Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Fabian Schwartau

Vielkanaliger controller-gesteuerter Zellspannungsgenerator zur Kalibrierung und zum Test von Batteriesensoren

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Fabian Schwartau

Vielkanaliger controller-gesteuerter Zellspannungsgenerator zur Kalibrierung und zum Test von Batteriesensoren

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. -Ing. Jürgen Vollmer

Abgegeben am 29. März 2012

Fabian Schwartau

Thema der Bachelorthesis

Vielkanaliger controller-gesteuerter Zellspannungsgenerator zur Kalibrierung und zum Test von Batteriesensoren

Stichworte

Mikrocontroller, Ethernet, Spannungsquelle, Stellaris, C, MATLAB

Kurzzusammenfassung

Diese Arbeit beschreibt die Konstruktion einer Spannungsquelle zum Testen und Kalibrieren von Batteriesensoren. Für die Kalibrierung muss eine sehr hohe Genauigkeit des Zellspannungsgenerators erreicht werden. Außerdem soll die Wiedergabe von Spannungssequenzen ermöglicht werden, wodurch Eingangssignale für die Sensoren reproduziert werden können. Die einzelnen Generatoren sind galvanisch getrennt und es können nahezu beliebig viele dieser Generatoren über Ethernet angesteuert werden.

Fabian Schwartau

Title of the paper

Multi-channel controller controlled cell voltage generator for calibration and testing of battery sensors

Keywords

Microcontroller, Ethernet, voltage source, Stellaris, C, MATLAB

Abstract

This work describes the construction of a voltage source for testing and calibration of battery sensors. For the calibration a high accuracy of the generator is required. Also the play back of a voltage sequence should be possible, by what input signals for the sensors are repeatable. Each generator is electrically isolated and a nearly unlimited count of generators can be controlled over Ethernet.

Danksagung

An dieser Stelle möchte ich mich zunächst bei Herrn Prof. Dr. -Ing. Karl-Ragmar Riemschneider bedanken, der es mir im Rahmen des Forschungsvorhabens BATSEN ermöglicht hat, diese Arbeit zu schreiben und auch als Erstprüfer dieser Arbeit fungiert hat. Der Dank geht auch an den Zweitprüfer, Herrn Prof. Dr. -Ing. Jürgen Vollmer.

Außerdem möchte ich mich bei Herrn Dipl. -Ing. Günter Müller und Herrn Dipl. -Ing. Matthias Schneider bedanken, die stets eine Hilfe waren und mich tatkräftig unterstützt haben.

Mein Dank gilt ebenfalls Herrn Dipl. -Ing. (FH) Martin Krey und meinem Kommilitonen Herrn B.Eng. Robert Ostermann, die mir stets mit guten Ratschlägen zur Seite standen.

Ein ganz besonderer Dank gilt meinen Eltern, die es mir dieses Studium ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung

In vielen Bereichen werden heutzutage bereits große Batterien eingesetzt, die aus mehreren Zellen bestehen, wie beispielsweise die Starter- oder Pufferbatterie in einem Auto oder als Antriebsbatterie in Gabelstaplern. Um die Lebensdauer dieser Batterien zu erhöhen ist es nicht nur notwendig genaue Informationen über den Zustand der Batterie, sondern insbesondere auch über die einzelnen Zellen, aus der die Batterie zusammengesetzt ist, zu haben.

Im Rahmen des vom Bundesministerium für Bildung und Forschung geförderten Projektes ”BATSEN” an der Hochschule für Angewandte Wissenschaften Hamburg werden Sensoren zur Überwachung dieser Batteriezellen entwickelt und getestet. Die Überwachung jeder einzelnen Zelle einer Batterie soll es ermöglichen, eine genauere Aussage über den Zustand der Batterie und der einzelnen Zellen zu treffen. Somit soll es möglich werden, Defekte einzelner Zellen zu erkennen, um diese rechtzeitig austauschen zu können. Zudem arbeiten die Sensoren drahtlos, um einen hohen Anschlussaufwand jedes einzelnen Sensors zu verhindern, wodurch die Produktionskosten für eine Batterie gesenkt werden sollen. Ein weiterer Vorteil der drahtlosen Sensoren ist die galvanisch getrennte Messung einer jeden Zelle.

Es wird zwischen drei verschiedenen Typen oder Klassen von Sensoren unterschieden. Sensoren der Klasse 1 besitzen lediglich einen Uplink-Kanal vom Sensor zur Basisstation. Die Sensoren senden dabei unkoordiniert, wodurch es auf dem gemeinsam genutzten Funkkanal zu Kollisionen kommt. Der Erhalt von Daten der Sensoren kann somit nicht garantiert werden. Dieses Problem wird bei den Sensoren der Klasse 2 gelöst, indem ein zusätzlicher Downlink-Kanal mit geringer Bandbreite von der Basistation zu den Sensoren hinzugefügt wird. Dieser wird mit RFID-Technik realisiert, um den Stromverbrauch des Sensors während der Ruhephasen auf ein Minimum senken zu können. Der Kanal dient primär zur Synchronisation der Module, es können aber auch geringe Mengen an Daten übertragen werden. Bei Sensoren der Klasse 3, von denen es derzeit noch keine Umsetzung gibt, wird der RFID-Downlink durch eine vollwertige Funkverbindung, z.B. im 433MHz Band, wie der Uplink-Kanal, ersetzt. Dies ermöglicht höhere Datenraten, steigert jedoch den Stromverbrauch enorm, da der Empfänger auch im Standby-Modus zusätzlich Energie benötigt.

1.1 Motivation

In vorherigen Arbeiten sind bereits mehrere Modelle von Batteriesensoren entwickelt worden. In der Arbeit von Herrn Ilgin wird gezeigt, dass es möglich ist, die Genauigkeit der Sensoren durch eine Kalibrierung zu verbessern [6]. Hierbei werden nicht nur Bauteiltoleranzen sondern auch Temperaturabhängigkeiten der Sensoren kompensiert. In der zuvor erwähnten Arbeit wird die Kalibrierung der Sensoren vollständig per Hand vorgenommen, was einen hohen zeitlichen Aufwand bedeutet. Hier setzt diese Arbeit an, in der ein Spannungsgenerator entwickelt wird, um diese Kalibrierung zu automatisieren.

Ein weiteres Problem ist bislang die Reproduzierbarkeit von Messwerten. Dies ist unter anderem wichtig beim Test der Kommunikation der Module über den gemeinsamen Funkkanal. Da hier nur eine begrenzte Bandbreite zur Verfügung steht und besondere Ereignisse, wie z.B. ein Hochstromereignis beim Starten eines Verbrennungsmotors, besonders hochfrequent abgetastet werden müssen, ist die Analyse der Kommunikation in solchen Momenten von großer Wichtigkeit. Um hinterher verschiedene Messungen miteinander vergleichen zu können, ist eine Reproduzierbarkeit solcher besonderen Ereignisse nötig.

1.2 Aufgabe und Anforderungen

Es soll ein Zellspannungsgenerator entwickelt werden, der zwei Aufgaben erfüllt: Zum einen soll es möglich sein, Batteriesensoren zu kalibrieren, zum anderen sollen zuvor aufgenommene oder synthetisierte Spannungsverläufe wiedergegeben werden können, um das Verhalten der Sensoren untersuchen zu können.

Für die Kalibrierung der Sensoren ist es notwendig, dass der Zellspannungsgenerator konstante oder nur langsam veränderliche Spannungen mit hoher Genauigkeit ausgeben kann. Die Genauigkeit des Generators muss daher im Bereich der Auflösung der Sensoren liegen. Die Auflösung des Generators ist in diesem Fall nicht entscheidend, lediglich die Genauigkeit, denn es werden nicht viele Messpunkte benötigt, um eine vollständige Kalibrierung durchzuführen.

Anders ist es bei der Ausgabe von transienten Spannungen, wie beispielsweise dem zuvor erwähnten Startvorgang eines Autos. Hierbei spielt die Genauigkeit des Generators keine so große Rolle, es ist jedoch ein Mindestmaß an Amplitudenauflösung gefordert, um den Verlauf ausreichend genau nachbilden zu können. Des weiteren wird natürlich eine gewisse Bandbreite bzw. Samplerate des Systems benötigt.

Alle diese Anforderungen (Auflösung, Genauigkeit, Bandbreite und Samplerate) sind abhängig von den Parametern der Sensoren und den zu reproduzierenden Spannungsverläufen. Die genauen Anforderungen, die sich hieraus an die Soft- und Hardware ergeben, müssen noch erarbeitet werden.

Eine weitere Anforderung an das System ist die Modularität. Es soll möglich sein, auch große Batterien mit bis zu 40 Zellen zu simulieren. Hierfür bietet es sich an, die Zellspannungsgeneratoren in einem 19-Zoll Rack unterzubringen. Um bei solch großen Batterien auch die Beeinflussung der Sensoren untereinander zu untersuchen, sollen die Generatoren zu einer Batterie zusammen geschaltet werden können. Hierfür ist eine galvanische Trennung der einzelnen Module untereinander erforderlich.

Um eine Automatisierung der Messungen zu ermöglichen, sollen die Generatoren über MATLAB steuerbar sein. Das heißt, sowohl die Konfiguration der Module als auch die Spannungskurven müssen über MATLAB einstellbar sein.

Diese Anforderungen an die Soft- und Hardware werden nun im folgenden Kapitel konkretisiert. Dafür wird primär die Hardware der Sensoren untersucht.

2 Konzeption

Für die Konzeption der Zellspannungsgeneratoren sind zunächst einige Analysen der Sensoren notwendig. Auch ist die Untersuchung eines typischen Spannungsverlaufs, der mit den Generatoren erzeugt werden soll, für die Auslegung des Systems von großer Bedeutung. Anschließend soll ein Konzept erarbeitet werden, das den Ansprüchen aus den Analysen und der Aufgabenstellung gerecht wird.

2.1 Sensoren

Wie zuvor erläutert, unterteilen sich die Sensoren in drei Klassen. Jede dieser Klassen hat verschiedene Hardware und somit verschiedene Kenndaten, die bei der Konzeption berücksichtigt werden müssen. Da bislang nur Sensoren der Klasse 1 und 2 umgesetzt wurden, sollen diese beiden Klassen in den folgenden beiden Kapiteln genauer untersucht werden.

Bei den Klasse 1 Sensoren gibt es verschiedene Varianten von Hard- und Software. Hier soll nur der Sensor nach Plaschke untersucht werden, da der Sensor, der aus der Bachelorarbeit von Ilgin auf Grund des hohen Eingangsspannungsbereichs als Zellensensor nicht in Frage kommt [12], [6]. Somit sind die Sensoren nach Plaschke die neuste Generation der Klasse-1 für den Betrieb an einzelnen Batteriezellen [12]. Zukünftige Erwähnungen des Klasse-1 Sensors beziehen sich daher immer auf diesen. Bislang wurde nur ein Sensor der Klasse 2 entworfen, welcher auch genauer untersucht werden soll, siehe [7].

2.1.1 Spannungsbereich

Eine wichtige Kenngröße für den Zellspannungsgenerator ist der benötigte Ausgangsspannungsbereich. Der Sensor der Klasse 1 ist mit einem L6920 Step-Up-Converter der Firma STMicroelectronics ausgestattet. Dieser läuft bei einer Eingangsspannung von $1\,\mathrm{V}$ an und arbeitet dann mit einer minimalen Eingangsspannung von $0{,}6\,\mathrm{V}$. Maximal dürfen am Eingang des Reglers $5{,}5\,\mathrm{V}$ liegen [15].

Im bislang einzigen Klasse-2 Sensor ist ein TPS61201 Step-Up/Down-Converter verbaut. Dieser schaltet automatisch zwischen Step-Down- und Step-Up-Betrieb um. Er ist in der Lage, Spannungen am Eingang von $0{,}3\,\mathrm{V}$ bis $5{,}5\,\mathrm{V}$ zu verarbeiten. Dabei wird eine Spannung von mindestens $0{,}5\,\mathrm{V}$ benötigt, damit der Regler anläuft.

Somit ergibt sich ein gesamter Spannungsbereich von $0{,}3\,\mathrm{V}$ bis $5{,}5\,\mathrm{V}$. Da es sich hierbei um die Simulation einer Batteriezelle handelt, werden Spannungen kleiner $1{,}0\,\mathrm{V}$ nicht erreicht, denn selbst ein leerer Bleiakku hat bei starker Belastung noch über $1\,\mathrm{V}$. Lithium-Zellen haben gegenüber Bleizellen eine noch höhere untere Spannungsgrenze [8]. Der Generator sollte daher in der Lage sein, Spannungen von mindestens $0{,}5\,\mathrm{V}$ bis $5{,}5\,\mathrm{V}$ auszugeben.

2.1.2 Strombedarf

Neben dem Spannungsbereich, der für die Nachbildung von Spannungssequenzen benötigt wird, ist die maximale Stromaufnahme der Sensoren ein weiterer wichtiger Punkt. Dieser Wert wird später für die Auslegung des Ausgangsverstärkers benötigt. Um diesen Wert zu bestimmen, werden zwei Messungen durchgeführt, um den Stromverbrauch der beiden Sensortypen zu ermitteln. In den Abbildungen 2.3 und 2.2 sind jeweils Ausschnitte gezeigt, in denen man den Punkt des maximalen Stromverbrauchs sehen kann. Die Messung wird über einen $0{,}1\,\Omega$ Shunt-Widerstand in der Masse des Sensors gemessen. Die Messschaltung ist in Abbildung 2.1 verdeutlicht.

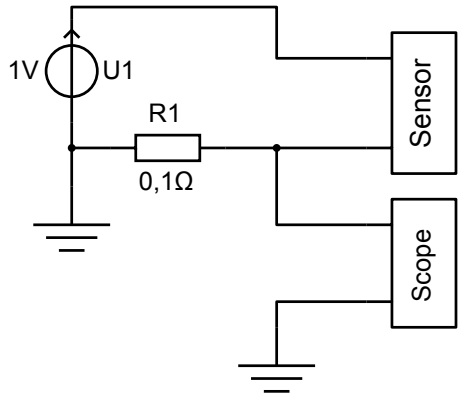


Abbildung 2.1: Schematischer Aufbau für Strommessung der Sensoren

Es ist zu erkennen, dass der Sensor der Klasse 1 maximal $174\,\mathrm{mA}$ benötigt, während der Klasse 2 Sensor einen Spitzenstrom von $504\,\mathrm{mA}$ erreicht. Bei dem Klasse-2 Sensor ist zu bedenken, dass der sehr hohe Impuls nur auftritt, nachdem der Step-Up/Down-Converter für einige Millisekunden abgeschaltet war. Es treten hier besonders hohe Ströme auf, da der Sensor in dieser Zeit nur über Pufferkondensatoren betrieben wurde, die anschließend wieder aufgeladen werden müssen. In der Zeit während der Wandler abgeschaltet ist, misst der

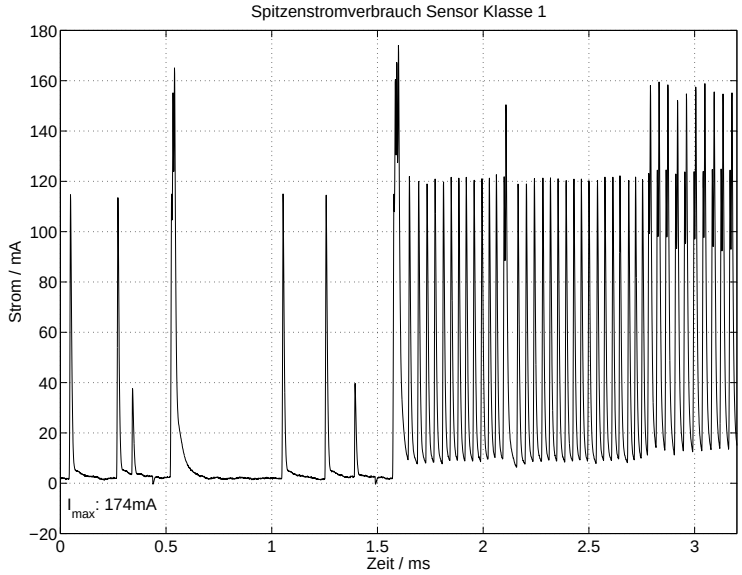


Abbildung 2.2: Spitzenstrommessung für Sensoren der Klasse 1

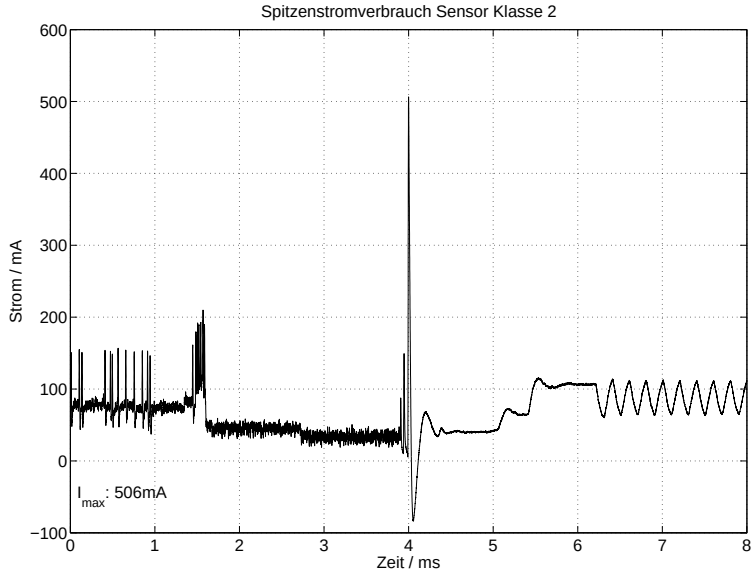


Abbildung 2.3: Spitzenstrommessung für Sensoren der Klasse 2

Sensor die Zellenspannung mit dem internen AD-Umsetzer. Das heißt, sollte die Spannung des Generators im Anschluss auf Grund dieses starken Strompulses kurzzeitig einbrechen, stört dies den Betrieb des Sensors in keiner Weise, da in diesem Zeitraum keine Messungen der Zellspannung vorgenommen werden. Die restlichen Impulse reichen nur knapp über $200\,\mathrm{mA}$. Somit ist davon auszugehen, dass eine Quelle mit maximal $250\,\mathrm{mA}$ ausreichend ist, um die aktuellen Sensoren mit Strom zu versorgen. Es ist jedoch erstrebenswert, eine Quelle zu entwerfen, die höhere Ströme liefern kann, um auch zukünftige Sensoren versorgen zu können. Insbesondere die Stromaufnahme der Klasse 3 Sensoren wird voraussichtlich noch höher ausfallen als bei der Klasse 2.

2.1.3 Auflösung

Der dritte und letzte Parameter, der einen Sensor beschreibt, ist die Auflösung des ADC (Analog Digital Converter), den er verwendet, um die Spannung der Batteriezelle zu messen. Beim Sensor der Klasse 1 kommt eine $2{,}5\,\mathrm{V}$ Referenzspannungsquelle und ein Spannungsteiler, der die Zellspannung um den Faktor $v_1 = \frac{1}{3}$ herabsetzt, zum Einsatz. Der ADC hat eine Auflösung von $N_1 = 10\,\mathrm{Bit}$ [12], [17]. Somit ergibt sich für die Auflösung der Klasse 1 Sensoren:

$$U_{LSB,1} = \frac{1}{v_1} \cdot \frac{U_{ref,1}}{2^{N_1}} = 3 \cdot \frac{2{,}5\,\mathrm{V}}{2^{10}} \approx 7{,}32\,\mathrm{mV} \tag{2.1}$$

Bei den Sensoren der Klasse 2 wird ein AD-Umsetzer mit $N_2 = 12\,\mathrm{Bit}$ genutzt und der Spannungsteiler vor dem ADC reduziert die Spannung um den Faktor $v_2 = \frac{1}{2}$. Als Referenz wird ebenfalls eine $2{,}5\,\mathrm{V}$ Spannungsquelle verwendet [7], [22]. Somit ergibt sich die Auflösung wie zuvor zu:

$$U_{LSB,2} = \frac{1}{v_2} \cdot \frac{U_{ref,2}}{2^{N_2}} = 2 \cdot \frac{2{,}5\,\mathrm{V}}{2^{12}} \approx 1{,}22\,\mathrm{mV} \tag{2.2}$$

Da der Klasse-2 Sensor hier die deutlich bessere Auflösung hat, ist er mit $U_{LSB,2} = 1{,}22\,\mathrm{mV}$ maßgebend für die Konstruktion des Zellspannungsgenerators.

2.1.4 Fehlerursachen

Bei einer Messung des Sensors mit dem ADC gibt es eine Reihe von Fehlerursachen, die eine Messung verfälschen können. Eine dieser Ursachen ist der Quantisierungsfehler, der auf Grund der begrenzten Auflösung entsteht. Entscheidend für die Genauigkeit des Sensors ist zudem die Genauigkeit der einzelnen Komponenten, wie dem ADC selbst und dessen

Analogbeschaltung. Hier lassen sich auch die Ursachen finden, auf Grund derer eine Kalibrierung der Sensoren notwendig ist.

Die entstehende Toleranz des Gesamtsystems soll hier exemplarisch am Zellensensor der Klasse 1 gezeigt und berechnet werden. Zunächst hat die interne Referenzspannungsquelle des Mikrocontrollers eine Toleranz. Diese beträgt bei einer Nominalspannung von $2{,}5\,\mathrm{V}$ laut dem Datenblatt $\pm 9\,\mathrm{mV}$ [17]. Zwei weitere Parameter beeinflussen die Genauigkeit des ADC, der Offset- und der Verstärkungsfehler. Diese sind mit $\pm 4\,\mathrm{LSB}$ beziehungsweise $\pm 2\,\mathrm{LSB}$ angegeben. Eine weitere Ungenauigkeit kommt noch durch die externe Beschaltung hinzu, die Toleranz des verwendeten Spannungsteilers. Gewöhnliche SMD-Metallschichtwiderstände haben eine Toleranz von $1\,\%$. Es kommt natürlich noch die INL (Integrale Nicht Linearität) und die DNL (Differentielle Nicht Linearität) hinzu, die mit einer aufwendigeren Kalibrierung ebenfalls korrigiert werden können. Um diese Werte und deren Auswirkungen zu bestimmen, werden die MATLAB-Skripte unter A.3.2.1 und A.3.2.2 verwendet. Diese Skripte berechnen den schlimmsten Fall der Abweichungen für den gesamten Spannungsbereich. Das Ergebnis ist in Abbildung 2.4 zu sehen. Auf Grund des Offsets steigt der prozentuale Fehler für kleine Werte stark an und geht für $U_{in} = 0\,\mathrm{V}$ gegen unendlich. Ab ca. $400\,\mathrm{mV}$ liegt der Fehler unter $10\,\%$ und pendelt sich für größere Eingangswerte bei knapp unter $4\,\%$ ein.

Hinzu kommt die Temperaturabhängigkeit, die die Referenzspannungsquelle stark beeinflussen kann. Laut [17] wird hier ein Wert von $\pm 100\,\frac{\mathrm{ppm}}{^\circ\mathrm{C}}$ angegeben. Da der Sensor später hauptsächlich im Automotive-Bereich eingesetzt werden soll, muss dieser auch für den Temperaturbereich von $-40\,^\circ\mathrm{C}$ bis $+125\,^\circ\mathrm{C}$ tauglich sein. Der MSP430 Prozessor ist für einen Temperaturbereich von $-40\,^\circ\mathrm{C}$ bis $+85\,^\circ\mathrm{C}$ spezifiziert. Betrachtet man die Abweichung in Relation zu der Abweichung bei Raumtemperatur ($20\,^\circ\mathrm{C}$), ergibt sich hierdurch eine zusätzliche Abweichung von:

$$e = \pm 100\,\frac{\mathrm{ppm}}{^\circ\mathrm{C}} \cdot 65\,^\circ\mathrm{C} = \pm 6500\,\mathrm{ppm} = \pm 0{,}65\,\% \tag{2.3}$$

Die Meisten der hier aufgeführten Fehler lassen sich mit einer Offset- und Gain-Kompensation beheben. Für die Korrektur des Temperaturfehlers muss mehr Aufwand betrieben werden, z.B. können verschiedene Korrekturfaktoren abgelegt werden, die je nach Umgebungstemperatur verwendet werden. Die INL und die DNL können zwar mit einer Kalibrierung ebenfalls deutlich verbessert werden, jedoch ist dies mit einer Offset- und Gain-Kompensation nicht möglich. Hierfür ist eine Kalibrierung nötig, die durch mehr als zwei Punkte bestimmt wird.

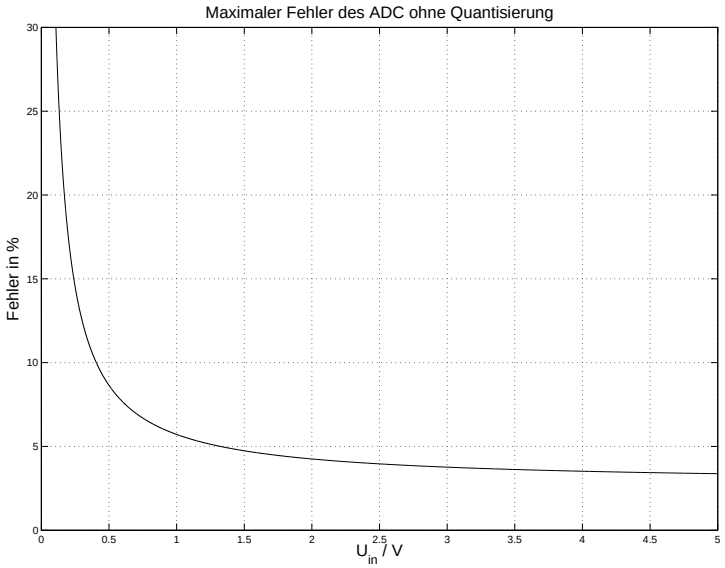


Abbildung 2.4: Maximaler ADC Fehler für einen Klasse-1 Sensor, ohne Temperaturabhängigkeit

2.2 Kalibrierung der Sensoren

Für eine gute Kalibrierung eines Sensors ist die Genauigkeit des Zellspannungsgenerators von großer Bedeutung. Diese muss im Bereich der Auflösung des Sensors liegen. Der Klasse-2 Sensor hat, wie im Kapitel 2.1.3 gezeigt, die höchste Auflösung mit $U_{LSB} = 1{,}22\,\mathrm{mV}$. Um auch diesen noch ausreichend genau kalibrieren zu können, müsste die Genauigkeit des Generators folglich unter $1{,}22\,\mathrm{mV}$ liegen.

Um diese Genauigkeit zu erreichen, ist es nötig, ein ausreichend präzises Signal zu generieren und dies zu verstärken. Die Generierung erfolgt mit einem DA-Umsetzer, welcher eine Referenzspannungsquelle benötigt. Anschließend muss das Signal gefiltert und verstärkt werden. Abbildung 2.5 verdeutlicht den Aufbau. Jede dieser vier Kernkomponenten (Referenzspannungsquelle, DA-Umsetzer, Formfilter und Verstärker) erzeugt Offset- und Verstärkungsfehler im Signal. Besonders betroffen sind hier der Formfilter und der Verstärker, da hier mit Widerständen gearbeitet wird, die Verstärkungsfehler verursachen. Zudem hat jeder Verstärker einen Offset Fehler.

Um dieses Problem zu umgehen, wird das Signal hinter dem Verstärker wieder abgegriffen und mit einem AD-Umsetzer zurück zum Mikrocontroller geführt, wie in Abbildung 2.6 gezeigt. So kann z.B. eine Regelschleife aufgebaut werden. Nun spielen zwar die vorher genannten Fehler keine Rolle mehr, da diese kompensiert werden können, es sind jedoch zwei neue dazu gekommen. Zum einen wieder die Referenzspannung für den AD-Umsetzer und

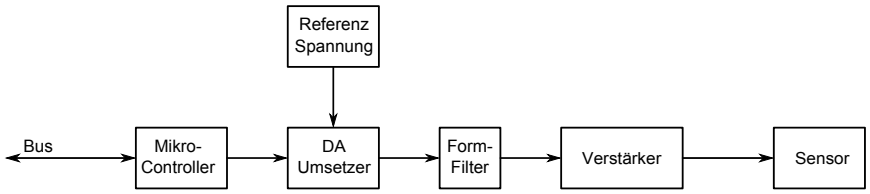


Abbildung 2.5: Signalpfad vom Mikrocontroller zum Generatorausgang

zum anderen der AD-Umsetzer selbst. Somit bleibt der Fehler durch eine Referenzspannungsquelle erhalten, der Fehler des AD-Umsetzers lässt sich mit dem des DA-Umsetzers vergleichen. Es fallen jedoch alle Ungenauigkeiten weg, die durch den Formfilter und den Verstärker erzeugt werden.

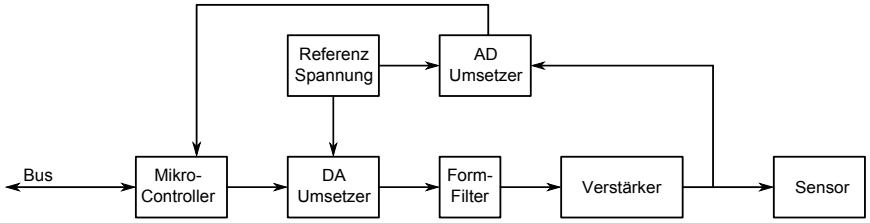


Abbildung 2.6: Signalpfad vom Mikrocontroller zum Generatorausgang und Rückführung über AD-Umsetzer

Um auch die Fehler der Referenzspannungsquelle und des AD-Umsetzers zu eliminieren, ist eine Kalibrierung jedes einzelnen Zellspannungsgenerators notwendig. Somit hängt die Genauigkeit nur noch von dem externen Kalibrierwerkzeug ab. Ist die Referenzspannungsquelle und der AD-Umsetzer einmal kalibriert, ist der Generator in der Lage, mit einer Regelschleife die Ausgangsspannung exakt einzustellen oder er kann den Signalpfad vom Mikrocontroller bis zum Verstärkerausgang selbst kalibrieren, wodurch eine Regelung entfällt.

2.3 Transientes Verhalten

Der zweite Modus, in dem der Zellspannungsgenerator betrieben werden soll, ist der Transienten-Modus. Hierbei sollen zeitliche Verläufe der Ausgangsspannung ausgegeben werden, die zuvor mit einem Oszilloskop an einer echten Batterie aufgenommen oder synthetisch erzeugt wurden. Diese Funktion soll primär verwendet werden, um das Verhalten der Sensoren zu testen. Ein Ziel dabei ist, das Abtastverhalten der Sensoren zu untersuchen. Beispielsweise muss der Startvorgang eines Autos, in dem deutlich mehr Informationen stecken als in langen Ruhephasen, hochfrequenter abgetastet werden. Um das Verhalten der

Sensoren in solchen Momenten vergleichen zu können, ist die Reproduzierbarkeit dieser Ereignisse von großer Bedeutung. Der zweite wichtige Punkt ist die Analyse der Funkübertragungsstrecke. Da die Bandbreite des Kanals stark beschränkt ist, ist eine effiziente Nutzung jener Bandbreite erforderlich. Speziell in den zuvor erwähnten Hochstromereignissen, in denen ein höheres Datenaufkommen besteht, ist die Analyse des Funkverhaltens der Sensoren sehr wichtig. Um dieses Verhalten optimieren zu können, ist ebenfalls eine Reproduzierbarkeit der Eingangsspannungsverläufe an den Sensoren nötig.

Für beide Fälle ist die absolute Genauigkeit der Signale nicht so entscheidend wie bei der Kalibrierung der Sensoren. Auf Grund dessen soll der AD-Umsetzer nicht als Rückkopplung genutzt werden, um eine Regelung der Ausgangsspannung aufzubauen. Es soll bei der Kalibrierung des Generators eine Wertetabelle von Ausgangsspannung zu DA-Umsetzer-Wert erstellt werden, die dann bei der Wiedergabe der Samples genutzt wird. Somit ist es möglich, deutlich höhere Bandbreiten zu erzielen, da die Rückkopplung über den AD-Umsetzer zu viel Zeit benötigt. Obwohl die Rückführung nicht verwendet wird, kann durch die Nutzung der Wertetabelle dennoch eine hohe Genauigkeit erreichen werden.

Im Gegensatz zur Kalibrierung ist bei der Wiedergabe von veränderlichen Spannungssequenzen die Auflösung, genau wie die Bandbreite des Signals, von größerer Bedeutung. Die Auflösung sollte hier im Bereich der Auflösung des Sensors liegen, um die Fähigkeiten des Sensors voll auszuschöpfen. Somit sollte die Auflösung nach Kapitel 2.1.3 mindestens $1{,}22\,\mathrm{mV}$ betragen. Zur Ermittlung der Bandbreite sind genauere Analysen notwendig. Hierfür wird der Spannungsverlauf des Startvorgangs eines PKW genauer untersucht. Das Signal wurde im Rahmen der Arbeit von S. Püttjer an einem Mercedes Benz Vito L, 4 Zylinder Diesel, 95kW, 2,0l gemessen [13].

Das Signal, welches im unteren Teil der Abbildung 2.7 dargestellt ist, soll mit Hilfe einer FFT in den Frequenzbereich transformiert werden, um dort die Bandbreite des Signals zu begrenzen. Anschließend soll das gefilterte Spektrum zurück in den Zeitbereich transformiert werden, wo es dann mit dem unveränderten Zeitsignal verglichen werden kann. Das Ziel ist es eine Bandbreite zu finden, bei der das Signal ohne zu große Unterschied zum unveränderten Signal wiedergegeben werden kann.

Transformiert man das Signal so wie es ist in den Frequenzbereich, filtert es dort und transformiert es wieder zurück, entstehen auf Grund des Leck-Effekts nicht erwünschte Anteile im Signal. Dies liegt an der periodischen Fortsetzung des Signals durch die FFT. Es entsteht am Anfang beziehungsweise am Ende des Signals ein Sprung, welcher sich primär in hochfrequenten Anteilen im Spektrum bemerkbar macht. Da diese Anteile jedoch herausgefiltert werden, wird das Signal bei der Rücktransformation in den Zeitbereich verfälscht. Um diesen Effekt zu vermeiden, wird das Zeitsignal mit einem Fenster multipliziert. Dabei wird das Signal zum Anfang und zum Ende hin nach 0 ausgeblendet, wodurch der Sprung vermieden wird. Da viele der benötigten Informationen zu Beginn des Signals vorhanden sind, wird ein Fenster gewählt, welches sehr schnell den Faktor eins erreicht. Hierfür eignet

sich das Tukey-Fenster mit dem Parameter $r = 0{,}1$, wodurch das Fenster nach einer Sekunde den Wert eins erreicht. In Abbildung 2.7 ist das Fenster und das unveränderte Zeitsignal zu sehen.

Das gefensterte Signal und dessen Spektrum wird in Abbildung 2.8 gezeigt. In Abbildung 2.9 wird dann das Spektrum bei einer bestimmten Frequenz (hier $2{,}5\,\mathrm{kHz}$) abgeschnitten und zurück in den Zeitbereich transformiert. Im Zeitbereich ist wieder das zuvor angewendete Fenster zu erkennen, das Signal blendet zu den Seiten hin nach 0 aus. Anschließend soll die Differenz des Originalsignals zu dem tiefpassgefilterten Signal im Zeitbereich bestimmt werden. Da das gefilterte Signal jedoch mit dem Fenster beaufschlagt ist, muss die Auswirkung des Fensters entweder wieder rückgängig gemacht werden, oder das unveränderte Signal muss vor dem Vergleich ebenfalls mit dem Fenster multipliziert werden. Um die Auswirkungen des Fensters rückgängig zu machen, kann man das gewonnene Zeitsignal durch das Fenster teilen, jedoch führt dies um die Nullstellen am Anfang und am Ende des Signals zu numerischen Problemen. Daher wird das unveränderte Signal ebenfalls mit dem Fenster multipliziert und anschließend vom gefilterten Signal subtrahiert.

Es soll eine Grenzfrequenz gefunden werden, bei der die Abweichungen zwischen dem unveränderten und dem gefilterten Signal im Zeitbereich ausreichend gering sind. Es zeigt sich, dass eine Grenzfrequenz von $2{,}5\,\mathrm{kHz}$ ausreichend ist, um den Verlauf des Signals nachbilden zu können, ohne, dass zu große Abweichungen auftreten. In der Abbildung 2.10 ist die Differenz der beiden Signale dargestellt. Würde man anhand dieser Grafik einen Bereich suchen, in dem die meisten Abweichungen liegen, käme man auf einen Wert von ca. $-10\,\mathrm{mV}$ bis $5\,\mathrm{mV}$. Dies ist jedoch nur eine Täuschung durch die Darstellung, da einige große Werte viele kleinere überdecken, wie in Abbildung 2.11 zu sehen ist. Hier würde man eher einen Wert von $-3\,\mathrm{mV}$ bis $2\,\mathrm{mV}$ angeben. Stellt man die Werte in einem Histogramm dar, wird die Verteilung der Abweichungen noch deutlicher, wie Abbildung 2.12 zeigt. Die vorherige Schätzung stimmt mit dieser Darstellung gut überein. Um dies auch rechnerisch zu untermauern, wird die Standardabweichung der Differenz berechnet. Dies geschieht mit MATLAB, sie beträgt $\sigma = 871\ \mu\mathrm{V}$, wie das Skript aus Anhang A.3.2.5 berechnet. Geht man von einer Normalverteilung aus, bedeutet dies, dass $68{,}3\,\%$ der Samples eine Abweichung im Bereich von $\pm\sigma$ haben. Da es sich in diesem Fall um keine perfekte Normalverteilung handelt, sind es sogar $77{,}5\,\%$, wie das MATLAB-Programm zeigt.

Bei der Wahl der Samplerate sind nun noch zwei Faktoren zu beachten. Zum einen muss das Abtasttheorem eingehalten werden, was dann eine Samplerate von mindestens $5\,\mathrm{kSPS}$ ergeben würde und zum anderen muss bedacht werden, dass hier mit einem nahezu idealen Filter gerechnet wurde, der in der Praxis nicht realisierbar ist. Es wird stattdessen ein analoger Filter erster oder zweiter Ordnung zum Einsatz kommen. Folglich muss die Abtastrate nochmals höher gewählt werden, wobei dies nicht beliebig möglich ist. Vor allem ist hier die begrenze Übertragungsrate zu den Modulen und deren Speicherfähigkeit ausschlaggebend, denn es sollen Sequenzen von bis zu 60 Sekunden wiedergegeben werden können. Es wird daher eine Samplerate von $20\,\mathrm{kSPS}$ gewählt. Dies ist eine vier fache Überabtastung und ist

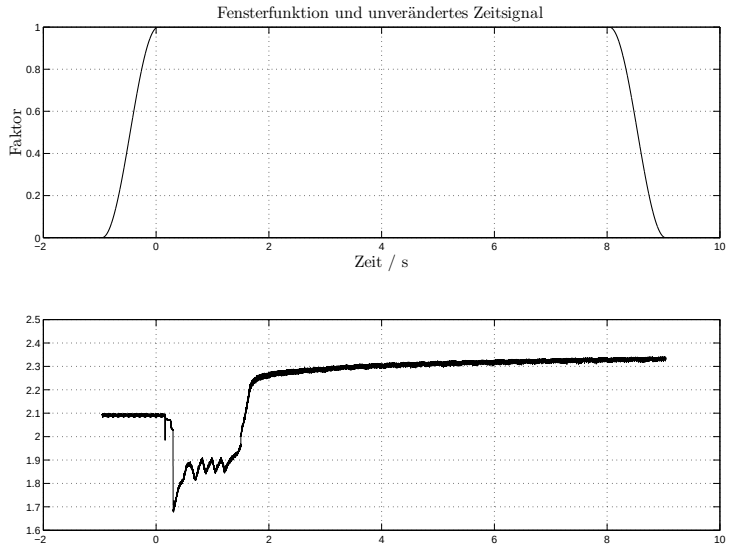


Abbildung 2.7: Oben: Gewähltes Tukey-Fenster, Unten: Unverändertes Zeitsignal, wie es vom Oszilloskop aufgenommen wurde multipliziert mit dem Fenster

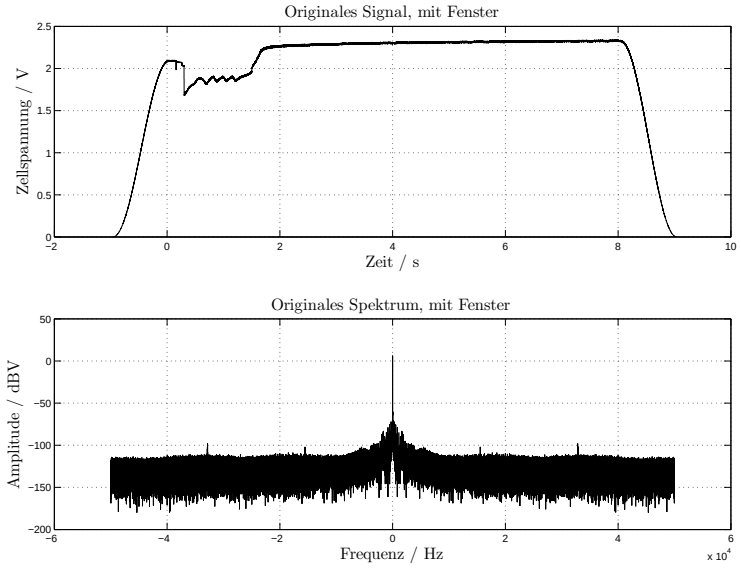


Abbildung 2.8: Oben: Gefenstertes Zeitsignal des Startvorgangs eines PKW, Unten: Spektrum des gefensterten Signals

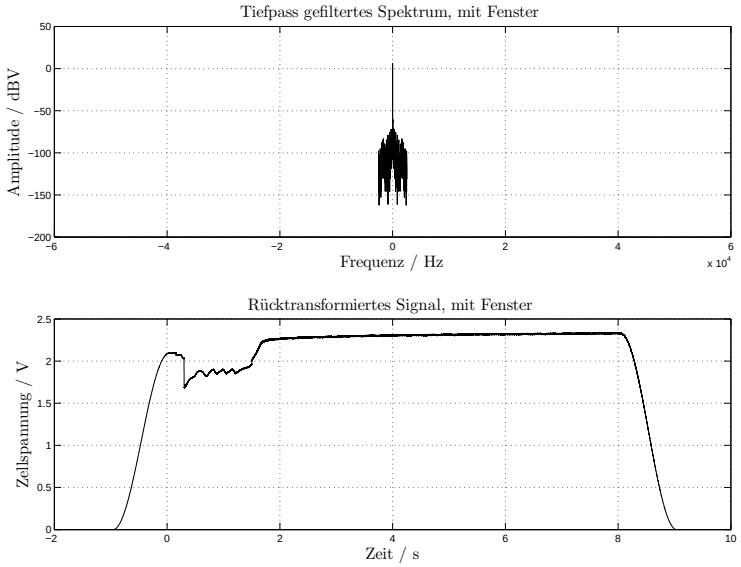


Abbildung 2.9: Oben: Bandbegrenztes Spektrum des gefensterten Startvorganges, Unten: Rücktransformiertes Signal nach der Bandbegrenzung im Spektrum, $f_g = 2{,}5\,\mathrm{kHz}$

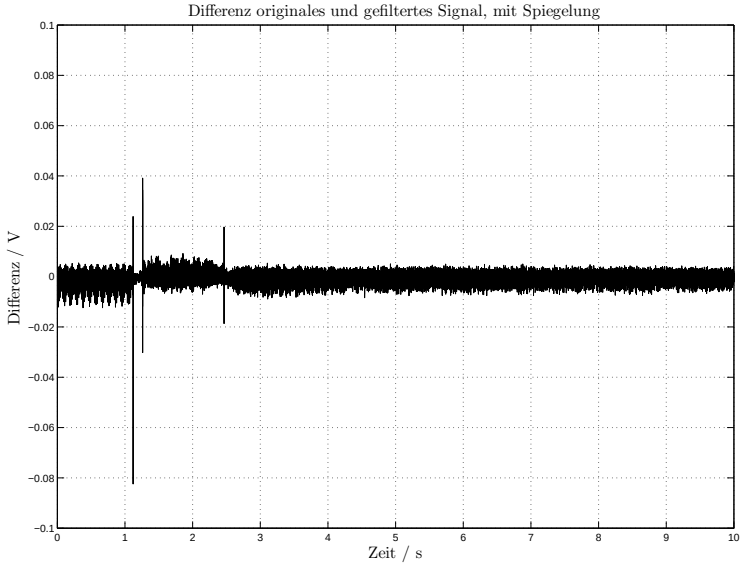


Abbildung 2.10: Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$

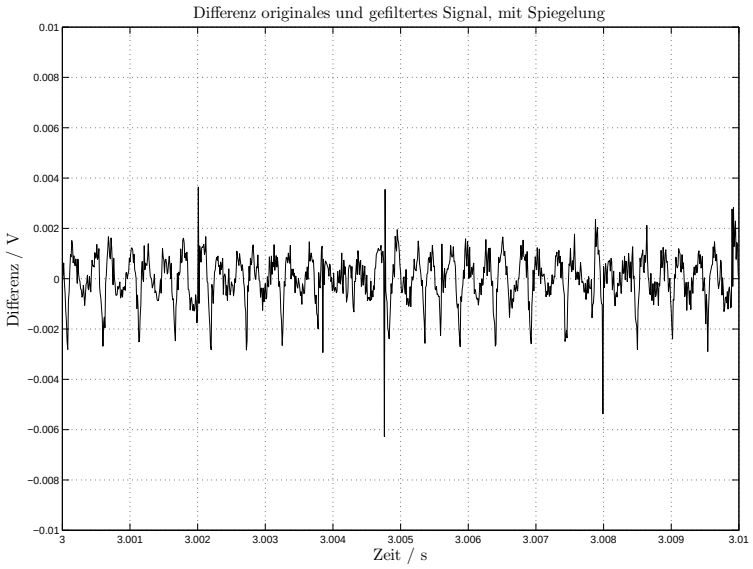


Abbildung 2.11: Kleiner Ausschnitt der Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$

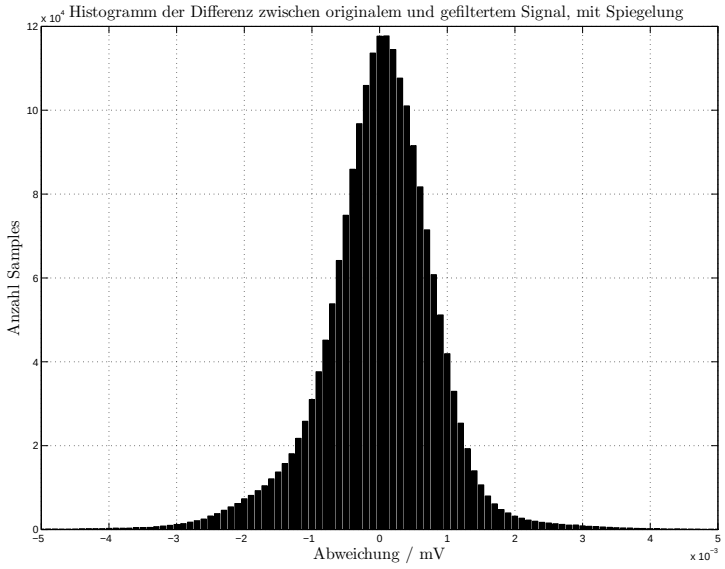


Abbildung 2.12: Histogramm der Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$

für einen Filter zweiter Ordnung ausreichend, wie sich später in Kapitel 4.2 noch zeigen wird.

2.4 Zusammenfassung der Anforderungen

Die Anforderungen an den Zellspannungsgenerator sind nun definiert und sollen folgend nochmals übersichtlich in der Tabelle 2.1 zusammen gefasst werden.

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

Tabelle 2.1: Zusammenfassung der Anforderungen an den Zellspannungsgenerator

2.5 Vernetzung der Module

Die einzelnen Zellspannungsgeneratoren sollen modular aufgebaut sein und es soll so möglich sein, mindestens 40 dieser Module gleichzeitig zu betreiben. Diese Anzahl an Zellen kommt beispielsweise bei Gabelstaplerbatterien zum Einsatz, die eine Nennspannung von $80\,\mathrm{V}$ haben, bei einer Nennspannung von $2\,\mathrm{V}$ je Blei-Zelle. Um die Module mit Daten zu versorgen ist ein Bussystem notwendig, welches eine Verbindung zwischen den einzelnen Modulen herstellt und auch die Integration in MATLAB ermöglicht. Um eine Wahl für das Bussystem zu treffen, soll zunächst die benötigte Datenrate und Datenmenge untersucht werden.

2.5.1 Übertragungsrate und Datenmenge

Ein wichtiger Gesichtspunkt ist die Übertragungsrate des Busses zu jedem der Module. Um diese bewerten zu können, muss zunächst die benötigten Datenrate ermittelt werden. Bekannt ist die Samplerate mit $v = 20\,\mathrm{kSPS}$ und die minimale Auflösung von $1{,}22\,\mathrm{mV}$

bei einer Maximalamplitude von $5{,}5\,\mathrm{V}$. Zunächst wird die Anzahl der Stufen bestimmt, die benötigt werden, um die Auflösung zu erreichen:

$$N_{min} = \frac{U_{ref}}{U_{LSB}} = \frac{5{,}5\,\mathrm{V}}{1{,}22\,\mathrm{mV}} \approx 4508 \tag{2.4}$$

Woraus sich dann die Anzahl an Bits pro Sample errechnen lassen:

$$d_{min} = \lceil ld(N_{min}) \rceil\,\mathrm{Bit} = \lceil ld(4508) \rceil\,\mathrm{Bit} = \lceil 12{,}14 \rceil Bit = 13\,\mathrm{Bit} \tag{2.5}$$

Somit lässt sich die minimale Datenrate wie folgt berechnen:

$$C_{min} = d_{min} \cdot v = 13\,\mathrm{Bit} \cdot 20\,\mathrm{kSPS} = 260\,\frac{\mathrm{kBit}}{\mathrm{s}} \tag{2.6}$$

Dabei ist zu beachten, dass die Bandbreite C_{min} von einem einzigen Modul benötigt wird. Da es zusätzlich möglich sein soll, verschiedene Daten an die einzelnen Module zu schicken, wird bei $n = 40$ Modulen eine Datenrate von insgesamt

$$C_{min,40} = C_{min} \cdot n = 260\,\frac{\mathrm{kBit}}{\mathrm{s}} \cdot 40 = 10{,}4\,\frac{\mathrm{MBit}}{\mathrm{s}} \tag{2.7}$$

benötigt. Diese Daten in Echtzeit zu verschicken, ist nahezu ausgeschlossen, zumal es nicht einfach ist, ein echtzeitfähiges Programm in MATLAB zu implementieren, welches in der Lage ist, diese Datenmengen aufzubereiten und an alle Module mit minimalem Puffer zu senden. Dies führt dazu, dass die Daten auf den Modulen zwischengespeichert werden müssen und dann synchron ausgegeben werden. Somit kann die Datenübertragung langsamer sein als die Samplerate der einzelnen Module. Dies führt zum einen dazu, dass auf jedem Modul ein ausreichend großer Speicher vorhanden sein muss, der die gesamte Sequenz zwischenspeichern kann, zum anderen stellt sich die Frage, wie lange die Programmierung der Module mit den Daten dauern darf. Um dies genauer zu betrachten, muss zunächst die zu übertragene Datenmenge pro Modul bestimmt werden. Hierfür ist die Samplerate mit $v = 20\,\mathrm{kSPS}$, die Größe pro Sample mit $d_{min} = 13\,\mathrm{Bit}$ und die Sequenzdauer von $T_{max} = 60\,\mathrm{s}$ bekannt:

$$B = d_{min} \cdot v \cdot T_{max} = 13\,\mathrm{Bit} \cdot 20\,\mathrm{kSPS} \cdot 60\,\mathrm{s} = 15{,}6\,\mathrm{MBit} \tag{2.8}$$

Bei $n = 40$ Modulen ergibt sich somit eine zu übertragende Gesamtgröße von:

$$B_{40} = B \cdot n = 15{,}6\,\mathrm{MBit} \cdot 40 = 624\,\mathrm{MBit} \tag{2.9}$$

2.5.2 Datenkompression

Um die Menge der zu übertragenden Daten weiter zu reduzieren, ist eine Quellencodierung möglich. Hierfür wird der Startvorgang eines Mercedes Benz Vito L, 4 Zylinder Diesel, 95kW, 2,0l als Vorlage für ein typisches Signal verwendet [13]. Zunächst wird das Signal in die DAC-Werte umgerechnet, sprich auf die Referenzspannung von $5{,}5\,\mathrm{V}$ bezogen und in $13\,\mathrm{Bit}$ umkodiert. Das quantisierte Signal ist in Abbildung 2.13 dargestellt. Anschließend wird das Signal digital differenziert, es wird also immer die Differenz zwischen zwei aufeinander folgenden Sampeln gebildet. Das Ergebnis ist ebenfalls in Abbildung 2.13 dargestellt. Bereits hier sieht man, dass sich die Anzahl der verschiedenen Werte bis auf wenige Ausreißer auf einen kleinen Bereich beschränkt. Die Verteilung der Werte wird zur Verdeutlichung nochmals in der Abbildung 2.14 in einem Histogramm dargestellt. Es ist zu erkennen, dass die meisten Werte ($98{,}78\,\%$) im Bereich von -5 bis 5 liegen. Kodiert man nun diese Werte im Bereich von -7 bis 7 benötigt man für jeden Sample 4 Bits und deckt damit $99{,}78\,\%$ aller Samples ab. Um die restlichen $0{,}22\,\%$ abdecken zu können, kann beispielsweise ein Escape-Zeichen verwendet werden, hierfür eignet sich der Wert -8. Nach der Escape-Sequenz kann der Abtastwert mit den vollen $13\,\mathrm{Bit}$ übertragen werden. Missachtet man die $0{,}22\,\%$ an Daten, die nicht direkt kodiert werden können, so kann die Datenrate um folgenden Faktor gesenkt werden:

$$l = \frac{4\,\mathrm{Bit}}{13\,\mathrm{Bit}} \approx 0{,}31 \tag{2.10}$$

Somit wird die benötigte Datenrate auf

$$C_{min,kod} = C_{min} \cdot l = 260\,\frac{\mathrm{kBit}}{\mathrm{s}} \cdot 0{,}31 = 80{,}6\,\frac{\mathrm{kBit}}{\mathrm{s}} \tag{2.11}$$

pro Sensor reduziert. Ebenso verringert sich die gesamt zu sendende Datenmenge für 40 Module um den Faktor l:

$$B_{40,kod} = B_{40} \cdot l = 624\,\mathrm{MBit} \cdot 0{,}31 \approx 193\,\mathrm{MBit} \tag{2.12}$$

Um die Kompression noch weiter zu verbessern, ist es möglich, das Signal wie zuvor digital zu differenzieren und zusätzlich mit dem Huffman-Code zu kodieren. Das Skript aus Anhang A.3.1.1 berechnet, dass bei einer Huffman-Kodierung $3{,}11\,\mathrm{Bit}$ pro Sample benötigt würden. Somit kann der Faktor l nochmals gesenkt werden:

$$l_{Huff} = \frac{3{,}11\,\mathrm{Bit}}{13\,\mathrm{Bit}} \approx 0{,}24 \tag{2.13}$$

Die Datenrate reduziert sich ebenfalls um den Faktor l:

$$C_{min,kod,huff} = C_{min} \cdot l = 260\,\frac{\mathrm{kBit}}{\mathrm{s}} \cdot 0{,}24 = 62{,}4\,\frac{\mathrm{kBit}}{\mathrm{s}} \tag{2.14}$$

Die Datenmenge reduziert sich auf:

$$B_{40,kod,Huff} = B_{40} \cdot l = 624\,\mathrm{MBit} \cdot 0{,}24 \approx 150\,\mathrm{MBit} \tag{2.15}$$

Bei beiden Arten der Kodierung ist jedoch zu bedenken, dass sie auf das hier verwendete Signal optimiert sind. Dies gilt speziell für die Huffman-Kodierung. Es kann also passieren, dass andere Signale deutlich mehr Daten bei der Übertragung benötigen und unter Umständen sogar mehr als das nicht komprimierte Signal.

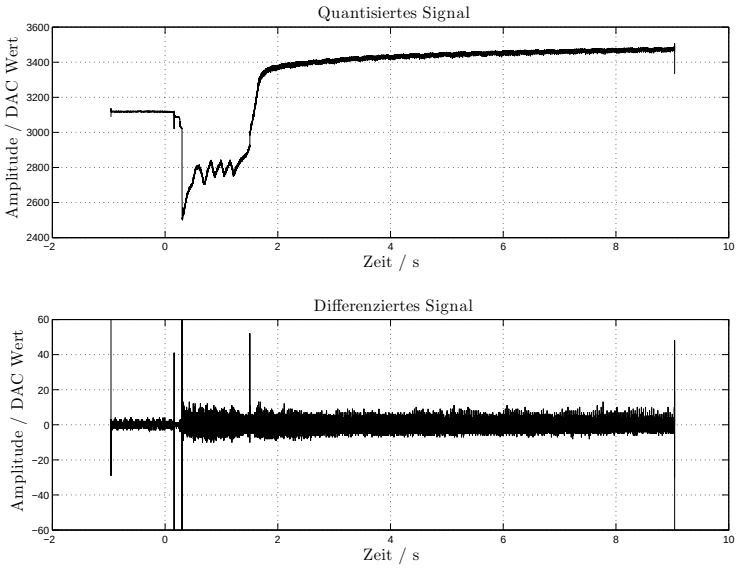


Abbildung 2.13: Umkodiertes Zellspannungssignal eines Startvorganges auf $13\,\mathrm{Bit}$ und $5{,}5\,\mathrm{V}$ Referenzspannung. Oben: Zeitsignal, welches umkodiert werden soll, unten: Differenzen eines Samples zu seinem vorherigen

2.5.3 Bussystem und Mikrocontroller

Für die Umsetzung stehen zwei verschiedene Prozessoren zur Auswahl. Ein 16-Bit Mikrocontroller der MSP430-Familie, die sich im Rahmen des Projektes BATSEN bereits vielfach bewährt hat oder ein 32-Bit ARM Cortex M3 Stellaris Mikrocontroller der Firma Texas Instruments. Beide Mikrocontroller bieten Standard-Bussysteme wie I^2C und SPI, der Stellaris hat zudem einen integrierten Ethernet MAC und PHY. Der Stellaris-Controller verfügt darüber hinaus über zwei CAN-Bus-Module. Der CAN Bus kann bei dem MSP430 auch mit einem externen IC auf einfache Weise nachgerüstet werden.

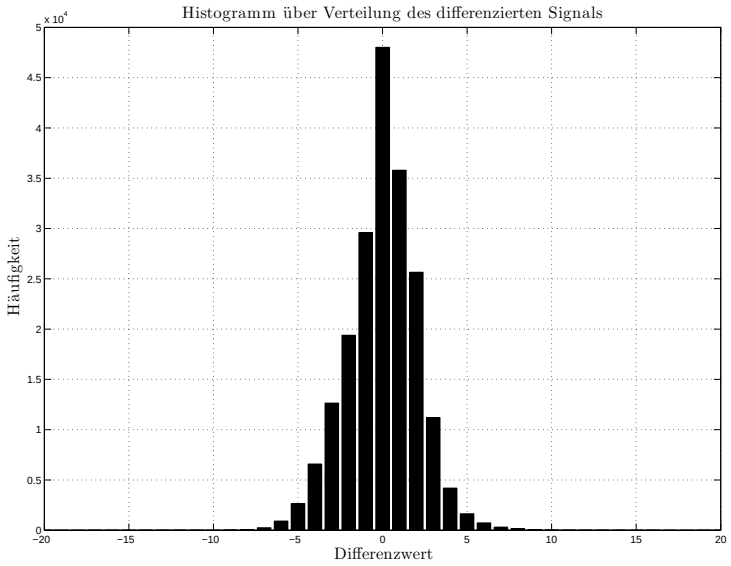


Abbildung 2.14: Histogramm der umkodierten Zellspannung eines Startvorganges auf $13\,\mathrm{Bit}$ und $5{,}5\,\mathrm{V}$ Referenz. $0{,}22\,\%$ können nicht direkt mit $4\,\mathrm{Bit}$ kodiert werden

Es stehen also insgesamt vier verschiedene Bussysteme zur Auswahl, um die Vernetzung der Generatoren vorzunehmen. Deren Vor- und Nachteile sollen nun genauer betrachtet werden. Generell wird an den Bus die Anforderung gestellt, mindestens 40 Module ansprechen zu können und eine Bandbreite zur Verfügung zu stellen, die die Übertragung der Daten in einer angemessenen Zeit erlaubt. Zudem muss es möglich sein, den Bus galvanisch zu trennen.

I²C

I²C ist ein serieller 2-Draht Datenbus, entwickelt von Philips Semiconductors (heute NXP Semiconductors), der primär schaltungsintern verwendet wird, um zwischen einzelnen ICs zu kommunizieren [10]. Er ist daher kein Feldbus und hat somit bei der Übertragung über größere Strecken Probleme, die eventuell durch geringere Taktraten ausgeglichen werden müssen. Nach [10] ist ein I²C-Master in der Lage, bis zu 127 Slaves einzeln oder im Broadcast anzusprechen. Der Broadcast ist ein wichtiger Vorteil, der es ermöglicht, die Programmierzeit der Module stark zu reduzieren, wenn alle Module die gleichen Daten bekommen sollen. Der in Frage kommende MSP430 Mikrocontroller unterstützt Taktraten von bis zu $400\,\mathrm{kHz}$ [22], was bei I²C für große Datenmengen direkt in die maximale Datenrate übertragen werden kann. Somit würde das Programmieren von 40 Sensoren mit unterschiedlichen

Daten unter Verwendung der einfachen Kompression

$$T_{I2C,40} = \frac{B_{40,kod}}{v_{I2C}} = \frac{150\,\mathrm{MBit}}{400\,\frac{\mathrm{kBit}}{\mathrm{s}}} = 375\,\mathrm{s} = 6:15\,\mathrm{Min} \tag{2.16}$$

in Anspruch nehmen. Der Soft- und Hardwareaufwand ist für I^2C minimal, da die meisten Mikrocontroller einen integrierten I^2C-Baustein besitzen, der den Großteil der Arbeit abnimmt, so auch der MSP430F235 [22]. Extern werden lediglich zwei Pull-Up-Widerstände benötigt, um die beiden Leitungen (Daten und Takt) auf einem festen Potential zu halten. Für die galvanische Trennung der Module untereinander wird zusätzlich nochmal jeweils ein IC benötigt, das diese Aufgabe übernimmt. Da sowohl die Datenleitung als auch die Taktleitung bidirektional ist, kann die Trennung nicht mit Optokopplern durchgeführt werden. Hierfür wird ein spezielles IC benötigt, wie zum Beispiel der ADUM1250 von Analog Devices [5].

SPI

SPI ist ebenfalls ein synchroner serieller Datenbus, ursprünglich entwickelt von Motorola. Im Gegensatz zu I^2C ist SPI Vollduplex fähig, wodurch eine Leitung mehr benötigt wird. Die Adressierung der Slaves erfolgt über Chip-Select-Leitungen von denen jeweils eine pro Slave benötigt wird. Dies würde bei über 40 Sensoren zu einem zu hohen Anschlussaufwand führen, wie Abbildung 2.15 verdeutlicht. Um dieses Problem zu umgehen, gibt es zwei Lösungen. Bei der ersten werden die Slaves alle in Serie geschaltet, somit ist ein Chip-Select überflüssig, da quasi alle Chips gleichzeitig arbeiten. Die zweite Variante ist, dass alle Slaves die ganze Zeit aktiv sind und auf Grund eines Befehls über die Empfangs-Datenleitung ihren Sende-Pin einschalten. Dies wird in Abbildung 2.17 dargestellt. Die erste Variante hat, wie Abbildung 2.16 zeigt, einen sehr geringen Hardware-Aufwand. Die galvanische Trennung kann mit einfachen Optokopplern vorgenommen werden. Das Software-Protokoll wird jedoch umfangreicher, wenn man auch Daten von den Slaves zum Master übertragen will, da diese Übertragungen dann koordiniert werden müssen. Zudem ist das System beim Ausfall eines einzigen Moduls vollständig unbrauchbar, da der Ring unterbrochen ist. Die zweite Variante benötigt etwas mehr externe Hardware, ist jedoch ausfallsicherer und hat eine einfachere Software. Natürlich ist es auch möglich, den SDO-Pin der Slaves in der Software direkt zwischen Ausgang und Tri-State umzuschalten, da jedoch eine galvanische Trennung erforderlich ist, muss der Treiber auf der anderen Seite der Trennung an- und abgeschaltet werden. Die galvanische Trennung selbst kann z.B. mit einfachen Optokopplern erfolgen. Bei beiden der vorgestellten Varianten kann man, mit einer passenden Software-Implementierung, den SPI-Bus als Broadcast und Multicast fähig betrachten.

Laut [22] ist der MSP430-Controller in der Lage, mit einer SPI-Taktrate von bis zu $16\,\mathrm{MBit/s}$ zu arbeiten. Wie aus dem Datenblatt hervorgeht läuft, der Mikrocontroller je-

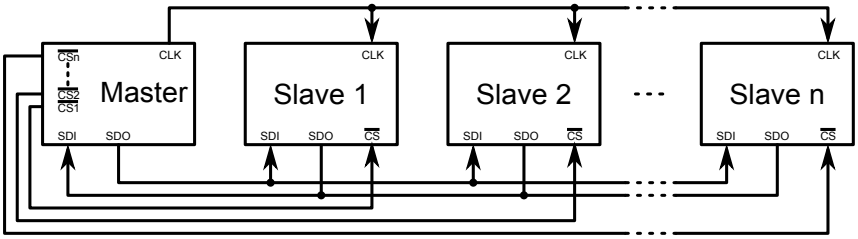


Abbildung 2.15: Klassischer Anschluss von mehreren SPI-Slaves an einen Master mit mehreren Chip-Select-Leitungen

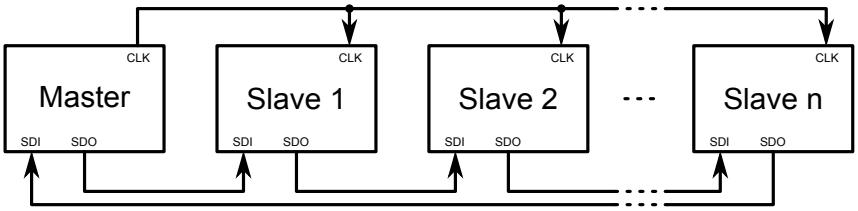


Abbildung 2.16: Anschluss von mehreren SPI-Slaves an einen Master durch Kaskadierung

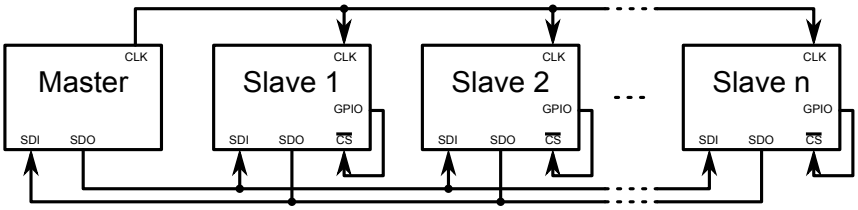


Abbildung 2.17: Anschluss von mehreren SPI-Slaves an einen Master, Implementierung des Chip-Selects in Software

doch nur mit maximal $16\,\mathrm{MHz}$, daher ist es fraglich, ob er diese Datenmengen auch verarbeiten kann. Denn jedes erhaltene Sample muss in einen Wert für den DA-Umsetzer umkodiert und abgespeichert werden. Dennoch soll hier die Zeit berechnet werden, die benötigt werden würde, um 40 Zellspannungsgeneratoren mit den einfach komprimierten Daten zu programmieren:

$$T_{SPI,40} = \frac{B_{40,kod}}{v_{SPI}} = \frac{150\,\mathrm{MBit}}{16\,\frac{\mathrm{MBit}}{\mathrm{s}}} \approx 9{,}38\,\mathrm{s} \tag{2.17}$$

CAN

Das Controller Area Network (CAN) wurde von Bosch ursprünglich für den Einsatz in Automobilen entwickelt. Der CAN-Bus ist sehr robust und ausfallsicher, da es sich um einen Feldbus handelt. Er kann mit maximal $1\,\mathrm{MHz}$ betrieben werden, wobei zu beachten ist, dass die Datenraten deutlich geringer sind, denn eine CAN-Nachricht beinhaltet maximal 64 Bit Daten, während der Header konstant 53 Bit lang ist (11 Bit Adresse). Somit werden maximal

$$p_{CAN} = \frac{64\,\mathrm{Bit}}{64\,\mathrm{Bit} + 53\,\mathrm{Bit}} = 54{,}7\,\% \tag{2.18}$$

der Zeit effektiv Daten übertragen. Was eine Datenrate von maximal $v_{CAN} = 547\,\frac{\mathrm{kBit}}{\mathrm{s}}$ ergibt. Überträgt man nun die Daten für 60 Sekunden an 40 Module mit einfacher Kompression, ergibt sich eine benötigte Zeit von

$$T_{SPI,40} = \frac{B_{40,kod}}{v_{CAN}} = \frac{150\,\mathrm{MBit}}{547\,\frac{\mathrm{kBit}}{\mathrm{s}}} \approx 274\,\mathrm{s} = 4:34\,\mathrm{Min} \tag{2.19}$$

Da beim CAN-Bus nicht der Master den Slave adressiert, sondern jeder Slave entscheiden kann, von welchem Master er Nachrichten erhalten möchte, muss die Adressierung der Slaves mit in den Daten stehen, was die effektive Bandbreite weiter verringert, wodurch dann allerdings auch Broadcast und Multicast Übertragungen möglich sind. Die galvanische Trennung des Busses erfolgt durch ein zusätzliches IC, wie zum Beispiel dem ADM3054 von Analog Devices [4].

Ethernet

Ethernet ist ein sehr schneller und flexibler Bus, der ursprünglich gedacht war, um Computer, Drucker und ähnliche Geräte miteinander zu verbinden. Es werden Datenraten von bis zu $10\,\mathrm{GBit/s}$ unterstützt. Der Stellaris Mikrocontroller unterstützt jedoch nur $100\,\mathrm{MBit/s}$. Ethernet bietet beispielsweise durch UDP Multicast und Broadcast Übertragungen an. Des

weiteren bringt Ethernet durch die Übertrager in den Buchsen direkt eine galvanische Trennung mit und es ist, im Gegensatz zu allen anderen bislang vorgestellten Bussystemen, nicht nötig, einen Adapter zu konstruieren, der den Bus für den PC beispielsweise in USB übersetzt, da die meisten PCs bereits eine Ethernet-Netzwerkkarte integriert haben. Zudem ist Ethernet der bislang schnellste Bus, wodurch es bei $100\,\mathrm{MBit/s}$ möglich sein sollte, 40 Module mit den einfach komprimierten Daten innerhalb von

$$T_{Ethernet,40} = \frac{B_{40,kod}}{v_{Ethernet}} = \frac{150\,\mathrm{MBit}}{100\,\frac{\mathrm{MBit}}{\mathrm{s}}} = 1{,}5\,\mathrm{s} \tag{2.20}$$

zu programmieren. Ein Nachteil von Ethernet ist der recht hohe Hardware- und Software-Aufwand, welcher jedoch stark reduziert werden kann durch die Verwendung des Evalboards EKI-LM3S9B92 von Texas Instruments. Auf diesem Board ist ein Stellaris LM3S9B92-Mikrocontroller verbaut, sowie eine Ethernetbuchse. Die meisten GPIOs sind auf Stiftleisten herausgeführt, was den Anschluss auf ein Baseboard als Steckmodul ermöglicht. Zudem wird ein JTAG-Programmiergerät mitgeliefert. Texas Instruments bietet mit StellarisWare ein umfangreiches Softwarepaket zur Steuerung des Mikrocontrollers und dessen Peripherie. Insbesondere ist in diesem Softwarepaket schon ein TCP/IP-Stack implementiert und mit Beispielen gut dokumentiert. Durch Verwendung des Evalboards und dem Softwarepaket StellarisWare kann so der Soft- und Hardwareaufwand als gering eingestuft werden.

Vergleich der Systeme

In Tabelle 2.2 sollen nun nochmals alle vorgestellten Bussysteme verglichen werden, um anschließend eine Entscheidung zu treffen.

Tabelle 2.2: Vergleich der Bussysteme

	I²C	**SPI**	**CAN**	**Ethernet**
Geschwindigkeit	$400\,\mathrm{kBit/s}$	$16\,\mathrm{MBit/s}$	$547\,\mathrm{kBit/s}$	$100\,\mathrm{MBit/s}$
Bustakt	$400\,\mathrm{kHz}$	$16\,\mathrm{Mhz}$	$1\,\mathrm{MHz}$	$125\,\mathrm{MHz}$
Programmierzeit	$6:15\,\mathrm{Min}$	$9{,}38\,\mathrm{s}$	$4:34\,\mathrm{Min}$	$1{,}5\,\mathrm{s}$
Hardwareaufwand	hoch	hoch	hoch	gering
Softwareaufwand	gering	gering	mittel	gering
Mikrocontroller	MSP/ARM	MSP/ARM	MSP+CAN-IC/ARM	ARM
Broadcast	ja	Software	Software	ja
Multicast	Software	Software	Software	ja

Beim Hardwareaufwand ist zu beachten, dass alle Konzepte bis auf die Nutzung von Ethernet eine zusätzliche Platine benötigen, um den Bus mit dem PC zu verbinden. Anhand

dieser Tabelle ist relativ schnell klar, welches der vorgestellten Konzepte am einfachsten umzusetzen ist und gleichzeitig einen geringen Softwareaufwand bedeutet. Das Stellaris-Mikrocontroller Evalboard ist die Wahl mit dem geringsten Hardwareaufwand und bietet mit umfangreichen Softwarebeispielen und Bibliotheken eine gute Basis für die Entwicklung der Software. Ein weiterer Vorteil des Stellaris Mikrocontrollers gegenüber dem MSP430 ist, dass dieser deutlich schneller läuft und einen effizienteren Prozessor hat, also weniger Takte pro Instruktion benötigt [25], [22].

Synchronisation

Werden mehrere Module parallel betrieben, die gemeinsam eine Batterie simulieren sollen, so ist die Synchronität der Module von großer Wichtigkeit. Ethernet bietet im Gegensatz zu allen anderen Bussystemen keine Möglichkeit der hoch genauen Synchronisation. Beispielsweise bei SPI oder I^2C kann die Taktleitung als Synchronisation verwendet werden. Um eine exakte Synchronisation auch bei Ethernet zu ermöglichen, ist eine weitere Leitung notwendig, die ebenfalls galvanisch getrennt werden muss. Eines der Module muss dann als Master arbeiten und den Takt für die anderen Module vorgeben. Die Auswahl des Masters kann über einen Jumper auf der Platine realisiert werden. Die galvanische Trennung erfolgt beispielsweise über einfache Optokoppler, die dann jedoch noch eine extra Versorgungsspannung benötigen.

2.6 Arten von Verstärkern

Wie zuvor in Kapitel 2.1.2 gezeigt, soll der Zellspannungsgenerator bis zu $250\,\mathrm{mA}$ Strom an einen angeschlossenen Sensor liefern können. Da ein normaler Operationsverstärker nicht mehr ausreicht, um diesen Strom zu liefern, muss hinter dem Formfilter ein Leistungs-Verstärker geschaltet werden. Dabei gibt es drei verschiedene Arten von Verstärkern, die genauer untersucht werden sollen.

2.6.1 Diskrete Verstärker

Die erste Möglichkeit ist, den Verstärker diskret aufzubauen. Dabei wird eine Gegentakt-Verstärkerstufe aufgebaut, die mit einem Operationsverstärker nicht invertierend zurück gekoppelt wird. Dies wird in Abbildung 2.18 veranschaulicht. Es ist zusätzlich möglich, in den Rückkoppelzweig einen Spannungsteiler zu bauen, um eine Spannungsverstärkung größer 1 zu wählen.

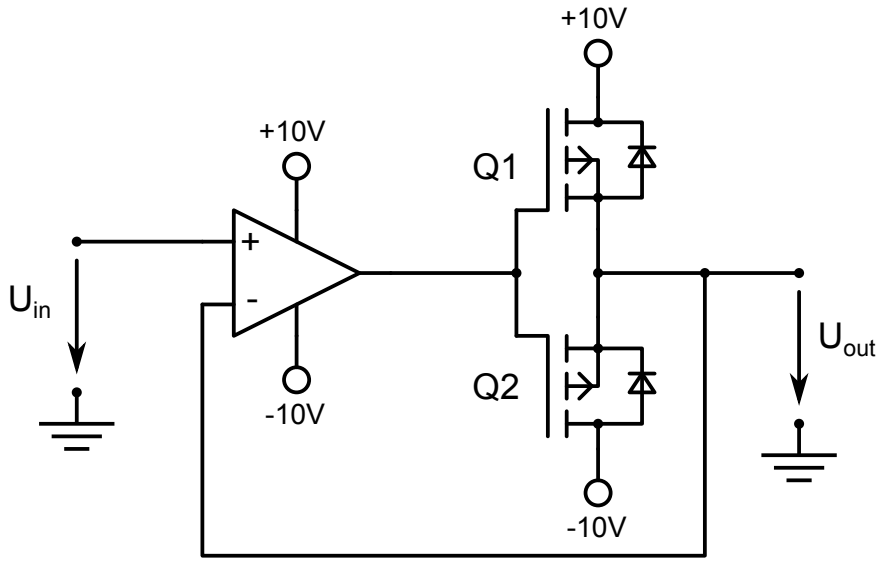


Abbildung 2.18: Schematischer Aufbau einer diskreten Verstärkerstufe

Obwohl nur positive Spannungen am Ausgang erzeugt werden, ist Q2 dennoch nötig, da er für das Entladen kapazitiver Lasten bei Verringerung der Spannung am Ausgang benötigt wird. Da die Threshold-Spannung U_{th} bei solchen Leistungs-MOSFETs für gewöhnlich über $1\,\mathrm{V}$ liegt, ist auch eine negative Spannungsversorgung, zumindest für den Operationsverstärker, unbedingt erforderlich. Der Drain-Anschluss von Q2 kann auch an die Masse angeschlossen werden, damit die negative Versorgungsspannung nur wenig Strom liefern muss. Schließt man den Ausgang der Schaltung gegen Masse kurz, so ist keine Strombegrenzung vorhanden und es werden unter Umständen Bauteile beschädigt. Es ist zwar möglich, eine Strombegrenzung einzubauen, jedoch erfordert dies eine deutlich komplexere Schaltung [26].

2.6.2 Integrierte Verstärker

Um die Komplexität des diskreten Verstärkers zu verringern, können auch integrierte Leistungs-Operationsverstärker eingesetzt werden. Diese können zwar im Allgemeinen nicht so viel Strom liefern wie eine diskret aufgebaute Gegentaktendstufe, jedoch wird für diese Anwendung nicht so viel Strom benötigt. Integrierte Verstärker, wie der OPA548 oder der OPA564 von Texas Instruments, bieten eine Reihe von Vorteilen. Hierzu gehört eine integrierte Strombegrenzung und eine Temperaturüberwachung. Außerdem sind diese beiden Verstärker kurzschlussfest, was einen großen Vorteil gerade bei Anwendungen in einem Labor bringt. Der OPA564 hat sowohl gegenüber der diskreten Variante als auch dem OPA548 einen weiteren Vorteil. Laut dem Datenblatt kommt er für Ausgangsströme bis $500\,\mathrm{mA}$

typisch auf eine untere Ausgangsspannung von $300\,\mathrm{mV}$ über der negativen Versorgungsspannung, was bedeutet, dass keine negative Versorgungsspannung benötigt wird [24]. Der integrierte Leistungs-Operationsverstärker kann wie ein gewöhnlicher Operationsverstärker als Impedanzwandler beschaltet werden, dies ist in Abbildung 2.19 dargestellt.

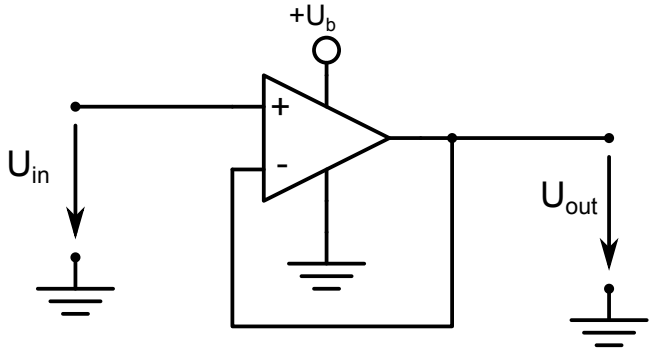


Abbildung 2.19: Leistungs-Operationsverstärker beschaltet als Impedanzwandler

2.6.3 Verstärker mit Parallelregler

Die dritte und letzte Möglichkeit, den Verstärker zu realisieren, ist die Verwendung eines Parallelreglers, auch Querregler genannt. Dabei wird ein steuerbarer Verbraucher, wie ein Transistor, parallel zur Last geschaltet. Es ist somit möglich, einen Teil des Stromflusses auf den Regler umzuleiten. Um damit eine Spannungsregelung zu realisieren, muss die Quelle über eine Strombegrenzung verfügen. Im einfachsten Fall ist dies ein Widerstand. Ein idealisierter Aufbau ist in Abbildung 2.20 gezeigt. Der Parallelregler arbeitet im Prinzip wie ein regelbarer Spannungsteiler. Der schematische Aufbau, wie er eingesetzt werden soll, wird in Abbildung 2.21 gezeigt.

Der Parallelregler hat gegenüber der diskreten Gegentaktendstufe den Vorteil, dass eine Strombegrenzung von

$$I_{max} = \frac{U_b}{R_i} \tag{2.21}$$

integriert ist. Zudem ist der Regler in der Lage, auch Spannungen von 0V zu erzeugen, da hierfür einfach der Transistor Q1 voll durchgeschaltet werden muss. Nachteilig ist, dass für die Versorgung immer eine viel größere Spannung benötigt wird als später maximal ausgegeben werden soll. Einerseits muss der Widerstand R_i möglichst klein gehalten werden, um die benötigte Versorgungsspannung U_b gering zu halten, andererseits dürfen die Verluste in R_i für den Fall der Ausgangsspannung $U_{out} = 0\,\mathrm{V}$ nicht zu groß werden. Ein guter Mittelweg ist hier R_i so zu wählen, dass bei maximalem Strom die Hälfte der Versorgungsspannung U_b an ihm abfällt. Somit muss die Versorgungsspannung doppelt so hoch

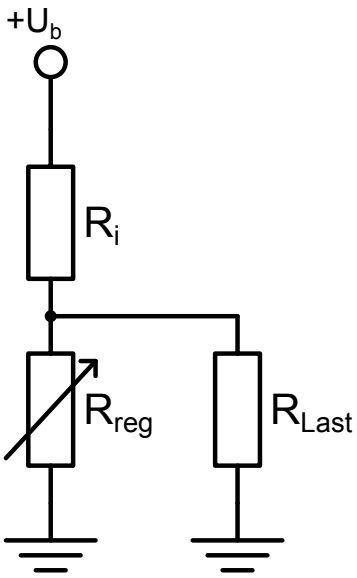


Abbildung 2.20: Idealisierter Aufbau eines Parallelreglers

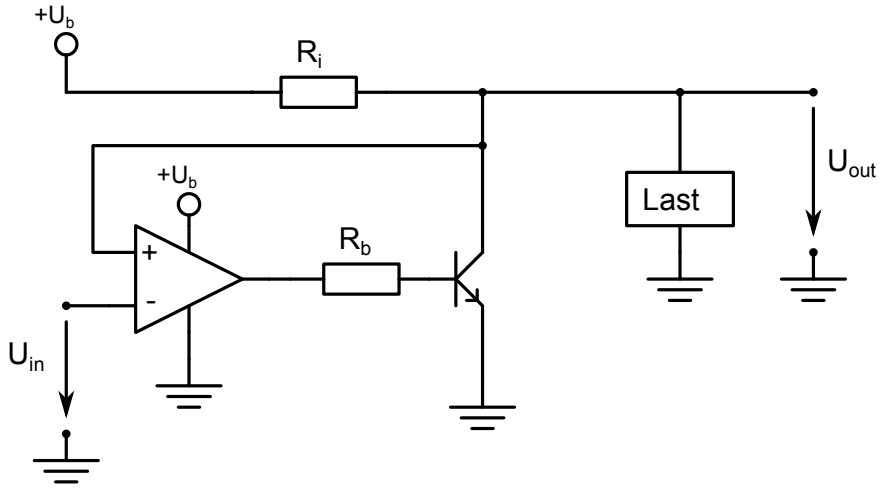


Abbildung 2.21: Schematischer Aufbau eines Parallelreglers

sein wie die maximal benötigte Ausgangsspannung. Hieraus ergibt sich ein weiterer Nachteil. Betrachtet man die Versorgungsspannung U_b und den Innenwiderstand R_i als lineare Quelle, ergibt sich die maximale Effizienz des Reglers bei Leistungsanpassung und beträgt $50\,\%$. Ein weiterer Fakt muss noch bei der Auslegung des Widerstandes R_i beachtet werden. Sollen mit dieser Quelle veränderliche Spannungen ausgegeben werden, ergibt sich bei kapazitiven Lasten ein Tiefpassfilter. Der Widerstand muss folglich ausreichend klein gewählt werden, um die Grenzfrequenz nach Kapitel 2.3 über $2{,}5\,\mathrm{kHz}$ zu halten. Bei den Sensoren der Klasse 2 ist am Eingang ein Kondensator von $47\,\mu\mathrm{F}$ angeschlossen. Mit einer maximalen Versorgungsspannung von $5{,}5\,\mathrm{V}$ ergeben sich folgende Werte:

$$U_b = 2 \cdot U_{out,max} = 2 \cdot 5{,}5\,\mathrm{V} = 11\,\mathrm{V} \tag{2.22}$$

$$R_{i,max} = \frac{1}{2 \cdot \pi \cdot f_g \cdot C} = \frac{1}{2 \cdot \pi \cdot 2{,}5\,\mathrm{kHz} \cdot 47\,\mu\mathrm{F}} \approx 1{,}35\,\Omega \tag{2.23}$$

$$P_{R_i,max} = \frac{{U_b}^2}{R_{i,max}} = \frac{(11\,\mathrm{V})^2}{1{,}35\,\Omega} \approx 89{,}6\,\mathrm{W} \tag{2.24}$$

Die maximale Verlustleistung in R_i ist für ein einziges Modul nicht akzeptabel. Somit muss der Widerstand deutlich größer gewählt werden, was die Grenzfrequenz herabsetzt, wodurch es nicht mehr möglich ist, den Startvorgang eines Autos zu simulieren. Bei den Berechnungen ist außerdem zu bedenken, dass der Regler in der Lage ist, die Spannungen bei fallenden Flanken durchaus schnell genug zu erzeugen, jedoch nicht bei steigenden Flanken, da hier über R_i geladen werden muss. Somit ist der Parallelregler für die Wiedergabe von transienten Spannungen nicht tauglich, er könnte sich aber für die Kalibrierung der Sensoren eigenen.

Im Kapitel 3 sollen daher die beiden vorgestellten integrierten Verstärker und der Parallelregler genauer untersucht werden. Der diskrete Verstärker soll nicht weiter untersucht werden, da er zu viele Nachteile gegenüber den integrierten Leistungs-Operationsverstärkern hat und praktisch keine Vorteile.

2.7 Speicherung von Spannungssequenzen

Wie zuvor in Kapitel 2.5.1 gezeigt, muss auf den Generatoren ausreichend Speicher vorhanden sein, um Samples für bis zu 60 Sekunden zwischen zu speichern. Bei $20\,\mathrm{kSPS}$ und einer Samplegröße von 2 Bytes ergibt sich ein Speichervolumen von mindestens:

$$B_{min} = 20\,\mathrm{kSPS} \cdot 2\,\mathrm{Byte} \cdot 60\,\mathrm{s} = 2{,}4\,\mathrm{MByte} \tag{2.25}$$

Mit einer Kompression kann die benötigte Größe nochmal reduziert werden, jedoch wird dann auch die Verarbeitung der Daten aufwendiger. Dennoch wird die Datenmenge dadurch

nicht klein genug, um sie im RAM des Stellaris Mikrocontroller zwischen zu speichern, welcher hierfür $96\,\mathrm{kByte}$ bietet. Es wird somit ein zusätzlicher externer Speicher benötigt. Der Stellaris LM3S9B92 besitzt einen internen Speichercontroller, welcher dazu dient, externe Speicher wie beispielsweise SDRAM anzusprechen. Dies bietet eine einfache Möglichkeit, den benötigten Speicher anzuschließen und auch aus der Software anzusprechen, denn er wird direkt in den bestehenden Adressraum integriert. Somit kann man den Speicher genau wie den internen RAM ansprechen. Da jedoch das Evalboard verwendet werden soll, ist dies keine Lösung, denn es sind nicht alle benötigten Pins des Controllers auf die Stiftleisten geführt. Weiterhin ist fraglich, ob ein Anschluss von SDRAM über die Stiftleisten möglich ist, da hier die Leitungen relativ lang sind und keine Impedanzkontrolle durchgeführt werden kann. Eine Alternative ist die Verwendung von einem externen Flash-Speicher. Diese Speicher gibt es in kleinen Gehäusen mit ausreichend Speicherkapazität. Die Ansteuerung erfolgt für gewöhnlich über SPI, I^2C oder ein paralleles Interface. Bei Verwendung von SPI und I^2C ist es daher ohne Probleme möglich, auch mehrere Speicher an einen Bus anzuschließen. Daher ist der externe Flash die einfachste Möglichkeit, ausreichend Speicher zur Verfügung zu stellen und ist somit die Wahl für den Zellspannungsgenerator.

2.8 Strommessung

Ein weiteres Feature, das zumindest in der Hardware realisiert werden soll, ist die Messung des Stromverbrauchs. Dabei soll der Stromverbrauch während der Transientenwiedergabe hochfrequent aufgenommen werden, ohne dass die Messung die Ausgangsspannung beeinflusst. Eine Strommessung über einen Hall-Effekt basierten Chip kommt nicht in Frage, da diese eine zu geringe Auflösung haben. Die Messung über einen Shunt-Widerstand ist deutlich genauer, jedoch erzeugt der eingesetzte Widerstand einen Spannungsabfall, welcher unbedingt vermieden oder sehr gering gehalten werden muss, da sonst die Ausgangsspannung abhängig vom Strom einbricht. Um den Spannungsabfall über den Widerstand vollständig zu kompensieren, kann dieser beispielsweise hinter den Verstärker gesetzt werden und anschließend erst die Rückführung des Verstärkers abgenommen werden. Somit regelt der Verstärker die entstehenden Fehler durch den Widerstand aus.

2.9 Zusammenfassung des Konzepts

Alle wesentlichen Punkte des Konzeptes sind nun bekannt und sollen folgend nochmals zusammengefasst werden. Jeder einzelne Zellspannungsgenerator soll in der Lage sein, eine Auflösung und eine Genauigkeit von unter $1{,}22\,\mathrm{mV}$ zu erreichen, eine Ausgangsspannung von $0{,}5\,\mathrm{V}$ bis $5{,}5\,\mathrm{V}$ und einem maximalen Strom von $250\,\mathrm{mA}$ zu liefern. Die Generatoren

werden untereinander mit Ethernet verbunden, worüber auch der steuernde PC angeschlossen ist. Hierfür kommt das EKS-LM3S9B92 Evalboard von Texas Instruments zum Einsatz, auf dem der Stellaris LM3S9B92 ARM Cortex M3 Mikrocontroller verbaut ist. Die Spannungswiedergabe soll mit mindestens $20\,\mathrm{kSPS}$ möglich sein, wobei eine analoge Bandbreite von etwa $2{,}5\,\mathrm{kHz}$ benötigt wird. Außerdem soll die Hardware erlauben, den aktuellen Stromverbrauch des angeschlossenen Sensors zu messen, wobei auch hier die Samplerate bei $20\,\mathrm{kSPS}$ liegen soll. Die Synchronisation der einzelnen Module geschieht über eine einzelne galvanisch getrennte Leitung, die von einem der Module getrieben wird. In Abbildung 2.22 ist hierfür nochmals das vollständige Blockschaltbild dargestellt.

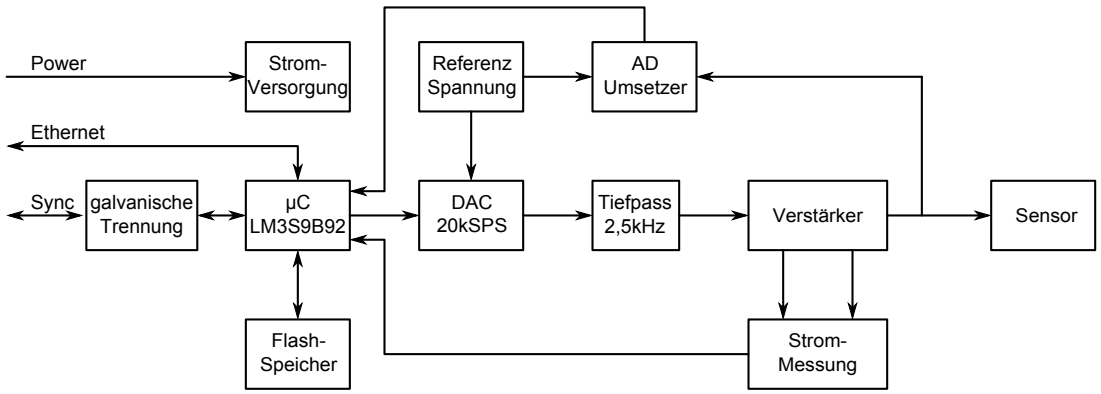


Abbildung 2.22: Vollständiges Blockschaltbild für das finale Konzept

3 Voruntersuchungen

Um die Auswahl des Verstärkers zu treffen, sollen die drei zuvor in Kapitel 2.6 eingeführten Verstärker genauer untersucht werden. Hierzu gehören die beiden integrierten Verstärker OPA548 und OPA564 von Texas Instruments und der Parallelregler. Alle Verstärker werden zunächst mit Simulationen untersucht, anschließend sollen Evaluations-Boards erstellt werden, die das Verhalten in der Realität bestätigen oder widerlegen sollen. Spezielles Augenmerk liegt hierbei auf dem Verhalten der Verstärker im Falle einer Laständerung. Dieses Verhalten ist besonders wichtig bei der Kalibrierung der Sensoren, da hier die Spannung trotz Laständerung möglichst konstant bleiben soll. Dabei ist nicht nur die stationäre Spannung in Abhängigkeit der Last von Interesse, sondern auch ein eventuell auftretendes Überschwingen bei der Laständerung selbst. Offset- und Verstärkungsfehler interessieren bei diesen Voruntersuchungen erst einmal nicht, da diese Fehler später noch mit einer Kalibrierung korrigiert werden können.

3.1 Analyse des Parallelreglers

Der in der Abbildung 2.21 zuvor gezeigte Parallelregler hat sich für die Transientenwiedergabe nicht als tauglich erwiesen, jedoch soll er nochmals genauer untersucht werden, da er für die Wiedergabe von stationären Spannungen generell geeignet ist.

3.1.1 Simulation

Der Parallelregler soll in einer Simulation zeigen, wie er sich bei Lastenänderungen am Ausgang verhält. Die Simulation wird mit PSpice durchgeführt, dabei soll der Regler eine konstante Spannung von $2\,\mathrm{V}$ ausgeben. Der Regler wird konstant mit $100\,\Omega$ belastet, zusätzlich werden periodisch $47\,\Omega$ parallel zugeschaltet. Somit fließt ein Ruhestrom von $20\,\mathrm{mA}$ und zusätzlich ein pulsartiger Strom von ca. $43\,\mathrm{mA}$. Der Aufbau der Schaltung, wie sie simuliert wird, ist in Abbildung 3.1 gezeigt. Betrachtet man nun die in Abbildung 3.2 und 3.3 dargestellte Ausgangsspannung, ist schnell klar, dass der Regler von der Simulation her relativ schlechte Eigenschaften hat. In Abbildung 3.2 ist das Verhalten des Reglers beim Zuschalten der $47\,\Omega$-Last zu sehen, der Ausgang bricht hier zunächst um über $350\,\mathrm{mV}$ ein und schwingt sich anschließend innerhalb von ca. $4\,\mu\mathrm{s}$ wieder ein. Das in Abbildung 3.3

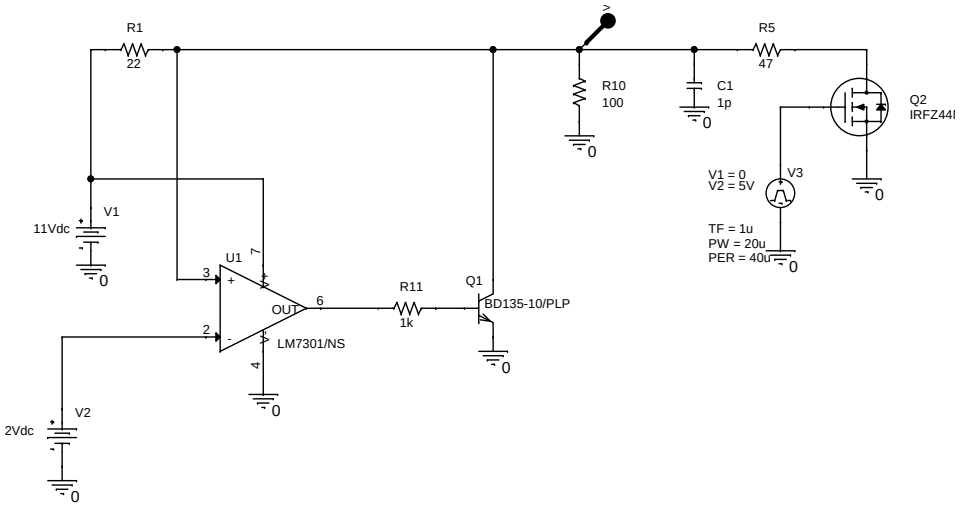


Abbildung 3.1: Schaltplan für die Simulation des Parallelreglers

dargestellte Abschalten der Last führt zu einem ähnlichen Effekt, die erste Spitze beträgt zwar nur ca. $200\,\mathrm{mV}$, jedoch dauert es über $10\,\mu\mathrm{s}$ bis der Regler wieder in einem stationären Zustand ist. Das unterschiedliche Verhalten des Reglers bei Zu- und Abschalten der Last liegt vermutlich an der unterschiedlichen Art der Quelle bzw. Senke. Für die Quelle wird ein Widerstand verwendet, als Senke ein gesteuerter Transistor.

Ein weiteres Problem, das sich bereits in der Simulation zeigt, ist die Belastung mit einer Kapazität. Auch hierfür wird eine Simulation wie in Abbildung 3.4 durchgeführt, wobei das System instabil wird und anfängt zu schwingen. Das Ausgangssignal ist in Abbildung 3.5 dargestellt. Das Schwingen wird durch die zusätzliche kapazitive Last verursacht. Diese erzeugt eine Phasendrehung des gesamten Systems, wird diese zu groß, kommt es bei rückgekoppelten Systemen zur Mitkopplung und das System wird instabil.

3.1.2 Hardwareentwurf und Aufbau

Zusätzlich zu der Simulation wird auch noch eine Version in Hardware aufgebaut. Der Schaltplan und das Layout sind in Anhang B.1.1 und B.2.1 zu finden. Der Schaltplan hat als Parallellast einen MOSFET verbaut, welcher bei den Tests jedoch durch einen Bipolartransistor des Typs BD649 ersetzt wird. Zudem wird, wie in der Simulation, ein Widerstand in Serie vor die Basis des Transistors gelötet.

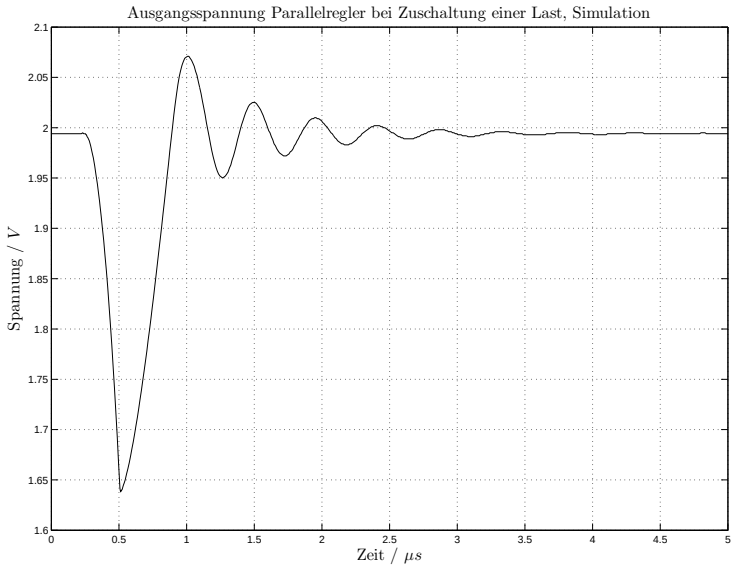


Abbildung 3.2: Ausgangsspannung des simulierten Parallelreglers beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

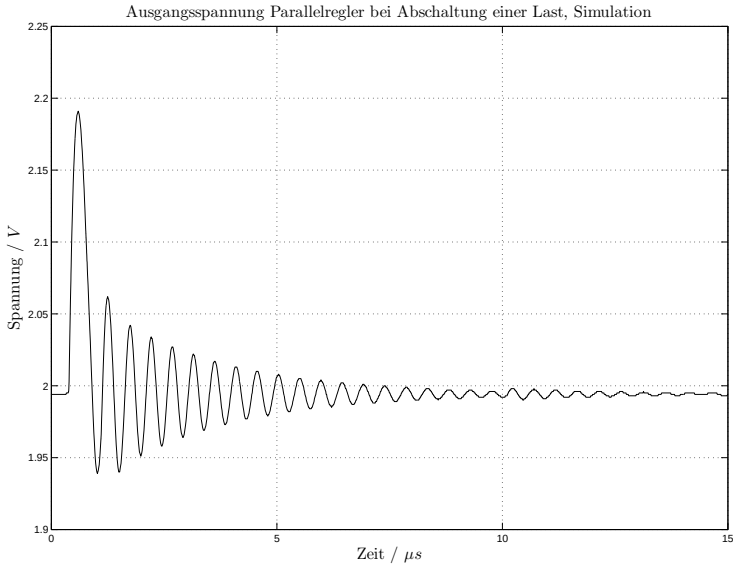


Abbildung 3.3: Ausgangsspannung des simulierten Parallelreglers beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

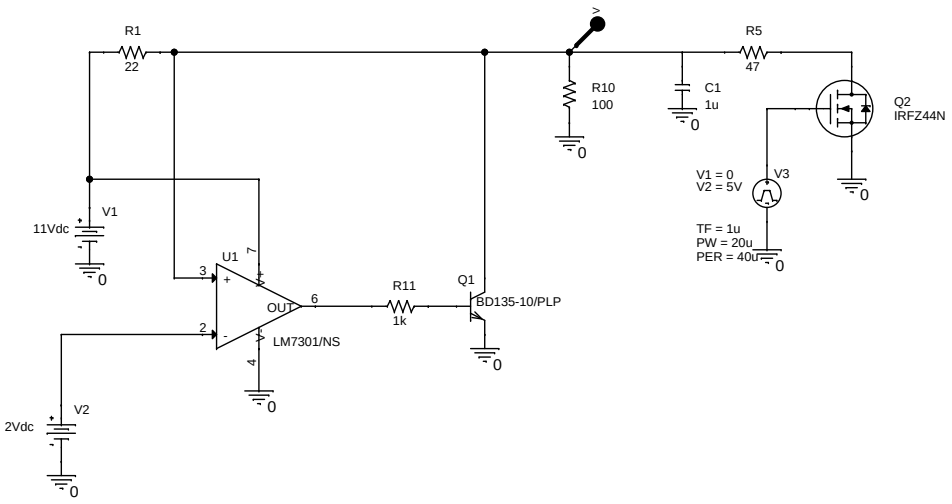


Abbildung 3.4: Schaltplan für die Simulation des Parallelreglers

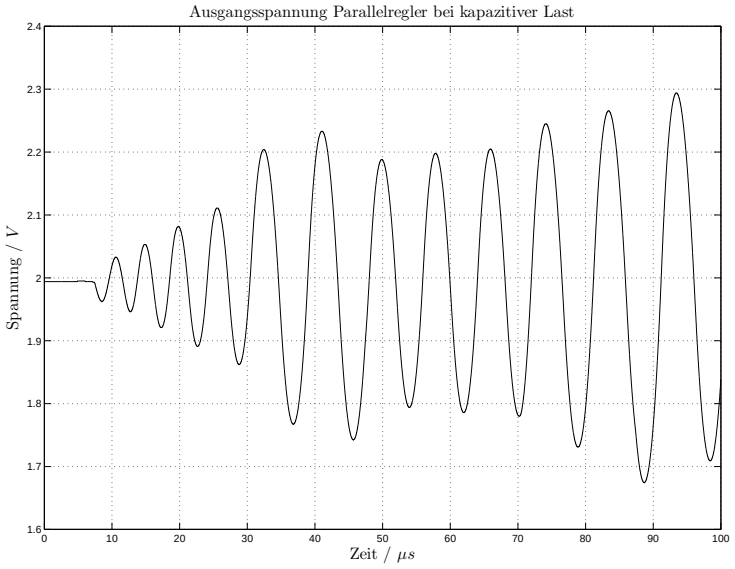


Abbildung 3.5: Ausgangsspannung des simulierten Parallelreglers mit zusätzlicher kapazitiver Last von $1\ \mu\mathrm{F}$

Wie die Simulation bereits vermuten lässt, ist das System nicht stabil. Selbst ohne kapazitive Last schwingt der Regler, wie Abbildung 3.6 zeigt. Es werden daher keine Weiteren Messungen durchgeführt.

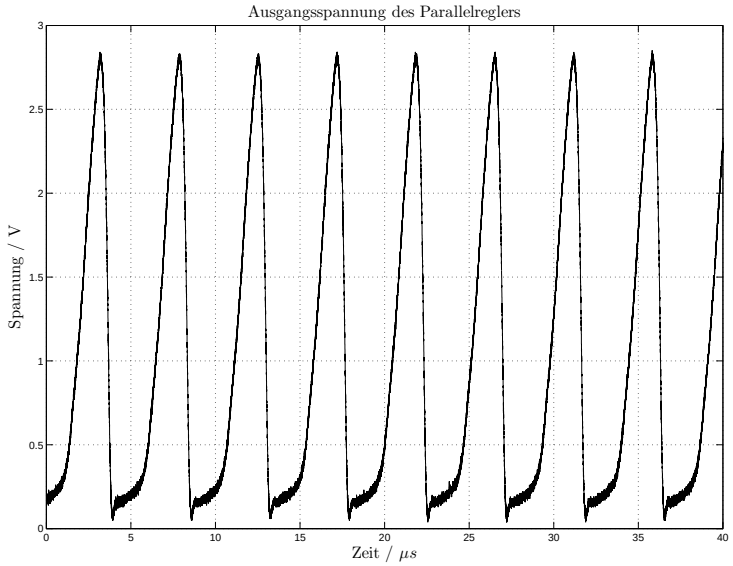


Abbildung 3.6: Ausgangsspannung des Parallelreglers ohne kapazitive Last, Messung an der Hardware

3.1.3 Auswertung der Simulationen und Messungen

Der Parallelregler hat eine Reihe von Nachteilen. Der Grundaufbau erlaubt keine schnellen Veränderungen der Ausgangsspannungen, aufgrund der begrenzen Ladegeschwindigkeit einer kapazitiven Last. Außerdem scheint der Regler in der Form, wie er hier aufgebaut ist, instabil zu sein, was jedoch kein allgemeines Problem von Parallelreglern ist. Daneben existieren noch einige weitere kleinere Nachteile, wie der schlechte Wirkungsgrad des Reglers. Es wird daher die Designentscheidung getroffen, das Konzept des Parallelreglers nicht weiter zu verfolgen, um Reglern Vorrang einzuräumen, die vielversprechender sind und auch für die Wiedergabe von Spannungssequenzen eingesetzt werden können.

3.2 Analyse integrierter Verstärker

Deutlich bessere Chancen auf ein stabiles Verhalten haben die beiden integrierten Leistungs-Operationsverstärker OPA548 und OPA564 von Texas Instruments. Auch hierfür sollen zu-

nächst Simulationen durchgeführt werden und anschließend jeweils eine Testplatine entworfen und aufgebaut werden, um damit die Simulationen zu bestätigen.

3.2.1 Simulationen

Die Abbildungen 3.7 und 3.8 zeigen die Basisschaltung der beiden Simulationen. Auch hier werden die Verstärker mit jeweils $100\,\Omega$ Grundlast betrieben und eine Last von $47\,\Omega$ hinzugeschaltet. Bei der Schaltung des OPA564 ist noch eine Besonderheit. In der Rückführung ist parallel zu dem Widerstand R6 noch ein Kondensator von $1\,\mathrm{nF}$ geschaltet. Dies ist für die Kompensation von kapazitiven Lasten am Ausgang, da der Regler sonst zumindest in der Simulation schnell instabil wird. Beim OPA548 besteht dieses Problem nicht.

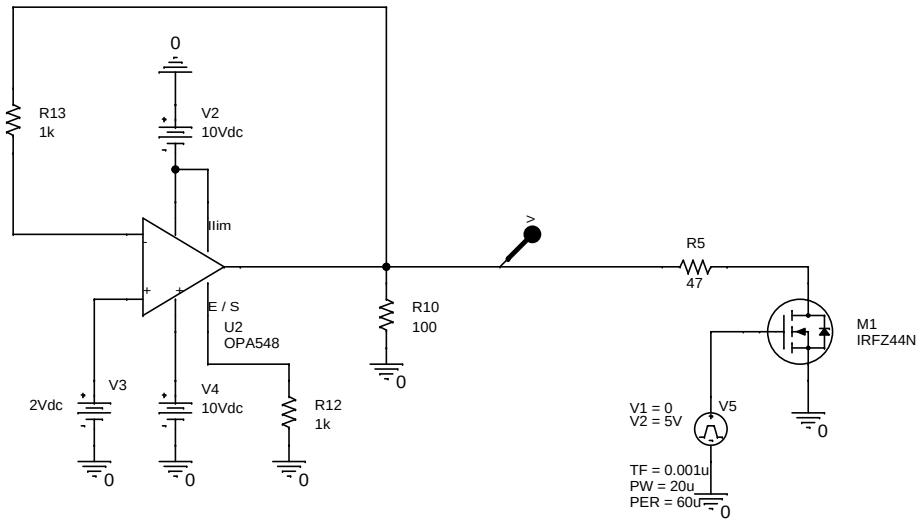


Abbildung 3.7: Schaltplan für die Simulation des integrierten Verstärkers OPA548

Auch hierbei wird wieder zwischen Zuschalten und Abschalten der Last unterschieden, da hier durchaus unterschiedliche Ergebnisse zu erwarten sind. Die Abbildungen 3.9 und 3.10 zeigen den Verlauf der Ausgangsspannung für das Zuschalten der Last jeweils für den OPA548 und OPA564. Der Verlauf der beiden Kurven ist sehr ähnlich, der OPA564 ist jedoch in der Lage, etwas schneller in die Ruhelage zurückzukehren und die maximale Amplitudenabweichung ist nur etwa halb so groß im Vergleich zum OPA548. Beim Abschalten der Last ist die Einschwingphase bei beiden Verstärkern, im Vergleich zum Zuschalten der

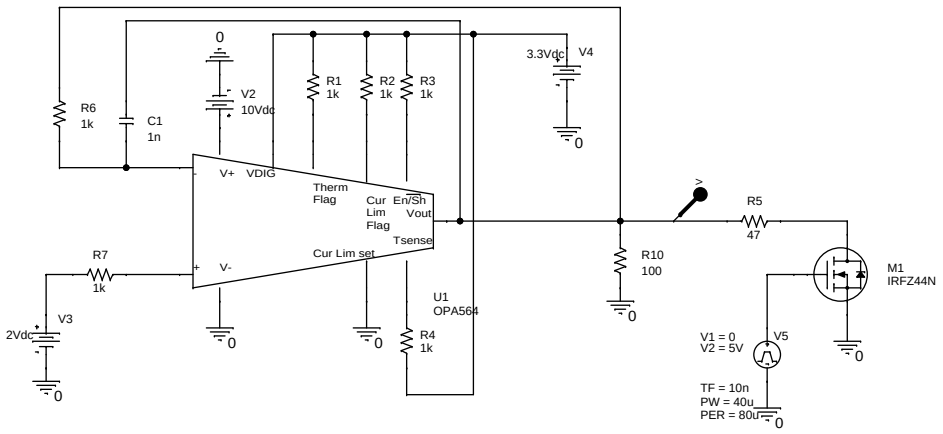


Abbildung 3.8: Schaltplan für die Simulation des integrierten Verstärkers OPA564

Last, deutlich länger. Die Amplitudenabweichungen sind jedoch etwas geringer. Wie zuvor ist auch hier der OPA564 in Bezug auf die Einschwingzeit und die maximale Abweichung besser.

Um das Überschwingen bei Laständerung noch weiter zu verringern, kann eine Kapazität an den Ausgang der Verstärker geschaltet werden. Bei dem OPA564 ist am Ausgang zusätzlich ein Serienwiderstand angebracht, da PSpice sonst Konvergenzprobleme hat. Die Kondensatoren glätten die Ausgangsspannung deutlich, wie in den Abbildungen 3.15 und 3.16 für den Fall der zuschaltenden Last zu erkennen ist. Auch hier schneidet der OPA564 wieder deutlich besser ab als der OPA548. Der Einschwingvorgang dauert zwar minimal länger als beim OPA548, jedoch liegt die maximale Abweichung von der Ruhelage gerade einmal bei ca. $6\,\mathrm{mV}$, wärend sie beim OPA548 ca. $15\,\mathrm{mV}$ beträgt. Das Gleiche gilt nach den Abbildungen 3.17 und 3.18 auch für das Abschalten der Last, auch hier schwingt der OPA564 etwa $2\,\mathrm{mV}$ über.

Ein weiterer wichtiger Punkt ist die Abweichung im eingeschwungen Zustand. Wobei nicht die absolute Abweichung zu den $2\,\mathrm{V}$ von Interesse sind, sondern, ob der Regler mit und ohne Last die gleiche Spannung am Ausgang erzeugt. Also, ob die Ausgangsspannung nach dem Einschwingen unabhängig von der Last ist. Bei beiden Reglern ist dies zumindest in der Simulation gegeben, was eine Grundvoraussetzung für die genaue Arbeitsweise der Reg-

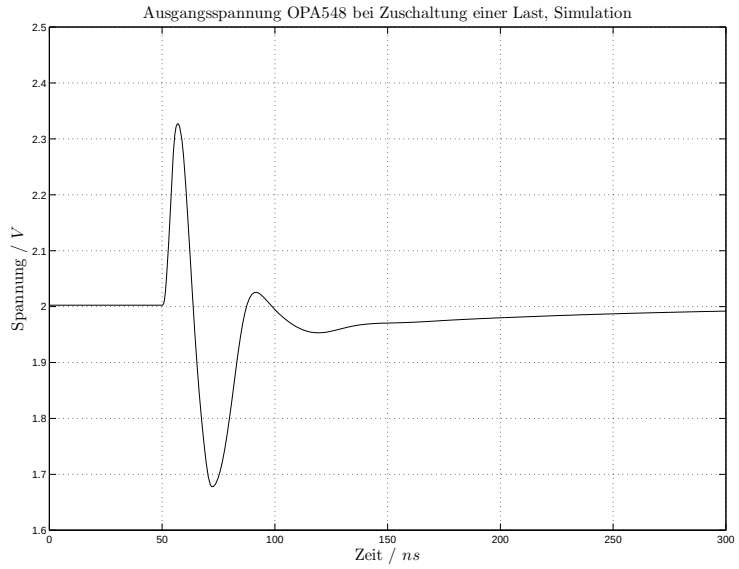


Abbildung 3.9: Ausgangsspannung des simulierten OPA548 beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

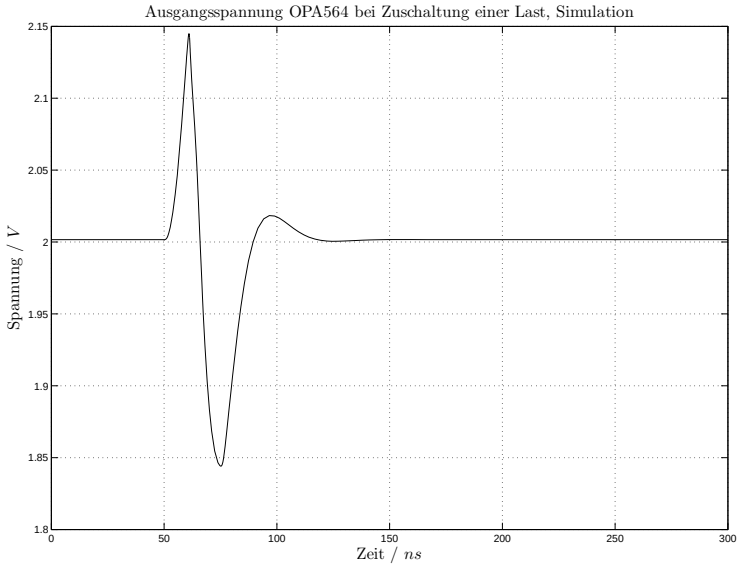


Abbildung 3.10: Ausgangsspannung des simulierten OPA564 beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

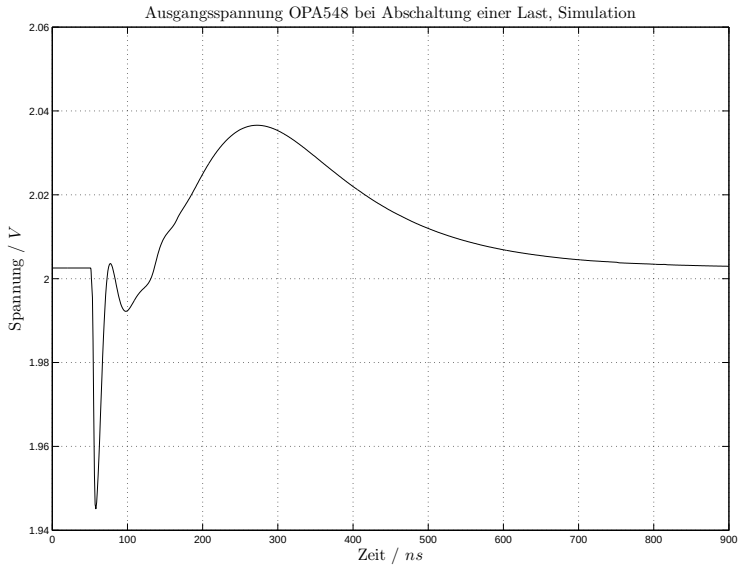


Abbildung 3.11: Ausgangsspannung des simulierten OPA548 beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

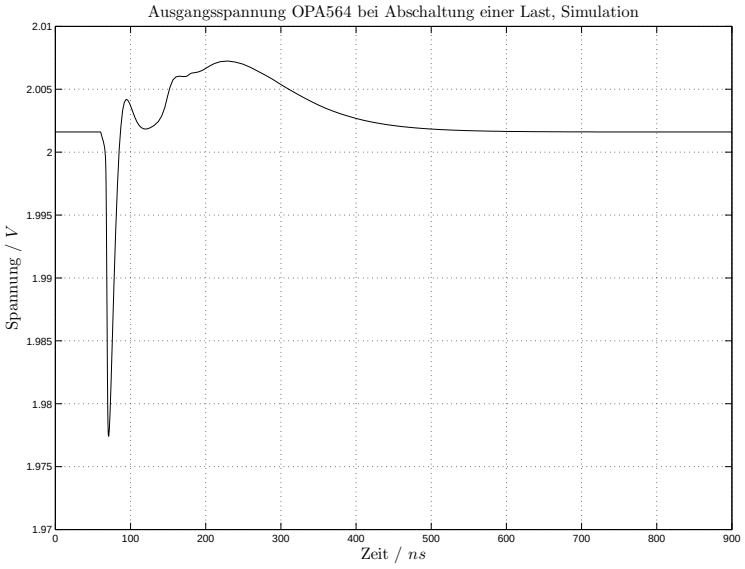


Abbildung 3.12: Ausgangsspannung des simulierten OPA564 beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

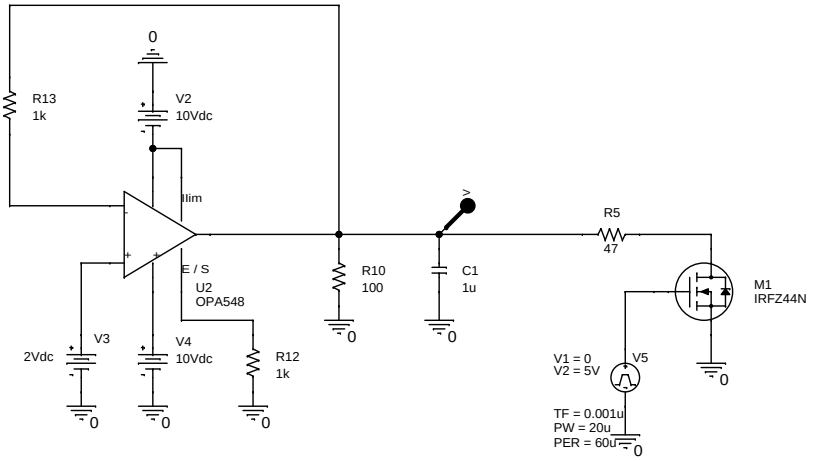


Abbildung 3.13: Schaltplan für die Simulation des integrierten Verstärkers OPA548 mit kapazitiver Last

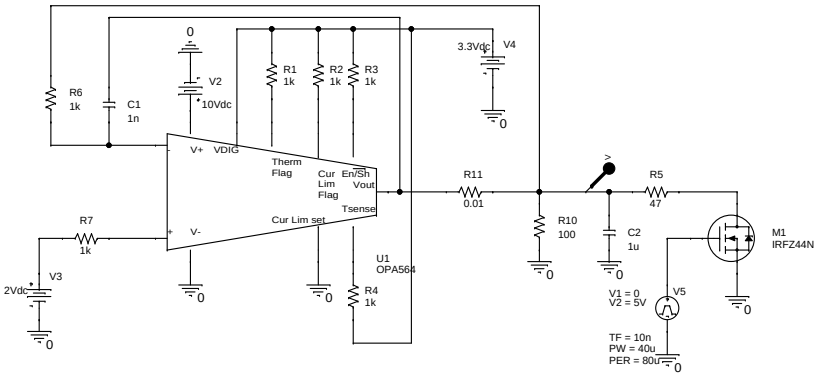


Abbildung 3.14: Schaltplan für die Simulation des integrierten Verstärkers OPA564 mit kapazitiver Last

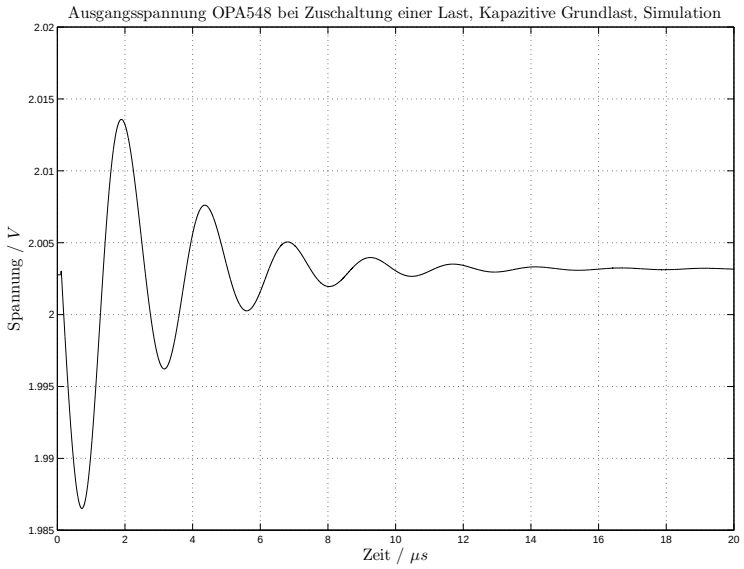


Abbildung 3.15: Ausgangsspannung des simulierten OPA548 beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $1\,\mu$F

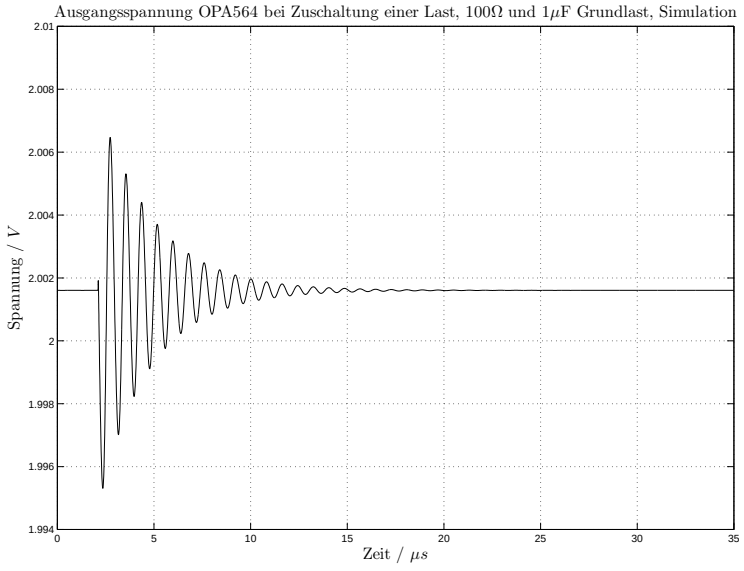


Abbildung 3.16: Ausgangsspannung des simulierten OPA564 beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $10\,\mu$F

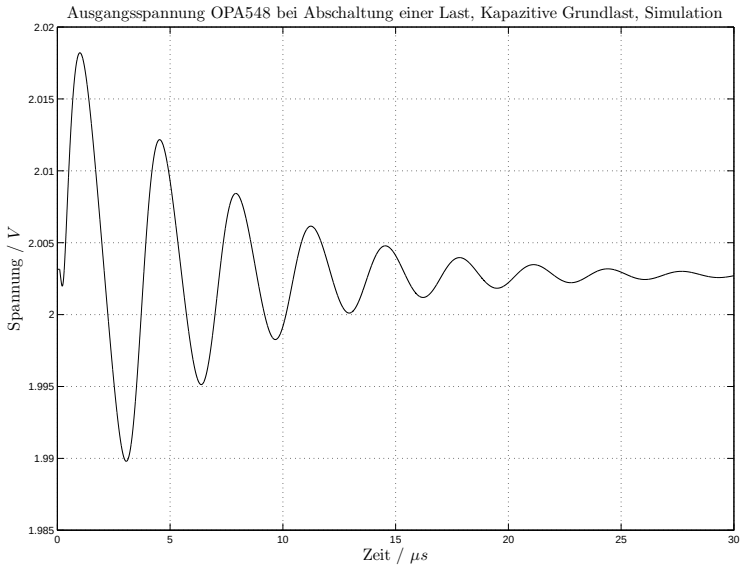


Abbildung 3.17: Ausgangsspannung des simulierten OPA548 beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $1\,\mu$F

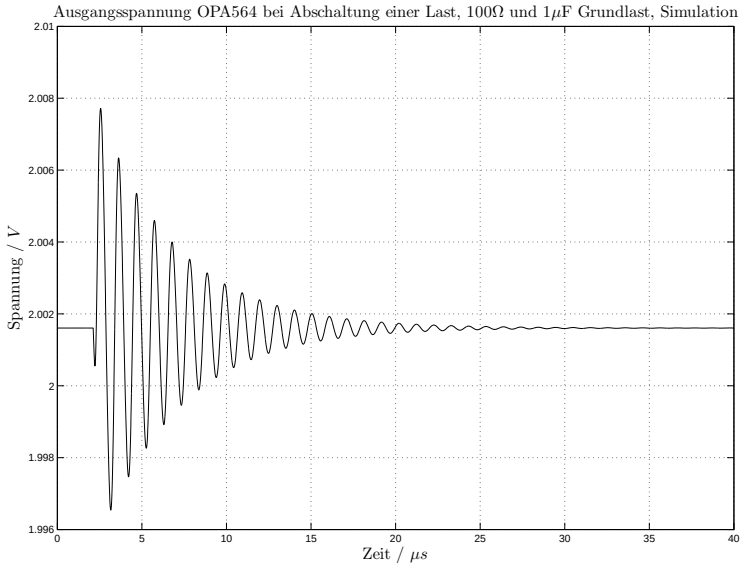


Abbildung 3.18: Ausgangsspannung des simulierten OPA564 beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $10\,\mu$F

ler ist. Um diese Ergebnisse auch in der Realität verifizieren zu können, werden für beide integrierten Verstärker Testplatinen erstellt.

3.2.2 Hardwareentwurf und Aufbau

Im Anhang B.1.2 befindet sich der Schaltplan für die Testplatine des OPA548, hier wird im Wesentlichen die Schaltung aus der Simulation übernommen. Hinzu kommen am Ausgang einige Kondensatoren, die mittels Jumper zuschaltbar sind, um das Verhalten bei verschiedenen kapazitiven Lasten analysieren zu können. Anhang B.1.3 enthält den Schaltplan für den OPA564, hierbei wurde ebenfalls primär der Schaltplan aus der Simulation übernommen, es kommen jedoch noch einige Komponenten hinzu. Hierzu zählen unter anderem zwei verschiedene Anschlüsse des Sensors, die hier getestet werden sollen. Zum einen ist dies die Verbindung über BNC-Kabel, zum anderen der Anschluss der Sensoren über Kabel mit einem Bananenstecker. Da die Kabel zu den Sensoren unter Umständen bis zu einem Meter lang sein können und in diesem Fall einen Widerstand von mehreren Milliohm haben, können bei Strömen von bis zu $250\,\mathrm{mA}$ einige Millivolt über die Kabel verloren gehen. Um diesen Effekt zu kompensieren ist eine differentielle Rückführung der Spannung direkt am Sensor sinnvoll, um den Spannungsabfall über die Kabel zu kompensieren. Hierbei muss nicht nur die positive Versorgungsspannung zurückgeführt werden, sondern auch die Masse des Sensors. In Anhang B.2.2 und B.2.3 sind jeweils die Layouts der beiden Testplatinen zu finden.

3.2.3 Messungen an der Hardware

Um die beiden Verstärker miteinander vergleichen zu können, werden, wie in der Simulation, beide mit einer Grundlast von $100\,\Omega$ belastet. Zusätzlich wird eine Last von $47\,\Omega$ periodisch an- und abgeschaltet. Die Aufzeichnung der Ausgangsspannung erfolgt dabei mit einem Tektronix MSO 3034 Oszilloskop. Um eine höhere Auflösung zu erhalten, werden die Eingänge des Oszilloskops AC-gekoppelt. Die Simulation mit einem zusätzlichen Kondensator von $1\,\mu\mathrm{F}$ am Ausgang (Abbildung 3.15) ist der Messung (Abbildung 3.20) sehr ähnlich. Die Simulation schneidet jedoch bezüglich der Einschwingdauer und der maximalen Amplitudenabweichung etwas besser ab.

Abbildung 3.19 zeigt den OPA548 mit einem maximalen Überschwingen von ca. $55\,\mathrm{mV}$. Im Vergleich zu der dazugehörigen Simulation aus Bild 3.9 ist das Verhalten in der Realität deutlich anders. Die erste Spitze schwingt in der Simulation etwa $330\,\mathrm{mV}$ über. Zudem unterscheidet sich das Abklingverhalten zwischen Simulation und der realen Messung, wobei es in der Simulation deutlich besser ist, da das Signal sehr schnell wieder eingeschwungen ist.

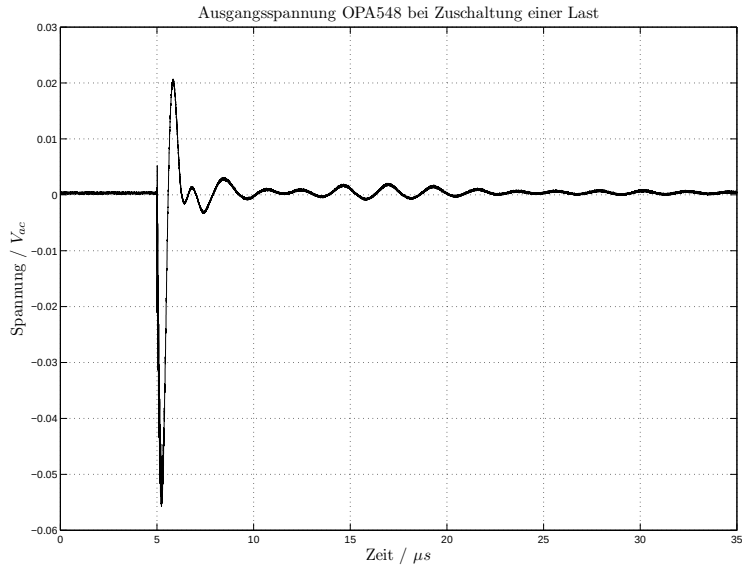


Abbildung 3.19: Ausgangsspannung des OPA548-Testboards beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

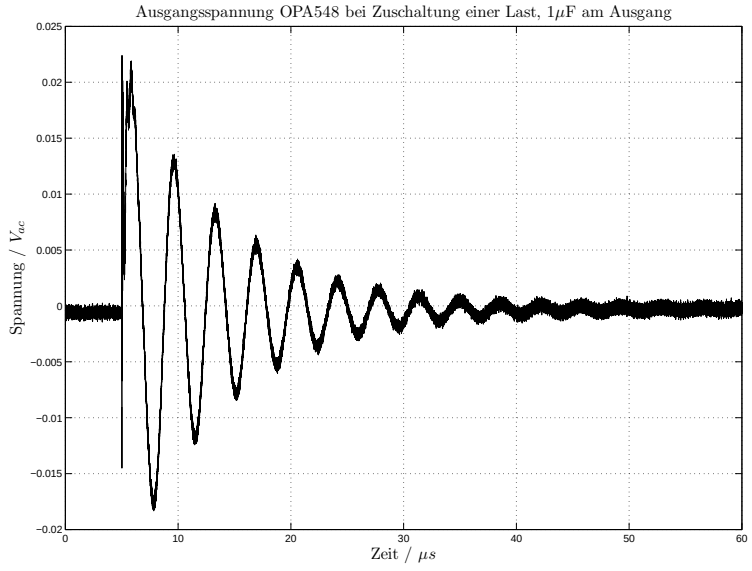


Abbildung 3.20: Ausgangsspannung des OPA548-Testboards beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $1\,\mu$F

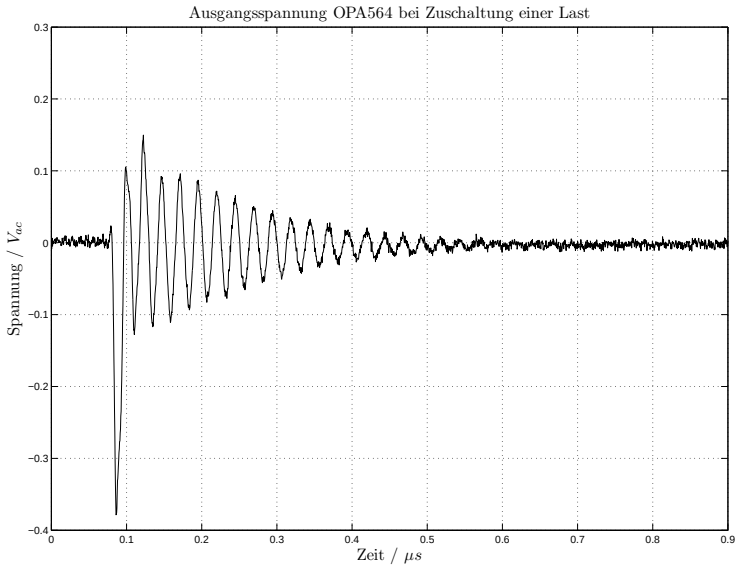


Abbildung 3.21: Ausgangsspannung des OPA564-Testboards beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$

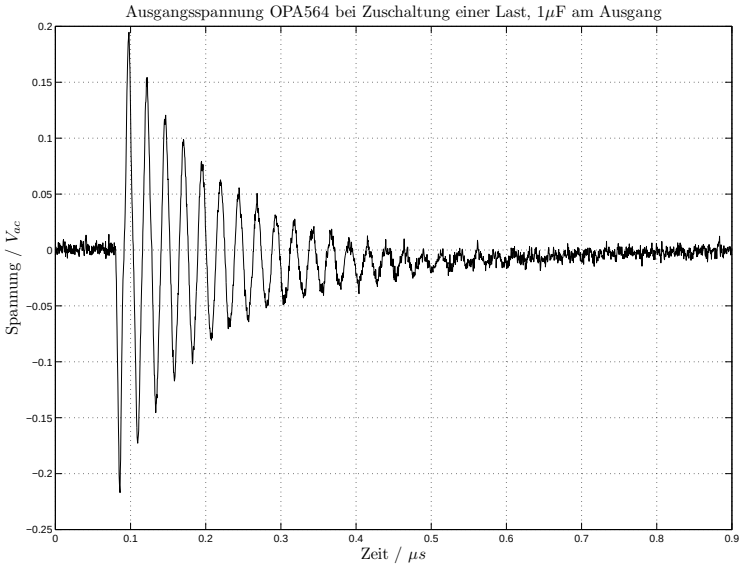


Abbildung 3.22: Ausgangsspannung des OPA564-Testboards beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $1\,\mu$F

Beim OPA564 hingegen sind kaum Unterschiede zwischen der Simulation aus Abbildung 3.10 und der Realität auszumachen. Bei beiden liegt der maximale Überschwinger etwa bei $18\,\mathrm{mV}$. Das Einschwingen dauert in der Simulation $15\,\mu\mathrm{s}$, in der Realtiät etwa $40\,\mu\mathrm{s}$. Das Einzige was hierbei und auch schon zuvor beim OPA548 auffällt, ist, dass das erste Überschwingen in der Praxis eine umgekehrte Polarität hat im Vergleich zur Simulation. Woran dies genau liegt, ist unklar.

Bei den Messungen des OPA564 lassen sich ebenfalls gravierende Unterschiede zu den Simulationen feststellen. Vergleicht man die Simulation ohne Kondensator am Ausgang aus Abbildung 3.9 mit dem gemessenen Verlauf aus Abbildung 3.21, wird deutlich, dass zwar die maximalen Abweichungen auf Grund des Überschwinges ähnlich sind, der Verstärker in der Realität jedoch deutlich länger braucht bis er wieder eingeschwungen ist. Betrachtet man das Verhalten des Verstärkers in der Realität mit einem zusätzlichen Kondensator von $1\,\mu\mathrm{F}$ am Ausgang (Abbildung 3.22) unterscheidet sich dies kaum zu dem Verhalten ohne Kondensator mit der Ausnahme, dass die erste Spitze des Überschwingens deutlich gedämpft ist.

Bei der Messungen mit dem OPA564 zeigt sich ein weiterer Effekt, welcher der finalen Implementierung zu Gute kommen kann. Bei genauerer Untersuchung der Verstärkerplatine kommt heraus, dass die Amplitude des Überschwingens sich proportional zu der Amplitude des Generatorsignals zum Schalten des Last-MOSFETs verhält. Verringert man am Generator die Steilheit dieser Flanken, so kann das Überschwingen nochmals deutlich reduziert werden, wie in Abbildung 3.23 zu sehen ist. Interessant ist auch, dass bereits ein deutliches Überschwingen entsteht, wenn der Verstärker gar nicht mit Strom versorgt wird. Es handelt sich somit bei den Überschwingern vermutlich um unerwünschte Messfehler auf Grund von Reflexionen oder anderen parasitären Effekten.

Neben dem Verhalten der Verstärker soll auch noch das Verhalten bei verschiedenen kapazitiven Lasten untersucht werden. Hierfür sind auf der Testplatine des OPA548 verschiedene, mit Jumper zuschaltbare, Kondensatoren vorhanden. Um das Verhalten zu vergleichen, werden Kondensatoren der Größen $1\mu F$, $10\mu F$, $100\mu F$ und $1000\mu F$ am Ausgang dazugeschaltet und das Verhalten bei einer Laständerung beobachtet. Abbildung 3.24 zeigt die verschiedenen AC-gekoppelten Ausgangsspannungen beim Abschalten der Last von $47\,\Omega$. Dabei ist deutlich zu sehen, dass größere Kapazitäten am Ausgang ein besseres Verhalten bei Laständerungen zeigen. Für die Ausgabe von transieten Spannungen ist jedoch zu bedenken, dass der benötigte Strom zum Umladen der Kondensatoren nicht zu hoch werden darf. Nimmt man an, dass ein Sinus mit der maximalen Frequenz von $2{,}5\,\mathrm{kHz}$ bei einer maximalen Amplitude von $A_{max} = 2{,}5\,\mathrm{V}$ ausgegeben werden soll, so ergibt sich die maximale Flankensteilheit r_{max} wie folgt:

$$U(t) = A_{max} \cdot sin\,(2 \cdot \pi \cdot f_{max} \cdot t) \tag{3.1}$$

$$r(t) = \frac{dU(t)}{dt} = A_{max} \cdot sin\,(2 \cdot \pi \cdot f_{max} \cdot t) \cdot \frac{d}{dt} \tag{3.2}$$

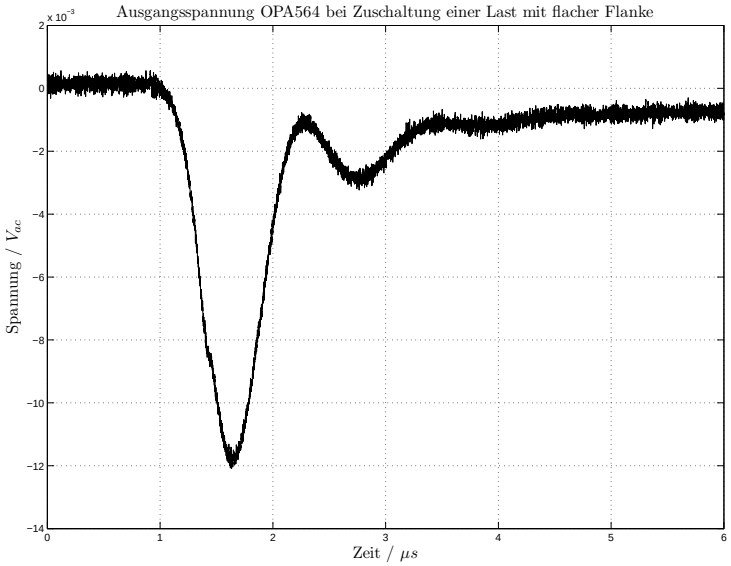


Abbildung 3.23: Ausgangsspannung des OPA564-Testboards beim Zuschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ und $1\,\mu$F, Flankensteilheit von $1\,\mu$s am Gate des Last-Mosfets

$$r(t) = A_{max} \cdot 2 \cdot \pi \cdot f_{max} \cdot cos\left(2 \cdot \pi \cdot f_{max} \cdot t\right) \tag{3.3}$$

$$r_{max} = r(t)\big|_{t=0\,\mathrm{s}} = 2{,}5\,\mathrm{V} \cdot 2 \cdot \pi \cdot 2{,}5\,\mathrm{kHz} \cdot cos\left(2 \cdot \pi \cdot 2{,}5\,\mathrm{kHz} \cdot 0\,\mathrm{s}\right) \tag{3.4}$$

$$r_{max} = 2{,}5\,\mathrm{V} \cdot 2 \cdot \pi \cdot 2{,}5\,\mathrm{kHz} \tag{3.5}$$

$$r_{max} = 39{,}3\,\frac{\mathrm{kV}}{\mathrm{s}} \tag{3.6}$$

Der OPA548 liefert maximal $5\,\mathrm{A}$ Spitzenstrom. Somit ergibt sich eine maximale Kapazität von:

$$I = C \cdot \frac{dU(t)}{dt} = C \cdot r_{max} \tag{3.7}$$

$$C_{max} = \frac{I_{max}}{r_{max}} \tag{3.8}$$

$$C_{max,548} = \frac{5\,\mathrm{A}}{39{,}3\,\frac{\mathrm{kV}}{\mathrm{s}}} \approx 127\,\mu\mathrm{F} \tag{3.9}$$

Ähnliches gilt nach Formel 3.8 für den OPA564, der maximal $1{,}5\,\mathrm{A}$ Strom liefern kann:

$$C_{max,564} = \frac{1{,}5\,\mathrm{A}}{39{,}3\,\frac{\mathrm{kV}}{\mathrm{s}}} \approx 38\,\mu\mathrm{F} \tag{3.10}$$

Bei beiden Reglern handelt es sich bei dem Maximalstrom um eine Strombegrenzung. Somit wird die Hardware auch mit höheren Kapazitäten arbeiten, Flanken werden jedoch unter Umständen abgeflacht, sollte die Kapazität am Ausgang die zuvor in 3.9 bzw. 3.10 errechneten Werte übersteigen.

3.2.4 Auswertung der Simulationen und Messungen

Aus der Arbeit von N. Jegenhorst geht für Sensoren der Klasse 2 hervor, dass am Eingang vor dem AD-Umsetzer ein passiver RC-Tiefpassfilter 1. Ordnung mit einer Grenzfrequenz von

$$f_{g,2} = \frac{1}{2 \cdot \pi \cdot R \cdot C} = \frac{1}{2 \cdot \pi \cdot 50\,\mathrm{k\Omega} \cdot 10\,\mathrm{nF}} \approx 318\,\mathrm{Hz} \tag{3.11}$$

verbaut ist [7]. Ähnliches gilt nach der Abschlussarbeit von Ilgin für die Sensoren der Klasse 1, bei denen ebenfalls ein passiver Tiefpass 1. Ordnung verbaut ist [6]. Die dort verwendete Schaltung ist in Abbildung 3.25 zu sehen.

Die Widerstände sind mit $R_1 = 6{,}8\,\mathrm{k\Omega}$ und $R_2 = 13{,}6\,\mathrm{k\Omega}$ bestückt. Somit ergibt sich eine Grenzfrequenz von:

$$f_{g,2} = \frac{1}{2 \cdot \pi \cdot \frac{R_1 \cdot R_2}{R_1 + R_2} \cdot C} = \frac{1}{2 \cdot \pi \cdot \frac{6{,}8\,\mathrm{k\Omega} \cdot 13{,}6\,\mathrm{k\Omega}}{6{,}8\,\mathrm{k\Omega} + 13{,}6\,\mathrm{k\Omega}} \cdot 100\,\mathrm{nF}} \approx 351\,\mathrm{Hz} \tag{3.12}$$

Da das Einschwingen am Ausgang der Verstärker bei einer Laständerung sehr kurz und hochfrequent ist, kann davon ausgegangen werden, dass die Tiefpassfilter dies ausreichend stark dämpfen werden und diese kleine Störung den Betrieb und die Genauigkeit der Sensoren nicht stört. Für zukünftige Sensoren muss jedoch davon ausgegangen werden, dass sie einen Tiefpass mit einer Grenzfrequenz von bis zu $2{,}5\,\mathrm{kHz}$ verwenden. Aber selbst dabei ist die Dämpfung durch einen Tiefpass erster Ordnung noch ausreichend, da die Frequenz des Schwingens circa zwei bis drei Dekaden höher liegt. Ein weiterer bereits zuvor angesprochener Punkt ist, dass die Ausgangsspannung unabhängig von der Last konstant bleibt. Auch dies ist den Messungen zu Folge gegeben.

Da der OPA564 in den meisten untersuchten Eigenschaften ein besseres Verhalten zeigt als der OPA548, ist dieser auch die Wahl für die Implementierung des Zellspannungsgenerators. Hierzu zählt das maximale Überschwingen bei einer Laständerung und die Dauer bis der Regler wieder eingeschwungen ist. Er bietet außerdem ein paar weitere eventuell nützliche Features wie jeweils einen Pin zur Überstrom- und Übertemperatur-Anzeige. Nachteilig ist der geringere Maximalstrom des Verstärkers. Wie sich in der Berechnung 3.10 zeigt, sollte eine Ausgangskapazität von $38\,\mu\mathrm{F}$ nicht überschritten werden. Dies ist jedoch bei den Sensoren der Klasse 2 der Fall, hier ist ein Kondensator mit $47\,\mu\mathrm{F}$ verbaut. Somit kann es bei steilen Flanken zu Dämpfungen kommen, dabei ist jedoch zu bedenken, dass

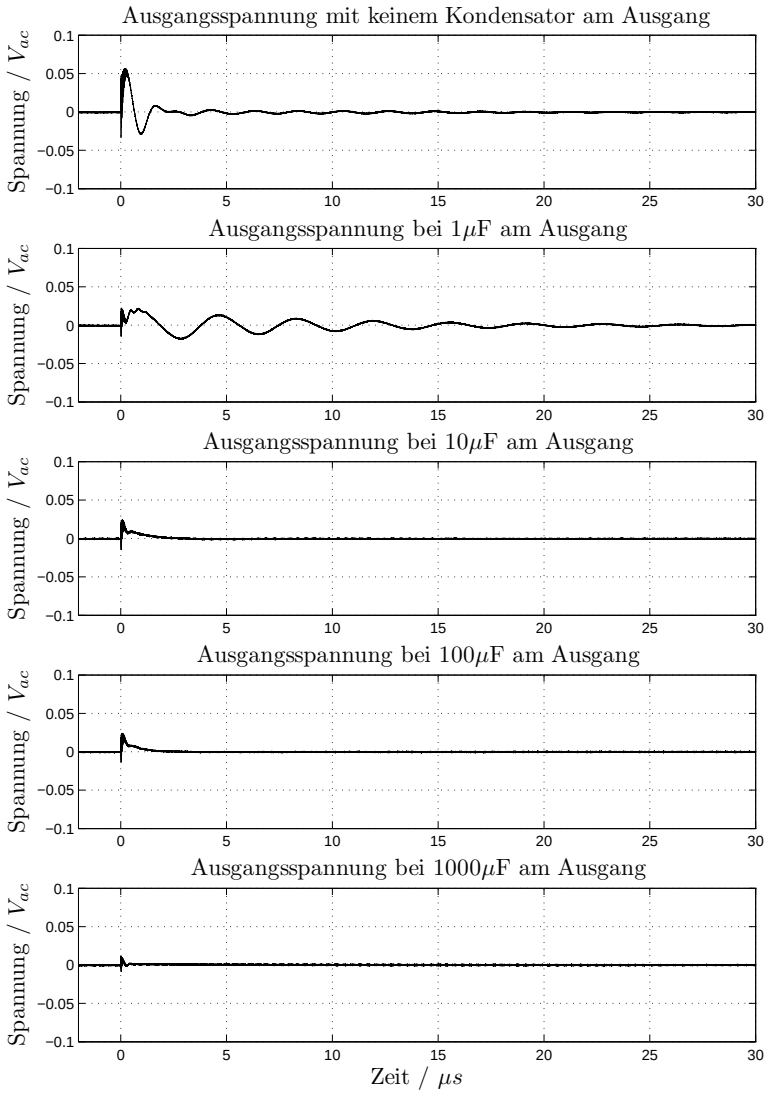


Abbildung 3.24: Ausgangsspannung des OPA548-Testboards beim Abschalten einer Last von $47\,\Omega$, Grundlast $100\,\Omega$ mit verschiedenen Kondensatoren am Ausgang

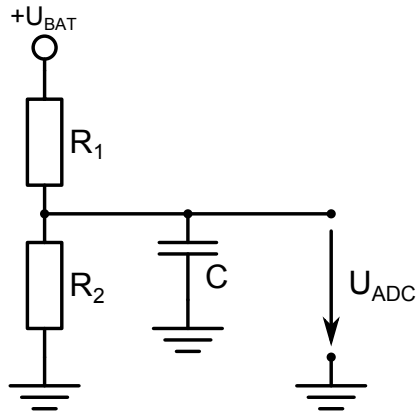


Abbildung 3.25: Im Klasse 1 Sensor verbaute Schaltung für den Antialiasing Filter vor dem AD-Umsetzer

bei den Berechnungen der ungünstigsten Fall angenommen wurde. Es wird in der Praxis nicht vorkommen, dass ein Sinus mit $2{,}5\,\mathrm{kHz}$ bei maximaler Amplitude ausgegeben werden soll.

4 Hardware

Um die Entwicklung und den Aufbau der Hardware zu beschleunigen wird, der Verstärker auf eine extra Platine gebaut, die dann als Steckmodul wie das Stellaris Evaluationsboard auf eine Hauptplatine gesteckt werden kann. Dies bringt des Weiteren den Vorteil, dass der Verstärker einfacher für neue Sensoren angepasst werden kann, ohne den Rest neu entwerfen und aufbauen zu müssen.

Jedes einzelne der Module soll in einem Slot eines 19-Zoll Racks untergebracht werden. Dies bedeutet, dass für jeden der Zellspannungsgeneratoren eine Eurokarte mit $160\,\mathrm{mm}$ mal $100\,\mathrm{mm}$ zur Verfügung steht. Die Platine wird zweilagig gefertigt. Neben den im Kapitel 2 bereits vorgestellten Komponenten sind noch ein paar genauere Spezifizierungen nötig, die in den folgenden Kapiteln ausgeführt werden sollen. Begonnen wird mit dem DA-Umsetzer, über den Tiefpass und Verstärker, weiter in den AD-Umsetzer und zurück zum Mikrocontroller. Anschließend werden die übrigen Komponenten wie die Referenzspannungsquelle, die Strom-Messung und die Stromversorgung genauer betrachtet. Zur Veranschaulichung ist in Abbildung 4.1 nochmals das Blockschaltbild aus der Konzeption dargestellt.

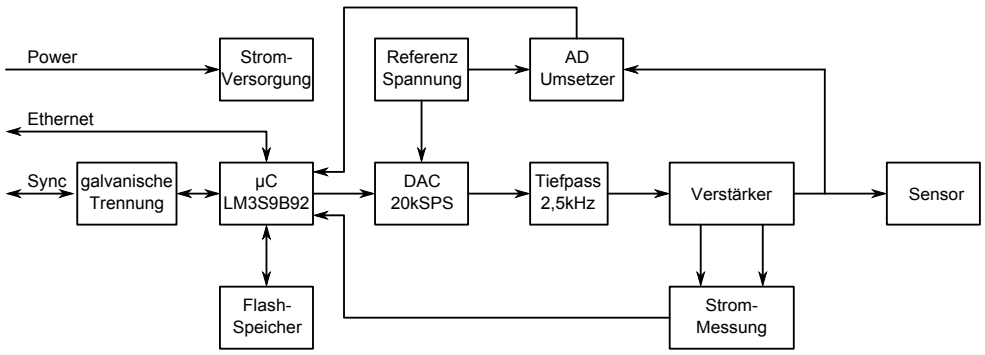


Abbildung 4.1: Vollständiges Blockschaltbild für das finale Konzept der Hardware

4.1 DA-Umsetzer

Der DA-Umsetzer ist maßgeblich an der späteren Auflösung des Generators beteiligt. Nach Kapitel 2.5.1 ergibt sich eine Auflösung von mindestens $13\,\mathrm{Bit}$ für den DA-Umsetzer, wo-

bei noch etwas Reserve eingeplant werden sollte. Angesichts der Tatsache, dass $13\,\mathrm{Bit}$ eine ungewöhnliche Auflösung für einen DA-Umsetzer ist, wird einer mit mindestens $14\,\mathrm{Bit}$ gesucht. Betrachtet man zusätzlich noch die INL und DNL der Umsetzer, muss die Anzahl der Bits unter Umständen noch höher gewählt werden, um zu starke Abweichungen von der idealen Kennlinie zu vermeiden. Natürlich muss er die Mindestabtastrate von $20\,\mathrm{kSPS}$ unterstützen, was jedoch für die meisten Umsetzer gegeben ist. Die Ansteuerung sollte seriell geschehen, also beispielsweise über SPI oder I^{2}C. Die Wahl fällt auf den AD5541 von Analog Devices. Er hat eine Auflösung von $N = 16\,\mathrm{Bit}$, was noch Spiel nach oben lässt [2]. Es wird eine Versorgungsspannung von $2{,}7\,\mathrm{V}$ bis $5{,}5\,\mathrm{V}$ für die digitale Schnittstelle benötigt. Als Referenzspannung können minimal $2\,\mathrm{V}$ und maximal bis zur digitalen Versorgungsspannung verwendet werden. Die INL und DNL liegt typisch jeweils bei $0{,}5\,\mathrm{LSB}$, was es ermöglicht, eine deutlich höhere Auflösung als die geforderten $1{,}22\,\mathrm{V}$ zu erreichen.

4.2 Tiefpass hinter dem DA-Umsetzer

Hinter dem DA-Umsetzer muss ein Formfilter verbaut werden, um das ursprüngliche Spektrum wieder herzustellen. Dieser soll nach Kapitel 2.3 eine Grenzfrequenz von $2{,}5\,\mathrm{kHz}$ haben. Neben der Filterung des Signals soll dieses in der ersten Stufe außerdem auch verstärkt werden. Wie später in Kapitel 4.6 noch erläutert wird, wird eine Referenzspannung von $U_{ref} = 2{,}5\,\mathrm{V}$ für den DA-Umsetzer verwendet. Da eine Ausgangsspannung von mindestens $5{,}5\,\mathrm{V}$ erzeugt werden soll, ist eine Verstärkung von mindestens $v_{min} = 2{,}2$ nötig. Um hier noch etwas Spielraum nach oben zu haben und auch neuere Sensoren zu unterstützten, wird die Verstärkung zu $v = 3$ gewählt. Da der DA-Umsetzer eine Auflösung von $16\,\mathrm{Bit}$ hat, entsteht dadurch noch keine nennenswerte Einschränkung der Auflösung am Ausgang des Generators:

$$U_{LSB} = v \cdot \frac{U_{ref}}{2^N} = 3 \cdot \frac{2{,}5\,\mathrm{V}}{2^{16}} \approx 114\,\mu\mathrm{V} \tag{4.1}$$

Der Ausgang des DA-Umsetzers ist nach dem Datenblatt relativ hochohmig und muss zunächst gepuffert werden [2]. Hierbei wird gleichzeitig die Verstärkung um den Faktor drei durchgeführt. Anschließend wird das Signal durch einen passiven RC-Tiefpassfilter 1. Ordnung gefiltert. Danach folgt nochmals eine Impedanzwandlung des Signals, bevor es nochmals auf einen weiteren RC-Tiefpassfitler gegeben wird. Mit diesem ist es möglich, die Ordnung der Filterung auf zwei anzuheben. Das Signal wird anschließend ungepuffert auf das Verstärkermodul gegeben. Abbildung 4.2 veranschaulicht diesen Signalfluss in einem Blockschaltbild, in Abbildung 4.3 ist der dazugehörige Schaltplan abgebildet.

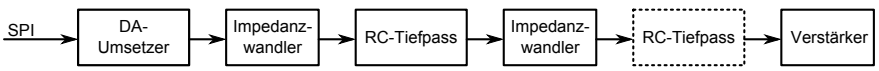


Abbildung 4.2: Schematischer Aufbau von DAC, Formfilter und Verstärker

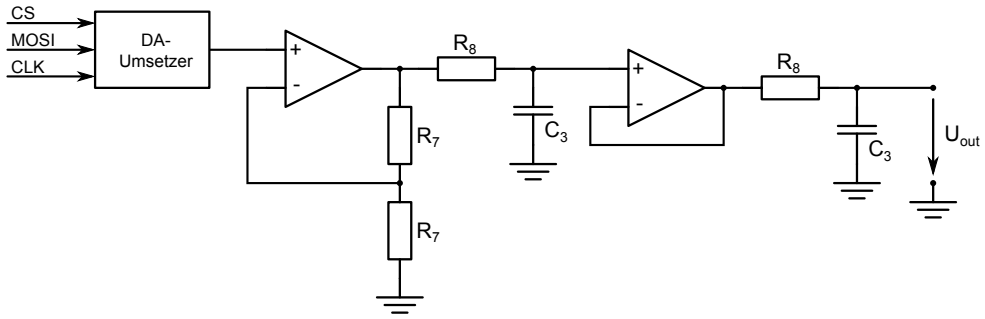


Abbildung 4.3: Schaltplan von DA-Umsetzer, Formfilter und Verstärker

Es sind nun noch die Werte der Widerstände und Kondensatoren zu bestimmen, so dass das Gesamtsystem eine Grenzfrequenz von $2{,}5\,\mathrm{kHz}$ ausweist. Hierfür wird zunächst der Frequenzgang eines einzelnen RC-Tiefpassfilters bestimmt:

$$\underline{U}_{out}(\omega) = \underline{U}_{in}(\omega) \cdot \frac{\frac{1}{j \cdot \omega \cdot C}}{R + \frac{1}{j \cdot \omega \cdot C}} = \underline{U}_{in} \cdot \frac{1}{j \cdot \omega \cdot R \cdot C + 1} \tag{4.2}$$

$$\underline{F}_{RC}(\omega) = \frac{\underline{U}_{out}(\omega)}{\underline{U}_{in}(\omega)} = \frac{1}{j \cdot \omega \cdot R \cdot C + 1} \tag{4.3}$$

Für die weiteren Untersuchungen ist jedoch nur der Amplitudengang interessant:

$$|\underline{F}_{RC}(\omega)| = \frac{1}{\sqrt{1 + (\omega \cdot R \cdot C)^2}} \tag{4.4}$$

Da die beiden in Serie geschalteten Tiefpässe durch einen Impedanzwandler getrennt sind, beeinflussen sie sich gegenseitig nicht. Unter der Voraussetzung, dass die beiden Widerstände und die beiden Kondensatoren identisch sind, können folglich die beiden Frequenzgänge miteinander multipliziert werden, um den Gesamtfrequenzgang zu ermitteln:

$$\underline{F}(\omega) = \underline{F}_{RC}{}^2(\omega) = \frac{1}{1 + (\omega \cdot R \cdot C)^2} \tag{4.5}$$

Um die $-3\,\mathrm{dB}$ Grenzfrequenz zu bestimmen, wird der Frequenzgang mit $\frac{1}{\sqrt{2}}$ gleichgesetzt.

$$\frac{1}{\sqrt{2}} = \frac{1}{1 + (\omega_g \cdot R \cdot C)^2} \tag{4.6}$$

Anschließend wird der Gesamtfrequenzgang nach C umgestellt:

$$\sqrt{2} = 1 + (\omega_g \cdot R \cdot C)^2 \tag{4.7}$$

$$\sqrt{\sqrt{2} - 1} = \omega_g \cdot R \cdot C \tag{4.8}$$

$$C = \frac{\sqrt{\sqrt{2} - 1}}{\omega_g \cdot R} = \frac{\sqrt{\sqrt{2} - 1}}{2 \cdot \pi \cdot f_g \cdot R} \tag{4.9}$$

Für eine Grenzfrequenz von $2{,}5\,\mathrm{kHz}$ und einem gewählten Widerstand von $10\,\mathrm{k\Omega}$ ergibt sich der zu verwendende Kondensator:

$$C = \frac{\sqrt{\sqrt{2} - 1}}{2 \cdot \pi \cdot 2{,}5\,\mathrm{kHz} \cdot 10\,\mathrm{k\Omega}} \approx 4{,}1\,\mathrm{nF} \tag{4.10}$$

Das Signal hat eine Bandbreite von $\pm 2{,}5\,\mathrm{kHz}$, bei einer Abtastung mit $20\,\mathrm{kHz}$ befindet sich somit die erste Störfrequenz auf Grund der periodischen Wiederholung des Spektrums durch die Abtastung bei $17{,}5\,\mathrm{kHz}$. In Abbildung 4.4 wird dies veranschaulicht. Ebenfalls in Abbildung 4.4 dargestellt sind die Dämpfungen durch den Tiefpassfilter und die gewichtete si-Funktion auf Grund der Wiedergabe in Form von Rechtecken durch den DA-Umsetzer. Das Skript aus Anhang A.3.2.7 berechnet bei $17{,}5\,\mathrm{kHz}$ eine Dämpfung von $43{,}7\,\mathrm{dB}$. Diese Dämpfung ist für die Anwendung im Zellspannungsgenerator ausreichend.

4.3 Leistungsverstärker

Hinter den Formfiltern und Vorverstärkern wird direkt der Leistungsverstärker geschaltet. Wie in Kapitel 3 bereits geklärt, kommt als Verstärker der OPA564 von Texas Instruments zum Einsatz. Der Verstärker wird im Prinzip als einfacher Impedanzwandler betrieben, es gibt aber auch einige Bestückungsoptionen, um beispielsweise die differentielle Rückführung der Ausgangsspannung zum Verstärker zu ermöglichen. Somit ist es dem Verstärker möglich, die Spannungsabfälle über den Zuleitungskabeln zu den Sensoren zu kompensieren. Es werden des Weiteren eine Reihe von Bestückungsoptionen hinter dem Verstärker vorgesehen, um das Ausgangssignal weiter zu filtern. Hierzu gehört ein Serienwiderstand sowie eine Serieninduktivität und einige Kapazitäten gegen Masse. Die Rückführung kann neben der externen Verbindung von verschiedenen Stellen abgenommen werden. Am Ausgang werden zudem noch 6 Dioden in Flussrichtung nach Masse vorgesehen, um auf einfache Weise eine Überspannung am Ausgang zu verhindern. Abbildung 4.6 verdeutlicht dies.

Soll der Verstärker seine Ausgangsspannung anhand der herausgeführten differentiellen Leitungen SEN- und SEN+ regeln, wird R_4 nicht bestückt, $R_3 = R_5$ und $R_1 = R_2$. Die po-

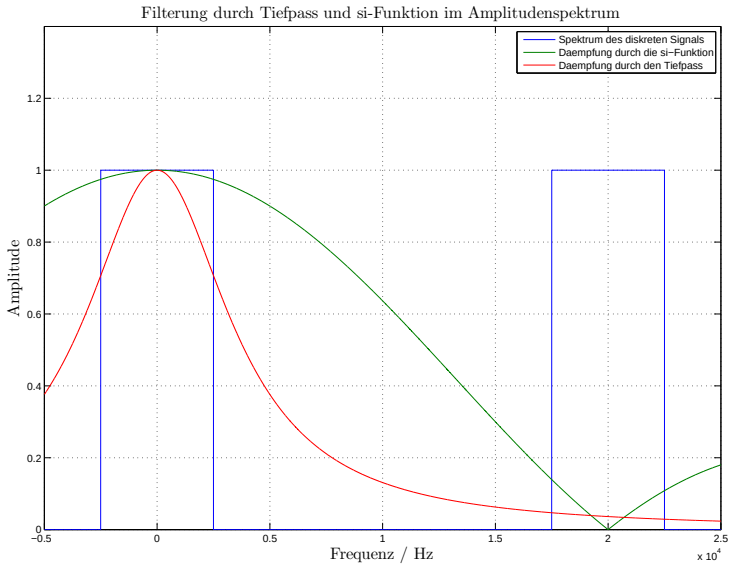


Abbildung 4.4: blau: Spektrum des diskreten Signals mit Bandbegrenzung auf $2{,}5\,\mathrm{kHz}$ und einer Abtastrate von $20\,\mathrm{kHz}$, rot: Dämpfung durch den Tiefpass, grün: Dämpfung durch die si-Funktion

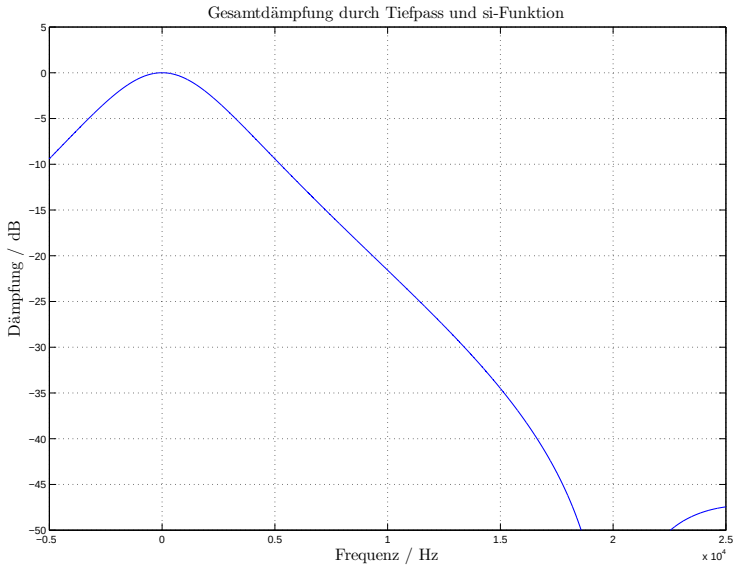


Abbildung 4.5: Dämpfung der si-Funktion und des Tiefpasses am Ausgang der DA-Umsetzers

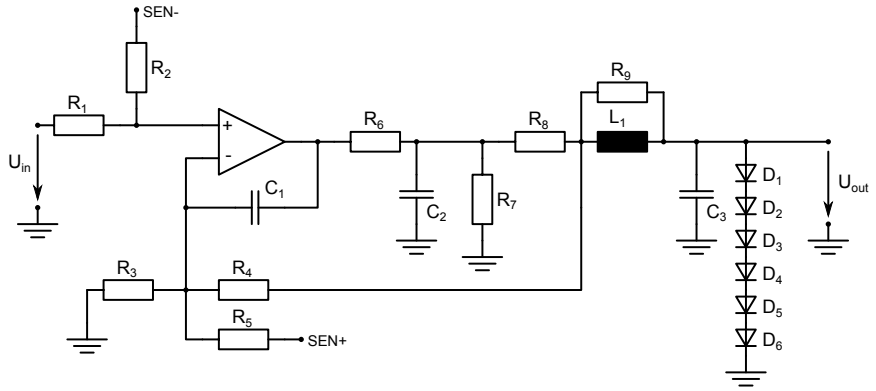


Abbildung 4.6: Schaltplan des Verstärkers mit differentieller Rückführung der Ausgangsspannung

sitive Rückführung ersetzt dabei einfach die normale Rückführung, die Rückführung der Masse muss jedoch noch auf das Signal aufaddiert werden. Hierfür wird zunächst der Mittelwert des Eingangssignals und der zurückgeführten Masse durch R_1 und R_2 erzeugt. In der Rückführung des Ausgangssignals zum Verstärker wird ein Spannungsteiler mit R_3 und R_5 realisiert, der das Signal wieder um den Faktor zwei verstärkt. Dies wird auch als Summierverstärker bezeichnet. Neben dem eigentlichen Verstärker soll auf dem Steckmodul auch die Strommessung realisiert werden, da diese mit an den Verstärker gebaut werden muss.

4.4 Strommessung

Wie bereits im Kapitel 2.8 geklärt, ist eine Messung des Stromes durch den Hall-Effekt nicht möglich. Stattdessen soll ein Shunt-Widerstand verwendet werden, der hinter den Ausgang des Verstärkers geschaltet wird. Die Rückkopplung für den Leistungsverstärker erfolgt dann erst hinter dem Widerstand, um diesen zu kompensieren. Weiterhin sollte hinter dem Widerstand kein Kondensator platziert werden, da sonst bei Veränderungen der Spannung der Blindstrom in den Kondensator mit gemessen wird. Da nicht auszuschließen ist, dass auf dem Sensor ebenfalls ein Kondensator verbaut ist, muss damit gerechnet werden, dass auch negative Ströme fließen.

Mit dem Einbau des Widerstandes zur Messung des Stroms hinter den Verstärker ergibt sich ein weiteres Problem. Die Spannung über dem Widerstand muss differentiell abgegriffen und verstärkt werden, was mit normalen Operationsverstärkern nur mit größerem Aufwand möglich ist. Eine einfachere Lösung bieten Instrumentenverstärker. Sie sind in der Lage, die Spannung differentiell zu messen und unabhängig vom Massepotential zu verstärken. Einer dieser Instrumentenverstärker ist der AD8226 von Analog Devices. Die Verstärkung ist laut dem Datenblatt mit einem Widerstand frei wählbar zwischen 1 und 1000 [3]. Er bietet

weiterhin einen Referenzeingang, um den Nullpunkt der Ausgangsspannung festzulegen. So ist es möglich, auch negative Spannungen messen zu können, ohne einen AD-Umsetzer mit bipolarer Versorgung zu benötigen. Hierfür wird der Referenz-Eingang mit einem Spannungsteiler auf die halbe Betriebsspannung gelegt, wobei diese Spannung vorher noch durch einen Impedanzwandler gepuffert werden muss, denn der AD8226 braucht eine niederohmige Quelle [3].

Für die Analog-Digital Wandlung zur Strommessung kann der interne AD-Umsetzer des Stellaris Mikrocontrollers verwendet werden. Dieser arbeitet mit einer Referenzspannung von $3{,}3\,\mathrm{V}$, wodurch sich eine Referenzspannung für den Instrumentenverstärker von $1{,}65\,\mathrm{V}$ ergibt. Verwendet man im Pfad des Verstärkers einen Shunt-Widerstand von $1\,\Omega$ so entspricht der maximale Strom von $250\,\mathrm{mA}$ einem Spannungsabfall von $250\,\mathrm{mV}$ am Shunt-Widerstand. Will man dies verstärken auf maximal $U_{FS} = 1{,}65\,\mathrm{V}$ folgt daraus eine Verstärkung von:

$$v_{1,max} = \frac{U_{out}}{U_{in}} = \frac{1{,}65\,\mathrm{V}}{250\,\mathrm{mV}} = 6{,}6 \tag{4.11}$$

Da der Instrumentenverstärker mit seinem Ausgang nicht ganz an die Versorgungsspannungen heran kommt und auch noch etwas Reserve eingeplant werden sollte, muss die Verstärkung etwas kleiner gewählt werden und wird somit zu $v_1 = 6$ festgelegt. Aus [25] geht hervor, dass der Stellaris Mikrocontroller einen AD-Umsetzer mit $N = 10\,\mathrm{Bit}$ hat. Somit ergibt sich eine Auflösung von:

$$U_{LSB,1} = \frac{1}{v_1} \cdot \frac{U_{FS}}{2^N} = \frac{1}{6} \cdot \frac{1{,}65\,\mathrm{V}}{2^{10}} \approx 269\,\mu\mathrm{V} \tag{4.12}$$

Die Spannungsauflösung entspricht einer Stromauflösung von $I_{LSB,1} = 269\,\mu\mathrm{A}$.

Es soll weiterhin möglich sein, auch den Ruhestrom von den Sensoren zu messen. Ein Klasse-2 Sensor benötigt im Standby ungefähr $100\,\mu\mathrm{A}$. Mit einer Auflösung von $269\,\mu\mathrm{A}$ ist es nicht möglich so kleine Ströme zu messen. Es wird also noch ein zweiter Verstärker mit einer höheren Verstärkung benötigt. Es ist zudem davon auszugehen, dass zukünftige Sensoren im Standby noch weniger Strom verbrauchen. Die Auflösung sollte daher im Bereich von einigen μA liegen. Legt man einen Bereich von $I_{FS,2} = \pm 2\,\mathrm{mA}$ fest, so ergibt sich eine Verstärkung von:

$$v_{2,max} = \frac{U_{out}}{U_{in}} = \frac{1{,}65\,\mathrm{V}}{2\,\mathrm{mV}} = 825 \tag{4.13}$$

Diese Verstärkung wird auf $v_2 = 800$ abgerundet. Die Auflösung beträgt dann:

$$U_{LSB,2} = \frac{1}{v_2} \cdot \frac{U_{FS}}{2^N} = \frac{1}{800} \cdot \frac{1{,}65\,\mathrm{V}}{2^{10}} = 2{,}014\,\mu\mathrm{V} \tag{4.14}$$

Dies entspricht einer Stromauflösung von $I_{LSB,2} = 2{,}014\,\mu\mathrm{A}$. Bei einer solch hohen Ver-

stärkung ist damit zu rechnen, dass das Signal stark verrauscht ist und einen hohen Offset aufweist.

Bevor das Signal von dem Shunt-Widerstand auf die Verstärker geht, muss es noch gefiltert werden. Hierfür kommt ein einfacher RC-Tiefpass 1. Ordnung zum Einsatz mit einer Grenzfrequenz von $1\,\mathrm{kHz}$. In Abbildung 4.7 wird der Schaltplan der Strommessung dargestellt.

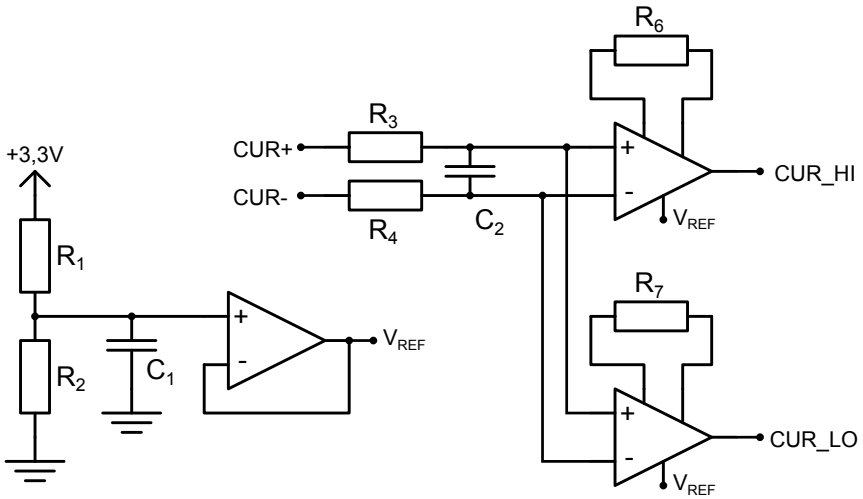


Abbildung 4.7: Schaltplan der Strommessung hinter dem Verstärker. Shuntwiderstand zwischen CUR+ und CUR-

Die Signale V_{REF}, CUR_LO und CUR_HI werden zum Stellaris Mikrocontroller geführt, um diese mit dem internen AD-Umsetzer zu messen. Dabei werden dann in der Software die Differenzen von CUR_LO und CUR_HI jeweils zu V_{REF} gebildet, um die einzelnen Ströme zu bestimmen.

4.5 AD-Umsetzer

Hinter dem Verstärker kommt der AD-Umsetzer zum Messen der Ausgangsspannung. An diesen AD-Umsetzer werden ähnliche Anforderungen wie an den DA-Umsetzer gestellt. Er sollte ebenfalls eine Auflösung von mindestens $14\,\mathrm{Bit}$ haben und dabei eine geringe INL und DNL. Für diese Anforderungen eignet sich der AD7798 von Analog Devices hervorragend. Laut dem Datenblatt hat er eine Auflösung von $16\,\mathrm{Bit}$ bei einer Versorgung von $2{,}7\,\mathrm{V}$ bis $5{,}25\,\mathrm{V}$ jeweils für die digitale Schnittstelle und das analoge Frontend [1]. Als Referenzspannung werden zusätzlich $2{,}5\,\mathrm{V}$ benötigt [1]. Der Umsetzer arbeitet mit maximal 470 Samples pro Sekunde [1]. Dies ist zwar nicht viel im Vergleich zu dem DA-Umsetzer, der AD-Umsetzer wird jedoch auch nur benötigt, um eine konstante Spannung genau einzustellen bzw. für die Kalibrierung.

Da das Ausgangssignal bis zu $5{,}5\,\mathrm{V}$ haben kann, muss ein Spannungsteiler vorgesehen werden, da der AD-Umsetzer nur mit Spannungen bis $2{,}5\,\mathrm{V}$ umgehen kann. Dieser Spannungsteiler bringt einen hohen Fehler in die Messung. Selbst wenn man Hochpräzisionswiderstände mit $0{,}1\,\%$ Toleranz benutzt, ergibt dies für einen Eingasspannungsbereich von $7{,}5\,\mathrm{V}$ eine maximale Abweichung von $7{,}5\,\mathrm{mV}$, was für eine genaue Messung ungenügend ist. Somit kommt man um eine Kalibrierung des AD-Umsetzers nicht herum, wenn man eine genauere Messung durchführen möchte, wobei auch die Ungenauigkeit der Referenzspannungsquelle noch hinzu kommt.

4.6 Referenzspannungsquelle

Sowohl der AD-Umsetzer als auch der DA-Umsetzer benötigen eine Referenzspannungsquelle mit einer Ausgangsspannung von $2{,}5\,\mathrm{V}$. Dabei ist ein möglichst geringes Rauschen von hohem Interesse. Die absolute Genauigkeit sollte ebenfalls hoch sein. Die REF50xx Serie von Texas Instruments bietet Referenzspannungsquellen mit einer sehr hohen Genauigkeit von $0{,}05\,\%$ und einem sehr geringen Rauschen [20]. Der REF5025 hat aus der Serie die korrekte Spannung von $2{,}5\,\mathrm{V}$. Ein weiterer Vorteil dieser Reihe ist die sehr geringe Temperaturabhängigkeit mit maximal $3\,\mathrm{ppm}/^{\circ}\mathrm{C}$ für die High-Grade Variante [20].

Trotz der sehr hohen Genauigkeit von $0{,}05\,\%$ kommt auch hierbei eine maximale Abweichung der Wandler von $3{,}75\,\mathrm{mV}$ bei einer Verstärkung bzw. Teilung von 3 hinzu. Auch dies spricht nochmals für eine externe Kalibrierung zumindest von dem AD-Umsetzer. Um die Kalibrierung auch über die Temperatur zu ermöglichen, ist ein Temperatursensor und ein höherer Softwareaufwand nötig.

4.7 Temperatursensor

Für den Temperatursensor wird der Chip TMP102 von Texas Instruments verwendet, da mit diesem Chip bereits Erfahrungen im Rahmen des Projektes BATSEN gesammelt wurden. Aus dem Datenblatt [23] geht hervor, dass der Sensor über $\mathrm{I}^2\mathrm{C}$ angesprochen wird und eine Genauigkeit von $0{,}5\,^{\circ}\mathrm{C}$ bei einem Temperaturbereich von $-25\,^{\circ}\mathrm{C}$ bis $85\,^{\circ}\mathrm{C}$ bietet.

Neben der Anpassung von Korrekturwerten der analogen Schaltungsteile kann der Temperatursensor auch zur Erkennung von zu hohen Temperaturen im Gehäuse verwendet werden. Somit ist es möglich, Bauteile vor einer Beschädigung zu schützen, indem man rechtzeitig Wärme erzeugende Komponenten wie den Verstärker und den Mikrocontroller abschaltet.

4.8 Speicher

Nachdem alle analogen und Mixed-Signal Komponenten behandelt sind, gibt es noch ein paar weitere digitale Bauteile. Eines davon ist der Flash-Speicher zum Zwischenspeichern der Samples für eine Spannungs-Sequenz. Er soll nach Kapitel 2.7 mindestens $2{,}4\,\mathrm{MByte}$ speichern können. Die Lesegeschwindigkeit muss ausreichend sein, um 20000 Samples pro Sekunde lesen zu können, was jedoch für die meisten Flash-ICs kein Problem darstellt. Da es keine Flash-Speicher mit $2{,}4\,\mathrm{MByte}$ oder $3\,\mathrm{MByte}$ gibt, muss einer mit $4\,\mathrm{MByte}$ genommen werden. Hier bietet sich der SST25VF032B von SST an. Aus dem Datenblatt geht hervor, dass er die benötigten $4\,\mathrm{MByte}$ Speicher hat und über SPI bei bis zu $80\,\mathrm{MHz}$ ansprechbar ist [14]. Eine Chip-Select-Leitung ermöglich das Anschließen mehrerer dieser Speicher an einem gemeinsamen Bus. Um auch die gemessenen Ströme abspeichern zu können, werden in der Hardware zwei dieser Flash-Speicher vorgesehen, die gemeinsam an einem Bus angeschlossen sind.

4.9 Synchronisation

Bei der Synchronisation handelt es sich um eine einfache Leitung, die alle Mikrocontroller miteinander verbindet. Die Leitung muss zudem an jedem Controller galvanisch getrennt werden. Hierfür wird der IC ADUM1201 vom Analog Devices verwendet. Dies ist ein IC, welches eine galvanische Trennung von zwei Leitungen in jeweils zwei unterschiedliche Richtungen ermöglicht. Die Ausgänge sind kurzschlussfest und benötigen nicht wie die meisten Optokoppler einen externen Pull-Up Widerstand. Der IC arbeitet zudem nicht wie ein Optokoppler mit Licht, sondern über kleine integrierte Transformatoren. Die beiden Leitungen im IC werden benötigt, da eines der Module als Master arbeiten soll. Eines der Module muss also auch auf der Synchronisationsleitung senden können. Um auszuwählen welches der Module der Master sein soll, wird ein einfacher Jumper verwendet.

Da die Module auf bis zu 40 Stück erweitert werden sollen und diese unter Umständen in mehreren Schränken untergebracht werden, ist davon auszugehen, dass die Synchronisationsleitung Strecken von einigen Metern überwinden muss. Der ADUM1201 erzeugt am Ausgang Flanken mit einer Steilheit von bis zu $10\,\mathrm{ns}$ bei einer Versorgung von $3{,}3\,\mathrm{V}$, wie aus dem Datenblatt hervorgeht [5]. Auf Grund der hohen Flankensteilheit und der relativ langen Kabel werden eine Reihe von Abschlüssen in der Hardware vorgesehen, wie Abbildung 4.8 zeigt. Da die Leitung vermutlich sternförmig verlegt wird, kann es passieren, dass es trotz der korrekten Bestückung nicht möglich ist, alle Reflexionen zu verhindern. Eine Alternativlösung ist die Unterdrückung doppelter Interrupts, die auf Grund von Reflexionen augetreten sind, durch die Software. Der Interrupt kann beispielsweise nach dem Erkennen einer Flanke für einige Mikrosekunden abgeschaltet werden.

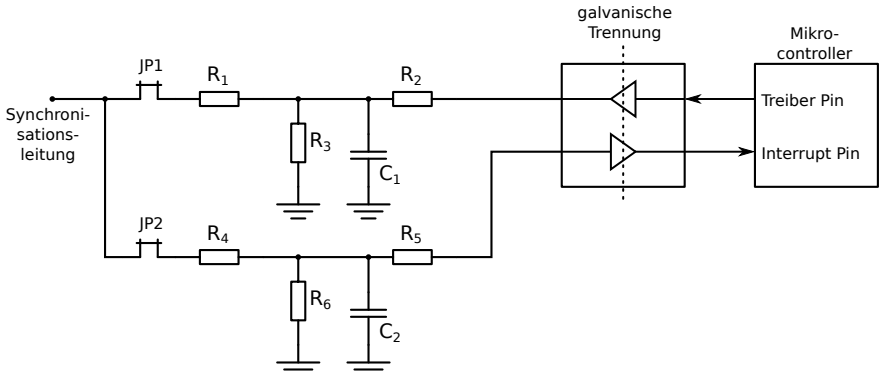


Abbildung 4.8: Schaltplan für die Synchronisation der Module untereinander

4.10 Stromversorgung

Um den Stellaris Mikrocontroller, den Verstärker und die übrigen ICs mit Strom zu versorgen, müssen einige Regler dimensioniert werden. Dabei ist zu beachten, dass die Versorgung der analogen Komponenten über gesonderte Regler geschieht, um die Einkopplung von Spannungsspitzen durch die digitalen Komponenten zu verhindern. Hierfür wird auch eine getrennte analoge Masse eingeführt.

Die meisten ICs wie der Flash und der AD-Umsetzer benötigen als Versorgung $3{,}3\,\mathrm{V}$ für die digitale Schnittstelle. Das Evaluations-Board des Stellaris Mikrocontrollers benötigt eine $5\,\mathrm{V}$ Spannungsversorgung, wobei der Controller selbst nur $3{,}3\,\mathrm{V}$ braucht. Diese $3{,}3\,\mathrm{V}$ werden jedoch auf dem Evaluations-Board erzeugt.

Die analogen Komponenten wie der AD-Umsetzer benötigen ebenfalls eine $3{,}3\,\mathrm{V}$-Versorgung für das analoge Frontend. Diese soll jedoch von den digitalen $3{,}3\,\mathrm{V}$ getrennt sein, daher wird dafür ein Linearregler verwendet. Es wird weiter die Versorgung für den Verstärker, die Vorverstärker und Filter benötigt. Diese Spannung soll ebenfalls über einen Linearregler erzeugt werden, da diese im Vergleich zu Step-Down-Reglern eine deutlich sauberere Spannung erzeugen. Da der Verstärker bis zu $250\,\mathrm{mA}$ liefern können soll, ist es sinnvoll, einen Low-Drop-Out (LDO) Linearregler zu verwenden, um möglichst wenig Verlust im Regler zu haben. LDO-Regler benötigen nur eine um einige hundert Millivolt höhere Eingangsspannung im Vergleich zur Ausgangsspannung. Um die Ausgangsspannung des LDO dennoch variabel zu halten, wird ein zusätzlicher Step-Down-Regler vorgeschaltet, der in der Lage ist, die benötigte Eingangsspannung des LDO sehr verlustarm zu erzeugen. Somit wird eine sehr saubere Spannung mit einer hohen Effizienz erzeugt.

Um die galvanische Trennung der Module zu erreichen, muss die Versorgung ebenfalls galvanisch getrennt werden. Hierfür eignet sich ein DC/DC-Wandler. Diese gibt es voll integriert in einem einzigen Gehäuse für verschiedene Spannungen. In dem Blockschaltbild 4.9 ist das Gesamtkonzept der Stromversorgung nochmals zusammengefasst.

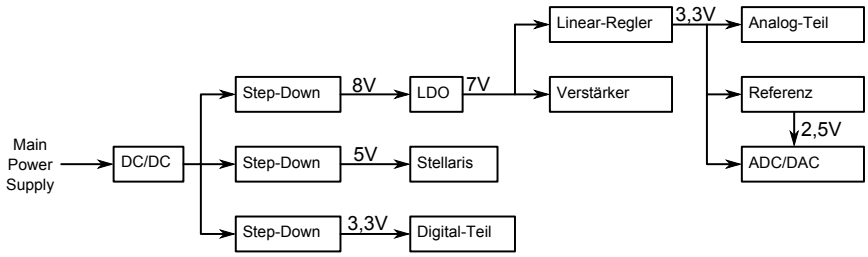


Abbildung 4.9: Blockschaltbild der Stromversorgung

Es sollen nun die Komponenten aus dem Blockschaltbild 4.9 einzeln genauer betrachtet werden. Begonnen wird im linken Teil des Schaltbildes mit dem DC/DC-Wandler, der Struktur des Blockschaltbildes weiter folgend. Bei der Auswahl des DC/DC-Wandlers ist recht viel Freiraum vorhanden. Die einzige Voraussetzung ist, dass die Step-Down-Regler, die hinter dem DC/DC-Wandler sitzen, mit der Ausgangsspannung arbeiten können müssen. Für gewöhnlich arbeiten solche Step-Down-Regler mit Spannungen bis mindestens $24\,\mathrm{V}$. Die Ausgangsspannung muss zudem über $8\,\mathrm{V}$ liegen, da der oberste Regler noch $8\,\mathrm{V}$ erzeugen können muss. Einzig ist sonst noch die Leistung des Wandlers entscheidend. Der Verstärker alleine benötigt eine Leistung von mindestens

$$P_{AMP} = 8\,\mathrm{V} \cdot 250\,\mathrm{mA} = 2\,\mathrm{W} \tag{4.15}$$

Hinzu kommt der Verbrauch des Stellaris Evaluationsboards, der nicht genau angegeben werden kann, da dies stark von der Beschaltung und der Software abhängig ist. Es sollte also, um ausreichend Reserve zu haben, ein DC/DC-Wandler mit mindestens $4\,\mathrm{W}$ verbaut werden. Die Wahl fällt auf den NMXS1215UC von Murata. Er hat eine Eingangsspannung von $12\,\mathrm{V}$, eine Ausgangsspannung von $15\,\mathrm{V}$ und liefert maximal $5\,\mathrm{W}$ [9].

Bei den Step-Down-Wandlern fällt die Wahl auf den TPS5410 von Texas Instruments. Dieser hat einen Eingangsspannungsbereich von $5{,}5\,\mathrm{V}$ bis $36\,\mathrm{V}$, einen maximalen Ausgangsstrom von $1\,\mathrm{A}$ und einen integrierten MOSFET [18]. Außerdem ist er in der Lage, alle benötigen Spannungen zu erzeugen. Die Berechnung der einzelnen externen Bauteile erfolgt mit dem Programm SwitcherPro von Texas Instruments. Wie im Kapitel 6.1 noch gezeigt wird, wird der Step-Down-Regler IC2 durch den kompatiblen Chip TPS5420 ersetzt.

An den Linearregler für den Verstärker wird die Forderung gestellt, einen Spannungsabfall von einigen hundert Millivolt bei einer Ausgangsspannung von ca. $7\,\mathrm{V}$ zu ermöglichen. Gleichzeitig muss er in der Lage sein, einen Strom von mindestens $250\,\mathrm{mA}$ liefern zu können, wobei hier noch etwas Reserve nach oben sein sollte. Diese Anforderungen erfüllt der TPS73801 von Texas Instruments. Er hat eine maximale Dropout-Spannung von $300\,\mathrm{mV}$, liefert bis zu $1\,\mathrm{A}$ und hat eine einstellbare Ausgangsspannung von $1{,}21\,\mathrm{V}$ bis $20\,\mathrm{V}$ [21]. Somit ist es auch möglich, die Hardware einfach auf einen größeren Spannungsbereich für den Ausgang des Generators anzupassen.

Der Linearregler für die Versorgung der analogen Komponenten, wie beispielsweise dem AD-Umsetzer, benötigt nur wenig Strom und eine konstante Spannung von $3{,}3\,\mathrm{V}$. Hier eignet sich der Texas Instruments Chip TPS76133. Der Linearregler bestitzt eine feste Ausgangsspannung von $3{,}3\,\mathrm{V}$ und liefert bis zu $100\,\mathrm{mA}$ bei einem Eingangsspannungsbereich von $3{,}68\,\mathrm{V}$ bis $16\,\mathrm{V}$ [16].

4.11 Backplane

Um die Module in einem Rack miteinander zu verbinden, ist eine Backplane eine einfache Lösung. Es handelt sich dabei um eine Platine, die auf der Rückseite des Racks montiert wird und die Stecker hält, in die die einzelnen Module gesteckt werden. Dabei ist darauf zu achten, dass die Backplane nur die Hälfte der Höhe in Anspruch nimmt, da im unteren Teil die Ethernet-Stecker herausgeführt werden müssen. Neben den vier Versorgungsleitungen für die einzelnen Module ($12\,\mathrm{V}$) und der Synchronisationstreiber ($3{,}3\,\mathrm{V}$) muss noch eine Leitung für die Synchronisation vorgesehen werden. Die fertige Platine ist in Anhang B.2.6 zu finden, der dazugehörige Schaltplan in Anhang B.1.6. Sowohl im Schaltplan als auch im Layout sind noch zwei Synchronisationsleitungen vorgesehen. Diese beiden Leitungen müssen über den vorgesehenen Jumper kurzgeschlossen werden.

5 Software

Die Software besteht aus zwei Teilen. Zum einen benötigt der Stellaris Mikrocontroller eine umfangreiche Software zur Steuerung der Hardware und zur Kommunikation mit dem PC, zum anderen ist eine Software auf dem PC nötig, um die Generatoren einfach anzusprechen. Die Stellaris-Software wird in C geschrieben und es wird die Bibliothek StellarisWare von Texas Instruments verwendet. Auf dem PC sollen die Generatoren über MATLAB steuerbar sein, es wird daher in MATLAB eine kleine Bibliothek geschrieben, um die Generatoren zu verwalten und zu steuern.

Die Steuerung der Generatoren erfolgt über den TCP/IP-Stack. Sowohl MATLAB als auch StellarisWare bieten hierfür eine umfangreiche und einfache Bibliothek. Auf dem Mikrocontroller wird hierbei der lwIP TCP/IP-Stack benutzt, der Teil von StellarisWare ist. Die Entwicklung der Mikrocontrollersoftware erfolgt im Code Composer Studio 5.1.0 (CCS), auf dem PC wird MATLAB in der Version R2010b eingesetzt.

5.1 Allgemeiner Aufbau der Kommunikation

Die einzelnen Generatoren sollen über eine TCP-Verbindung gesteuert werden. Der Ablauf dieser Verbindung soll so gestaltet sein, dass es möglich ist, sich mit einem einfachen Telnet-Client zu jedem der Module zu verbinden. Dies bedeutet, dass alle Daten, die übertragen werden, ASCII kodiert werden müssen. Dies bedeutet zwar einen Mehraufwand an Datenübertragung, hält das Protokoll jedoch sehr einfach und einfach erweiterbar. Die Geschwindigkeit von Ethernet ist dennoch mehr als ausreichend, um die Module in einer angemessenen Zeit zu programmieren.

Ein Problem stellt das Auffinden der Module im Netzwerk dar, denn es ist nicht effizient zu versuchen, zu jeder existierenden IP-Adresse eine Verbindung aufzubauen. Dies lässt sich durch die Verwendung von Broadcast Nachrichten lösen. Hierfür eignet sich UDP, es wird eine Broadcast Nachricht an alle Geräte im Netzwerk verschickt. Jedes der Module antwortet dann auf diese Nachricht, wodurch der PC in der Lage ist, alle im Netzwerk vorhandenen Generatoren zu identifizieren.

5.2 Software auf dem Stellaris Mikrocontroller

Die Software auf dem Mikrocontroller ist sehr umfangreich und soll hier in funktionalen Blöcken beschrieben werden. Als erstes erfolgen in der Funktion main() eine Reihe von Initialisierungen wie der LEDs, des Netzwerkes oder des TCP-Sockets für die Kommunikation. Anschließend werden alle weiteren Aktionen interruptbasiert ausgeführt, die main()-Funktion erfüllt also keine weitere Aufgaben.

5.2.1 Netzwerkverbindung

Um eine Verbindung mit dem Modul aufbauen zu können, ist zunächst eine Netzwerkverbindung erforderlich. Die lwIP-Bibliothek bietet hierfür die nötigen Funktionen. Dazu gehören neben einem TCP/IP-Stack und sonstiger nötiger Protokolle wie beispielsweise ARP die Implementierung eines DHCP-Clients. Der DHCP-Client kann eingesetzt werden, um die IP-Adresse für das Modul automatisch zu beziehen, jedoch ist dafür ein DHCP-Server erforderlich. Da dieser DHCP-Server im Labor nicht vorhanden ist, wird die automatische IP-Adressen-Verteilung nicht genutzt. Stattdessen werden statische Adressen verwendet. Damit nicht für jedes Modul eine eigene Firmware mit einer anderen IP-Adresse erstellt werden muss, werden 8 externe SMD-Lötbrücken verwendet, um eine feste IP-Adresse einzuprogrammieren. Dabei wird davon ausgegangen, dass die IP-Adresse immer die Struktur *192.168.0.xxx* hat. Die letzte Stelle wird durch die Jumper bestimmt.

Um die Module mit DHCP zu betreiben ist, nur eine kleine Änderung in der Software nötig. In der Datei *ethernet.c* sind in der Initialisierungsfunktion *ethernet_init()* zwei Zeilen, die die Funktion *lwIPInit(...)* aufrufen. Je nachdem welche der beiden Funktionsaufrufe genutzt wird, wird das Netzwerk über DHCP oder mit einer statischen IP-Adresse initialisiert. Die Subnetmask für die statische IP-Adresse ist *255.255.255.0*.

5.2.2 Finden der Module

Wie bereits zuvor erklärt, sollen die Module über eine UDP Broadcast Nachricht gefunden werden. Hierfür wird beim Start des Controllers nach der Netzwerkinitialisierung ein UDP-Socket erstellt, der auf dem Port 56936 läuft. Erhält dieser Socket ein Paket, wird durch lwIP eine Funktion aufgerufen, in der die weitere Kommunikation abgewickelt wird. Unabhängig vom Inhalt des Paketes wird eine Antwort an den Absender geschickt in der sich die eigene IP-Adresse befindet. Die Tatsache, dass die eigene IP-Adresse mitgeschickt werden muss, hängt mit einem Problem in MATLAB zusammen, dass später in Kapitel 5.3.1 noch genauer erklärt wird.

5.2.3 Steuerung der Module

Die Steuerung der Module geschieht über einen TCP-Socket, der ebenfalls auf dem Port 56936 auf eine Verbindung wartet. Wird eine neue Verbindung hergestellt, wird diese automatisch zur neuen Steuer-Verbindung. Eine eventuell bereits geöffnete Verbindung wird geschlossen. Die Software verbleibt nach dem Verbindungsaufbau, in dem Zustand in dem sie aus der vorherigen Verbindung zurückgelassen wurde. Es ist also die Aufgabe der Steuerung, nach Verbindungsaufbau für einen definierten Zustand zu sorgen.

Nach dem erfolgreichen Verbindungsaufbau wird zunächst ein Dollarzeichen an den Client gesendet, so dass dieser weiß, dass die Verbindung erfolgreich hergestellt wurde und der Generator bereit ist, Befehle zu empfangen. Dieses Dollarzeichen wird zudem immer nach erfolgreichem Ausführen eines Befehls gesendet. Der Client wartet vor dem Absenden eines neuen Befehls immer auf das Dollarzeichen, somit wird dem Verlust von Daten durch Pufferüberläufe vorgebeugt.

5.2.4 Annahme und Verarbeitung von Befehlen

Der Kern der Steuerungs-Software ist der Automat zur Verwaltung und der Interpretierung der Befehle, die über den TCP-Socket empfangen werden. Werden Daten empfangen, werden diese zunächst in einem FIFO zwischengespeichert, da es möglich ist, dass ein einziger Befehl über mehrere Pakete verteilt eintrifft. Anschließend wird der Automat aus Abbildung 5.1 ausgeführt. Dieser prüft zunächst, ob ein vollständiger Befehl im FIFO vorhanden ist. Ist dies der Fall, wird der Befehl aus dem FIFO ausgelesen und interpretiert. In den folgenden Kapiteln soll auf die einzelnen Befehle dieses Diagramms genauer eingegangen werden.

5.2.5 Flash-Speicher

Die beiden jeweils $4\,\mathrm{MByte}$ großen Flash-Speicher sind an einen gemeinsamen SPI-Bus angeschlossen. Der erste Flash dient zum Speichern der auszugebenden Spannungswerte, der zweite für die Speicherung von gemessenen Strömen während der Spannungswiedergabe. Im ersten Speicher wird zudem der erste Sektor von $4\,\mathrm{kByte}$ für Konfigurationen reserviert. Hier werden zwei Structs abgespeichert, die die Ergebnisse der Kalibrierung und ein paar weitere Einstellungen enthalten.

Die Konfigurationen werden zusätzlich durch einen 32-Bit CRC gesichert. Stimmt der CRC beim Start nicht überein, werden die Standardeinstellungen wiederhergestellt und abgespeichert. Mit dem Befehl *config* ist es möglich, sich die aktuelle Konfiguration ausgeben zu lassen.

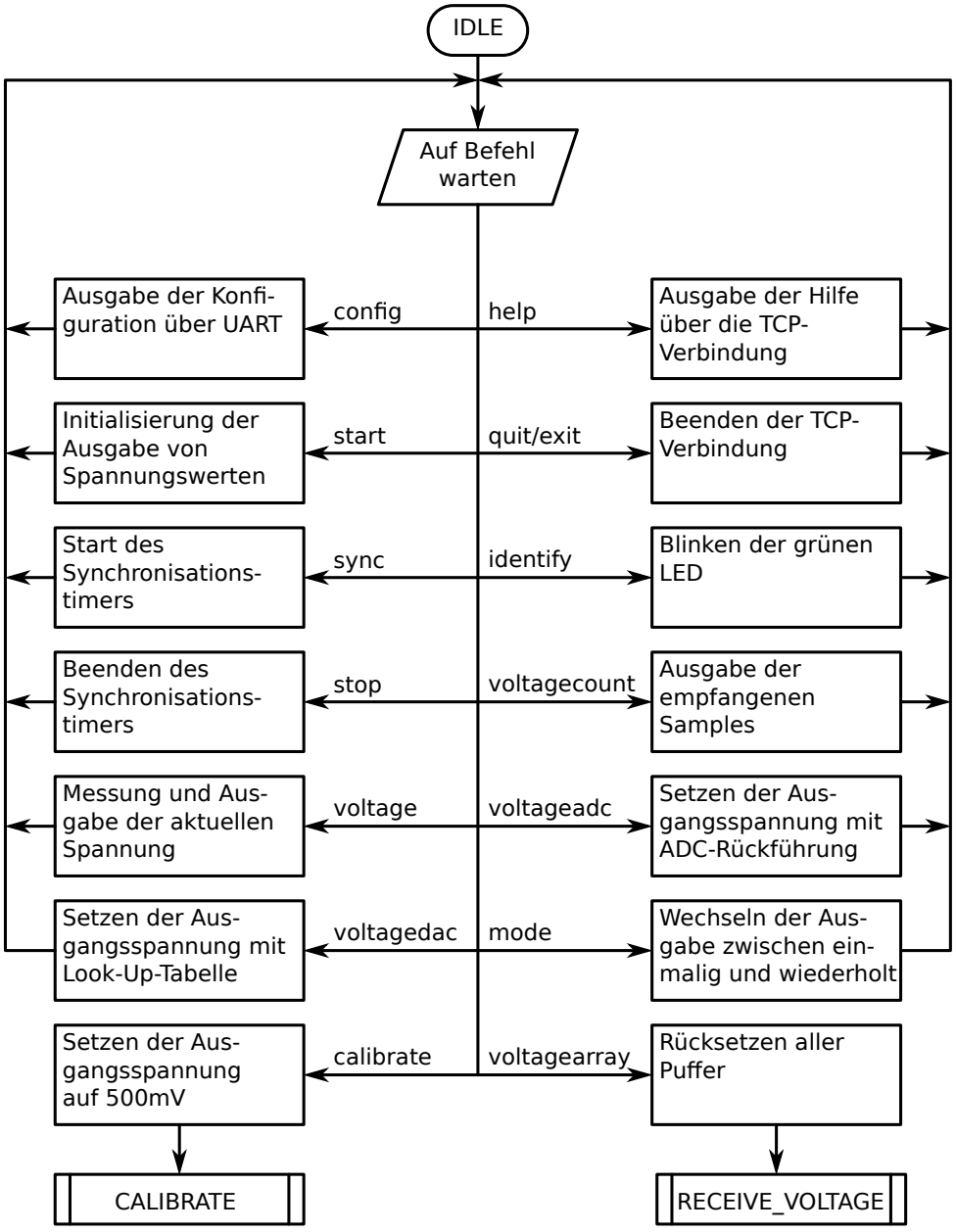


Abbildung 5.1: Flussdiagramm zur Auswertung der über TCP empfangenen Befehle

5.2.6 Kalibrierung

Mit dem Befehl *calibrate* ist es möglich, den externen AD-Umsetzer und die Strecke vom DA-Umsetzer bis zum Verstärker zu kalibrieren. Die Kalibrierung erfolgt dabei in zwei Schritten. Als erstes wird der AD-Umsetzer kalibriert, wobei hier eine externe Messung von Spannungen nötig ist, die dann an das Modul zurück gegeben werden. Im zweiten Schritt kalibriert der Zellspannungsgenerator die Strecke vom DA-Umsetzer bis zum Verstärker selbstständig.

Für die Kalibrierung des AD-Umsetzers werden am Ausgang zwei Spannungen ausgegeben, eine sehr geringe von $500\,\mathrm{mV}$ und eine hohe von $5{,}5\,\mathrm{V}$. Bei jeder Spannung muss eine externe Messung durchgeführt werden, die dann über die TCP-Verbindung an den Controller zurück gesendet wird. Der Mikrocontroller misst dann die Spannung mit dem AD-Umsetzer, die auf Grund der Toleranzen der Bauteile von der tatsächlichen Spannung abweicht. Sind dem Controller die beiden gemessenen und tatsächlichen Spannungen bekannt, kann er einen Offset- und Verstärkungsfehler berechnen. Diese beiden Werte werden in der Konfiguration abgespeichert und auf den Flash geschrieben.

Die Abfolge der Aktionen wird im Flussdiagramm in Abbildung 5.2 dargestellt. Dabei ist zu beachten, dass die beiden Punkte „Auf gemessene Spannung warten“ interruptbasiert sind und den Controller nicht blockieren.

Es ist nun möglich, mit einer Regelung die Ausgangsspannung unter Zuhilfenahme der Rückführung durch den externen AD-Umsetzer sehr genau einzustellen. Diese Regelung ist in dem Flussdiagramm 5.3 dargestellt. Zunächst wird der Wert für den DA-Umsetzer geschätzt. Durch eine Messung mit dem AD-Umsetzer kann eine Abweichung bestimmt werden, die dann in einen DA-Umsetzer-Wert umgerechnet und auf den vorhandenen Wert aufaddiert wird. Dies wird so lang wiederholt, bis die gemessene und die gewünschte Spannung identisch sind. Diese Funktion kann mit dem Befehl *voltagadc* auch einzeln ausgeführt werden.

Nach der Kalibrierung des AD-Umsetzers erfolgt die Erstellung einer Look-Up-Tabelle in der zu bestimmten Spannungen die passenden Werte des DA-Umsetzers eingetragen werden. Die Tabelle besteht aus den Spannungswerten $100\,\mathrm{mV}$, $500\,\mathrm{mV}$ und ab dort in $500\,\mathrm{mV}$-Schritten bis $6\,\mathrm{V}$. Jeder dieser Werte wird mit Hilfe der zuvor erläuterten Regelung am Ausgang angelegt und der dazugehörige Wert für den DA-Umsetzer in der Tabelle abgelegt. So ist es möglich, die Spannungen ohne eine Regelung genau zu setzen.

5.2.7 Spannungsprogrammierung

Für das Programmieren einer Spannungssequenz ist der Befehl *voltagearray* verantwortlich. Nach der Eingabe des Befehls wird zunächst der FIFO für die Spannungswerte initialisiert

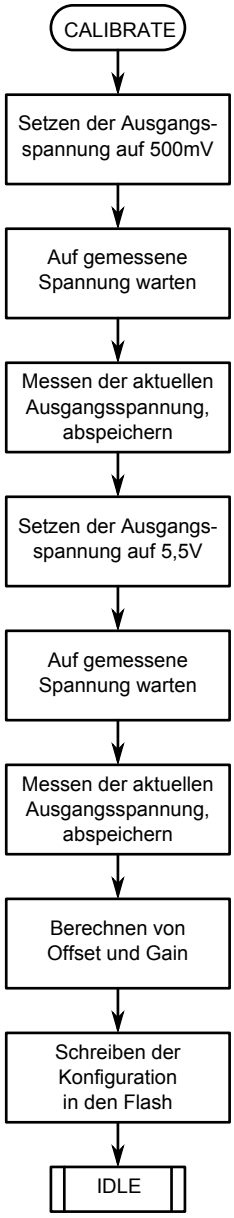


Abbildung 5.2: Vereinfachtes Flussdiagramm für die Kalibrierung von AD-Umsetzer und DA-Umsetzer bis Verstärker

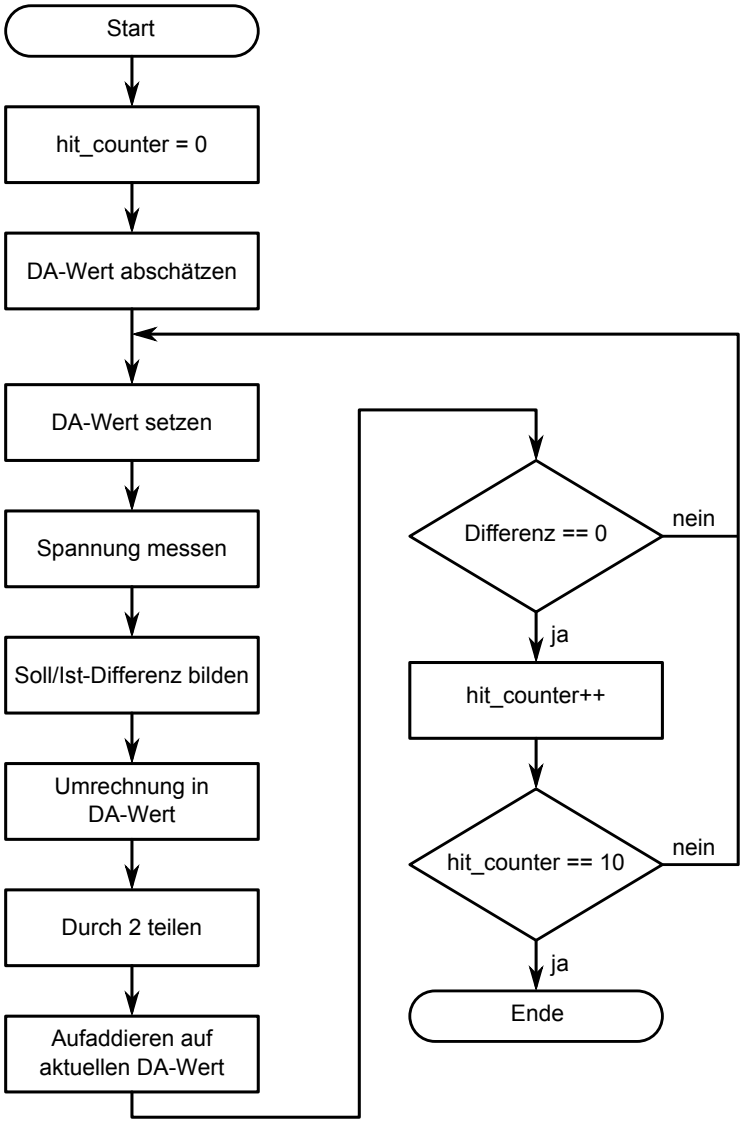


Abbildung 5.3: Flussdiagramm für das Einstellen der Ausgangsspannung mit Rückkopplung über den externen AD-Umsetzer

und der zweite Block des ersten Flash-Speichers gelöscht. Die Übertragung der Spannungswerte erfolgt über den TCP-Socket in ASCII-Kodierung. Die einzelnen Werte werden in μV übertragen und durch ein Semikolon getrennt. Da der FIFO für den TCP-Stream nur eine Größe von $5000\,\mathrm{Byte}$ hat, darf eine Zeile nicht mehr als $5000\,\mathrm{Byte}$ haben. Ist eine vollständige Zeile beim Mikrocontroller angekommen, werden die Spannungen dekodiert und mit Hilfe der zuvor durch eine Kalibrierung berechneten Offset- und Gain-Fehler in den entsprechenden Wert des DA-Umsetzers gewandelt. Die umgewandelten Samples werden dann zunächst in einem FIFO zwischengespeichert. Hat der FIFO eine Größe von $4\,\mathrm{kByte}$ erreicht, werden diese in einem Satz in den Flash geschrieben. Um die Übertragung zu beenden kann der Wert *EOF* übertragen werden. Es werden anschließend die übrigen Samples in den Flash geschrieben. Nach jeder gesendeten Zeile wird vom Mikrocontroller das Zeichen > übertragen, um der anderen Seite zu signalisieren, dass er bereit ist, neue Daten zu empfangen. Außerdem wird die Anzahl der erhaltenen Samples gezählt. Dieser Wert kann dann später mit dem Befehl *voltagecount* abgefragt werden, so ist es möglich, eine fehlerhafte Übertragung in vielen Fällen zu erkennen. Der gesamte Ablauf im Mikrocontroller für die Übertragung ist in Abbildung 5.4 in einem Flussdiagramm dargestellt.

5.2.8 Spannungswiedergabe

Für die Wiedergabe der zuvor programmierten Spannungen sind insgesamt drei Befehle nötig. Als erstes wird der Befehl *start* benötigt, welcher zunächst einen FIFO für die Wiedergabesamples initialisiert und bereits mit den ersten Werten aus dem Flash füllt. Zudem wird der erste Sample über den DA-Umsetzer ausgegeben. Dieser Befehl wird auf allen Modulen ausgeführt, die sich an der Wiedergabe beteiligen sollen. Anschließend wird auf einem Modul, welches mit dem Jumper JP2 als Master definiert ist, der Befehl *sync* ausgeführt. Dieser startet den Taktgenerator für die Synchronisationsleitung. Dieser Taktgenerator ist durch einen einfachen Timer realisiert, der einen Pin toggelt. Nachdem die Sequenz beendet ist oder der Nutzer die Sequenz vorzeitig abbrechen will, kann der Befehl *stop* an den Master gesendet werden, um die Synchronisationsleitung wieder abzustellen.

Es ist alternativ möglich nochmals den Befehl *start* zu senden, um die Sequenz erneut auszugeben. Dabei sind die einzelnen Module jedoch nicht synchronisiert, da der Befehl unter Umständen zu unterschiedlichen Zeitpunkten in den Modulen ausgeführt wird. Will man also die gleiche Sequenz nochmals auf allen Modulen synchron ausgeben, muss zuerst der *stop* Befehl ausgeführt werden, gefolgt von den Befehlen *start* und *sync*.

Das Nachladen der Samples aus dem Flash geschieht in einem eigenen Timer, da es wichtig ist, dass diese Aufgabe durch einen Interrupt ausgeführt wird, der eine höhere Priorität hat als der Ethernet-Interrupt, da es sonst zu Kollisionen beim Zugriff auf den Flash kommen kann. Ob Daten nachgeladen werden sollen oder nicht, wird über die globale Variable *reload_fill* gesteuert. Diese Variable wird nach dem Aufruf von dem Befehl *start* auf *1* gesetzt, um dem Timer mitzuteilen, dass er Daten nachladen soll. Ist das Ende der Sequenz

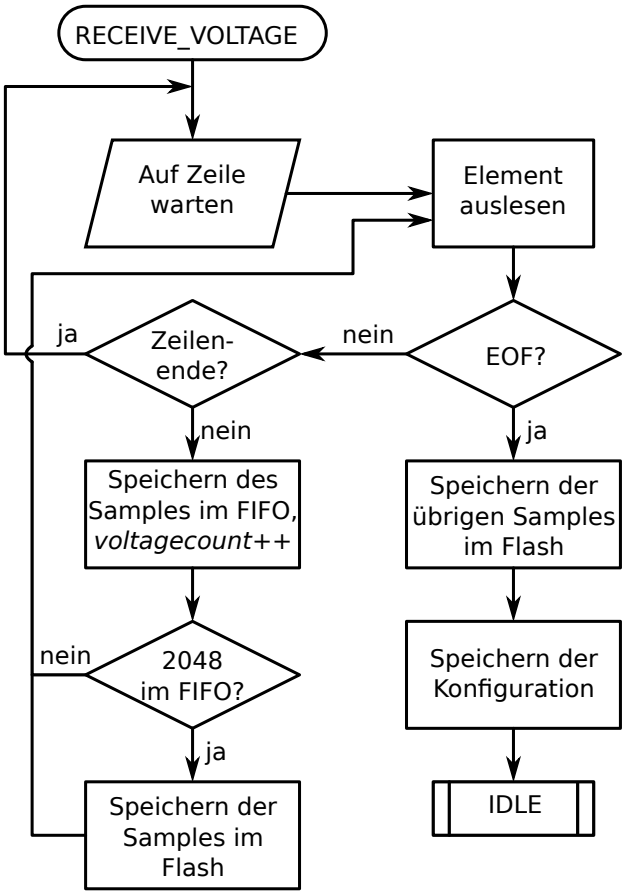


Abbildung 5.4: Flussdiagramm für die Übertragung von Spannungswerten an den Mikrocontroller

erreicht, setzt der Interrupt selbst diese Variable wieder auf 0, um ein weiteres Laden zu verhindern bzw. um mitzuteilen, dass die Wiedergabe abgeschlossen ist. Zudem kann dem Timer über die Konfigurationsvariable *repeat* mitgeteilt werden, ob er nach dem Erreichen des Endes wieder von vorne beginnen soll. Diese Variable kann mit dem Befehl *mode* und den Parametern *single* oder *free* gesetzt werden. Der Ablauf für das Laden der Samples ist im Flussdiagramm 5.5 dargestellt.

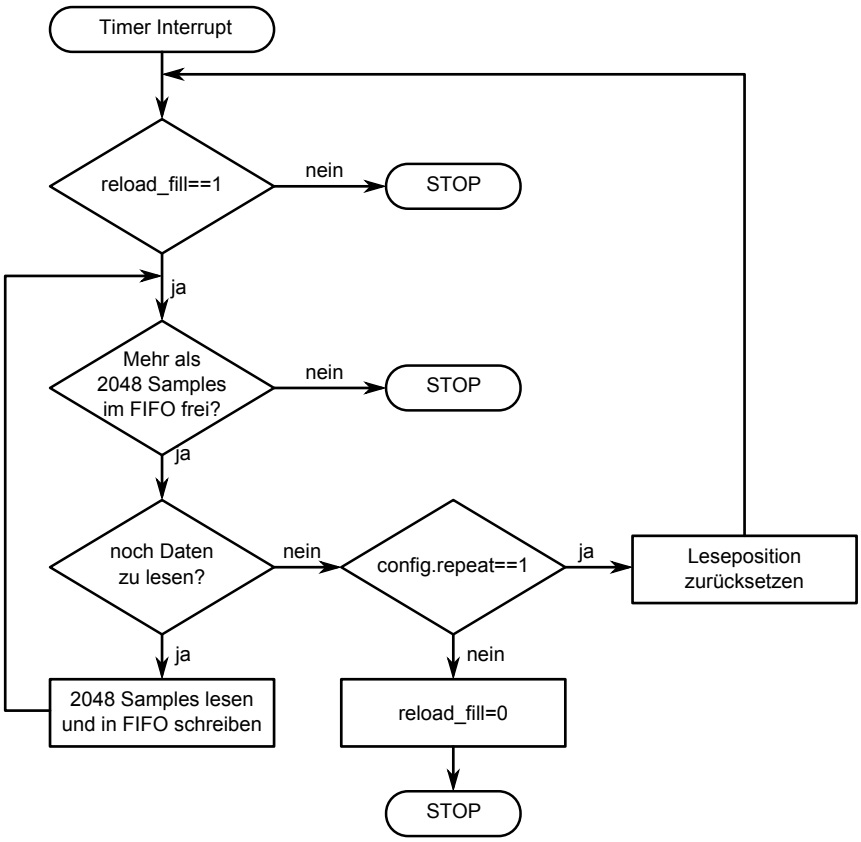


Abbildung 5.5: Flussdiagramm für die Transientenwiedergabe

5.2.9 Sonstige Befehle

Neben den bereits beschriebenen Befehlen gibt es noch ein paar weitere Befehle, die im Folgenden kurz beschrieben werden sollen.

Identifikation der Module

Bei der Methode nach Kapitel 5.2.2 zum Auffinden der Module über eine UDP-Broadcast Nachricht besteht das Problem, dass speziell bei der Verwendung von DHCP zum Zuweisen der IP-Adressen nicht klar ist, welches Modul welche IP-Adresse hat. Um die Module identifizieren zu können, gibt es den Befehl *identify*. Wird der Befehl ausgeführt, blinkt die grüne LED für einige Sekunden, wodurch die Identifikation eines Moduls möglich ist.

Ausgabe der Konfiguration

In einigen Fällen kann es nützlich sein, die aktuelle Konfiguration, sprich die Offset- und Gain-Werte für den AD-Umsetzer oder die Übersetzungstabelle für den DA-Umsetzer, auszulesen. Hierfür existiert der Befehl *config*, der die aktuelle Konfiguration über den TCP-Socket zurückgibt.

Ende der Verbindung

Mit Hilfe der Befehle *quit* oder *exit* kann die Verbindung beendet werden. Hierbei wird die Verbindung einfach seitens des Zellspannungsgenerators getrennt. Eine Beendigung durch den Client kann ebenfalls erfolgen, indem er die Verbindung trennt.

Abfrage der aktuellen Spannung

Der Befehl *voltage* führt eine Messung der aktuellen Ausgangsspannung mit dem externen AD-Umsetzer durch und gibt den gemessenen Wert in Mikrovolt über die TCP-Verbindung zurück.

Setzen der Ausgangsspannung

Neben der Ausgabe einer Spannung mit Rückkopplung über den externen AD-Umsetzer existiert auch noch ein Befehl zum Setzen der Ausgangsspannung über die Look-Up-Table. Der Befehl lautet *voltagedac* und setzt sich primär aus zwei Funktionen zusammen. Als Erstes wird der übermittelte Spannungswert über die Look-Up-Tabelle in den passenden Wert für den DA-Umsetzer umgewandelt. Mit einer zweiten Funktion wird der umgerechnete Wert in den DA-Umsetzer geschrieben.

5.3 Software auf dem PC

Um mit der implementierten Schnittstelle seitens des Zellspannungsgenerators bequem umgehen zu können, wird ein API (Application Programming Interface) in MATLAB implementiert. Das Ziel ist es, in MATLAB synthetisierte oder zuvor mit einem Oszilloskop aufgenommene Spannungsverläufe einfach an die Module senden zu können.

5.3.1 Suchen von Generatoren im Netzwerk

Um die Generatoren im Netzwerk finden zu können, muss die aus Kapitel 5.2.2 bekannte Methode mit der UDP Broadcast Nachricht angewendet werden. Es wird dabei ein UDP Paket an den Broadcast gesendet, worauf alle Module antworten. Da keine Möglichkeit gefunden werden konnte in MATLAB den Absender eines UDP-Paketes zu ermitteln, senden die Module ihre eigene IP-Adresse in den Daten des Paketes. Die Implementierung der Funktion zum Auffinden der Module heißt *gen_search()* und ihr wird die IP-Adresse des Broadcasts übergeben, welcher für gewöhnlich *255.255.255.255* ist. Die Funktion gibt dann eine Zelle zurück, die die erhaltenen IP-Adressen als Strings beinhaltet.

5.3.2 Verbindungsaufbau zu den Generatoren

Nachdem die Generatoren gefunden sind oder ihrer Adressen manuell definiert wurden, kann eine Verbindung aufgebaut werden. Dies geschieht über die Funktion *gen_connect()*, welche als Parameter die zuvor mit *gen_find()* erzeugte Zelle der IP-Adressen bekommt. In der Funktion wird eine TCP-Verbindung zu jeder der Adressen auf dem Port 56936 hergestellt. Es werden außerdem noch ein paar Einstellungen an dem TCP-Socket vorgenommen. Hierzu zählt zum Beispiel das Einstellen der Zeitüberschreitung für die Funktion *fread()*. Da diese Funktion blockend ist, kann so das Abbrechen des Funktionsaufrufs nach einer bestimmten Zeit erzwungen werden. Dies ist insbesondere in einem Fehlerfall seitens des Netzwerks oder eines Zellspannungsgenerators wichtig, damit das Programm nicht hängen bleibt.

5.3.3 Senden von einer Spannungssequenz

Mit der Funktion *gen_send_voltages()* ist es möglich, eine Spannungssequenz an die Generatoren zu senden. Die Sequenz kann zuvor mit einem Oszilloskop aufgenommen oder mit MATLAB synthetisch hergestellt sein. Als Parameter bekommt die Funktion als Erstes den Vektor mit den TCP-Sockets, die zuvor mit *gen_connect()* erstellt wurden. Der zweite Parameter beinhaltet die Spannungssequenz, die den Generatoren übermittelt werden soll. Dabei

kann ein einfacher Zeilenvektor übergeben werden, der an alle Generatoren gesendet wird, oder eine Matrize bestehend aus mehreren Zeilenvektoren für jeden einzelnen Generator. Es muss also ein Zeilenvektor oder eine Matrize mit einer Zeilenanzahl gleich der Anzahl der Generatoren übergeben werden.

In der Funktion werden die Spannungen jeweils in Blöcke zusammengefasst und in ASCII umkodiert. Die Anzahl der Samples pro Block und gesendetem Paket beträgt 120, da so die maximale Durchsatzrate erreicht wird. Nachdem der Block, beziehungsweise die Blöcke bei verschiedenen Sequenzen, umkodiert sind, werden diese an alle Generatoren verschickt. Erst nachdem alle Blöcke verschickt sind, wird auf die Bestätigung der einzelnen Module gewartet, was der effektiven Übertragungsgeschwindigkeit zugute kommt. Sind alle Samples übertragen, wird die Übertragung mit dem Befehl EOF beendet. Anschließend wird mit dem Befehl *voltagecount* die Anzahl der am Generator eingetroffenen Samples überprüft, um Fehler in der Übertragung zu erkennen. Da die Generatoren Spannungswerte unterhalb von $100\,\mathrm{mV}$ ignorieren, ist so auch eine Prüfung des Datensatz auf seine logische Korrektheit möglich. Dieser Ablauf ist nochmals in den Flussdiagramm 5.6 veranschaulicht.

5.3.4 Weitere Funktionen

Neben den oben erläuterten Funktionen sind noch einige weitere Funktionen in MATLAB implementiert, die den Umgang mit den Generatoren erleichtern sollen und teilweise auch innerhalb der API verwendet werden. Diese zusätzlichen Funktionen werden im folgenden kurz erläutert.

Identifizierung der Module

Mit dem Befehl *identify* ist es möglich, die Module durch das Blinken der grünen LED zu identifizieren. Dieser Befehl wird mit der MATLAB-Funktion *gen_identify()* umgesetzt, welche einen einzelnen TCP-Socket als Parameter bekommt.

Senden eines Befehls

Zum Senden eines einzelnen Befehls an einen oder mehrere Generatoren ist die Funktion *gen_cmd()* verantwortlich. Sie sendet den Befehl an die im ersten Parameter übergebenen TCP-Sockets und wartet anschließend auf eine Bestätigung der Module. Erhält die Funktion eine Bestätigung durch alle Module gibt sie 0 zurück, ansonsten 1.

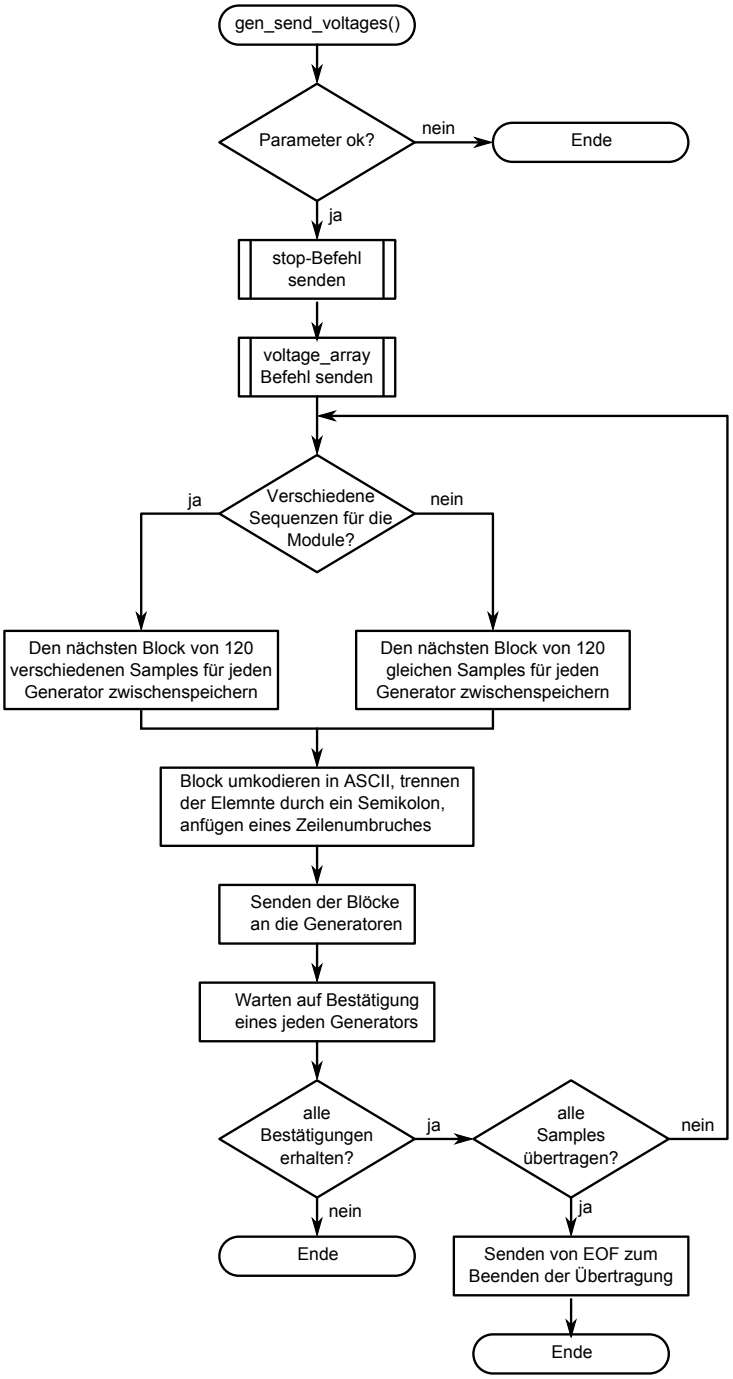


Abbildung 5.6: Flussdiagramm für die Beschreibung des Sendevorgangs von einer Spannungssequenz in MATLAB

Umschalten zwischen einfacher und wiederholter Ausgabe

In der Software für den Mikrocontroller ist mit dem Befehl *mode* die Möglichkeit vorgesehen, die Wiedergabe der Spannungssequenzen zwischen einfacher Ausgabe und wiederholter Ausgabe umzuschalten. Der Befehl wird mit den MATLAB-Funktionen *gen_repeat()* beziehungsweise *gen_single()* implementiert. Als Parameter bekommen die Funktionen jeweils eine Liste von TCP-Sockets.

Kontrolle der Ausgabe

Für die Kontrolle der Ausgabe sind drei Befehle verantwortlich, *start* zum Initialisieren der Widergabe, *sync* zur Generierung des Synchronisationstaktes auf dem Master Modul und *stop* zum Beenden der Synchronisation. Die Befehle sind mit den MATLAB-Funktionen *gen_start()*, *gen_sync()* und *gen_stop()* ausführbar, wobei *gen_start()* und *gen_stop()* jeweils eine Liste von TCP-Sockets als einzigen Parameter übergeben bekommen. Die Funktion *gen_sync()* bekommt hingegen nur einen Socket, da nur eines der Module als Master fungieren darf.

Beenden der Verbindung

Um die Verbindung nach der Ausführung von Befehlen wieder zu beenden, kann die Funktion *gen_disconnect()* verwendet werden. Es wird dabei die TCP-Verbindung mit *fclose()* abgebaut.

6 Erprobung und Messungen

In den vorherigen Kapiteln wurde das Konzept, die Hardware und die Software für die Zellspannungsgeneratoren besprochen. Es sollen nun mit der aufgebauten Hardware Untersuchungen durchgeführt werden, die die Funktion der Zellspannungsgeneratoren belegen. Es werden zwei Arten von Messungen durchgeführt. Zunächst soll die Genauigkeit des Generators für die Kalibrierung untersucht werden. Hier ist wichtig, dass die Spannung der Generatoren sehr genau eingestellt werden kann und auch bei einer Belastung konstant bleibt. Die zweite Art von Messungen bezieht sich auf die Wiedergabe von Spannungssequenzen. Hier ist ebenfalls wichtig, dass sich die Abweichungen vom Originalsignal in Grenzen halten. Bevor genauer auf die Messungen eingegangen wird, wird noch ein Problem mit einem der Step-Down-Regler genauer betrachtet.

6.1 Fehlverhalten einer der Step-Down-Regler

Bei der Inbetriebnahme kam es häufiger vor, dass der Step-Down-Regler, der vor dem Linearregler für den Verstärker sitzt (IC2), nicht korrekt funktioniert. Dies äußert sich dadurch, dass der Regler alle $20\,\mathrm{ms}$ versucht einzuschalten, jedoch jedes mal nach kurzer Zeit abbricht. Entfernt man große Kapazitäten hinter dem Regler, läuft dieser einwandfrei an. Es handelt sich bei dem Verhalten des Reglers vermutlich um einen Schutzmechanismus, der greift, wenn ein Überstrom erkannt wird. Die Schwelle hierfür liegt laut dem Datenblatt bei $1{,}5\,\mathrm{A}$ [18]. Der Regler versucht beim Start die Ausgangsspannung innerhalb von $8\,\mathrm{ms}$ in einer Rampe zu erhöhen [18]. Am Ausgang befinden sich hinter der Spule des Reglers $150\,\mu\mathrm{F}$ und hinter dem Linearregler nochmal $527\,\mu\mathrm{F}$. Somit wird der Step-Down-Regler im schlimmsten Fall mit $677\,\mu\mathrm{F}$ belastet. Mit der folgenden Formel lässt sich dann der Maximalstrom errechnen, der im Einschaltmoment fließt:

$$I = C \cdot \frac{du}{dt} = 677\,\mu\mathrm{F} \cdot \frac{11\,\mathrm{V}}{8\,\mathrm{ms}} \approx 931\,\mathrm{mA} \tag{6.1}$$

Dieser Wert ist zwar noch unter dem Maximalstrom von $1{,}5\,\mathrm{A}$, jedoch handelt es sich hier um einen Schaltregler, der am Ausgang Impulse erzeugt. Es kann also passieren, dass in einem Schaltmoment ein höherer Strom fließt, der dann hoch genug ist, um den Überstromschutz zu aktivieren.

Um dieses Problem zu lösen wird ein anderer Regler verbaut. Der TPS5420 ist bezüglich der Anschlüsse und der Beschaltung mit dem TPS5410 kompatibel, liefert aber bis zu $2\,\mathrm{A}$ Strom und schaltet erst bei $3\,\mathrm{A}$ ab [18], [19]. Somit ist es möglich, deutlich größere kapazitive Lasten an den Regler anzuschließen, ohne, dass dieser wegen Überstrom abschaltet.

6.2 Kalibrierung von Generatoren

Um die Sensoren zu kalibrieren, ist eine möglichst präzise Spannung nötig, deren Genauigkeit laut Kapitel 2.1.3 im Bereich von $1{,}22\,\mathrm{mV}$ liegen sollte. Da diese Genauigkeit durch eine Kalibrierung der Generatoren erreicht wird, kann sie natürlich nur so genau sein, wie das externe Werkzeug zur Kalibrierung. Hierfür kommt das Tischmultimeter PM 2519 von Philips zum Einsatz (Inventarnummer: 305). Laut dem Datenblatt [11] besitzt dieses Multimeter im Bereich bis $1\,\mathrm{V}$ eine Genauigkeit von

$$U_{tol,1V} = U_{mess} \cdot 0{,}1\,\% + 0{,}2\,\mathrm{mV} \tag{6.2}$$

und im Bereich bis $10\,\mathrm{V}$ eine Genauigkeit von

$$U_{tol,10V} = U_{mess} \cdot 0{,}1\,\% + 2\,\mathrm{mV} \tag{6.3}$$

Dabei ist U_{mess} der auf dem Messgerät abgelesene Wert. Somit liegt bei einer Eingangsspannung von $U_{mess} = 5{,}5\,\mathrm{V}$ die Toleranz bei $\Delta U_{5,5V} = 7{,}5\,\mathrm{mV}$, was deutlich außerhalb der ursprünglich geforderten $1{,}22\,\mathrm{mV}$ ist. Für eine Eingangsspannung von $U_{mess} = 1\,\mathrm{V}$ sieht es schon besser aus, hier liegt die Toleranz bei $\Delta U_{1V} = 1{,}2\,\mathrm{mV}$ und somit knapp innerhalb der Anforderung. Somit kann mit diesem Multimeter bei Spannungen über $1\,\mathrm{V}$ zwar nicht die geforderte Genauigkeit erreicht werden, die Auflösung ist mit $1\,\mathrm{mV}$ jedoch ausreichend.

6.2.1 Linearität des Generators

Bei der Kalibrierung wird der Offset- und Verstärkungsfehler korrigiert. Das heißt, die Kennlinie wird anhand zweier Punkte ermittelt. Es soll hier gezeigt werden, dass diese Art der Kalibrierung ausreichend ist. Hierfür wird zunächst der Offset- und Verstärkungsfehler mit dem Tischmultimeter ausgeglichen. Anschließend werden Spannungen über den gesamten Bereich von $500\,\mathrm{mV}$ bis $5{,}5\,\mathrm{V}$ mit Rückkopplung über den AD-Umsetzer eingestellt und mit dem Tischmultimeter gemessen. Es wird dann die Differenz zwischen gemessener und idealer Kennlinie bestimmt. Die Messung erfolgt in $250\,\mathrm{mV}$-Schritten. Das Ergebnis ist in Abbildung 6.1 zu sehen, es ist dabei jedoch zu beachten, dass bei den Werten über $1\,\mathrm{V}$ die Auflösung des Messgerätes nur noch $1\,\mathrm{mV}$ beträgt und die Genauigkeit noch deutlich schlechter ist. Man kann also bei Abweichungen von maximal $2\,\mathrm{mV}$ sagen, dass die

Kalibrierung durch zwei Punkte der Kennlinie ausreichend ist, da keine nennenswerten Abweichungen von der Kennlinie mit Hilfe der zur Verfügung stehenden Messinstrumente gemessen werden können.

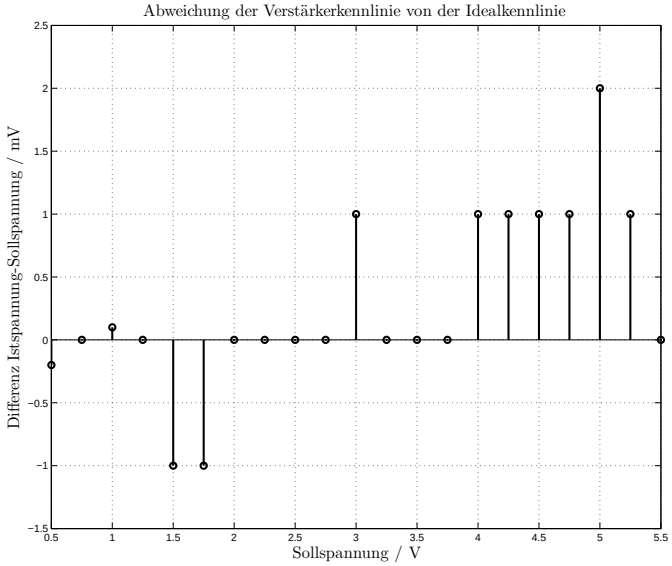


Abbildung 6.1: Abweichung der Ausgangskennlinie von der idealen Kennlinie des AD-Umsetzers nach Kalibrierung des Generators, über $1\,\mathrm{V}$ starke Quantisierung durch Auflösung von $1\,\mathrm{mV}$ des Tischmultimeters

6.2.2 Abhängigkeit der Spannung von einer Last

Der zweite wichtige Punkt neben der Linearität ist die Beeinflussung der Kennlinie durch eine Last. Hierfür wird eine Ausgangsspannung von $1\,\mathrm{V}$ ausgegeben und dann mit $10\,\Omega$ belastet, so dass $100\,\mathrm{mA}$ fließen. Es ist zu beobachten, dass beim Zuschalten der Last die Spannung von $999{,}7\,\mathrm{mV}$ auf $994{,}7\,\mathrm{mV}$ einbricht. Dies ist eine Differenz von $5\,\mathrm{mV}$. Des weiteren ist zu beobachten, dass die Spannung bei höheren Ausgangsspannungen stärker einbricht. Bei $2\,\mathrm{V}$ am Ausgang sind es bereits $10\,\mathrm{mV}$ und für eine Ausgangsspannung von $3\,\mathrm{V}$ bricht sie bei einer $10\,\Omega$-Belastung um $15\,\mathrm{mV}$ ein. Der Wert der Differenz scheint in etwa proportional zur Ausgangsspannung beziehungsweise in diesem Fall auch zum Ausgangsstrom zu sein. Bei diesen Messungen ist zu beachten, dass die Rückkopplung des Verstärkers auf dem Verstärker direkt passiert. Also der Widerstand *R3* ist bestückt, *R6*, *R7* und *R9* nicht (siehe Anhang B.1.4). Da der Spannungsabfall proportional zum Strom ist, ist eine naheliegende Vermutung, dass die Spannung über die Kabel und Leitungen vom Verstärker bis zur Bananenbuchse abfällt. Dies kann man umgehen, indem man die Rückkopplung für den Verstärker ebenfalls an die Bananenbuchsen führt. Hierfür müssen *R6*,

R7 und *R9* bestückt und *R3* entfernt werden. Einige Messungen ergeben, dass das Problem dadurch tatsächlich erheblich reduziert werden kann. Für eine Spannung von $1\,\mathrm{V}$ bricht der Ausgang bei einer Belastung mit $10\,\Omega$ nur noch um $0{,}7\,\mathrm{mV}$ statt $5\,\mathrm{mV}$ ein. Bei höheren Spannung können ähnliche Verbesserungen beobachtet werden.

6.2.3 Abhängigkeit der Spannung von der Temperatur

Ein weiteres Verhalten, das untersucht werden soll, ist die Temperaturabhängigkeit des Generators. Hierfür wird eine einfache Messung durchgeführt, bei welcher die Ausgangsspannung fest eingestellt und über einen längeren Zeitraum beobachtet wird. Das Multimeter zur Messung der Spannung am Ausgang ist mehrere Stunden warmgelaufen, der Zellspannungsgenerator wurde erst zu Beginn der Messung eingeschaltet. Es wird eine Ausgangsspannung von etwa $1\,\mathrm{V}$ über einen längeren Zeitraum beobachtet. Nach etwa einer Stunde ist die Spannung um $0{,}5\,\mathrm{mV}$ geringer als beim Start des Versuchs. Dieser Versuch wird bei Raumtemperatur durchgeführt, ohne, dass das Modul in einem Gehäuse verbaut ist. Es ist also damit zu rechnen, dass die Temperaturabweichungen noch größer werden, wenn das Modul in einem geschlossenen Gehäuse verbaut ist. Folglich müssen die Module eine Vorlaufzeit von ein bis zwei Stunden haben. Erst dann sollte mit den Modulen gearbeitet werden, wobei natürlich nach dem Erwärmen und vor der Benutzung eine Kalibrierung durchgeführt werden sollte.

6.3 Wiedergabe von Spannungssequenzen

Nachdem gezeigt wurde, dass die Wiedergabe von stationären Spannungen mit ausreichender Genauigkeit möglich ist, soll nun das Verhalten bei der Wiedergabe von transienten Spannungen untersucht werden. Zunächst soll der Amplitudengang des Generators bestimmt werden. Anschließend werden Messungen mit dem bereits im Kapitel 2.3 vorgestellten Spannungsverlauf des Startvorgangs eines Mercedes Benz Vito L, 4 Zylinder Diesel, 95kW, 2,0l durchgeführt [13]. Dabei wird das auf $2{,}5\,\mathrm{kHz}$ tiefpassgefilterte Signal, welches im Kapitel 2.3 erzeugt wurde, wiedergegeben. Das Signal am Ausgang des Generators wird mit dem Oszilloskop MSO 3034 von Tektronix aufgenommen, um es anschließend mit dem Original vergleichen zu können.

6.3.1 Messung des Amplitudengangs

In der Konzeption ist festgelegt, dass der analoge Teil der Schaltung Tiefpassverhalten mit einer Grenzfrequenz von $2{,}5\,\mathrm{kHz}$ haben soll. Um dies mit einer Messung zu bestätigen und

den Einfluss parasitärer Effekte zu beurteilen, wird der Amplitudengang des Systems aufgenommen. Hierfür wird eine Cosinus-Schwingung mit einer Amplitude von $2{,}5\,\mathrm{V}$ und einem Offset von $3\,\mathrm{V}$ und variabler Frequenz in den Zellspannungsgenerator übertragen und wiedergegeben. Der Wechselanteil der Ausgangsspannung wird mit einem FLUKE 45 Tischmultimeter gemessen, welches über eine serielle Schnittstelle an den PC angebunden ist. Die Schwingung wird mit einer Frequenz von $10\,\mathrm{Hz}$ bis $10\,\mathrm{kHz}$ erzeugt und logarithmisch skaliert.

Die gesamte Messung ist in dem MATLAB-Skript aus Anhang A.3.1.3 vollständig automatisiert. Es wird dafür ein Vektor mit den auszugebenden Frequenzen erzeugt, die anschließend einzeln an den Generator übertragen und wiedergegeben werden. Nach dem Start der Wiedergabe wird jeweils für einige Sekunden gewartet und anschließend eine Messung mit dem Tischmultimeter durchgeführt, die in einem zweiten Vektor abgelegt wird. Sowohl der Frequenzvektor als auch der Vektor mit den gemessenen Amplituden wird in einer Datei zwischengespeichert und anschließend mit dem Skript aus Anhang A.3.1.4 ausgewertet. Mit dieser Methode ist es nicht möglich, den Phasengang des Systems zu bestimmen, es wird ausschließlich der Amplitudengang betrachtet.

Das Ergebnis dieser Messung ist in Abbildung 6.2 dargestellt. Die gemessene Grenzfrequenz liegt bei etwa $2{,}24\,\mathrm{kHz}$. Um den Fehler zu bestimmen muss zunächst die Grenzfrequenz des Tiefpasses bestimmt werden, denn hier konnten nicht der exakt passende Kondensator von $4{,}1\,\mathrm{nF}$ verbaut werden, sondern eine Parallelschaltung, die eine Gesamtkapazität von $4{,}12\,\mathrm{nF}$ hat. Somit ergibt sich nach Formel 4.9 eine Grenzfrequenz von:

$$f_g = \frac{\sqrt{\sqrt{2}-1}}{2 \cdot \pi \cdot C \cdot R} = \frac{\sqrt{\sqrt{2}-1}}{2 \cdot \pi \cdot 4{,}12\,\mathrm{nF} \cdot 10\,\mathrm{k\Omega}} \approx 2{,}458\,\mathrm{kHz} \tag{6.4}$$

Die Abweichung beträgt somit $8{,}87\,\%$. Ein Großteil der Abweichungen lässt sich durch die Toleranz der Bauteile begründen. Die Widerstände haben jeweils eine Toleranz von $1\,\%$, bei den Kondensatoren sind es sogar $10\,\%$. Bezieht man diese Toleranzen mit in die vorherige Formel ein, ergibt sich eine Untergrenze der Grenzfrequenz zu:

$$f_{g,min} = \frac{\sqrt{\sqrt{2}-1}}{2 \cdot \pi \cdot 4{,}12\,\mathrm{nF} \cdot (1+0{,}1) \cdot 10\,\mathrm{k\Omega} \cdot (1+0{,}01)} \approx 2{,}238\,\mathrm{kHz} \tag{6.5}$$

Die gemessene Grenzfrequenz liegt somit knapp innerhalb der Toleranz der Bauteile. Dabei ist zu beachten, dass einige Toleranzen noch nicht berücksichtigt wurden. Hierzu gehört der Verstärkungsfehler durch Widerstände, Offset-Fehler der Verstärker und Messtoleranzen in dem Tischmultimeter. Ein weiterer Einfluss der bedacht werden muss, ist jener der Dämpfung durch die si-Funktion des DA-Umsetzers. Dieser liegt jedoch nach dem Skript aus Anhang A.3.2.7 lediglich bei $0{,}22\,\mathrm{dB}$ und kann somit vernachlässigt werden.

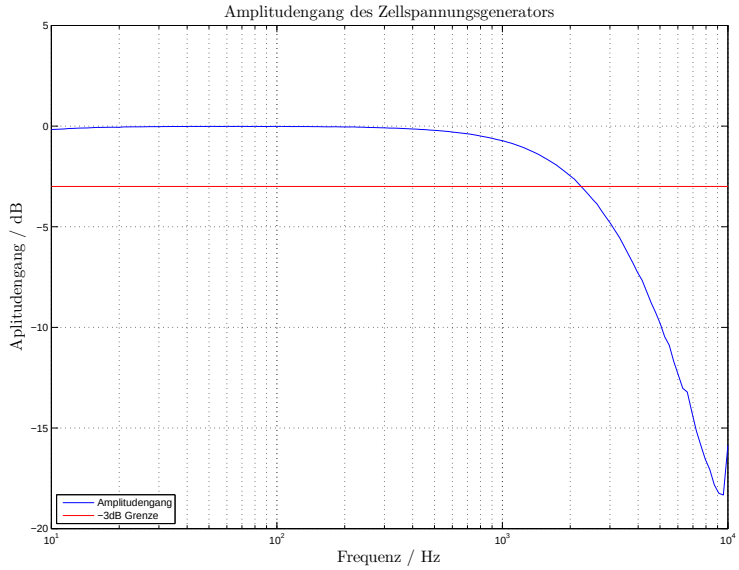


Abbildung 6.2: Gemessener Amplitudengang eines Zellspannungsgenerators mit eingezeichneter $-3\,\mathrm{dB}$ Grenze. Der Anstieg bei $f = 10\,\mathrm{kHz}$ kommt durch das nicht eingehaltene Abtasttheorem ($f_{max} < f_A/2$)

6.3.2 Simulation des Startvorgangs eines PKW

Um nun eine genauere Aussage über die Transientenwiedergabe der Generatoren zu fällen, wird der Startvorgang eines PKW wiedergegeben und anschließend mit dem Originalsignal verglichen. In den Zellspannungsgenerator wird das tiefpassbegrenzte und mit $20\,\mathrm{kHz}$ abgetastete Signal gegeben. Das am Ausgang aufgenommen Signal wird jedoch mit dem nicht gefilterten Originalsignal verglichen. Die Abweichungen, die durch die begrenzte Bandbreite entstehen, werden also mit berücksichtigt. In Abbildung 6.3 ist das ungefilterte Originalsignal aus Kapitel 2.3 in rot nochmals dargestellt, das blaue Signal ist das mit dem Oszilloskop am Generator gemessene Signal. Bereits hier lässt sich erkennen, dass eine Abweichung zwischen den beiden Signalen, vor allem am Anfang und am Ende, besteht. Das gemessene Signal scheint einen Offset zu haben. Dies bestätigt ebenfalls die Differenz der beiden Signale, die in Abbildung 6.4 zu sehen ist. Nach dem MATLAB Skript aus Anhang A.3.1.9, welches ebenfalls diese Grafiken erstellt hat, beträgt die Differenz im Mittel $2{,}71\,\mathrm{mV}$. Betrachtet man die statistische Verteilung der gemessenen Differenzen in Abbildung 6.5, ist der Offset ebenfalls zu erkennen. Die meisten Abweichungen liegen zwischen $-5\,\mathrm{mV}$ und $10\,\mathrm{mV}$. Das Ergebnis ist also deutlich schlechter als zunächst durch die Simulation aus Kapitel 2.3 angenommen, wo sich eine mittlere Streuung von $-3\,\mathrm{mV}$ bis $2\,\mathrm{mV}$ ergeben hatte. Dies hat mehrere Gründe, zum einen die bereits erwähnte Abweichung durch die Messinstrumente, wobei hier primär das Oszilloskop im Verdacht steht den Großteil des

Fehlers zu verursachen. Zum anderen spielt ein weiterer Effekt mit in die Messung ein, der bislang nicht betrachtet wurde. Betrachtet man den Anfang und das Ende des originalen und des gemessenen Signals, so kann man erkennen, dass die beiden Signale nicht genau übereinander liegen, wie in Abbildung 6.6 zu sehen ist. Am Anfang sind sie noch in etwa synchron, da die Signale am Anfang händisch im MATLAB Skript synchronisiert wurden, zum Ende hin eilt die gemessene Spannung dem Original jedoch voraus. Diese Diskrepanz liegt an den beiden Taktgebern des Mikrokontrollers und des Oszilloskops, die nicht synchron laufen. Die Aussagen die aus dem Vergleich der beiden Spannungen zuvor gezogen wurden sind also nur bedingt korrekt. Vermutlich ist die Abweichung der beiden Signale in Wirklichkeit deutlich geringer. Auch die Messdiffernz zwischen Oszilloskop und Tischmultimeter kann für einen großen Fehler sorgen.

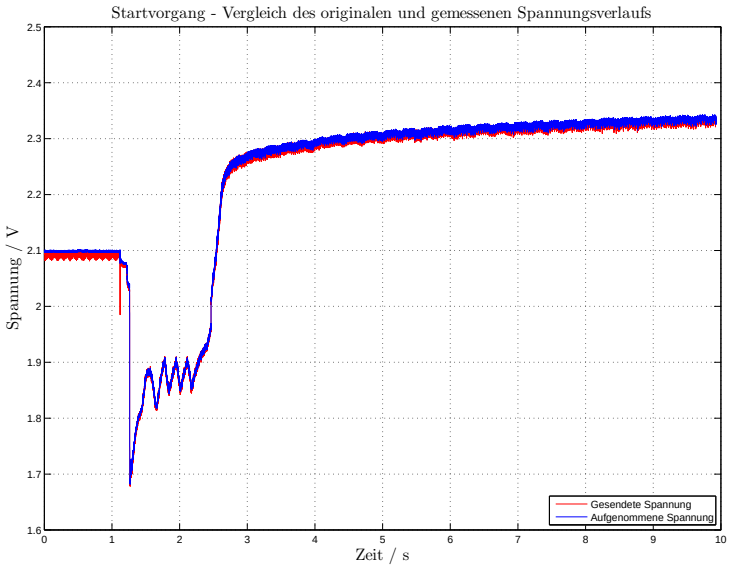


Abbildung 6.3: Vergleich des unveränderten und des am Generatorausgang gemessenen Signals des Startvorgangs eines PKW, bandbegrenzt auf $2{,}5\,\mathrm{kHz}$ mit $20\,\mathrm{kSPS}$ wiedergegeben

Es kann also gesagt werden, dass die meisten Samples bei der Wiedergabe einer Spannungssequenz eine Abweichung zwischen $-5\,\mathrm{mV}$ und $10\,\mathrm{mV}$ ausweisen, während in der mathematischen Annäherung aus Kapitel 2.3 ein Abweichung von $-3\,\mathrm{mV}$ bis $2\,\mathrm{mV}$ prognostiziert wurde. Vernachlässigt man den Gleichanteil dieser beiden Abweichungen, so ist die Abweichung in der Realität etwa um den Faktor 3 größer als die theoretische. Bedenkt man, dass bei der theoretischen Betrachtung ein idealer Filter eingesetzt wurde, während in der Realität nur ein Tiefpass 2. Ordnung verwendet wird und die Messtoleranzen des Oszilloskops, so ist dieses Ergebnis zufriedenstellend.

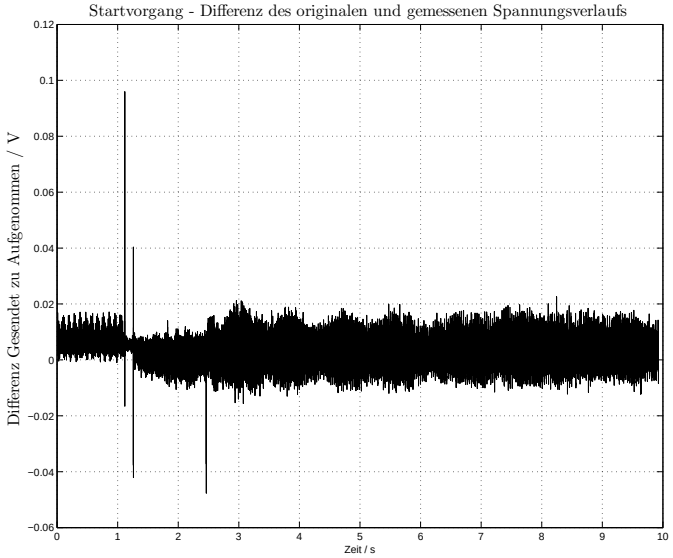


Abbildung 6.4: Differenz des originalen Signals zu dem gemessenen Signal des Startvorgangs eines PKW, bandbegrenzt auf $2{,}5\,\mathrm{kHz}$ mit $20\,\mathrm{kSPS}$ wiedergegeben

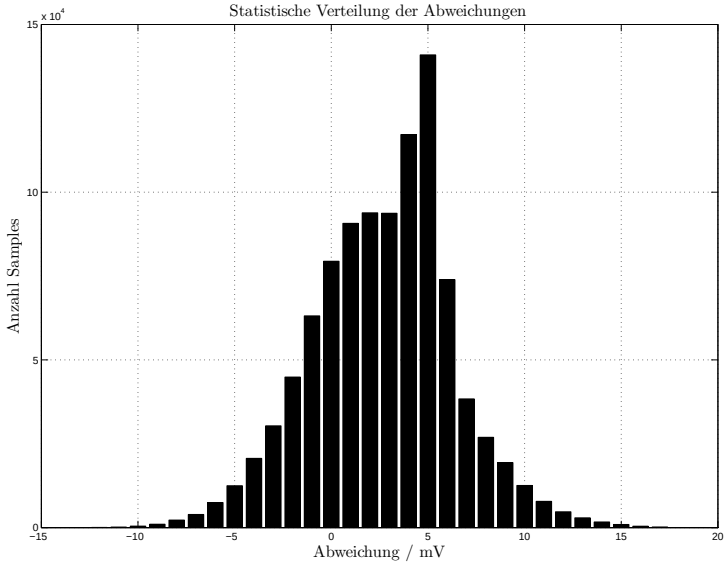


Abbildung 6.5: Statistische Verteilung der Differenz des originalen Signals zu dem gemessenen Signal des Startvorgangs eines PKW, bandbegrenzt auf $2{,}5\,\mathrm{kHz}$ mit $20\,\mathrm{kSPS}$ wiedergegeben

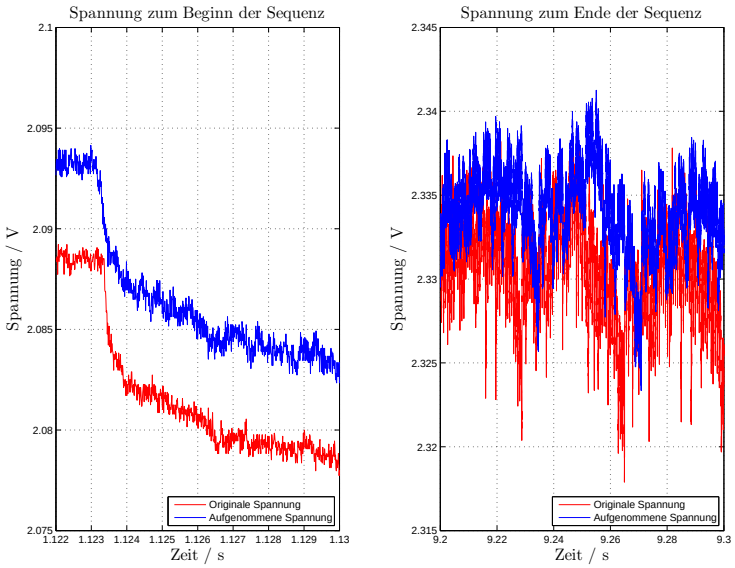


Abbildung 6.6: Vergleich des originalen Signals zu dem gemessenen Signal des Startvorgangs eines PKW am Anfang und am Ende, bandbegrenzt auf $2{,}5\,\mathrm{kHz}$ mit $20\,\mathrm{kSPS}$ wiedergegeben

6.3.3 Messungenauigkeiten des Oszilloskops

Die Differenz zwischen Oszilloskop und Tischmultimeter, die in der vorherigen Abbildung 6.4 zu erkennen ist, soll genauer untersucht werden. Es handelt sich dabei um eine Abweichung auf Grund von Messtoleranzen zwischen dem Oszilloskop und dem Tischmultimeter. Um diese Abweichung genauer zu untersuchen werden Spannungen von $0{,}2\,\mathrm{V}$ bis $3\,\mathrm{V}$ in $100\,\mathrm{mV}$-Schritten mit Hilfe des Zellspannungsgenerators erzeugt und mit dem Oszilloskop gemessen. Dabei wird davon ausgegangen, dass der Zellspannungsgenerator eine ähnliche Genauigkeit hat, wie das Tischmultimeter. Das MATLAB Skript aus Anhang A.3.1.6 erledigt dies voll automatisch. Die Ausgabe des Generators wird mit dem Oszilloskop aufgenommen und mit dem Skript aus Anhang A.3.1.7 analysiert. Dabei wird jede der einzelnen Spannungsstufen, wie sie in Abbildung 6.7 zu sehen sind, gemittelt und in einem Vektor gespeichert. Die Mittlung erfolgt jeweils vom roten zum grünen Strich. Anschließend wird die Differenz zwischen den gemessenen und den ausgegebenen Spannungen berechnet und in Abhängigkeit der ausgegebenen Spannung dargestellt. Das Ergebnis ist in Abbildung 6.8 zu sehen.

Die Abweichungen des Oszilloskops gegenüber dem Tischmultimeter sind sehr hoch und betragen je nach Eingangsspannung bis zu $20\,\mathrm{mV}$. Hieraus lassen sich auch die unterschiedlichen mittleren Spannungsdifferenzen für die Transientenwiedergabe aus Abbildung 6.4 erklären. Die Abweichungen sind dort während des starken Einbruches im Startvorgang ge-

ringer als in der restlichen Zeit. Während dieses Zeitraums ist eine Spannung von ungefähr $1{,}8\,\mathrm{V}$ bis $1{,}9\,\mathrm{V}$ vorhanden, genau in diesem Bereich ist die Abweichung des Oszilloskops am geringsten. Sowohl bei höheren als auch bei niedrigeren Spannungen ist die Abweichung höher.

Die Messung des Fehlers des Oszilloskops wurde mit einer Auflösung von $0{,}5\,\mathrm{V/div}$ durchgeführt. Es hat sich gezeigt, dass die Wahl der Auflösung einen großen Einfluss auf den Offset des Oszilloskops hat. Bei $2\,\mathrm{V/div}$ liegt die Nulllinie bereits bei etwa $60\,\mathrm{mV}$, für eine Auflösung von $10\,\mathrm{V/div}$ sogar bei etwa $300\,\mathrm{mV}$.

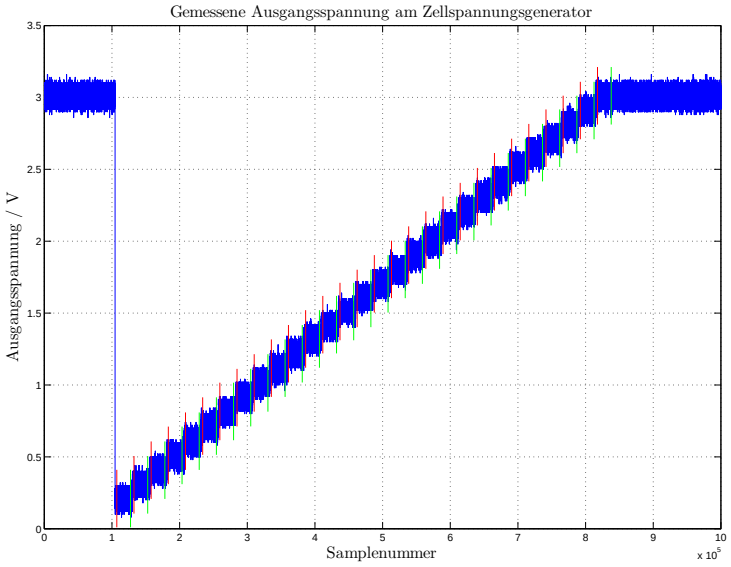


Abbildung 6.7: Treppenförmige Ausgangsspannung am Zellspannungsgenerator zur Bestimmung des Fehlers des Oszilloskops

6.3.4 Serienschaltung mehrerer Generatoren

Eine weitere Messung soll zeigen, dass es auch möglich ist, die Zellspannungsgeneratoren in Serie geschaltet zu betreiben. Die Serienschaltung der Generatoren wird durch die galvanische Trennung jedes einzelnen Generators mit einem DC/DC-Wandler ermöglicht. Es wird wieder die Spannungssequenz von S. Püttjer des Mercedes Benz Vito L, 4 Zylinder Diesel, 95kW, 2,0l genutzt [13]. Die Sequenz wird auf vier Generatoren gegeben, die alle in Serie geschaltet sind. Die Messung erfolgt mit einem Tektronix MSO 3034 Oszilloskop mit einer gemeinsamen Masse für alle Kanäle. In Abbildung 6.9 sind die bereits von einander subtrahierten Messwerte dargestellt. Hierbei fällt primär die Spannung der 4. Zelle

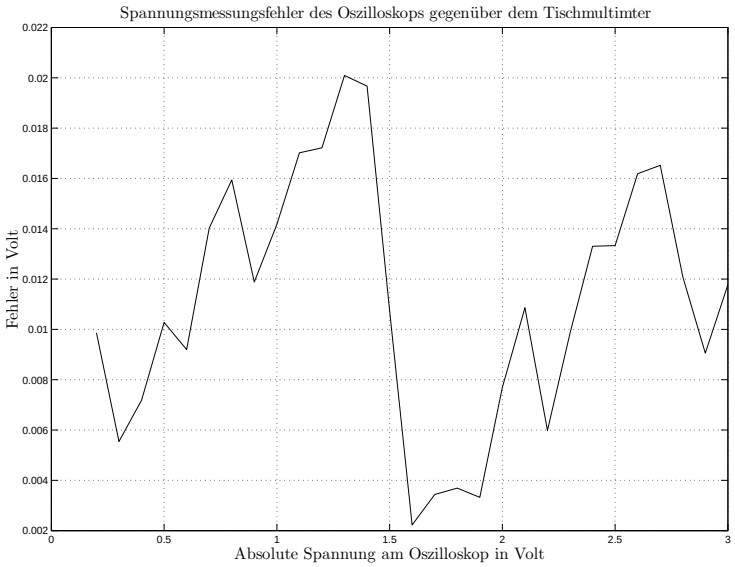


Abbildung 6.8: Fehler des Oszilloskops gegenüber dem Tischmultimeter in Abhängigkeit der Eingangsspannung

auf, die eine Abweichung von etwa $15\,\mathrm{mV}$ zu den anderen hat. Weitere Messungen am Oszilloskops bestätigen diese Abweichungen. Insbesondere bei Spannungen um $1{,}9\,\mathrm{V}$ weicht der 4. Kanal des Oszilloskops um etwa $20\,\mathrm{mV}$ ab. Die hier gemessenen Toleranzen lassen sich also größtenteils auf Messungenauigkeiten des Oszilloskops zurückführen. Es kann jedoch gezeigt werden, dass die Generatoren in der Lage sind, auch in der Serienschaltung zu arbeiten.

6.3.5 Sensor der Klasse 1 als Last

Bei der letzten Messung wird mit der gleichen Spannungssequenz wie zuvor der Start eines PKW simuliert, dabei wird als Last zusätzlich ein Sensor der Klasse 1 angeschlossen, um die Auswirkungen des pulsartigen Stromflusses zu untersuchen. Das Ergebnis zeigt Abbildung 6.10. Es ist deutlich zu erkennen, dass das pulsartige Schalten des Spannungsreglers auf dem Sensor einen starken Einfluss auf die Ausgangsspannung hat. Leuchtet die LED des Sensors, wird nochmals deutlich mehr Strom benötigt. Dies wirkt sich jedoch nicht in einem höheren Spitzenstrom aus, sondern die Impulse des Reglers sind häufiger, wie Abbildung 6.11 zeigt. Es entstehen Spannungsspitzen und Spannungseinbrüche im Bereich von circa $\pm 60\,\mathrm{mV}$. Ein Teil dieses Spannungseinbruches, in etwa die Hälfte, entsteht auf der Leitung vom Verstärker bis zum Sensor auf Grund des hohen Stromflusses. Obwohl die Rückkopplung des Verstärkers direkt an den Bananenbuchsen abgegriffen wird, ist der Verstärker

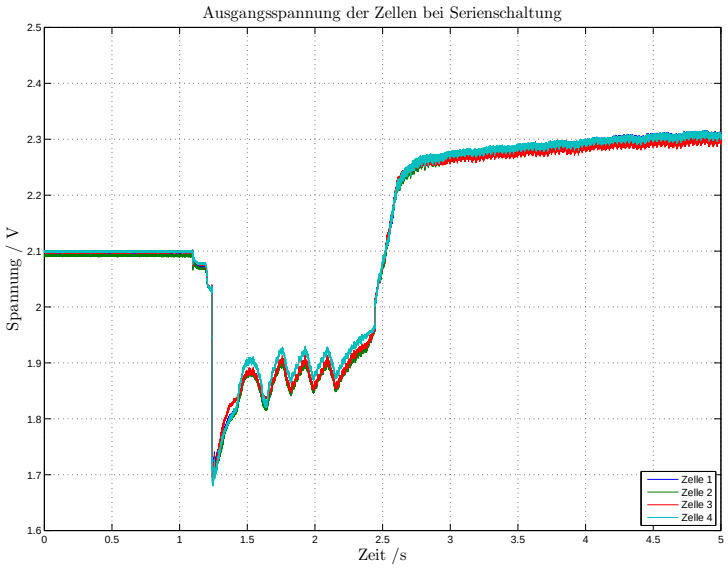


Abbildung 6.9: Messung von 4 in Serie geschalteten Generatoren, jeder der Generatoren gibt den selben Startvorgang eines PKW wieder

nicht in der Lage, so schnelle Veränderungen auszugleichen, denn solch ein Impuls dauert nur wenige Mikrosekunden lang. Als zweite Ursache für das starke Überschwingen kann der Widerstand R5 des Verstärkers ausgemacht werden. Dieser dient zur Messung des Stromes und liegt daher im Signalpfad und erhöht so den gesamten Leitungswiderstand um $1\,\Omega$. Überbrückt man diesen kann der Impuls auf etwa die Hälfte reduziert werden. Somit sind die beiden Widerstände R5 und der Leitungswiderstand die Hauptursachen für die starken Spannungseinbrüche. Es sollte also versucht werden, in einer späteren Überarbeitung der Platine die Leitungslängen vom Verstärker bis zum Sensor deutlich kürzer zu halten. Eine weitere Möglichkeit das Verhalten zu verbessern ist es den Widerstand für die Strommessung zu verkleinern. Die führt jedoch zu höheren Ungenauigkeiten, speziell für die Messung des Ruhestroms.

6.3.6 Kalibrierung der Strommessung

Für die Strommessung kommen, wie in Kapitel 2.8 beschreiben, zwei Verstärker zum Einsatz. Die beiden Verstärker haben unterschiedliche Verstärkungen, um einerseits den Strom im Bereich von $\pm 250\,\mathrm{mA}$ zu messen und andererseits den Ruhestrom der Sensoren messen. Da dieser Ruhestrom sehr gering ist, hat der zweite Verstärker eine sehr hohe Verstärkung von etwa 800, um einen Messbereich von $\pm 2\,\mathrm{mA}$ zu erhalten.

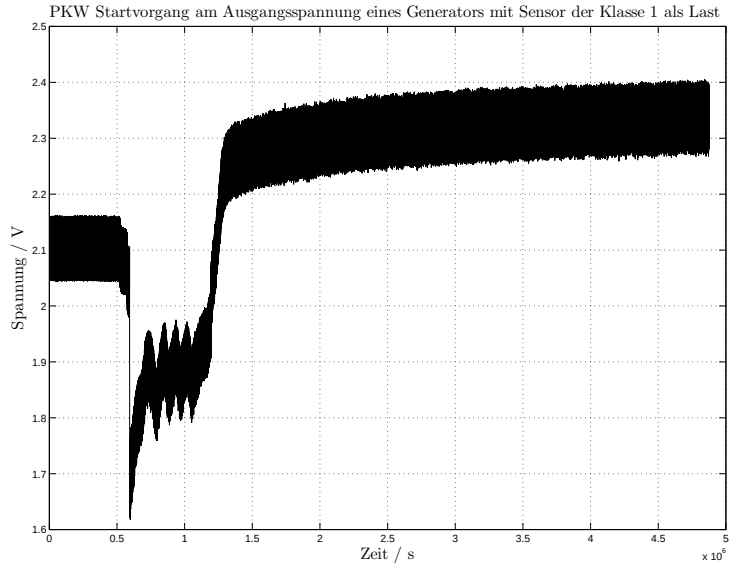


Abbildung 6.10: Messung der Ausgangsspannung des Generators bei Wiedergabe eines PKW-Startvorgangs mit einem Sensor der Klasse 1 als Last

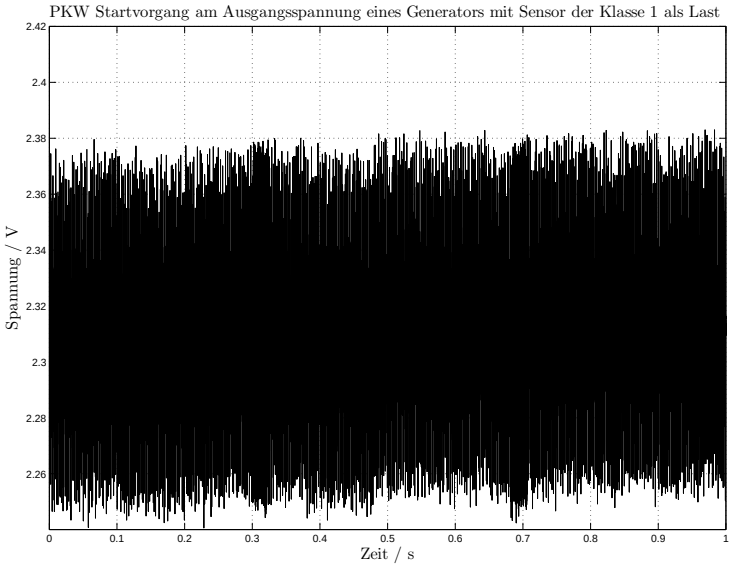


Abbildung 6.11: Zeitlich begrenzter Ausschnitt der Messung der Ausgangsspannung des Generators bei Wiedergabe eines PKW-Startvorgangs mit einem Sensor der Klasse 1 als Last

Auf Grund dieser sehr hohen Verstärkung zeigen sich bei Messungen einige unerwünschte Effekte. Zum einen ist der Offset sehr hoch und zum anderen ist dieser auch abhängig von der aktuellen Ausgangsspannung des Generators. Um einen Zusammenhang zwischen Ausgangsspannung und Offset zu erhalten, wird eine Messreihe aufgenommen, in der für sieben verschiedene Spannungen von $100\,\mathrm{mV}$ bis $6\,\mathrm{V}$ jeweils Ströme von $0\,\mu\mathrm{A}$ bis $2000\,\mu\mathrm{A}$ erzeugt und gemessen werden. In Abbildung 6.12 sind die durch den Generator gemessenen Ströme bei verschiedenen Spannungen in Abhängigkeit des tatsächlich fließenden Stromes dargestellt. Bereits hier lässt sich erahnen, dass der Offset des Stromes proportional zur Ausgangsspannung ist. Klarer wird dies, wenn man für jede der Kennlinien den Offset und Verstärkungsfehler berechnet und jeweils in einem Diagramm über die Ausgangsspannung aufträgt. Dies ist in Abbildung 6.13 und 6.14 dargestellt. Hier ist deutlich zu erkennen, dass sich der Offset linear in Abhängigkeit der Ausgangsspannung verhält, jedoch nicht proportional, da er selbst auch noch einen Offset hat. Der Verstärkungsfehler ist mit einem Faktor von $1{,}2$ sehr hoch, jedoch ist dieser unabhängig von der Ausgangsspannung. Es sollte also möglich sein diese Messfehler in der Software größtenteils zu kompensieren.

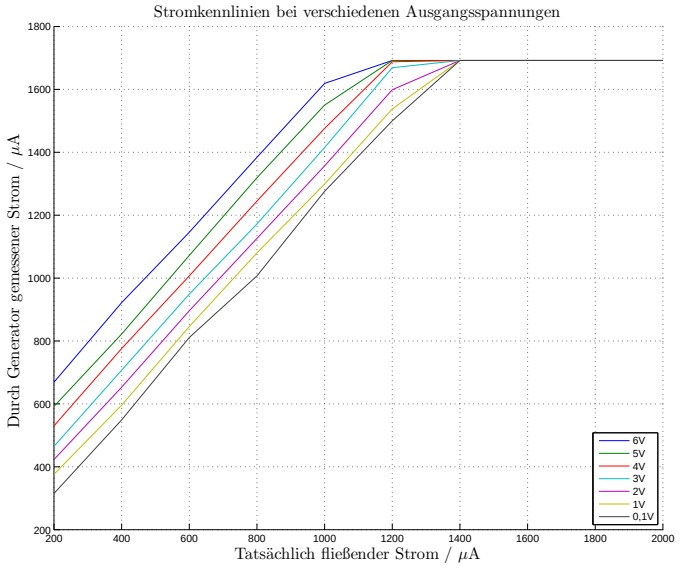


Abbildung 6.12: Gemessener Strom in Abhängigkeit des tatsächlichen Stroms für verschiedene Ausgangsspannungen

6.4 Störungen am Ausgang durch den DC/DC-Wandler

Zum Schluss der Arbeit ist noch eine Störung im Ausgangssignal aufgefallen, die den Betrieb unter Umständen stören kann. Löst man die Zeitachse beim Oszilloskop deutlich höher

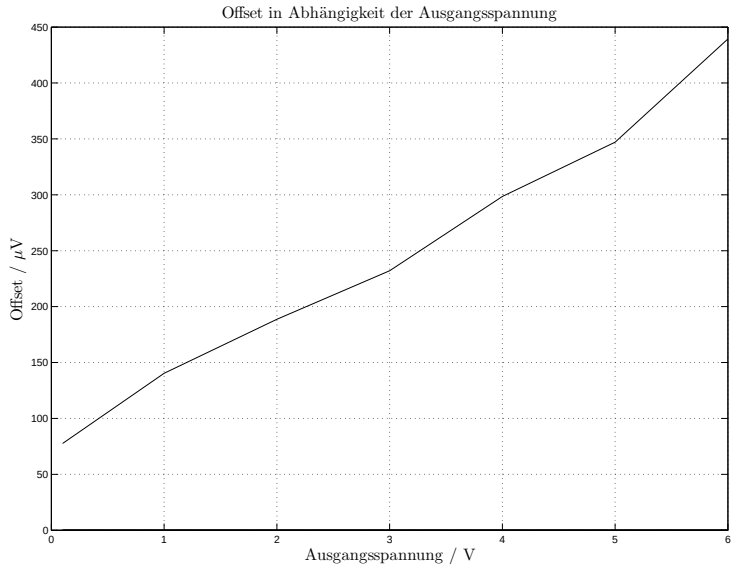


Abbildung 6.13: Offsetfehler des gemessenen Stroms in Abhängigkeit der Ausgangsspannung des Generators

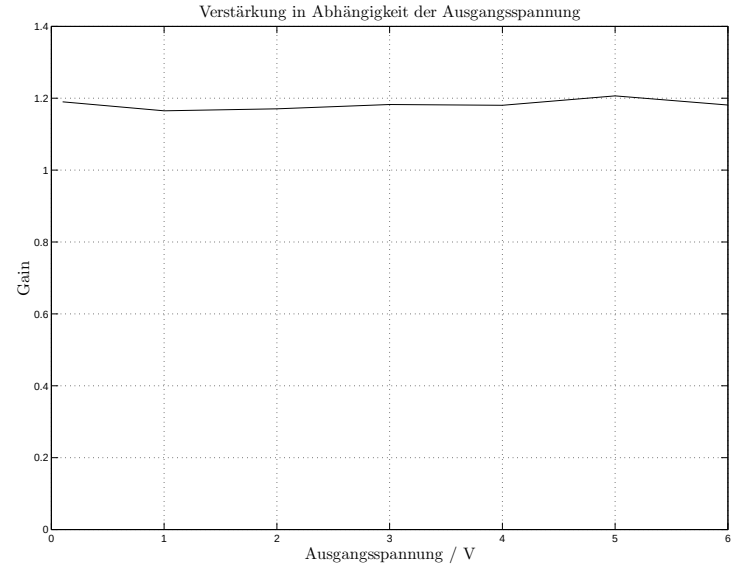


Abbildung 6.14: Verstärkungsfehler des gemessenen Stroms in Abhängigkeit der Ausgangsspannung des Generators

auf, fallen, selbst bei stationären Spannungen, sich periodisch wiederholende Spannungsspitzen auf. Eine Aufnahme dieser Störung ist in Abbildung 6.15 dargestellt. Es handelt sich dabei höchst wahrscheinlich um den DC/DC-Wandler, welcher typisch mit einer Frequenz von $70\,\mathrm{kHz}$ arbeitet [9]. In der Aufnahme in Abbildung 6.15 entsteht alle $8\,\mu\mathrm{s}$ eine Spannungsspitze. Dies entspricht einer Frequenz von $125\,\mathrm{kHz}$. Bedenkt man, dass der Regler vermutlich jedes mal beim An- und Abschalten eine Spitze erzeugt, halbiert sich die gemessene Frequenz auf $62{,}5\,\mathrm{kHz}$ und liegt somit sehr nahe bei typischen Frequenz des Wandlers. Betrachet man die Abbildung genauer, erkennt man, dass die Spitzen immer abwechselnd groß und klein sind, was die Vermutung der halben Frequenz bestätigt.

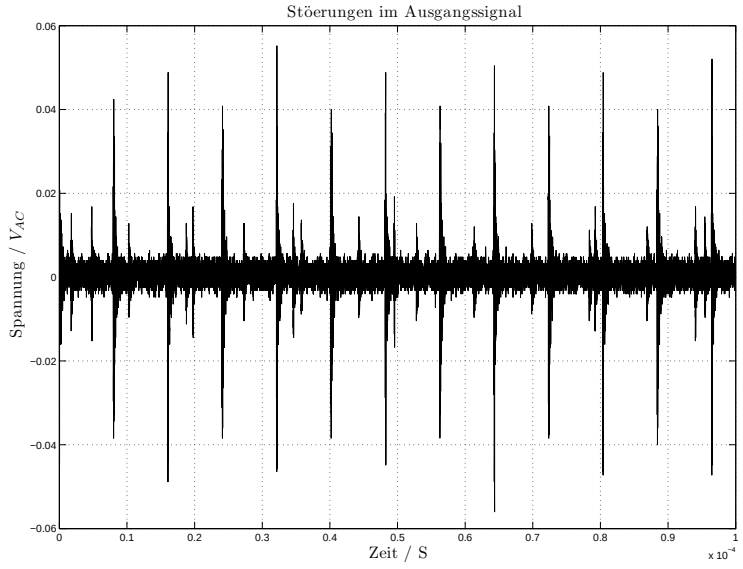


Abbildung 6.15: Störungen der Ausgangsspannung durch den DC/DC-Wandler

7 Fazit

In diesem Kapitel soll das Ergebnis der Arbeit betrachtet werden. Dabei gilt es zum einen zu klären, ob die Aufgabenstellung der Bachelorarbeit erfüllt ist. Zum anderen soll ein Ausblick gegeben werden, inwieweit das Ergebnis der Arbeit noch verbessert und weitergeführt werden kann.

7.1 Erreichen der Aufgabenstellung

Das Ziel der Arbeit ist es, einen Generator zu entwerfen, der die hoch präzise und schnelle Wiedergabe von Spannungssequenzen erlaubt. Die Auflösung und Genauigkeit des Generators soll im Bereich der Auflösung der aktuell zur Verfügung stehenden Batteriesensoren liegen. Bei der Wiedergabe von Spannungssequenzen soll zudem eine ausreichende Bandbreite des Signals möglich sein, um typische Ereignisse, wie den Startvorgang eines PKW, ohne große Unterschiede wiedergeben zu können. Das gesamte System soll auf mindestens 40 Generatoren erweiterbar sein, um auch große Batterien durch Serienschaltung der Module nachbilden zu können. Hierfür ist eine galvanische Trennung der einzelnen Generatoren nötig.

Es ist zunächst eine Untersuchung der aktuell umgesetzten Sensoren der Klassen 1 und 2 durchgeführt worden, um deren Kenndaten, wie beispielsweise den Spannungsbereich oder den Strombedarf, zu ermitteln. Anhand dieser Kenndaten wurde ein Konzept für den Generator entworfen, welches in der Lage sein sollte, die Anforderungen an den Zellspannungsgenerator zu erfüllen. Anschließend sind Untersuchungen durchgeführt worden, um den Verstärker für den Ausgang des Generators zu definieren und zu dimensionieren. Nach dem Entwurf der Hardware und dem Aufbau selbiger wurde eine umfangreiche Software zur Steuerung und Kalibrierung der Generatoren angefertigt.

Für die genaue Wiedergabe von einzelnen Samples erfolgt das Setzen der Ausgangsspannung mit einer Rückkopplung über einen AD-Umsetzer, wodurch Abweichungen und Offsets von Verstärkern und passiven Bauelementen vermieden werden können. Um auch transiente Spannungssequenzen genau wiedergeben zu können, wird zunächst eine Wertetabelle mit Hilfe des externen AD-Umsetzers erstellt, wodurch es möglich wird, Samples mit hoher Genauigkeit und hoher Geschwindigkeit wiederzugeben.

Es wurden insgesamt 6 der Generatoren aufgebaut, in Betrieb genommen und getestet. Bei Messungen an der Hardware zeigte sich, dass die Anforderungen an die Wiedergabe stationärer Spannungen in vollem Umfang erfüllt werden, wobei für die Genauigkeit des Generators primär das zur Kalibrierung verwendete Multimeter ausschlaggebend ist. Bei der Wiedergabe transienter Vorgänge konnte zwar keine so hohe Genauigkeit wie bei der Wiedergabe stationärer Spannungen erreicht werden, dies war jedoch auch nicht gefordert und ein Großteil der Abweichungen kann durch die Toleranz des Oszilloskops begründet werden.

Die galvanische Trennung durch den DC/DC Wandler auf jedem Generator erlaubt eine einfache und effektive galvanische Trennung der Generatoren untereinander. Ethernet bietet als Bussystem eine einfache und schnelle Möglichkeit mit den Modulen zu kommunizieren, wodurch zusätzlich der Bau einer Adapterplatine zum Anschluss an den PC entfällt. Die in der Aufgabenstellung definierte Mindestanforderung an den Spannungsbereich und den Mindeststrom der Module konnte sogar weit übertroffen werden. So liefert der Generator Spannungen von $100\,\mathrm{mV}$ bis $7{,}5\,\mathrm{V}$ und einen Ausgangsstrom von bis zu $1{,}5\,\mathrm{A}$. Die aktuelle Samplerate liegt, wie auch in der Aufgabenstellung, bei $20\,\mathrm{kSPS}$ und erfüllt somit alle Anforderungen. Hierzu ist noch zu sagen, dass die Samplerate mit einfachen Änderungen in der Software noch deutlich angehoben werden kann.

Für die Steuerung der Generatoren ist eine umfangreiche MATLAB-Bibliothek entstanden, die den Umgang mit den Generatoren stark vereinfacht. Es sind so eine Handvoll Beispiele entstanden, die die Benutzung der Bibliothek veranschaulichen und auch wie die Signalaufbereitung für die Generatoren durchgeführt werden kann.

Alles in allem konnte mit dieser Arbeit ein wichtiger Beitrag zum gelingen des Forschungsprojekts BATSEN geleistet werden. Es ist mit der in der Arbeit entworfenen Soft- und Hardware möglich, die Genauigkeit der Sensoren zu erhöhen und genauere Analysen der Sensoren durchzuführen.

7.2 Ausblick

Um den Zellspannungsgenerator weiter zu verbessern, können noch einige Veränderungen vorgenommen werden, die aufgrund der begrenzten Zeit für diese Arbeit nicht umgesetzt werden konnten. Die wichtigsten Punkte sollen in den folgenden Absätzen kurz dargelegt werden.

7.2.1 Verringern der Störung durch den DC/DC-Wandler

In Kapitel 6.4 hatte sich gezeigt, dass das Ausgangssignal stark durch die Schaltvorgänge mit Spannungsspitzen und Spannungseinbrüchen belastet wird. Es ist zu klären warum diese Störungen nicht durch die Spannungsregler hinter dem DC/DC-Wandler herausgefiltert werden und wie diese Störungen vermieden werden können.

7.2.2 Verringern der Störung durch einen Sensor

Neben den Störungen durch den DC/DC-Wandler sind im Kapitel 6.3.5 ähnliche Störungen der Ausgangsspannung durch den Schaltregler auf dem Sensor aufgefallen. Auch hier gilt es zu prüfen, ob die Hardware so verändert werden kann, dass das Ausgangssignal weniger anfällig für solche Schaltimpulse ist.

Fehlende Implementierung der Strommessung

In der Hardware sind bereits alle Komponenten enthalten, um den Stromverbrauch der Sensoren zu bestimmen. Die Software ist jedoch nur teilweise umgesetzt. Insbesondere fehlt die Kalibrierung für die Strommessung, wozu in dieser Arbeit lediglich einige Voruntersuchungen durchgeführt wurden. Des Weiteren fehlt die Abspeicherung der gemessenen Ströme während einer Transientenwiedergabe und eine Funktion zum Auslesen der gespeicherten Werte.

Fehlende Messung und Auswertung der Temperatur

Ebenfalls bereits auf der Platine vorgesehen ist die Temperaturmessung durch einen externen über I^2C angeschlossenen Temperatursensor. Diese Informationen könnten nützlich sein für die Kalibrierung jedes Generators, um eine Temperaturabhängigkeit zu kompensieren, allerdings wird der Temperatursensor bislang in der Software weder angesprochen, noch die Temperatur ausgewertet.

7.2.3 Ineffiziente Übertragung von Spannungssequenzen

Aktuell ist es möglich bei der Programmierung eines Spannungssequenz etwa 6000 Samples pro Sekunde zu übertragen. Je Sample werden bei Spannungen zwischen $1\,\mathrm{V}$ und $10\,\mathrm{V}$ durch die ASCII-Kodierung und das Trennzeichen $8\,\mathrm{Byte}$ benötigt. Somit ergibt sich eine Datenrate von $48\,\frac{\mathrm{kB}}{\mathrm{s}}$. Der Ethernet-Controller unterstützt theoretisch jedoch bis zu $12{,}5\,\frac{\mathrm{MB}}{\mathrm{s}}$.

Seitens der Hardware ist die maximale Übertragungsgeschwindigkeit also noch lange nicht erreicht. Es muss also an der Software liegen, dass die Übertragung so langsam ist. In zukünftigen Arbeiten sollte die Software zur Übertragung der Spannungswerte nochmals genauer untersucht werden, um diese zu optimieren. Die Übertragung von Binärdaten an Stelle einer ASCII-Kodierung könnte hier einen großen Unterschied machen, da so nur noch ein Viertel der Datenmenge übertragen werden muss (2 Byte statt 8 Byte je Sample) und die Umwandlung zwischen ASCII- und Binärdaten nicht mehr nötig ist.

Vereinfachung der Hardware

Sowohl für den externen AD-Umsetzer als auch den externen DA-Umsetzer kommen aktuell $16\,\mathrm{Bit}$-Varianten zum Einsatz. Es ist zu prüfen, ob die Verwendung von Umsetzern mit $14\,\mathrm{Bit}$ ausreichend ist. Des Weiteren kann die relativ teure externe Referenzspannungsquelle durch eine einfachere ersetzt werden oder auch komplett entfernt werden, denn es ist keine hoch präzise Quelle nötig. Es werden alle Abweichungen durch die Kalibrierung beseitigt. Wobei man bei Verwendung einer einfacheren Quelle unter Umständen eine höhere Temperaturabhängigkeit erhält.

Kalibrierung der Sensoren

Das Ziel des Zellspannungsgenerator-Projektes ist es, die Sensoren einfach kalibrieren zu können und eine Analyse des Funkverkehrs zu ermöglichen. Hierfür muss die Software auf den Sensoren noch angepasst werden, um beispielsweise eine Kalibrierung zu ermöglichen.

Literaturverzeichnis

[1] ANALOG DEVICES: *AD7798 - 3-Channel, Low Noise, Low Power, 16-/24-Bit, Σ - Δ ADC with On-Chip In-Amp*. Version: 2007. http://www.analog.com/static/imported-files/data_sheets/AD7798_7799.pdf, Abruf: 08.02.2012

[2] ANALOG DEVICES: *AD5541 - 2.7 V to 5.5 V, Serial-Input, Voltage-Output, 16-Bit DACs*. Version: 2011. http://www.analog.com/static/imported-files/data_sheets/AD5541_5542.pdf, Abruf: 08.02.2012

[3] ANALOG DEVICES: *AD8226 - Wide Supply Range, Rail-to-Rail Output Instrumentation Amplifier*. Version: 2011. http://www.analog.com/static/imported-files/data_sheets/AD8226.pdf, Abruf: 08.02.2012

[4] ANALOG DEVICES: *ADM3054 - 5 kV rms Signal Isolated High Speed CAN Transceiver with Bus Protection*. Version: 2011. http://www.analog.com/static/imported-files/data_sheets/ADM3054.pdf, Abruf: 29.03.2012

[5] ANALOG DEVICES: *ADUM1201 - Dual-Channel Digital Isolators*. Version: 2012. http://www.analog.com/static/imported-files/data_sheets/ADuM1200_1201.pdf, Abruf: 17.03.2012

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

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

[8] LINDEN, David ; REDDY, Thomas B.: In: *Handbook of batteries* Bd. 3. New York : McGraw-Hill, 2001. – ISBN 0–07–135978–8

[9] MURATA POWER SOLUTIONS: *NMXS1215UC - Isolated 5W Single and Dual Output DC/DC Converters*. Version: 2012. http://www.murata-ps.com/data/power/ncl/kdc_nmxu.pdf, Abruf: 07.02.2012

[10] NXP: *I^2C-bus specification and user manual*. Version: 2012. http://www.nxp.com/documents/user_manual/UM10204.pdf, Abruf: 22.02.2012

[11] PHILIPS: *Automatik-Multimeter PM 2519 - Die Messanlage für den Praktiker*. Version: 1988. http://www.helmut-singer.de/pdf/phpm2519.pdf, Abruf: 16.02.2012

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

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

[14] SST: *SST25VF032B - 32 Mbit SPI Serial Flash*. Version: 2011. http://www.sst.com/dotAsset/40373.pdf, Abruf: 08.02.2012

[15] STMICROELECTRONICS: *1V high efficiency syncronous step up converter*. Version: 2005. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00002171.pdf, Abruf: 16.02.2012

[16] TEXAS INSTRUMENT: *TPS76130, TPS76132, TPS76133, TPS76138, TPS76150 LOW-POWER 100-mA LOW-DROPOUT LINEAR REGULATORS*. Version: 2001. http://www.ti.com/lit/ds/symlink/tps76133.pdf, Abruf: 07.02.2012

[17] TEXAS INSTRUMENT: *MSP430F1232 Datasheet*. Version: 2006. http://www.ti.com/lit/ds/symlink/msp430f1232.pdf, Abruf: 17.02.2012

[18] TEXAS INSTRUMENT: *TPS5410 - 1-A, WIDE INPUT RANGE, STEP-DOWN SWIFT CONVERTER*. Version: 2006. http://www.ti.com/lit/ds/slvs675a/slvs675a.pdf, Abruf: 07.02.2012

[19] TEXAS INSTRUMENT: *TPS5420 - 2-A, WIDE INPUT RANGE, STEP-DOWN SWIFT CONVERTER*. Version: 2007. http://www.ti.com/lit/ds/symlink/tps5420.pdf, Abruf: 07.02.2012

[20] TEXAS INSTRUMENT: *REF5025 - Low-Noise, Very Low Drift, Precision VOLTAGE REFERENCE*. Version: 2010. http://www.ti.com/lit/ds/symlink/ref5025.pdf, Abruf: 07.02.2012

[21] TEXAS INSTRUMENT: *TPS73801 - 1.0-A LOW-NOISE FAST-TRANSIENT-RESPONSE LOW-DROPOUT REGULATOR*. Version: 2010. http://www.ti.com/lit/ds/slvs915/slvs915.pdf, Abruf: 07.02.2012

[22] TEXAS INSTRUMENT: *MSP430F235 Datasheet*. Version: 2011. http://www.ti.com/lit/ds/symlink/msp430f235.pdf, Abruf: 19.02.2012

[23] TEXAS INSTRUMENTS: *TMP102 - Low Power Digital Temperature Sensor With SMBus/Two-Wire Serial Interface in SOT563*. Version: 2007. http://www.ti.com/lit/ds/symlink/tmp102.pdf, Abruf: 09.02.2012

[24] TEXAS INSTRUMENTS: *1.5A, 24V, 17MHz, power operational amplifier*. Version: 2011. http://www.ti.com/lit/gpn/opa564, Abruf: 25.02.2012

[25] TEXAS INSTRUMENTS: *Stellaris LM3S9B92 Microcntroller - Datasheet*. Version: 2012. http://www.ti.com/lit/ds/symlink/lm3s9b92.pdf, Abruf: 07.02.2012

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

A Quellcodes

A.1 Software für den Mikrocontroller in C

A.1.1 adcdac.h

```
/*
 * adcdac.h
 *
 *  Created on: 27.12.2011
 *      Author: fabian
 */

#ifndef ADCDAC_H_
#define ADCDAC_H_

extern volatile unsigned short voltage_adc;
extern volatile unsigned long voltage_volt; // current voltage in mV

int adcdac_init(void);
void adcdac_clear_fifo(void);

void adc_select(char select);
void dac_select(char select);

unsigned char adcdac_write_read(unsigned char data);

unsigned char adc_status(void);  // read the status register from the ADC
unsigned short adc_read_data(void); // read the data register from the ADC
void adc_set_config_reg(unsigned short value);
unsigned short adc_read_config_reg(void);
void adc_set_mode_reg(unsigned short value);
unsigned short adc_read_offset_reg(void);
void adc_set_offset_reg(unsigned short value);
unsigned short adc_read_fs_reg(void);
void adc_set_fs_reg(unsigned short value);
long adc_oneshot(void);  // read a new value from the ADC

void dac_set_data(unsigned short data);

#endif /* ADCDAC_H_ */
```

A.1.2 adcdac.c

```
/*
 * adcdac.c
 *
 *  Created on: 27.12.2011
 *      Author: fabian
 *
 *  Ansteuerung des externen AD-Umsetzers und DA-Umsetzers.
 *  Die Ansteuerung geschieht über den Hardware-SPI 0.
 *  Die Chip-Selects liegen auf den Pins PA3 und PA6.
 *  Nach der Initialisierung durch addac_init() sind
 *  hauptsächlich die beiden Funktionen adc_oneshot()
 *  und dac_set_data() interessant. Die restlichen
 *  Funktionen werden primär nur intern verwendet.
 *
```

```
 *  Es muss bei der Benutzung darauf geachtet werden,
 *  dass nur der ADC oder der DAC zur gleichen Zeit
 *  angesteuert wird. Es kommt ansonsten zu Fehlern,
 *  da beide Chips über einen Bus angesprochen werden.
 */

#include "adcdac.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 <utils/uartstdio.h>
#include <inc/lm3s9b92.h>

#include "system_timer.h"
#include "config.h"

// Gemessener Wert aus letztem Aufruf von adc_oneshot(),
// direkter ADC-Wert, ihne Umwandlung in Spannung
volatile unsigned short voltage_adc;
// Gemessener Spannung aus letztem Aufruf von adc_oneshot(),
// kodiert in Mikrovolt
volatile unsigned long voltage_volt;

/**
 * Initialisierung der SPI-Hardware und der beiden Umsetzer.
 * Zudem wird geprüft, ob der ADC ansprechbar ist, indem das
 * Statusregister geschrieben und anschließend wieder gelesen
 * wird.
 *
 * Return: 0 bei Fehler, 1 bei Erfolg
 */
int adcdac_init(void)
{
    volatile unsigned long ulLoop;

    voltage_adc = 0;

    UARTprintf("Initializing ADC and DAC...");
    // Initialisierung der Pins
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOA; // Clock enable on port A
    ulLoop = SYSCTL_RCGC2_R;
    GPIO_PORTA_DIR_R |= (1<<3) | (1<<6);  // set chip selects as output
    GPIO_PORTA_DEN_R |= (1<<3) | (1<<6);

    // Sicherstellen, dass beide Chip-Selects high sind (nicht ausgewählt)
    adc_select(0);
    dac_select(0);

    // Initialisierung von SSI0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    // Kein Hardware Chipselect, dies wird per Software gemacht,
    // da zwei Chipselects benötigt werden
    //GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA4_SSI0RX);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);

    // Setzen der Pins in den SSI Modus
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);

    // Den Takt auf 1MHz stellen (hier ist noch Luft nach oben)
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3, SSI_MODE_MASTER, 2000000, 8);  // ADC=
          SSI_FRF_MOTO_MODE_3

    // Einschalten des SPI Interfaces
    SSIEnable(SSI0_BASE);

    adcdac_clear_fifo();  // Löschen des Hardware FIFOs

    // Reset des ADC durch senden von 32 Einsen
    adc_select(1);
    adcdac_write_read(0xFF);
    adcdac_write_read(0xFF);
    adcdac_write_read(0xFF);
    adcdac_write_read(0xFF);
    adc_select(0);

    // Ein Dummyread vom Statusregister des ADCs.
```

```
    // Es ist nicht klar warum das nötig ist, aber beim ersten
    // Auslesen des Registers wird sonst Müll zurückgegeben
    adc_status();

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

     // In den "power down" Modus gehen, um das "mode register" zu ändern
    adc_set_mode_reg(0x600A);
    adc_set_offset_reg(0);  // Initialisierung des "offset registers" mit 0
    adc_set_fs_reg(0); // Initialisierung des "full scale registers" mit 0

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

    // Durchführung der internen Kalibrierung des ADCs
    adc_set_mode_reg(0x800A); // Internal zero offset calibration
    while((adc_status() & (1<<7)));
    adc_set_mode_reg(0xA00A); // Internal full scale calibration
    while((adc_status() & (1<<7)));

    // Prüfen, ob die Kalibierung erfolgreich war
    // (sich die Werte geändert haben)
    if(adc_read_offset_reg() == 0)
    {
        UARTprintf("failed: offset calibration failed!\n");
        return 0;
    }
    if(adc_read_fs_reg() == 0)
    {
        UARTprintf("failed: full scale calibration failed!\n");
        return 0;
    }

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

    // "config register" zur Überprüfung lesen
    if(adc_read_config_reg() != 0x1010)
    {
        UARTprintf("failed: config register check failed!\n");
        return 0;
    }

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

/*
 * Löschen des Hardware FIFOs vom SPI-Interface
 */
void adcdac_clear_fifo(void)
{
    unsigned long dummy;
    while(SSIDataGetNonBlocking(SSI0_BASE, &dummy));  // clear the fifo
}

/*
 * An- und Abwahl des ADC.
 * select: 1=Chip ausgewählt, CS low
 *         0=Chip abgewählt, CS high
 */
void adc_select(char select)
{
    if(select)
    {
        GPIO_PORTA_DATA_R |= (1<<3);  // make shure dac is unselected
        GPIO_PORTA_DATA_R &= ~(1<<6);  // select adc
    }
    else
        GPIO_PORTA_DATA_R |= (1<<6);
}

/*
```

```
 * An- und Abwahl des DAC.
 * select: 1=Chip ausgewählt, CS low
 *         0=Chip abgewählt, CS high
 */
void dac_select(char select)
{
    if(select)
    {
        GPIO_PORTA_DATA_R |= (1<<6);  // make shure adc is unselected
        GPIO_PORTA_DATA_R &= ~(1<<3);  // select dac
    }
    else
        GPIO_PORTA_DATA_R |= (1<<3);
}

/*
 * Schreiben und Lesen auf dem SPI-Interface in
 * einem Schritt.
 */
unsigned char adcdac_write_read(unsigned char data)
{
    unsigned long ans;
    SSIDataPut(SSI0_BASE, data);
    SSIDataGet(SSI0_BASE, &ans);
    return (unsigned char)(ans&0xff);
}

/*
 * Lesen des Status Registers vom ADC.
 * Return: 1 Byte Status Register
 */
unsigned char adc_status(void)
{
    adc_select(1);
    adcdac_clear_fifo();
    adcdac_write_read(0x40);
    unsigned char status_reg = adcdac_write_read(0x00);
    adc_select(0);
    return status_reg;
}

/*
 * Auslesen der Daten vom ADC.
 * Es werden hierfür zwei Bytes gelesen und anschließend
 * zusammengesetzt. Diese Funktion wird nur intern
 * verwendet!
 */
unsigned short adc_read_data(void)
{
    unsigned short data;
    adc_select(1);
    adcdac_write_read(0x58);
    data = (adcdac_write_read(0x00))<<8;
    data |= adcdac_write_read(0x00);
    adc_select(0);
    return data;
}

/*
 * Auslesen des Konfigurations Registers des ADCs.
 * Es werden hierfür zwei Bytes gelesen und anschließend
 * zusammengesetzt.
 */
unsigned short adc_read_config_reg(void)
{
    unsigned short data;
    adc_select(1);
    adcdac_write_read(0x50);
    data = (adcdac_write_read(0x00))<<8;
    data |= adcdac_write_read(0x00);
    adc_select(0);
    return data;
}

/*
 * Schreiben des Konfigurationsregsters
 * Das Register ist 2 Bytes groß, es muss daher ein
 * zwei Byte short Wert übergeben werden. Es erfolgt
 * keine Prüfung.
 */
void adc_set_config_reg(unsigned short value)
{
    adc_select(1);
    adcdac_write_read(0x10);
```

```
    adcdac_write_read((unsigned char)(value>>8));
    adcdac_write_read((unsigned char)(value&0xff));
    adc_select(0);
}

/*
* Schreiben des Mode Registers.
* Das Register ist 2 Bytes groß, es muss daher ein
* zwei Byte short Wert übergeben werden. Es erfolgt
* keine Prüfung.
 */
void adc_set_mode_reg(unsigned short value)
{
    adc_select(1);
    adcdac_write_read(0x08);
    adcdac_write_read((unsigned char)(value>>8));
    adcdac_write_read((unsigned char)(value&0xff));
    adc_select(0);
}

/*
 * Auslesen des Offsetregisters des ADC.
 * Dieses ist zwei Bytes groß, die hier automatisch
 * beide gelsen und zusammen gesetzt werden.
 */
unsigned short adc_read_offset_reg(void)
{
    unsigned short data;
    adc_select(1);
    adcdac_write_read(0x70);
    data = (adcdac_write_read(0x00))<<8;
    data |= adcdac_write_read(0x00);
    adc_select(0);
    return data;
}

/*
 * Setzen des Offsetregisters des ADC.
 * Es erfolgt keine Prüfung!
 */
void adc_set_offset_reg(unsigned short value)
{
    adc_select(1);
    adcdac_write_read(0x30);
    adcdac_write_read((unsigned char)(value>>8));
    adcdac_write_read((unsigned char)(value));
    adc_select(0);
}

/*
 * Auslesen des Fullscale Registers des ADC.
 * Dieses ist zwei Bytes groß, die hier automatisch
 * beide gelsen und zusammen gesetzt werden.
 */
unsigned short adc_read_fs_reg(void)
{
    unsigned short data;
    adc_select(1);
    adcdac_write_read(0x78);
    data = (adcdac_write_read(0x00))<<8;
    data |= adcdac_write_read(0x00);
    adc_select(0);
    return data;
}

/*
 * Setzen des Fullscale Registers des ADC.
 * Es erfolgt keine Prüfung!
 */
void adc_set_fs_reg(unsigned short value)
{
    adc_select(1);
    adcdac_write_read(0x38);
    adcdac_write_read((unsigned char)(value>>8));
    adcdac_write_read((unsigned char)(value));
    adc_select(0);
}

/*
 * Lesen von einem Sample aus dem externen ADC.
 * Es wird zunächst darauf gewatet, dass ein Sample
 * aufgenommen wurde und bereit ist zum Lesen. Daher
 * muss vorher eine Wandlung gestartet worden sein
 * oder der ADC muss im Freerun-Modus sein.
```

```
 * Der gemessene Wert wird dann in eine Spannung
 * umgerechnet, die ideal berechnet wird. Hierfür
 * ist zugrudegelegt, dass eine 2,5V Referenz angeschlossen
 * ist, der ADC 16 Bit hat und am Eingang ein Spannungsteiler
 * mit dem Faktor 1/3 angebracht ist.
 *
 * Return: Gemessene Spannung in Mikrovolt
 */
long adc_oneshot(void)
{
    adc_read_data();  // make shure last data is read
    while(adc_status() & (1<<7));  // wait for new sample
    voltage_adc = adc_read_data();

    // ADC value to voltage
    voltage_volt = (long)voltage_adc*15625/2048*15;   // voltage_adc*3/65536*2.5*1000000

    voltage_volt = (long)((long long)voltage_volt*(long long)config.voltage_gain/(long long)1000000)+config.
         voltage_offset;

    return voltage_volt;
}

/*
 * Setzen des aktuellen DAC-Werts.
 * Der DAC muss nicht initialisiert werden und es
 * kann auch nicht geprüft werden, ob er korrekt
 * angesprochen wird. Das Übernehmen des Wertes erfolgt
 * beim wechsel der Chipselect Leitung von low nach high.
 * data: Samplewert der ausgegeben werden soll
 */
void dac_set_data(unsigned short data)
{
    adcdac_clear_fifo();
    dac_select(1);
    adcdac_write_read((unsigned char)(data>>8));
    adcdac_write_read((unsigned char)(data&0xff));
    dac_select(0);
}
```

A.1.3 calibrate.h

```
/*
 * calibrate.h
 *
 *  Created on: 30.12.2011
 *      Author: fabian
 */

#ifndef CALIBRATE_H_
#define CALIBRATE_H_

void calibrate(void);
unsigned short calibrate_set_voltage(unsigned long microvolt);
unsigned short microvolt2dac(unsigned long microvolt);

#endif /* CALIBRATE_H_ */
```

A.1.4 calibrate.c

```
/*
 * calibrate.c
 *
 *  Created on: 30.12.2011
 *      Author: fabian
 *
 *  In dieser Datei sind eine Reihe von Funktionen zum Kalibieren
 *  und andwenden der Kalibrierung. Es geht dabei um die Kalibrierung
 *  des DA-Umsetzers und Verstärkers als Gesamtsystem.
 */

#include "calibrate.h"
```

```
#include "adcdac.h"
#include "system_timer.h"
#include "config.h"

/**
 * Diese Funktion erstellt eine Tabelle mit Kalibrierungswerten.
 * Hierfür wird der AMP angewiesen Spannungen von 0,1V bis 6,0V zu
 * erzeugen. Die gemessenen Spannungen dazu werden in der Tabelle
 * abgelegt und dienen später zur Kalibrierung.
 * Zuvor ist der ADC kalibriert worden, um eine maximale Genauigkeit
 * zu erreichen.
 */
void calibrate(void)
{
    // restore default values
    memcpy(mv2dac, mv2dac_default, sizeof(mv2dac));
    unsigned short i;
    for(i=0; i<CALIBRATION_NUMBER; i++)
    {
        mv2dac[i][1]=calibrate_set_voltage(mv2dac[i][0]);
    }
}

/*
 * Setzen der Ausgangsspannung mit Rückkopplung über den AD-Umsetzer.
 * Hier wird eine einfache Regelung impelemtiert, um die Ausgangsspanung
 * anhand der gemessenen Spannung über den AD-Umsetzer einzustellen.
 * Dies erfolgt in wenigen Schritten:
 *   1) Abschätzen der Ausgangsspannung
 *   2) Setzen der Ausgangsspannung
 *   3) Messen der Ausgangsspannung mit dem ADC
 *   4) Bilden der Soll/Ist Differenz
 *   5) Umrechnen der Differenz in einen Wert für den DA-Umsetzer
 *   6) Aufaddieren der umgerechneten Differenz auf den aktuellen DAC-Wert
 *   7) Wenn die Differenz 10 insgesamt 10 mal 0 war, beenden
 *   8) Rücksprung zu Punkt 2
 * Param: microvolt: Auszugebene Spannung in Mikrovolt
 * Return: Zuletzt eingestellter DAC-Wert
 */
unsigned short calibrate_set_voltage(unsigned long microvolt)
{
    // Den Startwert abschätzen
    //unsigned short dac_value = microvolt*10/381;
    unsigned short dac_value = microvolt/114;
    unsigned short i;
    unsigned char hit_counter = 0;
    long diff;
    for(i=0; i<1000; i++)
    {
        dac_set_data(dac_value);  // aktuellen Wert setzen
        //sleep(1);  // Einen Moment warten bis der AMP sich eingeschwungen hat
        long voltage = adc_oneshot();
        diff = microvolt - voltage;  // diff in uV
        diff = diff/144; // diff in LSB
        long step = diff/2;  // Maxmal um den halben Fehler nachregeln
        if(step == 0)
        {
            hit_counter++;
            if(diff>0)
                step = 1;
            else
                step = -1;
        }
        dac_value += step;
        if(hit_counter>10)
            return dac_value;
    }
    return dac_value;
}

/*
 * Umrechnung einer Spannung in einen DAC-Wert.
 * Die Umrechnung erfolgt anhand der mit calibrate()
 * aufgenommenen Wertetabelle.
 * Param: microvolt: Zu wandelde Spannung
 * Return: Zur Spannung passender DAC-Wert
 */
unsigned short microvolt2dac(unsigned long microvolt)
{
    // ermitteln des unteren Werts in der Tabelle
    unsigned short i;
    for(i=0; i<CALIBRATION_NUMBER-1; i++)
    {
```

```
        if(mv2dac[i][0] <= microvolt && mv2dac[i+1][0] > microvolt)
            break;
    }
    unsigned long voltage0 = mv2dac[i][0];
    unsigned long voltage1 = mv2dac[i+1][0];
    if(voltage0==microvolt)
        return mv2dac[i][1];

    unsigned long dac0 = mv2dac[i][1];
    unsigned long dac1 = mv2dac[i+1][1];
    unsigned long dv = voltage1 - voltage0;
    unsigned long dd = dac1 - dac0;

    return dac0 + (long)((long long)dd*(long long)(microvolt-voltage0)/(long long)dv);
}
```

A.1.5 config.h

```
/*
 * config.h
 *
 *  Created on: 01.02.2012
 *      Author: Fabian
 */

#ifndef CONFIG_H_
#define CONFIG_H_


// sizeof(config_t) must be an even number! Otherwise the program
// will fail to save and/or reload the configuration!
typedef struct
{
    long current_lo_offset;
    long current_lo_gain;
    long current_hi_offset;
    long current_hi_gain;
    long voltage_offset;
    long voltage_gain;
    long voltage_count;
    short repeat;
} config_t;

extern config_t config;


#define CALIBRATION_NUMBER 13
extern unsigned long mv2dac[CALIBRATION_NUMBER][2];
extern const unsigned long mv2dac_default[CALIBRATION_NUMBER][2];

int config_read(void);
int config_write(void);

// Für die interne Nutzung
void config_erase(void);
int config_save_struct(void* data, long length, long position);
int config_load_struct(void* data, long length, long position);
void config_print_uart(void);

#endif /* CONFIG_H_ */
```

A.1.6 config.c

```
/*
 * config.c
 *
 *  Created on: 01.02.2012
 *      Author: Fabian
 *
 *  Verwalten, Speichern, Laden, Prüfungen der Konfiguration.
 *  Im Flash "0" werden in den ersten 4kB für zwei Strukturen reserviert
 *  in denen Konfigurationen abgelegt werden können, die auch
```

```
 *  nach einem Neustart erhalten bleiben sollen.
 *  Im ersten Struct werden allgemeine Informationen gespeichert,
 *  wie Offset- und Gain-Kalibrierungswerte. Das zweite ist eine Tabelle,
 *  in der Werte für die Umwandlung einer Spannung in einen
 *  DAC-Wert liegen. Bei beiden Strukturen ist es notwendig, dass diese
 *  eine gerade Anzahl an Bytes haben, da es sonst zu Problemen beim
 *  Schreibem auf den Flash kommen kann!
 *  Beide Konfigurationen werden durch einen 32-Bit CRC gesichert.
 */

#include "config.h"
#include "flash.h"
#include "crc.h"
#include <utils/uartstdio.h>

// Instanz der Konfiguration
config_t config;

// Instanz der Wertetabelle für die Umwandlung Spannung -> DAC-Wert
unsigned long mv2dac[CALIBRATION_NUMBER][2];

const unsigned long mv2dac_default[CALIBRATION_NUMBER][2]={
        { 100000, 4369/5},
        { 500000, 4369*1},
        {1000000, 4369*2},
        {1500000, 4369*3},
        {2000000, 4369*4},
        {2500000, 4369*5},
        {3000000, 4369*6},
        {3500000, 4369*7},
        {4000000, 4369*8},
        {4500000, 4369*9},
        {5000000, 4369*10},
        {5500000, 4369*11},
        {6000000, 4369*12}
};

/*
 * Löschen der Konfiguration
 */
void config_erase(void)
{
    flash_erase_block_4k(0, 0x000000);
    flash_wait(0);
}

/*
 * Auslesen und überprüfen der Konfiguration.
 * Nach dem Auslesen der Konfiguration wird diese mit den beiden
 * CRC Werten überprüft. Bei einem Fehler, werden die default-Werte
 * wiederhergestellt.
 * Return: 1=Konfiguration erfolgreich gelesen
 *         0=CRC oder Flash Fehler
 */
int config_read(void)
{
    UARTprintf("Initializing configuration...");
    char rewrite = 0;
    if(!config_load_struct(&config, sizeof(config_t), 0x000000))
    {
        config.current_hi_gain = 0;
        config.current_hi_offset = 0;
        config.current_lo_gain = 0;
        config.current_lo_offset = 0;
        config.voltage_gain = 1000000;
        config.voltage_offset = 0;
        config.voltage_count = 0;
        config.repeat = 0;
        rewrite = 1;
        UARTprintf("new offset/gain configuration created, ");
    }
    if(!config_load_struct(mv2dac, sizeof(mv2dac), sizeof(config_t)+2))
    {
        rewrite = 1;
        memcpy(mv2dac, mv2dac_default, sizeof(mv2dac));  // load default values
        UARTprintf("new voltage2dac configuration created, ");
    }
    if(rewrite)
    {
        config_write();
        // Check the written config again
        if(!config_load_struct(&config, sizeof(config_t), 0x000000) || !config_load_struct(mv2dac, sizeof(mv2dac),
             sizeof(config_t)+2))
        {
```

```
            UARTprintf("ERROR: Unable to save new configurations!");
            return 0;
        }
    }

    config_print_uart();

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

/*
 * Speichern eines Datenblocks im Flash.
 * Es wird zuvor noch ein CRC der Daten erstellt und vorne an die Daten
 * angehängt.
 * Param: data: Pointer auf die zu speichernden Daten
 *        length: Länge der Daten in Bytes
 *        position: Position im Flash 0, nur gerade Zahlen!
 */
int config_save_struct(void* data, long length, long position)
{
    unsigned short crc = crc16((unsigned char*)data, length);

    flash_write_block(0, (char*)&crc, 2, position);
    flash_write_block(0, data, length, position+2);

    return 1;
}

/*
 * Auslesen eines Datenblocks mit Prüfung des CRC.
 * Param: data: Pointer, wo die Daten abgelegt werden sollen
 *        length: Länge der Daten ohne CRC
 *        position: Position der Daten im Flash 0 (zeigt auf CRC)
 * Return: 0=CRC Fehler
 *         1=CRC ok
 */
int config_load_struct(void* data, long length, long position)
{
    unsigned short crc;
    flash_read_block(0, (char*)&crc, 2, position);
    flash_read_block(0, data, length, position+2);

    if(crc16((unsigned char*)data, length) != crc)
        return 0;
    else
        return 1;
}

/*
 * Schreiben der aktuellen Konfiguration.
 * Schreib den Struct config und die Wertetabelle in den
 * Flash 0. Dabei werden alle CRCs erstellt.
 */
int config_write(void)
{
    //flash_status_enable(0);
    //flash_status_write(0, 0x00);

    flash_erase_block_4k(0, 0x000000);
    flash_wait(0);

    config_save_struct(&config, sizeof(config), 0x000000);
    config_save_struct(mv2dac, sizeof(mv2dac), sizeof(config)+2);

    return 1;
}

/*
 * Ausgabe der Konfiguration über UART.
 * Gibt tabellarisch alle Werte der aktuellen Konfiguration über
 * die UART Verbindung aus.
 */
void config_print_uart(void)
{
    int i;
    UARTprintf("\n  config.current_hi_gain: %duV/V\n", config.current_hi_gain);
    UARTprintf("  config.current_hi_offset: %duV\n", config.current_hi_offset);
    UARTprintf("  config.current_lo_gain: %duV/V\n", config.current_lo_gain);
    UARTprintf("  config.current_lo_offset: %duV\n", config.current_lo_offset);
    UARTprintf("  config.voltage_gain: %duV/V\n", config.voltage_gain);
    UARTprintf("  config.voltage_offset: %duV\n", config.voltage_offset);
    UARTprintf("  config.voltage_count: %d\n", config.voltage_count);
```

```
177     UARTprintf("  config.repeat: %d\n", config.repeat);

        UARTprintf("             Volt     DAC\n");
        for(i=0; i<CALIBRATION_NUMBER; i++)
        {
182         UARTprintf("  mv2dac[%d]: %d, %d\n", i, mv2dac[i][0], mv2dac[i][1]);
        }
    }
```

A.1.7 control.h

```
    /*
     * control.h
     *
     *  Created on: 25.12.2011
5    *      Author: fabian
     */

    #ifndef CONTROL_H_
    #define CONTROL_H_
10
    #include <lwip/tcp.h>

    void control_init(void);
    err_t control_new_con(void* arg, struct tcp_pcb* newpcb, err_t err);
15  err_t control_rx(void* arg, struct tcp_pcb* tpcb, struct pbuf* p, err_t err);
    void control_tick();
    void control_interprete_line(char* line);

    typedef enum
20  {
        IDLE,
        CALIBRATE_VOLTAGE_LOWER_GET,
        CALIBRATE_VOLTAGE_UPPER_GET,
        CALIBRATE_CURRENT_LO_GET,
25      CALIBRATE_CURRENT_HI_GET,
        RECEIVE_VOLTAGE
    } control_state_t;

    #endif /* CONTROL_H_ */
```

A.1.8 control.c

```
1   /*
     * control.c
     *
     *  Created on: 25.12.2011
6    *      Author: fabian
     *
     * In dieser Datei befindet sich der Kern des Programms.
     * Hier wird die TCP-Verbindugn initialisiert und verarbeitet.
     * Dabei werden die Befehle, die über die TCP-Verbindung kommen
     * ausgewertet und ausgeführt.
11   */

    #include "control.h"
    #include "fifo8.h"
    #include "calibrate.h"
16  #include "adcdac.h"
    #include "config.h"
    #include "led.h"
    #include "system_timer.h"
    #include "iadc.h"
21  #include "voltage.h"
    #include "sync.h"
    #include "reloader_timer.h"

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

```
struct tcp_pcb* control_tpcb;  // Server-Socket für die TCP-Verbindung
// Stream zur Speicherung aller erhaltenen Daten über den TCP-Socket
fifo8* control_stream;
// TCP-Socket für die aktuelle Steuerungs-Verbindung
struct tcp_pcb* control_connection = (struct tcp_pcb*)NULL;

// Variablen zur Kalibrierung des ADCs.
long voltage_set1;
long voltage_get1;
long voltage_set2;
long voltage_get2;


// String für die Ausgabe bei dem Befehl "help"
const char help_string[] = "Help: \n"
        "  help                     Displays this message\n"
        "  quit                     Closes the TCP control connection\n"
        "  voltagearray             Starts a voltage transmission. After that the\n"
        "                           prompt will change to > and you can send the\n"
        "                           voltage array separated by semicolons, scaled\n"
        "                           in micro Volts. An element EOF terminates the\n"
        "                           transmission of voltages. This typically looks\n"
        "                           like this:\n"
        "                           $voltagearray\n"
        "                           >1000;2000;3000;4000;5000\n"
        "                           >6000;7000;8000;9000;10000;EOF\n"
        "                           $\n"
        "                           You should not send more than 1kB per line,\n"
        "                           otherwise data might be lost. Use voltagecount\n"
        "                           to check for lost data\n"
        "  voltagecount             Gives the number of correct voltages received\n"
        "                           Can be called after voltagearray to verify the\n"
        "                           number of received voltages.\n"
        "  voltageadc <microvolt>   Set the output to <microvolt> micro Volt\n"
        "                           using the ADC as reference.\n"
        "  voltagedac <microvolt>   Set the output to <microvolt> micro Volt\n"
        "                           using the calibrated values as reference.\n"
        "  mode <mode>              Set the operation mode, possible values for\n"
        "                           <mode> are:\n"
        "                             free (default): repeats the transient\n"
        "                                  output set be voltagearray for ever\n"
        "                             single: repeats the transient output once.\n"
        "                                  The output will keep the last sample.\n"
        "  start                    Initializes the transient output. This has to\n"
        "                           be send to all cards, including the master!\n"
        "                           One card has to become master. Use sync\n"
        "                           for that.\n"
        "  sync                     Will start the sync output. This must be\n"
        "                           called on one card to synchronize the\n"
        "                           outputs. There will be NO output until one\n"
        "                           card becomes master!\n"
        "  stop                     Stops the transient output. The output will\n"
        "                           keep the last value. This command can only be\n"
        "                           send to the master, which also got the sync\n"
        "                           command.\n"
        "  config                   Print the current configuration\n";


/*
 * Initialisierung der Kommunikation.
 * Hier wird der Server-TCP-Socket und der UDP-Socket initialisiert.
 * Zudem wird auch der FIFO für die Befehle initialisiert.
 */
void control_init(void)
{
    UARTprintf("Initializing control...");
    control_stream = fifo8_create(5000);
    control_tpcb = tcp_new();
    tcp_bind(control_tpcb, IP_ADDR_ANY, 56936);
    control_tpcb = tcp_listen(control_tpcb);
    tcp_accept(control_tpcb, &control_new_con);
    reload_fill = 0;
    UARTprintf("done\n");
}

/*
 * "Interrupt" bei Erhalt einer neuen TCP-Verbindung.
 * Diese Funktion wird vom lwIP-Stack ausgeführt, wenn eine
 * neue TCP-Verbindung angefordert wird. Zunächst wird eine evtl.
 * bestehende alte Verbindung getrennt. Anschließend wird die
 * neue Verbindung angenommen.
 */
err_t control_new_con(void* arg, struct tcp_pcb* newpcb, err_t err)
{
```

```
    if(control_connection)
        tcp_close(control_connection);
    control_connection = newpcb;
    tcp_accepted(newpcb);
    tcp_recv(newpcb, &control_rx);
    UARTprintf("New control connection established!\n");
    tcp_write(control_connection, "$", 1, TCP_WRITE_FLAG_COPY | TCP_WRITE_FLAG_MORE);
    return ERR_OK;
}

/*
 * "Interrupt" bei Erhalt von Daten über den TCP-Socket.
 * Diese Funktion wird vom lwIP-Stack ausgeführt, wenn neue
 * Daten über den TCP-Socket eingetroffen sind.
 * Die Daten werden hier zunächst im FIFO zwischengespeichert.
 * Anschließend wird die Funktion control_tick() ausgeführt, um
 * die erhaltenen Daten auszuwerten.
 */
err_t control_rx(void* arg, struct tcp_pcb* tpcb, struct pbuf* p, err_t err)
{
    //led_on(1);
    if(p == NULL)
    {
        UARTprintf("Control connection closed!\n");
        control_connection = NULL;
        return ERR_OK;
    }
    struct pbuf* current_p=p;
    //while(current_p->next)
    //  current_p=current_p->next;
    do
    {
        fifo8_push_elements(control_stream, (char*)current_p->payload, current_p->len);
    } while((current_p=current_p->next) != NULL);
    //process_commands(tpcb);
    control_tick();
    tcp_recved(tpcb, p->tot_len);
    pbuf_free(p);
    //led_off(1);
    return ERR_OK;
}

/*
 * Auswertung der im FIFO enthaltenen Befehle.
 * Es wird zunächst geprüft, ob eine vollständige Zeile
 * im FIFO vorhanden ist. Wenn ja, wird diese an die Funktion
 * control_interprete_line() übergeben. Wenn nicht, beendet
 * sich die Funktion einfach.
 */
void control_tick()
{
    long i;
    char hadCommand = 1;
    while(hadCommand)
    {
        hadCommand = 0;
        for(i = 0; i < fifo8_used(control_stream); i++)
        {
            char element = fifo8_read_element(control_stream, i);
            if(element == '\n' || element == '\r')
            {
                // There is a new command, read it
                char* line = (char*)malloc(i+1);
                fifo8_pop_elements(control_stream, line, i);
                line[i] = 0;
                //UARTprintf("New line: %s\n", line);
                control_interprete_line(line);
                free(line);
                hadCommand = 1;
                while(fifo8_read_element(control_stream, 0) == '\r' || fifo8_read_element(control_stream, 0) == '\n
                      ')
                    fifo8_pop_element(control_stream);
                break;
            }
        }
    }
}

/*
 * Interpretierung einer einzelnen Anweisung.
 * Es ist unbedingt zu beachten, dass diese Funktion noch
 * innerhalb des Ethernet-Interrupts ausgeführt wird, also
 * auf keinen Fal zu lange blockieren darf, da sonst Programme
 * wie Putty oder Telnet die Verbindung nach einem Timeout
```

```
 * beenden.
 */
void control_interprete_line(char* line)
{
    unsigned long start, stop;
    long i, currentStart;
    long lineLength;
    char buffer[100];
    long current;
    float gainf, offsetf;
    long gain, offset;
    char* argv[10];
    unsigned long argc;
    static control_state_t state = IDLE;  // Zustandsvariable für den Automaten
    if(state != RECEIVE_VOLTAGE)
        UARTprintf("New line: %s\n", line);
    led_set_mode(0, MODE_SINGLE);

    switch(state)
    {
    /*
     * IDLE: Es wird ein neuer normaler Befehl erwartet.
     * Die Zeile wird zunächst in die aus C bekannten Variablen
     * argv und argc zerlegt, um eine einfache Interpretierung
     * zu ermöglichen.
     */
    case IDLE:
        //argv[0]=line;

        // split the line into typical known variables argc and argv
        argc = 0;
        currentStart = 0;
        lineLength = strlen(line);
        for(i=0; i<=lineLength; i++)
        {
            if(line[i]==' ' || line[i]==0)
            {
                line[i]=0;
                if(currentStart==i)
                {
                    currentStart++;
                    continue;
                }
                argv[argc] = &line[currentStart];
                currentStart = i+1;
                argc++;
            }
        }

        // Ausgabe des interpretierten Befehls über UART
        for(i=0; i<argc; i++)
        {
            UARTprintf("argv[%d]: %s\n", i, argv[i]);
        }

        // Wenn dies eine leere Zeile war, abbrechen
        if(argc==0)
            break;

        // quit/exit
        if(strcmp(argv[0], "quit") == 0 || strcmp(argv[0], "close") == 0)
        {
            tcp_close(control_connection);
            control_connection = NULL;
            fifo8_clear(control_stream);
        }
        // identify
        else if(strcmp(argv[0], "identify") == 0)
        {
            UARTprintf("identifying this card by blinking\n");
            led_set_mode(0, MODE_IDENT);
        }
        // calibrate
        /*
         * Kalibierung des AD- und DA-Umsetzers mit Verstärker etc.
         * Da das ganze interruptbasiert ist, wird primär hierfür
         * der Automat benötigt.
         */
        else if(strcmp(argv[0], "calibrate")==0)
        {
            UARTprintf("Doing a calibration...\n");
            // Setzen der Konfigurationswerte auf den default Wert
            config.voltage_gain = 1000000;
            config.voltage_offset = 0;
```

```
            // Setzen der Ausgangsspannung auf 500mV
            calibrate_set_voltage(500000);
            voltage_set1 = adc_oneshot();
            UARTprintf("Voltage set: %d\n", voltage_set1);
            state = CALIBRATE_VOLTAGE_LOWER_GET;
        }
        // voltagearray
        /*
         * Übertragen von Spannungswerten, die im Flash
         * gespeichert werden sollen.
         * Auch hierfür wird der Automat benötigt, da die Übertragung
         * interruptbasiert ist. Nach dem Befehl können Spannungen
         * in Mikrovolt kodiert und durch ein Semikolon getrennt
         * übertragen werden. Jede Zeile wird mit einem > bestätigt.
         */
        else if(strcmp(argv[0], "voltagearray")==0)
        {
            state = RECEIVE_VOLTAGE;
            voltage_reset();  // Rücksetzen der Spannungsprogrammierung/-wiedergabe
            UARTprintf("waiting for voltages...\n");
        }
        // voltagecount
        /*
         * Ausgabe der Anzahl der empfangenen Spannung nach dem Befehl
         * voltagecount. Der Wert wird über den TCP-Socket ausgegeben.
         */
        else if(strcmp(argv[0], "voltagecount") == 0)
        {
            UARTprintf("Sending voltage count to control client: %d\n", config.voltage_count);
            sprintf(buffer, "%d\n", config.voltage_count);
            tcp_write(control_connection, buffer, strlen(buffer), TCP_WRITE_FLAG_COPY | TCP_WRITE_FLAG_MORE);
        }
        // help
        else if(strcmp(argv[0], "help") == 0)
        {
            UARTprintf("help requested...");
            tcp_write(control_connection, help_string, strlen(help_string), TCP_WRITE_FLAG_COPY |
                 TCP_WRITE_FLAG_MORE);
        }
        // voltagedac
        /*
         * Ausgabe der in Mikrovolt kodierten Spannung im ersten
         * Parameter über die Look-Up-Tabelle.
         */
        else if(strcmp(argv[0], "voltagedac") == 0)
        {
            if(argc != 2)
            {
                UARTprintf("Incorrect number of parameters for command \"voltagedac\"!\n");
                break;
            }
            int voltage = atoi(argv[1]);
            if(voltage<100000 || voltage > 6000000)
            {
                UARTprintf("Wrong voltage value: Value must be between 500000 and 6000000\n");
                break;
            }
            dac_set_data(microvolt2dac(voltage));
        }
        // voltageadc
        /*
         * Ausgabe der in Mikrovolt kodierten Spannung im ersten
         * Parameter über Rückkopplung über den ADC.
         */
        else if(strcmp(argv[0], "voltageadc") == 0)
        {
            if(argc != 2)
            {
                UARTprintf("Incorrect number of parameters for command \"voltageadc\"!\n");
                break;
            }
            int voltage = atoi(argv[1]);
            // begrenzen der Spannung nach oben und unten
            if(voltage<100000 || voltage > 6000000)
            {
                UARTprintf("Wrong voltage value: Value must be between 500000 and 6000000\n");
                break;
            }
            calibrate_set_voltage(voltage);
        }
        // mode
        /*
         * Setzen des Ausgabemodus. Wechsel zwischen einmaliger und wiederholter
         * Ausgabe der Spannungssequenz.
```

```
         * Param: "free"=Wiederholende Ausgabe
         *        "single"=Einmalige Ausgabe
         */
        else if(strcmp(argv[0], "mode") == 0)
        {
            if(argc != 2)
            {
                UARTprintf("Incorrect number of parameters for command \"mode\"!\n");
                break;
            }
            if(strcmp(argv[1], "free") == 0)
            {
                UARTprintf("Switching to free run mode\n");
                config.repeat = 1;
            }
            else if(strcmp(argv[1], "single") == 0)
            {
                UARTprintf("Switching to single run mode\n");
                config.repeat = 0;
            }
            else
            {
                UARTprintf("Wrong mode value\n");
            }
        }
        // config
        /*
         * Ausgabe der aktuellen Konfiguration über die UART Schnittstelle
         */
        else if(strcmp(argv[0], "config") == 0)
        {
            config_print_uart();
        }
        // start
        /*
         * Vorbereiten einer Transientenwiedergabe.
         * Es wird der FIFO für die Wiedergabesamples initialisiert
         * und mit ersten Werten geladen. Dem reload timer wird gesagt,
         * dass er ab sofort Samples nachladen soll.
         */
        else if(strcmp(argv[0], "start") == 0)
        {
            reload_fill = 0;
            voltage_start();
            reload_fill = 1;
        }
        // sync
        /*
         * Start des Synchronisationstimers.
         * Dieser Befehl muss nach start auf dem Master ausgeführt werden.
         */
        else if(strcmp(argv[0], "sync") == 0)
        {
            sync_out_enable();
        }
        // stop
        /*
         * Stoppen des Synchronisationstimers.
         */
        else if(strcmp(argv[0], "stop") == 0)
        {
            sync_out_disable();
            reload_fill = 0;
        }
        // voltage
        /*
         * Messung der Spannung am Ausgang über externen ADC.
         * Die Ausgangsspannung wird über den externen ADC
         * gemessen und über den TCP-Socket zurück gegeben.
         */
        else if(strcmp(argv[0], "voltage") == 0)
        {
            long voltage = adc_oneshot();
            UARTprintf("Sending voltage to control client: %d\n", voltage);
            sprintf(buffer, "%d\n", voltage);
            tcp_write(control_connection, buffer, strlen(buffer), TCP_WRITE_FLAG_COPY | TCP_WRITE_FLAG_MORE);
        }
        // current
        /*
         * Messung des aktuellen Stromes.
         * Der aktuell fließende Strom wird über den internen
         * ADC gemessen, umgewandelt und sowohl über UART
         * als auch die TCP-Verbindung gesendet.
         */
```

```
        else if(strcmp(argv[0], "current") == 0)
        {
            iadc_oneshot();
            UARTprintf("Sending current to control client: %d, %d\n", current_hi, current_lo);
            sprintf(buffer, "%d\n", current_lo);
            tcp_write(control_connection, buffer, strlen(buffer), TCP_WRITE_FLAG_COPY | TCP_WRITE_FLAG_MORE);
        }
        break;

        /*
         * RECEIVE_VOLTAGE: Es werden Zeilen mit Spannungswerten erwartet.
         * Die Spannugnswerte sind in ADCII kodiert in Mikrovolt und
         * durch ein Semikolon getrennt. Wird die Sequenz EOF gelesen,
         * ist dies das Ende der Übertragungen von Spannungen.
         */
    case RECEIVE_VOLTAGE:
        start = 0;
        lineLength = strlen(line);
        while(1)
        {
            char* str_element;
            stop = start;
            // Heraussuchen des nächsten Semikolons oder ende der Zeile
            while(line[stop]!=';' && stop<lineLength)
                stop++;

            // Beenden des elementes mit \0
            line[stop] = 0;
            // str_element enthält Zeiger auf Element
            str_element = &line[start];

            // Prüfen, ob dies das Ende der Übertragung ist
            if(str_element[0] == 'E' && str_element[0] != '\0' && str_element[1] == 'O' && str_element[1] != '\0'
                  && str_element[2] == 'F' && str_element[2] != '\0')
            {
                state = IDLE;
                if(voltage_count == 0)
                {
                    //UARTprintf("fill: %d\n", reload_fill);
                    UARTprintf("THIS SHOULD NEVER HAPPEN!");
                }
                UARTprintf("Received voltages: %d\n", voltage_count);
                voltage_finish();
                break;
            }
            else
            {
                // Wenn das Ende noch nicht erreicht ist, ist in buffer
                // nun ein Zahlenwert, der umgewandelt wird und im FIFO
                // gepseichert wird.
                unsigned long voltage = atoi(str_element);
                //UARTprintf(".");
                //UARTprintf("Received voltage: %d\n", voltage);
                //UARTprintf("%d ", voltage);
                voltage_add(voltage);
                //UARTprintf("%d, ", voltage);
                // Wenn dies das Ende der Zeile war, die Interpretierung
                // beenden.
                if(stop==lineLength)
                    break;
            }
            start = stop+1;
        }
        break;
        /*
         * CALIBRATE_VOLTAGE_LOWER_GET: Der Nutzer hat den Wert bei der Kalibrierung
         * für den unteren Wert von 500mV eingegeben.
         * Es ist nun die gesetzte Spannung und die "tatsächliche" Spannung für etwa
         * 500mV bekannt. Nun wird der gleiche Vorgang nochmals für 5,5V druchgeführt.
         */
    case CALIBRATE_VOLTAGE_LOWER_GET:
        voltage_get1 = atoi(line);

        // Setzen der Ausgangsspannung von 5,5V nach dem ADC
        calibrate_set_voltage(5500000);

        // Messen der aktuellen Spannung (diese sollte sehr exakt
        // 5,5V betragen, hier könnte man evtl. nochmals eine
        // Plausibilitätsprüfung einbauen)
        voltage_set2 = adc_oneshot();

        // In den nächsten Zustand wechseln
        state = CALIBRATE_VOLTAGE_UPPER_GET;
        break;
```

```
531         /*
            * CALIBRATE_VOLTAGE_UPPER_GET: Der Nutzer hat auch die zweite externe Spannung
            * von 5,5V gemessen und eingegeben.
            * Nun sind Zwei gesetzte Spannungen und zwei dazugehörige extern gemessene
            * Spannungen bekannt, woraus ein Offset- und Verstärkungsfehler berechnet
536         * werden kann. Diese beiden Korrekturwerte werden in der Konfiguration
            * abgespeichert und anschlie ß end wird diese in den Flash geschrieben.
            */
        case CALIBRATE_VOLTAGE_UPPER_GET:
            voltage_get2 = atoi(line);
541         UARTprintf("Voltage set2: %d\n", voltage_set2);
            UARTprintf("Voltage get2: %d\n", voltage_get2);

            // Berechnung des Verstärkungsfehlers
            gainf = ((float)voltage_get2-(float)voltage_get1)/((float)voltage_set2-(float)voltage_set1)*1000000.0f;
546         gain = (long)gainf;
            UARTprintf("Gain: %d uV/Volt\n", gain);
            config.voltage_gain = gain;

            // Berechnung des Offsetfehlers
551         offsetf = (float)voltage_get1-gainf*(float)voltage_set1/1000000.0f;
            offset = (long)offsetf;
            config.voltage_offset = offset;
            UARTprintf("Voltage set1: %d\n", voltage_set1);
            UARTprintf("Voltage get1: %d\n", voltage_get1);
556         UARTprintf("Offset: %d\n", config.voltage_offset);

            // Durchführung der Kalibrierung der Strecke vom DA-Umsetzer bis zum
            // Verstärker. Diese Kalibrierung dauert sehr lange, es sollte in
            // Erwägung gezogen werden die nicht im Ethernet-Interrupt zu erledigen.
561         calibrate();  // Do the calibration of the DAC


            // Kalibrierung des Stromes
            // Dies ist noch nicht implementiert bzw. ungetestet und daher auskommentiert
566         /*dac_set_data(microvolt2dac(1000000));  // The output should be open
            sleep(10);

            iadc_oneshot();
            config.current_hi_offset = current_hi;
571         config.current_lo_offset = current_lo;*/

            config_write();
            state = IDLE;
            //state = CALIBRATE_CURRENT_LO_GET;
576         break;

        // Im Moment nicht in Verwendung!
        case CALIBRATE_CURRENT_LO_GET:  // The user should have connected a 1k Ohm resistor and entered the current in
              uA which is flowing
            current = atoi(line);
581         iadc_oneshot();
            config.current_lo_gain = current_lo*1000000/current; // in uA/1000000uA

            state = CALIBRATE_CURRENT_HI_GET;

586         // Now the user should plug in a 10 Ohm resistor, 100mA should flow
            break;

        // Im Moment nicht in Verwendung!
        case CALIBRATE_CURRENT_HI_GET:  // The user should have entered a current value in uA
591         current = atoi(line);
            iadc_oneshot();
            config.current_hi_gain = current_hi*1000000/current;  // in mA/1000000mA

            config_write();
596         state = IDLE;
            break;
        }

        // Ausgabe der Bestätigung
601     // Je nach Befehl werden verschiedene Bestätigungen an den Nutzer gesendet.
        // Werden weitere Parameter erwatet wird ein > gesendet, wird ein normaler
        // Befehl erwartet ein Dollarzeichen.
        switch(state)
        {
606     case RECEIVE_VOLTAGE:
        case CALIBRATE_VOLTAGE_LOWER_GET:
        case CALIBRATE_VOLTAGE_UPPER_GET:
        case CALIBRATE_CURRENT_LO_GET:
        case CALIBRATE_CURRENT_HI_GET:
611         tcp_write(control_connection, ">", 1, TCP_WRITE_FLAG_COPY);
            break;
        default:
```

```
        tcp_write(control_connection, "$", 1, TCP_WRITE_FLAG_COPY);
        break;
    }
}
```

A.1.9 crc.h

```
/*
 * crc.h
 *
 *  Created on: 01.02.2012
 *      Author: Fabian
 */

#ifndef CRC_H_
#define CRC_H_

#define CRC16_POLYNOM       0x1021

unsigned short crc16_internal(unsigned char data_in, unsigned short data_out);
unsigned short crc16(unsigned char* buffer, unsigned char length);

#endif /* CRC_H_ */
```

A.1.10 crc.c

```
/*
 * crc.c
 *
 *  Created on: 01.02.2012
 *      Author: Robert Ostermann, Fabian Schwartau
 */

#include "config.h"
#include <stdio.h>
#include "crc.h"
#include "uart.h"

/*
 * Funktion zum berechnen eines 16 Bit CRC.
 * Die Funktion berechnet einen 16 Bit CRC mit dem Polynom
 * CRC16_POLYNOM von beliebigen und beliebig langen Daten.
 */
unsigned short crc16(unsigned char* buffer, unsigned char length)
{
    unsigned char i;
    unsigned short crc = 0;

    for(i=0; i<length; i++)
    {
        crc = crc16_internal(buffer[i], crc);
    }
    return crc;
}


/*
 * Intern verwendete Funktion zum inkrementellen berechnen des CRC
 */
unsigned short crc16_internal(unsigned char data_in, unsigned short data_out)
{
    unsigned char i;
    for(i=8; i>0; i--)
    {
        if((data_out>>15)&0x0001 != (data_in>>(i-1))&0x01)
        {
            data_out <<= 1;
            data_out ^= CRC16_POLYNOM;
        }
        else
            data_out <<= 1;
```

```
    }

    return data_out;
}
```

A.1.11 discover.h

```
/*
 * discover.h
 *
 *  Created on: 24.12.2011
 *      Author: fabian
 */

#ifndef DISCOVER_H_
#define DISCOVER_H_

#include <lwip/udp.h>
#include <inc/lm3s9b92.h>
#include <utils/uartstdio.h>

void discover_init(void);
void discover_rx(void* arg, struct udp_pcb* upcb, struct pbuf* p, struct ip_addr* addr, u16_t port);
void ipToString(char* string, unsigned long ip, int maxLen);

#endif /* DISCOVER_H_ */
```

A.1.12 discover.c

```
/*
 * discover.c
 *
 *  Created on: 24.12.2011
 *      Author: fabian
 *
 *  In dieser Datei werden die Funktionen zum Auffinden der Module
 *  über eine Ethernet Broadcast  Nachricht definiert. Hierfür wird
 *  ein UDP-Socket erstellt und initialisiert, der auf dem Port
 *  56936 allen ankommenden Paketen eine Antwort mit der IP-Adresse
 *  des Moduls sendet.
 */

#include "discover.h"

#include <string.h>
#include <stdio.h>
#include <utils/lwiplib.h>

// UDP-Socket für die UDP-Nachrichten
struct udp_pcb* g_upcb;

/*
 * Initialisierung des UDP-Sockets.
 */
void discover_init(void)
{
    volatile unsigned long ulLoop;
    UARTprintf("Initializing discover...");
    g_upcb = udp_new();
    err_t error = udp_bind(g_upcb, IP_ADDR_ANY, 56936);
    if(error != ERR_OK)
    {
        UARTprintf("failed: Unable to bind udp socket for discover: %d\n", error);
        return;
    }
    udp_recv(g_upcb, &discover_rx, (void*)0);
    UARTprintf("done\n");
}

/*
 * "Interrupt" beim Erhalten eines neuen Pakets.
```

```
 * Diese Funktion wird durch den lwIP-Stack ausgeführt, wenn ein
 * neues UDP-Paket auf dem Socket g_upcb angekommen ist.
 * Die Funktion antwortet jedem Paket mit der IP-Adresse des Moduls.
 */
void discover_rx(void* arg, struct udp_pcb* upcb, struct pbuf* p, struct ip_addr* addr, u16_t port)
{
    char buffer[30];
    UARTprintf("Got discovery packet:\n");

    UARTprintf("  length: %d\n", p->len);
    UARTprintf("  payload: %s\n", p->payload);

    //udp_connect(g_upcb, addr, port);
    ipToString(buffer, lwIPLocalIPAddrGet(), 29);
    struct pbuf* ipPacket = pbuf_alloc(PBUF_TRANSPORT, strlen(buffer), PBUF_RAM);
    strcpy((char*)ipPacket->payload, buffer);
    udp_sendto(g_upcb, ipPacket, addr, port);

    UARTprintf("  SRC IP: %d.%d.%d.%d\n", addr->addr & 0xff, (addr->addr >> 8) & 0xff, (addr->addr >> 16) & 0xff, (
        addr->addr >> 24) & 0xff);
    UARTprintf("  SRC Port: %d\n", port);
    UARTprintf("  Answer sent\n");
    /*UARTprintf("  SRC2 IP: %d.%d.%d.%d\n", upcb->remote_ip.addr & 0xff, (upcb->remote_ip.addr >> 8) & 0xff, (upcb
        ->remote_ip.addr >> 16) & 0xff, (upcb->remote_ip.addr >> 24) & 0xff);
    UARTprintf("  SRC2 Port: %d\n", upcb->remote_port);
    UARTprintf("  DST IP: %d.%d.%d.%d\n", upcb->local_ip.addr & 0xff, (upcb->local_ip.addr >> 8) & 0xff, (upcb->
        local_ip.addr >> 16) & 0xff, (upcb->local_ip.addr >> 24) & 0xff);
    UARTprintf("  DST Port: %d\n", upcb->local_port);*/
    pbuf_free(p);
    pbuf_free(ipPacket);
}

/*
 * Funktion zur Konvertierung einer in einem unsigned long gespeicherten
 * IP-Adresse in einen String.
 */
void ipToString(char* string, unsigned long ip, int maxLen)
{
    sprintf(string, "%d.%d.%d.%d", ip & 0xff, (ip >> 8) & 0xff, (ip >> 16) & 0xff, (ip >> 24) & 0xff);
}
```

A.1.13 ethernet.h

```
/*
 * ethernet.h
 *
 *  Created on: 24.12.2011
 *      Author: fabian
 */

#ifndef ETHERNET_H_
#define ETHERNET_H_

#include <utils/lwiplib.h>
#include <utils/uartstdio.h>

#include "discover.h"
#include "control.h"

//*****************************************************************************
//
// Defines for setting up the system clock.
//
//*****************************************************************************
#define SYSTICKHZ               100
#define SYSTICKMS               (1000 / SYSTICKHZ)

int ethernet_init(void);
void lwIPHostTimerHandler(void);
void SysTickIntHandler(void);

#endif /* ETHERNET_H_ */
```

A.1.14 ethernet.c

```
/*
 * ethernet.c
 *
 *  Created on: 24.12.2011
 *      Author: fabian
 *  Hier findet die Initialisierung des Ethernet Controllers
 *  und die Adresszuweisung für Ethernet statt. Nachdem
 *  Ethernet initialisiert wurde und eine IP Adresse erhalten
 *  hat, werden von hier aus die Initialisierungsfunktionen
 *  für den TCP-Steuerungsport und die UDP Broadcastnachrichten
 *  aufgerufen.
 */

#include "ethernet.h"
#include "system_timer.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 <inc/hw_ints.h>
#include <driverlib/interrupt.h>
#include <lwip/ip_addr.h>

// Zur Anzeige einer Aktivität in der Konsole
static char g_pcTwirl[4] = { '\\', '|', '/', '-' };
static unsigned long g_ulTwirlPos = 0;

// Aktuelle IP Adresse
static unsigned long g_ulLastIPAddr = 0;

unsigned long ulUser0, ulUser1;
// MAC Adresse
unsigned char pucMACArray[8];

int ethernet_init(void)
{
    UARTprintf("Initializing ethernet:\n");

    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOE;
    volatile unsigned long ulLoop;
    ulLoop = SYSCTL_RCGC2_R;

    GPIO_PORTE_DIR_R = 0x00;  // Auf Input setzen
    GPIO_PORTE_DEN_R = 0xFF;  // Einschalten der GPIOs
    GPIO_PORTE_PUR_R = 0xFF; // Pull Ups aktivieren

    // Einschalten und Resetten des Eternet Controllers
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ETH);
    ROM_SysCtlPeripheralReset(SYSCTL_PERIPH_ETH);

    // Aktivieren von Port F für die Ethernet LEDs
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinConfigure(GPIO_PF2_LED1);
    GPIOPinConfigure(GPIO_PF3_LED0);
    GPIOPinTypeEthernetLED(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);

    // Konfiguration von SysTick für einen periodischen Interrupt
    ROM_SysTickPeriodSet(ROM_SysCtlClockGet() / SYSTICKHZ);
    ROM_SysTickEnable();
    ROM_SysTickIntEnable();

    // ROM_IntMasterEnable();  // Enable processor interrupts.
    IntPrioritySet(INT_ETH, 0xE0); // lowes priority
    IntPrioritySet(INT_SYSCTL, 0xE0);

    // Lesen der MAC-Adresse aus den User-Registern
    // Diese scheint ab Werk programmiert zu sein
    ROM_FlashUserGet(&ulUser0, &ulUser1);
    if((ulUser0 == 0xffffffff) || (ulUser1 == 0xffffffff))
    {
        // We should never get here.  This is an error if the MAC address has
        // not been programmed into the device.  Exit the program
        UARTprintf("  MAC Address Not Programmed!\n");
        return 0;
    }

    // Umwandlung der 24/24 MAC Adresse aus dem NV RAM in eine 32/16 MAC
    // Adresse. Anchlie ß end wird die MAC dem Ethernet Controller
```

```
    // übergeben
    pucMACArray[0] = ((ulUser0 >>  0) & 0xff);
    pucMACArray[1] = ((ulUser0 >>  8) & 0xff);
    pucMACArray[2] = ((ulUser0 >> 16) & 0xff);
    pucMACArray[3] = ((ulUser1 >>  0) & 0xff);
    pucMACArray[4] = ((ulUser1 >>  8) & 0xff);
    pucMACArray[5] = ((ulUser1 >> 16) & 0xff);

    // Initialisierung von lwIP, mit DHCP.
    //lwIPInit(pucMACArray, 0, 0, 0, IPADDR_USE_DHCP);

    // Initialisierung von lwIP, mit statischer IP.
    unsigned long lowerIP = GPIO_PORTE_DATA_R;
    struct ip_addr local_addr;
    IP4_ADDR(&local_addr,lowerIP,0,168,192);
    lwIPInit(pucMACArray, local_addr.addr, 0xFFFFFF00, 0, IPADDR_USE_STATIC);

    // Setup the device locator service
    LocatorInit();
    LocatorMACAddrSet(pucMACArray);
    LocatorAppTitleSet("EK-LM3S9B92 enet_lwip");

    return 1;
}


/*
 * Timer Interrupt für Ethernet spezifische Aktionen.
 * Hier wird u.a. erkannt, wenn eine neue IP-Adresse zugewiesen
 * wurde. Die Neue IP wird dann über UART ausgegeben.
 */
void lwIPHostTimerHandler(void)
{
    unsigned long ulIPAddress;

    // Get the local IP address.
    ulIPAddress = lwIPLocalIPAddrGet();

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

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

    // Check if IP address has changed, and display if it has
    else if(ulIPAddress != g_ulLastIPAddr)
    {
        // Ausgabe der neuen MAC Adresse
        UARTprintf("\n  MAC: %x:%x:%x:%x:%x:%x\n", pucMACArray[0],
                pucMACArray[1], pucMACArray[2], pucMACArray[3],
                pucMACArray[4], pucMACArray[5]);
        //
        // Ausgabe der neuen IP Adresse
        UARTprintf("  IP: %d.%d.%d.%d\n", ulIPAddress & 0xff,
                (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
                (ulIPAddress >> 24) & 0xff);

        // Speichern der neuen IP-Adresse
        g_ulLastIPAddr = ulIPAddress;

        // Ausgabe der neuen Netzmaske
        ulIPAddress = lwIPLocalNetMaskGet();
        UARTprintf("  Netmask: %d.%d.%d.%d\n", ulIPAddress & 0xff,
                (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
                (ulIPAddress >> 24) & 0xff);

        //
        // Ausgabe der neuen Gateway Adresse
        //
        ulIPAddress = lwIPLocalGWAddrGet();
        UARTprintf("  Gateway: %d.%d.%d.%d\n", ulIPAddress & 0xff,
                (ulIPAddress >> 8) & 0xff, (ulIPAddress >> 16) & 0xff,
                (ulIPAddress >> 24) & 0xff);

        discover_init();  // Initialisierung des UDP Broadcast Dienstes
        control_init();   // Initialisierung der TCP-Steuerung

        // Initialisierung abgeschlossen, einschalten der grünen LED
        led_set_mode(0, MODE_ON);
```

```
166         led_set_mode(1, MODE_OFF);

            UARTprintf("Initialization complete..\n");
        }
    }
171
    /*
     * System-Timerinterrupt für Ethernet
     */
    void SysTickIntHandler(void)
176 {
        lwIPTimer(SYSTICKMS);
    }
```

A.1.15 fifo16.h

```
1   /*
     * fifo.h
     *
     *  Created on: 25.12.2011
     *      Author: fabian
6    */

    #ifndef FIFO16_H_
    #define FIFO16_H_

11  typedef struct
    {
        volatile unsigned short* buffer;
        volatile long readPtr;
        volatile long writePtr;
16      volatile long size;
    } fifo16;


    fifo16* fifo16_create(long size);
21  void fifo16_delete(fifo16* ff);
    long fifo16_used(fifo16* ff);
    long fifo16_free(fifo16* ff);
    void fifo16_clear(fifo16* ff);
    void fifo16_push_elements(fifo16* ff, unsigned short* data, long size);
26  void fifo16_pop_elements(fifo16* ff, unsigned short* data, long size);
    void fifo16_read_elements(fifo16* ff, unsigned short* data, long size);
    unsigned short fifo16_read_elememt(fifo16* ff, long pos);
    unsigned short fifo16_pop_element(fifo16* ff);

31  #endif /* FIFO_H_ */
```

A.1.16 fifo16.c

```
    /*
     * fifo.c
     *
4    *  Created on: 25.12.2011
     *      Author: fabian
     *  Implementierung eines FIFOs mit variabler Grö ß e und
     *  festen 2 Byte pro Element. Der FIFO ist in sofern
     *  interruptsicher, dass ein Interrupt lesen und der
9    *  andere schreiben darf, ohne dass Kollisionen auftreten.
     *  Zum erstellen des FIFOs wird der HEAP verwendet,
     *  es ist also darauf zu achten, dass beim ständigen
     *  neu erstellen keine Fragmentierung im Speicher
     *  auftritt.
14   */

    #include "fifo16.h"

    #include <stdlib.h>
19
    /*
     * Erstellung eiens neuen FIFOs.
```

```
 * Mit dieser Funktion kann ein neuer FIFO erstellt und
 * initialisiert werden.
 */
fifo16* fifo16_create(long size)
{
    fifo16* ff = (fifo16*)malloc(sizeof(fifo16));
    if(ff==NULL)
        return NULL;
    ff->size = size;
    ff->readPtr = 0;
    ff->writePtr = 0;
    ff->buffer = (unsigned short*)malloc(sizeof(unsigned short)*(size+1));  // one more element needed
    if(ff->buffer == NULL)
    {
        free(ff);
        return NULL;
    }
    return ff;
}

/*
 * Löschen des FIFOs.
 * Mit dieser Funktion können die reservierten Speicher
 * für den FIFO wieder freigegeben werden. Hierzu zählt
 * zum einen der Puffer des FIFOs und zum anderen der
 * FIFO-Struct selbst.
 */
void fifo16_delete(fifo16* ff)
{
    free((void*)ff->buffer);
    free(ff);
}

/*
 * Zurücksetzen des FIFOs.
 * Die read und write Pointer werden auf 0 zurück gesetzt,
 * was bedeutet, dass der FIFO leer ist.
 */
void fifo16_clear(fifo16* ff)
{
    ff->readPtr = 0;
    ff->writePtr = 0;
}

/*
 * Aktuell belgeter Speicher im FIFO.
 * Diese Funktion ermittelt anhand der beiden Lese- und
 * Schreibzeiger wie viele Bytes aktuell im FIFO belegt
 * sind.
 */
long fifo16_used(fifo16* ff)
{
    long used = ff->writePtr-ff->readPtr;
    if(used<0)
        used += ff->size;
    return used;
}

/*
 * Freier Platz im FIFO.
 * Ermittelt mit Hilfe der Funktion fifo16_used() und der
 * Grö ß e des FIFOs wie viel Platz noch im FIFO frei ist.
 */
long fifo16_free(fifo16* ff)
{
    return ff->size-fifo16_used(ff);
}

/*
 * Anfügen von einer Liste an Elementen.
 * Diese Funktion fügt die in data enthaltenden Elemente
 * an den FIFO an.
 * Param: data: Pointer auf anzufügendende Elemente
 *        size: Anzahl der Elemente
 */
void fifo16_push_elements(fifo16* ff, unsigned short* data, long size)
{
    long i;
    if(fifo16_free(ff)<=size)
        return;
    for(i=0; i<size; i++)
    {
        ff->buffer[ff->writePtr] = data[i];
        ff->writePtr++;
```

```
        if(ff->writePtr == ff->size)
            ff->writePtr = 0;
    }
}

/*
 * Auslesen einer Liste von Elementen.
 * Liest aus dem FIFO "size" Elemente aus und speichert
 * diese in data.
 * Sind nicht genügend Daten vorhanden werden keine Daten
 * ausgelesen.
 * Param: data: Pointer auf Speicher in dem die Elemente
 *              abgelegt werden sollen.
 *        size: Anzahl der auszulesenden Elemente
 */
void fifo16_pop_elements(fifo16* ff, unsigned short* data, long size)
{
    long i;
    if(fifo16_used(ff)<size)
        return;
    for(i=0; i<size; i++)
    {
        data[i] = ff->buffer[ff->readPtr++];
        if(ff->readPtr == ff->size)
            ff->readPtr = 0;
    }
}

/*
 * Auslesen einer Liste von Elementen ohne Löschen.
 * List aus dem FIFO "size" Elemente aus und speichert
 * diese in data, dabei werden die ausgelesenen Elemente
 * nicht aus dem FIFO entfernt.
 * Sind nicht genügend Daten vorhanden werden keine Daten
 * ausgelesen.
 * Param: data: Pointer auf Speicher in dem die Elemente
 *              abgelegt werden sollen.
 *        size: Anzahl der auszulesenden Elemente
 */
void fifo16_read_elements(fifo16* ff, unsigned short* data, long size)
{
    long readPtr = ff->readPtr;
    long i;
    if(fifo16_used(ff)<size)
        return;
    for(i=0; i<size; i++)
    {
        data[i] = ff->buffer[readPtr++];
        if(readPtr == ff->size)
            readPtr = 0;
    }
    //ff->readPtr = readPtr;
}

/*
 * Auslesen eines Elementes im FIFO an einer bestimmten
 * Position. Sind nicht genügend Daten im FIFO wird 0
 * zurück gegeben. Das Element wird nicht gelöscht.
 */
unsigned short fifo16_read_elememt(fifo16* ff, long pos)
{
    if(fifo16_used(ff) <= pos)
        return 0;
    long readPtr = ff->readPtr;
    readPtr += pos;
    if(readPtr>=ff->size)
        readPtr -= ff->size;
    return ff->buffer[readPtr];
}

/*
 * Auslesen des vordersten Elements im FIFO.
 * Gibt das erste Element im FIFO zurück und löscht
 * es aus dem FIFO.
 */
unsigned short fifo16_pop_element(fifo16* ff)
{
    if(fifo16_used(ff) == 0)
        return 0;
    short element = ff->buffer[ff->readPtr];
    ff->readPtr++;
    if(ff->readPtr >= ff->size)
        ff->readPtr = 0;
    return element;
```

```
}
```

A.1.17 fifo8.h

```
/*
 * fifo.h
 *
 *  Created on: 25.12.2011
 *      Author: fabian
 */

#ifndef FIFO8_H_
#define FIFO8_H_

typedef struct
{
    char* buffer;
    volatile long readPtr;
    volatile long writePtr;
    volatile long size;
} fifo8;


fifo8* fifo8_create(long size);
void fifo8_delete(fifo8* ff);
long fifo8_used(fifo8* ff);
long fifo8_free(fifo8* ff);
void fifo8_clear(fifo8* ff);
void fifo8_push_elements(fifo8* ff, char* data, long size); // append new data to the fifo
void fifo8_pop_elements(fifo8* ff, char* data, long size);  // read and remove the data from the fifo
void fifo8_read_elements(fifo8* ff, char* data, long size); // read without removing the data from the fifo
char fifo8_read_element(fifo8* ff, long pos); // read a single byte in the fifo without deleting it, 0 is returned
     when the position is out of dimension
char fifo8_pop_element(fifo8* ff); // read the next byte, returns 0 if there are no more bytes

#endif /* FIFO_H_ */
```

A.1.18 fifo8.c

```
/*
 * fifo.c
 *
 *  Created on: 25.12.2011
 *      Author: fabian
 *  Implementierung eines FIFOs mit variabler Größe und
 *  festen 1 Byte pro Element. Der FIFO ist in sofern
 *  interruptsicher, dass ein Interrupt lesen und der
 *  andere schreiben darf, ohne dass Kollisionen auftreten.
 *  Zum erstellen des FIFOs wird der HEAP verwendet,
 *  es ist also darauf zu achten, dass beim ständigen
 *  neu erstellen keine Fragmentierung im Speicher
 *  auftritt.
 */

#include "fifo8.h"

#include <stdlib.h>

/*
 * Erstellung eiens neuen FIFOs.
 * Mit dieser Funktion kann ein neuer FIFO erstellt und
 * initialisiert werden.
 */
fifo8* fifo8_create(long size)
{
    fifo8* ff = (fifo8*)malloc(sizeof(fifo8));
    ff->size = size;
    ff->readPtr = 0;
    ff->writePtr = 0;
    ff->buffer = (char*)malloc(size+1);  // one more byte needed
    return ff;
}
```

```
34
/*
 * Löschen des FIFOs.
 * Mit dieser Funktion können die reservierten Speicher
 * für den FIFO wieder freigegeben werden. Hierzu zählt
39 * zum einen der Puffer des FIFOs und zum anderen der
 * FIFO-Struct selbst.
 */
void fifo8_delete(fifo8* ff)
{
44     free(ff->buffer);
    free(ff);
}

/*
49 * Zurücksetzen des FIFOs.
 * Die read und write Pointer werden auf 0 zurück gesetzt,
 * was bedeutet, dass der FIFO leer ist.
 */
void fifo8_clear(fifo8* ff)
54 {
    ff->readPtr = 0;
    ff->writePtr = 0;
}

59 /*
 * Aktuell belgeter Speicher im FIFO.
 * Diese Funktion ermittelt anhand der beiden Lese- und
 * Schreibzeiger wie viele Bytes aktuell im FIFO belegt
 * sind.
64 */
long fifo8_used(fifo8* ff)
{
    long used = ff->writePtr-ff->readPtr;
    if(used<0)
69      used += ff->size;
    return used;
}

/*
74 * Freier Platz im FIFO.
 * Ermittelt mit Hilfe der Funktion fifo16_used() und der
 * Grö ß e des FIFOs wie viel Platz noch im FIFO frei ist.
 */
long fifo8_free(fifo8* ff)
79 {
    return ff->size-fifo8_used(ff);
}

/*
84 * Anfügen von einer Liste an Elementen.
 * Diese Funktion fügt die in data enthaltenden Elemente
 * an den FIFO an.
 * Param: data: Pointer auf anzufügendende Elemente
 *        size: Anzahl der Elemente
89 */
void fifo8_push_elements(fifo8* ff, char* data, long size)
{
    long i;
    if(fifo8_free(ff)<=size)
94      return;
    for(i=0; i<size; i++)
    {
        ff->buffer[ff->writePtr++] = data[i];
        if(ff->writePtr == ff->size)
99          ff->writePtr = 0;
    }
}

/*
104 * Auslesen einer Liste von Elementen.
 * Liest aus dem FIFO "size" Elemente aus und speichert
 * diese in data.
 * Sind nicht genügend Daten vorhanden werden keine Daten
 * ausgelesen.
109 * Param: data: Pointer auf Speicher in dem die Elemente
 *              abgelegt werden sollen.
 *        size: Anzahl der auszulesenden Elemente
 */
void fifo8_pop_elements(fifo8* ff, char* data, long size)
114 {
    long i;
    if(fifo8_used(ff)<size)
        return;
```

```
    for(i=0; i<size; i++)
    {
        data[i] = ff->buffer[ff->readPtr++];
        if(ff->readPtr == ff->size)
            ff->readPtr = 0;
    }
}

/*
 * Auslesen einer Liste von Elementen ohne Löschen.
 * List aus dem FIFO "size" Elemente aus und speichert
 * diese in data, dabei werden die ausgelesenen Elemente
 * nicht aus dem FIFO entfernt.
 * Sind nicht genügend Daten vorhanden werden keine Daten
 * ausgelesen.
 * Param: data: Pointer auf Speicher in dem die Elemente
 *              abgelegt werden sollen.
 *        size: Anzahl der auszulesenden Elemente
 */
void fifo8_read_elements(fifo8* ff, char* data, long size)
{
    long readPtr = ff->readPtr;
    long i;
    if(fifo8_used(ff)<size)
        return;
    for(i=0; i<size; i++)
    {
        data[i] = ff->buffer[readPtr++];
        if(readPtr == ff->size)
            readPtr = 0;
    }
}

/*
 * Auslesen eines Elementes im FIFO an einer bestimmten
 * Position. Sind nicht genügend Daten im FIFO wird 0
 * zurück gegeben. Das Element wird nicht gelöscht.
 */
char fifo8_read_element(fifo8* ff, long pos)
{
    if(fifo8_used(ff) <= pos)
        return 0;
    long readPtr = ff->readPtr;
    readPtr += pos;
    if(readPtr>=ff->size)
        readPtr -= ff->size;
    return ff->buffer[readPtr];
}

/*
 * Auslesen des vordersten Elements im FIFO.
 * Gibt das erste Element im FIFO zurück und löscht
 * es aus dem FIFO.
 */
char fifo8_pop_element(fifo8* ff)
{
    if(fifo8_used(ff) == 0)
        return 0;
    char element = ff->buffer[ff->readPtr];
    ff->readPtr++;
    if(ff->readPtr >= ff->size)
        ff->readPtr = 0;
    return element;
}
```

A.1.19 flash.h

```
/*
 * flash.h
 *
 *  Created on: 26.12.2011
 *      Author: fabian
 *
 *
 *  Flash structures:
 *  Flash 0:
 *      Bytes                       Value
 *      2                           configuration CRC
```

```
 *      sizeof(config_t)           configuration data
 *      2                          calibration array CRC
 *      sizeof(mv2dac)             calibration array data
 *      4096-(sizeof(config_t)+2)  unused data
 *      rest                       voltage values, coded in dac values, each 16 bits
 *  Flash 1:
 *      4096                       reserved for later use
 *      rest                       current values
 */

#ifndef FLASH_H_
#define FLASH_H_

// Commands
#define READ        0x03  // Read Memory
#define ERASE4K     0x20  // Erase 4k Block
#define CHIPERASE   0x60  // Erase Full Memory Array
#define BYTEPROGRAM 0x02  // To Program One Data Byte
#define AAI         0xAD  // Auto Address Increment Programming
#define RDSR        0x05  // Read-Status-Register
#define EWSR        0x50  // Enable-Write-Status-Register
#define WRSR        0x01  // Write Status-Register
#define WREN        0x06  // Write-Enable
#define WRDI        0x04  // Write-Disable
#define RDID        0x90  // Read ID

// Status register bits
#define SBUSY 0
#define SWEL  1
#define SBP0  2
#define SBP1  3
#define SBP2  4
#define SBP3  5
#define SAAI  6
#define SBPL  7

int flash_init(void);
unsigned char flash_write_read(unsigned char data);

void flash_write_enable(char chip);
void flash_erase_chip(char chip);
void flash_wait(char chip);

void flash_status_enable(char chip);
unsigned char flash_status_read(char chip);
void flash_status_write(char chip, unsigned char reg);

void flash_erase_block_4k(char chip, long address);

void flash_write_block(char chip, char* data, long size, long address);
void flash_read_block(char chip, char* data, long size, long address);

void flash_hold(char chip, char high);
void flash_cs(char chip, char high);
void flash_wp(char chip, char high);



#endif /* FLASH_H_ */
```

A.1.20 flash.c

```
/*
 * flash.c
 *
 *  Created on: 26.12.2011
 *      Author: fabian
 *  API zur Ansteuerung der beiden externen Flash-Speicher.
 *  Es werden eine Reihe von Funktionen zur Verfügung gestellt,
 *  um den externen Flash zu initialisieren, zu löschen, zu
 *  beschreiben und zu lesen.
 */

#include "flash.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 <utils/uartstdio.h>
#include <inc/lm3s9b92.h>

int flash_init(void)
{
    volatile unsigned long ulLoop;
    unsigned long dummy;

    UARTprintf("Initializing SPI flash...");
    // Initialisierung der Pins
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOH; // Clock enable on port H
    ulLoop = SYSCTL_RCGC2_R;
    GPIO_PORTH_DIR_R |= (1<<0) | (1<<1) | (1<<2) | (1<<5);
    GPIO_PORTH_DEN_R |= (1<<0) | (1<<1) | (1<<2) | (1<<5);


    /* !!WARNING!!
     * Pin F2 wird auch von Ethernet für eine der Status-LED verwendet.
     * Dies ist nicht behebbar solange das Eval Board verwendet wird,
     * sollte aber in einer späteren Version ohne das Eval Board
     * behoben werden! Diese Funktion muss daher nach der
     * Initialisierungsfunktion des Ethernets aufgerufen werden!
     */
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOF; // Clock enable on port F
    ulLoop = SYSCTL_RCGC2_R;
    GPIO_PORTF_DIR_R |= (1<<0) | (1<<1);
    GPIO_PORTF_DEN_R |= (1<<0) | (1<<1);

    // Initialisierung der IOs auf Standardwerte
    flash_hold(0, 1);
    flash_cs(0, 1);
    flash_wp(0, 1);

    flash_hold(1, 1);
    flash_cs(1, 1);
    flash_wp(1, 1);

    // Initialisierung von SSI1
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    GPIOPinConfigure(GPIO_PF2_SSI1CLK);
    // Hardware Chipselect abschalten, da dies per Software gelöst werden muss
    //GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PH6_SSI1RX);
    GPIOPinConfigure(GPIO_PH7_SSI1TX);

    // Setzen der Pins in SSI Modus (SPI)
    GPIOPinTypeSSI(GPIO_PORTF_BASE, GPIO_PIN_2);
    GPIOPinTypeSSI(GPIO_PORTH_BASE, GPIO_PIN_6 | GPIO_PIN_7);

    // 1MHz Takt für SPI (hier ist noch Luft nach oben)
    SSIConfigSetExpClk(SSI1_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, 5000000, 8);

    // Einschalten des SPI Interfaces
    SSIEnable(SSI1_BASE);

    // Löschen aller Elemente im Hardware SPI FIFO
    while(SSIDataGetNonBlocking(SSI1_BASE, &dummy));

    unsigned char reg = flash_status_read(0);
    UARTprintf("Status register: %d\n", (unsigned long)reg);

    flash_status_enable(0);  // Aktivieren von Schreiben in Status register
    flash_status_write(0, 0x00);  // Entfernen aller "write protects"

    flash_status_enable(1);  // Aktivieren von Schreiben in Status register
    flash_status_write(1, 0x00);  // Entfernen aller "write protects"

    // Das Status Register sollte bei der Initialisierung 0 sein,
    // da dies zuvor geschreiben wurde. Ist das nicht der Fall,
    // ist ein Fehler aufgetreten.
    reg = flash_status_read(0);
    if(reg!=0)
    {
        UARTprintf("failed: Wrong status register(%d) value of flash 0\n", (unsigned long)reg);
        return 0;
    }
    else
    {
```

```
        reg = flash_status_read(1);
        if(reg!=0)
        {
            UARTprintf("failed: Wrong status register(%d) value of flash 1\n", (unsigned long)reg);
            return 0;
        }
        else
        {
            UARTprintf("done\n");
        }
    }

    //flash_status_enable(0);
    //flash_status_write(0, 0x00);

    //flash_status_enable(0);  // enable status register writes
    //flash_status_write(0, 0x00);  // clear write protects
    /*UARTprintf("Status register (1): %d\n", (unsigned long)flash_status_read(0));
    flash_erase_block_4k(0, 0x000000);
    //flash_erase_chip(0);
    UARTprintf("Status register (2): %d\n", (unsigned long)flash_status_read(0));
    flash_wait(0);
    UARTprintf("Status register (3): %d\n", (unsigned long)flash_status_read(0));
    flash_write_block(0, "Hier kommt die 1. Konfiguration hin...", 38, 0x000000);

    flash_erase_block_4k(0, 0x001000);
    flash_wait(0);
    flash_write_block(0, "Hier kommt die 2. Konfiguration hin...", 38, 0x001000);

    char buffer[100];
    flash_read_block(0, buffer, 38, 0x000000);
    UARTprintf("Message 1. read: %s\n", buffer);
    flash_read_block(0, buffer, 38, 0x001000);
    UARTprintf("Message 2. read: %s\n", buffer);*/

    /*UARTprintf("Status register (1): %d\n", (unsigned long)flash_status_read(0));

    flash_status_enable(0);  // enable status register writes
    flash_status_write(0, 0x00);  // clear write protects
    UARTprintf("Status register (2): %d\n", (unsigned long)flash_status_read(0));
    flash_erase_chip(0);  // do a chip erase
    UARTprintf("Status register (3): %d\n", (unsigned long)flash_status_read(0));
    flash_wait(0);  // wait for chip erase to be finished
    UARTprintf("Status register (4): %d\n", (unsigned long)flash_status_read(0));

    // write test data
    flash_write_block(0, "Das ist ein langer Text", 24, 0x000000);
    UARTprintf("Status register (5): %d\n", (unsigned long)flash_status_read(0));

    // read test data
    char data[100];
    flash_read_block(0, data, 24, 0x000000);
    UARTprintf("Message read: %s\n", data);

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

/*
 * Schreiben und lesen eines Bytes auf dem SPI Interface
 */
unsigned char flash_write_read(unsigned char data)
{
    unsigned long ans;
    SSIDataPut(SSI1_BASE, data);
    SSIDataGet(SSI1_BASE, &ans);
    return (unsigned char)(ans&0xff);
}

/*
 * Lesen des Status Registers eines Flash.
 */
unsigned char flash_status_read(char chip)
{
    unsigned long ans;
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, RDSR);
    SSIDataPut(SSI1_BASE, 0x00);  // don't care byte
    SSIDataGet(SSI1_BASE, &ans);
    SSIDataGet(SSI1_BASE, &ans);
    flash_cs(chip, 1);
    return (unsigned char)(ans&0xff);
}
```

```
/*
 * Aktivieren des Status Registers.
 * Dies muss aufgerufen werden, bevor in das Status
 * Register geschreiben werden darf. Siehe Datenblatt.
 */
void flash_status_enable(char chip)
{
    unsigned long ans;
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, EWSR);
    SSIDataGet(SSI1_BASE, &ans);
    flash_cs(chip, 1);
}

/*
 * Das schreiben auf den Chip erlauben.
 * Dies muss einmalig ausgeführt werden bevor Daten
 * auf den Flash geschrieben werden oder er gelöscht
 * wird.
 */
void flash_write_enable(char chip)
{
    unsigned long ans;
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, WREN);
    SSIDataGet(SSI1_BASE, &ans);
    flash_cs(chip, 1);
}

/*
 * Löschen des gesamten Chips.
 */
void flash_erase_chip(char chip)
{
    unsigned long ans;
    flash_write_enable(chip);
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, CHIPERASE);
    SSIDataGet(SSI1_BASE, &ans);
    flash_cs(chip, 1);
}

/*
 * Warten auf laufende Aktionen in einem Flash.
 * Diese Funktion liest das Status Register so
 * lange aus, bis das BUSY Flag nicht mehr
 * gesetzt ist.
 */
void flash_wait(char chip)
{
    unsigned long ans;
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, RDSR);
    SSIDataGet(SSI1_BASE, &ans);
    do
    {
        SSIDataPut(SSI1_BASE, 0x00);  // don't care byte
        SSIDataGet(SSI1_BASE, &ans);
    }
    while(ans & (1<<SBUSY));
    flash_cs(chip, 1);
}

/*
 * Schreiben des Status Registers.
 */
void flash_status_write(char chip, unsigned char reg)
{
    unsigned long ans;
    flash_cs(chip, 0);
    SSIDataPut(SSI1_BASE, WRSR);
    SSIDataPut(SSI1_BASE, reg);
    SSIDataGet(SSI1_BASE, &ans);
    SSIDataGet(SSI1_BASE, &ans);
    flash_cs(chip, 1);
}

/*
 * Steuerung der HOLD Leitung.
 * Wird aktuell nicht direkt verwendet, lediglich bei
 * der Initialisierung, um die Leitung in den richtigen
 * Zustand zu bringen.
 */
void flash_hold(char chip, char high)
```

```
{
    if(chip == 0)
    {
        if(high)
            GPIO_PORTH_DATA_R |= (1<<2);
        else
            GPIO_PORTH_DATA_R &= ~(1<<2);
    }
    else if(chip == 1)
    {
        if(high)
            GPIO_PORTF_DATA_R |= (1<<1);
        else
            GPIO_PORTF_DATA_R &= ~(1<<1);
    }
}

/*
 * Steuerung der Chip-Select Leitungen.
 * Hiermit können die Chip-Select Leitungen der
 * beiden Flash gesteuert werden.
 * Es wird dabei nicht sichergesetllt, dass nicht
 * versehentlich beide Chips auf mal selektiert werden!
 * Param: high: 0=Chip ist selektiert
 *              1=Chip ist nicht selektiert
 */
void flash_cs(char chip, char high)
{
    if(chip == 0)
    {
        if(high)
            GPIO_PORTH_DATA_R |= (1<<0);
        else
            GPIO_PORTH_DATA_R &= ~(1<<0);
    }
    else if(chip == 1)
    {
        if(high)
            GPIO_PORTH_DATA_R |= (1<<5);
        else
            GPIO_PORTH_DATA_R &= ~(1<<5);
    }
}

/*
 * Steuerung der Write Protect Leitung.
 * Diese Funktion wird aktuell nur in der Initialisierung
 * verwendet, um den Zustand der Leitung korrekt zu setzen.
 */
void flash_wp(char chip, char high)
{
    if(chip == 0)
    {
        if(high)
            GPIO_PORTH_DATA_R |= (1<<1);
        else
            GPIO_PORTH_DATA_R &= ~(1<<1);
    }
    else if(chip == 1)
    {
        if(high)
            GPIO_PORTF_DATA_R |= (1<<0);
        else
            GPIO_PORTF_DATA_R &= ~(1<<0);
    }
}

/*
 * Löschen eines 4kB Block in einem Flash.
 * Löscht einen 4kB Block im Flash, danach muss
 * bevor geschreiben werden kann noch mit flash_wait()
 * auf die Beendigung des Vorganges gewartet werden.
 */
void flash_erase_block_4k(char chip, long address)
{
    // make sure address is a multiple of 4k
    //address -= (address%4096);
    //flash_status_enable(0);  // enable status register writes
    //flash_status_write(0, 0x00);  // clear write protects

    flash_write_enable(chip);
    flash_cs(chip, 0);
    flash_write_read(ERASE4K);  // enter ERASE 4K mode
    flash_write_read((address>>16)&0xff);  // write the address
```

```
    flash_write_read((address>>8)&0xff);
    flash_write_read((address)&0xff);
    flash_cs(chip, 1);
}

/*
 * Schreiben eines Blocks auf den Flash.
 * Der block kann beliebig groß sein, die Größe muss
 * allerdings durch 2 teilbar sein!
 */
void flash_write_block(char chip, char* data, long size, long address)
{
    long i;
    unsigned char ans;
    flash_write_enable(chip);  // write enable
    flash_cs(chip, 0);

    // Die ersten zwei Bytes müssen gesondert behandelt weden
    flash_write_read(AAI);  // enter AAI mode
    flash_write_read((address>>16)&0xff);  // write the address
    flash_write_read((address>>8)&0xff);
    flash_write_read((address)&0xff);
    flash_write_read(data[0]);  // write the first two bytes
    flash_write_read(data[1]);
    flash_cs(chip, 1);

    // Die restlichen Bytes werden alle im AAI Modus
    // geschrieben. Siehe Datenblatt.
    for(i=2; i<size; i++)
    {
        flash_cs(chip, 0);

        // Prüfen, ob vorherige Operation beendet ist
        flash_write_read(RDSR);
        do
        {
            ans = flash_write_read(0x00);
        }
        while(ans & (1<<SBUSY));
        flash_cs(chip, 1);

        // Schreiben der zwei Datenbytes
        flash_cs(chip, 0);
        flash_write_read(AAI);  // go on in AAI mode
        flash_write_read(data[i++]);
        flash_write_read(data[i]);
        flash_cs(chip, 1);
    }

    // Warten bis das letzte Byte geschreiben ist
    flash_cs(chip, 0);
    flash_write_read(RDSR);
    do
    {
        ans = flash_write_read(0x00);
    }
    while(ans & (1<<SBUSY));
    flash_cs(chip, 1);


    // Beenden des AAI Modus
    flash_cs(chip, 0);
    flash_write_read(WRDI);
    flash_cs(chip, 1);
}

/*
 * Lesen eines Blockes aus einem Flash.
 * Es wird ein block beliebiger Größe vom Flash gelesen.
 * Die Größe und die Position muss dabei lediglich durch
 * zwei teilbar sein.
 */
void flash_read_block(char chip, char* data, long size, long address)
{
    long i;
    flash_cs(chip, 0);
    flash_write_read(READ);  // enter READ mode
    flash_write_read((address>>16)&0xff);  // write the address
    flash_write_read((address>>8)&0xff);
    flash_write_read((address)&0xff);
    for(i=0; i<size; i++)
    {
        data[i] = flash_write_read(0x00);
    }
```

```
    flash_cs(chip, 1);
}
```

A.1.21 iadc.h

```
/*
 * adc.h
 *
 *  Created on: 29.12.2011
 *      Author: fabian
 */

#ifndef ADC_H_
#define ADC_H_

void iadc_init(void);
void iadc_oneshot(void);
void iadc_interrupt(void);

// Für interne Benutzung
extern volatile long current_ref_adc, current_hi_adc, current_lo_adc;
extern volatile long current_hi, current_lo;

#endif /* ADC_H_ */
```

A.1.22 iadc.c

```
/*
 * adc.c
 *
 *  Created on: 29.12.2011
 *      Author: fabian
 * Initialisierung und Ansteuerung des internen ADCs.
 * Der interne ADC wird hier initialisiert, läuft mit maximaler
 * Geschwindigkeit und macht bei den Pins PD1, PD2, PD3 eine
 * 8-fach Überabtastung. An diesen Pins liegen die beiden
 * Strommessungen vom Verstärker und die Referenzsspannung.
 */

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

#include "iadc.h"

#include "led.h"
volatile char iadc_sampled;
volatile long current_hi, current_lo;

/*
 * Initialisierung des ADC.
 * Der ADC läuft im Signleshot Betrieb. Ein Vorgang wird
 * per Software ausgelö ß t.
 */
void iadc_init(void)
{
    UARTprintf("Initializing internal ADC...");
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    GPIOPinTypeADC(GPIO_PORTD_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);

    ADCSequenceConfigure(ADC0_BASE, 2, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 2, 0, ADC_CTL_CH14);
    ADCSequenceStepConfigure(ADC0_BASE, 2, 1, ADC_CTL_CH12);
    ADCSequenceStepConfigure(ADC0_BASE, 2, 2, ADC_CTL_CH13 | ADC_CTL_IE | ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 2);
    ADCHardwareOversampleConfigure(ADC0_BASE, 8);
```

```
46
    ADCIntClear(ADC0_BASE, 2);
    ADCIntEnable(ADC0_BASE, 2);
    IntEnable(INT_ADC0SS2);
    UARTprintf("done\n");
51  }

    /*
     * Eine Messung und Mittelung aller drei Analogeingänge
     * mit anschließender Wandlung der ADC Werte in Ströme.
56   * Ist die Wandlung beendet wird der utnerstehende
     * Interrupt ausgeführt, der dann dieser Funktion
     * mitteilt, dass die Werte umgesetzt wurden.
     */
    void iadc_oneshot(void)
61  {
        iadc_sampled = 0;
        ADCProcessorTrigger(ADC0_BASE, 2);
        while(!iadc_sampled);

66      //float fcurrent_hi=(float)(current_hi_adc-current_ref_adc)/1024.0f*3.3f/7.024f*1000.0f;
        //float fcurrent_lo=(float)(current_lo_adc-current_ref_adc)/1024.0f*3.3f/838.0f*1000000.0f;

        current_hi = (current_hi_adc-current_ref_adc)*6600/14385;
        current_lo = (current_lo_adc-current_ref_adc)*103125/26816;
71  }

    // Globale Variablen für den aktuellen Strom, Referenz etc.
    // Die Werte sind nicht umgewandelt, also reine ADC-Werte
    volatile long current_ref_adc, current_hi_adc, current_lo_adc;
76
    /*
     * Interrupt bei vollenden der internen ADC Wandlung.
     * Die einzelnen Werte werden in den obrigen Variablen
     * abgespeichert und anschließend von der Funktion
81   * iadc_oneshot() ausgwertet, welche auf diesen Interrupt
     * wartet.
     */
    void iadc_interrupt(void)
    {
86      unsigned long values[3];
        //led_toggle();
        //UARTprintf("interrupt called\n");
        ADCIntClear(ADC0_BASE, 2);
        ADCSequenceDataGet(ADC0_BASE, 2, values);
91      current_ref_adc = values[0];
        current_hi_adc = values[1];
        current_lo_adc = values[2];
        iadc_sampled = 1;
    }
```

A.1.23 led.h

```
    /*
     * led.h
     *
5    *  Created on: 26.12.2011
     *      Author: fabian
     */

    #ifndef LED_H_
    #define LED_H_
10
    // Grün
    #define LED0_DIR  GPIO_PORTC_DIR_R
    #define LED0_DEN  GPIO_PORTC_DEN_R
    #define LED0_DAT  GPIO_PORTC_DATA_R
15  #define LED0_PIN  6

    // Rot
    #define LED1_DIR  GPIO_PORTC_DIR_R
    #define LED1_DEN  GPIO_PORTC_DEN_R
20  #define LED1_DAT  GPIO_PORTC_DATA_R
    #define LED1_PIN  7

    // Intern Mitte
    #define LED2_DIR  GPIO_PORTD_DIR_R
25  #define LED2_DEN  GPIO_PORTD_DEN_R
```

```
#define LED2_DAT  GPIO_PORTD_DATA_R
#define LED2_PIN  0

// Intern Rechts
#define LED3_DIR  GPIO_PORTF_DIR_R
#define LED3_DEN  GPIO_PORTF_DEN_R
#define LED3_DAT  GPIO_PORTF_DATA_R
#define LED3_PIN  3

void led_on(unsigned char led);
void led_off(unsigned char led);
void led_toggle(unsigned char led);

void led_init(void);

#endif /* LED_H_ */
```

A.1.24 led.c

```
/*
 * led.c
 *
 *  Created on: 26.12.2011
 *      Author: fabian
 *  Einige Funktionen zu ein und anschalten der LEDs auf
 *  dem Mainbaord.
 */

#include "led.h"

#include <inc/lm3s9b92.h>

volatile unsigned long* led_dirs[4]={&LED0_DIR, &LED1_DIR, &LED2_DIR, &LED3_DIR};
volatile unsigned long* led_data[]={&LED0_DAT, &LED1_DAT, &LED2_DAT, &LED3_DAT};
const unsigned long  led_pins[4]={LED0_PIN, LED1_PIN, LED2_PIN, LED3_PIN};

/*
 * Initialisierung der Pins für die LEDs.
 */
void led_init(void)
{
    volatile unsigned long ulLoop;

    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOD;
    ulLoop = SYSCTL_RCGC2_R;

    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOC;
    ulLoop = SYSCTL_RCGC2_R;

    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOF;
    ulLoop = SYSCTL_RCGC2_R;

    LED0_DIR |= (1<<LED0_PIN);
    LED0_DEN |= (1<<LED0_PIN);

    LED1_DIR |= (1<<LED1_PIN);
    LED1_DEN |= (1<<LED1_PIN);

    LED2_DIR |= (1<<LED2_PIN);
    LED2_DEN |= (1<<LED2_PIN);

    LED3_DIR |= (1<<LED3_PIN);
    LED3_DEN |= (1<<LED3_PIN);
}

/*
 * Einschalten einer LED
 */
void led_on(unsigned char led)
{
    *(led_data[led]) |= (1<<led_pins[led]);
}

/*
 * Ausschalten einer LED
 */
void led_off(unsigned char led)
{
```

```
    *(led_data[led]) &= ~(1<<led_pins[led]);
}

/*
 * Toggeln einer LED
 */
void led_toggle(unsigned char led)
{
    *(led_data[led]) ^= (1<<led_pins[led]);
}
```

A.1.25 lwipopts.c

```
//*****************************************************************************
//
// lwipopts.h - Configuration file for lwIP
//
// 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.
//
//*****************************************************************************
//
// NOTE:  This file has been derived from the lwIP/src/include/lwip/opt.h
// header file.
//
// For additional details, refer to the original "opt.h" file, and lwIP
// documentation.
//
//*****************************************************************************

#ifndef __LWIPOPTS_H__
#define __LWIPOPTS_H__

//*****************************************************************************
//
// ---------- Stellaris / lwIP Port Options ----------
//
//*****************************************************************************
#define HOST_TMR_INTERVAL               100         // default is 0
//#define DHCP_EXPIRE_TIMER_MSECS         (60 * 1000)
//#define INCLUDE_HTTPD_SSI
//#define INCLUDE_HTTPD_CGI
#define DYNAMIC_HTTP_HEADERS
//#define INCLUDE_HTTPD_DEBUG

//*****************************************************************************
//
// ---------- Platform specific locking ----------
//
//*****************************************************************************
#define SYS_LIGHTWEIGHT_PROT            1           // default is 0
#define NO_SYS                          1           // default is 0
//#define MEMCPY(dst,src,len)             memcpy(dst,src,len)
//#define SMEMCPY(dst,src,len)            memcpy(dst,src,len)

//*****************************************************************************
//
// ---------- Memory options ----------
//
//*****************************************************************************
//#define MEM_LIBC_MALLOC                 0
#define MEM_ALIGNMENT                   4           // default is 1
#define MEM_SIZE                        (22 * 1024)  // default is 1600, was 16K
```

```
//#define MEMP_OVERFLOW_CHECK             0
//#define MEMP_SANITY_CHECK               0
//#define MEM_USE_POOLS                   0
//#define MEMP_USE_CUSTOM_POOLS           0

//*****************************************************************************
//
// ---------- Internal Memory Pool Sizes ----------
//
//*****************************************************************************
#define MEMP_NUM_PBUF                     24    // Default 16, was 16
//#define MEMP_NUM_RAW_PCB                4
//#define MEMP_NUM_UDP_PCB                4
#define MEMP_NUM_TCP_PCB                  16    // Default 5, was 12
//#define MEMP_NUM_TCP_PCB_LISTEN         8
//#define MEMP_NUM_TCP_SEG                16
//#define MEMP_NUM_REASSDATA              5
//#define MEMP_NUM_ARP_QUEUE              30
//#define MEMP_NUM_IGMP_GROUP             8
//#define MEMP_NUM_SYS_TIMEOUT            3
//#define MEMP_NUM_NETBUF                 2
//#define MEMP_NUM_NETCONN                4
//#define MEMP_NUM_TCPIP_MSG_API          8
//#define MEMP_NUM_TCPIP_MSG_INPKT        8
#define PBUF_POOL_SIZE                    24    // Default 16, was 36

//*****************************************************************************
//
// ---------- ARP options ----------
//
//*****************************************************************************
//#define LWIP_ARP                        1
//#define ARP_TABLE_SIZE                  10
//#define ARP_QUEUEING                    1
//#define ETHARP_TRUST_IP_MAC             1

//*****************************************************************************
//
// ---------- IP options ----------
//
//*****************************************************************************
//#define IP_FORWARD                      0
//#define IP_OPTIONS_ALLOWED              1
#define IP_REASSEMBLY                   0           // default is 1
#define IP_FRAG                         0           // default is 1
//#define IP_REASS_MAXAGE                 3
//#define IP_REASS_MAX_PBUFS              10
//#define IP_FRAG_USES_STATIC_BUF         1
//#define IP_FRAG_MAX_MTU                 1500
//#define IP_DEFAULT_TTL                  255

//*****************************************************************************
//
// ---------- ICMP options ----------
//
//*****************************************************************************
//#define LWIP_ICMP                       1
//#define ICMP_TTL                       (IP_DEFAULT_TTL)

//*****************************************************************************
//
// ---------- RAW options ----------
//
//*****************************************************************************
//#define LWIP_RAW                        1
//#define RAW_TTL                        (IP_DEFAULT_TTL)

//*****************************************************************************
//
// ---------- DHCP options ----------
//
//*****************************************************************************
#define LWIP_DHCP                       1           // default is 0
//#define DHCP_DOES_ARP_CHECK             ((LWIP_DHCP) && (LWIP_ARP))

//*****************************************************************************
//
// ---------- UPNP options ----------
//
//*****************************************************************************
//#define LWIP_UPNP                       0

//*****************************************************************************
//
```

```
// ---------- PTPD options ----------
//
//*****************************************************************************
//#define LWIP_PTPD                       0

//*****************************************************************************
//
// ---------- AUTOIP options ----------
//
//*****************************************************************************
#define LWIP_AUTOIP                     1           // default is 0
#define LWIP_DHCP_AUTOIP_COOP           ((LWIP_DHCP) && (LWIP_AUTOIP))
                                                    // default is 0
#define LWIP_DHCP_AUTOIP_COOP_TRIES     5           // default is 9

//*****************************************************************************
//
// ---------- SNMP options ----------
//
//*****************************************************************************
//#define LWIP_SNMP                       0
//#define SNMP_CONCURRENT_REQUESTS        1
//#define SNMP_TRAP_DESTINATIONS          1
//#define SNMP_PRIVATE_MIB                0
//#define SNMP_SAFE_REQUESTS              1

//*****************************************************************************
//
// ---------- IGMP options ----------
//
//*****************************************************************************
//#define LWIP_IGMP                       0

//*****************************************************************************
//
// ---------- DNS options -----------
//
//*****************************************************************************
//#define LWIP_DNS                        0
//#define DNS_TABLE_SIZE                  4
//#define DNS_MAX_NAME_LENGTH             256
//#define DNS_MAX_SERVERS                 2
//#define DNS_DOES_NAME_CHECK             1
//#define DNS_USES_STATIC_BUF             1
//#define DNS_MSG_SIZE                    512

//*****************************************************************************
//
// ---------- UDP options ----------
//
//*****************************************************************************
//#define LWIP_UDP                        1
//#define LWIP_UDPLITE                    0
//#define UDP_TTL                         (IP_DEFAULT_TTL)

//*****************************************************************************
//
// ---------- TCP options ----------
//
//*****************************************************************************
//#define LWIP_TCP                        1
//#define TCP_TTL                         (IP_DEFAULT_TTL)
#define TCP_WND                         1024   // default is 2048
//#define TCP_MAXRTX                      12
//#define TCP_SYNMAXRTX                   6
//#define TCP_QUEUE_OOSEQ                 1
#define TCP_MSS                         1024        // default is 128
//#define TCP_CALCULATE_EFF_SEND_MSS      1
#define TCP_SND_BUF                      (4 * TCP_MSS)
                                                    // default is 256, was 6 *
//#define TCP_SND_QUEUELEN                (4 * (TCP_SND_BUF/TCP_MSS))
//#define TCP_SNDLOWAT                    (TCP_SND_BUF/2)
//#define TCP_LISTEN_BACKLOG              0
//#define TCP_DEFAULT_LISTEN_BACKLOG      0xff
//#define TCP_OVERSIZE TCP_MSS

//*****************************************************************************
//
// ---------- API options ----------
//
//*****************************************************************************
//#define LWIP_EVENT_API                  0
//#define LWIP_CALLBACK_API               1
```

```
//******************************************************************************
//
// ---------- Pbuf options ----------
//
//******************************************************************************
#define PBUF_LINK_HLEN                  16          // default is 14
#define PBUF_POOL_BUFSIZE               256
                                                    // default is LWIP_MEM_ALIGN_SIZE(TCP_MSS+40+PBUF_LINK_HLEN)
#define ETH_PAD_SIZE                    2           // default is 0

//******************************************************************************
//
// ---------- Network Interfaces options ----------
//
//******************************************************************************
//#define LWIP_NETIF_HOSTNAME             0
//#define LWIP_NETIF_API                  0
//#define LWIP_NETIF_STATUS_CALLBACK      0
//#define LWIP_NETIF_LINK_CALLBACK        0
//#define LWIP_NETIF_HWADDRHINT           0

//******************************************************************************
//
// ---------- LOOPIF options ----------
//
//******************************************************************************
//#define LWIP_HAVE_LOOPIF                0
//#define LWIP_LOOPIF_MULTITHREADING      1

//******************************************************************************
//
// ---------- Thread options ----------
//
//******************************************************************************
//#define TCPIP_THREAD_NAME              "tcpip_thread"
//#define TCPIP_THREAD_STACKSIZE          0
//#define TCPIP_THREAD_PRIO               1
//#define TCPIP_MBOX_SIZE                 0
//#define SLIPIF_THREAD_NAME             "slipif_loop"
//#define SLIPIF_THREAD_STACKSIZE         0
//#define SLIPIF_THREAD_PRIO              1
//#define PPP_THREAD_NAME                "pppMain"
//#define PPP_THREAD_STACKSIZE            0
//#define PPP_THREAD_PRIO                 1
//#define DEFAULT_THREAD_NAME            "lwIP"
//#define DEFAULT_THREAD_STACKSIZE        0
//#define DEFAULT_THREAD_PRIO             1
//#define DEFAULT_RAW_RECVMBOX_SIZE       0
//#define DEFAULT_UDP_RECVMBOX_SIZE       0
//#define DEFAULT_TCP_RECVMBOX_SIZE       0
//#define DEFAULT_ACCEPTMBOX_SIZE         0

//******************************************************************************
//
// ---------- Sequential layer options ----------
//
//******************************************************************************
//#define LWIP_TCPIP_CORE_LOCKING         0
#define LWIP_NETCONN                    0           // default is 1

//******************************************************************************
//
// ---------- Socket Options ----------
//
//******************************************************************************
#define LWIP_SOCKET                     0           // default is 1
//#define LWIP_COMPAT_SOCKETS             1
//#define LWIP_POSIX_SOCKETS_IO_NAMES     1
//#define LWIP_TCP_KEEPALIVE              0
//#define LWIP_SO_RCVTIMEO                0
//#define LWIP_SO_RCVBUF                  0
//#define SO_REUSE                        0

//******************************************************************************
//
// ---------- Statistics options ----------
//
//******************************************************************************
//#define LWIP_STATS                      1
//#define LWIP_STATS_DISPLAY              0
//#define LINK_STATS                      1
//#define ETHARP_STATS                    (LWIP_ARP)
//#define IP_STATS                        1
//#define IPFRAG_STATS                    (IP_REASSEMBLY || IP_FRAG)
```

```
//#define ICMP_STATS                    1
//#define IGMP_STATS                    (LWIP_IGMP)
//#define UDP_STATS                     (LWIP_UDP)
//#define TCP_STATS                     (LWIP_TCP)
//#define MEM_STATS                     1
//#define MEMP_STATS                    1
//#define SYS_STATS                     1

//----------------------------------------------------------------------------
//
// ---------- PPP options ----------
//
//----------------------------------------------------------------------------
//#define PPP_SUPPORT                   0
//#define PPPOE_SUPPORT                 0
//#define PPPOS_SUPPORT                 PPP_SUPPORT

#if PPP_SUPPORT
//#define NUM_PPP                       1
//#define PAP_SUPPORT                   0
//#define CHAP_SUPPORT                  0
//#define MSCHAP_SUPPORT                0
//#define CBCP_SUPPORT                  0
//#define CCP_SUPPORT                   0
//#define VJ_SUPPORT                    0
//#define MD5_SUPPORT                   0
//#define FSM_DEFTIMEOUT                6
//#define FSM_DEFMAXTERMREQS            2
//#define FSM_DEFMAXCONFREQS            10
//#define FSM_DEFMAXNAKLOOPS            5
//#define UPAP_DEFTIMEOUT               6
//#define UPAP_DEFREQTIME               30
//#define CHAP_DEFTIMEOUT               6
//#define CHAP_DEFTRANSMITS             10
//#define LCP_ECHOINTERVAL              0
//#define LCP_MAXECHOFAILS              3
//#define PPP_MAXIDLEFLAG               100

//#define PPP_MAXMTU                    1500
//#define PPP_DEFMRU                    296
#endif

//----------------------------------------------------------------------------
//
// ---------- checksum options ----------
//
//----------------------------------------------------------------------------
//#define CHECKSUM_GEN_IP               1
//#define CHECKSUM_GEN_UDP              1
//#define CHECKSUM_GEN_TCP              1
//#define CHECKSUM_CHECK_IP             1
//#define CHECKSUM_CHECK_UDP            1
//#define CHECKSUM_CHECK_TCP            1

//----------------------------------------------------------------------------
//
// ---------- Debugging options ----------
//
//----------------------------------------------------------------------------
#if 0
#define U8_F "c"
#define S8_F "c"
#define X8_F "x"
#define U16_F "u"
#define S16_F "d"
#define X16_F "x"
#define U32_F "u"
#define S32_F "d"
#define X32_F "x"
extern void UARTprintf(const char *pcString, ...);
#define LWIP_DEBUG
#endif

//#define LWIP_DBG_MIN_LEVEL              LWIP_DBG_LEVEL_OFF
#define LWIP_DBG_MIN_LEVEL             LWIP_DBG_LEVEL_OFF
//#define LWIP_DBG_MIN_LEVEL              LWIP_DBG_LEVEL_WARNING
//#define LWIP_DBG_MIN_LEVEL              LWIP_DBG_LEVEL_SERIOUS
//#define LWIP_DBG_MIN_LEVEL              LWIP_DBG_LEVEL_SEVERE

//#define LWIP_DBG_TYPES_ON               LWIP_DBG_ON
#define LWIP_DBG_TYPES_ON              (LWIP_DBG_ON|LWIP_DBG_TRACE|LWIP_DBG_STATE|LWIP_DBG_FRESH)

//#define ETHARP_DEBUG                    LWIP_DBG_ON     // default is OFF
//#define NETIF_DEBUG                     LWIP_DBG_ON     // default is OFF
```

```
//#define PBUF_DEBUG                      LWIP_DBG_OFF
//#define API_LIB_DEBUG                   LWIP_DBG_OFF
//#define API_MSG_DEBUG                   LWIP_DBG_OFF
//#define SOCKETS_DEBUG                   LWIP_DBG_OFF
//#define ICMP_DEBUG                      LWIP_DBG_OFF
//#define IGMP_DEBUG                      LWIP_DBG_OFF
//#define INET_DEBUG                      LWIP_DBG_OFF
//#define IP_DEBUG                        LWIP_DBG_ON     // default is OFF
//#define IP_REASS_DEBUG                  LWIP_DBG_OFF
//#define RAW_DEBUG                       LWIP_DBG_OFF
//#define MEM_DEBUG                       LWIP_DBG_OFF
//#define MEMP_DEBUG                      LWIP_DBG_OFF
//#define SYS_DEBUG                       LWIP_DBG_OFF
//#define TCP_DEBUG                       LWIP_DBG_OFF
//#define TCP_INPUT_DEBUG                 LWIP_DBG_OFF
//#define TCP_FR_DEBUG                    LWIP_DBG_OFF
//#define TCP_RTO_DEBUG                   LWIP_DBG_OFF
//#define TCP_CWND_DEBUG                  LWIP_DBG_OFF
//#define TCP_WND_DEBUG                   LWIP_DBG_OFF
//#define TCP_OUTPUT_DEBUG                LWIP_DBG_OFF
//#define TCP_RST_DEBUG                   LWIP_DBG_OFF
//#define TCP_QLEN_DEBUG                  LWIP_DBG_OFF
//#define UDP_DEBUG                       LWIP_DBG_ON     // default is OFF
//#define TCPIP_DEBUG                     LWIP_DBG_OFF
//#define PPP_DEBUG                       LWIP_DBG_OFF
//#define SLIP_DEBUG                      LWIP_DBG_OFF
//#define DHCP_DEBUG                      LWIP_DBG_ON     // default is OFF
//#define AUTOIP_DEBUG                    LWIP_DBG_OFF
//#define SNMP_MSG_DEBUG                  LWIP_DBG_OFF
//#define SNMP_MIB_DEBUG                  LWIP_DBG_OFF
//#define DNS_DEBUG                       LWIP_DBG_OFF

#endif /* __LWIPOPTS_H__ */
```

A.1.26 main.c

```
#include <inc/lm3s9b92.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/ethernet.h"
#include "driverlib/flash.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "utils/locator.h"
#include "utils/lwiplib.h"
#include <utils/uartstdio.h>

#include "ethernet.h"
#include "discover.h"
#include "control.h"
#include "flash.h"
#include "led.h"
#include "uart.h"
#include "adcdac.h"
#include "system_timer.h"
#include "sync.h"
#include "iadc.h"
#include "config.h"
#include "voltage.h"
#include "calibrate.h"
#include "reloader_timer.h"

/**
 * Prioritäten (Hoechste nach niedrigste)
 * 0x00   Sync
 * 0xC0   SystemTimer
 * 0xD0   ReloadTimer
 * 0xE0   Ethernet
 */

/**
 * Diese Funktion kann bei der Initialisierung der Hardware
 * aufgerufen werden, wenn ein kritischer Fehler auftritt.
```

```
 * Die Funktion deaktiviert global alle Interrupts,
 * schaltet die rote LED an, die gruene LED aus und dreht
 * sich anschliessend in einer Endlosschleife.
 */
void fatal_error()
{
    ROM_IntMasterDisable();
    led_on(1);
    led_off(0);
    while(1);
}


int main(void)
{
    //ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);  //
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);  // Sollte mit 50
         MHz laufen

    led_init();
    led_set_mode(0, MODE_BLINK);
    led_set_mode(1, MODE_BLINK);
    led_set_mode(2, MODE_OFF);

    system_timer_init();
    ROM_IntMasterEnable();  // Einschalten der globalen Interrupts
    sleep(1000);

    uart_init();
    reloader_timer_init();
    if(!voltage_init())
        fatal_error();
    // Ethernet muss vor dem Flash initialisiert werden, da es eine Doppel-
    // belegung des Pins F2 gibt.
    if(!ethernet_init())
        fatal_error();
    if(!flash_init())
        fatal_error();
    if(!config_read())
        fatal_error();
    if(!adcdac_init())
        fatal_error();
    sync_init();
    iadc_init();
    iadc_oneshot();

    dac_set_data(0x0000);  // Setzen der Ausgangsspannung auf 0V fuer den Start

    while(1);  // Alles andere wird in Interrupts erledigt
}
```

A.1.27 reloader_timer.h

```
/*
 * reloader.h
 *
 *  Created on: 28.02.2012
 *      Author: Fabian
 */

#ifndef RELOADER_TIMER_H_
#define RELOADER_TIMER_H_

extern volatile int reload_finished;

void reloader_timer_init(void);

#endif /* RELOADER_H_ */
```

A.1.28 reloader_timer.c

```
/*
 * reloader.c
 *
 *  Created on: 28.02.2012
 *      Author: Fabian
 *  Timer zum Nachladen von Samples aus dem Flash.
 *  Dieser Timer wird über die Variablen reload_fill und
 *  config.repeat gesteuert und ist dafür verantwortlich
 *  Samples aus dem Flash in den Sample-FIFO zu laden.
 *  Dieser Timer hat eine höhere Priorität  als der
 *  Ethernet Interrupt, um Kollisionen zu vermeiden.
 *  Der Ethernet Interrupt stellt sicher, dass dieser Timer
 *  abgeschaltet ist für den Fall dass der Ethernet
 *  Interrupt etwas an dem Sample-FIFO ändern möchte.
 */

#include "reloader_timer.h"
#include "voltage.h"
#include "config.h"

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


/*
 * Initialisierung des Sample Reload Timers.
 * Der Timer läuft mit 1kHz.
 */
void reloader_timer_init(void)
{
    reload_fill = 0;
    //reload_finished = 1;
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
    TimerConfigure(TIMER2_BASE, TIMER_CFG_16_BIT_PAIR |  TIMER_CFG_B_PERIODIC);
    TimerLoadSet(TIMER2_BASE, TIMER_B, SysCtlClockGet() / 1000);
    IntMasterEnable();
    TimerIntEnable(TIMER2_BASE, TIMER_TIMB_TIMEOUT);
    IntEnable(INT_TIMER2B);
    TimerEnable(TIMER2_BASE, TIMER_B);
    IntPrioritySet(INT_TIMER2B, 0xD0);
}


/*
 * Interrupt für den Sample-Reloader.
 * Wenn reload_fill auf 1 gesetzt ist werden
 * neue Samples aus dem Flash geladen. Ist
 * config.repeat ebenfalls auf 1 gesetzt beginnt
 * der Interrupt auch wieder von vorne wenn er das
 * Ende der Samples im FIFO erreicht hat.
 * Ist config.repeat auf 0 gesetzt wird der Timer
 * beim Erreichen des Endes sich selbst abschalten
 * und keine weiteren Samples laden.
 */
void reloader_timer_int(void)
{
    if(reload_fill)  // Prüfen, ob Samples gerade nachgeladen werden sollen
    {
        if(!voltage_fill_fifo())
        {
            // Prüfen ob sich die Samples wiederholen sollen
            if(config.repeat == 1)
            {
                voltage_pos = 4096;  // Aktuelle Position im Flash
                voltage_count = config.voltage_count;
            }
            else
            {
                reload_fill = 0;
            }
        }
    }

    // Löschen des Interrupt Flags
    TimerIntClear(TIMER2_BASE, TIMER_TIMB_TIMEOUT);
}
```

A.1.29 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.
//
//*****************************************************************************

//*****************************************************************************
//
// 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(void);
extern void SysTickIntHandler(void);
extern void Timer0BIntHandler(void);
extern void syny_in_interrupt(void);
extern void sync_out_interrupt(void);
extern void iadc_interrupt(void);
extern void reloader_timer_int(void);

//*****************************************************************************
//
// 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
    IntDefaultHandler,                      // 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
    iadc_interrupt,                     // ADC Sequence 2
    IntDefaultHandler,                          // ADC Sequence 3
    IntDefaultHandler,                      // Watchdog timer
    IntDefaultHandler,                      // Timer 0 subtimer A
    Timer0BIntHandler,                      // Timer 0 subtimer B
    IntDefaultHandler,                      // Timer 1 subtimer A
    sync_out_interrupt,                     // Timer 1 subtimer B
    IntDefaultHandler,                      // Timer 2 subtimer A
    reloader_timer_int,                     // 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
    syny_in_interrupt                       // 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");
}
```

```
167 //*****************************************************************************
    //
    // 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.
172 //
    //*****************************************************************************
    static void
    NmiSR(void)
    {
177     //
        // Enter an infinite loop.
        //
        while(1)
        {
182     }
    }

    //*****************************************************************************
    //
187 // 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.
    //
    //*****************************************************************************
192 static void
    FaultISR(void)
    {
        //
        // Enter an infinite loop.
197     //
        while(1)
        {
        }
    }
202
    //*****************************************************************************
    //
    // This is the code that gets called when the processor receives an unexpected
    // interrupt.  This simply enters an infinite loop, preserving the system state
207 // for examination by a debugger.
    //
    //*****************************************************************************
    static void
    IntDefaultHandler(void)
212 {
        //
        // Go into an infinite loop.
        //
        while(1)
217     {
        }
    }
```

A.1.30 sync.h

```
1  /*
    * sync.h
    *
    *  Created on: 29.12.2011
    *      Author: fabian
6   */

    #ifndef SYNC_H_
    #define SYNC_H_

11  void sync_init(void);
    void syny_in_interrupt(void);

    void sync_out_enable(void);
    void sync_out_disable(void);
16  void sync_out_interrupt(void);

    #endif /* SYNC_H_ */
```

A.1.31 sync.c

```
/*
 * sync.c
 *
 *  Created on: 29.12.2011
 *      Author: fabian
 *  Initialisierung und API für den Synchronisationstimer.
 *  Der Synchronisationstimer ist für die Synchronisation
 *  der einzelnen Generatormodule verantwortlich. Dafür wird
 *  eines der Module zum Master (per Softwarebefehl und
 *  Jumper). Anschlie ß end wird ein Takt von 20kHz erzeugt,
 *  um die Module zu synchronisieren.
 *  Des weiteren ist hier der Interrupt Pin für das
 *  Empfangen der Synchronisation defniert.
 */

#include "sync.h"

#include <inc/hw_types.h>
#include <driverlib/sysctl.h>
#include <driverlib/gpio.h>
#include <inc/hw_memmap.h>
#include <driverlib/interrupt.h>
#include <inc/hw_ints.h>

#include <inc/hw_timer.h>
#include <inc/hw_ints.h>
#include <driverlib/timer.h>
#include <utils/uartstdio.h>

#include <inc/lm3s9b92.h>

#include <math.h>

#include "led.h"
#include "iadc.h"
#include "adcdac.h"
#include "voltage.h"
#include "system_timer.h"

/*
 * Initialisierung des Synchronisationstimers und des
 * Synchronisations-Interrupts.
 * Der Interrupt für den Interrupt Pin hat die höchste
 * Priorität aller Interrupts, da dieser ohne
 * Verzögerung ausgeführt werden muss, da es sonst
 * zu Jitter im ausgegebenen Datenstrom kommt.
 */
void sync_init(void)
{
    volatile unsigned long ulLoop;

    UARTprintf("Initializing sync...");

    // Initialize the sync input
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
    GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_1);

    GPIODirModeSet(GPIO_PORTJ_BASE, GPIO_PIN_1, GPIO_DIR_MODE_IN);
    GPIOIntTypeSet(GPIO_PORTJ_BASE, GPIO_PIN_1, GPIO_RISING_EDGE);

    GPIOPinIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_1);

    IntPrioritySet(INT_GPIOJ, 0x80);
    IntEnable(INT_GPIOJ);
    IntMasterEnable();
    IntPrioritySet(INT_TIMER1B, 0x00);  // absolute highest priority

    // Initialize the sync output, default disabled
    // Timer
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
    TimerConfigure(TIMER1_BASE, TIMER_CFG_16_BIT_PAIR |  TIMER_CFG_B_PERIODIC);
    TimerLoadSet(TIMER1_BASE, TIMER_B, SysCtlClockGet() / 40000);
    //TimerLoadSet(TIMER1_BASE, TIMER_B, SysCtlClockGet());  // Sekunden Takt
    IntMasterEnable();
    TimerIntEnable(TIMER1_BASE, TIMER_TIMB_TIMEOUT);
    IntEnable(INT_TIMER1B);
    TimerDisable(TIMER1_BASE, TIMER_B);
    IntPrioritySet(INT_TIMER1B, 0x20);  // second highest priority
    // SYNC_OUT pin
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOJ;
    ulLoop = SYSCTL_RCGC2_R;
```

```
82
    GPIO_PORTJ_DIR_R |= (1<<0);
    GPIO_PORTJ_DEN_R |= (1<<0);

    UARTprintf("done\n");
87  }

unsigned short counter = 0;
float pos = 0.0f;

92  /*
 * Interrupt des Interruptpins zum Ausgeben
 * eines neues Samples.
 */
void syny_in_interrupt(void)
97  {
    //led_set_mode(2, MODE_ON);
    voltage_set_next();
    GPIOPinIntClear(GPIO_PORTJ_BASE, GPIO_PIN_1);
    //led_set_mode(2, MODE_OFF);
102 }

/*
 * Einschalten der Synchronisation.
 * Diese Funktion wird extern aufgerufen und startet
107 * den Timer für die Synchronisation. Diese Funktion
 * darf nur auf einem der Module ausgeführt werden,
 * nämlich bei dem Modul welches der Master ist.
 */
void sync_out_enable(void)
112 {
    TimerEnable(TIMER1_BASE, TIMER_B);
}

/*
117 * Abschalten der Synchronisation.
 * Diese Funktion wird extern aufgerufen und schaltet
 * den Timer für die Synchronisation ab.
 */
void sync_out_disable(void)
122 {
    TimerDisable(TIMER1_BASE, TIMER_B);
}

/*
127 * Interrupt des Synchronisationstimers.
 * Diese Funktion wird mit 40kHz ausgeführt
 * und toggelt jedes mal den Pin J0.
 */
void sync_out_interrupt(void)
132 {
    GPIO_PORTJ_DATA_R ^= (1<<0);
    TimerIntClear(TIMER1_BASE, TIMER_TIMB_TIMEOUT);
}
```

A.1.32 system_timer.h

```
/*
 * system_timer.h
 *
4 *  Created on: 28.12.2011
 *      Author: fabian
 */

#ifndef SYSTEM_TIMER_H_
9 #define SYSTEM_TIMER_H_

#define MODE_OFF    0
#define MODE_ON     1
#define MODE_BLINK  2
14 #define MODE_SINGLE 3
#define MODE_IDENT  4

// Zeiten für das Blinken getrennt nach
// an/aus Dauer und dauerhaften bzw. einmaligem
19 // Blinken
#define BLINK_ON_TIME  100
#define BLINK_OFF_TIME 100
```

```
#define SINGLE_ON_TIME 30
#define SINGLE_OFF_TIME 30

// Anzahl des Blinkens für die Identifizierung
#define IDENTIFY_COUNT 20

// Systemzeit in Millisekunden
extern volatile unsigned long system_time;

void sleep(unsigned long ms);
void led_set_mode(unsigned char led, unsigned char mode);

// Interne Funktionen
void system_timer_init(void);


#endif /* SYSTEM_TIMER_H_ */
```

A.1.33 system_timer.c

```
/*
 * system_timer.c
 *
 *  Created on: 28.12.2011
 *      Author: fabian
 *  Der Systemtimer erfüllt zwei Aufgaben:
 *  * Bereitstellung eines Systemzählers für die
 *    Funktion sleep()
 *  * Erzeugung von komplexeren LED Signalen.
 *    Hier kann beispielsweise einer LED gesagt
 *    werden, dass sie einmal blinken soll,
 *    dauerhaft blinken soll oder an/abgeschaltet
 *    ist.
 */

#include "system_timer.h"

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

#include "led.h"

volatile unsigned long system_time;  // Systemzeit in Millisekunden

// Aktueller Zustand der LEDs
volatile unsigned short led_modes[4]={MODE_OFF, MODE_OFF, MODE_OFF, MODE_OFF};
// Vorheriger Zustand der LEDs
volatile unsigned short led_oldmodes[4]={MODE_OFF, MODE_OFF, MODE_OFF, MODE_OFF};
// Anzahl an Wiederholungen eines Blinkzyklus
volatile unsigned short led_count[4]={0, 0, 0, 0};
// Anzahl von Aufrufen des Timers seit letztem Umschalten der LED
volatile unsigned long  led_times[4]={0, 0, 0, 0};


/*
 * Initialsierung des Systemtimers.
 * Hier wird der Timer0 mit 1kHz initialisiert und
 * gestartet.
 */
void system_timer_init(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_16_BIT_PAIR |  TIMER_CFG_B_PERIODIC);
    TimerLoadSet(TIMER0_BASE, TIMER_B, SysCtlClockGet() / 1000);
    IntMasterEnable();
    TimerIntEnable(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    IntEnable(INT_TIMER0B);
    TimerEnable(TIMER0_BASE, TIMER_B);
    IntPrioritySet(INT_TIMER0B, 0xC0); // 2nd lowest priority
    system_time = 0;
}
```

```
/*
 * Interrupt Handler für den Systemtimer.
 * Hier wird zunächst die Systemzeit erhöht und anschließend
 * die Verarbietung der LEDs durchgeführt.
 */
void Timer0BIntHandler(void)
{
    int led;
    system_time++;

    // Durchgehen aller LEDs
    for(led = 0; led<4; led++)
    {
        // Einmaliges Blinken der LED, anschließende
        // Zurückkehr in den vorherigen Modus
        if(led_modes[led] == MODE_SINGLE)
        {
            if(led_times[led] + SINGLE_ON_TIME == system_time)
                led_off(led);
            else if(led_times[led] + SINGLE_ON_TIME + SINGLE_OFF_TIME == system_time)
            {
                led_set_mode(led, led_oldmodes[led]);
            }
        }
        // Dauerhaftes Blinken der LED
        else if(led_modes[led] == MODE_BLINK)
        {
            if(led_times[led] + BLINK_ON_TIME == system_time)
                led_off(led);
            else if(led_times[led] + BLINK_ON_TIME + BLINK_OFF_TIME == system_time)
            {
                led_times[led] = system_time;
                led_on(led);
            }
        }
        // Blinken einer gewissen Anzahl für die
        // Identifizierung des Moduls
        else if(led_modes[led] == MODE_IDENT)
        {
            if(led_times[led] + BLINK_ON_TIME == system_time)
                led_off(led);
            else if(led_times[led] + BLINK_ON_TIME + BLINK_OFF_TIME == system_time)
            {
                led_times[led] = system_time;
                led_on(led);
                led_count[led]++;
            }
            if(led_count[led] > IDENTIFY_COUNT)
            {
                led_count[led] = 0;
                led_set_mode(led, led_oldmodes[led]);
            }
        }
    }

    // Löschen des Timer Interrupt Flags
    TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
}

/*
 * Funktion zum Warten einer bestimmten
 * Anzahl an Millisekunden.
 * Vorsicht: Wird in diese Funktion wärend des
 * Debuggens gesprungen wird sie vermutlich niemals
 * beendet, da der Timer weiter läuft, die CPU jedoch
 * nicht!
 */
void sleep(unsigned long ms)
{
    unsigned long target = system_time+ms;
    while(system_time != target);
}

/*
 * Setzen des LED Modes.
 */
void led_set_mode(unsigned char led, unsigned char mode)
{
    if(led_modes[led] == mode)
        return;
    if(mode != led_modes[led])
    {
        led_oldmodes[led] = led_modes[led];
```

```
        led_modes[led] = mode;
    }
    switch(mode)
    {
147 case MODE_OFF:
        led_off(led);
        break;
    case MODE_ON:
        led_on(led);
152     break;
    case MODE_SINGLE:
        led_on(led);
        led_times[led] = system_time;
        break;
157 case MODE_BLINK:
        led_on(led);
        led_times[led] = system_time;
        break;
    case MODE_IDENT:
162     led_on(led);
        led_times[led] = system_time;
        led_count[led] = 0;
        break;
    default:
167     led_off(led);
        led_modes[led] = MODE_OFF;
        break;
    }
}
```

A.1.34 uart.h

```
/*
 * uart.h
3 *
 *  Created on: 26.12.2011
 *      Author: fabian
 */

8 #ifndef UART_H_
#define UART_H_

void uart_init(void);

13 #endif /* UART_H_ */
```

A.1.35 uart.c

```
/*
2 * uart.c
 *
 *  Created on: 26.12.2011
 *      Author: fabian
 *  Initialisierung der UART Schnittstelle.
7 *  Baud: 115200
 *  Bits: 8
 *  Parität: keine
 *  Stop: 1
 *
12 *  Hinweis: Beim Debuggen über den Stellaris Prommer
 *  muss die Hardware Flusskontroller aktiviert werden!
 */

#include "uart.h"
17

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

```
void uart_init(void)
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTStdioInit(0);

    // Ausgabe der ersten Nachricht und Versionsnummer
    UARTprintf("\n\nBattery simulator 0.1a by Fabian Schwartau\nPart of project BATSEN\n");
    UARTprintf("Initializing UART...done\n");
}
```

A.1.36 voltage.h

```
/*
 * voltage.h
 *
 *  Created on: 06.02.2012
 *      Author: Fabian
 */

#ifndef VOLTAGE_H_
#define VOLTAGE_H_

#include "fifo16.h"

extern fifo16* voltage_stream;
extern volatile long voltage_pos;
extern volatile long voltage_count;
extern volatile int reload_fill;

int voltage_init(void);   // Initialisierung des Fifo, ...

// Speichern von Samples
void voltage_reset(void);  // Von Neuem anfangen den Flash zu beschreiben
int voltage_add(unsigned long voltage); // Hinzufügen eines neuen Sample
int voltage_finish(void);

// Wiedergabe von Samples
void voltage_start(void);
int voltage_fill_fifo(void);
int voltage_set_next(void);

#endif /* VOLTAGE_H_ */
```

A.1.37 voltage.c

```
/*
 * voltage.c
 *
 *  Created on: 06.02.2012
 *      Author: Fabian
 *  Funktionen zur Speicherung und Wiedergabe
 *  der Samples im Flash.
 *  Es wird ein 5000 Byte großer FIFO erstellt in dem die
 *  noch zu speichenden Samples liegen oder im Falle
 *  einer Wiedergabe die noch wiederzugebenden Samples
 *  liegen. Der FIFO wird also mehrfach verwendet, um
 *  Speicher zu sparen.
 */

#include "voltage.h"
#include "flash.h"
#include "calibrate.h"
#include "config.h"
#include "adcdac.h"
#include "led.h"

#include <inc/hw_types.h>
#include <inc/hw_memmap.h>
#include <utils/uartstdio.h>
```

```
#include <driverlib/gpio.h>

// Steuerungsvariable für das nachladen von Samples aus dem FIFO
// Diese Variable wird in control.c geschrieben und in reload_timer.c
// gelesen.
volatile int reload_fill;
fifo16* voltage_stream;

// Aktuelle Position im Flash sowohl beim lesen als auch beim Schreiben
volatile long voltage_pos;

// Schreiben in Flash: Aktuelle Anzahl erhaltener Samples
// Lesen aus dem Flash: Anzahl noch zu lesender Samples
volatile long voltage_count;

// Puffer zum zwischenspeichern von 4kB Samples
unsigned short voltage_buffer[2048];

/*
 * Initialsierung des FIFOs für die Spannungswiedergabe und
 * -speicherung.
 */
int voltage_init(void)
{
    voltage_stream = fifo16_create(5000);
    if(!voltage_stream)
    {
        UARTprintf("error: malloc failed!\n");
        return 0;
    }
    // Überspringen des ersten Blocks, da hier Konfigurationen gespeichert werden
    voltage_pos = 4096;
    voltage_count = 0;
    return 1;
}

/*
 * Initialisierung vor dem Schreiben von neuen Samples
 * in den Flash.
 */
void voltage_reset(void)
{
    // Überspringen des ersten Blocks, da hier Konfigurationen gespeichert werden
    voltage_pos = 4096;
    voltage_count = 0;

    // Abschalten des automatischen Nachladen in den FIFO
    reload_fill = 0;

    // Löschen des ersten Blocks veranlassen
    flash_erase_block_4k(0, voltage_pos);

    // Löschen des FIFOs
    fifo16_clear(voltage_stream);
}

/*
 * Hinzufügen eines neuen Samples in den FIFO
 * zum Schreiben auf den Flash.
 */
int voltage_add(unsigned long voltage)
{
    if(voltage_pos>=4*1024*1024)
        return 0;
    unsigned short dacval = microvolt2dac(voltage);
    fifo16_push_elements(voltage_stream, &dacval, 1);
    if(fifo16_used(voltage_stream)>=2048)
    {
        fifo16_pop_elements(voltage_stream, voltage_buffer, 2048);
        flash_wait(0); // Warten bis der Flash Block gelöscht ist
        flash_write_block(0, (char*)voltage_buffer, 4096, voltage_pos);
        voltage_pos+=4096;
        // Der Flash ist noch nicht voll, den nächsten Block löschen
        if(voltage_pos<4*1024*1024)
            flash_erase_block_4k(0, voltage_pos);
    }
    voltage_count++;
    return 1;
}

/*
 * Muss aufgerufen werden, um die letzten verbliebenen Elemente zu speichern
 */
int voltage_finish(void)
```

```
{
    if(voltage_pos>=4*1024*1024)
        return 0;
    long used = fifo16_used(voltage_stream);
    if(used>0)
    {
        fifo16_pop_elements(voltage_stream, voltage_buffer, used);
        flash_wait(0);
        flash_write_block(0, (char*)voltage_buffer, used*2, voltage_pos);
        voltage_pos+=used*2;
        //UARTprintf("written finish to flash\n");
    }
    config.voltage_count = voltage_count;
    UARTprintf("%d samples written\n", voltage_count);
    config_write();
    return 1;
}

/*
 * Initialisierung für die Wiedergabe von Samples aus dem Flash.
 * Hier wird der FIFO zurück gesetzt, die ersten Werte aus dem
 * Flash geladen und der reload_timer gesagt, dass er anfangen
 * soll Werte nachzuladen.
 */
void voltage_start(void)
{
    GPIOPinIntDisable(GPIO_PORTJ_BASE, GPIO_PIN_1);  // Sicherstellen, dass kein Sync Interrupt kommt
    reload_fill = 0;
    voltage_count = config.voltage_count;  // Anzahl der insgesamt zu lesenden Elemente bzw. noch zu lesenden
         Elemente
    UARTprintf("%d samples to read\n", voltage_count);
    voltage_pos = 4096;  // Aktuelle Position im Flash

    fifo16_clear(voltage_stream);
    voltage_fill_fifo();
    voltage_set_next();  // Setzen des ersten Spannungswertes
    GPIOPinIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_1);
}

/*
 * Funktion zum auffüllen des FIFOs bei der Wiedergabe von
 * Samples aus dem Flash. Diese Funktion wird vom reload_timer
 * regelmä ß ig aufgerufen.
 */
int voltage_fill_fifo(void)
{
    // TODO: Mal das hier testen: if(fifo16_free(voltage_stream)<2048)
    if(fifo16_used(voltage_stream)>2000)
        return 1;

    // Elemente, die in diesem Zyklus gelesen werden sollen
    long to_read = voltage_count;
    if(to_read>2048)  // begrenzen auf maximal einen Block
        to_read = 2048;
    flash_read_block(0, (char*)voltage_buffer, to_read*2, voltage_pos);
    fifo16_push_elements(voltage_stream, voltage_buffer, to_read);

    //for(i=0; i<to_read; i++)
    //  UARTprintf("%d ", voltage_buffer[i]);

    //UARTprintf("Read voltages from flash: ");
    //for(i=0; i<to_read; i++)
    //  UARTprintf("%d, ", voltage_buffer[i]);
    //UARTprintf("\n");

    // Aktualisieren der Leseposition und der Anzahl
    // der noch zu lesenden Samples.
    voltage_pos += to_read*2;
    voltage_count -= to_read;
    if(voltage_count == 0) // Dies waren die letzten Daten
        return 0;
    else
        return 1;
}

/*
 * Lesen eines Werts aus dem Fifo und Ausgabe auf den DAC.
 * Liest den vordersten Sample aus dem FIFO und gibt ihn
 * an den DAC weiter. die Funktion wird aus dem
 * Synchronisationstimer aufgerufen.
 */
int voltage_set_next(void)
{
    if(!fifo16_used(voltage_stream))
```

```
        return 0;
    unsigned short dacval = fifo16_pop_element(voltage_stream);
    dac_set_data(dacval);
    return 1;
}
```

A.1.38 voltage.h

```
/*
 * voltage.h
 *
 *  Created on: 06.02.2012
 *      Author: Fabian
 */

#ifndef VOLTAGE_H_
#define VOLTAGE_H_

#include "fifo16.h"

extern fifo16* voltage_stream;
extern volatile long voltage_pos;
extern volatile long voltage_count;
extern volatile int reload_fill;

int voltage_init(void);   // Initialisierung des Fifo, ...

// Speichern von Samples
void voltage_reset(void);  // Von Neuem anfangen den Flash zu beschreiben
int voltage_add(unsigned long voltage); // Hinzufügen eines neuen Sample
int voltage_finish(void);

// Wiedergabe von Samples
void voltage_start(void);
int voltage_fill_fifo(void);
int voltage_set_next(void);

#endif /* VOLTAGE_H_ */
```

A.1.39 voltage.c

```
/*
 * voltage.c
 *
 *  Created on: 06.02.2012
 *      Author: Fabian
 *  Funktionen zur Speicherung und Wiedergabe
 *  der Samples im Flash.
 *  Es wird ein 5000 Byte großer FIFO erstellt in dem die
 *  noch zu speichenden Samples liegen oder im Falle
 *  einer Wiedergabe die noch wiederzugebenden Samples
 *  liegen. Der FIFO wird also mehrfach verwendet, um
 *  Speicher zu sparen.
 */

#include "voltage.h"
#include "flash.h"
#include "calibrate.h"
#include "config.h"
#include "adcdac.h"
#include "led.h"

#include <inc/hw_types.h>
#include <inc/hw_memmap.h>
#include <utils/uartstdio.h>
#include <driverlib/gpio.h>

// Steuerungsvariable für das nachladen von Samples aus dem FIFO
// Diese Variable wird in control.c geschrieben und in reload_timer.c
// gelesen.
volatile int reload_fill;
fifo16* voltage_stream;
```

```
// Aktuelle Position im Flash sowohl beim lesen als auch beim Schreiben
volatile long voltage_pos;

// Schreiben in Flash: Aktuelle Anzahl erhaltener Samples
// Lesen aus dem Flash: Anzahl noch zu lesender Samples
volatile long voltage_count;

// Puffer zum zwischenspeichern von 4kB Samples
unsigned short voltage_buffer[2048];

/*
 * Initialsierung des FIFOs für die Spannungswiedergabe und
 * -speicherung.
 */
int voltage_init(void)
{
    voltage_stream = fifo16_create(5000);
    if(!voltage_stream)
    {
        UARTprintf("error: malloc failed!\n");
        return 0;
    }
    // Überspringen des ersten Blocks, da hier Konfigurationen gespeichert werden
    voltage_pos = 4096;
    voltage_count = 0;
    return 1;
}

/*
 * Initialisierung vor dem Schreiben von neuen Samples
 * in den Flash.
 */
void voltage_reset(void)
{
    // Überspringen des ersten Blocks, da hier Konfigurationen gespeichert werden
    voltage_pos = 4096;
    voltage_count = 0;

    // Abschalten des automatischen Nachladen in den FIFO
    reload_fill = 0;

    // Löschen des ersten Blocks veranlassen
    flash_erase_block_4k(0, voltage_pos);

    // Löschen des FIFOs
    fifo16_clear(voltage_stream);
}

/*
 * Hinzufügen eines neuen Samples in den FIFO
 * zum Schreiben auf den Flash.
 */
int voltage_add(unsigned long voltage)
{
    if(voltage_pos>=4*1024*1024)
        return 0;
    unsigned short dacval = microvolt2dac(voltage);
    fifo16_push_elements(voltage_stream, &dacval, 1);
    if(fifo16_used(voltage_stream)>=2048)
    {
        fifo16_pop_elements(voltage_stream, voltage_buffer, 2048);
        flash_wait(0); // Warten bis der Flash Block gelöscht ist
        flash_write_block(0, (char*)voltage_buffer, 4096, voltage_pos);
        voltage_pos+=4096;
        // Der Flash ist noch nicht voll, den nächsten Block löschen
        if(voltage_pos<4*1024*1024)
            flash_erase_block_4k(0, voltage_pos);
    }
    voltage_count++;
    return 1;
}

/*
 * Muss aufgerufen werden, um die letzten verbliebenen Elemente zu speichern
 */
int voltage_finish(void)
{
    if(voltage_pos>=4*1024*1024)
        return 0;
    long used = fifo16_used(voltage_stream);
    if(used>0)
    {
        fifo16_pop_elements(voltage_stream, voltage_buffer, used);
```

```
        flash_wait(0);
        flash_write_block(0, (char*)voltage_buffer, used*2, voltage_pos);
        voltage_pos+=used*2;
        //UARTprintf("written finish to flash\n");
    }
    config.voltage_count = voltage_count;
    UARTprintf("%d samples written\n", voltage_count);
    config_write();
    return 1;
}

/*
 * Initialisierung für die Wiedergabe von Samples aus dem Flash.
 * Hier wird der FIFO zurück gesetzt, die ersten Werte aus dem
 * Flash geladen und der reload_timer gesagt, dass er anfangen
 * soll Werte nachzuladen.
 */
void voltage_start(void)
{
    GPIOPinIntDisable(GPIO_PORTJ_BASE, GPIO_PIN_1);  // Sicherstellen, dass kein Sync Interrupt kommt
    reload_fill = 0;
    voltage_count = config.voltage_count;  // Anzahl der insgesamt zu lesenden Elemente bzw. noch zu lesenden
         Elemente
    UARTprintf("%d samples to read\n", voltage_count);
    voltage_pos = 4096;  // Aktuelle Position im Flash

    fifo16_clear(voltage_stream);
    voltage_fill_fifo();
    voltage_set_next();  // Setzen des ersten Spannungswertes
    GPIOPinIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_1);
}

/*
 * Funktion zum auffüllen des FIFOs bei der Wiedergabe von
 * Samples aus dem Flash. Diese Funktion wird vom reload_timer
 * regelmäß ig aufgerufen.
 */
int voltage_fill_fifo(void)
{
    // TODO: Mal das hier testen: if(fifo16_free(voltage_stream)<2048)
    if(fifo16_used(voltage_stream)>2000)
        return 1;

    // Elemente, die in diesem Zyklus gelesen werden sollen
    long to_read = voltage_count;
    if(to_read>2048)  // begrenzen auf maximal einen Block
        to_read = 2048;
    flash_read_block(0, (char*)voltage_buffer, to_read*2, voltage_pos);
    fifo16_push_elements(voltage_stream, voltage_buffer, to_read);

    //for(i=0; i<to_read; i++)
    //  UARTprintf("%d ", voltage_buffer[i]);

    //UARTprintf("Read voltages from flash: ");
    //for(i=0; i<to_read; i++)
    //  UARTprintf("%d, ", voltage_buffer[i]);
    //UARTprintf("\n");

    // Aktualisieren der Leseposition und der Anzahl
    // der noch zu lesenden Samples.
    voltage_pos += to_read*2;
    voltage_count -= to_read;
    if(voltage_count == 0) // Dies waren die letzten Daten
        return 0;
    else
        return 1;
}

/*
 * Lesen eines Werts aus dem Fifo und Ausgabe auf den DAC.
 * Liest den vordersten Sample aus dem FIFO und gibt ihn
 * an den DAC weiter. die Funktion wird aus dem
 * Synchronisationstimer aufgerufen.
 */
int voltage_set_next(void)
{
    if(!fifo16_used(voltage_stream))
        return 0;
    unsigned short dacval = fifo16_pop_element(voltage_stream);
    dac_set_data(dacval);
    return 1;
}
```

A.2 Software auf dem PC für MATLAB

A.2.1 gen_find.m

```
% Sucht nach Generatormodule im lokalen Neztwerk.
% Es wird ein UDP Paket an den Port 56936 der
% IP 'broadcast_ip' gesendet, wobei diese IP-
% Adresse fuer gewoehnlich ein Broadcast ist mit
% der Adresse '255.255.255.255'. Anschliessend
% wird auf Antworten gewartet. In den Antworten
% steht die jeweilige IP-Adresse des Absenders.
% Zurueckgegeben wird eine 'cell', welche die
% IP-Adressen als Strings enthaelt.
function addresses = gen_find(broadcast_ip)
    clientsid = 1;
    u = udp(broadcast_ip, 56936);
    u.Timeout = 1.0;
    fopen(u);
    fwrite(u, 'Who is there?');
    %warning off instrument:fread:unsuccessfulRead
    while 1
        [message count]=fread(u);
        if(count > 6 && count < 16)
            new_ip = sprintf('%s', message);
            %fprintf(1, 'New client: %s\n', new_ip);
            addresses{clientsid} = new_ip;
            clientsid = clientsid + 1;
        else
            break;
        end
    end
    %warning on instrument:fread:unsuccessfulRead
    fclose(u);
end
```

A.2.2 gen_connect.m

```
% Aufbau der Verbindungen zu den Generatoren
% in der Zelle 'addresses'.
% Es wird eine Verbindung zu jedem der in
% addresses enthaltenen Adressen eine TCP-
% Verbindung zum Port 56936 hergestellt.
% Es wird ausserdem der Timeout der Verbindungen
% auf eine Sekunde gesetzt, da die spaeter
% benoetigte Funktion fread blockend ist und
% fuer den Fall, dass keine Daten ausgelesen
% werden koennen nicht zu lange wartet.
function [connections failed] = gen_connect(addresses)
    failed = 0;
    for i=1:length(addresses)
        connections(i) = tcpip(addresses{i}, 56936);
        connections(i).OutputBufferSize = 15000;
        connections(i).InputBufferSize = 15000;
        %connections(i).TransferDelay = 'off';
        connections(i).Timeout = 1;
        fopen(connections(i));
        if(length(connections(i).status) ~= 4)
```

```
            fprintf('Unable to connect to %s\n', addresses{i});
            failed = 1;
        end
        fread(connections(i), 20);  % Puffer leeren
    end
end
```

A.2.3 gen_disconnect.m

```
% Abbau der Verbindungen, die in dem Parameter
% connections als Vektor uebergeben werden.
function gen_disconnect(connections)
    for i=1:length(connections)
        fclose(connections(i));
    end
end
```

A.2.4 gen_cmd.m

```
% Senden eines Befehls an einen oder mehrere Generatoren.
% Param: cons: Vektor mit den geoeffneten TCP-Verbindungen
%        cmd: Befehl mit Zeilenumbruch am Ende
% Return: 0=Alles OK, Bestaetigung empfangen
%         1=Fehler, keine Bestaetigung empfangen
function error = gen_cmd(cons, cmd)
    for i=1:length(cons)
        t = cons(i);
        fwrite(t, cmd);
        [line count] = fread(t,1);
        if(count < 1)
            fprintf('Error: Unable to send following command to %s: %s', t.RemoteHost,
                 cmd);
            fclose(t);
            error = 0;
        else
            error = 1;
        end
    end
end
```

A.2.5 gen_count.m

```
% Auslesen der Anzahl der empfangenen Samples.
% Die Funktion bekommt als Parameter einen
% TCP-Socket mit einer offenen Verbindung
% zu einem Generator.
function count = gen_count(con)
    fwrite(con, sprintf('voltagecount\n'));
    [line number] = fread(con,8);
    if(number < 3)
        count = -1;
```

```
        return;
    else
        count = sscanf(sprintf('%s', line(1:length(line)-2)), '%d');
        return;
    end
end
```

A.2.6 gen_identify.m

```
% Funktion zur Identifizierung eines Moduls.
% Es kann ein TCP-Verbindung uebergeben werden,
% um dort den "identify" Befehl auszufuehren.
% Die gruene LED sollte dann einige Male blinken.
function gen_identify(con)
    gen_cmd(con, sprintf('identify\n'));
end
```

A.2.7 gen_repeat.m

```
% Setzen des Modus auf wiederholende Wiedergabe.
% Setzt mit dem Befehl "mode" den Modus auf "free".
% Nach den Befehlen "start" und "sync" wird
% die Spannungssequenz in einer Endlosschleife
% wiederholt.
function error = gen_repeat(con)
    gen_cmd(con, sprintf('mode free\n'));
end
```

A.2.8 gen_send_voltages.m

```
% Senden von Spannungswerten an die Zellspannungsgeneratoren.
% Diese Funktion sendet an alle Verbindungen 'con' die
% Spannungen aus der Matrix 'vol'. Dabei werden die Spannungen
% in Mikrovolt umkodiert, in ASCII-Text gewandelt, durch ein
% Semikolon getrennt und in Bloecken von 120 Werten an die
% Generatoren gesendet. Nachdem ein Block an jeden Generator
% gesendet wurde, wird auf die Bestaetigung der Generatoren
% gewartet. Die Funktion gibt 1 zurueck, wenn ein Fehler
% aufgetreten ist, 0 wenn kein Fehler aufgetraten ist.
% con: Array mit den TCP-Verbindungen
% vol: n*m Matrix mit n Zeilenvektoren mit je m Spannungswerten in Volt
%      oder 1*m Vektor mit m Spannungwerten in Volt, die an jedes der
%      Module gesendet werden.
%
function error = gen_send_voltages(con, vol)
    [num2 voltage_count] = size(vol);
    if(length(con) ~= num2 && num2 ~= 1)
        fprintf('Wrong number of voltage arrays\n');
        error = 1;
        return;
    end
```

```
22
    % Senden des stop Befehls, um sicher zu stellen, dass alle vorherigen
    % Aktionen beendet sind und sich das Modul in einem klar definierten
    % Zustand befindet.
    for i=1:length(con)
27      if(gen_cmd(con(i), sprintf('stop\n')) == 0)
            error = 1;
            return;
        end
    end
32
    % Senden des voltagearray Befehls zur Beginn der Datenuebertragung
    % an die einzelnen Module.
    for i=1:length(con)
        if(gen_cmd(con(i), sprintf('voltagearray\n')) == 0)
37          error = 1;
            return;
        end
    end

42  % Setzen der Blockgroesse.
    blocksize_max = 120;

    % Senden der Spannungswerte im 'blocksize_max'er Block pro Zeile.
    % Jeder Block wird zunaechst an alle Sensoren geschickt, dann
47  % wird die Bestaetigung abgewartet, damit dieser Vorgang effizienter
    % wird.

    for pos = 1:blocksize_max:voltage_count
        % Ausgabe des Fortschritts
52      fprintf('%2.1f%%\n', pos/voltage_count*100);
        % Ermittlung der Blockgroesse fuer die naechste Uebertragung
        blocksize = voltage_count-pos+1;
        % Verringern der Blockgroesse auf maximal blocksize_max Elemente
        if(blocksize > blocksize_max)
57          blocksize = blocksize_max;
        end

        % Erstellen der Datenblocks
        block = zeros(length(con), blocksize);
62      if(num2 == 1)  % Ein Datensatz fuer alle Generatoren
            for i = 1:length(con)
                block(i, 1:blocksize) = vol(pos:pos+blocksize-1);
            end
        else
67          for i = 1:length(con)
                block(i, 1:blocksize) = vol(i, pos:pos+blocksize-1);
            end
        end
        %newline = sprintf('\n');
72      for i = 1:length(con)
            voltagelist{i} = '';
            for element = 1:blocksize-1
                voltagelist{i} = [voltagelist{i} sprintf('%.0f;', block(i, element)
                    *1000000)];
                %voltagelist{i} = [voltagelist{i} int2str(block(i, element)*1000000)
                    ';'];
77          end
            % Anhaengen des letzten Elements ohne ; und mit Zeilenumbruch
            voltagelist{i} = [voltagelist{i} sprintf('%.0f\n', block(i, blocksize)
                *1000000)];
            %voltagelist{i} = [voltagelist{i} int2str(block(i, blocksize)*1000000)
                newline];
```

```
        end
82
        % Senden der Spannung an alle Module
        for i=1:length(con)
            fwrite(con(i), voltagelist{i});
        end
87
        % Warten auf Bestaetigung der Module
        for i=1:length(con)
            [line count] = fread(con(i),1);
            if(count < 1)
92              fprintf('Error: Unable to get ack from module %d\n', i);
                %fclose(con(i));
                error = 1;
                return;
            end
97      end
    end
    fprintf('100.0%%\n');

    % Senden der Bestaetigung fuer das Ende der Uebertragung
102 for i=1:length(con)
        if(gen_cmd(con(i), sprintf('EOF\n')) == 0)
            error = 1;
            return;
        end
107 end

    % Ueberpruefung auf korrekt empfangene Anzahl von Samples
    for i=1:length(con)
        count = gen_count(con(i));
112     if(count ~= voltage_count)
            fprintf('Wrong voltage count received (%d of %d) to module %d\n', count,
                voltage_count, i);
        end
    end
end
```

A.2.9 gen_single.m

```
% Wechseln des Modus in die einfache Wiedergabe.
% Nach den Befehlen "start" und "sync" wird die
% Spannungssequenz nur einmalig wiederholt.
4 function error = gen_single(con)
    gen_cmd(con, sprintf('mode single\n'));
end
```

A.2.10 gen_start.m

```
% Initialisieren der Spannungssequenzwiedergabe.
% Der Befehl "start" initialisiert die in "con"
% angegebenen Module fuer die Wiedergabe der zuvor
4 % programmierten Spannungssequenz
function error = gen_start(con)
    for i = 1:length(con)
```

```
        gen_cmd(con(i), sprintf('start\n'));
    end
end
```

A.2.11 gen_stop.m

```
% Beenden der Synchronisation.
% Dieser Befehl bekommt als Parameter
% eine einzelne TCP-Verbindung, die des
% Masters. Er beendet die Synchronisation.
function error = gen_stop(con)
    gen_cmd(con, sprintf('stop\n'));
end
```

A.2.12 gen_sync.m

```
% Start der Synchronisation.
% Dieser Befehl bekommt eine einzelne
% TCP-Verbindung, die des Masters.
% Er sendet den "sync" Befehl, um die
% Synchronisation zu starten.
function error = gen_sync(con)
    gen_cmd(con, sprintf('sync\n'));
end
```

A.2.13 APIBeispiel.m

```
% Beispielprogramm zum Uebertragen der Spannungssequenz
% aus der Arbeit von Herrn Puettcher. Alternativ kann
% auch ein Sinus uebertragen werden.

clear all;
close all;
clc;

load('signalDownsampled.mat');
addr = gen_find('255.255.255.255');
con = gen_connect(addr);
time = 0:1/20000:1;
gen_stop(con);
tic;

% Uebertragen eiens sinus
%voltages = sin(2*pi*10*time)+1.5;
%gen_send_voltages(con, voltages);

% Ubertragen des Startvorganges
gen_send_voltages(con, signalNeu');

% Messen der benoetigten Zeit
elapsed = toc;
```

```
fprintf('Needed time: %.1f\n', elapsed);

% Starten der Wiedergabe
gen_start(con);
gen_repeat(con);

for i=1:length(addr)
    if(strcmp(addr{i},'192.168.0.128'))
        gen_sync(con(i));
        fprintf('Synced\n');
    end
end

% Beenden der Verbindung
gen_disconnect(con);
```

A.3 MATLAB-Skripte für diese Arbeit

A.3.1 Messungen

A.3.1.1 Kodierung.m

```
% Dieses Skript benutzt das Signal, welches im Rahmen der
% Arbeit von S. Puettcher aufgenommen wurde als typisches
% Signal und errechnet wie stark das Signal mit einer
% Kodierung durch digitale Differenzierung und den
% Huffmancode komprimiert werden kann.

close all;
clear all;
clc;

load('signal.mat');

resolution = 13;  % resolution of the signal in bits
ref_volt = 5.5;  % reference voltage (maximum value of the signal)

% Downsamplen des Signals
dt_orig = abs(time(2)-time(1));
dt_neu = 1/20000; % work with 20kSPS

down_sample_factor = round(dt_neu/dt_orig);
dt = dt_orig*down_sample_factor;

signal_ds = zeros(length(signal_filtered)/down_sample_factor, 1);
time_ds = zeros(length(signal_ds), 1);

for i=1:length(signal_filtered)/down_sample_factor
    signal_ds(i) = signal_filtered(i*down_sample_factor);
    time_ds(i) = time(i*down_sample_factor);
end

% Auf [resolution] Bits begrenzen
for i=1:length(signal_ds)
    signal_ds(i) = round(signal_ds(i)/ref_volt*(2^resolution)); % 5 volt reference
```

```
end

% plot des downsampled und quantisiertem Signal
fig1 = figure(1);
subplot(2,1,1);
plot(time_ds, signal_ds, 'k');
grid on;
title('Quantisiertes Signal', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Amplitude / DAC Wert', 'fontsize', 16, 'interpreter', 'latex');

signal_dif = zeros(length(signal_ds), 1);

signal_dif(1) = signal_ds(1);

% Differenzierung des Signals
for i=2:length(signal_ds)
    signal_dif(i) = signal_ds(i)-signal_ds(i-1);
end

subplot(2,1,2);
plot(time_ds, signal_dif, 'k');
grid on;
ylim([-60 60]);
title('Differenziertes Signal', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Amplitude / DAC Wert', 'fontsize', 16, 'interpreter', 'latex');

fig2 = figure(2);
[sym_count, sym] = hist(signal_dif, -31:31);
bar(sym, sym_count, 'k');
xlim([-20 20]);
grid on;
xlabel('Differenzwert', 'fontsize', 16, 'interpreter', 'latex');
ylabel('H\"aufigkeit', 'fontsize', 16, 'interpreter', 'latex');
title('Histogramm \"uber Verteilung des differenzierten Signals', 'fontsize', 16, '
    interpreter', 'latex');

export_fig(fig1, 'Kodierung_Time.pdf', 'A4L');
export_fig(fig2, 'Kodierung_Hist.pdf', 'A4L');

counter5 = 0;
for i=1:length(signal_dif)
    if(signal_dif(i) <= 5 && signal_dif(i) >= -5)
        counter5 = counter5 + 1;
    end
end

counter7 = 0;
for i=1:length(signal_dif)
    if(signal_dif(i) <= 7 && signal_dif(i) >= -7)
        counter7 = counter7 + 1;
    end
end

% Huffman-Code
[dict, avglen] = huffmandict(sym, sym_count/length(signal_dif));

fprintf('Samples zwischen -5 und 5: %.1f%%\n', counter5/length(signal_dif)*100);
fprintf('Samples zwischen -7 und 7: %.2f%%\n', counter7/length(signal_dif)*100);
fprintf('Benoetigte Bits pro Sample bei Huffman: %.2f Bit\n', avglen);
```

A.3.1.2 ADCLin1.m

```
% Analyse der Liniaritaet des externen AD-Umsetzers.
% Es werden mit dem Zellspannungsgenerator Spannungen
% von 500mV bis 5,5V in 250mV Schritten erzeugt.
% Diese Spannungen werden mit dem Tischmultimeter
% gemssen und in "gemessen" eingetrage. Anschliessend
% wird die Differenz der idealen zur gemessenen Kennlinie
% errechnet und ausgegeben.

clear all;
close all;
clc;

ideal = 0.5:0.25:5.5;
gemessen = [0.4998 0.7500 1.0001 1.250 1.499 1.749 2.000 2.250 2.500 ...
    2.750 3.001 3.250 3.500 3.750 4.001 4.251 4.501 4.751 5.002 ...
    5.251 5.500];

fig1 = figure(1);
stem(ideal, (gemessen-ideal)*1000, 'k', 'LineWidth', 2);
grid on;
ylim([-1.5 2.5]);
xlabel('Sollspannung / V', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Differenz Istspannung-Sollspannung / mV', 'fontsize', 16, 'interpreter', '
    latex');
title('Abweichung der Verst\"arkerkennlinie von der Idealkennlinie', 'fontsize', 16, '
    interpreter', 'latex');

export_fig(fig1, 'Differenz.pdf', 'A4L');
```

A.3.1.3 Frequenzgang_Messung.m

```
% Dieses Script erzeugt eine Reihe von coinus-
% foermigen Spannungen mit einer Amplitude von
% 2,5V bei einem Offset von 3V fuer einen Generator.
% Es wird dann mit einem Fluke 45 Tischmultimeter,
% welches ueber die Seriele Schnittstelle mit
% dem PC verbunden ist die Wechelspannung
% gemessen. So kann ein Ampludengang aufgestellt
% werden. Die Auswertung erfolgt in einem
% anderen Script

clear all;
close all;
clc;

% Initialisierung des COM-Ports
fclose(instrfind);
serial_port = serial('COM11','BaudRate',9600,'Terminator','CR');
set(serial_port,'OutputBufferSize',512);
fopen(serial_port);
set(serial_port,'Timeout',1);

fprintf(serial_port,'rems')     % remote mode
echo = fscanf(serial_port,'%s');
fprintf(serial_port,'vac')     % resistance measurement
echo = fscanf(serial_port,'%s');
fprintf(serial_port,'rate m')   % medium measure mode
```

```
echo = fscanf(serial_port,'%s');

addr = gen_find('255.255.255.255');
con = gen_connect(addr);
gen_repeat(con);

% Erzeugen einer logarithmischen Frequenzachse mit
% 50 Werten pro Dekade von 10Hz bis 40kHz
frequencies=10.^(1:1/50:4);

amplitudes = zeros(1, length(frequencies));

count = 1;
for freq=frequencies
    % Erzeugen der Zeitachse
    if(freq<100)
        time=0:1/20000:3/freq-1/20000;  % <100Hz: 3 Perioden
    elseif(freq<1000)
        time=0:1/20000:20/freq-1/20000; % 100Hz-1kHz: 20 Perioden
    elseif(freq<10000)
        time=0:1/20000:200/freq-1/20000; % 1kHz-10kHz: 200 Perioden
    elseif(freq<100000)
        time=0:1/20000:2000/freq-1/20000; % 10kHz-100kHz: 2000 Perioden
    end
    % Erzeugen der Schwingung
    cosinus = 2.5*cos(2*pi*freq*time)+3;

    % Uebertragen der Schwingung
    gen_send_voltages(con, cosinus);
    gen_start(con);
    gen_sync(con);

    % Einen Moment warten
    fprintf('Total: %.1f%%\n', count/length(frequencies)*100);
    pause(2);

    % Messen der Amplitude
    amplitudes(count) = fluke_read(serial_port);
    count = count + 1;
end

gen_stop(con);
gen_disconnect(con);

fclose(serial_port);

save('amplitudes.mat', 'amplitudes', 'frequencies');
```

A.3.1.4 Frequenzgang_Auswertung.m

```
% Mit diesem Script koennen die Daten, die mit
% Frequenzgang_Messung.m gewonnen wurden
% ausgwertet und dargestellt werden.

close all;
clear all;
clc;

Uin = 2.5/sqrt(2);
load('amplitudes.mat');
```

```
gain = 20*log10(amplitudes/Uin);

freqspec = txtread('Spektrum.txt');
freq = freqspec(:,1);
spec = freqspec(:,2);

fig1 = figure(1);
semilogx(frequencies, gain);
hold on;
grid on;
plot([min(freq) max(freq)], [-3 -3], 'r');
legend('Amplitudengang', '-3dB Grenze', 'location', 'southwest');
title('Amplitudengang des Zellspannungsgenerators', 'fontsize', 16, 'interpreter', '
    latex');
xlabel('Frequenz / Hz', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Aplitudengang / dB', 'fontsize', 16, 'interpreter', 'latex');
ylim([-20 5]);

export_fig(fig1, 'Amplitudengang.pdf', 'A4L');
```

A.3.1.5 fluke_read.m

```
% Funktion zu lesen eines Samples
% aus einem Fluke 45 Tischmultimeter.
% Der Parameter port muss einen
% geoeffneten COM-Port bekommen.

function value = fluke_read(port)
    fprintf(port,'val1?');
    volt = fscanf(port,'%s');
    echo = fscanf(port,'%s');
    value = str2double(volt);
end
```

A.3.1.6 ScopeTisch_SendVolages.m

```


clear all;
voltages = 0.2:0.1:3;

addr{1} = '192.168.0.128';
con = gen_connect(addr);

for i=1:length(voltages)
    gen_cmd(con, sprintf('voltagedac %.0f\n', voltages(i)*1000000));
    pause(1);
end
```

A.3.1.7 ScopeTisch.m

```
% Skript zur Bestimmung des Messfehlers des Oszilloskops
% gegenueber dem Tischmultimeter bzw. dem Generator.

clear all;
close all;
clc;

voltages = 0.2:0.1:3;

timesignal = csvread('tek0001CH1.csv', 15, 0);

signal = timesignal(:,2);
time = timesignal(:,1);

sample_time = 1; % Zeit in Sekunden zwischen zwei Spannungen
dt = time(2)-time(1);  % Zeit zwischen zwei Samples
sample_periode = 1/dt;  % Anzahl der Samples zwischen zwei Spannungen

% Korrekturfaktor fuer die Laenge einer Spannung (anhand des Plots geschaetzt)
sample_periode = sample_periode*1.015;

fig1 = figure(1);
plot(signal);
hold on;
grid on;

% Nach dem ersten Sample suchen, der kleiner 2V ist
for sample_offset=1:length(signal)
    if(signal(sample_offset)<2)
        break;
    end
end

measured = zeros(1, length(voltages));
for i=1:length(voltages)
    start = round(sample_offset+(i-1)*sample_periode+sample_periode/10);
    stop = round(sample_offset+(i)*sample_periode-sample_periode/10);
    measured(i) = mean(signal(start:stop));
    plot([start start], [measured(i)-0.2 measured(i)+0.2], 'r');
    plot([stop stop], [measured(i)-0.2 measured(i)+0.2], 'g');
end

xlabel('Samplenummer', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Ausgangsspannung / V', 'fontsize', 16, 'Interpreter','Latex');
title('Gemessene Ausgangsspannung am Zellspannungsgenerator', 'fontsize', 16, '
    Interpreter','Latex');

fig2 = figure(2);
plot(voltages, measured-voltages, 'k');
grid on;
xlabel('Absolute Spannung am Oszilloskop in Volt', 'fontsize', 16, 'Interpreter','
    Latex');
ylabel('Fehler in Volt', 'fontsize', 16, 'Interpreter','Latex');
title('Spannungsmessungsfehler des Oszilloskops gegen\"uber dem Tischmultimter', '
    fontsize', 16, 'Interpreter','Latex');

export_fig(fig1, 'ScopeTisch_Treppe.pdf', 'A4L');
export_fig(fig2, 'ScopeTisch_Diff.pdf', 'A4L');
```

A.3.1.8 StromAuswertung.m

```
% Skript zur Auswertung des Fehlers in der Strommessung.
% Die Werte fuer die Stroeme wurden per Hand gemessen
% und hier eingetragen.
% Das Skript analysiert die Abhaengigkeit des gemessenen
% Stromes von der Ausgangsspannung des Generators.

close all;
clear all;
clc;

current = [200 400 600 800 1000 1200 1400 1600 1800 2000];
voltage = [0.1 1 2 3 4 5 6];
colors = ['r', 'g', 'y', 'b', 'k', 'c', 'm'];
measured = [
315 549  811    1007    1276    1500    1692    1692    1692    1692;
376 596  846    1080    1299    1538    1692    1692    1692    1692;
423 653  896    1126    1357    1599    1692    1692    1692    1692;
465 707  949    1172    1415    1669    1692    1692    1692    1692;
530 776 1007    1245    1476    1688    1692    1692    1692    1692;
592 822 1072    1319    1550    1692    1692    1692    1692    1692;
669 922 1146    1384    1619    1692    1692    1692    1692    1692];

fig1 = figure(1);
hold on;
grid on;
plot(current, flipud(measured));
legend('6V', '5V', '4V', '3V', '2V', '1V', '0,1V', 'location', 'southeast');
title('Stromkennlinien bei verschiedenen Ausgangsspannungen', 'fontsize', 16, '
    interpreter', 'latex');
xlabel('Tats\"achlich flie{\ss}ender Strom / $\mu$A', 'fontsize', 16, 'interpreter', '
    latex');
ylabel('Durch Generator gemessener Strom / $\mu$A', 'fontsize', 16, 'interpreter', '
    latex');

current2 = [200 400 600 800 1000];
voltage2 = [0.1 1 2 3 4 5 6];
measured2 = [
315 549  811    1007    1276;
376 596  846    1080    1299;
423 653  896    1126    1357;
465 707  949    1172    1415;
530 776 1007    1245    1476;
592 822 1072    1319    1550;
669 922 1146    1384    1619];

offsets = zeros(length(voltage2));
gains = zeros(length(voltage2));
for i=1:length(voltage2)
    p = polyfit(current2, measured2(i,:), 1);
    offsets(i) = p(2);
    gains(i) = p(1);
end

fig2 = figure(2);
plot(voltage2, offsets, 'k');
grid on;
xlabel('Ausgangsspannung / V', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Offset / $\mu$V', 'fontsize', 16, 'interpreter', 'latex');
title('Offset in Abh\"angigkeit der Ausgangsspannung', 'fontsize', 16, 'interpreter',
    'latex');
```

```
fig3 = figure(3);
plot(voltage2, gains, 'k');
grid on;
xlabel('Ausgangsspannung / V', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Gain', 'fontsize', 16, 'interpreter', 'latex');
title('Verst\"arkung in Abh\"angigkeit der Ausgangsspannung', 'fontsize', 16, '
    interpreter', 'latex');

export_fig(fig1, 'Stromkompensation_Kennlinien.pdf', 'A4L');
export_fig(fig2, 'Stromkompensation_Offset.pdf', 'A4L');
export_fig(fig3, 'Stromkompensation_Gain.pdf', 'A4L');
```

A.3.1.9 Transient4.m

```
% Skript zur Auswertung der gemessenen Spannung
% bei der Simulation eiens Startvorganges.
% Das Skript subtrahiert die am Generator gemssene
% Spannung von der origianlen Spannung wie sie
% in der Arbeit von S. Puettcher verwendet wurde.
% Anschliessend werden eine Reihe an Grafigen
% fuer die Ausertung des Ergebnis erstellt.

clear all;
close all;
clc;
load('original.mat');
original = signal;
aufgenommen = csvread('aufgenommen.csv', 15, 0);

rec_time = aufgenommen(:,1);
rec_sig = aufgenommen(:,2);

dt = rec_time(2)-rec_time(1);

start_rec = 7227;  % Offset im aufgenommenen Signal

rec_sig_short = rec_sig(start_rec:length(original));

if(length(rec_sig_short)<length(original))
    original = original(1:length(rec_sig_short));
end

time = 0:dt:(length(rec_sig_short)-1)*dt;

fig1 = figure(1);
plot(time, original, 'r');
hold on;
plot(time, rec_sig_short, 'b');
xlabel('Zeit / s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / V', 'fontsize', 16, 'interpreter', 'latex');
title('Startvorgang - Vergleich des originalen und gemessenen Spannungsverlaufs', ...
    'fontsize', 16, 'interpreter', 'latex');
legend('Gesendete Spannung', 'Aufgenommene Spannung', 'Location', 'SouthEast');
grid on;

fig2 = figure(2);
plot(time, rec_sig_short-original, 'k');
xlabel('Zeit / s');
ylabel('Differenz Gesendet zu Aufgenommen / V', 'fontsize', 16, 'interpreter', 'latex'
    );
```

```
title('Startvorgang - Differenz des originalen und gemessenen Spannungsverlaufs', ...
    'fontsize', 16, 'interpreter', 'latex');
grid on;

fig3 = figure(3);
subplot(1,2,1);
plot(time, original, 'r');
hold on;
plot(time, rec_sig_short, 'b');
grid on;
xlim([1.122 1.13]);
xlabel('Zeit / s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / V', 'fontsize', 16, 'interpreter', 'latex');
title('Spannung zum Beginn der Sequenz', 'fontsize', 16, 'interpreter', 'latex');
legend('Originale Spannung', 'Aufgenommene Spannung', 'Location', 'SouthEast');

subplot(1,2,2);
plot(time, original, 'r');
hold on;
plot(time, rec_sig_short, 'b');
grid on;
xlim([9.2 9.3]);
xlabel('Zeit / s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / V', 'fontsize', 16, 'interpreter', 'latex');
title('Spannung zum Ende der Sequenz', 'fontsize', 16, 'interpreter', 'latex');
legend('Originale Spannung', 'Aufgenommene Spannung', 'Location', 'SouthEast');

% Histogramm
fig4 = figure(6);
[n,xout] = hist(rec_sig_short-original, -0.1:0.001:0.1);
bar(xout*1000,n, 'k')
xlim([-0.015 0.02]*1000);
grid on;
xlabel('Abweichung / mV', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Anzahl Samples', 'fontsize', 16, 'Interpreter','Latex');
title('Statistische Verteilung der Abweichungen', 'fontsize', 16, 'Interpreter','Latex
    ');

fprintf('Mittlere Abweichung: %f\n', mean(rec_sig_short-original));

export_fig(fig1, 'Transient_Verlauf.pdf', 'A4L');
export_fig(fig2, 'Transient_Differenz.pdf', 'A4L');
export_fig(fig3, 'Transient_Anfang_Ende.pdf', 'A4L');
export_fig(fig4, 'Transient_Histogramm.pdf', 'A4L');
```

A.3.1.10 Transient6_Aufnahme.m

```
% Uebertragung des Startvorganges eines PWK
% an vier verschiedene Module und Start der
% Wiedergabe.

clear all;
close all;
clc;

load('signalDownsampled.mat');
%addr = gen_find('255.255.255.255');
addr = {'192.168.0.128' '192.168.0.129' '192.168.0.130' '192.168.0.131'};
con = gen_connect(addr);
time = 0:1/20000:1;
```

```
gen_cmd(con, sprintf(';EOF\n'));
gen_stop(con);
tic;
gen_send_voltages(con, signalNeu');
elapsed = toc;
fprintf('Needed time: %.1f\n', elapsed);
gen_start(con);
gen_repeat(con);

for i=1:length(addr)
    if(strcmp(addr{i},'192.168.0.128'))
        gen_sync(con(i));
        fprintf('Synced\n');
    end
end
gen_disconnect(con);
```

A.3.1.11 Transient6_Auswertung.m

```
% Skript zur Auswertung der gemessenen Spannung
% bei der Simulation eiens Startvorganges.
% Das Skript subtrahiert die am Generator gemssene
% Spannung von der origianlen Spannung wie sie
% in der Arbeit von S. Puettcher verwendet wurde.
% Anschliessend werden eine Reihe an Grafigen
% fuer die Ausertung des Ergebnis erstellt.

clear all;
close all;
clc;

timevoltage = csvread('tek0009ALL.csv', 15, 0);

samplecut = 22000;

time = timevoltage(:,1);
voltage1 = timevoltage(:,2);
voltage2 = timevoltage(:,3);
voltage3 = timevoltage(:,4);
voltage4 = timevoltage(:,5);

voltage4 = voltage4-voltage3;
voltage3 = voltage3-voltage2;
voltage2 = voltage2-voltage1;

voltage1 = voltage1(samplecut:length(voltage1));
voltage2 = voltage2(samplecut:length(voltage2));
voltage3 = voltage3(samplecut:length(voltage3));
voltage4 = voltage4(samplecut:length(voltage4));

dt = time(2)-time(1);
time = 0:dt:dt*length(voltage1)-dt;

fig1 = figure(1);
plot(time, [voltage1 voltage2 voltage3 voltage4]);
legend('Zelle 1', 'Zelle 2', 'Zelle 3', 'Zelle 4', 'location', 'southeast');
grid on;
xlim([0 5]);
title('Ausgangsspannung der Zellen bei Serienschaltung', 'fontsize', 16, 'interpreter'
    , 'latex');
```

```
xlabel('Zeit /s', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / V', 'fontsize', 16, 'interpreter', 'latex');

export_fig(fig1, 'Ausgangsspannung_Serienschaltung.pdf', 'A4L');
```

A.3.2 Simulationen

A.3.2.1 ADCFehler.m

```
clear all;

Uin = 0:0.0001:5;

Umax0 = calcuout(1, Uin, 0);
%Umax1 = calcuout(1, Uin, 1);

%plot(Uin, (Umax1-Uin)./Uin*100, 'k');
%hold on;
plot(Uin, (Umax0-Uin)./Uin*100, 'k');
ylim([0 30]);
grid on;
title('Maximaler Fehler des ADC ohne Quantisierung', 'fontsize', 16);
xlabel('U_{in} / V', 'fontsize', 16);
ylabel('Fehler in %', 'fontsize', 16);
export_fig(gcf, 'Maximum_ADC_Error.pdf', 'A4L');
```

A.3.2.2 calcuout.m

```
function Uout = calcuout(dir, Uin, quant)
    R1 = 6800;
    R2 = 2*6800;
    Uref = 2.5;
    N = 10;
    Offset = 4;
    Gain = 4;

    Uadc = Uin*R1*(1+dir*0.01)/((R1+R2)*(1-dir*0.01));

    ADCout = Uadc*2^N/(Uref-dir*0.009)*((1+dir*Gain/2^N)) + dir*Offset;

    if(quant==1)
        ADCout = round(ADCout);
    end

    Uout = ADCout/2^N*Uref*3;
end
```

A.3.2.3 OPA548.m

```
% Dieses Skript wertet die Simulationen des
% OPA548 Power-OP aus und stellt sie dar.
% Die Simulationen sind mit PSpice erstellt.

clear all;
close all;

timevoltage = txtread('OPA548_ohne_c.txt');
time = timevoltage(:,1);
voltage = timevoltage(:,2);

fig1 = figure(1);
plot(time*1000000000-50, voltage, 'k');
grid on;
xlim([0 300]);
title('Ausgangsspannung OPA548 bei Zuschaltung einer Last, Simulation', 'fontsize',
    16, 'interpreter', 'latex');
xlabel('Zeit / $ns$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

fig2 = figure(2);
plot(time*1000000000-2050, voltage, 'k');
grid on;
xlim([0 900]);
title('Ausgangsspannung OPA548 bei Abschaltung einer Last, Simulation', 'fontsize',
    16, 'interpreter', 'latex');
xlabel('Zeit / $ns$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');


timevoltage = txtread('OPA548_mit_c.txt');
time = timevoltage(:,1);
voltage = timevoltage(:,2);

fig3 = figure(3);
plot(time*1000000, voltage, 'k');
grid on;
xlim([0 20]);
title('Ausgangsspannung OPA548 bei Zuschaltung einer Last, Kapazitive Grundlast,
    Simulation', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / $\mu s$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

fig4 = figure(4);
plot(time*1000000-20, voltage, 'k');
grid on;
xlim([0 30]);
title('Ausgangsspannung OPA548 bei Abschaltung einer Last, Kapazitive Grundlast,
    Simulation', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / $\mu s$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

export_fig(fig1, 'OPA548_Rising.pdf', 'A4L');
export_fig(fig2, 'OPA548_Falling.pdf', 'A4L');
export_fig(fig3, 'OPA548_Rising_C.pdf', 'A4L');
export_fig(fig4, 'OPA548_Falling_C.pdf', 'A4L');
```

A.3.2.4 OPA564.m

```
% Dieses Skript wertet die Simulationen des
% OPA564 Power-OP aus und stellt sie dar.
% Die Simulationen sind mit PSpice erstellt.

clear all;
close all;

timevoltage = txtread('OPA564_ohne_c.txt');
time = timevoltage(:,1);
voltage = timevoltage(:,2);

fig1 = figure(1);
plot(time*1000000000-8050, voltage, 'k');
grid on;
xlim([0 300]);
title('Ausgangsspannung OPA564 bei Zuschaltung einer Last, Simulation', 'fontsize',
    16, 'interpreter', 'latex');
xlabel('Zeit / $ns$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

fig2 = figure(2);
plot(time*1000000000-4050, voltage, 'k');
grid on;
xlim([0 900]);
title('Ausgangsspannung OPA564 bei Abschaltung einer Last, Simulation', 'fontsize',
    16, 'interpreter', 'latex');
xlabel('Zeit / $ns$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');


timevoltage = txtread('OPA564_mit_c.txt');
time = timevoltage(:,1);
voltage = timevoltage(:,2);

fig3 = figure(3);
plot(time*1000000-78, voltage, 'k');
grid on;
xlim([0 35]);
title('Ausgangsspannung OPA564 bei Zuschaltung einer Last, 100$\Omega$ und 1$\mu$F
    Grundlast, Simulation', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / $\mu s$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

fig4 = figure(4);
plot(time*1000000-38, voltage, 'k');
grid on;
xlim([0 40]);
title('Ausgangsspannung OPA564 bei Abschaltung einer Last, 100$\Omega$ und 1$\mu$F
    Grundlast, Simulation', 'fontsize', 16, 'interpreter', 'latex');
xlabel('Zeit / $\mu s$', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Spannung / $V$', 'fontsize', 16, 'interpreter', 'latex');

export_fig(fig1, 'OPA564_Rising.pdf', 'A4L');
export_fig(fig2, 'OPA564_Falling.pdf', 'A4L');
export_fig(fig3, 'OPA564_Rising_C.pdf', 'A4L');
export_fig(fig4, 'OPA564_Falling_C.pdf', 'A4L');
```

A.3.2.5 Startvorgang_Fenster.m

```
% Dieses Skript ist fuer die Berechnungen und
% Analysen des Startvorganges verantwortlich.
% Das Ziel ist es heraus zu finden, wie gross
% die Bandbreite des Generators sein muss,
% um Momente wie den Startvorgang eines PKW
% ausreichend genau nachbilden zu koennen.

% Diese Version analysiert das Signal mit Hilfe
% der Fensterung im Zeitbereich, Transformation
% in den Frequenzbereich, Filterung des Signls,
% Zuruecktransformation in den Zeitbereich
% und Bildung der Differenz.

close all;
clear all;
clc;
load('vito 10s.mat');
time = data_scaled(:,1);
signal = data_scaled(:,2);

% Fenstern und plotten des unveraenderten Signals
mywindow = tukeywin(length(signal), 1*2/(time(length(time))-time(1)));
fig1 = figure(1);
subplot(2,1,1);
plot(time, mywindow, 'k');
title('Fensterfunktion und unver\"andertes Zeitsignal', 'fontsize', 16, 'Interpreter',
    'Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Faktor', 'fontsize', 16, 'Interpreter','Latex');
grid on;
subplot(2,1,2);
plot(time, signal, 'k');
grid on;

signal = signal.*mywindow;

% Plot des unveraenderten Signals
fig2 = figure(2);
subplot(2,1,1);
plot(time, signal, 'k');
grid on;
title('Originales Signal, mit Fenster', 'fontsize', 16, 'Interpreter','Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Zellspannung / V', 'fontsize', 16, 'Interpreter','Latex');


% Berechnung der FFT des gefensterten Signals
dt = time(2)-time(1);
fa = 1/dt;
df = fa/length(time);
f = -fa/2:df:fa/2-df;

spec = fft(signal);
subplot(2,1,2);
plot(f, 20*log10(abs(fftshift(spec)/length(time))), 'k');
grid on;
ylabel('Amplitude / dBV', 'fontsize', 16, 'Interpreter','Latex');
xlabel('Frequenz / Hz', 'fontsize', 16, 'Interpreter','Latex');
title('Originales Spektrum, mit Fenster', 'fontsize', 16, 'Interpreter','Latex');

% Abschneiden der Frequenzen >cutoff
cutoff = 2500;  % cutoff frequency in Hz
```

```
spec_filtered = fftshift(spec);

lowerI = 1;
for i = 1:length(spec_filtered)
    if(f(i) > -(cutoff-1))
        lowerI = i;
        break;
    end
end

upperI = length(spec_filtered)-lowerI+2;

spec_filtered(1:lowerI) = 0;
spec_filtered(upperI:length(spec_filtered)) = 0;

% Plot des beschraeankten Spektrums
fig3 = figure(3);
subplot(2,1,1);
plot(f, 20*log10(abs(spec_filtered)/length(time)), 'k');
grid on;
ylabel('Amplitude / dBV', 'fontsize', 16, 'Interpreter','Latex');
xlabel('Frequenz / Hz', 'fontsize', 16, 'Interpreter','Latex');
title('Tiefpass gefiltertes Spektrum, mit Fenster', 'fontsize', 16, 'Interpreter','
    Latex');

% Zurueckwandeln des beschraenkten Spektrums in den Zeitbereich
spec_filtered = ifftshift(spec_filtered);
signal_filtered = ifft(spec_filtered);

% Plot des gefilterten Signals
subplot(2,1,2);
plot(time, abs(signal_filtered), 'k');
grid on;
title('R\"ucktransformiertes Signal, mit Fenster', 'fontsize', 16, 'Interpreter','
    Latex', 'Interpreter','Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Zellspannung / V', 'fontsize', 16, 'Interpreter','Latex');


% Plot der Differenz der beiden Signale
signal_diff = signal-signal_filtered;
% komplettes Signal
fig4 = figure(4);
plot(time, signal-signal_filtered, 'k');
grid on;
title('Differenz originales und gefiltertes Signal, mit Fenster', 'fontsize', 16, '
    Interpreter','Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','Latex');
ylim([-0.1 0.1]);
% y beschraenktes Signal
fig5 = figure(5);
plot(time, signal-signal_filtered, 'k');
grid on;
title('Differenz originales und gefiltertes Signal, mit Fenster', 'fontsize', 16, '
    Interpreter','Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','Latex');
ylim([-0.01 0.01]);
% x und y beschraenktes Signal
fig6 = figure(6);
plot(time, signal-signal_filtered, 'k');
grid on;
```

```
title('Differenz originales und gefiltertes Signal, mit Fenster', 'fontsize', 16, '
    Interpreter','Latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','Latex');
ylim([-0.01 0.01]);
xlim([3 3.01]);

% Histogramm
fig7 = figure(7);
[n,xout] = hist(signal-signal_filtered, -0.006:0.0001:0.006);
bar(xout,n, 'k')
xlim([-0.005 0.005]);
grid on;
xlabel('Abweichung / mV', 'fontsize', 16, 'Interpreter','Latex');
ylabel('Anzahl Samples', 'fontsize', 16, 'Interpreter','Latex');
title('Histogramm der Differenz zwischen originalem und gefiltertem Signal, mit
    Fenster', 'fontsize', 16, 'Interpreter','Latex');
std_abw = std(signal-signal_filtered);
fprintf('Standardabweichung: %.3fmV\n', std_abw*1000);

counter = 0;
for i=1:length(signal_diff)
    if(signal_diff(i) <= std_abw && signal_diff(i) >= -std_abw)
        counter = counter + 1;
    end
end
fprintf('Samples in sigma: %.1f%%\n', counter/length(signal_diff)*100);

% Berechnung des Effektivwerts der Abweichung mit Parsevall
N = length(signal);
Etime = sum(signal.^2);
Espec = sum(abs(spec).^2)/N;
Espec_filtered = sum(abs(spec_filtered).^2)/N;
e=1-Espec_filtered/Espec;
E_diff = Espec-Espec_filtered;
dAb = sqrt(E_diff/N);  % best case
dAw = sqrt(E_diff);  % worst case
fprintf('Best case Effektivabweichung: %.4fmV\n', dAb*1000);
fprintf('Worst case Effektivabweichung: %.1fmV\n', dAw*1000);

% Speichern der Bilder in PDFs
export_fig(fig1, 'Start_Original_Unbearbeitet.pdf', 'A4L');
export_fig(fig2, 'Start_Original.pdf', 'A4L');
export_fig(fig3, 'Start_LP.pdf', 'A4L');
export_fig(fig4, 'Start_Diff_Komplett.pdf', 'A4L');
export_fig(fig5, 'Start_Diff_x_lim.pdf', 'A4L');
export_fig(fig6, 'Start_Diff_xy_lim.pdf', 'A4L');
export_fig(fig7, 'Start_Hist.pdf', 'A4L');
```

A.3.2.6 Startvorgang_Spiegel.m

```
% Dieses Skript ist fuer die Berechnungen und
% Analysen des Startvorganges verantwortlich.
% Das Ziel ist es heraus zu finden, wie gross
% die Bandbreite des Generators sein muss,
% um Momente wie den Startvorgang eines PKW
% ausreichend genau nachbilden zu koennen.

% Diese Version analysiert das Signal mit Hilfe
% der Spiegelung des Signals im Zeitberich, Trans-
```

```
% formation in den Freuqnezbereich, Filterung des
% Signals, Zuruecktransformation in den Zeitbereich
% und Bildung der Differenz.

close all;
clear all;
clc;
load('vito 10s.mat');
time = data_scaled(:,1);
signal = data_scaled(:,2);

fig1 = figure(1);
plot(time, signal, 'k');
grid on;
title('Spannungsverlauf eines Startvorgangs', 'fontsize', 16, 'Interpreter','latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','latex');
ylabel('Spannung / V', 'fontsize', 16, 'Interpreter','latex');

% Abspeichern des originalen Signals
save('original.mat', 'signal');

signal = [signal;flipud(signal)];
dt = time(2)-time(1);
time = 0:dt:dt*length(signal)-dt;

% Plot des unveraenderten Signals
fig2 = figure(2);
subplot(2,1,1);
plot(time, signal, 'k');
grid on;
title('Originales Signal, mit Spiegelung', 'fontsize', 16, 'Interpreter','latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','latex');
ylabel('Zellspannung / V', 'fontsize', 16, 'Interpreter','latex');


% Berechnung der FFT des gefensterten Signals
dt = time(2)-time(1);
fa = 1/dt;
df = fa/length(time);
f = -fa/2:df:fa/2-df;

spec = fft(signal);
subplot(2,1,2);
plot(f, 20*log10(abs(fftshift(spec)/length(time))), 'k');
grid on;
ylabel('Amplitude / dBV', 'fontsize', 16, 'Interpreter','latex');
xlabel('Frequenz / Hz', 'fontsize', 16, 'Interpreter','latex');
title('Originales Spektrum, mit Spiegelung', 'fontsize', 16, 'Interpreter','latex');

% Abschneiden der Frequenzen >cutoff
cutoff = 2500;  % cutoff frequency in Hz
spec_filtered = fftshift(spec);

lowerI = 1;
for i = 1:length(spec_filtered)
    if(f(i) > -(cutoff-1))
        lowerI = i;
        break;
    end
end

upperI = length(spec_filtered)-lowerI+2;
```

```
spec_filtered(1:lowerI) = 0;
spec_filtered(upperI:length(spec_filtered)) = 0;

% Plot des beschraeankten Spektrums
fig3 = figure(3);
subplot(2,1,1);
plot(f, 20*log10(abs(spec_filtered)/length(time)), 'k');
grid on;
ylabel('Amplitude / dBV', 'fontsize', 16, 'Interpreter','latex');
xlabel('Frequenz / Hz', 'fontsize', 16, 'Interpreter','latex');
title('Tiefpass begrenztes Spektrum, mit Spiegelung', 'fontsize', 16, 'Interpreter','
    latex');

% Zurueckwandeln des beschraenkten Spektrums in den Zeitbereich
spec_filtered = ifftshift(spec_filtered);
signal_filtered = ifft(spec_filtered);

% Plot des gefilterten Signals
subplot(2,1,2);
plot(time, abs(signal_filtered), 'k');
grid on;
title('Ruecktransformiertes Signal, mit Spiegelung', 'fontsize', 16, 'interpreter', '
    latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter', 'latex');
ylabel('Zellspannung / V', 'fontsize', 16, 'Interpreter', 'latex');


% Plot der Differenz der beiden Signale
signal_diff = signal-signal_filtered;
% komplettes Signal
fig4 = figure(4);
plot(time, signal-signal_filtered, 'k');
grid on;
title('Differenz originales und gefiltertes Signal, mit Spiegelung', 'fontsize', 16, '
    Interpreter','latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','latex');
ylim([-0.1 0.1]);
xlim([0 10]);
% y beschraenktes Signal
fig5 = figure(5);
plot(time, signal-signal_filtered, 'k');
grid on;
title('Differenz originales und gefiltertes Signal, mit Spiegelung', 'fontsize', 16, '
    Interpreter','latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','latex');
ylim([-0.01 0.01]);
% x und y beschraenktes Signal
fig6 = figure(6);
plot(time, signal-signal_filtered, 'k');
grid on;
title('Differenz originales und gefiltertes Signal, mit Spiegelung', 'fontsize', 16, '
    Interpreter','latex');
xlabel('Zeit / s', 'fontsize', 16, 'Interpreter','latex');
ylabel('Differenz / V', 'fontsize', 16, 'Interpreter','latex');
ylim([-0.01 0.01]);
xlim([3 3.01]);

% Histogramm
fig7 = figure(7);
[n,xout] = hist(signal-signal_filtered, -0.006:0.0001:0.006);
bar(xout,n, 'k')
```

```
xlim([-0.005 0.005]);
grid on;
xlabel('Abweichung / mV', 'fontsize', 16, 'Interpreter','latex');
ylabel('Anzahl Samples', 'fontsize', 16, 'Interpreter','latex');
title('Histogramm der Differenz zwischen originalem und gefiltertem Signal, mit
    Spiegelung', 'fontsize', 16, 'Interpreter','latex');
std_abw = std(signal-signal_filtered);
fprintf('Standardabweichung: %.3fmV\n', std_abw*1000);

counter = 0;
for i=1:length(signal_diff)
    if(signal_diff(i) <= std_abw && signal_diff(i) >= -std_abw)
        counter = counter + 1;
    end
end
fprintf('Samples in sigma: %.1f%%\n', counter/length(signal_diff)*100);


% Berechnung des Effektivwerts der Abweichung mit Parsevall
N = length(signal);
Etime = sum(signal.^2);
Espec = sum(abs(spec).^2)/N;
Espec_filtered = sum(abs(spec_filtered).^2)/N;
e=1-Espec_filtered/Espec;
E_diff = Espec-Espec_filtered;
dAb = sqrt(E_diff/N);  % best case
dAw = sqrt(E_diff);  % worst case
fprintf('Best case Effektivabweichung: %.4fmV\n', dAb*1000);
fprintf('Worst case Effektivabweichung: %.1fmV\n', dAw*1000);

% Speichern des gefilterten Signals
signal_filtered = signal_filtered(1:length(signal_filtered)/2);
time = time(1:length(time)/2);
save('signal_filtered.mat', 'signal_filtered');
save('time_filtered.mat', 'time');

% Speichern der Bilder in PDFs
export_fig(fig1, 'Start_Original_Unbearbeitet.pdf', 'A4L');
export_fig(fig2, 'Start_Original.pdf', 'A4L');
export_fig(fig3, 'Start_LP.pdf', 'A4L');
export_fig(fig4, 'Start_Diff_Komplett.pdf', 'A4L');
export_fig(fig5, 'Start_Diff_x_lim.pdf', 'A4L');
export_fig(fig6, 'Start_Diff_xy_lim.pdf', 'A4L');
export_fig(fig7, 'Start_Hist.pdf', 'A4L');
```

A.3.2.7 Tiefpass.m

```
% Veranschaulichung der Filterung durch den Tiefpass
% hinter dem DA-Umsetzer und die si-Funktion des DA-
% Umsetzers selbst im Spektrum.

close all;
clear all;
clc;

% Parameter
freq = -5000:1:25000;
R=10000;
C=0.0000000041;
```

```
% Berechnung des Signals
signal = trapez(freq, 0, 2500, 2500) + trapez(freq, 20000, 2500, 2500);

% Berechnung der Daempfungen
lowpass = 1./(1 + (2*pi*freq*R*C).^2);
si = abs(sinc(freq/20000));

fig1 = figure(1);
plot(freq, signal, freq, si, freq, lowpass);

ylim([0 1.4]);
grid on;
hold on;
xlabel('Frequenz / Hz', 'fontsize', 16, 'interpreter', 'latex');
ylabel('Amplitude', 'fontsize', 16, 'interpreter', 'latex');
title('Filterung durch Tiefpass und si-Funktion im Amplitudenspektrum', 'fontsize',
    16, 'interpreter', 'latex');
legend('Spektrum des diskreten Signals', 'Daempfung durch die si-Funktion', 'Daempfung
     durch den Tiefpass');

fig2 = figure(2);
plot(freq, 20*log10(si.*lowpass));
ylim([-50 5]);
grid on;
xlabel('Frequenz / Hz', 'fontsize', 16, 'interpreter', 'latex');
ylabel('D\"ampfung / dB', 'fontsize', 16, 'interpreter', 'latex');
title('Gesamtd\"ampfung durch Tiefpass und si-Funktion', 'fontsize', 16, 'interpreter'
    , 'latex');

export_fig(fig1, 'Tiefpass_Spektrum.pdf', 'A4L');
export_fig(fig2, 'Tiefpass_Daempfung.pdf', 'A4L');

fprintf('Daempfung bei 17,5kHz: %.1fdB\n', -20*log10(1./(1 + (2*pi*17500*R*C).^2)*abs(
    sinc(17500/20000))));
fprintf('Daempfung bei 2,5kHz durch si: %.2fdB\n', -20*log10(abs(sinc(2500/20000))));
```

A.3.2.8 trapez.m

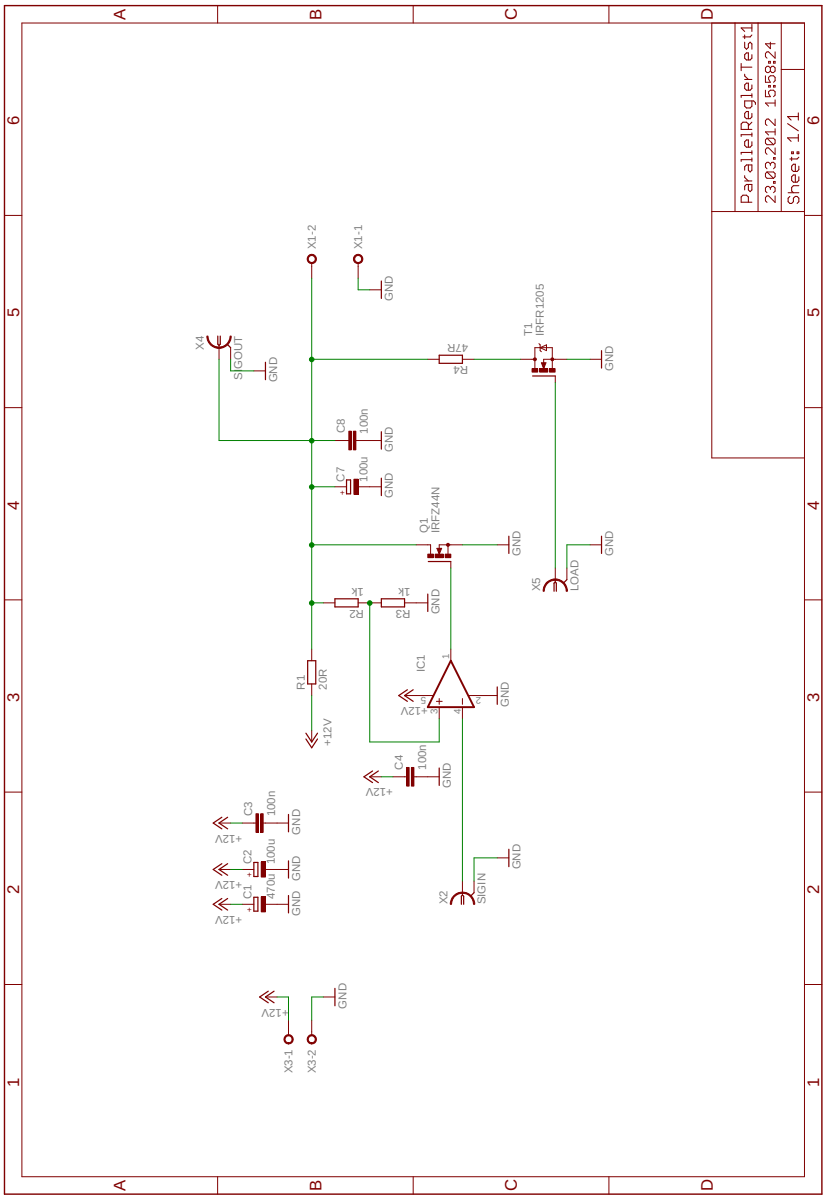
```
% Erstellen eines Trapezes.
% mitte ist die Mitte des Trapezes, bezogen auf den
% Vektor x.
% b1 ist der Abstand von der Mitte bis zum aeusseren
% Knick der Kurve.
% b2 ist der Abstand von der Mitte bis zum inneren
% Knick der Kurve
%      _______
%     /       \
% ___/         \____
%   b1 b2 |  b2 b1

function trap = trapez(x, mitte, b1, b2)
    trap = zeros(1, length(x));
    counter = 1;
    for pos=x
        if(pos<mitte-b1)  % links
            trap(counter) = 0;
        elseif(pos<mitte-b2)  % steigende flanke
            trap(counter) = (pos-(mitte-b1))/(b1-b2);
        elseif(pos<mitte+b2)  % oben
            trap(counter) = 1;
```

```
        elseif(pos<mitte+b1)  % rechte flanke
            trap(counter) = 1-(pos-(mitte+b2))/(b1-b2);
        else  % rechts
            trap(counter) = 0;
        end
        counter = counter + 1;
    end
end
```

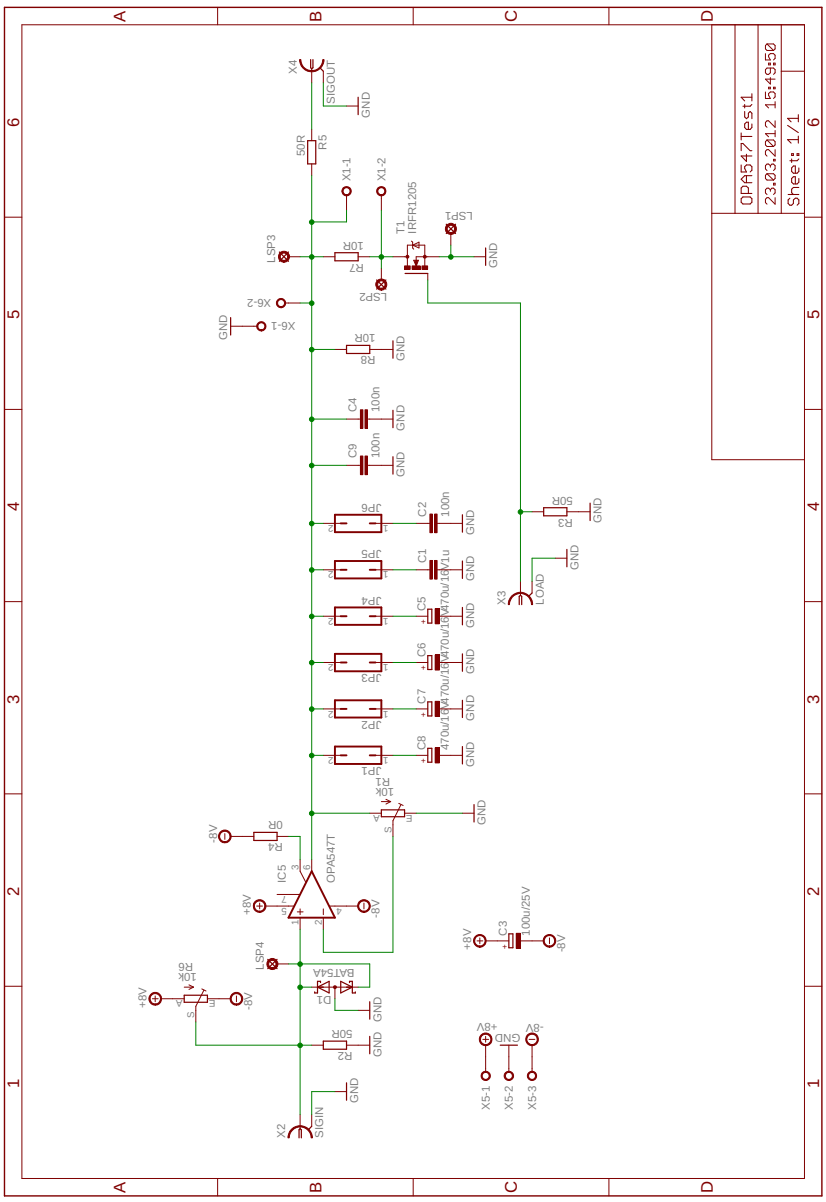
B Hardware

B.1 Schaltpläne

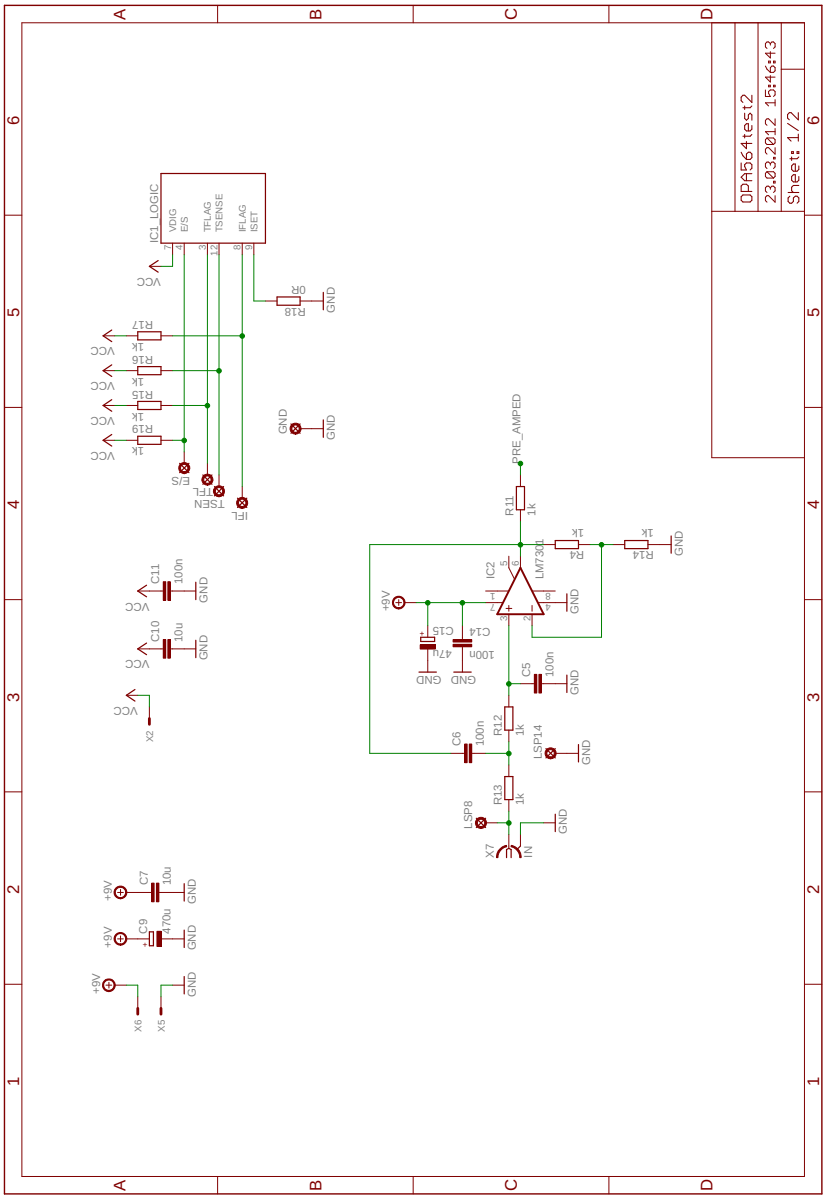
B.1.1 ParallelRegler.sch

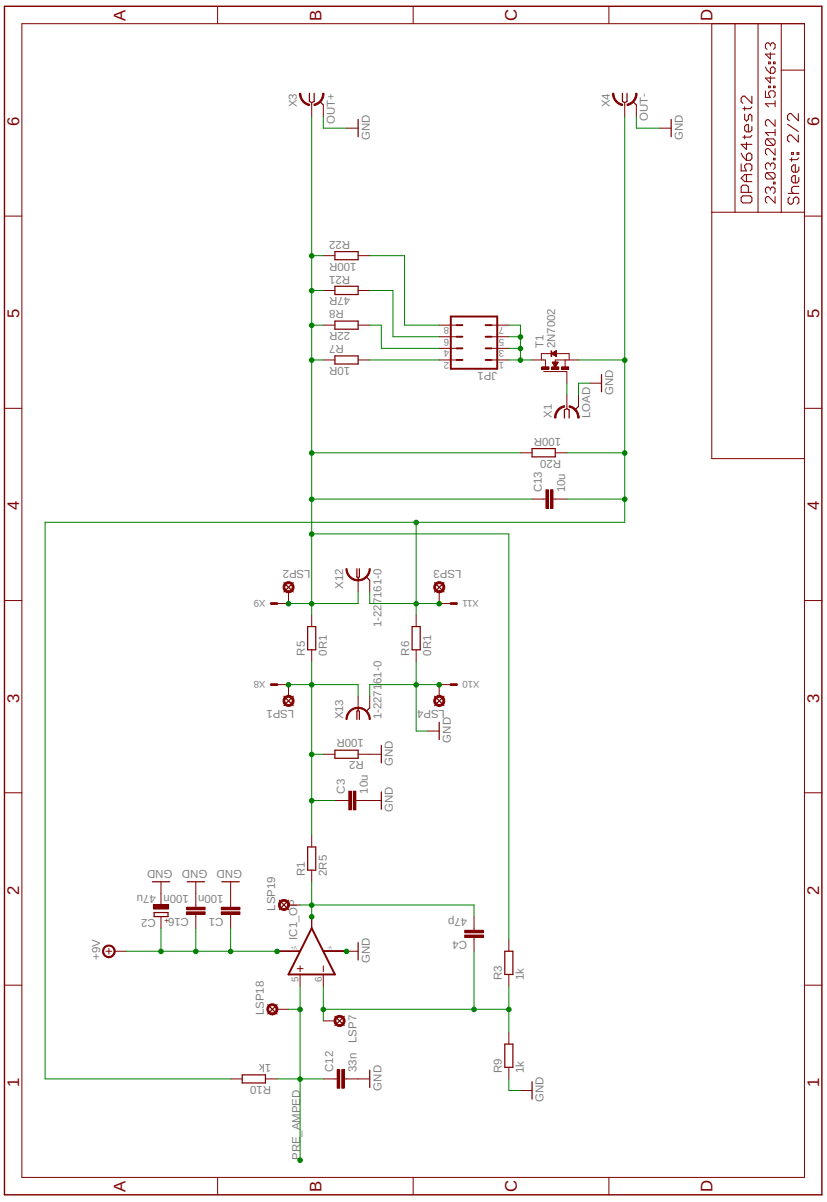


B.1.2 OPA547Test.sch

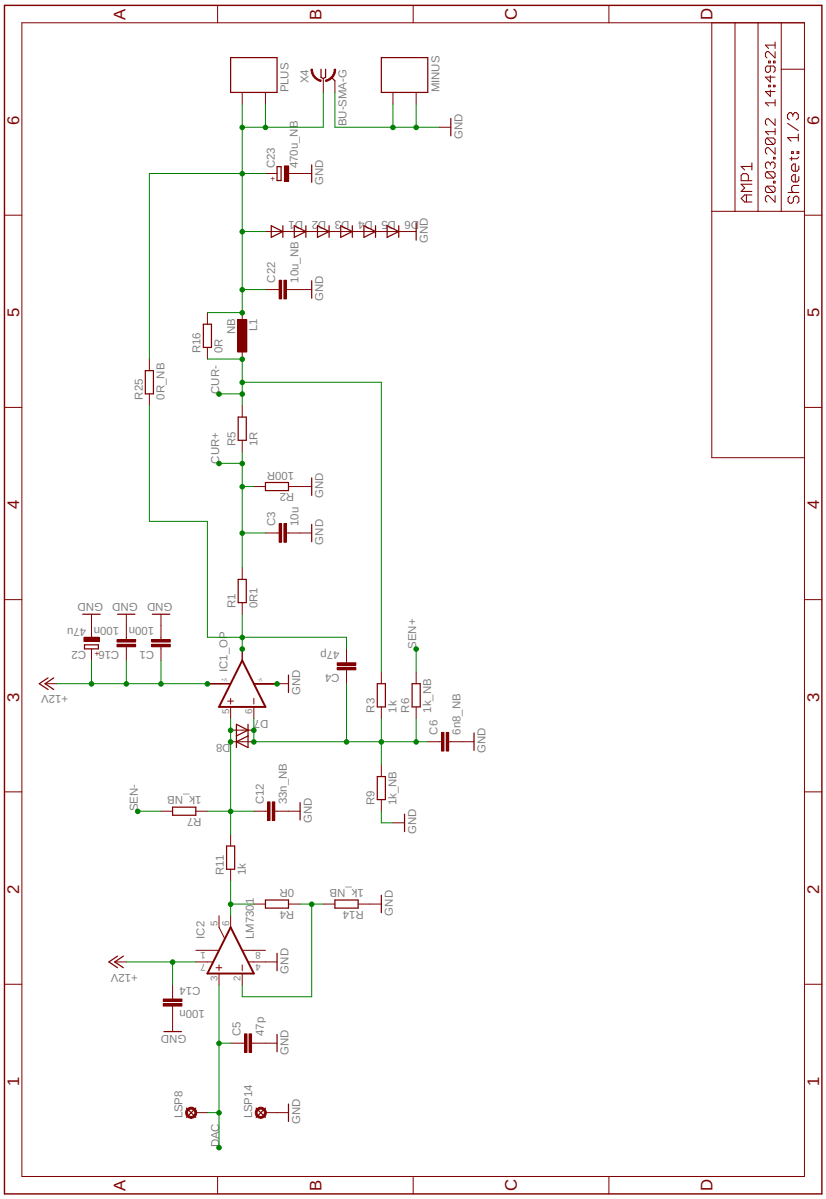


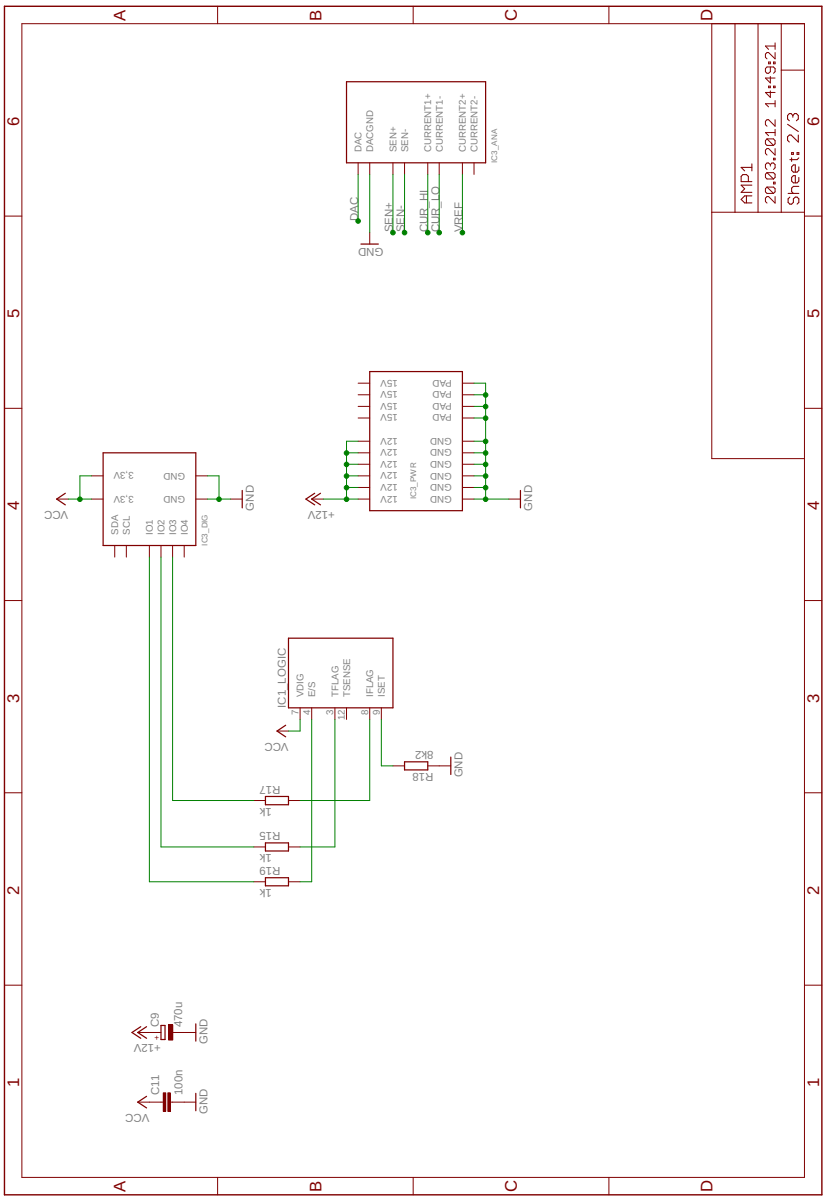
B.1.3 OPA564Test2.sch

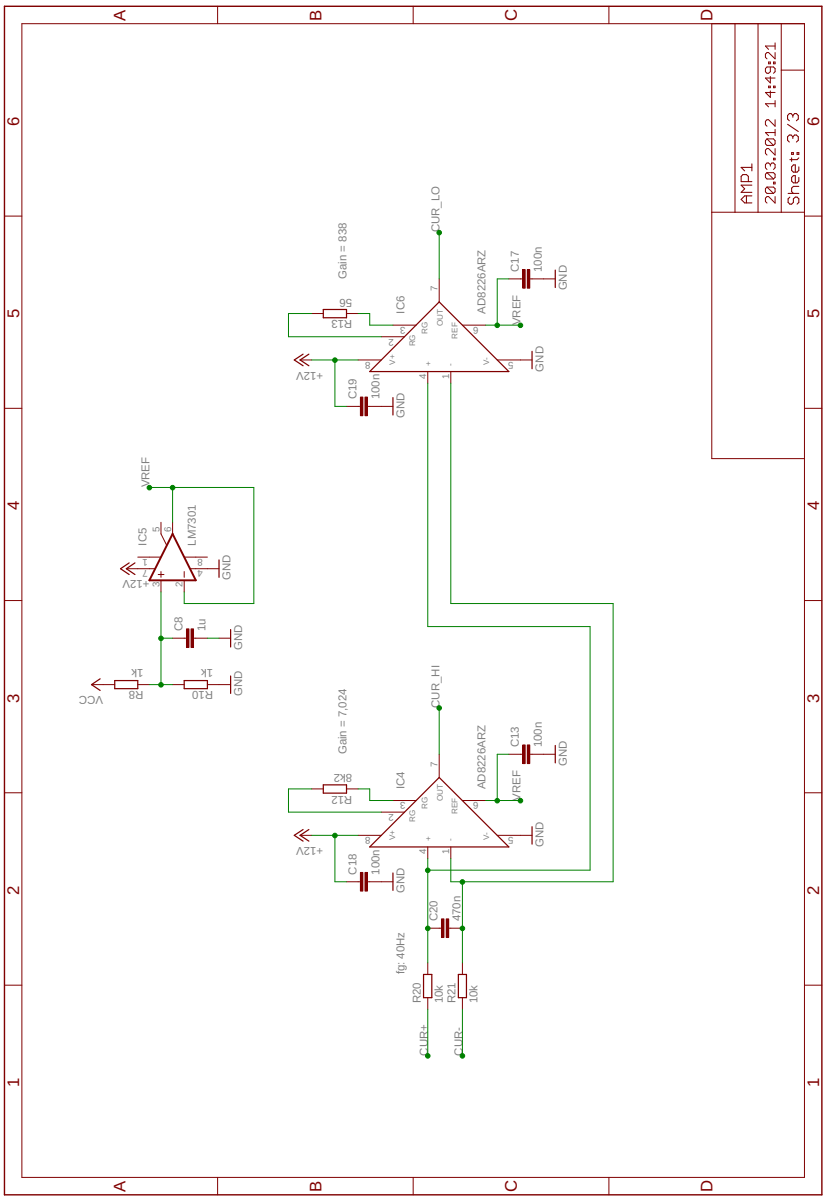



OPA564test2
23.03.2012 15:46:43
Sheet: 2/2
OUT+
OUT-
GND
LOAD
2N7002
JP1
PRE_AMPED
IC1_OP

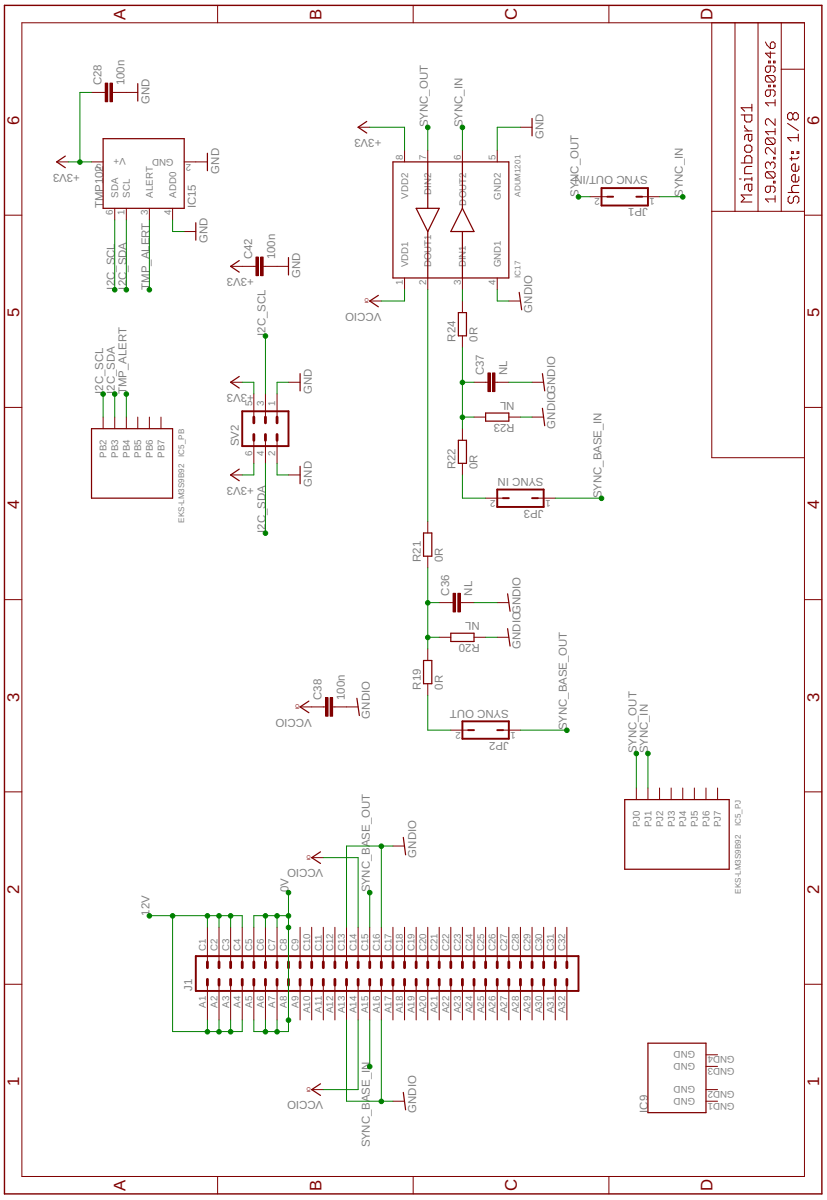
B.1.4 AMP1.sch

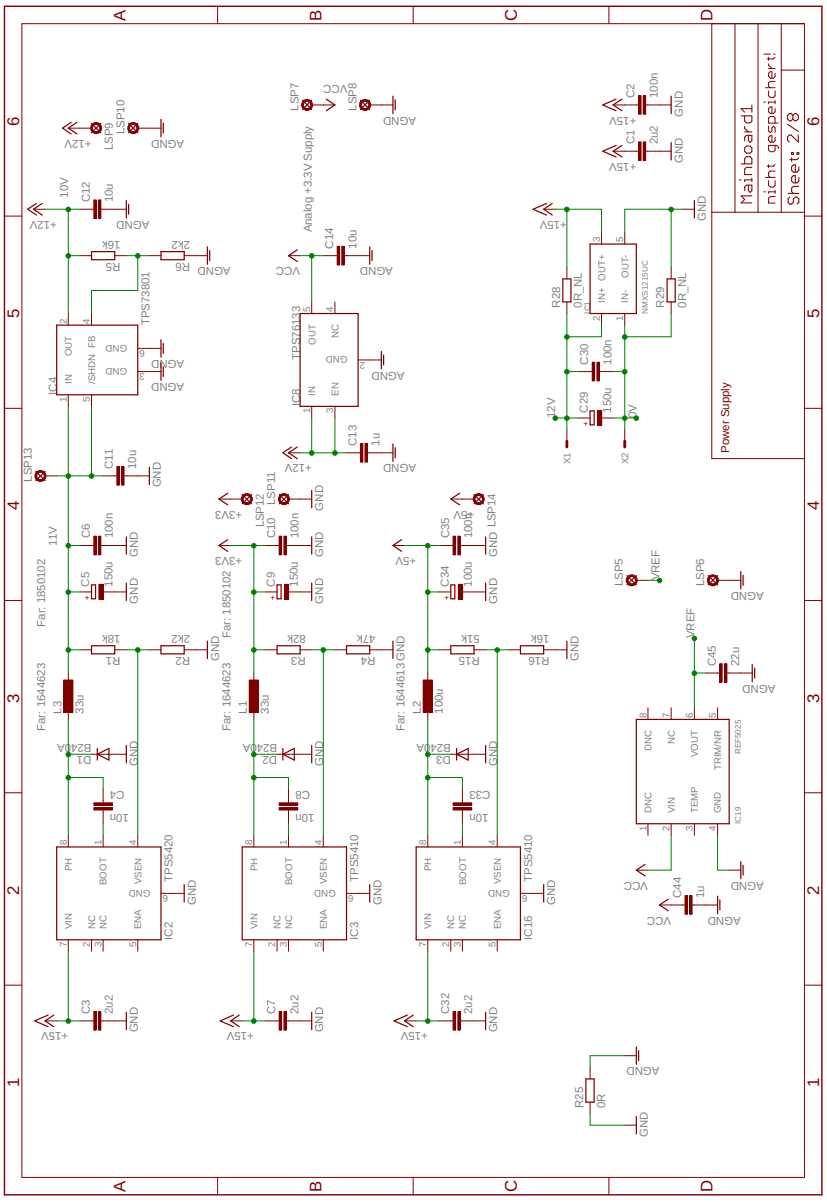

AMP1
20.03.2012 14:49:21
Sheet: 1/3
PLUS
MINUS
X4
BU-SMA-G
GND
C23
470u_NB
C22
10u_NB
L1
R16
0R
R25
0R_NB
CUR
CUR+
R5
1R
R2
100R
C3
10u
R1
0R1
IC1_OP
C4
47p
R3
1k
R6
1k_NB
C6
6n8_NB
SEN+
SEN-
R7
1k_NB
C12
33n_NB
R9
1k_NB
R11
1k
R24
0R
R14
1k_NB
IC2
LM7301
+12V
C14
100n
C5
47p
LSP8
LSP14
DAC
C2
47u
C16
100n
C1
100n

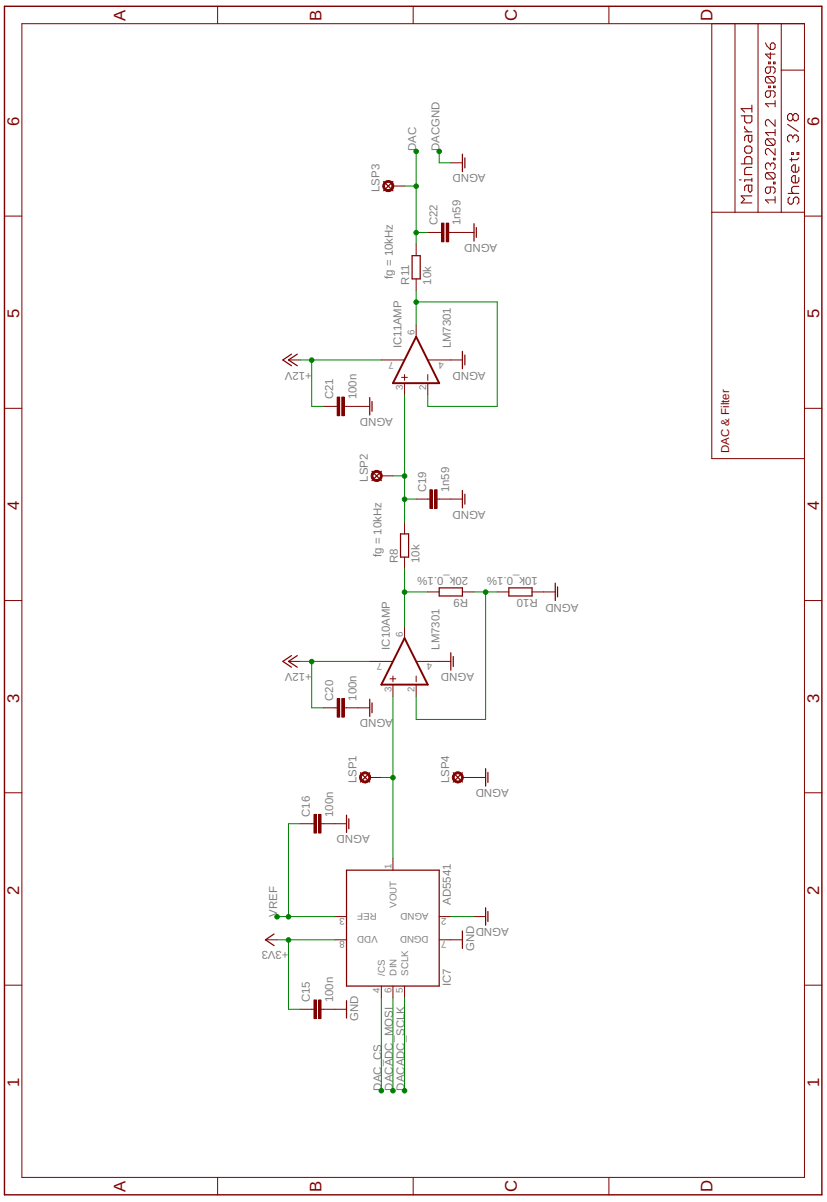

AMP1
20.03.2012 14:49:21
Sheet: 2/3
DAC
DACGND
SEN+
SEN-
CURRENT1+
CURRENT1-
CURRENT2+
CURRENT2-
IC3_ANA
DAC
SEN+
SEN-
CUR_HI
CUR_LO
VREF
GND
15V
12V
GND
PAD
IC3_PWR
+12V
3.3V
SDA
SCL
IO1
IO2
IO3
IO4
IC3_GPIO
VCC
IC1_LOGIC
VDIG
E/S
TFLAG
TSENSE
IFLAG
ISET
R17
R15
R19
R18
8K2
1K
C9
470u
C11
100n

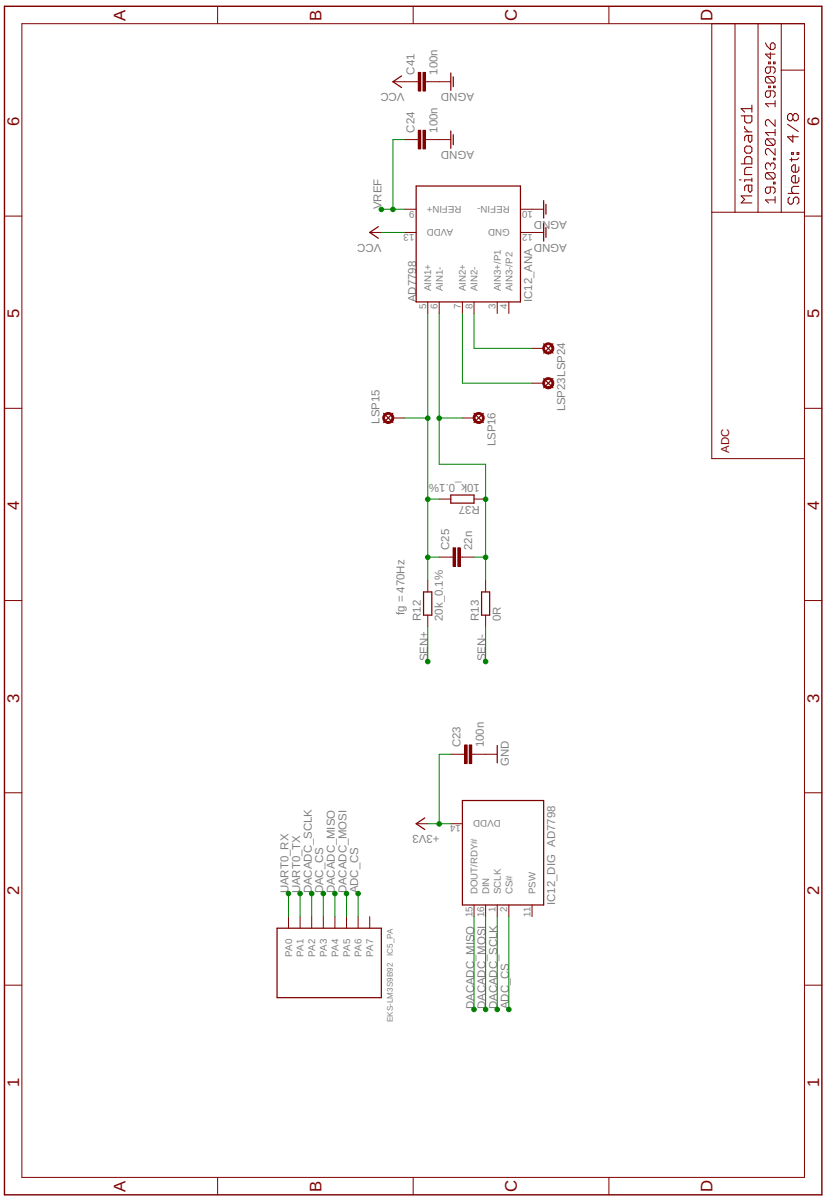

AMP1
20.03.2012 14:49:21
Sheet: 3/3
CUR_LO
CUR_HI
Gain = 838
Gain = 7,024
IC6
IC4
IC5
AD8226ARZ
LM7301
VREF
+12V
GND
VCC
C17
C13
C19
C18
C8
C20
R13
R12
R20
R21
R10
R8
fg: 40Hz
CUR+
CUR-

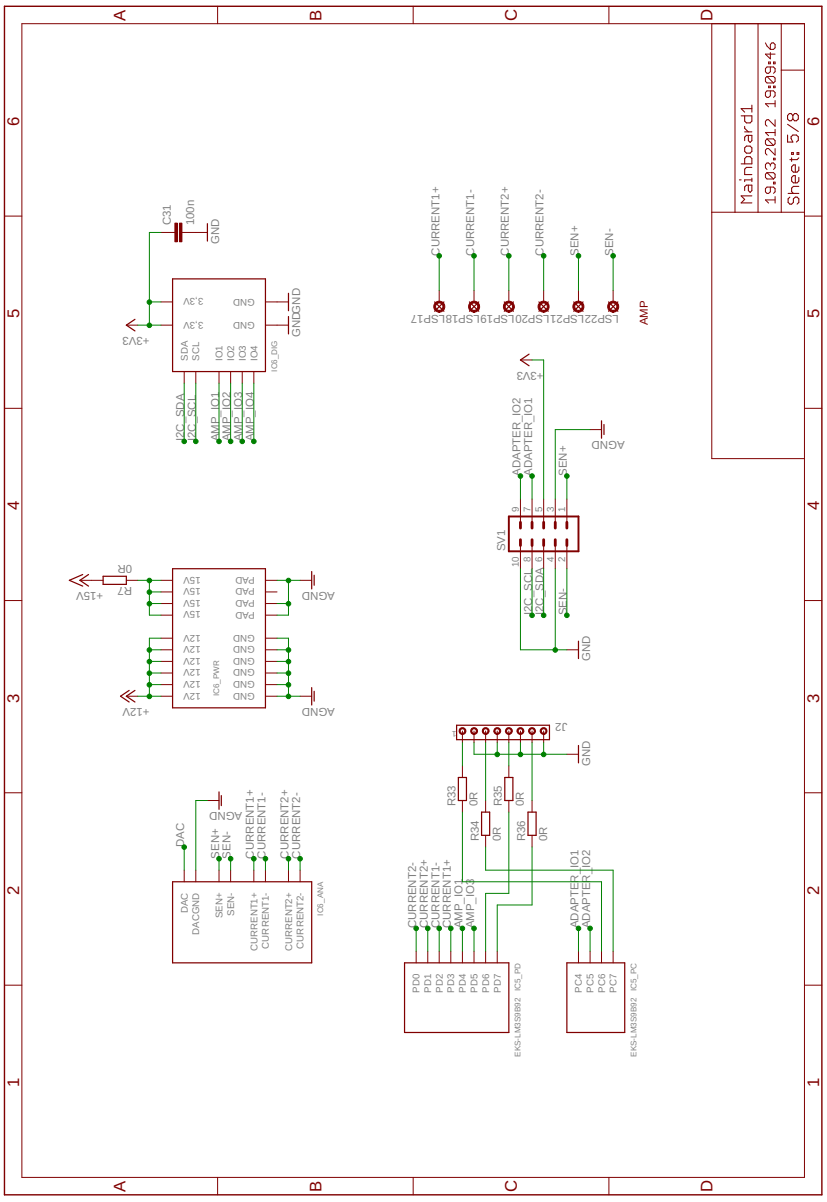
B.1.5 Mainboard1.sch

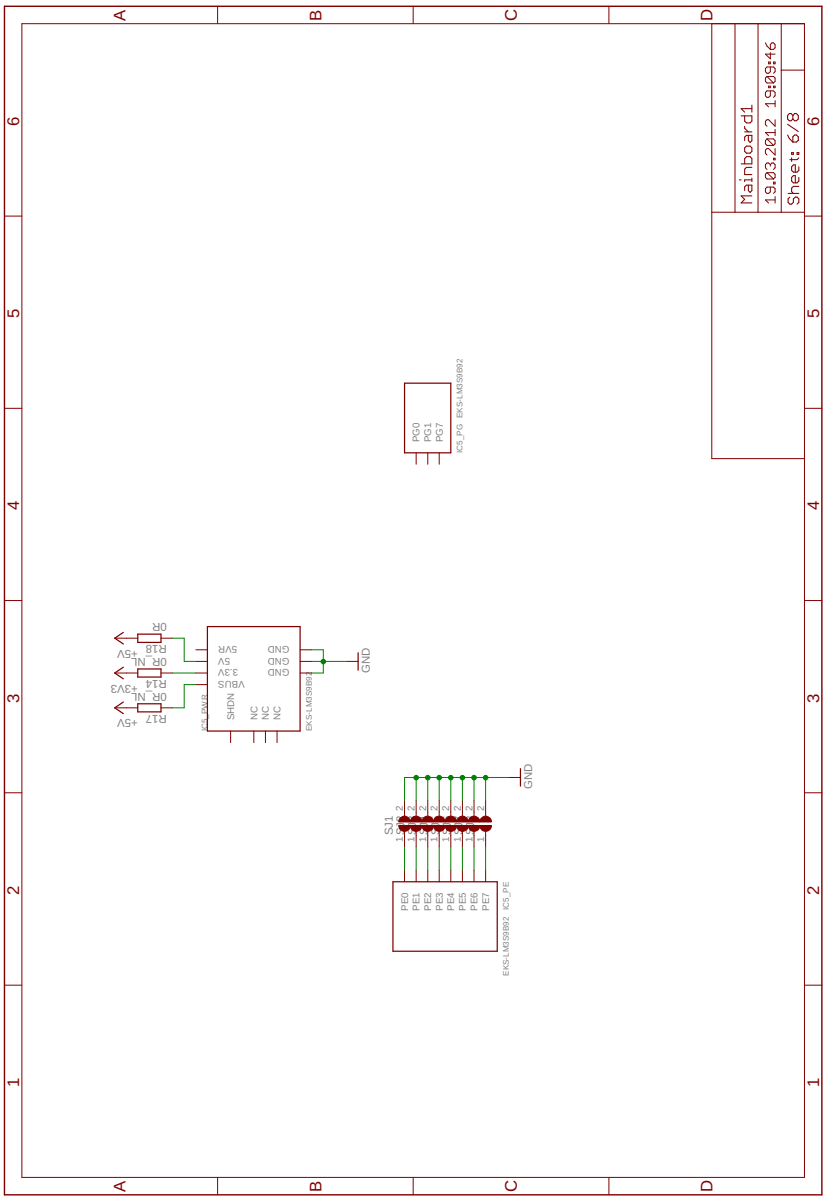


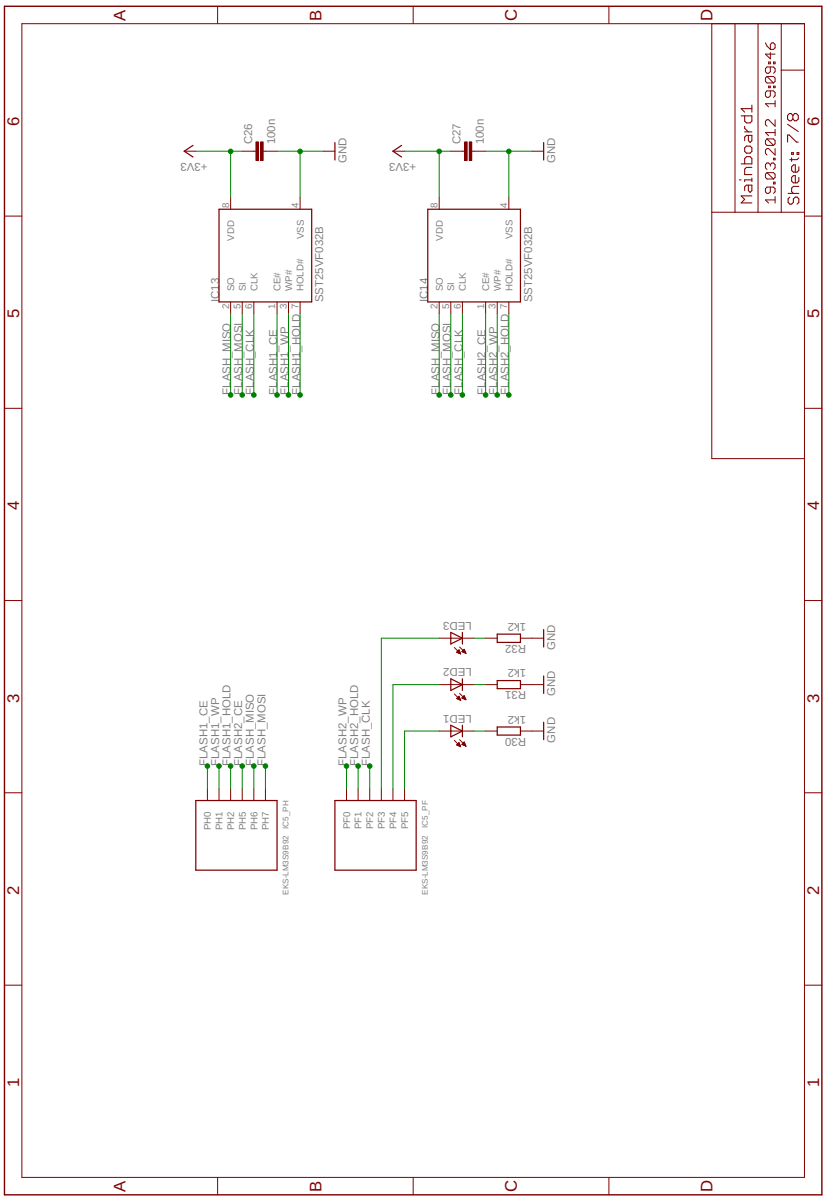

Mainboard1
nicht gespeichert!
Sheet: 2/8
Power Supply
Analog +3.3V Supply
TPS73601
TPS62113
TPS5420
TPS5410
NMXS1215UC
REF5025
Far: 1850102
Far: 1644623
Far: 1644613
B240A
+12V
+15V
+5V
+3V3
AGND
GND
VCC
VREF

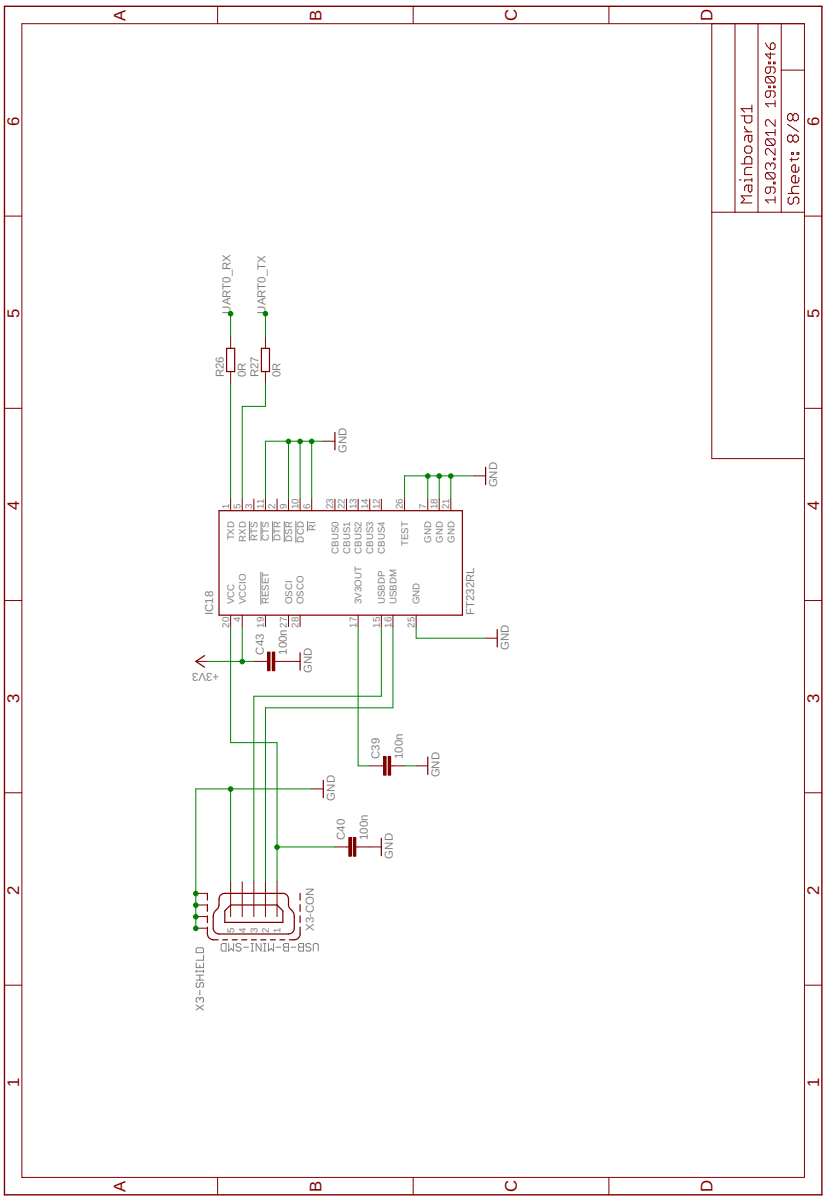

Mainboard1
19.03.2012 19:09:46
Sheet: 3/8
DAC & Filter
DAC
PACGND
AGND
LSP3
C22
1n59
fg = 10kHz
R11
10k
IC11AMP
LM7301
+12V
C21
100n
LSP2
C19
R8
R9
R10
20k_0.1%
10k_0.1%
IC10AMP
C20
LSP1
LSP4
C16
VREF
+3V3
C15
GND
IC7
AD5541
VOUT
REF
VDD
DGND
/CS
DIN
SCLK
DAC_CS
DACADC_MOSI
DACADC_SCLK


Mainboard1
19.03.2012 19:09:46
Sheet: 4/8
ADC
C41
100n
VCC
AGND
C24
100n
AGND
VREF
REFIN+
REFIN-
AVDD
GND
AD7798
AIN1+
AIN1-
AIN2+
AIN2-
AIN3+/P1
AIN3-/P2
IC12_ANA
LSP15
LSP16
LSP23
LSP24
R37
10K_0.1%
C25
22n
fg = 470Hz
R12
20k_0.1%
R13
0R
SEN+
SEN-
C23
100n
GND
+3V3
DVDD
DOUT/RDY#
DIN
SCLK
CS#
PSW
IC12_DIG : AD7798
UART0_RX
UART0_TX
DACADC_SCLK
DAC_CS
DACADC_MISO
DACADC_MOSI
ADC_CS
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
IC5_PA
EK-S-LM3S9B92

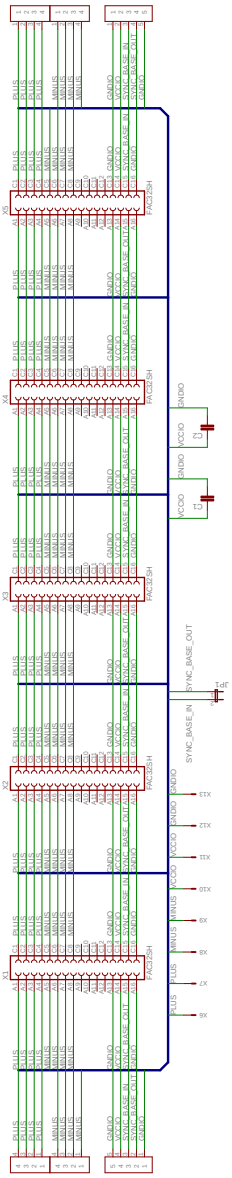

Mainboard1
19.03.2012 19:09:46
Sheet: 5/8
CURRENT1+
CURRENT1-
CURRENT2+
CURRENT2-
SEN+
SEN-
AMP
+3V3
GND
AGND
+15V
+12V
ADAPTER_IO2
ADAPTER_IO1
I2C_SCL
I2C_SDA
SV1
J2
R33
R34
R35
R36
0R
DAC
AMP_IO1
AMP_IO2
AMP_IO3
AMP_IO4


A
B
C
D
1
2
3
4
5
6
Mainboard1
19.03.2012 19:09:46
Sheet: 6/8
PG0
PG1
PG7
IC5_PG
EKS-LM3S9B92
0R
R18
+5V
0R_NL
R14
+3V3
0R_NL
R17
+5V
SHDN
VBUS
3.3V
5V
5VR
GND
GND
GND
NC
NC
NC
EKS-LM3S9B92
GND
S11
PE0
PE1
PE2
PE3
PE4
PE5
PE6
PE7
IC5_PE
EKS-LM3S9B92


Mainboard1
19.03.2012 19:09:46
Sheet: 7/8
C26
100n
C27
100n
+3V3
GND
IC13
IC14
SST25VF032B
FLASH_MISO
FLASH_MOSI
FLASH_CLK
FLASH1_CE
FLASH1_WP
FLASH1_HOLD
FLASH2_CE
FLASH2_WP
FLASH2_HOLD
LED1
LED2
LED3
R30
R31
R32
1K2
PH0
PH1
PH2
PH5
PH6
PH7
PF0
PF1
PF2
PF3
PF4
PF5

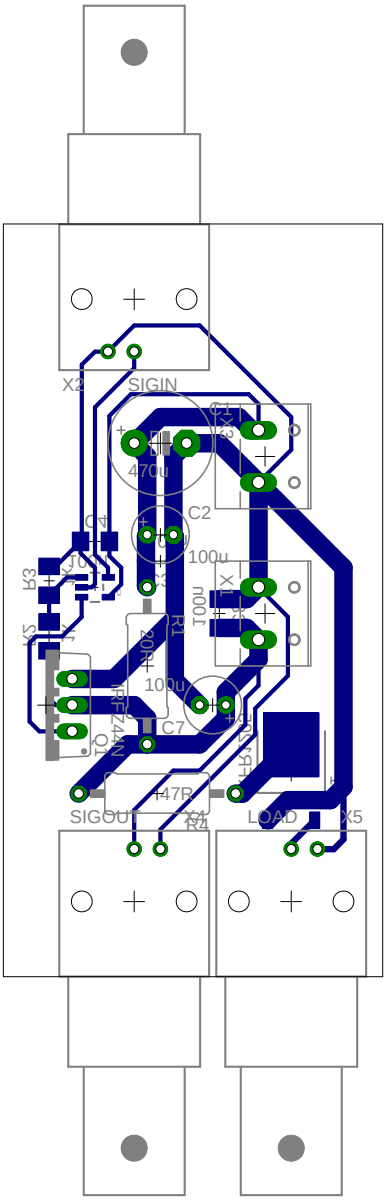

Mainboard1
19.03.2012 19:09:46
Sheet: 8/8
UART0_RX
UART0_TX
R26
0R
R27
0R
GND
IC18
FT232RL
TXD
RXD
RTS
CTS
DTR
DSR
DCD
RI
CBUS0
CBUS1
CBUS2
CBUS3
CBUS4
TEST
VCC
VCCIO
RESET
OSCI
OSCO
3V3OUT
USBDP
USBDM
C43
100n
+3V3
C39
100n
C40
100n
X3-CON
USB-B-MINI-SMD
X3-SHIELD

B.1.6 Backplane1.sch

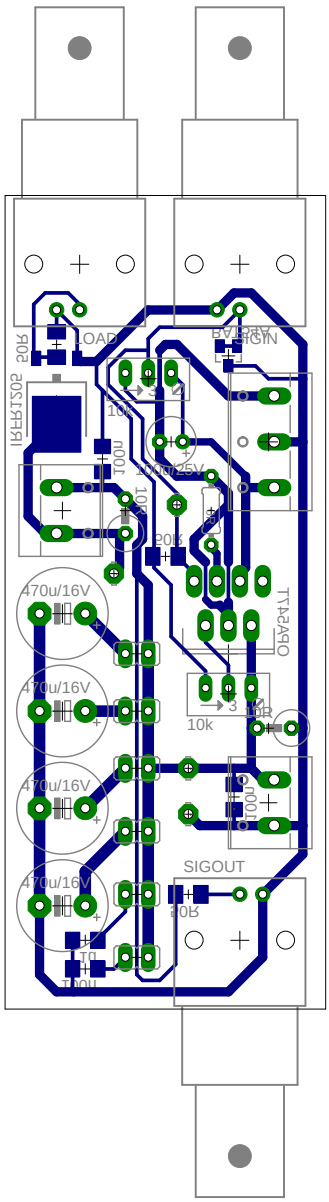


B.2 Layouts

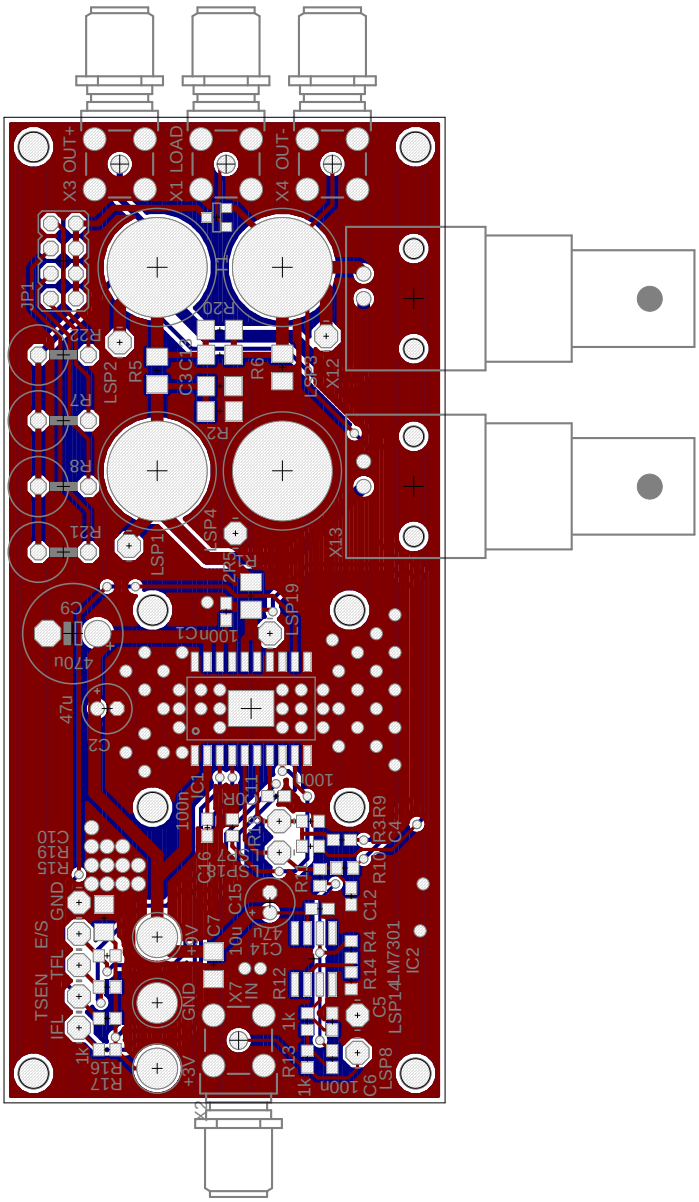
B.2.1 ParallelRegler.brd



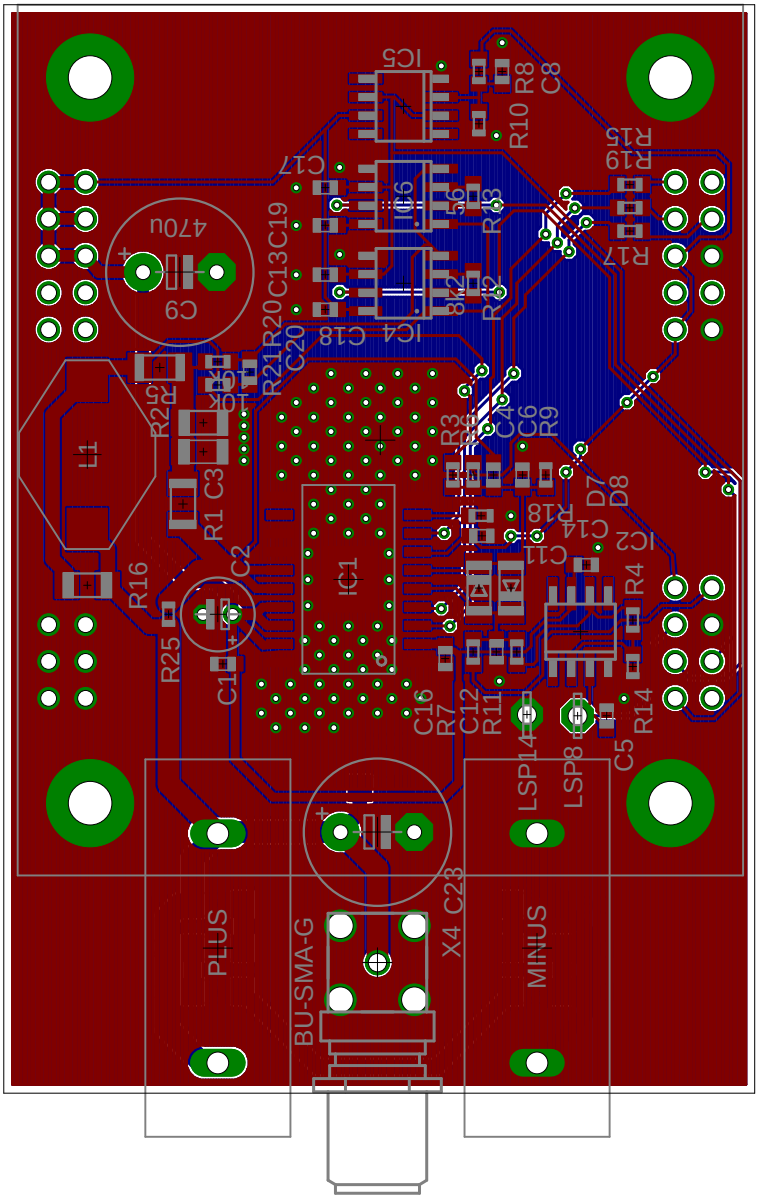
B.2.2 OPA547Test.brd



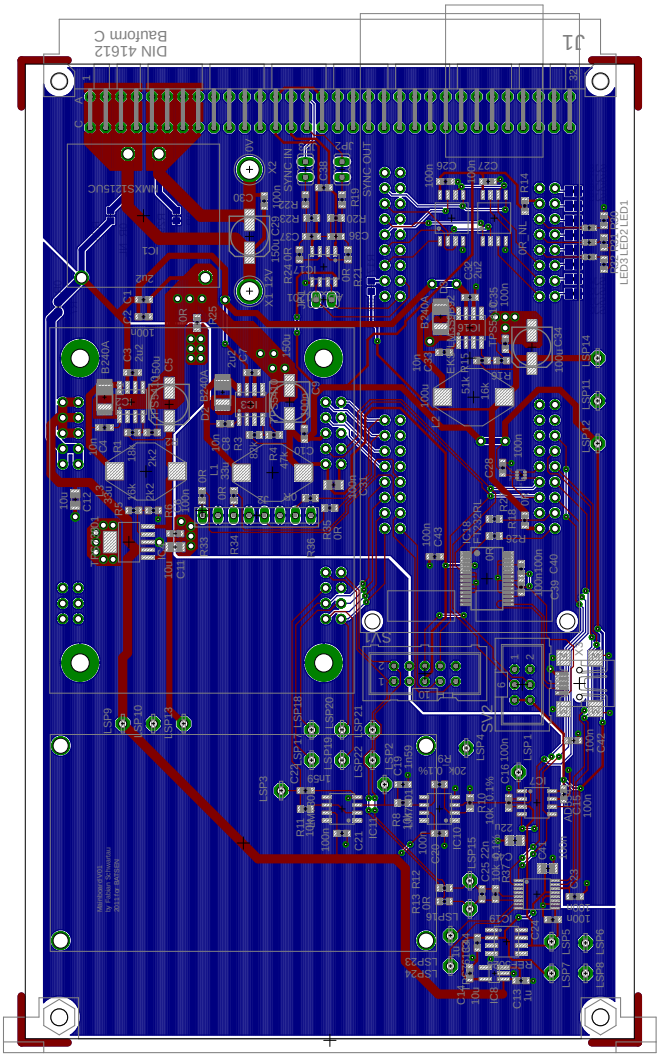
B.2.3 OPA564Test2.brd



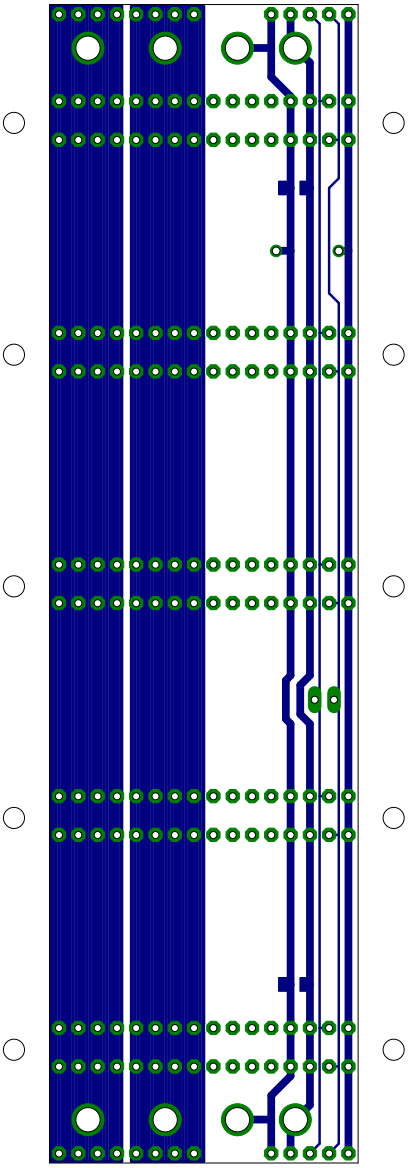
B.2.4 AMP1.brd


IC5
R8
C8
R10
R15
R19
C17
C19
C13
R20
C20
R21
R22
C18
IC4
R12
R13
R17
C9
470u
L1
C3
R1
C2
IC1
R16
R25
C1
C16
R7
C12
R11
C6
R9
R3
C14
C11
D7
D8
IC2
R4
R14
C5
LSP14
LSP8
PLUS
BU-SMA-G
X4
C23
MINUS

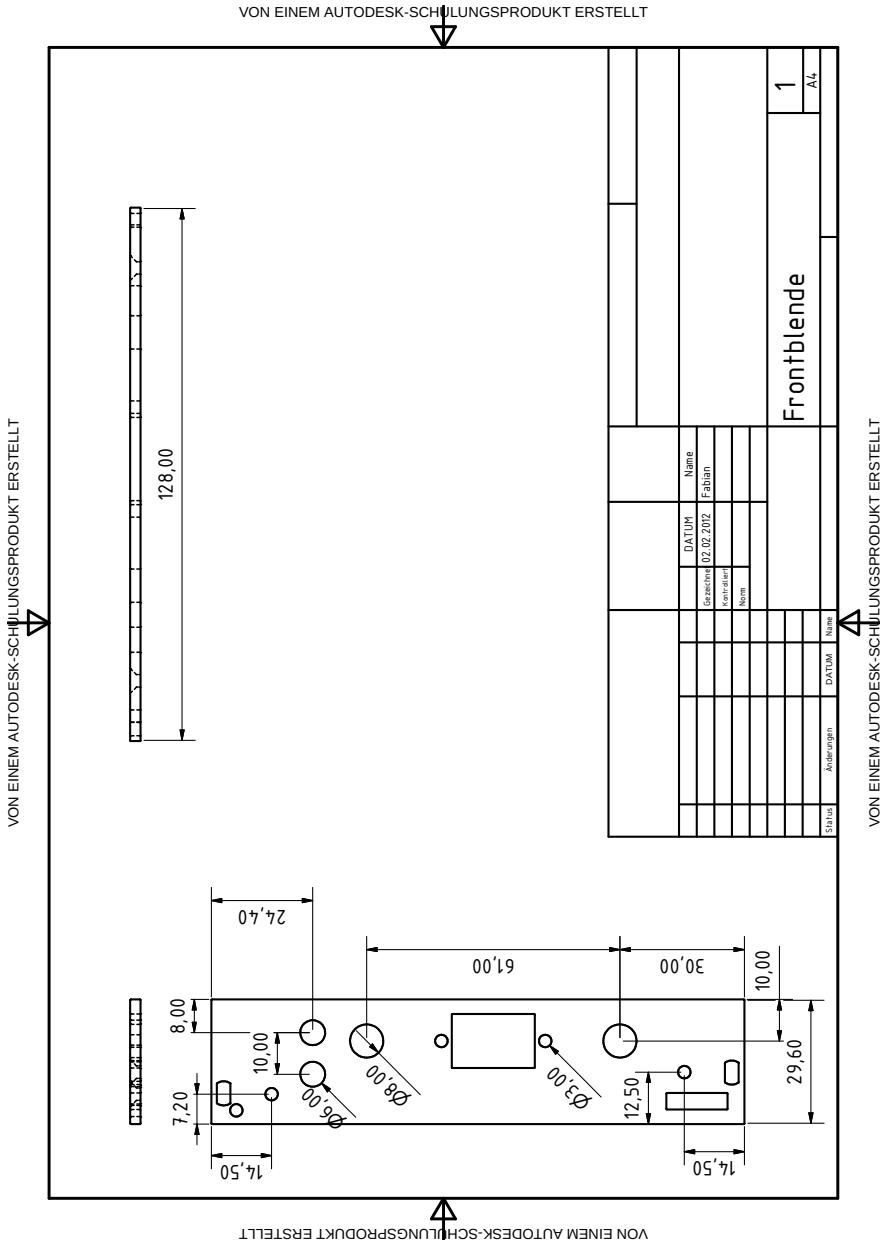
B.2.5 Mainboard1.brd



B.2.6 Backplane1.brd



B.3 Frontblende



C Alternative Methode zur Bandbreitenbestimmung

In Kapitel 2.3 ist die benötigte Bandbreite für den Generator abgeschätzt worden. Hierfür ist das Signal mit einem Fenster multipliziert worden, mit einer FFT in den Frequenzbereich gebracht, dort gefiltert und mit einer inversen FFT wieder zurück in den Zeitbereich transformiert worden. Die Fensterung hat den Nachteil, dass zu den Seiten hin die Differenz des Originalsignals und zu Null abklingt, genau wie das Fenster. Es ist jedoch nötig dafür zu sorgen, dass durch die periodische Wiederholung des Signals keine Sprünge entstehen, da dies zu unerwünschten Effekten führt.

Eine andere Möglichkeit den Sprung zu vermeiden, ohne das Signal fenstern zu müssen, ist die Fortsetzung des Signals durch seine gespiegelte Version. So kann der Sprung verhindert werden, es ist jedoch nicht genau klar, inwieweit dadurch das Ergebnis beeinflusst wird. Auf Grund dieser intuitiven Lösung und deren guten Resultaten, sollen die Ergebnisse hier präsentiert werden, wobei die Auswirkungen dieses Verfahrens mathematisch nicht bewiesen sind.

Das Verfahren bleibt ansonsten dem aus Kapitel 2.3 gleich. In Abbildung C.1 ist das gespiegelte Signal und dessen Spektrum zu sehen. Im Amplitudenspektrum lassen sich auf den ersten Blick keine nennenswerten Unterschiede zu dem Spektrum der gefensterten Zeitfunktion aus Abbildung 2.8 erkennen. Anschließend wird das Signal mit $2{,}5\,\mathrm{kHz}$ bandbegrenzt und in den Zeitbereich zurück transformiert. Dies ist in Abbildung C.2 zu sehen. Auch hier ist im Zeitbereich wieder das gespiegelte Signal zu sehen. Da nun der gespiegelte rechte Teil des Signals nicht mehr von Interesse ist wird dieser abgeschnitten und mit dem linken Teil der Vergleich mit dem unveränderten Signal durchgeführt. Abbildung C.3 zeigt die Differenz der beiden Signale, in Abbildung C.4 nochmal etwas detaillierter. Hier ist zu der Differenz mit der Fensterung ebenfalls kein nennenswerter Unterschied feststellbar, mit der Ausnahme, dass die Differenz nicht wie in Abbildung 2.10 zu den Seiten hin immer geringer wird. Es kann also gesagt werden, dass mit der Methode des Spiegelns ähnliche Ergebnisse erzielt werden, wie mit der Fensterung. Die Spiegelung hat jedoch den großen Vorteil, dass es keine sichtbaren Randeffekte gibt.

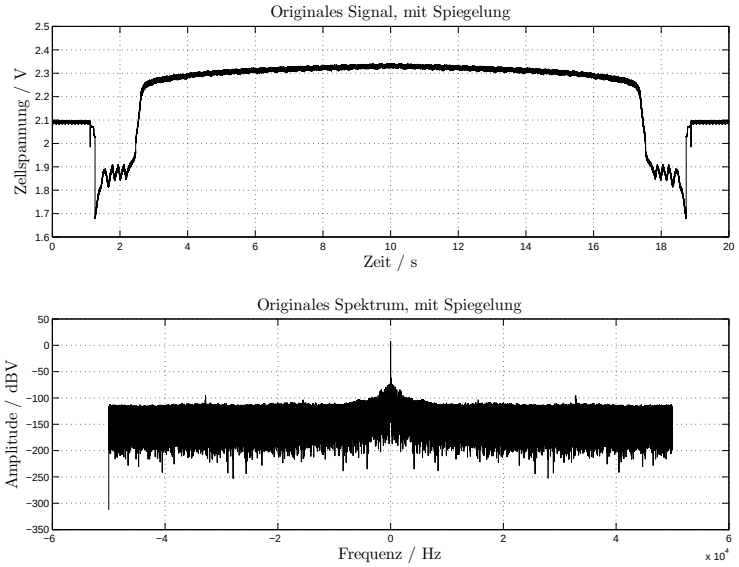


Abbildung C.1: Oben: Gefenstertes Zeitsignal des Startvorgangs eines PKW, Unten: Spektrum des gespiegelten Signals

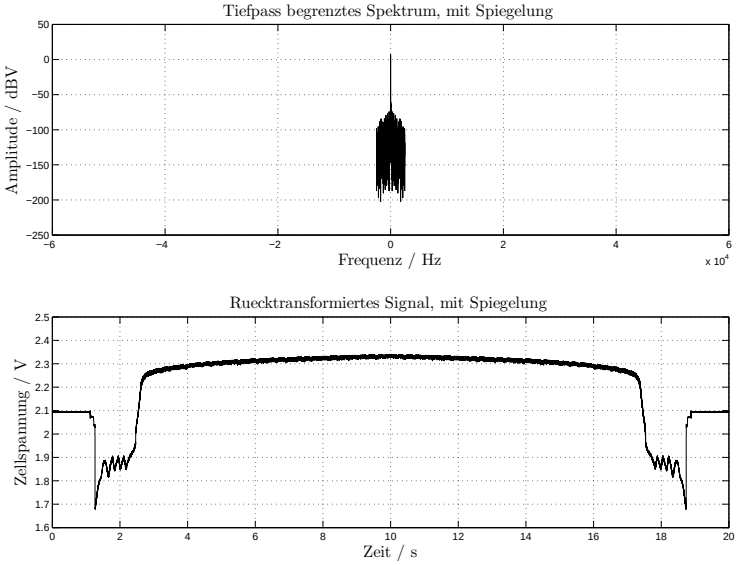


Abbildung C.2: Oben: Bandbegrenztes Spektrum des gespiegelten Startvorganges, Unten: Rücktransformiertes Signals nach der Bandbegrenzung im Spektrum, $f_g = 2{,}5\,\mathrm{kHz}$

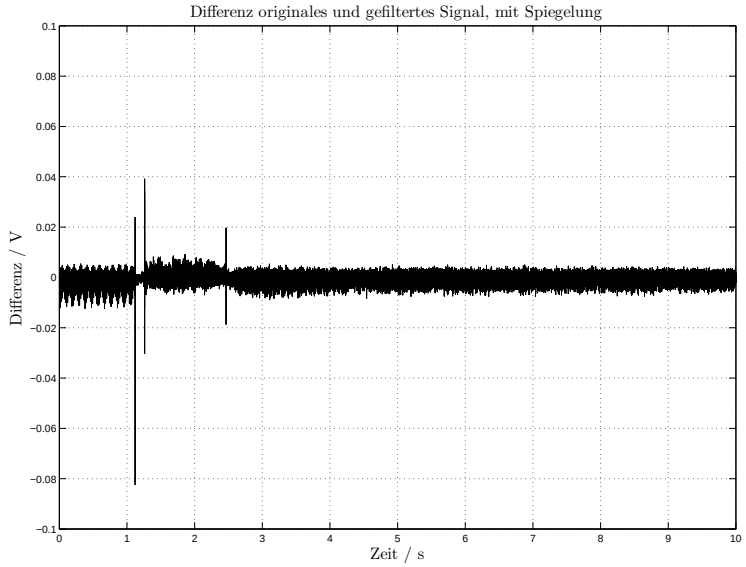


Abbildung C.3: Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$, Verwendung der Spiegelung im Zeitbereich

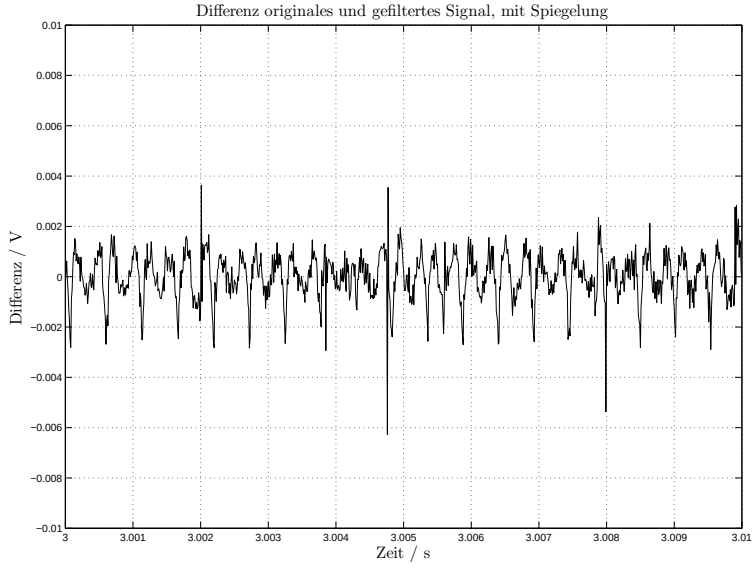


Abbildung C.4: Kleiner Ausschnitt der Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$, Verwendung der Spiegelung im Zeitbereich

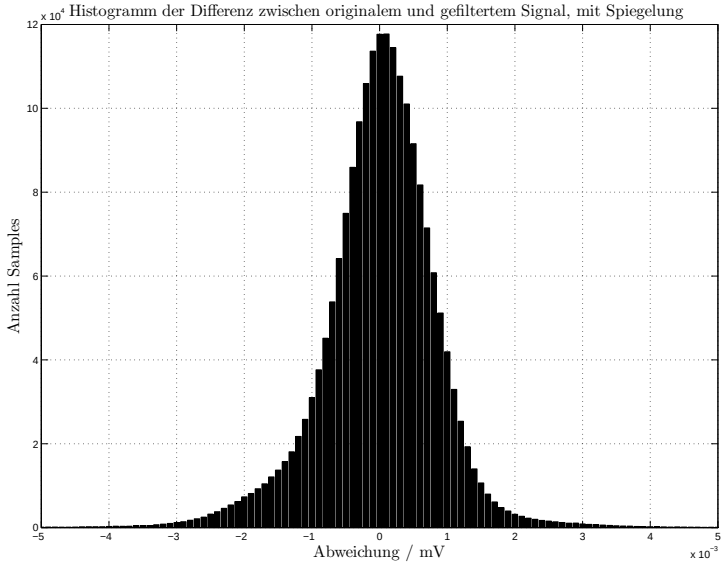


Abbildung C.5: Histogramm der Differenz der originalen und der tiefpassbegrenzten Zellspannung im Zeitbereich eines PKW Startvorgangs, $f_g = 2{,}5\,\mathrm{kHz}$, Verwendung der Spiegelung im Zeitbereich

Abkürzungsverzeichnis

ADC . . . . . Analog-Digital-Umsetzer

ARP . . . . . Address Resolution Protocol

DAC . . . . . Digital-Analog-Umsetzer

DHCP . . . . . Dynamic Host Configuration Protocol

DNL . . . . . Differentieller Nicht Linearitätsfehler

FFT . . . . . Fast Fourier Transformation

FIFO . . . . . First In First Out, Ringspeicher

INL . . . . . Integraler Nicht Linearitätsfehler

IP . . . . . Internet Protocol

TCP . . . . . Transmission Control Protocol

UDP . . . . . User Datagram Protocol

Tabellenverzeichnis

Abbildungsverzeichnis

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, 29. März 2012	
Ort, Datum	Unterschrift