Entwicklung einer Motorsteuereinheit für ein Fahrmodul

Christian Schröder

17. November 2005

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund

Studienarbeit

Entwicklung einer Motorsteuereinheit für ein Fahrmodul

von
cand. informations-systemtechnik Christian Schröder

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

Braunschweig, den 17. November 2005

Erklärung

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

Braunschweig, den 17. November 2005

Kurzfassung

Diese Studienarbeit behandelt die Entwicklung einer Motorsteuereinheit für ein Fahrmodul, welches am Institut für Betriebssysteme und Rechnerverbund (IBR) der TU Braunschweig für ein Praktikum eingesetzt wird. Das in dieser Arbeit entworfene Board ist als Kernelement mit einen P89C664-Mikrocontroller der Firma Philips bestückt. Die Motorsteuereinheit steuert zwei modifizierte Modellbau-Servos an, die als Fahr-Servos dienen. Dabei erzeugt der Mikrocontroller exakte Steuer-Impulse für die Elektronik der Servos. Außerdem ist es möglich, einen weiteren (nicht modifizierten) Servo beispielsweise für Sensor-Bewegungen zu steuern. Die modifizierten Fahr-Servos liefern Pulse, die dem Mikrocontroller die Messung der zurückgelegten Strecke erlauben. Die Befehle für die Motorsteuerung werden über einen I^2C-Bus von der Hauptplatine des Fahrmoduls gesendet. Diese Arbeit beinhaltet ein eigenes Protokoll, welches die Steuer-Befehle spezifiziert.

Abstract

This paper deals with the development of an engine control unit embedded in a vehicle used for a lab at the IBR at the TU Braunschweig (Technical University of Braunschweig). The designed board contains as main unit a microcontroller P89C664 of Philips company. The engine control unit controls two modified modelbuilding servo which serve as driving engines. The microcontroller produces exact control pulses for the servo electronics. Furthermore one unmodified servo may be controlled for example to turn a sensor. The modified driving servos produce pulses which allow the controller to detect the covered distance. The commands for the engine control unit are sent over I^2C bus and come from the main unit of the vehicle. This paper contains an own protocol specifying these commands.

Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Prof. Dr. L. Wolf

TU Braunschweig · Institut für Betriebssysteme und Rechnerverbund · Postfach 3329 · 38023 Braunschweig

Mühlenpfordtstr. 23
38106 Braunschweig
Telefon: (05 31) 3 91 – 3283
Telefax: (05 31) 3 91 – 5936
WWW: http://www.ibr.cs.tu-bs.de/

Prof. Dr. L. Wolf

Braunschweig, 25.08.2005

Aufgabenstellung für die Studienarbeit

Entwicklung einer Motorsteuereinheit für ein Fahrmodul

vergeben an

Herrn cand. informations-systemtechnik Christian Schröder
Matr.-Nr. 2691192, Email: ch.schroeder@tu-bs.de

Aufgabenstellung

Im Mikroprozessorlabor des Institutes für Betriebssysteme und Rechnerverbund wird das Praktikum "Ubiquitous Computing" für den Studiengang Informations-Systemtechnik durchgeführt. Im Laufe des Praktikums wird auch ein Fahrmodul eingesetzt. Für dieses Fahrmodul soll eine Motorsteuereinheit entwickelt und implementiert werden.

Die Motorsteuereinheit soll zwei für diese Zwecke modifizierte Servomotoren ansteuern, welche als Räder für den Antrieb des Fahrmoduls dienen. Dabei müssen sowohl Drehrichtung als auch Geschwindigkeit der Servomotoren einzeln steuerbar sein. Zur Drehwinkelbestimmung werden von der modifizierten Elektronik des Servomotors Pulse geliefert.

Es ist eine Schaltung zu entwickeln, welche als Mikrocontroller den P89C664 der Firma Philips enthält. Desweiteren soll die Schaltung die Schnittstelle zu den Servomotoren enthalten, eine serielle Schnittstelle für Debug- und Kontrollausgaben sowie einen I2C-Bus zur Übermittlung von Befehlen und zur Ausgabe von Status-Meldungen.

Diese Schaltung ist als gedruckte Schaltung aufzubauen und in Betrieb zu nehmen. Die für die Motorsteuereinheit zu entwickelnde Software muss die Servomotoren ansteuern und die Pulse zur Drehwinkelbestimmung auswerten. Für die Kommunikation über den I2C-Bus ist ein Protokoll zu entwerfen und zu implementieren. Hierdurch sollen mittels einer Befehlsstruktur die Servomotoren kontrolliert steuerbar sein und jederzeit der Status der Motorsteuereinheit abgefragt werden können.

Laufzeit: 3 Monate

Die Hinweise zur Durchführung von Studien- und Diplomarbeiten am IBR sind zu beachten (siehe http://www.ibr.cs.tu-bs.de/lehre/arbeiten-howto/).

Aufgabenstellung und Betreuung:

Prof. Dr. Lars Wolf _____

Dipl.-Ing. Dieter Brökelmann _____

cand. informations-systemtechnik Christian Schröder _____

Inhaltsverzeichnis

1 Einleitung **1**

2 Hardware **3**
2.1 Servos . . . 3
2.1.1 Impuls-Ansteuerung . . . 3
2.1.2 Puls-Generierung . . . 5
2.2 Anschluss an das Fahrmodul . . . 6
2.3 Board . . . 7
2.4 Mikrocontroller . . . 12

3 Software **15**
3.1 Definitionen (MOTOR_DEFS.H) . . . 15
3.2 Hauptprogramm (main) . . . 16
3.3 Warteschlange (fifo) . . . 17
3.4 Impulse generieren (servotiming) . . . 17
3.5 Flash (programDataByte und eraseBlock) . . . 20
3.6 Puls-Zählung (tickcounttiningTimer) . . . 21
3.7 Protokoll . . . 23
3.7.1 Aufbau der Befehle . . . 23
3.7.2 Befehl-Spezifikation . . . 24

4 Anwendung **33**
4.1 Motortest-Programm . . . 33
4.2 Kalibrierung . . . 34
4.2.1 Anwendungsvorschlag Kalibrierung . . . 36

5 Zusammenfassung und Ausblick **39**

Literaturverzeichnis **41**

A Listings **43**

B Board **49**

Abbildungsverzeichnis

2.1 Impulse für die Servo-Ansteuerung 4
2.2 Anschluss-Schema nicht modifizierter Servo 5
2.3 Modifizierter Servo Innenansicht 5
2.4 Anschluss-Schema modifizierter Servo 6
2.5 Fahrmodul 7
2.6 Schematischer Anschluss der Motorsteuereinheit an das Fahrmodul 8
2.7 Schaltbild der Spannungsversorgung 9
2.8 Anschluss-Schema I^2C-Steckverbinder 10
2.9 Schaltbild der Reset-Schaltung 11
2.10 Anschluss-Schema Port-Steckverbinder 12

3.1 Drei mit einem Timer erzeugte Impulse. 18

B.1 Bauteile-Ansicht 49
B.2 Löt- und Bestückungsseiten 49
B.3 Bestückungsseite 50
B.4 Lötseite 50
B.5 Schaltbild 51

Tabellenverzeichnis

3.1 Protokoll - Byteweiser Aufbau eines Befehls bzw. einer Nachricht 23
3.2 Protokoll - I^2C-Befehle bzw. -Nachrichten mit ID 24

B.1 Stückliste Teil 1 53
B.2 Stückliste Teil 2 54

Listings

A.1 servotiming.c: onTimer0() 44
A.2 programDataByte.c: programDataByte(...) 46
A.3 eraseBlock.c: eraseBlock() 47

1 Einleitung

In dieser Studienarbeit wird eine Motorsteuereinheit beschrieben, die in dem Fahrzeug (Fahrmodul) des Praktikums „Ubiquitous Computing“ (Informations-Systemtechnik II) am Institut für Betriebssysteme und Rechnerverbund der TU Braunschweig eingesetzt werden soll.

Das Fahrzeug besitzt eine Hauptplatine, deren Controller von den Praktikanten programmiert wird. Die Hauptplatine steuert über einen I^2C-Bus die hier entwickelte Motorsteuereinheit, welche die Befehle in Signale für die Antriebsmotoren umwandelt. Das Protokoll ist so entwickelt worden, dass es leicht verständlich ist, da für die Durchführung des Praktikums den Teilnehmern begrenzte Zeit zur Verfügung steht. Die Motoren sind modifizierte Modellbau-Servos und müssen über zeitlich sehr genaue Signale angesprochen werden. Diese Signal-Generierung ist die Hauptaufgabe der Motorsteuerung. Das Fahrzeug kann über verschieden gewählte Geschwindigkeiten der Motoren oder durch gegenläufig drehende Motoren gesteuert werden. Zur Kontrolle der Bewegung liefert die Motorsteuereinheit Informationen über die zurückgelegte Strecke der beiden Motoren über den I^2C-Bus an die Hauptplatine zurück. Zusätzlich ist es möglich, einen dritten (auch nicht modifizierten) Servo zu steuern, der beispielsweise einen Sensor bewegen kann.

Die Motorsteuereinheit ist eine Platine, auf welcher als Hauptkomponente der Mikrocontroller P89C664 der Firma Philips eingesetzt wird. Es sind Schnittstellen für drei Servomotoren, für die serielle Ausgabe von Kontroll-Daten und für die I^2C-Kommunikation mit der Hauptplatine zur Übermittlung von Steuer-Befehlen und Status-Meldungen vorhanden.

Der Grund für die Entwicklung einer neuen Motorsteuerung zu dem existierenden Fahrmodul ist eine andere Art, die Servos anzusteuern: Zuvor wurden die Servos als Gleichstrommotor betrieben. Zu diesem Zweck musste die Servo-Elektronik entfernt werden und auf der Motorsteuerplatine eine Treiber-Elektronik aufgebaut werden. In der hier entwickelten Motorsteuerung werden die Servos direkt über ihre eigene Servo-Elektronik angesprochen. Dadurch ist es nicht nötig, eigene Steuer-Elektronik auf der Motorsteuerplatine bereitzustellen. Der originale Signal-Eingang des Servos wird direkt an einen Mikrocontroller-Port angeschlossen, lediglich verstärkt durch einen Inverter. Die Servo-Elektronik muss folglich nicht entfernt werden. Der Servo wird um eine Puls-Elektronik erweitert (siehe Abschnitt 2.1 in Kapitel 2). Durch Entfernen einer Blockierung im Servo wird ein Rundumdrehen ermöglicht. Neben der Motorsteuerung, die Inhalt dieser Arbeit ist, soll das Fahrmodul selbst neu konstruiert werden.

Die Studienarbeit dokumentiert die Entwicklung der Hard- und Software, die in dieser Motorsteuereinheit zum Einsatz kommt. Sie ermöglicht den Nachbau des Boards, beschreibt die Software und beinhaltet ein separates Benutzerhand-

buch, dessen Lektüre für die Verwendung der Motorsteuerung im Fahrmodul ausreicht.

Kapitel 2 beschreibt zunächst die umgebende Hardware, mit der die Motorsteuerung kommunizieren muss. Das sind die Servos (Abschnitt 2.1) und das Fahrmodul (Abschnitt 2.2) selbst. Anschließend behandelt das Kapitel die im Rahmen dieser Arbeit entwickelte Hardware. In Kapitel 3 wird die Realisierung des Softwareteils der Aufgabe beschrieben. Neben der Funktionsweise des Programms ist das Protokoll aufgeführt, mit dessen Hilfe die Motorsteuereinheit über den I^2C-Bus angesprochen wird. Das Kapitel 4 zeigt Anwendungen auf, die die Motorsteuerung ermöglicht.

2 Hardware

In diesem Kapitel wird die Hardware beschrieben, die in dieser Studienarbeit zur Realisierung der Motorsteuereinheit verwendet wird. Das sind die Servos, die als die anzusteuernden Motoren verwendet werden, das Fahrmodul, der Mikrocontroller der Firma Philips mit seinen für dieses Projekt nützlichen Eigenschaften, sowie das Board als Ganzes.

2.1 Servos

Die von der Motorsteuereinheit angesteuerten Servos erfüllen zwei Funktionen. Sie drehen sich, um als Motor oder herkömmlicher Servo zu agieren. Zusätzlich liefern sie Pulse zur Drehwinkelbestimmung.

2.1.1 Impuls-Ansteuerung

Die hier beschriebene Motorsteuereinheit soll bis zu drei Servos ansteuern. Handelsübliche Modellbau-Servos sind dazu vorgesehen, sich lediglich in einem Bereich von 120 Grad zu drehen. Je nach Modell kann dies auch etwas mehr sein. Um aber für das Praktikum von Nutzen zu sein, müssen sie sich um 360 Grad fortlaufend drehen können. Aus diesem Grund werden sie modifiziert. Die „Nase“ im Getriebe des Servos, die ein Herumdrehen verhindert, wird entfernt. Dadurch ist es mechanisch möglich, den Servo-Kopf rundum zu drehen. Die Elektronik im Servo erkennt anhand eines Potentiometers die aktuelle Position des Kopfes. Damit der Servo die Rundum-Bewegung durchführen kann, wird zusätzlich das Potentiometer festgeklebt und vom drehenden Kopf entkoppelt. Nun kann die Elektronik nicht mehr feststellen, in welcher Position sich der Servo befindet. Für die so modifizierten Servos ist es möglich, als Motor Rundum-Drehungen auszuführen. Die modifizierten Servos wurden von meinem Betreuer zur Verfügung gestellt.

Nicht modifizierte Servos werden durch Impulse an ihrer Steuersignal-Leitung gesteuert[1]. Diese HIGH-Impulse (+5 Volt) haben einen Abstand von etwa 20 ms. Die Impulse sind zwischen 1,0 ms und 2,0 ms lang und stehen für eine Position innerhalb der 120 Grad. Die Mittelposition ist folglich bei etwa 1,5 ms. Abbildung 2.1 zeigt eine Ansteuerung mit 2 ms-Signalen. Da die hier verwendeten Servos eine größere Drehbewegung beherrschen, kann die Software Impulse zwischen 0,5 ms und 2,5 ms erzeugen, was zu einer größeren Drehbewegung führt (> 120 Grad). Wichtig ist, dass der Abstand zwischen den Impulsen nicht exakt 20 ms betragen muss. Dieser Wert darf schwanken.

Werden die Servos nun wie oben beschrieben modifiziert, führen die Impulse verschiedener Länge nicht mehr zu einer absoluten Winkel-Position, die der

[1] Quelle: Servotext bereitgestellt von Dieter Brökelmann. Braunschweig 2004

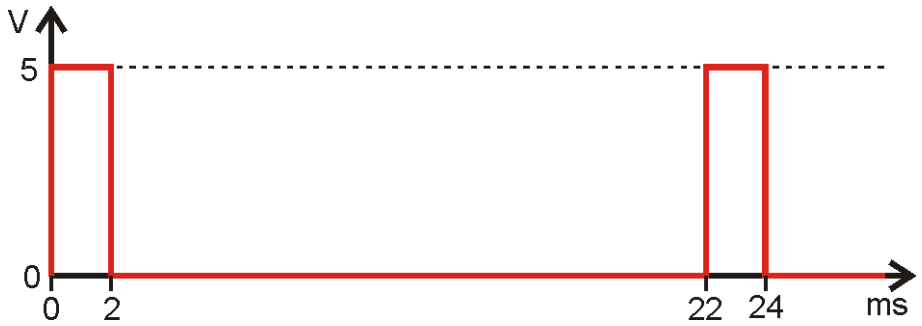

Abbildung 2.1: Impulse für die Servo-Ansteuerung

Servo einnimmt, sondern bewirken verschiedene Geschwindigkeiten. Durch die Arretierung des Potentiometers geht die Servo-Elektronik davon aus, ständig in Mittelposition zu sein, und steuert die angelegten Positionen dann mit verschiedenen Geschwindigkeiten an. Da die Position nie erreicht wird, entsteht eine Drehbewegung mit einer bestimmten Geschwindigkeit. Ein Impuls nahe dem Mittelwert von 1,5 ms veranlasst die Servo-Elektronik dazu, eine kleine Winkel-Änderung durchzuführen, weswegen sich der Servo sehr langsam dreht. Da diese Position nie erreicht wird, dreht sich der Servo ständig mit langsamer Geschwindigkeit, bis der Impuls wieder die Mittelstellung angibt. Ein Impuls nahe dem Maximalausschlag lässt die Servo-Elektronik eine große Winkel-Änderung durchführen, die zu einer höheren Geschwindigkeit führt. Analog zur niedrigen Geschwindigkeit wird die Position nie erreicht, folglich dreht der Servo ständig mit hoher Geschwindigkeit.

Die Software der Motorsteuerung ist auf die Steuerung zweier modifizierter Servos und eines nicht modifizierten ausgelegt. Die zwei modifizierten Servos sollen als Antrieb (rechts, links) für das im Praktikum verwendete Fahrzeug dienen. Der nicht modifizierte kann frei verwendet werden, beispielsweise zur Bewegung eines Sensors. Da weder Soft- noch Hardware erkennen können, ob ein Servo modifiziert ist oder nicht, sind diese beliebig austauschbar, wenn die Anwendung Sinn ergibt. Die Antriebs-Servos haben im Programm und der Dokumentation die Nummern 0 und 1, der zusätzliche Servo die Nummer 2. Die Steuerung der Servos geschieht über den I/O-Port 2 des Mikrocontrollers. Servo 0 wird über den Ausgangs-Pin P2.4, Servo 1 über Pin P2.5 und Servo 2 über Pin P2.6 angesteuert (vgl. Schaltplan in Abbildung B.5).

Für die Ansteuerung der Servos sind drei Leitungen notwendig, die auch in handelsüblichen Servos schon vorhanden sind. Servo 2 besitzt diese Anschlüsse unmodifiziert und kann mit dem original gelieferten Stecker (J/R-Anschluss) auf die Motorsteuereinheit aufgesteckt werden. Die Anschlüsse der Servos 0 und 1 werden im folgenden Unterabschnitt aufgeführt. Die drei Anschluss-Leitungen für Servo 2 (siehe auch Abbildung 2.2) sind

Pin 1: (Impuls-)Signal (hier gelb).

Pin 2: Versorgungsspannung V_{EE} +5V (hier rot)

Pin 3: Masse 0V (hier schwarz)

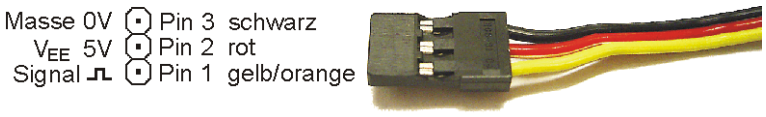

Abbildung 2.2: Anschluss-Schema nicht modifizierter Servo (Servo 2)

2.1.2 Puls-Generierung

Die zwei Fahr-Servos des Fahrmoduls sind zusätzlich zur oben beschriebenen Modifikation mit einer Puls-Generierung erweitert. Das bedeutet es steht ein zusätzliches Signal zur Verfügung, das die Drehwinkelbestimmung ermöglichen soll.

Ein Getriebezahnrad im Gehäuse des Servos ist mit Markierungen versehen. Diese werden von einem Infrarot-Sensor abgetastet und mit Hilfe eines Operationsverstärkers umgewandelt. Die Platine ist in Abbildung 2.3 im Innern des Servos zu sehen. Die Signalpegel, die an der herausgeführten Leitung „Puls“ anliegen, sind 0V und 5V, je nach abgetastetem Bereich des Zahnrades. Der Counter der Motorsteuerung verarbeitet die negativen Flanken (siehe Abschnitt 3.6).

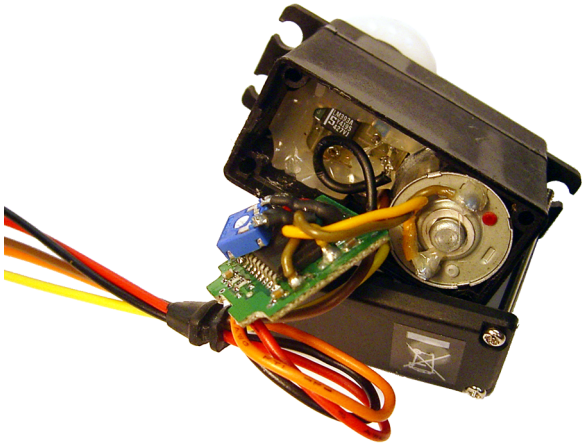

Abbildung 2.3: Modifizierter Servo Innenansicht

Für den Anschluss der zwei modifizierten Fahr-Servos sind zusätzlich zu den drei Anschlüssen eines unmodifizierten Servos (vgl. Unterabschnitt 2.1.1) zwei weiterere Anschlüsse für den Puls-Ausgang und die 5 V-Spannungsversorgung für die Pulserzeuger-Platine notwendig. Deswegen sind die Anschlüsse für die Fahr-Servos ebenfalls modifiziert. Es wird ein 5-poliger Platinen-Steckverbinder verwendet, welcher die folgende Belegung aufweist (siehe auch Abbildung 2.4):

Pin 1: Masse 0V (schwarz)

Pin 2: Puls (vom Servo) (braun)

Pin 3: Impuls/Signal (zum Servo) (rot)

Pin 4: V_{EE} (Versorgungsspannung Servo +5V) (orange)

Pin 5: V_{CC} (Versorgungsspannung Puls-Generator +5V) (gelb)

Die an diesem Stecker vorhandene Versorgungsspannung V_{CC} wird von der Puls-Elektronik benötigt. Die Spannung V_{EE}, mit der die drei Servos versorgt werden, wird mit einem zusätzlichen Spannungsregler erzeugt, damit von den Servo-Motoren erzeugte Störungen die Puls-Elektronik und den Mikrocontroller nicht beeinflussen. An dieser Platine ist einheitlich an jedem Platinen-Steckverbinder an Pin 1 Masse und am höchsten Pin V_{CC} angelegt.

Die Puls-Signale, die von den Servos 0 und 1 kommen, werden in den Mikrocontroller an den Eingangs-Pins der Timer/Counter 2 und Timer/Counter PCA eingespeist. Das ist Mikrocontroller-Pin P1.0 (Timer/Counter 2) für Servo 0 und Pin P1.2 (Timer/Counter PCA) für Servo 1 (vgl. Schaltplan in Abbildung B.5).

Die in diesem Abschnitt beschriebenen modifizierten Servos wurden von meinem Betreuer zur Verfügung gestellt.

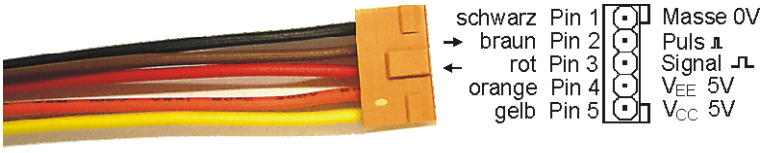

Abbildung 2.4: Anschluss-Schema modifizierter Servo (Servo 0 oder Servo 1)

2.2 Anschluss an das Fahrmodul

Die in dieser Arbeit beschriebene Motorsteuereinheit befindet sich im Gesamt-Kontext des Fahrmoduls für das Praktikum. Dieses Fahrmodul verfügt über die zwei erwähnten modifizierten Servomotoren, die als Antrieb und Steuerung dienen. Ein drittes nicht steuerbares Rad verhindert ein Kippen des Fahrzeugs. Neben den Fahr-Servos befindet sich ein dritter Servo zur Steuerung eines Abstands-Sensors an der Front des Fahrmoduls. Ein Display ermöglicht das Darstellen von Informationen, die folglich auch ohne (seriellen) Anschluss an den PC ablesbar sind. Ein Akku (7,2 Volt) übernimmt die Stromversorgung. Die drei Servos werden durch die Motorsteuereinheit gesteuert. Die Gesamtsteuerung des Moduls übernimmt die Hauptplatine mit einem P87C552-Controller, welcher das Programm der Praktikums-Teilnehmer enthält. Die Kommunikation zwischen den beiden Platinen wird über den I^2C-Bus realisiert.

Außerdem verfügt das Fahrmodul an vielen Stellen über Erweiterungsmöglichkeiten, die individuell von den Praktikanten genutzt werden können.

Abbildung 2.5 zeigt ein ausgestattetes Fahrmodul im Einsatz. Schematisch verdeutlicht Abbildung 2.6 den Anschluss der Motorsteuereinheit an das Fahrmodul. Die seriellen Anschlüsse an den PC dienen jeweils zur Programmierung, Befehlsübermittlung und Ausgabe von Debug-Informationen. Je nach Anwendung kommunizieren verschiedene Programme über die Schnittstelle. Die Programmierung beider Controller in den Sprachen C oder Assembler wird in μVision [2] vorgenommen. Das Hochladen des Programms für die Hauptplatine wird von μVision im Monitor-Modus vorgenommen. Dort ist auch die Eingabe von Befehlen und das Ablesen von Ausgaben möglich. Die Software der Motorsteuereinheit dagegen wird mit Flash Magic[3] hochgeladen. Anschließend wird die Kommunikation über ein beliebiges Terminal-Programm (z.B. das HyperTerminal von MS Windows) durchgeführt. Die einzustellenden Parameter finden sich im Benutzerhandbuch[3].

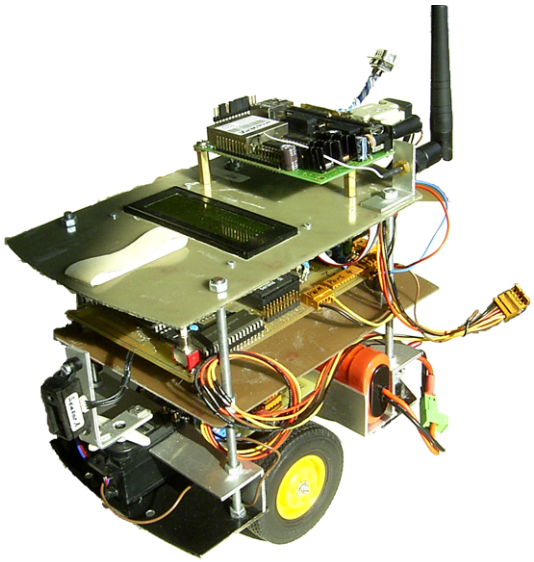

Abbildung 2.5: Fahrmodul, dessen Motoren von der Motorsteuereinheit gesteuert werden. In der Mitte ist die große Hauptplatine zu sehen. Darunter befindet sich neben dem Akku die Motorsteuer-Platine. Dieses Fahrmodul ist zusätzlich mit einem WLAN-Modul ausgestattet.

2.3 Board

Es folgt eine Beschreibung des entwickelten Boards für die Motorsteuerung. Es basiert auf dem bisherigen Board motor_v3 vom 02.10.2003, da dieses das von mir verwendete Experimentier-Exemplar ist und die meisten Elemente schon enthält und wenige weggelassen werden. Abbildung B.5 im Anhang zeigt das

[2] μVision ist eine Entwicklungsumgebung der Firma Keil Software.
[3] Flash Magic ist eine kostenlose ISP Software für Philips Flash Mikrocontroller. Bezugsquelle: `http://www.esacademy.com/software/flashmagic/`

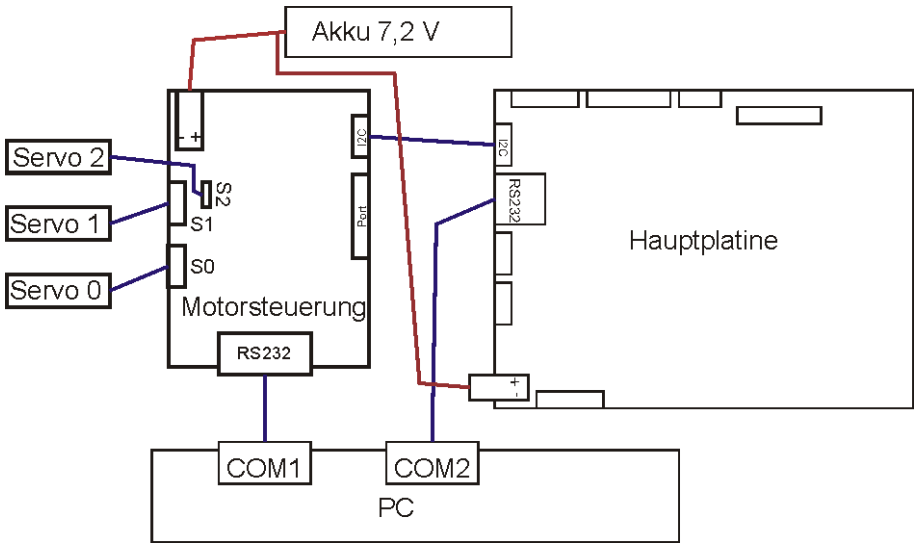

Abbildung 2.6: Schematischer Anschluss der Motorsteuereinheit an das Fahrmodul

Schaltbild des von mir entwickelten Boards, anhand dessen im Folgenden die Elemente beschrieben sind. Ebenfalls im Anhang B befinden sich einige Grafiken zum Aufbau der Platine, auf die hier nicht näher eingegangen wird.

Das Motorsteuer-Board hat die Elemente:

- Spannungsversorgung
- Mikrocontroller P89C664 von Philips (siehe Abschnitt 2.4)
- Servo-Treiber
- Puls-Aufarbeitung
- Serielle Schnittstelle
- I^2C-Bus
- Reset-Schalter
- Oszillator
- zusätzliche Schnittstelle für Erweiterungen

Die auf dem Board integrierte **Spannungsversorgung** verteilt die ankommenden 7,2 Volt auf zwei unabhängige 5 Volt auf. V_{EE} dient der Versorgung der Servo-Motoren und wird mit dem Spannungsregler U2 (78S05) erzeugt, welcher 2 Ampere liefert und mit einem Kühlkörper gekühlt werden muss. Der Spannungsregler U1 (7805) erzeugt ebenfalls 5 Volt (V_{CC}), die der Elektronik zugeführt werden. Aufgrund des geringeren Stromverbrauchs der Elektronik liefert U1 lediglich 1 Ampere und braucht nicht gekühlt zu werden. Die rote LED zeigt die Existenz der Versorgungsspannung an. In der Abbildung 2.7 ist der Spannungsversorgungs-Teil des Motorsteuer-Boards abgebildet. Die

Kondensatoren um die Spannungsregler herum stabilisieren die Spannung. Die Elektrolytkondensatoren mit hoher Kapazität C5 und C6 filtern Verbrauchsspitzen heraus, die Kondensatoren C4, C7, C12 und C13 filtern hochfrequente Störungen.

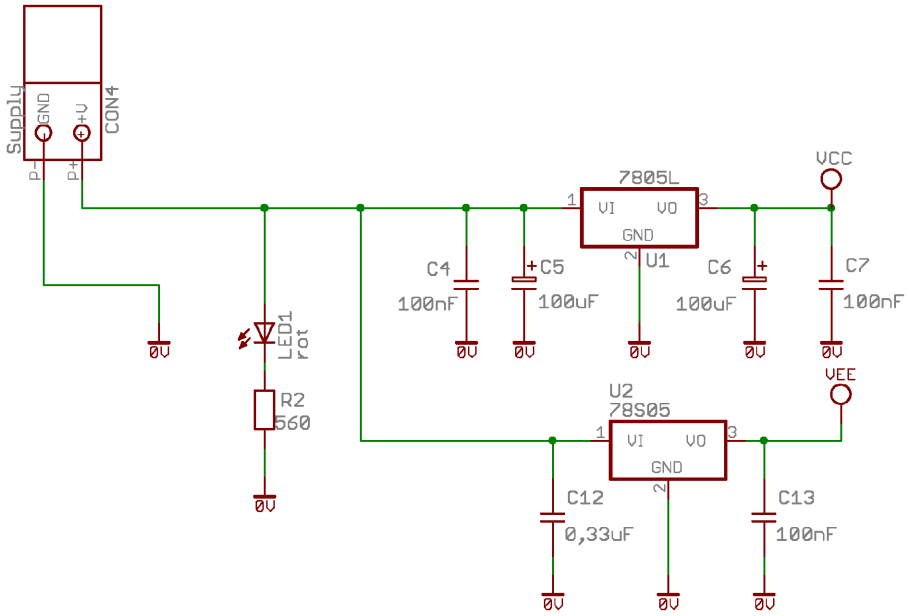

Abbildung 2.7: Schaltbild der Spannungsversorgung

Wie in Abschnitt 2.1.1 beschrieben werden die Servos vom Mikrocontroller über die Ports P2.4, P2.5 und P2.6 angesteuert. Um die Servo-Elektronik vom Controller zu entkoppeln, werden **Servo-Treiber** eingesetzt. Das Signal wird mit Hilfe von drei (der sechs) Schmitt-Trigger-Invertern im Baustein IC1 (IC 7414) verstärkt und dann den Servos zugeführt. Die Genauigkeit des Signals wird dadurch erhöht, dass es sich um Inverter handelt. Ein Impuls, der im Programm durch `HIGH` gekennzeichnet ist, wird im Modul `MOTOR_DEFS.H` durch eine 0 umgesetzt (entspricht 0 V). Die Controller-Elektronik erzeugt diese 0 sauberer und schneller als eine 1 (d.h. +5 V). Im Inverter wird diese 0 wieder umgesetzt in einen sauberen positiven Impuls von +5 V. Da der Servo 2 nicht modifiziert ist, kann auch der herkömmliche Stecker auf das Board gesteckt werden (CON2). Die Belegung ist in Abbildung 2.2 dargestellt. Die beiden modifizierten Fahr-Servos dagegen besitzen modifizierte Steckverbinder (CON0 und CON 1). Mit Ausnahme des gerade beschriebenen Steckers für Servo 2 besitzen alle auf dem Board herausgeführten Steckverbinder am Pin 1 0 V und am höchsten Pin V_{CC}. Der Stecker der Fahr-Servos ist in Abbildung 2.4 dargestellt. Die verstärkten Signale der Steuer-Pins sind am Stecker-Pin 3 bereitgestellt.

Die von den Servos generierten Pulse für die Messung von Entfernungsschritten werden in der **Puls-Aufarbeitung** für die Mikrocontroller-Eingänge vorbereitet. Zu diesem Zweck glätten Kondensatoren (C14 und C15) das Signal, so dass hochfrequente Störungen nicht als Puls gezählt werden. Die nachge-

schalteten Schmitt-Trigger-Inverter im Baustein IC1 sorgen ebenfalls für ein gesäubertes Signal.

Die **serielle Schnittstelle** (CON5) ist über einen MAX232 (IC2) an den Controller angeschlossen. Der Controller verfügt über eine interne full-duplex serielle Schnittstelle, die die benötigten Signale RXD für den Empfang (receive) und TXD für das Senden (transmit) zur Verfügung stellt. Die Schnittstelle wird im Mode 1 betrieben, d.h. 8-Bit UART mit variabler Baud-Rate. Die Signale RXD und TXD des Controllers verwenden TTL-Pegel 0 Volt und 5 Volt. Die serielle RS-232-Schnittstelle eines Computers verwendet dagegen die Pegel +/-12 Volt (eine logische 0 wird dargestellt durch +12 V, eine logische 1 durch -12 V). Der integrierte Schaltkreis MAX232 wird benötigt, um die Pegel anzupassen. Er wandelt das vom Controller kommende Signal TXD in TX um, welches an Pin 3 des 9-poligen D-SUB-Steckers CON5 herausgeführt wird. Das vom Computer kommende Signal RX (Pin 2 von CON5) wird zu RXD gewandelt.

Da der Mikrocontroller über eine **I²C-Bus**-Einheit verfügt, genügt es, auf dem Board einen Stecker (CON6) zu platzieren, der direkt vom Controller angesprochen wird. Die Pins 2 (SDA) und 3 (SCL) sind die Signalisierungs-Leitungen des I^2C-Busses. Pin 1 (0V) führt die Masse. An Pin 5 lässt sich per Jumper V_{CC} zuschalten, was in dieser Anwendung nicht notwendig ist. Pin 4 (INT) ist eine zusätzliche Leitung, die zum Interrupt-Eingang $\overline{\text{INT0}}$ (P3.2) führt. Dieser Pin hat ebenfalls keine Funktion in dieser Anwendung, ermöglicht es aber, zusätzliche Funktionen hinzuzufügen. Außerdem verfügt die Hauptplatine des Praktikums über einen entsprechend passenden Stecker, der mit dieser Konfiguration nicht verändert werden muss.

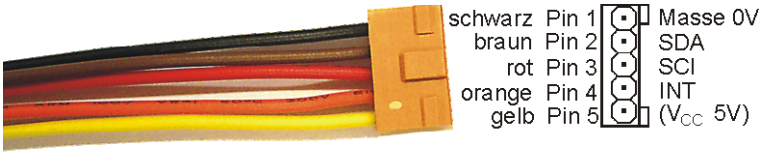

Abbildung 2.8: Anschluss-Schema I^2C-Steckverbinder

Zum Takten des On-Chip-Oszillators des Controllers wird ein **Quarz** benötigt. Dieses ist der Baustein Q1, welcher mit der Frequenz (Clock-Zyklus) $f_{OSC} = 11,0592$ MHz schwingt. Das führt zu einer Maschinen-Takt-Frequenz von $f_{MA} = \frac{1}{T_{MA}} = \frac{1}{6} * f_{OSC} = 1,8432$MHz, da die Bearbeitung eines Maschinen-Befehls sechs Clock-Zyklen benötigt[1].

Der **Reset-Schalter** JP6 (ResetSwitch) schaltet bei Betätigung V_{CC} an den Controller-Eingang RST durch und verursacht damit einen Reset. Das Schaltbild in Abbildung 2.9 zeigt den Reset-Taster SW2 samt Beschaltung. Der Kondensator C3 bewirkt den Power-On-Reset. Unter Verwendung der oben berechneten Frequenz f_{MA} beträgt die Dauer eines Maschinen-Zyklus $T_{MA} = \frac{1}{f_{MA}} = \frac{1}{1,8432\text{ MHz}} = 542,5$ ns. Laut Spezifikation[4] muss das Reset-Signal zwei Maschinen-Zyklen lang anliegen, damit der Controller das Reset ausführt, das

[4]Reset-Signal-Dauer vgl. [1], S. 10

sind folglich $2 * 542,5$ ns $= 1,085$ ms. Zusätzlich muss dem Oszillator genügend Zeit für den Start-Up gegeben werden. Deswegen sind einige weitere Millisekunden zu überbrücken. Diese Zeit muss der Kondensator beim Einschalten der Versorgungsspannung die Spannung V_{CC} an den Controller-Eingang RST liefern. Der interne Widerstand des RST-Eingangs des Controllers liegt zwischen 40 kΩ und 225 kΩ[5]. Parallel dazu ist der Widerstand R1 mit 10 kΩ geschaltet. Zusammen ergibt die Parallelschaltung minimal (das heißt bei einem Innenwiderstand von 40 kΩ) $R = \frac{1}{\frac{1}{10\ \text{k}\Omega} + \frac{1}{40\ \text{k}\Omega}} = 8$ kΩ. Der Kondensator ist mit 10 μF so gewählt, dass er minimal die Dauer von $\tau = R * C = 8\ \text{k}\Omega * 10\ \mu\text{F} = 80$ ms überbrücken kann. Die Bedingungen für ein Reset sind somit großzügig erfüllt.

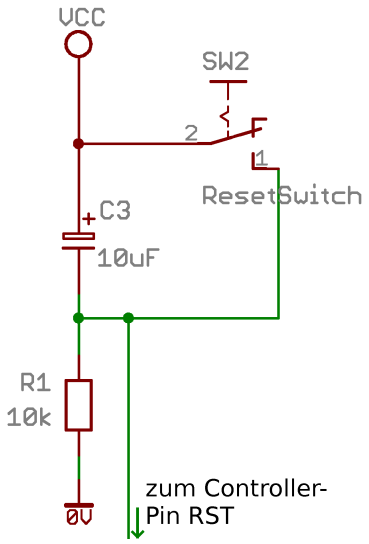

Abbildung 2.9: Schaltbild der Reset-Schaltung

Die Schnittstelle CON3 ist für die Motorsteuerung nicht notwendig. Es handelt sich um eine **zusätzliche Schnittstelle**, die die freien Pins 2.0, 2.1, 2.2 und 2.3 des Controller-Ports 2 nach außen führt. Die Wahl fiel auf diesen Port, weil er über interne Pull-ups verfügt und die Pins somit direkt als Ein- oder Ausgänge verwendet werden können. An diesem Steckverbinder steht zusätzlich neben 0 V die Versorgungsspannung (V_{CC}) zur Verfügung. Die Pin-Belegung wird in Abbildung 2.10 ersichtlich. Vier nicht angeschlossene Ausgänge ermöglichen das Anlöten von weiteren Drähten, die beispielsweise weitere Pins des Controllers oder andere Signale an dem Steckverbinder zur Verfügung stellen. So ist es möglich, an dem 10-poligen Steckverbinder acht Pins zur Verfügung zu stellen. Diese Modifikationen bieten Erweiterungsmöglichkeiten für zukünftige Verwendungen. Die Motorsteuereinheit könnte weitere Aufgaben übernehmen (z.B. Motoren oder Sensoren steuern), ohne ein neues Design der Platine notwendig zu machen.

[5]Interner Reset Pull-Down Widerstand vgl. [1], S. 76

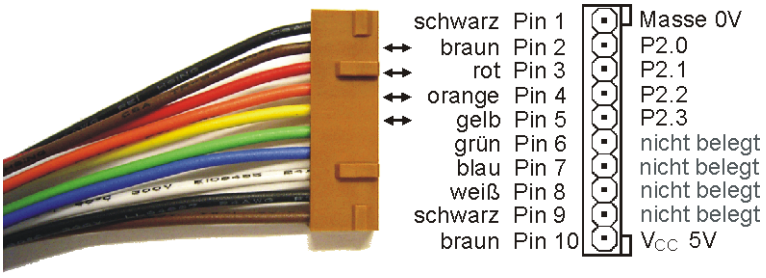

Abbildung 2.10: Anschluss-Schema Port-Steckverbinder

2.4 Mikrocontroller

Bei dem verwendeten Controller handelt es sich um einen Mikrocontroller vom Typ P89C664 der Firma Philips[1], der einen 8051-Kern enthält. Dieser in der Aufgabenstellung vorgegebene Mikrocontroller hat verschiedene Vorteile für dieses Projekt:

Eine für die Programmierung des Controllers geeignete *Entwicklungsumgebung* ist μVision von der Firma Keil Software[6]. Diese wird bisher schon im Praktikum verwendet, ist folglich vorhanden und muss nicht angeschafft werden. Des Weiteren ist es den Praktikums-Teilnehmern möglich ohne lange Einarbeitungszeit Veränderungen an der Motorsteuerung vorzunehmen. Durch die Bereitstellung eines universellen Protokolls (siehe Abschnitt 3.7) soll dieses aber überflüssig sein. Ein weiterer Vorteil ist die genau passende Ausstattung des Controllers. Er verfügt über einen *Flash-Speicher*, der *In-System Programming* (ISP) und *In-Application Programming* (IAP) unterstützt, kann also ohne ausgebaut zu werden sogar von der Endbenutzer-Applikation beschrieben werden (siehe auch Abschnitt 3.5). Verschiedene *Interrupts* lassen sich vier *Prioritäts-Stufen* zuweisen. Dadurch wird es möglich, die hoch priore Impuls-Generierung sehr genau erfolgen zu lassen, da die Interrupt-Routine für die Steuerung dieser Impulse nicht durch andere Interrupts niedrigerer Priorität unterbrochen werden kann. Die im Mikrocontroller vorhandenen Schnittstellen entlasten die Programmierung und die Bestückung der Platine erheblich. So ist eine *serielle Schnittstelle* vorhanden, die in diesem Projekt zur Programmierung und Fehlerkontrolle eingesetzt wird. Die ebenfalls vorhandene serielle *I^2C-Schnittstelle* wird für die Steuerung der Motorsteuereinheit und Status-Meldungen verwendet. Die 8-Bit *I/O-Ports* dienen der Ansteuerung der Servos, sowie der Aufnahme der Rückmeldungen der Servos über deren Drehbewegungen. Die vier zur Verfügung stehenden *Timer* werden alle benötigt.

In dieser Motorsteuereinheit wird der Mikrocontroller mit einer Frequenz von 11,0592 MHz betrieben, was einer Befehlsausführung von etwa 1,8 MHz[7] entspricht.

[6] Keil Software, Inc., `http://www.keil.com`
[7] Sechs Clocks pro Maschinen-Operation, d.h. $11,0592\text{MHz}/6 = 1,8432\text{MHz}$

Flash

Der Mikrocontroller besitzt 64 KB Flash-Speicher in vier Blöcken. Die in dieser Studienarbeit programmierte Software nutzt die Eigenschaft, dass per *In-Application Programming* ein Block gelöscht werden kann und dann byteweise adressiert wieder beschrieben werden kann. Die Eigenschaften eines Flash-Speichers lassen es nur zu, einen gesamten Block zu löschen, nicht weniger. Ist nach dem Löschen der gesamte Block mit Einsen (HEX-Wert 0xF) überschrieben, kann die Applikation dann byteweise adressiert wahlweise einige Bits in Nullen umwandeln.

In-Application Programming

In der Entwicklungsumgebung μVision ist es möglich, mit Assembler-Code das Löschen eines Flash-Blockes sowie das oben beschriebene byteweise Schreiben durchzuführen. Da der Programmcode sich ebenfalls im Flash-Speicher befindet, ist bei der Programmierung darauf zu achten, dass nicht ein Block gelöscht wird, in welchem sich noch auszuführender Programmcode befindet. In-Application Programming wird in der Motorsteuerung verwendet, um Kalibrierungsdaten zu speichern.

Timer

Der 664-Mikrocontroller verfügt über insgesamt vier Timer, die durch die Motorsteuerung optimal ausgenutzt werden. Es handelt sich um die Timer mit den Bezeichnungen 0, 1, 2, PCA.

- Der Timer 1 wird von der seriellen Schnittstelle verwendet und ist somit nicht weiter verfügbar.
- Der Timer 0 wird für die Erzeugung der Servo-Steuerimpulse verwendet. Dafür wird der 16-Bit-Modus verwendet, um einen möglichst großen Zählbereich zu erreichen.

  Näheres zur Realisierung in Software siehe Abschnitt 3.4.
- Die Timer/Counter 2 und PCA werden für die Aufnahme der Pulse der beiden Fahr-Servos verwendet. Da beide Timer über einen externen Zähl-Eingang an den Ports des Controllers verfügen, ist es möglich, die Pulse auf diese Art zu zählen. Es handelt sich dabei um den Pin T2 (P1.0) für Timer 2 und den Pin ECI (P1.2) für Timer PCA, jeweils am Port 1.

  Näheres zur Software-Realisierung befindet sich im Abschnitt 3.6.

3 Software

In diesem Kapitel wird die Umsetzung der Aufgabenstellung der Software auf der im vorigen Kapitel beschriebenen Plattform beschrieben. Die Programme für den P89C664-Mikrocontroller werden mit der Entwicklungsumgebung μVision der Firma Keil entwickelt. Die dort verwendete Programmiersprache ist C[8], erweiterbar durch beispielsweise Assembler, welches mit Hilfe der `#pragma`-Richtlinie eingebunden wird.

Die Software ist in verschiedene Module aufgeteilt, die in Dateien gekapselt sind:

- `MOTOR_DEFS.H` enthält globale Einstellungen,
- `main` enthält das Hauptprogramm,
- `fifo` stellt eine Warteschlange zur Verwaltung der Motor-Befehle zu Verfügung,
- `servotiming` erzeugt die Impulse und stellt die Kalibrierung zu Verfügung,
- `tickcounttimingTimer` wertet die Pulse aus,
- `programDataByte` schreibt mit Hilfe von eingebundenem Assembler-Code ein Byte in den Flash-Speicher,
- `eraseBlock` löscht einen Block im Flash-Speicher,
- `i2c_slave` ist ein mir zur Verfügung gestelltes Modul von Markus Rilk zum Empfang der I^2C-Daten[9].

Die Module werden im Folgenden detailliert beschrieben.

3.1 Definitionen (MOTOR_DEFS.H)

Das Modul `MOTOR_DEFS.H` ist die zentrale Stelle, an der das Verhalten der Motorsteuerung geregelt ist und globale Datentypen und Variablen definiert sind.

Es folgt eine Beschreibung der wichtigen Einstellungs-Möglichkeiten.

Die Konstante `DBG` bestimmt, ob die vielfältigen Debug-Ausgaben an der seriellen Schnittstelle ausgegeben werden sollen (1) oder nicht (0). Bei Aktivieren dieser Funktion ist zu bedenken, dass die häufige Ausführung des `printf(...)`-Befehls zu Fehlern in der Ausführung führen kann! Die per Protokoll-Befehl

[8] Die verfügbaren Funktionen lassen sich in der C51 Library Reference [2] nachlesen, die als Hilfe-Datei in μVision verfügbar ist.

[9] Das Modul i2c_slave wird hier nicht näher beschrieben.

ENABLE_DEBUG zuschaltbaren Ausgaben bleiben von dieser Konstante unberührt.

Die Konstante `SLAVE_ADDR` bestimmt die I^2C-Bus-Adresse, auf die der Baustein reagieren soll. Standard ist 0x10.

Die Konstanten `HIGH` und `LOW` bestimmen, wie die Signale (Impulse), mit denen die Servos angesteuert werden, ausgerichtet sind. Bei HIGH=1, LOW=0 ist ein Impuls +5V, umgekehrt ist ein Impuls 0V. Die Einstellung ermöglicht das Einsetzen eines Inverters vor den Servos, um diese vom Mikrocontroller zu entkoppeln.

In den folgenden Zeilen des Moduls werden die Befehle des Protokolls (vgl. auch Abschnitt 3.7) definiert und Strukturen für die Verarbeitung der Befehle zur Verfügung gestellt.

3.2 Hauptprogramm (main)

Das Modul `main` ist das Hauptprogramm der Motorsteuerung. Zu Beginn werden in der Funktion `main()` sämtliche Module sowie die serielle Schnittstelle initialisiert. Anschließend wird in einer Endlos-Schleife auf ankommende I^2C-Datagramme gewartet. In der Funktion `processCommand()` werden diese Datagramme wie im Protokoll vorgegeben verarbeitet. Falls der Befehl eine Antwort erwartet, wird diese mit Hilfe einer der send...-Funktionen über den Bus gesendet.

Im laufenden Betrieb gibt die Motorsteuerung Ausgaben an der seriellen Schnittstelle (COM-Port) aus. Auf jeden Fall werden bei einem Reset einige Zeilen mit Identifikation und Version ausgegeben. Wenn die Debug-Option aktiviert ist (siehe dazu Befehl **ENABLE_DEBUG**), werden zusätzliche Ausgaben gemacht, die die Fehlersuche erleichtern.

Diese zusätzlichen Ausgaben betreffen über den I^2C-Bus kommende Befehle, gesendete Nachrichten, aufgetretene Fehler, die Kalibrierung, in die Warteschlange eingereihte sowie entfernte und ausgeführte Befehle. Der Aufruf der zuständigen Funktionen (`printf(...)` bzw. `puts(...)`) wird durch eine Mutex-ähnliche Variable[10] `g_as` (Bit-adressierbar, Bit 1 von `g_debug`) geschützt, damit keine gleichzeitigen Aufrufe erfolgen. Gleichzeitige Aufrufe dieser Funktionen (auch einer dieser Funktionen) müssen verhindert werden, da sie sonst den selben Speicherbereich gleichzeitig für verschiedene Ausgaben verwenden würden. Durch die Verwendung des Mutex ist es möglich, dass verschiedene schnell aufeinander folgende Debug-Ausgaben einfach nicht erfolgen. Die angezeigten I^2C-Bus-Daten werden als unsigned char (byte) ausgegeben und müssen unter Umständen umgerechnet werden, falls sie als Befehlsparameter interpretiert werden sollen. Trotz des Schutzes durch `g_as` können Seiteneffekte, die den korrekten Programmablauf verhindern, nicht ausgeschlossen werden. Dies ist zu beachten.

[10] Mutex: mutual exclusion - gegenseitiger Ausschluss.

3.3 Warteschlange (fifo)

Das Protokoll sieht für die Steuerung der zwei Fahr-Servos eine gemeinsame Ansteuerung vor, d.h. mit einem Befehl werden für beide Servos Steuerbefehle übermittelt. Dieses ist der Befehl **MOVE**. Er wird dargestellt in der Struktur `struct MOVE_COMMAND`. Da es dem Anwender möglich sein soll, mehrere **MOVE**-Befehle nacheinander abzusenden, werden diese in eine Warteschlange nach dem Prinzip First-in-First-out (FiFo)[11] organisiert. Diese Aufgabe übernimmt das Modul `fifo`.

Die Funktionen `pushFifo(...)` und `popFifo()` erfüllen die Aufgaben, ein Element hinzuzufügen bzw. herauszuholen.

Ein Element besteht, wie im Modul `MOTOR_DEFS` zu sehen, aus zwei Werten je Servo: Die *Geschwindigkeit*, mit der sich der Servo drehen soll und die *Ticks*, die er sich fortbewegen soll. Die Ticks sind die vom Servo gezählten Pulse (Entfernungsschritte). Durch Kombinationen der Werte lassen sich auch unendliche Bewegungen oder Bewegungen nur eines der Motoren erreichen (siehe Protokoll).

Die Größe des FiFo-Speichers lässt sich zu Beginn des Moduls in der Konstante `FIFO_SIZE` festlegen und ist standardmäßig auf 100 gesetzt.

Die Funktion `signed char pushFifo(struct MOVE_CMD *dat)` fügt ein Element der Warteschlange hinzu, falls dort freier Speicherplatz zur Verfügung steht. Ist die Warteschlange voll, wird der Befehl verworfen und eine „0“ zurückgegeben, woraufhin über den I^2C-Bus die Fehlernachricht **ERROR_FIFO_FULL** gesendet wird. Sobald der Befehl erfolgreich in die Warteschlange eingereiht wurde, wird die Funktion `startNewMove()` aufgerufen, welche überprüft, ob der Befehl sofort ausgeführt werden kann (siehe dazu auch Abschnitt 3.6). Rückgabewert von `pushFifo(...)` ist im Erfolgsfall „1“.

Die Funktion `struct MOVE_CMD *popFifo()` gibt einen Pointer auf das älteste Element der Warteschlange zurück. Falls der Speicher leer ist, erfolgt die Rückgabe von NULL.

3.4 Impulse generieren (servotiming)

Die wichtigste (weil zeitkritische) Funktion ist die Generierung der Impulse, die die in Abschnitt 2.1 beschriebenen Servos steuern. Diese Generierung wird im Modul `servotiming` vorgenommen. In dieser Funktion ist es sehr wichtig, die jeweilige Länge der Impulse konstant zu halten. Außerdem soll die Ausführungsdauer des Codes nicht die anderen Aufgaben beeinträchtigen. Aus diesem Grund fiel die Entscheidung auf die folgende Funktionsweise mit Hilfe von hoch prioren Interrupts:

Der Mikrocontroller verfügt (neben dem für die serielle Schnittstelle verwendeten) über drei verfügbare Timer, die sich anbieten, für die Generierung der Impulse zu dienen, da sie extrem genau sind und nicht viel Rechenzeit in Anspruch nehmen. Da aber die in Abschnitt 3.6 beschriebene Puls-Verarbeitung

[11] FiFo: Das erste ankommende Element wird auch als erstes wieder entfernt.

ebenfalls optimal mit je einem Timer funktioniert, verbleibt für die Generierung der drei Impulse nur ein Timer.

Um ein Zittern des Servos zu verhindern bzw. eine ungestörte Bewegung zu erreichen, muss jeweils die *Länge* jedes Impulses konstant bleiben, allerdings nicht deren Abstand zueinander. Dadurch ist es möglich, alle drei Impulse mit einem Timer zu erzeugen. Hierfür wird der Timer 0 verwendet.

Um die drei Impulse mit nur einem Timer generieren zu können, werden sie nacheinander erzeugt. Ihre Länge beträgt jeweils maximal 2,5 ms und zwischen den Impulsen für einen Servo sollen etwa 20 ms liegen, mehr ist nicht problematisch. Es bleibt genügend Zeit, die Impulse aneinander zu reihen. Es entsteht – wie in Abbildung 3.1 zu sehen – dreimal ein Impuls von 0,5 ms bis 2,5 ms an einem jeweils verschiedenen Port, gefolgt von einer konstanten Pause von 21,5 ms. Die Länge der Pause ist nicht von großer Bedeutung, darf also schwanken. In dieser Konfiguration schwankt die Pause zwischen 23 ms und 29 ms, je nach Impulslänge der drei Servos.

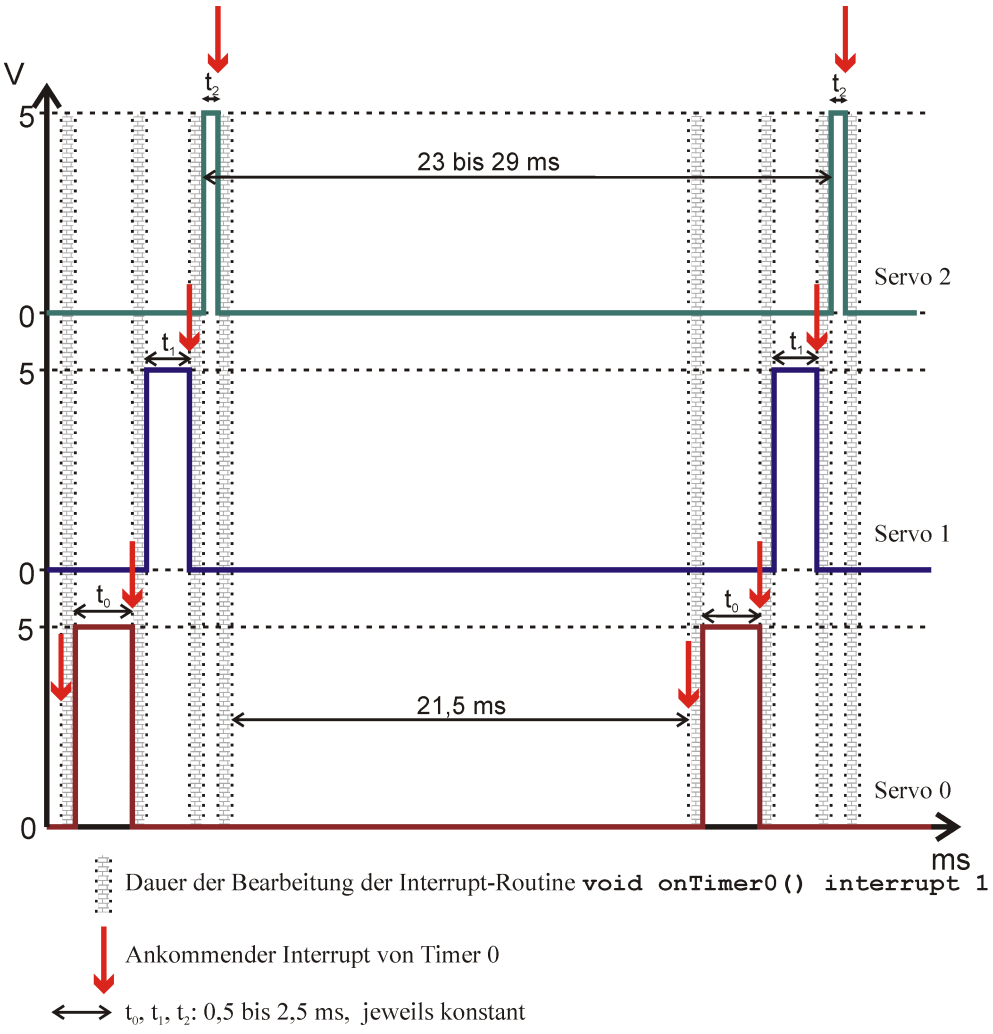

Abbildung 3.1: Drei mit einem Timer erzeugte Impulse.

Dies ist wie folgt in Software umgesetzt:
Der Timer 0 erzeugt beim Überlauf (dem Wechsel von 0xFFFF nach 0x0000) einen Interrupt. Sofort wird aufgrund dieses Interrupts die zugehörige Interrupt-Routine `void onTimer0()interrupt 1` ausgeführt. Diese Funktion stellt das Hauptelement der Impuls-Generierung dar. Hier wird die Länge des nächsten Intervalls berechnet und die Pins angesteuert. Da die drei Impulse für die drei Servos nacheinander erzeugt werden, muss die Interrupt-Routine reihum die entsprechenden Pins auf HIGH setzen.

Bei der Initialisierung von Timer 0 wird die Priorität des Interrupts auf die höchste Stufe (=3) gesetzt. Dadurch wird die Interrupt-Routine sofort ausgeführt, wenn der Timer überläuft – auch wenn der Mikrocontroller gerade eine andere Aufgabe ausführt. Sogar andere Interrupt-Bearbeitungen werden unterbrochen. Das sorgt für die geforderte höchste Genauigkeit und Zuverlässigkeit für die Impuls-Generierung.

Bevor es nun um die detaillierte Beschreibung der Interrupt-Routine geht, folgt eine Beschreibung, wie der Timer eingestellt wird, um eine genaue Zeitspanne zu erzeugen. Der Timer zählt, nachdem er aktiviert wurde, automatisch alle sechs Takte um eins hoch. Wie erwähnt ist der Übergang von 0xFFFF nach 0x0000 (Überlauf) der Zeitpunkt, mit dessen Hilfe man eine Aktion durchführen kann, weil der Interrupt die Interrupt-Routine aufruft. Um einen bestimmten Zeitraum zu erzeugen, kann man den Timer mit einem Wert vorladen und dann aktivieren. Je nach gewähltem Vorlade-Wert vergeht jetzt eine genau bestimmte Zeit bis zum nächsten Timer-Interrupt. Gleichzeitig mit der Aktivierung des Timers wird, falls es sich nicht um eine Pause handelt, das Signal für den entsprechenden Servo auf HIGH gesetzt. An dieser Stelle ist zu beachten, dass im Modul `MOTOR_DEFS.H` das Signal invertiert wird, ein HIGH also 0V entspricht. Sobald der Timer überläuft (Übergang von 0xFFFF=65535 zu 0x0000=0), wird der vorher aktivierte Impuls von der Interrupt-Routine beendet, d.h. das Signal auf LOW gesetzt. Der Vorlade-Wert errechnet sich aus

$$\text{Wert} = (65535 - \text{steps}),$$

mit

$$\text{steps} = \frac{\text{Impulsdauer}}{\left(\frac{1}{6} * \text{clock}\right)^{-1}},$$

$$\text{clock} = 11,0592\text{MHz},$$

$$\text{Impulsdauer} = \text{Dauer des Impulses}.$$

Der 16-Bit-Timer kann also in Abständen von 35,555 ms (entspricht Vorladen mit 0) bis 0 ms (entspricht Vorladen mit 65535) Interrupts erzeugen.

Bei der Initialisierung in der Funktion `initTimer0()` wird der Timer mit 0x0000 vorgeladen, das bedeutet eine Pause von 35,555 ms vor dem ersten Interrupt. In der Interrupt-Routine `onTimer0()interrupt 1` (siehe Listing A.1) werden die Impulse erzeugt. Die Variable `servo_count` zeigt an, welcher Servo zur Zeit angesteuert wird. Der Wert -1, welcher auch bei der Initialisierung verwendet wird, bedeutet kein aktivierter Servo, also die Pause zwischen den Impuls-Folgen. Sobald ein Interrupt auftritt, werden in jedem Fall sämtliche

Pins auf LOW gesetzt. Dadurch wird sichergestellt, dass das Auftreten des Interrupts sofort zum Ende des aktiven Impulses führt. Die dann folgenden Berechnungen sind also zeitunkritisch, da sie keinen Einfluss auf die Länge eines möglicherweise zu generierenden Impulses haben. In der Abbildung 3.1 sind sie gesondert markiert.

In den folgenden Zeilen der Funktion geschieht die Unterscheidung zwischen einer zu generierenden Pause und einem der drei Impulse. Im Falle der Pause wird `servo_count` auf -1 gesetzt (Überlaufschutz), kein Pin aktiviert, der Timer mit dem Wert für die Pause (`DEFAULT_PAUSE`) vorgeladen und anschließend gestartet. Im Falle eines zu generierenden Impulses wird der Timer mit dem Wert für den entsprechenden Impuls (`servo_position[servo]`) vorgeladen. Bei den Servos 0 und 1 wird der Impuls unterdrückt, falls der Befehl **`STOP`** aktiviert ist. Dadurch stellt der Servo jede Bewegung[12] ein. Das erlaubt den Servo mit der Hand zu bewegen, was beim Anliegen eines Impulses nicht möglich ist, da der Servo ständig nachstellt. Wichtig ist eine genaue Kalibrierung der Mittelposition, da der Servo sonst möglicherweise langsam weiter läuft. Um bei der Programmierung einer Anwendung sicher zu gehen, kann nach Stoppen des Servos eine Anwendung des Befehls **`STOP`** dafür sorgen, dass garantiert keine Bewegung mehr stattfindet. Die Impulse für Servo 2 werden wie bei den anderen Servos bei dem zugehörigen **`STOP_2`**-Befehl unterdrückt.

Anschließend wird in der `switch`-Anweisung der entsprechende Pin aktiviert, falls nicht die Bedingung für ein Unterdrücken erfüllt ist. Sofort im Anschluss wird der Timer aktiviert, wodurch die Zeit für einen genauen Impuls läuft.

Die kurze Funktion `setServoSpeed(unsigned char servo, signed char speed)` rechnet die vom Anwender verwendeten Geschwindigkeits-Werte (zwischen -128 und +127) in die intern verwendeten Werte zum Vorladen des Timers um. Dieser umgerechnete Wert wird in der Variable `servo_position[servo]` gespeichert. Dabei beachtet die Funktion die bei der Kalibrierung gespeicherten Maximal- und Mittel-Werte.

3.5 Flash (programDataByte und eraseBlock)

Zusammen mit der Impuls-Generierung befindet sich die Kalibrierung im Modul `servotiming`. Die dort eingestellten Kalibrierungs-Werte (vgl. Abschnitt 4.2) für die Servo-Motoren können in den Flash-Speicher des Mikrocontrollers geschrieben werden. Hier wird beschrieben, wie das Speichern und Löschen im Flash realisiert ist.

Im Flash kann nur blockweise gelöscht und byteweise geschrieben werden. Ein Byte kann aber im Allgemeinen nur einmal beschrieben werden und dann nicht wieder überschrieben werden. Eine Ausnahme stellt der Fall dar, in dem beim Überschreiben die Bits so geändert werden, dass nur zusätzliche Einsen in Nullen umgewandelt werden. Eine einmal geschriebene Null kann nur durch Löschen des gesamten Blockes wieder in eine Eins verwandelt werden. Da die

[12] Ein ausbleibendes Signal führt tatsächlich nicht *sofort* zum Stillstand, sondern erst etwas zeitverzögert, wenn nämlich die Servo-Elektronik feststellt, dass keine weiteren Impulse kommen. Ein sofortiger Stopp kann durch Senden der Geschwindigkeit 0 erwirkt werden.

Kalibrierungs-Daten von der Anwendung geändert bzw. gelöscht werden sollen, sind sie in einem eigenen Block gespeichert. Dieser Block darf keine Programm-Daten enthalten, da diese sonst bei einem neuen Schreiben der Kalibrierungs-Werte gelöscht werden würden. Der von mir gewählte Block ist Block 3, der sich im Adressraum von 0x8000 bis 0xBFFF erstreckt.

Die 16-Bit-Integer Kalibrierungs-Variablen sind in der folgenden Reihenfolge ab der Adresse 0x8000 gespeichert:
`CALIBRATION_0_UPPER`, `CALIBRATION_0_MIDDLE`, `CALIBRATION_0_LOWER`,
`CALIBRATION_1_UPPER`, `CALIBRATION_1_MIDDLE`, `CALIBRATION_1_LOWER`,
`CALIBRATION_2_UPPER`, `CALIBRATION_2_MIDDLE`, `CALIBRATION_2_LOWER`

Hier werden die zwei Module bzw. Funktionen beschrieben, die vom Modul `servotiming` aufgerufen werden, wenn der Speicher gelöscht und neu beschrieben werden soll.

Das Modul `eraseBlock` löscht in der Funktion `eraseBlock()` (siehe Listing A.3) den dort voreingestellten Block 3 im Flash-Speicher des Mikrocontrollers. Hier wird mit einem In-Application Programming (IAP) Call die Funktion ERASE BLOCK ausgeführt (vgl. [1], S. 69). Der im User Guide vorgegebene Assembler-Code wird mit Hilfe der `#pragma`-Richtlinie (vgl. [2], S. 163ff.) eingebunden. Der zu löschende Block ist fest eincompiliert.

Die Funktion `ProgramDataByte(unsigned int address, unsigned char mydata)` (siehe Listing A.2) im Modul `ProgramDataByte` schreibt das übergebene Byte `mydata` an die Adresse `address` des Flash-Speichers. Auch hier wird der Assembler-Code mit dem IAP Call der Funktion PROGRAM DATA BYTE mit der `#pragma-` Richtlinie eingebunden. Im Gegensatz zur Lösch-Funktion müssen hier Parameter übergeben werden. In der Parameter-Kombination Integer, Byte werden die Werte in den Registern R6, R7 und R5 übergeben. Der Integer steht in R6 (MSB[13]) und R7 (LSB[14]), das Byte in R5 (vgl. [2], S. 164).

Die Durchführung der Kalibrierung ist im Abschnitt 4.2 beschrieben.

3.6 Puls-Zählung (tickcounttiningTimer)

Im Modul `tickcounttimingTimer` werden die von den Fahr-Servos gelieferten Pulse ausgewertet. Dafür werden die zwei Timer/Counter 2 und PCA verwendet. Beide Timer funktionieren als Counter und verfügen über einen externen Eingang, an dessen Pin jeweils ein Fahr-Servo-Puls-Ausgang angeschlossen ist. Die Timer zählen bei jeder auftretenden negativen Puls-Flanke um eins hoch. Beim Überlauf erzeugen auch sie einen Interrupt, der jeweils die zugehörige Interrupt-Routine aufruft. Die Prioritäten dieser Interrupts müssen < 3 betragen, damit sie nicht die Generierung der Steuer-Impulse stören. Die Prioritäten werden deswegen in den Initialisierungs-Funktionen `initTickTimer2()` und `initTickTimerPCA()` auf „1“ gesetzt. Außerdem werden die Counter noch nicht gestartet, um nicht ungewollte Interrupts zu erhalten. Um den Eingang nutzbar zu machen, muss der Eingangs-Pin auf HIGH (1) gesetzt werden. Wird der Pin nun extern vom Servo mit Masse (0V) verbunden, lässt die Flanke den

[13] Most Significant Byte
[14] Least Significant Byte

Counter zählen. Counter 2 (für Servo 0) zählt an Pin T2 (P1.0), Counter PCA (für Servo 1) verwendet Pin ECI (P1.2).

Mit dieser Timer-Konfiguration ist es möglich, eine bestimmte Puls-Zahl festzulegen, nach der ein Interrupt auftreten soll. Die Pulse werden auf diese Weise sehr Ressourcen schonend gezählt.

Die beiden Interrupt-Routinen `onTimer2()interrupt 7` und `onTimerPCA()interrupt 6` funktionieren im Detail wie folgt: Zuerst setzen sie das Overflow-Flag zurück, um den Counter für einen neuen Durchgang vorzubereiten. Anschließend wird der Timer/Counter deaktiviert und mit „0“ vorbelegt. Mit Hilfe der Funktion `setSpeed(...)` wird der zugehörige Servo gestoppt. An dieser Stelle ist die Ausführung des vergangenen Befehls sauber beendet. Falls beide Servos ihre Befehle abgearbeitet haben, müssen die nachfolgenden bearbeitet werden. Deswegen wird am Ende der Interrupt-Routine die Funktion `startNewMove()` aufgerufen.

Die Funktion `startNewMove()` überprüft zunächst anhand der Counter-Register TR2 und CR, ob noch einer dieser Counter läuft, also noch nicht beide Servos ihren Befehl zu Ende ausgeführt haben. Befindet sich einer der Counter noch in Ausführung, beendet die Funktion `startNewMove()`. Somit ist die Interrupt-Bearbeitung beendet. Es wird auf den zweiten Servo gewartet, der bei Beendigung seines Befehls ebenfalls seinen Interrupt auslöst und wieder in der `startNewMove()`-Funktion ankommt. In dem Fall wird festgestellt, dass beide Counter ausgeschaltet sind, also beide Servo-Befehle fertig bearbeitet sind. Da jetzt der nächste neue Befehl aus der Warteschlange ausgeführt werden muss, wird dieser von der Funktion `popFifo()` angefordert. Befindet sich ein Befehl in der Warteschlange, werden die Counter erneut mit den Werten für den folgenden Befehl vorgeladen und die Servos mit der entsprechenden Geschwindigkeit gestartet. Befindet sich kein weiterer Befehl in der Warteschlange, bleiben die Servos auf Geschwindigkeit 0 und die Counter bleiben gestoppt. Die Funktion `startNewMove()` wird dann erst beim nächsten über den I^2C-Bus kommenden Befehl von der Funktion `pushFifo(struct MOVE_CMD *dat)` aufgerufen.

Durch Ausführen der Funktion `startNewMove()` nach jedem ausgeführten Befehl und bei jedem Eintreffen eines neuen Befehls ist sichergestellt, dass ein Befehl, der sich in der Warteschlange befindet, auch ausgeführt wird.

3.7 Protokoll

Im Folgenden wird das Protokoll spezifiziert, mit dem die Servomotoren über den I^2C-Bus gesteuert und die Statusinformationen abgerufen werden können.

Die I^2C-Adresse der Motorsteuerung (Slave) ist voreingestellt auf 0x10. Die verwendete I^2C-Bus-Datenrate ist 100 kbit/s.

3.7.1 Aufbau der Befehle

Die Befehle setzen sich wie folgt byteweise zusammen:

Adresse	Befehl (Param0)	Param1 ... Paramn

bzw.

Adresse	Nachricht (Param0)	Param1 ... Paramn

Adresse:	I^2C-Adresse der Motorsteuerung (Slave)
Befehl:	Befehls-ID ist hier dezimal angegeben Befehls-ID < 50 (siehe Tabelle 3.2): Master -> Slave (Steuerung -> Motorsteuerung)
Nachricht:	Nachrichten-ID ist hier dezimal angegeben Nachrichten-ID $\mathbf{> 50}$ (siehe Tabelle 3.2): Slave -> Master (Motorsteuerung -> Steuerung, d.h. Antwort, Statusmeldung ...)
Param:	Parameter siehe Befehl-Spezifikation

Tabelle 3.1: Protokoll - Byteweiser Aufbau eines Befehls bzw. einer Nachricht

Steuerung allgemein	
RESET	1
RESET_ALL	2
ENABLE_DEBUG	3
Kalibrieren	
CALIBRATE_SET_UPPER	11
CALIBRATE_SET_MIDDLE	12
CALIBRATE_SET_LOWER	13
CALIBRATE_ADD_UPPER	14
CALIBRATE_ADD_MIDDLE	15
CALIBRATE_ADD_LOWER	16
CALIBRATE_GET_VALUES	17
CALIBRATE_VALUES	**51**
CALIBRATE_ERASE	18
CALIBRATE_WRITE_FLASH	19
CALIBRATE_SET_SPEED	31
CALIBRATE_GET_SPEED_VALUE	30
CALIBRATE_SPEED_VALUE	**55**
Steuerbefehle für die Servos	
MOVE	21
GET_STATUS	22
STATUS	**52**
STOP	23
START	24
MOVE_2	25
GET_STATUS_2	26
STATUS_2	**53**
STOP_2	27
START_2	28
SET_SPEED	29
Fehler	
ERROR	**54**

Tabelle 3.2: Protokoll - I^2C-Befehle bzw. -Nachrichten mit ID

3.7.2 Befehl-Spezifikation

Aufbau Befehl-Spezifikation

Im Folgenden wird für jeden Befehl bzw. jede Nachricht der Tabelle 3.2 der Aufbau und die Befehl-Spezifikation erläutert.

ID	**NAME_DES_BEFEHLS_ODER_DER_NACHRICHT**			
Beschreibung was dieser Befehl bewirkt bzw. die Nachricht enthält				
PARAMx	Funktion	Datentyp	erlaubte Werte	Beschreibung

Der Datentyp „unsigned char“ entspricht dem Datentyp „byte“.
ID ist als Dezimal-Wert angegeben.

Befehle Steuerung allgemein

1	**RESET**
Setzt alle Werte in den Ursprungszustand. Das beinhaltet Löschen der Warteschlange, Zurücksetzen aller Geschwindigkeiten und Ticks, Stoppen der Timer. Die Kalibrierung wird *nicht* zurückgesetzt.	
kein Parameter	

2	**RESET_ALL**
Wie **RESET**, setzt zusätzlich die Kalibrierung zurück auf die Standart-Werte aus dem Flash.	
kein Parameter	

3	**ENABLE_DEBUG**			
Aktiviert oder deaktiviert den Debug-Modus. Aktionen der Motorsteuerung werden über die serielle Schnittstelle bestätigt, falls der Debug-Modus aktiviert ist. Bei deaktiviertem Debug-Modus gibt die serielle Schnittstelle nur eine Begrüßung aus.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	1|0	1: aktivert, 0: deaktiviert (default)

Befehle zum Kalibrieren

11 12 13	**CALIBRATE_SET_UPPER** **CALIBRATE_SET_MIDDLE** **CALIBRATE_SET_LOWER**			
Setzt den Wert für den oberen/mittleren/unteren Wert (maximale Vorwärts-Geschwindigkeit / keine Bewegung / maximale Rückwärts-Geschwindigkeit) für den angegebenen Motor. Dieser Befehl kann verwendet werden, um bereits bekannte Werte für die vorliegende Kombination Platine - Motoren in der Initialisierungsphase zu setzen. Zur Kalibrierung selbst können die Befehle **CALIBRATE_SET_UPPER**, **-MIDDLE**, **-LOWER** verwendet werden. **Achtung**: Der hier gesetzte Wert wirkt erst bei der nächsten Geschwindigkeitsänderung des Motors (z.B. nächster Befehl der Warteschlange oder neues **SET_SPEED**).				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, der kalibriert wird
PARAM2	HI-Byte	unsigned integer	[0, 65535]	Wert
PARAM3	LOW-Byte			
Hier kann das vorgegebene struct `CALIBRATE_SET_CMD` verwendet werden.				

14 15 16	**CALIBRATE_ADD_UPPER** **CALIBRATE_ADD_MIDDLE** **CALIBRATE_ADD_LOWER**			
Verändert den Wert für den oberen/mittleren/unteren Wert (maximale Vorwärts-Geschwindigkeit / keine Bewegung / maximale Rückwärts-Geschwindigkeit) um den angegeben Wert für den angegebenen Motor. Es sind Schritte in maximal -128- bzw. +127-Sprüngen möglich. Dieser Befehl kann zur Fein-Justierung verwendet werden. Achtung: Der hier gesetzte Wert wirkt erst bei der nächsten Geschwindigkeitsänderung des Motors (z.B. nächster Befehl der Warteschlange oder neues **SET_SPEED**).				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, der kalibriert wird
PARAM2		signed char	[-128, 127]	Wert, um den der Kalibrierungs-Wert geändert werden soll

17	**CALIBRATE_GET_VALUES**			
Fordert die aktuellen Kalibrierungswerte für den angegebenen Motor an. Die Antwort ist die Nachricht **CALIBRATE_VALUES**.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, dessen Werte angefordert werden

51	**CALIBRATE_VALUES**			
Rückantwort Slave -> Master (Motorsteuerung -> Steuerung) Liefert die von **CALIBRATE_GET_VALUES** angeforderten Kalibrierungswerte für den entsprechenden Motor zurück.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, dessen Werte folgen
PARAM2	HI-Byte	unsigned integer	[0, 65535]	UPPER-Wert
PARAM3	LOW-Byte			
PARAM4	HI-Byte	unsigned integer	[0, 65535]	MIDDLE-Wert
PARAM5	LOW-Byte			
PARAM6	HI-Byte	unsigned integer	[0, 65535]	LOW-Wert
PARAM7	LOW-Byte			
Hier kann das vorgegebene struct `CALIBRATE_VALUES_MSG` verwendet werden.				

18	**CALIBRATE_ERASE**
Löscht den Kalibrierungs-Variablen-Speicher des Flash, damit neue Werte geschrieben werden können. **Achtung:** Nach Ausführung dieses Befehls bitte per Hand einen Reset durchführen! **Achtung:** Nach Ausführung dieses Befehls müssen die Kalibrierungs-Variablen neu gesetzt (**CALIBRATE_SET_xxx**) und neu gespeichert (**CALIBRATE_WRITE_FLASH**) werden, sonst werden die Default-Werte verwendet.	
kein Parameter	

19	**CALIBRATE_WRITE_FLASH**
Schreibt die aktuell eingestellten Kalibrierungs-Werte in den Flash-Speicher, so dass sie nach einem Neustart nicht verloren gehen. Wenn alte Werte überschrieben werden sollen, muss vorher unbedingt ein **CALIBRATE_ERASE** durchgeführt werden, da sonst die neuen Werte nicht korrekt gespeichert werden (die 0-Bits der alten Werte würde erhalten bleiben).	
kein Parameter	

31	**CALIBRATE_SET_SPEED**			
Setzt die Geschwindigkeit für den angegebenen Motor direkt auf den angegebenen Wert. Dieser Befehl umgeht die sonst angewendeten [-128, 127], die im normalen Betrieb innerhalb der Motorsteuerung in den hier direkt übermittelten Wert umgerechnet werden. Dieser Befehl kann verwendet werden, um Kalibrierungswerte auszuprobieren. **Achtung:** Mit diesem Befehl können die Kalibrierungsgrenzen (UPPER und LOWER) übergangen werden.				
PARAM1		unsigned char	0|1|2	Motor, dessen Wert gesetzt wird
PARAM2	HI-Byte	unsigned integer	[0, 65535]	Wert
PARAM3	LOW-Byte			
Hier kann das vorgegebene struct `CALIBRATE_SET_CMD` verwendet werden.				

30	**CALIBRATE_GET_SPEED_VALUE**			
Fordert den Wert (vom Datentyp „unsigned integer“) der aktuellen Geschwindigkeit vom angegebenen Motor an. Dieser Wert kann für die Kalibrierung nützlich sein, wenn die Geschwindigkeit beispielsweise mit **SET_SPEED** gesetzt wurde (siehe auch **CALIBRATE_SET_SPEED**). Die Antwort ist die Nachricht **CALIBRATE_SPEED_VALUE**.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, dessen Wert angefordert wird

55	**CALIBRATE_SPEED_VALUE**			
Rückantwort Slave -> Master (Motorsteuerung -> Steuerung) Liefert den von **CALIBRATE_GET_SPEED_VALUE** angeforderten Geschwindigkeits-Wert für den entsprechenden Motor zurück.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, dessen Wert folgt
PARAM2	HI-Byte	unsigned integer	[0, 65535]	Geschwindigkeits-Wert
PARAM3	LOW-Byte			
Hier kann das vorgegebene struct `CALIBRATE_SET_CMD` verwendet werden.				

Steuerbefehle für die Motoren

21	**MOVE**			
Steuerbefehl für die Motoren 0 und 1. Diese Motoren können nur gleichzeitig angesprochen werden. Diese Befehle werden in die Warteschlange eingereiht. Die Geschwindigkeit ist ein Wert zwischen -128 und 127, der den kalibrierten Maximalgeschwindigkeiten entspricht. Die Mittelposition ist 0 (der Motor steht). Die Ticks sind die Anzahl der Bewegungsschritte, die sich der Motor fortbewegen soll.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		signed char	[-128, 127]	Geschwindigkeit Motor 0
PARAM2	HI-Byte	unsigned integer	[0, 65535]	Ticks Motor 0
PARAM3	LOW-Byte			
PARAM4		signed char	[-128, 127]	Geschwindigkeit Motor 1
PARAM5	HI-Byte	unsigned integer	[0, 65535]	Ticks Motor 1
PARAM6	LOW-Byte			
Hier kann das vorgegebene struct `MOVE_CMD` verwendet werden.				

22	**GET_STATUS**
Fordert von der Motorsteuerung den Befehl der Warteschlange, der sich aktuell in der Ausführung befindet, und die vergangenen Ticks an. Die Antwort ist die Nachricht **STATUS**. Die Ausführung der Warteschlange wird nicht beeinflusst.	
kein Parameter	

52	**STATUS**

Rückantwort Slave -> Master (Motorsteuerung -> Steuerung)
Liefert den aktuellen Befehl der Warteschlange, der sich in der Ausführung befindet. Die Parameter 1 bis 6 liefern genau den Befehl wie er mit dem Befehl **MOVE** in Auftrag gegeben wurde. Die folgenden Werte sind die aktuellen Zählerstände, wie viele Ticks schon vergangen sind, das heißt wie viele Entfernungs-Schritte der Motor seit Beginn des Befehls schon zurückgelegt *hat*. Die Ausführung der Warteschlange wird nicht beeinflusst.
Wenn die Motoren mit dem Befehl **STOP** gestoppt wurden, werden an dieser Stelle für Geschwindigkeiten und Ticks Nullen zurückgegeben. Die vergangenen Ticks bleiben durch das **STOP** unbeeinflusst.

Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		signed char	[-128, 127]	Geschwindigkeit Motor 0
PARAM2	HI-Byte	unsigned integer	[0, 65535]	Ticks Motor 0
PARAM3	LOW-Byte			
PARAM4		signed char	[-128, 127]	Geschwindigkeit Motor 1
PARAM5	HI-Byte	unsigned integer	[0, 65535]	Ticks Motor 1
PARAM6	LOW-Byte			
PARAM7	HI-Byte	unsigned integer	[0, 65535]	vergangene Ticks Motor 0
PARAM8	LOW-Byte			
PARAM9	HI-Byte	unsigned integer	[0, 65535]	vergangene Ticks Motor 1
PARAM10	LOW-Byte			

Die Parameter 1 bis 6 sind identisch mit denen des **MOVE**-Befehls.
Hier kann das vorgegebene struct `STATUS_MSG` verwendet werden.

23	**STOP**

Pausiert die Ausführung der Befehle in der Warteschlange für die Motoren 0 und 1 und verwirft den aktuell ausgeführten Befehl. Das Signal an den Motorausgängen 0 und 1 wird unterbrochen. Der in diesem Zustand zurückgegebene Status liefert für die Geschwindigkeiten und Ticks Nullen, die vergangenen Ticks bleiben unbeeinflusst. Motor 2 bleibt unbeeinflusst.

kein Parameter

24	**START**

Führt die Ausführung der Warteschlange bei dem Befehl fort, der dem mit **STOP** unterbrochenen folgt. Motor 2 bleibt unbeeinflusst.

kein Parameter

25	**MOVE_2**			
Steuerbefehl für den Motor 2. Dieser Befehl wird sofort ausgeführt, also nicht in der Warteschlange der Motoren 0 und 1 geführt. Der aktuell vom Motor 2 ausgeführte Befehl wird unterbrochen. Der aktuell ausgeführte Befehl kann durch **GET_STATUS_2** abgefragt werden. Für diesen Motor können keine Entfernungs-Schritte (Ticks) angegeben werden! Die Position (eines nicht modifizierten Servos) kann kann nur über die MOVE_2-Befehle gesetzt werden.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		signed char	[-128, 127]	Position Servo 2

26	**GET_STATUS_2**
Fordert von der Motorsteuerung den aktuellen von Motor 2 ausgeführten Befehl an. Die Antwort ist die Nachricht **STATUS_2**. Die Ausführung der Warteschlange (der Motoren 0 und 1) und des Befehls für Motor 2 werden nicht beeinflusst.	
kein Parameter	

53	**STATUS_2**			
Rückantwort Slave -> Master (Motorsteuerung -> Steuerung) Liefert den von **GET_STATUS_2** angeforderten aktuellen von Motor 2 ausgeführten Befehl. Die Ausführung der Warteschlange (der Motoren 0 und 1) und des Befehls für Motor 2 werden nicht beeinflusst. Falls sich der Motor im STOP-Zustand befindet, wird Geschwindigkeit Null zurückgegeben.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		signed char	[-128, 127]	Geschwindigkeit Motor 2
Die Parameter sind identisch mit denen des **MOVE_2**-Befehls.				

27	**STOP_2**
Pausiert die Ausführung des Befehls für Motor 2. Das Signal am Motorausgang 2 wird unterbrochen. Die Motoren 0 und 1 bleiben unbeeinflusst. Der in diesem Zustand zurückgegebene Status liefert die Geschwindigkeit Null.	
kein Parameter	

28	**START_2**
Führt die Ausführung des aktuellen Befehls von Motor 2 fort. Die Motoren 0 und 1 bleiben unbeeinflusst.	
kein Parameter	

29	**SET_SPEED**			
Aktiviert sofort den entsprechenden Motor mit der angegebenen Geschwindigkeit und unendlicher Bewegung (also ohne Zählung von Entfernungs-Schritten). **Achtung:** Dieser Befehl sollte *nicht* gleichzeitig mit der Warteschlange verwendet werden (die mit dem Befehl **MOVE** gefüllt wird), da er die Befehle dort überholt und deren Ausführung verhindert. Der Befehl **SET_SPEED 2** entspricht dem Befehl **MOVE_2**.				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	0|1|2	Motor, der gestartet wird
PARAM2		signed char	[-128, 127]	Geschwindigkeit

Der leere Befehl für keine Aktion (siehe Befehle **MOVE**, **STATUS**, **MOVE_2**, **STATUS_2**) ist durch die Parameter Geschwindigkeit=0 und Ticks=0 möglich. Eine unendliche Bewegung der Motoren 0 oder bzw. und 1 wird durch Setzen der Ticks = 0 erreicht. Diese Bewegung wird unterbrochen sobald der andere Motor seine Ticks abgearbeitet hat. Wenn beide Motoren unendliche Befehle haben (Ticks=0), wird die Ausführung abgebrochen sobald ein nachfolgender Befehl in die Warteschlange eingereiht wird.

54	**ERROR**			
Zeigt einen Fehler in der Motorsteuerung an. Die Behandlung der Fehler ist im Benutzerhandbuch beschrieben. Fehler-Code 61 = `ERROR_WRITE_FLASH`, Fehler-Code 62 = `ERROR_FLASH_IS_NOT_ERASED`, Fehler-Code 63 = `ERROR_FIFO_FULL`				
Byte Nr.	Funktion	Datentyp	Werte	Beschreibung
PARAM1		unsigned char	61|62|63	Fehler-Code
Hier kann das vorgegebene struct `ERROR_MSG` verwendet werden.				

4 Anwendung

4.1 Motortest-Programm

Um die entwickelte Motorsteuerung testen zu können, wurde das zur Verfügung gestellte Motortest-Programm modifiziert und erweitert. Das Programm liegt im Quellcode für μVision vor und kann auf einem P87C552 ausgeführt werden, wie er sich auf der Praktikums-Platine befindet.

Anschluss

Die Praktikums-Platine wird wie zur normalen Steuerung der Motorsteuereinheit über den I^2C-Bus an diese angeschlossen (siehe Abbildung 2.6). Das Motortest-Programm wird über die serielle Schnittstelle geladen und gesteuert. Im Target Setup muss eventuell der anzusprechende Port umgestellt werden, da in diesem Projekt COM2 verwendet wurde.

Starten

Um das Programm auszuführen, wird das Target „Monitor“ gewählt und dann der Debug-Modus gestartet. Dieser lädt das Programm automatisch in den Controller und führt den Befehl `g 0x2000` aus. Dieser Sprungbefehl startet das Testprogramm. Steuern lässt sich das Programm über Eingaben an der seriellen Schnittstelle. Diese werden umgesetzt in I^2C-Bus-Befehle entsprechend dem Motorsteuerungs-Protokoll für die Motorsteuerung.

Bedienen

Das Programm sendet die im Protokoll spezifizierten I^2C-Bus-Befehle, mit deren Hilfe die Motorsteuerung getestet werden kann. Eine Übersicht der Befehle, die das Programm senden kann, wird nach Eingabe von „h“ oder „hilfe“ angezeigt.

Reaktionen der Motorsteuerung können mit Eingabe von „receive <bytes>“ angezeigt werden. Dieser Befehl zeigt <bytes> Bytes des I^2C-Bus-Empfangs-Puffers an. Die als unsigned char (0 bis 255) dargestellten Bytes müssen falls nötig selbst in andere Datentypen (z.B. Integer) umgerechnet werden. Falls das Testprogramm eine empfangene Nachricht identifiziert, wird diese zusätzlich fertig umgerechnet angezeigt.

Das Haupt-Einsatzgebiet dieses Testprogramms ist die Kalibrierung. Sämtliche Kalibrierungs-Befehle stehen zur Verfügung. Die Status-Meldungen können (falls benötigt) direkt im Debug-Modus an der seriellen Schnittstelle der Motorsteuereinheit oder im Motortest-Programm nach Eingabe von „receive“ abgelesen werden. Für eine komfortablere Kalibrierung ist im folgenden Abschnitt ein Ansatz beschrieben.

4.2 Kalibrierung

Jeder Servo verhält sich geringfügig (bei verschiedenen Modellen und Herstellern auch stark) anders. Deswegen ist es wichtig, dass sich die Ansteuerung der verschiedenen Servos anpassen lässt, damit die Maximalgeschwindigkeiten und Mittelpositionen der beiden Motoren rechts und links identisch sind. Außerdem wird die Mittelstellung durch den Widerstandswert des arretierten Potentiometers bestimmt, welcher bei jedem Exemplar anders ist.

Die Motorsteuerung sieht zu diesem Zweck die Kalibrierung vor.

Die Kalibrierung ist der einzige Punkt, bei dem sich die interne Darstellung der Impulse dem Anwender zeigt. Wie in Abschnitt 3.4 beschrieben, wird für jeden Servo die Impulslänge durch eine Zahl zwischen 0 und 65535 dargestellt. Zu beachten ist an dieser Stelle, dass eine höhere Zahl einen kürzeren Impuls bedeutet, da der Timer bei höherem Vorladewert weniger lange braucht, um überzulaufen.

Es können für jeden Servo separat drei verschiedene Werte festgelegt werden:

- UPPER: Wert für die Maximalgeschwindigkeit in Vorwärts- bzw. Rückwärtsrichtung, d.h. der längste bzw. kürzeste Impuls (je nach Einbaurichtung des Servos)

- MIDDLE: Wert für die Mittelstellung, in der sich der Servo nicht bewegt

- LOWER: Wert für die Maximalgeschwindigkeit in Rückwärts- bzw. Vorwärtsrichtung, d.h. der kürzeste bzw. längste Impuls (je nach Einbaurichtung des Servos)

Wenn bisher keine Werte in den Flash-Speicher geschrieben wurden, bzw. der Speicher gelöscht wurde, sind die oben genannten Werte nach einem Hardware-Reset oder dem Software-Befehl **RESET_ALL** für jeden Servo auf Default-Werte gesetzt. Diese Werte erzeugen Impulse von der Länge 2,5 ms (UPPER (LOWER) =60927), 1,5 ms (MIDDLE= 62770) und 0,5 ms (LOWER (UPPER)= 64613). Für Servo 1 sind UPPER und LOWER vertauscht.

Die Default-Werte sehen für die Servos 0 und 2 die gleiche Drehrichtung vor. Bestimmt wird sie durch die Werte UPPER > MIDDLE > LOWER. Durch die entgegengesetzte Einstellung LOWER > MIDDLE > UPPER für Servo 1 ist seine Drehrichtung umgekehrt. Da die Servos spiegelverkehrt am Fahrmodul angebracht sind, ist diese Umkehrung notwendig, damit vorwärts jeweils Geschwindigkeiten größer Null und rückwärts Geschwindigkeiten kleiner Null sind. Bei der eigenen Kalibrierung kann die Drehrichtung durch Änderung der Verhältnisse LOWER >< UPPER eingestellt werden.

Die Kalibrierung läuft nun wie folgt ab:

Falls bereits Werte aus früheren Versuchen für die Servos bekannt sind, können sie mit den Befehlen

- **CALIBRATE_SET_UPPER**
- **CALIBRATE_SET_MIDDLE**
- **CALIBRATE_SET_LOWER**

für jeden Servo separat in den Speicher geschrieben werden. Bis zum nächsten Reset bleiben sie erhalten.

Die Ermittlung der richtigen Kalibrierungswerte für UPPER, MIDDLE und LOWER für jeden Servo kann auf verschiedene Weise vorgenommen werden.

Zuerst folgen Möglichkeiten, wie ein gewünschter Wert zur Probe übermittelt und zur Ausführung gebracht werden kann.

- **CALIBRATE_SET_SPEED**:
  Die direkte Art und Weise, einen Servo in der gewünschten Geschwindigkeit (Achtung: 16-Bit-Geschwindigkeit!) drehen zu lassen, ist der Befehl **CALIBRATE_SET_SPEED**.
- **SET_SPEED**:
  Ist eine grobe Kalibrierung bereits vorgenommen, kann auch der Befehl **SET_SPEED** verwendet werden, um eine Geschwindigkeit zu setzen. Hier ist zu beachten, dass keine Werte jenseits der aktuellen Werten für UPPER und LOWER möglich sind. Außerdem ist die Genauigkeit unter Umständen sehr viel schlechter als bei Verwendung des oben genannten Befehls. Sind die Werte UPPER und LOWER schon in den RAM geschrieben, bietet sich diese Methode zum Finden der Mittelposition an. Ist die gewünschte Geschwindigkeit eingestellt, kann der Kalibrierungswert anschließend mit dem Befehl **CALIBRATE_GET_SPEED_VALUE** angefordert werden.
- **CALIBRATE_ADD_xxx**:
  Falls eine Anwendung es erfordert, können die Befehle **CALIBRATE_ADD_UPPER**, **CALIBRATE_ADD_MIDDLE** und **CALIBRATE_ADD_LOWER** mit einem anschließenden **SET_SPEED** verwendet werden. (Erst **SET_SPEED** macht die Änderungen wirksam.)

Welche Werte nach obigen Methoden ausprobiert werden sollten, bleibt dem Anwender, der die Kalibrierung durchführt, überlassen.

Eine schnelle aber auch sehr ungenaue Methode ist beispielsweise die Ermittlung der Maximalwerte nach Gehör. In dem Fall wird der Wert für UPPER oder LOWER erhöht oder verringert bis keine Geräusch-Änderung mehr stattfindet.

Eine sehr viel genauere Methode ist das Zählen der zurückgelegten Entfernungsschritte (Pulse). Zu diesem Zweck sind die Befehle **MOVE** und **GET_STATUS** nützlich. Mit **MOVE** kann eine unendliche Bewegung einer bestimmten Geschwindigkeit initiiert werden, woraufhin nach einer festen Zeit mit Hilfe des

Befehls **GET_STATUS** die zurückgelegten Entfernungsschritte abgefragt werden. Ein Vergleich zwischen den Servos liefert Informationen, welcher Servo schneller oder langsamer ist und kalibriert werden muss. Eine detailliertere Programmidee ist in Abschnitt 4.2.1 dargestellt.

Weitere praktische Informationen zur Anwendung, speziell zum anschließenden Speichern der Werte im Flash, befinden sich im Benutzerhandbuch[3].

4.2.1 Anwendungsvorschlag Kalibrierung

In diesem Abschnitt wird ein Vorschlag beschrieben, wie ein Programm bei der Kalibrierung zweier Motoren vorgehen könnte. Besonderer Wert wird auf genaue Abstimmung der maximalen Geschwindigkeiten der zwei Fahr-Servos gelegt, damit das Fahrzeug nach der Kalibrierung bei gleichem angesteuerten Geschwindigkeitswert der zwei Motoren auch geradeaus fährt.

Das Prinzip hinter dieser Idee ist die Verwendung der Puls-Zählung in Verbindung mit einer genau festgelegten Messzeit dieser Entfernungsschritte. Der Kern ist ein Algorithmus, der Befehle in bestimmter Reihenfolge und exaktem Timing über den I^2C-Bus sendet. Dabei müssen folgende Schritte eingehalten werden:

- **Kalibrierungsdaten setzen**: Die zu testenden Kalibrierungswerte für die Servos sind zu setzen. Dafür sind die Befehle **CALIBRATE_ADD_xxx** oder **CALIBRATE_SET_xxx** geeignet. Hier ist darauf zu achten, dass sich die Servos bei spiegelverkehrter Montage in die gleiche Richtung bewegen (siehe Abschnitt 4.2).

- **Zeit nehmen**: Zunächst muss die Zeit genommen werden, da die tatsächliche Geschwindigkeit der Servos von der Software nur über die Kombination aus zurückgelegter Strecke (Entfernungsschritte bzw. Pulse) und der Zeit ermittelt werden kann.

- **Servos starten**: Mit dem Befehl **MOVE** werden anschließend beide Motoren mit einer unendlichen (oder hinreichend weiten) Bewegung (welche keinesfalls vor Ablauf der Zeit erreicht sein darf) und maximaler Geschwindigkeit gestartet. Je nach zu testender Richtung ist das +127 für vorwärts oder -128 für rückwärts.

- **Zeit abgelaufen**: Sobald eine bestimmte Zeit abgelaufen ist (z.B. 2 Sekunden), wird mit dem Befehl **GET_STATUS** der Zählerstand der beiden Servos abgefragt. Die Servos laufen während dieser Abfrage noch. Anschließend können sie gestoppt werden.

- **Zählerstände auswerten**: Diese beiden Zählerstände sind auszuwerten. Ziel ist es, bei gleicher vorgegebener Geschwindigkeit an dieser Stelle gleiche Werte für beide Servos zu erhalten. Dann bewegen sie sich gleich schnell. Weiteres Ziel ist es, hier möglichst hohe Werte für die fortgelegten Entfernungsschritte zu erhalten, weil das einer hohen Geschwindigkeit entspricht. Für die Kalibrierung der Mittelstellung muss der Wert Null sein.

Mit dieser Methode können beide Motoren gleichzeitig gestartet und ausgewertet werden. Das hat den Vorteil, dass keine unterschiedlichen Be-arbeitungszeiten der Befehle die Messung beeinträchtigen.

Ein Algorithmus um diesen gerade beschriebenen Kernalgorithmus herum muss die Kalibrierungswerte sinnvoll wählen und die Auswertung der erhaltenen Zählerstände vornehmen.

Beispielsweise kann sich der Algorithmus von zwei Seiten mit immer kleineren Schritten an den Maximalwert herantasten. Die Vorgehensweise kann wie folgt aussehen:

- Mittelposition (Stand) kalibrieren:
  - ◇ Servo 0:
    - ○ Den Wert schrittweise (z.B. 1000er-Schritte) von LOWER an UPPER annähern (oder umgekehrt) und die Zählerstände vergleichen. Sobald sich der bisherige Trend umkehrt (z.B. Wert wird zuerst kleiner, dann wieder größer), die Schrittweite verkleinern und zwischen den letzten beiden 1000er-Schritten diesen Punkt mit kleineren Schritten wiederholen.
    - ○ *Alternativ* den Mittelwert zwischen LOWER und UPPER berechnen (oder Default-Wert 62770) und als Ausgangspunkt verwenden.
    - ○ Eventuell ist es sinnvoll zu beachten, dass je nach Annäherung aus Richtung UPPER oder LOWER verschiedene (aber nahe beieinander liegende) Werte zum Stehen des Servos führen. In dem Fall kann der Mittelwert zwischen den beiden verwendet werden, um sicher gegenüber kleinen Veränderungen der Servos zu sein (z.B. Temperaturschwankungen, die zu einer Verschiebung der Mittelposition führen).
    - ○ Das Ergebnis ist der Wert für MIDDLE von Servo 0.
  - ◇ Servo 1:
    - ○ Die Mittelstellung wie bei Servo 0 ermitteln.
    - ○ Das Ergebnis ist der Wert für MIDDLE von Servo 1.
- Vorwärts-Kalibrierung:
  - ◇ Servo 0:
    - ○ Mit Default-Wert 60927[15] beginnend in 1000er-Schritten größer werden .
    - ○ Sobald der Servo zwischen zwei Messungen langsamer wird in 100er-Schritten vom kleineren Wert ausgehend erhöhen.
    - ○ Sobald der Servo zwischen zwei Messungen langsamer wird, den letzten Punkt mit immer kleineren Schritten wiederholen.

[15] Default-Wert 60927: falls nötig kleineren Wert verwenden, d.h. falls der Servo die Maximalgeschwindigkeit bei diesem Standart-Wert noch nicht erreicht hat.

        - Das Ergebnis ist die Vorwärts-Maximalgeschwindigkeit (UPPER bzw. LOWER je nach Drehrichtung) für Servo 0. Zusätzlich den Zählerstand zwischenspeichern.
    - Servo 1:
        - Entsprechend Servo 0 die Vorwärts-Maximalgeschwindigkeit ermitteln. Falls der Servo spiegelverkehrt angebracht ist, also andersrum drehen soll, wie bei der Rückwärts-Maximalgeschwindigkeit von Servo 0 vorgehen.
    - Gemeinsame Vorwärts-Maximalgeschwindigkeit:
        - Die Zählerstände bei Maximalgeschwindigkeit der Servos 0 und 1 vergleichen. Die Maximalgeschwindigkeit des Servo mit dem höheren (niedrigeren) Wert muss verringert (vergrößert) werden (je nach Drehrichtung), damit sich beide Servos in ihrer Maximalstellung gleich schnell bewegen.
        - Zu diesem Zweck den Geschwindigkeitswert des schnelleren Servos Richtung Mittelposition (MIDDLE) schrittweise verändern, bis der Zählerstand gleich dem des langsameren Servos ist. Hierbei kann sich wie bereits beschrieben in veränderbaren Schrittweiten an den gewünschten Wert herangetastet werden.

- Rückwärts-Kalibrierung:
    - Servo 0:
        - Die drei ersten Punkte der Vorwärts-Kalibrierung mit dem Default-Wert 64613^{16} wiederholen. Dabei den Wert jedoch jeweils verkleinern.
        - Das Ergebnis ist die Rückwärts-Maximalgeschwindigkeit (LOWER bzw. UPPER je nach Drehrichtung) für Servo 0.
    - Servo 1:
        - Entsprechend Servo 0 die Rückwärts-Maximalgeschwindigkeit ermitteln. Falls der Servo spiegelverkehrt angebracht ist, also andersrum drehen soll, wie bei der Vorwärts-Maximalgeschwindigkeit von Servo 0 vorgehen.
    - Gemeinsame Rückwärts-Maximalgeschwindigkeit:
        - Identisch zur Vorgehensweise bei der Ermittlung der gemeinsamen Vorwärts-Maximalgeschwindigkeit.

Nach dem Durchlaufen dieses Algorithmus sind die Kalibrierungswerte der Servos 0 und 1 bekannt. Servo 2 besitzt keine Puls-Zählung und kann nicht automatisch kalibriert werden, da die Software keinerlei Information über die aktuelle Position des Servos erhalten kann.

[16] Default-Wert 64613: falls nötig größeren Wert verwenden

5 Zusammenfassung und Ausblick

Die Studienarbeit behandelt die Entwicklung einer Motorsteuereinheit für das Fahrmodul vom Informations-Systemtechnik-Praktikum. Bei den angesteuerten Motoren handelt es sich um drei Modellbau-Servos. Ein handelsüblicher Servo ist zur Bewegung eines Sensors oder ähnlichem vorgesehen. Zwei modifizierte Servos stellen den Antrieb des Fahrmoduls dar. Sie können sich fortlaufend rundum drehen und sind durch eine Einheit (Platine mit Messaufnehmer) erweitert, die Pulse liefert, anhand derer die Motorsteuerung eine Drehwinkelbestimmung des Servos machen kann. Diese Puls-Verarbeitung ist mittels zweier Counter realisiert, die zum Zählen der Impulse keine Rechenzeit des P89C664-Mikrocontrollers beanspruchen.

Die Generierung der Steuerimpulse für die Servos ist eine wesentliche Aufgabe der Software. Sie wird Ressourcen sparend mit Hilfe eines einzelnen Timers erfüllt. Die im Abstand von etwa 20 ms zu erzeugenden 0,5 ms bis 2,5 ms langen Impulse für jeden Servo werden nacheinander mit dem gleichen Timer berechnet. So verbleibt genügend Rechenzeit für weitere Aufgaben wie die Puls-Verarbeitung und die Kommunikation.

Die Motorsteuerung verfügt über zwei Schnittstellen zur Kommunikation mit anderen Elementen. Die serielle RS-232-Schnittstelle wird verwendet, um Debug-Ausgaben an einem Terminal sichtbar zu machen und um das mit μVision erstellte Programm hochzuladen. Die I^2C-Bus-Schnittstelle ist vorgesehen, um Befehle von der Hauptplatine des Fahrmoduls zu empfangen. Diese Befehle sind in einem Protokoll spezifiziert, so dass sie von den Teilnehmern des Praktikums mit ihrem Programm erzeugt werden können. Die Befehle sehen eine unabhängige Steuerung der drei Servos vor. Es ist zudem ständig möglich, den Status der Servos und der Puls-Zählung abzufragen.

Ein wesentliches Merkmal der Software der Motorsteuerung ist die Möglichkeit, die angeschlossenen Servos zu kalibrieren. Da sich jeder Servo bezüglich seiner Maximalausschläge bzw. erreichten Geschwindigkeiten und seiner Mittelposition unterschiedlich verhält, können diese drei Werte für jeden der drei angeschlossenen Servos separat festgelegt werden. Erst ein korrekt kalibriertes Fahrmodul verhält sich berechenbar. Andernfalls ist beispielsweise ein geordnetes Geradeausfahren nicht möglich.

Ein mögliches Verfahren zur automatischen Kalibrierung dient als Idee für eine mögliche Anwendung und weitere Beschäftigung mit dieser Motorsteuereinheit. Ein bisher nicht verwendeter 10-poliger Steckverbinder auf dem entwickelten Board erlaubt die Verwendung von vier dort angeschlossenen Pins des I/O-Ports 2 vom Mikrocontroller sowie die zusätzliche Verdrahtung von vier weiteren Signalen. Für ein so erweitertes Board ist die Programmierung von zusätzlichen Modulen möglich, mit denen die Motorsteuerung weitere Aufgaben übernehmen kann. Durch die sparsame Ressourcen-Nutzung der Impuls-Generierung und Puls-Zählung ist weitere Rechenzeit verfügbar.

Literaturverzeichnis

[1] Philips Semiconductors: Data Sheet P89C660/P89C662/P89C664/P89C668. Oktober 2002, U.S.A.

[2] Keil Software: Cx51 Compiler User's Guide. September 2001, Hilfe-Datei der Software μVision.

[3] Christian Schröder: Benutzerhandbuch Motorsteuereinheit. Institut für Betriebssysteme und Rechnerverbund, Technische Universität Braunschweig, 2005.
`http://www.ibr.cs.tu-bs.de/theses/broeke/SA_Motorsteuereinheit_Prakt/Dokumente.html`

[4] Brian W. Kerninghan, Dennis M. Ritchie: Programmieren in C. Zweite Auflange, ANSI C. Carl Hauser Verlag München Wien, 1990.

A Listings

Listing A.1: servotiming.c: onTimer0() - Interrupt-Routine für Impuls-Generierung

```
/**
 * Interrupt-Routine für Interrupt Timer 0
 */
void onTimer0() interrupt 1
{
    OUT_PORT24 = OUT_PORT25 = OUT_PORT26 = LOW; // Servo-Ausgänge deaktivieren
    servo_count++;
                                    // Timer 0 stoppen
    TR0 = 0;
                                    // Ein Durchgang Servo-Ansteuerung ist zu Ende
    if (servo_count + 1 > SERVO_MAX) {
                                    // aktiviere beim nächsten Interrupt Servo 0
        servo_count = -1;
                                    // Reset Timer 0 (vorladen mit DEFAULT_PAUSE);
        TH0 = (DEFAULT_PAUSE >> 8) & 0x00FF;
        TL0 =  DEFAULT_PAUSE       & 0x00FF;
                                    // Start Timer 0 (starten für Pause)
        TR0 = 1;
    }
                                    // wenn ein Impuls erzeugt werden soll
                                    //  (die -1 kann nie vorkommen, da immer ++ am Anfang des Interrupts ausgeführt wird)
    else {
                                    // Timer 0 vorladen mit Wert für Servo servo_count
                                    //  hier wird die Impuls-Länge bestimmt!!
        TH0 = (servo_position[servo_count] >> 8) & 0x00FF;
        TL0 =  servo_position[servo_count]       & 0x00FF;
                                    // Servo servo_count-Ausgang aktivieren, Timer 0 starten
        // Den entsprechenden Port ansteuern und Timer starten
        //  (aus Gründen der Zeitgenauigkeit sofort nach jedem Aktivieren statt nach der switch-Anweisung
        switch (servo_count) {
            case 0: // Servo 0
                    // Signal bleibt weg bei STOP
                    if (!stop01)
                        OUT_PORT24 = HIGH;
                    TR0 = 1;        // Start Timer 0 (anwerfen für Servo servo_count)
                    break;
            case 1: // Servo 1
                    // Signal bleibt weg bei STOP
```

```
39                      if (!stop01)
40                          OUT_PORT25 = HIGH;
41                      TR0 = 1;        // Start Timer 0 (anwerfen für Servo servo_count)
42                      break;
43              case 2: // Servo 2
44                      if (!stop2)      // Signal bleibt weg bei STOP
45                          OUT_PORT26 = HIGH;
46                      TR0 = 1;        // Start Timer 0 (anwerfen für Servo servo_count)
47                      break;
48              default:                // default verhindert, dass der Timer bei nicht-Zutreffen
49                                      //  von 0, 1, 2 nie wieder einen Interrupt wirft
50                      if (g_debug_bit0 && g_as == 0) {g_as = 1;puts(" default!!");g_as = 0;}
51                      TR0 = 1;        // Start Timer 0 (anwerfen für nächsten Interrupt)
52                      break;
53          }
54      }
55  }
```

Listing A.2: programDataByte.c: programDataByte(unsigned int address, unsigned char mydata) - Schreibt ein Byte in den Flash

```
/**
 * Schreibt ein byte mydata in den Flash an die Adresse address
 *
 * unsigned int address : Adresse, an die geschrieben werden soll
 * unsigned char mydata : Daten, die geschrieben werden sollen
 *
 * return 0x00  : erfolgreich
 *        sonst : nicht erfolgreich
 */
unsigned char ProgramDataByte (unsigned int address, unsigned char mydata)
{

    #pragma asm
        MOV   AUXR1,#20H      ; set the ENBOOT bit

        MOV   DPH,R6          ; MOV   DTPR,Address
        MOV   DPL,R7
        MOV   R0,#11          ; FOSC 11 MHz
        MOV   R1,#02H         ; function code
        MOV   A,R5
        CALL  0FFF0H          ; Call Philips firmware
        MOV   R7,A            ; C51 expects return in R7

        ANL   AUXR1,#NOT 20H ;
        RET
    #pragma endasm

}
```

Listing A.3: eraseBlock.c: eraseBlock() - Löscht einen Block des Flash-Speichers während der Laufzeit (In Application Programming, IAP)

```
1
2  /** Löscht den Block 3 des Flash-Speichers
3   *  Nach Ausführung dieser Funktion führt der Prozessor einen RESET durch!!!
4   *
5   * byte block: 0x0000 : block 0
6   *             0x2000 : block 1
7   *             0x4000 : block 2
8   *             0x8000 : block 3
9   *             0xC000 : block 4
10  */
11 void eraseBlock ()
12 {
13
14     #pragma asm
15         MOV   AUXR1,#20H     ; Setze das ENBOOT bit
16         MOV   R0,#11         ; FOSC 11 MHz ODER 0 für Quick Erase
17         MOV   R1,#01H        ; Funktions-Code (QuickErase #81H, sonst #01H)
18         MOV   DPTR, #8000H
19         CALL  0FFF0H         ; Aufruf Philips Firmware (fest im 1 kByte Mikrocontroller ROM)
20         RET
21     #pragma endasm
22
23 }
```

B Board

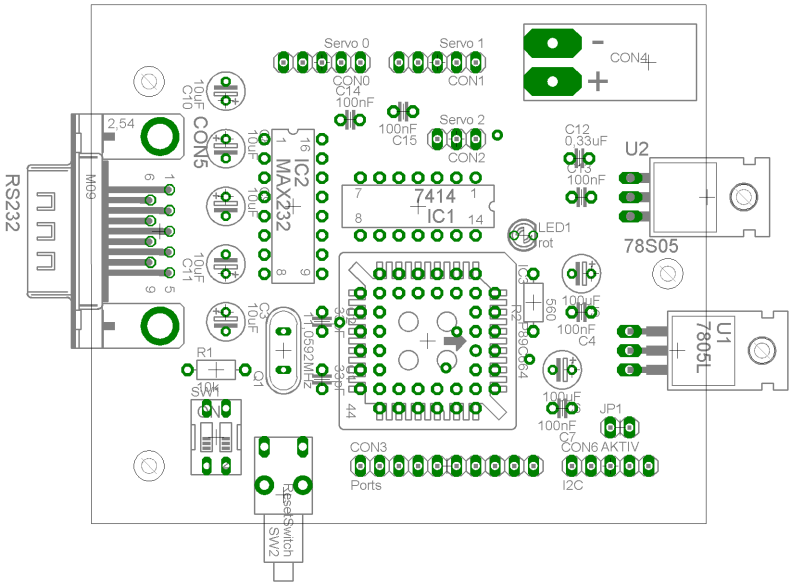

Abbildung B.1: Bauteile-Ansicht

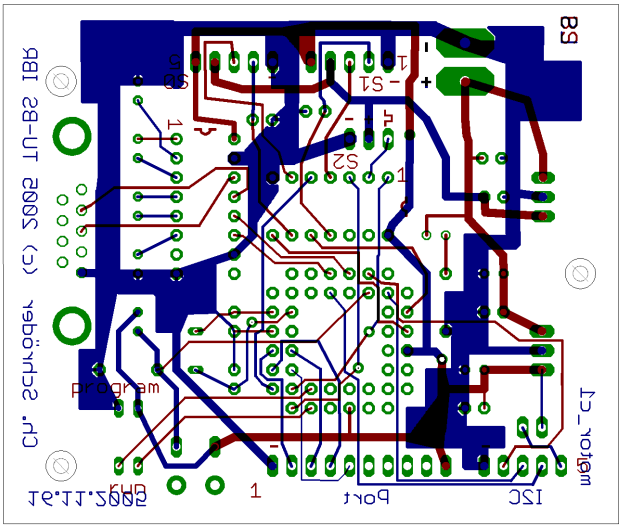

Abbildung B.2: Löt- und Bestückungsseiten

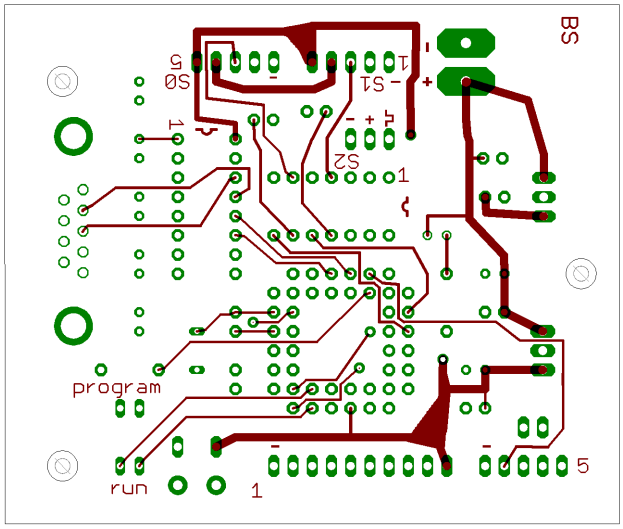

Abbildung B.3: Bestückungsseite

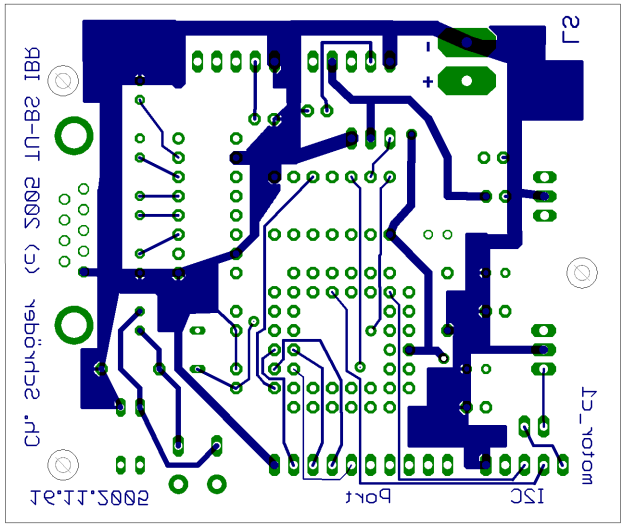

Abbildung B.4: Lötseite

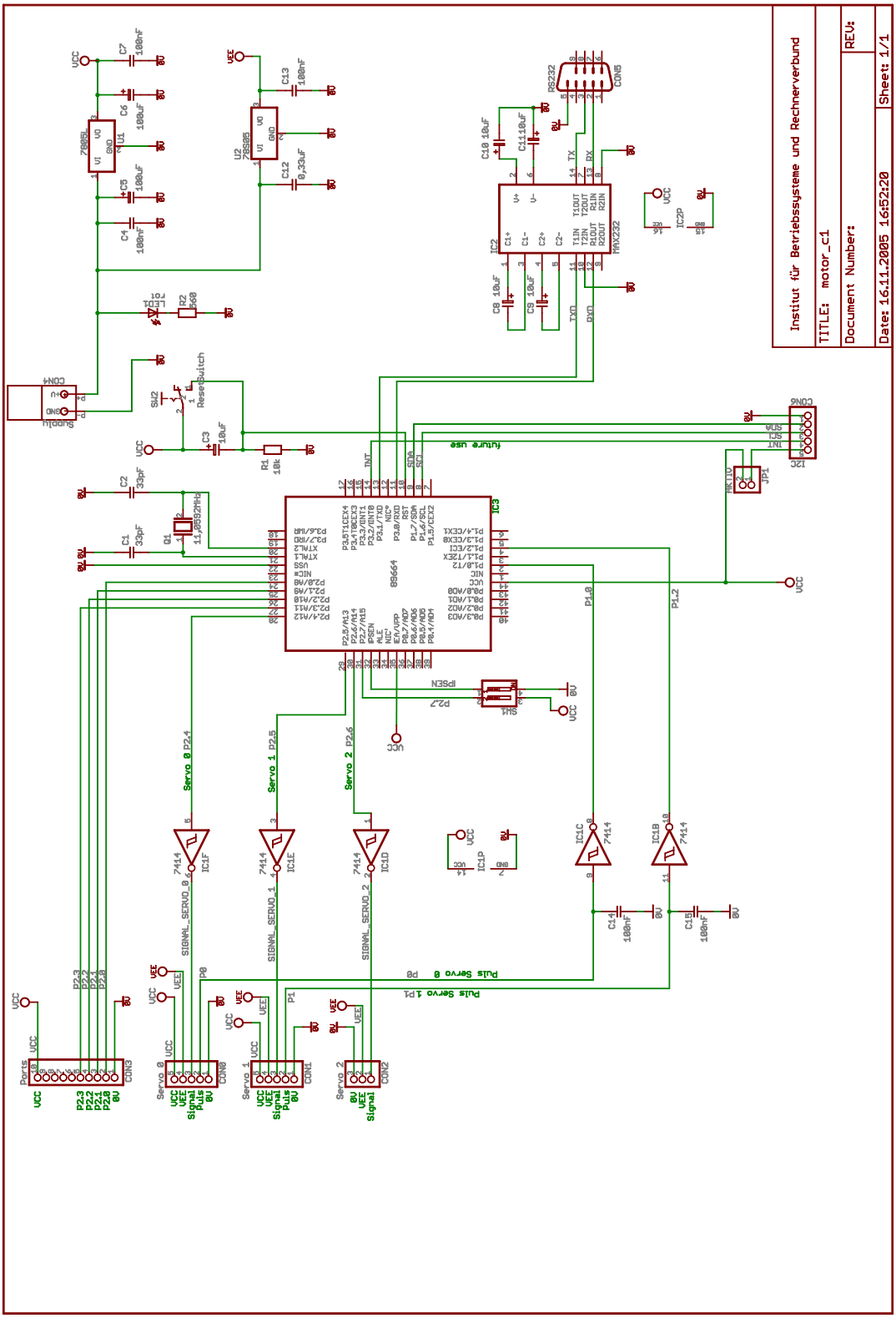

Abbildung B.5: Schaltbild [Diese Seite wird ersetzt durch faltbare DIN A3-Seite mit Schaltbild im Querformat]

Kondensatoren		
C1	33 pF	Keramik-Kondensator
C2	33 pF	Keramik-Kondensator
C3	10 μF	Elko, radial, 10 μF, 35V
C4	100 nF	Keramik-Kondensator Vielschicht 10%
C5	100 μF	Elko, radial, 100 μF, 16V
C6	100 μF	Elko, radial, 100 μF, 16V
C7	100 nF	Keramik-Kondensator Vielschicht 10%
C8	10 μF	Elko, radial, 10 μF, 35V
C9	10 μF	Elko, radial, 10 μF, 35V
C10	10 μF	Elko, radial, 10 μF, 35V
C11	10 μF	Elko, radial, 10 μF, 35V
C12	0,33 μF	Keramik-Kondensator, 0,33 μF (oder Elko)
C13	100 nF	Keramik-Kondensator Vielschicht 10%
C14	100 nF	Keramik-Kondensator Vielschicht 10%
C15	100 nF	Keramik-Kondensator Vielschicht 10%
Widerstände		
R1	10 kΩ	1/4W, 5%
R2	560 Ω	1/4W, 5%
Spannungsregler		
U1	7805	1A positiv, TO-220
U2	78S05	2A positiv, TO-220 + Kühlkörper
Leuchtdiode		
LED1	3mm rot	2V, 20mA

Tabelle B.1: Stückliste Teil 1

Schalter		
SW1 SW2	2-polig Taster	Dip-Schalter, stehend Drucktaster, Schließer, gewinkelt, Printanschluss
Quarz		
Q1	11,0592 MHz	Standardquarz, Grundton, Gehäuse HC49/U-S/U
ICs		
IC1	7414	Hex Schmitt-Trigger Inverter, DIL 14-polig + Fassung (evtl. mit Kondensator)
IC2	MAX232	RS-232 Driver/Receiver (+5V), DIL 16-Polig + Fassung mit Kondensator
IC3	P89C664	Philips Mikrocontroller, Plastic Leaded, Chip Carrier S44 + Fassung (evtl. mit Kondensator)
Jumper		
JP1	2-polig	2-polige Stiftleiste, gerade, 1-reihig, Rastermaß 2,54mm
Steckverbinder		
CON0	5-polig	Platinen-Steckverbinder 5-polig 90 Grad gewinkelt
CON1	5-polig	Platinen-Steckverbinder 5-polig 90 Grad gewinkelt
CON2	3-polig	3-polige Stiftleiste, gerade, 1-reihig, Rastermaß 2,54mm
CON3	10-polig	Platinen-Steckverbinder 10-polig 90 Grad gewinkelt
CON4	2-polig	gewinkelter Wannenstecker und Anschlussklemmensystem 2-pol, Rastermaß 5,08mm
CON5	D-SUB 9-polig	D-SUB-Stecker, 9-polig, gewinkelt
CON6	5-polig	Platinen-Steckverbinder 5-polig 90 Grad gewinkelt

Tabelle B.2: Stückliste Teil 2