Betriebssoftware für eine Fahrplattform unter besonderer Berücksichtigung der Echtzeitbedingungen

Christoph Peltz

25. September 2009

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Bachelorarbeit

Betriebssoftware für eine Fahrplattform unter besonderer Berücksichtigung der Echtzeitbedingungen

von
Christoph Peltz

Aufgabenstellung und Betreuung:
Prof. Dr.-Ing. L. Wolf und Dipl.-Ing. Dieter Brökelmann.

Braunschweig, den 25. September 2009

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 25. September 2009

Kurzfassung

Diese Bachelorarbeit umfasst die Entwicklung, die Implementierung und die Untersuchung einer Betriebssoftware für eine vorgegebene Plattform mit Blick auf den späteren Verwendungszweck im Praktikum ”Programmierung eingebetteter Systeme”. Ziel ist es, die aktuell verwendete Betriebssoftware durch eine wartungsfreundlichere, performante und erweiterbare Alternative zu ersetzen. Dazu wird auch ein neues Protokoll entworfen, um die Befehlsverarbeitung und die Zusammensetzung dieser Befehle auf beiden Kommunikationsseiten einfach und effizient zu gestalten.

Abstract

This bachelorthesis contains the development, implementation and examination of the operation-software for a given platform in light of the later use in the internship ”Programming of embedded systems”. The goal of this is to replace the software currently in use with a software that is better maintainable, faster and easier to extend. For this purpose a new protocol was developed to make the processing and creating of orders simpler on both sides involved in the communication process.

[Hier wird später die Aufgabenstellung eingefügt.]

Inhaltsverzeichnis

1 Einleitung **1**

2 Überblick über die Hardware **3**
2.1 Einsatz im Praktikum 3
2.2 Die Motorplatine 4
2.2.1 Mikrocontroller 4
2.2.2 Ein-/Ausgabemöglichkeiten zur Praktikumsplatine 5
2.2.3 Interne Ein-/Ausgabe-Ports 5

3 Design und Design-Entscheidungen **7**
3.1 System 7
3.2 Debug 7
3.3 Drive, Motor und PID 8
3.4 IO, I2C und UART 8
3.5 Order und Queue 9
3.6 Timer, Parser und Options 9

4 Protokoll **11**
4.1 Grundlegende Konzepte 11
4.2 Eingebaute Befehle 12
4.2.1 Extended Instruction 12
4.2.2 Control 12
4.2.3 Query 13
4.2.4 Drive 13
4.2.5 Advanced Drive 13
4.2.6 SetPID 13
4.2.7 Option 13

5 Implementierung der Betriebssoftware **15**
5.1 Die Hauptschleife 15
5.1.1 process_orders() 15
5.1.2 lcd_update_screen() 15
5.1.3 parser_update() 16
5.1.4 queue_update() 16
5.2 Das Aktive Brems-System (ABS) 17
5.3 Befehle: Struktur und Funktionen 17
5.4 Datenpfad von Befehlen 18
5.5 Debug-Ausgaben 18
5.6 IO-Framework 19
5.7 Unterstützende Bibliothek für die Praktikumsplatine 20

6 Untersuchung des Laufzeitverhaltens **21**
6.1 Erste Messungen 21
6.2 Das LCD-Problem 22
6.3 Optimierung 23
6.3.1 Inlining von Funktionen 23
6.3.2 Eliminierung von Modulo-Operatoren 23
6.3.3 Effizienteres Initialisieren des Speichers 25
6.3.4 Bedingtes Kompilieren von Debugausgaben 25
6.4 Verlieren/Verpassen von Interrupts 25
6.5 Vergleich mit der Original-Software 26

7 Zusammenfassung und Ausblick **31**

Literaturverzeichnis **33**

A Module und ihre Beziehung **35**

Abbildungsverzeichnis

2.1 Die Motorplatine 4

3.1 Die Hauptschleife 8

5.1 Die Verteilerfunktion 15
5.2 Befehlsstruktur 17
5.3 order_init()-Funktion 18
5.4 order_copy()-Funktion 18
5.5 Debug-Defines und die Verwendung im Code 19

6.1 Die Hauptschleife mit Debugausgaben und Pin-Operationen 22
6.2 Vergleich von mehreren Befehlen 26

A.1 Die Module und ihre Beziehungen 35

Tabellenverzeichnis

4.1 Optionen des Control-Befehls 12
4.2 Optionen des Queue-Befehls 13

6.1 Benötigte Takte/Zeit für Pin-Operationen 21
6.2 Ergebnisse der ersten Messungen 27
6.3 LCD Ergebnis 27
6.4 Auswirkung von -finline-functions 28
6.5 Ergebnisse der io_get()-Messungen 28
6.6 Ergebnisse der order_init()-Optimierung 28
6.7 Vergleich der Speicheranforderung 28
6.8 Vergleich der Geschwindigkeiten 29

1 Einleitung

Im Praktikum ”Programmierung eingebetteter Systeme” wird eine Motorplatine [1] eingesetzt, damit die Studenten die Motoren benutzen können, um ihr Fahrzeug in Bewegung zu setzen. Diese Motorplatine benötigt für ihre Operation eine Betriebssoftware, die die Befehle der Praktikumsplatine entgegen nimmt, interpretiert und in Aktionen umsetzen kann. Die Entwicklung dieser Software und die qualitative Untersuchung ihres Laufzeitverhaltens ist Gegenstand dieser Arbeit.
Zuerst wird auf die Hardware eingegangen, für die die Betriebssoftware geschrieben wird, und die in einer eigenen Arbeit für diesen Einsatzzweck speziell entwickelt wurde. Der Mikrocontroller und die Kommunikationseinrichtungen werden genauer betrachtet, um einen Überblick über die Möglichkeiten zu haben, die die Motorplatine bietet.
Dann wird die Motorplatine mit der Hardware, die im Praktikum eingesetzt wird, in Verbindung gebracht. Das beinhaltet die Beziehung zwischen der Praktikumsplatine, dem WLAN-Modul und der Motorplatine. Dies ist die Plattform, die im Praktikum benutzt wird. Sie stellt somit die Umgebung dar, in der die Software eingesetzt wird. Die Software muss den Anforderungen dieser Umgebung genügen.
Für die Implementierung dieser Betriebssoftware wurde als Programmiersprache C [2] vorgegeben, deren Standard C99 ausgewählt wurde. Da diese Software für längere Zeit eingesetzt werden soll, waren die Wartbarkeit und die Möglichkeit, die Software einfach erweitern zu können, dominante Designaspekte. Kaum weniger wichtig war die Anforderung, dass die Software ihre Arbeit performant und zuverlässig ausführt.
Zur Untersuchung der Performance und der Zuverlässigkeit wurden ein Logikanalyzer und ein Oszilloskop benutzt, die die Flanken und deren Länge an nach außen gelegten Pins gemessen haben. Diese Pins wurden für die Untersuchung von der Betriebssoftware an wichtigen bzw. kritischen Stellen im Programmcode gesetzt und wieder gelöscht.

2 Überblick über die Hardware

Die Hardware, die in dieser Arbeit zum Einsatz kommt, wurde eigens für den Zweck des Praktikums ”Programmierung eingebetteter Systeme” entwickelt.

2.1 Einsatz im Praktikum

In diesem Praktikum wird den Studenten der Umgang mit kleinen Systemen näher gebracht, die sich wesentlich von den bekannten IBM-PC-kompatiblen Systemen unterscheiden. Das schließt die Rechenleistung, den Speicherplatz und die Kommunikationsmöglichkeiten mit ein. Der Großteil der Praktikanten kennt nur das Schreiben von Programmen, die auf Betriebssystemen laufen. Dies ist mit gewissen Vorteilen verbunden. Es gibt Dateien, der Speicher wird verwaltet und vieles mehr. Auf diesen kleinen Systemen, mit denen die Studenten im Praktikum konfrontiert werden, läuft kein Betriebssystem, sondern lediglich ein Bootloader. Dieser ermöglicht es, neue Programme auf die Platine zu laden, um diese dann ausführen zu können. Im Laufe des Praktikums lernen die Studenten, wie man Programme in der Sprache C schreibt und wie die Timer sowie andere angeschlossene Geräte verwendet werden, zu denen z.B. ein LCD und ein Servomotor zählen. Außerdem wird den Studenten vermittelt, wie der I2C-Bus funktioniert und wie man diesen benutzt.
Im zweiten Teil des Praktikums können die Studenten sich für ein Projekt entscheiden, welches sie durchführen möchten (die Projektideen kommen von den Studenten). Als Plattform für ihre Projekte wird ihnen ein Fahrzeug zur Verfügung gestellt. Dieses Fahrzeug besitzt die aus dem ersten Teil des Praktikums bekannte Platine (von nun an Praktikumsplatine genannt) und ein WLAN-Modul, das das Programmieren der Praktikumsplatine ohne Kabel ermöglicht. Außerdem können hierüber Daten mit der Praktikumsplatine während der Laufzeit ausgetauscht werden. Zusätzlich besitzt das Fahrzeug eine Motorplatine mit zugehörigen Motoren und Rädern, die zur Fortbewegung des Fahrzeugs genutzt werden. Die Motorplatine wird mithilfe von Befehlen, welche die Praktikumsplatine sendet, gesteuert, und diese wiederum steuert die Motoren.
Die Motorplatine, für die die Betriebssoftware in der vorliegenden Arbeit entwickelt wurde, wird im Allgemeinen von den Studenten nur benutzt, aber nicht modifiziert (auch wenn ihnen dies frei steht). Die Leistungen, die die Motorplatine bereitstellt, sollen einfach anzusprechen sein, damit die Studenten sich auf das Implementieren ihres eigenen Projektes konzentrieren können.

2.2 Die Motorplatine

Die Motorplatine wurde im Zuge der Studienarbeit von Timo Klingeberg [1] entwickelt.

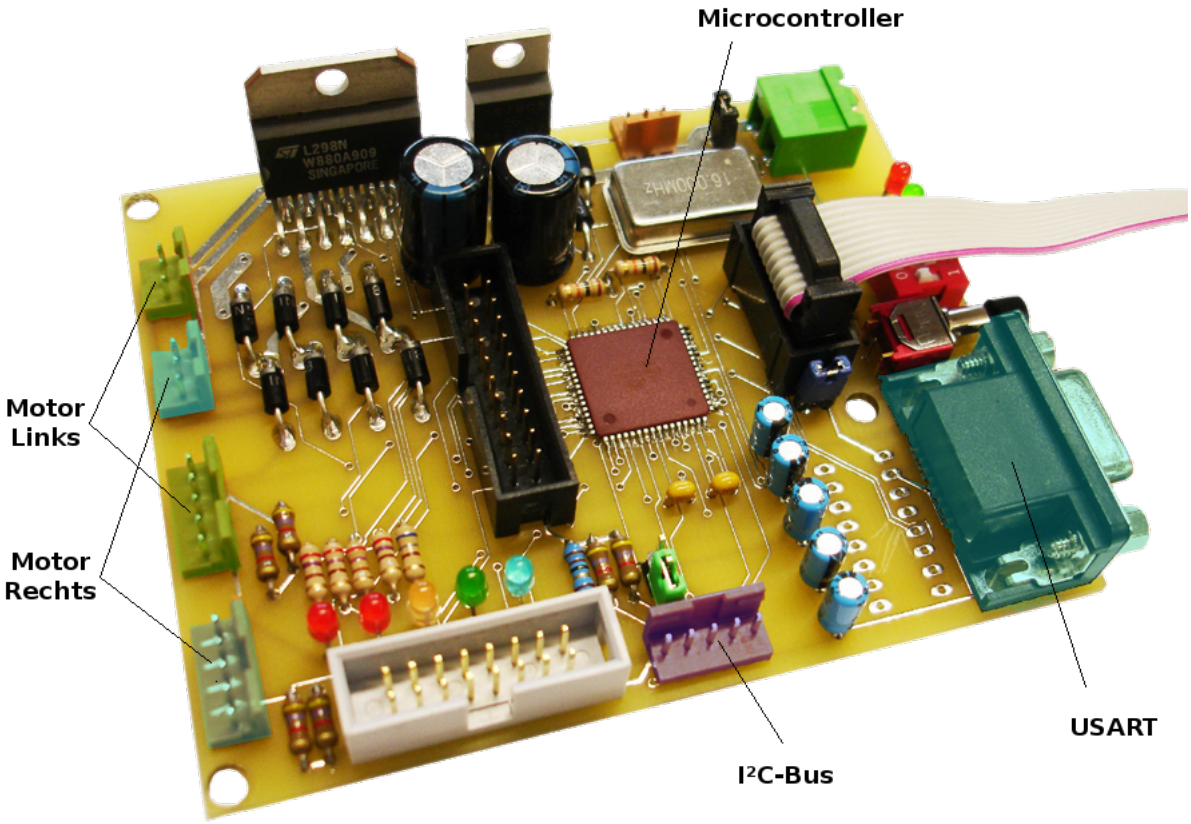

Abbildung 2.1: Die Motorplatine

2.2.1 Mikrocontroller

Das Herzstück der Platine bildet ein Mikrocontroller der Firma Atmel. Es handelt sich hierbei um einen ATMEGA2561[3], der 256-KiB-Speicher für Programme (Flash) hat, sowie 8-KiB-Speicher für Variablen (SRAM). Der maximale Takt für diesen Mikrocontroller liegt bei 16 MHz, der auch ausgenutzt wird. Das bedeutet, dass ein Takt 62,5 ns benötigt. Da der Großteil der Instruktionen des Mikrocontrollers nur einen Takt benötigt, kann dieser theoretisch 16 MIPS leisten. Durch die Benutzung von Funktionen, Pin-IO und bedingten Sprüngen bleibt dies aber nur eine theoretische Zahl. Vorteilhafterweise unterstützt der Controller das ISP (In-System Programming). Das war bei der Entwicklung der Betriebssoftware von großem Nutzen, da hierdurch schnell und relativ unkompliziert neue Versionen auf die Platine überspielt werden konnten.
Der Controller verfügt über 6 Timer. Zwei von diesen haben nur 8 Bit, die anderen vier 16 Bit. Das entspricht bei dem gegebenen Takt von 16 MHz einem maximalen

Timer-Intervall von 262 ms. Zusätzlich kann der Controller sechs Pulsweitenmodulationen betreiben, von denen zwei für die Servomotoren benutzt werden, die die Räder antreiben.

2.2.2 Ein-/Ausgabemöglichkeiten zur Praktikumsplatine

Die Motorplatine verfügt sowohl über einen UART-Port, als auch einen I2C-Bus [4], über die die Praktikumsplatine mit der Motorplatine kommunizieren kann. Der UART-Port wird außerdem für die Ausgabe von Debug-Informationen benutzt, falls diese aktiviert wird. Im Allgemeinen wird allerdings nur der I2C-Bus zur Kommunikation zwischen den beiden Platinen verwendet. Da der Mikrocontroller über eingebaute Hardwarelogiken für den I2C-Bus und auch den UART-Port verfügt, hält sich der administrative Aufwand für die Kommunikation in engen Grenzen. Es können lediglich Interrupt Service Routinen (ISR) für diese Kommunikations-Einrichtungen zur Verfügung gestellt werden. Dadurch ist es möglich, schnell auf Situationen zu reagieren, ohne Informationen verlieren zu können.

2.2.3 Interne Ein-/Ausgabe-Ports

Zusätzlich zu den externen Kommunikationsmöglichkeiten besitzt der ATMEGA2561 54 programmierbare IO Kanäle, die in Ports mit je 8 Kanälen zusammenfasst werden. 16 von diesen Kanälen sind für die Steuerung der Servomotoren zuständig, die die Räder und die Hall-Sensoren betreiben. Diese Sensoren, die an den Motoren befestigt sind, lösen bei Bewegung der Räder Unterbrechungen aus. Damit liefern sie Informationen, aus denen auf die Drehrichtung der Räder und die Strecke, die zurückgelegt wurde, geschlossen werden kann. Für jede Umdrehung eines Rades werden 360 Interrupts ausgelöst, d.h. für jedes Grad ein Interrupt. Je nach Größe des Raddurchmessers ist eine Steckenauflösung von wenigen Millimetern möglich.

3 Design und Design-Entscheidungen

Die Betriebssoftware wurde in Module aufgeteilt. Ein Modul kann mehrere Code-Dateien umfassen. Eine Code-Datei ist aber nur einem Modul zugeordnet. Es wurde besonderer Wert darauf gelegt, dass Module so wenig wie möglich andere Module aufrufen, und dies auch nur durch Funktionen, nicht durch die Variablen. Dieses Prinzip musste allerdings während der Optimierungs- und Testphase geringfügig modifiziert werden. Entweder wurde der Code dadurch unleserlich, oder die Relation Preis zu Nutzen je Funktionsaufruf unangemessen hoch. Als wesentliche Verbesserung wurden vier bis fünf globale Variablen eingeführt, die sich während der Laufzeit ändern können, und die in unterschiedlichen Modulen direkt referenziert werden. Geändert werden diese allerdings nur in sehr wenigen Funktionen, in denen dies auch explizit dokumentiert wurde. Außerdem wurde in den Coding-Guidelines darauf hingewiesen, möglichst keine Funktionen zu schreiben, die die globalen Variablen verändern. Anderenfalls ist dies ausdrücklich hervorzuheben. Neben diesen maximal fünf globalen Variablen, die das Systemverhalten verändern, gibt es vier weitere Variablen in zwei verschiedenen Modulen, die jeweils aus einem anderen Modul gesetzt werden. Dies sind die Trigger-Werte für die Position und die Zeit. Diese werden nur durch den Drive bzw. den Advanced Drive Befehl gesetzt und während der Motorunterbrechungen nach und nach dekrementiert. Auf die Module wird in den Unterkapiteln genauer eingegangen.

3.1 System

Das als System bezeichnete Modul beinhaltet die Hauptschleife der Betriebssoftware und die Initialisierung aller anderen Untermodule. Es benutzt nur wenige Module, um seine Aufgabe in einem abstrakten Maße zu erfüllen. Während der Hauptschleife, die in gekürzter Fassung (d.h., ohne Kommentare und Debug-Ausgaben) in Abb. 3.1 zu sehen ist, werden nur einige wenige Funktionen aufgerufen. Dadurch werden die wichtigen Module auf dem aktuellen Stand gehalten. Außerdem wird es ermöglicht, dass Daten zwischen den Modulen fließen können.

3.2 Debug

Eigene Mechanismen zum Debugging sind gerade dann wichtig, wenn man dies nicht mit den gewohnten Werkzeugen durchführen kann. Es ist unabdingbar wichtig zu wissen, welche Vorgänge sich in dem kleinen Mikrocontroller während der Laufzeit abspielen. Dies ist aber nicht ganz einfach, denn die einzigen Möglichkeiten, die die Platine zur Kommunikation mit der Außenwelt hat, beschränken sich auf 5 LED, ein

```
while(1) {
    copy_timer_flags();

    process_orders();

    if (LCD_PRESENT)
        lcd_update_screen();

    parser_update();

    queue_update();
}
```

Abbildung 3.1: Die Hauptschleife

4*20 Zeichen LCD, eine UART und eine I2C-Bus Schnittstelle. Die LED funktionieren nicht zuverlässig und das LCD ist in seinem Aussagegehalt sehr begrenzt, weil man nicht viele Informationen auf einem 4*20 Zeichen LCD unterbringen kann. Die Wahl fiel dann auf die UART Schnittstelle, da der Arbeitscomputer, an dem das Debugging durchgeführt wurde, einen solchen Anschluss besitzt, aber über keinen I2C Anschluss verfügt. Mithilfe der Debugausgaben kann man protokollieren, was das System in jedem Schleifendurchlauf getan hat, und dadurch das Verhalten analysieren, um schlussendlich Fehler aufzuspüren.

3.3 Drive, Motor und PID

Das Motor bindet zum einen das System an die Motoren und an die anderen nötigen Fahrelemente an, wohingegen das Drive Modul die Services, die das Motor Modul anbietet abstrahiert und in einer weise Zusammenfasst, die es dem System erlaubt auf sehr einfache Art und Weise die Motoren zu bedienen. Das PID Modul, welches größtenteils aus der vorhergegangen Studienarbeit übernommen wurde [1], ist für den Fehlerausgleich der Radbewegungen zuständig. Genauer: Es errechnet die korrigierten Werte für die Räder, die dann vom Drive Modul an diese auch weiter geleitet werden. Die Fehler die auftreten können sind vielfältig und nicht besonders groß, wie zum Beispiel eine kleine Varianz in der Geschwindigkeit eines Rades oder ähnliches. Damit aber das Fahr-verhalten stabiler wird müssen diese Fehler korrigiert werden.

3.4 IO, I2C und UART

Das IO Modul stellt für das System einheitliche Funktionen zur Verfügung um Daten zu lesen als auch zu schreiben. Hierbei kann es dem Benutzer der Funktionen egal sein, welche Schnittstelle letztendlich für die Kommunikation genutzt wird. Das IO Modul kümmert sich um die Unterschiede zwischen UART und I2C. Sowohl das I2C als auch das UART Modul kümmern sich hauptsächlich um das Initialisieren der Hardware und das behandeln der Interrupt Service Routinen.

3.5 Order und Queue

Das Order Modul ist das größte aller Module. Es beinhaltet zum einen den Typ, mit dem Befehle intern dargestellt und verarbeitet werden. Aufbauend auf den Typ, dem einige unterstützende Funktionen zugeteilt sind um wiederkehrende Aufgaben zu erleichtern, existiert im Order Modul auch die Order Funktionen. Diese Order Funktionen sind Handler-Funktionen für die im Protokoll spezifizierten Befehle. Falls ein Befehl länger benötigt, bis er als beendet gelten kann, wird die entsprechende Order Funktion mit dem korrespondierenden Befehl in jeder Iteration der Hauptschleife aufgerufen. Diese kann dann eventuell Wartungsarbeiten an dem Befehl durchführen und überprüfen, ob dieser beendet ist und entsprechend seinen Status ändern. Durch diesen Aufbau ist eine Iteration der Hauptschleife sehr kurz, aber auch komplizierte Befehle oder solche deren Parameter und Durchführung überwacht werden müssen sind hierdurch möglich.
Durch das Queue Modul ist es möglich der Motorplatine mehrere Befehle direkt hintereinander zu übermitteln, die dann einer nach dem anderen ausgeführt werden, außerdem kümmert die Queue sich darum, das Prioritäts-Befehle nicht angereiht werden sondern bei der nächsten Iteration der Hauptschleife ausgeführt werden.

3.6 Timer, Parser und Options

Diese drei Module haben hauptsächlich eine unterstützende Funktion, so bringt das Timer Modul die Möglichkeit mit in bestimmten zeitlichen Intervallen Befehle auszuführen. Der Parser fasst einzelne Bytes logisch zu Befehlen zusammen und übergibt diese der Queue. Das Options Modul beinhaltet die Einstellungen, die das Verhalten des gesamten Systems beeinflussen, wie z.B. ob Debugging Ausgaben aktiviert sind, mit was für einer Geschwindigkeit das ABS bremst und einiges mehr.

4 Protokoll

Die Praktikumsplatine muss, um das Fahrzeug in Bewegung zu setzen, mit der Motorplatine kommunizieren. Diese Kommunikation muss sollte möglichst effizient erfolgen. Deswegen ist die Entwicklung eines guten Protokolls sehr wichtig für das gesamte System.
Das Verläufer-Protokoll benutzte Zeichenketten, um Instruktionen zu übermitteln. Dies hat zwei wichtige Nachteile. Der erste ist, dass das Zusammensetzen der Zeichenketten mithilfe der verfügbaren Prozessoren, sehr viel Rechenleistung in Anspruch nimmt. Der zweite Nachteil ist, dass durch die Verwendung der Zeichenketten die Informationsdichte der Instruktionen nicht sehr hoch ist und damit zusätzliche Wartezeit bei der Übermittlung anfällt.

4.1 Grundlegende Konzepte

Da die Kommunikationseinrichtungen der Hardware Byte-orientiert sind, wurde das Protokoll ebenfalls Byte-orientiert aufgebaut. Im Gegensatz zu dem Vorgänger-Protokoll werden also keine Zeichenketten benutzt, um die Informationen zu repräsentieren, sondern nur der Wert der einzelnen Bytes.
Ein Befehl ist eine Folge von n Bytes, wobei n mindestens 1 und maximal die im Code eingestellte Größe der Befehlsstruktur entspricht. Von den eingebauten Befehlen besitzt keiner eine maximale Länge von mehr als 9 Bytes. Das erste Byte eines Befehls hat eine besondere Bedeutung. Dieses Byte wird Kommando-Byte genannt. Dieses Kommando-Byte ist in zwei Teile geteilt. Der erste Teil umfasst die niederwertigsten 4 Bit und wird Befehlscode genannt. Dieser Befehlscode spezifiziert die Art des Befehls. Der zweite Teil beinhaltet die höchstwertigen 4 Bit. In ihm werden Optionen angegeben, die den im Befehlscode angegebenen Befehl modifizieren. Die Kombination von Befehlscode und Optionen legt auch die Länge des Befehls fest. Alle auf dem Kommando-Byte folgende Bytes sind Parameter, wie z.B. die Geschwindigkeit der Räder.
Da 4 Bit für Befehlscodes zur Verfügung stehen, sind 16 verschiedene Befehle möglich. Sechs Befehle wurden im Zuge dieser Arbeit implementiert, das würde noch Platz für 10 weitere Befehle lassen. Das Protokoll sollte allerdings noch mehr Freiheiten für zukünftige Erweiterungen bieten, deswegen wurde einer der Befehlscodes reserviert. Dieser reservierte Befehlscode und dessen Behandlung im Code, ermöglichen es, dass es mehr als ein Kommando-Byte gibt. Damit ist es möglich Befehle einfach hinzuzufügen, die nicht auf das ”ein Kommando-Byte, viele Parameter”-Schema passen.
Wenn Parameter übertragen werden müssen, die mehr als ein Byte benötigen, wird zuerst das höherwertigste Byte übertragen. Danach absteigend nach der Wertigkeit die anderen Bytes.
Durch die volle Ausnutzung der Bytes gibt es kein Protokoll-spezifisches STOP- oder

START-Byte. Das bedeutet, dass das Protokoll sich auf die Flusskontrolle der zugrunde liegende Hardware verlässt.

4.2 Eingebaute Befehle

Sechs Befehle und die Infrastruktur für den reservierten Befehlscode wurden implementiert. Im nachfolgenden werden die Befehle einzeln vorgestellt. Dabei wurd der Befehlscode bei jeden Befehl in hexadezimaler Schreibweise als ganzes Byte dargestellt.

4.2.1 Extended Instruction

Dieser Befehl ist ein Platzhalter für zukünftige Befehle, die mehr als ein Kommandobyte benötigen, oder wenn alle normalen Befehlscodes bereits vergeben sind. Der Befehlscode ist 0x00.
Im Code werden Befehle dieser Art mithilfe von besonderen Funktionen behandelt. Die parser_extended_order_complete-Funktion ist ein Beispiel hierfür. Diese besonderen Versionen von Funktionen gibt es allerdings nur im Parser, danach wird eine Extended Instruction genauso behandelt wie ein normaler Befehl. Diese zusätzlichen Funktionen waren nötig, da sich diese Befehle im Aufbau sehr von normalen Befehlen unterscheiden.

4.2.2 Control

Mithilfe dieses Befehls kann das System gesteuert werden. Darunter fallen Aufgaben wie das Reseten des gesamten Programs, das Anhalten der Befehlsausführung und die Manipulation der Befehls-Warteschlange. Der Befehlscode für diesen Befehl ist 0x01. Außerdem ist der Control-Befehl ein priorisierter Befehl, d.h. er wird bei dem nächsten Hauptschleifendurchlauf ausgeführt und nicht an die Warteschlange angereiht.
Die in Tabelle 4.1 beschriebene Bitmaske wird mit dem Befehlscode verundet und ergibt das Kommandobyte. Bei manchen Befehlen können mehrere Optionen zur gleichen Zeit gewählt werden, dies ist hier nicht der Fall. Die Optionen dieses Befehlsschließen sich gegenseitig aus. Die Länge der Befehle mit diesem Befehlscode beläuft sich auf 1 Byte, also lediglich das Kommandobyte muss gesendet werden.

Option	**Bitmaske**	**Beschreibung**
Reset	0x10	Resetet die Hardware
Stop Queue	0x20	Der Aktuelle Befehl wird verworfen. Queue wird angehalten, aber nicht verworfen.
Continue Queue	0x30	Der nächste Befehl in der Queue wird ausgeführt.
Clear Queue	0x40	Die Warteschlange wird gelöscht.
Stop Drive	0x50	Befehl wird angehalten, Fahrzeug geht zum aktiven Bremsen über.

Tabelle 4.1: Optionen des Control-Befehls

4.2.3 Query

Manchmal ist es für die kontrollierende Praktikumsplatine wichtig verschiedene Laufzeitwerte der Motorplatine zu kennen. Dieser Befehl, dem der Befehlscode 0x02 zugeordnet ist, ermöglicht es die Geschwindigkeit der Räder, die Anzahl der Befehle in der Warteschlange und den aktuellen Befehl abzufragen. Die Optionen schließen sich gegenseitig aus und die Länge des Befehls beträgt immer 1 Byte. Der Befehl ist wie der Control-Befehl ein priorisierter Befehl.
Wenn dieser Befehl an die Motorplatine gesendet wurde und kurz darauf versucht wird über den I2C-Bus das Ergebniss zu lesen, kann es vorkommen, dass das Ergebniss noch nicht bereit ist. Dies ist insbesondere der Fall, wenn der aktuelle Befehl angefordert wurde. In diesem Fall muss die Lese-Operation nochmal gestartet werden.
Damit die Praktikumsplatine etwas mit dem aktuellen Befehl, den die Motorplatine zurückgibt, anfangen kann, muss die Länge des Befehls mit übertragen werden. Deswegen ist es nötig, dass die Praktikumsplatinen zwei Lese-Operationen durchführt. Die erste hat eine Länge von einem Byte und spezifiziert wie lang die zweite Antwort in Bytes ist.

Bitmaske	**Beschreibung**	**Antwortlänge**
0x10	Geschwindigkeit des linken Rades	1
0x20	Geschwindigkeit des rechten Rades	1
0x30	Anzahl der Befehl in der Queue	1
0x40	Aktueller Befehl	2 - 16

Tabelle 4.2: Optionen des Queue-Befehls

4.2.4 Drive

4.2.5 Advanced Drive

4.2.6 SetPID

4.2.7 Option

5 Implementierung der Betriebssoftware

Die Betriebssoftware wurde, wie in der Aufgabenstellung festgelegt in C geschrieben. Hierbei fiel die Wahl auf den Standard C99, der mit einigen sprachlichen Aktualisierungen gegenüber C89 aufwarten kann. Als Compiler wurde eine Version der GNU Compiler Collection (GCC) mit einem Backend für AVR-kompatiblen Assembler benutzt. Außer der C Standard Library und der AVR IO Library besitzt die Software keine externen Abhängigkeiten im Code.

5.1 Die Hauptschleife

Die Hauptschleife ist in dieser Implementierung eine Endlosschleife, da die Beendigung dieser Schleife dazu führen würde, dass das System nicht mehr reagiert. Wie in Abb. 3.1 zu sehen ist, werden in der Hauptschleife vier wichtige Funktionen aufgerufen.

5.1.1 process_orders()

Die process_orders()-Funktion bearbeitet die Befehle, die bereits in der Queue sind. Dafür holt sich die Funktion den aktuellen Befehl von der Queue. Falls es solch einen Befehl gibt ruft die Funktion eine Verteilerfunktion auf. Diese wiederum ruft die zugehörige Befehlsfunktion auf, indem die untersten vier Bits des ersten Befehlsbytes als Index für eine Call-Table benutzt werden (siehe Abb. 5.1).
Falls kein Befehl vorliegt oder die Queue angehalten wurde, ruft die process_orders()-

```
// That means, call the right function
if(order_array[order->data[0] & 0x0f] != 0) {
    order_array[order->data[0] & 0x0f](order);
} else { // Set the Order status to done if there is no function for this order
    order->status |= ORDER_STATUS_DONE;
}
```

Abbildung 5.1: Die Verteilerfunktion

Funktion die Funktion zum aktiven Bremsen auf. Das aktive Bremsen wird in später noch diskutiert.

5.1.2 lcd_update_screen()

In dem Fall, dass ein LCD angeschlossen ist, können dort Informationen ausgegeben werden. Ob ein LCD angeschlossen ist wird über die Stellung eines DIP-Schalters auf

der Platine geregelt.
Da das synchrone Aktualisieren des LCD sehr viel Zeit benötigt (vgl. ¡Untersuchung-LCD-Problem¿), wird während dieser Funktion maximal ein Zeichen an das LCD geschickt. Dies wird erreicht indem das Busy-Flag, welches signalisiert, dass das LCD noch beschäftigt ist, abgefragt wird. Falls es nicht gesetzt ist und es noch Daten zum aktualisieren gibt, wird das nächste Zeichen an das LCD gesendet.
Auf dem LCD werden die Versionsnummer der Betriebssoftware, der Status einiger system-weiter Variablen und der aktuelle ausgeführte Befehl angezeigt. Falls nun ein Befehl bearbeitet wird, der länger als einen Schleifendurchlauf benötigt (das sind z.B. alle Fahr-Befehle), ruft die lcd_update_screen()-Funktion die lcd_update_info()-Funktion auf, die diese Informationen in einem Puffer konstruiert. Nach und nach gibt die lcd_update_screen()-Funktion den Inhalt dieses Puffer an das LCD weiter.
Befehle die innerhalb eines Hauptschleifendurchlaufs abgearbeitet sind werden nicht ausgegeben und generieren auch keinen Aufruf von lcd_update_info(). Dies ist nötig, da diese Befehle zu schnell abgearbeitet werden. Es können vier bis fünf dieser Befehle abgearbeitet werden, bevor das LCD auch nur einmal vollständig aktualisiert werden kann.

5.1.3 parser_update()

Der Parser ist dafür zuständig aus den Bytes, die über I2C oder UART gelesen werden, Befehle in Form von order_t-Strukturen zu erstellen. Die parser_update()-Funktion fragt beim IO-Modul nach, wie viele Bytes zum Abholen bereit stehen. Diese werden dann geholt und an die Funktion parser_add_byte() übergeben.
Diese parser_add_byte()-Funktion fügt das Byte an die korrekte Stelle im Puffer ein. Wenn ein Befehl komplett ist, was mit der parser_order_complete()-Funktion überprüft wird, gilt der Befehl als fertig und alle weiteren Bytes, die hinzugefügt werden, landen in einer neuen order_t-Struktur.
Damit erkannt werden kann, wann ein Befehl zu ende ist benutzt die parser_order_complete()-Funktion die bytes_needed()-Funktion, in der fest codiert ist, welcher Befehlscode mit welchen Optionen wie viele Bytes benötigt. Dies ist auch eine der Stellen, die angepasst werden müssen, wenn ein neuer Befehl hinzugefügt wird, oder bestehende verändert werden.

5.1.4 queue_update()

Diese Funktion führt Wartungsarbeiten an der Befehlswarteschlange (Queue) durch. Dies beinhaltet neue Befehle beim Parser-Modul abzuholen und diese korrekt einzureihen. Es gibt zwei Möglichkeiten, wie die Queue diese neuen Befehle einreihen kann. Zum einen als normale Befehle, diese werden einer nach dem anderen abgearbeitet, zum anderen als priorisierte Befehl. Es kann nur ein priorisierter Befehl in der Queue sein. Diese Befehle werden umgehend in der nächsten Hauptschleifeniteration ausgeführt. In die Kategorie der priorisierten Befehle fallen alle Queue-Kontroll-Befehle, wie z.B. pausieren, löschen, aktuellen Befehl verwerfen, etc. (vgl. Protokoll-Spezifikation im Anhang).

5.2 Das Aktive Brems-System (ABS)

Das aktive Bremssystem bewirkt, dass die Räder versuchen ihre Position nicht zu verlassen. Dies wird realisiert indem eine Referenz-Position für jedes Rad gespeichert wird. Während der Hauptschleife wird nun die tatsächliche Position mit der Referenz-Position verglichen. In dem Fall, dass diese Positionen nicht übereinstimmen, werden die Motoren mit einer einstellbaren Geschwindigkeit betrieben, um die Räder wieder auf die Referenz-Position zu bringen.
Die Referenz-Positionen werden an drei verschiedenen Stellen im Code gesetzt. Zum einen in der process_orders()-Funktion in der Hauptschleife, wenn der aktuelle Befehl beendet wurde, zum anderen in den Fahr-Befehls-Funktionen, falls ein Rad früher als das andere seine Stopp-Bedingung erreicht hat.
Das ABS kann vom Benutzer während das System läuft, angepasst werden. So kann man die Geschwindigkeit ändern, mit der die Motoren die Positions-Differenz ausgleichen. Außerdem kann man Teile des ABS deaktivieren oder auch wieder reaktivieren sowie das gesamte ABS abschalten, bzw. wieder anschalten. Damit kann der Benutzer das ABS an seine Wünsche anpassen.

5.3 Befehle: Struktur und Funktionen

Die Struktur (Abb. 5.2), die einen Befehl im System repräsentiert, besteht hauptsächlich aus einem Array, in dem die eigentlichen Daten gespeichert sind, und einem Status-Byte, in dem Statusinformationen in Form von Flags gespeichert werden. Das erste

```
typedef struct ORDER {
    uint8_t data[ORDER_TYPE_MAX_LENGTH];
    uint8_t status;
} order_t;
```

Abbildung 5.2: Befehlsstruktur

Byte dieses Arrays ist das Typ-Byte, das die Art des Befehls und die zugehörigen Optionen spezifiziert. Der Befehlscode (die unteren vier Bits des Typ-Bytes) 0 ist für zukünftige Erweiterungen Reserviert, die mehr als ein Typ-Byte benötigen. Des weiteren sind die Befehlscodes 1 bis 6 durch diese Arbeit bereits definiert und mit Funktionalität erfüllt. Die Befehlscodes 7 bis 15 sind noch nicht definiert und können für zukünftige Erweiterungen benutzt werden, die mit einem Typ-Byte auskommen.
Alle auf das Typ-Byte (oder die Typ-Bytes im Falle des Befehlscodes 0) folgenden Bytes sind Parameter. Die Anzahl und Länge dieser hängt von der Spezifikation des Befehls ab. Grundsätzlich gilt aber, dass bei Parametern, mit mehr als ein Byte Länge, zuerst das höchstwertige Byte im Array gespeichert wird, dann absteigend bis zum niederwertigsten Byte.
Oft benutze Aktionen bezüglich der Befehlsstruktur wurden zusammengefasst (siehe Abb. 5.3 und 5.4). Vor der Untersuchung des Laufzeitverhaltens und der damit einhergehenden Optimierungen, wurden diese beiden Funktionen durch for-Schleifen implementiert, was sich aber als langsamer als die Standard C Funktionen memset() und memcpy() herausstellte.

```
void order_init(order_t *order) {
    // Function to Initialize a order_t structure
    // Overwrite every piece of memory with 0
    memset(order->data, 0, ORDER_TYPE_MAX_LENGTH);
    // Set the Status of the order to Initialized
    order->status = ORDER_STATUS_INITIALIZED;
}
```

Abbildung 5.3: order_init()-Funktion

```
void order_copy(const order_t * const from, order_t *to) {
    // Used to copy the order data from one to another
    memcpy(to->data, from->data, ORDER_TYPE_MAX_LENGTH);
    to->status = from->status;
}
```

Abbildung 5.4: order_copy()-Funktion

5.4 Datenpfad von Befehlen

Befehle werden entweder über die UART oder die I2C Schnittstelle byteweise empfangen. Diese Schnittstellen werden über Interrupt Service Routinen bearbeitet, um zeitnah auf eingehende Daten zu reagieren, da diese Übertragung die größte Latenz-Quelle darstellt (I2C: ca. 270 µs pro Byte; UART: ca 139 /textmus pro Byte). In diesen Interrupt Service Routinen wird das empfangene Byte in den Eingangspuffer gelegt. Wenn die Hauptschleife wieder die parser_update()-Funktion erreicht, wird die bisher empfangene Bytes abgeholt und in der Parser-Puffer abgelegt, um aus den Bytes order_t Struktur Instanzen zu generieren. Wenn der Befehl fertig im Parser vorliegt, ruft ihn queue_update() ab und reiht ihn in die Warteschlange ein (siehe Kapitel 5.1.4). Von der Warteschlange holt sich die process_orders()-Funktion den aktuellen Befehl und führt die zugeordnete order_function-Funktion solange aus, bis im Status-Byte (vgl. Abb. 5.2) das ORDER_STATUS_DONE Flag gesetzt wurde und entfernt diese dann aus der Warteschlange.
Falls der Befehl, der bearbeitet wird, eine Ausgabe von Daten bewirkt, wird in der entsprechenden order_function-Funktion die auszugebenden Daten in den Ausgabepuffer des IO-Framework angereiht. Dieses Framework wird dann, sobald möglich, diese Daten über die ausgewählte Schnittstelle senden (bei UART wird sofort angefangen zu senden, bei I2C muss gewartet werden, bis die Daten vom Benutzer abgerufen werden).

5.5 Debug-Ausgaben

Das Problem mit Debug-Ausgaben ist, dass sie notwendig sind, insbesondere wenn die Benutzung eines üblichen Debuggers nicht ohne weiteres möglich ist. Debug-Ausgaben benötigen entsprechenden Code, nicht nur um die Ausgaben überhaupt zu ermöglichen, sonder auch an vielen Stellen im Code, um wirklich etwas zu bestimmten Zeitpunkten auszugeben. Diese Ausgaben, selbst wenn sie nicht gemacht werden müssen, indem sie mit if-Statements umschlossen werden (siehe Abb. 5.5), verlangsamen das gesamte System. Während der normalen Operation der Betriebssoftware im

Praktikum, sind diese Ausgaben nicht nötig und für die meisten Teilnehmer des Praktikums, nicht informativ. Damit die Ausgaben zum Zweck der Fehlerbehebung benutzt werden können, muss der Leser eine entsprechende Kenntnisse des Codes vorweisen. Aufgrund dieser fragwürdigen Hilfe, die diese Ausgaben einem durchschnittlichen Praktikanten bieten, und der negativen Auswirkung der Ausgaben auf die Performance des Systems, wurde ein Prä-Prozessor Trick angewandt, der zusammen mit der eingestellten Stufe der Code-Optimierung des Compilers diese beiden Probleme löst. Durch diesen Trick werden die Debug-Ausgaben nur dann mit kompiliert, wenn dem Compiler der Parameter -DDEBUG übergeben wird. Selbst dann ist es noch möglich mit einem DIP-Schalter auf der Platine die Ausgaben auszuschalten, dann hat man aber immer noch die leichten negativen Auswirkungen auf die Performance.

```
#ifndef DEBUG
    #define DEBUG_ENABLE 0
#endif
-----------------------------------------------
#ifdef DEBUG

uint8_t DEBUG_ENABLE;
#endif
-----------------------------------------------
if (DEBUG_ENABLE) {
    debug_WriteInteger(PSTR("queue.c : queue_update() : order opcode is = "), local_order.data[0]);
    debug_WriteInteger(PSTR("queue.c : queue_update() : order status is = "), local_order.status);
}
```

Abbildung 5.5: Debug-Defines und die Verwendung im Code

5.6 IO-Framework

Das Ziel des IO-Frameworks war die Abstraktion der Ein- und Ausgabe von Daten von den zu Grunde liegenden Schnittstellen. Die UART und die I2C Schnittstelle sind in der Benutzung sehr unterschiedlich. Statt überall im Code wo E/A stattfindet jeweils für beide Schnittstellen Code einzufügen, wurde das IO-Framework als Zwischenschicht entwickelt. Es verfügt über sowohl über einen Ausgabe- als auch über einen Eingabepuffer. Diese sind jeweils auf 256 Bytes festgelegt. Durch diese Definition konnte das normale Überlaufverhalten der 8-Bit-Variablen ausgenutzt werden, um Modulo-Operationen zu ersetzen, welche ungerechtfertigt viel Zeit in Anspruch nehmen.
Eine Besonderheit ist die Objekt-basierte Ausgabe von Daten. Ein Objekt hat mindestens ein Byte und maximal 256 Byte. Objekte werden bei entweder komplett Übertragen oder gar nicht. Wenn ein Objekt nicht komplett übertragen werden konnte, wird bei der nächsten Übertragen vom Anfang der Objektes wieder angefangen. Außerdem bewirkt die Ausgabe von Objekten bei Benutzung der I2C Schnittstelle, dass für jede Lese-Operation, die von der Praktikumsplatine eingeleitet wird, ein Objekt übermittelt wird.
E/A-Operationen geschehen nicht-blockend und gepuffert. Damit verbraucht die E/A nur Prozessor-zeit, wenn es nötig ist und verschwendet diese nicht mit aktiven Warten.

5.7 Unterstützende Bibliothek für die Praktikumsplatine

6 Untersuchung des Laufzeitverhaltens

Die Untersuchung der Laufzeitverhaltens wurde mithilfe eines Logik- Analysers durchgeführt. Ein Logikanalyser misst wie lange und wann ein Pin 1 und/oder 0 ist. Die Pins, die nötig sind um diese Messungen durchzuführen, dürfen noch nicht belegt sein. Vorteilhafterweise besitzt die Platine zwei Ports, die noch nicht belegt sind aber trotzdem nach draußen gelegt wurden, somit konnte der Logik- Analyser an die Platine angeschlossen und ein kleines Modul geschrieben werden, um diese Pins setzen und wieder löschen zu können. Diese Operationen wurden als Makros implementiert um den Messoverhead so gering wie möglich zu halten. Wie hier beschrieben sind die Zeiten für

Makro	benötigte Takte	benötigte Zeit bei 16 MHz
pin_set()	2	125 ns
pin_clear()	2	125 ns
pin_toggle()	4	250 ns

Tabelle 6.1: Benötigte Takte/Zeit für Pin-Operationen

das Ausführen der Instruktion zum setzen, löschen und umschalten von einzelnen Pins ziemlich gering. Doch bei den Messungen insbesondere mit einem Leistungsfähigen und sehr genauen Oszilloskop konnte herausgefunden werden, dass das eigentliche Wechsel des Stroms am Pin verhältnismäßig langsam durchgeführt wird, insbesondere das Abfallen des Stromes, also bei einer fallenden Flanke benötigt ungefähr 2 µs von denen allerdings, und hier liegt das Problem, zwischen 0.5 und 1 µs fälschlicherweise als ”high” gemessen wird. D.h. der Logikanalyser misst eine gewisse Zeitspanne einen ”falschen” Wert. (Er ist nicht physikalisch falsch nur logisch). Denn wenn der Strom abfällt ist der bin schon nicht mehr gesetzt, der Logikanalyser allerdings betrachtet dies teilweise immer noch als gesetzt. Aufgrund dieses Umstandes als auch der Tatsache, dass das System nicht untersucht werden kann ohne einen geringen Fußabdruck zu hinterlassen, sind die Messungen mit einem abschätzbaren aber nicht genau vorhersagbaren Fehler im Vergleich zur Wirklichkeit behaftet.

6.1 Erste Messungen

Durch die ersten Messungen wurde der Startpunkt für die Untersuchung der Software festgelegt. Dafür wurde in der Hauptschleife (siehe Abb. 6.1) zum einen ein pin_toggle() eingebaut, um die Länge einer Schleifeniteration zu messen, zum anderen wurde die einzelnen Funktionsaufrufe in der Hauptschleife mit pin_set() und pin_clear() umgeben. In der Tabelle 6.2 sind die Wichtigsten Daten der ersten Messrei-

```
while(1) {
    pin_toggle_A(0);
    // Copy global timer flags to a local copy, which will be used throughout the program.
    // This is done to not miss a timer tick.
    local_time_flags = timer_global_flags;
    timer_global_flags = 0;

    // Processes the next or current order
    if (DEBUG_ENABLE)
        debug_WriteString_P(PSTR("main.c : main() :  process_orders()\n"));
    pin_set_A(1);
    process_orders();
    pin_clear_A(1);

    // If a LCD is pluged in we get nice status messages on it
    pin_set_A(2);
    if (LCD_PRESENT)
        lcd_update_screen();
    pin_clear_A(2);

    // Update the order parser
    if (DEBUG_ENABLE)
        debug_WriteString_P(PSTR("main.c : main() :  parser_update()\n"));
    pin_set_A(3);
    parser_update();
    pin_clear_A(3);

    // Housekeeping for the order queue
    if (DEBUG_ENABLE)
        debug_WriteString_P(PSTR("main.c : main() :  queue_update()\n"));
    pin_set_A(4);
    queue_update();
    pin_clear_A(4);
}
```

Abbildung 6.1: Die Hauptschleife mit Debugausgaben und Pin-Operationen

hen zusammen gefasst. Die Spalte ”Rahmenbedingungen” beschreibt Bedingungen während der Messung geherrscht haben. Hierbei bezeichnet die Bedingung ”AB:EIT”, dass das aktive Bremsen (AB:) eingeschaltet (E; Enable) ist und sowohl aktiviert wird, wenn kein Befehl bearbeitet wird(I; Idle) oder eines der Räder seinen Trigger früher erreicht hat als das andere (T; Trigger). ”I2C an” bezeichnet, dass der I2C-Bus für die E/A-Operationen verwendet wurde und nicht die UART Schnittstelle.
Interessant ist bei diesen Daten, dass das aktualisieren der Warteschlange, wenn ein Befehl vorliegt, gut eine halbe Millisekunde benötigt.

6.2 Das LCD-Problem

Wie in der Tabelle 6.2 zu sehen ist, benötigt das System zum aktualisieren der Informationen des LCD über drei Millisekunden. Damit ist diese Funktion die mit Abstand zeit intensivste im gesamten System. Zwar ist das Aktualisieren recht selten, nämlich nur, wenn ein neuer Befehl gestartet wird, aber haben drei Millisekunden Latenz eine negative Wirkung auf die Reaktion des Systems auf Ereignisse. So können während der drei Millisekunden ungefähr zwölf Bytes auf dem I2C Bus eintreffen, was im schlimmsten Fall sechs eigenständige Befehle wären. Wenn diese Befehle priorisierte Befehle sind, so könnte sich das System unerwartet verhalten.
Damit dieses Problem gelöst werden konnte, musste die Arbeitsweise des LCD, welches durch eine fertige Programm-Bibliothek gesteuert wird, analysiert werden. Da-

durch wurde herausgefunden, dass die Funktionen, die Daten auf das LCD ausgeben, aktives Warten betreiben. Ein LCD benötigt natürlich Zeit, um ein übermitteltes Zeichen auch anzeigen zu können. Während dieser Zeit wird das sog. Busy-Flag gesetzt, welches anzeigt, dass das LCD noch beschäftigt ist. Erst wenn dieses Flag nicht mehr gesetzt ist darf ein neues Zeichen übermittelt werden. Beim aktiven Warten verbraucht die Funktion unnötig CPU-Zeit, die sinnvoll genutzt werden könnte.
Mit diesem Wissen wurde die ursprüngliche lcd_update_screen()-Funktion in zwei Funktionen aufgeteilt. Die eine wurde lcd_update_info() genannt und aktualisierte die LCD-Informationen in einem Speicherpuffer. Die andere, immer noch lcd_update_screen() genannt, überprüft das Busy-Flag des LCD's und schreibt das nächste Zeichen aus dem Speicherpuffer aufs LCD, wenn es nicht gesetzt ist. Wie in der Tabelle 6.3 zu sehen, hatten diese Maßnahmen ihre gewünschte Wirkung. Über 98% weniger Zeit wird nun im zeitintensivsten Fall benötigt. Dies hat seinen Nachteil in einer leicht erhöhten Laufzeit für das Aktualisieren des LCD im Ruhezustand. Diese Zeit hat sich von durchschnittlich 4 µs auf 9 µs erhöht.

6.3 Optimierung

Damit die Latenz des Systems möglichst niedrig ist, müssen die Funktionen, die in der Hauptschleife aufgerufen werden, unter allen Bedingungen möglichst effizient ihre Arbeit verrichten. Dazu wurde jede dieser Funktionen, mit dem Pin setzen und wieder löschen Verfahren, auf Möglichkeiten untersucht sie zu optimieren.

6.3.1 Inlining von Funktionen

Die erste Möglichkeit, die in Betracht gezogen wurde, um die Latenz des Systems zu verringern, war die Verwendung von Compiler-Optimierungen. Zu Beginn wurde das System auf Code-Größe optimiert (-Os Option des Compilers). Da das System wesentlich weniger Speicher benötigt als auf der Platine zur Verfügung stehen, wurde nun der Compiler auf die Optimierung der Geschwindigkeit eingestellt (-O2 Option). Außerdem wurden die Debug-Informationen, die der Compiler dort platziert hat, entfernt (-g Option entfernt). Dadurch änderten sich die Parameter des Systems nicht wesentlich. Die Größe blieb nahezu identisch und die Geschwindigkeit wurde etwas besser, besonders bei langen Funktionen.
Nachdem verifiziert werden konnte, dass Funktionsaufrufe zwischen 1 und 2 µs Overhead mit sich bringen, wurde die Compiler-Option -finline-functions aktiviert, die bewirkt, dass der Code genügend simpler Funktionen direkt in die aufrufende Funktion eingefügt wird, ohne dass dabei Register gerettet werden müssten oder ähnliches. Dies hatte deutlich merkbare Konsequenzen für die Geschwindigkeit des gesamten Systems, wie man in Tabelle 6.4 sehen kann.

6.3.2 Eliminierung von Modulo-Operatoren

Während der Untersuchung der parser_update()-Funktion wurde festgestellt, dass eine innere Funktion unverhältnismäßig viel Zeit benötigt, um ihre Aufgabe zu erfüllen. Dies war die io_get()-Funktion, deren Aufgabe es ist das nächste Byte im Eingangspuffer an den Aufrufer zurück zu liefern. Für diese simple Operation benötigte über 36

µs. Mehr als die gesamte Hauptschleife im Idle Zustand. Eine genaue Untersuchung der Funktion lies darauf schließen, dass die Anweisungen mit Modulo-Operationen für diesen Aufwand verantwortlich waren. Dies konnte zum einen durch Kontrolle des generierten Assembler Codes, zum anderen durch das Auslagern der Modulo-Operation in eine eigene Zeile bestätigt werden. Der Compiler generiert für die Modulo-Operation eine extra Funktion, die sehr kompliziert ist und offensichtlich eine Menge Zeit benötigt.

```
uint8_t io_get(uint8_t* value) {
  pin_set_C(6);
  if ((inpos_begin + 1) % IO_INBUFFER_SIZE == inpos_end)
    return 0;
  pin_clear_C(6);
  pin_set_C(7);
  *value = in_buffer[inpos_begin];
  pin_clear_C(7);
  pin_set_C(1);
  inpos_begin = (inpos_begin + 1) % IO_INBUFFER_SIZE;
  pin_clear_C(1);
  return 1;
}
```

Es gibt zwei Möglichkeiten dies zu korrigieren: Zum einen kann die Modulo-Operation durch eine Reihe simplerer Operationen ersetzt werden. Zum anderen kann auf die Modulo-Operation komplett verzichtet werden, wenn man das Überlaufverhalten der Variablen ausnutzen kann. Die zweite Methode setzt voraus, dass der Puffer so viele Elemente hat wie der größte darstellbare Wert der Variablen plus eins. Da hier Variablen vom Typ uint8_t verwendet werden (unsigned 8-Bit integer; vorzeichenlose 8-Bit Ganzzahl) muss der Puffer 256 Elemente umfassen. Da diese Zahl bereits eingestellt war und diese Funktion sehr häufig verwendet wird, wurde hier die zweite Methode benutzt. Es wurde dadurch nicht mehr Speicher verwendet als vorher, aber die Methode lief daraufhin wesentlich schneller. Der Code sieht nun wie folgt aus:

```
uint8_t io_get(uint8_t* value) {
  uint8_t temp = (inpos_begin + 1);
  if (temp == inpos_end)
    return 0;
  *value = in_buffer[inpos_begin];
  inpos_begin = temp;
  return 1;
}
```

Das selbe Problem existierte in der Queue, dort wurde allerdings die Modulo Operationen

```
queue_writeposition %= QUEUE_SIZE;
```

durch simplere Operationen ersetzt.

```
queue_writeposition -= (queue_writeposition
  / QUEUE_SIZE) * QUEUE_SIZE;
```

6.3.3 Effizienteres Initialisieren des Speichers

Die Funktionen zum initialisieren und kopieren von Befehlen wurden, aufgrund ihrer häufigen Verwendung ebenfalls untersucht. Die Befehlsstrukturen wurden einfach mithilfe von Schleifen mit dem Wert 0 initialisiert und beim Kopieren wurde ebenfalls mit einer Schleife die Werte von einem Befehl in den Nächsten kopiert. Diese Schleifen wurden durch die memset()- bzw. die memcpy()-Funktion aus der Standardbibliothek ersetzt. Dies führte zu einer 40%igen Geschwindigkeitszuwachs (siehe 6.6).

6.3.4 Bedingtes Kompilieren von Debugausgaben

Wie in Kapitel 5.5 beschrieben wurde ein Sprachen-spezifischer Trick verwendet, um die Möglichkeit zu besitzen einfach Diagnoseausgaben auszugeben, die im normalen Betrieb keine Auswirkung auf das Laufzeitverhalten des Systems haben. So generiert der Compiler für die Ausgaben keinen Code, solange nicht das Compilerflag -DDEBUG gegeben ist. Selbst wenn es gegeben ist und die Ausgaben mithilfe des entsprechenden DIP-Schalters deaktiviert sind, bringt jede Ausgabe lediglich weniger als eine µs größere Laufzeit.

6.4 Verlieren/Verpassen von Interrupts

Während den Messungen kam die Frage auf, was passiert, wenn ein Interrupt auftritt, wenn ein anderer bereits läuft. Wenn dies geschehen kann, was passiert mit den beiden Interrupts, geht eventuell sogar einer verloren?
Die Antwort auf diese Frage wurde von dem Handbuch des Mikrocontrollers [3] beantwortet. Mit den richtigen Einstellungen, die bereits eingestellt waren, werden Interrupts, die während eines anderen Interrupts auftreten, gespeichert und nach der Beendigung des laufenden Interrupts werden die gespeicherten ihrer Priorität nach ausgeführt.
Der schnellste Interrupt tritt alle 138 µs auf, das ist der UART-Interrupt. Im schlimmsten Fall treten alle Interrupts zur selben Zeit auf und der am schnellsten wieder auftretende hat die niedrigste Priorität. Er wird also erst nachdem alle anderen Interrupts abgearbeitet wurden ausgeführt. Bei vier Hall-Sensoren-Interrupts, einem I2C-Bus-Interrupt, zwei Timer-Interrupts und zwei UART-Interrupts und einer oberen Schranke von 8 µs pro Interrupt plus 2 µs Umschaltungszeit zwischen zwei Interrupts kommt man auf eine Zeit von 90 µs. Das bedeutet, dass bevor der schnellste Interrupt wieder auftreten kann, wurden alle Interrupt-Service-Routinen abgearbeitet und mindestens ein Hauptschleifendurchlauf beendet. Anzumerken ist hier noch, dass dieser konstruierte schlimmste Fall unmöglich ist. Entweder ist der UART-Empfang-Interrupt oder der I2C-Bus-Interrupt ausgeschaltet, sie sind niemals zur selben Zeit aktiv. Außerdem benötigt der Großteil der Interrupts weniger als 5 µs, um seine Arbeit zu beenden. Ganz zu schweigen davon, dass die Hall-Sensoren-Interrupts, durch ihre Anordnung an den Rädern, nicht alle zur selben Zeit auftreten können.
Die Hall-Sensoren-Interrupts treten sehr häufig auf, wenn die Räder in Bewegung sind. Die Auswirkung dieser Interrupts auf das Echtzeitverhalten des Systems wurde daher untersucht. Dabei wurde durch Messungen herausgefunden, dass bei einer Spannung von 12 V und maximaler eingestellter Geschwindigkeit der Räder, die Hall-Sensoren

jeder alle 1,7 ms einen Interrupt generieren. Das sind 1700 µs und dieser Interrupt benötigt maximal 8 µs. Während dieser 1700 µs treten insgesamt 4 Interrupts auf, jeweils einer von jedem Sensor, d.h. alle 425 µs wird ein Interrupt der Länge mit 8 µs. Das bedeutet, dass zwischen zwei Interrepts der Hall-Sensoren im Durschnitt 9 bis 10 Hauptschleifeniterationen durchgeführt werden.

6.5 Vergleich mit der Original-Software

Der harte Vergleich mit Nummern der Original-Software mit der in dieser Arbeit geschriebenen Software ist nur schwer möglich, da das zugrunde liegende Design von beiden Projekten so unterschiedlich ist, dass hauptsächlich die Dauer der Hauptschleife eine Idee davon geben kann, inwiefern die eine Software schneller arbeitet als die Andere (siehe Tabelle 6.8).
Zusätzlich kann man noch den Speicherfußabdruck der Systeme vergleichen (siehe Tabelle 6.7). Hierbei fällt auf, dass der generierte Maschinencode im Vergleich zum Original etwas größer ist, was aber unter Anbetracht der benutzten Compiler-Flags bedeutet, dass der Code mit gleichen Flags kleiner wäre als das Original. Dies ist auch demonstriert worden, indem die Original-Software mit den Compiler-Flags der jetzigen Software übersetzt wurde. Zusätzlich wurde noch verglichen, wie lange ein Befehl von der Praktikumsplatine bis zur Motorplatine benötigt. Hier ist die Übertragungsgeschwindigkeit des I2C-Busses das größte Hindernis und dank der höheren Datendichte der Befehlesübertragung im neuen System ist dieses wesentlich schneller (vgl. Tabelle 6.8 und Abb. 6.2).

```
'42'  'D'  'N'     'N'     ','  '1'  '2'  '7'  ','  '1'  '2'  '7'  '\0'
'42'  '3'  '127'   '127'


'42'  'D'   'P'     'P'     ','     '-'    '1'    '2'    '7'  ','  '-'  '1'
'42'  '163'  '-127'  '-127'  '-127'  '-127' '-127' '-127'
'2'  '7'  ','  '-'  '3'  '2'  '7'  '6'  '8'  ','  '-'  '3'  '2'  '7'  '6'  '8'  '\0'
```

Abbildung 6.2: Vergleich von mehreren Befehlen

Funktion	Zeit	Rahmenbedingungen
Hauptschleife	27,916 µs	Idle, LCD an, DEBUG aus, I2C an, AB:EIT
process_orders()	11,964 µs	
lcd_update_screen()	3,988 µs	
parser_update()	3,988 µs	
queue_update()	3,988 µs	

Hauptschleife	47,856 µs	ABS aktiv im Idle Status, ansonsten wie oben
process_orders()	31,904 µs	
lcd_update_screen()	3,988 µs	
parser_update()	3,988 µs	
queue_update()	3,988 µs	

Hauptschleife	718,843 µs	LCD aus, Byte in Parser, Befehl bereit
process_orders()	39,880 µs	I2C-Bus ISR (Daten) aufgetreten
lcd_update_screen()	0,997 µs	
parser_update()	175,474 µs	
queue_update()	505,484 µs	

Hauptschleife	3614,000 µs	LCD an, Befehl in Queue, aber nicht gestartet
process_orders()	358,923 µs	
lcd_update_screen()	3250,000 µs	
parser_update()	6,032 µs	
queue_update()	5,583 µs	

I2C-Bus ISR	4,586 µs	Adresse empfangen
I2C-Bus ISR	21,934 µs	Daten empfangen
100ms ISR	8,076 µs	

Tabelle 6.2: Ergebnisse der ersten Messungen

Funktion	Zeit	Verbesserung	Rahmenbedingungen
Hauptschleife	421,158 µs	-88,32%	LCD an, Befehl in Queue, aber nicht gestartet
process_orders()	354,627 µs		
lcd_update_screen()	57,884 µs	-98,22%	

Tabelle 6.3: LCD Ergebnis

Funktion	Zeit	Verbesserung	Rahmenbedingungen
Hauptschleife	420,659 µs		LCD an, Befehl in Queue, aber nicht gestartet
process_orders()	355,289 µs		
lcd_update_screen()	59,481 µs		
parser_update()	4,146 µs		
queue_update()	4,640 µs		
Hauptschleife	285,643 µs	-32,1%	LCD an, Befehl in Queue, aber nicht gestartet
process_orders()	224,836 µs	-36,72%	
lcd_update_screen()	51,647 µs	-13,17%	
parser_update()	5,384 µs	+29,86%	
queue_update()	4,187 µs	-9,76%	

Tabelle 6.4: Auswirkung von -finline-functions

Funktion	Zeit	Verbesserung
io_get()	36,527 µs	
io_get() (umgeschrieben)	3,393 µs	75,19%

Tabelle 6.5: Ergebnisse der io_get()-Messungen

Funktion	Zeit	Verbesserung
order_init()	13,968 µs	
order_init() (memset)	7,976 µs	42,9%

Tabelle 6.6: Ergebnisse der order_init()-Optimierung

Speicherart	Belegung
Program (Original)	14968 Bytes (5,7%)
Data (Original)	1952 Bytes (23,8%)
Program (Original + mod. Makefile)	17808 Bytes (6,8%)
Data (Original + mod. Makefile)	1952 Bytes (23,8%)
Program (Jetzt)	15362 Bytes (5,9%)
Data (Jetzt)	1248 Bytes (15,2%)

Tabelle 6.7: Vergleich der Speicheranforderung

Funktion	**Zeit**	**Verb.**	**Bedingung**
Hauptschleife (Original)	121,885 µs		Idle
Hauptschleife (mod Orig)	121,635 µs	0,20%	Idle
Hauptschleife (Jetzt)	23,535 µs	80,69%	Idle
Befehl (kurz, Original)	3655 µs (13 Byte)		I2C-Bus, kürzester Befehl
Befehl (kurz, Jetzt)	1023 µs (4 Byte)		I2C-Bus, kürzester Befehl
Befehl (lang, Original)	7747 µs (27 Byte)		I2C-Bus, längster Befehl
Befehl (lang, Jetzt)	2191 µs (8 Byte)		I2C-Bus, längster Befehl

Tabelle 6.8: Vergleich der Geschwindigkeiten

7 Zusammenfassung und Ausblick

Das Ergebnis dieser Arbeit ist, dass die Platine auf Befehle schnell und korrekt reagiert, solange diese nicht gravierend von den Spezifikation abweichen. Es gehen keine Interrupts verloren, und die Platine reagiert auf Änderungen in Bruchteilen einer Sekunde.
Dennoch gibt es einige Punkte, die aufgrund der begrenzten Zeit nicht getestet oder implementiert werden konnten.
So ist beispielsweise das Verhalten des ABS abhängig von der Dauer der Hauptschleife. Normalerweise ist dies kein Problem. Wenn allerdings Debug-Ausgaben vorgenommen werden, kann das ABS sich auf eine Art und Weise verhalten, die nicht erwünscht ist (dauerndes hin und her schwenken der Räder, da kein Schleifendurchlauf während des Nullpunktes stattfindet). Um dies zu verhindern, kann man einen Timer-Interrupt mit einer möglichst niedrigen Auflösung benutzen. Der Vorteil wäre hierbei, dass das ABS nicht mehr abhängig von der Geschwindigkeit der Hauptschleife ist. Außerdem ist das Abschalten des ABS gleichzusetzen mit dem Abschalten des zugehörigen Interrupts. Dieses wiederum eliminiert den Overhead des ABS komplett, wenn es abgeschaltet ist. Dieser Overhead wird auf ungefähr 2,5 µs pro Schleifendurchlauf geschätzt. Der Nachteil besteht in einem geringen Mehraufwand, da ein Funktionsaufruf durch eine Interrupt-Service-Routine ersetzt wird.
Eine weitere Erweiterungsmöglichkeit für das System besteht in erhöhter Robustheit und erweiterter Fehlererkennung. So ist es möglich eintreffende Befehle auf komplette syntaktische Korrektheit zu überprüfen und nur solche Befehle zu akzeptieren, die diese Tests bestehen. Diese Erweiterung muss allerdings möglichst effizient und einfach erweiterbar implementiert werden, um zum einen nicht entgegen der Designprinzipien des Systems zu handeln, als auch das Laufzeitverhalten nicht schwerwiegend zu beeinträchtigen. Zusätzlich zu dieser Fehlererkennung kann die Robustheit des Systems durch ein Überwachungssystem erhöht werden, welches in periodischen Abständen, ermöglicht durch die eingebauten Timer, die einzelnen Module des Gesamtsystems auf Anzeichen von Problemen untersucht, wie z.B. volle Puffer, runaway Befehle, Parser Status Korruption und dergleichen. Um die Möglichkeit von runaway Code auszuschließen, ist die Benutzung des eingebauten Watchdog-Timers möglich, dessen timeout Wert allerdings sehr sorgfältig gewählt werden muss. Der timeout Wert darf nicht kleiner sein als die längste Hauptschleifen Iteration plus ein entsprechendes Sicherheitspolster.
Ein Bereich, der während der Arbeit vollkommen vernachlässigt wurde, ist die Möglichkeit die Motorplatine in einen Idle-Modus zu schicken, in diesem Modus verbraucht wie Platine wesentlich weniger Strom. Dies würde die Zeit verlängern, die eine Batterie an solch einem Fahrzeug hält, während die Praktikanten an ihm arbeiten und die Motorplatine längere Zeit nichts zu tun hat. In dieser Hinsicht wäre es auch interessant den Idle-Modus bzw. das Verhalten um dem Idle-Modus durch Befehle während der Laufzeit steuern zu können.

Wie im Kapitel über die Laufzeituntersuchung beschrieben ist die Datenrate über den I2C-Bus der größte limitierende Faktor bei der Übermittlung von Fahrbefehlen von der Praktikumsplatine an die Motorplatine. Damit dieses Problem minimiert wird kann man zum einen erwägen den I2C-Bus im ”high-speed” Modus operieren zu lassen, oder eine eigene Punkt-zu-Punkt Kommunikation mithilfe der freien Ports auf der Motorplatine zu realisieren. Solch ein Maßgeschneiderter Port mit einem eigens dafür entwickelten Protokoll könnte die Latenz, die durch das Senden des Befehls entsteht, erheblich verringern.

Literaturverzeichnis

[1] Timo Klingeberg. Entwicklung einer motorsteuerung für zwei getriebemotoren. Technical report, Technische Universität Braunschweig, 2008.

[2] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall Software, 1988.

[3] ATMEL. *8-bit AVR Microcontroller with 64K/128K/256K • Bytes In-System • Programmable Flash*.

[4] Wikipedia. http://de.wikipedia.org/wiki/i2c. Internet.

A Module und ihre Beziehung

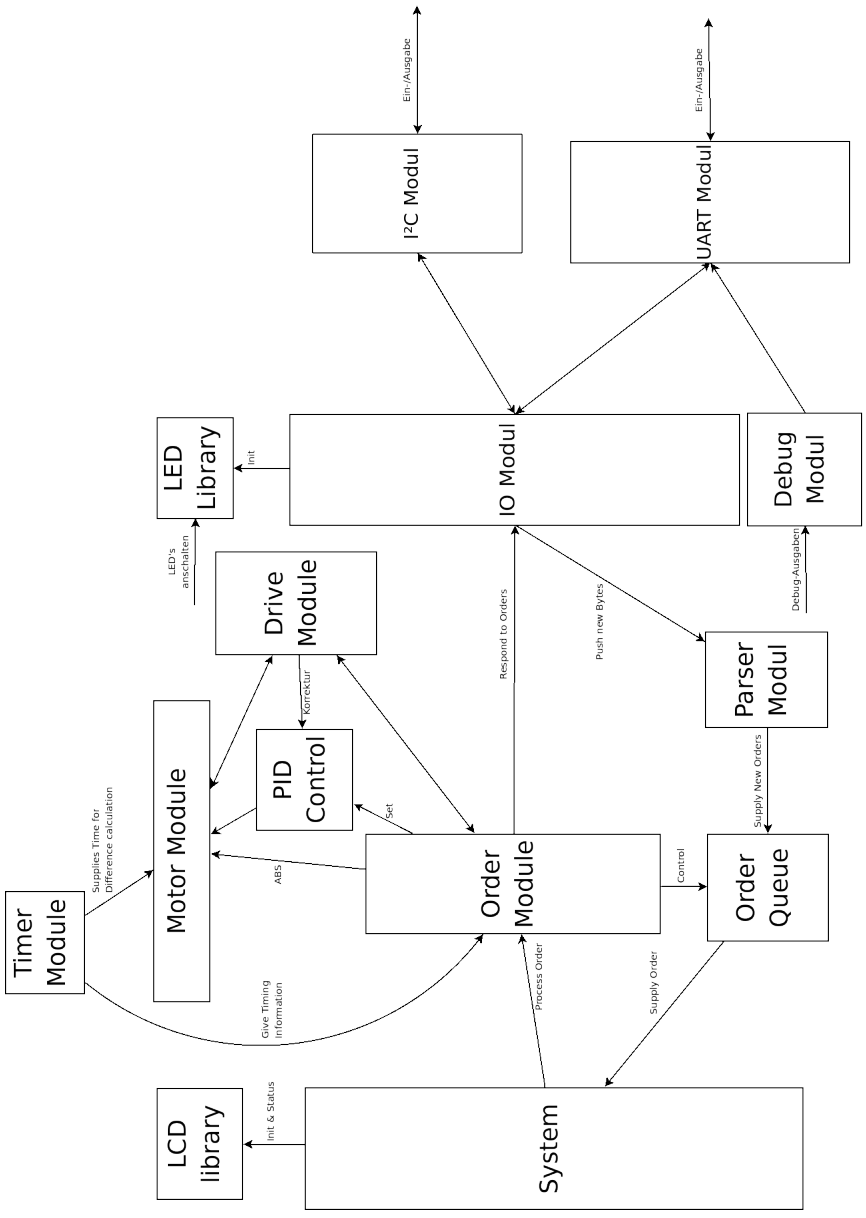

Abbildung A.1: Die Module und ihre Beziehungen