOpen(CURRENT);
148 }

150 /**
     * Automatische Spannungskorrektur aktivieren
152  */
    void outputEnableAutoAdjustment()
154 {
```

```
    TimerEnable(TIMER2_BASE, TIMER_A);
}

/**
 * Automatische Spannungskorrektur deaktivieren
 */
void outputDisableAutoAdjustment()
{
    TimerDisable(TIMER2_BASE, TIMER_A);
}

/**
 * Sequenzwiedergabe starten
 */
void cmdOutputStart()
{
    outputSampleNum = 0;

    if(settings.outputMode == STATIC) {
        telnetWrite("STATIC mode selected, no sequenze output possible\n");
        return;
    }

    switch(settings.syncMode) {
        case STANDALONE:
            rs485SetMode(STANDALONE);
            TimerEnable(TIMER0_BASE, TIMER_B);      // Start 20kHz output timer
            break;

        case MASTER:
            rs485SetMode(MASTER);
            rs485EnableRxInterrupt();
            TimerEnable(TIMER0_BASE, TIMER_B);      // Start 20kHz output timer
            break;

        case SLAVE:
            rs485SetMode(SLAVE);
            rs485EnableRxInterrupt();
            break;
    }
}

/**
 * Sequenzwiedergabe stoppen
 */
void cmdOutputStop()
{
    rs485DisableRxInterrupt();
    rs485SetMode(STANDALONE);

    // Stop 20kHz output timer
    TimerDisable(TIMER0_BASE, TIMER_B);
    dacSetValue(0);

    ledEnable(LED1);
}

/**
 * Naechstes Sample an DAC senden
 */
void outputNextSample()
{
    // Put voltage value to DAC
```

```
218     unsigned int microvolt = sdramGetVoltageValue(outputSampleNum);
        dacSetVoltage(microvolt);
220
        if(outputSampleNum < sdramGetVoltageSampleCnt()) {
222         outputSampleNum++;
        } else {
224         if(settings.outputMode == REPEAT) {
                outputSampleNum = 0;
226         } else if(settings.outputMode == SINGLE) {
                cmdOutputStop();
228         }
        }
230
        // Toggle LED to measure output clock
232     ledToggle(LED6);

234     static volatile unsigned int ledBlink = 0;
        if(ledBlink == 10000) {
236         ledToggle(LED1);
            ledBlink = 0;
238     } else {
            ledBlink++;
240     }
    }
242
    /**
244  * Interrupt Handler zur Sequenzwiedergabe
     */
246 void Timer0BIntHandler()
    {
248     switch(settings.syncMode) {
            case STANDALONE:
250             outputNextSample();
                break;
252
            case MASTER:
254             rs485SendSyncCommand();
                break;
256     }

258     // Clear interrupt flag
        TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
260 }

262 /**
     * Interrupt Handler fuer Korrektur der Ausgangsspannung
264  */
    void Timer2AIntHandler()
266 {
        unsigned int dacVolt = dacGetVoltage();
268     unsigned int adcVolt = adcExternalGetVoltage();

270     UARTprintf("dac volt %d\n", dacVolt);
        UARTprintf("adc volt %d\n", adcVolt);
272
        int diff = ((int)dacVolt - (int)adcVolt) / 90;
274     UARTprintf("diff %d\n", diff);

276     dacCorrectValue(diff);

278     // Clear interrupt flag
        TimerIntClear(TIMER2_BASE, TIMER_TIMA_TIMEOUT);
280 }
```

relais.h

```
/*
 * relais.h
 *
 *  Created on: 13.07.2012
 *      Author: Tobias
 */

#ifndef RELAIS_H_
#define RELAIS_H_

typedef enum
{
    BUFFER,
    CURRENT
} relais_t;

void relaisInit();
void relaisClose(relais_t relais);
void relaisOpen(relais_t relais);

#endif /* RELAIS_H_ */
```

relais.c

```
/*
 * relais.c
 *
 *  Created on: 13.07.2012
 *      Author: Tobias
 */

#include "relais.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_epi.h"
#include "inc/hw_gpio.h"

#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"
#include "utils/uartstdio.h"
#include <stdlib.h>

/**
 * Relais GPIOs initialisieren
 */
void relaisInit()
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);

    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTJ_BASE, GPIO_PIN_7);

    relaisOpen(BUFFER);
    relaisOpen(CURRENT);
```

```
}

/**
 * Relais schliessen
 */
void relaisClose(relais_t relais)
{
    switch (relais) {
        case BUFFER:
            ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);
            break;
        case CURRENT:
            ROM_GPIOPinWrite(GPIO_PORTJ_BASE, GPIO_PIN_7, GPIO_PIN_7);
            break;
        }
}

/**
 * Relais oeffnen
 */
void relaisOpen(relais_t relais)
{
    switch (relais) {
        case BUFFER:
            ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0);
            break;
        case CURRENT:
            ROM_GPIOPinWrite(GPIO_PORTJ_BASE, GPIO_PIN_7, 0);
            break;
        }
}
```

rs485.h

```
/*
 * rs485.h
 *
 *  Created on: 18.07.2012
 *      Author: Tobias
 */

#ifndef RS485_H_
#define RS485_H_

#include "settings.h"

void rs485Init();
void rs485SetMode(sync_mode_t mode);
void rs485EnableRxInterrupt();
void rs485DisableRxInterrupt();
void rs485SendSyncCommand();
void rs485RxIntHandler();

#endif /* RS485_H_ */
```

rs485.c

```
/*
 * rs485.c
 *
 *  Created on: 18.07.2012
 *      Author: Tobias
 */

#include "rs485.h"
#include "led.h"
#include "output.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/gpio.h"
#include "driverlib/uart.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include <utils/uartstdio.h>

/**
 * RS485 Connection initialisieren
 */
void rs485Init()
{
    // Enable peripherals
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    // Driver/Receiver Enable Pins
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_0);

    // Set GPIO A0 and A1 as UART pins.
    GPIOPinConfigure(GPIO_PB0_U1RX);
    GPIOPinConfigure(GPIO_PB1_U1TX);
    ROM_GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // UART Config: 8N1, 1MBaud
    UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 1000000,
                            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                             UART_CONFIG_PAR_NONE));

    // Enable UART1 interrupts
    ROM_IntEnable(INT_UART1);

    // Set UART1 RX to highest priority
    IntPrioritySet(INT_UART1, 0x00);

    // Set transceiver mode
    rs485SetMode(STANDALONE);
}

/**
 * RS485 Treiber und Empfaenger je nach Modus de-/aktivieren
 */
void rs485SetMode(sync_mode_t mode)
{
```

```
    switch(mode) {
        case STANDALONE:
            // Disable Driver and Receiver
            ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0, 0);
            break;
        case MASTER:
            // Enable Driver and Receiver
            ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0, GPIO_PIN_0);
            break;
        case SLAVE:
            // Disable Driver and enable Receiver
            ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_0, GPIO_PIN_0);
            break;
    }
}

/**
 * UART Receive Interrupt aktivieren
 */
void rs485EnableRxInterrupt()
{
    ROM_UARTIntEnable(UART1_BASE, UART_INT_RX);
}

/**
 * UART Receive Interrupt deaktivieren
 */
void rs485DisableRxInterrupt()
{
    ROM_UARTIntDisable(UART1_BASE, UART_INT_RX);
}

/**
 * Sync Kommando an UART ausgeben
 */
void rs485SendSyncCommand()
{
    UARTCharPut(UART1_BASE, 'S');
}

/**
 * Interrupt Handler: RS485 UART Empfang
 */
void rs485RxIntHandler()
{
    unsigned char recv = (unsigned char)ROM_UARTCharGet(UART1_BASE);
    if(recv == 'S') {
        ledToggle(LED5);
        outputNextSample();
    }

    // Clear interrupt flag
    ROM_UARTIntClear(UART1_BASE, UART_INT_RX);
}
```

sdram.h

```
/*
 * sdram.h
 *
 *  Created on: 11.07.2012
 *      Author: Tobias
 */

#ifndef SDRAM_H_
#define SDRAM_H_

#define EPI_PORTC_PINS (GPIO_PIN_7 | GPIO_PIN_6 | GPIO_PIN_5 | GPIO_PIN_4)
#define EPI_PORTE_PINS (GPIO_PIN_1 | GPIO_PIN_0)
#define EPI_PORTF_PINS (GPIO_PIN_5 | GPIO_PIN_4)
#define EPI_PORTG_PINS (GPIO_PIN_7 | GPIO_PIN_1 | GPIO_PIN_0)
#define EPI_PORTH_PINS (GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 | GPIO_PIN_2 | \
                        GPIO_PIN_1 | GPIO_PIN_0)
#define EPI_PORTJ_PINS (GPIO_PIN_6 | GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 | \
                        GPIO_PIN_2 | GPIO_PIN_1 | GPIO_PIN_0)

// The starting and ending address for the 32MB SDRAM chip (16Meg x 16bits)
#define SDRAM_START_ADDRESS 0x000000
#define SDRAM_END_ADDRESS 0xFFFFFF

void sdramInit();
void sdramTest();

void sdramWriteVoltage(unsigned long voltage);
void sdramGetPosition();
void sdramResetPointer();
void sdramReadVoltage();
void sdramReadPos(int argc, char *argv[]);
unsigned int sdramGetVoltageValue(unsigned int index);
unsigned int sdramGetVoltageSampleCnt();
unsigned long long sdramGetVoltageChecksum();

#endif /* SDRAM_H_ */
```

sdram.c

```
/*
 * sdram.c
 *
 *  Created on: 11.07.2012
 *      Author: Tobias
 */

#include "sdram.h"
#include "helper.h"
#include "settings.h"
#include "ethernet.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_epi.h"
#include "inc/hw_gpio.h"
#include "driverlib/epi.h"
#include "driverlib/gpio.h"
```

```
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include <stdlib.h>

static volatile unsigned short *g_pusEPISdram;
static volatile unsigned long sdramPos;
static volatile unsigned long voltageSampleCnt;

/**
 * SDRAM GPIOs und Peripherie initialisieren
 */
void sdramInit()
{
    UARTprintf("Initialize SDRAM...");

    // Enable EPI0 peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_EPI0);

    // Enable ports for EPI0 usage
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);

    // Configure pins for EPI0 usage
    GPIOPinConfigure(GPIO_PH3_EPI0S0);
    GPIOPinConfigure(GPIO_PH2_EPI0S1);
    GPIOPinConfigure(GPIO_PC4_EPI0S2);
    GPIOPinConfigure(GPIO_PC5_EPI0S3);
    GPIOPinConfigure(GPIO_PC6_EPI0S4);
    GPIOPinConfigure(GPIO_PC7_EPI0S5);
    GPIOPinConfigure(GPIO_PH0_EPI0S6);
    GPIOPinConfigure(GPIO_PH1_EPI0S7);
    GPIOPinConfigure(GPIO_PE0_EPI0S8);
    GPIOPinConfigure(GPIO_PE1_EPI0S9);
    GPIOPinConfigure(GPIO_PH4_EPI0S10);
    GPIOPinConfigure(GPIO_PH5_EPI0S11);
    GPIOPinConfigure(GPIO_PF4_EPI0S12);
    GPIOPinConfigure(GPIO_PG0_EPI0S13);
    GPIOPinConfigure(GPIO_PG1_EPI0S14);
    GPIOPinConfigure(GPIO_PF5_EPI0S15);
    GPIOPinConfigure(GPIO_PJ0_EPI0S16);
    GPIOPinConfigure(GPIO_PJ1_EPI0S17);
    GPIOPinConfigure(GPIO_PJ2_EPI0S18);
    GPIOPinConfigure(GPIO_PJ3_EPI0S19);
    GPIOPinConfigure(GPIO_PJ4_EPI0S28);
    GPIOPinConfigure(GPIO_PJ5_EPI0S29);
    GPIOPinConfigure(GPIO_PJ6_EPI0S30);
    GPIOPinConfigure(GPIO_PG7_EPI0S31);

    // Configure pins for EPI0 mode with 8mA drive strength in push-pull operation
    GPIOPinTypeEPI(GPIO_PORTC_BASE, EPI_PORTC_PINS);
    GPIOPinTypeEPI(GPIO_PORTE_BASE, EPI_PORTE_PINS);
    GPIOPinTypeEPI(GPIO_PORTF_BASE, EPI_PORTF_PINS);
    GPIOPinTypeEPI(GPIO_PORTG_BASE, EPI_PORTG_PINS);
    GPIOPinTypeEPI(GPIO_PORTH_BASE, EPI_PORTH_PINS);
    GPIOPinTypeEPI(GPIO_PORTJ_BASE, EPI_PORTJ_PINS);

    // Set EPI clock to sysclock
    EPIDividerSet(EPI0_BASE, 0);
```

```
82      // Enable SDRAM mode
        EPIModeSet(EPI0_BASE, EPI_MODE_SDRAM);
84
        // Configure SDRAM mode
86      EPIConfigSDRAMSet(EPI0_BASE, EPI_SDRAM_CORE_FREQ_30_50 | EPI_SDRAM_FULL_POWER |
            EPI_SDRAM_SIZE_256MBIT, 1024);
88      // Set the address mapping
        EPIAddressMapSet(EPI0_BASE, EPI_ADDR_RAM_SIZE_256MB | EPI_ADDR_RAM_BASE_6);
90
        // Wait for SDRAM to be ready
92      while(HWREG(EPI0_BASE + EPI_O_STAT) &  EPI_STAT_INITSEQ)
        {
94      }
96      // Set EPI memory pointer to EPI base address
        g_pusEPISdram = (unsigned short *)0x60000000;
98
        sdramPos = 0;
100     voltageSampleCnt = 0;
102     UARTprintf("done\n");
    }
104
    /**
106  * SDRAM Testroutine, untere und oberes Bytes schreiben und wieder lesen
     */
108 void sdramTest()
    {
110     //
        // Read the initial data in SDRAM, and display it on the console.
112     //
        UARTprintf("  SDRAM Initial Data:\n");
114     UARTprintf("     Mem[0x6000.0000] = 0x%4x\n",
                   g_pusEPISdram[SDRAM_START_ADDRESS]);
116     UARTprintf("     Mem[0x6000.0001] = 0x%4x\n",
                   g_pusEPISdram[SDRAM_START_ADDRESS + 1]);
118     UARTprintf("     Mem[0x603F.FFFE] = 0x%4x\n",
                   g_pusEPISdram[SDRAM_END_ADDRESS - 1]);
120     UARTprintf("     Mem[0x603F.FFFF] = 0x%4x\n\n",
                   g_pusEPISdram[SDRAM_END_ADDRESS]);
122
        //
124     // Display what writes we are doing on the console.
        //
126     UARTprintf("  SDRAM Write:\n");
        UARTprintf("     Mem[0x6000.0000] <- 0xabcd\n");
128     UARTprintf("     Mem[0x6000.0001] <- 0x1234\n");
        UARTprintf("     Mem[0x603F.FFFE] <- 0xdcba\n");
130     UARTprintf("     Mem[0x603F.FFFF] <- 0x4321\n\n");

132     //
        // Write to the first 2 and last 2 address of the SDRAM card.  Since the
134     // SDRAM card is word addressable, we will write words.
        //
136     g_pusEPISdram[SDRAM_START_ADDRESS] = 0xabcd;
        g_pusEPISdram[SDRAM_START_ADDRESS + 1] = 0x1234;
138     g_pusEPISdram[SDRAM_END_ADDRESS - 1] = 0xdcba;
        g_pusEPISdram[SDRAM_END_ADDRESS] = 0x4321;
140

142     //
        // Read back the data you wrote, and display it on the console.
```

```
144     //
        UARTprintf("  SDRAM Read:\n");
146     UARTprintf("     Mem[0x6000.0000] = 0x%4x\n",
                g_pusEPISdram[SDRAM_START_ADDRESS]);
148     UARTprintf("     Mem[0x6000.0001] = 0x%4x\n",
                g_pusEPISdram[SDRAM_START_ADDRESS + 1]);
150     UARTprintf("     Mem[0x603F.FFFE] = 0x%4x\n",
                g_pusEPISdram[SDRAM_END_ADDRESS - 1]);
152     UARTprintf("     Mem[0x603F.FFFF] = 0x%4x\n\n",
                g_pusEPISdram[SDRAM_END_ADDRESS]);
154
156     //
        // Check the validity of the data.
158     //
        if((g_pusEPISdram[SDRAM_START_ADDRESS] == 0xabcd) &&
160             (g_pusEPISdram[SDRAM_START_ADDRESS + 1] == 0x1234) &&
                (g_pusEPISdram[SDRAM_END_ADDRESS - 1] == 0xdcba) &&
162             (g_pusEPISdram[SDRAM_END_ADDRESS] == 0x4321))
        {
164         //
            // Read and write operations were successful.  Return with no errors.
166         //
            UARTprintf("Read and write to external SDRAM was successful!\n");
168     }
    }
170
    /**
172  * 24 Bit Spannungswert speichern
     */
174 void sdramWriteVoltage(unsigned long voltage)
    {
176     voltageSampleCnt++;

178     static unsigned char voltageBuffer[6];
        static unsigned char voltageBufferCnt = 0;
180
        // Spannungswert in 3 Byte zerlegen
182     voltageBuffer[voltageBufferCnt++] = (voltage & 0xFF0000) >> 16;
        voltageBuffer[voltageBufferCnt++] = (voltage & 0x00FF00) >> 8;
184     voltageBuffer[voltageBufferCnt++] = (voltage & 0x0000FF);

186     if(voltageBufferCnt == 6) {
            int i;
188         for(i=0; i<6; i=i+2) {
                g_pusEPISdram[sdramPos++] = (voltageBuffer[i] << 8) | (voltageBuffer[i+1])
                    ;
190         }
            voltageBufferCnt = 0;
192     }
    }
194
    /**
196  * Aktuelle Speicherposition abfragen
     */
198 void sdramGetPosition()
    {
200     telnetWrite(itoa(sdramPos));
    }
202
    /**
204  * Schreibposition ruecksetzen
     */
```

```
206 void sdramResetPointer()
    {
208     sdramPos = 0;
        voltageSampleCnt = 0;
210 }

212 /**
     * Spannungswert auslesen
214  */
    void sdramReadVoltage()
216 {
        sdramResetPointer();
218     unsigned long value;

220     unsigned char readBuffer[6];
        readBuffer[0] = (g_pusEPISdram[sdramPos] & 0xFF00) >> 8;
222     readBuffer[1] = (g_pusEPISdram[sdramPos] & 0x00FF);
        readBuffer[2] = (g_pusEPISdram[sdramPos+1] & 0xFF00) >> 8;
224     readBuffer[3] = (g_pusEPISdram[sdramPos+1] & 0x00FF);
        readBuffer[4] = (g_pusEPISdram[sdramPos+2] & 0xFF00)  >> 8;
226     readBuffer[5] = (g_pusEPISdram[sdramPos+2] & 0x00FF);

228     //UARTprintf("voltage bytes: %d %d %d\n", readBuffer[0], readBuffer[1], readBuffer
            [2]);
        //UARTprintf("voltage bytes: %d %d %d\n", readBuffer[3], readBuffer[4], readBuffer
            [5]);
230
        int i;
232     for(i=0; i<6; i=i+3)  {
            value = (readBuffer[i+0] << 16) | (readBuffer[i+1] << 8) | readBuffer[i+2];
234         UARTprintf("voltage value: %d\n", value);
        }
236


238     for(i=0; i<voltageSampleCnt; i++) {

240     }


242
    }
244
    /**
246  * Wert von Position auslesen
     */
248 void sdramReadPos(int argc, char *argv[])
    {
250     if(argc == 0) {
            telnetWrite("Parameter missing\n");
252     }

254     unsigned long index = atoi(argv[0]);
        unsigned int value = sdramGetVoltageValue(index);
256     UARTprintf("voltage index value: %d %d\n", index, value);
    }
258
    /**
260  * 24-Bit Wert anhand des Index aus SDRAM auslesen und zurueckgeben
     */
262 unsigned int sdramGetVoltageValue(unsigned int index)
    {
264     unsigned char byte[3];

266     // calculate memory location from value index
```

```
    sdramPos = index*3/2;

    if(index%2 == 0) {  // even index
        byte[0] = (g_pusEPISdram[sdramPos] & 0xFF00) >> 8;
        byte[1] = (g_pusEPISdram[sdramPos] & 0x00FF);
        byte[2] = (g_pusEPISdram[sdramPos+1] & 0xFF00) >> 8;
    } else {            // odd index
        byte[0] = (g_pusEPISdram[sdramPos] & 0x00FF);
        byte[1] = (g_pusEPISdram[sdramPos+1] & 0xFF00) >> 8;
        byte[2] = (g_pusEPISdram[sdramPos+1] & 0x00FF);
    }

    return (byte[0] << 16) | (byte[1] << 8) | byte[2];
}

/**
 * Anzahl an gespeicherten Samples abfragen
 */
unsigned int sdramGetVoltageSampleCnt()
{
    return voltageSampleCnt;
}

/**
 * Checksumme berechnen
 */
unsigned long long sdramGetVoltageChecksum()
{
    unsigned long long checksum = 0;
    unsigned int i;
    for(i=0; i<voltageSampleCnt; i++) {
        checksum += sdramGetVoltageValue(i);
    }

    return checksum;
}
```

settings.h

```
/*
 * settings.h
 *
 *  Created on: 28.06.2012
 *      Author: Tobias
 */

#ifndef SETTINGS_H_
#define SETTINGS_H_

#include <stdint.h>

#define MACADDR 0xACDE48000001;     // AC:DE:48:00:00:01
#define IPADDR  0xC0A800C8;         // 192.168.0.200

// Modus fuer Ausgangsspannung
typedef enum
{
    STATIC,     // Statische Ausgangsspannung
    SINGLE,     // Sequenz einmal ausgeben
```

```
    REPEAT      // Sequenz wiederholt ausgeben
} output_mode_t;

// Sync Mode
typedef enum
{
    STANDALONE,
    MASTER,
    SLAVE
} sync_mode_t;

// DAC Korrekturwerte
typedef struct
{
    unsigned int microvolt;
    unsigned short dacValue;
} dac_calibration_t;

// Board-Einstellungen (werden in Soft-EEPROM gespeichert)
typedef struct
{
    unsigned int ipAddress;
    unsigned long long macAddress;
    output_mode_t outputMode;
    sync_mode_t syncMode;
    int voltageOffset;
    int voltageGain;
    dac_calibration_t dacTable[13];
} settings_t;

// Globale Instanz der Board-Einstellungen
extern settings_t settings;


void settingsInit();
void settingsLoad();
void settingsSave();
void cmdSettings(int argc, char *argv[]);
void cmdSetOutputMode(int argc, char *argv[]);
void cmdSetSyncMode(int argc, char *argv[]);

#endif /* SETTINGS_H_ */
```

settings.c

```
/*
 * settings.c
 *
 *  Created on: 28.06.2012
 *      Author: Tobias
 */

#include "settings.h"
#include "ethernet.h"
#include "helper.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_timer.h"
```

```
#include "inc/hw_ints.h"
#include "inc/hw_gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/gpio.h"
#include "softeeprom.h"
#include <utils/uartstdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Instanz der Board-Einstellungen
settings_t settings;

// Pointer auf Settings Instanz
uint8_t *pSettings = (uint8_t*) &settings;

/**
 * SoftEEPROM initialisieren und Einstellungen laden
 */
void settingsInit()
{
    UARTprintf("Initializing Soft EEPROM...");

    long eepromInitStatus;
    eepromInitStatus = SoftEEPROMInit(0x3E000, 0x40000, 0x1000);

    // Check for initialization success
    if(eepromInitStatus != 0)
    {
        UARTprintf("\rAn error occurred during Soft EEPROM initialization!");
    }
    settingsLoad();

    settings.ipAddress = IPADDR;
    settings.macAddress = MACADDR;

    UARTprintf("done\n");
}

/**
 * Einstellungen aus SoftEEPROM laden
 */
void settingsLoad()
{
    uint8_t i;
    uint16_t buffer;
    long eepromReadStatus;
    tBoolean idFound;

    UARTprintf("Load board settings (%d bytes)...", sizeof(settings));

    for(i=0; i<sizeof(settings); i++) {

        eepromReadStatus = SoftEEPROMRead(i, &buffer, &idFound);

        if(eepromReadStatus != 0 || idFound == false) {
            UARTprintf("\rAn error occurred during a soft EEPROM read operation");
        }

        memcpy(pSettings+i, &buffer, 1);
    }
```

```
78      UARTprintf("done\n");
}
80
/**
82   * Einstellungen in SoftEEPROM speichern
 */
84 void settingsSave()
{
86      uint8_t i;
        uint8_t buffer;
88      //uint16_t buffer;
        long eepromWriteStatus;
90
        UARTprintf("Save board settings (%d bytes)...", sizeof(settings));
92
        for(i=0; i<sizeof(settings); i++) {
94
            memcpy(&buffer, pSettings+i, 1);
96          eepromWriteStatus = SoftEEPROMWrite(i, buffer);

98          if(eepromWriteStatus != 0) {
                UARTprintf("\rAn error occurred during a soft EEPROM read operation");
100         }
        }
102
        UARTprintf("done\n");
104 }

106 /**
     * Board-Einstellungen per Telnet ausgeben
108  */
    void cmdSettings(int argc, char *argv[])
110 {
        if(argc == 0) {
112         char buffer[100];
            char dacTable[650];
114         dacTable[0] = '\0';

116         //sprintf(buffer, "IP Adresse: %s\n", itoa(settings.ipAddress));
            //telnetWrite(buffer);
118         //sprintf(buffer, "MAC Adresse: %lld\n", settings.macAddress);
            //telnetWrite(buffer);
120
            if(settings.outputMode == SINGLE) {
122             telnetWrite("Output Mode: SINGLE\n");
            } else if(settings.outputMode == STATIC) {
124             telnetWrite("Output Mode: STATIC\n");
            } else if(settings.outputMode == REPEAT) {
126             telnetWrite("Output Mode: REPEAT\n");
            } else {
128             telnetWrite("Output Mode: not set\n");
            }
130
            if(settings.syncMode == STANDALONE) {
132             telnetWrite("Sync Mode: STANDALONE\n");
            } else if(settings.syncMode == MASTER) {
134             telnetWrite("Sync Mode: MASTER\n");
            } else if(settings.syncMode == SLAVE) {
136             telnetWrite("Sync Mode: SLAVE\n");
            } else {
138             telnetWrite("Sync Mode: not set\n");
            }
140
```

```
        sprintf(buffer, "Voltage Offset: %d\n", settings.voltageOffset);
        telnetWrite(buffer);
        sprintf(buffer, "Voltage Gain: %d\n", settings.voltageGain);
        telnetWrite(buffer);

        telnetWrite("DAC Calibration values:\n");
        int i;
        for(i=0; i<13; i++) {
            sprintf(buffer, "Microvolt: %d DAC value: %d\n", settings.dacTable[i].
                microvolt, settings.dacTable[i].dacValue);
            strcat(dacTable, buffer);
        }
        telnetWrite(dacTable);
        telnetWrite("\n");

    } else if(strcmp(argv[0], "load") == 0) {
        settingsLoad();
        telnetWrite("Settings loaded\n");
    } else if(strcmp(argv[0], "save") == 0) {
        settingsSave();
        telnetWrite("Settings saved\n");
    } else {
        telnetWrite("Wrong parameter\n");
    }
}

/**
 * Output Mode aendern
 */
void cmdSetOutputMode(int argc, char *argv[])
{
    if(argc == 0) {
        telnetWrite("Parameter missing\n");
    }

    if(strcmp(argv[0], "static") == 0) {
        settings.outputMode = STATIC;
    } else if(strcmp(argv[0], "single") == 0) {
        settings.outputMode = SINGLE;
    } else if(strcmp(argv[0], "repeat") == 0) {
        settings.outputMode = REPEAT;
    } else {
        telnetWrite("Wrong parameter\n");
    }
}

/**
 * Sync Mode aendern
 */
void cmdSetSyncMode(int argc, char *argv[])
{
    if(argc == 0) {
        telnetWrite("Parameter missing\n");
    }

    if(strcmp(argv[0], "standalone") == 0) {
        settings.syncMode = STANDALONE;
    } else if(strcmp(argv[0], "master") == 0) {
        settings.syncMode = MASTER;
    } else if(strcmp(argv[0], "slave") == 0) {
            settings.syncMode = SLAVE;
    } else {
        telnetWrite("Wrong parameter\n");
```

```
    }
}
```

spi.h

```
/*
 * spi.h
 *
 *  Created on: 16.07.2012
 *      Author: Tobias
 */

#ifndef SPI_H_
#define SPI_H_

// SPI chip selects
typedef enum
{
    ADC,
    DAC,
    NONE
} spi_cs_t;

void spiInit();
void spiChipSelect(spi_cs_t chip);
void spiClearFifo();
unsigned char spiReadWrite(unsigned char data);

#endif /* SPI_H_ */
```

spi.c

```
/*
 * spi.c
 *
 *  Created on: 16.07.2012
 *      Author: Tobias
 */

#include "spi.h"

#include <inc/hw_memmap.h>
#include <inc/hw_ssi.h>
#include <inc/hw_types.h>
#include <driverlib/ssi.h>
#include <driverlib/gpio.h>
#include <driverlib/sysctl.h>
#include <driverlib/rom.h>
#include <utils/uartstdio.h>
#include <inc/lm3s9b92.h>

/**
 * SPI-Interface initialisieren
 */
void spiInit()
```

```
{
    // Configure pots for chip selects and set them to high
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);

    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_6);
    ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, GPIO_PIN_6);

    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTD_BASE, GPIO_PIN_1);
    ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, GPIO_PIN_1);


    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA4_SSI0RX);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);

    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

    //SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
        SSI_MODE_MASTER, 2000000, 8);  // ADC=SSI_FRF_MOTO_MODE_3
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
        SSI_MODE_MASTER, 1000000, 8);  // ADC=SSI_FRF_MOTO_MODE_3

    SSIEnable(SSI0_BASE);
}

/**
 * Steuerung der SPI Chip Selects fuer ADC und DAC
 */
void spiChipSelect(spi_cs_t chip)
{
    switch (chip) {
        case ADC:
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, GPIO_PIN_1);
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0);
            break;
        case DAC:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, GPIO_PIN_6);
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, 0);
            break;
        case NONE:
            ROM_GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, GPIO_PIN_6);
            ROM_GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_1, GPIO_PIN_1);
            break;
    }
}

/*
 * Loeschen des SPI FIFOs
 */
void spiClearFifo()
{
    unsigned long dummy;
    while(SSIDataGetNonBlocking(SSI0_BASE, &dummy));
}

/*
 * SPI Schreib-/Lesevorgang
 */
unsigned char spiReadWrite(unsigned char data)
```

```
{
    spiClearFifo();

    unsigned long result;
    SSIDataPut(SSI0_BASE, data);
    SSIDataGet(SSI0_BASE, &result);
    return (unsigned char)(result&0xff);
}
```

startup_ccs.c

```
//*****************************************************************************
//
// startup_ccs.c - Startup code for use with TI's Code Composer Studio.
//
// Copyright (c) 2009-2011 Texas Instruments Incorporated.  All rights reserved.
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
// This is part of revision 8264 of the EK-LM3S9B92 Firmware Package.
//
//*****************************************************************************

#include "led.h"

//*****************************************************************************
//
// Forward declaration of the default fault handlers.
//
//*****************************************************************************
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
static void IntDefaultHandler(void);

//*****************************************************************************
//
// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****************************************************************************
extern void _c_int00(void);

//*****************************************************************************
//
// Linker variable that marks the top of the stack.
//
```

```
//*****************************************************************************
extern unsigned long __STACK_TOP;

//*****************************************************************************
//
// External declarations for the interrupt handlers used by the application.
//
//*****************************************************************************
extern void lwIPEthernetIntHandler();
extern void SysTickIntHandler();
extern void Timer0AIntHandler();
extern void Timer0BIntHandler();
extern void Timer1AIntHandler();
extern void Timer2AIntHandler();
extern void rs485RxIntHandler();

//*****************************************************************************
//
// The vector table.  Note that the proper constructs must be placed on this to
// ensure that it ends up at physical address 0x0000.0000 or at the start of
// the program if located at a start address other than 0.
//
//*****************************************************************************
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((unsigned long)&__STACK_TOP),
                                            // The initial stack pointer
    ResetISR,                               // The reset handler
    NmiSR,                                  // The NMI handler
    FaultISR,                               // The hard fault handler
    IntDefaultHandler,                      // The MPU fault handler
    IntDefaultHandler,                      // The bus fault handler
    IntDefaultHandler,                      // The usage fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // SVCall handler
    IntDefaultHandler,                      // Debug monitor handler
    0,                                      // Reserved
    IntDefaultHandler,                      // The PendSV handler
    SysTickIntHandler,                      // The SysTick handler
    IntDefaultHandler,                      // GPIO Port A
    IntDefaultHandler,                      // GPIO Port B
    IntDefaultHandler,                      // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                      // UART0 Rx and Tx
    rs485RxIntHandler,                      // UART1 Rx and Tx
    IntDefaultHandler,                      // SSI0 Rx and Tx
    IntDefaultHandler,                      // I2C0 Master and Slave
    IntDefaultHandler,                      // PWM Fault
    IntDefaultHandler,                      // PWM Generator 0
    IntDefaultHandler,                      // PWM Generator 1
    IntDefaultHandler,                      // PWM Generator 2
    IntDefaultHandler,                      // Quadrature Encoder 0
    IntDefaultHandler,                      // ADC Sequence 0
    IntDefaultHandler,                      // ADC Sequence 1
    IntDefaultHandler,                      // ADC Sequence 2
    IntDefaultHandler,                      // ADC Sequence 3
    IntDefaultHandler,                      // Watchdog timer
    IntDefaultHandler,                      // Timer 0 subtimer A
```

```
    Timer0BIntHandler,                      // Timer 0 subtimer B
    Timer1AIntHandler,                      // Timer 1 subtimer A
    IntDefaultHandler,                     // Timer 1 subtimer B
    Timer2AIntHandler,                      // Timer 2 subtimer A
    IntDefaultHandler,                     // Timer 2 subtimer B
    IntDefaultHandler,                      // Analog Comparator 0
    IntDefaultHandler,                      // Analog Comparator 1
    IntDefaultHandler,                      // Analog Comparator 2
    IntDefaultHandler,                      // System Control (PLL, OSC, BO)
    IntDefaultHandler,                      // FLASH Control
    IntDefaultHandler,                      // GPIO Port F
    IntDefaultHandler,                      // GPIO Port G
    IntDefaultHandler,                      // GPIO Port H
    IntDefaultHandler,                      // UART2 Rx and Tx
    IntDefaultHandler,                      // SSI1 Rx and Tx
    IntDefaultHandler,                      // Timer 3 subtimer A
    IntDefaultHandler,                      // Timer 3 subtimer B
    IntDefaultHandler,                      // I2C1 Master and Slave
    IntDefaultHandler,                      // Quadrature Encoder 1
    IntDefaultHandler,                      // CAN0
    IntDefaultHandler,                      // CAN1
    IntDefaultHandler,                      // CAN2
    lwIPEthernetIntHandler,                 // Ethernet
    IntDefaultHandler,                      // Hibernate
    IntDefaultHandler,                      // USB0
    IntDefaultHandler,                      // PWM Generator 3
    IntDefaultHandler,                      // uDMA Software Transfer
    IntDefaultHandler,                      // uDMA Error
    IntDefaultHandler,                      // ADC1 Sequence 0
    IntDefaultHandler,                      // ADC1 Sequence 1
    IntDefaultHandler,                      // ADC1 Sequence 2
    IntDefaultHandler,                      // ADC1 Sequence 3
    IntDefaultHandler,                      // I2S0
    IntDefaultHandler,                      // External Bus Interface 0
    IntDefaultHandler                       // GPIO Port J
};

//*****************************************************************************
//
// This is the code that gets called when the processor first starts execution
// following a reset event.  Only the absolutely necessary set is performed,
// after which the application supplied entry() routine is called.  Any fancy
// actions (such as making decisions based on the reset cause register, and
// resetting the bits in that register) are left solely in the hands of the
// application.
//
//*****************************************************************************
void
ResetISR(void)
{
    //
    // Jump to the CCS C initialization routine.
    //
    __asm("    .global _c_int00\n"
          "    b.w     _c_int00");
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives a NMI.  This
// simply enters an infinite loop, preserving the system state for examination
// by a debugger.
//
```

```
//*****************************************************************************
static void
NmiSR(void)
{
    ledEnable(LED2);
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives a fault
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
FaultISR(void)
{
    ledEnable(LED2);
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
//
// This is the code that gets called when the processor receives an unexpected
// interrupt.  This simply enters an infinite loop, preserving the system state
// for examination by a debugger.
//
//*****************************************************************************
static void
IntDefaultHandler(void)
{
    ledEnable(LED2);
    //
    // Go into an infinite loop.
    //
    while(1)
    {
    }
}
```

temp.h

```
/*
 * temperature.h
 *
 *  Created on: 27.07.2012
 *      Author: Tobias
 */
```

```
#ifndef TEMPERATURE_H_
#define TEMPERATURE_H_

#define TMP102_ADDR 0x48        // TMP102 slave address

void tempInit();
void cmdGetTemperature();

#endif /* TEMPERATURE_H_ */
```

temp.c

```
/*
 * temperature.c
 *
 *  Created on: 27.07.2012
 *      Author: Tobias
 */

#include "temp.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_i2c.h"
#include "driverlib/i2c.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"
#include "ethernet.h"
#include <stdio.h>

/**
 * I2C Interface fuer Temperaturfuehler initialisieren
 */
void tempInit()
{
    // Enable I2C0 peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

    // Enable Port
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    // Configure I2C pins
    GPIOPinConfigure(GPIO_PB2_I2C0SCL);
    GPIOPinConfigure(GPIO_PB3_I2C0SDA);
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);

    // Enable and initialize the I2C0 master module
    I2CMasterInitExpClk(I2C0_MASTER_BASE, SysCtlClockGet(), false);
}

/**
 * Temperatur abfragen
 */
void cmdGetTemperature()
{
    unsigned char result[2];

```

```
    // Enable I2C write mode
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, TMP102_ADDR, false);

    // Select temperature register 0x00
    I2CMasterDataPut(I2C0_MASTER_BASE, 0x00);
    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    // Enable I2C read mode
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, TMP102_ADDR, true);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    // Read 2 bytes...
    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    result[0] = I2CMasterDataGet(I2C0_MASTER_BASE);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    result[1] = I2CMasterDataGet(I2C0_MASTER_BASE);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);
    while (I2CMasterBusy(I2C_MASTER_BASE));

    // Calculate temp (one lsb equals 0.0625 degree celsius)
    unsigned short temp = (result[0] << 4) | ((result[1] & 0xF0) >> 4);
    unsigned short tempInteger = temp * 625 / 10000;
    unsigned short tempDecimal = temp * 625 - (tempInteger * 10000);

    char buffer[100];
    sprintf(buffer, "Temperature: %d.%d degree Celsius\n", tempInteger, tempDecimal);
    telnetWrite(buffer);
}
```

uart.h

```
/*
 * uart.h
 *
 *  Created on: 25.06.2012
 *      Author: Tobias
 */

#ifndef UART_H_
#define UART_H_

void uartInit();

#endif /* UART_H_ */
```

uart.c

```
/*
 * uart.c
 *
 *  Created on: 25.06.2012
 *      Author: Tobias
 */

#include "uart.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include <utils/uartstdio.h>

/**
 * Initialize UART
 */
void uartInit()
{
    // Enable peripherals
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Set GPIO A0 and A1 as UART pins.
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // Initialize UARTStdio
    UARTStdioInit(0);

    UARTprintf("UART initialized...\n");
}
```

Versicherung über die Selbstständigkeit

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

Hamburg, 24. August 2012

Ort, Datum