Betriebssoftware für eine Fahrplattform unter besonderer Berücksichtigung der Echtzeitbedingungen

Christoph Peltz

9. Oktober 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 9. Oktober 2009

Erklärung

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

Braunschweig, den 9. Oktober 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 der Motor- und Praktikumsplatine einfach und effizient zu gestalten. Außerdem wurde eine eingehende Untersuchung des Laufzeitverhaltens der Betriebssoftware durchgeführt.

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 the internship-board and the motor-board. Additionally a study of the behavior of the operation-software during the run-time was conducted.

[Hier wird später die Aufgabenstellung eingefügt.]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

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 entgegennimmt, 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. Nicht weniger wichtig war die Anforderung, dass die Software ihre Arbeit performant und zuverlässig ausführt.
Zur Untersuchung der Performance und des Echtzeitverhaltens der Betriebssoftware wurden an besonders wichtigen bzw. kritischen Stellen im Programmcode verschiedene nach außen gelegte Portpins gesetzt bzw. zurückgesetzt. An diesen Portpins wurden Flanken und Signaldauer gemessen und ausgewertet. Der hierfür benötigte Programmcode wird als Makro eingefügt und ist so kurz, dass eine Verfälschung der Laufzeit des Betriebsprogramms vernachlässigt werden kann.

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

Im vorgenannten 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 Praktikumsplatine und ein WLAN-Modul, das das Programmieren der Praktikumsplatine ohne Kabel ermöglicht. Außerdem können hierüber Daten zwischen der Praktikumsplatine und dem Entwicklungs-PC 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 freisteht). 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.

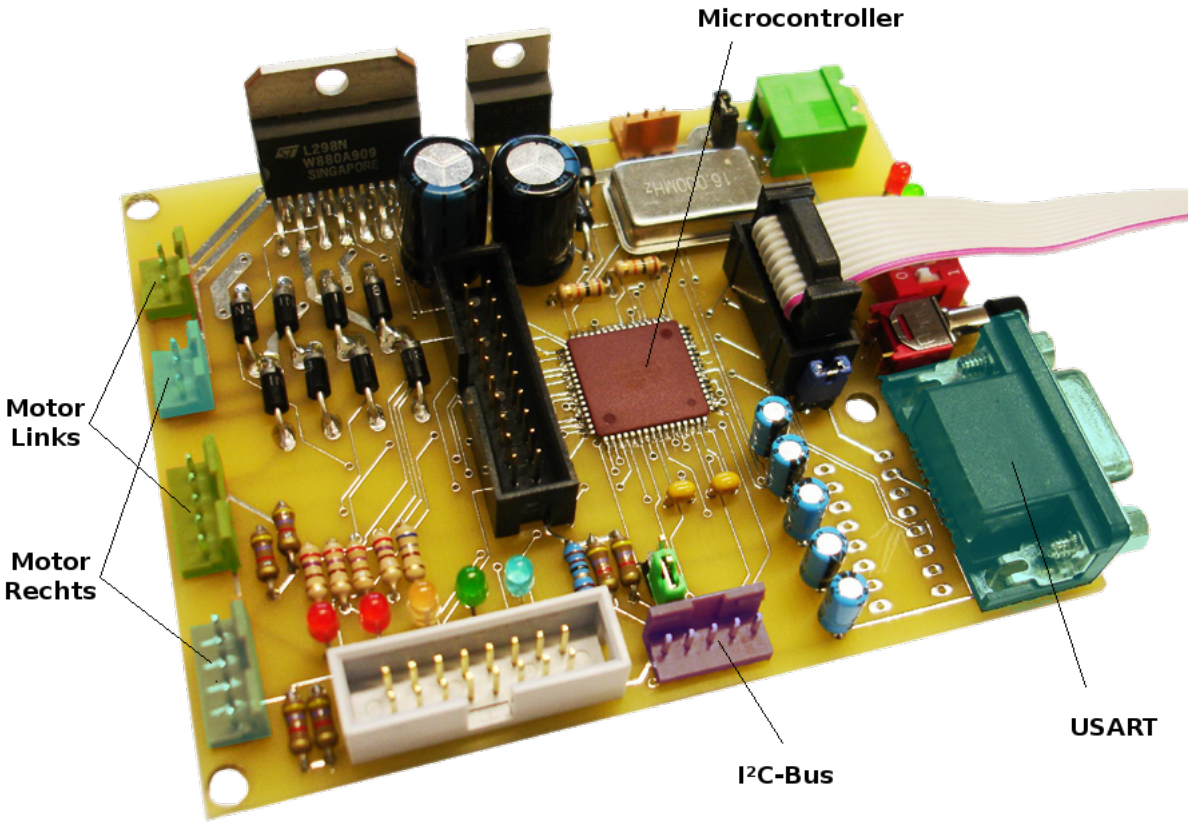

Abbildung 2.1: Die Motorplatine

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 Bits, die anderen vier 16 Bits. 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 genutzt 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 kann außerdem für die Ausgabe von Debug-Informationen benutzt werden, wenn 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 müssen 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 Ein-/Ausgabe-Ports

Zusätzlich zu den externen Kommunikationsmöglichkeiten mit der Praktikumsplatine besitzt der ATMEGA2561 54 programmierbare IO-Kanäle, die in Ports mit je 8 Kanälen zusammengefasst werden. 16 von diesen Kanälen sind für die Steuerung der Servomotoren zuständig, die die Räder und die Hall-Sensoren betreiben. Je ein Servomotor in Verbindung mit zwei Sensoren belegt einen Port. Der Servomotor benötigt 3 Kanäle des Ports, und die Sensoren benötigen 5 Kanäle. 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. Durch die Rückmeldung der Sensoren ist es möglich, die Servomotoren nicht nur zu steuern, sondern zu regeln. Zusätzlich kann die Geschwindigkeit der einzelnen Rädern aus diesen Informationen ermittelt werden. Die Informationen werden genutzt, um die Pulsweiten-Modulation zu regeln, mit der die Servomotoren angetrieben werden.
Neben diesen zwei Ports zum Regeln der Motoren sind weitere zwei Ports nach außen gelegt. Mithilfe der Pins dieser zwei nicht belegten Ports konnte eine weitgehend unverfälschte Untersuchung des Echtzeitverhaltens der Betriebssoftware durchgeführt werden.

2.2.4 LCD

Die Motorplatine verfügt über einen Anschluss für ein LCD. Dieses LCD kann zur autonomen Ausgabe von Informationen genutzt werden. Die Betriebssoftware nutzt diese Möglichkeit, wenn ein LCD angeschlossen ist, und der zugehörige DIP-Schalter aktiviert wurde. Es werden die aktuelle Version der Betriebssoftware, die Optionen und, falls vorhanden, der aktuelle Befehl in hexadezimaler Darstellung ausgegeben.

Zu den Optionen zählen der Status der Debug-Ausgaben, welche Kommunikations-Einrichtung verwendet wird, und der Zustand des ABS. Diese Optionen werden groß geschrieben, wenn sie aktiviert sind, und klein geschrieben, wenn sie deaktiviert sind. Die Betriebssoftware ist auf LCDs eingestellt, die 4 Zeilen und 20 Zeichen je Zeile besitzen. Sollte ein anderes LCD angeschlossen werden, ist eine korrekte Ausgabe – ohne Änderungen an der Software – nicht 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 Zeit 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 (siehe Anhang C) 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. 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 Motormodul seinerseits bindet das System an die Motoren und an die anderen nötigen Fahrelemente an. Dagegen abstrahiert das Drive-Modul die Services, die das Motormodul anbietet, und fasst diese so zusammen, dass das System auf sehr einfache Art und Weise die Motoren bedienen kann. Das PID-Modul, welches größtenteils aus der vorhergegangen Studienarbeit übernommen wurde [1], ist für den Fehlerausgleich der Radbewegungen zuständig. Genauer gesagt: Es errechnet die korrigierten Werte für die Räder, die dann vom Drive-Modul an diese weitergeleitet werden.
Mögliche Fehlerquellen sind: Verschiedenheiten in der Ausführung oder Anbringung der Räder, unterschiedliche Bodenbeschaffenheiten, ungleiche Gewichtsverteilung auf dem Fahrzeug, und unterschiedliches Verhalten der Servomotoren bei gleichen Parametern. Damit das Fahrverhalten aber stabiler wird, müssen diese Fehler korrigiert werden. Insbesondere bei ”Geradeaus-Fahrt” sind selbst kleine Fehler schnell zu erkennen.

3.4 IO, I2C und UART

Das IO-Modul stellt für das System einheitliche Funktionen zur Verfügung um Daten zu lesen oder zu schreiben. Dabei spielt es für den Benutzer der Funktionen keine Rolle, welche Schnittstelle für die Kommunikation genutzt wird. Es werden in beiden Fällen dieselben Funktionen verwendet. Das IO-Modul hebt die sichtbaren Unterschiede zwischen UART und I2C auf. Es besitzt Puffer für den Ein- und Ausgang, um Daten zwischenzuspeichern, bevor diese verarbeitet werden können. Sowohl das I2C- als auch das UART-Modul initialisieren hauptsächlich die Hardware und behandeln die Interrupt-Service-Routinen, die die eigentliche Kommunikation ermöglichen.

3.5 Order und Queue

Das Order-Modul ist das größte aller Module. Es beinhaltet unter anderem den Typ, mit dem Befehle intern dargestellt und verarbeitet werden. Aufbauend auf dem Typ, dem einige unterstützende Funktionen zugeteilt sind, um wiederkehrende Aufgaben zu erleichtern, existieren im Order-Modul auch die Order-Funktionen. Diese Order-Funktionen sind Handler-Funktionen für die im Protokoll spezifizierten Befehle. Falls ein Befehl längere Zeit 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 nacheinander ausgeführt werden. Außerdem sorgt die Queue dafür, dass 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 bietet das Timer-Modul die Möglichkeit in bestimmten zeitlichen Intervallen Befehle auszuführen. Benutzt wird nur **ein** Timer, der alle 100 ms eine Unterbrechung auslöst. Er speichert die Anzahl der Räder-Ticks für das PID-Modul ab, welches mit diesen Informationen Nachregelungen durchführt. Außerdem erhöht dieser Timer nach je 10 Unterbechungen den Sekunden-Zähler.
Der Parser holt eingegangene Bytes beim IO-Modul ab und konvertiert diese zu Befehlen. Dabei achtet er auf die Länge, die bestimmte Kommando-Bytes vorgeben (vgl. Kapitel 4). Wenn ein Befehl vollständig empfangen ist, wird er von der Queue abgeholt und an die Warteschlange angereiht.
Das Options-Modul beinhaltet die Einstellungen, die das Verhalten des gesamten Systems beeinflussen, wie z.B., ob Debugging-Ausgaben aktiviert sind, mit welcher Geschwindigkeit das ABS (siehe Kapitel 5.2) standardmäßig bremst, die maximale Länge eines Befehls, oder die Größe der Ringpuffer für das Parser- und Queue-Modul.

4 Protokoll

Die Praktikumsplatine muss mit der Motorplatine kommunizieren, um das Fahrzeug in Bewegung zu setzen. Diese Kommunikation sollte möglichst effizient erfolgen. Die Entwicklung eines guten Protokolls ist deshalb sehr wichtig für das gesamte System.
Das Vorläufer-Protokoll benutzte Zeichenketten, um Instruktionen zu übermitteln. Dies hat zwei gravierende Nachteile. Zum einen nimmt das Zusammensetzen der Zeichenketten mithilfe der verfügbaren Prozessoren sehr viel Rechenleistung in Anspruch. Zum anderen ist durch die Verwendung der Zeichenketten die Informationsdichte der Instruktionen nicht sehr hoch, und es fällt zusätzliche Wartezeit bei der Übermittlung an.

4.1 Grundlegende Konzepte

Da die Kommunikations-Einrichtungen 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 der im Code eingestellten Größe der Befehlsstruktur entspricht. Die maximale Länge der einzelnen eingebauten Befehle beträgt 9 Bytes. Das erste Byte eines Befehls hat eine besondere Bedeutung. Dieses Byte wird Kommando-Byte genannt. Das Kommando-Byte ist in zwei Teile geteilt. Der erste Teil umfasst die 4 Bits mit der niedrigsten Wertigkeit und wird Befehlscode genannt. Der Befehlscode spezifiziert die Art des Befehls. Der zweite Teil beinhaltet die höchstwertigen 4 Bits. Darin werden Optionen gesetzt, die den im Befehlscode angegebenen Befehl modifizieren. Die Kombination von Befehlscode und Optionen legt auch die Länge des Befehls fest. Alle dem Kommando-Byte folgende Bytes sind Parameter, wie z.B. die Geschwindigkeit der Räder.
Da 4 Bits 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 geben kann. Damit ist es möglich, Befehle einfach hinzuzufügen, die nicht in das Schema ”ein Kommando-Byte, viele Parameter” passen.
Wenn Parameter übertragen werden müssen, die mehr als ein Byte benötigen, wird zuerst das höchstwertige Byte übertragen. Danach, absteigend nach der Wertigkeit, folgen 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

liegenden 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 wurde der Befehlscode bei jedem Befehl in hexadezimaler Schreibweise als ganzes Byte dargestellt.

4.2.1 Extended-Instruction - 0x00

Dieser Befehl ist ein Platzhalter für zukünftige Befehle, die mehr als ein Kommando-Byte benötigen, oder für den Fall, dass alle normalen Befehlscodes bereits vergeben sind. Der Befehlscode ist **0x00**.
Im Code werden solche Befehle mit besonderen Funktionen behandelt. Die parser_extended_order_complete-Funktion ist ein Beispiel hierfür. Diese besonderen Funktionsvarianten gibt es allerdings nur im Parser. Nach dem Verlassen des Parsers wird eine Extended-Instruction wie ein normaler Befehl behandelt. Die zusätzlichen Funktionen waren nötig, weil sich diese Befehle im Aufbau von normalen Befehlen stark unterscheiden.

4.2.2 Control - 0x01

Mit diesem Befehl kann das System gesteuert werden. Darunter fallen Aufgaben wie das Reseten des gesamten Programms, 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 verodert und ergibt das Kommando-Byte. Bei manchen Befehlen können mehrere Optionen zur gleichen Zeit gewählt werden; dies ist hier nicht der Fall. Die Optionen dieses Befehls schließen sich gegenseitig aus. Die Länge der Befehle mit vorgenanntem Befehlscode beläuft sich auf 1 Byte, sodass lediglich das Kommando-Byte gesendet werden muss.

4.2.3 Query - 0x02

Manchmal ist es wichtig, dass die kontrollierende Praktikumsplatine verschiedene Laufzeitwerte der Motorplatine kennt. 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. Er wird also nicht an die Warteschlange angereiht.
Wenn nach Senden dieses Befehls an die Motorplatine kurz darauf versucht wird, über den I2C-Bus das Ergebnis zu lesen, kann der Fall eintreten, dass es noch nicht vorliegt. Das geschieht besonders dann, wenn der aktuelle Befehl angefordert wurde. In diesem Fall muss die Lese-Operation nochmals gestartet werden.

Option	Bitmaske	resultierendes Kommando-Byte	Beschreibung
Reset	0x10	0x11	Hardware-Reset
Stop Queue	0x20	0x21	Der Aktuelle Befehl wird verworfen. Die Warteschlange wird angehalten, aber nicht verworfen.
Continue Queue	0x30	0x31	Der nächste Befehl in der Warteschlange wird ausgeführt.
Clear Queue	0x40	0x41	Die Warteschlange wird gelöscht.
Stop Drive	0x50	0x51	Befehl wird angehalten, Fahrzeug geht zum aktiven Bremsen über.

Tabelle 4.1: Optionen des Control-Befehls

Die Länge des aktuellen Befehls muss mit übertragen werden, damit die Praktikumsplatine diesen Befehl, den die Motorplatine zurückgibt, auch verwerten kann. Es ist deshalb nötig, dass die Praktikumsplatine zwei Lese-Operationen durchführt, wobei die erste eine Länge von einem Byte hat und außerdem spezifiziert, wie lang die zweite Antwort in Bytes ist.

Bitmaske	resultierendes Kommando-Byte	Beschreibung	Antwortlänge in Bytes
0x10	0x12	Geschwindigkeit des linken Rades	1
0x20	0x22	Geschwindigkeit des rechten Rades	1
0x30	0x32	Anzahl der Befehle in der Warteschlange	1
0x40	0x42	Aktueller Befehl	2 - 16

Tabelle 4.2: Optionen des Query-Befehls

4.2.4 Drive - 0x03

Der am häufigsten benutzte Befehl ist der Drive-Befehl. Durch ihn werden die Räder des Fahrzeugs in Bewegung gesetzt. Ihm ist der Befehlscode **0x03** zugeteilt, und seine Länge beträgt zwischen 3 und 7 Bytes. Im Minimalfall besteht der Befehl aus dem Kommando-Byte und je einem Byte für die gewünschte Geschwindigkeit der einzelnen Räder. Ohne die Angabe einer Abbruchbedingung für beide Räder würde dieser Befehl endlos laufen, oder solange, bis mithilfe des Control-Befehls die Ausführung

unterbrochen wird. Die Abbruchbedingungen werden Trigger genannt. Die Räder sind mit zwei Trigger-Arten ausgerüstet: Die Positions-Trigger und die Zeit-Trigger. Die Positions-Trigger stoppen das Rad, dem sie zugeordnet wurden, nachdem eine bestimmte Anzahl an Interrupts von dem Rad ausgelöst wurden. Die Zeit-Trigger stoppen das Rad, nachdem die angegebene Fahrzeit erreicht wurde. Dadurch kann jedes Rad für sich entweder durch keinen Trigger, durch einen Zeit-Trigger oder durch einen Positions-Trigger beeinflusst werden. Triggerwerte sind 2 Bytes lang und nur positiv. Die Geschwindigkeit ist 1 Byte lang und kann auch negativ sein. Dem Kommando-Byte folgt daher das Byte für die Geschwindigkeit des linken Rades, dann das Byte für die Geschwindigkeit des rechten. Falls Triggerwerte angegeben werden müssen, erfolgt dies nach den Geschwindigkeitsangaben. Man beginnt ebenfalls mit dem Triggerwert für das linke Rad, dann folgt der Wert für das rechte.
Zusätzlich zu diesen Fahrmöglichkeiten gibt es einen speziellen Fahrmodus, der für die ”Geradeaus-Fahrt” optimiert ist. Er wird als ”Fahrt mit Differenzausgleich” bezeichnet. Hierbei wird versucht, die Differenz beider Räder möglichst auf 0 zu halten. Wegen der Geradeaus-Fahrt entfällt die zweite Geschwindigkeitsangabe, und die Bitmasken für die Trigger des rechten Rades gelten für beide Räder. Eine weitere Bitmaske erlaubt es, den Startwert des Differenzwertes festzulegen. Dieser ist dann der einzige Parameter, 2 Bytes lang und vorzeichenbehaftet.
Zur Konstruktion des Kommando-Bytes für diesen Befehls wird der Befehlscode mit der Bitmaske für das linke und rechte Rad verodert. Dabei ist zu beachten, dass eine der drei Möglichkeiten ausgewählt wird, die in dem oberen Teil der Tabelle 4.4 vorgestellt werden. Das gilt nur, wenn keine die Bitmaske 0x30 oder 0xc0 verwendet wird.

Byte-Anzahl	**Option**	**Wertebereich**	**Beschreibung**
1	-	-128 bis 127	Geschwindigkeit des linken Rades
1	-	-128 bis 127	Geschwindigkeit des rechten Rades
2	Zeit-Trigger	0 bis 65535	die Fahrzeit des linken Rades (in 100 ms)
2	Positions-Trigger	0 bis 65535	Anzahl der Ticks für die Fahrt des linken Rades
2	Zeit-Trigger	0 bis 65535	die Fahrzeit des rechten Rades (in 100 ms)
2	Positions-Trigger	0 bis 65535	Anzahl der Ticks für die Fahrt des rechten Rades

Tabelle 4.3: Parameter des Drive-Befehls

4.2.5 Advanced-Drive - 0x04

Der Advanced-Drive-Befehl, der den Befehlscode **0x04** besitzt, verhält sich grundsätzlich wie der normale Drive-Befehl. Er bietet allerdings mehr Möglichkeiten für den

Option	Bitmaske links	Bitmaske rechts	Beschreibung
Kein Trigger	0x00	0x00	Endlosfahrt
Zeit-Trigger	0x10	0x40	zeitlich begrenzte Fahrt
Positions-Trigger	0x20	0x80	fährt eine bestimmte Strecke
Fahrt mit Differenz-Ausgleich	0x30	-	beide Räder fahren mit derselben Geschwindigkeit geradeaus
Differenz setzen	0xc0	-	setzt die Differenz der Räder

Tabelle 4.4: Optionen des Drive-Befehls

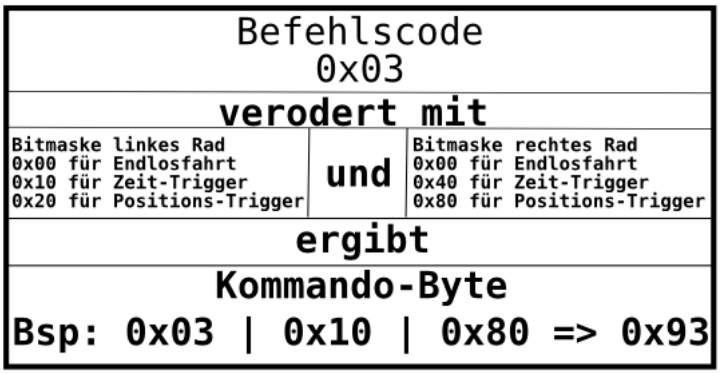

Abbildung 4.1: Das grundlegende Konzept zur Erstellung eines Drive-Kommando-Bytes

Einsatz der Trigger. Wird der Befehl ohne Trigger verwendet, so ist er mit dem Drive-Befehl identisch. Werden diese jedoch verwendet, müssen für jedes Rad zwei Trigger-Werte angegeben werden. Es werden also immer Zeit- und Positions-Trigger verwendet. Je nach gewählter Option werden sie miteinander mit einem logischen UND oder einem logischen ODER verknüpft. So ist es möglich, Fahrbefehle zu erteilen, die entweder eine bestimmte Strecke oder eine bestimmte Zeit fahren.

4.2.6 SetPID - 0x05

Das PID-Modul benutzt mehrere Parameter, um den Fehlerausgleich durchzuführen. Diesen Parametern sind im Programmcode bereits Standardwerte zugewiesen worden. Es kann aber vonnöten sein, diese Parameter anzupassen. Dafür ist der SetPID-Befehl erforderlich. Dieser besitzt den Befehlscode **0x05**. Seine Länge beträgt immer 9 Bytes. Es müssen alle Werte dem SetPID-Befehl übergeben werden. Jeder Wert benötigt 2 Bytes. Die Werte können auch negativ sein. Sie werden in der Reihenfolge angegeben, wie sie im oberen Teil der Tabelle 4.6 zu finden sind. Die genaue Auswirkung dieser Einstellungen sollte vom Benutzer in der Studienarbeit von Timo Klingeberg [1] nachgelesen werden.

Option	Bitmaske Links	Bitmaske Rechts	Beschreibung
Kein Trigger	0x00	0x00	Endlosfahrt
Zeit-Trigger ODER Positions-Trigger	0x10	0x40	Fahrt bis Erreichen eines Triggers
Zeit-Trigger UND Positions-Trigger	0x20	0x80	Fahrt bis Erreichen beider Trigger

Tabelle 4.5: Optionen des Advanced-Drive-Befehls

Byte-Anzahl	Position	Wertebereich	Beschreibung
2	1	-32768 bis 32767	**P**roportionaler Faktor
2	2	-32768 bis 32767	**I**ntegraler Faktor
2	3	-32768 bis 32767	**D**ifferentieller Faktor
2	4	-32768 bis 32767	maximaler Fehlersummen-Modifikator

Option	Bitmaske	resultierendes Kommando-Byte
linkes Rad	0x00	0x05
rechtes Rad	0x10	0x15
beide Räder	0x20	0x25

Tabelle 4.6: Parameter und Optionen des SetPID-Befehls

4.2.7 Option - 0x06

Der Option-Befehl besitzt den Befehlscode **0x06** und hat eine Länge von 2 Bytes. Durch ihn können bestimmte Variablen auf der Motorplatine während der Laufzeit geändert werden. Momentan kann damit das ABS (siehe Kapitel 5.2) angepasst werden. Nach dem Kommando-Byte folgt der neue Wert der Variablen.

Bitmaske	Default	Wertebereich (Bytes)	Beschreibung
0x10	40	1 bis 127 (1)	Geschwindigkeitswert beim aktiven Bremsen
0x20	1	0 oder 1 (1)	Schalter zum Aktivieren der aktiven Bremsung (eine 1 bedeutet die Aktivierung)
0x30	1	0 oder 1 (1)	Schalter zum Aktivieren der aktiven Bremsung **des** Rades, das seinen Trigger erreicht hat
0x40	1	0 oder 1 (1)	Schalter zum Aktivieren der aktiven Bremsung, wenn kein Befehl bearbeitet wird

Tabelle 4.7: Optionen des Option-Befehls

5 Implementierung der Betriebssoftware

Die Betriebssoftware wurde, wie in der Aufgabenstellung festgelegt, in C geschrieben. Hierbei fiel die Wahl auf den Standard C99, der gegenüber C89 einige sprachliche Aktualisierungen aufweist. 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 sonst dazu führen würde, dass das System nicht mehr reagiert. Wie in Abb. 5.1 zu sehen ist, werden in der Hauptschleife vier wichtige Funktionen aufgerufen.

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

    process_orders();

    if (LCD_PRESENT)
        lcd_update_screen();

    parser_update();

    queue_update();
}
```

Abbildung 5.1: Die Hauptschleife

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. Wenn es solch einen Befehl gibt, ruft die Funktion eine Verteilerfunktion auf. Diese wiederum ruft die zugehörige Befehlsfunktion auf, indem der Befehlscode als Index für eine Call-Table benutzt wird (siehe Abb. 5.2).
Falls kein Befehl vorliegt, oder die Queue angehalten wurde, ruft die process_orders()-Funktion die Funktion zum aktiven Bremsen auf. Das aktive Bremsen wird in Kapitel 5.2 behandelt.

```
// 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 5.2: Die Verteilerfunktion

5.1.2 lcd_update_screen()

Wenn ein LCD angeschlossen ist, können dort Informationen ausgegeben werden. Der Anschluss des LCD wird dem System mit einem DIP-Schalter auf der Platine mitgeteilt.
Da das synchrone Aktualisieren des LCD sehr viel Zeit benötigt (vgl. Kapitel 6.2), wird pro Aufruf dieser Funktion maximal ein Zeichen an das LCD geschickt. Dies wird durch Abfrage des Busy-Flags erreicht, welches signalisiert, dass das LCD noch beschäftigt ist. Falls es nicht gesetzt ist und noch Daten zu aktualisieren sind, wird das nächste Zeichen an das LCD gesendet.
Auf dem LCD werden die Versionsnummer der Betriebssoftware, der Status einiger globaler Variablen, und der aktuell ausgeführte Befehl angezeigt. Wenn 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 Puffers an das LCD weiter.
Befehle, die innerhalb eines Hauptschleifendurchlaufs abgearbeitet sind, werden nicht ausgegeben und generieren auch keinen Aufruf von lcd_update_info(). Das ist nötig, weil diese Befehle zu schnell abgearbeitet werden. Es könnten zwischenzeitlich 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 richtige Stelle im Puffer ein. Wenn ein Befehl komplett ist, dies wird mit der parser_order_complete()-Funktion überprüft, gilt der Befehl als fertig und alle weiteren Bytes, die hinzugefügt werden, landen in einer neuen order_t-Struktur.
Zum Erkennen, wann ein Befehl zu Ende ist, benutzt die parser_order_complete()-Funktion die bytes_needed()-Funktion. In dieser ist fest codiert, welcher Befehlscode mit welchen Optionen wie viele Bytes benötigt. Das ist auch eine der Stellen, die angepasst werden müssen, wenn neue Befehle hinzugefügt oder bestehende verändert werden sollen.

5.1.4 queue_update()

Diese Funktion führt Wartungsarbeiten an der Befehlswarteschlange (Queue) durch. Das beinhaltet, neue Befehle beim Parser-Modul abzuholen und diese korrekt einzureihen. Es gibt zwei Möglichkeiten, wie die neuen Befehle an die Queue angereiht werden können. Zum einen als normale Befehle, die einer nach dem anderen abgearbeitet werden, zum anderen als priorisierter Befehl. Es kann nur ein priorisierter Befehl in der Queue sein. Die 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. Kapitel 4).

5.2 Das Aktive-Brems-System (ABS)

Das aktive Bremssystem soll bewirken, dass die Räder im praktischen Betrieb ihre Position nicht verlassen können. Dies wird realisiert, indem eine Referenz-Position für jedes Rad gespeichert wird. Während der Hauptschleife wird die tatsächliche Position mit der Referenz-Position verglichen. Für den Fall, dass diese Positionen nicht übereinstimmen, werden die Motoren mit einer einstellbaren Geschwindigkeit so betrieben, dass die Räder wieder auf die Referenz-Position gebracht werden.
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 bei laufendem System 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 reaktivieren, oder das gesamte ABS abschalten bzw. wieder anschalten. So kann der Benutzer das ABS an seine Wünsche anpassen.

5.3 Befehle: Struktur und Funktionen

Die Struktur (Abb. 5.3), 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 5.3: Befehlsstruktur

Byte dieses Arrays ist das Kommando-Byte, welches die Art des Befehls und die zugehörigen Optionen spezifiziert. Der Befehlscode 0x00 ist für zukünftige Erweiterungen reserviert, die mehr als ein Kommando-Byte benötigen. Des Weiteren sind die Befehlscodes 0x01 bis 0x06 durch diese Arbeit bereits definiert und mit Funktionalität erfüllt. Die Befehlscodes 0x07 bis 0x0f sind noch nicht definiert und können für

zukünftige Erweiterungen benutzt werden, die mit **einem** Kommando-Byte auskommen.
Alle auf das Kommando-Byte (oder die Kommando-Bytes im Falle des Befehlscodes 0x00) folgenden Bytes sind Parameter. Deren Anzahl und Länge hängt von der Spezifikation des Befehls ab. Bei Parametern, die länger als ein Byte sind, wird zuerst das höchstwertige Byte im Array gespeichert. Dann folgen die übrigen Bytes mit absteigender Wertigkeit.
Oft benutzte Aktionen bezüglich der Befehlsstruktur wurden zusammengefasst (siehe Abb. 5.4 und 5.5). Vor der Untersuchung des Laufzeitverhaltens, und der damit einhergehenden Optimierungen, wurden diese beiden Funktionen durch for-Schleifen implementiert. Wie sich bei der Untersuchung herausstellte, waren die for-Schleifen aber langsamer als die Standard-C-Funktionen memset() und memcpy().

```
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 5.4: 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 5.5: 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 µs pro Byte). In diesen Interrupt-Service-Routinen wird das empfangene Byte in den Eingangspuffer gelegt. Wenn die Hauptschleife wieder die parser_update()-Funktion erreicht, werden die bisher empfangenen Bytes abgeholt und in dem 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. 5.3) das ORDER_STATUS_DONE-Flag gesetzt wurde. Anschließend entfernt sie diesen aus der Warteschlange.
Falls der zu bearbeitende Befehl eine Ausgabe von Daten bewirkt, werden diese in der entsprechenden order_function-Funktion in dem Ausgabepuffer des IO-Frameworks

angereiht. Dieses Framework wird dann, sobald wie möglich, diese Daten über die ausgewählte Schnittstelle senden (bei UART wird sofort mit dem Senden begonnen; bei I2C muss gewartet werden, bis die Daten vom Benutzer abgerufen werden).

5.5 Debug-Ausgaben

Es ist nicht ohne weiteres möglich, einen üblichen Debugger zur Fehlerfindung zu verwenden; stattdessen sind Debug-Ausgaben notwendig, um die Funktion des Debuggers zu ersetzen. Im Gegensatz zu einem Debugger muss zusätzlicher Programmcode eingefügt werden, um Debug-Ausgaben realisieren zu können. Diese Ausgaben beeinträchtigen die Geschwindigkeit des gesamten Systems auch dann noch, wenn diese mit if-Statements umschlossen werden (siehe Abb. 5.6).

Während der normalen Operation der Betriebssoftware im Praktikum sind diese Ausgaben nicht nötig, und für die meisten Teilnehmer des Praktikums nicht informativ. Der Leser muss eine entsprechende Kenntnis des Codes besitzen, damit die Ausgaben für die Fehlerbehebung benutzt werden können.
Wegen der geringen Hilfe, die diese Ausgaben im Normalfall dem Praktikanten bieten, und der negativen Auswirkung der Ausgaben auf die Performance des Systems, wurde ein Prä-Prozessor-Technik angewandt, die zusammen mit der eingestellten Stufe der Code-Optimierung des Compilers die beiden Probleme löst.

Durch diese Technik werden die Debug-Ausgaben nur dann kompiliert, wenn dem Compiler der Parameter -DDEBUG übergeben wird. Zwar ist es auch dann noch möglich, mit einem DIP-Schalter auf der Platine die Ausgaben auszuschalten, aber es bleiben immer noch die leicht negativen Auswirkungen auf die Performance.

Normalerweise wird die Betriebssoftware ohne diesen Parameter kompiliert. Das bedeutet, dass die Software neu kompiliert werden muss, wenn diese Debug-Ausgaben erwünscht sind.

```
#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 5.6: 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 der 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 sowohl über einen Ausgabe- als auch 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 unangemessen viel Zeit in Anspruch nehmen.
Eine Besonderheit ist die Ausgabe von Daten auf Objekt-Basis. Ein Objekt hat mindestens ein Byte und maximal 256 Byte. Objekte werden entweder komplett übertragen oder gar nicht. Wenn ein Objekt nicht komplett übertragen werden konnte, wird bei der nächsten Übertragung vom Anfang des 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 Prozessorzeit, wenn es nötig ist, und vermeidet so nutzlosen Zeitverbrauch bedingt durch aktives Warten.

5.7 Unterstützende Bibliothek für die Praktikumsplatine

Die Benutzung der Motorplatine soll für die Studierenden möglichst einfach sein. Aus diesem Grund wurde neben der Betriebssoftware für die Motorplatine auch eine Bibliothek für die Praktikumsplatine geschrieben. Diese Bibliothek stellt Funktionen und Definitionen zur Verfügung, um Befehle an die Motorplatine senden oder empfangen zu können. Hierfür wurde eine modifizierte Version der Befehls-Struktur für die Praktikumsplatine geschrieben. Es sind drei Funktionsaufrufe nötig, um jeden möglichen Befehl zusammenzustellen und zu versenden. Die erste Funktion setzt das Kommando-Byte. Die zweite setzt die Parameter; dies wurde mit einer Funktion gelöst, die eine variable Liste von Argumenten erhält. Die dritte und letzte Funktion sendet den Befehl an die Motorplatine. Vorher muss allerdings die Befehls-Struktur einmal initialisiert werden.

```
order_t order;
order_init(&order);
order_set_type(&order, ORDER_DRIVE_P_P);
order_add_params(&order, "1122", 127, -100, 32700, 16768);
order_send(&order);
```

6 Untersuchung des Laufzeitverhaltens

Die Untersuchung des Laufzeitverhaltens wurde mithilfe eines Logik- Analysers des Typs ”Agilent Logicwave E9340A” durchgeführt. Dieser misst, wann und wie lange ein Portpin gesetzt oder nicht gesetzt ist. Die Pins, die zur Durchführung dieser Messungen nötig sind, 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 die Pins setzen und zurücksetzen zu können. Diese Operationen wurden als Makros implementiert, um den Mess-Overhead so gering wie möglich zu halten. Wie hier beschrieben, sind die Zeiten für die Ausführung der Instruktionen –

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

setzen, löschen und umschalten von einzelnen Pins – ziemlich gering. Doch vor allem bei den Messungen mit einem leistungsfähigen und sehr genauen Oszilloskop konnte herausgefunden werden, dass der eigentliche Wechsel der Spannung am Pin verhältnismäßig langsam durchgeführt wird. Insbesondere das Abfallen der Spannung, 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. Das bedeutet, dass der Logik-Analyser eine gewisse Zeitspanne einen ”falschen” Wert misst (Er ist nicht physikalisch falsch, aber **logisch** falsch). Denn: Wenn die Spannung abfällt, ist der Pin schon nicht mehr gesetzt; der Logik-Analyser allerdings betrachtet diesen teilweise immer noch als gesetzt.

6.1 Erste Messungen

Durch die ersten Messungen wurde der Startpunkt für die Untersuchung der Software festgelegt. Dafür wurde in der Hauptschleife (siehe Abb. 6.2) zum einen ein pin_toggle() eingebaut, um die Länge einer Schleifeniteration zu messen; zum anderen wurden die einzelnen Funktionsaufrufe in der Hauptschleife mit pin_set() und pin_clear() umgeben. In der Tabelle 6.2 sind die wichtigsten Daten der ersten Messreihen zusammengefasst. Die Spalte ”Rahmenbedingungen” beschreibt Bedingungen, die während der Messung geherrscht haben. Hierbei bezeichnet die Bedingung ”AB:EIT”,

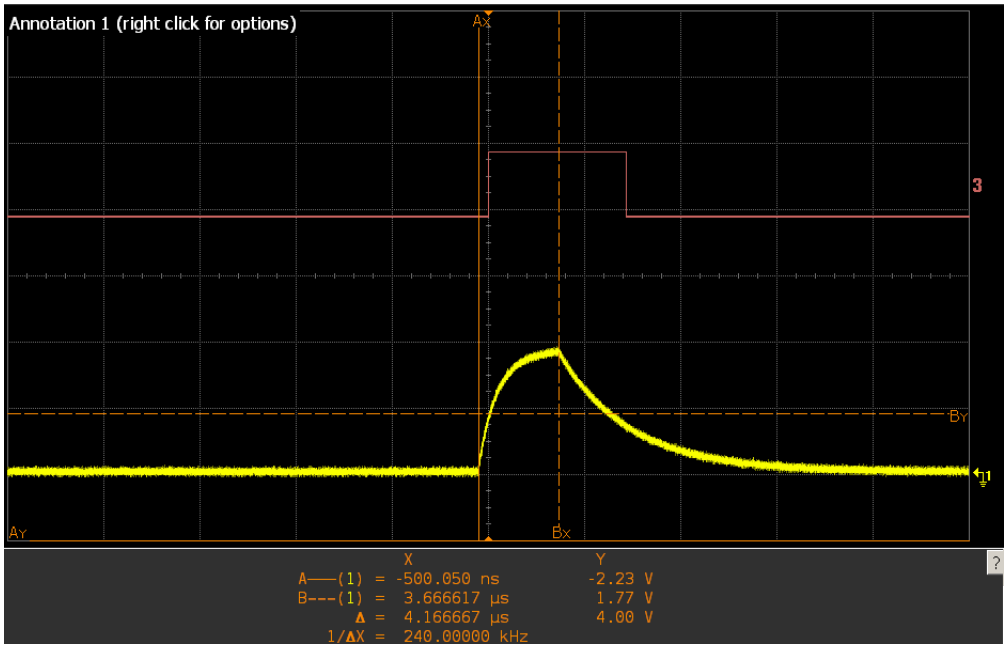

Abbildung 6.1: Reales und logisches Spannungsverhalten der Portpins

Anmerkung: In diesem Bild sind sowohl der reale Spannungsverlauf (gelb) als auch der logische Spannungsverlauf (pink) auf derselben zeitlichen Basis dargestellt. Die horizontale braune Linie zeigt den Schwellenwert an, ab dem der gemessene Portpin als logisch ”high” betrachtet wird. Man erkennt hier deutlich, dass die Spannung beim Zurücksetzen des Pins nur langsam abfällt.

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 bei diesen Daten ist, dass das Aktualisieren der Warteschlange bei Vorliegen eines Befehls 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 zeitintensivste im gesamten System. Das Aktualisieren erfolgt zwar recht selten, nämlich nur, wenn ein neuer Befehl gestartet wird; aber schon bei drei Millisekunden Latenz wird die Reaktion des Systems auf Ereignisse negativ beeinflusst. So könnten während der drei Millisekunden ungefähr zwölf Bytes auf dem I2C-Bus eintreffen. Dies wären im schlimmsten Fall sechs eigenständige Befehle. Wenn diese Befehle priorisierte Befehle sind, könnte sich das System anders verhalten als erwartet .
Zur Lösung dieses Problems musste die Arbeitsweise des LCD, welches durch eine fertige Programm-Bibliothek gesteuert wird, analysiert werden. Dabei wurde heraus-

```
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 6.2: Die Hauptschleife mit Debug-Ausgaben und Pin-Operationen

gefunden, 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 aktualisiert die LCD-Informationen in einem Speicherpuffer. Die andere, immer noch lcd_update_screen() genannt, überprüft das Busy-Flag des LCDs und schreibt das nächste Zeichen aus dem Speicherpuffer aufs LCD, wenn es nicht gesetzt ist. Diese Lösung des Problems ist unter den gegebenen Bedingungen die effektivste. Die eleganteste und effizienteste Lösung wäre die Benutzung eines Interrupts auf Basis des Busy-Flags; dies ist aber aufgrund der Konstruktion des LCDs nicht möglich. Wie in der Tabelle 6.3 zu sehen ist, hatten diese Maßnahmen aber ihre gewünschte Wirkung. Über 98% weniger Zeit wird nun im zeitintensivsten Fall benötigt. Die nachteillige Folge ist eine leicht erhöhte Laufzeit für das Aktualisieren des LCD im Ruhezustand, deren Dauer sich von durchschnittlich 4 µs auf 9 µs erhöht hat.

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-zurücksetzen-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 steht, wurde anschließend der Compiler auf die Optimierung der Geschwindigkeit eingestellt (-O2 Option). Außerdem wurden die Debug-Informationen (-g Option) entfernt, die der Compiler im Objektcode platziert hat. Dadurch änderten sich die Parameter des Systems nicht wesentlich. Die Größe blieb nahezu identisch und die Geschwindigkeit stieg leicht an, 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. Diese bewirkt, dass der Code von ausreichend simplen Funktionen direkt in die aufrufende Funktion eingefügt wird, ohne dass dabei Register gerettet werden müssten oder Ähnliches. Dies hatte deutliche 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 zurückzuliefern. Für diese simple Operation wurden über 36 µs benötigt. Das ist länger, als die gesamte Hauptschleife im Idle Zustand braucht. Eine genaue Untersuchung der Funktion ließ darauf schließen, dass die Anweisungen mit Modulo-Operatoren für diesen Aufwand verantwortlich waren. Das konnte zum einen durch Kontrolle des generierten Assembler-Codes, zum anderen durch das Auslagern der Modulo-Operatoren in eine eigene Zeile bestätigt werden. Der Compiler generiert für die Modulo-Operatoren eine Extra-Funktion, die sehr kompliziert ist und offensichtlich bemerkenswert viel 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 Methoden, 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 bzw. 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 Funktion lief daraufhin wesentlich schneller. Der Code sieht nun aus wie folgt:

```
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;
}
```

Dasselbe Problem existierte in der Queue, dort wurden allerdings die Modulo-Operatoren

```
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 wegen ihrer häufigen Verwendung ebenfalls untersucht. Die Befehlsstrukturen wurden einfach mithilfe von Schleifen mit dem Wert 0 initialisiert, und beim Kopieren wurden 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 einem 40%igen Geschwindigkeitszuwachs (siehe 6.6).

6.3.4 Bedingtes Kompilieren von Debug-Ausgaben

Wie in Kapitel 5.5 beschrieben, wurde eine Sprachen-spezifische Technik verwendet, um einfache Diagnoseausgaben zu ermöglichen, 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. Wenn es aber gegeben wurde, und die Ausgaben mithilfe des entsprechenden DIP-Schalters deaktiviert sind, bringt jede Ausgabe lediglich eine Erhöhung der Laufzeit um weniger als eine µs.

6.4 Verlieren/Verpassen von Interrupts

Während der Messungen kam die Frage auf, was passieren würde, wenn ein Interrupt auftritt und ein anderer bereits läuft. Wenn dies möglich sein sollte, was geschieht dann mit den beiden Interrupts? Geht eventuell sogar einer davon verloren?
Die Antwort auf diese Fragen wurde von dem Handbuch des Mikrocontrollers [3] gegeben. Mit den richtigen Einstellungen, die bereits vorgenommen wurden, werden Interrupts, die während eines anderen Interrupts auftreten, gespeichert. Nach der Beendigung des laufenden Interrupts werden die anderen gespeicherten ihrer Priorität nach ausgeführt.
Der schnellste Interrupt ist der UART-Interrupt. Er tritt maximal aller 138 µs auf. Im ungünstigsten Fall treten alle Interrupts zur selben Zeit auf, und der danach am schnellsten wieder auftretende hat die niedrigste Priorität. Er wird also erst ausgeführt, nachdem alle anderen Interrupts abgearbeitet wurden. Bei vier Hall-Sensoren-Interrupts, einem I2C-Bus-Interrupt, einem Timer-Interrupt und zwei UART-Interrupts sowie einer oberen Schranke von 8 µs pro Interrupt, plus 2 µs Umschaltzeit zwischen zwei Interrupts, kommt man auf eine Zeit von 80 µs. Das bedeutet: 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 ungünstigste Fall nicht möglich ist. Entweder ist der UART-Empfangs-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. Hinzu kommt, dass die Hall-Sensoren-Interrupts, bedingt 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. Messungen ergaben, dass bei einer Spannung von 12 V und maximal eingestellter Geschwindigkeit der Räder jeder Hall-Sensor aller 1,7 ms (d.s. 1700 µs) einen Interrupt generiert. Dieser Interrupt benötigt maximal 8 µs. Während der 1700 µs treten insgesamt 4 Interrupts auf, jeweils einer pro Sensor; d.h. aller 425 µs tritt ein Interrupt mit einer Länge von 8 µs auf. Somit werden zwischen zwei Interrupts der Hall-Sensoren im Durchschnitt 9 bis 10 Hauptschleifen-Iterationen durchgeführt.

6.5 Vergleich mit der Vorläufer-Software

Ein strenger Vergleich zwischen der Vorläufer-Software und der in dieser Arbeit geschriebenen Software ist nur sehr bedingt möglich. Das zugrunde liegende Design von beiden Projekten ist so unterschiedlich, dass hauptsächlich die Dauer der Hauptschleife andeuten kann, inwiefern die eine Software schneller arbeitet als die andere (siehe Tabelle 6.8).
Zusätzlich kann man noch den Speicherbedarf der Systeme vergleichen (siehe Tabelle

6.7). Hierbei fällt auf, dass der generierte Maschinencode im Vergleich zum Vorläufer etwas größer ist. In Anbetracht der benutzten Compiler-Flags bedeutet das aber, dass der Code mit gleichen Flags kleiner wäre als der des Vorläufers. Dies ist auch demonstriert worden, indem die Vorläufer-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; aber dank der höheren Datendichte der Befehlesübertragung im neuen System ist dieses wesentlich schneller (vgl. Tabelle 6.8 und Abb. 6.3).

```
'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 6.3: Vergleich von mehreren Befehlen im alten und neuen Protokoll

Anmerkung: Die vorliegende Abbildung zeigt den kürzesten und den längsten Drive-Befehl im Format des alten und des neuen Protokolls. Eine Zahl oder ein Buchstabe, die von ' eingeschlossen sind, repräsentieren den Wert eines Bytes. Die Abbildung verdeutlicht die Reduktion der zu übertragenden Bytes und die erhöhte Informationsdichte des neuen Protokolls.

Funktion	**vor der Code-Optimierung**	**nach der Code-Optimierung**	**Rahmenbedingungen**
Hauptschleife	27,916 µs	24,858 µs	Idle, LCD an, DEBUG aus, I2C an, AB:EIT
process_orders()	11,964 µs	8,375 µs	
lcd_update_screen()	3,988 µs	5,583 µs	
parser_update()	3,988 µs	5,584 µs	
queue_update()	3,988 µs	4,187 µs	
Hauptschleife	47,856 µs	42,473 µs	ABS aktiv im Idle-Status, ansonsten wie oben
process_orders()	31,904 µs	26,221 µs	
lcd_update_screen()	3,988 µs	5,484 µs	
parser_update()	3,988 µs	5,683 µs	
queue_update()	3,988 µs	4,686 µs	
Hauptschleife	718,843 µs	119,143 µs	LCD aus, Byte in Parser, Befehl bereit
process_orders()	39,880 µs	18,544 µs	I2C-Bus-ISR (Daten) aufgetreten
lcd_update_screen()	0,997 µs	<0,040 µs	
parser_update()	175,474 µs	47,258 µs	
queue_update()	505,484 µs	52,443 µs	
Hauptschleife	3614,000 µs	118,644 µs	LCD an, Befehl in Queue, aber nicht gestartet
process_orders()	358,923 µs	11,167 µs	
lcd_update_screen()	3250,000 µs	97,109 µs	
parser_update()	6,032 µs	5,783 µs	
queue_update()	5,583 µs	3,988 µs	
I2C-Bus-ISR	4,586 µs	6,600 µs	Adresse empfangen
I2C-Bus-ISR	21,934 µs	6,600 µs	Daten empfangen
100ms-ISR	8,076 µs	5,583 µs	

Tabelle 6.2: Ergebnisse der Messungen vor und nach den Code-Optimierungen

Funktion	Zeit	Verbesserung	Rahmenbedingungen
Hauptschleife	421,158 µs	-88,35%	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

Anmerkung: Für diese Tabelle und alle noch folgenden mit einer Rubrik "Verbesserung" gilt:
Prozentwerte der Verbesserung mit Minuszeichen bedeuten eine Reduzierung, mit Pluszeichen eine Erhöhung der ursprünglichen Werte.

Funktion	Zeit	Rahmenbedingungen
Hauptschleife	420,659 µs	LCD an, Befehl in Queue, aber nicht gestartet, ohne -finline-functions
process_orders()	355,289 µs	
lcd_update_screen()	59,481 µs	
parser_update()	4,146 µs	
queue_update()	4,640 µs	

Funktion	Zeit	Verbesserung	Rahmenbedingungen
Hauptschleife	285,643 µs	-32,1%	LCD an, Befehl in Queue, aber nicht gestartet, mit -finline-functions
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	-90,71%

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

Tabelle 6.7: Vergleich der Speicheranforderung mit der Vorläufer-Software

Funktion	Zeit	Verb.	Bedingung
Hauptschleife (Vorläufer)	121,885 µs		Idle
Hauptschleife (mod. Vorl.)	121,635 µs	-0,21%	Idle
Hauptschleife (aktuell)	23,535 µs	-80,69%	Idle
Befehl (kurz, Vorläufer)	3655 µs (13 Byte)		I2C-Bus, kürzester Befehl
Befehl (kurz, aktuell)	1023 µs (4 Byte)		I2C-Bus, kürzester Befehl
Befehl (lang, Vorläufer)	7747 µs (27 Byte)		I2C-Bus, längster Befehl
Befehl (lang, aktuell)	2191 µs (8 Byte)		I2C-Bus, längster Befehl

Tabelle 6.8: Vergleich der Geschwindigkeiten mit der Vorläufer-Software

7 Zusammenfassung und Ausblick

Das Ergebnis dieser Arbeit zeigt, dass die Platine auf Befehle schnell und korrekt reagiert, solange diese nicht gravierend von der 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 Herschwenken 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. Das wiederum eliminiert den Overhead des ABS komplett, wenn dieses abgeschaltet ist. Der 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.
Noch eine Erweiterungsmöglichkeit für das System besteht in erhöhter Robustheit und zusätzlicher 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 ausbaubar implementiert werden, um einerseits nicht gegen die Designprinzipien des Systems zu handeln, andererseits 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 gänzlich ausgeklammert wurde, ist die Möglichkeit, die Motorplatine in einen Idle-Modus zu schicken. In diesem Modus verbraucht die Platine wesentlich weniger Strom. Das würde die Nutzungsdauer der Batterie erhöhen, wenn die Praktikanten an solch einem Fahrzeug arbeiten und die Motorplatine sich für längere Zeit im Leerlauf befindet. In dieser Hinsicht wäre es auch interessant, den Idle-Modus – bzw. das Verhalten des Idle-Modus – durch Befehle während der Laufzeit steuern zu können.

Wie im Kapitel über die Laufzeituntersuchung beschrieben wurde, 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*. 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

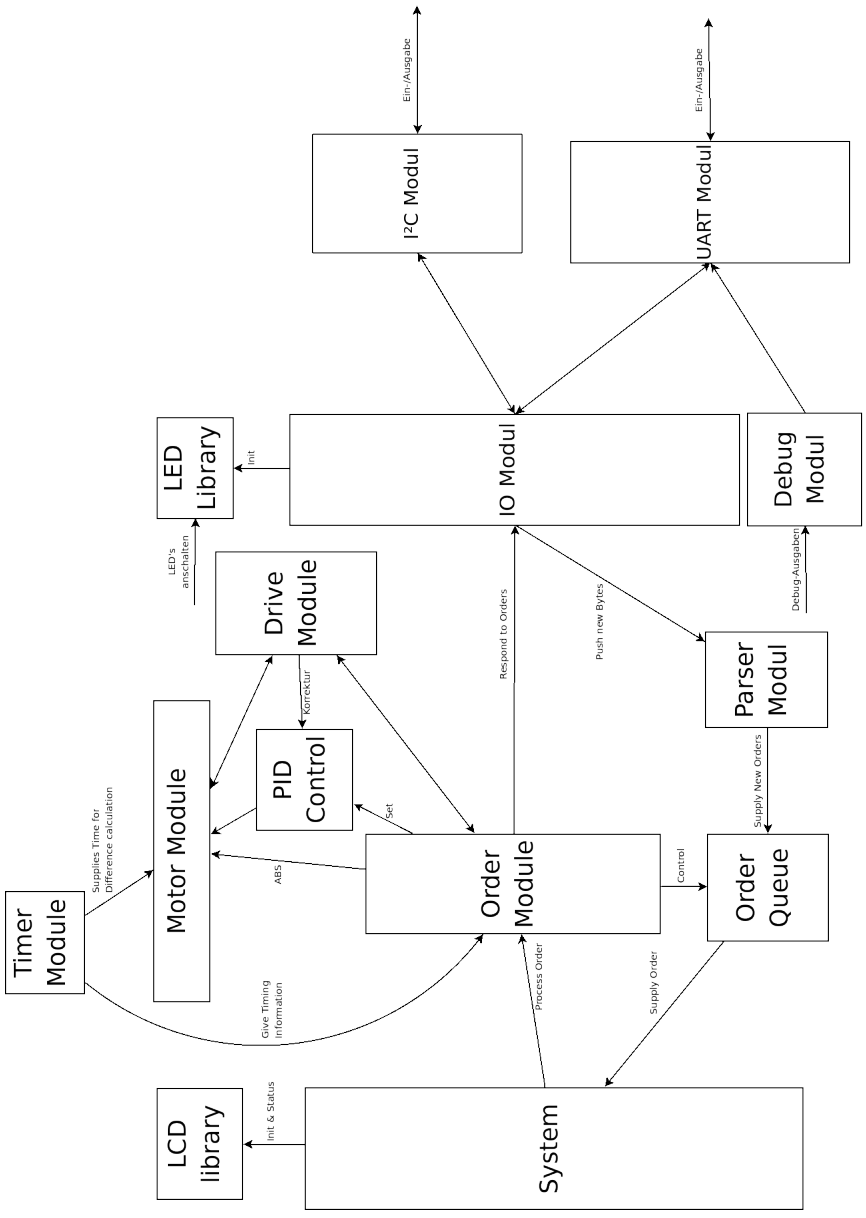

Abbildung A.1: Die Module und ihre Beziehung

B Inhalt der zugehörigen CD

- Der Quellcode der Betriebssoftware (inklusive Makefile).
- Die Quelldateien der vorliegenden Arbeit.
- Die vorliegende Arbeit im PDF.
- Die Quelldateien des Benutzerhandbuches.
- Das Benutzerhandbuch im PDF.
- Die Testreihen, die während der Optimierungsphase erstellt wurden.
- Die Programmcode-Dokumentation als doxygen-HTML-Dokumentation.
- Das Datenblatt des Mikrocontrollers.
- Das Benutzerhandbuch von Timo Klingeberg für die Motorplatine.
- Die Coding-Guidelines als Text-Datei.

C Coding-Guidelines

- Jede Einrück-Ebene wird mit einem Tabulator eingerückt.
- Nach Kommentar-Zeichen folgt ein Leerzeichen.
- Geschweifte Klammern werden direkt nach dem entsprechenden Statement geöffnet.

Bsp.:

```
if (a < b) {
    // Do something
} else if (a > b) {
    // Do another thing
} else {
    // Whatever
}
```

- Funktionen werden nach folgendem Schema benannt:
  modul_namensteil1_..._namensteilN();

Bsp.:

```
    io_obj_start();
```

- Defines und globale Variablen (keine Datei-globale, nur System-globale) werden groß geschrieben.
- Jede Funktion muss einen Doxygen-Kommentar haben.
- Jede nicht-triviale Funktion muss im Code dokumentiert werden.
- Funktionen, die System-globale Variablen ändern, müssen auf diesen Umstand explizit in der Dokumentation hinweisen.
- Parameter sind, wenn möglich, mit const zu deklarieren.
- Lokale Variablen werden sinnvoll benannt und einzelne Namensteile durch _ getrennt.

Bsp.:

```
    uint8_t inbuf_start;
```

- Ein-Zeichen-Variablen sollen nur für for-Schleifen verwendet werden.
- Der Typ der Variablen muss explizit mit Bit-Größe angegeben werden (uint8_t, int8_t, uint16_t, int16_t, etc).