HAW HAMBURG

BACHELORTHESIS
Fabian Mahler

Optische Datenkommunikation im Batteriemodul von Elektrofahrzeugen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

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

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Fabian Mahler

Optische Datenkommunikation im Batteriemodul von Elektrofahrzeugen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr. Pawel Adam Buczek

Eingereicht am: 30. September 2021

Fabian Mahler

Thema der Arbeit

Optische Datenkommunikation im Batteriemodul von Elektrofahrzeugen

Stichworte

Mikrocontroller, Datenübertragung, Kommunikation, IrDA, UART, Funktionsaufbau, Batteriezelle, Batteriemodul, Lichtleitkörper, IrDA-Transceiver, IrDA-SIR, Matlab, C-Quellcode, Sensoren

Kurzzusammenfassung

Die vorliegende Arbeit befasst sich mit den Thematiken der optischen Datenkommunikation mittels Infrarot-Transceiver-Modulen und eines Lichtleitkörpers. Es wird ein Funktionsaufbau konstruiert, der sich an den Abmessungen einer realen Fahrzeugbatterie orientiert. Die Datenübertragung ist eine Anlehnung an die IrDA-SIR-Spezifikation. Die Kommunikation erfolgt nach dem Master-Slave Prinzip. Für die Konzeption des Übertragungsprotokolls werden reale Datenmengen betrachtet und evaluiert.

Fabian Mahler

Title of Thesis

Optical data communication in the battery module of electric vehicles

Keywords

microcontroller, data transfer, communication, IrDA, UART, function model, battery cell, battery module, light conductor, IrDA transceiver, IrDA SIR, Matlab, c source code, sensors

Abstract

The present thesis deals with the topics of optical data communication by means of infrared transceiver modules and a light conductor. A function model is designed that is based on the dimensions of a real vehicle battery. The data transmission is based on the IrDA-SIR specification. Communication takes place according to the master slave principle. Real data amounts are considered and evaluated for the conception of the transmission protocol.

Inhaltsverzeichnis

1 Einführung

1.1 Ausgangssituation

Elektrofahrzeuge werden weltweit immer beliebter als Alternative zu Verbrennungsmotoren. Die Gründe hierfür sind sehr vielfältig. Im Fokus stehen dabei Nachhaltigkeitsbewusstsein, staatliche Subventionen und steuerliche Vorteile sowie weniger anfallende Kosten pro gefahrenem Kilometer.

Die anfänglichen Nachteile, wie etwa eine geringe Reichweite und oder eine lange Ladezeit der Akkumulatoren, konnten in den vergangenen Jahren durch innovative Lösungen deutlich verbessert werden. So sind die aktuellen Reichweiten von Elektrofahrzeugen für den Nahbereich ausreichend, für Langstrecken ist jedoch noch Spielraum nach oben.

Die Komplexität, der in einem Elektrofahrzeug installierten Technik, ist dabei deutlich höher, als in einem vergleichbarem Fahrzeug mit Verbrennungsmotor. Die Leistungselektronik, die für den Antrieb benötigt wird, wird nach aktuellem Stand mit Lithium-Ionen-Batterien realisiert. Ein Batteriemodul besteht dabei aus mehreren verschalteten Batteriezellen. In einem BMW i3 sind so beispielsweise 96 Batteriezellen in 8 Batteriemodulen mit einer Gesamtsystemspannung von 360 Volt verbaut [9].

Für die Funktionalität eines Elektrofahrzeuges ist es notwendig, dass Daten von den Batteriezellen abgefragt werden können. Die Steuerelektronik benötigt beispielsweise Informationen über den Ladestand, die Temperatur der Zelle, die Leistungsabfrage uvm.

Die Übertragung der benötigten Daten, kann mittels diverser Ansätze realisiert werden. Aktueller Stand der Technik ist eine Übertragung mittels elektrischer Leitungen. Es gibt dabei noch andere Ansätze der Datenübertragung, etwa die Datenübertragung mittels elektromagnetischer Strahlung und der optischen Übertragung, die im Abschnitt 2.1 genauer untersucht werden, mit Darstellung der jeweiligen Vor- und Nachteile.

Im folgenden Abschnitt werden die Ziele dieser Arbeit definiert.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Laboraufbau auf Basis von Vorarbeiten zu entwickeln. Dazu gehört die Optimierung des Lichtleitkörpers durch Simulation und Labormuster, die Entwicklung und der Aufbau von Zellcontrollerplatinen, die Neuentwicklung des Modulcontrollers und der Entwurf entsprechender Controller- und Auswertesoftware.

Als Teilaufgabe soll eine Implementierung eines Datenformats erfolgen. Dieses soll die Aufgaben der Synchronisation, Addressierung, Datenübertragung und Fehlersicherung umfassen. Hierbei ist auf eine effiziente Nutzung der verfügbaren Datenrate zu achten. Als übergeordneter Teil soll eine Abschätzung des Datenvolumen im Modul erfolgen. Dem soll gegenübergestellt werden, welche Datenraten im IrDA-Format praktisch umsetzbar sind.

Anhand des entstehenden Laboraufbaus sollen wichtige Fragesellungen für die zukünftige Anwendung erarbeitet werden. Dazu gehören beispielsweise Abschätzungen der Sendeleistung und der mechanischen Toleranzen, dazu Aussagen zu Einflussfaktoren auf die Zuverlässigkeit der Übertragung.

Die folgende Übersicht zeigt die Inhalte dieser Thesis mit Entwicklungsstand.

	Theoretische Überlegung	Konzeptionell	Implementiert
Funktionsdemonstrator			
Betrachtung von Zeitverhalten und Synchronität			
Fehlersicherungskonzepte bei der Datenübertragung			
Abschätzung des Overhead des zu adaptierenden Übertragungsprotokolls			
Erweiterung der Batteriemodulcontrollersoftware um UDP-Schnittstelle			
PC-Software zur Visualisierung und Steuerung des Batteriemodulcontrollers			
Test der Komponenten und des gesamten Aufbaus			

Abbildung 1.1: Inhalte dieser Thesis mit Entwicklungsstand

2 Grundlagen und Vorarbeiten

2.1 Vorarbeiten

In diesem Abschnitt der Thesis wird auf die Vorarbeiten zu der gegebenen Thematik eingegangen.

Die Vorarbeiten dieser Thesis beschäftigen sich mit der Fragestellung, wie die Übertragung von Informationen zwischen einem Batteriezellencontroller und einem Haupsteuergerät in einer Batterie eines Elektrofahrzeugs realisiert werden kann.

Wichtige Aspekte dabei sind die Robustheit, Störsicherheit und Realisierbarkeit des jeweiligen Kommunikationsansatzes.

2.1.1 Übertragung mit Hilfe von elektrischen Leitern

Bei dieser Art der Kommunikation handelt es sich um das Daisy-Chain Prinzip. Hier sind die Busteilnehmer in einer Kette seriell miteinander verbunden. Die Busteilnehmer beziehungsweise Module besitzen dabei jeweils einen Eingang, an dem der vorgeschaltete Busteilnehmer angeschlossen ist und einen Ausgang an den der nachfolgende Busteilnehmer angeschlossen ist. Potenzialtrennende Komponenten der Zellcontroller dienen der Verbindung des Datenbusses mit den Zellcontrollern und des Haupsteuergeräts.

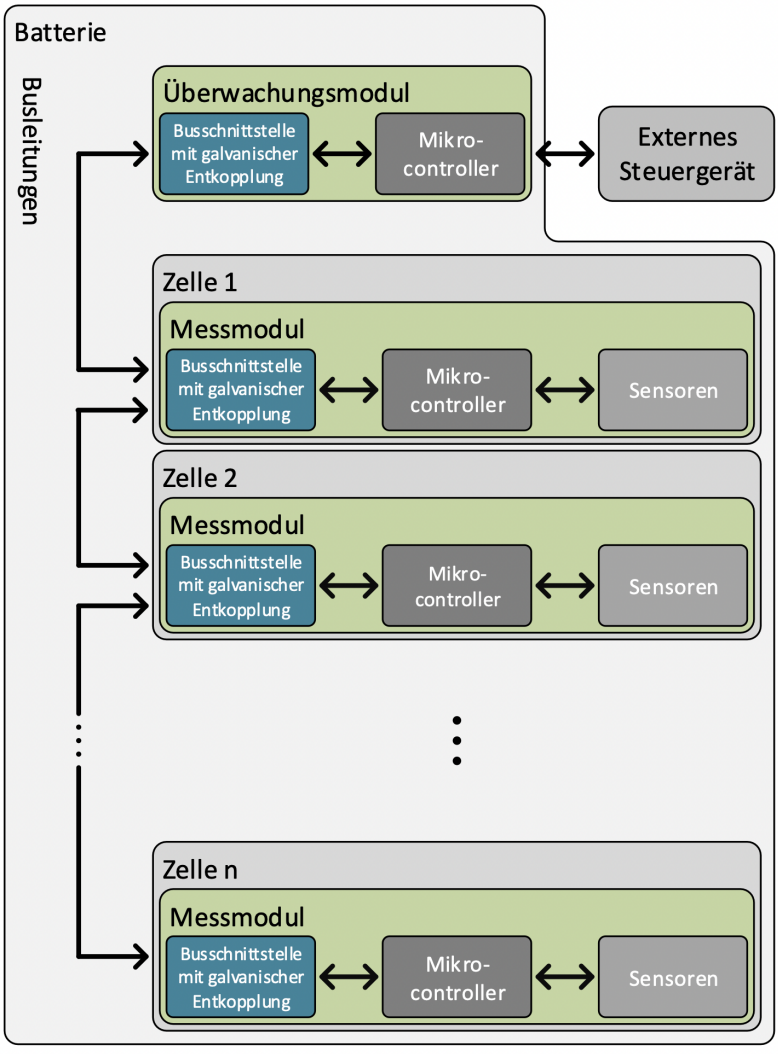


Abbildung 2.1: Aufbau Kommunikation der Busteilnehmer mittels Daisy-Chain Topologie [8, S. 3]

2.1.2 Übertragung mit Hilfe von Power-Line-Communication

Bei der Datenübertragung mit Hilfe der Power-Line-Communication (PLC) wird die Hauptstromleitung der Batterie als Übertragungsmedium verwendet. Das Datensignal kann von den Busteilnehmern kapazitiv oder induktiv ein- und ausgekoppelt werden.

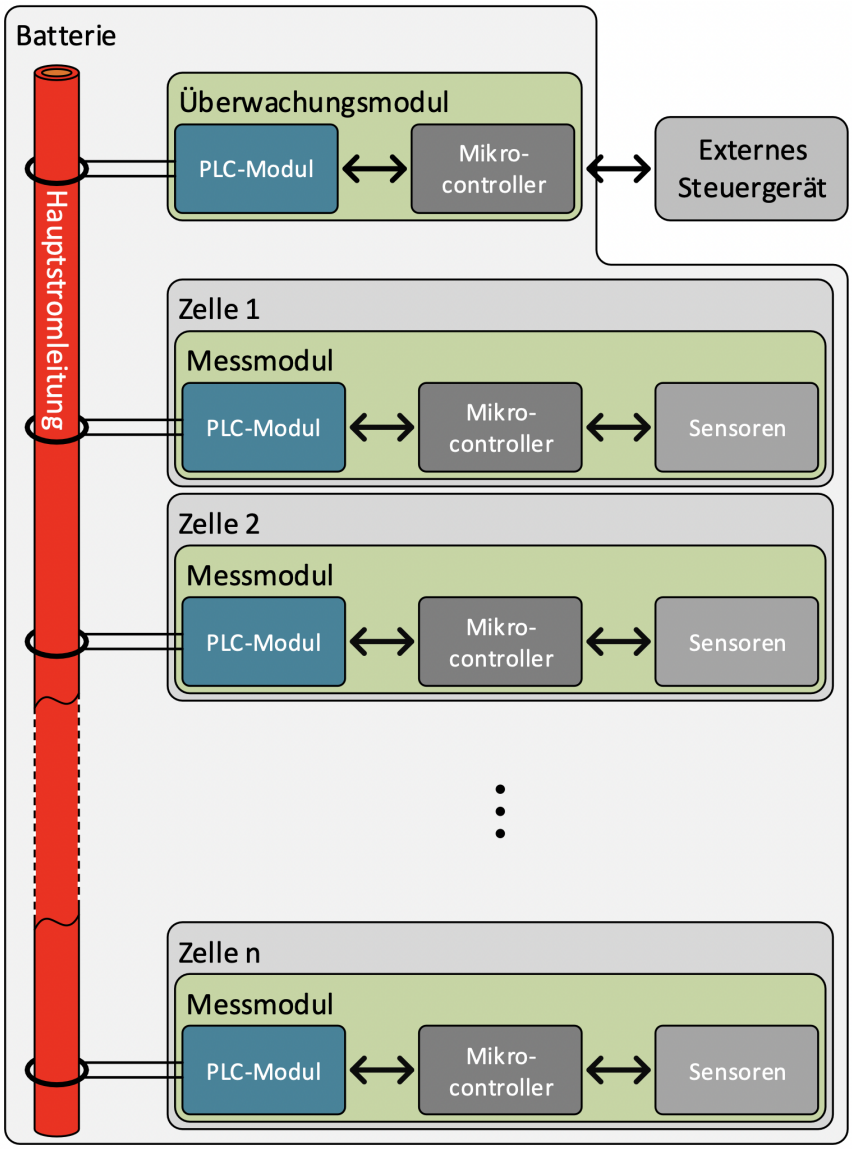


Abbildung 2.2: Aufbau Kommunikation der Busteilnehmer mittels Power-Line-Communication Topologie [8, S. 4]

2.1.3 Übertragung mit Hilfe von elektromagnetischen Radiowellen

Bei dieser Art der Datenübertragung werden die Daten mit Hilfe von elektromagnetischen Radiowellen übertragen. Die Zellcontroller sowie das Hauptsteuergerät sind dabei mit einer Antenne ausgestattet, die zum Senden und Empfangen der Daten benötigt wird.

Diese Thematik wird umfassend im Rahmen einer Bachelor- und Masterthesis von Nico Sassano bearbeitet [10] [11].

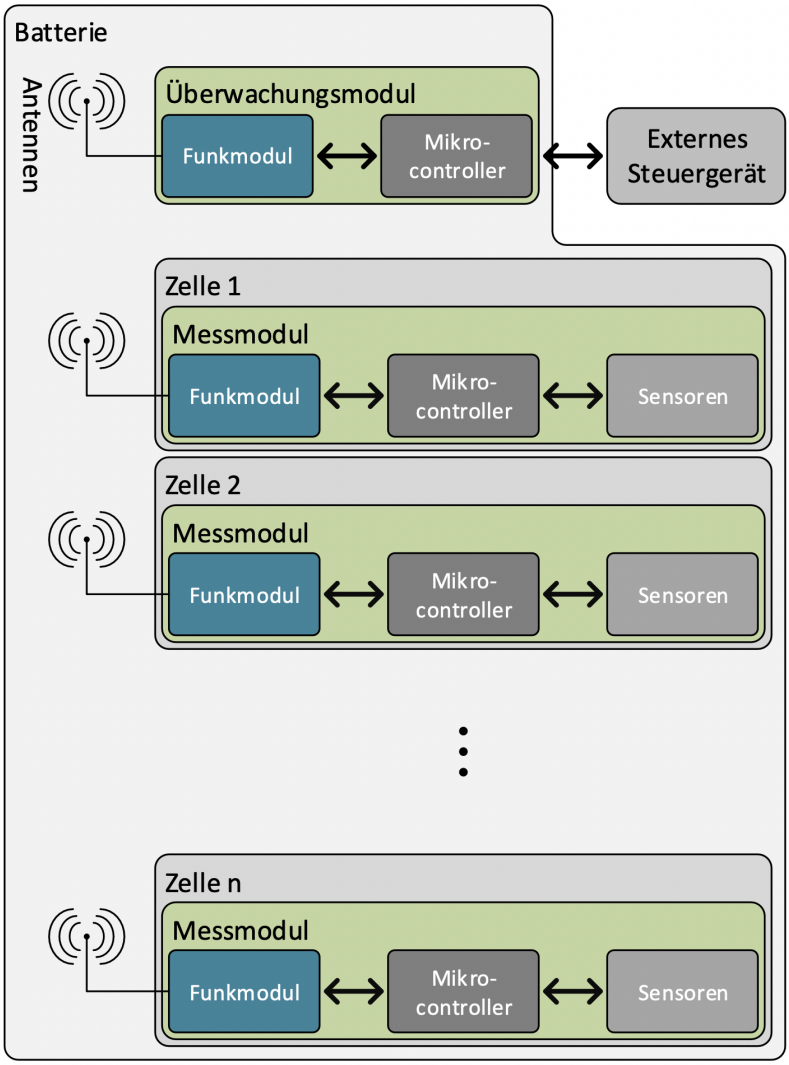


Abbildung 2.3: Aufbau Kommunikation der Busteilnehmer mit Hilfe elektromagnetischer Radiowellen [8, S. 5]

2.1.4 Übertragung mit Hilfe von einem Lichtleitkörper

Bei dieser Art der Datenübertragung werden die Daten mittels Lichtimpulsen auf einen Lichtleitkörper gegeben. Die Busteilnehmer sind mit Infrarotdioden an den Lichtwellenleiter gekoppelt, die zum Senden und Empfangen der Lichtimpulse dienen.

Diese Thematik wird im Rahmen einer Bachelorthesis von Jonas Ernsting [8] behandelt und dient als Ausgangspunkt für diese Thesis.

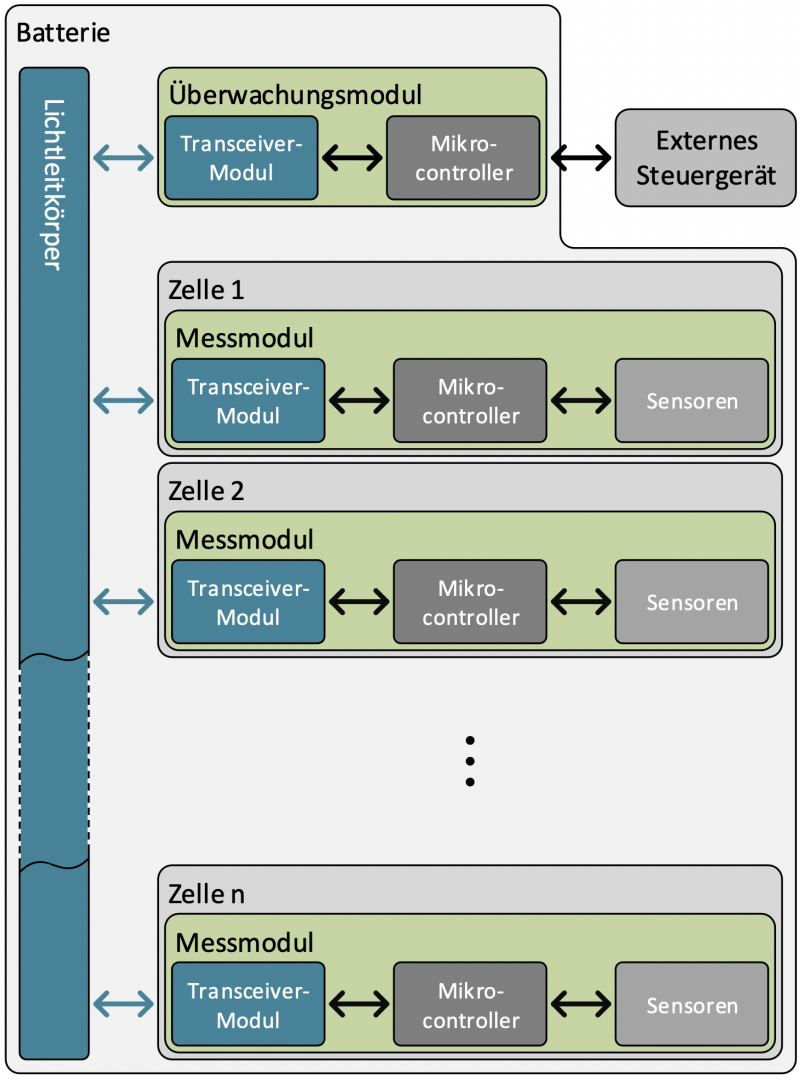


Abbildung 2.4: Aufbau Kommunikation der Busteilnehmer mit Hilfe eines Lichtleitkörpers [8, S. 6]

2.1.5 Vergleich der Ansätze der Datenübertragung

Grundlegend betrachtet, besitzt jeder Ansatz der Datenübertragung seine spezifischen Vor- und Nachteile. Diese werden nachfolgend qualitativ bewertet und gegenübergestellt.

Der größte Vorteil der Daisy-Chain Topologie ist seine einfache Implementierbarkeit. Es werden für jeden Busteilnhemer zwei Leitungen benötigt, um eine Busanbindung zu realisieren. Ein Nachteil hingegen ist ein nicht-dynamisches Prioritätsschema, dass die Priorität der Busteilnehmer regelt.

Die Übertragung mit Hilfe der Power-Line-Communication hat den Vorteil, dass keine zusätzlichen Leitungen benötigt werden. Da die Daten ein- und ausgekoppelt werden, muss auf EMV-Einflüsse geachtet werden.

Der Ansatz der Übertragung mit elektromagnetischen Wellen besitzt den Vorteil, dass er sich durch eine gute Erweiterbarkeit des virtuellen Busses auszeichnet, da neue Busteilnehmer nicht wie bei den anderen Ansätzen verdrahtet werden müssen. Ebenfalls hier sind EMV-relevante Aspekte bei der Implementierung zu berücksichtigen.

Die Datenübertragung über einen Lichtleitkörper bietet den Vorteil, dass der Bus galvanisch getrennt ist von den Busteilnehmern. Ebenfalls sind EMV-relevante Aspekte hier nicht vorhanden. Zu den Nachteilen gehören hier Störanfälligkeiten durch Ablagerungen von Staub und Dreck, sowie die eventuelle Funktionsbeeinträchtigung durch Kondensatbildung. Weitere relevante Aspekte dieses Ansatzes sind Gegenstand dieser Thesis.

Zusammenfassend betrachtet, lässt sich sagen, dass für die Implementierung in einem Elektrofahrzeug mehrere Aspekte in Hinsicht auf Störanfälligkeit, Robustheit, und Implementierbarkeit zu beachten sind. So ist davon auszugehen, dass in einem Elektrofahrzeug die Störanfälligkeit durch EMV-Einflüsse sehr hoch ist. Zudem muss bei der Implementierung eines Ansatzes auf mechanische Belastbarkeit, beziehungsweise Robustheit geachtet werden. Sonstige Einflüsse wie Temperatureinfluss, Kondensatentwicklung oder Verschmutzung der Komponenten sind ebenfalls zu betrachten.

Darüber hinaus muss darauf geachtet werden, dass der Installationsaufwand in einem definierten Rahmen bleibt. Das heißt, dass Komponenten ohne großen Aufwand installiert und gegebenenfalls einfach getauscht werden können, ohne dass zum Beispiel ein ganzes Batteriemodul entfernt werden muss.

2.2 IrDA

2.2.1 Einführung

IrDA (Infrared Data Association) ist ein Verbund von Unternehmen, die sich mit der Standardisierung von Infrarottransceivern und Protokollspezifikationen auseinandersetzen. Ziel ist die Schaffung eines kostengünstigen, energiesparenden halbduplex Übertragungsstandard für Infrarottransceiver.

2.2.2 IrDA Protocol Stack

Der IrDA Protocol Stack ist vergleichbar mit dem OSI-Referenzmodell für Netzwerkprotokolle. In der folgenden Abbildung dargestellt ist der IrDA Protocol Stack.

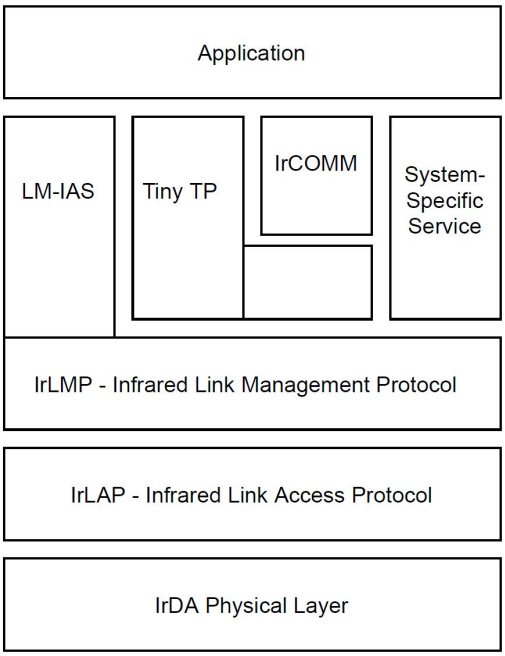


Abbildung 2.5: Aufbau des IrDA Protocol Stack, modifiziert nach: [12, S. 21]

Der unterste Layer des Protocol Stack besteht aus dem IrDA-Physical-Layer. In diesem Layer sind die Informationen über Übertragungsgeschwindigkeit, Modulation und Codierung definiert. Die folgende Tabelle fasst diese Informationen zusammen.

Bezeichnung	Datenrate	Modulation	Codierung
Serial Infrared SIR	9.6 kbit/s bis 115.2 kbit/s	RZI 3/16 Pulse	16 Bit CRC-CCIT
Medium Infrared MIR	0.576 Mbit/s bis 1.152 Mbit/s	RZI 1/4 Pulse	16 Bit CRC-CCIT
Fast Infrared FIR	4 Mbit/s	4 PPM	32 Bit CRC
Very Fast Infrared VFIR	16 Mbit/s	NRZI HHH(1,13)	32 Bit CRC
Ultra Fast Infrared UFIR	96 Mbit/s	NRZI 8B10B Code	32 Bit CRC
Giga IR	512 Mbit/s und 1Gbit/s	2- bzw. 4 ASK	32 Bit CRC

Tabelle 2.1: Übersicht: Übertragungsraten im IrDA-Physical-Layer mit Modulation und Codierung

Für eine Verbindung mit UART-Schnittstellen wird das Serial-Infrared verwendet, da dieses äquivalente Übertragungsraten aufweist.

Die zweite Schicht der IrDA-Protocol-Stack bildet das Infrared-Link-Acess-Protocol. IrLAP ist eine Adaption des High-Level Data Link Control (HDLC), welches ein normiertes Netzwerkprotokoll der OSI-Referenzmodellschicht 2 zuzuordnen ist. Die Aufgabe des IrLAP ist es, Kommunikationspartner zu ermitteln, die Zugriffskontrolle zu regeln, Adresskonflikte zu lösen, den Verbindungs- und Informationsaustausch zu initiieren und die Trennung zu verwalten. Die nächste Abbildung zeigt das Funktionsprinzip des IrLAP.

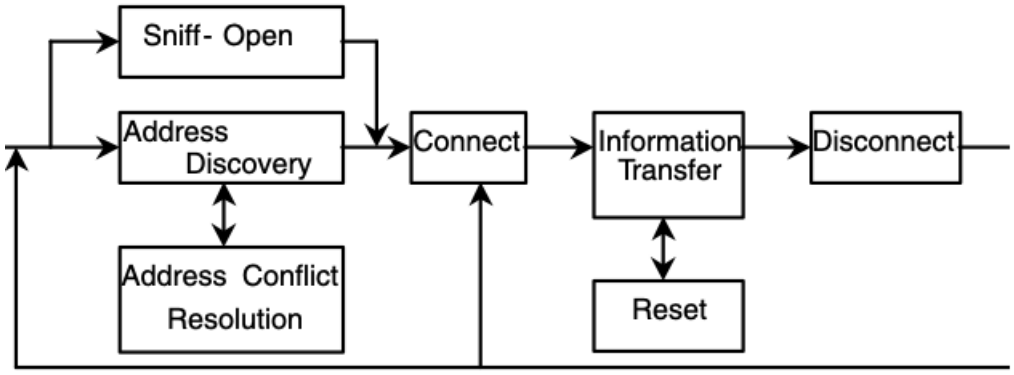


Abbildung 2.6: Funktionsweise des IrLAP [12, S. 21]

Zu Beginn des Verbindungsaufbaus wird nach verfügbaren Kommunikationsteilnehmern gesucht (Sniffing). Parallel dazu werden die Adressen der potenziellen Teilnehmer überprüft. Bei Auftreten eines Adressierungsfehlers, wird eine Adressenskonfliktlösung initiiert. Nach der erfolgreichen Verbindung mit einem Kommunikationspartner beginnt der Datentransfer. Während der Datenübertragung ist IrLAP für Fehlererkennung, erneute

Übertragung und Datenflusskontrolle verantwortlich. Zuletzt wird die aktive Verbindung getrennt.

Die dritte Schicht des IrDA-Protocol-Stack besteht aus dem Infrared-Link-Management-Protocol. Die Aufgaben des IrLAP sind die Verwaltung der verfügbaren Anwendungen und Funktionen zwischen den kommunizierenden Endgeräten. Darüber hinaus wird die fehlerfreie Datenübertragung überwacht. Zudem stellt das IrLAP Funktionen zum Multiplexen bereit, sodass mehrere Geräte gleichzeitig Daten senden und empfangen können.

Die übergeordneten Schichten des IrDA-Protocol-Stack werden hier nicht weiter erläutert.

2.2.3 Datenübertragung

Die IrDA-Datenübertragung verwendet am häufigsten eine USB-zu-UART oder die serielle Schnittstelle eines Mikrocontrollers. Die angestrebte Übertragungsgeschwindigkeit beträgt 115.2 kbit/s (SIR), die ebenfalls die maximale Übertragungsgeschwindigkeit von UART ist. Die verwendete Wellenlänge für die Übertragung beträgt 850 nm bis 900 nm.

2.2.4 IrDA-Hardware

In diesem Abschnitt wird auf die für diese Thesis relevante IrDA-Hardware eingegangen. Dabei handelt es sich um den TFDU4101 von Vishay Semiconductors [13], der zum Senden und Empfangen der Infrarot Lichtimpulse verwendet wird. Dieser ist in der folgenden Abbildung dargestellt.

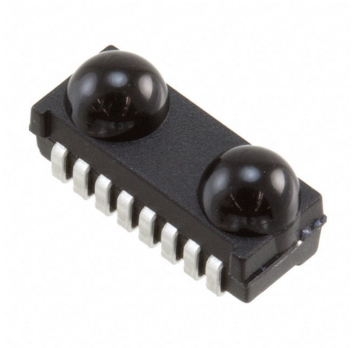


Abbildung 2.7: Infrarot Transceiver-Modul TFDU 4101 von Vishay Semiconductors [7]

Dieser Transceiver ist für Serial Infrared konzipiert mit einer Übertragungsrate von 115.2 kbit/s. In das Gehäuse integriert sind eine Foto-Diode zum Empfangen von Daten, ein Infrarot-Emitter (IRED) zum Senden von Daten und ein energiesparender Steuer-IC. Aufgrund der verwendeten internen Hardware sind Übertragungsreichweiten von über einen Meter ohne Lichtleitkörper erzielbar.

2.3 Komponenten

2.3.1 Mikrocontroller

Im Rahmen dieser Thesis werden Mikrocontroller programmiert, die nach dem Master-Slave Prinzip miteinander kommunizieren. Es werden dabei der Infineon XMC 2Go und der Infineon XMC4700 verwendet. In den folgenden Abbildungen sind die Mikrocontroller abgebildet.

Abbildung 2.8: Mikrocontroller Infineon XMC 2Go [5]

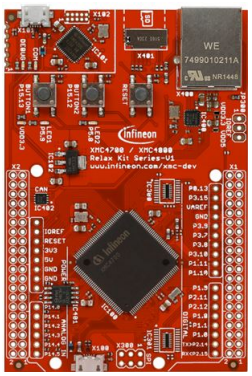


Abbildung 2.9: Mikrocontroller Infieon XMC4700 Relax Kit [6]

In Abbildung 2.8 dargestellt ist der XMC 2Go. Dieser wird später in einem Master-Slave-Netz als Slave eingesetzt. In Abbildung 2.9 dargestellt ist der XMC4700. Dieser Mikrocontroller wird später als Master eingesetzt und dient als Kommunikationsschnittstelle mit einem Computer. Der XMC4700 verfügt über eine RJ45-Ethernet-Schnittstelle, die für die Kommunikation mit einem Computer eingesetzt wird. Beide Mikrocontroller verfügen über einen ARM-Cortex-Prozessor mit ARM-Architektur.

2.3.2 Batteriezellencontroller

Für diese Arbeit wird im späteren Verlauf ein Funktionsdemonstrator auf Basis der Vorarbeit von Jonas Ernsting [8] konstruiert. Die Aufgabe des Funktionsdemonstrators ist es, einen Batterieblock mit zwölf Batteriezellen zu simulieren. Für die Batteriezellen wird ein Batteriezellencontroller entwickelt, der beispielhaft Sensordaten, wie etwa Batterietemperatur, Leistungsabruf, Ladestand usw., mit zwei Potentiometern abruft und anschließend an den Mastercontroller übermittelt. Auf dem Batteriezellencontroller befindet sich ein XMC 2Go, der die Daten verwaltet. Nachfolgend dargestellt ist der Batteriezellencontroller.

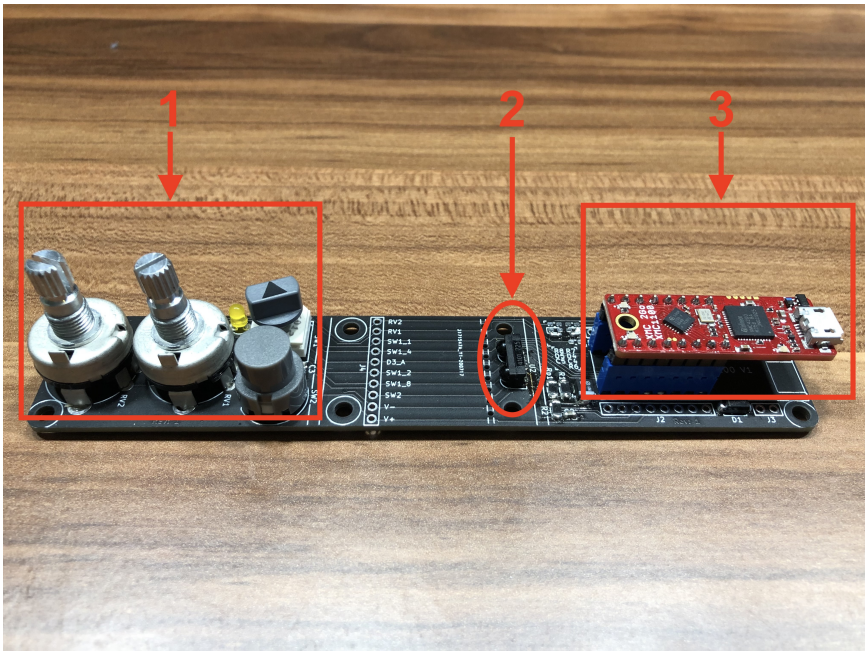


Abbildung 2.10: Batteriezellencontroller für den Funktionsdemonstrator. 1 Zeigt die Bedienelemente zur Simulation von Sensorwerten, 2 zeigt den IrDA-Transceiver, 3 zeigt den Mikrocontroller

Auf der Platine des Batteriezellencontrollers ist rechts der XMC 2Go Mikrocontroller zu erkennen (Nummer 3 im Bild). Mittig auf der Platine, vor dem XMC 2Go, ist der IrDA Transceiver zu erkennen (Nummer 2 im Bild) . Darüber hinaus sind links auf der Platine die zwei Potentiometer zu erkennen, mit denen beispielhafte Sensorwerte an den Mikrocontroller übergeben werden können. Ebenfalls befindet sich ein hexadezimaler Kodierschalter auf der Platine, mit dem eine Absolutadresse eingestellt werden kann.

Diese dient später der Adressierung. Unter dem Kodierschalter ist ein Druckknopf zu erkennen. Mit dem Druckknopf kann eine Änderung der eingestellten Adresse mit dem Kodierschalter quittiert werden. Zuletzt befindet sich eine gelbe LED auf der Platine. Die LED dient als Aktivitätsanzeige (Alle Elemente befinden sich in Bereich 1 im Bild).

2.3.3 Überwachungsmodul

Das Überwachungsmodul dient dem Master als Kommunikationsschnittstelle mit den Batteriezellencontrollern. Die Platine besteht dabei aus der Basis der Batteriezellencontrollerplatine. Das Überwachungsmodul ist in der folgenden Abbildung dargstellt.

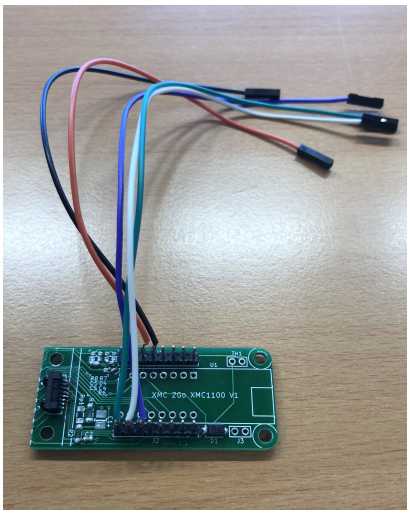


Abbildung 2.11: Überwachungsmodul als Schnittstelle für den Mastercontroller

Auf der Platine zu erkennen sind diverse Leitungen, die für die Kommunikation mit dem Mastercontroller dienen. Darüber hinaus ist vorne an der Platine ein TFDU4101 IrDA Transceiver angebracht, um mit den Batteriezellencontrollern kommunizieren zu können.

2.3.4 Lichtleitkörper

Da die Batteriezellencontroller über optische Signale miteinander kommunizieren sollen ist es essentziell, dass die Lichtsignale über ein Medium übertragen werden, damit alle Batteriezellencontroller miteinander verbunden sind. Zwar lassen sich mit dem

TFDU4101 Übertragungsreichweiten von über einem Meter erreichen, dies setzt aber voraus, dass die Kommunikationspartner sich direkt sehen. Da die Batteriezellencontroller nebeneinander angeordnet werden, wären somit nicht mehr alle Controller ansprechbar. Demnach ist es erforderlich, dass es einen Lichtleitkörper gibt, der die optischen Signale an alle Batteriezellencontroller gezielt reflektiert. Der Lichtleitkörper ist in der folgenden Abbildung dargestellt.

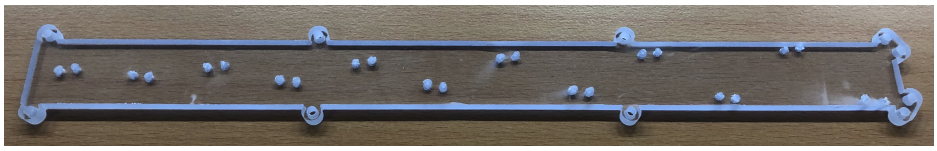


Abbildung 2.12: Lichtleitkörper als Datenbus, der dargestellte Lichtleitköroer basiert auf Basis von Reflexbohrungen

Der Lichtleitkörper besteht aus gefrästem Plexiglas und basiert auf dem Prinzip der Reflexbohrung. An den Stellen wo die IrDA-Transceiver platziert sind befinden sich jeweils zwei Bohrungen, die die Lichtimpulse so reflektieren, dass diese sich im Bereich der Linsen der Transceiver sammeln.

Im Rahmen der Vorarbeit [8] wurde ein Lichtleitkörper auf der Basis ausgerichteter Reflektorflächen entworfern, der für diese Arbeit verwendet wird. Der alternative Lichtleitkörper ist in der folgenden Abbildung zu erkennen.

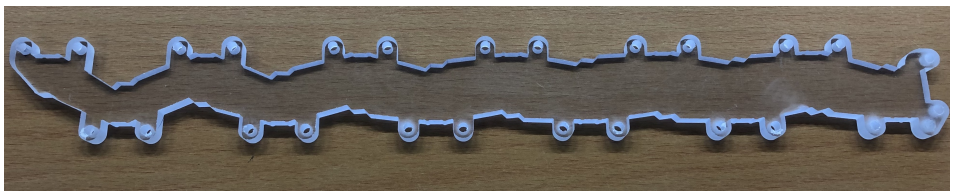


Abbildung 2.13: Lichleitkörper auf der Basis von ausgerichteten Reflektorflächen

3 Konzeption

3.1 Datenübertragung

3.1.1 Master-Slave-Prinzip

Die Mikrocontroller sind alle mittels der IrDA-Transceiverdioden an den Lichtleitkörper angebunden. Damit es auf dem Datenbus keine Datenkollisionen gibt, muss die geteilte Ressource durch ein hierarchisches Zugriffsverfahren aufgeteilt werden. Es gibt insgesamt 12 Batteriezellencontroller, die von einem übergeordneten Mikrocontroller gesteuert werden sollen. Es bietet sich daher an, die Kommunikation mittels Master-Slave-Prinzip zu realisieren. Der Master ist dabei der XMC4700 Controller, die Slavecontroller bestehen aus den steuerungs XMC 2Go Controllern auf dem Batteriezellencontroller. Dieser Controller übernimmt die Regelung des Zugriffs des Datenbusses. Ein großer Vorteil dieser Topologie ist, dass der Mastercontroller den Ressourcenzugriff verwaltet. Somit ist das Risiko der Datenkollision um einiges reduziert. Ein Nachteil dieser Topologie ist, dass die Slavecontroller untereinander nicht kommunizieren können.

Der Mastercontroller spricht einen Batteriezellencontroller an, der dann mit einer enstprechenden Antwort antwortet. Ein Batteriezellencontroller kann von sich aus nicht auf die geteilte Ressource zugreifen, bis er explizit vom Mastercontroller angesprochen wird.

3.1.2 Adressierung

Wenn ein Datenpaket auf dem Datenbus gesendet wird, können alle Busteilnehmer dieses Datenpaket empfangen. Auf das Master-Slave-Prinzip bezogen bedeutet das, dass ein vom Mastercontroller gesendetes Datenpaket von jedem Batteriezellencontroller gelesen werden kann. Es ist somit notwendig, ein Datenpaket mit einer Empfangsadresse zu versehen. So wird sichergestellt, dass nur der Batteriezellencontroller die Nachricht empfängt, für den sie vorgesehen ist.

Dabei lassen sich verschiedene Ansätze der Adressierung realisieren. Zum einen gibt es den Ansatz der fixen Adressierung. Hier lassen sich die vergeben Adressen nach einmaliger Vergabe nicht mehr ändern. Ein anderer Ansatz sieht eine variable Adressierung vor. Bei diesem Ansatz lassen sich Adressen nach der Erstvergabe wieder ändern. Dies ließe sich zum Beispiel in der Steuersoftware implementieren, sodass von einem externen Steuergerät die Adressen der Batteriezellencontroller nachträglich geändert werden.

Eine variable Adressierung besitzt zudem den Vorteil, dass sich Batteriezellencontroller beliebig austauschen und umplatzieren lassen, ohne dass die Kommunikation durch eine Änderung der festen Adressen unterbrochen würde.

Der Adressbereich für diese Arbeit beschränkt sich auf 12 Busteilnehmer, den Mastercontroller eingeschlossen. Somit ist die Adressierung innerhalb eines halben Byte möglich:

$$2^4 = 16 \tag{3.1}$$

Es stehen nach Gleichung (3.1) 16 mögliche Adressen zur Verfügung. Diese sind in der folgenden Tabelle zusammengefasst mit ensprechender binärer Codierung.

Modul	**Adresse(n)**	**binäre Codierung**
Mastermodul	0	0000
Batteriezellencontroller 1 bis 12	1 bis 12	0001 bis 1100
Freie Adressen	13 bis 15	1101 bis 1111

Tabelle 3.1: Darstellung der Adressierung der einzelnen Module mit entsprechender binärer Codierung

Der Mastercontroller kann von den Batteriezellencontrollern mit der Adresse 0 angesprochen werden. Die Batteriezellencontroller können mit den binär codierten Adressen 1 ... 12 vom Mastercontroller angesprochen werden. Der Adressbereich 13 ... 15 ist nicht vergeben. Es ist denkbar eine Broadcastadresse zu implementieren, mit der alle Batteriezellencontroller gleichzeitig angesprochen werden können.

3.1.3 Synchronisierung

Für die Datenübertragung zwischen einem Sender und einem Empänger ist es notwendig, dass der Sender und der Empfänger synchronisiert sind. Wenn dies nicht gewährleistet ist, sind Bitfehler in der Übertragung die Folge und eine Auswertung der übertragenen Nachricht ist nicht mehr möglich.

Konkret hier bedeutet das, dass der Master beim Senden eines Datenpakets mit den Batteriezellencontrollern synchronisiert sein muss.

Die Kommunikation mit dem UART-Übertragungsprotokoll erfolgt seriell und asynchron. Das heißt, dass die zu übertragenden Daten nacheinander übertragen werden. Es wird zu Beginn einer Nachricht ein Startbit gesendet, mit dessen Hilfe dem Empfänger mitgeteilt wird, dass ein Datenpaket zu erwarten ist. Das Ende eines Datenpakets wird mit einem Stoppbit angezeigt.

Bei den meisten Übertragungsprotokollen werden neben dem Datenbus keine zusätzlichen Taktleitungen verlegt, mit der der Sender und der Empfänger synchronisiert werden. Es ist also für eine synchronisierte Übertragung der Daten erforderlich, dass der Takt aus dem Datenpaket entnommen werden kann. Das heißt, es muss eine Taktrückgewinnung des Sendetakts erfolgen.

Für die Übertragung des Sendetakts gibt es verschiedene Ansätze. Mit Hilfe eines Datenheaders kann der Takt zurückgewonnen werden. Der Nachteil eines Datenheaders ist, dass dieser bereits zu dem Payload, also den Nutzdaten gehört. Aus diesem Grund wird alternativ eine Präambel verwendet. Die Präambel ist dem Payload vorangestellt, sodass mehr Bits für den Payload zur Verfügung stehen. Die Präambel besteht aus einer fest definierten, wechselhaften Abfolge von Nullen und Einsen, bei deren Flankenwechsel der Takt aus den gesendeten Daten zurückgewonnen werden kann. In der folgenden Abbildung ist eine Präambel dargestellt.

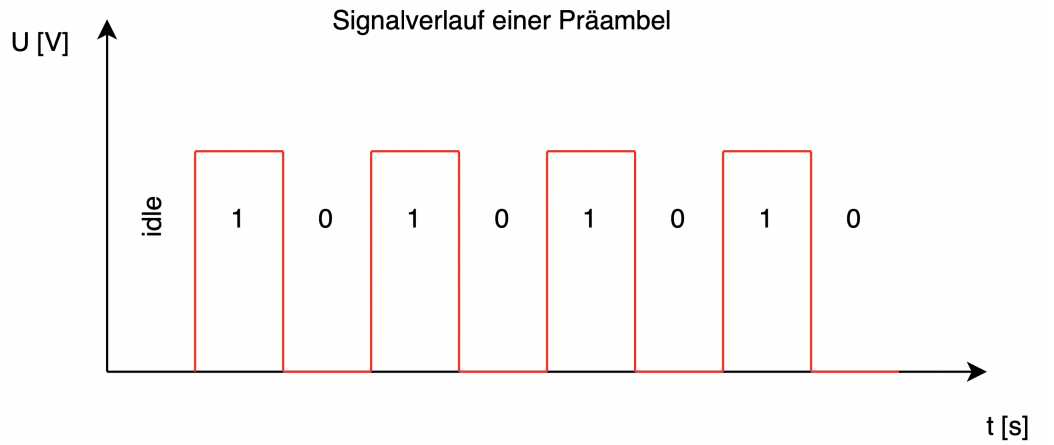


Abbildung 3.1: Signalverlauf einer ein Byte großen Präambel eines Datenpakets zum Einsynchronisieren des Datenempfängers

In der Abbildung ist der Signalverlauf einer Präambel eines Datenpakets auf dem Datenbus mit den entsprechenden Signalpegeln zu erkennen. Mit der Bitfolge 10101010 kann der Empfänger der Nachricht den Sendetakt rekonstruieren, indem er die Flankenwechsel der Bitfolge auswertet. Die Präambel des Datenpakets lässt sich so innerhalb eines Byte übertragen. Nach der Präambel folgen weitere zum Overhead des Datenpakets gehörende Daten. In der Abbildung 3.1 wird von unipolaren Signalpegeln ausgegangen, dies kann aber auf bipolare Signale äquivalent übertragen werden.

3.1.4 Kalibrierung des Systemtakts

Für die Überlegungen der Synchronisation ist es darüber hinaus notwendig, die Taktcharakteristiken der Mikrocontroller zu betrachten. Taktabweichungen zwischen Sender und Empfänger können für eine fehlerhafte Übertragung sorgen, da der Abtastzeitpunkt mit jedem weiteren Bit vor oder hinter den Sendesignalpuls verschoben werden kann. Der XMC4700 Relax Kit verfügt über einen internen RC- Oszillator, der über Fast-Clock-Parameter und Slow-Clock-Parameter verfügt. Dieser Oszillator ist für die CPU und die On-Chip Peripherie verfügbar. Ebenfalls lässt sich eine externe Taktquelle zur Erzeugung des Systemtakts anschließen. Der interne RC-Oszillator weist zum einen eine nominale Frequenz von 36.5 MHz und zum anderen eine geteilte nominale Frequenz von 32.768 kHz [4, S. 82 ff.]. Die Taktfrequenz unterliegt dabei Schwankungen, die durch verschiedene Kalibrierungsansätze und Schwankungen der Versorgungsspannung geschuldet sind. Die Abweichungen sind in den beiden folgenden Tabellen zusammengefasst.

Parameter	Wert			Einheit	Anmerkung
	Min.	Typ.	Max.		
Nominale Frequenz	-	36.5	-	MHz	nicht kalibriert
	-	24	-	MHz	kalibriert
Genauigkeit	-0.5	-	0.5	%	automatische Kalibrierung
	-15	-	15	%	Fabrik-Kalibrierung
	-25	-	25	%	unkalibriert
	-7	-	7	%	Schwankung der Versorgungsspannung 3.13 V Vddp 3.63 V
Start-up Zeit	-	50	-	µs	

Tabelle 3.2: Zusammenfassung des Fast-Internal-Clock Parameter [4, S. 82]

Parameter	Wert			Einheit	Anmerkung
	Min.	Typ.	Max.		
Nominale Frequenz	-	32.768	-	kHz	
Genauigkeit	-4	-	4	%	VBat = const. 0 °C TA 85 °C
	-5	-	5	%	VBat = const. TA 0°C or TA 85 °C
	-5	-	5	%	2.4 V VBat , TA = 25 °C
	-7	-	7	%	1.95 V VBat $<$ 2.4 V, TA = 25 °C
Start-up Zeit	-	50	-	µs	

Tabelle 3.3: Zusammenfassung des Slow-Internal-Clock Parameter [4, S. 83]

Anhand der beiden Tabellen 3.2 und 3.3 wird ersichtlich, dass der Takt stark beeinflussbar durch Temperaturschwankungen und Schwankungen der Versorgungsspannung ist. Aus Tabelle 3.2 lässt sich entnehmen, dass ein unkalibrierter Mikrocontroller eine Taktabweichung von bis zu ±25 Prozent aufweisen kann. Bei Abweichungen der Versorgungsspannung sind Abweichungen von ±7 Prozent möglich. Der Mikrocontroller XMC 2Go verfügt über ähnliche Taktcharakteristiken wie der XMC 4700 Relax Kit [2].

Für die Kalibrierung des internen Oszillators gibt es verschiedene Ansätze. Einer davon ist die Kalibrierung mit Hilfe eines externen Referenztakts. Dabei wird der externe Referenztakt von der Capture Compare Unit 4 (CCU4) des Mikrocontrollers erfasst und mit dem internen Takt verglichen. Bei einer Abweichung wird mit einer Auswertesoftware ein Frequenzteiler auf den internen Takt angewendet. Diese Methode der Kalibrierung ist in der folgenden Abbildung schematisch dargestellt.

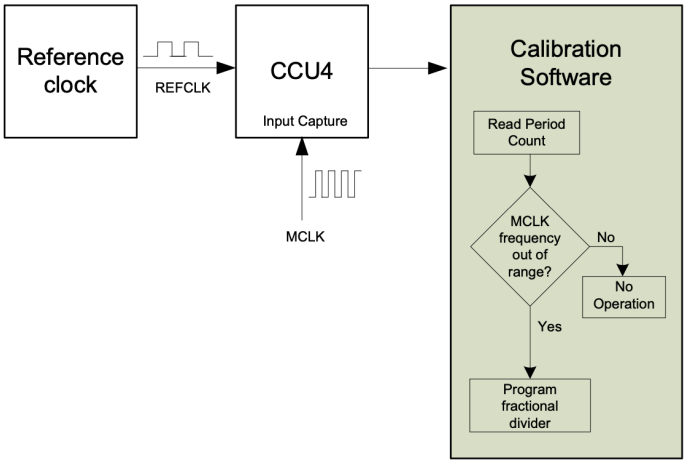


Abbildung 3.2: Schematische Darstellung der Kalibrierung des internen Takts MCLK mit Hilfe einer externen Taktquelle REFCLK [3, S. 3]

In Abbildung 3.3 ist der Ablauf der Kalibrierung des internen Takts des Mikrocontrollers zu erkennen. Die Capture Compare Unit 4 liest den externen eingespeisten Takt ein und eine Auswertesoftware, in Abbildung 3.3 grün hinterlegte Box, vergleicht anschließend die Abweichung des internen mit dem externen Takt. Bei der Abweichung eines bestimmten Bereichs wird ein Frequenzteiler auf den internen Takt angewendet, bis der interne Takt innerhalb einer bestimmten Toleranz ist.

3.1.5 Übertragungsprotokoll

Als Übertragungsprotokoll wird ein serielles Protokoll gewählt. Die Batteriezellencontroller kommunizieren mit dem Batteriemodulcontroller mit Hilfe der Universal-Interface-Channel (USIC) der Mikrocontroller. Dieses Interface ist als Universal-Asynchronous-Receive-Transmit (UART) mit der 8E1 Notation konfiguriert. Die 8 gibt die Anzahl der Datenbits an, E gibt an, dass es sich um eine Even-Parity, also gerade Parität handelt und die 1 gibt ein Stoppbit an.

Im Rahmen der Vorarbeit von Jonas Ernsting ist ein Übertragungsprotokoll für die Kommunikation mittels IrDA-Transceivern entwickelt worden [8, S. 17 ff.]. Grundlage ist eine UART-Standardübertragung, bei dem das Sendesignalpegel invertiert und die Bitdauer gekürzt ist. Dieses Protokoll sieht jeweils ein Start- und Stoppbit vor, 4 Bit für Daten beziehungsweise ein Kommando, 4 Bit für die Empfangsadresse und ein Paritätsbit.

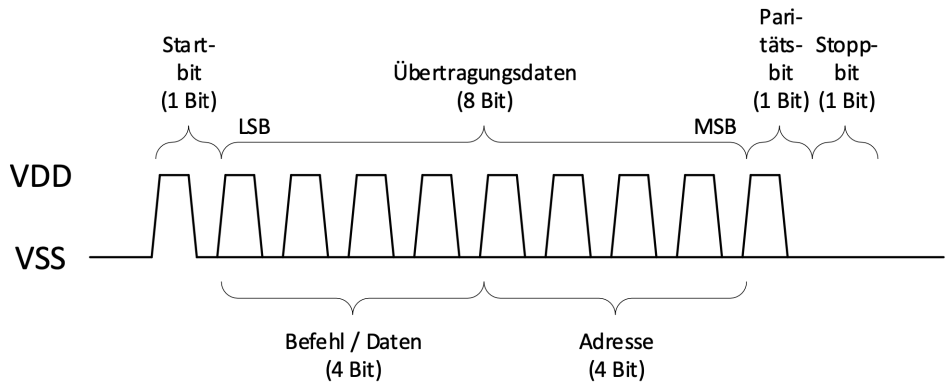


Abbildung 3.3: Paketaufbau und Signalverlauf der UART-Übertragung in Anlehnung an eine IrDA-SIR-Spezifikation nach Jonas Ernsting [8, S. 20]

Untersuchungen haben ergeben, dass eine Datenübertragung fehlerfrei bis zu einer Taktrate von

$f_{MAX} = 162006\ Bit/s$

möglich ist [8, S. 23].

Eine weitere Untersuchung hat ergeben, dass die Anhebung der Datenrate zur Reduzierung des Einflusses der Taktabweichung nicht effizient ist [8, S. 24]. Eine bessere Variante ist die direkte Reduzierung der Taktabweichung, die mit der in Abschnitt 3.1.3

beschriebenen Methode zu Kalibrierung des internen Oszillators des Mikrocontrollers vorgenommen werden kann.

Das von Jonas Ernsting entwickelte Übertragungsprotokoll soll an das von Nico Sassano adaptiert werden. Es ist im Rahmen seiner Bachelorthesis ein Übertragungsprotokoll entwickelt worden, dass für die Übertragung mittels Radiowellen konzipiert wurde [10, S. 37 ff.]. Die Besonderheit bei dem Protokoll ist, dass die Nutzdatenlänge dynamisch je nach zu sendenden Daten variiert, sodass es keine fest definierte Länge der Nutzdaten gibt.

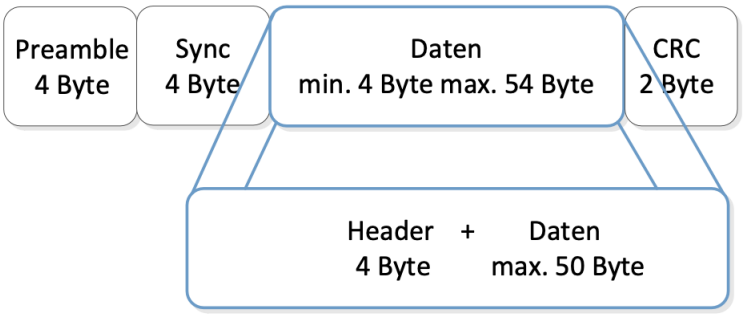


Abbildung 3.4: Paketaufbau einer Datenübertragung nach Nico Sassano [10, S. 38]

Abbildung 3.5: Header einer Uplink-Übertragung nach Nico Sassano [10, S. 39]

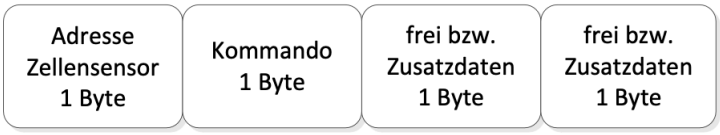


Abbildung 3.6: Header einer Downlink-Übertragung nach Nico Sassano [10, S. 39]

Das Übertragungsprotoll von Nico Sassano sieht eine Paketgröße von 64 Byte vor. Diese ist die maximale Paketgröße, die von seiner verwendeten Hardware gesendet werden kann. Das Paket besteht dabei aus einer 4 Byte großen Präambel, einem 4 Byte großen

Synchronisierer, einem 4 Byte großen Header, 50 Byte Nutzdaten und einem 2 Byte großen CRC Prüffeld, wie in Abbildung 3.4 dargestellt.

Der Header des Datenpakets ist dabei davon abhängig, ob ein Paket von einem Batteriezellencontroller zum Batteriemodulcontroller oder entgegengesetzt gesendet wird. In Abbildung 3.5 ist der Header einer Uplink-Datenübertragung zu erkennen und bedeutet, dass ein Batteriezellencontroller an den Batteriemodulcontroller sendet. Dabei ist ein Byte für die Adresse des Batteriemodulcontrollers, ein Byte für die Absenderadresse des Batteriezellencontrollers und 2 Byte Reserve vorgesehen. In Abbildung 3.6 ist der Header für die Downlink-Datenübertragung abgebildet. Hier ist ein Byte für die Zieladresse des Batteriezellencontrollers, ein Byte des verschickten Kommandos und zwei Byte Reserve vorgesehen. Der Overhead ist definiert als alle Daten eines Datenpakets, die nicht zu den Nutzdaten gehören. Das können beispielsweise Daten für Fehlererkennung, Adressierung oder Synchronisierung sein. Der Overhead bei diesem Übertragungsprotokoll beträgt konstant 14 Byte bei Uplink- und Downlink-Übertragung.

Adaption der Übertragungsprotokolle

Für die Adaption des Übertragungsprotokolls soll zunächst abgeschätzt werden, welcher Overhead für das Datenpaket zu erwarten ist. Dazu werden die Sensoren aus der Tabelle [1] betrachtet. Die Sensoren sind in der folgenden Grafik zusammenfassend dargestellt.

Sensor	Sensortyp	Anbringung	Messrate	Schnittstelle	I2C Adressen	Messsignal	Bit-Auflösung
Temperatursensor	Hallleitersensor (AS6204)	Sensoren in der Zelle	0,25 - 10,7 Conv/s	I2C (4 Adressen); GPIO Alert nicht vorgesehen	1001000 (0x48) 1001001 (0x49) 1001010 (0x4A) 1001011 (0x4B)	digital	12
Optischer Sensor (HAW)	Glasfasersensor	Sensoren in der Zelle	1 Hz	Seriell, UART	n/a	analog ausgewertet, dig. Schnittstelle	10
–	–	Sensoren an der Zelle	5 Perioden je Freq.Linie (5 Hz - 5 kHz)	ISO-UART (2MBit/s)per Jumper zu PC oder µC	n/a	digital	12-16
Spannungssensor	VDC	Sensoren an der Zelle	20 Hz	ISO-UART (2MBit/s)per Jumper zu PC oder µC	n/a	digital	14-16
Temperatursensor	Temp on Chip	Sensoren an der Zelle	10 Hz	ISO-UART (2MBit/s)per Jumper zu PC oder µC	n/a	digital	9-11
Spannungssensor	HAW- VAC (mit OPs)	Sensoren an der Zelle	2-10 Perioden je Freq.Linie (0,1 Hz - 10 kHz)	SPI + 2x CS, DAC, ADC	n/a	0 V - 3,3 V	12
Spannungssensor	HAW- VDC	Sensoren an der Zelle	variabel bis 10 kHz	ADC	n/a	0 V - 3,3 V	12
–	HAW- Hallsensor (ACS716)	Sensoren an der Zelle	120 kHz	ADC	n/a	0 V - 3,3 V	12
Zellinnendruck und Zellausdehnung	Drucksensor TPMS (basierend auf SP37/40)	Sensoren an der Zelle	0.5 Hz	I2C (4 Adressen); GPIO Alert nicht vorgesehen	0110110 (0x36)	analog ausgewertet, dig. Schnittstelle	7-8
Strom- und Spannungsmessensorik	Strommess-IC	Sensoren an der Zelle	1 0 kHz RC Tiefpass (änderbar)	ISO-UART + 2x ADC (Vl u. NTC) + I2C zu EEPROM + 1x GPIO (Gain)	1010000 (0x50)	0,5 V - 4,5 V	16

Abbildung 3.7: Tabellarische Darstellung ausgewählter Sensoren zur Abschätzung des Overheads des adaptierten Übertragungsprotokolls [1].

In der Abbildung 3.7 ist eine Auswahl an Sensoren tabellarisch dargestellt. Mit - gekennzeichnete Zeilen sind dabei anonymisiert. Es handelt sich hierbei um Sensoren, die auf einer Multisensorzelle angebracht werden sollen. Die Sensoren haben diverse Aufgaben wie Spannungs- und Strommessung, Temperaturmessung und Druckmessung. Die Sensoren sind dabei sowohl aus analog- als auch digitaler Technik aufgebaut. Zudem sind einige dieser Sensoren mit einer festen 7 Bit Adresse adressierbar. Andere wiederum verzichten auf eine Adresse. Darüber hinaus ist bei einigen Sensoren eine Kalibrierung notwendig. Auffällig ist, dass die Sensoren mit einer geringen Messfrequenz arbeiten. Eine Ausnahme bilden die Sensoren für die Spannungs- und Strommessung, die mit einer deutlich höheren Messfrequenz betrieben werden.

Für die Abschätzung des Overheads, wird nun die effektive Nutzdatengröße der in der Tabelle 3.7 genannten Sensoren betrachtet. Dazu wird zum einen die minimale effektive Nutzdatengröße und die maximale effektive Nutzdatengröße betrachtet. Es ergibt sich eine Nutzdatengröße von mindesten 116 Bit und maximal 125 Bit. Die Nutzdaten sind somit in 16 Byte darstellbar.

In dem Übertragungsprotokoll der Vorarbeit [10], ist eine CRC-16 Überprüfung implementiert. Mit dieser Fehlererkennung lassen sich Mehrfach- und Bündelfehler bis zu einer Länge von 16 Bit eindeutig erkennen. Für den Overhead bedeutet das, dass 2 Byte für die CRC Prüfung als Trailer an die Nutzdaten innerhalb des Datenpakets angehängt werden. Es ist zu untersuchen, ob eine Einzelfehlererkennung und -korrektur mittels Hamming-Code effizient ist oder, ob bei einer Einzelfehlererkennung die Daten verworfen werden und neu angefragt werden. Die Einzelfehlererkennung würde zusätzlichen Overhead generieren und die Ressourcen des Mikrocontrollers belasten.

Darüber hinaus ist zu untersuchen, ob eine 1 Byte große Präambel ausreichend ist, um Sender und Empfänger bei der Datenübertragung mit SIR IrDA zu synchronisieren. Ein Wichtiger Aspekt dabei ist die Messsynchronisierung. In der Übersicht 3.7 ist angegeben mit welcher Messfrequenz die Sensoren operieren. Hier ist darauf zu achten, dass die Messung und die Auswertung der Daten synchron sind. Dazu müssen die Übertragung zwischen externem Computer zum Batteriemodulcontroller und die Übertragung vom Batteriemodulcontroller zum Batteriezellencontroller betrachtet werden.

Bei der Zusammensetzung des Datenpakets ist es sinnvoll, das Datenpaket bei Uplink- und Downlink-Übertragung unterschiedlich aufzubauen. Bei der Downlink-Übertagung vom Batteriemodulcontroller zum Batteriezellencontroller ist es notwendig, dass eine Anweisung übertragen wird. Dies ist bei der Uplink-Sendung nicht erforderlich, da die

Antwort im Nutzdatenblock zurückgegeben wird, sodass das Kommandobyte für eine andere Funktionalität verwendet werden kann. Folgender Paketaufbau ist möglich:

Abbildung 3.8: Paketaufbau des adaptierten Übertragungsprotokolls

Abbildung 3.9: Header einer Uplink-Datenübertragung des adaptierten Übertragungsprotokolls

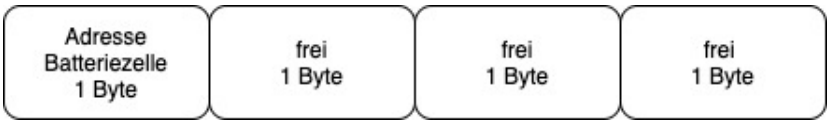


Abbildung 3.10: Header einer Downlink-Datenübertragung des adaptierten Übertragungsprotokolls

In Abbildung 3.8 ist der Paketaufbau des adaptierten Übertragungsprotokolls zu erkennen. Der Overhead des Datenpakets beträgt insgesamt 7 Byte, was geringer ist als bei der Vorarbeit [10] mit 14 Byte Overhead. Darüber hinaus ist der Header bei der Uplink- und Downlink-Übertragung unterschiedlich. Bei der Downlink-Übertragung wird neben der Zellencontrolleradresse zusätzlich ein Kommando übergeben. Für die Uplink-Übertragung ist das nicht notwendig, sodass das Byte für eine andere Funktion genutzt werden kann.

Das Übertragen der Zellencontroller-Adresse bei der Uplink-Übertragung hat den Vorteil, dass bei einem beispielhaften Broadcast-Befehl des Batteriemodulcontrollers zugeordnet werden kann, welcher Batteriezellencontroller geantwortet hat. Das ist somit eine extra Information, da bei einem Broadcast in festen Zeitfenstern für einen Zellencontrollern geantwortet werden sollte.

Mit einem 1 Byte großen Zellenadressblock, lassen sich insgesamt 256 verschiedene Batteriezellencontroller adressieren. Betrachtet man das Beispiel, welches auch in dem Einführungsabschnitt aufgeführt wird, so ist das eine für die Praxis ausreichende Größe.

Bei dem Beispiel sind 96 Batteriezellen in insgesamt 8 Batteriemodulen verbaut. Auf ein Batteriemodul skaliert, ergeben sich 12 Batteriezellen [9].

Mit einem ein Byte großen Kommandoblock lassen sich außerdem 256 verschiedene Kommandos und Anweisungen implementieren. Es ist denkbar, dass bei einer Antwort eines Batteriezellencontrollers, das gesendete Kommando als Quittierung mit in den Header der Uplink-Übertragung gelegt wird.

3.2 Konzeption des Funktionsdemonstrators

Der Funktionsdemonstrator wird auf der Basis der Vorarbeit von Jonas Ernsting weiterentwickelt [8]. Dieser soll für diese Arbeit als zentrales Anschauungsobjekt fungieren. Dabei stellt der Funktionsdemonstrator ein Batteriezellenmodul mit insgesamt 12 Batteriezellen und einem Batteriemodulcontroller dar.

Die Batteriezellen werden mit den Batteriezellencontrollern realisiert. Ebenfalls befindet sich der Batteriemodulcontroller auf dem Funktionsdemonstrator, der aus dem XMC 4700 Relax Kit und dem dazugehörigen Überwachungsmodul besteht. Auf den Batteriezellencontrollern wird der Lichtleitkörper installiert, um die Batteriezellencontroller und den Batteriemodulcontroller optisch miteinander zu verbinden.

Die zentrale Spannungsversorgungseinheit wird mit zwei USB-Hubs realisiert, die die Spannung an die Batteriezellencontroller verteilen. Der Modulcontroller wird über die USB-Debugschnittstelle mit Spannung versorgt, kann jedoch bei Bedarf ebenfalls über eines der USB-Hubs versorgt werden.

Das Grundgerüst bildet eine Holzplatte, auf der die Komponenten im Verlauf der Konstruktion angebracht werden. In der nachfolgenden Abbildung ist der Aufbau des Funktionsdemonstrators schematisch dargestellt.

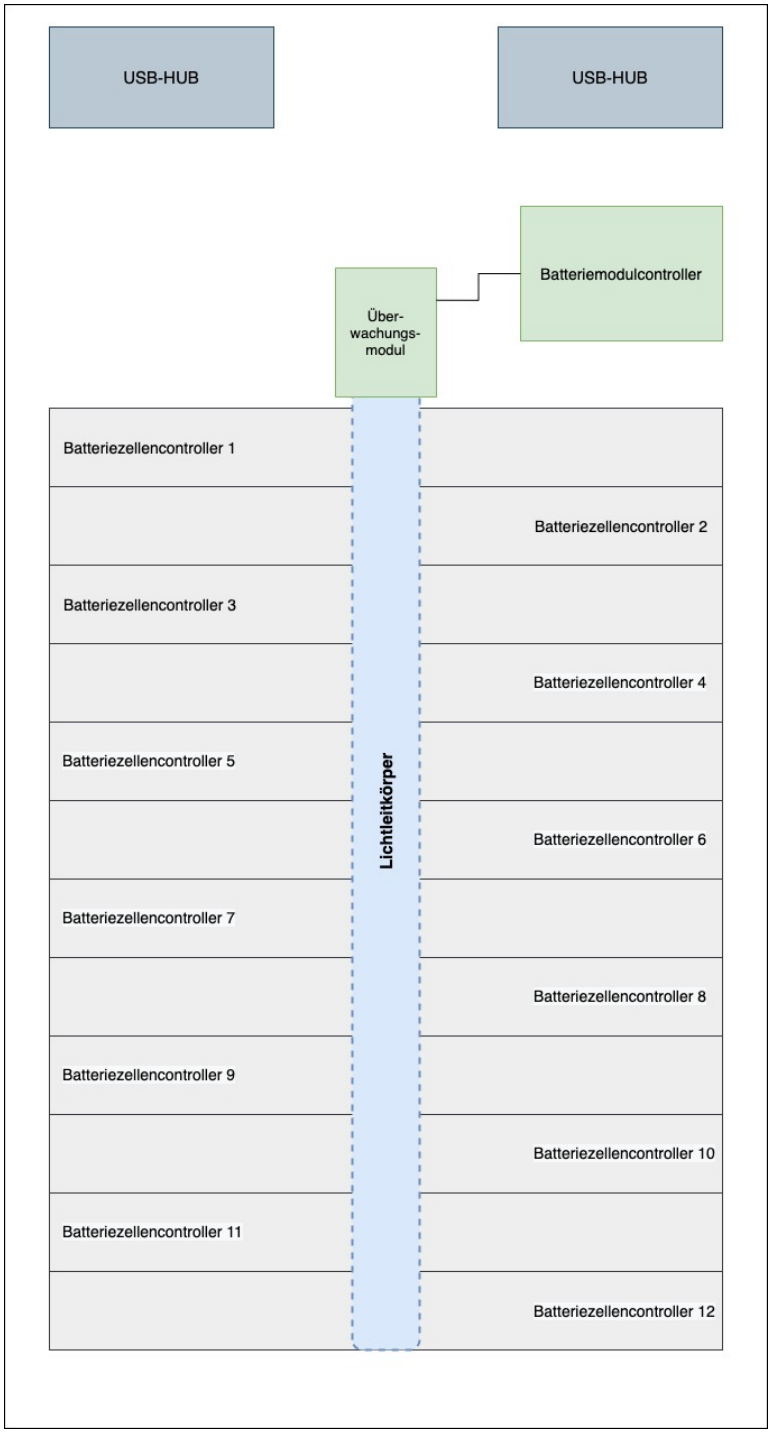


Abbildung 3.11: Schematischer Aufbau des Funktionsdemonstrators mit Anordnung der Komponenten

In Abbildung 3.11 ist der schematische Aufbau des Funktionsdemonstrators abgebildet. Die Komponenten sind so angeordnet, wie bei der späteren Installation. Es ist zu erkennen, dass der Lichtleitkörper (in Abbildung 3.11 blau dargestellt) über den Batteriezellencontrollern (in Abbildung 3.11 grau dargestellt) platziert wird. Das Überwachungsmodul des Batteriemodulcontrollers wird in 90° Versatz zu den Batteriezellencontrollern angebracht (in Abbildung 3.11 beides grün dargestellt). Die Batteriezellencontroller sind dabei abwechselnd zueinander angeordnet. Das Kabelmanagementsystem wird auf der Unterseite der Holzplatte angebracht. Dieses System regelt die Verteilung der Spannungsversorgung der Batteriezellencontroller. Die USB-Hubs (in Abbildung 3.11 dunkelgrau dargestellt) werden an der Oberseite des Aufbaus platziert.

Um die Batteriezellencontroller und das Überwachungsmodul besser auf der Holzplatte installieren zu können, wird eine Installationshilfe entwickelt. Die Installationshilfe wird später mit einem 3D- Drucker hergestellt. Diese hilft dabei, die einzelnen Platinen besser fixieren und bei Bedarf austauschen zu können. Die Installationshilfen sind in den beiden unteren Abbildungen dargestellt.

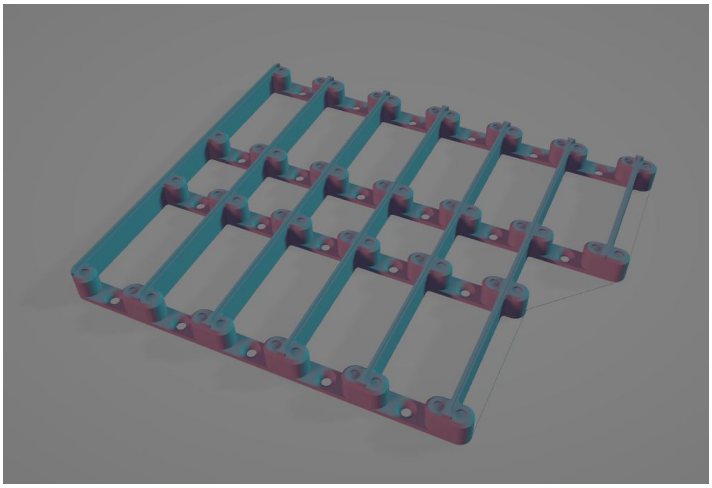


Abbildung 3.12: 3D-Ansicht der Installationshilfe für die Batteriezellencontroller

In Abbildung 3.12 ist das 3D-Modell der Installationshilfe für die Batteriezellencontroller zu erkennen. Es werden insgesamt zwei Stück nebeneinander angeordnet, um 12 Batteriezellencontroller auf der Holzplatte zu befestigen. In Abbildung 3.13 ist das 3D-Modell der Installationshilfe für das Überwachungsmodul dargestellt. Dieses wird, wie in Abbildung 3.11 zu erkennen, an der Stelle des Überwachungsmoduls auf der Holzplatte angebracht.

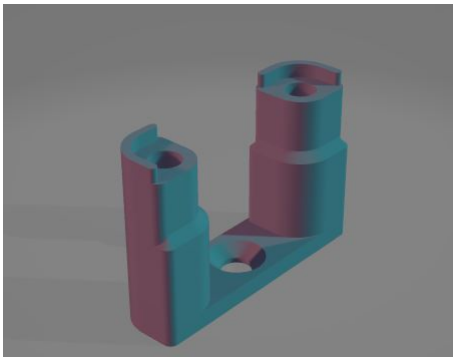


Abbildung 3.13: 3D-Ansicht der Installationshilfe für das Überwachugsmodul des Batteriemodulcontrollers

Auf der Unterseite der Holzplatte werden darüber hinaus Abstandsrollen angebracht, um die Holzplatte von der Abstellfläche zu erhöhen.

4 Entwicklung und Implementierung

4.1 Konstruktion des Funktionsdemonstrators

Der Funktionsdemonstrator wird gemäß der schematischen Darstellung 3.11 in Abschnitt 3.2 nachgebaut. Als Bodenplatte wird eine 20 mm dicke, weiß lackierte Sperrholzplatte verwendet. Diese Holzplatte wird auf ein Maß von 510 mm x 280 mm zugeschnitten. Die Installationshilfen der Controller werden mit einem 3D-Drucker produziert und anschließend auf der Holzplatte befestigt. An den Stellen, wo die spannungsversorgenden USB-Leitungen durch die Holzplatte gezogen werden, werden Bohrungen von 10 mm gefertigt. Diese Bohrungen befinden sich an den USB-Hubs und direkt vor den Batteriezellencontrollern. Der Konstruktionsfortschritt ist in der folgenden Abbildung zu erkennen.

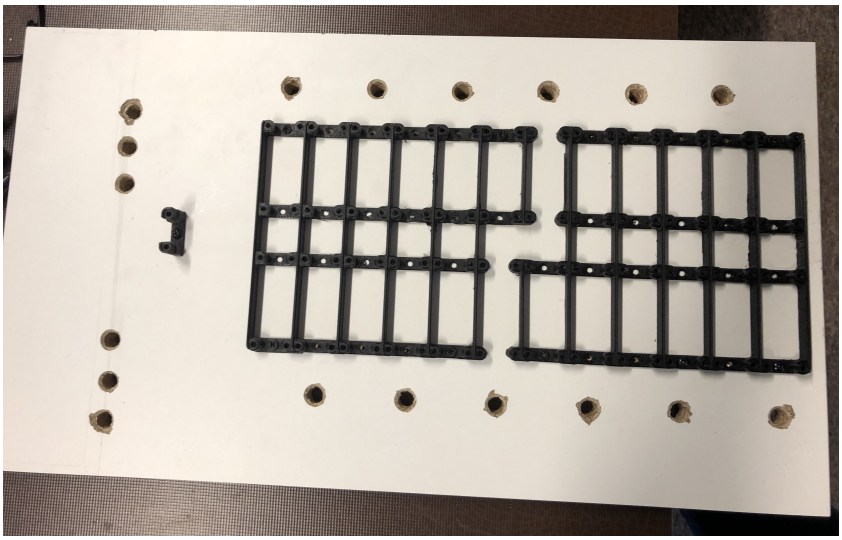


Abbildung 4.1: Zugeschnittene Holzplatte mit angebrachter Installallationshilfe für die Batteriezellencontroller des Überwachungsmoduls und Bohrungen für die USB-Leitungen

Als Nächstes werden die zwei USB-Hubs an der Holzplatte angebracht. Danach werden die Batteriezellencontrollern auf der Installationshilfe platziert und mit Schrauben befestigt. Nun können die USB-Leitungen auf der Unterseite der Holzplatte angebracht und an den Batteriezellencontrollern angeschlossen werden. Um die Holzplatte von einer Ablagefläche zu erhöhen, werden an der Unterseite Abstandhalter befestigt. Daraufhin werden das Überwachungsmodul und der Batteriezellencontroller auf der Installationshilfe und auf dem Holzbrett verschraubt. Zum Schluss wird der Lichtleitkörper über den Batteriezellencontrollern platziert und befestigt. Der fertige Aufbau ist in der nachfolgenden Abbildung dargestellt.

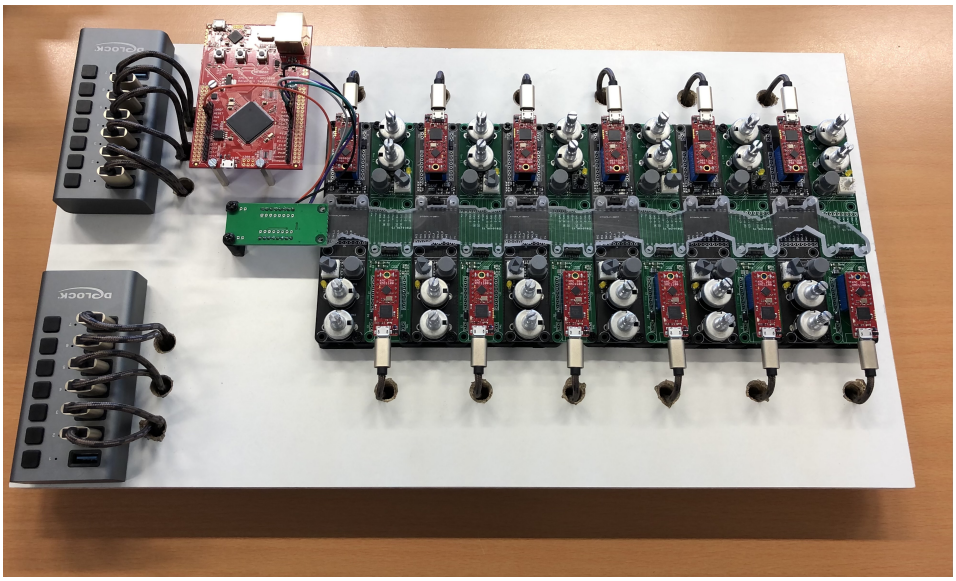


Abbildung 4.2: Fertiggestellter Aufbau des Funktionsdemonstrators

In der Abbildung sind links auf dem Brett die beiden USB-Hubs erkennbar. Der rote Controller daneben ist der Batteriemodulcontroller, der mit dem darunterliegenden Überwachungsmodul verbunden ist. Rechts daneben befinden sich die Batteriezellencontroller mit dem daraufliegenden Lichtleitkörper mit ausgerichteten Reflektorflächen.

4.2 Controllersoftware

In diesem Abschnitt wird auf die Entwicklung der Batteriemodulcontroller- und Batteriezellencontrollersoftware, sowie auf die Entwicklung der PC-Software zur Steuerung und Visualisierung eingegangen.

4.2.1 Software des Batteriezellen- und Batteriemodulcontrollers

Die Controllersoftware wird auf Basis der Vorarbeit weiterentwickelt. Die Software des Batteriemodulcontrollers soll um eine UDP-Schnittstelle erweitert werden, damit der Batteriemodulcontroller von einem Computer gesteuert und die gesendeten Daten des Batteriemodulcontrollers ausgewertet werden können. Die Quellcodedateien befinden sich im Anhang, sowie auf der CD.

Darüber hinaus soll die Software des Batteriemodulcontrollers und die der Batteriezellencontroller aus Gründen der Übersichtlichkeit in mehrere C-Quell- und Headerdateien aufgeteilt werden, da sich der Quellcode der Software zum Abschlusspunkt der Vorarbeit noch in einer Datei befindet.

Für die Programmierung wird die Entwicklungsumgebung DAVE von Infineon Technologies AG verwendet. Diese verfügt über eine komponentenbasierte Programmierung, was das Hinzufügen bestimmter Funktionen der Controllerhardware vereinfacht. Zudem verfügt die Entwicklungsumgebung über eine grafische Oberfläche der DAVE-APPs, mit der automatisch der Quellcode für die Initialisierung der Hardware generiert werden kann. Die Quellcode-Dateien und die Konfiguration der Entwicklungsumgebung befinden sich im Anhang.

Für die Übertragung der Daten mittels UDP-Verbindung werden verschiedene Befehle implementiert, die von einem externen Computer gesendet werden können. Diese sind in der folgenden Tabelle dargestellt.

Befehl/Antwort	Zeichenkette	Anmerkung
Sende Daten eines spezifischen Batteriezellencontroller	0101,No,0	Dieser Befehl sendet alle Daten eines ausgewählten Batteriezellencontrollers über die UDP-Schnittstelle
Sende Daten aller verbundenen Batteriezellencontroller	0102,0	Dieser Befehl sendet alle Daten aller verbundenen Batteriezellencontroller über die UDP-Schnittstelle
Sende alle Daten	0103,0	Dieser Befehl sendet alle Daten aller Batteriezellencontroller, unabhängig davon ob sie verbunden sind, über die UDP-Schnittstelle
Sende Adressen aller verbundenen Batteriezellencontroller	0104,0	Dieser Befehl sendet die Adressen aller verbundenen Batteriezellencontroller über die UDP-Schnittstelle
Sende Antwort: Unbekannter Befehl	UNKNOWN CMD	Wenn kein dem Batteriemodulcontroller bekannter Befehl übergeben wurde, wird mit UNKNOWN CMD geantwortet

Tabelle 4.1: Übersicht der implementierten Befehle, die von einem externen Computeran den Batteriemodulcontroller gesendet werden können

Die Befehle sind immer gleich aufgebaut. Der Befehl beginnt mit einem 4 Zeichen langen Befehl-Indentifier, gefolgt von einem Nullterminator. Beim ersten Befehl besteht die Besonderheit, dass dem Batteriemodulcontroller die Nummer des anzusprechenden Batteriezellencontrollers mit übergeben wird.

Der erste Befehl in Tabelle 4.1 fordert den Batteriemodulcontroller auf, die Daten eines bestimmten Batteriezellencontrollers an die UDP-Schnittstelle zu übergeben, an der ein externer Computer zur Auswertung der Daten angebunden ist.

Der zweite Befehl in Tabelle 4.1 fordert den Batteriemodulcontroller auf, die Daten aller Batteriezellencontroller einzuholen, die mit dem Batteriemodulcontroller verbunden sind.

Der dritte Befehl in Tabelle 4.1 fordert den Batteriemodulcontroller auf, alle verfügbaren Daten der Batteriezellencontroller einzuholen und an die UDP-Schnittstelle zu übergeben.

Der letzte Befehl in Tabelle 4.1 fordert den Batteriemodulcontroller auf, die Adressen aller verbundenen Batteriezellencontroller an die UDP-Schnittstelle zu übergeben.

Der Programmablaufplan wird gemäß der erweiterten Funktionalität bezüglich einer UDP-Schnittstelle angepasst und ist in den folgenden Grafiken dargestellt.

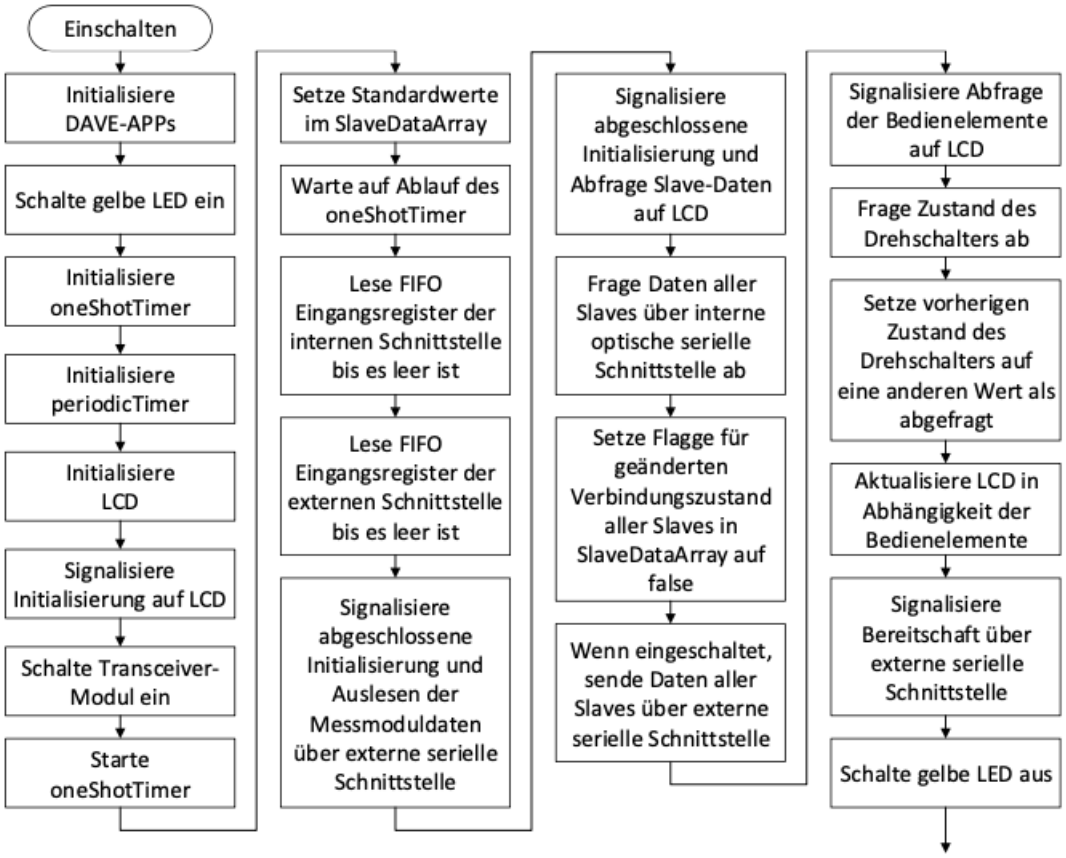


Abbildung 4.3: Vereinfachtes Flussdiagramm des Hauptprogrammablaufs im Überwachungsmodul als Master, Teil 1 [8, S. 92]

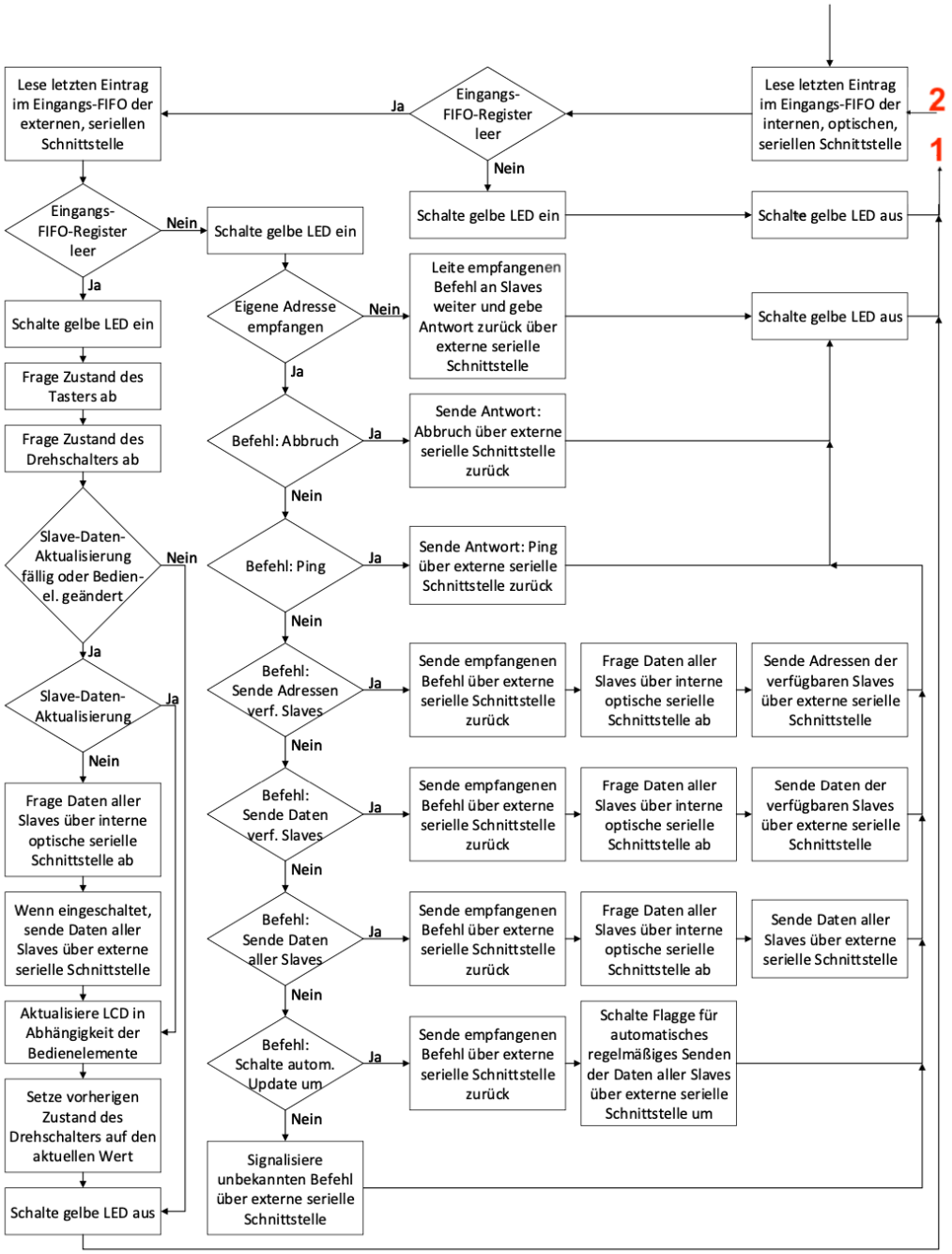


Abbildung 4.4: Vereinfachtes Flussdiagramm des Hauptprogrammablaufs im Überwachungsmodul als Master, Teil 2. Auf der rechten Seite, mit den roten Zahlen 1 und 2 gekennzeichnet, befindet sich der Einschub für die Erweiterung der UDP-Funktionalität, modifiziert nach [8, S. 93]

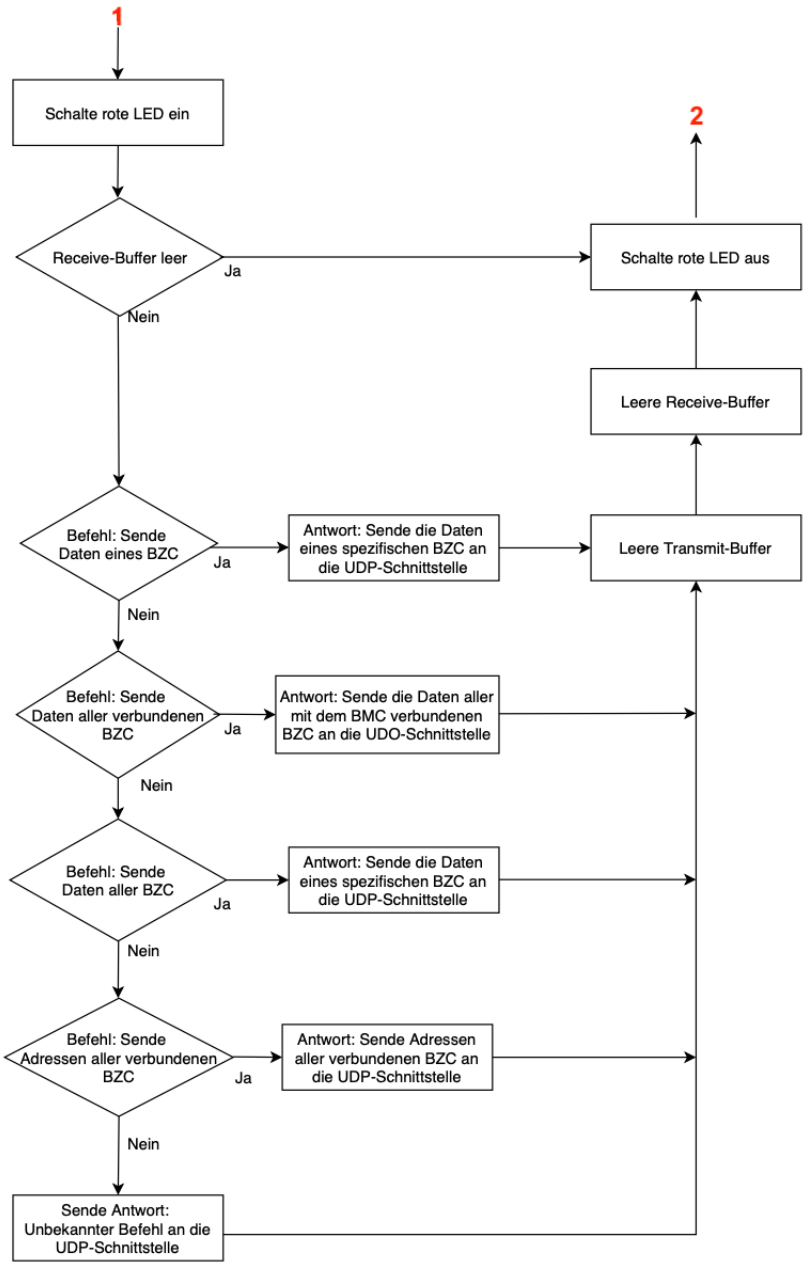


Abbildung 4.5: Vereinfachtes Flussdiagramm des Hauptprogrammablaufs im Überwachungsmodul als Master, Teil 3 mit Erweiterung um eine UDP-Schnittstelle. Die Rote 1 markiert den Beginn der Programmsequenz, die Rote 2 markiert das Ende der Programmsequenz mit Rückkehr zum Programmablauf in Abbildung4.4. Die Abkürzung BMC steht für Batteriemodulcontroller, die Abkürzung BZC steht für Batteriezellencontroller

In Abbildung 4.5 ist der vereinfachte Programmablauf der Erweiterung um die UDP-Schnittstelle dargestellt. Zu Beginn wird eine rote Indikator-LED eingeschaltet. Daraufhin wird überprüft, ob der Receive-Buffer eine Nachricht enthält. Ist das nicht der Fall, wird die Indikator-LED wieder ausgeschaltet und die Programmschleife verlassen.

Befindet sich jedoch eine Nachricht im Receive-Buffer, so wird die Empfangene Nachricht auf ein übertragenes Kommando ausgewertet. Der Batteriemodulcontroller übergibt je nach erhaltenem Kommando die entsprechende Antwort an die UDP-Schnittstelle. Wenn kein bekanntes Kommando übermittelt wurde, antwortet der Batteriemodulcontroller mit einem *UNKNOWN CMD*.

Nach der Übergabe der Antwort an die UDP-Schnittstelle, wird der Transmit-Buffer wieder geleert und ebenso der Receive-Buffer. Anschließend wird die Indikator LED ausgeschaltet und die Programmschleife verlassen.

4.3 PC Software zur Steuerung und Visualisierung

Im Rahmen dieser Thesis soll eine Computer Software entwickelt werden, die über eine UDP-Schnittstelle mit dem Batteriemodulcontroller verbunden ist und die übertragenen Daten des Batteriemodulcontrollers auswertet und visualisiert, sowie den Batteriemodulcontroller mit ausgewählten Befehlen steuern kann. Die Software wird innerhalb von Matlab von The MathWorks Inc. Entwickelt. Die Quellcodedatei befindet sich im Anhang, sowie auf der CD.

Die Software soll aus Gründen der einfachen Bedienbarkeit, eine Benutzergeführte Eingabe mit Hilfe des Kommandofensters innerhalb von Matlab erhalten. Die Ausgabe der Daten soll tabellarisch und wahlweise zyklisch oder sequenziell erfolgen. Darüber hinaus soll es dem Benutzer der Software möglich sein, die zu sendenden Steuerbefehle mit einer Konsoleneingabe auszuwählen.

Die Befehle, die dabei implementiert werden sollen, sind in Tabelle 4.1 dargestellt. Der Quellcode befindet sich im Anhang.

In der nachfolgenden Grafik ist der Programmablauf dargestellt.

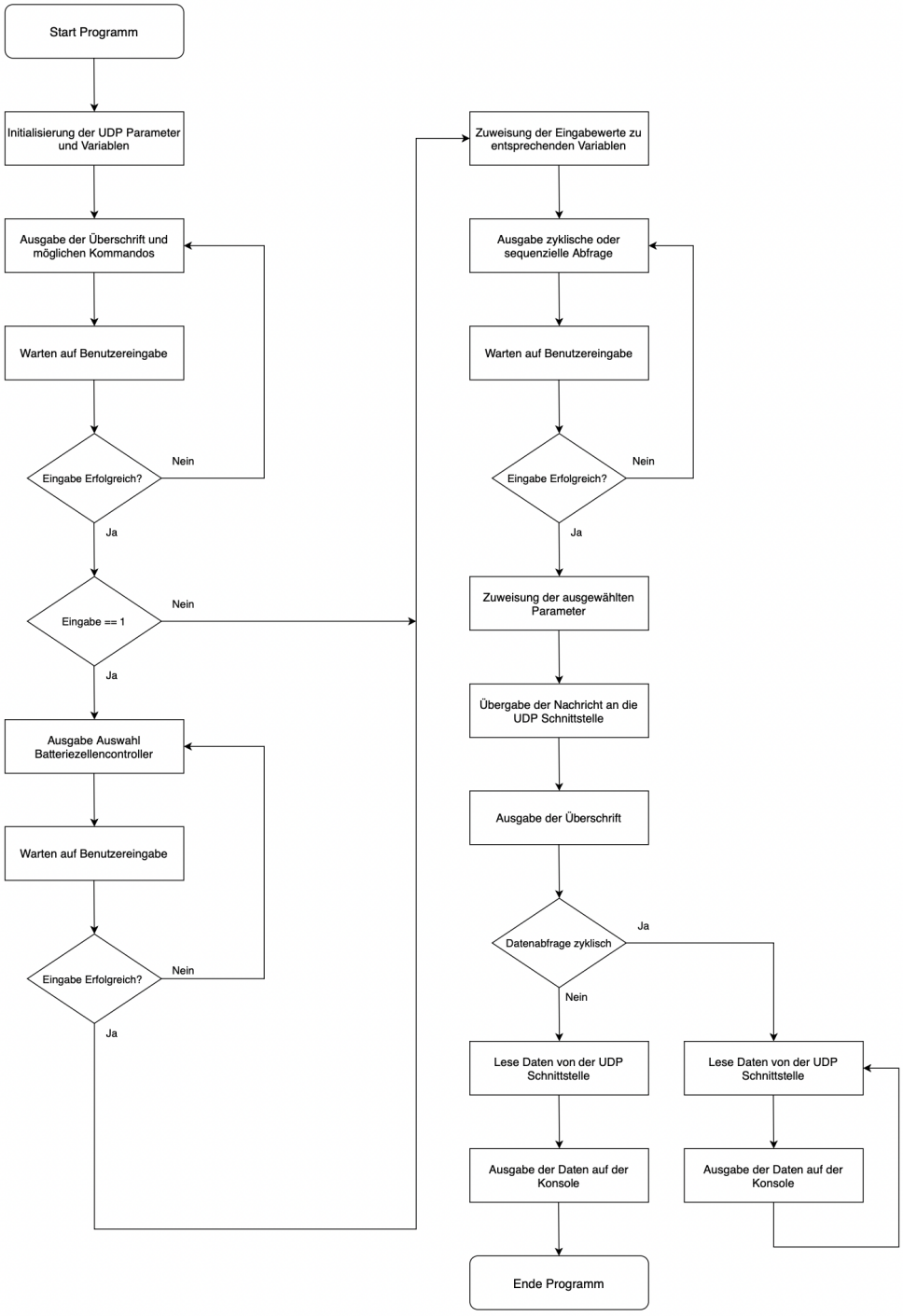


Abbildung 4.6: Vereinfachtes Flussdiagramm der PC Software zur Visualisierung und Steuerung des Batteriemodulcontrollers

Nach dem Start der Software, werden zunächst die Parameter für die UDP-Verbindung initialisiert, sowie die benötigten Hilfsvariablen. Daraufhin wird eine Überschrift ausgegeben, die zudem die verfügbaren Kommandos ausgibt. Es erfolgt danach eine benutzergeführte Eingabe, die außerdem fehlergesichert ist.

Wenn das Kommando für einen spezifischen Batteriezellencontroller ausgewählt wird, wird anschließend abgefragt, welcher Batteriezellencontroller angesteuert werden soll.

Als nächstes kann ausgewählt werden, ob die Abfrage der Daten zyklisch, oder sequenziell erfolgen soll. Nach der Benutzerabfrage werden die eingegeben Informationen in den jeweiligen Hilfsvariablen gespeichert.

Nun wird das Kommando an die UDP-Schnittstelle übergeben und anschließend die Daten abgefragt. Die Software lässt sich mit der Tastenkombination STRG+C beenden.

5 Test und Erprobung

5.1 Funktionstest

5.1.1 Software Batteriezellencontroller

Die Software der Batteriezellencontroller wird umfangreich auf Ihre Funktionalität geprüft. Dazu werden diverse Tests durchgeführt.

Zunächst wird überprüft, dass die Gelbe LED, die zur Indikation der Aktivität dient, funktioniert. In diesem Zusammenhang wird ebenfalls überprüft, dass die Rote LED, ebenfalls leuchtet. Damit wird festgestellt, dass die Batteriezellencontroller die eigene Adresse korrekt erkennen und den Datenverkehr ermöglichen. Darüber hinaus wird getestet, ob die Änderung der Adresse mittels des Kodierschalters und des Tasters funktioniert.

Daraufhin wird überprüft, dass der Controller die Werte der zwei Potentiometer korrekt einliest und speichert.

Für weitere Tests muss das der gesamte Aufbau in Betrieb genommen werden. Im Anhang befindet sich ein Prüfprotokoll zu dem Test der Batteriezellencontroller.

5.1.2 Software Batteriemodulcontroller

Wie die Software des Batteriezellencontrollers wird ebenfalls die Software des Batteriemodulcontrollers umfangreich getestet.

Zunächst wird überprüft, ob der Batteriemodulcontroller über die UART-Schnittstelle mit einem Computer kommuniziert. Dazu werden die vordefinierten Kommandos mit Hilfe der Software HyperTerminal an den Batteriemodulcontroller übergeben.

Für weitere Test muss der gesamte Aufbau in Betrieb genommen werden. Das Prüfprotokoll befindet sich im Anhang.

5.1.3 PC Software zur Steuerung und Visualisierung

So wie in den beiden vorigen Abschnitten wird ebenfalls die Software zur Steuerung und Visualisierung umgehend getestet. Zu dem Test gehört die Überprüfung der richtigen Parametrierung der UDP Parameter, welche mit der Kommandozeile von Matlab überprüft wird.

Weiterführende Tests bedürfen der Inbetriebnahme des gesamten Aufbaus. Das Prüfprotokoll befindet sich im Anhang.

5.1.4 Gesamtes Messsystems

Zunächst wird die mechanische Robustheit des gesamten Aufbaus untersucht. Dabei lässt sich feststellen, dass die Installationsplatte stabil genug ist, um mechanischen Belastungen stand zu halten. Ebenfalls sind die spannungsführenden Leitungen so installiert, dass diese mechanischen Belastungen standhalten. Die einzelnen Komponenten sind so installiert, dass diese einen festen Sitz haben.

Als nächstes werden die Batteriezellencontroller und der Batteriemodulcontroller weiter untersucht. Es ist zu erkennen, dass der Datentransfer zwischen den Controller funktioniert. Dies ist ebenfalls an der Indikator LED des Batteriemodulcontrollers zu erkennen.

Als nächstes wird die Kommunikation des Batteriemodulcontrollers mit einem Computer über die UDP-Schnittstelle getestet. Dazu werden nacheinander alle Kommandos an den Modulcontroller übergeben und die Antwort desselben ausgewertet. Dabei werden ebenfalls die Bedienbarkeit und Funktionalität der Matlab-Software getestet, sowie die richtige Parametrierung der PC-Software und die Modulcontroller-Software.

5.2 Auswertung

Bei den Funktionstest hat sich herausgestellt, dass die Funktionalität der Komponenten wie gefordert, gewährleistet und sichergestellt ist. Darüber hinaus weist der gesamte Aufbau eine gute mechanische Robustheit und Stabilität auf.

Es stehen weitere Tests zum Zeitverhalten der Komponenten aus, darunter die Anforderungen an das adaptierte Übertragungsprotokoll. Ebenfalls stehen weitere Tests zur Robustheit aus, wie etwa der Einfluss von Kondensat auf die Komponenten des Demonstrationsaufbaus.

6 Zusammenfassung

6.1 Bewertung der Thesis

Im Rahmen dieser Thesis konnte ein Funktionsdemonstrator auf Basis von Vorarbeiten reproduziert werden, um eigene Tests mit demselben durchzuführen. Des Weiteren sind theoretische Überlegungen zu den Thematiken der Synchronität und Übertragung von Nachrichten durchgeführt worden. In diesem Zusammenhang wurde betrachtet, wie das Übertragungsprotokoll der Vorarbeit an das einer anderen Vorarbeit adaptiert werden kann.

Mit Hilfe von ausgesuchten Sensoren wurde dann eine Abschätzung des zu erwartenden Overheads gemacht, bei der ebenfalls Konzepte zu der fehlersicheren Übertragung von Nachrichten diskutiert wurden.

Die bereits bestehende Software des Batteriemodulcontrollers wurde um eine lwIP-Ethernet Schnittstelle erweitert, sodass die Daten von den Batteriezellencontrollern an einen externen Computer übertragen werden können. Dabei wurde ein Matlab-Skript entwickelt, dass die Daten von dem Batteriemodulcontroller abfragen und darstellen kann. Dieses verfügt zudem über eine Benutzerführung zur einfachen Bedienbarkeit.

Abschließend wurden aussagefähige Tests der einzelnen Komponenten und des gesamten Funktionsaufbaus, anhand von aussagefähigen Testmustern, durchgeführt, um dessen Funktion nachweisen zu können.

Zusammenfassend konnten so weitere wertvolle Erkenntnisse über das Thema der optischen Datenkommunikation von Batteriemodulen in Elektrofahrzeugen gewonnen werden, die einen guten Ausgangspunkt für weitere Arbeiten darstellen.

6.2 Offene Punkte und Ausblick

Im Rahmen von weiteren Arbeiten ist zu untersuchen, wie der Lichtleitkörper verbessert werden kann hinsichtlich Übertragung, Robustheit, Fertigungsvarianten usw. Darüber hinaus sind weitere Untersuchungen zu der verwendeten Hardware durchzuführen, etwa wie die Sendeleistung verbessert werden kann, Energieverbrauch gesenkt werden kann und wie die Effizienz verbessert werden kann.

Außerdem sind weitere Untersuchungen zum Thema Zeitverhalten des theoretisch betrachteten Übertragungsprotokolls durchzuführen und weitere Konzepte der Synchronisation zwischen Batteriemodulcontroller und Batteriezellencontroller zu erarbeiten. In diesem Zusammenhang ist auch die Thematik der Messsynchronisation für den Funktionsaufbau weiter zu untersuchen.

Literaturverzeichnis

[1] *E4.1.1 Anforderungen-Sensorverfahren-v03-2020-09-04*. Hochschule für Angewandte Wissenschaften Hamburg, 2021

[2] AG, Infineon T.: *Evaluation Board For XMC1000 Family XMC 2Go Kit with XMC1100 Kit Version 1.0*. Infineon Technologies AG, 2014. – URL https://www.infineon.com/dgdl/Board_Users_Manual_XMC_2Go_Kit_with_XMC1100_R1.0.pdf?fileId=db3a3043444ee5dc014453d6c75078c6

[3] AG, Infineon T.: *Oscillator handling*. Infineon Technologies AG, 2016. – URL https://www.infineon.com/dgdl/Infineon-ApplicationNote_AP32321_Oscillator_Handling-AN-v01_00-EN.pdf?fileId=5546d46253f65057015471bc512f774b

[4] AG, Infineon T.: *XMC4700 / XMC4800 Microcontroller Series for Industrial Applications*. Infineon Technologies AG, 2018. – URL https://www.infineon.com/dgdl/Infineon-XMC4700-XMC4800-DS-v01_01-EN.pdf?fileId=5546d462518ffd850151908ea8db00b3

[5] AG, Infineon T.: *KIT-XMC-2GO-XMC1100-V1*. Infineon Technologies AG, 2021. – URL https://www.infineon.com/cms/de/product/evaluation-boards/kit_xmc_2go_xmc1100_v1/

[6] AG, Infineon T.: *KIT-XMC47-RELAX-V1*. Infineon Technologies AG, 2021. – URL https://www.infineon.com/cms/de/product/evaluation-boards/kit_xmc47_relax_v1/

[7] Electronics, Digi-Key: *TFDU4101-TR3*. Digi-Key Electronics, 2021. – URL https://www.digikey.ch/product-detail/de/vishay-semiconductor-opto-division/TFDU4101-TR3/751-1070-1-ND/1681420

[8] Ernsting, Jonas: *Funktionsdemonstrator für die optische Messdatenübertragung in Fahrzeugbatterien.* Hochschule für Angewandte Wissenschaften Hamburg, 2021

[9] H, M.: *Anzahl Akkuzellen in E-Autos.* e-auto-journal.de, 2018. – URL https://e-auto-journal.de/anzahl-akkuzellen-in-e-autos/

[10] Sassano, Nico: *Hard- und Softwareentwicklung für einen drahtlos kommunizierenden Batterie-Zellensensor mit funksynchronisierter Messung.* Hochschule für Angewandte Wissenschaften Hamburg, 2013. – URL https://reposit.haw-hamburg.de/handle/20.500.12738/6310

[11] Sassano, Nico: *Entwicklung eines Messsystems zur funksynchronisierten elektrochemischen Impedanzspektroskopie an Batterie-Zellen.* Hochschule für Angewandte Wissenschaften Hamburg, 2015. – URL https://reposit.haw-hamburg.de/handle/20.500.12738/75849

[12] Semiconductors, Vishay: *Infrared Data Communication According the IrDA® Standard Part 2: Protocol IrDA Protocol Stack.* Vishay Semiconductors, 2021. – URL https://www.vishay.com/docs/82504/protocol.pdf

[13] Semiconductors, Vishay: *TFDU4101 Product Information.* Vishay Semiconductors, 2021. – URL https://www.vishay.com/product?docid=81288&tab=documents

A Anhang

A.1 Prüfprotokolle

A.1.1 Batteriezellencontroller

Test Zellcontroller-Software

Lfd. Nr.	Inhalt	Bemerkung
Z1	Rote LED Datentransfer	Erfolgt (visuell)
Z2	Gelbe LED Datentransfer	Erfolgt (visuell)
Z3	Variation Potentiometer	Erfolgt (Debugger)
Z4	Kodierschalter + Taster	Adressänderung + Bestätigung (Debugger)
Z5	IrDA Transceiver Übertragung	Erfolgt (Debugger)
Z6	Auswertung erhaltene Kommando	Erfolgt (Debugger)
Z7	DAVE APPS korrekt parametriert	Erfolgt (Debugger)
Z8	Korrekte Datenübergabe	Erfolgt (Debugger)

A.1.2 Batteriemodulcontroller

Test Modulcontroller-Software

Lfd. Nr.	Inhalt	Bemerkung
M1	Rote LED Datentransfer IrDA	Erfolgt (visuell)
M2	Rote LED Datentransfer UDP	Erfolgt (visuell)
M3	IrDA Transceiver Übertragung	Erfolgt (Debugger)
M4	Auswertung erhaltene Kommando UART	Erfolgt (Debugger)
M5	Auswertung erhaltene Kommando UDP	Erfolgt (Debugger)
M6	DAVE APPS korrekt parametriert	Erfolgt (Debugger)
M7	Korrekte Datenabfrage der Zellencontroller	Erfolgt (Debugger)

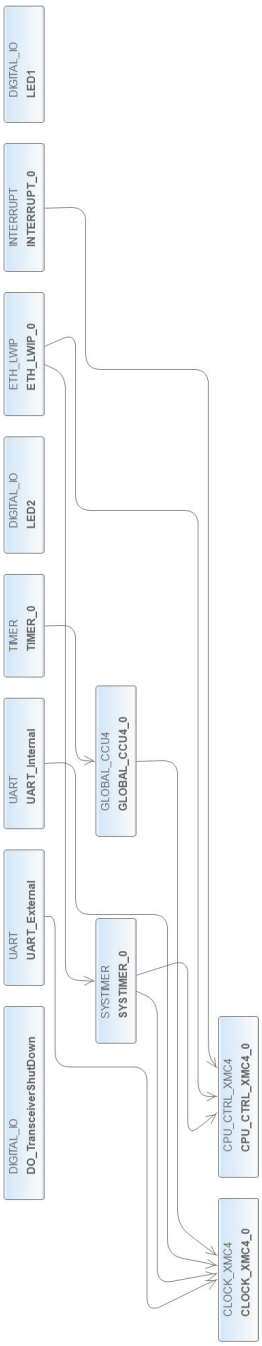
A.1.3 PC Software zur Steuerung und Visualisierung

Test Matlab-Software

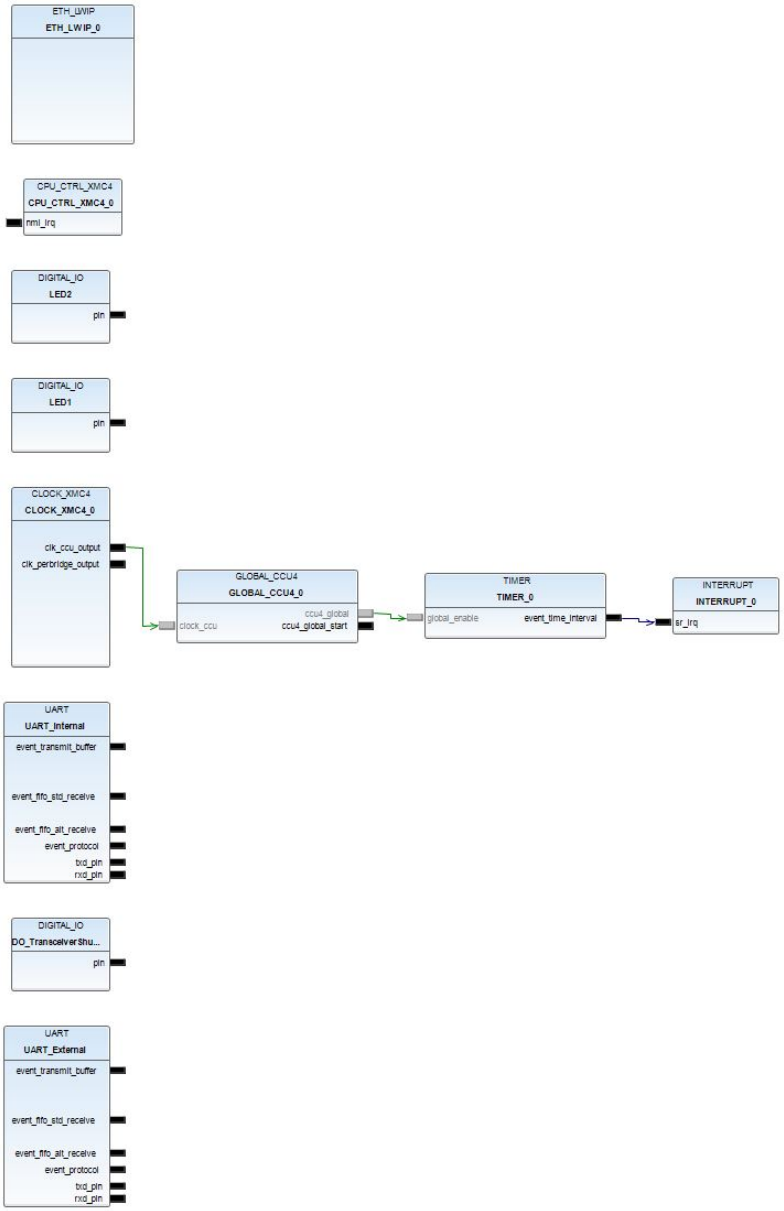
Lfd. Nr.	Inhalt	Bemerkung
MA1	Ausgabe der Überschriften	Erfolgt (Debugger)
MA2	Verarbeitung der Benutzereingaben	Erfolgt (Debugger)
MA3	Fehlerbehandlung	Erfolgt (Debugger)
MA4	Übergabe der Nachricht an UDP Schnittstelle	Erfolgt (Debugger)
MA5	Visualisierung der empfangenen Daten	Erfolgt (Debugger)
MA6	UDP Schnittstelle korrekt parametriert	Erfolgt (Debugger)
MA7	Übergabe von Kommandos an UDP-Schnittstelle	Erfolgt (Debugger)

A.2 Konfiguration

A.2.1 APP Abhängigkeit der DAVE-APPs des Batteriemodulcontrollers



A.2.2 Hardwaresignalverbindungen der DAVE-APPs des Batteriemodulcontrollers



A.2.3 Konfiguration der DAVE-APPs des Batteriemodulcontrollers

Registerkarte	Einstellung	Wert
Interrupt Settings	Enable interrupt at initialization	true
Interrupt priority	Preemption priority	63
Interrupt priority	Subpriority	0
Interrupt handler	Timer0_Handler	

Tabelle A.1: Konfiguration der DAVE-APP INTERRUPT mit dem Namen INTERRUPT_0

Registerkarte	Einstellung	Wert
General Settings	Select timer module	CCU4
General Settings	Time interval [usec]	1000
General Settings	Start after initialization	false
Event Settings	Time interval event	true

Tabelle A.2: Konfiguration der DAVE-APP TIMER mit dem Namen TIMER_0

Registerkarte	Einstellung	Wert
General Settings	Enable netif hostname	XMC4700
General Settings	Enable callback to notify netif status changed	false
General Settings	Enable RTOS	false
General Settings	Enable debug messages	false
General Settings	Enable assertions and error messages	false
General Settings	Enable statistics collection	false
Network Interface	MII interface mode	RMII
Network Interface	PHY device	KSZ8081RNB
Network Interface	PHY address	0
Network Interface	Enable autonegotation	true
Network Interface	MAC address	00:03:19:45:00:00
Network Interface	Number of receive buffers	4
Network Interface	Number of transmit buffers	4
Network Interface	Enable promiscous mode	false
Network Interface	Accept broadcast frames	true
Network Interface	Poll for received data	false
Network Interface	Preemption priority	62
Network Interface	Subpriority	0
IP Settings	Use DHCP	false
IP Settings	IP address	192.168.10
IP Settings	Subnet mask	255.255.255.0
IP Settings	Gareway address	192.168.0.1
IP Settings	Enable IP options	false
IP Settings	Enable IP fragmentation	false
IP Settings	Enable IP reassembly	false
IP Settings	Max. transmission unit (MTU)	1500
IP Settings	Default timo to live (TTL)	255
Protocol Settings	ARP table size	10
Protocol Settings	Enable AUTOIP	false
Protocol Settings	Enable ICMP	false
Protocol Settings	Enalbe IGMP	false
Protocol Settings	Enable DNS	false
Protocol Settings	Enable SNMP	false
Protocol Settings	Maximum segment size (TCP_MSS)	536
Protocol Settings	Window size (TCP_WND)	2144
Protocol Settings	TCP sender buffer space [bytes] (TCP_SND_BUF)	1072
Protocol Settings	TCP sender buffer space [pbufs] (TCP_SND_QUEUELEN)	8
Protocol Settings	Enable UDP	true
Memory Settings	Heap size (MEM_SIZE)	1600
Memory Settings	Packet buffer pool size (PBUF_POOL_SIZE)	16
Memory Settings	Number of memp struct pbufs (MEMP_NUM_BPUF)	16
Memory Settings	Number of raw connection PCBs (MEMP_SUM_RAW_PCB)	4
Memory Settings	Simulatenously active UDP connections (MEMP_NUM_UDP_PCB)	4
Memory Settings	Simulatenously active TCP connections (MEMP_NUM_TCP_PCB)	5
Memory Settings	Listening TCP connections (MEMP_NUM_TCP_PCB_LISTEN)	5
Memory Settings	Simultaneously queued TCP segments (MEM_NUM_TCP_SEG)	8
Memory Settings	Simultaneously queued pbufs waiting for ARP response (MEM_NUM_ARP_QUEUE)	30

Abbildung A.1: Konfiguration der DAVE-APP ETH_LWIP mit dem Namen ETH_-LWIP_0

A.3 Quelltexte

A.3.1 Quelltexte Batteriemodulcontroller

Listing A.1: Quelltext zum lesen der ersten Daten aus dem UART-Eingangspuffer um die Adresse und die ersten Daten zu extrahieren und zu speichern

breaklines

```
1  /*
2   * external_uart_receive_first.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "external_uart_receive_first.h"
11 #include "globals.h"
12
13
14
15 /**
16  * If the UART input buffer is not empty, read the first (oldest) data from the
       UART input buffer to extract and save the address and first (oldest) data.
17  * returns true if data was available
18  */
19 bool externalUART_receiveFirst()
20 {
21         bool dataAvailable = false;
22
23         if(!UART_IsRXFIFOEmpty(&UART_External)) // read oldest data from receive
               FIFO and thereby shift all receive FIFO data
24         {
25                 externalUART_inputBuffer = UART_GetReceivedWord(&UART_External);
26                 externalUART_inputAddress = (uint8_t) ((externalUART_inputBuffer &
                       EXTERNAL_UART_MASK_ADDRESS) >>
                       EXTERNAL_UART_NUMBER_OF_BITS_DATA);
27                 externalUART_inputData = (uint8_t) (externalUART_inputBuffer &
                       EXTERNAL_UART_MASK_DATA);
28                 dataAvailable = true; // indicate that data has been received
29         }
30
31         return dataAvailable; // indicate that no data has been received
32 }
```

Listing A.2: Headerfile mit Prototyp für das Listing A.1

breaklines

```
1  /*
2   * external_uart_receive_first.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef EXTERNAL_UART_RECEIVE_FIRST_H_
9  #define EXTERNAL_UART_RECEIVE_FIRST_H_
10
11 // Prototype
12 extern bool externalUART_receiveFirst();
13
14 #endif /* EXTERNAL_UART_RECEIVE_FIRST_H_ */
```

Listing A.3: Quelltext zum lesen der letzten Daten aus dem UART-Eingangspuffer

breaklines

```
1  /*
2   * external_uart_receive_last.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "external_uart_receive_last.h"
11 #include "globals.h"
12
13 /**
14  * First check whether reception is in progress and if so, wait until it's
        finished.
15  * Then, if the UART input buffer is not empty, read the first (oldest) data from
        the UART input buffer until it is empty to save the last (newest) data.
16  * This discards all older received data in the UART input buffer.
17  * returns true if data was available
18  */
19
20 bool externalUART_receiveLast()
21 {
22         bool dataAvailable = false;
23
24         while(UART_IsRxBusy(&UART_External)); // wait for any running reception to
               finish
25
26         while(!UART_IsRXFIFOEmpty(&UART_External)) // read oldest data from
               receive FIFO and thereby shift all receive FIFO data until the receive
               FIFO is empty to get the latest received data
27         {
28                 externalUART_inputBuffer = UART_GetReceivedWord(&UART_External);
29                 externalUART_inputAddress = (uint8_t) ((externalUART_inputBuffer &
                       EXTERNAL_UART_MASK_ADDRESS) >>
                       EXTERNAL_UART_NUMBER_OF_BITS_DATA);
30                 externalUART_inputData = (uint8_t) (externalUART_inputBuffer &
                       EXTERNAL_UART_MASK_DATA);
31                 dataAvailable = true; // indicate that data has been received
32         }
33
34         return dataAvailable; // indicate whether data has been received
35 }
```

Listing A.4: Headerfile mit Prototyp für das Listing A.3

breaklines

```
1  /*
2   * external_uart_receive_last.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef EXTERNAL_UART_RECEIVE_LAST_H_
9  #define EXTERNAL_UART_RECEIVE_LAST_H_
10
11 // Prototype
12 bool externalUART_receiveLast();
13
14 #endif /* EXTERNAL_UART_RECEIVE_LAST_H_ */
```

Listing A.5: Quelltext zum Übergeben eines Symbols mittels UART

breaklines

```
1  /*
2   * external_uart_transmit.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "external_uart_transmit.h"
11 #include "globals.h"
12
13 /**
14  * transmit one symbol via UART
15  */
16 void externalUART_transmit(void)
17 {
18         externalUART_outputBuffer = (externalUART_outputAddress <<
               EXTERNAL_UART_NUMBER_OF_BITS_DATA) | externalUART_outputData; // set
               output buffer
19         while(UART_IsTxBusy(&UART_External)); // wait for previous transmission to
               finish
20         UART_Transmit(&UART_External, &externalUART_outputBuffer, 1U);
21 }
```

Listing A.6: Headerfile mit Prototyp für das Listing A.5

breaklines

```
1  /*
2   * external_uart_transmit.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef EXTERNAL_UART_TRANSMIT_H_
9  #define EXTERNAL_UART_TRANSMIT_H_
10
11 // Prototype
12 extern void externalUART_transmit(void);
13
14 #endif /* EXTERNAL_UART_TRANSMIT_H_ */
```

Listing A.7: Quelltext zum Übergeben eines String mittels UART

breaklines

```
1  /*
2   * external_uart_transmit_string.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "external_uart_transmit_string.h"
11 #include "globals.h"
12
13 /**
14  * transmit a string via UART
15  */
16 void externalUART_transmitString(char *pointer)
17 {
18         UART_Transmit(&UART_External, (uint8_t*) pointer, strlen(pointer));
19 }
```

Listing A.8: Headerfile mit Prototyp für das Listing A.7

breaklines

```
1  /*
2   * external_uart_transmit_string.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef EXTERNAL_UART_TRANSMIT_STRING_H_
9  #define EXTERNAL_UART_TRANSMIT_STRING_H_
10
11 // Prototype
12 void externalUART_transmitString(char *pointer);
13
14
15 #endif /* EXTERNAL_UART_TRANSMIT_STRING_H_ */
```

Listing A.9: Quelltext zum Senden einer Nachricht vom externen Gerät über externen UART an die Batteriezellencontroller mittels internen UART und Rückgabe der ersten Antwort

breaklines

```
1  /*
2   * forward_command_and_wait_for_answer.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "forward_command_and_wait_for_answer.h"
11 #include "globals.h"
12
13
14 /**
15  * forward the message from the external device via external UART to the slaves
        via internal UART and return the first answer of slave
16  * or a timeout message if the slave didn't respond
17  */
18 void forwardCommandAndWaitForAnswer(void)
19 {
20         internalUART_outputAddress = externalUART_inputAddress;
21         internalUART_outputData = externalUART_inputData;
22         internalUART_transmit();
23
24         oneShotTimerStart(INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US, false); //
               start timer for response timeout
25
26         // wait for answer of any slave or until
               INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US has elapsed
27         while(true)
28         {
29                 // check and handle newly received command from the external
                       device
30                 if(internalUART_receiveFirst()) // check for new received data
31                 {
32                         // abort execution
33                         if(externalUART_inputData ==
                               EXTERNAL_UART_COMMANDMESSAGE_ABORT)
34                         {
35                                 oneShotTimerElapsed(); // stop timer
36                                 externalUART_outputAddress =
                                       EXTERNAL_UART_ADDRESS_EXTERNAL;
37                                 externalUART_outputData =
                                       EXTERNAL_UART_COMMANDMESSAGE_ABORT;
```

```
38                                  externalUART_transmit();
39                                  break; // exit while loop
40                          }
41                          // inform external device that command is being ignored
42                          else
43                          {
44                                  externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
45                                  externalUART_outputData =
                                        EXTERNAL_UART_MESSAGE_COMMAND_IGNORED;
46                                  externalUART_transmit();
47                          }
48                  }
49                  // transmission received from the slave
50                  else if(internalUART_receiveLast()) // check for new received data
51                  {
52                          oneShotTimerElapsed(); // stop timer
53                          externalUART_outputAddress = internalUART_inputAddress;
54                          externalUART_outputData = internalUART_inputData;
55                          externalUART_transmit();
56                          break; // exit while loop
57                  }
58                  // timeout; slave didn't answer
59                  else if (!oneShotTimer.running)
60                  {
61                          externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
62                          externalUART_outputData =
                                EXTERNAL_UART_MESSAGE_SLAVE_DOESNT_RESPOND;
63                          externalUART_transmit();
64                          break; // exit while loop
65                  }
66          }
67  }
```

Listing A.10: Headerfile mit Prototyp für das Listing A.9

breaklines

```
1  /*
2   * forward_command_and_wait_for_answer.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef FORWARD_COMMAND_AND_WAIT_FOR_ANSWER_H_
9  #define FORWARD_COMMAND_AND_WAIT_FOR_ANSWER_H_
10
11 // Prototype
12 extern void forwardCommandAndWaitForAnswer(void);
13
14 #endif /* FORWARD_COMMAND_AND_WAIT_FOR_ANSWER_H_ */
```

Listing A.11: Quelltext zum lesen der ersten Daten aus dem internen UART-Eingangspuffer umdie Adresse und die ersten Daten zu extrahieren und zu speichern

breaklines

```
1  /*
2   * internal_uart_receive_first.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
8  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
9  #include "internal_uart_receive_first.h"
10 #include "globals.h"
11
12 /**
13  * If the UART input buffer is not empty, read the first (oldest) data from the
        UART input buffer to extract and save the address and first (oldest) data.
14  * returns true if data was available
15  */
16 bool internalUART_receiveFirst()
17 {
18         bool dataAvailable = false;
19
20         if(!UART_IsRXFIFOEmpty(&UART_Internal)) // read oldest data from receive
               FIFO and thereby shift all receive FIFO data
21         {
22                 internalUART_inputBuffer = UART_GetReceivedWord(&UART_Internal);
23                 internalUART_inputAddress = (uint8_t) ((internalUART_inputBuffer &
                       INTERNAL_UART_MASK_ADDRESS) >>
                       INTERNAL_UART_NUMBER_OF_BITS_DATA);
24                 internalUART_inputData = (uint8_t) (internalUART_inputBuffer &
                       INTERNAL_UART_MASK_DATA);
25                 dataAvailable = true; // indicate that data has been received
26         }
27
28         return dataAvailable; // indicate that no data has been received
29 }
```

Listing A.12: Headerfile mit Prototyp für das Listing A.11

breaklines

```
1  /*
2   * internal_uart_receive_first.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef INTERNAL_UART_RECEIVE_FIRST_H_
9  #define INTERNAL_UART_RECEIVE_FIRST_H_
10
11 // Prototype
12 bool internalUART_receiveFirst();
13
14 #endif /* INTERNAL_UART_RECEIVE_FIRST_H_ */
```

Listing A.13: Quelltext zum lesen der letzten Daten aus dem internen UART-Eingangspuffer umdie Adresse und die ersten Daten zu extrahieren und zu speichern

breaklines

```
1  /*
2   * internal_uart_receive_last.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "internal_uart_receive_last.h"
11 #include "globals.h"
12
13 /**
14  * First check whether reception is in progress and if so, wait until it's
        finished.
15  * Then, if the UART input buffer is not empty, read the first (oldest) data from
        the UART input buffer until it is empty to save the last (newest) data.
16  * This discards all older received data in the UART input buffer.
17  * returns true if data was available
18  */
19 bool internalUART_receiveLast()
20 {
21         bool dataAvailable = false;
22
23         while(UART_IsRxBusy(&UART_Internal)); // wait for any running reception to
              finish
24
25         while(!UART_IsRXFIFOEmpty(&UART_Internal)) // read oldest data from
              receive FIFO and thereby shift all receive FIFO data until the receive
              FIFO is empty to get the latest received data
26         {
27                 internalUART_inputBuffer = UART_GetReceivedWord(&UART_Internal);
28                 internalUART_inputAddress = (uint8_t) ((internalUART_inputBuffer &
                       INTERNAL_UART_MASK_ADDRESS) >>
                       INTERNAL_UART_NUMBER_OF_BITS_DATA);
29                 internalUART_inputData = (uint8_t) (internalUART_inputBuffer &
                       INTERNAL_UART_MASK_DATA);
30                 dataAvailable = true; // indicate that data has been received
31         }
32
33         return dataAvailable; // indicate that no data has been received
34 }
```

Listing A.14: Headerfile mit Prototyp für das Listing A.13

breaklines

```
1  /*
2   * internal_uart_receive_last.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef INTERNAL_UART_RECEIVE_LAST_H_
9  #define INTERNAL_UART_RECEIVE_LAST_H_
10
11 // Prototype
12 extern bool internalUART_receiveLast();
13
14 #endif /* INTERNAL_UART_RECEIVE_LAST_H_ */
```

Listing A.15: Quelltext zum Übertragen eines Symbols mittels der internen UART-Schnittstelle

breaklines

```
1  /*
2   * internal_uart_transmit.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
8  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
9  #include "internal_uart_transmit.h"
10 #include "globals.h"
11
12
13 /**
14  * transmit one symbol via UART
15  * Any received data which was received while sending data is discarded.
16  */
17 void internalUART_transmit(void)
18 {
19         /*
20          * The while loops "while(UART_IsTxBusy(&UART_Internal) || UART_IsRxBusy(&
                UART_Internal))" before and after the function
21          * "UART_Transmit(const UART_t * const handle, uint8_t * data_ptr,
                uint32_t count)" shouldn't be necessary when receive mode
22          * is set to "direct" which blocks the CPU until all data is sent or
                received.
23          * (See description of function "UART_lStartTransmitPolling(const UART_t *
                const handle, uint8_t* data_ptr, uint32_t count)" and
24          * "UART_lStartReceivePolling(const UART_t *const handle, uint8_t*
                data_ptr, uint32_t count)" in "Dave\Generated\UART\uart.c".)
25          */
26         internalUART_outputBuffer = (internalUART_outputAddress <<
               INTERNAL_UART_NUMBER_OF_BITS_DATA) | internalUART_outputData; // set
               output buffer
27         while(UART_IsTxBusy(&UART_Internal) || UART_IsRxBusy(&UART_Internal)); //
               wait for previous transmission or reception to finish
28         UART_Transmit(&UART_Internal, &internalUART_outputBuffer, 1U); // transmit
               contents of output buffer
29         while(UART_IsTxBusy(&UART_Internal) || UART_IsRxBusy(&UART_Internal)); //
               wait for transmission or reception to finish
30         while(!internalUART_receiveLast()); // wait for echoed data from the IrDA
               transmitter and discard it
31 }
```

Listing A.16: Headerfile mit Prototyp für das Listing A.15

breaklines

```
1  /*
2   * internal_uart_transmit.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef INTERNAL_UART_TRANSMIT_H_
9  #define INTERNAL_UART_TRANSMIT_H_
10
11 // Prototype
12 extern void internalUART_transmit(void);
13
14 #endif /* INTERNAL_UART_TRANSMIT_H_ */
```

Listing A.17: Quelltext für Callback-Funktion des One Shot Timers

breaklines

```
1  /*
2   * one_shot_timer_elapsed.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
8  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
9  #include "one_shot_timer_elapsed.h"
10 #include "globals.h"
11
12 /**
13  * callback function of on shot timer
14  * stop timer and indicate that timer has elapsed
15  */
16 void oneShotTimerElapsed(void)
17 {
18         // stop software timer
19         SYSTIMER_StopTimer(oneShotTimer.id);
20
21         oneShotTimer.running = false;
22 }
```

Listing A.18: Headerfile mit Prototyp für das Listing A.17

breaklines

```
1  /*
2   * one_shot_timer_elapsed.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef ONE_SHOT_TIMER_ELAPSED_H_
9  #define ONE_SHOT_TIMER_ELAPSED_H_
10
11 // Prototype
12 extern void oneShotTimerElapsed(void);
13
14 #endif /* ONE_SHOT_TIMER_ELAPSED_H_ */
```

Listing A.19: Quelltext für Start und Restart des One Shot Timers

breaklines

```
1  /*
2   * one_shot_timer_start.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
8  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
9  #include "one_shot_timer_start.h"
10 #include "globals.h"
11
12
13 /**
14  * restart software timer and check whether software timer restart failed
15  */
16 void oneShotTimerStart(uint32_t timeToWait, bool waitForTimerToElapse)
17 {
18         if(SYSTIMER_RestartTimer(oneShotTimer.id, timeToWait) ==
               SYSTIMER_STATUS_SUCCESS)
19         {
20                 oneShotTimer.running = true;
21         }
22         else // error handler code
23         {
24                 XMC_DEBUG("\nTimer start failed\n");
25                 while(true);
26         }
27
28         // wait for one shot timer to elapse
29         if(waitForTimerToElapse)
30         {
31                 while(oneShotTimer.running);
32         }
33
34 }
```

Listing A.20: Headerfile mit Prototyp für das Listing A.19

breaklines

```
1  /*
2   * one_shot_timer_start.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef ONE_SHOT_TIMER_START_H_
9  #define ONE_SHOT_TIMER_START_H_
10
11 // Prototype
12 extern void oneShotTimerStart(uint32_t timeToWait, bool waitForTimerToElapse);
13
14 #endif /* ONE_SHOT_TIMER_START_H_ */
```

Listing A.21: Quelltext für Callback-Funktion des periodischen Timers

breaklines

```
1  /*
2   * slave_data_update_periodic_timer_elapsed.c
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR
       declaration)
9  #include <stdio.h> // for sprintf, (char *__restrict, const char *__restrict, ...)
10 #include "slave_data_update_periodic_timer_elapsed.h"
11 #include "globals.h"
12
13 volatile bool slaveDataUpdateFlag; // true if an update of slave data is required
14
15
16 /**
17  * callback function of on periodic timer
18  * set flag to indicate that the periodic timer has elapsed
19  */
20 void slaveDataUpdatePeriodicTimerElapsed(void)
21 {
22         slaveDataUpdateFlag = true; // set flag
23 }
```

Listing A.22: Headerfile mit Prototyp für das Listing A.21

breaklines

```
1  /*
2   * slave_data_update_periodic_timer_elapsed.h
3   *
4   *  Created on: 10 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef SLAVE_DATA_UPDATE_PERIODIC_TIMER_ELAPSED_H_
9  #define SLAVE_DATA_UPDATE_PERIODIC_TIMER_ELAPSED_H_
10
11 // Prototype
12 extern void slaveDataUpdatePeriodicTimerElapsed(void);
13
14 #endif /* SLAVE_DATA_UPDATE_PERIODIC_TIMER_ELAPSED_H_ */
```

Listing A.23: Quelltext für Handler des Timers0

breaklines

```
1  /*
2   * timer0_handler.c
3   *
4   *  Created on: 12 Aug 2021
5   *      Author: Fabian Mahler
6   */
7  #include <DAVE.h>                 //Declarations from DAVE Code Generation (
       includes SFR declaration)
8  #include <stdbool.h>
9  #include <stdint.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <math.h>
14 #include "timer0_handler.h"
15 #include "globals.h"
16
17 /**
18  * Timer handler for updating time stamp
19  *
20  */
21 void Timer0_Handler(void)
22 {
23         TIMER_ClearEvent(&TIMER_0);
24
25         t += 1000;       // update time
26
27 }
```

Listing A.24: Headerfile mit Prototyp für das Listing A.23

breaklines

```
1  /*
2   * timer0_handler.h
3   *
4   *  Created on: 12 Aug 2021
5   *      Author: Fabian Mahler
6   */
7
8  #ifndef TIMER0_HANDLER_H_
9  #define TIMER0_HANDLER_H_
10
11 // Prototype
12 extern void Timer0_Handler(void);
13
14 #endif /* TIMER0_HANDLER_H_ */
```

Listing A.25: Headerfile für die globalen Variablen

breaklines

```
/*
 * globals.h
 *
 *  Created on: 10 Aug 2021
 *      Author: Fabian Mahler
 */

#ifndef DAVE_GENERATED_GLOBALS_H_
#define DAVE_GENERATED_GLOBALS_H_


#define TRANSCEIVER_STARTUP_TIME_IN_US ((uint32_t) 10000) // 0,5 ms; time the transceiver needs after the SD (shutdown) signal goes low before it can operate

#define SLAVE_DATA_UPDATE_TIME_IN_US ((uint32_t) 500000) // 0,5 s; time between slave data update (time before flag for update request is set to true again)

#define REMOTE_PORT 45174
#define LOCAL_PORT 45175

//IP Address Host PC
#define IP_ADDR0_PC 192
#define IP_ADDR1_PC 168
#define IP_ADDR2_PC 0
#define IP_ADDR3_PC 5

#define CELL_ADDR "01"

//commands from master
#define CMD_GET_SINGLE_DATA "01"
#define CMD_GET_DATA_CONNECTED "02"
#define CMD_GET_ALL_DATA "03"
#define CMD_GET_ADDRESS_CONNECTED "04"


#define EXTERNAL_UART_NUMBER_OF_BITS_ADDRESS ((uint32_t) 4) // number of address bits
#define EXTERNAL_UART_NUMBER_OF_BITS_DATA    ((uint32_t) 4) // number of data bits

#define EXTERNAL_UART_MASK_ADDRESS ((uint8_t) 240) // (2 ^ EXTERNAL_UART_NUMBER_OF_BITS_ADDRESS - 1U) << EXTERNAL_UART_NUMBER_OF_BITS_DATA; mask for address
#define EXTERNAL_UART_MASK_DATA    ((uint8_t)  15) // 2 ^ EXTERNAL_UART_NUMBER_OF_BITS_DATA - 1U; mask for data

#define EXTERNAL_UART_ADDRESS_MASTER    ((uint8_t)  0) // address of master transceiver
```

```
41 #define EXTERNAL_UART_ADDRESS_EXTERNAL  ((uint8_t)  1) // address of external
       device
42
43 #define EXTERNAL_UART_MESSAGE_UNKNOWN_COMMAND                    ((uint8_t)  1) //
       message for unknown command
44 #define EXTERNAL_UART_COMMANDMESSAGE_PING                        ((uint8_t)  2) //
       command and message for requesting a response
45 #define EXTERNAL_UART_MESSAGE_SLAVE_DOESNT_RESPOND               ((uint8_t)  3) //
       message for no response received within timeout time
46 #define EXTERNAL_UART_MESSAGE_COMMAND_IGNORED                    ((uint8_t)  4) //
       message for not executed and ignored command
47 #define EXTERNAL_UART_COMMANDMESSAGE_ABORT                       ((uint8_t)  5) //
       command and message for aborting current action
48 #define EXTERNAL_UART_COMMAND_SHOW_ALL_AVAILABLE_SLAVE_ADDRESSES ((uint8_t)  7) //
       command for sending addresses of all currently connected slaves
49 #define EXTERNAL_UART_COMMAND_SHOW_DATA_OF_ALL_AVAILABLE_SLAVES  ((uint8_t)  8) //
       command for sending data of all currently connected slaves
50 #define EXTERNAL_UART_COMMAND_SHOW_DATA_OF_ALL_SLAVES            ((uint8_t)  9) //
       command for sending data of all slaves
51 #define EXTERNAL_UART_COMMAND_TOGGLE_AUTO_UPDATE_SLAVE_DATA      ((uint8_t) 10) //
       command for toggling periodic sending of all slave data after it was updated
52 //...
53 //#define EXTERNAL_UART_                                         ((uint8_t) 15)
       // ...
54
55 // transmission time = (1 + 8 + 1 + 1) / (115200 1/s) = 95,486 us
56 #define EXTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US ((uint32_t) 1000) // 1000  s
       ; maximum time a slave has to respond to any command
57
58
59 #define INTERNAL_UART_NUMBER_OF_BITS_ADDRESS ((uint32_t) 4) // number of address
       bits
60 #define INTERNAL_UART_NUMBER_OF_BITS_DATA    ((uint32_t) 4) // number of data bits
61
62 #define INTERNAL_UART_MASK_ADDRESS ((uint8_t) 240) // (2 ^
       INTERNAL_UART_NUMBER_OF_BITS_ADDRESS - 1U) <<
       INTERNAL_UART_NUMBER_OF_BITS_ADDRESS; mask for address
63 #define INTERNAL_UART_MASK_DATA    ((uint8_t)  15) // 2 ^
       INTERNAL_UART_NUMBER_OF_BITS_DATA - 1U; mask for data
64
65 #define INTERNAL_UART_ADDRESS_MASTER      ((uint8_t)  0) // address of master
       transceiver
66 #define INTERNAL_UART_ADDRESS_SLAVE_START ((uint8_t)  1) // lowest address of
       first cell slave
67 #define INTERNAL_UART_ADDRESS_SLAVE_END   ((uint8_t) 12) // highest address of
       last cell slave
68 #define INTERNAL_UART_ADDRESS_BROADCAST   ((uint8_t) 15) // 2 ^
       INTERNAL_UART_NUMBER_OF_BITS_ADDRESS - 1U; broadcast address to reach all bus
       participants
69
```

```
70 #define INTERNAL_UART_MESSAGE_UNKNOWN_COMMAND          ((uint8_t) 1) // message for
       unknown command
71 #define INTERNAL_UART_COMMANDMESSAGE_PING              ((uint8_t) 2) // command and
       message for requesting a response
72 #define INTERNAL_UART_COMMANDMESSAGE_ABORT             ((uint8_t) 3) // command and
       message for aborting current action
73 #define INTERNAL_UART_COMMAND_SEND_ALL_DATA_PACKAGES ((uint8_t) 4) // command for
       sending all available data
74 //...
75 //#define INTERNAL_UART_                                (INTERNAL_UART_MASK_DATA)
       // command for ...
76
77 #define INTERNAL_UART_COMMAND_SEND_ALL_DATA_PACKAGES_MAXIMUM_TRIES ((uint32_t) 2)
       // number of retries of command for sending all data packages
78
79 // transmission time of one character = (1 + 8 + 1 + 1) / (115200 1/s) = 95,486 us
80 #define INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US ((uint32_t) 1000) // 1000  s
       ; maximum time a slave has to respond to any command
81
82
83
84 #define SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH   ((uint32_t)  4) // number of
       bits sent by the slave for the rotary switch position (Should be multiple of
       INTERNAL_UART_NUMBER_OF_BITS_DATA.)
85 #define SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_1 ((uint32_t) 12) // number of
       bits sent by the slave for the ADC value of potentiometer 1 (or supply voltage
       ) (Should be multiple of INTERNAL_UART_NUMBER_OF_BITS_DATA.)
86 #define SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_2 ((uint32_t) 12) // number of
       bits sent by the slave for the ADC value of potentiometer 2 (or temperature) (
       Should be multiple of INTERNAL_UART_NUMBER_OF_BITS_DATA.)
87
88 #define SLAVE_DATA_MASK_ROTARY_SWITCH   ((uint32_t)  255) // 2 ^
       SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH - 1U; mask for rotary switch position
       bits to be send to the master
89 #define SLAVE_DATA_MASK_POTENTIOMETER_1 ((uint32_t) 4095) // 2 ^
       SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_1 - 1U; mask for ADC value of
       potentiometer 1 (or supply voltage) bits to be send to the master
90 #define SLAVE_DATA_MASK_POTENTIOMETER_2 ((uint32_t) 4095) // 2 ^
       SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_2 - 1U; mask for ADC value of
       potentiometer 2 (or temperature) bits to be send to the master
91
92 #define SLAVE_DATA_NUMBER_OF_PACKAGES ((SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH +
       SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_2 +
       SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_1) / INTERNAL_UART_NUMBER_OF_BITS_DATA
       ) // number of transmissions from the slave after
       COMMAND_SEND_ALL_DATA_PACKAGES
93
94
95 extern volatile uint8_t externalUART_inputBuffer; // only read and write in UART
       receive event handler; buffer for received data
```

```
96  extern uint8_t externalUART_outputBuffer; // only read and write in UART
        transmission function; buffer for transmission data
97
98  extern uint8_t externalUART_inputAddress; // source address were the received data
         came from
99  extern uint8_t externalUART_inputData; // data received from source
100
101 extern uint8_t externalUART_outputAddress; // destination address were data will
        be send to
102 extern uint8_t externalUART_outputData; // data to send to destination
103
104 extern volatile bool slaveDataUpdateFlag; // true if an update of slave data is
        required
105
106 extern volatile uint8_t internalUART_inputBuffer; // only read and write in UART
        receive event handler; buffer for received data
107 extern uint8_t internalUART_outputBuffer; // only read and write in UART
        transmission function; buffer for transmission data
108
109 extern uint8_t internalUART_inputAddress; // source address were the received data
         came from
110 extern uint8_t internalUART_inputData; // data received from source
111
112 extern uint8_t internalUART_outputAddress; // destination address were data will
        be send to
113 extern uint8_t internalUART_outputData; // data to send to destination
114
115 extern struct struct_slave {
116         uint8_t address; // address of slave
117         bool connectionState; // state of connection: false: not connected | true:
                 connected
118         bool connectionStateChanged; // state of connection changed: false: no
                change | true: if connectionState == true then newly connected else
                connection lost
119         bool connectionProblems; // indicates problems with the communication, if
                a connection is established
120         uint8_t rotarySwitch; // position of rotary switch
121         uint16_t potentiometer1ADC; // ADC value of potentiometer 1; this analog
                input can also be used for supply voltage measurement
122         uint16_t potentiometer2ADC; // ADC value of potentiometer 2; this analog
                input can also be used for temperature measurement
123 };
124
125
126 //extern volatile bool slaveDataUpdateFlag; // true if an update of slave data is
        required
127
128 extern struct struct_oneShotTimer {
129         uint32_t id; // timer ID for controlling one the timer
130         volatile bool running; // true while one shot timer is running
131 };
```

```
132
133 //extern struct struct_oneShotTimer oneShotTimer; // one shot timer parameters
134 struct struct_oneShotTimer oneShotTimer;
135
136 extern uint64_t t;                                    // time stored in MCU
137
138 #endif /* DAVE_GENERATED_GLOBALS_H_ */
```

Listing A.26: Quelltext main() Funktion mit udp Funktion

breaklines

```
1  #include <DAVE.h>                 //Declarations from DAVE Code Generation (
       includes SFR declaration)
2  #include <stdbool.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <math.h>
8
9
10 // include headerfiles
11 #include "globals.h"
12 #include "one_shot_timer_start.h"
13 #include "one_shot_timer_elapsed.h"
14 #include "slave_data_update_periodic_timer_elapsed.h"
15 #include "slave_data_update_periodic_timer_elapsed.h"
16 #include "external_uart_receive_first.h"
17 #include "external_uart_receive_last.h"
18 #include "external_uart_transmit.h"
19 #include "internal_uart_receive_first.h"
20 #include "internal_uart_receive_last.h"
21 #include "internal_uart_transmit.h"
22 #include "forward_command_and_wait_for_answer.h"
23 #include "timer0_handler.h"
24
25
26
27
28 // pointers for udp connection
29 struct udp_pcb *pcb_tr;
30 struct pbuf *p_tx;                                  // transmit buffer
31 ip_addr_t ip_hostpc;
32
33
34
35 uint64_t t = 0;                                     // time stored in MCU
36
37
38
39 struct struct_slave slaveData[INTERNAL_UART_ADDRESS_SLAVE_END -
       INTERNAL_UART_ADDRESS_SLAVE_START + 1U] = {}; // struct array for all slave
       data
40 uint32_t slaveDataNumberOfElements = sizeof(slaveData) / sizeof(slaveData[0]); //
       number of elements in slaveData
41
42
43
```

```
44 bool externalUART_autoUpdateSlaveData = false; // flag that indicates whether all
       slave data shall be send to the external device via UART after it was updated
45
46 uint32_t slavDataUpdatePeriodicTimerID = 0U; // timer ID for controlling the timer
47
48
49 volatile uint8_t externalUART_inputBuffer = 0U; // only read and write in UART
       receive event handler; buffer for received data
50 uint8_t externalUART_outputBuffer = 0U; // only read and write in UART
       transmission function; buffer for transmission data
51
52 uint8_t externalUART_inputAddress = 0U; // source address were the received data
       came from
53 uint8_t externalUART_inputData = 0U; // data received from source
54
55 uint8_t externalUART_outputAddress = 0U; // destination address were data will be
       send to
56 uint8_t externalUART_outputData = 0U; // data to send to destination
57
58
59 volatile uint8_t internalUART_inputBuffer = 0U; // only read and write in UART
       receive event handler; buffer for received data
60 uint8_t internalUART_outputBuffer = 0U; // only read and write in UART
       transmission function; buffer for transmission data
61
62 uint8_t internalUART_inputAddress = 0U; // source address were the received data
       came from
63 uint8_t internalUART_inputData = 0U; // data received from source
64
65 uint8_t internalUART_outputAddress = 0U; // destination address were data will be
       send to
66 uint8_t internalUART_outputData = 0U; // data to send to destination
67
68
69
70
71 /**
72  * send all slave data as a string (table format)
73  */
74 void sendSlaveDataString(bool onlyPrintIfSlaveAvailable)
75 {
76         char uart_outputBuffer[128U] = {};
77         // print header
78         sprintf(uart_outputBuffer, "\n Address | Connection | Rotary Switch |
               Potentiometer | Potentiometer");
79         externalUART_transmitString(uart_outputBuffer);
80         sprintf(uart_outputBuffer, "\n         | State      | Position      | 1
               ADC Value    | 2 ADC Value");
81         externalUART_transmitString(uart_outputBuffer);
```

```
82          sprintf(uart_outputBuffer, "\n
                ----------+----------+----------+----------+----------
            ");
83          externalUART_transmitString(uart_outputBuffer);
84          // print table content
85          for(uint32_t idx = 0U;idx < slaveDataNumberOfElements; idx += 1U) {
86                  // check and handle newly received command from the external
                        device
87                  if(internalUART_receiveFirst()) // check for new received data
88                  {
89                          // abort execution
90                          if(externalUART_inputData ==
                                EXTERNAL_UART_COMMANDMESSAGE_ABORT)
91                          {
92                                  externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
93                                  externalUART_outputData =
                                        EXTERNAL_UART_COMMANDMESSAGE_ABORT;
94                                  externalUART_transmit();
95
96                                  return; // exit function
97                          }
98                          // inform external device that command is being ignored
99                          else
100                         {
101                                 externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
102                                 externalUART_outputData =
                                        EXTERNAL_UART_MESSAGE_COMMAND_IGNORED;
103                                 externalUART_transmit();
104                         }
105                 }
106                 else if(!onlyPrintIfSlaveAvailable || slaveData[idx].
                        connectionState)
107                 {
108                         sprintf(uart_outputBuffer, "\n  0b%u%u%u%u | %s %s | %13hu
                                | %13hu | %13hu",
109                         (slaveData[idx].address & 8U) ? (1U) : (0U),
110                         (slaveData[idx].address & 4U) ? (1U) : (0U),
111                         (slaveData[idx].address & 2U) ? (1U) : (0U),
112                         (slaveData[idx].address & 1U) ? (1U) : (0U),
113                         (slaveData[idx].connectionProblems) ? ("ERR  ") : ("     "
                                ),
114                         (slaveData[idx].connectionState) ? ((slaveData[idx].
                                connectionStateChanged) ? (" New") : ("   C")) : ((
                                slaveData[idx].connectionStateChanged) ?
115 ("Lost") : ("  NC")),
116                         (unsigned short) slaveData[idx].rotarySwitch,
117                         (unsigned short) slaveData[idx].potentiometer1ADC,
118                 (unsigned short) slaveData[idx].potentiometer2ADC);
119                         externalUART_transmitString(uart_outputBuffer);
```

```
120                 }
121         }
122 }
123
124
125 /*
126  * send all slave addresses to the external device via UART
127  */
128 void showAllAvailableSlaveAddresses(void)
129 {
130         // iterate over all salves
131         for(uint32_t idx = 0U; idx < slaveDataNumberOfElements; idx += 1U)
132         {
133                 // check and handle newly received command from the external
                        device
134                 if(internalUART_receiveFirst()) // check for new received data
135                 {   // check command
136                         // abort execution
137                         if(externalUART_inputData ==
                                EXTERNAL_UART_COMMANDMESSAGE_ABORT)
138                         {
139                                 externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
140                                 externalUART_outputData =
                                        EXTERNAL_UART_COMMANDMESSAGE_ABORT;
141                                 externalUART_transmit();
142                                 break; // exit for loop
143                         }
144                         // inform external device that command is being ignored
145                         else
146                         {
147                                 externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
148                                 externalUART_outputData =
                                        EXTERNAL_UART_MESSAGE_COMMAND_IGNORED;
149                                 externalUART_transmit();
150                         }
151                 }
152                 // transmit all known slave addresses
153                 else if(slaveData[idx].connectionState)
154                 {
155                         externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
156                         externalUART_outputData = slaveData[idx].address;
157                         externalUART_transmit();
158                 }
159         }
160 }
161
162
163
```

```
164
165 /**
166  * request information from slave and save it if the slave responds
167  */
168 void slaveDataUpdate(uint8_t *address)
169 {
170         uint32_t commandTryCounter = 0U; // number of times the command has been
                sent
171         uint32_t receivedDataBuffer; // buffer for data from received
                transmissions
172         uint32_t packageCounter; // number of received transmissions from the
                slave
173         uint32_t slaveDataArrayIndex; // index of the slave in the slaveData array
                 with the given address
174
175         // search for correct address and save the index
176         for(slaveDataArrayIndex = 0U; slaveDataArrayIndex <
                slaveDataNumberOfElements; slaveDataArrayIndex += 1U)
177         {
178                 if(slaveData[slaveDataArrayIndex].address == *address)
179                 {
180                         break; // exit for loop
181                 }
182                 // if the slave address was not found in slaveData handle error
183                 else if(slaveDataArrayIndex == (slaveDataNumberOfElements - 1U))
184                 {
185                         // error handler code
186                         XMC_DEBUG("\nSlave not found.\n");
187                         while(true);
188                 }
189         }
190
191         // check command retry count
192         while(commandTryCounter <
                INTERNAL_UART_COMMAND_SEND_ALL_DATA_PACKAGES_MAXIMUM_TRIES)
193         {
194                 commandTryCounter += 1U; // increment number of command tries
195                 receivedDataBuffer = 0U; // reset received data
196                 packageCounter = 0U;// reset number of received data packages
197
198                 internalUART_receiveLast(); // clear UART input buffer by reading
                        it until it is empty
199
200                 internalUART_outputAddress = *address;
201                 internalUART_outputData =
                        INTERNAL_UART_COMMAND_SEND_ALL_DATA_PACKAGES;
202                 internalUART_transmit();
203                 oneShotTimerStart(INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US,
                        false); // start timer for response timeout
204
205                 // receive data packages and update slave data array
```

```
206                 while(packageCounter < SLAVE_DATA_NUMBER_OF_PACKAGES)
207                 {
208                         // check and handle newly received command from the slave
209                         if(internalUART_receiveFirst()) // check for new received
                                data
210                         {
211                                 oneShotTimerElapsed(); // stop timer
212                                 packageCounter += 1U; // increment data package
                                        count
213                                 receivedDataBuffer |= internalUART_inputData &
                                        INTERNAL_UART_MASK_DATA; // save received data
                                         to local buffer
214
215                                 /**
216                                  * save data and clear receivedDataBuffer
217                                  * or
218                                  * shift receivedDataBuffer for the data from the
                                         next transmission
219                                  */
220                                 switch(packageCounter)
221                                 {
222                                 case (SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH /
                                        INTERNAL_UART_NUMBER_OF_BITS_DATA):
223
224 SLAVE_DATA_MASK_ROTARY_SWITCH); // save the rotary switch position
225                                 receivedDataBuffer = 0U; // clear buffer
226                                 break;
227                                 case ((SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH +
                                        SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_1) /
                                        INTERNAL_UART_NUMBER_OF_BITS_DATA):
228
```

```
229 SLAVE_DATA_MASK_POTENTIOMETER_1); // save the potentiometer 1 ADC value
230                                  receivedDataBuffer = 0U; // clear buffer
231                                  break;
232                                  case ((SLAVE_DATA_NUMBER_OF_BITS_ROTARY_SWITCH +
                                         SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_1 +
                                         SLAVE_DATA_NUMBER_OF_BITS_POTENTIOMETER_2) /
                                         INTERNAL_UART_NUMBER_OF_BITS_DATA):
233
234 SLAVE_DATA_MASK_POTENTIOMETER_2); // save the potentiometer 2 ADC value
235                                  receivedDataBuffer = 0U; // clear buffer
236                                  break;
237                                  default:
238                                          receivedDataBuffer <<=
                                             INTERNAL_UART_NUMBER_OF_BITS_DATA; //
                                             shift bits to make room for the data
                                             from the next transmission
239                                          break;
240                                  }
241
242                                  // there are still packages left to receive
243                                  if(packageCounter < SLAVE_DATA_NUMBER_OF_PACKAGES)
244                                  {
```

```
245                                     oneShotTimerStart(
                                            INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US
                                            , false); // start timer for response
                                            timeout of a following package
246                                 }
247                                 // all packages received
248                                 else
249                                 {
250                                     // wait for any follow-up packages to
                                            detect a faulty communication
251                                     oneShotTimerStart(
                                            INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US
                                            , true); // restart timer for response
                                             timeout and wait for it to elapse
252                                     // if data is available, the communication
                                             had problems and the received data is
                                             likely incorrect
253                                     if(internalUART_receiveLast())
254                                     {
255                                         packageCounter += 1U; // increment
                                                data package count to
                                               indicate a incorrect amount of
                                                received packages
256                                         break; // exit inner while loop
257                                     }
258                                     else
259                                     {
260                                         commandTryCounter =
                                               INTERNAL_UART_COMMAND_SEND_ALL_DATA_PACKAGES_MA
                                               ; // exit outer while loop
                                               since all packages have been
                                               received correctly and
                                               therefore no retry is
261 required
262                                     }
263                                 }
264                             }
265                             // check if INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US
                                    has elapsed since the last data package was received
                                    from the slave
266                             else if(!oneShotTimer.running)
267                             {
268                                 oneShotTimerStart(
                                        INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US,
                                        true); // start timer for response timeout and
                                         wait for it to elapse
269                                 // The following line is only required if ...
270                                 // ...at least one transmission has been received
                                        while waiting for the timeout,
271                                 // ...no retry will follow and
272                                 // ...it was the last slave to be contacted.
```

```
273                                 internalUART_receiveLast(); // clear FIFO input
                                        buffer
274                                 break; // exit inner while loop
275                         }
276                 }
277         }
278
279         // update connection state
280         // slave responded
281         if(packageCounter > 0U)
282         {
283                 // slave responded with the correct number of packages
284                 if(packageCounter == SLAVE_DATA_NUMBER_OF_PACKAGES)
285                 {
286                         slaveData[slaveDataArrayIndex].connectionProblems = false;
                                // indicate no connection problems
287                 }
288                 // slave responded with a incorrect number of packages
289                 else
290                 {
291                         slaveData[slaveDataArrayIndex].connectionProblems = true;
                                // indicate connection problems
292                 }
293
294                 // update connection state: existing/new connection
295                 if(!slaveData[slaveDataArrayIndex].connectionState)
296                 {
297                         slaveData[slaveDataArrayIndex].connectionState = true; //
                                indicate connection
298                         slaveData[slaveDataArrayIndex].connectionStateChanged =
                                true; // indicate new connection
299                 }
300                 else
301                 {
302                         slaveData[slaveDataArrayIndex].connectionStateChanged =
                                false; // if slave was connected before, indicate no
                                change
303                 }
304         }
305         // slave did not respond
306         else
307         {
308                 slaveData[slaveDataArrayIndex].connectionProblems = false; //
                        indicate no connection problems
309
310                 // update connection state: no/lost connection
311                 if(slaveData[slaveDataArrayIndex].connectionState)
312                 {
313                         slaveData[slaveDataArrayIndex].connectionState = false; //
                                indicate no connection
```

```
314                         slaveData[slaveDataArrayIndex].connectionStateChanged =
                                true; // indicate lost connection
315                 }
316                 else
317                 {
318                         slaveData[slaveDataArrayIndex].connectionStateChanged =
                                false; // if slave was not connected before, indicate
                                no change
319                 }
320         }
321 }
322
323
324
325 /**
326  * update all slave information
327  * check each slave and save the data sent by them unless aborted by the external
        device or slave connection loss
328  */
329 void slaveDataUpdateAll(void)
330 {
331         for(uint8_t idx = 0U; idx < slaveDataNumberOfElements; idx += 1U)
332         {
333                 // check and handle newly received command from the external
                        device
334                 if(externalUART_receiveFirst()) // check for new received data
335                 {
336                         // abort execution
337                         if(externalUART_inputData ==
                                EXTERNAL_UART_COMMANDMESSAGE_ABORT)
338                         {
339                                 externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
340                                 externalUART_outputData =
                                        EXTERNAL_UART_COMMANDMESSAGE_ABORT;
341                                 externalUART_transmit();
342
343                                 return; // exit function
344                         }
345                         // inform external device that command is being ignored
346                         else
347                         {
348                                 externalUART_outputAddress =
                                        EXTERNAL_UART_ADDRESS_EXTERNAL;
349                                 externalUART_outputData =
                                        EXTERNAL_UART_MESSAGE_COMMAND_IGNORED;
350                                 externalUART_transmit();
351                         }
352                 }
353                 else
354                 {
```

```
355                         slaveDataUpdate(&slaveData[idx].address);
356                 }
357         }
358 }
359
360
361
362
363 void udp_receive(void *arg, struct udp_pcb *pcb, struct pbuf *p_rx, ip_addr_t *
        addr, u16_t port)
364 {
365
366         DIGITAL_IO_ToggleOutput(&LED1); // Indicates Activity
367
368         uint8_t j;
369
370
371         if (p_rx != NULL) // check if buffer not empty
372         {
373
374                 char *recv_data;                                                //
                        received data string
375                 recv_data = (char *)p_rx->payload;              // get payload
                    from receive buffer
376
377                 char recv_addr[2];                                              //
                        aux var for received address
378                 char recv_cmd[2];                                               //
                        aux var for received command
379                 char recv_sladdr[2];                    // aux var for received
                    slave address
380
381
382                 strncpy(recv_addr, recv_data, 2);               // get address
383                 recv_data += 2;                                                 //
                        remove first two characters
384                 strncpy(recv_cmd, recv_data, 2);                // get command
385                 recv_data += 2;                                                 //
                        remove first two characters
386                 strncpy(recv_sladdr, recv_data,2);
387
388
389
390                 char send_data[16];             // string for data to be send //
                    Besser  ndern  zu char*
391
392
393                 if(strncmp(recv_addr, CELL_ADDR,2)==0){
                                        // check cell addr
394
```

```
395                         if(strncmp(recv_cmd, CMD_GET_SINGLE_DATA,2)==0){
                                                        // cmd: Get data of
                                specific cellcontroller
396
397
398                             if(strncmp(recv_sladdr, "01",2) == 0){
399                                     j = 0;
400                             }else if(strncmp(recv_sladdr, "02",2) == 0){
401                                     j = 1;
402                             }else if(strncmp(recv_sladdr, "03",2) == 0){
403                                     j = 2;
404                             }else if(strncmp(recv_sladdr, "04",2) == 0){
405                                     j = 3;
406                             }else if(strncmp(recv_sladdr, "05",2) == 0){
407                                     j = 4;
408                             }else if(strncmp(recv_sladdr, "06",2) == 0){
409                                     j = 5;
410                             }else if(strncmp(recv_sladdr, "07",2) == 0){
411                                     j = 6;
412                             }else if(strncmp(recv_sladdr, "08",2) == 0){
413                                     j = 7;
414                             }else if(strncmp(recv_sladdr, "09",2) == 0){
415                                     j = 8;
416                             }else if(strncmp(recv_sladdr, "10",2) == 0){
417                                     j = 9;
418                             }else if(strncmp(recv_sladdr, "11",2) == 0){
419                                     j = 10;
420                             }else if(strncmp(recv_sladdr, "12",2) == 0){
421                                     j = 11;
422                             }
423
424                             snprintf(send_data, (sizeof(send_data)+1), "
                                    0101%02X%02X%04X%04X",slaveData[j].address,
                                    slaveData[j].connectionState, slaveData[j].
                                    potentiometer1ADC, slaveData[j].
                                    potentiometer2ADC);  //
425 write data string
426                             p_tx = pbuf_alloc(PBUF_TRANSPORT, sizeof(send_data
                                ), PBUF_RAM);          // allocate transmit
                                buffer
427                             memcpy (p_tx->payload, send_data, sizeof(send_data
                                ));                             // copy msg to
                                buffer
428
429                             udp_sendto(pcb_tr,p_tx, &ip_hostpc, REMOTE_PORT);
                                 // send message via udp
430                         }
431                         else if(strncmp(recv_cmd, CMD_GET_DATA_CONNECTED,2)==0){

                                // cmd: get data of connected cellcontroller
432
```

```
433
434                                 for(j = 0U; j < 12U; j += 1U){
435                                         if(slaveData[j].connectionState == true)
436                                         {
437
438                                                 snprintf(send_data, (sizeof(
                                                        send_data)+1), "0103%02X%02X
                                                        %04X%04X",slaveData[j].address
                                                        , slaveData[j].connectionState
                                                        , slaveData[j].
                                                        potentiometer1ADC,
439 slaveData[j].potentiometer2ADC);          // write data string: cell addr 8bit, msg
        8bit, time 64bit
440                                                 p_tx = pbuf_alloc(PBUF_TRANSPORT,
                                                        sizeof(send_data), PBUF_RAM);
                                                                // allocate transmit
                                                        buffer
441                                                 memcpy (p_tx->payload, send_data,
                                                        sizeof(send_data));
                                                                        // copy msg to
                                                         buffer
442
443                                                 udp_sendto(pcb_tr,p_tx, &ip_hostpc
                                                        , REMOTE_PORT);
444
445                                                 pbuf_free(p_tx);  // free transmit
                                                         buffer
446                                         }
447
448                                 }
449                             }
450                             else if(strncmp(recv_cmd, CMD_GET_ALL_DATA,2)==0){
                                                                                    // cmd
                                    : get all data
451
452
453                                 for(j = 0U; j < 12U; j += 1U){
454                                         snprintf(send_data, (sizeof(send_data)+1),
                                                 "0103%02X%02X%04X%04X",slaveData[j].
                                                address, slaveData[j].connectionState,
                                                 slaveData[j].potentiometer1ADC,
                                                slaveData[j].potentiometer2ADC);
455 // write data string: cell addr 8bit, msg 8bit, time 64bit
456                                         p_tx = pbuf_alloc(PBUF_TRANSPORT, sizeof(
                                                send_data), PBUF_RAM);          //
                                                allocate transmit buffer
457                                         memcpy (p_tx->payload, send_data, sizeof(
                                                send_data));
                                                        // copy msg to buffer
458
```

```
459                                     udp_sendto(pcb_tr,p_tx, &ip_hostpc,
                                            REMOTE_PORT);
460
461                                     pbuf_free(p_tx);  // free transmit buffer
462
463
464                             }
465                         }
466                         else if(strncmp(recv_cmd, CMD_GET_ADDRESS_CONNECTED,2)==0)
                                {
                                // cmd: get address of connected cellcontrollers
467
468
469                             for(j = 0U; j < 12U; j += 1U){
470                                     if(slaveData[j].connectionState == true)
471                                     {
472
473                                             snprintf(send_data, (sizeof(
                                                    send_data)+1), "0104%012X",
                                                    slaveData[j].address);   //
                                                    write data string: cell addr 8
                                                    bit, msg 8bit, time 64bit
474                                         p_tx = pbuf_alloc(PBUF_TRANSPORT,
                                                sizeof(send_data), PBUF_RAM);
                                                        // allocate transmit
                                                buffer
475                                         memcpy (p_tx->payload, send_data,
                                                sizeof(send_data));
                                                                // copy msg to
                                                 buffer
476
477                                         udp_sendto(pcb_tr,p_tx, &ip_hostpc
                                                , REMOTE_PORT);
478
479                                         pbuf_free(p_tx);  // free transmit
                                                 buffer
480
481                                     }
482
483                             }
484                         }
485                         else{   // return command unknown
486
487
488                             snprintf(send_data, (sizeof(send_data)+1), "
                                    UNKNOWN CMD");       // write data string: cell
                                     addr 8bit, msg 8bit, time 64bit
489                             p_tx = pbuf_alloc(PBUF_TRANSPORT, sizeof(send_data
                                ), PBUF_RAM);          // allocate transmit
                                buffer
```

```
490                                 memcpy (p_tx->payload, send_data, sizeof(send_data
                                        ));                                 // copy msg to
                                        buffer
491
492                                 udp_sendto(pcb_tr,p_tx, &ip_hostpc, REMOTE_PORT);
493
494                         }
495
496                         pbuf_free(p_tx);        // Free transmit buffer
497
498                 }
499
500         }
501
502         pbuf_free(p_rx);        // Free receive buffer
503
504         DIGITAL_IO_ToggleOutput(&LED1); // Indicates end of activity
505
506 }
507
508
509
510 int main(void)
511 {
512
513
514
515         // variables for initialization of DAVE APPs
516         DAVE_STATUS_t dave_init_status;
517
518         // initialize DAVE APPs
519         dave_init_status = DAVE_Init();
520
521         // check whether initialization of DAVE APPs was successful
522         if(dave_init_status != DAVE_STATUS_SUCCESS)
523         {
524                 // error handler code
525                 XMC_DEBUG("\nDAVE APPs initialization failed\n");
526                 while(true);
527         }
528
529
530         // indicate activity
531         DIGITAL_IO_ToggleOutput(&LED2);
532
533
534         // create a software timer which generates a single callback event after
                TRANSCEIVER_STARTUP_TIME_IN_US
535         oneShotTimer.id = SYSTIMER_CreateTimer(TRANSCEIVER_STARTUP_TIME_IN_US,
                SYSTIMER_MODE_ONE_SHOT, (void*)oneShotTimerElapsed, NULL);
536
```

```
537         // check whether software timer was created successfully
538         if(oneShotTimer.id == 0U) // error handler code
539         {
540                 XMC_DEBUG("\nTimer creation failed\n");
541                 while(true);
542         }
543
544
545         // create a software timer which generates a cyclic callback event after
                SLAVE_DATA_UPDATE_TIME_IN_US
546         slavDataUpdatePeriodicTimerID = SYSTIMER_CreateTimer(
                SLAVE_DATA_UPDATE_TIME_IN_US, SYSTIMER_MODE_PERIODIC, (void*)
                slaveDataUpdatePeriodicTimerElapsed, NULL);
547
548         // check whether software timer was created successfully
549         if(slavDataUpdatePeriodicTimerID == 0U) // error handler code
550         {
551                 XMC_DEBUG("\nTimer creation failed\n");
552                 while(true);
553         }
554
555         // enable transceiver
556         DIGITAL_IO_SetOutputLow(&DO_TransceiverShutDown);
557
558         // start software timer
559         oneShotTimerStart(TRANSCEIVER_STARTUP_TIME_IN_US, false);
560
561         // initialize slave data array while waiting for transceiver to start
562         for(uint32_t idx = 0U, address = INTERNAL_UART_ADDRESS_SLAVE_START; idx <
                slaveDataNumberOfElements; idx += 1U, address += 1U)
563         {
564                 slaveData[idx].address = (uint8_t) address;
565                 slaveData[idx].connectionState = false;
566                 slaveData[idx].connectionStateChanged = false;
567                 slaveData[idx].rotarySwitch = 0U;
568                 slaveData[idx].potentiometer1ADC = 0U;
569                 slaveData[idx].potentiometer2ADC = 0U;
570         }
571
572         // wait for one shot timer to elapse and transceiver is ready
573         while(oneShotTimer.running);
574
575         // clear UART receive FIFO
576         internalUART_receiveLast();
577         externalUART_receiveLast();
578
579         // communicate initialization complete and reading of slave data
580         externalUART_transmitString("\nInitialization complete.\n");
581         externalUART_transmitString("\nReading slave data...");
582
```

```
583         // look for available slaves and get their data and set slaveData[idx].
                connectionStateChanged = false; as this was the first execution
584         slaveDataUpdateAll();
585         for(uint32_t idx = 0; idx < slaveDataNumberOfElements; idx += 1) {
                slaveData[idx].connectionStateChanged = false; }
586
587         // start software timer and check whether software timer start failed
588         if(SYSTIMER_StartTimer(slavDataUpdatePeriodicTimerID) !=
                SYSTIMER_STATUS_SUCCESS) // error handler code
589         {
590                 XMC_DEBUG("\nTimer start failed\n");
591                 while(true);
592         }
593
594         // send updated slave data to the external device if enabled
595         if(externalUART_autoUpdateSlaveData)
596         {
597                 sendSlaveDataString(false);
598         }
599
600         externalUART_transmitString("done.\n");
601         externalUART_transmitString("\nReady.\n");
602
603
604
605         pcb_tr = udp_new();
                                                                    // create UDP
                PCB
606         IP4_ADDR(&ip_hostpc, IP_ADDR0_PC,IP_ADDR1_PC,IP_ADDR2_PC,IP_ADDR3_PC); //
                set host pc ip-addr
607         udp_bind(pcb_tr, IP4_ADDR_ANY, LOCAL_PORT);
                                            // bind UDP PCB
608         udp_recv(pcb_tr, udp_receive, NULL);
609
610
611         // indicate activity
612         DIGITAL_IO_ToggleOutput(&LED2);
613
614         // cyclic application code
615         while (true)
616         {
617
618                 // indicate activity
619                 DIGITAL_IO_ToggleOutput(&LED2);
620
621                 // check whether any slave sent something when it wasn't requested
                        to ensure a free bus
622                 if(internalUART_receiveLast()) // check for new received data
623                 {
624
625
```

```
626                         /*
627                          * This has the potential for an endless waiting loop if a
                                 slave keeps sending data, but the bus must be free
                                as it is only half-duplex.
628                          * Sending an abort message probably doesn't work as the
                                bus is only half-duplex and the slave might receive
                                corrupted data or nothing at all,
629                          * since it ignores any data received while transmitting.
630                          * Waiting for the communication cycle time could also
                                cause an unresponsive user interface.
631                          */
632                         //oneShotTimerStart(
                                INTERNAL_UART_COMMAND_RESPONSE_TIMEOUT_IN_US, true);
                                // start timer for response timeout and wait for it to
                                elapse
633
634
635                 }
636                 // check and handle newly received command from the external
                        device
637
638                 else if(externalUART_receiveFirst()) // check for new received
                        data
639                 {
640
641                         // check target address
642                         if(externalUART_inputAddress ==
                                EXTERNAL_UART_ADDRESS_MASTER)
643                         {
644                                 // check received command
645                                 // abort command
646                                 if(externalUART_inputData ==
                                        EXTERNAL_UART_COMMANDMESSAGE_ABORT)
647                                 {
648                                         externalUART_outputAddress =
                                                EXTERNAL_UART_ADDRESS_EXTERNAL;
649                                         externalUART_outputData =
                                                EXTERNAL_UART_COMMANDMESSAGE_ABORT;
650                                         externalUART_transmit();
651                                 }
652                                 // ping command
653                                 else if(externalUART_inputData ==
                                        EXTERNAL_UART_COMMANDMESSAGE_PING)
654                                 {
655                                         externalUART_outputAddress =
                                                EXTERNAL_UART_ADDRESS_EXTERNAL;
656                                         externalUART_outputData =
                                                EXTERNAL_UART_COMMANDMESSAGE_PING;
657                                         externalUART_transmit();
658                                 }
659                                 // show addresses of all available slaves
```

```
660                     else if(externalUART_inputData ==
                            EXTERNAL_UART_COMMAND_SHOW_ALL_AVAILABLE_SLAVE_ADDRESSES
                            )
661                     {
662                         externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
663                         externalUART_outputData =
                                externalUART_inputData;
664                         externalUART_transmit(); // echo command
665                         slaveDataUpdateAll();
666                         showAllAvailableSlaveAddresses();
667                     }
668                     // show data of all available slaves
669                     else if(externalUART_inputData ==
                            EXTERNAL_UART_COMMAND_SHOW_DATA_OF_ALL_AVAILABLE_SLAVES
                            )
670                     {
671                         externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
672                         externalUART_outputData =
                                externalUART_inputData;
673                         externalUART_transmit(); // echo command
674                         slaveDataUpdateAll();
675                         sendSlaveDataString(true);
676                     }
677                     // show data of all slaves
678                     else if(externalUART_inputData ==
                            EXTERNAL_UART_COMMAND_SHOW_DATA_OF_ALL_SLAVES)
679                     {
680                         externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
681                         externalUART_outputData =
                                externalUART_inputData;
682                         externalUART_transmit(); // echo command
683                         slaveDataUpdateAll();
684                         sendSlaveDataString(false);
685                     }
686                     // toggle flag that indicates whether all slave
                            data shall be send to the external device
                            after it was updated
687                     else if(externalUART_inputData ==
                            EXTERNAL_UART_COMMAND_TOGGLE_AUTO_UPDATE_SLAVE_DATA
                            )
688                     {
689                         externalUART_outputAddress =
                                EXTERNAL_UART_ADDRESS_EXTERNAL;
690                         externalUART_outputData =
                                externalUART_inputData;
691                         externalUART_transmit(); // echo command
692                         externalUART_autoUpdateSlaveData = !
                                externalUART_autoUpdateSlaveData;
```

```
693                                     }
694                                     // unknown command
695                                     else
696                                     {
697                                             externalUART_outputAddress =
                                                    EXTERNAL_UART_ADDRESS_EXTERNAL;
698                                             externalUART_outputData =
                                                    EXTERNAL_UART_MESSAGE_UNKNOWN_COMMAND;
699                                             externalUART_transmit();
700                                     }
701                             }
702                             else // since the master was not addressed, forward
                                    message to internal UART and return answer of slave or
                                     timeout message
703                             {
704                                     forwardCommandAndWaitForAnswer();
705                             }
706
707                     }
708                     // if no new data has been received, check whether the user input
                            changed or the timer for update interval has elapsed and
                            update the external device (if enabled) and the LCD
709                     else
710                     {
711
712
713                             if(slaveDataUpdateFlag)
714                             {
715                                     // if timer for update interval has elapsed update
                                            slave data
716                                     if(slaveDataUpdateFlag)
717                                     {
718                                             slaveDataUpdateAll(); // get current data
                                                    from all slaves
719
720                                             slaveDataUpdateFlag = false; // reset flag
721
722                                             // send updated slave data to the external
                                                    device if enabled
723                                             if(externalUART_autoUpdateSlaveData)
724                                             {
725                                                     sendSlaveDataString(false);
726                                             }
727                                     }
728                             }
729
730
731                             sys_check_timeouts();           // required for UDP
732
733                             // stop indicating activity
734                             DIGITAL_IO_ToggleOutput(&LED2);
```

```
735
736                         }
737
738             }
739
740 }
```

A.3.2 Quelltext Matlab

Listing A.27: Quelltext der Matlab Software zur Steuerung des Batteriemodulcontrollers und zur Visualisierung der empfangenen Daten

breaklines

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Author:       Fabian Mahler
% Filename:     lwipp_command.m
% Date: 30.09.21
% Description: This is a code
%               to send different
%               commands to XMC4700 mu
%               and display received
%               data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


clear;
close all;
clc;

% clear old connections
if(exist('u0', 'var'))
    flush(u0,'output');
    clear u0
end

% obtain network information
% [~, result] = system('ipconfig')

transmission_error = 0; % transmission error flag

% set local host, local port, timeout
LocalIP = '192.168.0.5';
LocalPort = 45174;
% open UDP connection
u0 = udpport('IPV4', 'LocalHost', LocalIP, 'LocalPort', LocalPort, 'Timeout', 10);
configureTerminator(u0, 0);     % set NULL terminator for UDP socket
%u0  % uncomment to check settings

% set remote ip and remote port
% change values only in agreement with XMC4700
RemoteIP = '192.168.0.10';
RemotePort = 45175;


select_cellcontroller = 0; %set default
user_input = 0; % set default
cyclic_flag = 2; %set default
```

```
45
46
47
48
49 % user input with error handling
50 while user_input < 1 || user_input > 4
51     clc;
52     fprintf("*****************************************************************\n")
           ;
53     fprintf("| This program allows you to control the XMC4700 Master Module. |\n")
           ;
54     fprintf("| You can select a command by typing in a number shown below    |\n")
           ;
55     fprintf("*****************************************************************\n\n
           ");
56     fprintf("Available commands:\n");
57     fprintf("1 = get data of one specific controller\n");
58     fprintf("2 = get data of all connected controllers\n");
59     fprintf("3 = get all controller data\n");
60     fprintf("4 = get the addresses of the connected controllers\n");
61
62
63     % selection of command with error handling
64     user_input = input('\ntype in one number of above: ');
65
66     if user_input < 1 || user_input > 4
67         clc;
68         fprintf("Your input is exceeding limits, try again\n");
69         pause
70     end
71 end
72
73 % select cellcontroller with error handling
74 if user_input == 1
75     while select_cellcontroller < 1 || select_cellcontroller > 12
76         clc;
77         select_cellcontroller = input('Type in the number of the Controller you
               want to select: ');
78
79         if select_cellcontroller < 1 || select_cellcontroller > 12
80             clc;
81             fprintf("Your input is exceeding limits, try again\n");
82             pause
83         end
84     end
85 end
86
87 % assign user input to string
88 switch select_cellcontroller
89     case 1
90         select_cellcontroller = "01";
```

```
91      case 2
92          select_cellcontroller = "02";
93      case 3
94          select_cellcontroller = "03";
95      case 4
96          select_cellcontroller = "04";
97      case 5
98          select_cellcontroller = "05";
99      case 6
100         select_cellcontroller = "06";
101     case 7
102         select_cellcontroller = "07";
103     case 8
104         select_cellcontroller = "08";
105     case 9
106         select_cellcontroller = "09";
107     case 10
108         select_cellcontroller = "10";
109     case 11
110         select_cellcontroller = "11";
111     case 12
112         select_cellcontroller = "12";
113 end
114
115 % select query mode with error handling
116 while cyclic_flag > 1 || cyclic_flag < 0
117     clc;
118     cyclic_flag = input('Do you want a cyclic query? [y = 1/n = 0]: ');
119
120     if  cyclic_flag > 1 || cyclic_flag < 0
121         clc;
122         fprintf("Wrong input, try again");
123         pause
124     end
125 end
126
127
128 % assing and assemble message for mu
129 switch user_input
130     case 1
131         t_str = strcat("0101" , select_cellcontroller , "\0"); % cmd to get data
                of one specific controller
132     case 2
133         t_str = strcat("0102", "\0"); % cmd get data of all connected controller
134     case 3
135         t_str = strcat("0103" , "\0"); % cmd get all data
136     case 4
137         t_str = strcat("0104" , "\0"); % cmd get address of all cellcontrollers
                connected
138 end
139
```

```
140 pause(0.1);
141
142 % write command to mu
143 write(u0, t_str, RemoteIP, RemotePort)  % send command
144
145 % print headline for output
146 if user_input == 1 || user_input == 2 || user_input == 3
147     clc;
148     fprintf("Address | Connected | Potentiometer 1 | Potentiometer 2\n")
149     fprintf("--------+-----------+-----------------+-----------------\n");
150
151
152
153     while  1
154
155
156         % assign received data to array
157         data_recv = char(read(u0,16));
158
159         % mask the data and assign to variable
160         address = hex2dec(data_recv(6));
161         connected = hex2dec(data_recv(8));
162         pot1 = hex2dec(data_recv(9:12));
163         pot2 = hex2dec(data_recv(13:16));
164
165
166
167
168         % formattet output of data
169         fprintf("   %2d   |     %d     |      %4d       |       %4d       \n",
              address, connected, pot1, pot2);
170
171
172
173         % repeat write while data is available
174         if u0.NumBytesAvailable == 0
175
176             write(u0, t_str, RemoteIP, RemotePort)  % send command
177
178         end
179         pause(.18);
180
181
182         % print headline
183         if user_input == 1
184             if cyclic_flag == 0
185                 pause;
186             end
187             clc;
188             fprintf("Address | Connected | Potentiometer 1 | Potentiometer 2\n")
189             fprintf("--------+-----------+-----------------+-----------------\n");
```

```
190         end
191
192         % print headline
193         if(address == 12)
194             if cyclic_flag == 0
195                 pause;
196             end
197             clc;
198             fprintf("Address | Connected | Potentiometer 1 | Potentiometer 2\n")
199             fprintf("--------+-----------+-----------------+-----------------\n");
200         end
201
202     end
203
204
205 elseif user_input == 4
206
207     % print headline
208     fprintf("Connected Addresses\n")
209     fprintf("--------------------\n");
210
211
212
213     while 1
214
215         % assign received data to array
216         data_recv = char(read(u0,16));
217
218         % mask the data and assign to variable
219         address = hex2dec(data_recv(15:16));
220
221         % formatted output of received data
222         fprintf("    %2d     \n ", address);
223
224         % repeat write while data is available
225         if u0.NumBytesAvailable == 0
226
227             write(u0, t_str, RemoteIP, RemotePort)  % send command
228
229         end
230         pause(.18);
231
232         % print headline
233         if(address == 12)
234             if cyclic_flag == 0
235                 pause;
236             end
237             clc;
238             fprintf("Connected Addresses\n")
239             fprintf("--------------------\n");
240         end
```

```
241         end
242     end
```

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original