Implementierung einer Betriebssoftware unter Berücksichtigung des Echtzeitverhaltens

Christoph Peltz

7. September 2009

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Bachelorarbeit

Implementierung einer Betriebssoftware unter Berücksichtigung des Echtzeitverhaltens

von
cand. inform. Christoph Peltz

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

Braunschweig, den 7. September 2009

Erklärung

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

Braunschweig, den 7. September 2009

Kurzfassung

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

Abstract

This example abstract is indeed very abstract.

[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 fahren zu lassen. Diese Motorplatine benötigt für seine Operation eine Betriebssoftware, die die Befehle der Praktikumsplatine entgegen nimmt, interpretiert und in Aktionen umsetzen kann. Die Entwicklung dieser Software, sowie 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 Microcontroller 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 Platform, die im Praktikum benutzt wird und somit die Umgebung in der die Software auch eingesetzt wird und deren Anforderungen sie genügen muss.
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 Aspekte der Wartbarkeit und die Möglichkeit die Software einfach erweitern zu können, dominante Designaspekte, kurz darauf folgt die Anforderung, dass die Software ihre Arbeit performant und zuverlässig ausführt.
Um die Performanz und Zuverlässigkeit zu Untersuchen wurde sowohl ein Logikanalyzer als auch ein Oszilloskop benutzt, die Flanken und deren Länge an nach außen gelegten Pins gemessen haben. Diese Pins wurden für den Zweck der Untersuchung von der Betriebssoftware an wichtigen bzw. kritischen Stellen im Programmcode gesetzt und wieder gelöscht.

2 Überblick über die Hardware

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

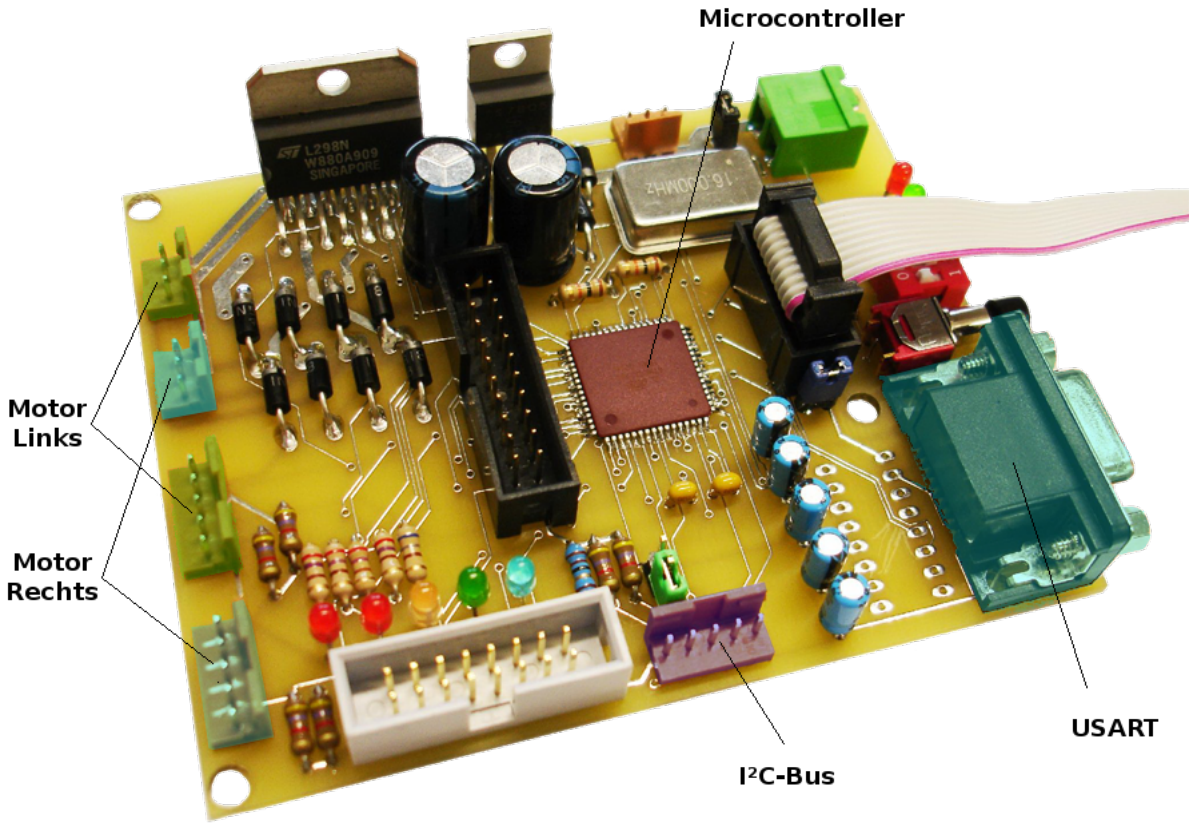

Abbildung 2.1: Die Motorplatine

2.1 Microcontroller

Das Herzstück der Platine bildet ein Microcontroller der Firma Atmel. Es handelt sich hierbei um einen ATMEGA2561[3], der 256 KB Speicher für Programme (Flash) hat, sowie 8 KB Speicher für Variablen (SRAM). Der maximale Takt für diesen Microcontroller liegt bei 16 MHz, dieser wird auch ausgenutzt. Außerdem unterstützt er In-System Programming (ISP), was bei der Entwicklung der Betriebssoftware von großem Nutzen war, da hierdurch schnell und relativ unkompliziert neue Versionen auf die Platine überspielt werden konnten.

2.2 Ein-/Ausgabemöglichkeiten zur Praktikumsplatine

Die Motorplatine verfügt sowohl über einen UART-Port, als auch einen I2C-Bus, über die die Praktikumsplatine mit der Motorplatine kommunizieren kann. Der UART-Port wird außerdem für die Ausgabe von Debug-Informationen benutzt, falls dies aktiviert wird. Im allgemeinen Fall wird allerdings nur der I2C-Bus zur Kommunikation zwischen den beiden Platinen verwendet. Da der Microcontroller über eingebaute Hardwarelogiken sowohl für den I2C-Bus als auch für den UART-Port verfügt, hält sich der administrative Aufwand für die Kommunikation sehr in Grenzen. Es können lediglich Interrupt Service Routinen (ISR) für diese zur Verfügung gestellt werden, die es dann ermöglichen schnell auf Situationen zu reagieren und ohne, dass Informationen verloren gehen könnten.

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 zusammgefasst werden.

3 Einsatz im Praktikum

Im Praktikum ”Programmierung eingebetteter Systeme” wird den Studenten der Umgang mit kleinen Systemen, die sich wesentlich von den bekannten IBM-PC-kompatiblen Systemen unterscheiden. Sowohl was die Rechenleisten, Speicherplatz, als auch Kommunikationsmöglichkeiten angeht. Die meisten Praktkanten kennen nur das Programmieren für ein Betriebssystem, bzw immer mit dem Hintergedanken, dass das Programm auf einem solchen Betriebssystem läuft. Auf diesen kleinen Systemen, mit denen die Studenten im Praktikum konfrontiert werden, läuft kein Betriebssystem, lediglich ein Bootloader, der es ermöglicht neue Programme auf die Platine zu laden, um diese dann auch ausführen zu können. Im Laufe des Praktikums lernen die Studenten, wie man Programme in C schreibt, wie die Timer und andere angeschlossene Geräte verwendet werden, wie z.B. ein LCD oder ein Servomotor. 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 (diese Projektideen kommen auch von den Studenten). Als Platform 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), 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 benutzt werden. Diese Motorplatine wird mithilfe von Befehlen, die die Praktikumsplatine sendet, gesteuert und diese wiederum steuert die Motoren.
Die Motorplatine, für die die Betriebssoftware in dieser Arbeit entwickelt wurde, wird im Allgemeinen von den Studenten nur benutzt, nicht modifiziert (auch wenn ihnen dies freisteht). Diese Leistungen, die die Motorplatine bereitstellt, sollten nicht die Studenten behindern, sondern auch einfach anzusprechen sein, damit diese sich auf das Implementieren ihres eigenes Projektes konzentrieren können.

4 Design und Designentscheidungen

Die Betriebssoftware wurde in Module aufgeteilt, wobei eine Code-Datei einem Modul zugeordnet ist und ein Module mehrere Code-Dateien umfassen kann. Es wurde besonderer Wert darauf gelegt, dass Module so wenig wie möglich andere Module aufrufen und dies auch nur durch Funktionen und nicht durch Variablen. Dieses Prinzip musste allerdings während der Optimierungs- und Testphase geringfügig aufgeweicht werden, da entweder der Code dadurch unleserlich wurde oder der Preis für einen Funktionsaufruf zu hoch war in Relation zu dem Nutzen, der dadurch erzielt wurde. Somit gibt es nun vier bis fünf globale Variablen, die sich während der Laufzeit ändern können und in unterschiedlichen Modulen direkt referenziert werden. Geändert werden diese allerdings nur in sehr wenigen Funktionen, in denen dies auch explizit Kommentiert wurde. Außerdem wurde in den Coding-Guidelines extra auf den Umstand hingewiesen wenn möglich keine Funktionen zu schreiben, die die globalen Variablen verändern und wenn doch dies explizit zu dokumentieren. Neben diesen maximal fünf Verhaltensverändernden globalen Variablen gibt es vier weitere Variablen in zwei verschiedenen Modulen, die jeweils aus einem anderen Modul explizit 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 Motor Interrupts nach und nach dekrementiert. Über die Module wird in den Unterkapiteln genauer eingegangen.

4.1 System

Das als System bezeichnete Modul beinhaltet die Hauptschleife der Betriebssoftware, sowie die Initialisierung aller anderen Submodule. Es benutzt ein paar wenige Module um seine Aufgabe in einem abstrakten Maße zu erfüllen. Während der Hauptschleife, die in gekürzter Fassung (Kommentare und Debug-Ausgaben entfernt) in Abb. 4.1 zu sehen, werden nur ein paar wenige Funktionen aufgerufen, die die wichtigen Module auf dem aktuellen Stand halten und damit es ermöglichen, dass Daten zwischen den Modulen fließen kann.

4.2 Debug

Eigene Mechanismen zum Debugging sind genau dann sehr wichtig, wenn man nicht mit den gewohnten Werkzeugen dies tun kann. Während der Laufzeit verstehen was in dem kleinen Microcontroller vor sich geht ist essentiel, allerdings auch nicht sehr simpel, denn die einzigen Möglichkeiten, die die Platine hat mit der Aussenwelt zu kommunizieren beschränken sich auf 5 LED's, ein 4*20 Zeichen LCD, eine USART und eine I2C-Bus Schnittstelle. Der Aussagegehalt von den LED's (insbesondere da diese sich geweigert haben zuverlässig zu funktionieren) und der LCD ist doch sehr

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

    process_orders();

    if (LCD_PRESENT)
        lcd_update_screen();

    parser_update();

    queue_update();
}
```

Abbildung 4.1: Die Hauptschleife

begrenzt (man kann nicht viele Informationen auf einem 4*20 Zeichen LCD unterbringen). Die Wahl fiel dann auf die USART 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.

4.3 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 Fahrverhalten stabiler wird müssen diese Fehler korrigiert werden.

4.4 IO, I2C und USART

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 USART und I2C. Sowohl das I2C als auch das USART Modul kümmern sich hauptsächlich um das Initialisieren der Hardware und das behandeln der Interrupt Service Routinen.

4.5 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 Hauptschleifeniteration 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 eingereiht werden sondern bei der nächsten Hauptschleifeniteration ausgeführt werden.

4.6 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.

5 Implementierung der Betriebssoftware

Die Betriebssoftware wurde, wie in der Aufgabenstellung festgelegt in C geschrieben. Hierbei viel die Wahl allerdings auf den Standard C99, der mit einigen sprachlichen Aktualisierungen gegenüber C89 aufwarten kann. Als Kompiler 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.

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 Messungsoverhead so gering wie möglich zu halten. Wie hier beschrieben sind

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

die Zeiten für 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 verhältnissmäßig langsam durchgeführt wird, insbesondere das Abfallen des Stromes, also bei einer fallenden Flanke benötigt ungefähr 2 us von denen allerdings, und hier liegt das Problem, zwischen 0.5 und 1 us 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 einzelen Funktionsaufrufe in der Hauptschleife mit pin_set() und pin_clear() umgeben.

```
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

6.2 Das LCD-Problem

6.3 Optimierung

6.3.1 Inlining von Funktionen

6.3.2 Eleminierung von Modulo-Operatoren

6.3.3 Effizienteres Initialisieren des Speichers

6.3.4 Bedingtes Kompilieren von Debugausgaben

6.4 Vergleich mit der Orginal-Software

7 Zusammenfassung und Ausblick

Am Ende dieser Arbeit ist die Platine in der Lage auf Befehle schnell und korrekt zu reagieren, solange diese nicht schwerwiegend fehlgeformt sind. Es gehen keine Interrupts verloren und die Platine reagiert auf Änderungen in Bruchteilen einer Sekunde. Dennoch gibt es einige Punkte, die aufgrund der Mangelnden 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 dieses 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, sowie, dass das Abschalten des ABS gleichzusetzen ist mit dem Abschalten des zugehörigen Interrupts, was wiederum den Overhead des ABS komplett eliminiert, wenn dieses abgeschaltet ist. Dieser Overhead wird auf ungefähr 2,5 us 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 Fehlererkennen 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. voll Buffer, runaway Befehle, Parser Status Korruptionen 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-speedM̈odus operieren zu lassen, oder eine eigene Punkt-zu-Punkt Kommunikation mithilfe der freien Ports auf der Motorplatine zu realisieren. Solch ein Maßgeschneideter 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*.