Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Gregor Manzke

Überwachungscontroller für einen Hochleistungs-Batterieprüfstand

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Gregor Manzke

Überwachungscontroller für einen Hochleistungs-Batterieprüfstand

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. Wilfried Wöhlke

Abgegeben am 16. November 2015

Gregor Manzke

Thema der Bachelorthesis

Überwachungscontroller für einen Hochleistungs-Batterieprüfstand

Stichworte

Steuerung, Überwachung, Sicherheit, Batterie-Zykliersystem, Temperaturregulierung, Hardwareaufbau

Kurzzusammenfassung

Im BATSEN Forschungsprojekt wurde ein Überwachungssystems für einen Hochleistungs-Batterieprüfstand entwickelt. Es wurde ein Konzept und das Design erstellt, mit dem ein Hardwareaufbau eines Batterieprüfstandes mit integriertem Schutzsystem realisiert wurde. Ebenfalls wurde eine Software für den Überwachungs-Mikrocontroller entwickelt, welche die Eigenschaften des Systems überwacht.

Gregor Manzke

Title of the paper

Monitoring-controller for a high-performance battery-tester

Keywords

control, monitoring, safety, battery-cyclingsystem, temperature-regulation, hardware-construction

Abstract

In the BATSEN research project a safety system for a high-performance battery-tester was developed. A concept, a design and the hardware for a battery testsystem with an integrated safety system was realized. Also, the software for the monitoring microcontroller was developed that monitors the characteristics of the system.

Danksagung

Als erstes möchte ich meinen Eltern Uwe & Nina Manzke danken, die mir durch Ihre seelische und finanzielle Unterstützung dieses Studium ermöglicht haben.

Meiner Freundin Catharina Schmidt möchte ich ebenfalls für Ihre seelische Unterstützung danken, denn sie ist auch in “Notsituationen“ zu konfusen Uhrzeiten immer für mich da gewesen.

Herrn Prof. Karl-Ragmar Riemschneider, für die Möglichkeit diese Bachelorarbeit in dem Projekt BATSEN schreiben zu dürfen.

Herrn Dipl. Ing. Günther Müller, der besonders mit seiner Korrekturlesung von Bachelorarbeiten nicht nur mir, sondern schon so vielen Kommilitonen im BATSEN Projekt eine große Hilfe war.

Ebenfalls möchte ich allen Mitarbeitern aus dem BATSEN Projekt meinen Dank aussprechen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einführung

1.1. Motivation

In dem BATSEN Projekt der HAW wird im Bereich: “Wirtschaftlichkeit, betriebliche Verfügbarkeit und Sicherheit der Batterie“ sowie “drahtlose Sensornetze zur Überwachung jeder Zelle einer Fahrzeugbatterie“ geforscht. Man kann dies als Anwendungsforschung von Fahrzeugbatterien allgemein zusammenfassen. Für diese Anwendungsforschung bedarf es spezieller Laborgeräte, mit denen man Batterien zum Beispiel Laden-, Entladen, Zyklieren oder deren optische bzw. physikalische Effekte sichtbar/messbar machen kann. Hier wurden bereits in einigen Vorarbeiten Zykliermaschinen erstellt, welche als Laborwerkzeuge für ein planmäßiges Laden- und Entladen verwendet werden können.

Im Zuge der jüngsten Entwicklungen im Bereich Elektro-Mobilität, wird die Batterie als zentrales Energiespeicherelement immer wichtiger. Mit den Forschungsergebnissen im Bereich Batterieanwendung und Verwendung lässt sich eine verbesserte Nutzung der Speicherelemente umsetzen. Die für die verschiedenen Batterietypen (Li-Po, Li-Fe) erzeugten Analysen, lassen in der praktischen Anwendung eine erhöhte Lebensdauer der Batterien oder am Beispiel eines Elektrofahrzeugs eine verlängerte Reichweite in Kilometern zu. Deshalb ist eine technische Anwendung der Forschungsergebnisse in diesem Bereich für die E-Mobilität unerlässlich. Durch die Berücksichtigung dieser Forschungsergebnisse lassen sich Wirtschaftlichkeit und Ökobilanzen von Elektromobilitätsprojekten deutlich verbessern. Aus diesem Grund sind die Ergebnisse des BATSEN Projektes auch für die Industrie wertvoll.

Die Motivation aus der diese Bachelorarbeit entstanden ist, kommt von den ständig steigenden Anforderungen an die Labormessgeräte für diese beschriebenen Anwendungen. Mit einem noch präziseren Laborgerät lassen sich noch bessere Analysen und Ergebnisse erzielen. Außerdem sollte es technisch so abgesichert sein, dass der Anwender sich auf die Anwendung des Gerätes konzentrieren kann, ohne zu tief in die technischen Feinheiten der verwendeten Laborgeräte einzutauchen. In dieser Bachelorarbeit soll daher ein Sicherheitskonzept erarbeitet werden, mit dem man eine zukünftige Neuauflage eines Zykliersystems absichert. Durch dieses Sicherheitskonzept kann sich der Anwender voll und ganz auf den Ablauf seiner Messungen konzentrieren, ohne sich Gedanken über die technischen Eigenschaften des dafür verwendeten Labormessgerätes zu machen. Die Einarbeitung in die bereits vorhandenen Zykliersysteme sind durch die Verwendung von den externen Quellen und

Senken relativ zeitaufwendig. Ein neues System, in dem alles intern vorhanden ist, dass zudem keine gravierenden Fehler durch den Anwender zulässt, würde daher den Fokus auf die Messungen selbst setzten. Dies wäre ein deutlicher Mehrwert für die weiterführenden Arbeiten im BATSEN Projekt.

1.2. Zykliersystem aus Vorarbeit

Das im Labor bereits verwendete Zykliersystem besteht aus einer Einheit, in dem ein Mikrocontroller den Lade- Entladevorgang einer extern angeschlossenen Batterie steuert/regelt. In dem Zykliersystem wird der Lade/Entladestrom über mehrere Relais gesteuert. Der Strom für den Ladevorgang wird von einem externen Netzteil erzeugt und über die internen Relais gesteuert. Der Entladestrom wird ebenfalls über die internen Relais gesteuert, auch hier wird dieser Strom über eine externe Last in Wärme umgewandelt. Diese externe Last ist ein Laborgerät, dass über die internen Widerstände die zugeführte Leistung verbraucht. Die Last verfügt über Lüfter zur Kühlung der Widerstände, da diese im Betrieb sehr heiß werden können. Das Zykliersystem hat durch seine Bauart nur eine begrenzte Anzahl von Schaltstufen, auch die Zusammenschaltung der einzelnen Komponenten ist etwas aufwendig und erfordert einiges an Verkabelung. Hier können dem Benutzer auch beim Messaufbau diverse Fehler unterlaufen.

1.3. Die Idee des neuen Zykliersystems

Das neue Zykliersystem soll über eine MOS-FET Schaltung eine deutlich genauere (stufenlose) Stromsteuerung ermöglichen. Außerdem soll der Vorgang des Entladens direkt im Zykliersystem erfolgen. Dies bedeutet, dass die Lastwiderstände direkt im neuen System verbaut werden sollen.

1.4. Autarkes Schutzsystem für das neue Zykliersystem

Die Aufgabe dieser Bachelorarbeit war, parallel zu der Entwicklung des neuen Zykliersystems der BATSEN Arbeitsgruppe, ein mit diesem Zykliersystem verknüpftes Schutzsystem zu entwickeln. Das Schutzsystem soll die Eigenschaften des Gesamtsystems (Schutzsystem & Zykliersystem) überwachen und absichern. Ein Schutzsystem ist deshalb notwendig, da hier bis zu 120 Ampere durch das Gerät in eine Batterie fließen sollen oder von einer Batterie 120 Ampere in dem Gerät in Wärme umgesetzt werden soll. Hier wird eine Leistung von

ca. 1KW umgesetzt. Der Umgang in diesem Leistungsbereich/Strombereich kann gefährlich sein, deshalb sollte hier ein Schutzsystem Abhilfe schaffen.

1.5. Kommerzielle Systeme

Es gibt Batterieprüfstände von verschiedenen Herstellern. Die Laborgeräte haben ebenfalls Schutzfunktionen und diverse Analysefunktionen integriert. Die Eigenschaften und Preise variieren je nach Hersteller. Hier gibt es alles von kleineren fertigen Laborgeräten bis hin zu kompletten Prüfstands auf Bestellung oder einer gesamten Prüfstraße, die in der Fabrik des Kunden aufgebaut wird.

Hersteller sind beispielsweise:

BaSyTec - www.basytec.de - Stand (07/2015)
BaSyTec bietet verschiedene Komplettsysteme zum Testen von Batterien an. Diese reichen von kleinen Systemen im mA, bis hin zu großen Testschränken im kA Bereich. Diese sind untereinander werksseitig kompatibel, so dass mit Einsatz eines Client-Server-Datenbankmoduls eine vernetzte Analyseumgebung erreicht werden kann.

Arbin - www.arbin.com - Stand (07/2015)
Arbin bietet für die Batterieanalyse die maßgeschneiderte Lösung BT-2000 an, diese wird nach Kundenanforderungen gebaut. Es werden außerdem weitere fertige Produktserien im Strombereich von BT-2043 (-10V|10V / 100mA) bis EVTS (700V / 2000A) angeboten.

Heiden Power - www.heidenpower.com - Stand (07/2015)
Heiden Power hat eine breite Produktpalette von DC-Quellen, DC-Lasten, DC-Quelle-Senke / Batteriesimulation und Ladetechnik, mit denen eine selbst zusammengestellte Batterie-Testumgebung aufgebaut werden kann. Außerdem bietet Heiden Power wie auch die Konkurrenten, kundenspezifische Systeme für die individuellen Anforderungen an.

FuelCon - www.fuelcon.com - Stand (07/2015)
FuelCon bietet neben kleinen Analysegeräten und Testzubehör ebenfalls große Testsysteme für Batteriezellen an. Hier werden der Evaluator-B30 (0-5V / 8 Kanäle mit max 320A) oder Evaluator-B100 (1000V / 1000A) angeboten. Auch FuelCon bietet Kundenspezifische Systeme für die individuellen Anforderungen an, welche hier “schlüsselfertige Testfelder“ genannt werden. Diese werden als große Testfelder in Räumen oder Hallen installiert.

Thyssen Krupp - www.thyssenkrupp-system-engineering.com - Stand (07/2015)
Thyssen Krupp bietet nur große Batterie-Testsysteme an, welche als Serienprüfstände in die Fabrikhallen oder Räume der Kunden installiert werden. In dieser Größenordnung gibt es ausschließlich kundenspezifische Lösungen.

Cadex - www.cadex.com - Stand (07/2015)
Cadex bietet nur ein einziges Batterie-Testsystem (C8000) an. Das C8000 beinhaltet eine Stromversorgung sowie eine Lastbank für den Ladevorgang (4 Kanäle mit je 10A Laden / 400W Leistung beim Entladen). Es lässt sich als Einzelsystem für die Batterieanalyse betreiben, hat zusätzlich aber die Möglichkeit, externe Geräte mit einzubinden und zu steuern. Es lassen sich folgende Zusatzgeräte mit dem C8000 ansteuern/verwenden: Wärmekammer, externes Ladegerät, digitale Lastbank, Heizelement, Druckmessgerät, Batteriezellen-Überwachung, Sicherheitsschaltung.

Maccor - www.maccor.com - Stand (07/2015)
Maccor bietet fertige Laborgeräte von Tisch- bis Schrankgröße an, die vom mA Bereich bis hin zu 2000A eine breite Palette an Testmöglichkeiten abdecken. Die Laborgeräte sind ein in sich geschlossenes System und bieten laut Hersteller alle Möglichkeiten zur automatisierten Laboranalyse von Batterien.

Chroma - www.chromaate.com - Stand (07/2015)
Chroma bietet mit der Modellserie 17011 ein großes modulares Komplettsystem an. In einem 19 Zoll Rack-System können die Komponenten vom Kunden zusammengestellt oder bei Bedarf auch nachgerüstet werden. Es gibt Quellen- und Senken-Module. Die Quellen-Module gibt es in drei Varianten, (5V / 20A), (5V / 30A), (5V / 100A).

NH Research - www.nhresearch.com - Stand (07/2015)
NH Research bietet 3 verschiedene große Komplett-Testsysteme mit integriertem Display an. Die Spezifikationen lauten: Model 4904 (0-40V / 600A), Model 4912 (0-120V / 200A), Model 4960 (0-600V / 40A). Die Komplett-Testsysteme sind aufgrund eines großen Displays besonders für den Alleinbetrieb geeignet.

1.6. Ladeverfahren

Durch ein neues stufenloses Zykliergerät wären verschiedene Ladeverfahren möglich. Hier sollen die verschiedenen Ladeverfahren kurz vorgestellt werden, welche vielleicht in einem neuen Gerät zum Einsatz kommen könnten.

Als Ladeverfahren werden die Formen der Spannungs- und Stromregelung bezeichnet, mit dem man einem Akkumulator eine Ladung zuführt. Auch die Ladungserhaltung spielt je nach Verfahren eine Rolle. Es existieren verschiedene Ladeverfahren, die sich für unterschiedliche Akkumulatortypen eignen. Die Art des Ladens hat einen direkten Einfluss auf die Lebensdauer in Form von Ladezyklen. Außerdem kann ein Ladevorgang, der die Batteriespezifikation nicht einhält, die Batterie dauerhaft schädigen oder zerstören.

Ladewirkungsgrad

Auch bei dem Laden von Batterien gibt es einen Wirkungsgrad. Die zugeführte Energie wird niemals komplett gespeichert. Ein Teil davon wird in Form von Wärme an die Umwelt abgegeben. Der Wirkungsgrad bei Akkus ist in der Größenordnung von 70-85% [15].

1.6.1. Konstantstrom-Ladeverfahren

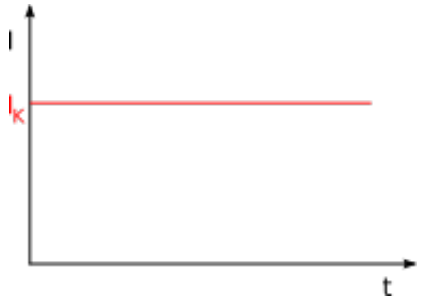


Abbildung 1.1.: Ladung mit Konstantstrom [15]

Bei dem Konstantstromverfahren wird über die gesamte Ladezeit ein konstanter Ladestrom zum Laden des Akkus verwendet (siehe Abb.: 1.1). Ist nach einer bestimmten Zeit t der Akku voll geladen, muss der Ladevorgang beendet werden. Wird nicht nach diesem Zeitpunkt abgeschaltet, kann eine Überladung den Akku zerstören. Die Ladezeit bei vollständig entladenem Akku, ergibt sich aus Formel

$$t = c * \frac{Q}{I_k} \tag{1.1}$$

t = Ladezeit, c = Ladefaktor,
Q = Akkukapazität (elektrische Ladung/Ladungsmenge), I_k = Ladestrom

Der Ladefaktor c wird je nach Akkumulatorart gewählt. Der zulässige Ladestrom hängt ebenfalls von der verwendeten Art ab [15].

1.6.2. Pulsladeverfahren

Das Pulsverfahren ist dem Konstantstromverfahren sehr ähnlich. Hier wird ebenfalls ein Konstantstrom zum Laden verwendet, dieser wird hier in Pulsen ausgegeben (siehe Abb.: 1.2). Dieses Verfahren hat im Vergleich zum normalen Konstantstromverfahren einige Vorteile. Hier kann man zum Beispiel in den stromlosen Pausen die Batteriespannung messen, ohne dass die Übergangs- und Leitungswiderstände die Messergebnisse verfälschen. Ebenfalls

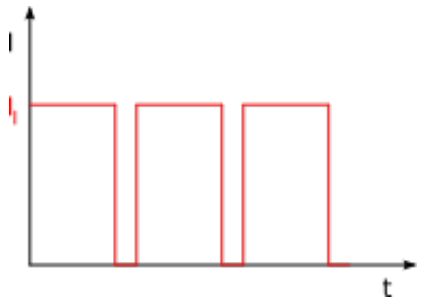


Abbildung 1.2.: Ladung mit Strompulsen [15]

mit der Manipulation des Tastverhältnisses bei einer Pulsweitenmodulation, kann man direkten Einfluss auf das Ladeverfahren nehmen, ohne die Stromstärke ändern zu müssen. Bei dem Ladevorgang kann man die stromlosen Pausen zur Messung verwenden, nach dem Ladevorgang dienen sehr kurze Strompulse, gefolgt von langen Pausen, der Ladungserhaltung [15].

1.6.3. Konstantspannungs-Ladeverfahren

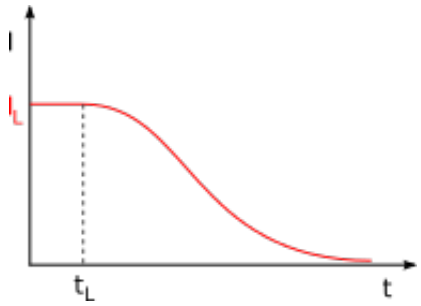


Abbildung 1.3.: Ladung mit Konstantspannung [15]

Bei dem Konstantspannungsverfahren wird über die gesamte Ladezeit eine konstante Ladespannung zum Laden des Akkus verwendet. Wegen der kleiner werdenden Spannungsdifferenz zwischen Akku und Ladegerät, sinkt der Ladestrom I mit fortschreitender Zeit t (siehe Abb.: 1.3). Der Ladestrom fließt auch nach Vollladung weiter, da dieser dann sehr kleine Ladestrom der Selbstentladung des Akkus entgegenwirkt. Dieses Ladeverfahren ist geeignet für: Bleiakkus, Li-Ion-Akkus, Rechargeable Alkali Manganese Zellen [15].

1.6.4. IU-Ladeverfahren (CCCV)

Das CCCV-Verfahren (constant current constant voltage) ist eine Kombination von Konstantstrom- und Konstantspannungsverfahren. Der Ladevorgang ist in zwei Phasen aufgeteilt. In der ersten Phase (Konstantstrom) wird der konstante Ladestrom nur durch das Ladegerät begrenzt. Im Vergleich zu dem Konstantspannungsverfahren wird der Anfangsstrom somit begrenzt. Nach Erreichen der Ladeschlussspannung wird statt Strom- auf Spannungsregelung umgestellt. In der zweiten Phase wird die Ladespannung konstant gehalten (Konstantspannung). Der Ladestrom nimmt bei fortschreitendem Ladezustand ab, bis nur noch ein kleiner, der Selbstentladung entgegenwirkender Ladestrom, übrig bleibt. [15]

1.6.5. IUoU-Ladeverfahren

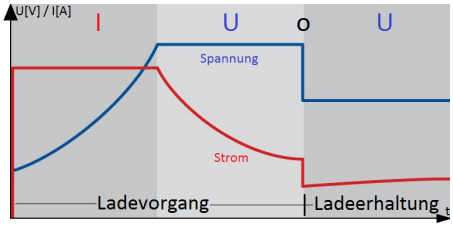


Abbildung 1.4.: IUoU-Ladeverfahren

Das IUoU-Verfahren (IUoU = konstant Strom/Spannung/Spannung) arbeitet ähnlich dem CCCV-Verfahren, allerdings wird hier nach dem Laden bis zur Ladekennspannung auf eine Erhaltungsladung umgeschaltet (siehe Abb. 1.4). Die Erhaltungsladung, die häufig gepulst und temperaturüberwacht ist, wirkt der Selbstentladung des Akkumulators entgegen. Das IUoU-Verfahren ist geeignet, Bleiakkumulatoren dauerhaft zu laden. [15]

1.6.6. Rückstromladen - Reflexladen

Rückstromladen oder Reflexladen ist ein Ladeverfahren, bei dem zum Laden periodische Stromimpulse ähnlich dem Pulsladeverfahren verwendet werden. Aber hier werden die Ladepulse von kurzen Entladestromimpulsen abgewechselt (siehe Abb.: 1.5). Dieses Ladeverfahren zählt zu den sogenannten Schnellladeverfahren. Das Reflex-Ladeverfahren eignet sich für NiCd- und NiMh-Akkus. [18]

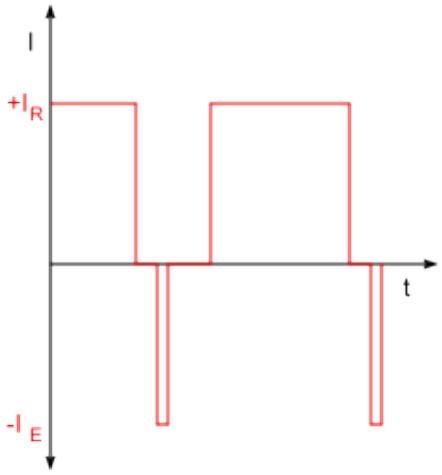


Abbildung 1.5.: Rückstromladen - Reflexladen [18]

1.6.7. Ladeverfahren für Lithium-Ionen-Akkumulatoren

Lithium-Ionen-Akkumulatoren benötigen eine Kombination aus den vorweg genannten CCCV-Ladeverfahren mit Beachtung ihrer besonderen Anfangseigenschaften. Li-Ion-Akkus vertragen am Anfang des Ladevorgangs, im Falle eines tiefen Entladezustandes, keinen hohen Ladestrom. Ein tiefer Entladezustand verringert auch die Lebensdauer dieses Akkutyps. Aus diesem Grund sollte man einen sehr niedrigen Ladezustand vermeiden. Eine passende Ladeschaltung für Li-Ion-Akkus misst daher vor dem Ladevorgang die Leerlaufspannung und passt den Ladestrom an einen Anfangswert an, welcher im Verlauf der Ladung bis zum vorgesehenen Maximalwert gesteigert werden kann. Um die Lebensdauer zur erhöhen, sollte eine tiefe Entladung vermieden werden sowie die Ladeschlussspannung unterhalb der Herstellerangabe gehalten werden. Ideal wäre hier ein Ladezustand von etwa 30-70%, dies verlängert die Lebensdauer deutlich im Vergleich zu einem Ladezustand von 10-100%. Die vom Hersteller angegebene obere Spannungsgrenze sollte niemals überschritten werden, da ab diesem Wert in der Zelle chemische Prozesse einsetzen. Hier wird die Zelle dauerhaft geschädigt und ihre Kapazität nimmt deutlich ab.

Balancing mit Batteriemanagementsystem (BMS)

Zur besseren Ermittlung der Einzelzellen wird die Ladespannung durch das Batteriemanagementsystem während des Ladens höher angesetzt, damit die Ladung akkuschonend schon vorzeitig abgebrochen werden kann. Danach folgt das Balancieren der Einzelzellen, bei dem höhere Spannungsbereiche gemieden werden. Hier besteht nur noch eine geringe Nachladung der Batterien.

Durch das serielle Zusammenschalten von mehreren Einzelzellen zur Erhöhung der Spannung, wird in der Nachladephase durch das Balancieren der Einzelzellen ein einheitliches Spannungsniveau gewährleistet [15].

1.6.8. Abschaltkriterium: Minus-Delta-U-Verfahren

In aktuellen Ladereglern wird die Spannung des zu ladenden Akkus über den gesamten Ladevorgang überwacht. Bei fortschreitendem Ladungsvorgang sinkt der differentielle Widerstand des Akkus, die an ihm abfallende Spannung steigt dagegen an. Bei Erreichen der Vollladung nimmt die Spannung nicht weiter zu und der Akku erwärmt sich, da die zugeführte Energie nicht mehr chemisch gebunden werden kann. Mit steigender Temperatur sinkt der differentielle Widerstand des Akkus weiter, deshalb sinkt die Ladespannung ab diesem Zeitpunkt, statt wie erwartet weiter zu steigen. Ab diesem Punkt müsste der Laderegler den Ladevorgang beenden. Dieses Abschaltkriterium wird allerdings nur bei NiCd- und NiMh-Akkus beobachtet. [15]

1.6.9. Abschaltkriterium: Temperaturkriterium

Bei diesem Verfahren wird nur auf die Temperatur des zu ladenden Akkus geachtet. Als Abschaltkriterium für ein Ladeverfahren ist dieses nur bedingt brauchbar, da über die Temperatur nur sehr ungenaue Ladestandaussagen getroffen werden können. Es ist allerdings sinnvoll, das Temperaturkriterium in Bezug auf die Sicherheit beim Ladevorgang zu verwenden. Eine erhöhte Temperatur beschädigt die chemische Akkumulatorstruktur und kann je nach Akku-Typ diesen auslaufen oder in Brand geraten lassen. [15]

1.6.10. Ladeschlussspannung

Die Ladeschlussspannungen sind für die verschiedenen Akku-Typen unterschiedlich. Bei 20°C liegen diese pro Zelle bei:

- Bleiakkumulator
  2,42 V (Lade-Erhaltung 2,23 V)
- NiCd/NiMH-Akku
  1,45 V
- Lithium-Cobaltdioxid-Akkumulator
  4,2 V

- Lithium-Polymer-Akku (LiPo)
  4,2 V
- Lithium-Eisenphosphat-Akku (LiFePO4)
  3,6 V (maximal 3,8 Volt)
- Nickel-Zink-Akkumulator
  1,90 V/Zelle

Wie schon erwähnt, ist bei Lithium-Akkus zur Verlängerung der Lebensdauer eine frühere Abschaltung des Ladevorgangs von Vorteil. Diese sollte hier etwa 0,3 Volt unterhalb der angegebenen Ladeschlussspannung liegen. [15]

2. Analyse

2.1. Sicherheitskonzept

Aufgabe dieser Bachelorarbeit war es, ein Sicherheitskonzept für ein neues Zykliersystem zu beschreiben. Es wurde entschieden, dass dies über einen zweiten Mikrocontroller realisiert werden sollte. Dieser Schutzcontroller soll autark von dem Zykliercontoller laufen. Entscheidet das Schutzsystem, dass ein Weiterlaufen in jeglicher Form unterbunden werden muss, soll dem Zykliersystem komplett die Möglichkeit der Leistungsregelung entzogen werden. Dies wurde über Leistungsrelais realisiert. Sind diese geöffnet, kann der Zykliercontroller keine Ströme/Spannungen mehr steuern. Somit ist jeglicher Fehlerfall in dieser Kategorie ausgeschlossen. Außerdem soll das Schutzsystem das Zykliersystem noch durch Informationen in Form von Messwerten unterstützen. Die beiden Teilsysteme sollten voneinander galvanisch getrennt laufen. Somit wäre eine Verbindung beider Systeme nur über Optokoppler sinnvoll. Die weiteren zu überwachenden Werte wie Temperatur, Spannung, Strom, welche einen Indikator für Fehlerfälle darstellen können, müssen daher ständig vom Schutzsystem auf ihre Werte überprüft werden. Sind diese Werte erhöht, sollen Warnungscodes, sind diese zu hoch, sollen Fehlercodes an das Zykliersystem übertragen werden. So kann das Zykliersystem bei Warnungen vorzeitig reagieren, bzw. bei Fehlern die Gründe eines eventuellen Laufzeitabbruchs protokollieren. Eine SD-Karte ist bei dem Zykliersystem zum Protokollieren der Messungen ohnehin vorgesehen, von daher würde eine zweite im Schutzsystem wenig Sinn ergeben. Die Einstellungen und letzten Fehlermeldungen sollen im ROM des Schutzsystems gespeichert werden, um diese bei Bedarf abrufen zu können. Das Hauptaugenmerk des Gesamtkonzeptes soll auf der autarken Überwachung liegen. Selbst wenn das Zykliersystem grobe Fehler produziert, die zur Zerstörung des Gesamtaufbaus führen könnten, soll das Schutzsystem das Auftreten solcher Fehler verhindern bzw. den entstehenden Schaden minimieren.

2.2. 1-wire

Zur Temperaturmessung sollen in dem Endgerät mehrere Temperatursensoren verwendet werden. Hierfür wurden 1-wire Sensoren vom der Firma Dallas Typ DS18B20 verwendet.

Der 1-wire Bus ist eine serielle Schnittstelle der Firma Dallas Semiconductor Corp, welche 2001 von Maxim Integrated aufgekauft wurde. Mit dieser Schnittstelle ist es möglich, über eine einzige Leitung (DQ) Daten zu senden, zu empfangen und die Stromversorgung für diesen Bus bereitzustellen. Eine zusätzliche Masseleitung (GND) ist ebenfalls erforderlich. Die Kommunikation über die DQ Leitung arbeitet bidirektional im Halbduplexverfahren. Die Datenübertragung läuft asynchron, es wird daher kein Taktsignal übertragen. Die Übertragung erfolgt im One-Master/Multi-Slave Prinzip. Dies bedeutet, es gibt nur einen Master, welcher bei 1-wire bis zu 100 Slaves ansteuern kann. Jedes 1-wire Bauteil besitzt eine 64-Bit-ROM-ID, welche aus einem 8-Bit-Family-Code, einer 48-Bit-Seriennummer und einer 8-Bit-CRC-Checksumme besteht. Über diese 64-Bit-ROM-ID, lassen sich die Bauteile in einem 1-wire Bus direkt ansteuern oder abfragen. Die DS18B20 Temperatursensoren verfügen zusätzlich über ein weiteres ROM, hier Scratchpad genannt, welches sich auslesen lässt. Dieses Scratchpad ist 9 Bytes lang und enthält neben der Temperatur noch weitere Informationen und Konfigurationseinstellungen (siehe Abb. 2.2) [13].

2.2.1. 1-wire via UART

Der in dieser Arbeit verwendete Mikrocontroller besitzt keine integrierte 1-wire Schnittstelle. Deshalb wurde hier mit Hilfe eines zusätzlichen Bauteils eine 1-wire Schnittstelle, unter Verwendung eines Dual-Buffers NC7WZ07 (Open-Drain Output) der Firma Fairchild, über die UART-Schnittstelle des Mikrocontrollers realisiert.

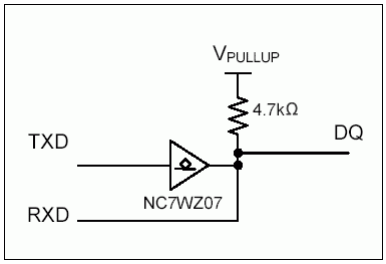


Abbildung 2.1.: UART zu 1-wire Schaltung [6]

In Abb. 2.1 ist der von Maxim beschriebene Aufbau zu sehen. Hier wird der Open Drain Buffer an den TXD Ausgang des Mikrocontrollers (MC) angeschlossen und der Ausgang des Open Drain Buffers wiederum mit dem RXD Eingang des MC. Somit hat man aus den zwei UART Leitungen des MC eine 1-wire Leitung geschaffen. Auf die besonderen Voraussetzungen des 1-wire-Busses muss jetzt softwareseitig eingegangen werden [6].

2.2.2. 1-wire CRC

Der CRC-Check wurde aus einem Teil der “Application Note 27“ der Firma Maxim Integrated übernommen. Hier gab es ein Beispiel zur Verwendung der von Maxim patentierten 1-wire Schnittstelle in Verbindung mit einem dazu passenden CRC-Check. Diese CRC-Prüfsumme dient zur sicheren Datenübertragung von dem Sensor an den Master. [7]

2.2.3. 1-wire Testprogramm

Es wurde zu Testzwecken ein Programm geschrieben, mit dem sich die einzeln angeschlossenen Temperatursensoren mit ihren ROM-Codes und Temperaturen auslesen lassen. Im weiteren Verlauf wurde dieses Testprogramm zu einem elementaren Teil des Gesamtaufbaus. Es ist zwar programmiertechnisch möglich, die angeschlossenen Temperatursensoren automatisch durchzunummerieren und danach abzufragen, in dem Gesamtaufbau macht es aber mehr Sinn, die ROM-Codes vorher zu kennen. Deshalb wurde auf eine automatische Adressierung verzichtet. Die ROM-Codes werden daher zuvor mit dem Testprogramm notiert und nummeriert, um danach von Hand nach dem Löten, auf den jeweiligen Sensor geschrieben zu werden. Somit wird bei der Verkabelung im Innenraum des Gesamtaufbaus sichergestellt, dass die Messpunkte nicht vertauscht werden. Ohne diese Nummerierungen auf der Verkabelung ist sonst zu schnell der Überblick verloren.

Testprogramm Beispiel

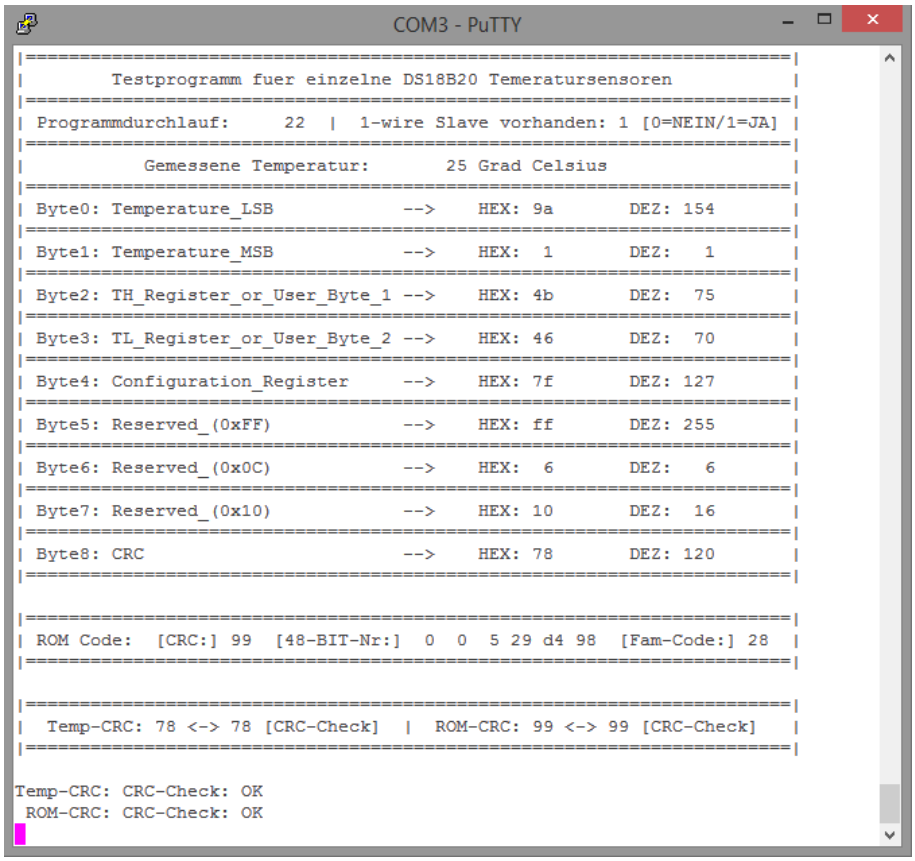


Abbildung 2.2.: 1-wire Testprogramm

In Abb. 2.2 ist die Bildschirmausgabe des Testprogramms dargestellt. Es wird angezeigt, ob ein Sensor angeschlossen ist oder nicht. Die Temperatur wird ebenfalls oben angezeigt. Des Weiteren werden hier die beiden ausgelesenen ROM-Codes/Scratchpads detailliert mit der Bedeutung jedes einzelnen Bytes ausgegeben. Die CRC-Werte der beiden ROM-Codes/Scratchpads werden berechnet und im Fehlerfall wird unten ein CRC Fehler angezeigt.

2.3. Wärmewiderstand

$$\Delta T = P_\theta * R_\theta \tag{2.1}$$

ΔT = Temperaturdifferenz zwischen Wärmequelle und Umgebung in K
P_θ = Wärmeleistung in W
R_θ = Wärmewiderstand in K/W

Der Wärmewiderstand R_θ ist die wichtigste Eigenschaft eines Kühlkörpers und wird in K/W, Kelvin pro Watt, angegeben. R_θ ist der Quotient des Unterschiedes einer Temperatur zwischen Wärmequelle und Umgebung in Kelvin, zu der abgeführten Wärmeleistung in Watt. Je effektiver der Kühlkörper, desto niedriger ist der Wärmewiderstand, da der Kühlkörper die Wärmeleistung mit einem geringeren Temperaturunterschied abführen kann. Das zu kühlende Bauteil wird bei einer niedrigeren Temperatur betrieben. Dies verlängert die Lebensdauer von elektronischen Bauteilen deutlich [9].

2.4. Widerstände

Die Widerstände stammen aus der Entwicklung des neuen Zykliergerätes der BATSEN Arbeitsgruppe. Es wurden zwei verschiedene Widerstandstypen für die Lade- und Entladeschaltung gewählt. Die Widerstände wurden in jeweils drei Widerstandspakete zusammengefasst, je drei für den Ladekreis und drei für den Lastkreis. Teil der Aufgabe dieser Bachelorarbeit ist die Kühlung und Temperaturüberwachung dieser Widerstandspakete.

2.4.1. Widerstandsmessung mit Stromfehlerschaltung (Realwert)

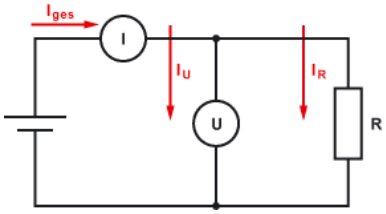


Abbildung 2.3.: Stromfehlerschaltung [3]

Zur Bestimmung der realen Widerstände der beiden Widerstandspakete, wurden mittels Stromfehlerschaltung die Realwerte bestimmt. Dazu waren zwei Messgeräte wie in Abb. 2.3 nötig. Parallel an den Gesamtwiderstand wurde ein Spannungsmesser angeschlossen, der aufgrund seines sehr hohen Innenwiderstandes fast keinen Einfluss auf den Gesamtstrom I_{ges} hat. In beiden Fällen haben die Lade- bzw. Lastwiderstände im Verhältnis zum Messgerät extrem kleine Widerstände. Das davor geschaltete Strommessgerät gibt somit einen sehr

exakten Strommesswert, mit dem man den realen Widerstand eines Widerstandspaketes berechnen kann.

2.4.2. Ladewiderstände

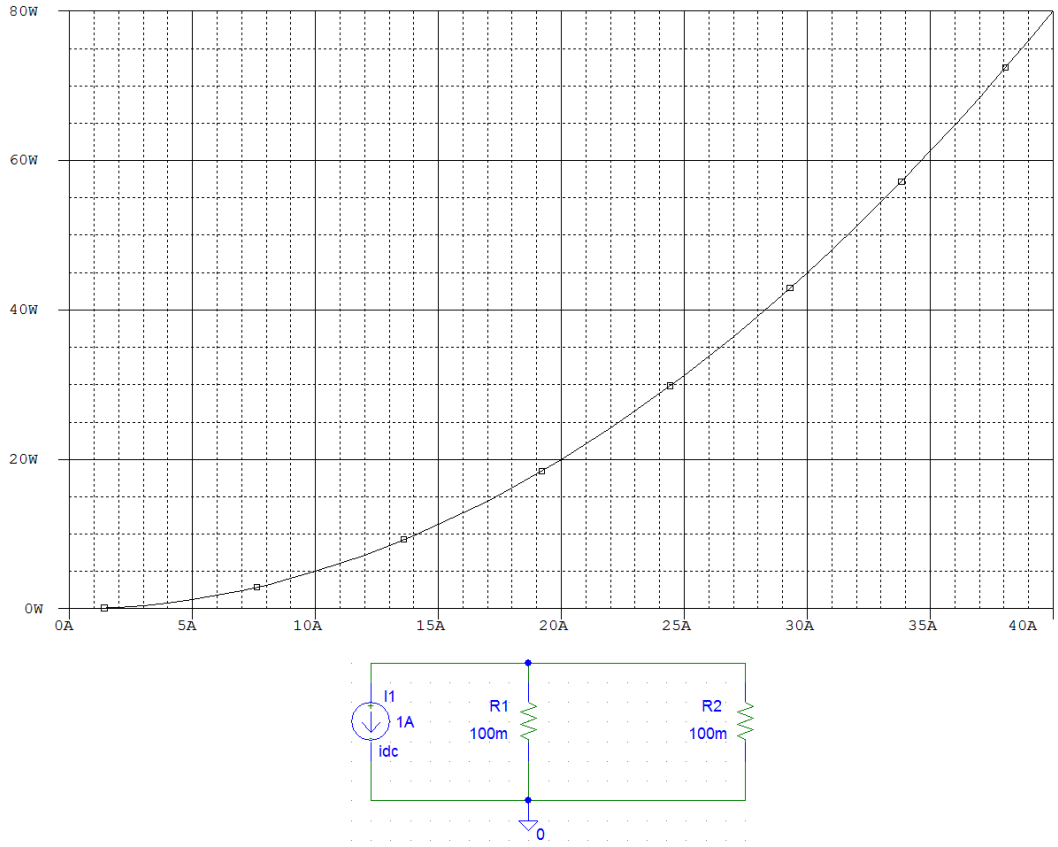


Abbildung 2.4.: umgesetzte Leistung, ideale Widerstände (Parallelschaltung)

Hier wurden zwei 0,1 Ohm Widerstände parallel geschaltet. Der Gesamtwiderstand dieses Widerstandspaketes ist rechnerisch 0,05 Ohm. Auf Abb. 2.4 ist die umgesetzte Leistung an den Widerständen aufgezeichnet. Diese kann sich je nach Aufbau (Übergangswiderstände) und Temperatur verändern und dient deshalb nur als Annahmewert. Im Falle einer vollen Belastung von 40 Ampere würden über die Widerstände ca. 80 Watt in Wärme umgewandelt werden.

Gemessener Widerstandswert

Der nach Abb. 2.3 gemessene reale Wert des Widerstandspaketes ist: 0,0556 Ohm. Der hier abweichende Widerstandswert kommt von den Übergangswiderständen, die durch die

Lötstellen hervorgerufen werden. Aufgrund der Befestigung der $2,5 mm^2$ Kabel an den Widerstandsklemmen, war hier die Verwendung von etwas mehr Lötzinn notwendig.

2.4.3. Lastwiderstände

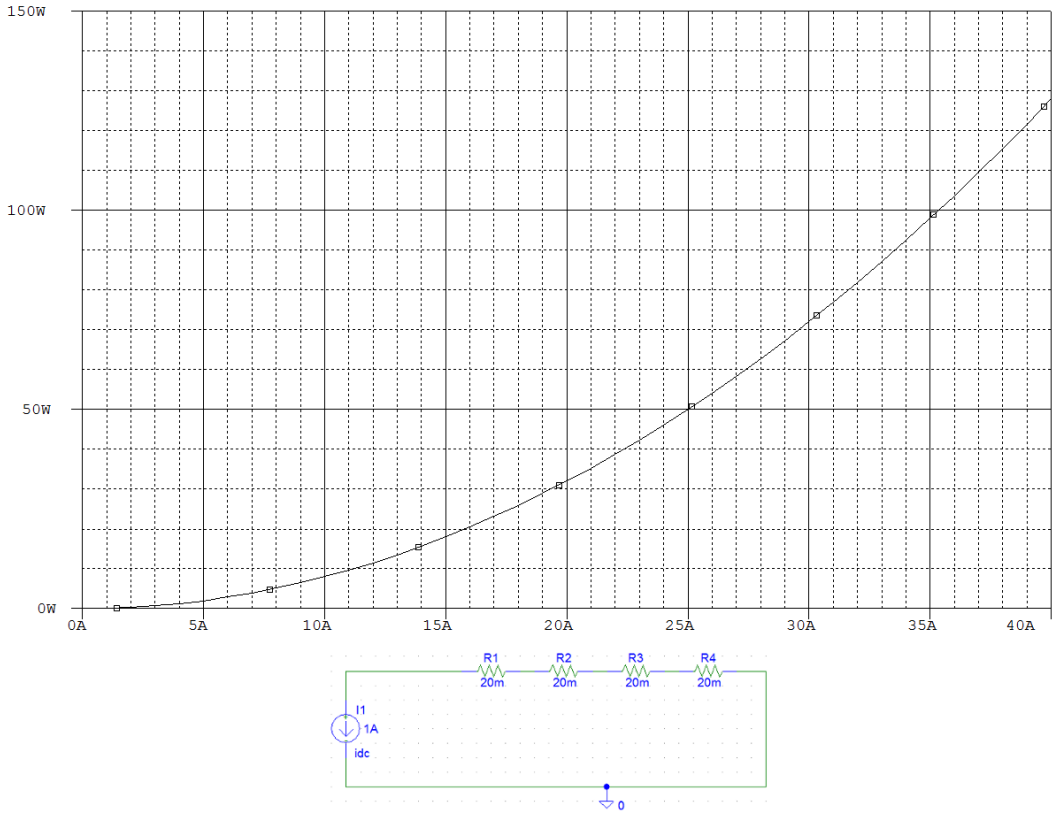


Abbildung 2.5.: umgesetzte Leistung, ideale Widerstände (Serienschaltung)

Hier wurden vier 0,02 Ohm Widerstände in Reihe geschaltet. Der Gesamtwiderstand dieses Widerstandspaketes ist rechnerisch 0,08 Ohm. Auf Abb. 2.5 ist die umgesetzte Leistung an den Widerständen aufgezeichnet. Diese kann sich je nach Aufbau (Übergangswiderstände) und Temperatur verändern und dient deshalb nur als Annahmewert. Im Falle einer vollen Belastung von 40 Ampere würden über die Widerstände ca. 130 Watt in Wärme umgewandelt werden.

Realer Widerstandswert

Der nach Abb. 2.3 gemessene reale Wert des Widerstandspaketes ist: 0,090 Ohm. Ebenso wie schon bei dem Realwiderstand des anderen Widerstandspaketes, sind hier ebenfalls die Übergangswiderstände, die durch die Lötstellen hervorgerufen werden, für die Abweichung

verantwortlich. Bei diesem Aufbau ist der veränderte Realwert im Verhältnis größer, da bei der Reihenschaltung von vier Widerständen mehr Lötstellen und somit auch mehr Übergangswiderstände vorhanden sind als bei der Parallelschaltung von zwei Widerständen.

2.5. Testaufbau Widerstandspakete

2.5.1. Aufbau

Zur Analyse des Temperaturverhaltens eines einzelnen Widerstand-Lüfter-Verbundes, wurde dieser an eine Konstantstromquelle angeschlossen. Des Weiteren wurden 6 Temperatursensoren an dem Bauteil verteilt. Dazu wurde ein Tischmultimeter (Fluke 45) parallel zur Spannungsprotokollierung an die Widerstände angeschlossen. Das Multimeter überträgt den aktuellen Messwert in Volt über einen RS232/TTL Wandler an den Mikrocontroller. Das Mikrocontrollerprogramm des Schutzsystems wurde auf das Nötigste zusammengestaucht und vereinfacht, so dass es alle fünf Sekunden die vergangene Zeit in Sekunden, die Messwerte der Temperatursensoren sowie den Messwert des Tischmultimeters über die serielle Schnittstelle im CSV-Format an den angeschlossenen Computer überträgt. Auf dem Rechner läuft das Terminalprogramm "PuTTy", mit dem die Messwerte als CSV-Datei gespeichert werden. Die CSV-Datei wird danach mit MATLAB ausgewertet, um den Wärmewiderstand des Widerstand-Lüfter-Verbundes zu bestimmen.

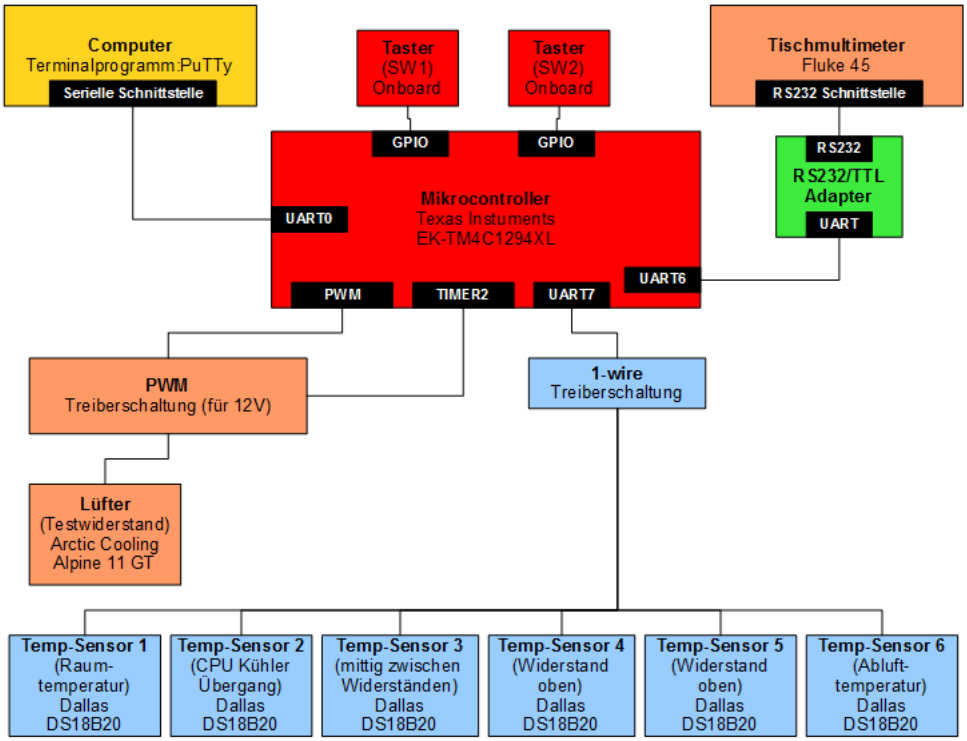


Abbildung 2.6.: Verschaltung der Komponenten

2.5.2. Temperaturmesspunkte am Testaufbau

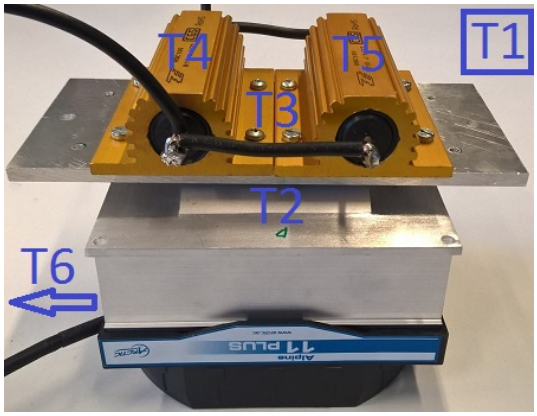


Abbildung 2.7.: Testaufbau - Temperaturmesspunkte am Widerstandspaket (Parallelschaltung)

Testaufbau - Temperaturmesspunkte Widerstandspaket (Parallelschaltung)

- Temperatursensor 1: Raumluft
- Temperatursensor 2: zwischen Befestigungsplatte und Kühlkörper
- Temperatursensor 3: mittig, zwischen den beiden Widerständen
- Temperatursensor 4: auf dem linken Widerstand
- Temperatursensor 5: auf dem rechten Widerstand
- Temperatursensor 6: Abluft des Kühlkörpers

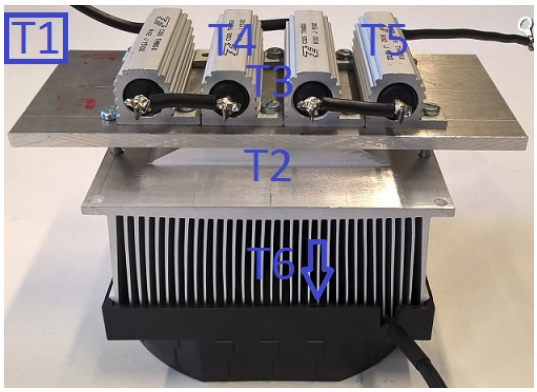


Abbildung 2.8.: Testaufbau - Temperaturmesspunkte am Widerstandspaket (Serienschaltung)

Testaufbau - Temperaturmesspunkte Widerstandspaket (Serienschaltung)

- Temperatursensor 1: Raumluft
- Temperatursensor 2: zwischen Befestigungsplatte und Kühlkörper
- Temperatursensor 3: mittig, zwischen den beiden mittleren Widerständen
- Temperatursensor 4: auf dem linken-mittleren Widerstand
- Temperatursensor 5: auf dem rechten-äußeren Widerstand
- Temperatursensor 6: Abluft des Kühlkörpers

2.5.3. Messpläne

Messplan 1

Zur Analyse des Widerstand-Lüfter-Verbundes wurden zwei Messpläne erstellt und durchgeführt. Die Stromvorgabewerte sowie die Lüfterdrehzahlen (PWM) mussten hierbei von Hand in einem Intervall von ca. 5 Minuten eingestellt werden. Eine Stoppuhr diente bei diesem Versuch als Zeitgeber.

Dauer	Strom	Lüfter
1 min.	0A	0 RPM
5 min.	5A	Aus
5 min.	5A	50% PWM
5 min.	5A	100% PWM
5 min.	10A	Aus
5 min.	10A	50% PWM
5 min.	10A	100% PWM
5 min.	15A	Aus
5 min.	15A	50% PWM
5 min.	15A	100% PWM
5 min.	20A	Aus
5 min.	20A	50% PWM
5 min.	20A	100% PWM
5 min.	0A	100% PWM
5 min.	0A	Aus

Tabelle 2.1.: Messplan 1

Messplan 2

Der zweite Versuch bezog sich auf die längere Volllast (20A) ohne Lüfter und den direkten Vergleich zu einer maximalen Lüfterdrehzahl (ca. 2100 RPM). Hier wurde der Lüfter bei einer Temperatur von 50° Celsius eingeschaltet.

Dauer	Strom	Lüfter
ca 30 min. (Lüfter bei 50°C eingeschaltet)	20A	Aus
ca 30 min. (Lüfter bei 50°C eingeschaltet)	20A	100% PWM

Tabelle 2.2.: Messplan 2

Messplan 3

Der dritte Versuch bezog sich auf die längere Volllast (20A) ohne Lüfter und den Vergleich zu drei verschiedenen Lüfterdrehzahlen (0 RPM, 5%PWM, 50%PWM, 100%PWM,). Um in den interessanten Bereich zu kommen, wurden die Widerstände etwa 30 Minuten lang "vorgeheizt". Erst Dann wurde mit dem Aufzeichnen der Messwerte begonnen.

Dauer	Strom	Lüfter
30 min. (ca.)	20A	Aus (Messwerte nicht aufgezeichnet!)
20 min.	20A	Aus
20 min.	20A	5% PWM
20 min.	20A	50% PWM
20 min.	20A	100% PWM

Tabelle 2.3.: Messplan 3

2.5.4. Messergebnisse

Die detaillierten Messergebnisse der sechs Messungen, welche mit MATLAB ausgewertet wurden, befinden sich im Anhang:

- Messung 1 - 2P (Abb. A.1)
- Messung 2 - 2P (Abb. A.2)
- Messung 3 - 2P (Abb. A.3)
- Messung 1 - 4R (Abb. A.4)
- Messung 2 - 4R (Abb. A.5)
- Messung 3 - 4R (Abb. A.6)

Es werden vier Plots dargestellt: Strom/Spannung, Tastgrad/Lüfterdrehzahl, Temperaturen 1-6, Wärmewiderstand. Die Ströme/Spannungen sowie Tastgrade/Lüfterdrehzahlen sind die Vorgabemesswerte aus den Messplänen. Die Temperaturen sind die Messewerte, die Wärmewiderstände resultieren aus der Formel 2.1.

Messergebnisse - Ladewiderstände - Wärmewiderstand

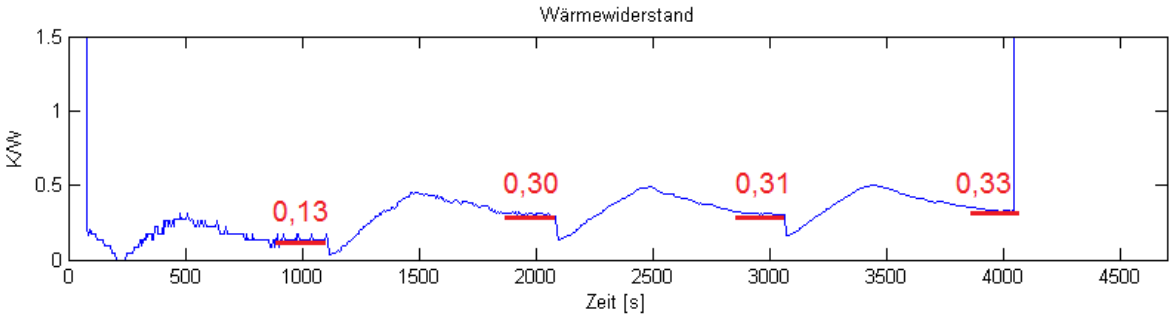


Abbildung 2.9.: Ergebnisse: Messung 1 - Ladewiderstände Wärmewiderstand (siehe Tab. 2.4)

Die Wärmewiderstände aus Abb. 2.9 liegen bei:

Strom	RPM	Wärmewiderstand
5A	2100	0,13 K/W
10A	2100	0,30 K/W
15A	2100	0,31 K/W
20A	2100	0,33 K/W

Tabelle 2.4.: Messung 1 Ladewiderstände - Wärmewiderstand

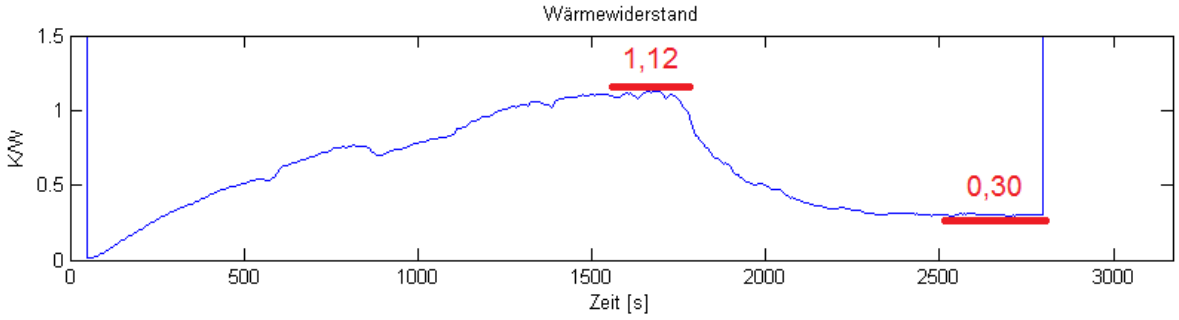


Abbildung 2.10.: Ergebnisse: Messung 2 - Ladewiderstände Wärmewiderstand (siehe Tab. 2.5)

Die Wärmewiderstände aus Abb. 2.10 liegen bei:

Strom	RPM	Wärmewiderstand
20A	0	1,12 K/W
20A	2100	0,30 K/W

Tabelle 2.5.: Messung 2 Ladewiderstände - Wärmewiderstand

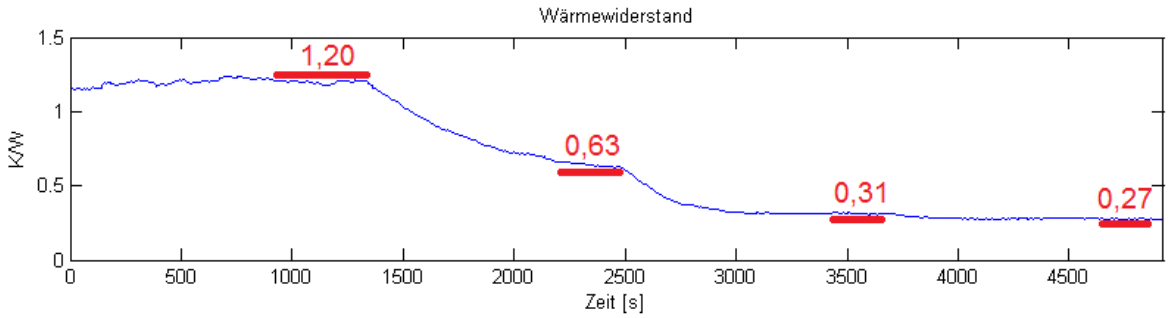


Abbildung 2.11.: Ergebnisse: Messung 3 - Ladewiderstände Wärmewiderstand (siehe Tab. 2.6)

Die Wärmewiderstände aus Abb. 2.11 liegen bei:

Strom	RPM	Wärmewiderstand
20A	0	1,20 K/W
20A	840	0,63 K/W
20A	1670	0,31 K/W
20A	2100	0,27 K/W

Tabelle 2.6.: Messung 3 Ladewiderstände - Wärmewiderstand

Messergebnisse - Lastwiderstände - Wärmewiderstand

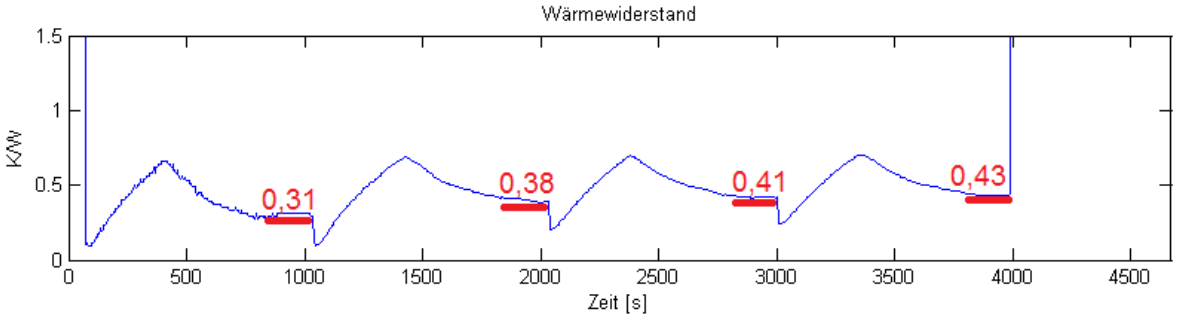


Abbildung 2.12.: Ergebnisse: Messung 1 - Lastwiderstände Wärmewiderstand (siehe Tab. 2.7)

Die Wärmewiderstände aus Abb. 2.12 liegen bei:

Strom	RPM	Wärmewiderstand
5A	2100	0,31 K/W
10A	2100	0,38 K/W
15A	2100	0,41 K/W
20A	2100	0,43 K/W

Tabelle 2.7.: Messung 1 Lastwiderstände - Wärmewiderstand

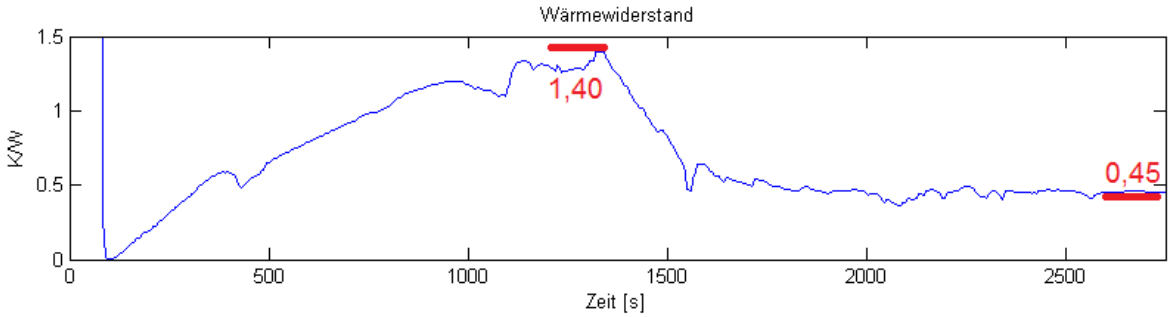


Abbildung 2.13.: Ergebnisse: Messung 2 - Lastwiderstände Wärmewiderstand (siehe Tab. 2.8)

Die Wärmewiderstände aus Abb. 2.13 liegen bei:

Strom	RPM	Wärmewiderstand
20A	0	1,40 K/W
20A	2100	0,45 K/W

Tabelle 2.8.: Messung 2 Lastwiderstände - Wärmewiderstand

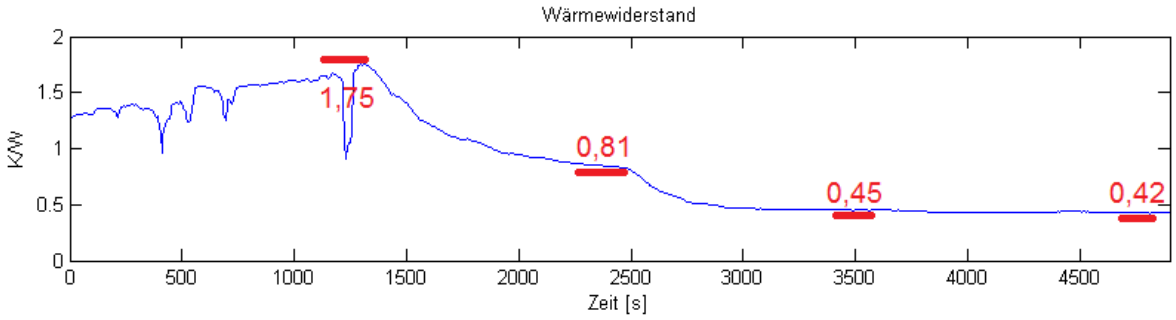


Abbildung 2.14.: Ergebnisse: Messung 3 - Lastwiderstände Wärmewiderstand (siehe Tab. 2.9)

Die Wärmewiderstände aus Abb. 2.14 liegen bei:

Strom	RPM	Wärmewiderstand
20A	0	1,75 K/W
20A	840	0,81 K/W
20A	1670	0,45 K/W
20A	2100	0,42 K/W

Tabelle 2.9.: Messung 3 Lastwiderstände - Wärmewiderstand

2.5.5. Zusammenfassung der Messergebnisse

Wie erwartet, konvergiert der Wärmewiderstandswert nach einer gewissen Zeit gegen einen festen Wert. Dieser ist abhängig von der zugeführten Leistung, welche für den Temperaturunterschied verantwortlich ist, der Umgebungstemperatur und der Lüfterdrehzahl, welche den Kühleffekt des Widerstandspaketes direkt beeinflusst. Bleiben hier der Strom und die Lüfterdrehzahl konstant, pendelt sich der Wärmewiderstandswert ebenfalls auf einen konstanten Wert ein.

Die Kühlung der Widerstände ist mit dem gewählten Aufbau problemlos möglich. Der Wärmewiderstandswert ist bei einer hohen Drehzahl der Lüfter in diesem Aufbau effektiv genug, die Bauteile in einem Temperaturlevel unter 50° Celsius zu halten. Im späteren Test soll der Widerstand-Lüfter-Verbund in dem Gesamtaufbau erneut auf den Temperaturverlauf hin untersucht werden. Es ist zu erwarten, dass die gewählten Gehäuselüfter noch einen Zu- und Abluftstrom erzeugen, mit denen die Bauteile auch bei höheren Strömen noch in einem tolerierbaren Temperaturbereich zu betreiben sind.

2.6. Vollständiges Abschalten des Schutzcontrollers

Hier soll auf die Frage des vollständigen Abschaltens des Schutzsystems zu Testzwecken eingegangen werden. Damit ist ein Modus gemeint in dem der Schutzcontroller alles frei gibt, ohne das System weiter zu überwachen.

Ein solcher Modus würde die Idee eines Schutzsystems insgesamt in Frage stellen. Es gibt hier trotzdem einen möglichen sinnvollen Grund, einen “Development Mode“ (DM) zu implementieren.

Bei dem Anschließen von externen Messgeräten zur Analyse des kompletten Zykliersystems, könnte ein Abschalten der Schutzfunktionen die Kalibrierung der Messgeräte oder eines Messaufbaus die Messung selbst erst ermöglichen. Eine Blockade seitens des Schutzsystems würde innere Analysen des Gesamtsystems deutlich erschweren.

In dieser Arbeit wurde sich gegen einen DM entschieden. Die Folgen und Fehler sind aufgrund des derzeitigen Entwicklungsstandes des Gesamtsystems noch nicht absehbar. Sollte das Gesamtsystem später einmal überarbeitet werden, könnte man sich erneut über einen Testmodus Gedanken machen. Hierfür wären aber Testläufe und Analysen des Gesamtsystems (Schutzsystem in Verbindung mit dem Zykliersystem) erforderlich, die zu diesem Zeitpunkt noch nicht möglich waren, da dass Zykliersystem noch nicht ausgereift zur Verfügung stand.

3. Design

3.1. Frontplattenkonzept und Vergleich

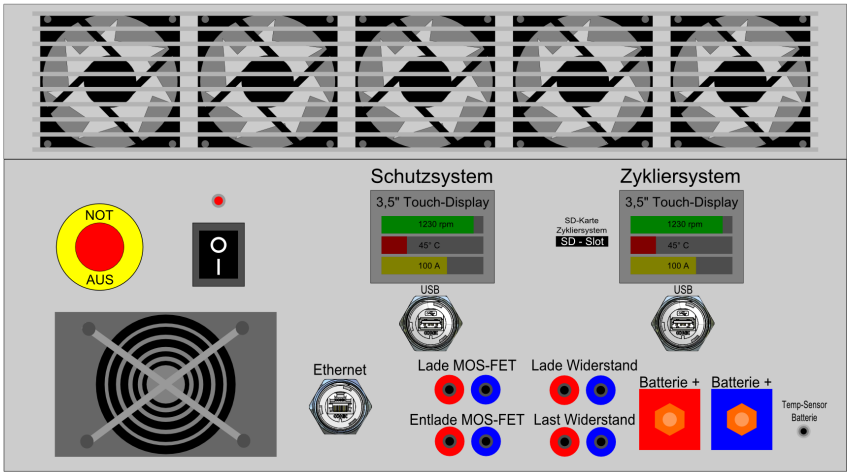
Hier sollen die Vorüberlegungen für das gesamte Design und die Entwicklung vorgestellt werden. Die ersten Konzeptideen dienten hier als Vorlage und wurden für die finale Frontplatte, bzw. für das finale Endgerät berücksichtigt. Das Endprodukt wich in einigen Punkten leicht von der Konzepterstellung ab, ohne die Grundidee zu verwerfen.

3.1.1. Frontplattenkonzept

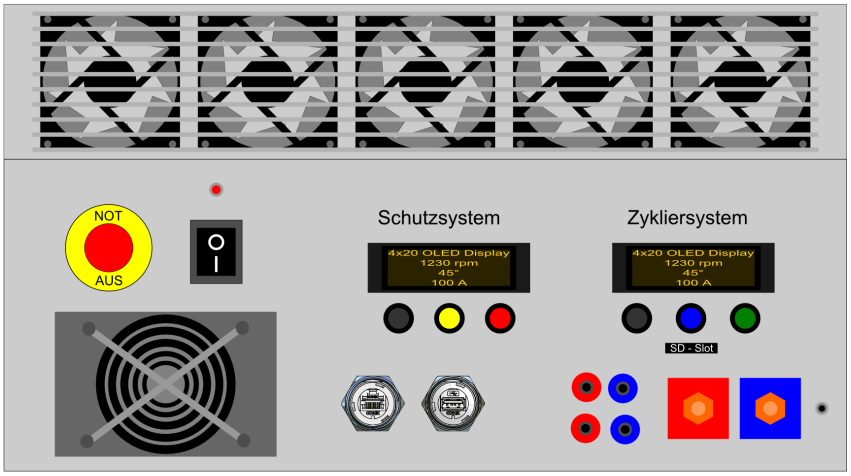
In Abb. 3.1 sind drei mögliche Frontplattenkonzepte gegenübergestellt. Es gibt verschiedene Möglichkeiten die Bereiche Steuerung & Schutz voneinander zu unterteilen oder diese in einem Gesamtkonzept zusammenzufassen. Außerdem sollte die Belüftung des Innenbereiches ebenfalls mit auf der Vorderseite des Gerätes berücksichtigt werden.

Die drei Konzepte sind für ein 2HE (Höheneinheiten) oben und ein 4HE unten erstellt worden. Bei allen drei Konzepten sind folgende Eigenschaften identisch. In dem oberen 2HE Element befinden sich fünf Lüfter, welche die Frischluft zum Kühlen in den Innenraum des Gerätes transportieren. Auf der linken Seite des 4HE Moduls liegen oben ein Not-Aus-Knopf und der Hauptschalter. Darunter befindet sich das Zykliernetzteil, welches ebenfalls die Luft nach Innen zieht. Unten rechts auf dem 4HE Modul befinden sich die gesamten Anschlüsse für Peripherie, Messgeräte und die zu zyklierende Batterie. Der rechte obere Bereich des 4HE Moduls enthält Bedienelemente sowie Anzeigeelemente.

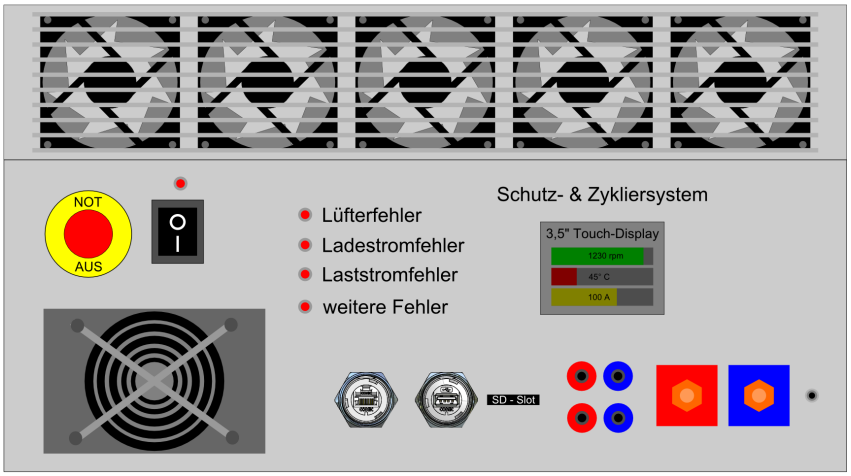
Die Konzepte unterscheiden sich nur im mittleren bis rechten Bereich des 4HE Moduls mit der Anordnung der Elemente und der Bauteilauswahl (Displays). Bei dem Vergleich der Konzepte ist deshalb der Schwerpunkt auf die Bedienung des Endgerätes gelegt worden, da sich die Anschlussmöglichkeiten für Peripherie oder Messinstrumente kaum voneinander unterscheiden.



Konzept 1



Konzept 2



Konzept 3

Abbildung 3.1.: Frontplattenkonzepte

3.1.2. Vergleich der verschiedenen Konzepte

Konzept 1 - 2 Touch Displays
Bei diesem Konzept hat das Endgerät zwei Touchdisplays, die die beiden Mikrocontroller getrennt ansteuern. Somit ist es möglich, die zwei Systeme komplett eigenständig zu bedienen. Auf weitere Anzeigeelemente in Form von LEDs wurde komplett verzichtet, da die Fehlermeldungen hier alle auf dem Display ausgegeben werden sollen. Auch auf Eingabeknöpfe wurde aus demselben Grund verzichtet.

Vorteile

- einfache Änderung der Benutzeroberfläche
- spart Knöpfe und Anzeigeelemente (LEDs)
- geringe Anschaffungskosten
- viele Möglichkeiten für neue Ideen durch die Farbdisplays
- sehr viel Platz zum Anzeigen aktueller Werte
- wirkt optisch modern

Nachteile

- benötigt aufgrund der zwei Touchdisplays eine komplexere Programmierung
- ist das Display kaputt, ist keine Eingabe durch Knöpfe möglich
- kein sofortiger Zugriff auf wichtige Funktion, wenn man sich in einem Untermenü befindet

Konzept 2 - 2 OLED Displays
Bei dem zweiten Konzept werden zwei OLED Displays für die Ausgabe von Informationen verwendet, für die Eingabe sind unter den OLED Displays jeweils drei Druckknöpfe angebracht. Wie auch bei dem ersten Konzept werden hier die beiden Teilsysteme getrennt angesteuert und eine komplett eigenständige Bedienung ist hier ebenfalls gegeben.

Vorteile

- bereits erprobt in vorigem Zykliersystem
- direkter Zugriff durch die Druckknöpfe
- Programmierung einfacher als die Touch-Varianten

Nachteile

- Teurer als Touch-Variante
- nicht so variationsreich wie Farbdisplay
- neue Ideen sind auf die 2-farbige Anzeige beschränkt
- wenig Platz zum Anzeigen aktueller Werte
- wirkt optisch veraltet

Konzept 3 - 1 Touch Display
Bei dem dritten Konzept ist nur ein Touchdisplay für beide Systeme vorhanden. Die Bedienung und Ausgabe von Informationen beider Teilsysteme wird über das Touchdisplay realisiert, unterstützend stehen noch LEDs für die direkte optische Fehlerrückmeldung zur Verfügung.

Vorteile

- minimalistische Bedienoberfläche
- günstigste Variante
- vereinfachte Änderung an Benutzeroberfläche
- belegt weniger Platz im inneren des Gerätes (Verkabelung)
- viele Möglichkeiten für neue Ideen durch das Farbdisplay
- viel Platz zum Anzeigen aktueller Werte
- wirkt optisch modern

Nachteile

- bei ausgelöstem Fehlerfall (Steuercontroller wird ausgeschaltet) problematische Fehleranzeige, falls dieser das Display ansteuert und abgeschaltet wird
- jeder Fehler benötigt eigene LED (Bei Änderungen müssten neue Löcher gebohrt werden.)
- benötigt aufgrund des Touchdisplays eine komplexere Programmierung

3.1.3. Frontplatte (Endprodukt)

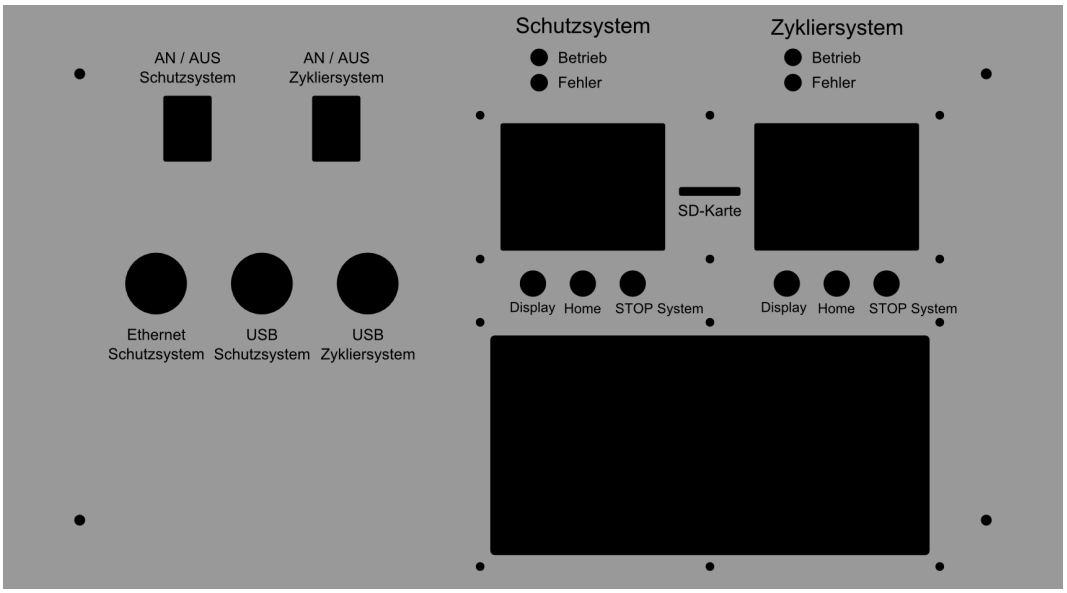


Abbildung 3.2.: Frontplatte: CAD Entwurf Endprodukt 6HE (oben)

Das endgültige Design der Frontplatte weicht deutlich von den ersten drei Entwürfen ab. Hier wurden die Vorteile der verschiedenen Konzepte in ein neues Frontplattendesign vereint. Die auffälligste Änderung betrifft die Größe des Gesamtaufbaus, es wurden statt 2HE (Höheneinheiten) oben und 4HE unten, 6HE oben und 3HE unten gewählt. Dies wurde nicht aufgrund der Anforderungen im Frontplattenbereich entschieden, sondern aufgrund der Platzaufteilung des Innenbereiches. Somit war mehr Spielraum für die inneren Bauteile gegeben. Auf der Frontplatte wurden zwei verschiedene Netzschalter verbaut, um die beiden Teilschaltungen voneinander unabhängig aktivieren zu können. Durch ein Relais ist es trotzdem nur möglich , dass Schutzsystem ohne das Zykliersystem zu betreiben, die umgekehrte Kombination ist nicht möglich. Ein Aktivieren des Schutzcontrollers ohne Zykliercontroller soll hier Test- und Analyseverfahren ermöglichen. Außerdem ist es somit möglich, das Schutzsystem vorweg zu konfigurieren und erst danach das Zykliersystem zuzuschalten. Die direkten Anzeigen via LEDs sind beibehalten worden, allerdings sind diese nur noch auf die zwei wesentlichen Eigenschaften “Betrieb“ und “Fehler“ reduziert. Alle weiteren Ausgaben sollen auf den Displays erfolgen. Trotz der möglichen Eingabe über die Touch-Displays, wurden unterhalb von diesen noch je drei Druckschalter vorgesehen. Über diese Direktschalter ist es möglich, auf die relevantesten Funktionen des Systems zuzugreifen, egal in welchem Untermenü man sich derzeit befindet. Die Taste “Display“ soll ein Ein- & Ausschalten des Displays im Betrieb ermöglichen, um dieses zu schonen. So muss nicht zwingend das Display aktiv sein, wenn das Gesamtsystem unbeaufsichtigt seinen störungsfreien Dienst verrichtet.

Die Taste “Home“ soll es ermöglichen, auf die Startseite des Systems zu gelangen, egal in welchem Untermenü man sich befindet. Die letzte und wichtigste Taste “STOP System“ soll in beiden Teilsystemen einen “System-Stop“ per Software auslösen. Somit ist es während des Betriebs möglich, die Systeme im vom Benutzer bemerkten Fehlerfall sofort anzuhalten, ohne die Hauptschalter betätigen zu müssen. Der Not-Aus-Schalter wurde aufgrund der Netzschalter weggelassen, da diese die Versorgungsspannung in dem Gerät bereits sicher trennen. Ein zusätzlicher Not-Aus-Schalter an dem Gerät hätte einen Mehraufwand in der Verkabelung im Innenraum bedeutet, ohne hier einen deutlichen Vorteil bei der Sicherheit zu bringen. Die Gehäuselüfter wurden ebenfalls verändert, statt fünf mit Gleichstrom gibt es nur noch zwei mit 230V Wechselspannung betriebene Gehäuselüfter auf der Vorderseite. Hier wurden durch diese Änderung zusätzliche fehleranfällige Schaltungsteile eingespart und die Luftleistung wurde durch die vergrößerten und leistungsstärkeren Lüfter verbessert. Das Leistungsnetzteil wurde aus Platzaufteilungsgründen im Innenraum, in die Mitte verlegt. Eine weitere deutliche Änderung ist die große Aussparung für den Mess- und Batterieanschlußbereich. Für die bessere Weiterentwicklung in zukünftigen Projekten, wurde diese mit einer großen Pertinaxplatte versehen. Damit werden Änderungen deutlich vereinfacht und zerstören nicht die teure Frontplatte, da keine weiteren Löcher in diese gebohrt werden müssen, um neue Ideen mit dem Endgerät zu realisieren. Auch Isolationsprobleme wurden durch die Verwendung der Pertinaxplatte gelöst. Insgesamt wurden die Vorteile der drei zuvor vorgestellten Designs kombiniert, daher wirkt das Design mit diesen Änderungen minimalistischer und ist für die zukünftige Weiterentwicklung im BATSEN Projekt bestens gerüstet.

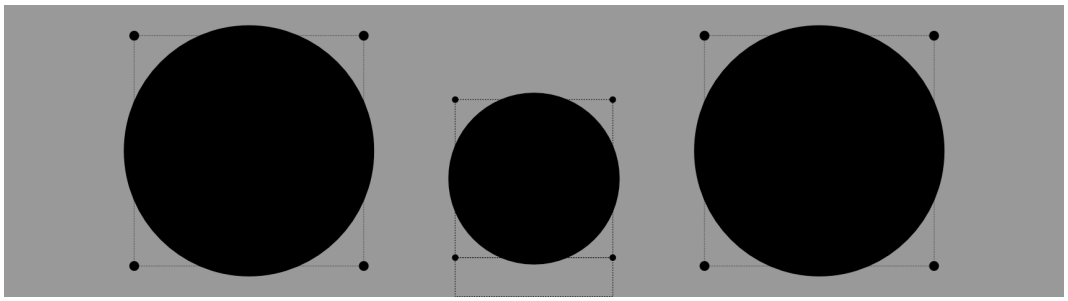


Abbildung 3.3.: Frontplatte: CAD Entwurf Endprodukt 3HE (unten)

3.1.4. Frontplatte (Endprodukt) - zukünftige Erweiterungsmöglichkeiten

Die große Aussparung mit hinterlegter Pertinaxplatte lässt noch viel Platz für weitere Anschlüsse. Weitere Möglichkeiten für einen späteren Komplettaufbau mit Zykliersystem wären:

- RS232 (universelle Schnittstelle für diverse Zusatzgeräte)
- RS485 (Ansteuerung Temperaturschrank)
- diverse Messpunkte für externe Messgeräte (Bananenstecker)
- 1-wire Stecker für die Überwachung der Batterietemperatur

3.2. Gehäuse

Hier werden die Vorüberlegungen für die Aufteilung und das Design des Gehäuses und die Innenraumaufteilung vorgestellt. Aufgrund der Größe und Anzahl der unterzubringenden Bauteile, wurde sich für ein 19 Zoll Rack-Element mit 9 HE (Höheneinheiten) entschieden.

3.2.1. Gesamtaufbau CAD Entwurf

Eine detaillierte Beschriftung der Komponenten befindet sich in Abb. 3.10, 3.11 und 3.12.

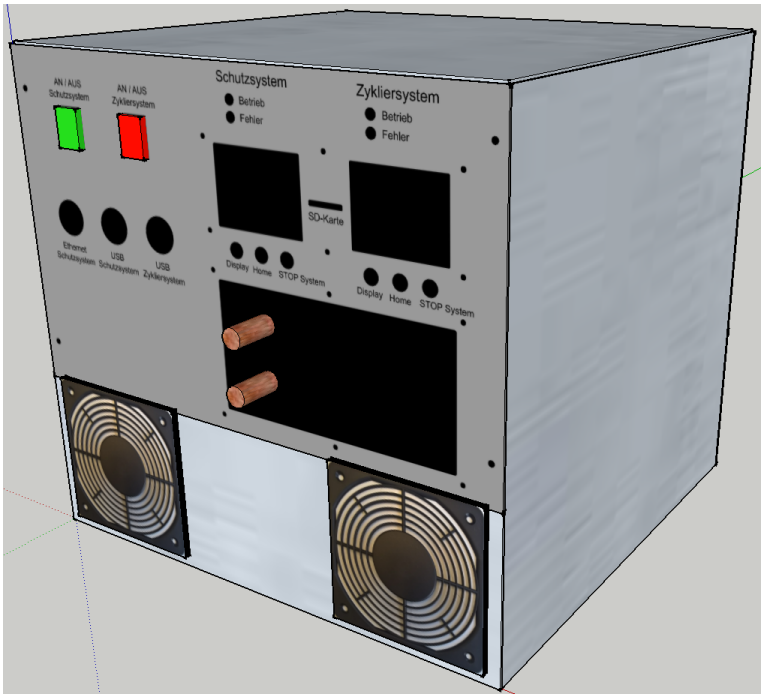


Abbildung 3.4.: CAD Entwurf - Vorderseite

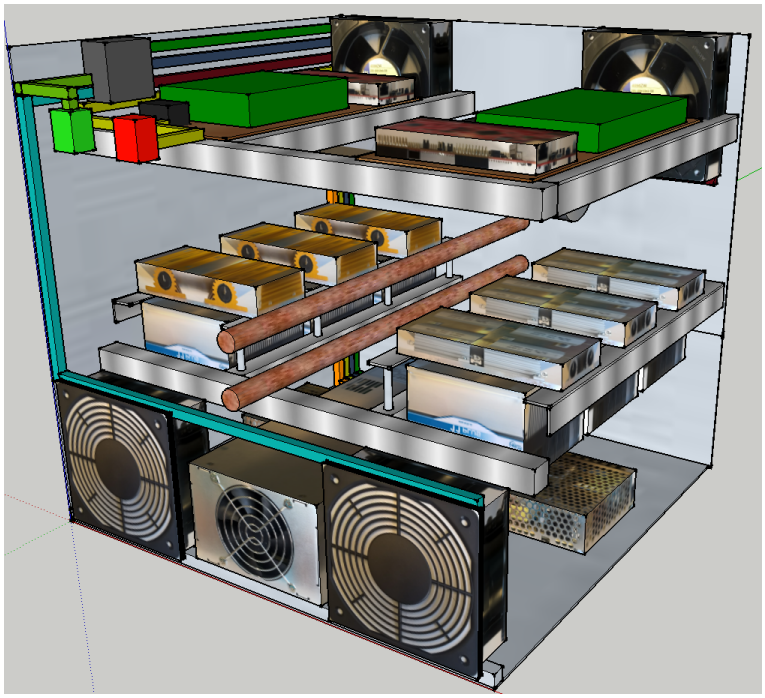


Abbildung 3.5.: CAD Entwurf - Vorderseite offen

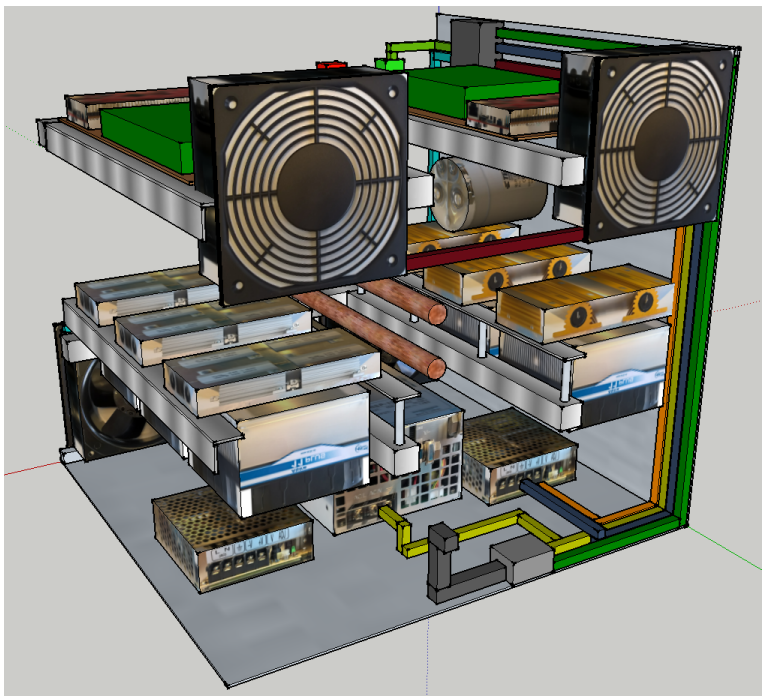


Abbildung 3.6.: CAD Entwurf - Rückseite

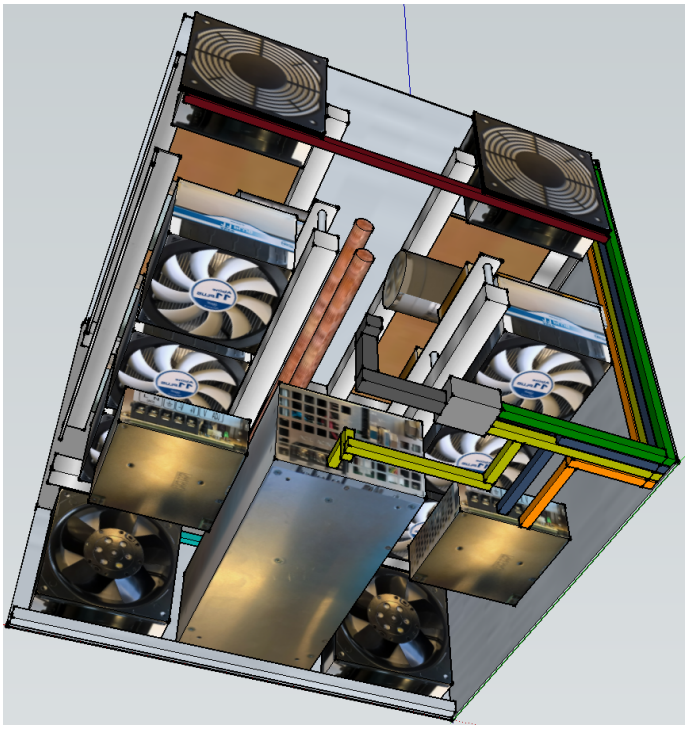


Abbildung 3.7.: CAD Entwurf - Rückseite unten

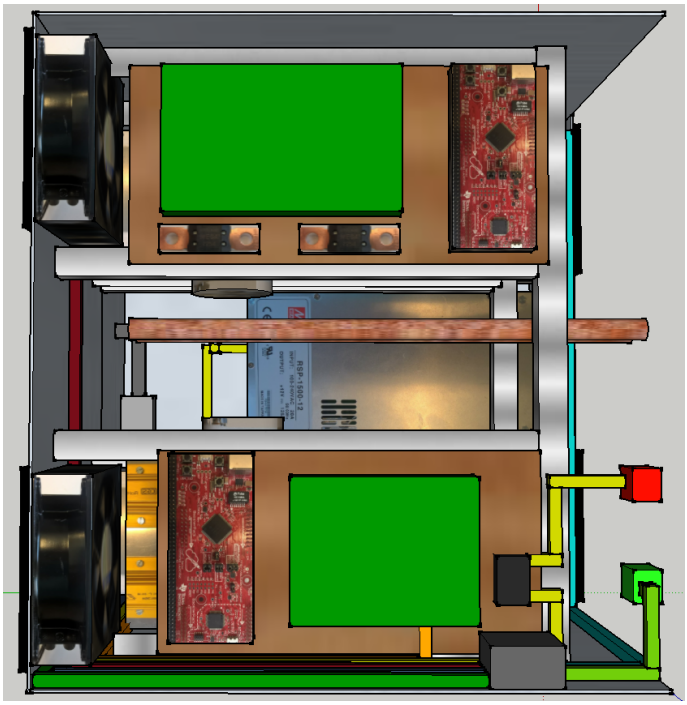


Abbildung 3.8.: CAD Entwurf - Ansicht obere Ebene

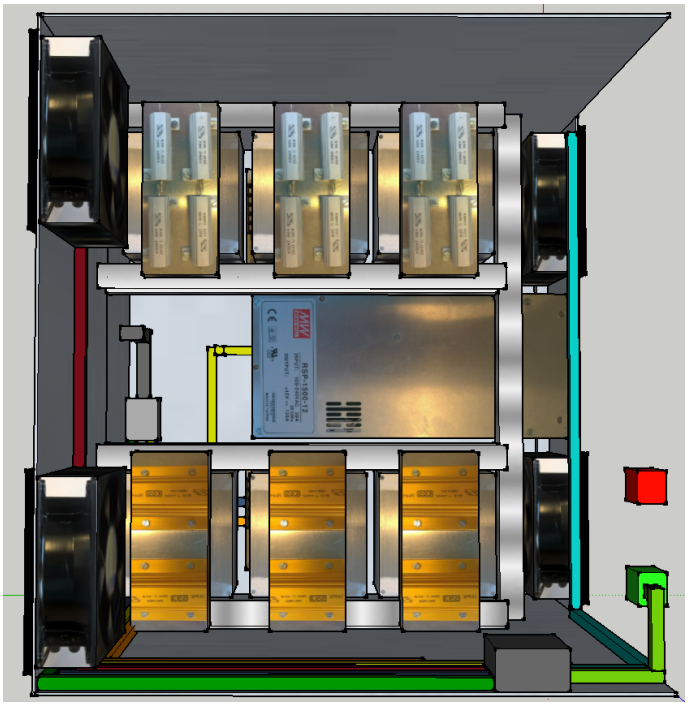


Abbildung 3.9.: CAD Entwurf - Ansicht mittlere Ebene

Verkabelung der Versorgungsspannung

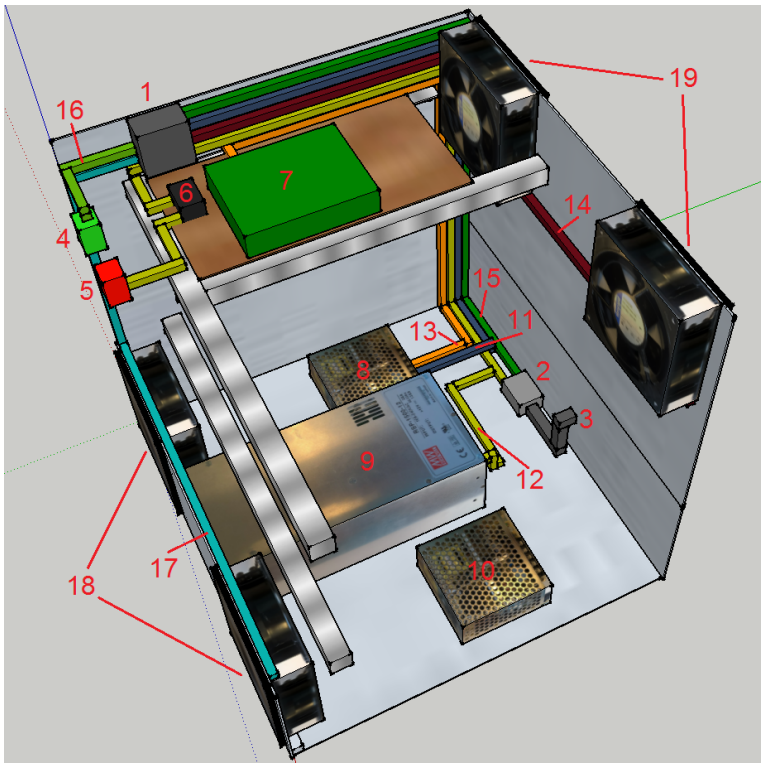


Abbildung 3.10.: Verkabelung der Versorgungsspannung

Die Versorgungsspannungen (Netzspannungskabel) sind in dem Endgerät an den Innenwänden entlanggeführt. So ist eine optische und räumliche Trennung von Netz- und Schaltungskabeln gewährleistet (siehe Abb. 3.10). Das Endgerät ist mittels eines 32A Drehstromkabels an die Steckdose angeschlossen. Das Drehstromkabel(3) ist an eine Verteilerschiene(2) angeschlossen, an der in diesem Aufbau nur L1 und L2 des Drehstromnetzes weitergeführt und verwendet werden. An die Verteilerschiene(2) sind an L2(15) das Nachlauf-Relais(1) und an L1(12) über ein Relais(6) und den Zykliersystem-Netzschalter(5) das Leistungsnetzteil(9) angeschlossen. L3 ist ungenutzt, soll aber später der Versorgung des Zykliersystem-Netzteils(10) dienen. Der Schutzsystem-Netzschalter(4) aktiviert das Nachlauf-Relais(1), welches sofort über die Leitungen (11,14,17) das Schutzsystem-Netzteil(8) und die Front-(18) & Rücklüfter(19) mit Spannung versorgt. Wird der Schutzsystem-Netzschalter(4) deaktiviert, ist die Versorgungsspannungsleitung(11) des Schutzsystem-Netzteils(8) sofort getrennt, während die Front-(18) & Rücklüfter(19) über die Leitungen(18,19) noch eine kurze Zeit weiter versorgt werden. Das Leistungsnetzteil(9) wird über Leitung-L2(12) mittels Zykliersystem-Netzschalter(5) und Relais(6) aktiviert. Das

Relais(6) schaltet nur bei aktiver Schutzsystem-Schaltung(7), daher ist ein Einschalten des Zykliersystems nur bei bereits aktivem Schutzsystem möglich.

Nr	Farbe	Eigenschaften
1	dunkelgrau	Verteilerdose mit Nachlauf-Relais
2	grau	Verteilerschiene Drehstromnetz
3	dunkelgrau	32A Drehstromkabel
4	hellgrün	Netzschalter Schutzsystem
5	hellrot	Netzschalter Zykliersystem
6	schwarz	Relais (Wird von Schutzsystem-Schaltung aktiviert)
7	grün	Schutzsystem-Schaltung
8	-	Netzteil Schutzsystem
9	-	Leistungsnetzteil Zykliersystem
10	-	Netzteil Zykliersystem
11	graublau	Versorgungsspannung 230V aus Nachlauf-Relais für Schutzsystem-Netzteil
12	gelb	Versorgungsspannung Leistungsnetzteil
13	orange	Versorgungsspannung DC Schutzsystem-Schaltung
14	dunkelrot	Versorgungsspannung 230V aus Nachlauf-Relais für Lüfter (Rückseite)
15	dunkelgrün	Versorgungsspannung 230V für Nachlauf-Relais
16	mintgrün	Schalter 230V für Nachlauf-Relais
17	türkis	Versorgungsspannung 230V aus Nachlauf-Relais für Lüfter (Vorderseite)
18	-	Lüfter (Vorderseite)
19	-	Lüfter (Rückseite)

Tabelle 3.1.: Verkabelung der Versorgungsspannung Abb. 3.10

Verkabelung der Hochstromkabel

Die Stromschienen, welche in dem Endgerät eine runde Form haben, sind in der Mitte des Endgerätes verbaut worden. Ein Großteil der inneren Verkabelung muss mit diesen Stromschienen direkt oder indirekt verbunden werden, sodass durch die mittige Position die sparsamste und sinnvollste Verkabelung der Komponenten möglich ist.

Die in Abb. 3.11 und Abb. 3.12 gezeigten Verschaltungen, sind für ein späteres Aufbauen mit fertigem Zykliersystem. Da dies zum Zeitpunkt dieser Arbeit noch nicht fertig war, ist die aufgezeigte Zusammenschaltung ein Grundkonzept für den späteren Gesamtaufbau.

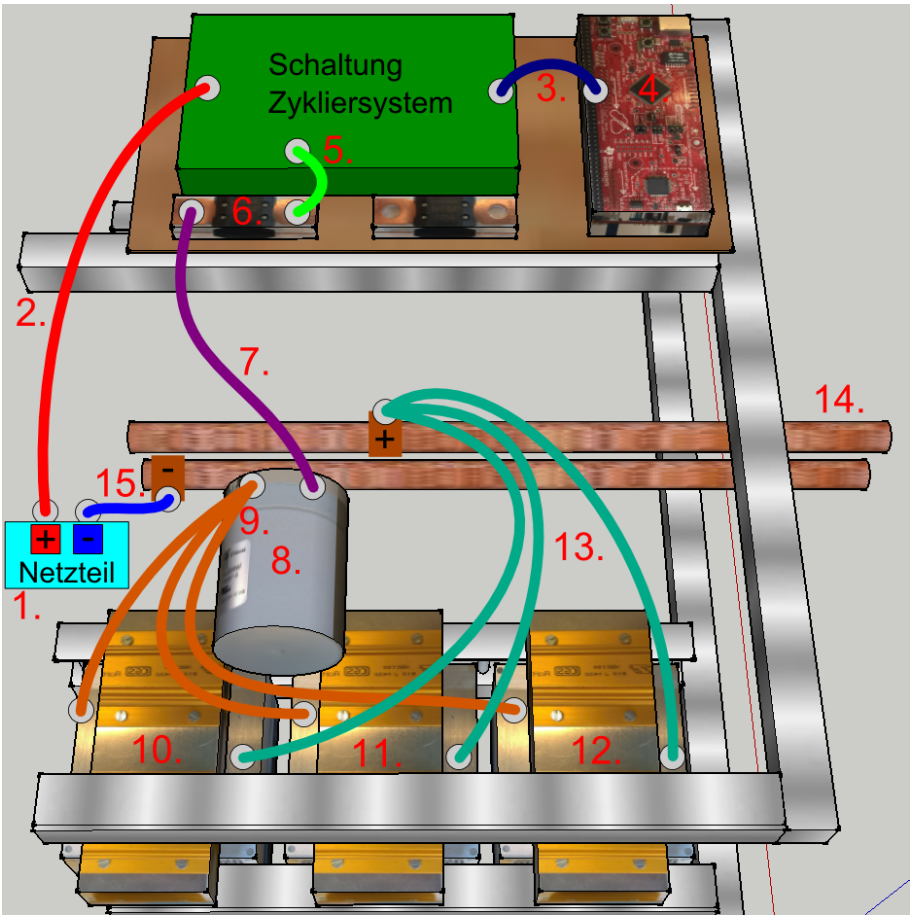


Abbildung 3.11.: Ladekreis Verkabelung

Nr	Farbe	Eigenschaften
1	helltürkis	Leistungsnetzteil
2	rot	Kabelverbindung zwischen Leistungsnetzteil und Zykliersystem-schaltung
3	dunkelblau	Kabelverbindung zwischen Zykliersystemschaltung und Mikrocontroller-Board
4	-	Mikrocontroller-Board
5	hellgrün	Kabelverbindung zwischen Zykliersystemschaltung und Schmelz-sicherung
6	-	Schmelzsicherung
7	lila	Kabelverbindung zwischen Schmelzsicherung und Leistungsre-lais (an diesem Kabel sollte ein Hallsensor zur Messung des La-destromes angebracht werden)
8	-	Leistungsrelais
9	orange	Kabelverbindungen (3x) zwischen Leistungsrelais und Ladewider-ständen
10	-	Ladewiderstand (2 Widerstände Parallelschaltung)
11	-	Ladewiderstand (2 Widerstände Parallelschaltung)
12	-	Ladewiderstand (2 Widerstände Parallelschaltung)
13	türkis	Kabelverbindungen (3x) zwischen Ladewiderständen und Strom-schiene
14	-	Stromschiene
15	blau	Kabelverbindung zwischen Stromschiene und Leistungsnetzteil

Tabelle 3.2.: Ladekreis Verkabelung Abb. 3.11

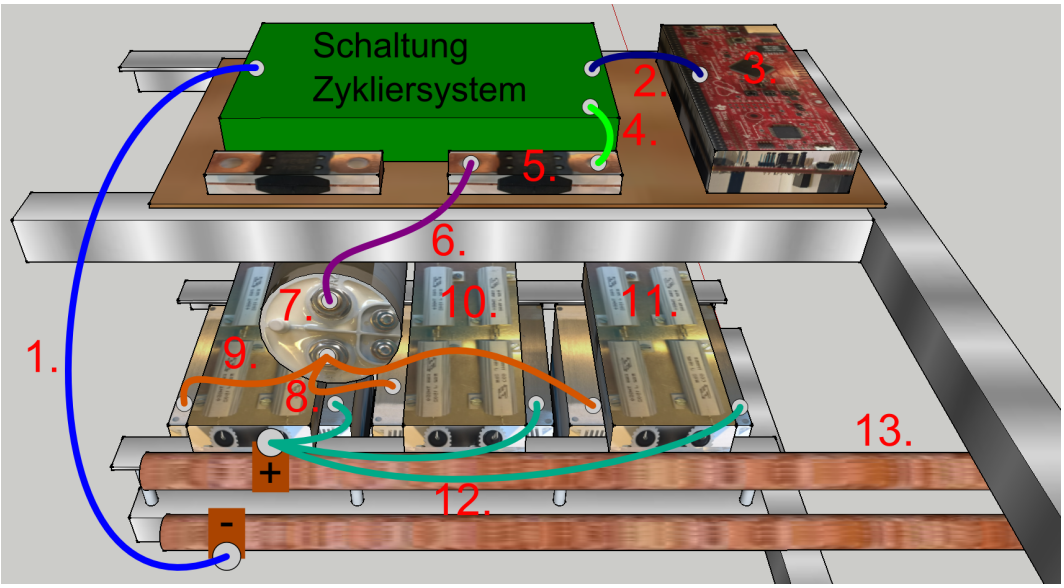


Abbildung 3.12.: Lastkreis Verkabelung

Nr	Farbe	Eigenschaften
1	blau	Kabelverbindung zwischen Stromschiene und Zykliersystem-schaltung
2	dunkelblau	Kabelverbindung zwischen Zykliersystemschaltung und Mikrocontroller-Board
3	-	Mikrocontroller-Board
4	hellgrün	Kabelverbindung zwischen Zykliersystemschaltung und Schmelz-sicherung
5	-	Schmelzsicherung
6	lila	Kabelverbindung zwischen Schmelzsicherung und Leistungsre-lais (an diesem Kabel sollte ein Hallsensor zur Messung des La-destromes angebracht werden)
7	-	Leistungsrelais
8	orange	Kabelverbindungen (3x) zwischen Leistungsrelais und Lastwider-ständen
9	-	Lastwiderstand (4 Widerstände Serienschaltung)
10	-	Lastwiderstand (4 Widerstände Serienschaltung)
11	-	Lastwiderstand (4 Widerstände Serienschaltung)
12	türkis	Kabelverbindungen (3x) zwischen Lastwiderständen und Strom-schiene
13	-	Stromschiene

Tabelle 3.3.: Lastkreis Verkabelung Abb. 3.12

3.2.2. Luftleistung

Für das Gehäuse wurden vier Lüfter vorgesehen, von den zwei die Luft in und zwei die Luft aus dem Gehäuse transportieren sollen. Diese schaffen pro Lüfterpaar einen Volumenstrom von $160 m^3/h$ durch das Gehäuse. Dies ergibt einen Gesamtvolumenstrom von $320 m^3/h$. Für die Kühlung der Widerstände sind CPU-Kühler vorgesehen, dessen Lüfter jeder je nach steuerbarer Umdrehungszahl bis zu $50 m^3/h$ an Volumenstrom auf den Kühlkörper schaffen. Würden alle sechs CPU-Kühler-Lüfter gleichzeitig mit voller Last laufen wäre, dies ein benötigter Gesamtvolumenstrom von ca. $300 m^3/h$. Obwohl dieser Fall im Normalbetrieb niemals auftreten sollte, würde der Volumenstrom an frischer Außenluft zu dem benötigten Volumenstrom für die Kühlung der Bauteile passen.

3.3. Verwendete Bauteile

3.3.1. Mikrocontroller Evaluation Board

Abbildung 3.13.: TM4C1294XL [4]

Das zentrale Steuerelement des Schutzsystems ist ein “TM4C1294XL Connected LaunchPad [4]“ von Texas Instruments. Die darauf verbaute TM4C1294NCPDT Mikrocontroller-Einheit besteht aus einer 120MHz 32-bit ARM Cortex-M4 CPU mit 1MB Flash, 256KB SRAM, 6KB EEPROM, integrierter 10/100 Ethernet Schnittstelle, acht 32-bit Timern, dual 12-bit 2MSPS ADCs, motion control PWMs, USB H/D/O und vielen weiteren seriellen Kommunikationsschnittstellen.

3.3.2. Nachlauf-Relais für die Gehäuselüfter

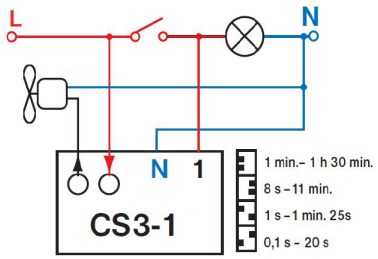


Abbildung 3.14.: Nachlauf-Relais

Für die Gehäuselüfter wurde ein Nachlauf-Relais (Abb. 3.14) eingesetzt. Dies ist ein Bauteil, dass sonst für die Belüftung von Badezimmern eingesetzt wird. Es ist direkt mit dem Hauptschalter des Schutzsystems verbunden. Wird der Hauptschalter eingeschaltet, werden das Netzteil des Schutzsystems und die Gehäuselüfter mit Strom versorgt. Nach dem Ausschalten wird der Strom von dem Netzteil des Schutzsystems sofort getrennt, während die Gehäuselüfter durch das Nachlauf-Relais noch einen einstellbaren Zeitbereich weiter versorgt werden. Dadurch laufen die Gehäuselüfter noch einen Zeitraum nach dem Ausschalten des Gerätes nach, um die Restwärme aus dem Gehäuseinneren zu transportieren.

3.3.3. Temperatursensoren (1-wire)

Verwendet wurden 1-wire Temperatursensoren der Firma Dallas. Da der verwendete Mikrocontroller keine integrierte 1-wire Schnittstelle besitzt, wurde unter Verwendung des Tutorials “Tutorial 214: Using a UART to Implement a 1-Wire Bus Master“ [6] eine 1-wire Schittstelle unter Verwendung der UART Schnittstelle emuliert.

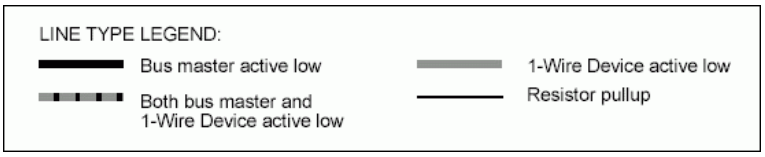


Abbildung 3.15.: 1-wire: Legende [6]

Um über eine UART Schnittstelle eine 1-wire Schnittstelle zu implementieren, müssen via UART Hexadezimalziffern übertragen und empfangen werden und dazu passende Zeitfenster eingehalten werden. Dies wurde nach der Vorgabe [6] von der Firma Maxim Integrated

implementiert. Die dafür vorgesehene Schaltung übersetzt die Signale von UART zu 1-wire und umgekehrt.

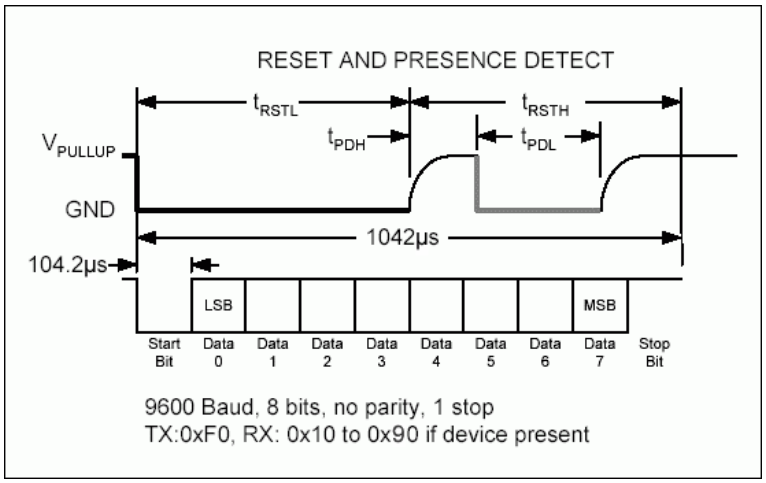


Abbildung 3.16.: 1-wire: Reset-Puls [6]

Der Puls in Abb. 3.16 wird erzeugt, um einen Reset der 1-wire Slaves zu erreichen. Dazu wird die UART Schnittstelle auf 9600 Baud gestellt und der Hexadezimalwert 0xF0 wird übertragen. Am 1-wire Ausgang entsteht dadurch der Reset-Puls, welcher den 1-wire Kommunikationskanal eine Zeit t_{RSTL} auf Masse zieht. Durch diesen Vorgang werden alle an dem 1-wire Kanal angeschlossenen Slaves zurückgesetzt.

Danach wird die UART Schnittstelle mit 115200 Baud weiter verwendet, da die langsamere Einstellung nur für den Reset-Puls benötigt wird. Die restliche Kommunikation läuft über die schnellere Geschwindigkeit.

Die weitere Kommunikation läuft zwar mit 115200 Baud, jedoch wird hier immer jeweils eine Hexadezimalzahl für 0 oder 1 gesendet. Die zu übertragenden Daten müssen daher binär aufgeschlüsselt und mittels bestimmter hexadezimaler Muster übertragen werden. Auch beim Empfang von Daten kommen diese nur als Hexadezimalmuster an, welche auch jeweils für 0 oder 1 uminterpretiert werden müssen.

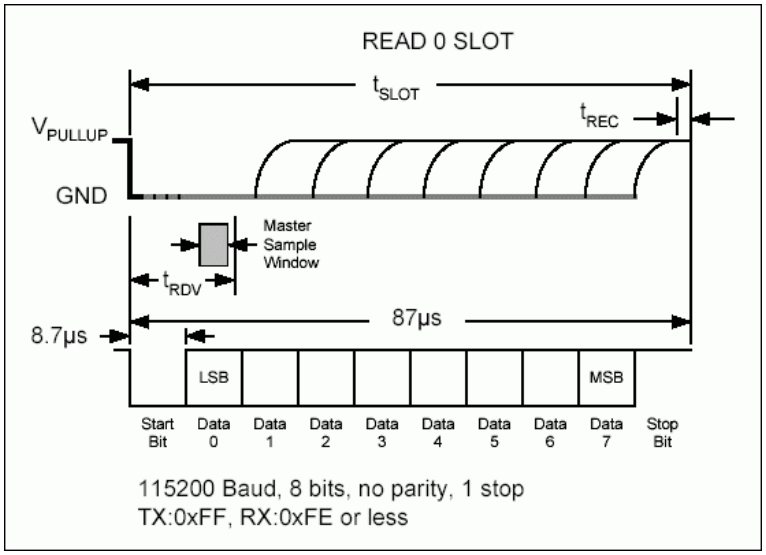


Abbildung 3.17.: 1-wire: Read 0 Slot [6]

In Abb. 3.17 ist der Sende-/Empfangspuls für das Einlesen einer logischen 0 zu sehen. Der Mikrocontroller sendet über die UART Schnittstelle (TX) die Hexadezimalzahl 0xFF an den 1-wire Bus und erhält als Antwort des 1-wire Busses an die UART Schnittstelle (RX) die Hexadezimalzahl 0xFE. Es wurde ein Lesebefehl gesendet, die Antwort war eine logische 0.

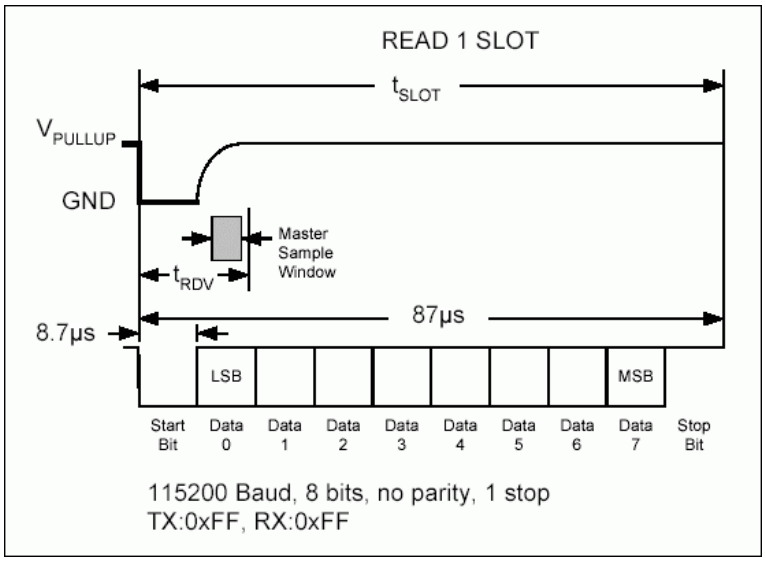


Abbildung 3.18.: 1-wire: Read 1 Slot [6]

In Abb. 3.18 ist der Sende-/Empfangspuls für das Einlesen einer logischen 1 zu sehen. Der Mikrocontroller sendet über die UART Schnittstelle (TX) die Hexadezimalzahl 0xFF an den 1-wire Bus und erhält als Antwort des 1-wire Busses an die UART Schnittstelle (RX) die Hexadezimalzahl 0xFF. Es wurde ein Lesebefehl gesendet, die Antwort war eine logische 1.

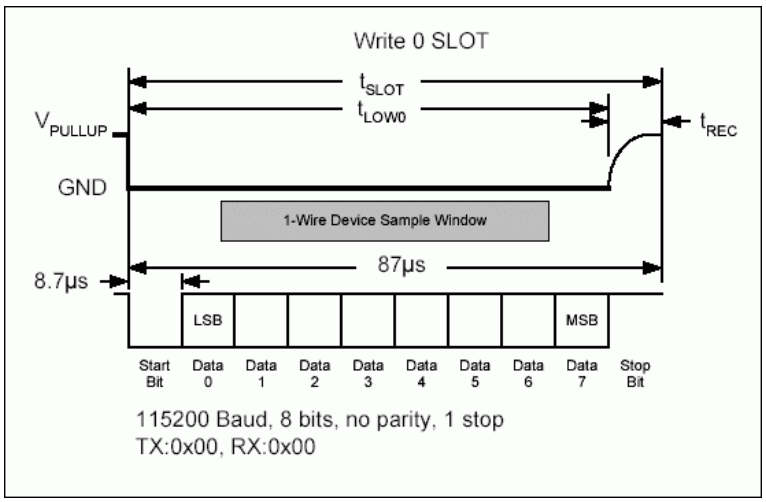


Abbildung 3.19.: 1-wire: Write 0 Slot [6]

In Abb. 3.19 ist der Sende-/Empfangspuls für das Schreiben einer logischen 0 zu sehen. Der Mikrocontroller sendet über die UART Schnittstelle (TX) die Hexadezimalzahl 0x00 an den 1-wire Bus und erhält als Antwort des 1-wire Busses an die UART Schnittstelle (RX) die Hexadezimalzahl 0x00. Die Antwort war identisch, da die 1-wire Slaves hier nicht antworten und RX und TX über die Übersetzerschaltung miteinander verbunden sind.

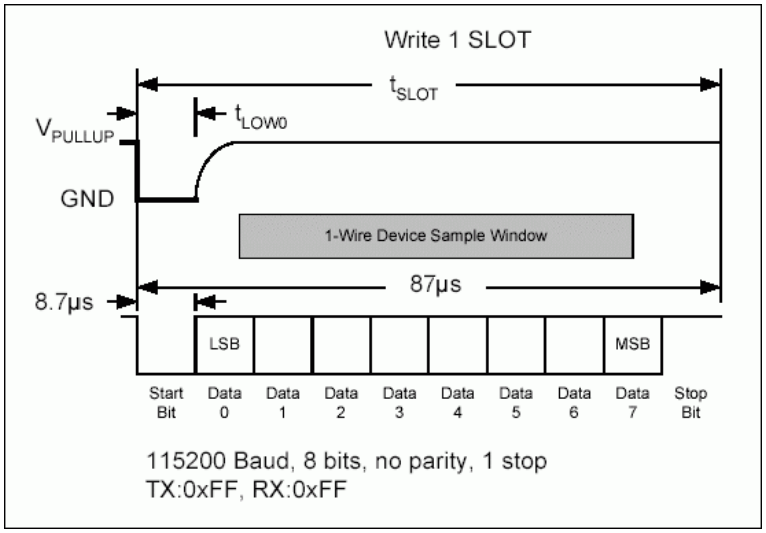


Abbildung 3.20.: 1-wire: Write 1 Slot [6]

In Abb. 3.20 ist der Sende-/Empfangspuls für das Schreiben einer logischen 1 zu sehen. Der Mikrocontroller sendet über die UART Schnittstelle (TX) die Hexadezimalzahl 0xFF an den 1-wire Bus und erhält als Antwort des 1-wire Busses an die UART Schnittstelle (RX) die Hexadezimalzahl 0xFF. Die Antwort war identisch, da die 1-wire Slaves hier nicht antworten und RX und TX über die Übersetzerschaltung miteinander verbunden sind.

3.3.4. Hall-Sensoren

Es wurde der Hallsensor DHAB S/25 der Firma LEM verwendet. Dieser benötigt eine Versorgungsspannung von 5 Volt und arbeitet daher auch in dem Spannungsbereich von 0 bis 5 Volt. Aus diesem Grund muss eine Spannungsteilerschaltung verwendet werden, welche die Betriebsspannung am Analogausgang auf die vom Mikrocontroller benötigten 3,3 Volt anpasst. Der Hallsensor hat zwei verschiedene Kanäle -75A/+75 Ampere und -500A/+500 Ampere.

3.3.5. PWM gesteuerte Lüfter

Bei den verwendeten Lüftern handelt es sich um handelsübliche CPU Lüfter. Diese verfügen über einen 4-adrigen Anschluss und sind bereits auf einen passenden Kühlkörper verbaut. Die 4 Anschlüsse eines Lüfters sind:

- +12 Volt Versorgungsspannung

- GND
- PWM Steuersignal
- PWM Tachosignal

PWM Steuersignal

Die Lüfter sind PWM gesteuert. Die Frequenz des Steuersignals liegt bei 25 kHz. Die Lüfter lassen sich in einem Bereich von 1000 - 2000 RPM steuern. Bei einem Tastgrad von 0 (PWM komplett aus) drehen die Lüfter mit 1000 RPM.

PWM Tachosignal

Das Tachosignal dient als Rückgabekanal eines Lüfters. Hier wird pro Umdrehung ein Rechteckpuls mit 2 Takten und einem Tastgrad von 50% ausgegeben. Aus diesem wird später die aktuelle Umdrehungszahl berechnet.

3.3.6. Touch-Display

In dem Gerät sind zwei Touch-Displays verbaut. Ein Display für das Schutzsystem und eins für das Zykliersystem. In Bezug auf die Aufgabenstellung dieser Bachelorarbeit wurde nur das Display für das Schutzsystem implementiert.

3.3.7. Relais

Die Relais in dem Gerät dienen der sicheren Abschaltung. Es wurden 2 Leistungsrelais verbaut, ein Relais im Ladekreis und eins im Lastkreis. Zu beachten ist bei der Verwendung von Relais der Widerstand bei geschlossenem Zustand, denn die verwendeten niederohmigen Widerstände der Zyklierschaltung sind bereits sehr klein und sollten durch die Relais in Ihrer Funktion nicht beeinträchtigt werden. Der dadurch auftretende Fehler wäre ein zu großer Widerstand im Lade- oder Lastkreis und würde einen geringeren Lade- bzw. Entladestrom zur Folge haben. Dies würde eine der zentralen Eigenschaften des neuen Systems verschlechtern und hat somit eine besondere Bedeutung, denn hier ist eine der wenigen direkten Verschaltungen von Schutz- und Zykliersystem gegeben.

Zur Ansteuerung der großen Leistungsrelais, wurde eine fertige Relaisschaltung verwendet. Diese Versorgt die großen Relais mit den benötigten 12 Volt.

3.3.8. Gehäuselüfter

In dem Gehäuse sind 4 Lüfter für die Luftkühlung verbaut. Zwei an der Vorderseite und zwei auf der Rückseite. Diese laufen mit 230V und werden durch Einschalten des Schutzsystems aktiviert. Ein Nachlauf-Relais lässt die Lüfter nach Abschalten des Schutzsystems noch eine kurze Zeit weiterlaufen.

3.3.9. Stromschienen

In im Laufe der Vorüberlegungen wurde sich für die Verwendung von Stromschienen entschieden. Aufgrund der hohen Ströme, die das Leistungsnetzteil liefert oder der zu verbrauchenden Rückströme, welche die Batterie zum Verbrauchen in das Gerät leitet. Von daher wird im Inneren des Gerätes ein sicherer Anschluss benötigt. Eine Stromschiene liefert eine sichere Möglichkeit, die einzelnen Komponenten miteinander zu verbinden. So ist im Fehlerfall gewährleistet, dass zum Beispiel ein überhitztes Kabel nur einen möglichst kleinen Teil des Gesamtaufbaus betrifft. Die Stromschiene ist aufgrund ihrer Dicke bezüglich der Überdimensionierung der sichere zentrale Teil der gesamten Verdrahtung.

Im ersten Entwurf wurde die Stromschiene als Rechteckleiste vorgesehen. Hier sollten die einzelnen Anschlüsse jeweils mit Bohrungen und Ringkabelanschlüssen realisiert werden. Aufgrund der damit stark begrenzten Änderungsmöglichkeiten und der Tatsache, dass die gebohrten Löcher eine Verringerung des Querschnittes der Stromschiene bedeuten, wurde sich gegen eine rechteckige Lösung entschieden.

Es wurde sich für die Verwendung von runden Stromschienen entschieden, da diese bessere Eigenschaften für einen Einbau und die Anschlussmöglichkeiten der Bauteile liefern. Die runden Stromschienen haben einen Durchmesser von 16mm, denselben Durchmesser der Anschlüsse von handelsüblichen Autobatterien. Durch diese Wahl können für jeden Anschluss die Standard-Batteriepolklemmen aus dem KFZ-Bereich verwendet werden. Diese eignen sich bereits für sehr hohe Ströme und es müssen nicht für jeden Anschluss Löcher in die Stromschiene gebohrt werden. Auch ein späteres Versetzen von Anschlüssen ist mit dieser Kombination problemlos möglich. Die auf der Frontseite herausgeführten Stromschienen können ebenfalls mittels Batteriepolklemmen verbunden werden. Da diese Anschlüsse an der Vorderseite in Zukunft sehr häufig an- und abgeschraubt werden, um die Batterien anzuschließen, ist eine Verbindung über die Standard-Polklemmen für das Material besonders schonend. Die Endstücke nutzen bei häufigem Gebrauch daher nicht so schnell ab.

3.4. Kommunikation zwischen Schutzcontroller & Zykliercontroller

Die Kommunikation der beiden Mikrocontroller läuft über einen UART Kanal und zwei GPIO Pins.

Ein GPIO Pin dient als Freigabe-Pin. Wird der Pin vom Schutzsystem auf logisch 0 gezogen, muss das Zykliersystem sofort reagieren. Hier wird der Zustand des Pins vom Schutzsystem vorgegeben.

Ein GPIO Pin dient als Rückmelde-Pin. Wird der Pin vom Zykliersystem auf logisch 0 gezogen, muss das Schutzsystem reagieren. Hier wird der Zustand des Pins vom Zykliersystem vorgegeben.

Die UART Schnittstelle des Schutzcontrollers überträgt immer nur eine vorgegebene Zeichenkette an den Zykliercontroller. Dies belastet beide Controller möglichst wenig, denn die Bedeutung der Warn- & Fehlercodes der Zeichenkette ist beiden Controllern gleichermaßen bekannt. Auch die Werte, welche die Warnungen oder Fehler ausgelöst haben, stecken mit in der Zeichenkette.

Alle Verbindungen zwischen den Mikrocontrollern sind über Optokoppler geführt, um eine galvanische Trennung beider Systeme zu gewährleisten.

3.5. Batteriespannung - Ober- & Untergrenze

Die Batteriespannung soll mit einem ADC überwacht werden. Die Werte für diese Grenzen sollen einstellbar oder auch komplett abschaltbar sein.

Für den Fall des sicheren Betriebes soll die Batteriespannung über Grenzen definierbar sein. Dies ermöglicht eine sichere Zyklierung, ohne dass die Batterie hierbei Schaden nimmt. Die Grenzen sollten mit denen des Zykliersystems vor dessen Betrieb verglichen werden. Die Untergrenze sollte etwas unter der des Zykliersystems liegen, die Obergrenze etwas oberhalb der des Zykliersystems. Somit werden ungewollte Abschaltungen seitens des Schutzcontrollers vermieden.

Für den gewollten Fall der Batteriezerstörung, wie es in Laborbedingungen durchaus vorkommen kann, sollen die Spannungsgrenzen auch komplett abschaltbar sein. Dadurch würde es für den Zykliercontoller auch möglich sein, die Batterie im Grenzbereich zu bearbeiten.

3.6. Fehlerfälle und Maßnahmen

Hier werden mögliche Warn- und Fehlerfälle aufgezählt und die beiden damit verbundenen Gegenmaßnahmen erläutert.

3.6.1. Warnungsfälle

Sind die Werte im System erhöht, soll der laufende Betrieb nicht unterbrochen und die betreffenden Warnungen über die UART Schnittstelle an den Zykliercontroller übertragen werden.

- Der Temperatursensor meldet einen erhöhten Wert.
- Der Hallsensor im Lade- oder Entladekreis misst einen Strom, der die Warngrenze überschreitet.

3.6.2. Fehlerfälle

Das Verfahren im Fehlerfall soll bei jedem auftretenden Fehler reproduzierbar/identisch sein. So soll der GPIO-Controller-Bereit-Pin gelöscht, eine Meldung mit Art des Fehlers über die UART Schnittstelle an den Zykliercontoller übertragen werden, der Buzzer gibt ein Warnsignal aus und die Fehler-LED (rot) blinkt. Auf dem Display soll dazu die Art des Fehlers erkennbar sein, außerdem sollen die Relais in dem Lade- & Entladekreis geöffnet werden.

- Der Temperatursensor meldet eine Überschreitung des Fehlergrenzwertes.
- Die gesetzten Grenzen der Batteriespannung werden über- bwz. unterschritten.
- Der Lade- oder Entladekreis Hallsensor misst einen Strom, der die Fehlergrenze überschreitet.
- Die Umdrehungen pro Minute (RPM) eines Lüfters sind unter die Mindestdrehzahl gesunken.

4. Realisierung

4.1. FMEA - Fehlermöglichkeits- und -einflussanalyse

Die FMEA (Fehlermöglichkeits- und -einflussanalyse) ist eine analytische Methode der Zuverlässigkeitstechnik. Hier werden Kennzahlen für Produktfehler auf Basis ihrer Auftretenswahrscheinlichkeit, Auswirkungen auf das Endprodukt und Entdeckungswahrscheinlichkeit, vergeben. Diese Kennzahlen dienen als Grundlage für die Berechnung einer Risikoprioritätszahl, welche eine Aussage über das Risiko des Gesamtsystems bezüglich des Fehlers unter Berücksichtigung seiner Eigenschaften und Gegenmaßnahmen trifft [14].

Die Bedeutung der Werte, welche in der FMEA vergeben und berechnet werden, ist in Tab. 4.1 verzeichnet.

4.1.1. FMEA - Zykliersystem mit integriertem Schutzsystem

Die erstellte FMEA bezieht sich auf den derzeitigen Entwicklungsstand und auf die zukünftige Weiterentwicklung, dass heißt in diesem Fall auf das fertige Zykliersystem mit integriertem Schutzsystem. Einige der aufgeführten Fehlermöglichkeiten sind daher bereits mit Kontrollmaßnahmen abgesichert. Es befinden sich ebenfalls potentielle Fehler in der Analyse, welche aufgrund des noch nicht vorhandenen Zykliersystems nur für die Weiterentwicklung des Gesamtsystems berücksichtigt werden sollen. Daher kann diese FMEA auch bei der Weiterentwicklung als Unterstützung dienen, zukünftigen Fehlern vorzubeugen.

A = Auftreten	**B = Bedeutung**	**E = Entdeckung**	**RPZ = Risiko-Prioritäts-Zahl**
Wahrscheinlichkeit des Auftretens unwahrscheinlich = 1 sehr gering = 2 - 3 gering = 4 - 6 mäßig = 7 - 8 hoch = 9 -10	Auswirkungen auf das Endprodukt kaum wahrnehmbar = 1 unbedeutender Fehler = 2 - 3 mäßig schwerer Fehler = 4 - 6 schwerer Fehler = 7 - 8 äußerst schwerer Fehler = 9 -10	Wahrscheinlichkeit der Entdeckung hoch = 1 mäßig = 2 - 3 gering = 4 - 6 sehr gering = 7 - 8 unwahrscheinlich = 9 - 10	hoch <= 1000 mittel <= 250 gering <= 125 kein = 1

Tabelle 4.1.: FMEA Abkürzungen

1	**Fehlerort:**	Software	A = 7
	Potentieller Fehler:	durchschalten aller MOS-FETs	B = 8
	Fehlerfolge:	Strom vom Netzteil fließt direkt über die Last	E = 7
	Fehlerursache:	Softwarefehler	
	Kontrollmaßnahmen:	mit 2 Hallsensoren überwachen	
	Status:	vorbereitet	**RPZ = 392**
2	**Fehlerort:**	elektronisches Bauteil	A = 6
	Potentieller Fehler:	Ausfall oder Fehlfunktion	B = 6
	Fehlerfolge:	Ausfall oder Fehlfunktion des Gesamtsystems	E = 3
	Fehlerursache:	Kurzschluss	
	Kontrollmaßnahmen:	regelmäßige Funktionskontrolle	
	Status:	offen	**RPZ = 108**
3	**Fehlerort:**	Leistungsnetzteil	A = 5
	Potentieller Fehler:	Strom fehlerhaft	B = 9
	Fehlerfolge:	zu hoher Strom	E = 3
	Fehlerursache:	Zerstörung/Brand der Batterie oder von Bauteilen	
	Kontrollmaßnahmen:	Überwachung durch Hallsensoren	
	Status:	umgesetzt	**RPZ = 135**
4	**Fehlerort:**	Batterie	A = 7
	Potentieller Fehler:	Batterieüberhitzung	B = 8
	Fehlerfolge:	Zerstörung/Brand der Batterie	E = 5
	Fehlerursache:	fehlerhafte Spannungsüberwachung	
	Kontrollmaßnahmen:	Überwachung: mit ADC Boosterpack und Temperatursensor	
	Status:	vorbereitet	**RPZ = 280**
5	**Fehlerort:**	Lade- / Entladeschaltung	A = 4
	Potentieller Fehler:	Batteriespannung bleibt an elektronischen Bauteilen trotz Abschaltung des Gerätes	B = 5
	Fehlerfolge:	Zerstörung/Brand von Schaltungsbauteilen	E = 5
	Fehlerursache:	fehlerhafte Zykliersystemschaltung oder Software des Zykliersystems	
	Kontrollmaßnahmen:	Leistungsrelais zum sicheren Abschalten durch Schutzsystem	
	Status:	vorbereitet	**RPZ = 100**

Tabelle 4.2.: FMEA - Teil 1

6	**Fehlerort:**	Messvorrichtung	A = 8
	Potentieller Fehler:	falsche Messwerte	B = 8
	Fehlerfolge:	Zerstörung von Batterie oder Bauteilen oder Fehlfunktionen des Systems	E = 4
	Fehlerursache:	falsche Verschaltung / fehlerhafte Umrechnung	
	Kontrollmaßnahmen:	vorheriger Abgleich mittels Messwerten	
	Status:	offen	**RPZ = 256**
7	**Fehlerort:**	Batterie	A = 2
	Potentieller Fehler:	Überladung der Batterie	B = 8
	Fehlerfolge:	Zerstörung der Batterie	E = 3
	Fehlerursache:	Spannungsmessung Fehler / Softwarefehler	
	Kontrollmaßnahmen:	doppelte Spannungsüberwachung durch Schutz- & Zykliersystem	
	Status:	offen	**RPZ = 48**
8	**Fehlerort:**	Schaltung	A = 4
	Potentieller Fehler:	Störspannungen	B = 7
	Fehlerfolge:	Mess- oder Steuerfehler	E = 9
	Fehlerursache:	unentdeckte Abstrahlungen oder Induktivitäten	
	Kontrollmaßnahmen:	einzelne Komponenten vor Inbetriebnahme testen / Inbetriebnahmetest des Gesamtsystems	
	Status:	offen	**RPZ = 252**
9	**Fehlerort:**	Zykliersystem	A = 5
	Potentieller Fehler:	Fehlbedienung durch Benutzer	B = 1
	Fehlerfolge:	diverse Fehler	E = 2
	Fehlerursache:	Benutzerfehler	
	Kontrollmaßnahmen:	Schutzsystem	
	Status:	teilweise umgesetzt	**RPZ = 10**
10	**Fehlerort:**	Batterie	A = 6
	Potentieller Fehler:	defekte Batterie	B = 2
	Fehlerfolge:	Überhitzung/Brand der Batterie	E = 3
	Fehlerursache:	Batterie zu alt	
	Kontrollmaßnahmen:	Spannungs- / Temperaturüberwachung	
	Status:	vorbereitet	**RPZ = 36**

Tabelle 4.3.: FMEA - Teil 2

11	**Fehlerort:**	Leistungsrelais	A = 5
	Potentieller Fehler:	Relais schaltet nicht mehr	B = 8
	Fehlerfolge:	ständig An/Aus, keine Schutzfunktion mehr	E = 1
	Fehlerursache:	Lebensdauer überschritten	
	Kontrollmaßnahmen:	Anzahl der Schaltzyklen speichern und überwachen	
	Status:	vorbereitet	**RPZ = 40**
12	**Fehlerort:**	Widerstände (Lade- & Last)	A = 7
	Potentieller Fehler:	Temperatur zu hoch	B = 5
	Fehlerfolge:	Widerstandswert verändert sich / oder Hitzeschaden	E = 1
	Fehlerursache:	Lüfter defekt / zu langsam	
	Kontrollmaßnahmen:	Temperatur & Drehzahlüberwachung	
	Status:	umgesetzt	**RPZ = 35**
13	**Fehlerort:**	Lade- oder Lastkreis	A = 4
	Potentieller Fehler:	Strom im Lade- oder Lastkreis zu hoch	B = 8
	Fehlerfolge:	Zerstörung/ Brand von Bauteilen/Batterie	E = 2
	Fehlerursache:	Software oder Schaltungsfehler	
	Kontrollmaßnahmen:	Stromüberwachung mittels Hallsensoren	
	Status:	umgesetzt	**RPZ = 64**
14	**Fehlerort:**	Temperatursensoren auf den Widerständen	A = 6
	Potentieller Fehler:	Temperatur viel höher als Messwert	B = 5
	Fehlerfolge:	Überhitzung, da RPM Ansteuerung zu gering	E = 2
	Fehlerursache:	Sensor löst sich vom Messobjekt	
	Kontrollmaßnahmen:	Sensoren mittels Klemmen festschrauben	
	Status:	umgesetzt	**RPZ = 60**
15	**Fehlerort:**	[EXTERN] Temperaturschrank	A = 2
	Potentieller Fehler:	Ausfall Temperaturschrank	B = 3
	Fehlerfolge:	Zykliervorgang / Messung fehlerhaft	E = 1
	Fehlerursache:	Ausfall oder Fehler des Temperaturschranks, der die zu zykliererenden Batterien enthält	
	Kontrollmaßnahmen:	Kommunikation mittels RS485 Schnittstelle	
	Status:	offen	**RPZ = 6**

Tabelle 4.4.: FMEA - Teil 3

4.2. Pin Belegung MC

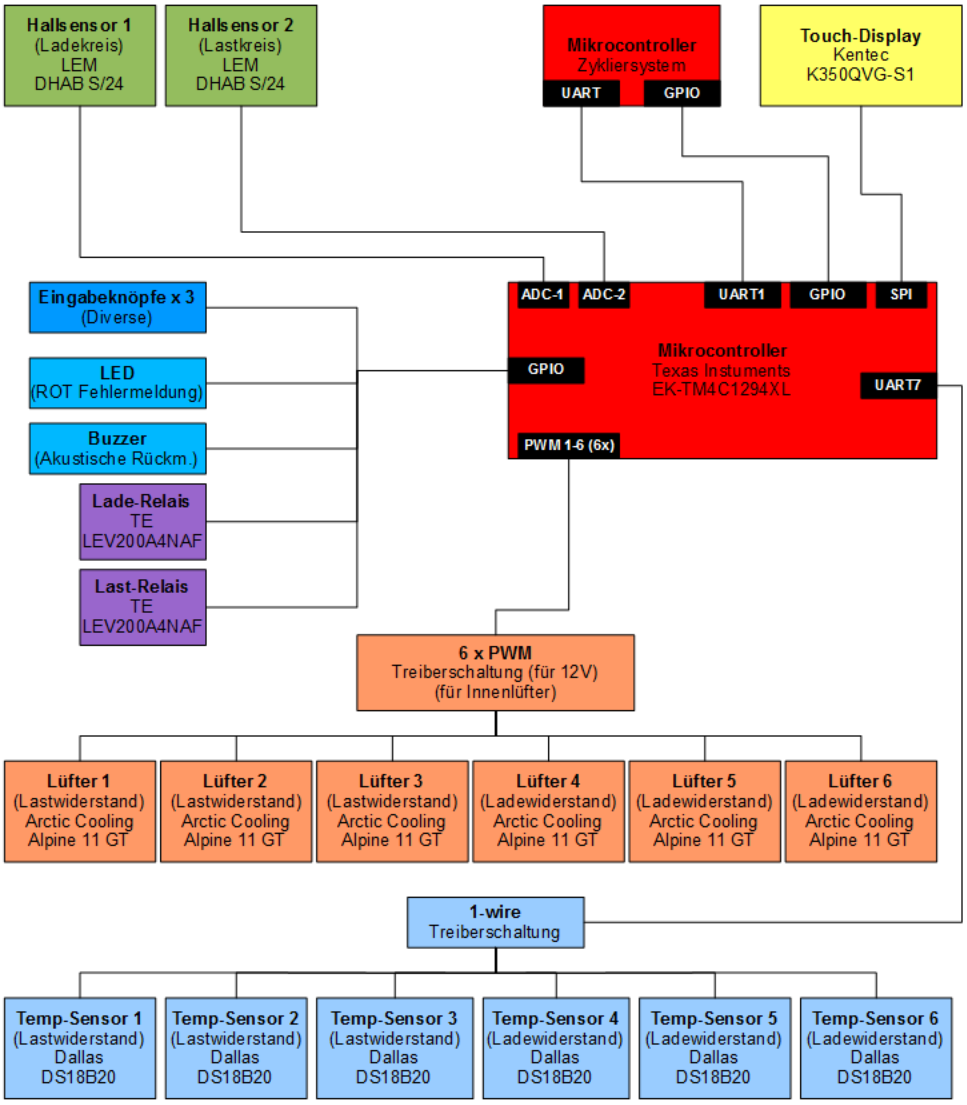


Abbildung 4.1.: Verschaltung der Komponenten

Die exakte PORT & Pin Belegung ist auf Tabelle A.1 vermerkt.

4.3. 1-wire ROM Codes der Sensoren

Die ROM-Codes der Temperatursensoren wurde vor dem Einbau in das Endgerät mit dem Testprogramm (siehe 2.2) ausgelesen und notiert. Die Sensoren wurden auf der Verkabelung von 1-6 durchnummeriert, was bei dem endgültigen Einbau einer falschen Anordnung vorbeugt. Die softwareseitige Zuordnung läuft in der Funktion: “one_wire_Devices“ ab.

Die Zuordnung der Lüfter zu den ROM-Codes der Temperatursensoren lautet wie folgt:

Lüfter 1 (Lastwiderstand)	0x28, 0xEB, 0x56, 0x37, 0x06, 0x00, 0x00, 0xD9
Lüfter 2 (Lastwiderstand)	0x28, 0x8A, 0x5D, 0x37, 0x06, 0x00, 0x00, 0x5D
Lüfter 3 (Lastwiderstand)	0x28, 0x83, 0x22, 0x37, 0x06, 0x00, 0x00, 0xC9
Lüfter 1 (Ladewiderstand)	0x28, 0x52, 0x66, 0x37, 0x06, 0x00, 0x00, 0xCC
Lüfter 2 (Ladewiderstand)	0x28, 0x4D, 0x76, 0x37, 0x06, 0x00, 0x00, 0xCF
Lüfter 3 (Ladewiderstand)	0x28, 0x53, 0x26, 0x37, 0x06, 0x00, 0x00, 0x12

Tabelle 4.5.: Zuordnung der ROM-Codes: Lüfter - Temperatursensor

4.4. Drehzahlmessungen & berechnete Drehzahl

Bei der Messung der Lüfterdrehzahlen und deren Verarbeitung gilt es, die Eigenschaften der Lüfter und Zeitfenster der Messwerte durch den Mikrocontroller zu beachten. Die Lüfter haben eine einstellbare Drehzahl im Bereich von ca. 800-2100 RPM (rounds-per-minute) und liefern zwei Takte pro Umdrehung. Die im Mikrocontroller verwendeten Timer, welche hier als Flankenzähler implementiert wurden, können daher mit einer steigenden Taktflanke eine halbe Umdrehung messen. Wählt man nun die Zeitfenster zu niedrig, erhält man bei der Berechnung der Drehzahl eine zu große Abweichung. Besonders bei niedrigeren Drehzahlen, wird der Fehler größer.

Hier ist beispielhaft die Messung von 800 RPM durch den Mikrocontroller aufgeführt.

Zeitfenster	RPM Lüfter	RPM pro Zeitfenster	Messwert (Ganzzahlig)	Berechnung	RPM (berechnet)	Abweichung
10 ms	800	0,267	0	(0 * 100 * 60) / 2	0	-800
100 ms	800	1,333	2	(2 * 10 * 60) / 2	600	-200
1 sek	800	13,333	26	(26 * 60) / 2	780	-20
5 sek	800	66,667	133	(133 * 12) / 2	798	-2

Tabelle 4.6.: Drehzahlmessungen & berechnete Drehzahl

Da die Umdrehungszahl der Lüfter verhältnismäßig träge ist und eine bessere Genauigkeit der RPM erzielt werden sollte, wurde in der Programmierung ein Zeitfenster von 5 Sekunden gewählt. Auch die Temperaturentwicklung an den Widerständen ist träge genug, so dass ein Lüftervorgabewert mit einer Verzögerungszeit von 5 Sekunden ausreichend erscheint.

4.5. Fehlerlevel

In der Software wurden verschiedene Fehlerlevel implementiert. Diese dienen zur Fehleranalyse (im Abschaltfall) oder in der Software selbst als Auslösegrund für Gegenmaßnahmen. Die Fehlerlevel werden auch immer über die UART Schnittstelle an den Zykliercontroller übertragen.

Fehlerlevel	Eskalationsstufen
L#00	kein Fehler, das System läuft einwandfrei.
L#01	Warnungslevel, hier werden die Warncodes an das Zykliersystem übertragen.
L#02	Fehlerfall, es werden Gegenmaßnahmen durchgeführt und die Fehlercodes an das Zykliersystem übertragen. Der GPIO Freigabepin wird gelöscht
L#03	Kommunikationsfehler, GPIO Freigabepin des Zykliercontrollers nicht gesetzt.

Tabelle 4.7.: Fehlerlevel

4.6. Variablen-, Fehler- & Warnungscodes

Zur einfacheren Kommunikation der beiden im System befindlichen Mikrocontroller wurden Variablen-, Fehler- & Warncode-Tabellen erstellt. Diese Tabellen müssen auf beiden Controllern implementiert sein. Somit kann mit einem relativ kurzen Befehl die gewünschte Information übertragen werden.

Die Variablencodes dienen der Identifizierung der übertragenen Variable. Hier wird erst der betreffende Variablencode, gefolgt von dem Variablenwert übertragen. Somit kann das Zykliersystem diese Messwerte bei Bedarf weiter verarbeiten.

Bei einem Fehlerfall, wird die dazu passende Softwaremaßnahme mit Systemstopp auf dem Schutzcontroller ausgeführt. Damit der Zykliercontroller eine Information über den Grund

des Systemstopps erhält, wird über die UART Schnittstelle der jeweilige Fehlercode übertragen.

Die Warncodes werden bei dem Über- oder Unterschreiten festgelegter Grenzwerte an den Zyklierconroller übertragen. Das Auslösen dieser Warnungen hat nicht den sofortigen Systemstopp zur Folge. Die Warnungen sollen nach Fehlerfällen dem Zykliercontroller lediglich zum Auffinden der Fehlerursachen dienen.

Eine detaillierte Auflistung der Variablen-, Fehler- & Warnungscodes befindet sich im Anhang.

4.7. Displayausgabe

Es wurden für die Anzeige des aktuellen Status drei durchschaltbare Bildschirme erstellt. Auf diesen sind die derzeitigen Werte und Eigenschaften ausgegeben.

4.7.1. Status - Displayausgabe

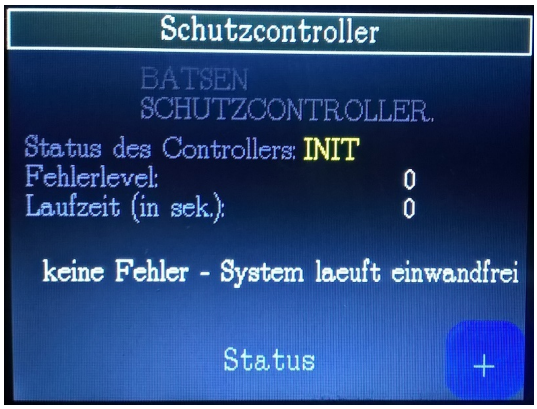


Abbildung 4.2.: Anzeige: Status (laufend)

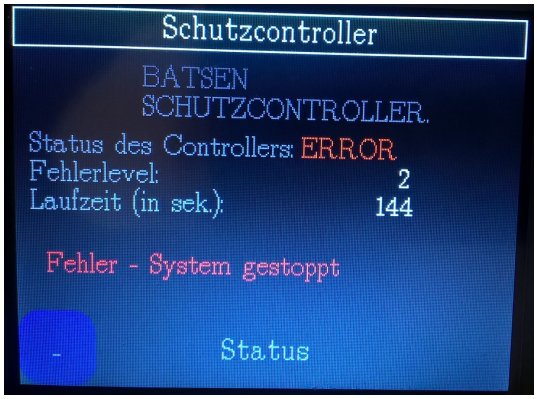


Abbildung 4.3.: Anzeige: Status (gestoppt)

Bei dem In Abb. 4.2 und Abb. 4.3 ist der Statusbildschirm dargestellt. Hier werden die nötigsten Informationen direkt angezeigt. Der Status des Controllers wird als Erstes ausgegeben, dieser kann die Werte “INIT“, “RUN“, “ERROR“ annehmen. Dieser Status wird farblich passend hervorgehoben. Des Weiteren werden noch der derzeitige Fehlerlevel und die Laufzeit in Sekunden angezeigt. Diese Werte sind farblich weiß hervorgehoben. Darunter wird in Form eines Textes der derzeitige Systemstatus noch einmal ausgegeben.

4.7.2. Lastkreis & Ladekreis - Displayausgabe

Abbildung 4.4.: Anzeige: Lastkreis

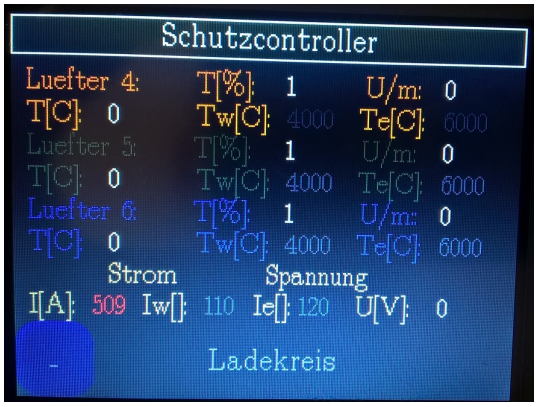


Abbildung 4.5.: Anzeige: Ladekreis

Auf den beiden Bildschirmen “Anzeige Lastkreis“ 4.4 und “Anzeige Ladekreis“ 4.5 werden die Messwerte und Grenzwerte zum Lade- bzw. Lastkreis dargestellt. Die Messwerte werden in weiß hervorgehoben, befindet sich ein Messwert im Warnbereich wird er gelb, hat dieser den Fehlerbereich überschritten wird er rot hervorgehoben. Die jeweiligen Warn- und Fehlergrenzwerte werden in einem gräulichen Farbton angezeigt.

4.8. ADC Booster-Pack zur Spannungsüberwachung der Batterie

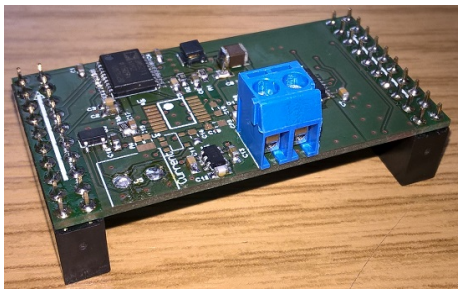


Abbildung 4.6.: ADC Booster-Pack

Zur Spannungsüberwachung sollte ein ADC (Analog-Digtial-Converter) verwendet werden, welcher einen galvanisch getrennten Messaufbau ermöglicht. Von der BATSEN Arbeitsgruppe wurde ein Booster-Pack mit fertiger Software entwickelt. In der Schaltung arbeiten drei ICs welche einen ADC realisieren, der die Anforderungen einer galvanischen Trennung erfüllt. Das fertige Booster-Pack ist ein ADC, der über die SPI Schnittstelle ausgelesen werden

kann. Booster-Packs sind fertige Schaltungen, welche direkt auf die Erweiterungsports des hier verwendeten Mikrocontroller-Evaluation-Boards von Texas Instruments gesteckt werden können.

Die drei IC's haben die folgenden Eigenschaften und Aufgaben:

Der ADuM4154 von Analog Devices ist ein SPI Isolator. Dieser unterstützt SPI Taktraten von bis zu 17 MHz, vier schnelle verzögerungsarme SPI Kanäle und die Ansteuerung von bis zu vier SPI-Slaves [1]. Dieser wird verwendet, um die Kommunikation mit dem ADC galvanisch zu trennen.

Der ADuM5211 von Analog Devices ist ein DC-DC Converter. Dieser unterstützt eine regulierbare Ausgangsspannung von 3,15 bis 5,25 Volt mit einer Ausgangsleistung von bis zu 150 Milliwatt [2]. Der DC-DC Converter wird verwendet, um den ADC mit seiner Versorgungsspannung zu betreiben. Dies trennt den ADC galvanisch von der Spannung des Mikrocontrollers, welcher auf der Kommunikationsebene durch den SPI Isolator vom ADC getrennt wird.

Der MAX1120 von Maxim Integrated ist ein ADC. Dieser hat eine Auflösung von 24 Bit und eine sehr geringe Stromaufnahme von unter 300 Mikroampere [5]. Der ADC ist zur direkten Spannungsmessung an den Stromschienen vorgesehen. Der ADC wird von dem DC-DC Konverter galvanisch getrennt mit Spannung versorgt und kommuniziert über den SPI Isolator galvanisch getrennt mit der SPI Schnittstelle des Mikrocontrollers.

Das Booster-Pack ist bereits fertig aufgebaut und die Software ist in einem Testprogramm lauffähig. Leider ist es im Zeitrahmen dieser Bachelorarbeit nicht mehr möglich gewesen, das Booster-Pack in die bestehende Schaltung/Software zu integrieren.

5. Detail

5.1. Hardware

5.1.1. Schaltung - PWM und Tacho-Signal (Lüfter)

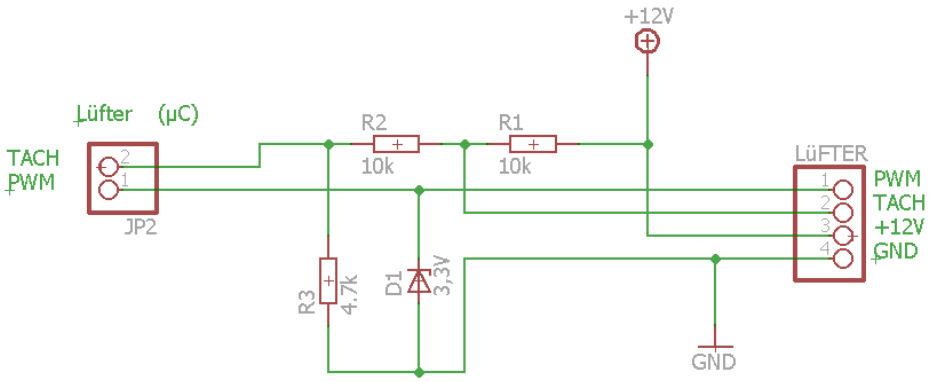


Abbildung 5.1.: Schaltung: Lüfter PWM & Tacho-Signal

Der PWM Kanal des Mikrocontrollers gibt ein 25 kHz Signal mit 3,3V aus, welches direkt in den PWM gesteuerten Eingang des Lüfters geht. Ein Standard CPU-Lüfter verträgt hier als Steuersignal Spannungswerte von 0,8 bis 5,25 Volt. Durch die Schaltung wird das Tacho-Signal zum Mikrocontroller Flankenzähler Timer Eingang auf eine Spannung von ca 2,5 Volt gebracht.

5.1.2. Schaltung - 1-wire (Temperatursensoren)

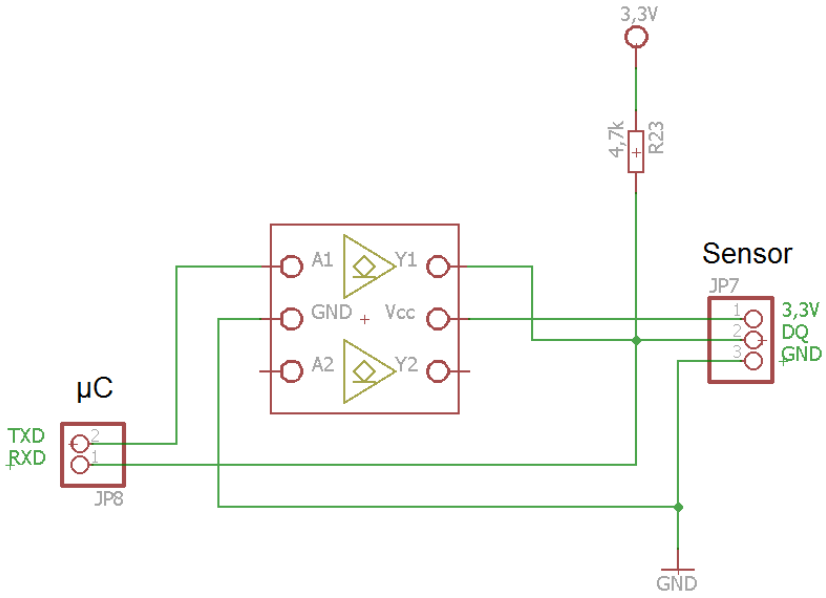


Abbildung 5.2.: Schaltung: Temperatursensoren 1-wire

Für diesen Schaltungsteil wurde ein NC7WZ07 TinyLogic UHS Dual Buffer der Firma Fairchild verwendet. Dieses Bauteil ermöglicht es, 1-wire Bauelemente an einer UART Schnittstelle zu verwenden. Voraussetzung hierfür ist eine softwareseitige Anpassung des UART Ein- / Ausgangs. Die Schaltung entspricht der Schaltung in Abb. 2.1 und ist direkt der dem “Tutorial 214“ [6] von Maxim entnommen. Der NC7WZ07 ist eine “open drain“ beschaltete CMOS-Transistor Schaltung und verhält sich daher hier wie ein Schaltausgang. Mit Hilfe eines Vorwiderstandes, lässt sich damit ein beliebiges Spannungsniveau schalten.

5.1.3. Schaltung - ADC (Hallsensoren)

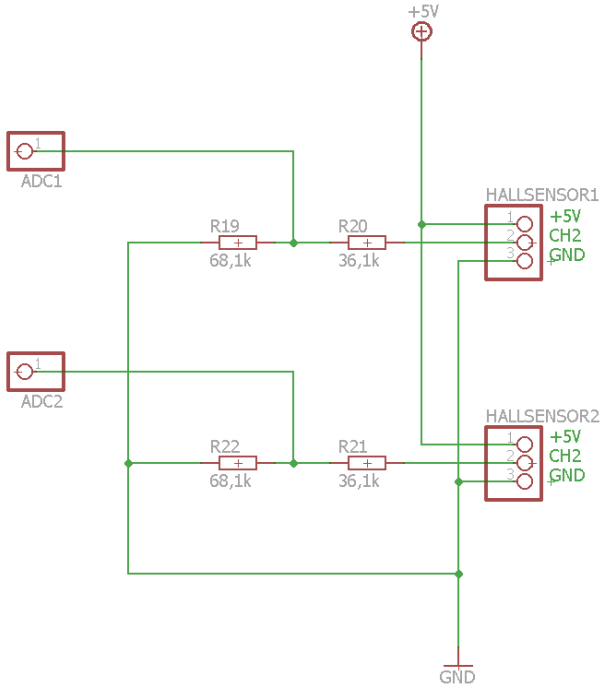


Abbildung 5.3.: Schaltung: Hallsensoren ADC

Für die Hallsensoren wurden zwei einfache Spannungsteiler verwendet, da die gewählten Hallsensoren einen Spannungbereich von 0-5 Volt zur analogen Übermittlung ihrer Messwerte verwenden. Der Messbereich ist von -500 bis +500 Ampere, wobei bei einer Spannung von 2,5 Volt die Nulllinie (0 Ampere) liegt. Der Spannungsteiler teilt diesen Bereich von 0-5 Volt auf 0-3,3 Volt herunter.

Mikrocontroller

Der ADC des Mikrocontrollers hat 12 Bit:

$$12Bit = 2^{12} = 4096Steps$$

Die Spannung mit dem der ADC arbeitet liegt bei 3,3V (gemessen wurde 3,252V):

$$\frac{3,3V}{4096Steps} = 0,000803V/Bit \quad , \quad \frac{3,252}{4096Steps} = 0,000794V/Bit$$

Dies ergibt eine Genauigkeit von:

$$0,8mV/Bit$$

Hallsensor

Laut Datenblatt hat der Hallsensor zwei Kanäle mit unterschiedlichen Auflösungen, hier wurde Kanal 2 (CH2) verwendet:

- CH1 | -75 bis +75 A | Sensivity 27,7 mV/A
- CH2 | -500 bis +500 A | Sensivity 4 mV/A

Spannungsteiler

$$U_{vor} = \frac{(U_{Hall} * R_1)}{(R_1 + R_2)} \quad , \quad U_{ADC} = \frac{(U_{Hall} * R_2)}{(R_1 + R_2)}$$

Die verwendeten Spannungsteiler-Widerstände sind, $R_1 = 36,1$ kOhm und $R_2 = 68,1$ kOhm. Dies ergibt einen Gesamtwiderstand von $R_{ges} = 104,2$ kOhm. Dies teilt die Maximalspannung des Hallsensors U_{Hall} von 5 Volt auf eine für den Mikrocontroller abtastbare Spannung U_{ADC} herunter.

$$U_{ADC-max} = \frac{5V * 68.100\Omega}{104.200\Omega} = 3,268V \quad , \quad U_{ADC-min} = 0V$$

$$U_{ADC-Null} = \frac{2,5V * 68.100\Omega}{104.200\Omega} = 1,634V$$

5.1.4. Schaltung - UART (galvanische Trennung)

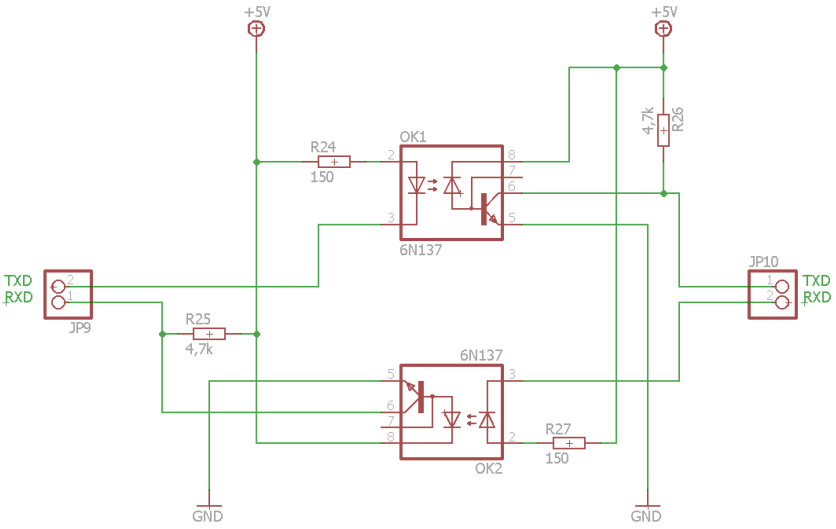


Abbildung 5.4.: Schaltung: galvanische Trennung UART

Diese Schaltung wurde entwickelt und aufgebaut, um eine galvanische Trennung der UART Kommunikation zwischen Schutzsystem und Zykliersystem zu realisieren. Die Kommunikationskanäle sind somit über Optokoppler voneinander potentialfrei getrennt. Da ein Zykliersystem zum Ende dieser Bachelorarbeit noch nicht vorhanden war, dient diese Schaltung nur als schon fertig aufgebauter Konzeptentwurf.

5.2. Software

Hier werden die wichtigsten Softwaremodule und deren Funktion erklärt.

5.2.1. State-Machine

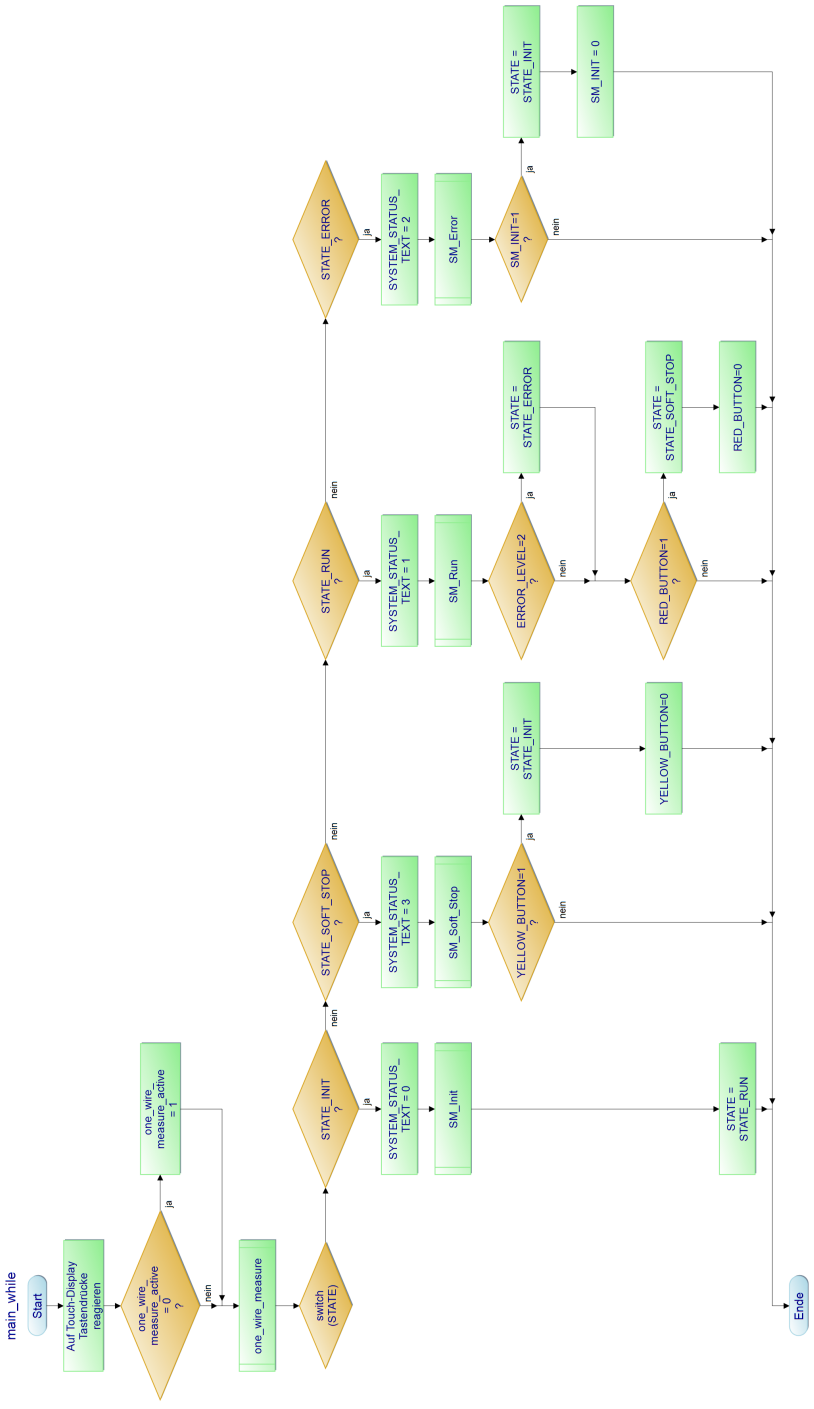


Abbildung 5.5.: PAP: State-Machine (in while(1) der main Funktion)

In der State-Maschine, welche sich in der Main Funktion befindet, wird der gesamte Ablauf des Programms gesteuert. (Abb. 5.5) Die Messungen von Temperaturen und Drehzahlen kommen aus der Timer5A Schleife, welche alle 500us ausgeführt wird. Die Messungen selbst werden hier, allerdings je nach Anforderungen, nur im Sekundentakt oder 5 Sekundentakt ausgeführt. Die State-Maschine ist nur eine Abarbeitungsschleife, welche Merker und Variablen abfragt und gegebenenfalls die gewünschten Funktionen aufruft.

Es wurden vier verschiedene State-Machine Zustände implementiert:

SM-Init
Dieser Zustand wird nur beim ersten Start oder beim Software Neustart des Systems aufgerufen. Hier werden die gespeicherten Werte aus dem ROM in die Variablen geschrieben, die Startwerte des Systems gesetzt und der Zustand SM-Run gesetzt.

SM-Run
Dies ist der Hauptzustand, in dem das gesamte System im Normalzustand läuft. Hier werden die Messwerte ständig überprüft und im Falle eines Messwertes, der sich im Warnbereich befindet, die betreffenden Maßnahmen ergriffen. Ist dieser Warnbereich erreicht, werden die Werte an das Zykliersystem über die UART Schnittstelle übertragen. Hat ein Messwert den Fehlerbereich erreicht, werden die Werte ebenfalls an das Zykliersystem übertragen und es wird sofort in den Zustand SM-error übergegangen.

SM-Error
Dieser Zustand gilt für den Fehlerfall. Hier wird das Gesamtsystem gestoppt.

GPIO Interrupts
Durch GPIO Interrupts (z.B. Taster), werden Variablen angesteuert. Diese werden in der Interruptfunktion auf 1 gesetzt, damit die Variablen in der State-Machine wiederum berücksichtigt werden können.

5.2.2. State-Machine Funktionen

Es wurden diverse Funktionen implementiert, die aus der State-Machine oder diesen Unterfunktionen selbst heraus aufgerufen werden können.

SM-Funktion: SM_Init

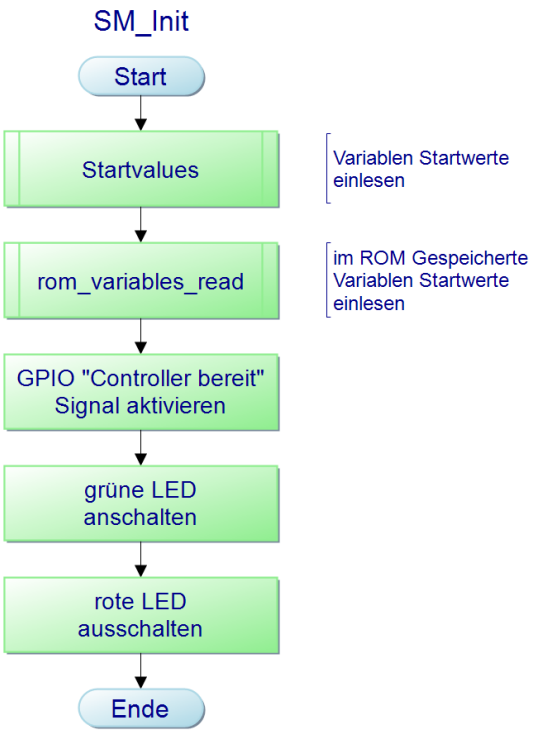


Abbildung 5.6.: PAP: SM_Init

Die Funktion SM_Init wird aus dem State-Machine Zustand SM-Init heraus aufgerufen. In SM_Init wird als Erstes eine weitere Funktion (Startvalues) heraus aufgerufen, welche wiederum die Startwerte für einen Großteil der im Gesamtprogramm verwendeten Variablen setzt. Nachdem die Startwerte gesetzt wurden, werden die im ROM gespeicherten Variablen ausgelesen. Hier können im Bedarfsfall auch aus den Startwerten geholte Variablen wieder überschrieben werden. Somit bleibt für die Weiterentwicklung des Gesamtgerätes die Option offen, die Werte aus dem letzten Gerätestart weiter zu verwenden. Als letztes werden via GPIO das “Controller bereit“ Signals gesetzt, die grüne LED ein und die rote LED ausgeschaltet.

SM-Funktion: SM_Soft_Stop

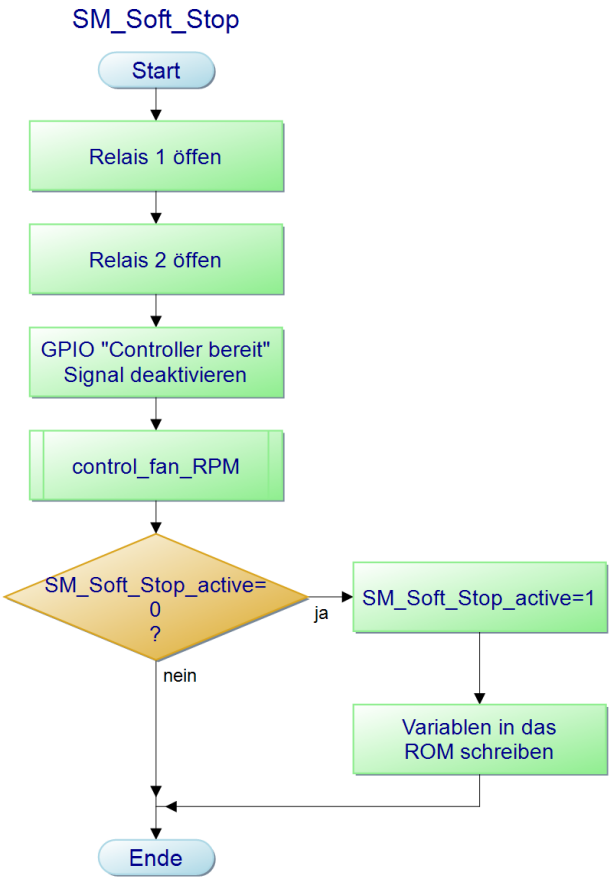


Abbildung 5.7.: PAP: SM_Soft_Stop

Die Funktion SM_Soft_Stop gilt für den Fall, dass der rote Button am Gerät gedrückt wird, um einen System-Stopp von Hand zu initiieren. Es werden GPIO Funktionen aufgerufen, welche Relais 1 und 2 öffnen und das “Controller bereit“ Signal löschen. Danach wird die Funktion zum Steuern der Lüfterdrehzahl auf Basis der Widerstandspaket-Temperaturen aufgerufen. Die in das ROM zu speichernden Variablen werden in dieses geschrieben, wenn die SM_Soft_Stop Funktion das erste Mal aufgerufen wurde.

SM-Funktion: SM_Run

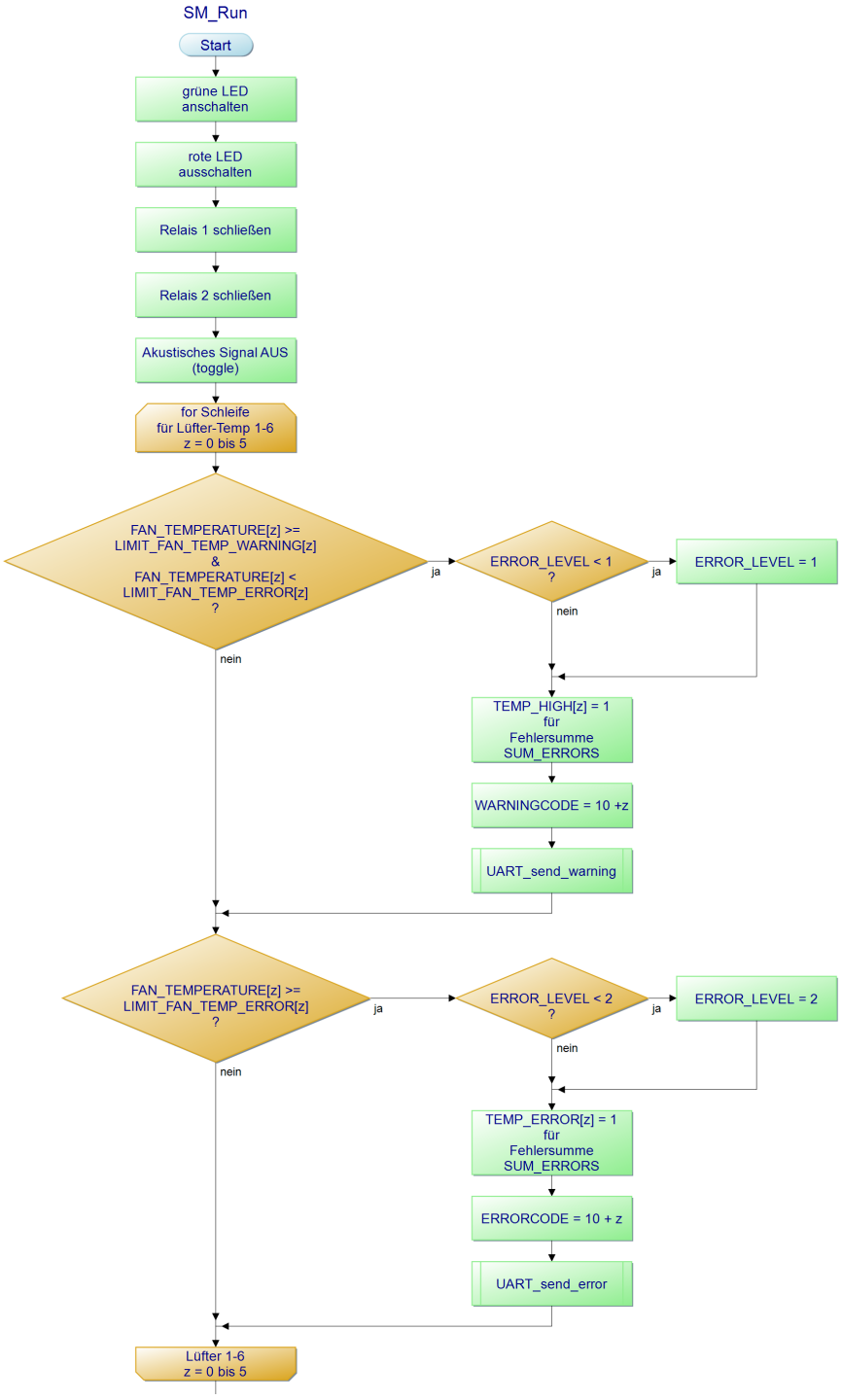


Abbildung 5.8.: PAP: SM_Run (Teil 1)

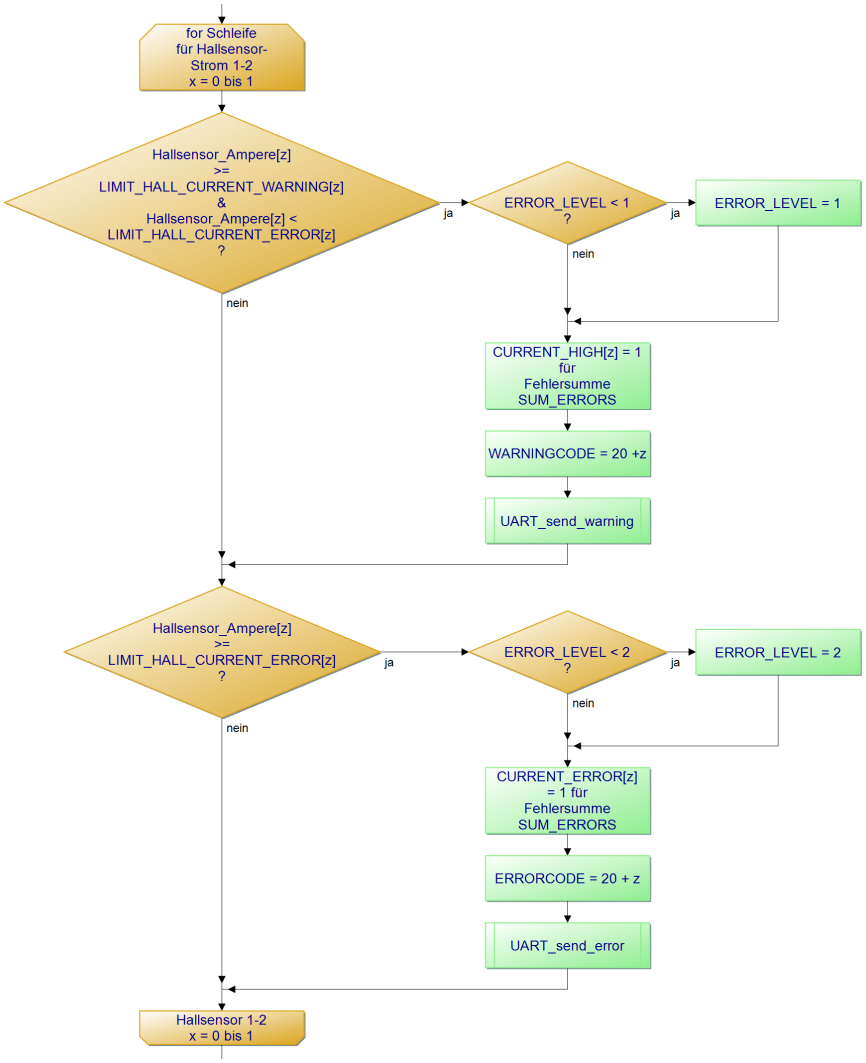


Abbildung 5.9.: PAP: SM_Run (Teil 2)

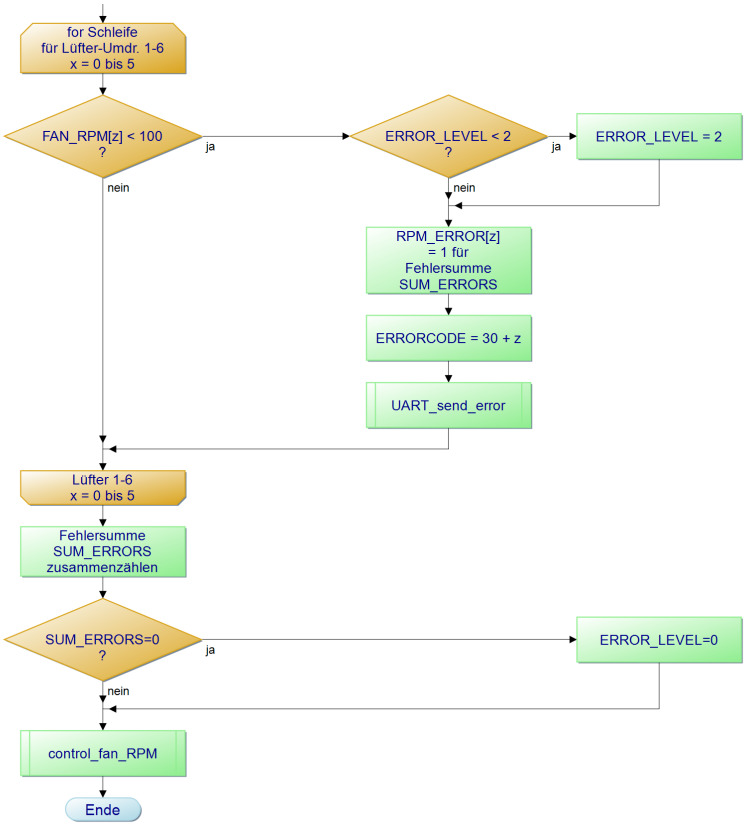


Abbildung 5.10.: PAP: SM_Run (Teil 3)

Die SM_Run ist die Funktion, welche in Normalfall bei fehlerlosem Betrieb ausgeführt wird. Gleich am Anfang dieser Funktion wird mittels der GPIO-Funktionen dafür gesorgt, dass die grüne LED angeschaltet, die rote LED ausgeschaltet, Relais 1 & 2 geschlossen und die akustische Rückmeldung ausgeschaltet werden. Danach werden nacheinander drei for-Schleifen ausgeführt. Die erste for-Schleife ist für das Auswerten der Temperaturen an den Lüfter-Widerstand-Verbunden zuständig. Hier werden jeweils für einen Temperatursensor die Temperaturen mit den definierten Warn- & Fehlergrenzwerten verglichen. Ist eine Temperatur erhöht (im Warnbereich) wird die Variable für den Fehlerlevel “ERROR_LEVEL“ auf 1 gesetzt, sofern der Fehlerlevel nicht bereits größer als 0 ist. Für den Fehlersummen-Merker am Ende dieser Funktion wird sich gemerkt, wenn eine Warnung vorlag. Nachfolgend wird der betreffende eindeutige Warncode mit Messwert über die UART Schnittstelle übertragen. Ist die Temperatur im Fehlerbereich, wird die Variable für den Fehlerlevel “ERROR_LEVEL“ auf 2 gesetzt, sofern der Fehlerlevel nicht bereits größer als 1 ist. Für den Fehlersummen-Merker am Ende dieser Funktion wird sich gemerkt, wenn ein Fehler vorlag. Nachfolgend wird der betreffende eindeutige Fehlercode mit Messwert über die UART Schnittstelle übertragen. Mit demselben Verfahren werden auch die Warn- & Fehlerwerte der beiden Hallsensoren und

die Fehlerwerte der Mindestdrehzahl für die sechs Lüfter überprüft. Am Ende wird mittels der Fehlersummen-Merker überprüft, ob überhaupt ein Fehler vorlag. Ist dies nicht der Fall, wird die Variable für den Fehlerlevel “ERROR_LEVEL“ auf 0 gesetzt. Abschließend wird die Funktion für die Drehzahlansteuerung der Lüfter ausgeführt.

SM-Funktion: SM_Error

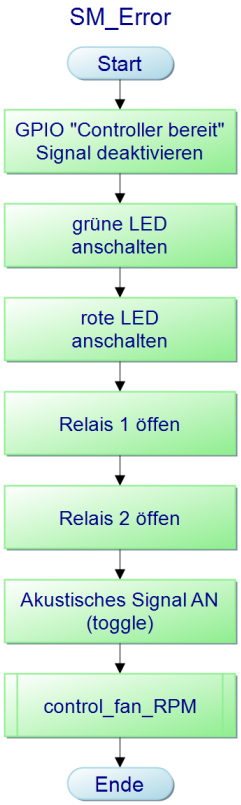


Abbildung 5.11.: PAP: SM_Error

Die Funktion SM_Error wird im State-Machine Zustand SM-Error aufgerufen. In dieser Funktion werden nacheinander sechs GPIO Pins angesteuert. Hier werden, das “Controller bereit“ Signal gelöscht, die grüne LED ausgeschaltet, die rote LED eingeschaltet, Relais 1 und 2 geöffnet und das akustische Signal (Buzzer) eingeschaltet. Am Ende der Funktion wird die Unterfunktion control_fan_RPM aufgerufen, welche die Lüfterdrehzahl auf Basis der Widerstandspaket-Temperaturen ansteuert.

SM-Funktion: Control-Fan-RPM

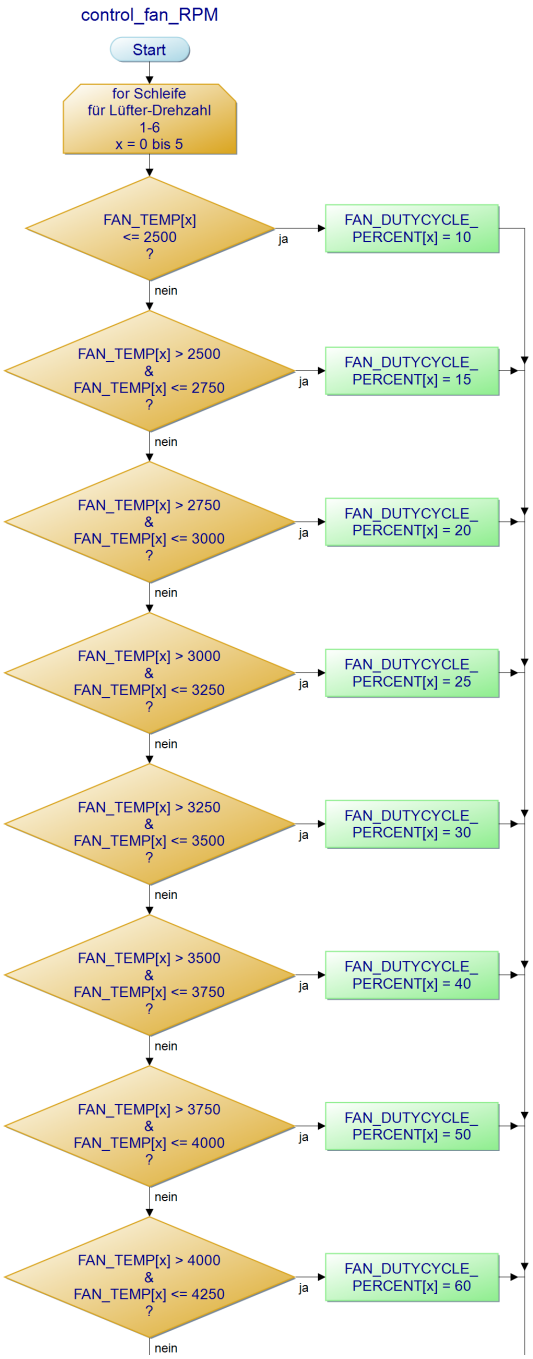


Abbildung 5.12.: PAP: Funktion: control_fan_RPM (Teil 1)

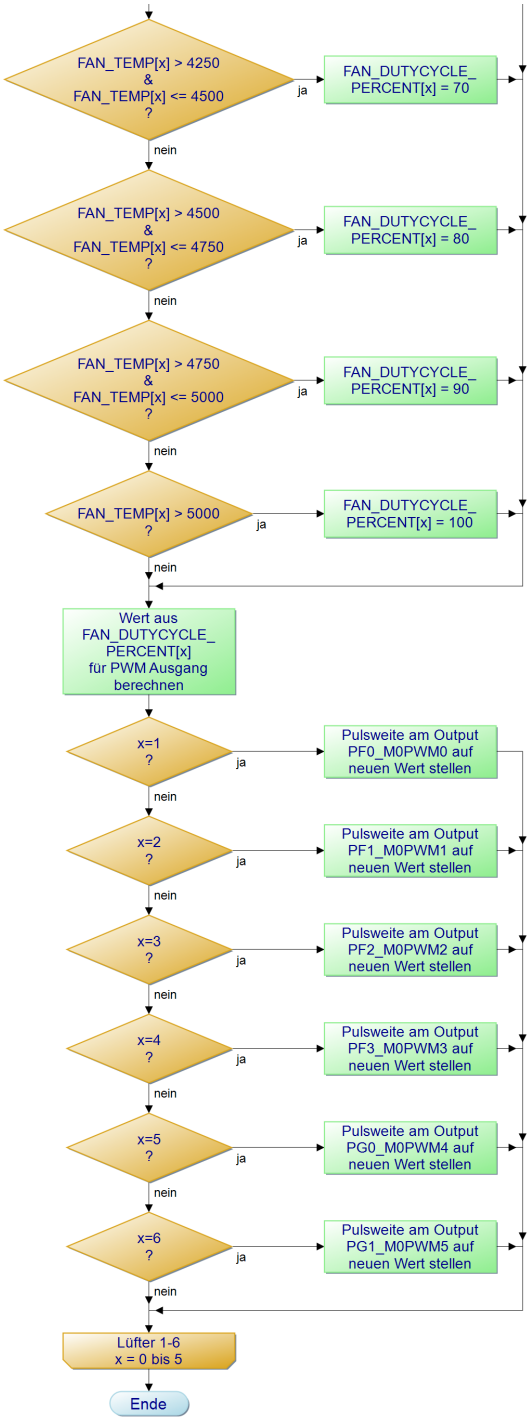


Abbildung 5.13.: PAP: Funktion: control_fan_RPM (Teil 2)

Die Funktion control_fan_RPM besteht aus einer for Schleife, welche 6 mal ausgeführt wird, jeweils einmal für jeden Temperaturpunkt auf einem Lüfter-Widerstands-Verbund. Somit wird für jeden dieser Punkte als erster Schritt der Temperaturbereich, in dem er sich befindet abgefragt, um diesem einen prozentualen Tastgrad für die Pulsweiten-Modulation (PWM) zum Lüfter zuordnen. Danach wird dieser prozentuale Wert in einen zu dem berechneten PWM passenden Setzwert umgerechnet. Im letzten Schritt wird der den Lüfter betreffende PWM-Ausgang mit dem Setzwert neu eingestellt.

SM-Funktionen: UART-Send warning/error

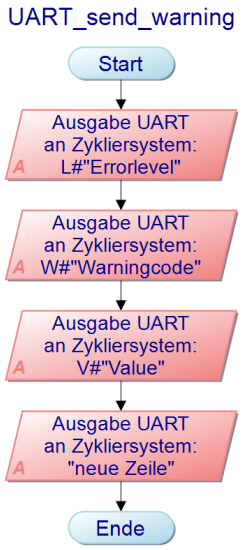


Abbildung 5.14.: PAP: Funktion: UART_send_warning

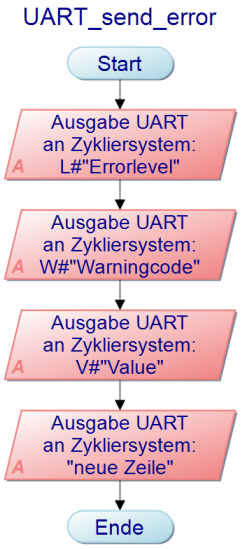


Abbildung 5.15.: PAP: Funktion: UART_send_error

Die Funktionen UART_send_warning und UART_send_error sind für die Kommunikation zum Zykliercontroller verantwortlich. Hier werden, wie in der schon aufgezeigten Vorgabe, den Fehlerlevel gefolgt von den Fehler- bzw. Warnungscodes, über die UART-Schnittstelle an den Zweitcontroller übertragen. Erst werden zwei “character“ gesendet, “L“ und “#“, gefolgt von dem einstelligen Integer-Wert des derzeitigen Fehlerlevels. Dann werden erneut zwei “character“ gesendet, “W“(Warning) oder “E“(Error) und “#“ diesmal gefolgt von einem zweistelligen Integer-Wert, welcher direkt einem Fehler bzw. einer Warnung entspricht. Abschließend werden wieder zwei “character“ gesendet, “V“ und “#“, mit einem darauffolgenden sechsstelligen Integer-Wert mit dem Variablenwert, welcher für den Fehler oder die Warnung verantwortlich war.

Beispiel: L#1W#10V#005500
(Temperatur an Sensor 1 (Lüfter 1) hat Warnung ausgelöst, da Temperaturwert bei 55 Grad liegt.)

Mit Kenntnis des Übertragungsmusters lässt sich so später an dem Zykliersystem eine Funktion implementieren, mit der die Informationen problemlos ausgelesen werden können.

SM-Funktionen: rom_variables_write und rom_variables_read

Es wurden zwei Funktionen implementiert, welche für das Speichern bzw. Auslesen des ROMs benötigt werden. Die Speicherfunktion sollte im Programm nicht hochfrequent auf-

gerufen werden, da die Schreibzyklen des internen ROMs auf etwa eine halbe Million beschränkt sind. In beiden Funktionen werden die zu speichernden Variablenwerte eingetragen.

SM-Funktionen: LED, Buzzer, Relay - on/off

Bei diesen Funktionen handelt es sich um einfache GPIO Setz-, Rücksetz-, Toggle-Funktionen. Hier werden lediglich Pins gesetzt oder gelöscht. Der Einfachheit halber in eigenen Funktionen, welche im Quellcode aufgrund ihres Namens schneller zu lesen und zu verstehen sind.

5.2.3. Drucktaster - (GPIO Interrupt Funktion)

Hier wurde sich für ein Entprellen via Software entschieden. Die drei Drucktaster wurden an GPIO Pins angeschlossen, für welche eine Interrupt-Funktion implementiert wurde. Wird an dem betreffenden PORT & Pin eine positive Flanke erkannt, wird die Interrupt Funktion aufgerufen. Hier wird mittels PORT & Pin Abfragen der gedrückte Knopf ermittelt, mit einer Delay-Funktion wird einige Mikrosekunden gewartet, um ein Entprellen des betreffenden Taster zu erreichen. Die dem Taster zugeordnete Variable wird = 1 gesetzt, diese werden dann in der State-Machine weiter verarbeitet.

5.2.4. im ROM gespeicherte Werte & Variablen

Damit nach dem Abschalten einige Werte und Variablen erhalten bleiben, werden diese nach Vorgabe der Software in den ROM des Mikrocontrollers geschrieben. Bei jedem Start des Systems werden diese Werte eingelesen. In diesem ersten Teil wird nur die Schaltanzahl der Relais gespeichert, da in dem derzeitigen Entwicklungsstand des Gesamtsystems (Zyklier & Schutzsystem) noch keine weiteren Werte als sinnvoll erscheinen. Aber durch die Art der Programmierung der ROM-Funktionen ist es sehr einfach möglich, die gewünschten Werte in den zwei read/write Funktionen einzufügen. Auch die Relais-Schaltzahl ist bei dem Prototyp nur zur Anschauung ohne weitere Verarbeitung verwendet worden, da in diesem Testmodell nur ein Relais verbaut ist und dieses dazu noch ein bereits gebrauchtes Bauteil war. Aus diesem Grund ich die wirkliche Schaltzahl hier auch noch unbekannt.

Bauteil	was wird gespeichert	Grund
Lade-Relais	Anzahl Schaltzyklen	Das Relais hat nur eine begrenzte Anzahl von Schaltzyklen
Last-Relais	Anzahl Schaltzyklen	Das Relais hat nur eine begrenzte Anzahl von Schaltzyklen
Temp. Sensoren	Warn-Grenzwerte	Temperaturen, ab denen eine Warnung gesendet wird
Temp. Sensoren	Fehler-Grenzwerte	Temperaturen, ab denen das System gestoppt wird
Hallsensoren	Warn-Grenzwerte	Ströme ab denen eine Warnung gesendet wird
Hallsensoren	Fehler-Grenzwerte	Ströme ab denen das System gestoppt wird
ADC Batteriespannung	Warn-Grenzwerte	Spannung ab der eine Warnung gesendet wird
ADC Batteriespannung	Fehler-Grenzwerte	Spannung ab der das System gestoppt wird

Tabelle 5.1.: ROM Werte und Variablen

5.2.5. Timer5A & Timer5B

Es wurden zwei Timer implementiert, die sich periodisch um den zeitkritischen Ablauf des Hauptprogramms kümmern. Ein Timer5A (Abb. 5.16) ist für den periodischen Ablauf der Messungen zuständig. Aufgrund des in ihm enthaltenen Quelltextes kann dieser Timer unterschiedlich lang für die Abarbeitung benötigten. Deshalb wurde ein weiterer Timer5B (Abb. 5.17) implementiert, welcher nur wenige Zeilen zur Abarbeitung enthält. Dieser steuert nur einige wenige Variable, welche für exaktere Zeitabfragen verwendet werden können.

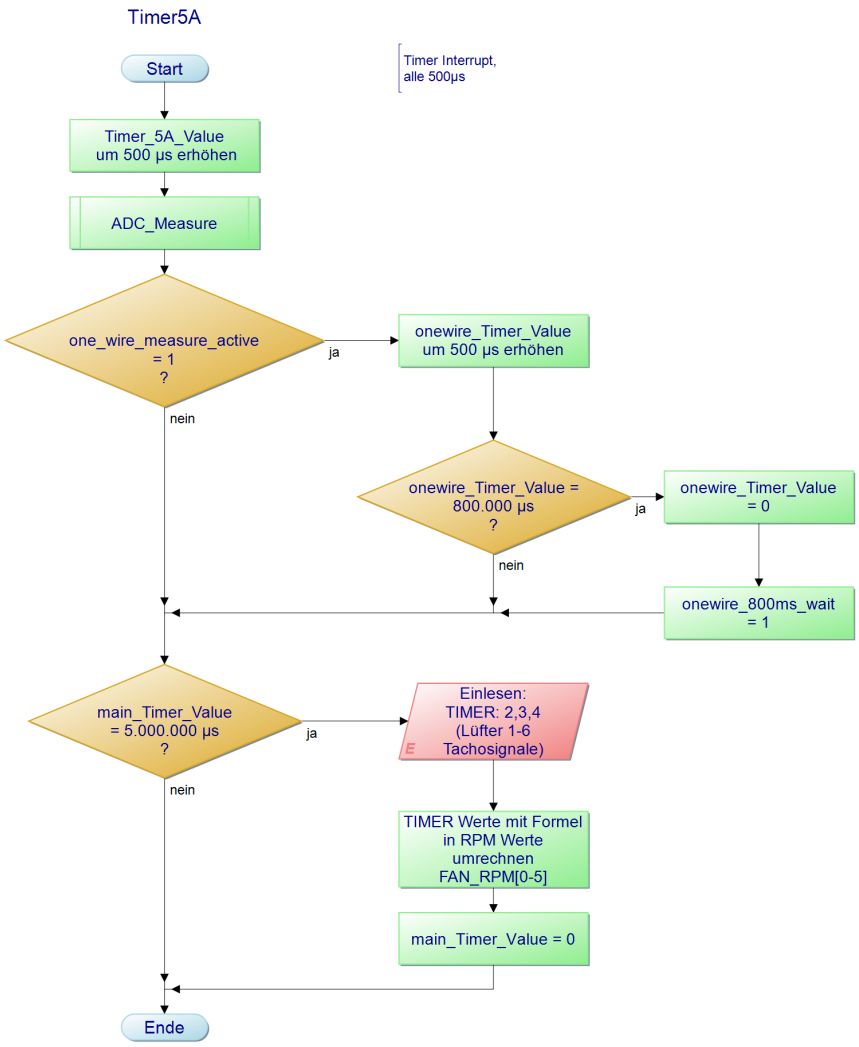


Abbildung 5.16.: PAP: Timer5A

Timer5A ist für die Zeitpunkte der Messungen und deren Aufruf zuständig. Diese Funktion wird alle 500 Mikrosekunden aufgerufen. Hier werden die ADC Werte gemessen und die 1-wire Wartezeit von 800 Millisekundenfür die 1-wire Schnittstelle geregelt. Abschließend wird im Fünf-Sekundentakt der Wert aus den Flankenzählern ausgewertet und in den zugehörigen Variablen gespeichert.

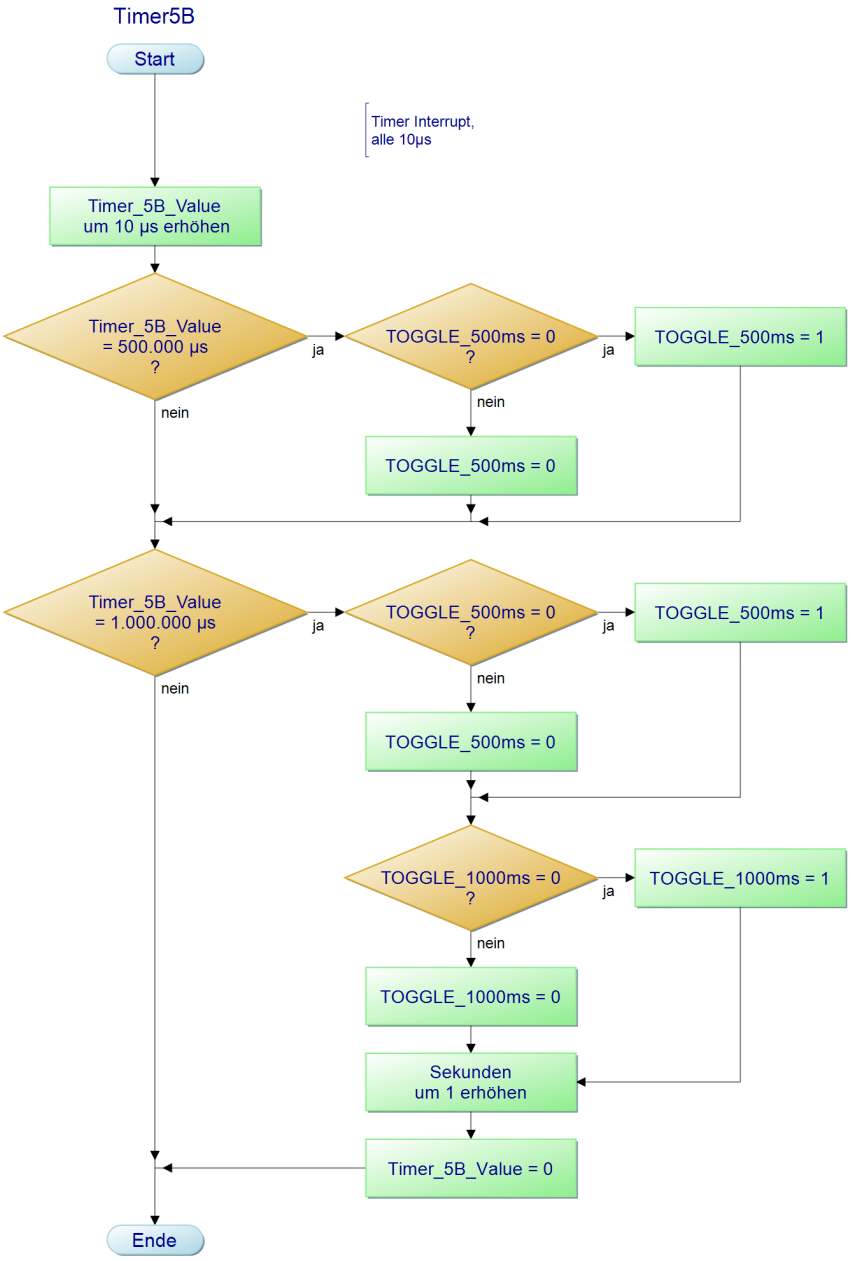


Abbildung 5.17.: PAP: Timer5B

5.2.6. 1-wire

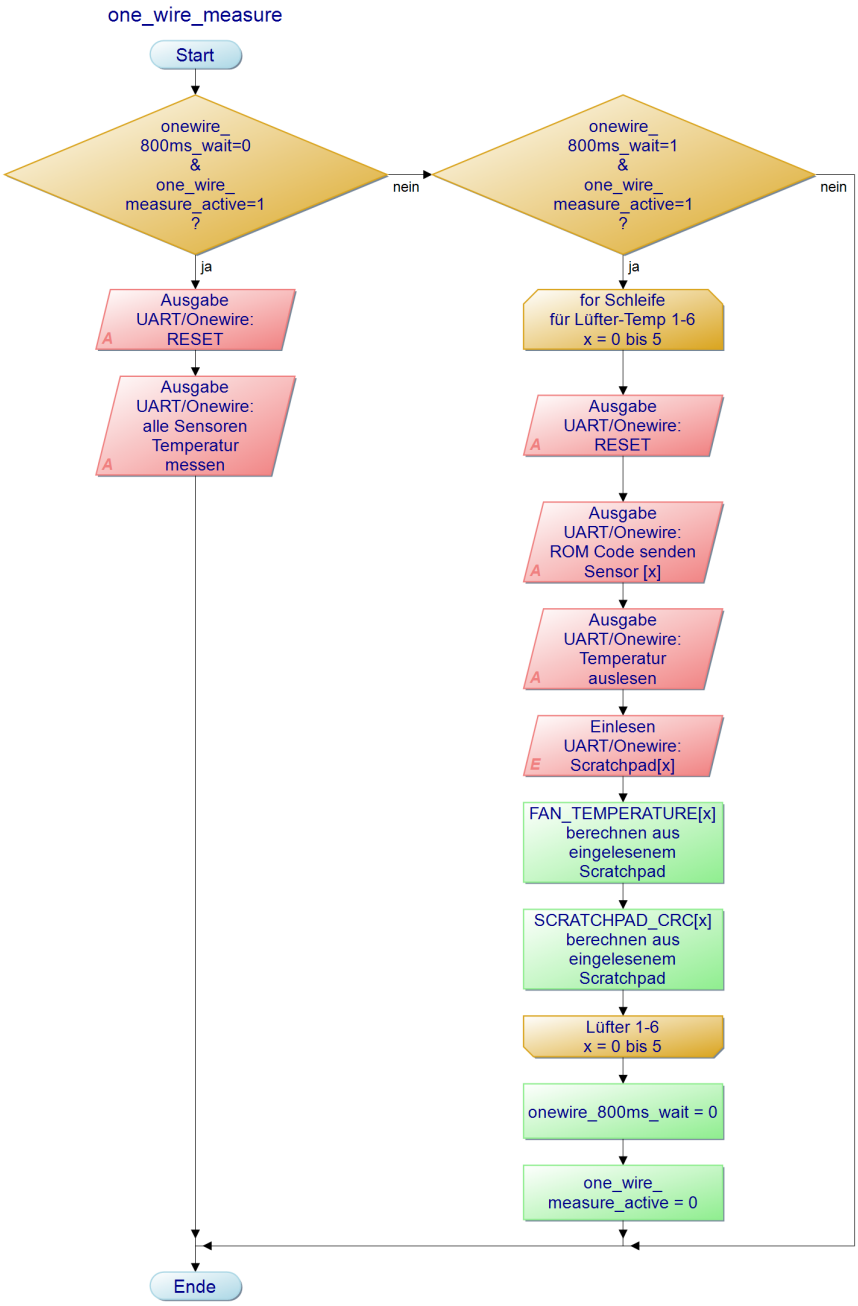


Abbildung 5.18.: PAP: Funktion: one_wire_measure

Die Funktion one_wire_measure (5.18) ist für die Messung der 1-wire Temperatursensoren zuständig. In der Funktion wird als Erstes abgefragt, ob die Merker one_wire_measure_active und onewire_800ms_wait gesetzt sind oder nicht. Diese Merker werden verwendet, um die für die UART Schnittstelle einzuhaltenden Zeitfenster nach Vorgabe für eine 1-wire Schnittstelle zu garantieren. Der Merker onewire_800ms_wait wird nicht in dieser Funktion gesetzt, diese Aufgabe ist in Timer5A vorgesehen.

Ist der Merker onewire_800ms_wait noch nicht gesetzt, so ist die erste if-Schleifen-Bedingung erfüllt. Hier wird eine neue Messung angestoßen. Dazu wird ein Reset an die 1-wire Sensoren gesendet, um danach die beiden hexadezimalen ROM-Kommandos 0xCC (alle Sensoren) und 0x44 (Temperaturen Messen) über die UART/1-wire Schnittstelle zu übertragen. Die Funktion wird danach verlassen.

Hier kommt der onewire_800ms_wait Merker zum Einsatz, solange die 800ms nach dem ersten Schleifendurchlauf noch nicht verstrichen sind, wird die Funktion one_wire_measure sofort wieder verlassen. Die 800ms sind eine Vorgabe aus dem Datenblatt und dem 1-wire Tutorial, denn die Temperatursensoren benötigen nach dem Befehl zur Messung mindestens 750ms, bis diese vollzogen ist.

Sind die 800ms verstrichen, werden mittels einer for-Scheife die sechs Temperatursensoren ausgelesen. Für jeden Sensor wird als erstes ein Reset über die 1-wire Schnittstelle gesendet, danach wird der ROM-Code von dem Sensor übertragen, der ausgelesen werden soll, gefolgt von dem Befehl “Temperatur auslesen“. Als Zweites wird ein Befehl gesendet, der dem Sensor ein Antworten über die 1-wire Schnittstelle ermöglicht. Die vom Temperatursensor empfangenen Daten werden in die betreffende Scratchpad Variable geschrieben. Aus der Scratchpad Variable werden dann die Temperatur und die CRC Nummer berechnet. Dieser Vorgang wird mit der for-Schleife für jeden Sensor wiederholt. Damit ist die Messung abgeschlossen und die beiden Variablen one_wire_measure_active und onewire_800ms_wait werden auf 0 gesetzt.

5.2.7. Funktion: ADC_Measure - (Hallsensor (ADC))

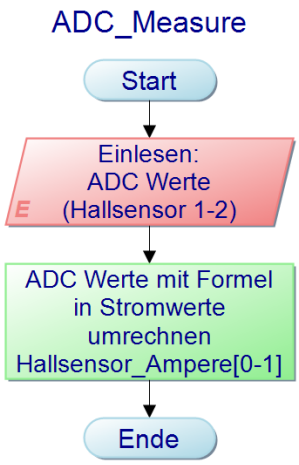


Abbildung 5.19.: PAP: Funktion: ADC_Measure

Die Funktion ADC_Measure, wird periodisch aus dem Timer5A heraus aufgerufen. Hier werden die an einer Spannungsteiler-Schaltung abfallenden Spannungswerte digital gemessen, um danach mittels einer Formel in einen Stromwert (in Ampere) umgerechnet und in eine Variable gespeichert zu werden.

6. Erprobung

6.1. Testaufbau

Abbildung 6.1.: Testaufbau: Gehäusefront

Das Endgerät wurde in einem Testaufbau mit allen Lastwiderständen in Reihe aufgebaut, da das Zykliersystem noch nicht vorhanden war. Die Stromschienen wurden für diesen Testaufbau mit einer Brücke kurzgeschlossen. Das Schutzsystem steuert ebenfalls das große Ladenetzteil an. Das Hauptprogramm wurde teilweise auskommentiert, um einzelne Bereiche oder Funktionen testen zu können. Es wurden in jedem Test nur 3 Lüfter, bzw. Widerstände verwendet (Lastwiderstände), daher war nur etwa die Hälfte der Messwerte interessant.

Weiter zu bemerken ist die Genauigkeit der Strommessung, die nur einen Bereich von +/- 1 Ampere erreicht. Diese für Messverhältnisse sehr starke Abweichung kommt von den Anforderungen an die Schutzschaltung, für welche eine Genauigkeit von +/- 1 Ampere vollkommen ausreicht. Im späteren Komplettsystem soll das Schutzsystem keine Ströme von ca. +120 Ampere zulassen. Dies wäre ein normaler Volllast-Strom, daher ist eine Obergrenze von +130 Ampere für den Fehlerfall denkbar. Bei diesen Größen war eine Genauigkeit von +/- 1 Ampere ausreichend, diese ist für die Testmessungen nicht verändert worden, da dies nicht mehr in den zeitlichen Rahmen gepasst hätte.

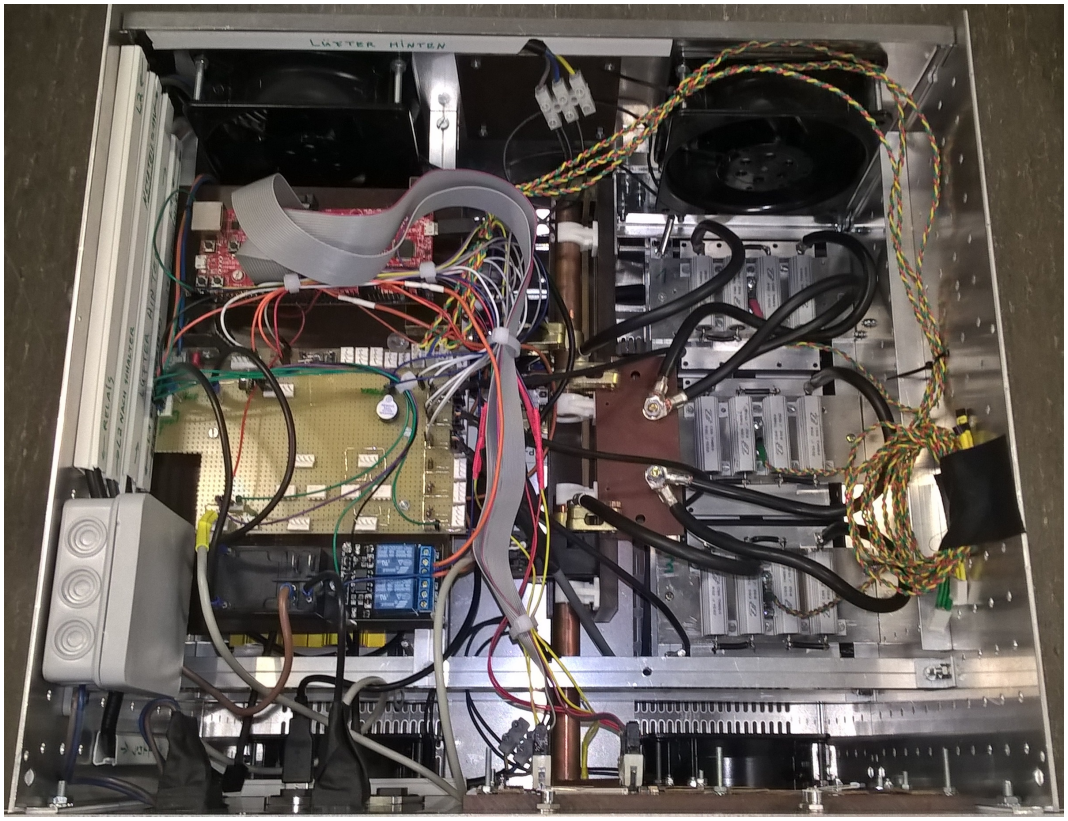


Abbildung 6.2.: Testaufbau: Innenansicht

6.2. 30-minütiger Volllasttest

Der erste Versuch des Gesamtaufbaus war der Test unter Volllast. Hier wurden die ca 40 Ampere, welche im Endsystem unter Volllast an jedem einzelnen Widerstandspaket auftreten können, für eine Dauer von ca. 30 Minuten an die 3 Lastwiderstände in Reihe geschlossen. Die vom Schutzsystem aufgezeichneten Werte wurden im Sekundentakt über die UART Schnittstelle ausgegeben und mit dem Programm PuTTy protokolliert.

Anhand des in Prozent ausgegebenen Tastgrades und der daraus resultierenden Umdrehungszahl, kann man die Temperaturregelung jedes einzelnen Widerstands-Lüfter-Verbundes verfolgen. Die Ansteuerung der Lüfter funktionierte erwartungsgemäß. Die Messung des Stromes funktionierte ebenfalls mit der Ausnahme eines Offsets, welcher ca 7 Ampere unterhalb des Realwertes lag. In der MATLAB Auswertung wurde dieser Messwert angepasst.

Nach Abschaltung des Leistungsnetzteils liefen die Lüfter mit voller Umdrehungszahl weiter, da hier in diesem Testmodus die Lüftersteuerung nach Abschalten des Netzteils außer Kraft gesetzt war, um eine zügige Kühlung der Bauteile zu erreichen.

Die Messergebnisse befinden sich im Anhang.

- Stromverlauf A.7
- Lüfter 1 - Temperatur, Tastgrad/RPM, Wärmewiderstand A.8
- Lüfter 2 - Temperatur, Tastgrad/RPM, Wärmewiderstand A.9
- Lüfter 3 - Temperatur, Tastgrad/RPM, Wärmewiderstand A.10

6.3. Test der Lüfteransteuerung

Bei diesem Testlauf wurde die Ansteuerung der Lüfterdrehzahlen getestet. Die von oben frei zugänglichen Temperatursensoren wurden von Hand mit Hilfe eines Heißlüfters erhitzt. Die Drehzahlen und Temperaturen wurden auch hier mittels dem Programm PuTTy protokolliert. Das Leistungsnetzteil wurde in diesem Versuch abgeschaltet, da die erhöhten Temperaturen auch ohne dieses von dem Hauptprogramm für die Drehzahlregelung der Lüfter als funktionsfähige Steuervorgabe verwendet werden können.

Wie in den MATLAB Plots A.11, A.12, A.13 in diesem Versuch zu sehen ist, funktionierte die temperaturabhängige Drehzahlansteuerung wie erwartet. Die jeweiligen Temperaturanstiege resultierten aus den Heißlüftererwärmungen. Das Gerät schaltete bei Temperaturen von 60 Grad in den Fehlermodus.

Die drei Lüfter wurden jeweils mit den Eigenschaften Temperatur in Grad Celsius, Tastgrad in % und Umdrehungszahl des Lüfters ausgegeben. Das obere Bild zeigt die Temperatur, dass mittlere Bild zeigt den Tastgrad und die Drehzahl eines einzelnen Lüfters und das untere Bild zeigt den Zustand der State-Machine.

Der Zustand der State-Machine ist in jedem der drei MATLAB Bildern derselbe. Die drei Lüfter liefen in jedem Testbild parallel zueinander, daher ist der State-Machine Zustand für

alle drei Lüfter gleichermaßen gültig. Die Zustände der State-Machine haben die folgende Bedeutung:

- 1 = RUN - Das System ist im normalen Überwachungszustand, es sind keine Fehler vorhanden.
- 2 = ERROR - Ein Fehler ist aufgetreten, das System ist im Fehlermodus.
- 3 = SOFT-STOP - Die Taste SOFT-STOP wurde gedrückt, damit startet das System neu, um in den RUN-Zustand zu gelangen.

Die Werte entsprechen der Variablen “SYSTEM_STATUS_TEXT“ in der State-Machine, zu sehen in Abb. 5.5 im vorigen Kapitel. Nach jedem Fehlerfall musste durch Drücken der SOFT-STOP Taste ein Rücksetzen des Systems erfolgen, um wieder in den Normalzustand wechseln zu können. Hierfür wurde ein weiterer Taster für die Tests umprogrammiert, um nach dem SOFT-STOP Zustand in den RUN-Zustand wechseln zu können.

Der erste Systemstopp wurde von Lüfter 2 ausgelöst, der zweite von Lüfter 1 und der dritte von Lüfter 3.

Die Messergebnisse befinden sich im Anhang.

- Lüfter 1 - Temperatur, Tastgrad/RPM, State-Machine Zustand A.11
- Lüfter 2 - Temperatur, Tastgrad/RPM, State-Machine Zustand A.12
- Lüfter 3 - Temperatur, Tastgrad/RPM, State-Machine Zustand A.13

6.4. Test der Stromfehlerabschaltung

Der letzte Test bezog sich auf die Stromüberwachung des Hallsensors an der Stromschiene. Diese wurde für diesen Test kurzgeschlossen, damit der Strom direkt durch die drei in Reihe geschlossenen Widerstände fließen konnte. Da die Grenzströme von +120 Ampere hier nicht erreicht werden konnten, wurde in der Software die Begrenzung auf 30 Ampere herabgesetzt. In dem Test diente der Hauptschalter auf der Frontseite des Gesamtaufbaus als Schalter für das Zuschalten des Leistungsnetzteils. Dies musste für den Test von Hand aktiviert und deaktiviert werden.

Bei jedem mechanischen Anschalten des Zykliersystem-Hauptschalters, wurde das Zyklier-netzteil eingeschaltet. Dadurch floss ein Strom von ca +35 Ampere durch die Widerstände. Der Mikrocontroller registrierte mit Hilfe des Hallsensor den Strom von über 30 Ampere und schaltete in den Fehlermodus. Das System schaltete wie erwartet bei einem zu hohen Strom in den Fehlermodus.

Der Zustand der State-Machine im unteren Bild entspricht der selben Logik wie bei dem Test der Lüfteransteuerung.

Die Messergebnisse befinden sich im Anhang.

- Stromfehlerabschaltung A.14

7. Schluss

7.1. Fazit

Das geforderte Schutzkonzept eines Überwachungscontrollers für einen Hochleistungs-Batterie Prüfstand wurde umgesetzt und teilweise aufgebaut. Die geforderten Aufgabenpunkte dieser Arbeit wurden je nach Anforderungen oder gegebenen Umständen konzeptionell oder praktisch realisiert. Die einzelnen in dem Schutzsystem und Gesamtaufbau verwendeten Komponenten wurden ausgewählt und auf ihre Eigenschaften hin untersucht. Die daraus resultierende konzeptionelle Vorarbeit für ein Fertigen eines Zykliersystems mit integriertem Schutzsystem wurde mit dieser Arbeit abgeschlossen. Die Tatsache, dass das Zykliersystem zum Ende dieser Arbeit noch nicht fertig war, ließ bei einigen Problemen nur ein Konzept für die Weiterentwicklung zu.

Der mechanische Aufbau des Systems war deutlich aufwendiger als gedacht und eine Weiterentwicklung des Gesamtsystems wird auch hier noch einiger kleinerer Änderungen bedürfen. Die mechanische Vorarbeit für das Endprodukt ist vollzogen und kann als Basis für dieses aufgefasst werden.

Schutzfunktionen wurden ebenfalls implementiert, auch diese werden sich in der Weiterentwicklung noch verfeinern und erweitern müssen.

Eine Displayausgabe wurde umgesetzt. Die relevanten bereits implementierten Messwerte werden auf dem Display angezeigt. Eine weitere Eingabe und vor allem eine Menüführung für das Schutzsystem konnte aus zeitlichen Gründen nicht implementiert werden, da die Verwendung eines Farb-Touch-Displays einen nicht zu unterschätzenden Zeitaufwand mit sich bringt. Es gibt hierfür zwar bereits vorgefertigte Widgets, diese müssen hingegen stark angepasst werden, so dass auch hier ein großer zeitlicher Aufwand entsteht.

Der Gesamtaufbau ist in einem 9HE 19 Zoll Rackelement realisiert worden, eine passende Frontplatte mit Ein- und Ausgabemöglichkeiten wurde ebenfalls geschaffen. Im Innenraum des Gesamtaufbaus wurden Haltemöglichkeiten für die Einzelkomponenten eingebaut. Auch die bereits verfügbaren Bauteile des Zykliersystems wurden integriert. Die Kühlung von relevanten Bauelementen und der Abtransport der Verlustleistung, wurden hinreichend durchgeführt. Der mechanische Aufbau wird mit Sicherheit noch kleineren Veränderungen bedürfen, wenn das Zykliersystem einen fortgeschritteneren Fertigungsstand erreicht hat.

Die aus Zeitgründen nicht mehr geschaffen Displayverfeinerungen, Verwendung des fertigen ADC Boosterpacks und die Weiterführung der elektronischen Schaltung, sollten in Folgearbeiten verbessert oder ausgearbeitet werden. Am besten in Verknüpfung mit einer Weiterentwicklung des Zykliersystems, um frühzeitig auf gegebene Entwicklungsstandänderungen eingehen zu können. Erst eine Zusammenfassung der beiden Teilsysteme kann weitere sinnvolle abschließende Tests ermöglichen.

Zusammenfassend ist der Gesamtaufbau eine gute Vorarbeit für die Weiterentwicklung eines Zykliersystems mit integriertem Schutzsystem, mit dem die BATSEN Arbeitsgruppe eine weitere Möglichkeit haben wird, ihre Forschungsarbeit weiterzuführen. Ein komplett fertiges Endsystem wird somit die Möglichkeit bieten, dass Studenten Batteriezyklen planen und durchführen können, ohne die Gefahr, das Gerät oder die Batterie zu zerstören.

7.2. Ausblick

Mit dem im Verlauf dieser Bachelorarbeit erstellten 19 Zoll Rack Gesamtaufbau, kann das zukünftige BATSEN Zykliersystem mit integriertem Schutzsystem weiterentwickelt werden. Das parallel dazu von der BATSEN Arbeitsgruppe entwickelte Zykliersystem kann in folgenden Projekten oder Bachelorarbeiten mit dem bereits fertigen Gerät kombiniert werden. Die nötigen Schutzfunktionen wurden in das Schutzsystem integriert, so dass ein sicheres Betreiben der Zykliersystems realisiert werden kann.

Da die Zykliersystem-Schaltung im Verlauf dieser Arbeit noch nicht fertig entwickelt wurde, ist diese Bachelorarbeit eine konzeptionelle Vorarbeit für die Endfertigung des Gesamtgerätes. Die in dieser Arbeit entwickelten Schaltungskonzepte und die dazugehörige Software sollten einer Weiterentwicklung unterzogen werden. Die weitere Entwicklung des zugehörigen noch nicht fertigen Zykliersystems wird mit Sicherheit neue Probleme aufzeigen, welche es noch abzusichern gilt. Für eine Vereinfachung der Verkabelung sollten die Schaltungskonzepte überarbeitet, erweitert und in eine SMD Platine überführt werden. Ebenfalls sollte das Anzeige- und Bedienkonzept des Displays verfeinert und erweitert werden, um mögliche Ein- und Ausgaben über das Display zu ermöglichen. Außerdem sollte die bereits fertige Booster-Pack Platine, mit ihrer ebenfalls fertigen Software zur Spannungsüberwachung der Batterie, in das System integriert werden.

Für die Finalisierung des Endproduktes ist eine Enderprobung nur mit fertigem Zykliersystemteil sinnvoll, da erst bei diesem Entwicklungsstand von einem kompletten Zykliersystem mit integriertem Schutzsystem ausgegangen werden kann.

Literaturverzeichnis

[1] DEVICES, Analog: *ADUM4154 - 5 kV, Dedicated Isolators for SPI Interfaces*. – URL http://www.analog.com/en/products/interface-isolation/isolation/spisolator/adum4154.html#product-overview. – Zugriffsdatum: 2015-11-01

[2] DEVICES, Analog: *ADUM5211 - Dual-Channel Isolators with Integrated DC-DC Converter*. – URL http://www.analog.com/en/products/interface-isolation/isolation/isopower/adum5211.html. – Zugriffsdatum: 2015-11-01

[3] ELEKTRONIK-KOMPENDIUM: *Stromfehlerschaltung (Strommesser vor Spannungsmesser)*. – URL http://www.elektronik-kompendium.de/sites/grd/0306091.htm. – Zugriffsdatum: 2015-09-02

[4] INSTRUMENTS, Texas: *TM4C1294 Connected LaunchPad*. – URL http://www.ti.com/tool/ek-tm4c1294xl. – Zugriffsdatum: 2015-11-01

[5] INTEGRATED, Maxim: *MAX11200 - ultra-low-power , high-resolution, serial output ADC*. – URL https://www.maximintegrated.com/en/products/analog/data-converters/analog-to-digital-converters/MAX11200.html. – Zugriffsdatum: 2015-11-01

[6] INTEGRATED, Maxim: *Using a UART to Implement a 1-Wire Bus Master*. – URL https://www.maximintegrated.com/en/app-notes/index.mvp/id/214. – Zugriffsdatum: 2015-09-27

[7] INTEGRATED, Maxim: *Understanding and Using Cyclic Redundancy Checks with Maxim 1-Wire and iButton Products*. 2001. – URL http://www.maximintegrated.com/en/app-notes/index.mvp/id/27

[8] MIKROCONTROLLER.NET: *Entprellung*. – URL http://www.mikrocontroller.net/articles/Entprellung. – Zugriffsdatum: 2015-09-10

[9] MIKROCONTROLLER.NET: *Wärmewiderstand*. – URL http://www.mikrocontroller.net/articles/K%C3%BChlk%C3%B6rper#W.C3.A4rmewiderstand. – Zugriffsdatum: 2015-09-02

[10] SCHULZ, Dieter: *Akkus und Ladetechniken*. Franzis, 2014. – ISBN 978-3-645-65258-2

[11] SCHUNK, Alexander: *LaTeX Das Praxisbuch*. Franzis, 2009. – ISBN 978-3-7723-6730-4

[12] STEIN, Ulrich: *Programmieren mit MATLAB*. Hanser, 2012. – ISBN 978-3-446-43243-7

[13] WIKIPEDIA: *1-wire*. – URL https://de.wikipedia.org/wiki/1-Wire. – Zugriffsdatum: 2015-09-27

[14] WIKIPEDIA: *Fehlermöglichkeits- und -einflussanalyse*. – URL https://de.wikipedia.org/wiki/FMEA. – Zugriffsdatum: 2015-10-09

[15] WIKIPEDIA: *Ladeverfahren*. – URL https://de.wikipedia.org/wiki/Ladeverfahren. – Zugriffsdatum: 2015-09-01

[16] WIKIPEDIA: *Lithium-Ionen-Akkumulator*. – URL https://de.wikipedia.org/wiki/Lithium-Ionen-Akkumulator. – Zugriffsdatum: 2015-09-01

[17] WIKIPEDIA: *Programmablaufplan*. – URL https://de.wikipedia.org/wiki/Programmablaufplan. – Zugriffsdatum: 2015-10-16

[18] WIKIPEDIA: *Rückstromladen*. – URL https://de.wikipedia.org/wiki/R%C3%BCckstromladen. – Zugriffsdatum: 2015-09-01

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

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

30. Juli 2015

Bachelorarbeit: Gregor Manzke

Überwachungscontroller für einen Hochleistungs-Batterieprüfstand

Motivation

Die Forschungsgruppe Batteriesensoren an der HAW Hamburg will Aussagen über das Batterieverhalten zukünftigen Elektrobussen treffen. Diese Prognosen sollen in die Betriebsplanung beim Busunternehmen Hamburger Hochbahn einfließen. Dafür ist es notwendig, im Vorfeld im Labor bestimmte Lade- und Entladevorgänge durchzuführen und dabei möglichst viele Messwerte, wie z.B. Zellspannung, Strom und Temperatur aufzuzeichnen.
Um diese zeitaufwändige Zyklierung der Batterie zu automatisieren, soll ein Hochleistungsprüfstand für einzelne Zellen der Batterie entstehen. Der sehr hohe maximale Lade- bzw. Entlade-Strom, der auf dem Prüfstand fließen kann, erfordert besondere Schutz- und Sicherheitsmaßnahmen.

Aufgabe

Herr Manzke erhält die Aufgabe, ein Schutzkonzept für diesen Hochleistungsprüfstand zu entwickeln. Dieses soll in einer eigenen separaten Funktionseinheit (Schutzmodul) umgesetzt werden. Es sollen hier die Ströme/Spannungen, die Temperatur und weitere relevante Größen überwacht werden. Die Funktionseinheit soll mit einen eigenen Mikrocontroller gesteuert werden. Die Fehlerfälle des Prüfstandes sollen erkannt und mit steuerungstechnischen Maßnahmen abgesichert werden. Ebenfalls Teil dieser Aufgabe ist es, eine Kommunikation des Schutzmoduls und des Controllers des Prüfstandes zu implementieren. Ein wesentlicher Teil sind praktische Aufgaben der Planung, Umsetzung und Erprobung im Zusammenhang mit dem Aufbau des Prüfstandes.
Die Bachelorarbeit soll folgende Arbeitspakete umfassen:

1) **Einführung, Erfassung der Vorarbeiten, Analyse der Rahmenbedingungen**
   - Einarbeitung in die Projektzielsetzung und Aufgabe der Bachelorarbeit
   - Studium der vorangegangenen Arbeiten zum Zyklierprüfstand und zum E-Busprojekt
   - Abschätzung von Rahmenbedingungen für den Prüfstand und das Schutzmodul
2) **Schutzkonzept des Gesamtsystems**
   - Fehlermöglichkeits- und Einflussanalyse (FMEA) für den Prüfstand
   - Auswahl und Dokumentation von geeigneten Schutzfunktionen, sofern passend dazu die Bewertung von Alternativen und Prioritäten
3) **Planung und Umsetzung des Schutzkonzeptes**
   - Entwurf von Software-Modulen auf dem Mikrocontroller für das Schutzmodul (Spannungsüberwachung, Temperaturregelung)
   - Aufbau von potentialgetrennten Messschaltungen und Einbindung in die Controller-Software
   - Entwurf der Erweiterungshardware z.B. als eine Booster-Pack-Schaltung für das Controllerboard

4) **Kommunikation zwischen Schutzcontroller und Prüfstand sowie mit weiteren Schnittstellen und Anzeigefunktionen**

   - Kommunikationskonzept der beiden Controller erstellen und schrittweise umsetzen
   - Bedienkonzept durch Display, LED, Taster, u.a. sowie Umsetzung in Hard- und Software
   - Sicherstellung der Schnittstellen-Entkopplung

5) **Aufbau des Prüfstandes, insbesondere des Schutzmoduls**

   - Konstruktion des mechanischen Aufbaus und der Gehäusefronten mit Hilfe eines CAD-Programms
   - Auswahl der Komponenten Lüfter, Kühlkörper, Widerstände, Spannungsversorgung
   - Planung der Abführung der Verlustleistung
   - Entwurf und Umsetzung der Leitungsführung der Hochstromkabel
   - Umsetzung der Sicherheitsfunktionen des Schutzkonzeptes

6) **Aufbau, Inbetriebnahme und Erprobung**

   - Mechanischer Aufbau, Teilefertigung und Montage, inkl. Gehäuse und Peripherie
   - Anschlussführung und Inbetriebnahme der Messplatinen und der Sensoren
   - Inbetriebnahme des Prüfstandes hinsichtlich der Schutzfunktionen
   - Funktionstests der Schutzfunktionen mit provozierten Fehlerfällen
   - Exemplarischer Testbetrieb der Regelfunktionen der Platinen im Labor, Behebung kleiner Software- und Hardware-Fehler
   - Aufbau des neuen Gerätes und der Verkabelung für die größeren Ströme
   - Inbetriebnahme des neuen Gerätes

7) **Gesamtbewertung und Fazit**

   - Diskussion der Lösung in Bezug auf die Zielsetzung
   - Darstellung offener Punkte und weitergehender Verbesserungsmöglichkeiten
   - Bewertung der Nutzbarkeit und Weiterführung

Dokumentation

Die Vorarbeiten und die kommerziellen Unterlagen sind zielgerichtet zu recherchieren und das Ergebnis zusammengefasst darzustellen. Die gewählten Lösungen und die Funktionsweise sind gut nachvollziehbar zu dokumentieren. Die gesetzten Rahmenbedingungen, die Grundkonzeption, auftretende Probleme und wesentliche Entwurfsentscheidungen sollen beschrieben werden. Die Erprobungsergebnisse sind in exemplarischem Umfang zu erfassen und auszuwerten.

A. Anhang

A.1. Quellcode: 1-wire Testprogramm

Listing A.1: 1-wire Testprogramm - main.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  //#define TARGET_IS_TM4C123_RA1
4  #include "inc/hw_memmap.h"
5  #include "inc/hw_types.h"
6  #include "driverlib/gpio.h"
7  #include "driverlib/pin_map.h"
8  #include "driverlib/sysctl.h"
9  #include "driverlib/uart.h"
10 #include "inc/hw_ints.h"
11 #include "utils/uartstdio.h"
12
13 #include "1_wire_crc.h"
14 //#include "driverlib/rom.h"
15
16 // hier wird UART3 verwendet U3RX->PA4 und U3TX->PA5
17
18 // ROM-Kommandos
19 #define SEARCH_ROM          0xf0 // identifiziert ROM Codes der verfügbaren Slaves im Bussystem.
20                                  // Auch die Anzahl ist bestimmbar
21
22 #define READ_ROM            0x33 // Kommando hat den selben Effekt wie SEARCH_ROM, kann aber nur genutzt werden,
23                                  // wenn nur ein Slave im Bussystem verfügbar ist. Sonst würden Datenkollisionen auftreten
24
25 #define MATCH_ROM           0x55 // dieses Kommando, gefolgt von einem 64-bit ROM-Code wird benutzt um ein bestimmten Slave
26                                  // im Bussystem anzusprechen. Wenn der Slave-Code mit dem empfangenen überein stimmt, wird
27                                  // er als einziger Slave im BUS antworten. Die anderen warten ab dann bis eine Reset Sequenz kommt.
28
29 #define SKIP_ROM            0xcc // wird benutzt um alle BUS Teilnehmer zur gleichen Zeit zu adressieren. Dies kann nützlich sein,
30                                  // um zum Beispiel das Kommando einer Temperaturabfrage zu starten.
31                                  // (BSP: CONVERT_TEMP, 0x44 ...wenn nur ein TN im BUS ist.)
32
33 #define ALARM_SEARCH        0xec // fast das Gleiche wie wie SEARCH_ROM,
34                                  // nur das hier nur die Teilnehmer mit einer gesetzten ALARM_FLAG antworten
35
```

```
36 // FUNKTIONS-Kommandos
37 #define CONVERT_TEMP        0x44 // startet eine Temperaturmessung. Die Temperatur wird in den ersten
       beiden BYTES des SCRATCHPAD gespeichert.
38                                   // Die "Konvertierungs"-Zeit ist Auflösungsabhängig
39                                   // ACHTUNG: Wenn man ein READ während eines Konvertierungsprozesses
                                        schickt, antwortet der Teilnehmer mit 0 (low).
40                                   // Wenn der Prozess abgschlossen ist, antwortet der Teilnehmer mit 1
                                        (high).
41                                   // (nur verfügbar, wenn nicht "parasitär"-Versorgt.)
42
43 #define WRITE_SCRATCHPAD    0x4e //
44
45 #define READ_SCRATCHPAD     0xbe //
46
47 #define COPY_SCRATCHPAD     0x48 //
48
49 #define RECALL_E_2          0xb8 //
50
51 #define READ_POWER_SUPPLY   0xb4 //
52
53
54 uint32_t ui32SysClkFreq;
55 // Hilfsvariable zum Speichern der SystemClockFrequenz
56
57 char temperature_MSB;
58 char temperature_LSB;
59 char buffer;
60 int temperature_celsius;
61 int PROGRAMM_COUNTER=0;
62 char DEVICE_PULSE;
63 int DEVICE_AVAILABLE=0; // speichern ob DEVICE Vorhanden ist 0=NEIN, 1=JA
64 int k=0;
65
66 //nur zur Hilfe
67 char B2_TH_Register;
68 char B3_TL_Register;
69 char B4_Config_Register;
70 char B5_Reserved_0xFF;
71 char B6_Reserved_0x0C;
72 char B7_Reserved_0x10;
73 char B8_CRC;
74
75
76 //////////////////////TEST
77 unsigned long ulIdx, ulValue, ulValue2;
78 unsigned char pucData[256];
79 //////////////////////TEST
80
81
82 // 8 chars * 8 Bit = 64 Bit
83 char ROM_CODE[8];
84
85 void delayMS(int ms) {
86     //ROM_SysCtlDelay( (ROM_SysCtlClockGet()/(3*1000))*ms ) ;  // more accurate
87     SysCtlDelay( (ui32SysClkFreq/(3*1000))*ms ) ;  // less accurate
88 }
89
90
91 void one_wire_reset(void)
92 {
```

```
93      //--1-WIRE Reset ------------------------------------------------------------
94      // (9600 BAUD)
95      // sende 0xF0 = 1111 0000 über TX, wenn 0xF0 = 1111 0000 über RX zurückkommt,
96      // dann ist kein Device vorhanden
97
98          UARTConfigSetExpClk(UART3_BASE, ui32SysClkFreq, 9600,(UART_CONFIG_WLEN_8 |
                UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
99          // Parameter für UART0 Schnittstelle einstellen: Baudrate=9600, 8-1-N-N
100
101         UARTCharPut(UART3_BASE, 0xF0);
102
103         DEVICE_PULSE = UARTCharGet(UART3_BASE);
104
105         if(DEVICE_PULSE == 0xF0)
106         {
107             DEVICE_AVAILABLE = 0; // NEIN
108         }
109         else if(DEVICE_PULSE == 0xE0)
110         {
111             DEVICE_AVAILABLE = 1; // JA
112             // wenn Devices vorhanden, dann auf 1-wire Daten-Geschwindigkeits-Timing
113             // Parameter für UART3 Schnittstelle einstellen: Baudrate=115200, 8-1-N-N
114             UARTConfigSetExpClk(UART3_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
115         }
116 }
117
118
119 void one_wire_bit(int bit)
120 {
121     if(bit==0)
122     {
123         UARTCharPut(UART3_BASE, 0x00);
124     }
125     else if(bit==1)
126     {
127         UARTCharPut(UART3_BASE, 0xFF);
128     }
129 }
130
131
132 void one_wire_write_byte(int byte)
133 {
134     int i;
135     for(i=0 ; i < 8 ; i++)
136     {
137     // sendet immer das aktuell letzte Bit über den Bus
138     one_wire_bit(byte & 0x01);
139     byte = byte >> 1;
140     }
141 }
142
143 char one_wire_read_byte(void)
144 {
145     int i;
146     char byte = 0x00;
147     for(i=0 ; i < 8 ; i++)
148     {
149         buffer=0;
150         // 8 MAL !!!!
```

```
151         // auf den BUS 1111 1111 schreiben damit der 1-wire weiss das er zu antworten hat
152         UARTCharPut(UART3_BASE, 0xFF);
153
154         // Byte zurückerhalten:
155         buffer = UARTCharGet(UART3_BASE);
156
157         //danach die zurückerhaltene Byte auswerten:
158         if(buffer==0xFC) // bedeutet eine 0 vom 1-wire Slave (FC = 1111 1100)
159         {
160             // eine 0 in das MSB schreiben
161             byte = byte | 0b00000000;
162             //oder Verknüpfung BSP: 1011 | 0000 = 1011
163         }
164         else if(buffer==0xFF) // bedeutet eine 1 vom 1-wire Slave
165         {
166             // eine 1 in das MSB schreiben
167             byte = byte | 0b10000000;
168             //oder Verknüpfung BSP: 0000 | 0001 = 0001
169         }
170
171         if(i<7)
172         {
173         // nach links schieben
174         // aber nur bei i=0 bis i=6
175         byte = byte >> 1;
176         }
177     }
178     return byte;
179 }
180
181
182
183 int main(void)
184 {
185     ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
            SYSCTL_CFG_VCO_480), 120000000);
186     // stellen der SystemClockFrequenz auf 120 MHz und speichern in Hilfsvariable "ui32SysClkFreq
            "=120.000.000
187
188     SysCtlPeripheralEnable(SYSCTL_PERIPH_UART3);
189     // aktiviert UART0 Modul
190     // siehe "Pin Mux Utility" bei der "aktivierung" von UART3
191     // genutzt soll hier werden: Port_A, da dort RX(an PA4) und TX(an PA5) vorhanden sind
192
193     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
194     // aktiviert GPIO Ports an Port_A
195     // benötigt für UART0 --> siehe: PINMUX --> (PA4_U3RX) und (PA5_U3TX)
196
197     GPIOPinConfigure(GPIO_PA4_U3RX);
198     // GPIO-Pin-PA4 als TX (Transmit) Pin einstellen
199
200     GPIOPinConfigure(GPIO_PA5_U3TX);
201     // GPIO-Pin-PA5 als RX (Recieve) Pin einstellen
202
203     GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_4 | GPIO_PIN_5);
204     // GPIO PINs von Port_A (PIN4 und PIN5) als UART PINs verwenden
205
206 //---------- UART0 für die Ausgabe übers Terminal ----------
207     SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
```

```
208     // aktiviert UART0 Modul: die UART0 Ein-/Ausgänge sind über PORT_B & PORT_H & PORT_K & PORT_P
            verteilt
209     // siehe "Pin Mux Utility" bei der "aktivierung" von UART0
210     // genutzt soll hier werden: Port_A, da dort TX(an PA0) und RX(an PA1) vorhanden sind
211
212     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
213     // aktiviert GPIO Ports an Port_A
214     // benötigt für UART0 --> siehe: PINMUX --> (PA0_U0RX) und (PA1_U0TX)
215
216     GPIOPinConfigure(GPIO_PA0_U0RX);
217     // GPIO-Pin-PA0 als TX (Transmit) Pin einstellen
218
219     GPIOPinConfigure(GPIO_PA1_U0TX);
220     // GPIO-Pin-PA1 als RX (Recieve) Pin einstellen
221
222     GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
223     // GPIO PINs von Port_A (PIN0 und PIN1) als UART PINs verwenden
224
225     UARTConfigSetExpClk(UART0_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE
            | UART_CONFIG_PAR_NONE));
226     // Parameter für UART0 Schnittstelle einstellen: Baudrate=115200, 8-1-N-N
227
228     UARTStdioConfig(0, 115200, ui32SysClkFreq);
229     // hier wird für die "UARTprintf" Funktion der UART Standard auf UART0 gestellt
230     // damit "UARTprintf" weiß, wo es die Zeichen hinschicken soll.....
231
232 //---------- UART0 für die Ausgabe übers Terminal ENDE----------------------------
233
234     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
235     // aktiviert GPIO Ports an Port_N
236
237     GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
238     // definiert an PORT_N --> PIN0 & PIN1 als Output Pins
239     // hier für die LEDs genutzt
240 //--------------------------------------------------------------------------------
241
242     while (1)
243     {
244         // Funktionsaufruf
245         one_wire_reset();
246
247         one_wire_write_byte(0xCC); // 0xCC = 1100 1100
248         one_wire_write_byte(0x44); // 0x44 = 0100 0100
249
250         // mindestens 750ms warten
251         delayMS(800);
252
253         // Funktionsaufruf
254         one_wire_reset();
255
256         one_wire_write_byte(0xCC); // 0xCC = 1100 1100
257         one_wire_write_byte(0xBE); // 0xBE = 1011 1110
258
259         // CODE, der den UART quasi resettet, sonst liest der RX nur den Reste der vorigen Zeile ein.
260         UARTConfigSetExpClk(UART3_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
261
262         //Bytes auslesen
263         temperature_LSB = one_wire_read_byte();
264         temperature_MSB = one_wire_read_byte();
```

```
265
266         B2_TH_Register = one_wire_read_byte();
267         B3_TL_Register = one_wire_read_byte();
268         B4_Config_Register = one_wire_read_byte();
269         B5_Reserved_0xFF = one_wire_read_byte();
270         B6_Reserved_0x0C = one_wire_read_byte();
271         B7_Reserved_0x10 = one_wire_read_byte();
272         B8_CRC = one_wire_read_byte();
273
274         // Funktionsaufruf
275         one_wire_reset();
276         //one_wire_write_byte(0xCC); // 0xCC = 1100 1100
277         one_wire_write_byte(0x33); // 0xBE = 1011 1110
278
279         // CODE, der den UART quasi resettet, sonst liest der RX nur den Reste der vorigen Zeile ein.
280         UARTConfigSetExpClk(UART3_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
281
282
283
284         for(k=0;k<8;k++)
285         {
286             ROM_CODE[k] = one_wire_read_byte();
287         }
288
289         temperature_celsius = ((temperature_MSB * 256) + temperature_LSB) / 16 ;
290
291
292
293 //////////////////////////////////////////////////TEST
294         //
295         // Fill pucData with some data.
296         //
297
298         pucData[0] = temperature_LSB;
299         pucData[1] = temperature_MSB;
300         pucData[2] = B2_TH_Register;
301         pucData[3] = B3_TL_Register;
302         pucData[4] = B4_Config_Register;
303         pucData[5] = B5_Reserved_0xFF;
304         pucData[6] = B6_Reserved_0x0C;
305         pucData[7] = B7_Reserved_0x10;
306
307         ulValue2 = CRC8_1_wire(0, pucData, 8);
308
309         for(k=0;k<7;k++)
310         {
311             pucData[k] = ROM_CODE[k];
312         }
313
314         ulValue = CRC8_1_wire(0, pucData, 7);
315
316 //////////////////////////////////////////////////TEST
317
318
319 //-----// Periodische Ausgabe der Daten auf UART0 -> Terminal Programm
320
321         UARTprintf("\033[2J\033[H"); // lösche Bildschirm, lege Cursor auf Position (0,0)
322         UARTprintf("|=========================================================================|\r\n");
323         UARTprintf("|        Testprogramm fuer einzelne DS18B20 Temeratursensoren             |\r\n");
```

```
324         UARTprintf("|=============================================================================|\r\n");
325         UARTprintf("| Programmdurchlauf: %6d  |  1-wire Slave vorhanden: %1d [0=NEIN/1=JA] |\r\n",
                PROGRAMM_COUNTER, DEVICE_AVAILABLE);
326         UARTprintf("|=============================================================================|\r\n");
327         UARTprintf("|           Gemessene Temperatur: %8d Grad Celsius                  |\r\n",
                temperature_celsius);
328         UARTprintf("|=============================================================================|\r\n");
329         UARTprintf("| Byte0: Temperature_LSB             -->    HEX: %2X       DEZ: %3d       |\r\n",
                temperature_LSB, temperature_LSB);
330         UARTprintf("|=============================================================================|\r\n");
331         UARTprintf("| Byte1: Temperature_MSB             -->    HEX: %2X       DEZ: %3d       |\r\n",
                temperature_MSB, temperature_MSB);
332         UARTprintf("|=============================================================================|\r\n");
333         UARTprintf("| Byte2: TH_Register_or_User_Byte_1 -->    HEX: %2X       DEZ: %3d       |\r\n",
                B2_TH_Register, B2_TH_Register);
334         UARTprintf("|=============================================================================|\r\n");
335         UARTprintf("| Byte3: TL_Register_or_User_Byte_2 -->    HEX: %2X       DEZ: %3d       |\r\n",
                B3_TL_Register, B3_TL_Register);
336         UARTprintf("|=============================================================================|\r\n");
337         UARTprintf("| Byte4: Configuration_Register     -->    HEX: %2X       DEZ: %3d       |\r\n",
                B4_Config_Register, B4_Config_Register);
338         UARTprintf("|=============================================================================|\r\n");
339         UARTprintf("| Byte5: Reserved_(0xFF)            -->    HEX: %2X       DEZ: %3d       |\r\n",
                B5_Reserved_0xFF, B5_Reserved_0xFF);
340         UARTprintf("|=============================================================================|\r\n");
341         UARTprintf("| Byte6: Reserved_(0x0C)            -->    HEX: %2X       DEZ: %3d       |\r\n",
                B6_Reserved_0x0C, B6_Reserved_0x0C);
342         UARTprintf("|=============================================================================|\r\n");
343         UARTprintf("| Byte7: Reserved_(0x10)            -->    HEX: %2X       DEZ: %3d       |\r\n",
                B7_Reserved_0x10, B7_Reserved_0x10);
344         UARTprintf("|=============================================================================|\r\n");
345         UARTprintf("| Byte8: CRC                        -->    HEX: %2X       DEZ: %3d       |\r\n",
                B8_CRC, B8_CRC);
346         UARTprintf("|=============================================================================|\r\n");
347         UARTprintf("\r\n");
348         UARTprintf("|=============================================================================|\r\n");
349         UARTprintf("| ROM Code:  [CRC:] %2X  [48-BIT-Nr:] %2X %2X %2X %2X %2X %2X  [Fam-Code:] %2X
                |\r\n", ROM_CODE[7], ROM_CODE[6], ROM_CODE[5], ROM_CODE[4], ROM_CODE[3], ROM_CODE[2],
                ROM_CODE[1], ROM_CODE[0]);
350         UARTprintf("|=============================================================================|\r\n");
351         UARTprintf("\r\n");
352         UARTprintf("|=============================================================================|\r\n");
353         UARTprintf("|  Temp-CRC: %2X <-> %2X [CRC-Check]  |  ROM-CRC: %2X <-> %2X [CRC-Check]    |\r\n
                ", B8_CRC, ulValue2, ROM_CODE[7], ulValue);
354         UARTprintf("|=============================================================================|\r\n");
355         UARTprintf("\r\n");
356
357
358         if(B8_CRC==ulValue2)
359         {
360         UARTprintf("Temp-CRC: CRC-Check: OK :-) \r\n");
361         }
362         else
363         {
364         UARTprintf("Temp-CRC: CRC-Check: Error !!! \r\n");
365         }
366
367
368         if(ROM_CODE[7]==ulValue)
369         {
```

```
370             UARTprintf(" ROM-CRC: CRC-Check: OK :-) \r\n");
371             }
372             else
373             {
374             UARTprintf(" ROM-CRC: CRC-Check: Error !!! \r\n");
375             }
376
377     //-----// Terminal Ausgabe Ende
378
379             // 1 s warten
380             delayMS(1000);
381
382             PROGRAMM_COUNTER++;
383
384         }
385     }
```

A.2. MATLAB Codes

Listing A.2: MATLAB-File für die Messung der Ladewiderstände

```
1  % Die CSV-Datei einlesen
2  a = csvread('Messung_3.csv');
3
4  % Zeit in Sekunden
5  % Die Werte der ersten Spalte
6  sekunden = a(:,1);
7
8  % Tastgrad in %
9  tastgrad = a(:,2);
10
11 % RPM des Lüfters
12 rpm      = a(:,3);
13
14 % Temperatur 1
15 temp_1   = a(:,4);
16
17 % Temperatur 2
18 temp_2   = a(:,5);
19
20 % Temperatur 3
21 temp_3   = a(:,6);
22
23 % Temperatur 4
24 temp_4   = a(:,7);
25
26 % Temperatur 5
27 temp_5   = a(:,8);
28
29 % Temperatur 6
30 temp_6   = a(:,9);
31
32 % Spannung über den beiden parallelen Widerständen
33 volt     = a(:,10);
34
35 r = 0.0556;
36
37 % Strom durch die beiden parallelen Widerständen
```

```
38  i = volt / r;
39
40  % Leistung, die an den Widerständen abfällt
41  p = (volt.* volt) / r;
42
43  % Berechnung von delta-T
44  delta_temp_1_3 = abs((temp_3/100) - (temp_1/100));
45
46  % Wärmewiderstand in K/W
47  R_0 = (delta_temp_1_3./p) ;
48
49  figure(1);
50
51
52
53  hold on
54  subplot(4,1,1); % Zeilen, Spalten, Plotnummer
55  [ax,p1,p2] = plotyy(sekunden,volt,sekunden,i);
56  set(p1(1), 'LineWidth', 2);
57  set(p2(1), 'LineWidth', 2);
58  xlim(ax(1),[0 max(sekunden)]);
59  xlim(ax(2),[0 max(sekunden)]);
60  %ylim(ax(2),[0 22]);
61  title('Spannung / Strom');
62  xlabel('Zeit [s]');
63  ylabel(ax(1),'Spannung [V]') % label left y-axis
64  ylabel(ax(2),'Strom [A]') % label right y-axis
65  grid(ax(1),'on') % grid x-axis ON
66  hold off
67
68  hold on
69  subplot(4,1,2); % Zeilen, Spalten, Plotnummer
70  [ax,p1,p2] = plotyy(sekunden,tastgrad,sekunden,rpm);
71  set(p1(1), 'LineWidth', 2);
72  set(p2(1), 'LineWidth', 2);
73  xlim(ax(1),[0 max(sekunden)]);
74  xlim(ax(2),[0 max(sekunden)]);
75  ylim(ax(1),[0 110]);
76  ylim(ax(2),[0 2200]);
77  title('Tastgrad / RPM');
78  xlabel('Zeit [s]');
79  ylabel(ax(1),'Tastgrad') % label left y-axis
80  ylabel(ax(2),'RPM') % label right y-axis
81  grid(ax(1),'on') % grid x-axis ON
82  hold off
83
84  % Zeilen, Spalten, Plotnummer
85
86  subplot(4,1,3);
87  plot(sekunden,temp_1/100,'Color','black');
88  hold on
89  plot(sekunden,temp_2/100,'Color','blue');
90  plot(sekunden,temp_3/100,'Color','green');
91  plot(sekunden,temp_4/100,'Color',[1,0.4,0.6]);
92  plot(sekunden,temp_5/100,'Color',[0.8,0.3,0.5]);
93  plot(sekunden,temp_6/100,'Color',[0.9,0.1,0.7]);
94  xlim([0 max(sekunden)]);
95  title('Temperaturen');
96  xlabel('Zeit [s]');
97  ylabel('Celsius');
```

```
98  legend('T1','T2','T3','T4','T4','T6');
99  hold off
100
101 % Wärmewiderstand in K/W
102 subplot(4,1,4);
103 plot(sekunden,R_0,'Color','blue');
104 hold on
105 xlim([0 max(sekunden)]);
106 ylim([0 1.5]);
107 title('Wärmewiderstand');
108 xlabel('Zeit [s]');
109 ylabel('K/W');
110 hold off
```

Listing A.3: MATLAB-File für die Messung der Lastwiderstände

```
1  % Die CSV-Datei einlesen
2  a = csvread('Messung_1.csv');
3
4  % Zeit in Sekunden
5  % Die Werte der ersten Spalte
6  sekunden = a(:,1);
7
8  % Tastgrad in %
9  tastgrad = a(:,2);
10
11 % RPM des Lüfters
12 rpm      = a(:,3);
13
14 % Temperatur 1
15 temp_1   = a(:,4);
16
17 % Temperatur 2
18 temp_2   = a(:,5);
19
20 % Temperatur 3
21 temp_3   = a(:,6);
22
23 % Temperatur 4
24 temp_4   = a(:,7);
25
26 % Temperatur 5
27 temp_5   = a(:,8);
28
29 % Temperatur 6
30 temp_6   = a(:,9);
31
32 % Spannung über den beiden parallelen Widerständen
33 volt     = a(:,10);
34
35 r = 0.09;
36
37 % Strom durch die beiden parallelen Widerständen
38 i = volt / r;
39
40 % Leistung, die an den Widerständen abfällt
41 p = (volt.* volt) / r;
42
43 % Berechnung von delta-T
44 delta_temp_1_3 = abs((temp_3/100) - (temp_1/100));
```

```
45
46 % Wärmewiderstand in K/W
47 R_0 = (delta_temp_1_3./p) ;
48
49 figure(1);
50
51
52
53 hold on
54 subplot(4,1,1); % Zeilen, Spalten, Plotnummer
55 [ax,p1,p2] = plotyy(sekunden,volt,sekunden,i);
56 set(p1(1), 'LineWidth', 2);
57 set(p2(1), 'LineWidth', 2);
58 xlim(ax(1),[0 max(sekunden)]);
59 xlim(ax(2),[0 max(sekunden)]);
60 %ylim(ax(2),[0 22]);
61 title('Spannung / Strom');
62 xlabel('Zeit [s]');
63 ylabel(ax(1),'Spannung [V]') % label left y-axis
64 ylabel(ax(2),'Strom [A]') % label right y-axis
65 grid(ax(1),'on') % grid x-axis ON
66 hold off
67
68 hold on
69 subplot(4,1,2); % Zeilen, Spalten, Plotnummer
70 [ax,p1,p2] = plotyy(sekunden,tastgrad,sekunden,rpm);
71 set(p1(1), 'LineWidth', 2);
72 set(p2(1), 'LineWidth', 2);
73 xlim(ax(1),[0 max(sekunden)]);
74 xlim(ax(2),[0 max(sekunden)]);
75 ylim(ax(1),[0 110]);
76 ylim(ax(2),[0 2200]);
77 title('Tastgrad / RPM');
78 xlabel('Zeit [s]');
79 ylabel(ax(1),'Tastgrad') % label left y-axis
80 ylabel(ax(2),'RPM') % label right y-axis
81 grid(ax(1),'on') % grid x-axis ON
82 hold off
83
84 % Zeilen, Spalten, Plotnummer
85
86 subplot(4,1,3);
87 plot(sekunden,temp_1/100,'Color','black');
88 hold on
89 plot(sekunden,temp_2/100,'Color','blue');
90 plot(sekunden,temp_3/100,'Color','green');
91 plot(sekunden,temp_4/100,'Color',[1,0.4,0.6]);
92 plot(sekunden,temp_5/100,'Color',[0.8,0.3,0.5]);
93 plot(sekunden,temp_6/100,'Color',[0.9,0.1,0.7]);
94 xlim([0 max(sekunden)]);
95 title('Temperaturen');
96 xlabel('Zeit [s]');
97 ylabel('Celsius');
98 legend('T1','T2','T3','T4','T4','T6');
99 hold off
100
101 % Wärmewiderstand in K/W
102 subplot(4,1,4);
103 plot(sekunden,R_0,'Color','blue');
104 hold on
```

```
105 xlim([0 max(sekunden)]);
106 ylim([0 1.5]);
107 title('Wärmewiderstand');
108 xlabel('Zeit [s]');
109 ylabel('K/W');
110 hold off
```

Listing A.4: MATLAB-File für die Volllast-Messung

```
1  % Die CSV-Datei einlesen
2  a = csvread('Messung_VL.csv');
3
4  % Zeit in Sekunden
5  % Die Werte der ersten Spalte
6  sekunden = a(:,1);
7
8  % Temperatur 1
9  temp_1   = a(:,2);
10
11 % Temperatur 2
12 temp_2   = a(:,3);
13
14 % Temperatur 3
15 temp_3   = a(:,4);
16
17 % Temperatur 4
18 temp_4   = a(:,5);
19
20 % Temperatur 5
21 temp_5   = a(:,6);
22
23 % Tastgrad L1 in %
24 tastgrad_1 = a(:,7);
25
26 % Tastgrad L1 in %
27 tastgrad_2 = a(:,8);
28
29 % Tastgrad L1 in %
30 tastgrad_3 = a(:,9);
31
32 % RPM des Lüfters 1
33 rpm_1      = a(:,10);
34
35 % RPM des Lüfters 2
36 rpm_2      = a(:,11);
37
38 % RPM des Lüfters 3
39 rpm_3      = a(:,12);
40
41 % ADC --> Strom in A
42 ampere      = a(:,13);
43
44 % System Status Text
45 % 1 = Init / 2 = Run / 3 = Soft-Stop
46 status_text      = a(:,14);
47
48 % System Status Text
49 % 0 = AUS / 1 = AN
50 status_power      = a(:,15);
51
```

```
52
53 % Berechnung von delta-T
54 delta_temp_L1 = abs((temp_5/100) - (temp_1/100));
55 delta_temp_L2 = abs((temp_5/100) - (temp_2/100));
56 delta_temp_L3 = abs((temp_5/100) - (temp_3/100));
57
58 % Widerstand Gesamt (pro Widerstandspaket 0.080 Ohm)
59 r = 0.240;
60
61 % Leistung an allen 3 Widerständen
62 % P = I^2 * R
63 p = ( ampere.* ampere ) * r;
64
65 % Wärmewiderstand in K/W
66 R_0_L1 = (delta_temp_L1./(p/3)) ;
67 R_0_L2 = (delta_temp_L2./(p/3)) ;
68 R_0_L3 = (delta_temp_L3./(p/3)) ;
69
70 figure(1);
71
72 hold on
73 %subplot(2,1,1); % Zeilen, Spalten, Plotnummer
74 plot(sekunden,ampere+6,'Color','red');
75 xlim([0 max(sekunden)]);
76 ylim([-1 40]);
77 title('Strom');
78 xlabel('Zeit [s]');
79 ylabel('Strom [A]');
80 grid('on');
81 hold off
82
83 % Lüfter 1
84 % Temperatur
85 % T / RPM
86 % Wärmewiderstand
87 figure(2);
88
89 hold on
90 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
91 plot(sekunden,temp_1/100,'Color','blue', 'LineWidth', 2);
92 hold on
93 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
94 xlim([0 max(sekunden)]);
95 ylim([18 60]);
96 title('Lüfter 1 - Temperatur');
97 xlabel('Zeit [s]');
98 ylabel('Celsius');
99 legend('Lüfter 1','Raum');
100 grid('on');
101 hold off
102
103 hold on
104 subplot(3,1,2); % Zeilen, Spalten, Plotnummer
105 [ax,p1,p2] = plotyy(sekunden,tastgrad_1,sekunden,rpm_1);
106 set(p1(1), 'LineWidth', 2);
107 set(p2(1), 'LineWidth', 2);
108 xlim(ax(1),[0 max(sekunden)]);
109 xlim(ax(2),[0 max(sekunden)]);
110 ylim(ax(1),[0 110]);
111 ylim(ax(2),[0 2400]);
```

```
112 title('Lüfter 1 - Tastgrad / RPM');
113 xlabel('Zeit [s]');
114 ylabel(ax(1),'Tastgrad') % label left y-axis
115 ylabel(ax(2),'RPM') % label right y-axis
116 grid(ax(1),'on') % grid x-axis ON
117 hold off
118
119 % Wärmewiderstand in K/W
120 subplot(3,1,3);
121 plot(sekunden,R_0_L1,'Color','blue');
122 hold on
123 xlim([0 max(sekunden)]);
124 ylim([0 1]);
125 title('Lüfter 1 - Wärmewiderstand');
126 xlabel('Zeit [s]');
127 ylabel('K/W');
128 hold off
129
130 % Lüfter 2
131 % Temperatur
132 % T / RPM
133 % Wärmewiderstand
134 figure(3);
135
136 hold on
137 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
138 plot(sekunden,temp_2/100,'Color','blue', 'LineWidth', 2);
139 hold on
140 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
141 xlim([0 max(sekunden)]);
142 ylim([18 60]);
143 title('Lüfter 2 - Temperatur');
144 xlabel('Zeit [s]');
145 ylabel('Celsius');
146 legend('Lüfter 2','Raum');
147 grid('on');
148 hold off
149
150 hold on
151 subplot(3,1,2); % Zeilen, Spalten, Plotnummer
152 [ax,p1,p2] = plotyy(sekunden,tastgrad_2,sekunden,rpm_2);
153 set(p1(1), 'LineWidth', 2);
154 set(p2(1), 'LineWidth', 2);
155 xlim(ax(1),[0 max(sekunden)]);
156 xlim(ax(2),[0 max(sekunden)]);
157 ylim(ax(1),[0 110]);
158 ylim(ax(2),[0 2400]);
159 title('Lüfter 2 - Tastgrad / RPM');
160 xlabel('Zeit [s]');
161 ylabel(ax(1),'Tastgrad') % label left y-axis
162 ylabel(ax(2),'RPM') % label right y-axis
163 grid(ax(1),'on') % grid x-axis ON
164 hold off
165
166 % Wärmewiderstand in K/W
167 subplot(3,1,3);
168 plot(sekunden,R_0_L2,'Color','blue');
169 hold on
170 xlim([0 max(sekunden)]);
171 ylim([0 1]);
```

```
172 title('Lüfter 2 - Wärmewiderstand');
173 xlabel('Zeit [s]');
174 ylabel('K/W');
175 hold off
176
177 % Lüfter 3
178 % Temperatur
179 % T / RPM
180 % Wärmewiderstand
181 figure(4);
182
183 hold on
184 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
185 plot(sekunden,temp_3/100,'Color','blue', 'LineWidth', 2);
186 hold on
187 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
188 xlim([0 max(sekunden)]);
189 ylim([18 60]);
190 title('Lüfter 3 - Temperatur');
191 xlabel('Zeit [s]');
192 ylabel('Celsius');
193 legend('Lüfter 3','Raum');
194 grid('on');
195 hold off
196
197 hold on
198 subplot(3,1,2); % Zeilen, Spalten, Plotnummer
199 [ax,p1,p2] = plotyy(sekunden,tastgrad_3,sekunden,rpm_3);
200 set(p1(1), 'LineWidth', 2);
201 set(p2(1), 'LineWidth', 2);
202 xlim(ax(1),[0 max(sekunden)]);
203 xlim(ax(2),[0 max(sekunden)]);
204 ylim(ax(1),[0 110]);
205 ylim(ax(2),[0 2400]);
206 title('Lüfter 3 - Tastgrad / RPM');
207 xlabel('Zeit [s]');
208 ylabel(ax(1),'Tastgrad') % label left y-axis
209 ylabel(ax(2),'RPM') % label right y-axis
210 grid(ax(1),'on') % grid x-axis ON
211 hold off
212
213 % Wärmewiderstand in K/W
214 subplot(3,1,3);
215 plot(sekunden,R_0_L3,'Color','blue');
216 hold on
217 xlim([0 max(sekunden)]);
218 ylim([0 1]);
219 title('Lüfter 3 - Wärmewiderstand');
220 xlabel('Zeit [s]');
221 ylabel('K/W');
222 hold off
```

Listing A.5: MATLAB-File für die Messung der Lüfteransteuerung

```
1 % Die CSV-Datei einlesen
2 a = csvread('Messung_LS.csv');
3
4 % Zeit in Sekunden
5 % Die Werte der ersten Spalte
6 sekunden = a(:,1);
```

```
7
8  % Temperatur 1
9  temp_1    = a(:,2);
10
11 % Temperatur 2
12 temp_2    = a(:,3);
13
14 % Temperatur 3
15 temp_3    = a(:,4);
16
17 % Temperatur 4
18 temp_4    = a(:,5);
19
20 % Temperatur 5
21 temp_5    = a(:,6);
22
23 % Tastgrad L1 in %
24 tastgrad_1 = a(:,7);
25
26 % Tastgrad L1 in %
27 tastgrad_2 = a(:,8);
28
29 % Tastgrad L1 in %
30 tastgrad_3 = a(:,9);
31
32 % RPM des Lüfters 1
33 rpm_1       = a(:,10);
34
35 % RPM des Lüfters 2
36 rpm_2       = a(:,11);
37
38 % RPM des Lüfters 3
39 rpm_3       = a(:,12);
40
41 % ADC --> Strom in A
42 ampere       = a(:,13);
43
44 % System Status Text
45 % 1 = Init / 2 = Run / 3 = Soft-Stop
46 status_text      = a(:,14);
47
48 % System Status Text
49 % 0 = AUS / 1 = AN
50 status_power      = a(:,15);
51
52 % Lüfter 1
53 % Temperatur
54 % Zustand
55
56 figure(1);
57
58 hold on
59 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
60 plot(sekunden,temp_1/100,'Color','blue', 'LineWidth', 2);
61 hold on
62 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
63 xlim([0 max(sekunden)]);
64 ylim([18 70]);
65 title('Lüfter 1 - Temperatur');
66 xlabel('Zeit [s]');
```

```
67  ylabel('Celsius');
68  legend('Lüfter 1','Raum');
69  grid('on');
70  hold off
71
72  hold on
73  subplot(3,1,2); % Zeilen, Spalten, Plotnummer
74  [ax,p1,p2] = plotyy(sekunden,tastgrad_1,sekunden,rpm_1);
75  set(p1(1), 'LineWidth', 2);
76  set(p2(1), 'LineWidth', 2);
77  xlim(ax(1),[0 max(sekunden)]);
78  xlim(ax(2),[0 max(sekunden)]);
79  ylim(ax(1),[0 110]);
80  ylim(ax(2),[0 2300]);
81  title('Lüfter 1 - Tastgrad / RPM');
82  xlabel('Zeit [s]');
83  ylabel(ax(1),'Tastgrad') % label left y-axis
84  ylabel(ax(2),'RPM') % label right y-axis
85  grid(ax(1),'on') % grid x-axis ON
86  hold off
87
88  hold on
89  subplot(3,1,3); % Zeilen, Spalten, Plotnummer
90  plot(sekunden,status_text,'Color','red', 'LineWidth', 2);
91  xlim([0 max(sekunden)]);
92  ylim([0 4]);
93  title('Zustand');
94  ylabel('Zustand');
95  xlabel('Zeit [s]');
96  hold off
97
98  % Lüfter 2
99  % Temperatur
100 % Zustand
101 figure(2);
102
103 hold on
104 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
105 plot(sekunden,temp_2/100,'Color','blue', 'LineWidth', 2);
106 hold on
107 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
108 xlim([0 max(sekunden)]);
109 ylim([18 70]);
110 title('Lüfter 2 - Temperatur');
111 xlabel('Zeit [s]');
112 ylabel('Celsius');
113 legend('Lüfter 2','Raum');
114 grid('on');
115 hold off
116
117 hold on
118 subplot(3,1,2); % Zeilen, Spalten, Plotnummer
119 [ax,p1,p2] = plotyy(sekunden,tastgrad_2,sekunden,rpm_2);
120 set(p1(1), 'LineWidth', 2);
121 set(p2(1), 'LineWidth', 2);
122 xlim(ax(1),[0 max(sekunden)]);
123 xlim(ax(2),[0 max(sekunden)]);
124 ylim(ax(1),[0 110]);
125 ylim(ax(2),[0 2400]);
126 title('Lüfter 2 - Tastgrad / RPM');
```

```
127 xlabel('Zeit [s]');
128 ylabel(ax(1),'Tastgrad') % label left y-axis
129 ylabel(ax(2),'RPM') % label right y-axis
130 grid(ax(1),'on') % grid x-axis ON
131 hold off
132
133 hold on
134 subplot(3,1,3); % Zeilen, Spalten, Plotnummer
135 plot(sekunden,status_text,'Color','red', 'LineWidth', 2);
136 xlim([0 max(sekunden)]);
137 ylim([0 4]);
138 title('Zustand');
139 xlabel('Zeit [s]');
140 ylabel('Zustand');
141 hold off
142
143 % Lüfter 3
144 % Temperatur
145 % Zustand
146 figure(3);
147
148 hold on
149 subplot(3,1,1); % Zeilen, Spalten, Plotnummer
150 plot(sekunden,temp_3/100,'Color','blue', 'LineWidth', 2);
151 hold on
152 plot(sekunden,temp_5/100,'Color','black', 'LineWidth', 2);
153 xlim([0 max(sekunden)]);
154 ylim([18 70]);
155 title('Lüfter 3 - Temperatur');
156 xlabel('Zeit [s]');
157 ylabel('Celsius');
158 legend('Lüfter 3','Raum');
159 grid('on');
160 hold off
161
162 hold on
163 subplot(3,1,2); % Zeilen, Spalten, Plotnummer
164 [ax,p1,p2] = plotyy(sekunden,tastgrad_3,sekunden,rpm_3);
165 set(p1(1), 'LineWidth', 2);
166 set(p2(1), 'LineWidth', 2);
167 xlim(ax(1),[0 max(sekunden)]);
168 xlim(ax(2),[0 max(sekunden)]);
169 ylim(ax(1),[0 110]);
170 ylim(ax(2),[0 2400]);
171 title('Lüfter 3 - Tastgrad / RPM');
172 xlabel('Zeit [s]');
173 ylabel(ax(1),'Tastgrad') % label left y-axis
174 ylabel(ax(2),'RPM') % label right y-axis
175 grid(ax(1),'on') % grid x-axis ON
176 hold off
177
178 hold on
179 subplot(3,1,3); % Zeilen, Spalten, Plotnummer
180 plot(sekunden,status_text,'Color','red', 'LineWidth', 2);
181 xlim([0 max(sekunden)]);
182 ylim([0 4]);
183 title('Zustand');
184 xlabel('Zeit [s]');
185 ylabel('Zustand');
186 hold off
```

Listing A.6: MATLAB-File für die Messung der Stromüberwachung

```
% Die CSV-Datei einlesen
a = csvread('Messung_CE.csv');

% Zeit in Sekunden
% Die Werte der ersten Spalte
sekunden = a(:,1);

% Temperatur 1
temp_1    = a(:,2);

% Temperatur 2
temp_2    = a(:,3);

% Temperatur 3
temp_3    = a(:,4);

% Temperatur 4
temp_4    = a(:,5);

% Temperatur 5
temp_5    = a(:,6);

% Tastgrad L1 in %
tastgrad_1 = a(:,7);

% Tastgrad L1 in %
tastgrad_2 = a(:,8);

% Tastgrad L1 in %
tastgrad_3 = a(:,9);

% RPM des Lüfters 1
rpm_1       = a(:,10);

% RPM des Lüfters 2
rpm_2       = a(:,11);

% RPM des Lüfters 3
rpm_3       = a(:,12);

% ADC --> Strom in A
ampere       = a(:,13);

% System Status Text
% 1 = Init / 2 = Run / 3 = Soft-Stop
status_text      = a(:,14);

% System Status Text
% 0 = AUS / 1 = AN
status_power      = a(:,15);

% Hallsensor Strom
% Zustand
figure(1);

hold on
subplot(2,1,1); % Zeilen, Spalten, Plotnummer
plot(sekunden,ampere+6,'Color','blue', 'LineWidth', 2);
```

```
59  xlim([0 max(sekunden)]);
60  title('Hallsensor - Strom');
61  xlabel('Zeit [s]');
62  ylabel('Strom [A]');
63  grid('on');
64  hold off
65
66  hold on
67  subplot(2,1,2); % Zeilen, Spalten, Plotnummer
68  plot(sekunden,status_text,'Color','red', 'LineWidth', 2);
69  xlim([0 max(sekunden)]);
70  ylim([0 4]);
71  title('Zustand');
72  ylabel('Zustand');
73  xlabel('Zeit [s]');
74  hold off
```

A.3. MATLAB Figures

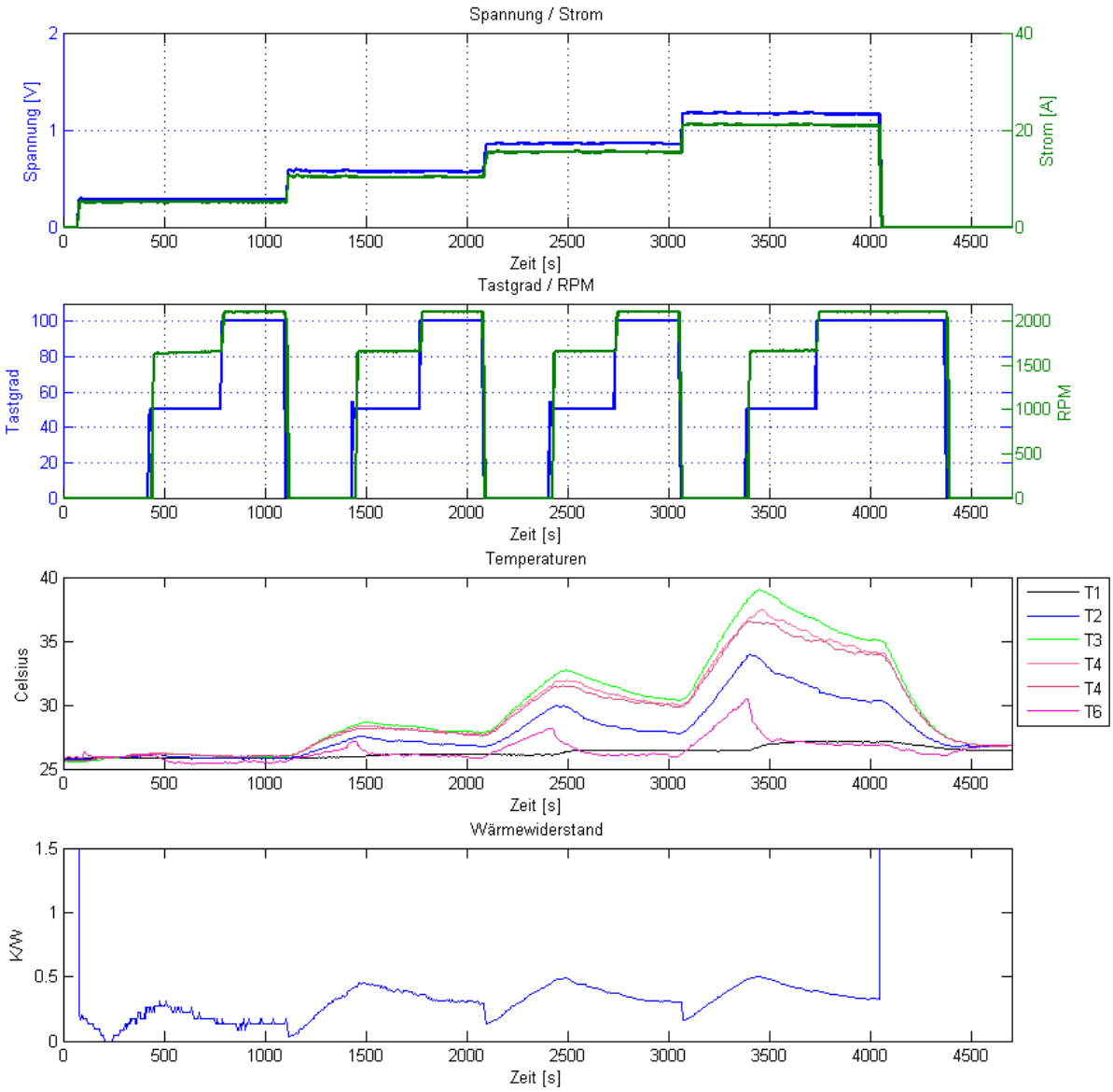


Abbildung A.1.: Ergebnisse: Messung 1 - Ladewiderstände (2 Parallel)

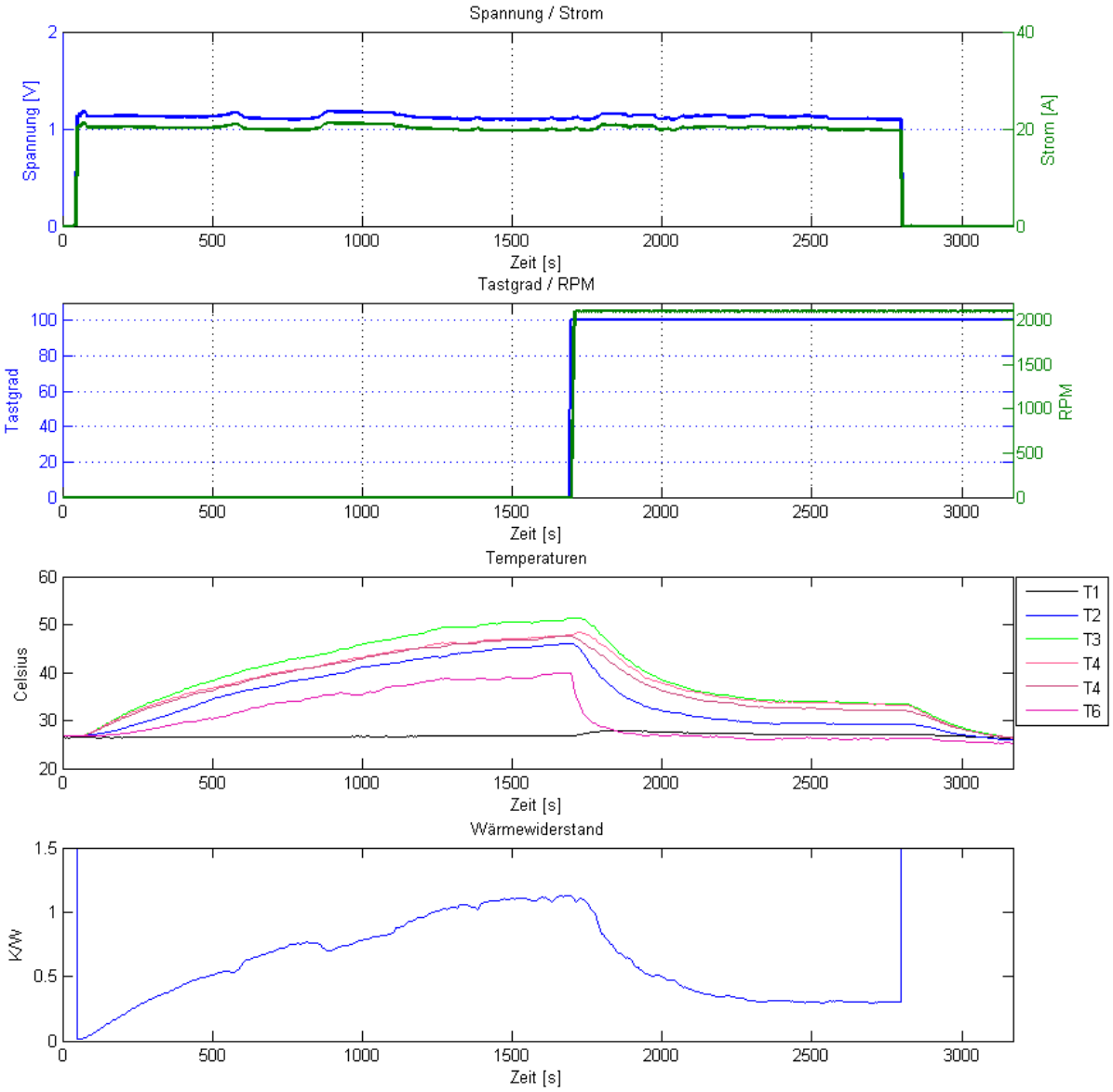


Abbildung A.2.: Ergebnisse: Messung 2 - Ladewiderstände (2 Parallel)

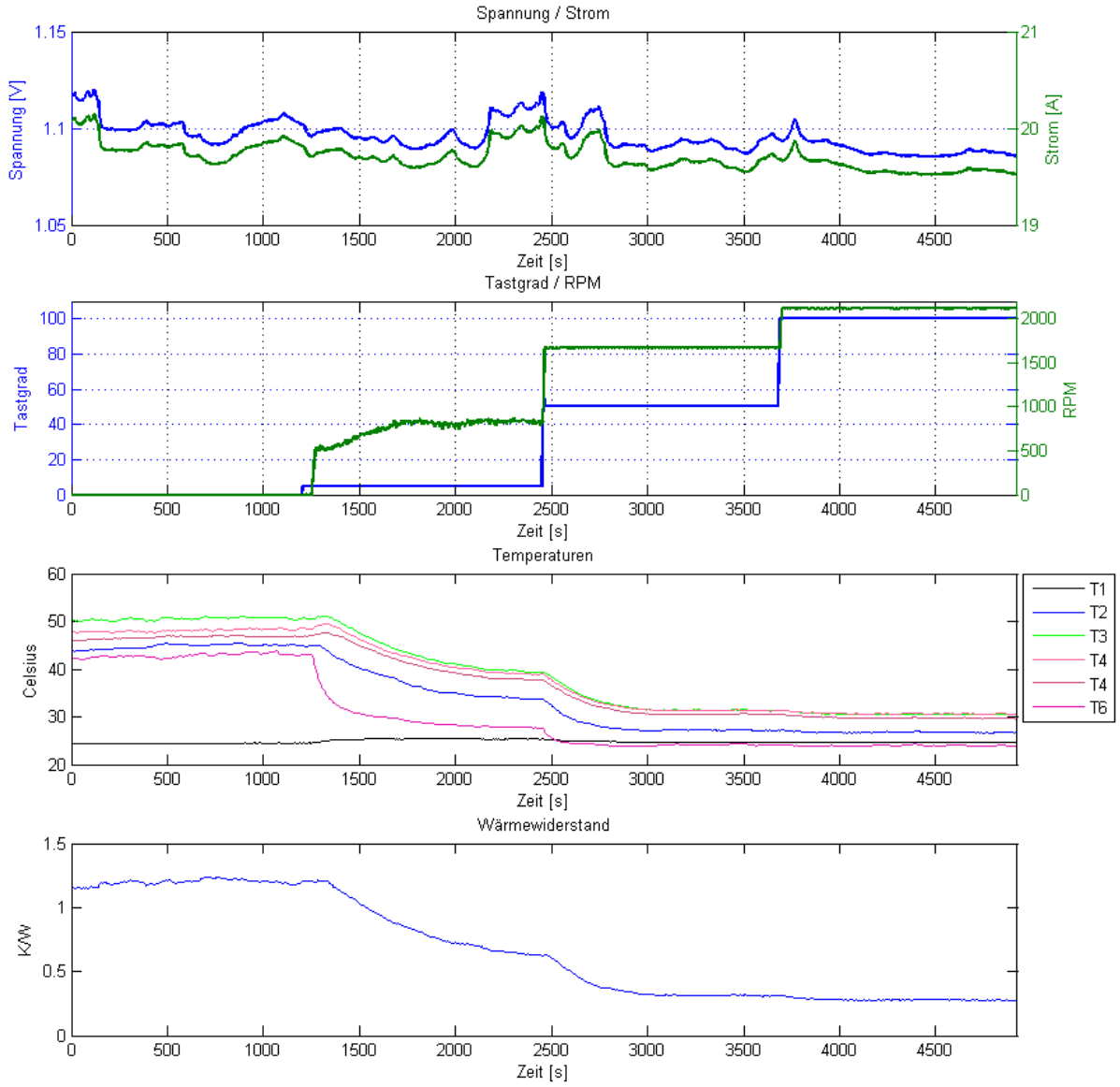


Abbildung A.3.: Ergebnisse: Messung 3 - Ladewiderstände (2 Parallel)

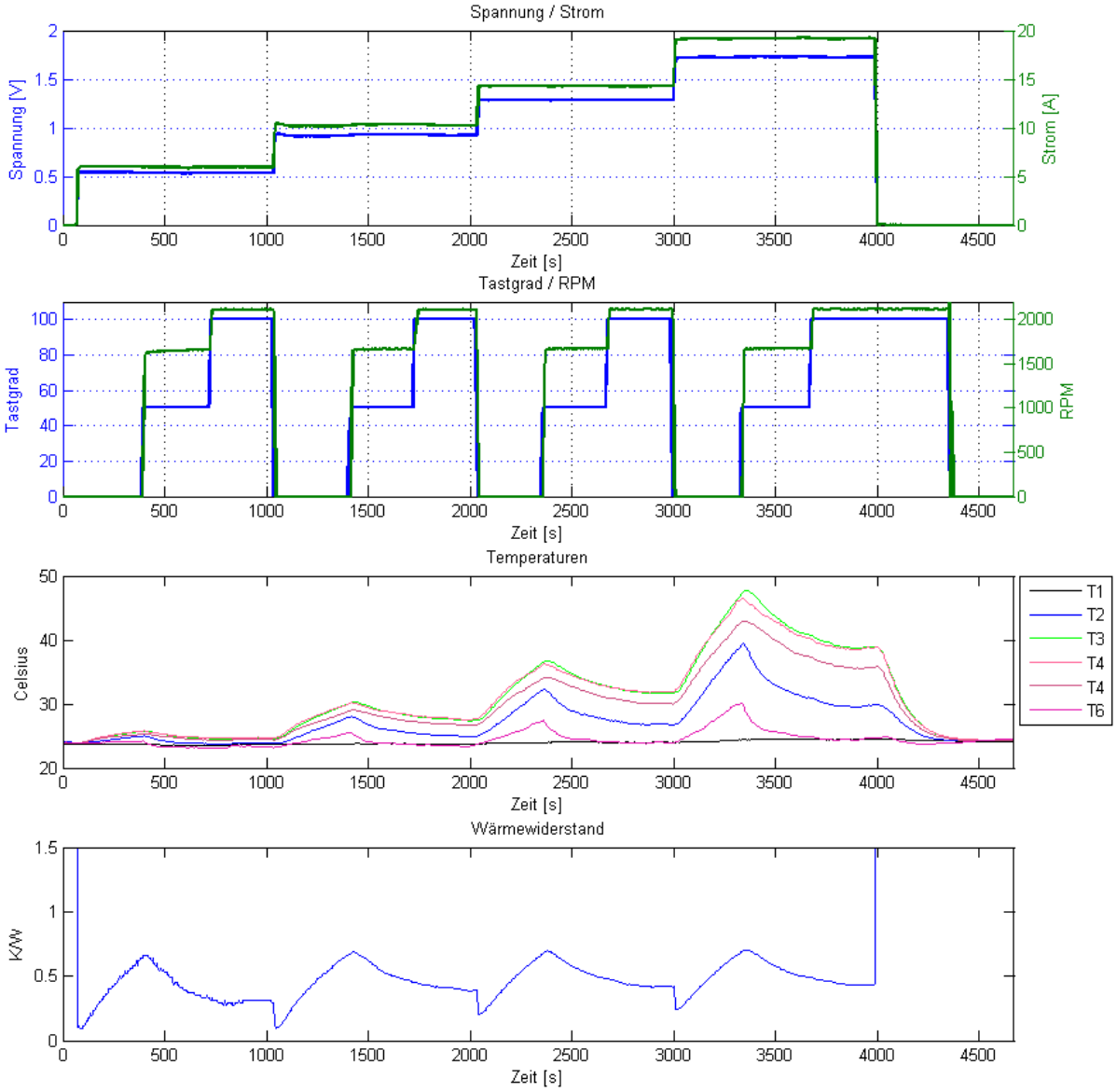


Abbildung A.4.: Ergebnisse: Messung 1 - Lastwiderstände (4 Reihe)

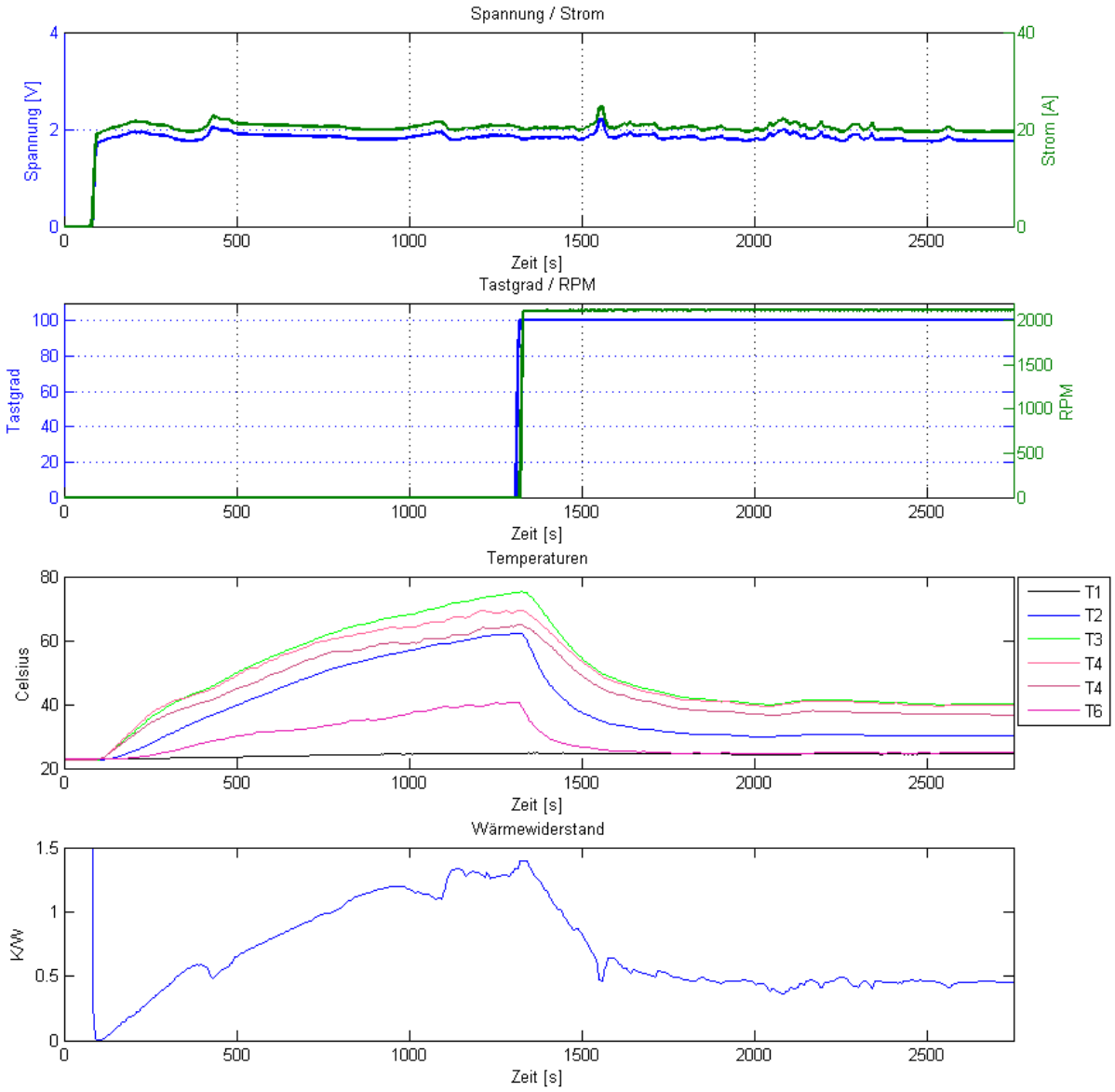


Abbildung A.5.: Ergebnisse: Messung 2 - Lastwiderstände (4 Reihe)

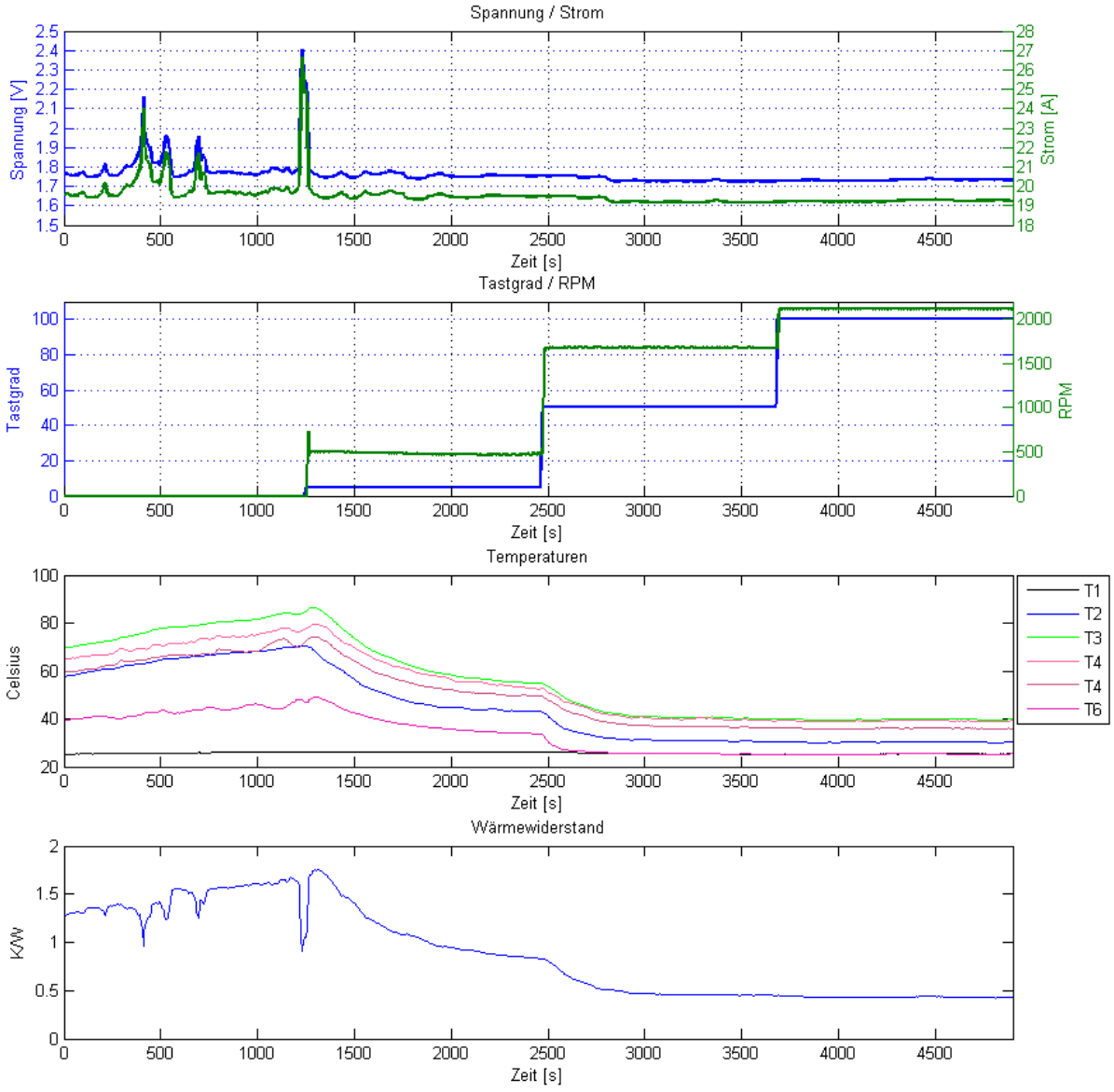


Abbildung A.6.: Ergebnisse: Messung 3 - Lastwiderstände (4 Reihe)

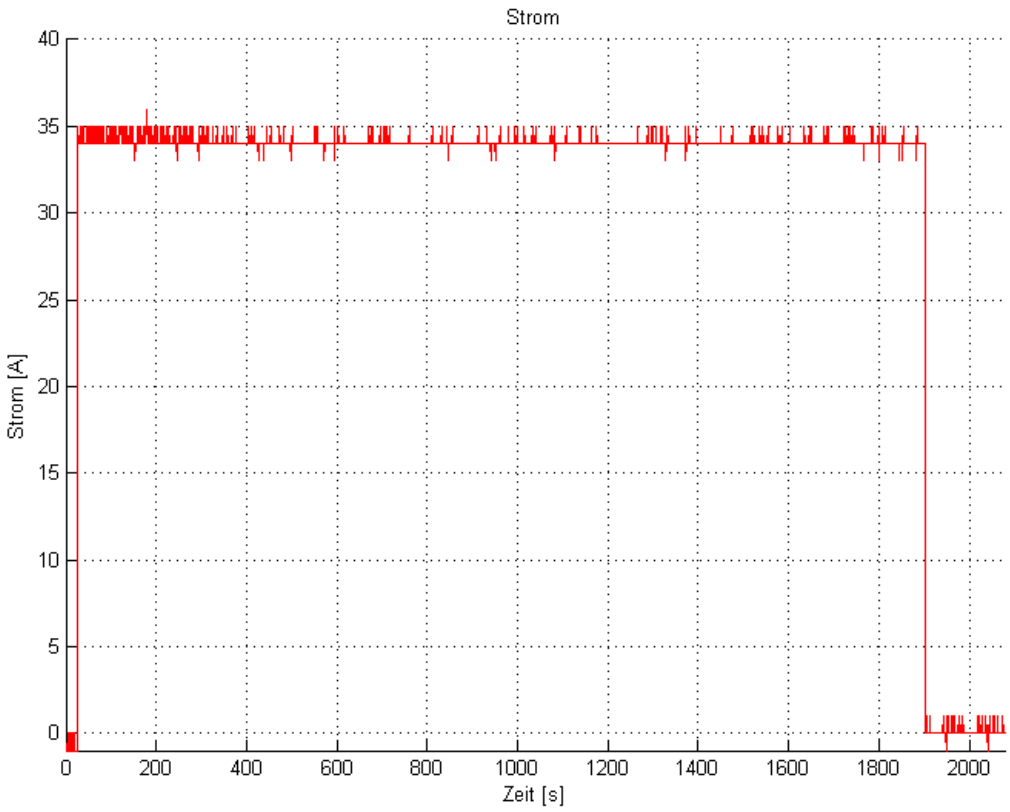


Abbildung A.7.: Ergebnisse: Messung Volllast - Testaufbau - Strom

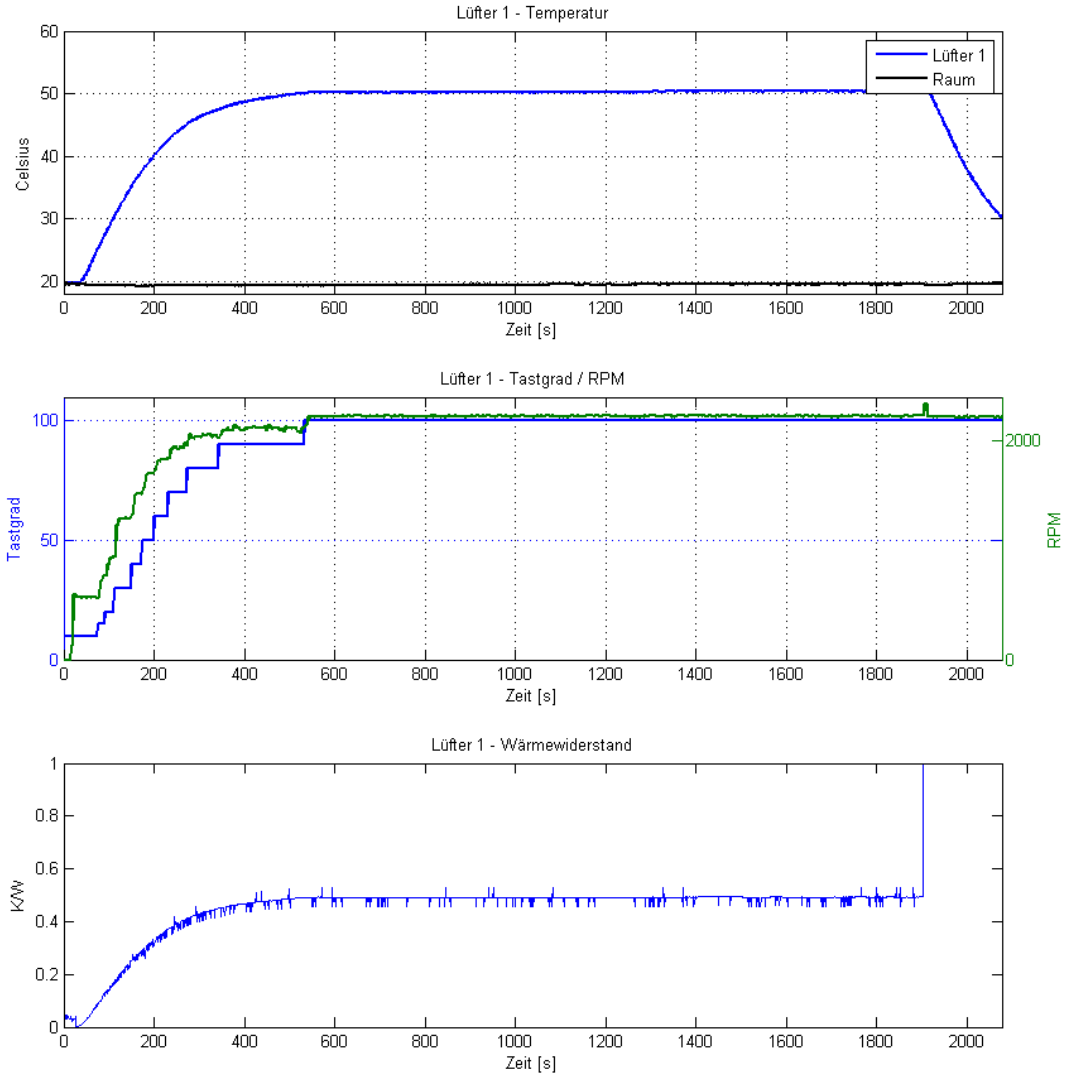


Abbildung A.8.: Ergebnisse: Messung Volllast - Testaufbau - Lüfter 1

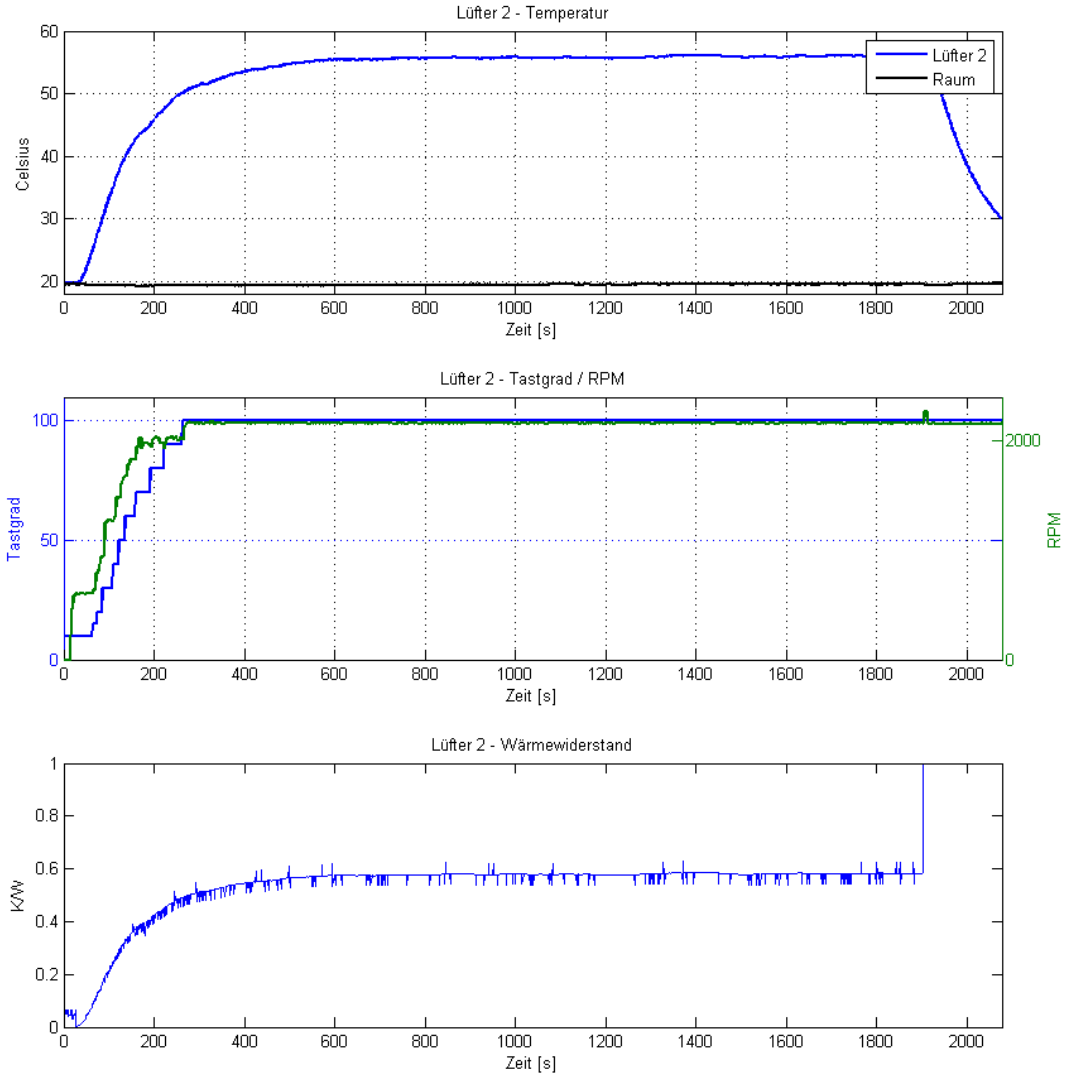


Abbildung A.9.: Ergebnisse: Messung Volllast - Testaufbau - Lüfter 2

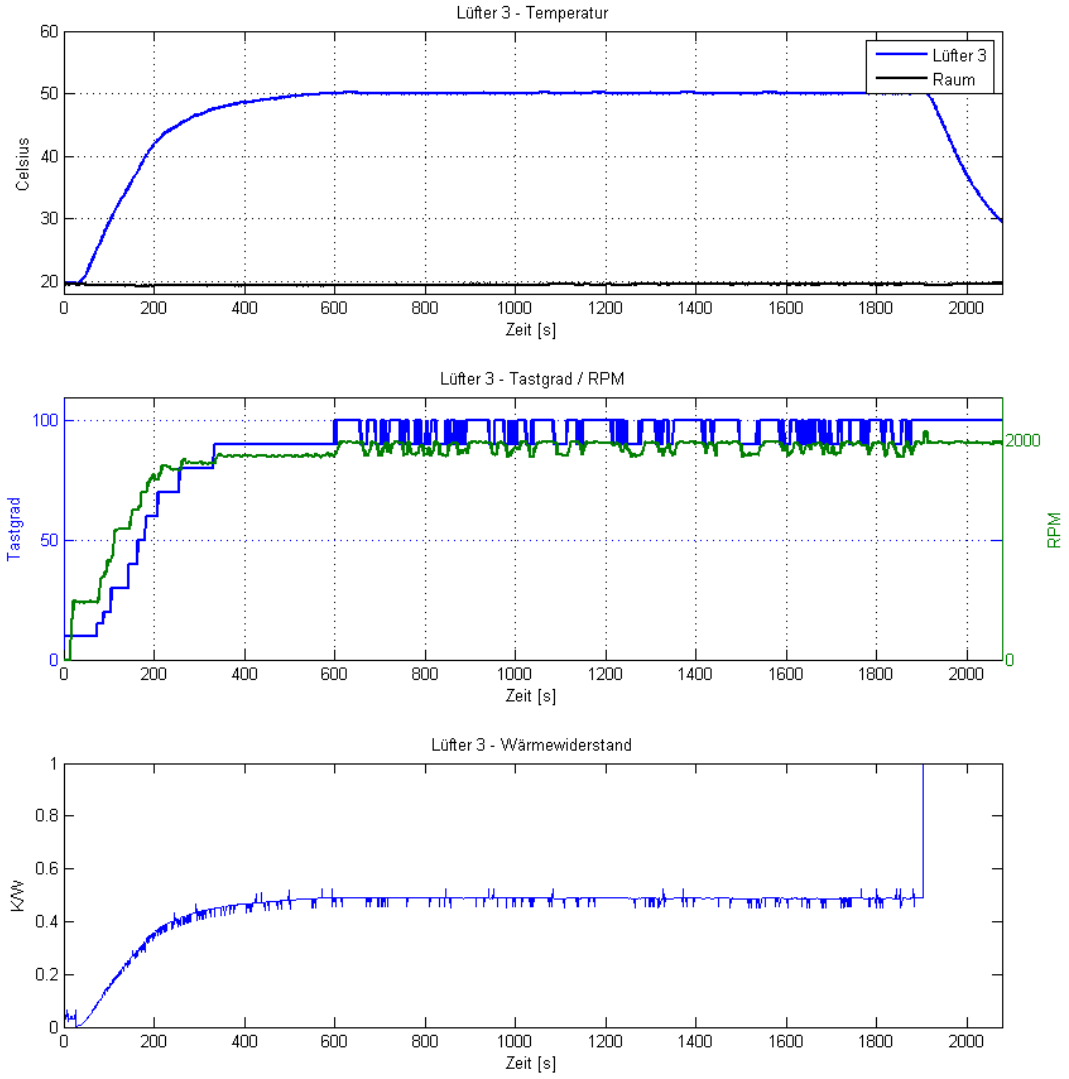


Abbildung A.10.: Ergebnisse: Messung Volllast - Testaufbau - Lüfter 3

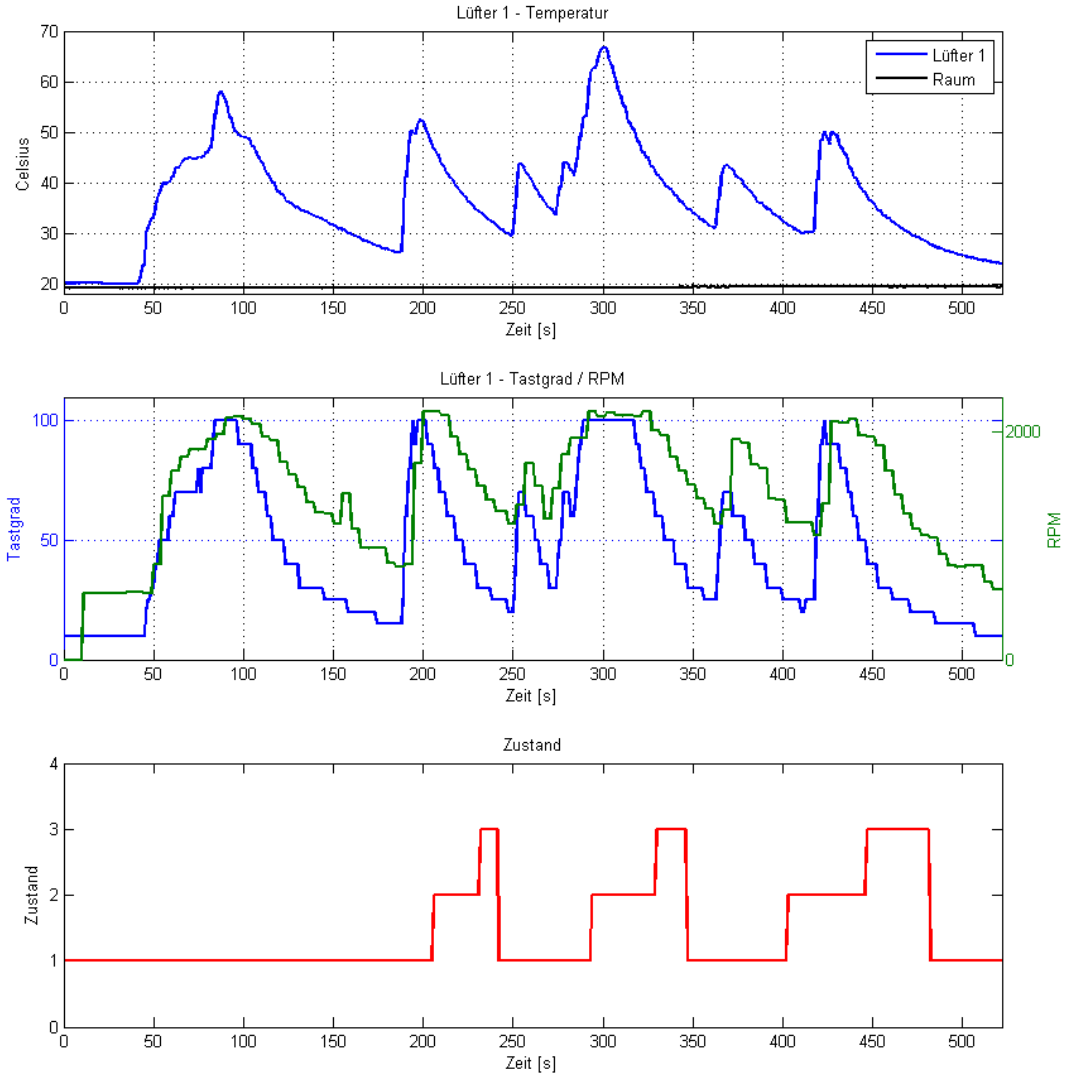


Abbildung A.11.: Ergebnisse: Messung Lüftersteuerung - Lüfter 1

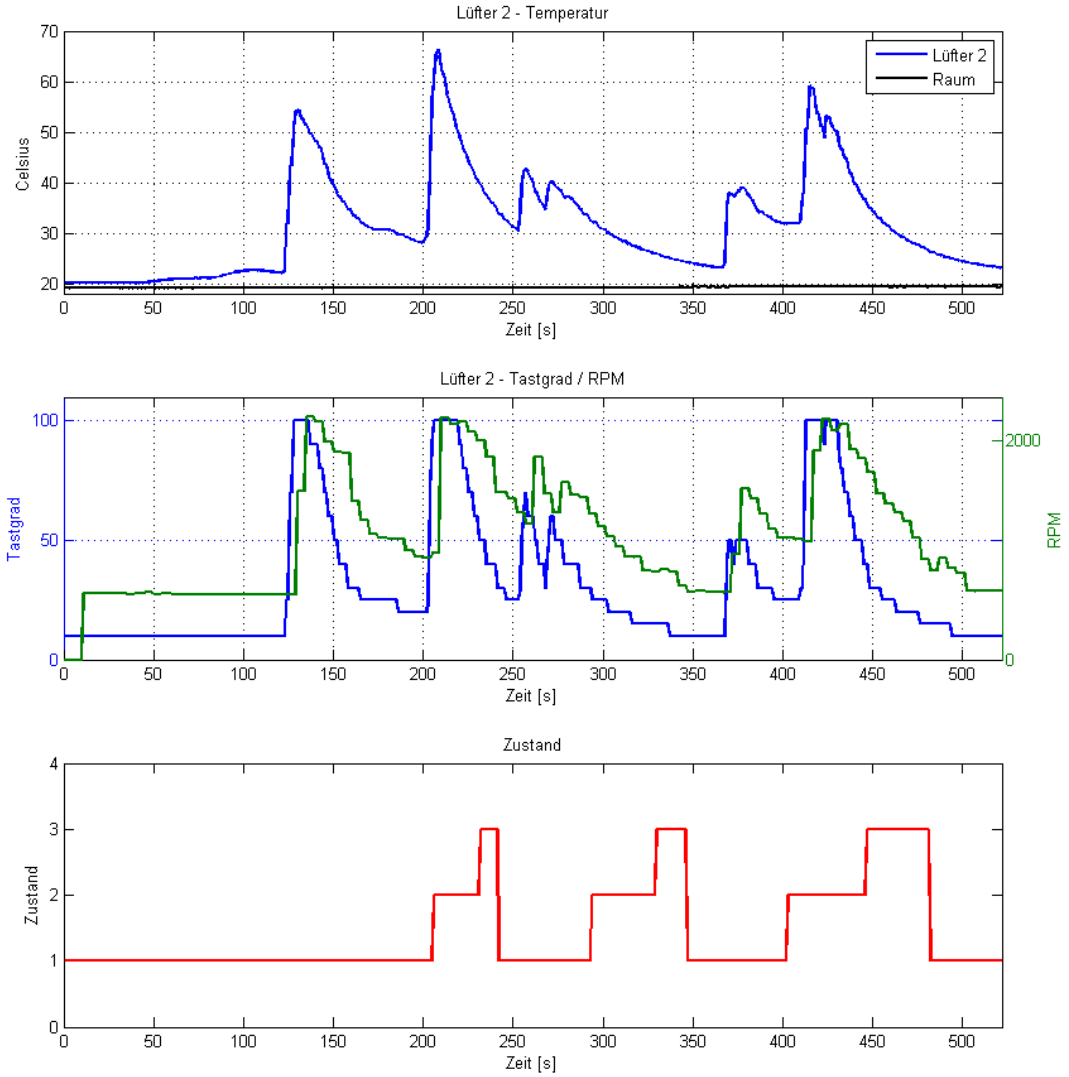


Abbildung A.12.: Ergebnisse: Messung Lüftersteuerung - Lüfter 2

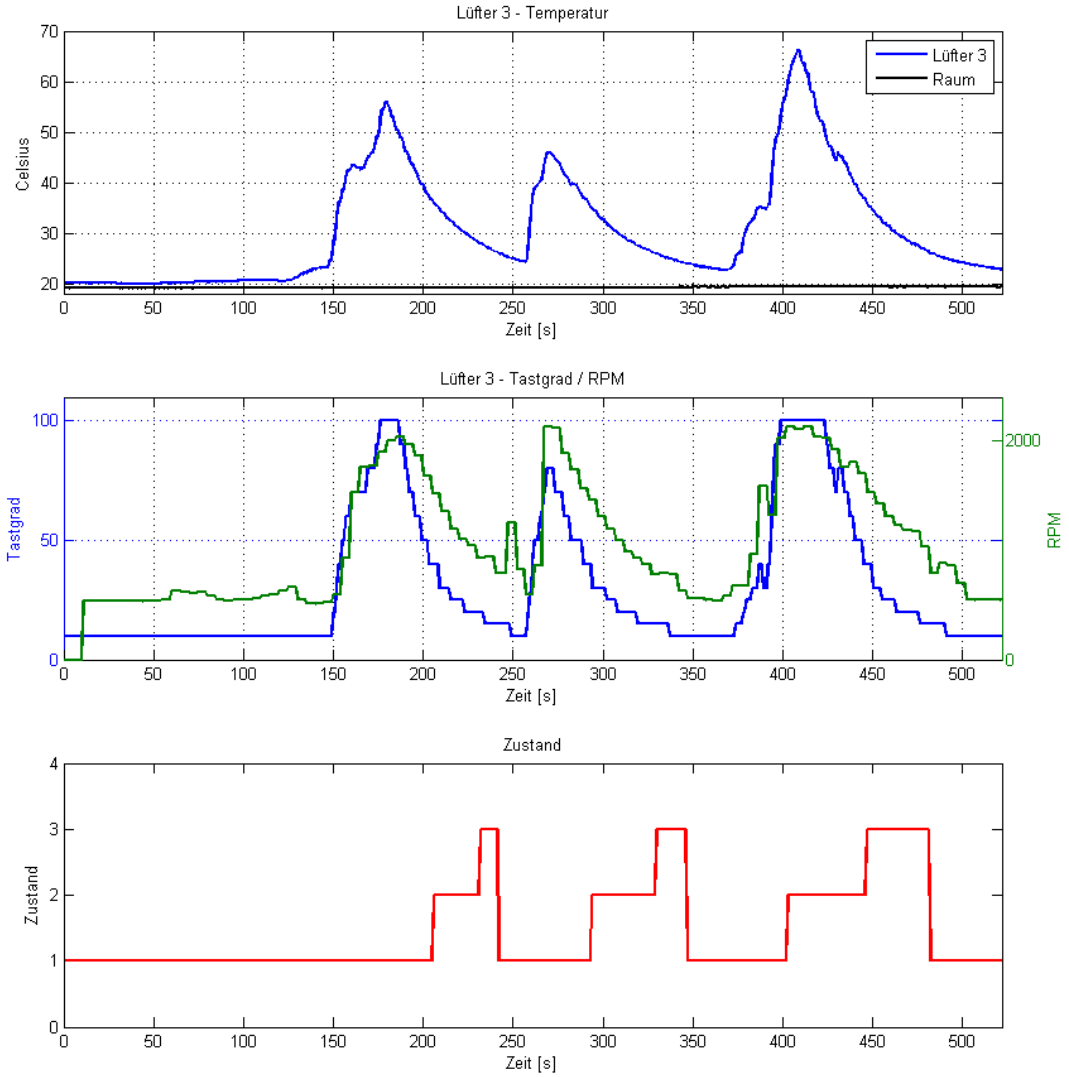


Abbildung A.13.: Ergebnisse: Messung Lüftersteuerung - Lüfter 3

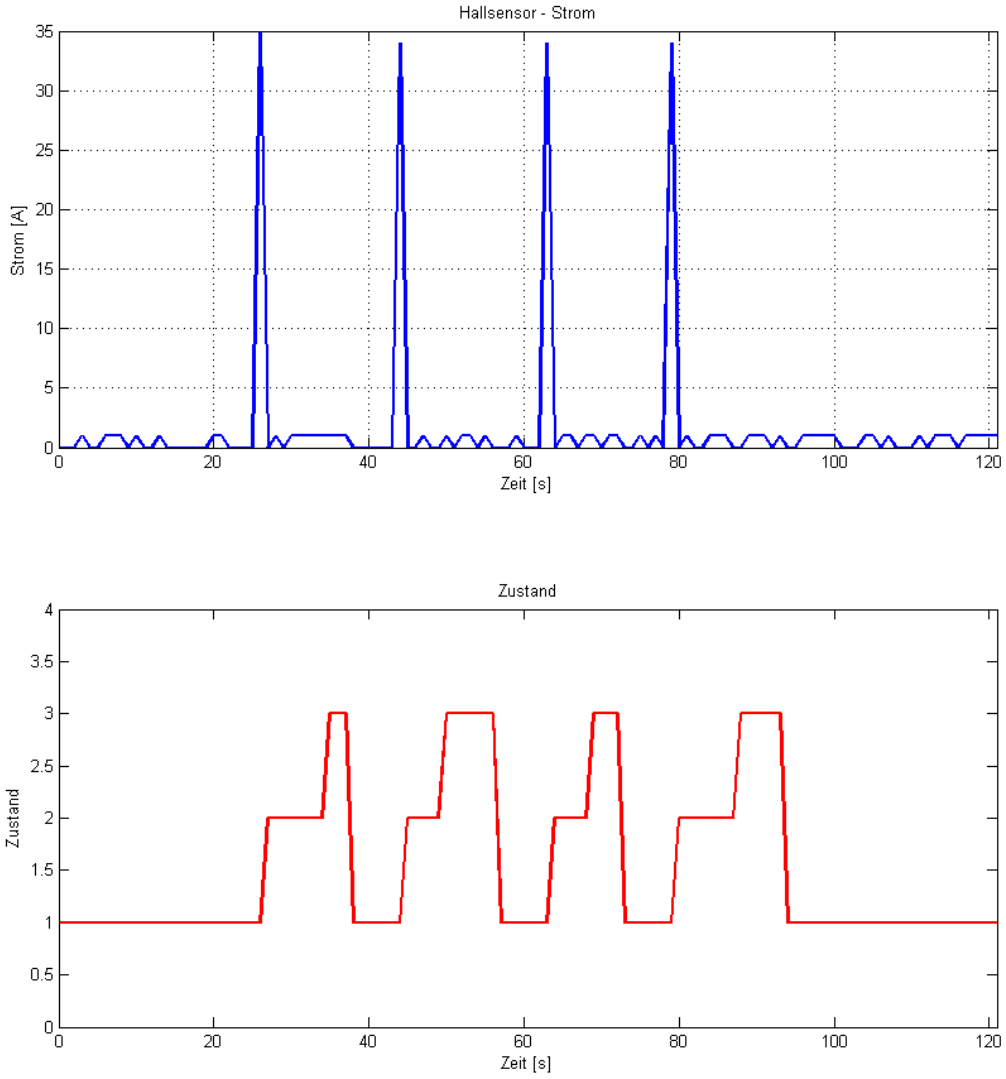


Abbildung A.14.: Ergebnisse: Messung Stromüberwachung

A.4. Tabelle PIN-Belegung

PORT	PIN	Verwendung	Eigenschaften	Funktion
A	0	UART0	RX	Kommunikation mit PC
A	1	UART0	TX	Kommunikation mit PC
C	4	UART7	RX	1-wire Temperatursensoren
C	5	UART7	TX	1-wire Temperatursensoren
F	0	PWM0	M0PWM0	Lüfter 1 Ansteuerung
F	1	PWM0	M0PWM1	Lüfter 2 Ansteuerung
F	2	PWM0	M0PWM2	Lüfter 3 Ansteuerung
F	3	PWM0	M0PWM3	Lüfter 4 Ansteuerung
G	0	PWM0	M0PWM4	Lüfter 5 Ansteuerung
G	1	PWM0	M0PWM5	Lüfter 6 Ansteuerung
-	-	Timer5	[32 Bit Mode]	Zeitgeberschleife
A	4	Timer2	CCP0	Lüfter 1 Flanken Zähler
A	5	Timer2	CCP1	Lüfter 2 Flanken Zähler
A	6	Timer3	CCP0	Lüfter 3 Flanken Zähler
A	7	Timer3	CCP1	Lüfter 4 Flanken Zähler
B	0	Timer4	CCP0	Lüfter 5 Flanken Zähler
B	1	Timer4	CCP1	Lüfter 6 Flanken Zähler
H	0	GPIO-H	Logischer Ausgang	LED rot
H	1	GPIO-H	Logischer Ausgang	LED grün
H	2	GPIO-H	Logischer Ausgang	Relay schalten (Ladekreis)
H	3	GPIO-H	Logischer Ausgang	Relay schalten (Lastkreis)
M	4	GPIO-H	Logischer Ausgang	Akustisches Signal
M	7	GPIO-H	Logischer Ausgang	Schutzcontroller bereit
M	5	GPIO-H	Logischer Ausgang	Netzteil AN/AUS (nur für Funktionstest)
L	0	GPIO-L	Logischer Eingang	Button rot
L	1	GPIO-L	Logischer Eingang	Button gelb
L	2	GPIO-L	Logischer Eingang	Button schwarz
L	3	GPIO-L	Logischer Eingang	Zykliercontroller bereit
L	4	GPIO-L	Logischer Eingang	Netzteil aktiv (nur für Funktionstest)
E	0	ADC1	AIN03	Hallsensor (Ladekreis)
E	1	ADC1	AIN02	Hallsensor (Lastkreis)
A	X	SPI0	SPI-1	Spannungsmesschaltung (Batterie)
-	-	Timer1	Timer-A	Display Eingabe
-	-	ADC0	CH6	Display Eingabe
-	-	ADC0	CH7	Display Eingabe
D	4	GPIO-D	Logischer Ausgang	Display Eingabe
D	5	GPIO-D	Logischer Ausgang	Display Eingabe
Q	1	GPIO-Q	Logischer Ausgang	Display Eingabe
M	6	GPIO-M	Logischer Ausgang	Display Eingabe
G	1	GPIO-G	Logischer Ausgang	Display Ausgabe (Backlight)

Tabelle A.1.: PIN-Belegung Schutzsystem

A.5. Fehlercodes

Fehlerblock 1X	**Temperaturfehler**
Fehlercode	Bedeutung
E#10	Grenzwert an Temperatursensor#1 überschritten
E#11	Grenzwert an Temperatursensor#2 überschritten
E#12	Grenzwert an Temperatursensor#3 überschritten
E#13	Grenzwert an Temperatursensor#4 überschritten
E#14	Grenzwert an Temperatursensor#5 überschritten
E#15	Grenzwert an Temperatursensor#6 überschritten
Fehlerblock 2X	**Stromfehler**
Fehlercode	Bedeutung
E#20	Grenzwert an Hallsensor#1 überschritten
E#21	Grenzwert an Hallsensor#2 überschritten
Fehlerblock 3X	**Drehzahlfehler**
Fehlercode	Bedeutung
E#30	Drehzahl an Lüfter 1 unter 100 U/min.
E#31	Drehzahl an Lüfter 2 unter 100 U/min.
E#32	Drehzahl an Lüfter 3 unter 100 U/min.
E#33	Drehzahl an Lüfter 4 unter 100 U/min.
E#34	Drehzahl an Lüfter 5 unter 100 U/min.
E#35	Drehzahl an Lüfter 6 unter 100 U/min.

Tabelle A.2.: Fehlercodes

A.6. Warnungscodes

Warnblock 1X	**Temperaturwarnungen**
Warncode	Bedeutung
W#10	Warnbereich an Temperatursensor#1 erreicht
W#11	Warnbereich an Temperatursensor#2 erreicht
W#12	Warnbereich an Temperatursensor#3 erreicht
W#13	Warnbereich an Temperatursensor#4 erreicht
W#14	Warnbereich an Temperatursensor#5 erreicht
W#15	Warnbereich an Temperatursensor#6 erreicht
Warnblock 2X	**Stromwarnungen**
Warncode	Bedeutung
W#20	Warnbereich an Hallsensor#1 erreicht
W#21	Warnbereich an Hallsensor#2 erreicht

Tabelle A.3.: Warncodes

A.7. Quellcode Hauptprogramm

Listing A.7: 1_wire_crc.c

```
1  //
2  //
3  //
4
5  #include <stdint.h>
6
7  //*****************************************************************************
8  // The CRC table for the polynomial C(x) = x^8 + x^5 + x^4 + 1 (CRC-8_1-wire).
9  //*****************************************************************************
10 static const uint8_t g_pui8CRC8_1_wire[256] =
11 {
12         0, 94, 188, 226, 97, 63, 221, 131, 194, 156, 126, 32, 163, 253, 31, 65,
13         157, 195, 33, 127, 252, 162, 64, 30, 95, 1, 227, 189, 62, 96, 130, 220,
14         35, 125, 159, 193, 66, 28, 254, 160, 225, 191, 93, 3, 128, 222, 60, 98,
15         190, 224, 2, 92, 223, 129, 99, 61, 124, 34, 192, 158, 29, 67, 161, 255,
16         70, 24, 250, 164, 39, 121, 155, 197, 132, 218, 56, 102, 229, 187, 89, 7,
17         219, 133, 103, 57, 186, 228, 6, 88, 25, 71, 165, 251, 120, 38, 196, 154,
18         101, 59, 217, 135, 4, 90, 184, 230, 167, 249, 27, 69, 198, 152, 122, 36,
19         248, 166, 68, 26, 153, 199, 37, 123, 58, 100, 134, 216, 91, 5, 231, 185,
20         140, 210, 48, 110, 237, 179, 81, 15, 78, 16, 242, 172, 47, 113, 147, 205,
21         17, 79, 173, 243, 112, 46, 204, 146, 211, 141, 111, 49, 178, 236, 14, 80,
22         175, 241, 19, 77, 206, 144, 114, 44, 109, 51, 209, 143, 12, 82, 176, 238,
23         50, 108, 142, 208, 83, 13, 239, 177, 240, 174, 76, 18, 145, 207, 45, 115,
24         202, 148, 118, 40, 171, 245, 23, 73, 8, 86, 180, 234, 105, 55, 213, 139,
25         87, 9, 235, 181, 54, 104, 138, 212, 149, 203, 41, 119, 244, 170, 72, 22,
26         233, 183, 85, 11, 136, 214, 52, 106, 43, 117, 151, 201, 74, 20, 246, 168,
27         116, 42, 200, 150, 21, 75, 169, 247, 182, 232, 10, 84, 215, 137, 107, 53
```

```
28  };
29
30  //*****************************************************************************
31  //
32  // This macro executes one iteration of the CRC8_1_wire.
33  //
34  //*****************************************************************************
35  #define CRC8_ITER(crc, data)    g_pui8CRC8_1_wire[(uint8_t)((crc) ^ (data))]
36  //*****************************************************************************
37
38
39  //
40  //! Calculates the CRC8_1_wire of an array of bytes.
41  //!
42  //! \param ui8Crc is the starting CRC8_1_wire value.
43  //! \param pui8Data is a pointer to the data buffer.
44  //! \param ui32Count is the number of bytes in the data buffer.
45  //!
46  //! This function is used to calculate the CRC8_1_wire of the input buffer.
47  //! The CRC8_1_wire is computed in a running fashion, meaning that the entire
48  //! data block that is to have its CRC8_1_wire computed does not need to be
49  //! supplied all at once.  If the input buffer contains the entire block of
50  //! data, then \b ui8Crc should be set to 0.  If, however, the entire block of
51  //! data is not available, then \b ui8Crc should be set to 0 for the first
52  //! portion of the data, and then the returned value should be passed back in
53  //! as \b ui8Crc for the next portion of the data.
54  //!
55  //! For example, to compute the CRC8_1_wire of a block that has been split into
56  //! three pieces, use the following:
57  //!
58  //! \verbatim
59  //!     ui8Crc = CRC8_1_wire(0, pui8Data1, ui32Len1);
60  //!     ui8Crc = CRC8_1_wire(ui8Crc, pui8Data2, ui32Len2);
61  //!     ui8Crc = CRC8_1_wire(ui8Crc, pui8Data3, ui32Len3);
62  //! \endverbatim
63  //!
64  //! Computing a CRC8_1_wire in a running fashion is useful in cases where the
65  //! data is arriving via a serial link (for example) and is therefore not all
66  //! available at one time.
67  //!
68  //! \return The CRC8_1_wire of the input data.
69  //
70  //*****************************************************************************
71  uint8_t
72  CRC8_1_wire(uint8_t ui8Crc, const uint8_t *pui8Data, uint32_t ui32Count)
73  {
74      uint32_t ui32Temp;
75
76      //
77      // If the data buffer is not 16 bit-aligned, then perform a single step of
78      // the CRC to make it 16 bit-aligned.
79      //
80      if((uint32_t)pui8Data & 1)
81      {
82          //
83          // Perform the CRC on this input byte.
84          //
85          ui8Crc = CRC8_ITER(ui8Crc, *pui8Data);
86
87          //
```

```
88          // Skip this input byte.
89          //
90          pui8Data++;
91          ui32Count--;
92      }
93
94      //
95      // If the data buffer is not word-aligned and there are at least two bytes
96      // of data left, then perform two steps of the CRC to make it word-aligned.
97      //
98      if(((uint32_t)pui8Data & 2) && (ui32Count > 1))
99      {
100         //
101         // Read the next 16 bits.
102         //
103         ui32Temp = *(uint16_t *)pui8Data;
104
105         //
106         // Perform the CRC on these two bytes.
107         //
108         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp);
109         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp >> 8);
110
111         //
112         // Skip these input bytes.
113         //
114         pui8Data += 2;
115         ui32Count -= 2;
116     }
117
118     //
119     // While there is at least a word remaining in the data buffer, perform
120     // four steps of the CRC to consume a word.
121     //
122     while(ui32Count > 3)
123     {
124         //
125         // Read the next word.
126         //
127         ui32Temp = *(uint32_t *)pui8Data;
128
129         //
130         // Perform the CRC on these four bytes.
131         //
132         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp);
133         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp >> 8);
134         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp >> 16);
135         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp >> 24);
136
137         //
138         // Skip these input bytes.
139         //
140         pui8Data += 4;
141         ui32Count -= 4;
142     }
143
144     //
145     // If there are 16 bits left in the input buffer, then perform two steps of
146     // the CRC.
147     //
```

```
148     if(ui32Count > 1)
149     {
150         //
151         // Read the 16 bits.
152         //
153         ui32Temp = *(uint16_t *)pui8Data;
154 
155         //
156         // Perform the CRC on these two bytes.
157         //
158         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp);
159         ui8Crc = CRC8_ITER(ui8Crc, ui32Temp >> 8);
160 
161         //
162         // Skip these input bytes.
163         //
164         pui8Data += 2;
165         ui32Count -= 2;
166     }
167 
168     //
169     // If there is a final byte remaining in the input buffer, then perform a
170     // single step of the CRC.
171     //
172     if(ui32Count != 0)
173     {
174         ui8Crc = CRC8_ITER(ui8Crc, *pui8Data);
175     }
176 
177     //
178     // Return the resulting CRC8_1_wire value.
179     //
180     return(ui8Crc);
181 }
```

Listing A.8: 1_wire_crc.h

```
1  #ifndef __DRIVERLIB_SW_CRC_H__
2  #define __DRIVERLIB_SW_CRC_H__
3  //*****************************************************************************
4  //
5  // Prototypes for the functions.
6  //
7  //*****************************************************************************
8  extern uint8_t CRC8_1_wire(uint8_t ui8Crc, const uint8_t *pui8Data, uint32_t ui32Count);
9 
10 #endif // __DRIVERLIB_SW_CRC_H__
```

Listing A.9: 1_wire_Devices.c

```
1  //
2  //   ROM-CODES der verwendeten 1-wire Devices
3  //
4 
5  #include <stdint.h>
6  #include <stdbool.h>
7 
8  // eigene Includes
9  #include "globals.h"
```

```
10
11  void one_wire_Devices(void)
12  {
13      //ROM_CODE_FAN_1[8] = { 0x28, 0xEB, 0x56, 0x37, 0x06, 0x00, 0x00, 0xD9 };
14      ROM_CODE_FAN[0][0] = 0x28;
15      ROM_CODE_FAN[0][1] = 0xEB;
16      ROM_CODE_FAN[0][2] = 0x56;
17      ROM_CODE_FAN[0][3] = 0x37;
18      ROM_CODE_FAN[0][4] = 0x06;
19      ROM_CODE_FAN[0][5] = 0x00;
20      ROM_CODE_FAN[0][6] = 0x00;
21      ROM_CODE_FAN[0][7] = 0xD9;
22
23      //ROM_CODE_FAN_2[8] = { 0x28, 0x8A, 0x5D, 0x37, 0x06, 0x00, 0x00, 0x5D };
24      ROM_CODE_FAN[1][0] = 0x28;
25      ROM_CODE_FAN[1][1] = 0x8A;
26      ROM_CODE_FAN[1][2] = 0x5D;
27      ROM_CODE_FAN[1][3] = 0x37;
28      ROM_CODE_FAN[1][4] = 0x06;
29      ROM_CODE_FAN[1][5] = 0x00;
30      ROM_CODE_FAN[1][6] = 0x00;
31      ROM_CODE_FAN[1][7] = 0x5D;
32
33      //ROM_CODE_FAN_3[8] = { 0x28, 0x83, 0x22, 0x37, 0x06, 0x00, 0x00, 0xC9 };
34      ROM_CODE_FAN[2][0] = 0x28;
35      ROM_CODE_FAN[2][1] = 0x83;
36      ROM_CODE_FAN[2][2] = 0x22;
37      ROM_CODE_FAN[2][3] = 0x37;
38      ROM_CODE_FAN[2][4] = 0x06;
39      ROM_CODE_FAN[2][5] = 0x00;
40      ROM_CODE_FAN[2][6] = 0x00;
41      ROM_CODE_FAN[2][7] = 0xC9;
42
43      //ROM_CODE_FAN_4[8] = { 0x28, 0x52, 0x66, 0x37, 0x06, 0x00, 0x00, 0xCC };
44      ROM_CODE_FAN[3][0] = 0x28;
45      ROM_CODE_FAN[3][1] = 0x52;
46      ROM_CODE_FAN[3][2] = 0x66;
47      ROM_CODE_FAN[3][3] = 0x37;
48      ROM_CODE_FAN[3][4] = 0x06;
49      ROM_CODE_FAN[3][5] = 0x00;
50      ROM_CODE_FAN[3][6] = 0x00;
51      ROM_CODE_FAN[3][7] = 0xCC;
52
53      //ROM_CODE_FAN_5[8] = { 0x28, 0x4D, 0x76, 0x37, 0x06, 0x00, 0x00, 0xCF };
54      ROM_CODE_FAN[4][0] = 0x28;
55      ROM_CODE_FAN[4][1] = 0x4D;
56      ROM_CODE_FAN[4][2] = 0x76;
57      ROM_CODE_FAN[4][3] = 0x37;
58      ROM_CODE_FAN[4][4] = 0x06;
59      ROM_CODE_FAN[4][5] = 0x00;
60      ROM_CODE_FAN[4][6] = 0x00;
61      ROM_CODE_FAN[4][7] = 0xCF;
62
63      //ROM_CODE_FAN_6[8] = { 0x28, 0x53, 0x26, 0x37, 0x06, 0x00, 0x00, 0x12 };
64      ROM_CODE_FAN[5][0] = 0x28;
65      ROM_CODE_FAN[5][1] = 0x53;
66      ROM_CODE_FAN[5][2] = 0x26;
67      ROM_CODE_FAN[5][3] = 0x37;
68      ROM_CODE_FAN[5][4] = 0x06;
69      ROM_CODE_FAN[5][5] = 0x00;
```

```
70      ROM_CODE_FAN[5][6] = 0x00;
71      ROM_CODE_FAN[5][7] = 0x12;
72  }
```

Listing A.10: 1_wire_Devices.h

```
1   #ifndef __1_WIRE_DEVICES_H_
2   #define __1_WIRE_DEVICES_H_
3   //*****************************************************************************
4   //
5   // Prototypes for the functions.
6   //
7   //*****************************************************************************
8   extern void one_wire_Devices(void);
9
10  #endif /* 1_WIRE_DEVICES_H_ */
```

Listing A.11: 1_wire_Functions.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12  #include "driverlib/uart.h"
13  #include "utils/uartstdio.h"
14
15  // eigene Includes
16  #include "globals.h"
17
18  void one_wire_reset(void)
19  {
20      //-1-WIRE Reset ------------------------------------------------------
21      // (9600 BAUD)
22      // sende 0xF0 = 1111 0000 über TX, wenn 0xF0 = 1111 0000 über RX zurückkommt,
23      // dann ist kein Device vorhanden
24
25          UARTConfigSetExpClk(UART7_BASE, ui32SysClkFreq, 9600,(UART_CONFIG_WLEN_8 |
                UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
26          // Parameter für UART0 Schnittstelle einstellen: Baudrate=9600, 8-1-N-N
27
28          UARTCharPut(UART7_BASE, 0xF0);
29
30          DEVICE_PULSE = UARTCharGet(UART7_BASE);
31
32          if(DEVICE_PULSE == 0xF0)
33          {
34              DEVICE_AVAILABLE = 0; // NEIN
35          }
36          else if(DEVICE_PULSE == 0xE0)
37          {
38              DEVICE_AVAILABLE = 1; // JA
39              // wenn Devices vorhanden, dann auf 1-wire Daten-Geschwindigkeits-Timing
```

```
40              // Parameter für UART3 Schnittstelle einstellen: Baudrate=115200, 8-1-N-N
41              UARTConfigSetExpClk(UART7_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
42          }
43  }
44
45
46  void one_wire_bit(int bit)
47  {
48      if(bit==0)
49      {
50          UARTCharPut(UART7_BASE, 0x00);
51      }
52      else if(bit==1)
53      {
54          UARTCharPut(UART7_BASE, 0xFF);
55      }
56  }
57
58
59  void one_wire_write_byte(int byte)
60  {
61      int i;
62      for(i=0 ; i < 8 ; i++)
63      {
64      // sendet immer das aktuell letzte Bit über den Bus
65      one_wire_bit(byte & 0x01);
66      byte = byte >> 1;
67      }
68  }
69
70  char one_wire_read_byte(void)
71  {
72      int i;
73      char byte = 0x00;
74      for(i=0 ; i < 8 ; i++)
75      {
76          buffer=0;
77          // 8 MAL !!!!
78          // auf den BUS 1111 1111 schreiben damit der 1-wire weiss das er zu antworten hat
79          UARTCharPut(UART7_BASE, 0xFF);
80
81          // Byte zurückerhalten:
82          buffer = UARTCharGet(UART7_BASE);
83
84          //danach die zurückerhaltene Byte auswerten:
85          if(buffer==0xFC) // bedeutet eine 0 vom 1-wire Slave (FC = 1111 1100)
86          {
87              // eine 0 in das MSB schreiben
88              byte = byte | 0b00000000;
89              //oder Verknüpfung BSP: 1011 | 0000 = 1011
90          }
91          else if(buffer==0xFF) // bedeutet eine 1 vom 1-wire Slave
92          {
93              // eine 1 in das MSB schreiben
94              byte = byte | 0b10000000;
95              //oder Verknüpfung BSP: 0000 | 0001 = 0001
96          }
97
98          if(i<7)
```

```
99          {
100         // nach links schieben
101         // aber nur bei i=0 bis i=6
102         byte = byte >> 1;
103         }
104     }
105     return byte;
106 }
```

Listing A.12: 1_wire_Functions.h

```
1  #ifndef __1_WIRE_FUNCTIONS_H_
2  #define __1_WIRE_FUNCTIONS_H_
3  //*****************************************************************************
4  //
5  // Prototypes for the functions.
6  //
7  //*****************************************************************************
8  extern void one_wire_reset(void);
9  extern void one_wire_bit(int bit);
10 extern void one_wire_write_byte(int byte);
11 extern char one_wire_read_byte(void);
12
13 #endif /* 1_WIRE_FUNCTIONS_H_ */
```

Listing A.13: 1_wire_Measure.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/uart.h"
13 #include "utils/uartstdio.h"
14
15 // eigene Includes
16 #include "globals.h"
17 #include "1_wire_Functions.h"
18 #include "delayMS.h"
19 #include "1_wire_ROM_Commandos.h"
20 #include "1_wire_crc.h"
21
22 void one_wire_measure(void)
23 {
24     int k,l;
25     //k=0-7/8 --> 8-9 ROM Code Längen
26     //l=0-5   --> 6 Sensoren -->
27
28     if(onewire_800ms_wait == 0 && one_wire_measure_active == 1)
29     {
30         // Funktionsaufruf
31         one_wire_reset();
32
```

```
33          // für alle 1-wire Slaves
34          one_wire_write_byte(SKIP_ROM);
35          one_wire_write_byte(CONVERT_TEMP);
36
37      }
38      // mindestens 750ms warten
39      else if(onewire_800ms_wait == 1 && one_wire_measure_active == 1)
40      {
41          //FAN 1-6 Temperaturen Auslesen
42          for(l=0;l<6;l++)
43          {
44              // Funktionsaufruf
45              one_wire_reset();
46
47              one_wire_write_byte(MATCH_ROM);
48              for(k=0;k<=7;k++)
49              {
50                  one_wire_write_byte(ROM_CODE_FAN[l][k]);
51              }
52              one_wire_write_byte(READ_SCRATCHPAD);
53
54              // CODE, der den UART quasi resettet, sonst liest der RX nur Reste der Zeile hier drüber
                    ein
55              UARTConfigSetExpClk(UART7_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
56
57              //Bytes auslesen: SCRATCHPAD_FAN[1-6] (9 Bytes)
58              for(k=0;k<9;k++)
59              {
60                  SCRATCHPAD_FAN[l][k] = one_wire_read_byte();
61              }
62
63              // CODE, der den UART quasi resettet, sonst liest der RX nur Reste der Zeile hier drüber
                    ein
64              UARTConfigSetExpClk(UART7_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 |
                    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
65
66              FAN_TEMPERATURE[l] = ((SCRATCHPAD_FAN[l][1] * 256) + SCRATCHPAD_FAN[l][0])*100 / 16 ;
67              //temperature_celsius = ((temperature_MSB * 256) + temperature_LSB) / 16 ;
68
69
70              // CRC aus SCRATCHPAD Daten berechen:
71              for(k=0;k<8;k++)
72              {
73                  CRC_Data[k] = SCRATCHPAD_FAN[l][k];
74              }
75              SCRATCHPAD_CRC[l] = CRC8_1_wire(0, CRC_Data, 8);
76
77          }
78          onewire_800ms_wait = 0;
79          one_wire_measure_active = 0;
80      }
81
82  }
```

Listing A.14: 1_wire_Measure.h

```
1  #ifndef __1_WIRE_MEASURE_H_
2  #define __1_WIRE_MEASURE_H_
3  //*****************************************************************************
```

```
4  //
5  // Prototypes for the functions.
6  //
7  //*****************************************************************************
8  extern void one_wire_measure(void);
9
10 #endif /* 1_WIRE_MEASURE_H_ */
```

Listing A.15: 1_wire_ROM_Commandos.h

```
1  #ifndef __1_WIRE_ROM_COMMANDOS_H_
2  #define __1_WIRE_ROM_COMMANDOS_H_
3
4  // ROM-Kommandos
5  #define SEARCH_ROM          0xf0 // identifiziert ROM Codes der verfügbaren Slaves im Bussystem.
6                                   // Auch die Anzahl ist bestimmbar
7
8  #define READ_ROM            0x33 // Kommando hat den selben Effekt wie SEARCH_ROM, kann aber nur genutzt werden,
9                                   // wenn nur ein Slave im Bussystem verfügbar ist. Sonst würden Datenkollisionen auftreten
10
11 #define MATCH_ROM           0x55 // dieses Kommando, gefolgt von einem 64-bit ROM-Code wird benutzt um ein bestimmten Slave
12                                  // im Bussystem anzusprechen. Wenn der Slave-Code mit dem empfangenen überein stimmt, wird
13                                  // er als einziger Slave im BUS antworten. Die anderen warten ab dann bis eine Reset Sequenz kommt.
14
15 #define SKIP_ROM            0xcc // wird benutzt um alle BUS Teilnehmer zur gleichen Zeit zu adressieren. Dies kann nützlich sein,
16                                  // um zum Beispiel das Kommando einer Temperaturabfrage zu starten.
17                                   // (BSP: CONVERT_TEMP, 0x44 ...wenn nur ein TN im BUS ist.)
18
19 #define ALARM_SEARCH        0xec // fast das Gleiche wie wie SEARCH_ROM,
20                                  // nur das hier nur die Teilnehmer mit einer gesetzten ALARM_FLAG antworten
21
22 // FUNKTIONS-Kommandos
23 #define CONVERT_TEMP        0x44 // startet eine Temperaturmessung. Die Temperatur wird in den ersten beiden BYTES des SCRATCHPAD gespeichert.
24                                  // Die "Konvertierungs"-Zeit ist Auflösungsabhängig
25                                  // ACHTUNG: Wenn man ein READ während eines Konvertierungsprozesses schickt, antwortet der Teilnehmer mit 0 (low).
26                                  // Wenn der Prozess abgschlossen ist, antwortet der Teilnehmer mit 1 (high).
27                                  // (nur verfügbar, wenn nicht "parasitär"-Versorgt.)
28
29 #define WRITE_SCRATCHPAD    0x4e //
30
31 #define READ_SCRATCHPAD     0xbe //
32
33 #define COPY_SCRATCHPAD     0x48 //
34
35 #define RECALL_E_2          0xb8 //
36
37 #define READ_POWER_SUPPLY   0xb4 //
38
39
40
```

```
41 #endif /* 1_WIRE_ROM_COMMANDOS_H_ */
```

Listing A.16: 1_wire_UART7_Init.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/uart.h"
13 #include "utils/uartstdio.h"
14
15 // eigene Includes
16 #include "globals.h"
17
18 void one_wire_UART7_Init(void)
19 {
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_UART7);
21     // aktiviert UART7 Modul
22     // siehe "Pin Mux Utility" bei der "aktivierung" von UART7
23     // genutzt soll hier werden: Port_C, da dort RX(an PC4) und TX(an PC5) vorhanden sind
24
25     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
26     // aktiviert GPIO Ports an Port_C
27     // benötigt für UART7 --> siehe: PINMUX --> (PC4_U7RX) und (PC5_U7TX)
28
29     GPIOPinConfigure(GPIO_PC4_U7RX);
30     // GPIO-Pin-PC4 als TX (Transmit) Pin einstellen
31
32     GPIOPinConfigure(GPIO_PC5_U7TX);
33     // GPIO-Pin-PC5 als RX (Recieve) Pin einstellen
34
35     GPIOPinTypeUART(GPIO_PORTC_BASE, GPIO_PIN_4 | GPIO_PIN_5);
36     // GPIO PINs von Port_C (PIN4 und PIN5) als UART PINs verwenden
37 }
```

Listing A.17: 1_wire_UART7_Init.h

```
1  #ifndef __1_WIRE_UART7_INIT_H_
2  #define __1_WIRE_UART7_INIT_H_
3  /*******************************************************************************
4   *
5   * Function Declarations
6   *
7   *******************************************************************************/
8  void one_wire_UART7_Init(void);
9
10 #endif /* 1_WIRE_UART7_INIT_H_ */
```

Listing A.18: ADC_Init.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
```

```
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pin_map.h"
10 #include "inc/hw_gpio.h"
11 #include "driverlib/adc.h"
12
13 #include "globals.h"
14
15 void ADC_Init(void)
16 {
17     SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC1);
18     //aktiviert ADC1: diese sind über viele verteilt
19     //siehe "Pin Mux Utility" bei der "aktivierung" von ADC1
20     //genutzt soll hier werden: AINXX an Port_E, PinXXX
21
22     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
23     // aktiviert GPIO Ports an Port_E
24
25     ADCHardwareOversampleConfigure(ADC1_BASE, 64);
26     // 64 Sample-Mess-Werte werden für einen Durchschnittswert verwendet.
27     // Dadurch "wackelt" der Messwert nicht so stark
28
29     GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 );
30     // aktiviert die GPIO Pins als ADC Input Pins
31     // GPIO_PIN_0 = PORT_E_PIN_0 = AIN03
32     // GPIO_PIN_1 = PORT_E_PIN_1 = AIN02
33     // GPIO_PIN_2 = PORT_E_PIN_2 = AIN01
34
35     ADCSequenceConfigure(ADC1_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
36     // ADC0 konfigurieren: (Benutze:ADC1, Sample Sequencer:1, Prozessor soll Sequence triggern,
           höchste Prirorität:0)
37
38     ADCSequenceStepConfigure(ADC1_BASE, 1, 0, ADC_CTL_CH3);
39     // Einstellungen für AIN03
40     //ADCSequenceStepConfigure(ui32Base, ui32SequenceNum, ui32Step, ui32Config);
41
42     ADCSequenceStepConfigure(ADC1_BASE, 1, 1, ADC_CTL_CH2);
43     // Einstellungen für AIN02
44
45     ADCSequenceStepConfigure(ADC1_BASE, 1, 2, ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END);
46     // Einstellungen für AIN01
47     // die extra Einstellungen hinten sind für: "Interrupt-Generation" und "finales-Sample-der-
           Sequence"
48
49     ADCSequenceEnable(ADC1_BASE, 1);
50     //aktiviert "ADC-Sequencer-1"
51 }
```

Listing A.19: ADC_Init.h

```
1  #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_ADC_INIT_H_
2  #define SCHUTZCONTROLLER_MIT_DISPLAY_ADC_INIT_H_
3
4  /*****************************************************************************
5   *
6   * Function Declarations
```

```
 *
 ******************************************************************************/
void ADC_Init(void);

#endif /* SCHUTZCONTROLLER_MIT_DISPLAY_ADC_INIT_H_ */
```

Listing A.20: ADC_Measure.c

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/fpu.h"
#include "driverlib/gpio.h"
#include "driverlib/debug.h"
#include "driverlib/pin_map.h"
#include "inc/hw_gpio.h"
#include "driverlib/adc.h"

#include "globals.h"

void ADC_Measure(void)
{
    ADCIntClear(ADC1_BASE, 1);
    // Der "Indikator", dass der Sequencer und ADC Prozess fertig sind, ist das "ADC-Interrupt-Status-Flag"
    // Es ist "gutes Programmieren" vorher immer zu überprüfen, ob das "Flag" gelöscht wurde.
    // dieser Schritt wird nach Beenden der Schleife das BIT löschen. (also am Anfang der nächsten Schleife)

    ADCProcessorTrigger(ADC1_BASE, 1);
    // triggert die ADC Konvertierung per Software
    // ADC Konvertierungen können auch von vielen anderen Quellen getriggert werden

    while(!ADCIntStatus(ADC1_BASE, 1, false))
    {
    }
    // reine Warteschleife !!!
    // Wartet, bis die Konvertierung fertig ist.
    // Der Indikator, der am Anfang der Schleife gelöscht wurde.
    // Offensichtlich könnte man das besser lösen (verschwendet CPU Zyklen), z.B. mit einem Interrupt.

    ADCSequenceDataGet(ADC1_BASE, 1, ADC_Value_Array);
    // Ist die vorige while() Schleife verlassen, wurde die Konvertierung abgeschlossen.
    // Jetzt kann der ADC Wert aus dem "ADC-Sample-Sequencer-1" ausgelesen werden.
    // Diese Funktion liest die Werte aus und speichert sie in dem Array "ADC_Value_Array" [4 "uint32_t" Elemente]

    ADC_Value_AIN3 = ADC_Value_Array[0];
    ADC_Value_AIN2 = ADC_Value_Array[1];
    ADC_Value_AIN1 = ADC_Value_Array[2];
    // schreibt die Variablenwerte aus dem Array in die Variablen

//  Hallsensor_3_Ampere = abs((2500 - ( ( ( ( ( ADC_Value_AIN3 * 803 ) / 1000 ) * ( 36100 + 67600 ) ) / 67600 ) ) )) / 4;
    // --> 803mV pro ADC-Step (bei 3,29V)
    // --> Sensivity = 4 mV/A (bei +- 500A Messbereich)
    // --> Offset = 2500 mV
```

```
48      // --> verwendete Spannungsteilerwiderstände: 36100 Ohm + 68100 Ohm
49      Hallsensor_Ampere[1] = abs((2500 - ( ( ( ( ADC_Value_AIN2 * 803 ) / 1000 ) * ( 36100 + 67600 )
            ) / 67600 ) ) )) / 4;
50      Hallsensor_Ampere[0] = abs((2500 - ( ( ( ( ADC_Value_AIN1 * 803 ) / 1000 ) * ( 36100 + 67600 )
            ) / 67600 ) ) )) / 4;
51      // ACHTUNG ZUR GENAUIGKEIT MUSS WIDERSTAND MÖGLICHST GENAU GEMESSEN WERDEN !!!!!!!!!!!!
52
53  }
```

Listing A.21: ADC_Measure.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_ADC_MEASURE_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_ADC_MEASURE_H_
3
4   /*****************************************************************************
5    *
6    * Function Declarations
7    *
8    *****************************************************************************/
9   void ADC_Measure(void);
10
11  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_ADC_MEASURE_H_ */
```

Listing A.22: delayMS.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12  #include "driverlib/uart.h"
13  #include "utils/uartstdio.h"
14
15  // eigene Includes
16  #include "globals.h"
17
18  void delayMS(int ms)
19  {
20      SysCtlDelay( (ui32SysClkFreq/(3*1000))*ms ) ;  // less accurate
21  }
```

Listing A.23: delayMS.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_DELAYMS_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_DELAYMS_H_
3   /*****************************************************************************
4    *
5    * Function Declarations
6    *
7    *****************************************************************************/
8   void delayMS(int);
9
10  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_DELAYMS_H_ */
```

Listing A.24: globals.h

```
#ifndef EXTERN
#define EXTERN extern
#endif

// durch den #define (in der main.c vor #include "globals.h") expandiert das Makro EXTERN in global.h zu
// einem Leerstring. Dadurch wird aus
// EXTERN uint32_t PWM_FREQUENCY;
// die Zeile
// uint32_t PWM_FREQUENCY;
// und das definiert die Variable PWM_FREQUENCY

EXTERN uint32_t FAN_PWM_FREQUENCY; // 25000 Hz Frequenz für die PWMs

EXTERN uint32_t ui32SysClkFreq; // Hilfsvariable zum Speichern der SystemClockFrequenz

EXTERN uint32_t schalter0;
EXTERN uint32_t schalter1;

EXTERN uint32_t TESTMERKER;

/******************** Timer Variablen Anfang ***************************/
EXTERN uint32_t timer_a_value;
EXTERN uint32_t timer_a_length;
EXTERN uint32_t timer_b_length;
EXTERN uint32_t TIMER2A_Value;
EXTERN uint32_t TIMER2B_Value;
EXTERN uint32_t TIMER3A_Value;
EXTERN uint32_t TIMER3B_Value;
EXTERN uint32_t TIMER4A_Value;
EXTERN uint32_t TIMER4B_Value;
EXTERN volatile uint32_t t; // Zählervariable für den Timer5A
EXTERN volatile uint32_t Sekunden; // --> 0 -  4.294.967.295 Sekunden // wird alle 5 Sekunden um 5 erhöht
// TIMER5B
EXTERN volatile uint32_t Timer_5B_Value;            // Wert in Microsekunden (ACHTUNG !!! wird periodisch um 500us erhöht)
EXTERN uint32_t TOGGLE_500ms;
EXTERN uint32_t TOGGLE_1000ms;
/******************** Timer Variablen Anfang ***************************/

/******************** Lüfter Variablen Anfang ***************************/
// Lüfter_1-6:
EXTERN uint32_t FAN_DUTYCYCLE_PERCENT[6];
EXTERN uint32_t FAN_RPM[6];

// 6 Sensoren
EXTERN uint32_t FAN_TEMPERATURE[6];

/******************** Lüfter Variablen Ende ***************************/

/******************** 1-wire Variablen Anfang ***************************/
EXTERN char DEVICE_PULSE;
EXTERN char buffer;
EXTERN int DEVICE_AVAILABLE; // speichern ob DEVICE Vorhanden ist 0=NEIN, 1=JA

EXTERN char SCRATCHPAD_FAN[6][9];
//SCRATCHPAD[0]: temperature_MSB;
```

```
56  //SCRATCHPAD[1]: temperature_LSB;
57  //SCRATCHPAD[2]: TH_Register;
58  //SCRATCHPAD[3]: TL_Register;
59  //SCRATCHPAD[4]: Config_Register;
60  //SCRATCHPAD[5]: Reserved_0xFF;
61  //SCRATCHPAD[6]: Reserved;
62  //SCRATCHPAD[7]: Reserved_0x10;
63  //SCRATCHPAD[8]: SCRATCHPAD_CRC;
64
65  EXTERN char ROM_CODE_FAN[6][8];
66  //ROM_CODE_FAN_1[0]: Family_Code;
67  //ROM_CODE_FAN_1[1]: ROM_Code_1;
68  //ROM_CODE_FAN_1[2]: ROM_Code_2;
69  //ROM_CODE_FAN_1[3]: ROM_Code_3;
70  //ROM_CODE_FAN_1[4]: ROM_Code_4;
71  //ROM_CODE_FAN_1[5]: ROM_Code_5;
72  //ROM_CODE_FAN_1[6]: ROM_Code_6;
73  //ROM_CODE_FAN_1[7]: ROM_Code_CRC;
74
75  //CRC Variablen
76  EXTERN unsigned long SCRATCHPAD_CRC[6];
77
78  EXTERN unsigned char CRC_Data[256];
79
80  //CRC Check Variablen
81  EXTERN char CRC_OK[6][5];
82
83  // Zählervariablen
84  uint32_t i,k;
85
86  /******************** 1-wire Variablen Ende ****************************/
87
88  /******************** PWM Variablen Anfang *****************************/
89  EXTERN volatile uint32_t ui32Load;
90  EXTERN volatile uint32_t ui32PWMClock;
91  /******************** PWM Variablen Ende *******************************/
92
93  /******************** ADC/Hallsensor Variablen Anfang ******************/
94  EXTERN uint32_t ADC_Value_Array[4]; // Array für die Variablen
95
96  EXTERN volatile uint32_t ADC_Value_AIN1; // Variable für Messung
97  EXTERN volatile uint32_t ADC_Value_AIN2;
98  EXTERN volatile uint32_t ADC_Value_AIN3;
99
100 EXTERN uint32_t Hallsensor_Ampere[2];
101 /******************** ADC/Hallsensor Variablen Ende ********************/
102
103 /********************     Display Variablen Anfang     ******************/
104 EXTERN uint32_t g_ui32Panel; // Nummer des derzeit aktiven Panels
105 /********************     Display Variablen Ende       ******************/
106
107 /******************** State Machine Variablen Anfang  ******************/
108
109 // Error Level
110 EXTERN volatile uint32_t ERROR_LEVEL;           // Status des aktuellen Errorlevels
111 // L#00 - kein Fehler
112 // L#01 - Warnungslevel
113 // L#02 - Fehlerfall
114 // L#03 - Kommunikationsfehler
115
```

```
116 // 0 oder 1 : 0 = Initialisierung bereits erfolgt / 1 = neu initialisieren
117 EXTERN volatile uint32_t SM_INIT;
118
119 EXTERN volatile uint32_t POWER_SUPPLY_ACTIVE;
120 EXTERN volatile uint32_t POWER_SUPPLY_ON_OFF;
121
122 // Hilfsvariable, die benötigt wird, das TOUCHEINGABE defekt...
123 EXTERN volatile uint32_t DISPLAY_ROTATER;
124
125 // Zeitvariablen
126 EXTERN volatile uint32_t main_Timer_Value;          // Wert in Microsekunden (ACHTUNG !!! wird
        periodisch um 500us erhöht)
127 EXTERN volatile uint32_t onewire_Timer_Value;       // Wert für die 800ms Wartezeit für die 1-wire
        Temperaturmessung
128 EXTERN volatile uint32_t onewire_800ms_wait;        // 0 = nicht abgeschlossen / 1 = abgeschlossen
129
130 // Semaphoren
131 EXTERN volatile uint32_t one_wire_measure_active;
132 EXTERN volatile uint32_t SM_Soft_Stop_active;
133
134 // Zähler
135 EXTERN volatile uint32_t z;
136
137 EXTERN volatile uint32_t SYSTEM_STATUS_TEXT;        // für die Textausgabe auf Display
138 // 0 = INIT
139 // 1 = RUN
140 // 2 = ERROR
141 // 3 = SOFT-STOP
142
143 /******************** State Machine Variablen Ende    ******************/
144
145
146 /******************** Grenzwerte Anfang    ******************************/
147
148 // Temperaturen der Lüfter
149 EXTERN volatile uint32_t LIMIT_FAN_TEMP_WARNING[6];
150 EXTERN volatile uint32_t LIMIT_FAN_TEMP_ERROR[6];
151
152 // Stromstärken
153 EXTERN volatile uint32_t LIMIT_HALL_CURRENT_WARNING[2];
154 EXTERN volatile uint32_t LIMIT_HALL_CURRENT_ERROR[2];
155
156 //
157
158 /******************** Grenzwerte Ende      ******************************/
159
160 /******************** EEPROM     Anfang    ******************************/
161
162 // Array für die zu speichernden Variablen
163 EXTERN uint32_t ROM_VARIABLES[2];
164 // ROM_VARIABLES[0] = RELAY_COUNTER[0]
165 // ROM_VARIABLES[1] = RELAY_COUNTER[1]
166
167 // ACHTUNG --- !!! in der SM_Functions.c wird dies zugewiesen
168 // in den Funktionen: rom_variables_read(); und rom_variables_write();
169
170 /******************** EEPROM     Ende      ******************************/
171
172 /******************** Relais Counter Anfang ******************************/
173
```

```
174 // Array für die Relais Variablen
175 EXTERN uint32_t RELAY_COUNTER[2];
176 // RELAY_COUNTER[0] = Relais 1
177 // RELAY_COUNTER[1] = Relais 2
178
179 /******************* Relais Counter Ende ******************************/
180
181 /******************* GPIO Buttons Anfang   ******************************/
182 EXTERN volatile uint32_t YELLOW_BUTTON;
183 EXTERN volatile uint32_t RED_BUTTON;
184 EXTERN volatile uint32_t BLACK_BUTTON;
185 /******************* GPIO Buttons Ende   ******************************/
```

Listing A.25: GPIO_Int_Handler.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/timer.h"
13
14 #include "globals.h"
15 #include "Timer_Init.h"
16 #include "delayMS.h"
17
18 void GPIO_Int_Handler()
19 {
20 //  uint32_t status=0;
21 //
22 //  status = GPIOIntStatus(GPIO_PORTJ_BASE, true);
23 //
24 //  GPIOIntClear(GPIO_PORTJ_BASE, status);
25 //
26 //  schalter0 = GPIOPinRead(GPIO_PORTJ_BASE, GPIO_PIN_0);
27 //  schalter1 = GPIOPinRead(GPIO_PORTJ_BASE, GPIO_PIN_1);
28 //
29 //  if( (schalter0 & GPIO_PIN_0)== 0 )
30 //  {
31 //  }
32 //
33 //  if( (schalter1 & GPIO_PIN_1)== 0 )
34 //  {
35 //  }
36
37 }
38
39 void GPIO_Int_Handler_PORT_L()
40 {
41     uint32_t status=0;
42     uint32_t red;
43     uint32_t yellow;
44     uint32_t black;
45
```

```
46      status = GPIOIntStatus(GPIO_PORTL_BASE, true);
47
48      GPIOIntClear(GPIO_PORTL_BASE, status);
49
50      delayMS(50);
51
52      red     = GPIOPinRead(GPIO_PORTL_BASE, GPIO_PIN_0);
53      yellow  = GPIOPinRead(GPIO_PORTL_BASE, GPIO_PIN_1);
54      black   = GPIOPinRead(GPIO_PORTL_BASE, GPIO_PIN_2);
55
56      if( (red & GPIO_PIN_0)== 0 )
57      {
58          RED_BUTTON = 1;
59      }
60
61      if( (yellow & GPIO_PIN_1)== 0 )
62      {
63          YELLOW_BUTTON = 1;
64      }
65
66      if( (black & GPIO_PIN_2)== 0 )
67      {
68          BLACK_BUTTON = 1;
69      }
70  }
```

Listing A.26: GPIO_Int_Handler.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_GPIO_INT_HANDLER_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_GPIO_INT_HANDLER_H_
3
4   /*****************************************************************************
5    *
6    * Function Declarations
7    *
8    *****************************************************************************/
9
10  void GPIO_Int_Handler(void);
11  void GPIO_Int_Handler_PORT_L(void);
12
13  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_GPIO_INT_HANDLER_H_ */
```

Listing A.27: GPIO_Ports_Init.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12
13  // eigene Includes
14  #include "GPIO_Int_Handler.h"
15  #include "globals.h"
```

```
16
17 void GPIO_Ports_Init(void)
18 {
19     // Ausgänge an PORTH ----------------------------------------------A
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
21     // aktiviert GPIO Ports an Port_H
22
23     // Ausgänge an PORTM
24     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOM);
25     // aktiviert GPIO Ports an Port_M
26
27     // Ausgang für LED-ROT
28     GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_0);
29     // setzt GPIO Port_H Pin0 als Output Pin
30
31     // Ausgang für LED-GRÜN
32     GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_1);
33     // setzt GPIO Port_H Pin1 als Output Pin
34
35     // Ausgang für RELAY-1
36     GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_2);
37     // setzt GPIO Port_H Pin2 als Output Pin
38
39     // Ausgang für RELAY-2
40     GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_3);
41     // setzt GPIO Port_H Pin3 als Output Pin
42
43     // Ausgang für AKUSTISCHES-SIGNAL
44     GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_4);
45     // setzt GPIO Port_H Pin4 als Output Pin
46
47     // Ausgang für SCHUTZCONTROLLER-BEREIT-SIGNAL
48     GPIOPinTypeGPIOOutput(GPIO_PORTH_BASE, GPIO_PIN_5);
49     // setzt GPIO Port_H Pin5 als Output Pin
50
51     // Ausgang für POWER_SUPPLY (hier für den TEST Notwendig, soll später der ZYklierconroller
           übernehmen)
52     GPIOPinTypeGPIOOutput(GPIO_PORTM_BASE, GPIO_PIN_5);
53     // setzt GPIO Port_H Pin6 als Output Pin
54
55     // ALLE Pins auf 0 setzen (zur Sicherheit)
56     GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_7 | GPIO_PIN_6 | GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
           GPIO_PIN_2 | GPIO_PIN_1 | GPIO_PIN_0, 0b00000000);
57
58     // konfiguriert PINS
59     GPIOPadConfigSet(GPIO_PORTM_BASE ,GPIO_PIN_4, GPIO_STRENGTH_12MA, GPIO_PIN_TYPE_STD);
60     //--------------------------------------------------------------A
61
62
63     // Eingänge an PORTL ----------------------------------------------E
64     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
65     // aktiviert GPIO Ports an Port_L
66
67     // Eingang für BUTTON-ROT
68     GPIOPinTypeGPIOInput(GPIO_PORTL_BASE, GPIO_PIN_0);
69     // setzt GPIO Port_L Pin0 als Input Pin
70
71     // Eingang für BUTTON-GELB
72     GPIOPinTypeGPIOInput(GPIO_PORTL_BASE, GPIO_PIN_1);
73     // setzt GPIO Port_L Pin1 als Input Pin
```

```
74
75      // Eingang für BUTTON-SCHWARZ
76      GPIOPinTypeGPIOInput(GPIO_PORTL_BASE, GPIO_PIN_2);
77      // setzt GPIO Port_L Pin2 als Input Pin
78
79      // Eingang für Zykliercontroller bereit
80      GPIOPinTypeGPIOInput(GPIO_PORTL_BASE, GPIO_PIN_3);
81      // setzt GPIO Port_L Pin3 als Input Pin
82
83      // Eingang für POWER_SUPPLY bereit (nur für den TESTLAUF)
84      GPIOPinTypeGPIOInput(GPIO_PORTL_BASE, GPIO_PIN_4);
85      // setzt GPIO Port_L Pin4 als Input Pin
86
87      // konfiguriert PINS
88      GPIOPadConfigSet(GPIO_PORTL_BASE ,GPIO_PIN_0, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
89      GPIOPadConfigSet(GPIO_PORTL_BASE ,GPIO_PIN_1, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
90      GPIOPadConfigSet(GPIO_PORTL_BASE ,GPIO_PIN_2, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
91      GPIOPadConfigSet(GPIO_PORTL_BASE ,GPIO_PIN_3, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
92      GPIOPadConfigSet(GPIO_PORTL_BASE ,GPIO_PIN_4, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
93
94      //-------------------------------------------------------------E
95
96      // Interrupts für die Buttons
97
98  // RED
99      GPIOIntTypeSet(GPIO_PORTL_BASE, GPIO_PIN_0, GPIO_BOTH_EDGES);
100     // GPIO Interrupt auf steigende Flanke stellen
101 // YELLOW
102     GPIOIntTypeSet(GPIO_PORTL_BASE, GPIO_PIN_1, GPIO_BOTH_EDGES);
103     // GPIO Interrupt auf steigende Flanke stellen
104 // BLACK
105     GPIOIntTypeSet(GPIO_PORTL_BASE, GPIO_PIN_2, GPIO_BOTH_EDGES);
106     // GPIO Interrupt auf steigende Flanke stellen
107
108     GPIOIntRegister(GPIO_PORTL_BASE, GPIO_Int_Handler_PORT_L);
109     // Für den GPIO-Interrupt die aufzurufende Funktion wählen
110
111     GPIOIntEnable(GPIO_PORTL_BASE, GPIO_INT_PIN_0 | GPIO_INT_PIN_1 | GPIO_INT_PIN_2);
112     // GPIO-Interrupt aktivieren
113 }
```

Listing A.28: GPIO_Ports_Init.h

```
1  #ifndef GPIO_PORTS_INIT_H_
2  #define GPIO_PORTS_INIT_H_
3  //*****************************************************************************
4  //
5  // Prototypes for the functions.
6  //
7  //*****************************************************************************
8  extern void GPIO_Ports_Init(void);
9
10 #endif /* GPIO_PORTS_INIT_H_ */
```

Listing A.29: main.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include <stdlib.h>
```

```
4  #include <stdio.h>
5  #include "inc/hw_memmap.h"
6  #include "inc/hw_types.h"
7  #include "inc/hw_nvic.h"
8  #include "inc/hw_sysctl.h"
9  #include "inc/hw_gpio.h"
10 #include "driverlib/sysctl.h"
11 #include "driverlib/fpu.h"
12 #include "driverlib/gpio.h"
13 #include "driverlib/debug.h"
14 #include "driverlib/pwm.h"
15 #include "driverlib/pin_map.h"
16 #include "driverlib/timer.h"
17 #include "driverlib/uart.h"
18 #include "driverlib/interrupt.h"
19 #include "driverlib/flash.h"
20 #include "driverlib/systick.h"
21 #include "driverlib/udma.h"
22 #include "driverlib/rom.h"
23 #include "driverlib/rom_map.h"
24 #include "utils/uartstdio.h"
25 #include "utils/ustdlib.h"
26 #include "grlib/grlib.h"
27 #include "grlib/widget.h"
28 #include "grlib/canvas.h"
29 #include "grlib/checkbox.h"
30 #include "grlib/container.h"
31 #include "grlib/pushbutton.h"
32 #include "grlib/radiobutton.h"
33 #include "grlib/slider.h"
34 #include "drivers/Kentec320x240x16_ssd2119_8bit.h"
35 #include "drivers/touch.h"
36 #include "driverlib/eeprom.h"
37
38 // eigene #include Dateien
39 #define EXTERN // benötigt für die globals.h
40 #include "globals.h"
41 #include "PWM_init.h"
42 #include "GPIO_Int_Init.h"
43 #include "GPIO_Int_Handler.h"
44 #include "Timer_Init.h"
45 #include "UART_Init.h"
46 #include "TIMER5A_Interrupt_Handler.h"
47 #include "TIMER5B_Interrupt_Handler.h"
48 #include "1_wire_crc.h"
49 #include "1_wire_ROM_Commandos.h"
50 #include "1_wire_Functions.h"
51 #include "1_wire_UART7_Init.h"
52 #include "1_wire_Measure.h"
53 #include "1_wire_Devices.h"
54 #include "delayMS.h"
55 #include "ADC_Init.h"
56 #include "ADC_Measure.h"
57 #include "images.h"
58 #include "SM_Functions.h"
59 #include "GPIO_Ports_Init.h"
60 #include "Startvalues.h"
61
62 //*****************************************************************************
63 //
```

```
64  // Forward declarations for the globals required to define the widgets at
65  // compile-time.
66  //
67  //*****************************************************************************
68  void OnPrevious(tWidget *psWidget);
69  void OnNext(tWidget *psWidget);
70  void On_1_Status(tWidget *psWidget, tContext *psContext);
71  void On_2_Lastkreis(tWidget *psWidget, tContext *psContext);
72  void On_3_Ladekreis(tWidget *psWidget, tContext *psContext);
73  void On_4_Details(tWidget *psWidget, tContext *psContext);
74  void OnCanvasPaint(tWidget *psWidget, tContext *psContext);
75  extern tCanvasWidget g_psPanels[];
76
77  //*****************************************************************************
78  //
79  // The first panel, which contains introductory text explaining the
80  // application.
81  //
82  //*****************************************************************************
83  Canvas(g_sIntroduction, g_psPanels, 0, 0, &g_sKentec320x240x16_SSD2119, 0, 24,
84         320, 166, CANVAS_STYLE_APP_DRAWN, 0, 0, 0, 0, 0, 0, On_1_Status);
85
86  //*****************************************************************************
87  //
88  // The second panel, which demonstrates the graphics primitives.
89  //
90  //*****************************************************************************
91  Canvas(g_sPrimitives, g_psPanels + 1, 0, 0, &g_sKentec320x240x16_SSD2119, 0,
92         24, 320, 166, CANVAS_STYLE_APP_DRAWN, 0, 0, 0, 0, 0, 0,
93         On_2_Lastkreis);
94
95  Canvas(g_sPrimitives2, g_psPanels + 2, 0, 0, &g_sKentec320x240x16_SSD2119, 0,
96         24, 320, 166, CANVAS_STYLE_APP_DRAWN, 0, 0, 0, 0, 0, 0,
97         On_3_Ladekreis);
98
99  //Canvas(g_details, g_psPanels + 3, 0, 0, &g_sKentec320x240x16_SSD2119, 0,
100 //        24, 320, 166, CANVAS_STYLE_APP_DRAWN, 0, 0, 0, 0, 0, 0,
101 //       On_4_Details);
102
103 //*****************************************************************************
104 //
105 // An array of canvas widgets, one per panel.  Each canvas is filled with
106 // black, overwriting the contents of the previous panel.
107 //
108 //*****************************************************************************
109 tCanvasWidget g_psPanels[] =
110 {
111     CanvasStruct(0, 0, &g_sIntroduction, &g_sKentec320x240x16_SSD2119, 0, 24,
112                  320, 166, CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0),
113     CanvasStruct(0, 0, &g_sPrimitives, &g_sKentec320x240x16_SSD2119, 0, 24,
114                  320, 166, CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0),
115     CanvasStruct(0, 0, &g_sPrimitives2, &g_sKentec320x240x16_SSD2119, 0, 24,
116                  320, 166, CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0),
117 //  CanvasStruct(0, 0, &g_details, &g_sKentec320x240x16_SSD2119, 0, 24,
118 //                320, 166, CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0),
119 };
120
121 //*****************************************************************************
122 //
123 // The number of panels.
```

```
124 //
125 //*****************************************************************************
126 #define NUM_PANELS              (sizeof(g_psPanels) / sizeof(g_psPanels[0]))
127
128 //*****************************************************************************
129 //
130 // The names for each of the panels, which is displayed at the bottom of the
131 // screen.
132 //
133 //*****************************************************************************
134 char *g_pcPanei32Names[] =
135 {
136         "       Status       ",
137         "      Lastkreis     ",
138         "      Ladekreis     ",
139         "       Details      "
140 };
141
142 //*****************************************************************************
143 //
144 // The buttons and text across the bottom of the screen.
145 //
146 //*****************************************************************************
147 RectangularButton(g_sPrevious, 0, 0, 0, &g_sKentec320x240x16_SSD2119, 0, 190,
148                   50, 50, PB_STYLE_FILL, ClrBlack, ClrBlack, 0, ClrSilver,
149                   &g_sFontCm20, "-", g_pui8Blue50x50, g_pui8Blue50x50Press, 0, 0,
150                   OnPrevious);
151
152 Canvas(g_sTitle, 0, 0, 0, &g_sKentec320x240x16_SSD2119, 50, 190, 220, 50,
153        CANVAS_STYLE_TEXT | CANVAS_STYLE_TEXT_OPAQUE, 0, 0, ClrSilver,
154        &g_sFontCm20, 0, 0, 0);
155
156 RectangularButton(g_sNext, 0, 0, 0, &g_sKentec320x240x16_SSD2119, 270, 190,
157                   50, 50, PB_STYLE_IMG | PB_STYLE_TEXT, ClrBlack, ClrBlack, 0,
158                   ClrSilver, &g_sFontCm20, "+", g_pui8Blue50x50,
159                   g_pui8Blue50x50Press, 0, 0, OnNext);
160
161 //*****************************************************************************
162 //
163 // The panel that is currently being displayed.
164 //
165 //*****************************************************************************
166
167
168
169 //*****************************************************************************
170 //
171 // Handles presses of the previous panel button.
172 //
173 //*****************************************************************************
174 void
175 OnPrevious(tWidget *psWidget)
176 {
177     //
178     // There is nothing to be done if the first panel is already being
179     // displayed.
180     //
181     if(g_ui32Panel == 0)
182     {
183         return;
```

```
184         }
185
186         //
187         // Remove the current panel.
188         //
189         WidgetRemove((tWidget *)(g_psPanels + g_ui32Panel));
190
191         //
192         // Decrement the panel index.
193         //
194         g_ui32Panel--;
195
196         //
197         // Add and draw the new panel.
198         //
199         WidgetAdd(WIDGET_ROOT, (tWidget *)(g_psPanels + g_ui32Panel));
200         WidgetPaint((tWidget *)(g_psPanels + g_ui32Panel));
201
202         //
203         // Set the title of this panel.
204         //
205         CanvasTextSet(&g_sTitle, g_pcPanei32Names[g_ui32Panel]);
206         WidgetPaint((tWidget *)&g_sTitle);
207
208         //
209         // See if this is the first panel.
210         //
211         if(g_ui32Panel == 0)
212         {
213             //
214             // Clear the previous button from the display since the first panel is
215             // being displayed.
216             //
217             PushButtonImageOff(&g_sPrevious);
218             PushButtonTextOff(&g_sPrevious);
219             PushButtonFillOn(&g_sPrevious);
220             WidgetPaint((tWidget *)&g_sPrevious);
221         }
222
223         //
224         // See if the previous panel was the last panel.
225         //
226         if(g_ui32Panel == (NUM_PANELS - 2))
227         {
228             //
229             // Display the next button.
230             //
231             PushButtonImageOn(&g_sNext);
232             PushButtonTextOn(&g_sNext);
233             PushButtonFillOff(&g_sNext);
234             WidgetPaint((tWidget *)&g_sNext);
235         }
236
237     }
238
239     //*****************************************************************************
240     //
241     // Handles presses of the next panel button.
242     //
243     //*****************************************************************************
```

```
244 void
245 OnNext(tWidget *psWidget)
246 {
247     //
248     // There is nothing to be done if the last panel is already being
249     // displayed.
250     //
251     if(g_ui32Panel == (NUM_PANELS - 1))
252     {
253         return;
254     }
255
256     //
257     // Remove the current panel.
258     //
259     WidgetRemove((tWidget *)(g_psPanels + g_ui32Panel));
260
261     //
262     // Increment the panel index.
263     //
264     g_ui32Panel++;
265
266     //
267     // Add and draw the new panel.
268     //
269     WidgetAdd(WIDGET_ROOT, (tWidget *)(g_psPanels + g_ui32Panel));
270     WidgetPaint((tWidget *)(g_psPanels + g_ui32Panel));
271
272     //
273     // Set the title of this panel.
274     //
275     CanvasTextSet(&g_sTitle, g_pcPanei32Names[g_ui32Panel]);
276     WidgetPaint((tWidget *)&g_sTitle);
277
278     //
279     // See if the previous panel was the first panel.
280     //
281     if(g_ui32Panel == 1)
282     {
283         //
284         // Display the previous button.
285         //
286         PushButtonImageOn(&g_sPrevious);
287         PushButtonTextOn(&g_sPrevious);
288         PushButtonFillOff(&g_sPrevious);
289         WidgetPaint((tWidget *)&g_sPrevious);
290     }
291
292     //
293     // See if this is the last panel.
294     //
295     if(g_ui32Panel == (NUM_PANELS - 1))
296     {
297         //
298         // Clear the next button from the display since the last panel is being
299         // displayed.
300         //
301         PushButtonImageOff(&g_sNext);
302         PushButtonTextOff(&g_sNext);
303         PushButtonFillOn(&g_sNext);
```

```
304             WidgetPaint((tWidget *)&g_sNext);
305         }
306
307     }
308
309     void OnRotateScreens()       // eigene Funktion, da TOUCHEINGABE KAPUTT ist !!!
310     {
311         //
312         // There is nothing to be done if the last panel is already being
313         // displayed.
314         //
315         if(g_ui32Panel == (NUM_PANELS - 1))
316         {
317             //
318             // Remove the current panel.
319             //
320             WidgetRemove((tWidget *)(g_psPanels + g_ui32Panel));
321
322             // wieder auf 0 setzen
323             g_ui32Panel = 0;
324
325             //
326             // Add and draw the new panel.
327             //
328             WidgetAdd(WIDGET_ROOT, (tWidget *)(g_psPanels + g_ui32Panel));
329             WidgetPaint((tWidget *)(g_psPanels + g_ui32Panel));
330
331             //
332             // Set the title of this panel.
333             //
334             CanvasTextSet(&g_sTitle, g_pcPanei32Names[g_ui32Panel]);
335             WidgetPaint((tWidget *)&g_sTitle);
336
337             //
338             // See if the previous panel was the first panel.
339             //
340             if(g_ui32Panel == 1)
341             {
342                 //
343                 // Display the previous button.
344                 //
345                 PushButtonImageOn(&g_sPrevious);
346                 PushButtonTextOn(&g_sPrevious);
347                 PushButtonFillOff(&g_sPrevious);
348                 WidgetPaint((tWidget *)&g_sPrevious);
349             }
350
351             //
352             // See if this is the last panel.
353             //
354             if(g_ui32Panel == (NUM_PANELS - 1))
355             {
356                 //
357                 // Clear the next button from the display since the last panel is being
358                 // displayed.
359                 //
360                 PushButtonImageOff(&g_sNext);
361                 PushButtonTextOff(&g_sNext);
362                 PushButtonFillOn(&g_sNext);
363                 WidgetPaint((tWidget *)&g_sNext);
```

```
364             }
365
366             return;
367         }
368
369         //
370         // Remove the current panel.
371         //
372         WidgetRemove((tWidget *)(g_psPanels + g_ui32Panel));
373
374         //
375         // Increment the panel index.
376         //
377         g_ui32Panel++;
378
379         //
380         // Add and draw the new panel.
381         //
382         WidgetAdd(WIDGET_ROOT, (tWidget *)(g_psPanels + g_ui32Panel));
383         WidgetPaint((tWidget *)(g_psPanels + g_ui32Panel));
384
385         //
386         // Set the title of this panel.
387         //
388         CanvasTextSet(&g_sTitle, g_pcPanei32Names[g_ui32Panel]);
389         WidgetPaint((tWidget *)&g_sTitle);
390
391         //
392         // See if the previous panel was the first panel.
393         //
394         if(g_ui32Panel == 1)
395         {
396             //
397             // Display the previous button.
398             //
399             PushButtonImageOn(&g_sPrevious);
400             PushButtonTextOn(&g_sPrevious);
401             PushButtonFillOff(&g_sPrevious);
402             WidgetPaint((tWidget *)&g_sPrevious);
403         }
404
405         //
406         // See if this is the last panel.
407         //
408         if(g_ui32Panel == (NUM_PANELS - 1))
409         {
410             //
411             // Clear the next button from the display since the last panel is being
412             // displayed.
413             //
414             PushButtonImageOff(&g_sNext);
415             PushButtonTextOff(&g_sNext);
416             PushButtonFillOn(&g_sNext);
417             WidgetPaint((tWidget *)&g_sNext);
418         }
419
420     }
421
422     //*****************************************************************************
423     //
```

```
424 // Handles paint requests for the introduction canvas widget.
425 //
426 //*****************************************************************************
427 void
428 On_1_Status(tWidget *psWidget, tContext *psContext)
429 {
430     char DISPLAY_ERROR_LEVEL[10];
431     sprintf(DISPLAY_ERROR_LEVEL, "%10d", ERROR_LEVEL);
432 
433     char DISPLAY_RUNTIME[10];
434     sprintf(DISPLAY_RUNTIME, "%10d", Sekunden);
435 
436 
437     GrContextFontSet(psContext, &g_sFontCm18);
438     GrContextForegroundSet(psContext, ClrSilver);
439 
440     //
441     GrContextForegroundSet(psContext, ClrSilver);
442     GrStringDraw(psContext, "BATSEN", -1, 80, 32, 0);
443     GrStringDraw(psContext, "SCHUTZCONTROLLER.", -1, 80, 50, 0);
444 
445     // erst den Text schreiben
446     GrStringDraw(psContext, "Status des Controllers:", -1, 10, 74, 0);
447     GrStringDraw(psContext, "Fehlerlevel:", -1, 10, 92, 0);
448     GrStringDraw(psContext, "Laufzeit (in sek.):", -1, 10, 110, 0);
449     //GrStringDraw(psContext, "", -1, 0, 128, 0);
450 
451     // dann die Variablen als Text schreiben
452     if(SYSTEM_STATUS_TEXT == 0)
453     {
454         GrContextForegroundSet(psContext, ClrYellow);
455         GrStringDraw(psContext, "INIT", -1, 180, 74, 0);
456     }
457     else if(SYSTEM_STATUS_TEXT == 1)
458     {
459         GrContextForegroundSet(psContext, ClrWhite);
460         GrStringDraw(psContext, "RUN", -1, 180, 74, 0);
461     }
462     else if(SYSTEM_STATUS_TEXT == 2)
463     {
464         GrContextForegroundSet(psContext, ClrRed);
465         GrStringDraw(psContext, "ERROR", -1, 180, 74, 0);
466     }
467     else if(SYSTEM_STATUS_TEXT == 3)
468     {
469         GrContextForegroundSet(psContext, ClrYellow);
470         GrStringDraw(psContext, "ERROR", -1, 180, 74, 0);
471     }
472     // 0 = INIT
473     // 1 = RUN
474     // 2 = ERROR
475     // 3 = SOFT-STOP)
476 
477     GrContextForegroundSet(psContext, ClrWhite);
478     GrStringDraw(psContext, DISPLAY_ERROR_LEVEL, -1, 180, 92, 0);
479     GrStringDraw(psContext, DISPLAY_RUNTIME,     -1, 180, 110, 0);
480 
481     if(ERROR_LEVEL == 0)
482     {
483         GrContextForegroundSet(psContext, ClrWhite);
```

```
484             GrStringDraw(psContext, "keine Fehler - System laeuft einwandfrei",     -1, 20, 150, 0);
485         }
486         else if(ERROR_LEVEL == 1)
487         {
488             GrContextForegroundSet(psContext, ClrYellow);
489             GrStringDraw(psContext, "Warnung aktiv - System laeuft",     -1, 20, 150, 0);
490         }
491         else if(ERROR_LEVEL == 2)
492         {
493             GrContextForegroundSet(psContext, ClrRed);
494             GrStringDraw(psContext, "Fehler - System gestoppt",     -1, 20, 150, 0);
495         }
496         else if(ERROR_LEVEL == 3)
497         {
498             GrContextForegroundSet(psContext, ClrRed);
499             GrStringDraw(psContext, "Kommunikationsfehler (Zykliersystem überprüfen) - System gestoppt",
                    -1, 20, 150, 0);
500         }
501
502     }
503
504     //*****************************************************************************
505     //
506     // Handles paint requests for the primitives canvas widget.
507     //
508     //*****************************************************************************
509     void
510     On_2_Lastkreis(tWidget *psWidget, tContext *psContext)
511     {
512         // Variablen in Strings schreiben:
513
514         // Lüfter 1 ------------------------------------------------------------
515         char FAN_1_DUTY[5];
516         sprintf(FAN_1_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[0]);
517
518         char FAN_1_RPM[5];
519         sprintf(FAN_1_RPM, "%d", FAN_RPM[0]);
520
521         char FAN_1_TEMP[5];
522         sprintf(FAN_1_TEMP, "%d", FAN_TEMPERATURE[0]);
523
524         char FAN_1_TEMP_WARN[5];
525         sprintf(FAN_1_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[0]);
526
527         char FAN_1_TEMP_ERROR[5];
528         sprintf(FAN_1_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[0]);
529
530         // Lüfter 2 ------------------------------------------------------------
531         char FAN_2_DUTY[5];
532         sprintf(FAN_2_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[1]);
533
534         char FAN_2_RPM[5];
535         sprintf(FAN_2_RPM, "%d", FAN_RPM[1]);
536
537         char FAN_2_TEMP[5];
538         sprintf(FAN_2_TEMP, "%d", FAN_TEMPERATURE[1]);
539
540         char FAN_2_TEMP_WARN[5];
541         sprintf(FAN_2_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[1]);
542
```

```
543     char FAN_2_TEMP_ERROR[5];
544     sprintf(FAN_2_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[1]);
545
546     // Lüfter 3 ------------------------------------------------------------
547     char FAN_3_DUTY[5];
548     sprintf(FAN_3_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[2]);
549
550     char FAN_3_RPM[5];
551     sprintf(FAN_3_RPM, "%d", FAN_RPM[2]);
552
553     char FAN_3_TEMP[5];
554     sprintf(FAN_3_TEMP, "%d", FAN_TEMPERATURE[2]);
555
556     char FAN_3_TEMP_WARN[5];
557     sprintf(FAN_3_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[2]);
558
559     char FAN_3_TEMP_ERROR[5];
560     sprintf(FAN_3_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[2]);
561
562     // Hallsensor 1 --------------------------------------------------------
563     char HALL_1_VALUE[5];
564     sprintf(HALL_1_VALUE, "%d", Hallsensor_Ampere[0]);
565
566     char HALL_1_WARN[5];
567     sprintf(HALL_1_WARN, "%d", LIMIT_HALL_CURRENT_WARNING[0]);
568
569     char HALL_1_ERROR[5];
570     sprintf(HALL_1_ERROR, "%d", LIMIT_HALL_CURRENT_ERROR[0]);
571
572
573     // Text
574     GrContextFontSet(psContext, &g_sFontCm18);
575     GrContextForegroundSet(psContext, ClrGold);
576     GrStringDraw(psContext, "Luefter 1:     T[%]:          U/m:", -1, 10, 30, 0);
577     GrStringDraw(psContext, "T[C]:          Tw[C]:        Te[C]:", -1, 10, 50, 0);
578     GrContextForegroundSet(psContext, ClrSeaGreen);
579     GrStringDraw(psContext, "Luefter 2:     T[%]:          U/m:", -1, 10, 70, 0);
580     GrStringDraw(psContext, "T[C]:          Tw[C]:        Te[C]:", -1, 10, 90, 0);
581     GrContextForegroundSet(psContext, ClrSlateBlue);
582     GrStringDraw(psContext, "Luefter 3:     T[%]:          U/m::", -1, 10, 110, 0);
583     GrStringDraw(psContext, "T[C]:          Tw[C]:        Te[C]:", -1, 10, 130, 0);
584     GrContextForegroundSet(psContext, ClrKhaki);
585     GrStringDraw(psContext, "Strom         Spannung", -1, 60, 150, 0);
586     GrStringDraw(psContext, "I[A]:      Iw[]:      Ie[]:      U[V]:", -1, 10, 170, 0);
587
588     // Variablen1
589     GrContextFontSet(psContext, &g_sFontCm18);
590     GrContextForegroundSet(psContext, ClrWhite);
591     GrStringDraw(psContext, FAN_1_DUTY, -1, 170, 30, 0);
592     GrStringDraw(psContext, FAN_1_RPM, -1, 270, 30, 0);
593     // Farbe anpassen ANFANG
594     if(FAN_TEMPERATURE[0] >= LIMIT_FAN_TEMP_WARNING[0] && FAN_TEMPERATURE[0] < LIMIT_FAN_TEMP_ERROR
            [0]){ GrContextForegroundSet(psContext, ClrYellow); }
595     else if(FAN_TEMPERATURE[0] >= LIMIT_FAN_TEMP_ERROR[0]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
596     // Farbe anpassen ENDE
597     GrStringDraw(psContext, FAN_1_TEMP, -1, 60, 50, 0);
598     GrContextForegroundSet(psContext, ClrSlateGray);
599     GrStringDraw(psContext, FAN_1_TEMP_WARN, -1, 170, 50, 0);
600     GrStringDraw(psContext, FAN_1_TEMP_ERROR, -1, 270, 50, 0);
```

```
601     GrContextForegroundSet(psContext, ClrWhite);
602     GrStringDraw(psContext, FAN_2_DUTY, -1, 170, 70, 0);
603     GrStringDraw(psContext, FAN_2_RPM, -1, 270, 70, 0);
604     // Farbe anpassen ANFANG
605     if(FAN_TEMPERATURE[1] >= LIMIT_FAN_TEMP_WARNING[1] && FAN_TEMPERATURE[1] < LIMIT_FAN_TEMP_ERROR
            [1]){ GrContextForegroundSet(psContext, ClrYellow); }
606     else if(FAN_TEMPERATURE[1] >= LIMIT_FAN_TEMP_ERROR[1]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
607     // Farbe anpassen ENDE
608     GrStringDraw(psContext, FAN_2_TEMP, -1, 60, 90, 0);
609     GrContextForegroundSet(psContext, ClrSlateGray);
610     GrStringDraw(psContext, FAN_2_TEMP_WARN, -1, 170, 90, 0);
611     GrStringDraw(psContext, FAN_2_TEMP_ERROR, -1, 270, 90, 0);
612     GrContextForegroundSet(psContext, ClrWhite);
613     GrStringDraw(psContext, FAN_3_DUTY, -1, 170, 110, 0);
614     GrStringDraw(psContext, FAN_3_RPM, -1, 270, 110, 0);
615     // Farbe anpassen ANFANG
616     if(FAN_TEMPERATURE[2] >= LIMIT_FAN_TEMP_WARNING[2] && FAN_TEMPERATURE[2] < LIMIT_FAN_TEMP_ERROR
            [2]){ GrContextForegroundSet(psContext, ClrYellow); }
617     else if(FAN_TEMPERATURE[2] >= LIMIT_FAN_TEMP_ERROR[2]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
618     // Farbe anpassen ENDE
619     GrStringDraw(psContext, FAN_3_TEMP, -1, 60, 130, 0);
620     GrContextForegroundSet(psContext, ClrSlateGray);
621     GrStringDraw(psContext, FAN_3_TEMP_WARN, -1, 170, 130, 0);
622     GrStringDraw(psContext, FAN_3_TEMP_ERROR, -1, 270, 130, 0);
623     GrContextForegroundSet(psContext, ClrWhite);
624     // Farbe anpassen ANFANG
625     if(Hallsensor_Ampere[0] >= LIMIT_HALL_CURRENT_WARNING[0] && Hallsensor_Ampere[0] <
            LIMIT_HALL_CURRENT_ERROR[0]){ GrContextForegroundSet(psContext, ClrYellow); }
626     else if(Hallsensor_Ampere[0] >= LIMIT_HALL_CURRENT_ERROR[0]){ GrContextForegroundSet(psContext,
            ClrRed); }
627     // Farbe anpassen ENDE
628     GrStringDraw(psContext, HALL_1_VALUE, -1, 50, 170, 0);
629     GrContextForegroundSet(psContext, ClrSlateGray);
630     GrStringDraw(psContext, HALL_1_WARN, -1, 120, 170, 0);
631     GrStringDraw(psContext, HALL_1_ERROR, -1, 180, 170, 0);
632     // Spannungswert
633     GrContextForegroundSet(psContext, ClrWhite);
634     GrStringDraw(psContext, "0", -1, 270, 170, 0); // VORERST (Spannung folgt)
635
636 }
637
638 void
639 On_3_Ladekreis(tWidget *psWidget, tContext *psContext)
640 {
641     // Variablen in Strings schreiben:
642
643     // Lüfter 4 ------------------------------------------------------------
644     char FAN_4_DUTY[5];
645     sprintf(FAN_4_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[3]);
646
647     char FAN_4_RPM[5];
648     sprintf(FAN_4_RPM, "%d", FAN_RPM[3]);
649
650     char FAN_4_TEMP[5];
651     sprintf(FAN_4_TEMP, "%d", FAN_TEMPERATURE[3]);
652
653     char FAN_4_TEMP_WARN[5];
654     sprintf(FAN_4_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[3]);
```

```
655
656         char FAN_4_TEMP_ERROR[5];
657         sprintf(FAN_4_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[3]);
658
659         // Lüfter 5 ------------------------------------------------------------
660         char FAN_5_DUTY[5];
661         sprintf(FAN_5_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[4]);
662
663         char FAN_5_RPM[5];
664         sprintf(FAN_5_RPM, "%d", FAN_RPM[4]);
665
666         char FAN_5_TEMP[5];
667         sprintf(FAN_5_TEMP, "%d", FAN_TEMPERATURE[4]);
668
669         char FAN_5_TEMP_WARN[5];
670         sprintf(FAN_5_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[4]);
671
672         char FAN_5_TEMP_ERROR[5];
673         sprintf(FAN_5_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[4]);
674
675         // Lüfter 6 ------------------------------------------------------------
676         char FAN_6_DUTY[5];
677         sprintf(FAN_6_DUTY, "%d", FAN_DUTYCYCLE_PERCENT[5]);
678
679         char FAN_6_RPM[5];
680         sprintf(FAN_6_RPM, "%d", FAN_RPM[5]);
681
682         char FAN_6_TEMP[5];
683         sprintf(FAN_6_TEMP, "%d", FAN_TEMPERATURE[5]);
684
685         char FAN_6_TEMP_WARN[5];
686         sprintf(FAN_6_TEMP_WARN, "%d", LIMIT_FAN_TEMP_WARNING[5]);
687
688         char FAN_6_TEMP_ERROR[5];
689         sprintf(FAN_6_TEMP_ERROR, "%d", LIMIT_FAN_TEMP_ERROR[5]);
690
691         // Hallsensor 1 --------------------------------------------------------
692         char HALL_2_VALUE[5];
693         sprintf(HALL_2_VALUE, "%d", Hallsensor_Ampere[1]);
694
695         char HALL_2_WARN[5];
696         sprintf(HALL_2_WARN, "%d", LIMIT_HALL_CURRENT_WARNING[1]);
697
698         char HALL_2_ERROR[5];
699         sprintf(HALL_2_ERROR, "%d", LIMIT_HALL_CURRENT_ERROR[1]);
700
701
702         // Text
703         GrContextFontSet(psContext, &g_sFontCm18);
704         GrContextForegroundSet(psContext, ClrGold);
705         GrStringDraw(psContext, "Luefter 4:      T[%]:          U/m:", -1, 10, 30, 0);
706         GrStringDraw(psContext, "T[C]:           Tw[C]:        Te[C]:", -1, 10, 50, 0);
707         GrContextForegroundSet(psContext, ClrSeaGreen);
708         GrStringDraw(psContext, "Luefter 5:      T[%]:          U/m:", -1, 10, 70, 0);
709         GrStringDraw(psContext, "T[C]:           Tw[C]:        Te[C]:", -1, 10, 90, 0);
710         GrContextForegroundSet(psContext, ClrSlateBlue);
711         GrStringDraw(psContext, "Luefter 6:      T[%]:          U/m::", -1, 10, 110, 0);
712         GrStringDraw(psContext, "T[C]:           Tw[C]:        Te[C]:", -1, 10, 130, 0);
713         GrContextForegroundSet(psContext, ClrKhaki);
714         GrStringDraw(psContext, "Strom         Spannung", -1, 60, 150, 0);
```

```
715     GrStringDraw(psContext, "I[A]:      Iw[]:      Ie[]:      U[V]:", -1, 10, 170, 0);
716
717     // Variablen1
718     GrContextFontSet(psContext, &g_sFontCm18);
719     GrContextForegroundSet(psContext, ClrWhite);
720     GrStringDraw(psContext, FAN_4_DUTY, -1, 170, 30, 0);
721     GrStringDraw(psContext, FAN_4_RPM, -1, 270, 30, 0);
722     // Farbe anpassen ANFANG
723     if(FAN_TEMPERATURE[3] >= LIMIT_FAN_TEMP_WARNING[3] && FAN_TEMPERATURE[3] < LIMIT_FAN_TEMP_ERROR
            [3]){ GrContextForegroundSet(psContext, ClrYellow); }
724     else if(FAN_TEMPERATURE[3] >= LIMIT_FAN_TEMP_ERROR[3]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
725     // Farbe anpassen ENDE
726     GrStringDraw(psContext, FAN_4_TEMP, -1, 60, 50, 0);
727     GrContextForegroundSet(psContext, ClrSlateGray);
728     GrStringDraw(psContext, FAN_4_TEMP_WARN, -1, 170, 50, 0);
729     GrStringDraw(psContext, FAN_4_TEMP_ERROR, -1, 270, 50, 0);
730     GrContextForegroundSet(psContext, ClrWhite);
731     GrStringDraw(psContext, FAN_5_DUTY, -1, 170, 70, 0);
732     GrStringDraw(psContext, FAN_5_RPM, -1, 270, 70, 0);
733     // Farbe anpassen ANFANG
734     if(FAN_TEMPERATURE[4] >= LIMIT_FAN_TEMP_WARNING[4] && FAN_TEMPERATURE[4] < LIMIT_FAN_TEMP_ERROR
            [4]){ GrContextForegroundSet(psContext, ClrYellow); }
735     else if(FAN_TEMPERATURE[4] >= LIMIT_FAN_TEMP_ERROR[4]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
736     // Farbe anpassen ENDE
737     GrStringDraw(psContext, FAN_5_TEMP, -1, 60, 90, 0);
738     GrContextForegroundSet(psContext, ClrSlateGray);
739     GrStringDraw(psContext, FAN_5_TEMP_WARN, -1, 170, 90, 0);
740     GrStringDraw(psContext, FAN_5_TEMP_ERROR, -1, 270, 90, 0);
741     GrContextForegroundSet(psContext, ClrWhite);
742     GrStringDraw(psContext, FAN_6_DUTY, -1, 170, 110, 0);
743     GrStringDraw(psContext, FAN_6_RPM, -1, 270, 110, 0);
744     // Farbe anpassen ANFANG
745     if(FAN_TEMPERATURE[5] >= LIMIT_FAN_TEMP_WARNING[5] && FAN_TEMPERATURE[5] < LIMIT_FAN_TEMP_ERROR
            [5]){ GrContextForegroundSet(psContext, ClrYellow); }
746     else if(FAN_TEMPERATURE[5] >= LIMIT_FAN_TEMP_ERROR[5]){ GrContextForegroundSet(psContext, ClrRed)
            ; }
747     // Farbe anpassen ENDE
748     GrStringDraw(psContext, FAN_6_TEMP, -1, 60, 130, 0);
749     GrContextForegroundSet(psContext, ClrSlateGray);
750     GrStringDraw(psContext, FAN_6_TEMP_WARN, -1, 170, 130, 0);
751     GrStringDraw(psContext, FAN_6_TEMP_ERROR, -1, 270, 130, 0);
752     GrContextForegroundSet(psContext, ClrWhite);
753     // Farbe anpassen ANFANG
754     if(Hallsensor_Ampere[1] >= LIMIT_HALL_CURRENT_WARNING[1] && Hallsensor_Ampere[1] <
            LIMIT_HALL_CURRENT_ERROR[1]){ GrContextForegroundSet(psContext, ClrYellow); }
755     else if(Hallsensor_Ampere[1] >= LIMIT_HALL_CURRENT_ERROR[1]){ GrContextForegroundSet(psContext,
            ClrRed); }
756     // Farbe anpassen ENDE
757     GrStringDraw(psContext, HALL_2_VALUE, -1, 50, 170, 0);
758     GrContextForegroundSet(psContext, ClrSlateGray);
759     GrStringDraw(psContext, HALL_2_WARN, -1, 120, 170, 0);
760     GrStringDraw(psContext, HALL_2_ERROR, -1, 180, 170, 0);
761     // Spannungswert
762     GrContextForegroundSet(psContext, ClrWhite);
763     GrStringDraw(psContext, "0", -1, 270, 170, 0); // VORERST (Spannung folgt)
764
765 }
766
```

```
767 void
768 On_4_Details(tWidget *psWidget, tContext *psContext)
769 {
770
771 }
772
773 // STATE MACHINE Variablen
774 typedef enum{
775     STATE_INIT,
776     STATE_SOFT_STOP,
777     STATE_RUN,
778     STATE_ERROR
779 }PROGRAMM_STATE;
780
781 // STATE MACHINE Startwert
782 static volatile PROGRAMM_STATE STATE = STATE_INIT;
783
784
785 int main(void)
786 {
787     ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
          SYSCTL_CFG_VCO_480), 120000000);
788     // Systemclock auf 120 MHz stellen
789     // den Wert des Systemtaktes außerdem in die Variable "ui32SysClkFreq" speichern. (vereinfacht
          spätere Berechnung von Werten)
790
791     UART_Init();
792     // UART Einstellungen
793
794     PWM_init();
795     // PWM einstellungen
796
797     GPIO_Int_Init();
798     // GPIO Interrupt Einstellungen
799
800     GPIO_Ports_Init();
801     // GPIO Ports Einstellungen
802
803     Timer_Init();
804     // Timer für die PWM Auswertung -> Einstellungen
805
806     one_wire_UART7_Init();
807     // UART3 für die 1-wire Ein- & Ausgabe Einstellen
808
809     ADC_Init();
810     // ADCs für die HALL Sensoren --> Einstellungen
811
812     SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
813     EEPROMInit();
814     // EEPROMMassErase(); // würde den gesamten ROM auf 0x FFFF FFFF zurücksetzten
815     // Aktiviert den EEPROM
816
817     IntMasterEnable();
818     // dies aktiviert die MASTER-Interrupt-API für alle Interrupts
819
820     //NUR ZUM TESTEN
821     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_4);
822     // setzt GPIO Port_F Pin4 als Output Pin
823     // an diesem Pin ist der LED1 angeschlossen (Launchpad)
824
```

```
825     // ROM-CODES Init.
826     one_wire_Devices();
827
828     tContext sContext;
829     tRectangle sRect;
830
831     //
832     // The FPU should be enabled because some compilers will use floating-
833     // point registers, even for non-floating-point code.  If the FPU is not
834     // enabled this will cause a fault.  This also ensures that floating-
835     // point operations could be added to this application and would work
836     // correctly and use the hardware floating-point unit.  Finally, lazy
837     // stacking is enabled for interrupt handlers.  This allows floating-
838     // point instructions to be used within interrupt handlers, but at the
839     // expense of extra stack usage.
840     //
841 //    FPUEnable();
842 //    FPULazyStackingEnable();
843
844     //
845     // Initialize the display driver.
846     //
847     Kentec320x240x16_SSD2119Init(ui32SysClkFreq);
848
849     //
850     // Initialize the graphics context.
851     //
852     GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
853
854     //
855     // Fill the top 24 rows of the screen with blue to create the banner.
856     //
857     sRect.i16XMin = 0;
858     sRect.i16YMin = 0;
859     sRect.i16XMax = GrContextDpyWidthGet(&sContext) - 1;
860     sRect.i16YMax = 23;
861     GrContextForegroundSet(&sContext, ClrDarkBlue);
862     GrRectFill(&sContext, &sRect);
863
864     //
865     // Put a white box around the banner.
866     //
867     GrContextForegroundSet(&sContext, ClrWhite);
868     GrRectDraw(&sContext, &sRect);
869
870     //
871     // Put the application name in the middle of the banner.
872     //
873     GrContextFontSet(&sContext, &g_sFontCm20);
874     GrStringDrawCentered(&sContext, "Schutzcontroller", -1, GrContextDpyWidthGet(&sContext) / 2, 8,
            0);
875
876     //
877     // Initialize the touch screen driver and have it route its messages to the
878     // widget tree.
879     //
880     TouchScreenInit(ui32SysClkFreq);
881     TouchScreenCallbackSet(WidgetPointerMessage);
882
883     //
```

```
884     // Add the title block and the previous and next buttons to the widget
885     // tree.
886     //
887     WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sPrevious);
888     WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sTitle);
889     WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sNext);
890
891     //
892     // Add the first panel to the widget tree.
893     //
894     g_ui32Panel = 0;
895     WidgetAdd(WIDGET_ROOT, (tWidget *)g_psPanels);
896     CanvasTextSet(&g_sTitle, g_pcPanei32Names[0]);
897
898     //
899     // Issue the initial paint request to the widgets.
900     //
901     WidgetPaint(WIDGET_ROOT);
902
903     //
904     // Loop forever handling widget messages.
905     //
906
907
908
909
910     Sekunden = 0;                   // 5 Sekunden Zähler
911
912     while(1)
913     {
914         // Auf die Touch-Display-Tastendrücke reagieren
915         WidgetMessageQueueProcess();
916
917         // Messen
918         if(one_wire_measure_active == 0)
919         {
920             one_wire_measure_active = 1;
921         }
922         one_wire_measure();
923
924         switch(STATE)
925         {
926
927             case STATE_INIT:
928
929                 SYSTEM_STATUS_TEXT = 0;
930
931                 SM_Init();          // Funktionsaufruf (Initialisierungen)
932
933                 STATE = STATE_RUN;  // Normalbetrieb starten
934
935             break;
936
937
938             case STATE_SOFT_STOP:
939
940                 SYSTEM_STATUS_TEXT = 3;
941
942                 SM_Soft_Stop();     //  in der Fu.: Relays öffnen, STOP weitergeben etc.
```

```
943                                 // der Status STATE_SOFT_STOP wird in der Interrupt Funktion
                                        gesetzt
944
945                 // falls der gelbe knopf gedrückt wurde, STATUS ändern um über STATE_INIT in
                        STATE_RUN zu kommen
946                 if(YELLOW_BUTTON==1)
947                 {
948                     STATE = STATE_INIT;
949                     YELLOW_BUTTON=0;
950                 }
951
952             break;
953
954
955             case STATE_RUN:
956
957                 SYSTEM_STATUS_TEXT = 1;
958
959                 SM_Run();           // Fu. Messungen verarbeiten, bei Fehler STATE wechseln!!!
960                                     // Warnings sind hier möglich (kein eigener STATE)
961                 if(ERROR_LEVEL == 2)
962                 {
963                     STATE = STATE_ERROR;
964                 } //ACHTUNG NUR ZUM TESTEN NICHT IN DEN ERROR MODE GEHEN !!!!!!
965
966                 // falls der rote knopf gedrückt wurde, STATUS ändern um in SOFT-STOP zu kommen
967                 if(RED_BUTTON==1)
968                 {
969                     STATE = STATE_SOFT_STOP;
970                     RED_BUTTON=0;
971                 }
972
973             break;
974
975
976             case STATE_ERROR:       // erst bei Errorlevel L#03 oder L#02
977
978                 SYSTEM_STATUS_TEXT = 2;
979
980                 SM_Error();         // um diesen Status zu verlassen muss in Fu. Bestätigt werden.
981                                     // dann in STATE_INIT wechseln
982                 if(SM_INIT == 1)
983                 {
984                     STATE = STATE_INIT;
985                     SM_INIT = 0;
986                 }
987
988                 // falls der rote knopf gedrückt wurde, STATUS ändern um in SOFT-STOP zu kommen
989                 if(RED_BUTTON==1)
990                 {
991                     STATE = STATE_SOFT_STOP;
992                     RED_BUTTON=0;
993                 }
994
995             break;
996
997             default:
998             break;
999         }
1000
```

```
1001            // Bildschirme durchschalten (da Display-Toucheingabe KAPUTT ist) :-(
1002            if(DISPLAY_ROTATER == 1)
1003            {
1004                DISPLAY_ROTATER = 0;
1005                OnRotateScreens();
1006            }
1007        }
1008    }
```

Listing A.30: PWM_init.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12
13  #include "globals.h"
14
15
16  void PWM_init(void)
17  {
18      FAN_PWM_FREQUENCY = 25000; // in Hz
19
20      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); // aktiviert GPIO Ports an PORT_F --> benötigt für
            PWM0
21      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG); // aktiviert GPIO Ports an PORT_G --> benötigt für
            PWM0
22      // benötigt für PWM0
23
24      SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
25      // aktiviert PWM0 Modul: diese PWMs sind über PORT_F & PORT_G & PORT_K & PORT_L verteilt
26      // das PWM0 Modul hat vier PWM Generatorblöcke, jeder Generatorblock kann 2 unabhängige Tastgrade
            der selben Frequenz erzeugen
27      // siehe "Pin Mux Utility" bei der "aktivierung" von PWM0
28      // genutzt soll hier werden: Port_F
29
30      PWMClockSet(PWM0_BASE,PWM_SYSCLK_DIV_64);
31      // stellt den Takt der PWM0 ein auf ein 64tel des Systemtaktes (hier: siehe oben: 120 MHz)
32      // 120/64 = 1,875 MHz = 1.875.000 Hz
33      // "/64" ist die "langsamste" Einstellung die es gibt.
34
35      GPIOPinConfigure(GPIO_PF0_M0PWM0); // PIN0 an PORT_F wird aktiviert (siehe Pinmux Utility PF0 IST
            : M0PWM0)
36      GPIOPinConfigure(GPIO_PF1_M0PWM1); // PIN1 an PORT_F wird aktiviert (siehe Pinmux Utility PF1 IST
            : M0PWM1)
37      GPIOPinConfigure(GPIO_PF2_M0PWM2); // PIN2 an PORT_F wird aktiviert (siehe Pinmux Utility PF2 IST
            : M0PWM2)
38      GPIOPinConfigure(GPIO_PF3_M0PWM3); // PIN3 an PORT_F wird aktiviert (siehe Pinmux Utility PF3 IST
            : M0PWM3)
39      GPIOPinConfigure(GPIO_PG0_M0PWM4); // PIN0 an PORT_G wird aktiviert (siehe Pinmux Utility PG0 IST
            : M0PWM4)
40      GPIOPinConfigure(GPIO_PG1_M0PWM5); // PIN1 an PORT_G wird aktiviert (siehe Pinmux Utility PG1 IST
            : M0PWM5)
```

```
41
42      GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3); // Port_G ->
            PIN_0 bis PIN_3 als GPIO-->PWM Ausgang
43      GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_0 | GPIO_PIN_1); // Port_G -> PIN_0 & PIN_1 als GPIO-->
            PWM Ausgang
44
45      // Dies dient zur Berechnung von Takt und PWM-lade-Werten:
46      ui32PWMClock = ui32SysClkFreq / 64;
47      // In "ui32PWMClock" wird der Wert des PWM-Taktes gespeichert (siehe oben: zuvor eingestellt auf
            120MHz/64)
48      // In der Variable "ui32SysClkFreq" wurde ganz am Anfang der Systemtakt eingestellt.
49      // 120/64 = 1,875 MHz = 1.875.000 Hz
50      ui32Load = (ui32PWMClock / FAN_PWM_FREQUENCY) - 1;
51      // die Variable "ui32Load" enthält die Nummer der PWM Takt Zyklen pro gewählte Ausgabeperiode
52      // Der Variable "ui32Load" wird der Wert (1.875.000 Hz / 25.000 Hz) = 75 Hz zugewiesen.
53      // "PWM_FREQUENCY" wurde am Anfang selbst definiert.(25.000) Hz
54      // "-1", da die Variable bei 0 zu zählen beginnt (75 - 1 = 74)
55
56      //PWM Konfiguration Generator 0:
57      PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN); // stellt PWM0 Generator.0 in den "
            count-down-mode"
58      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, ui32Load); // stellt die Periode von PWM0 Generator.0 auf
            den zuvor berechneten Wert (18.749)
59      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, ui32Load/2); // stellt die Pulsweite am Output PF0_M0PWM0
            auf Tastgrad
60      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, ui32Load/2); // stellt die Pulsweite am Output PF1_M0PWM1
            auf 1/2 Tastgrad (vorerst)
61      PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true); // wählt den gewünschten Output (hier: PF0_M0PWM0
            )
62      PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true); // wählt den gewünschten Output (hier: PF1_M0PWM1
            )
63      PWMGenEnable(PWM0_BASE, PWM_GEN_0); // startet den PWM-Generator.0
64
65      //PWM Konfiguration Generator 1:
66      PWMGenConfigure(PWM0_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN); // stellt PWM0 Generator.1 in den "
            count-down-mode"
67      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, ui32Load); // stellt die Periode von PWM0 Generator.0 auf
            den zuvor berechneten Wert (18.749)
68      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, ui32Load/2); // stellt die Pulsweite am Output PF2_M0PWM2
            auf 1/2 Tastgrad (vorerst)
69      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, ui32Load/2); // stellt die Pulsweite am Output PF3_M0PWM3
            auf 1/2 Tastgrad (vorerst)
70      PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT, true); // wählt den gewünschten Output (hier: PF2_M0PWM2
            )
71      PWMOutputState(PWM0_BASE, PWM_OUT_3_BIT, true); // wählt den gewünschten Output (hier: PF3_M0PWM3
            )
72      PWMGenEnable(PWM0_BASE, PWM_GEN_1); // startet den PWM-Generator.1
73
74      //PWM Konfiguration Generator 2:
75      PWMGenConfigure(PWM0_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN); // stellt PWM0 Generator.2 in den "
            count-down-mode"
76      PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, ui32Load); // stellt die Periode von PWM0 Generator.2 auf
            den zuvor berechneten Wert (18.749)
77      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ui32Load/2); // stellt die Pulsweite am Output PG0_M0PWM4
            auf 1/2 Tastgrad (vorerst)
78      PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, ui32Load/2); // stellt die Pulsweite am Output PG1_M0PWM5
            auf 1/2 Tastgrad (vorerst)
79      PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT, true); // wählt den gewünschten Output (hier: PG0_M0PWM4
            )
```

```
80      PWMOutputState(PWM0_BASE, PWM_OUT_5_BIT, true); // wählt den gewünschten Output (hier: PG1_M0PWM5
            )
81      PWMGenEnable(PWM0_BASE, PWM_GEN_2); // startet den PWM-Generator.2
82  }
```

Listing A.31: PWM_init.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_PWM_INIT_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_PWM_INIT_H_
3
4   /*****************************************************************************
5    *
6    * Function Declarations
7    *
8    *****************************************************************************/
9
10  void PWM_init(void);
11
12  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_PWM_INIT_H_ */
```

Listing A.32: SM_Functions.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12  #include "driverlib/uart.h"
13  #include "utils/uartstdio.h"
14  #include "driverlib/eeprom.h"
15
16  // eigene Includes
17  #include "globals.h"
18  #include "Startvalues.h"
19
20  // schaltet RED-LED AN
21  void red_led_on(void)
22  {
23      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_0, 0b00000001);
24  }
25
26  // schaltet RED-LED AUS
27  void red_led_off(void)
28  {
29      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_0, 0b00000000);
30  }
31
32  // schaltet GREEN-LED AN
33  void green_led_on(void)
34  {
35      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0b00000010);
36  }
37
```

```
38  // schaltet GREEN-LED AUS
39  void green_led_off(void)
40  {
41      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_1, 0b00000000);
42  }
43
44  // schaltet Relay 1 AN
45  void relay_1_on(void)
46  {
47      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_2, 0b00000000);
48  }
49
50  // schaltet Relay 1 AUS
51  void relay_1_off(void)
52  {
53      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_2, 0b00000100);
54  }
55
56  // schaltet Relay 2 AN
57  void relay_2_on(void)
58  {
59      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_3, 0b00000000);
60  }
61
62  // schaltet Relay 2 AUS
63  void relay_2_off(void)
64  {
65      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_3, 0b00001000);
66  }
67
68  // toggelt AKUSTISCHES-SIGNAL
69  void buzzer_toggle(void)
70  {
71      if(TOGGLE_500ms == 1)
72      {
73          GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_4, 0b00010000); //AN
74      }
75      else
76      {
77          GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_4, 0b00000000); //AUS
78      }
79  }
80
81  // schaltet AKUSTISCHES-SIGNAL AUS
82  void buzzer_off(void)
83  {
84      //AUS
85      GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_4, 0b00000000);
86  }
87
88  // schaltet SCHUTZCONTROLLER-BEREIT-SIGNAL AN
89  void controller_ready_on(void)
90  {
91      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_5, 0b00100000);
92  }
93
94  // schaltet SCHUTZCONTROLLER-BEREIT-SIGNAL AUS
95  void controller_ready_off(void)
96  {
97      GPIOPinWrite(GPIO_PORTH_BASE, GPIO_PIN_5, 0b00000000);
```

```
98  }
99
100 // schreibt die zu speichernden Werte in das ROM_VARIABLES-Array und schreibt dieses dann wiederum in
        das EEPROM
101 void rom_variables_write(void)
102 {
103     // Startwerte aus dem ROM
104     ROM_VARIABLES[0] = RELAY_COUNTER[0];
105     ROM_VARIABLES[1] = RELAY_COUNTER[1];
106
107     // ROM_VARIABLES Array in EEPROM schreiben
108     EEPROMProgram(ROM_VARIABLES, 0x0, sizeof(ROM_VARIABLES));
109
110 }
111
112 // liest das EEPROM aus und schreibt die zu speichernden Werte in das ROM_VARIABLES-Array
113 // dann werden die Wertte aus dem ROM_VARIABLES-Array in die gewünschten Werte geschrieben
114 void rom_variables_read(void)
115 {
116     // EEPROM Auslesen und in ROM_VARIABLES Array speichern
117     EEPROMRead(ROM_VARIABLES, 0x0, sizeof(ROM_VARIABLES));
118
119     // Startwerte aus dem ROM
120     RELAY_COUNTER[0] = ROM_VARIABLES[0];
121     RELAY_COUNTER[1] = ROM_VARIABLES[1];
122 }
123
124 void UART_send_warning(uint32_t UART_L, uint32_t UART_W, uint32_t UART_V)
125 {
126     // ERRORLEVEL
127     UARTprintf("L#%d", UART_L);
128
129     // Warning
130     UARTprintf("W#%2d", UART_W);
131
132     // Value
133     UARTprintf("V#%6d", UART_V);
134
135     // neue Zeile
136     UARTprintf("\r\n");
137 }
138
139 void UART_send_error(uint32_t UART_L, uint32_t UART_E, uint32_t UART_V)
140 {
141     // ERRORLEVEL
142     UARTprintf("L#%d", UART_L);
143
144     // Warning
145     UARTprintf("E#%2d", UART_E);
146
147     // Value
148     UARTprintf("V#%6d", UART_V);
149
150     // neue Zeile
151     UARTprintf("\r\n");
152 }
153
154 void control_fan_RPM(void)
155 {
156     int fan_number;
```

```
157     int pwm_value;
158
159     for(fan_number=0;fan_number<6;fan_number++)
160     {
161         // Temperaturen lesen und Werte
162         if(FAN_TEMPERATURE[fan_number] <= 2500  )                                    // bei 25
                ° und kälter
163         {
164             FAN_DUTYCYCLE_PERCENT[fan_number] = 10;
165         }
166         else if(FAN_TEMPERATURE[fan_number] > 2500 && FAN_TEMPERATURE[fan_number] <= 2750)  // bei
                +25° bis 27,5°
167         {
168             FAN_DUTYCYCLE_PERCENT[fan_number] = 15;
169         }
170         else if(FAN_TEMPERATURE[fan_number] > 2750 && FAN_TEMPERATURE[fan_number] <= 3000)  // bei
                +27,5° bis 30°
171         {
172             FAN_DUTYCYCLE_PERCENT[fan_number] = 20;
173         }
174         else if(FAN_TEMPERATURE[fan_number] > 3000 && FAN_TEMPERATURE[fan_number] <= 3250)  // bei
                +30° bis 32,5°
175         {
176             FAN_DUTYCYCLE_PERCENT[fan_number] = 25;
177         }
178         else if(FAN_TEMPERATURE[fan_number] > 3250 && FAN_TEMPERATURE[fan_number] <= 3500)  // bei
                +32,5° bis 35°
179         {
180             FAN_DUTYCYCLE_PERCENT[fan_number] = 30;
181         }
182         else if(FAN_TEMPERATURE[fan_number] > 3500 && FAN_TEMPERATURE[fan_number] <= 3750)  // bei
                +35° bis 37,5°
183         {
184             FAN_DUTYCYCLE_PERCENT[fan_number] = 40;
185         }
186         else if(FAN_TEMPERATURE[fan_number] > 3750 && FAN_TEMPERATURE[fan_number] <= 4000)  // bei
                +37,5° bis 40°
187         {
188             FAN_DUTYCYCLE_PERCENT[fan_number] = 50;
189         }
190         else if(FAN_TEMPERATURE[fan_number] > 4000 && FAN_TEMPERATURE[fan_number] <= 4250)  // bei
                +40° bis 42,5°
191         {
192             FAN_DUTYCYCLE_PERCENT[fan_number] = 60;
193         }
194         else if(FAN_TEMPERATURE[fan_number] > 4250 && FAN_TEMPERATURE[fan_number] <= 4500)  // bei
                +42,5° bis 45°
195         {
196             FAN_DUTYCYCLE_PERCENT[fan_number] = 70;
197         }
198         else if(FAN_TEMPERATURE[fan_number] > 4500 && FAN_TEMPERATURE[fan_number] <= 4750)  // bei
                +45° bis 47,5°
199         {
200             FAN_DUTYCYCLE_PERCENT[fan_number] = 80;
201         }
202         else if(FAN_TEMPERATURE[fan_number] > 4750 && FAN_TEMPERATURE[fan_number] <= 5000)  // bei
                +47,5° bis 50°
203         {
204             FAN_DUTYCYCLE_PERCENT[fan_number] = 90;
205         }
```

```
206         else if(FAN_TEMPERATURE[fan_number] > 5000)                                    // bei
                über 50°
207         {
208             FAN_DUTYCYCLE_PERCENT[fan_number] = 100;
209         }
210
211
212
213         // zu setztenden Wert berechnen
214         pwm_value = ( ( (1000*ui32Load)/100)* FAN_DUTYCYCLE_PERCENT[fan_number] ) / 1000;
215
216
217         // neuen Wert in betreffendem PWM Ausgang setzen
218         if(fan_number == 0)
219         {
220             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, pwm_value); // stellt die Pulsweite am Output
                    PF0_M0PWM0 auf neuen Wert
221         }
222         else if(fan_number == 1)
223         {
224             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, pwm_value); // stellt die Pulsweite am Output
                    PF1_M0PWM1 auf neuen Wert
225         }
226         else if(fan_number == 2)
227         {
228             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, pwm_value); // stellt die Pulsweite am Output
                    PF2_M0PWM2 auf neuen Wert
229         }
230         else if(fan_number == 3)
231         {
232             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, pwm_value); // stellt die Pulsweite am Output
                    PF3_M0PWM3 auf neuen Wert
233         }
234         else if(fan_number == 4)
235         {
236             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, pwm_value); // stellt die Pulsweite am Output
                    PG0_M0PWM4 auf neuen Wert
237         }
238         else if(fan_number == 5)
239         {
240             PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, pwm_value); // stellt die Pulsweite am Output
                    PG1_M0PWM5 auf neuen Wert
241         }
242
243     }
244
245 }
246
247 void power_supply_start(void)
248 {
249     // Status des Netzteils lesen (GREEN-CABLE)
250     POWER_SUPPLY_ACTIVE = GPIOPinRead(GPIO_PORTL_BASE, GPIO_PIN_4);
251
252     if( (POWER_SUPPLY_ACTIVE & GPIO_PIN_4) != 0 ) // wenn es AUS ist, AN schalten
253     {
254         // setze YELLOW-CABLE = 1 (PORT_M,PIN_5), wenn POWER_SUPPLY_ACTIVE == 0 (PORT_L,PIN_4)
255         GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_5, 0b00100000);
256         POWER_SUPPLY_ON_OFF = 1;
257     }
258     //else
```

```
259     // ist bereits AN, nichts ändern
260 }
261
262 void power_supply_stop(void)
263 {
264     // Status des Netzteils lesen (GREEN-CABLE)
265     POWER_SUPPLY_ACTIVE = GPIOPinRead(GPIO_PORTL_BASE, GPIO_PIN_4);
266
267     if( (POWER_SUPPLY_ACTIVE & GPIO_PIN_4) == 0 ) // wenn es AN ist, AUS schalten
268     {
269         // setze YELLOW-CABLE = = (PORT_M,PIN_5), wenn POWER_SUPPLY_ACTIVE != 0 (PORT_L,PIN_4)
270         GPIOPinWrite(GPIO_PORTM_BASE, GPIO_PIN_5, 0b00000000);
271         POWER_SUPPLY_ON_OFF = 0;
272     }
273     //else
274     // ist bereits AUS, nichts ändern
275 }
276
277
278 //_____________________________________________________________________
279 //                    STATE MACHINE Functions:
280 //_____________________________________________________________________
281
282
283 void SM_Init(void)
284 {
285     // Startwerte den Variablen zuweisen
286     Startvalues();
287
288     // Variablen aus dem ROM lesen
289     rom_variables_read();
290
291     // schaltet SCHUTZCONTROLLER-BEREIT-SIGNAL AN
292     controller_ready_on();
293
294     // (Grün) LED Betrieb AN
295     green_led_on();
296
297     // (Rot) LED Fehler AUS
298     red_led_off();
299
300 }
301
302 void SM_Soft_Stop(void)
303 {
304     // für den TEST Power Supply anhalten
305     power_supply_stop();
306
307     // Relais 1 öffnen
308     relay_1_off();
309
310     // Relais 2 öffnen
311     relay_2_off();
312
313     // schaltet SCHUTZCONTROLLER-BEREIT-SIGNAL AUS
314     controller_ready_off();
315
316     // Drehzahl steuern
317     control_fan_RPM();
318
```

```
319     // nur beim ersten aufrufen der Soft_Stop_Funktion
320     if(SM_Soft_Stop_active==0)
321     {
322         SM_Soft_Stop_active=1;
323 
324         // für TEST geändert
325         // Variablen in das ROM schreiben
326         //rom_variables_write();
327     }
328 }
329 
330 void SM_Run(void)
331 {
332     // lokale Variablen
333     int TEMP_HIGH[6]  = {0,0,0,0,0,0};
334     int TEMP_ERROR[6] = {0,0,0,0,0,0};
335     int CURRENT_HIGH[2] = {0,0};
336     int CURRENT_ERROR[2] = {0,0};    // Stromfehler im Lade / Lastkreis
337     int RPM_ERROR[6] = {0,0,0,0,0,0};
338     int VOLTAGE_ERROR = 0;       // Spannungsfehler
339     int SUM_ERRORS;
340     uint32_t WARNINGCODE = 0;
341     uint32_t ERRORCODE = 0;
342 
343     // (Grün) LED Betrieb AN
344     green_led_on();
345 
346     // (Rot) LED Fehler AUS
347     red_led_off();
348 
349     // Relais 1 öffnen
350     relay_1_on();
351 
352     // Relais 2 öffnen
353     relay_2_on();
354 
355     // schaltet AKUSTISCHES-SIGNAL AUS
356 //  buzzer_off();
357 
358     // für TEST
359     buzzer_toggle();
360 
361     // Netzteil starten, wenn es nicht bereits getan war (nur für den TEST, dies soll später der
        Zyklierconroller übernehmen)
362     power_supply_start();
363 
364     // Temperaturen überprüfen
365     for(z=0;z<6;z++)
366     {
367         // bei Warnungstemperatur
368         if(FAN_TEMPERATURE[z] >= LIMIT_FAN_TEMP_WARNING[z] && FAN_TEMPERATURE[z] <
              LIMIT_FAN_TEMP_ERROR[z])
369         {
370             if(ERROR_LEVEL < 1)      // wenn der Fehlerlevel nicht bereits höher war
371             {
372                 ERROR_LEVEL = 1;
373             }
374 
375             TEMP_HIGH[z] = 1; // für den Summenmerker
376 
```

```
377             // für Temperaturen gilt Warnblock 1X
378             WARNINGCODE = 10 + z;
379
380             // Nicht benötigt für TESTLAUF
381 //          //über UART senden ERRORLEVEL / ERROR / VALUE
382 //          UART_send_warning(ERROR_LEVEL, WARNINGCODE, FAN_TEMPERATURE[z]);
383
384         }
385         // bei Fehlertemperatur (also größer als oben)
386         else if(FAN_TEMPERATURE[z] >= LIMIT_FAN_TEMP_ERROR[z])
387         {
388             if(ERROR_LEVEL < 2)      // wenn der Fehlerlevel nicht bereits höher war
389             {
390                 ERROR_LEVEL = 2;
391             }
392
393             TEMP_ERROR[z] = 1; // für den Summenmerker
394
395             // für Temperaturen gilt Fehlerblock 1X
396             ERRORCODE = 10 + z;
397
398             // Nicht benötigt für TESTLAUF
399 //          //über UART senden ERRORLEVEL / ERROR / VALUE
400 //          UART_send_error(ERROR_LEVEL ,ERRORCODE , FAN_TEMPERATURE[z]);
401         }
402     }
403
404     // Strom überprüfen
405     for(z=0;z<2;z++)
406     {
407         // bei Warnungsstrom
408         if(Hallsensor_Ampere[z] >= LIMIT_HALL_CURRENT_WARNING[z] && Hallsensor_Ampere[z] <
                LIMIT_HALL_CURRENT_ERROR[z])
409         {
410             if(ERROR_LEVEL < 1)      // wenn der Fehlerlevel nicht bereits höher war
411             {
412                 ERROR_LEVEL = 1;
413             }
414
415             CURRENT_HIGH[z] = 1; // für den Summenmerker
416
417             // für Ströme gilt Warnblock 2X
418             WARNINGCODE = 20 + z;
419
420             // Nicht benötigt für TESTLAUF
421 //          //über UART senden ERRORLEVEL / ERROR / VALUE
422 //          UART_send_warning(ERROR_LEVEL, WARNINGCODE, Hallsensor_Ampere[z]);
423
424         }
425         // bei Fehlerstrom (also größer als oben)
426         else if(Hallsensor_Ampere[z] >= LIMIT_HALL_CURRENT_ERROR[z])
427         {
428             if(ERROR_LEVEL < 2)      // wenn der Fehlerlevel nicht bereits höher war
429             {
430                 ERROR_LEVEL = 2;
431             }
432
433             CURRENT_ERROR[z] = 1; // für den Summenmerker
434
435             // für Ströme gilt Fehlerblock 2X
```

```
436                 ERRORCODE = 20 + z;
437
438                 // Nicht benötigt für TESTLAUF
439 //              //über UART senden ERRORLEVEL / ERROR / VALUE
440 //              UART_send_error(ERROR_LEVEL ,ERRORCODE , Hallsensor_Ampere[z]);
441             }
442         }
443
444         // überprüfen, ob die Lüfter sich drehen
445 //      for(z=0;z<6;z++)
446         for(z=0;z<3;z++) // ZUM TESTEN !!! da hier nur 3 Lüfter angeschlossen sind
447         {
448             if(FAN_RPM[z] < 100)        // wenn die Drehzahl unter 100 U/min fällt
449             {
450                 if(ERROR_LEVEL < 2)     // wenn der Fehlerlevel nicht bereits höher war
451                 {
452                     ERROR_LEVEL = 2;
453                 }
454
455                 RPM_ERROR[z] = 1; // für den Summenmerker
456
457                 // für Drehzahlen gilt Fehlerblock 3X
458                 ERRORCODE = 30 + z;
459
460                 // Nicht benötigt für TESTLAUF
461 //              //über UART senden ERRORLEVEL / ERROR / VALUE
462 //              UART_send_error(ERROR_LEVEL ,ERRORCODE , FAN_RPM[z]);
463             }
464         }
465
466         SUM_ERRORS =
467                 TEMP_HIGH[0] + TEMP_HIGH[1] + TEMP_HIGH[2] + TEMP_HIGH[3] + TEMP_HIGH[4] + TEMP_HIGH[5] +
468                 TEMP_ERROR[0] + TEMP_ERROR[1] + TEMP_ERROR[2] + TEMP_ERROR[3] + TEMP_ERROR[4] +
                        TEMP_ERROR[5] +
469                 CURRENT_HIGH[0] + CURRENT_HIGH[1] +
470                 CURRENT_ERROR[0] + CURRENT_ERROR[1] +
471                 RPM_ERROR[0] + RPM_ERROR[1] + RPM_ERROR[2] + RPM_ERROR[3] + RPM_ERROR[4] + RPM_ERROR[5] +
472                 VOLTAGE_ERROR
473                 ;
474         if(SUM_ERRORS == 0)
475         {
476             ERROR_LEVEL = 0;
477         }
478
479         // Drehzahl steuern
480         control_fan_RPM();
481
482 }
483
484 void SM_Error(void)
485 {
486         // Netzteil stoppen, wenn es nicht bereits getan war (nur für den TEST, dies soll später der
                Zyklierconroller übernehmen)
487         power_supply_stop();
488
489         // schaltet SCHUTZCONTROLLER-BEREIT-SIGNAL AUS
490         controller_ready_off();
491
492         // (Grün) LED Betrieb AN
493         green_led_on();
```

```
494
495     // (Rot) LED Fehler AN
496     red_led_on();
497
498     // Relais 1 öffnen
499     relay_1_off();
500
501     // Relais 2 öffnen
502     relay_2_off();
503
504     // Piepton an
505     buzzer_toggle();
506
507     // Drehzahl steuern
508     control_fan_RPM();
509
510 }
```

Listing A.33: SM_Functions.h

```
1  #ifndef SM_FUNCTIONS_H_
2  #define SM_FUNCTIONS_H_
3  //*****************************************************************************
4  //
5  // Prototypes for the functions.
6  //
7  //*****************************************************************************
8  extern void temperature_limit_warning_time_error(uint32_t);
9  extern void temperature_limit_error(uint32_t number);
10 extern void red_led_on(void);
11 extern void red_led_off(void);
12 extern void green_led_on(void);
13 extern void green_led_off(void);
14 extern void relay_1_on(void);
15 extern void relay_1_off(void);
16 extern void relay_2_on(void);
17 extern void relay_2_off(void);
18 extern void buzzer_toggle(void);
19 extern void buzzer_off(void);
20 extern void controller_ready_on(void);
21 extern void controller_ready_off(void);
22 extern void rom_variables_write(void);
23 extern void rom_variables_read(void);
24 extern void SM_Init(void);
25 extern void SM_Soft_Stop(void);
26 extern void SM_Run(void);
27 extern void SM_Error(void);
28 extern void UART_send_warning(uint32_t UART_L, uint32_t UART_W, uint32_t UART_V);
29 extern void UART_send_error(uint32_t UART_L, uint32_t UART_E, uint32_t UART_V);
30 extern void control_fan_RPM(void);
31 extern void power_supply_start(void);
32 extern void power_supply_stop(void);
33
34 #endif /* SM_FUNCTIONS_H_ */
```

Listing A.34: startup_ccs.c

```
1  //*****************************************************************************
2  //
```

```
3  // startup_ccs.c - Startup code for use with TI's Code Composer Studio.
4  //
5  // Copyright (c) 2013-2014 Texas Instruments Incorporated.  All rights reserved.
6  // Software License Agreement
7  //
8  // Texas Instruments (TI) is supplying this software for use solely and
9  // exclusively on TI's microcontroller products. The software is owned by
10 // TI and/or its suppliers, and is protected under applicable copyright
11 // laws. You may not combine this software with "viral" open-source
12 // software in order to form a larger program.
13 //
14 // THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
15 // NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
16 // NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
17 // A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
18 // CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
19 // DAMAGES, FOR ANY REASON WHATSOEVER.
20 //
21 // This is part of revision 2.1.0.12573 of the EK-TM4C1294XL Firmware Package.
22 //
23 //*****************************************************************************
24
25 #include <stdint.h>
26 #include "inc/hw_nvic.h"
27 #include "inc/hw_types.h"
28
29 //*****************************************************************************
30 //
31 // Forward declaration of the default fault handlers.
32 //
33 //*****************************************************************************
34 void ResetISR(void);
35 static void NmiSR(void);
36 static void FaultISR(void);
37 static void IntDefaultHandler(void);
38
39 //*****************************************************************************
40 //
41 // External declaration for the reset handler that is to be called when the
42 // processor is started
43 //
44 //*****************************************************************************
45 extern void _c_int00(void);
46 extern void TIMER5A_Interrupt_Handler(void);                                    // HIER
      ÄNDERUNG !!! EIGENE INTERRUPT FUNKTION !!!!
47 extern void TIMER5B_Interrupt_Handler(void);                                    // HIER
      ÄNDERUNG !!! EIGENE INTERRUPT FUNKTION !!!!
48
49 //*****************************************************************************
50 //
51 // Linker variable that marks the top of the stack.
52 //
53 //*****************************************************************************
54 extern uint32_t __STACK_TOP;
55
56 //*****************************************************************************
57 //
58 // External declaration for the interrupt handler used by the application.
59 //
60 //*****************************************************************************
```

```
61  extern void TouchScreenIntHandler(void);
62
63  //*****************************************************************************
64  //
65  // The vector table.  Note that the proper constructs must be placed on this to
66  // ensure that it ends up at physical address 0x0000.0000 or at the start of
67  // the program if located at a start address other than 0.
68  //
69  //*****************************************************************************
70  #pragma DATA_SECTION(g_pfnVectors, ".intvecs")
71  void (* const g_pfnVectors[])(void) =
72  {
73      (void (*)(void))((uint32_t)&__STACK_TOP),
74                                              // The initial stack pointer
75      ResetISR,                               // The reset handler
76      NmiSR,                                  // The NMI handler
77      FaultISR,                               // The hard fault handler
78      IntDefaultHandler,                      // The MPU fault handler
79      IntDefaultHandler,                      // The bus fault handler
80      IntDefaultHandler,                      // The usage fault handler
81      0,                                      // Reserved
82      0,                                      // Reserved
83      0,                                      // Reserved
84      0,                                      // Reserved
85      IntDefaultHandler,                      // SVCall handler
86      IntDefaultHandler,                      // Debug monitor handler
87      0,                                      // Reserved
88      IntDefaultHandler,                      // The PendSV handler
89      IntDefaultHandler,                      // The SysTick handler
90      IntDefaultHandler,                      // GPIO Port A
91      IntDefaultHandler,                      // GPIO Port B
92      IntDefaultHandler,                      // GPIO Port C
93      IntDefaultHandler,                      // GPIO Port D
94      IntDefaultHandler,                      // GPIO Port E
95      IntDefaultHandler,                      // UART0 Rx and Tx
96      IntDefaultHandler,                      // UART1 Rx and Tx
97      IntDefaultHandler,                      // SSI0 Rx and Tx
98      IntDefaultHandler,                      // I2C0 Master and Slave
99      IntDefaultHandler,                      // PWM Fault
100     IntDefaultHandler,                      // PWM Generator 0
101     IntDefaultHandler,                      // PWM Generator 1
102     IntDefaultHandler,                      // PWM Generator 2
103     IntDefaultHandler,                      // Quadrature Encoder 0
104     IntDefaultHandler,                      // ADC Sequence 0
105     IntDefaultHandler,                      // ADC Sequence 1
106     IntDefaultHandler,                      // ADC Sequence 2
107     TouchScreenIntHandler,                  // ADC Sequence 3
108     IntDefaultHandler,                      // Watchdog timer
109     IntDefaultHandler,                      // Timer 0 subtimer A
110     IntDefaultHandler,                      // Timer 0 subtimer B
111     IntDefaultHandler,                      // Timer 1 subtimer A
112     IntDefaultHandler,                      // Timer 1 subtimer B
113     IntDefaultHandler,                      // Timer 2 subtimer A
114     IntDefaultHandler,                      // Timer 2 subtimer B
115     IntDefaultHandler,                      // Analog Comparator 0
116     IntDefaultHandler,                      // Analog Comparator 1
117     IntDefaultHandler,                      // Analog Comparator 2
118     IntDefaultHandler,                      // System Control (PLL, OSC, BO)
119     IntDefaultHandler,                      // FLASH Control
120     IntDefaultHandler,                      // GPIO Port F
```

```
121     IntDefaultHandler,                      // GPIO Port G
122     IntDefaultHandler,                      // GPIO Port H
123     IntDefaultHandler,                      // UART2 Rx and Tx
124     IntDefaultHandler,                      // SSI1 Rx and Tx
125     IntDefaultHandler,                      // Timer 3 subtimer A
126     IntDefaultHandler,                      // Timer 3 subtimer B
127     IntDefaultHandler,                      // I2C1 Master and Slave
128     IntDefaultHandler,                      // CAN0
129     IntDefaultHandler,                      // CAN1
130     IntDefaultHandler,                      // Ethernet
131     IntDefaultHandler,                      // Hibernate
132     IntDefaultHandler,                      // USB0
133     IntDefaultHandler,                      // PWM Generator 3
134     IntDefaultHandler,                      // uDMA Software Transfer
135     IntDefaultHandler,                      // uDMA Error
136     IntDefaultHandler,                      // ADC1 Sequence 0
137     IntDefaultHandler,                      // ADC1 Sequence 1
138     IntDefaultHandler,                      // ADC1 Sequence 2
139     IntDefaultHandler,                      // ADC1 Sequence 3
140     IntDefaultHandler,                      // External Bus Interface 0
141     IntDefaultHandler,                      // GPIO Port J
142     IntDefaultHandler,                      // GPIO Port K
143     IntDefaultHandler,                      // GPIO Port L
144     IntDefaultHandler,                      // SSI2 Rx and Tx
145     IntDefaultHandler,                      // SSI3 Rx and Tx
146     IntDefaultHandler,                      // UART3 Rx and Tx
147     IntDefaultHandler,                      // UART4 Rx and Tx
148     IntDefaultHandler,                      // UART5 Rx and Tx
149     IntDefaultHandler,                      // UART6 Rx and Tx
150     IntDefaultHandler,                      // UART7 Rx and Tx
151     IntDefaultHandler,                      // I2C2 Master and Slave
152     IntDefaultHandler,                      // I2C3 Master and Slave
153     IntDefaultHandler,                      // Timer 4 subtimer A
154     IntDefaultHandler,                      // Timer 4 subtimer B
155     TIMER5A_Interrupt_Handler,              // Timer 5 subtimer A          // HIER ÄNDERUNG !!!
            EIGENE INTERRUPT FUNKTION !!!!
156     TIMER5B_Interrupt_Handler,              // Timer 5 subtimer B          // HIER ÄNDERUNG !!!
            EIGENE INTERRUPT FUNKTION !!!!
157     IntDefaultHandler,                      // FPU
158     0,                                      // Reserved
159     0,                                      // Reserved
160     IntDefaultHandler,                      // I2C4 Master and Slave
161     IntDefaultHandler,                      // I2C5 Master and Slave
162     IntDefaultHandler,                      // GPIO Port M
163     IntDefaultHandler,                      // GPIO Port N
164     0,                                      // Reserved
165     IntDefaultHandler,                      // Tamper
166     IntDefaultHandler,                      // GPIO Port P (Summary or P0)
167     IntDefaultHandler,                      // GPIO Port P1
168     IntDefaultHandler,                      // GPIO Port P2
169     IntDefaultHandler,                      // GPIO Port P3
170     IntDefaultHandler,                      // GPIO Port P4
171     IntDefaultHandler,                      // GPIO Port P5
172     IntDefaultHandler,                      // GPIO Port P6
173     IntDefaultHandler,                      // GPIO Port P7
174     IntDefaultHandler,                      // GPIO Port Q (Summary or Q0)
175     IntDefaultHandler,                      // GPIO Port Q1
176     IntDefaultHandler,                      // GPIO Port Q2
177     IntDefaultHandler,                      // GPIO Port Q3
178     IntDefaultHandler,                      // GPIO Port Q4
```

```
179     IntDefaultHandler,                      // GPIO Port Q5
180     IntDefaultHandler,                      // GPIO Port Q6
181     IntDefaultHandler,                      // GPIO Port Q7
182     IntDefaultHandler,                      // GPIO Port R
183     IntDefaultHandler,                      // GPIO Port S
184     IntDefaultHandler,                      // SHA/MD5 0
185     IntDefaultHandler,                      // AES 0
186     IntDefaultHandler,                      // DES3DES 0
187     IntDefaultHandler,                      // LCD Controller 0
188     IntDefaultHandler,                      // Timer 6 subtimer A
189     IntDefaultHandler,                      // Timer 6 subtimer B
190     IntDefaultHandler,                      // Timer 7 subtimer A
191     IntDefaultHandler,                      // Timer 7 subtimer B
192     IntDefaultHandler,                      // I2C6 Master and Slave
193     IntDefaultHandler,                      // I2C7 Master and Slave
194     IntDefaultHandler,                      // HIM Scan Matrix Keyboard 0
195     IntDefaultHandler,                      // One Wire 0
196     IntDefaultHandler,                      // HIM PS/2 0
197     IntDefaultHandler,                      // HIM LED Sequencer 0
198     IntDefaultHandler,                      // HIM Consumer IR 0
199     IntDefaultHandler,                      // I2C8 Master and Slave
200     IntDefaultHandler,                      // I2C9 Master and Slave
201     IntDefaultHandler                       // GPIO Port T
202 };
203
204 //*****************************************************************************
205 //
206 // This is the code that gets called when the processor first starts execution
207 // following a reset event.  Only the absolutely necessary set is performed,
208 // after which the application supplied entry() routine is called.  Any fancy
209 // actions (such as making decisions based on the reset cause register, and
210 // resetting the bits in that register) are left solely in the hands of the
211 // application.
212 //
213 //*****************************************************************************
214 void
215 ResetISR(void)
216 {
217     //
218     // Jump to the CCS C initialization routine.  This will enable the
219     // floating-point unit as well, so that does not need to be done here.
220     //
221     __asm("    .global _c_int00\n"
222           "    b.w     _c_int00");
223 }
224
225 //*****************************************************************************
226 //
227 // This is the code that gets called when the processor receives a NMI.  This
228 // simply enters an infinite loop, preserving the system state for examination
229 // by a debugger.
230 //
231 //*****************************************************************************
232 static void
233 NmiSR(void)
234 {
235     //
236     // Enter an infinite loop.
237     //
238     while(1)
```

```
239     {
240     }
241 }
242
243 //*****************************************************************************
244 //
245 // This is the code that gets called when the processor receives a fault
246 // interrupt.  This simply enters an infinite loop, preserving the system state
247 // for examination by a debugger.
248 //
249 //*****************************************************************************
250 static void
251 FaultISR(void)
252 {
253     //
254     // Enter an infinite loop.
255     //
256     while(1)
257     {
258     }
259 }
260
261 //*****************************************************************************
262 //
263 // This is the code that gets called when the processor receives an unexpected
264 // interrupt.  This simply enters an infinite loop, preserving the system state
265 // for examination by a debugger.
266 //
267 //*****************************************************************************
268 static void
269 IntDefaultHandler(void)
270 {
271     //
272     // Go into an infinite loop.
273     //
274     while(1)
275     {
276     }
277 }
```

Listing A.35: Startvalues.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3
4  // eigene Includes
5  #include "globals.h"
6
7  void Startvalues(void)
8  {
9      FAN_DUTYCYCLE_PERCENT[0] = 10;
10     FAN_DUTYCYCLE_PERCENT[1] = 10;
11     FAN_DUTYCYCLE_PERCENT[2] = 10;
12     FAN_DUTYCYCLE_PERCENT[3] = 10;
13     FAN_DUTYCYCLE_PERCENT[4] = 10;
14     FAN_DUTYCYCLE_PERCENT[5] = 10;
15     DEVICE_AVAILABLE=0;             // speichern ob DEVICE Vorhanden ist 0=NEIN, 1=JA
16     main_Timer_Value = 0;           // 100ms Zähler
17     onewire_Timer_Value = 0;        // 1-wire Zähler
18     Timer_5B_Value = 0;
```

```
19      TOGGLE_500ms = 0;
20      TOGGLE_1000ms = 0;
21      YELLOW_BUTTON = 0;
22      RED_BUTTON = 0;
23      BLACK_BUTTON = 0;
24      ERROR_LEVEL = 0;
25      SM_INIT = 0;
26      DISPLAY_ROTATER = 0;
27      SYSTEM_STATUS_TEXT = 0;
28      SM_Soft_Stop_active = 0;
29
30      //ANFANGSWERTE
31      LIMIT_FAN_TEMP_WARNING[0] = 5500; // 50,00 Grad Celsius
32      LIMIT_FAN_TEMP_WARNING[1] = 5500; // 50,00 Grad Celsius
33      LIMIT_FAN_TEMP_WARNING[2] = 5500; // 50,00 Grad Celsius
34      LIMIT_FAN_TEMP_WARNING[3] = 9000; // 50,00 Grad Celsius // geändert für Testlauf
35      LIMIT_FAN_TEMP_WARNING[4] = 9000; // 50,00 Grad Celsius // geändert für Testlauf
36      LIMIT_FAN_TEMP_WARNING[5] = 9000; // 50,00 Grad Celsius // geändert für Testlauf
37
38      LIMIT_FAN_TEMP_ERROR[0] = 6000; // 60,00 Grad Celsius
39      LIMIT_FAN_TEMP_ERROR[1] = 6000; // 60,00 Grad Celsius
40      LIMIT_FAN_TEMP_ERROR[2] = 6000; // 60,00 Grad Celsius
41      LIMIT_FAN_TEMP_ERROR[3] = 9900; // 60,00 Grad Celsius // geändert für Testlauf
42      LIMIT_FAN_TEMP_ERROR[4] = 9900; // 60,00 Grad Celsius // geändert für Testlauf
43      LIMIT_FAN_TEMP_ERROR[5] = 9900; // 60,00 Grad Celsius // geändert für Testlauf
44
45      LIMIT_HALL_CURRENT_WARNING[0] = 1100;  // 110 Ampere // geändert für Testlauf auf 1100 A ! (nie
            erreicht)
46      LIMIT_HALL_CURRENT_WARNING[1] = 1200;  // 110 Ampere // geändert für Testlauf auf 1200 A ! (nie
            erreicht)
47
48      LIMIT_HALL_CURRENT_ERROR[0] = 1100;  // 120 Ampere // geändert für Testlauf auf 1100 A ! (nie
            erreicht)
49      LIMIT_HALL_CURRENT_ERROR[1] = 1200;  // 120 Ampere // geändert für Testlauf auf 1100 A ! (nie
            erreicht)
50  }
```

Listing A.36: Startvalues.h

```
1   #ifndef STARTVALUES_H_
2   #define STARTVALUES_H_
3   //*****************************************************************************
4   //
5   // Prototypes for the functions.
6   //
7   //*****************************************************************************
8   void Startvalues(void);
9
10  #endif /* STARTVALUES_H_ */
```

Listing A.37: Timer_Init.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
```

```
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/timer.h"
13 #include "driverlib/timer.c"
14
15 // eigene Includes
16 #include "globals.h"
17
18 void Timer_Init(void)
19 {
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
21     // aktiviert GPIO Ports an Port_A
22     // benötigt für TIMER2 --> siehe: PINMUX --> (PA4_T2CCP0) und (PA5_T2CCP1)
23     // benötigt für TIMER3 --> siehe: PINMUX --> (PA6_T3CCP0) und (PA7_T3CCP1)
24
25     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
26     // aktiviert GPIO Ports an Port_B
27     // benötigt für TIMER4 --> siehe: PINMUX --> (PB0_T4CCP0) und (PB1_T4CCP1)
28     // benötigt für TIMER5 --> siehe: PINMUX --> (PB2_T5CCP0) und (PB3_T5CCP1)
29
30     // TIMER5 ---> der Hochzähltimer ----------------------------------------------
31     //
32     SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER5);
33     // aktiviert TIMER5: (PORT_B, PIN2 & PIN3)
34     // siehe "Pin Mux Utility" bei der "aktivierung" von TIMER5
35
36     TimerConfigure(TIMER5_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC_UP |
           TIMER_CFG_B_PERIODIC_UP);
37     // konfiguriert den TIMER5 A&B als zwei von einander unabhängige 16 Bit Timer ( TIMER5A [16 Bit]
           & TIMER5B [16 Bit] )
38     // TIMER5: A&B werden auf Hochzählen eingestellt
39
40     timer_a_length = ui32SysClkFreq / 2000 ; // 0,000 500 sek
41     // Der MC läuft auf f = 120.000.000 Hz --> dies bedeutet, dass jeder "Clock-Pulse" T = 1 / f
           dauert.
42     // 1-Clock-Pulse: T = 1 / 120.000.000 Hz = 8,333 * 10^-9 sek (jeder Schritt dauert 8,3 ns)
43     // der Timer soll alle 500us auslösen => 0,000 500 s / 0,000 000 008 333 s = 60.024,009
44     // 120 Mhz / X = 0,060 024 MHz => X = 1999,2 = 2000
45     // (SysCtlClockGet = 120.000.000 Hz) / 2000 = 60.000 Hz
46     // ein 16 BIT Timer zählt MAX bis 65.535 !!!!!!
47
48     timer_b_length = ui32SysClkFreq / 100000 ; // 0,000 010 sek
49     // Der MC läuft auf f = 120.000.000 Hz --> dies bedeutet, dass jeder "Clock-Pulse" T = 1 / f
           dauert.
50     // 1-Clock-Pulse: T = 1 / 120.000.000 Hz = 8,333 * 10^-9 sek (jeder Schritt dauert 8,3 ns)
51     // der Timer soll alle 10us auslösen => 0,000 010 s / 0,000 000 008 333 s = 1.200,480
52     // 120 Mhz / X = 0,0012 MHz => X = 100.000
53     // (SysCtlClockGet = 120.000.000 Hz) / 100.000 = 1.200 Hz
54     // ein 16 BIT Timer zählt MAX bis 65.535 !!!!!!
55
56     TimerLoadSet(TIMER5_BASE, TIMER_A, timer_a_length);
57     // lässt den Timer bei 60.000 anfangen zu zählen, damit er alle 500us auf 0 läuft.
58
59     TimerLoadSet(TIMER5_BASE, TIMER_B, timer_b_length);
60     // lässt den Timer bei 1.200 anfangen zu zählen, damit er alle 10us auf 0 läuft.
61
62     IntEnable(INT_TIMER5A);
63     // hier wird eine Vektor der mit TIMER5 verbunden ist, aktiviert
```

```
64
65      IntEnable(INT_TIMER5B);
66      // hier wird eine Vektor der mit TIMER5 verbunden ist, aktiviert
67
68      TimerIntEnable(TIMER5_BASE, TIMER_TIMA_TIMEOUT);
69      // aktiviert ein "Event" im Timer, um ein "Interrupt" zu generieren
70      // HIER: wird ein "Interrupt" bei einem Ablauf von TIMER5A generiert
71
72      TimerIntEnable(TIMER5_BASE, TIMER_TIMB_TIMEOUT);
73      // aktiviert ein "Event" im Timer, um ein "Interrupt" zu generieren
74      // HIER: wird ein "Interrupt" bei einem Ablauf von TIMER5B generiert
75
76      //
77      // TIMER5  ENDE----------------------------------------------------------------------
78
79
80
81      // TIMER2 ---> die Flankenzähler TimerA und TimerB-----------------------------------
82      //
83      GPIOPinConfigure(GPIO_PA4_T2CCP0);
84      // GPIO Pin PA4 als T2CCP0 einstellen
85      // auf diesen PIN reagiert der Timer (Event Counter)
86      // siehe PinMux Utility
87
88      GPIOPinConfigure(GPIO_PA5_T2CCP1);
89      // GPIO Pin PA5 als T2CCP1 einstellen
90      // auf diesen PIN reagiert der Timer (Event Counter)
91      // siehe PinMux Utility
92
93      GPIOPinTypeTimer(GPIO_PORTA_BASE, GPIO_PIN_4 | GPIO_PIN_5);
94      // GPIO Pin PA4 als Timer PIN einstellen
95      // GPIO Pin PA5 als Timer PIN einstellen
96      // auf diesen PIN reagiert der Timer (Event Counter)
97      // siehe PinMux Utility
98
99      SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
100     // aktiviert TIMER2: (PORT_A, PIN4 & PIN5)
101     // siehe "Pin Mux Utility" bei der "aktivierung" von TIMER2
102
103     TimerConfigure(TIMER2_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_CAP_COUNT | TIMER_CFG_B_CAP_COUNT)
            ;
104     // konfiguriert den TIMER2 als 2 einzelne 16 Bit Timer ( TIMER2A [16 Bit] & TIMER2B [16 Bit] )
105     // TIMER2A: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
106     // TIMER2B: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
107
108     TimerControlEvent(TIMER2_BASE, TIMER_BOTH, TIMER_EVENT_POS_EDGE);
109     // der TIMER 2A & 2B werden auf steigende Flanken eingestellt
110     // TIMER2A: hiermit zählt er jede steigende Flanke
111     // TIMER2B: hiermit zählt er jede steigende Flanke
112     //
113     TimerLoadSet(TIMER2_BASE, TIMER_BOTH, 10000);
114     // lässt den Edge Counter von 10.000 an herunterzählen
115     //
116     // TIMER2  ENDE----------------------------------------------------------------------
117
118     // TIMER3 ---> die Flankenzähler TimerA und TimerB-----------------------------------
119     //
120     GPIOPinConfigure(GPIO_PA6_T3CCP0);
121     // GPIO Pin PA6 als T3CCP0 einstellen
122     // auf diesen PIN reagiert der Timer (Event Counter)
```

```
123     // siehe PinMux Utility
124
125     GPIOPinConfigure(GPIO_PA7_T3CCP1);
126     // GPIO Pin PA7 als T3CCP1 einstellen
127     // auf diesen PIN reagiert der Timer (Event Counter)
128     // siehe PinMux Utility
129
130     GPIOPinTypeTimer(GPIO_PORTA_BASE, GPIO_PIN_6 | GPIO_PIN_7);
131     // GPIO Pin PA6 als Timer PIN einstellen
132     // GPIO Pin PA7 als Timer PIN einstellen
133     // auf diesen PIN reagiert der Timer (Event Counter)
134     // siehe PinMux Utility
135
136     SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
137     // aktiviert TIMER3: (PORT_A, PIN6 & PIN7)
138     // siehe "Pin Mux Utility" bei der "aktivierung" von TIMER3
139
140     TimerConfigure(TIMER3_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_CAP_COUNT | TIMER_CFG_B_CAP_COUNT)
          ;
141     // konfiguriert den TIMER3 als 2 einzelne 16 Bit Timer ( TIMER3A [16 Bit] & TIMER3B [16 Bit] )
142     // TIMER3A: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
143     // TIMER3B: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
144
145     TimerControlEvent(TIMER3_BASE, TIMER_BOTH, TIMER_EVENT_POS_EDGE);
146     // der TIMER 3A & 3B werden auf steigende Flanken eingestellt
147     // TIMER3A: hiermit zählt er jede steigende Flanke
148     // TIMER3B: hiermit zählt er jede steigende Flanke
149     //
150     TimerLoadSet(TIMER3_BASE, TIMER_BOTH, 10000);
151     // lässt den Edge Counter von 10.000 an herunterzählen
152     //
153     // TIMER3  ENDE----------------------------------------------------------------------------
154
155     // TIMER4 ---> die Flankenzähler TimerA und TimerB-----------------------------------------
156     //
157     GPIOPinConfigure(GPIO_PB0_T4CCP0);
158     // GPIO Pin PB0 als T4CCP0 einstellen
159     // auf diesen PIN reagiert der Timer (Event Counter)
160     // siehe PinMux Utility
161
162     GPIOPinConfigure(GPIO_PB1_T4CCP1);
163     // GPIO Pin PB1 als T4CCP1 einstellen
164     // auf diesen PIN reagiert der Timer (Event Counter)
165     // siehe PinMux Utility
166
167     GPIOPinTypeTimer(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
168     // GPIO Pin PB0 als Timer PIN einstellen
169     // GPIO Pin PB1 als Timer PIN einstellen
170     // auf diesen PIN reagiert der Timer (Event Counter)
171     // siehe PinMux Utility
172
173     SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER4);
174     // aktiviert TIMER4: (PORT_B, PIN0 & PIN1)
175     // siehe "Pin Mux Utility" bei der "aktivierung" von TIMER4
176
177     TimerConfigure(TIMER4_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_CAP_COUNT | TIMER_CFG_B_CAP_COUNT)
          ;
178     // konfiguriert den TIMER4 als 2 einzelne 16 Bit Timer ( TIMER4A [16 Bit] & TIMER4B [16 Bit] )
179     // TIMER4A: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
180     // TIMER4B: wird auf Herunterzählen eingestellt (jedes Event wird gezählt)
```

```
181
182     TimerControlEvent(TIMER4_BASE, TIMER_BOTH, TIMER_EVENT_POS_EDGE);
183     // der TIMER 4A & 4B werden auf steigende Flanken eingestellt
184     // TIMER4A: hiermit zählt er jede steigende Flanke
185     // TIMER4B: hiermit zählt er jede steigende Flanke
186     //
187     TimerLoadSet(TIMER4_BASE, TIMER_BOTH, 10000);
188     // lässt den Edge Counter von 10.000 an herunterzählen
189     //
190     // TIMER4  ENDE----------------------------------------------------------------------
191
192     //TimerEnable(TIMER5_BASE, TIMER_A);
193     // hier wird der Timer aktiviert (jetzt beginnt er zu zählen)
194
195     TimerEnable(TIMER5_BASE, TIMER_BOTH);
196     // hier wird der Timer aktiviert (jetzt beginnt er zu zählen)
197
198     TimerEnable(TIMER2_BASE, TIMER_BOTH);
199     // hier wird der Timer aktiviert (jetzt beginnt er zu zählen)
200
201     TimerEnable(TIMER3_BASE, TIMER_BOTH);
202     // hier wird der Timer aktiviert (jetzt beginnt er zu zählen)
203
204     TimerEnable(TIMER4_BASE, TIMER_BOTH);
205     // hier wird der Timer aktiviert (jetzt beginnt er zu zählen)
206
207 }
```

Listing A.38: Timer_Init.h

```
1  #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_TIMER_PWM_DETECTION_INIT_H_
2  #define SCHUTZCONTROLLER_MIT_DISPLAY_TIMER_PWM_DETECTION_INIT_H_
3
4  /*****************************************************************************
5   *
6   * Function Declarations
7   *
8   *****************************************************************************/
9
10 void Timer_Init(void);
11
12 #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_TIMER_PWM_DETECTION_INIT_H_ */
```

Listing A.39: TIMER5A_Interrupt_Handler.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/timer.h"
13 #include "driverlib/interrupt.h"
14 #include "utils/uartstdio.h"
```

```
15  #include "driverlib/uart.h"
16
17  // eigene Includes
18  #include "globals.h"
19  #include "ADC_Measure.h"
20  #include "UART_Ausgabe.h"
21
22  // dieser Interrupt kommt alle 0,000 500s
23  void TIMER5A_Interrupt_Handler(void)
24  {
25      int STEP = 500; // 500 us Schrittweite
26
27      TimerIntClear(TIMER5_BASE, TIMER_TIMA_TIMEOUT);
28      // hier wird die "Interrupt"-Quelle "gelöscht" (für den Neuanfang am Ende der FKT)
29
30
31      // Hauptzähler um 500us erhöhen
32      main_Timer_Value = main_Timer_Value + STEP;
33
34
35      ADC_Measure();
36      // Achtung hier wird ebenfalls auf das Ende der Messung gewartet !!!
37      // while(!ADCIntStatus(ADC1_BASE, 1, false)) ----> Wartet, bis die Konvertierung fertig ist.
38
39
40      // 1-wire Zeitschleife
41      if(one_wire_measure_active == 1)
42      {
43          onewire_Timer_Value = onewire_Timer_Value + STEP;
44
45          if(onewire_Timer_Value == 800000)
46          {
47              onewire_Timer_Value = 0;
48              onewire_800ms_wait = 1;
49          }
50      }
51
52      // NUR FÜR TESTLAUF ----> Messwerte Ausgabe via Putty
53      if(main_Timer_Value == 5000000 || main_Timer_Value == 4000000 || main_Timer_Value == 3000000 ||
            main_Timer_Value == 2000000 || main_Timer_Value == 1000000)
54      {
55          // für Test geändert, sonst wird dies in Timer5B gemacht (Sekundenzähler)
56          Sekunden = Sekunden + 1;
57
58          // Sekunden
59          UARTprintf("%d,", Sekunden);
60
61          // Lüfter TEMP
62          UARTprintf("%d,", FAN_TEMPERATURE[0]);
63          UARTprintf("%d,", FAN_TEMPERATURE[1]);
64          UARTprintf("%d,", FAN_TEMPERATURE[2]);
65          UARTprintf("%d,", FAN_TEMPERATURE[3]);
66          UARTprintf("%d,", FAN_TEMPERATURE[4]);
67
68          // Lüfter DUTY
69          UARTprintf("%d,", FAN_DUTYCYCLE_PERCENT[0]);
70          UARTprintf("%d,", FAN_DUTYCYCLE_PERCENT[1]);
71          UARTprintf("%d,", FAN_DUTYCYCLE_PERCENT[2]);
72
73          // Lüfter RPM
```

```
74          UARTprintf("%d,", FAN_RPM[0]);
75          UARTprintf("%d,", FAN_RPM[1]);
76          UARTprintf("%d,", FAN_RPM[2]);
77
78          // ADC Strom (Sensor 1)
79          UARTprintf("%d,", Hallsensor_Ampere[1]);
80
81          // welcher System Status besteht ?
82          UARTprintf("%d,", SYSTEM_STATUS_TEXT);
83
84          // Netzteil AN/AUS
85          UARTprintf("%d", POWER_SUPPLY_ON_OFF);
86
87          // neue Zeile
88          UARTprintf("\r\n");
89
90      }
91
92      // alle 5 Sekunden ausführen, da die gemessenen Werte (Umdrehungen) sonst zu gering sind
93      // (wegen der ganzzahligen Rundungen)
94      if(main_Timer_Value == 5000000)
95      {
96          TIMER2A_Value = TimerValueGet(TIMER2_BASE, TIMER_A);
97          // Wert auslesen
98
99          TIMER2B_Value = TimerValueGet(TIMER2_BASE, TIMER_B);
100         // Wert auslesen
101
102         TIMER3A_Value = TimerValueGet(TIMER3_BASE, TIMER_A);
103         // Wert auslesen
104
105         TIMER3B_Value = TimerValueGet(TIMER3_BASE, TIMER_B);
106         // Wert auslesen
107
108         TIMER4A_Value = TimerValueGet(TIMER4_BASE, TIMER_A);
109         // Wert auslesen
110
111         TIMER4B_Value = TimerValueGet(TIMER4_BASE, TIMER_B);
112         // Wert auslesen
113
114         TimerLoadSet(TIMER2_BASE, TIMER_BOTH, 10000);
115         // Rücksetzen der Zählanfangsvariablen auf 10.000
116
117         TimerLoadSet(TIMER3_BASE, TIMER_BOTH, 10000);
118         // Rücksetzen der Zählanfangsvariablen auf 10.000
119
120         TimerLoadSet(TIMER4_BASE, TIMER_BOTH, 10000);
121         // Rücksetzen der Zählanfangsvariablen auf 10.000
122
123         // RPM Werte aktualisieren (alle 5 Sek)-------------------------------
124         FAN_RPM[0] = ((10000 - TIMER2A_Value) * 12) / 2;
125         FAN_RPM[1] = ((10000 - TIMER2B_Value) * 12) / 2;
126         FAN_RPM[2] = ((10000 - TIMER3A_Value) * 12) / 2;
127         FAN_RPM[3] = ((10000 - TIMER3B_Value) * 12) / 2;
128         FAN_RPM[4] = ((10000 - TIMER4A_Value) * 12) / 2;
129         FAN_RPM[5] = ((10000 - TIMER4B_Value) * 12) / 2;
130         // kommt alle 5sek *12 = 60 sek = RPM
131         // :2 da der Lüfter 2 Takte pro Umdrehung macht
132         //-------------------------------------------------------------------
133
```

```
134             // Hauptzähler wieder auf 0ms zurücksetzen
135             main_Timer_Value = 0;
136
137             //alle 5 Sekunden die Messwerte auf die serielle Schnittstelle ausgeben
138             //UART_Ausgabe();
139
140         }
141
142     }
```

Listing A.40: TIMER5A_Interrupt_Handler.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_TIMER5A_INTERRUPT_HANDLER_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_TIMER5A_INTERRUPT_HANDLER_H_
3
4   /*****************************************************************************
5    *
6    * Function Declarations
7    *
8    *****************************************************************************/
9
10  void TIMER5A_Interrupt_Handler(void);
11
12  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_TIMER5A_INTERRUPT_HANDLER_H_ */
```

Listing A.41: TIMER5B_Interrupt_Handler.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12  #include "driverlib/timer.h"
13  #include "driverlib/interrupt.h"
14  #include "UART_Ausgabe.h"
15  #include "utils/uartstdio.h"
16
17  // eigene Includes
18  #include "globals.h"
19
20  void TIMER5B_Interrupt_Handler(void)                // dieser Interrupt kommt alle 10us
21  {
22      int STEP_B = 10; // 10 us Schrittweite
23
24      TimerIntClear(TIMER5_BASE, TIMER_TIMB_TIMEOUT);
25      // hier wird die "Interrupt"-Quelle "gelöscht" (für den Neuanfang am Ende der FKT)
26
27      // Timer Zähler um 10us erhöhen
28      Timer_5B_Value = Timer_5B_Value + STEP_B;
29
30      // Hilfstoggle für das Display (TOUCH-KAPUTT)
31      if(Timer_5B_Value == 1000000)
32      {
```

```
33          DISPLAY_ROTATER = 1;
34      }
35
36      switch(Timer_5B_Value)
37      {
38          case 500000:     // halbe Sekunde
39
40              // TOGGLE  500ms Variable
41              if(TOGGLE_500ms==0) TOGGLE_500ms = 1;
42                  else TOGGLE_500ms = 0;
43
44              break;
45
46          case 1000000:    // eine Sekunde
47
48              // TOGGLE  500ms Variable
49              if(TOGGLE_500ms==0) TOGGLE_500ms = 1;
50                  else TOGGLE_500ms = 0;
51
52              // TOGGLE  1000ms Variable
53              if(TOGGLE_1000ms==0) TOGGLE_1000ms = 1;
54                  else TOGGLE_1000ms = 0;
55
56              // für TEST geändert
57              //Sekunden = Sekunden + 1;
58
59              Timer_5B_Value = 0; // zurücksetzen
60
61              break;
62
63          default:
64
65              break;
66      }
67
68  }
```

Listing A.42: TIMER5B_Interrupt_Handler.h

```
1  #ifndef TIMER5B_INTERRUPT_HANDLER_H_
2  #define TIMER5B_INTERRUPT_HANDLER_H_
3  /*****************************************************************************
4   *
5   * Function Declarations
6   *
7   *****************************************************************************/
8
9  void TIMER5B_Interrupt_Handler(void);
10
11 #endif /* TIMER5B_INTERRUPT_HANDLER_H_ */
```

Listing A.43: UART_Ausgabe.c

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_types.h"
5  #include "driverlib/sysctl.h"
6  #include "driverlib/fpu.h"
```

```
7  #include "driverlib/gpio.h"
8  #include "driverlib/debug.h"
9  #include "driverlib/pwm.h"
10 #include "driverlib/pin_map.h"
11 #include "inc/hw_gpio.h"
12 #include "driverlib/uart.h"
13 #include "utils/uartstdio.h"
14
15 // eigene Includes
16 #include "globals.h"
17
18
19 void UART_Ausgabe(void)
20 {
21         //------// Periodische Ausgabe der Daten auf UART0 -> Terminal Programm
22     UARTprintf("\033[2J\033[H"); // lösche Bildschirm, lege Cursor auf Position (0,0)
23     UARTprintf("|=============================================================================|\r\n");
24     UARTprintf("|         Testprogramm fuer Lueftersteuerung                                  |\r\n");
25     UARTprintf("|=============================================================================|\r\n");
26     UARTprintf("| Luefter 1 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[0], FAN_RPM[0], FAN_TEMPERATURE[0]);
27     UARTprintf("|=============================================================================|\r\n");
28     UARTprintf("| Luefter 2 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[1], FAN_RPM[1], FAN_TEMPERATURE[1]);
29     UARTprintf("|=============================================================================|\r\n");
30     UARTprintf("| Luefter 3 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[2], FAN_RPM[2], FAN_TEMPERATURE[2]);
31     UARTprintf("|=============================================================================|\r\n");
32     UARTprintf("| Luefter 4 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[3], FAN_RPM[3], FAN_TEMPERATURE[3]);
33     UARTprintf("|=============================================================================|\r\n");
34     UARTprintf("| Luefter 5 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[4], FAN_RPM[4], FAN_TEMPERATURE[4]);
35     UARTprintf("|=============================================================================|\r\n");
36     UARTprintf("| Luefter 6 |  Dutycycle in %%: %3d | RPM: %6d | Temperatur: %6d C |\r\n",
           FAN_DUTYCYCLE_PERCENT[5], FAN_RPM[5], FAN_TEMPERATURE[5]);
37     UARTprintf("|=============================================================================|\r\n");
38     UARTprintf("|=============================================================================|\r\n");
39     UARTprintf("|  Scratchpad CRC: Luefter 1:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[0][8], SCRATCHPAD_CRC[0]);
40     UARTprintf("|=============================================================================|\r\n");
41     UARTprintf("|  Scratchpad CRC: Luefter 2:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[1][8], SCRATCHPAD_CRC[1]);
42     UARTprintf("|=============================================================================|\r\n");
43     UARTprintf("|  Scratchpad CRC: Luefter 3:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[2][8], SCRATCHPAD_CRC[2]);
44     UARTprintf("|=============================================================================|\r\n");
45     UARTprintf("|  Scratchpad CRC: Luefter 4:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[3][8], SCRATCHPAD_CRC[3]);
46     UARTprintf("|=============================================================================|\r\n");
47     UARTprintf("|  Scratchpad CRC: Luefter 5:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[4][8], SCRATCHPAD_CRC[4]);
48     UARTprintf("|=============================================================================|\r\n");
49     UARTprintf("|  Scratchpad CRC: Luefter 6:  %3d <-> %3d                                    |\r\n",
           SCRATCHPAD_FAN[5][8], SCRATCHPAD_CRC[5]);
50     UARTprintf("|=============================================================================|\r\n");
51     UARTprintf("|=============================================================================|\r\n");
52     UARTprintf("|  ADC1-Messwert: %6d    Hallsensor-Strom: %6d                               |\r\n",
           ADC_Value_AIN1, Hallsensor_Ampere[0]);
53     UARTprintf("|=============================================================================|\r\n");
```

```
54      UARTprintf("|   ADC2-Messwert: %6d    Hallsensor-Strom: %6d                        |\r\n",
            ADC_Value_AIN2, Hallsensor_Ampere[1]);
55      UARTprintf("|================================================================================|\r\n");
56      UARTprintf("ROM Test  : %d\r\n", RELAY_COUNTER[0]);
57      UARTprintf("ROM Test  : %d\r\n", RELAY_COUNTER[1]);
58  }
```

Listing A.44: UART_Ausgabe.h

```
1   #ifndef SCHUTZCONTROLLER_UART_AUSGABE_H_
2   #define SCHUTZCONTROLLER_UART_AUSGABE_H_
3
4   /*******************************************************************************
5    *
6    * Function Declarations
7    *
8    *******************************************************************************/
9
10  void UART_Ausgabe(void);
11
12  #endif /* SCHUTZCONTROLLER_UART_AUSGABE_H_ */
```

Listing A.45: UART_Init.c

```
1   #include <stdint.h>
2   #include <stdbool.h>
3   #include "inc/hw_memmap.h"
4   #include "inc/hw_types.h"
5   #include "driverlib/sysctl.h"
6   #include "driverlib/fpu.h"
7   #include "driverlib/gpio.h"
8   #include "driverlib/debug.h"
9   #include "driverlib/pwm.h"
10  #include "driverlib/pin_map.h"
11  #include "inc/hw_gpio.h"
12  #include "driverlib/uart.h"
13  #include "utils/uartstdio.h"
14
15  // eigene Includes
16  #include "globals.h"
17
18  void UART_Init(void)
19  {
20      SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
21      // aktiviert UART0 Modul: die UART0 Ein-/Ausgänge sind über PORT_B & PORT_H & PORT_K & PORT_P
            verteilt
22      // siehe "Pin Mux Utility" bei der "aktivierung" von UART0
23      // genutzt soll hier werden: Port_A, da dort RX(an PA0) und TX(an PA1) vorhanden sind
24
25      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
26      // aktiviert GPIO Ports an Port_A
27      // benötigt für UART0 --> siehe: PINMUX --> (PA0_U0RX) und (PA1_U0TX)
28
29      GPIOPinConfigure(GPIO_PA0_U0RX);
30      // GPIO-Pin-PA0 als TX (Transmit) Pin einstellen
31
32      GPIOPinConfigure(GPIO_PA1_U0TX);
33      // GPIO-Pin-PA1 als RX (Recieve) Pin einstellen
34
```

```
35      GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
36      // GPIO PINs von Port_A (PIN0 und PIN1) als UART PINs verwenden
37
38      UARTConfigSetExpClk(UART0_BASE, ui32SysClkFreq, 115200,(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE
            | UART_CONFIG_PAR_NONE));
39      // Parameter für UART0 Schnittstelle einstellen: Baudrate=115200, 8-1-N-N
40
41      UARTStdioConfig(0, 115200, ui32SysClkFreq);
42      // hier wird für die "UARTprintf" Funktion der UART Standard auf UART0 gestellt
43      // damit "UARTprintf" weiß, wo es die Zeichen hinschicken soll.....
44  }
```

Listing A.46: UART_Init.h

```
1   #ifndef SCHUTZCONTROLLER_MIT_DISPLAY_UART_INIT_H_
2   #define SCHUTZCONTROLLER_MIT_DISPLAY_UART_INIT_H_
3
4   /*****************************************************************************
5    *
6    * Function Declarations
7    *
8    *****************************************************************************/
9
10  void UART_Init(void);
11
12  #endif /* SCHUTZCONTROLLER_MIT_DISPLAY_UART_INIT_H_ */
```

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, 16. November 2015	
Ort, Datum	Unterschrift