UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

# Regelung eines Linearaktors mit digitalen Messschieber und Mikrocontroller

## Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Beneditk Jöbgen

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Hannes Frey<br>Institut für Informatik |
| Zweitgutachter: | Dr. Merten Joost<br>Institut für Integrierte Naturwissenschaften,<br>Abteilung Physik |

Koblenz, im März 2013

**Erklärung**

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

. . . . . .

(Ort, Datum) (Unterschrift)

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Entwicklung eines Regelsystems zur genauen Ansteuerung eines Linearaktors. Hierzu wurde an einer Linearführung zusätzlich ein digitaler Messschieber angebracht, welcher von einem Mikrocontroller ausgelesen wird. Außerdem wurde der Schrittmotor der Linearführung durch einen Gleichstrommotor ersetzt. Im Verlauf der schriftlichen Ausführung wird zunächst einmal überblickhaft darüber informiert, wie ein Regelkreis im Allgemeinen funktioniert (Kapitel 3.1) und auf welche Weise das vorliegende System ein solcher ist (Kapitel 3.2). Weiterführend wird der PID-Regler vorgestellt (Kapitel 3.3). Ein grundlegender Teil der Arbeit bestand im Entwickeln der Hardware (Kapitel 4). Hierzu war es zunächst erforderlich das Protokoll des Messschiebers zu verstehen, um das Auslesen von diesem mittels Mikrocontroller in Angriff zu nehmen (Kapitel 4.1). Daran anschließend wird der Antrieb der Linearführung mittels Gleichstrommotor näher in Augenschein genommen sowie dessen notwendige galvanische Abkopplung (Kapitel 4.2). Abschließend werden in Kapitel 4.3 noch weitere Details der Hardware genannt sowie ein Gesamtüberblick gegeben. Im folgenden Kapitel 5 wird die Struktur der Software näher beleuchtet. Es wird erklärt aus welchen Modulen das Programm besteht (Kapitel 5.1), wie eine eventbasierte Parallelität implementiert wurde (Kapitel 5.2) und inwiefern es sich hierbei um einen Zustandsautomaten handelt (Kapitel 5.3). Im Kapitel 6 wird sich nun mit den speziellen Reglern befasst, welche im vorliegenden System zum Einsatz kommen. Hierbei handelt es sich um drei verschiedene Regelalgorithmen, die jeweils für einen Systemzustand konzipiert wurden. Daran anschließend wird in Kapitel 7 der vollständige Quellcode der einzelnen

Module vorgestellt und erklärt. Inwiefern das System veränderbar ist und wie einzelne Module ausgetauscht werden können, ist in Kapitel 8 einzusehen. Abschließend ist noch zu prüfen, ob die Ziele des Projektes erreicht wurden (Kapitel 9).

# Kapitel 2

# Motivation

Ziel dieses Projektes ist es, eine mechanische Platinenfräse der Universität Koblenz zu verbessern. Da die Linearführungen des Kreuztisches der Fräse aus Preisgründen keine genauen, aber auch teure Kugelumlaufspindeln verwenden, besteht ein mehrere Zehntelmillimeter großes Umkehrspiel. Dieses Spiel ist vor allem an der z-Achse, also bei dem Einsenken der Fräse in das Werkstück, stark ausgeprägt, wenn nicht Platinen sondern andere, härtere Materialien zu fräsen sind.
Diese Arbeit soll nun untersuchen, ob es möglich ist dieses Spiel durch ein geregeltes System auszugleichen. Durch die an der Linearführung angebrachte Schieblehre soll ein Mikrocontroller das aktuelle Umkehrspiel erkennen und ihm durch genaue Motoransteuerung entgegenwirken. Des Weiteren soll der zurzeit genutzte Schrittmotor des Linearaktors durch einen Gleichstrommotor ersetzt werden, welcher in einem ungeregelten System nicht verwendbar wäre. Vorteile des Wechselns wären eine höhere Drehzahl, ein höheres Drehmoment und der geringere Preis der Gleichstrommotoren.
Die Herausforderung hierbei ist es, die sehr sensible Messeinrichtung mit den starken Induktionsspannungen des Motors in einem System zu vereinen sowie die schrittmotorspezifischen Steuerungssignal in Echtzeit zu interpretieren und modifiziert an den Gleichstrommotor weiterzuleiten.

# Kapitel 3

# Regelkreis

## 3.1 Der Regelkreis im Allgemeinen

Unter einem Regelkreis versteht man ein geschlossenes System aus Messen, Vergleichen und Regeln. Es dient in erster Linie dazu einen Wert möglichst schnell auf einen Sollwert zu bringen und ihn dort zu halten. Solche Systeme lassen sich auch zahlreich im Alltag vorfinden. Hier lassen sich einige Beispiele anbringen: Ein Tempomat im Kraftfahrzeug, welcher die Geschwindigkeit auf einen bestimmten Wert beschleunigt. Ebenso die Körpertemperatur von warmblütigen Lebewesen, die automatisch einen spezifischen Wert hält. Innerhalb der freien Marktwirtschaft kann man auch die Preisbildung als Beispiel für einen Regelkreis nennen. Innerhalb dessen wird der aktuelle Preis so geregelt, dass ein maximaler Gewinn erreicht werden kann. Ein weiteres Beispiel ist ein Regelkreis, der einen Linearantrieb reguliert, welcher in dieser Arbeit im Vordergrund steht.
Im Folgenden wird sich nun dem allgemeinen Aufbau des Regekreises gewidmet. Ein Regelkreis besteht aus insgesamt vier Komponenten. Wie Abbildung 3.1 zeigt, handelt es sich hierbei um einen Regler, eine Stelleinheit, eine Regelstrecke sowie eine Messeinheit.
Die **Regelstrecke** ist das Objekt, welches die Eigenschaft besitzt, die von dem Regelkreis gesteuert wird. In den oben genannten Beispielen wäre dies das Kraftfahrzeug mit seiner Geschwindigkeit oder der Körper mit seiner Temperatur. Diese Eigenschaft, auch Regelgröße genannt, wird von der Stelleinheit beeinflusst und von der Messeinheit gemessen.
Unter einer **Messeinheit** versteht man einen Sensor, der die Regelgröße misst

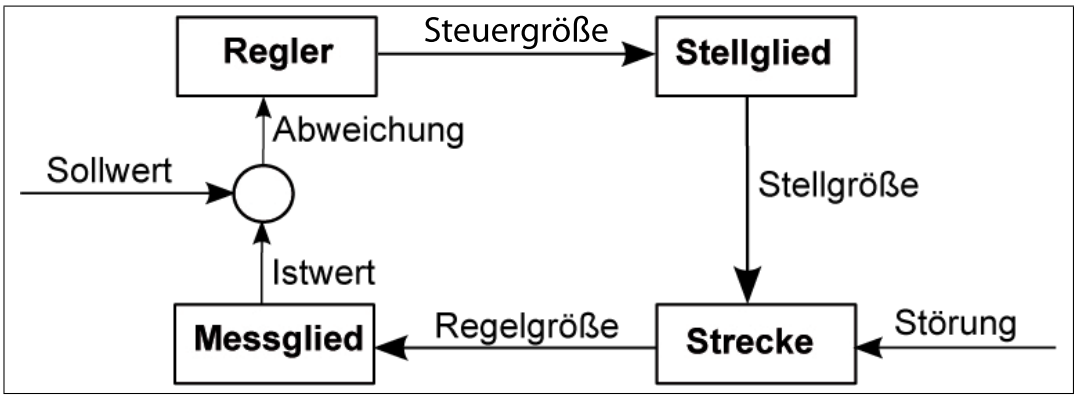


Abbildung 3.1: Regelkreis

und ihren Wert in ein vom Regler lesbares Signal umwandelt. Dieses Signal stellt den Istwert dar, welcher nun mit dem Sollwert zu der sogenannten Abweichung verrechnet wird.
Der **Regler** ist nun dafür zuständig aus dieser Abweichung eine Steuergröße zu errechnen, mit dieser die Regelgröße bestmöglich auf den Sollwert gebracht wird. Für diese Regelung existieren in der Digital- und Elektrotechnik bereits gut bewehrte Methoden - die PID-Regler -, welche in Kapitel 3.3 näher beschrieben werden.
Die errechnete Steuergröße ist vorgesehen für die vierte Komponente, die **Stelleinheit**. Dieses Segment stellt den Aktuator im System. Er beeinflusst die Regelstrecke über die Stellgröße und verändert somit die Regelgröße. In den Beispielen wären diese der Motor, der das Fahrzeug antreibt, oder Diejenigen, die in der Wirtschaft die Preise festlegen. Nun ist der Kreislauf geschlossen.
Die Rückführung der Regelgröße über die Messeinheit wäre grundsätzlich nicht notwendig, da man die Auswirkungen der Stellgröße auf die Regelgröße errechnen oder empirisch testen könnte. Dies ist allerdings nicht möglich, wenn Störungen auf die Regelstrecke einwirken, die nicht genau vorhersagbar sind. Äußere Einflüsse, wie zum Beispiel die Außentemperatur oder im Fall des Tempomats Steigungen und Gefälle der Fahrbahn, können solche Störungen sein. Da solche Einflüsse in den meisten Stellsystemen vorhanden sind, besitzen Regelkreise eine so enorme Wichtigkeit.

## 3.2 Der Regelkreis im Speziellem

Diese Arbeit befasst sich im Grunde mit einer Instanz des Regelkreises. Eine Schienenführung soll auf eine bestimmte Position gefahren und dort gehalten werden, bis vom Host neue Anweisungen eintreffen (siehe Abbildung 3.2).
Die Regelstrecke ist hierbei eine circa 20cm lange Linearführung, welche über einen Gewindetrieb angetrieben wird. Ihre Position stellt die Regelgröße. Diese Regelgröße wird von einer an der Schiene angebrachten Schieblehre gemessen und über einem digitalen Ausgang von einem Atmel ATmega16 ([atm13]) ausgelesen. Dieses Signal mit dem Positionswert ist der Istwert. Die Angaben zur Zielposition gelangen über einen Host-PC an den Mikrocontroller. Da diese eigentlich für eine Schrittmotorsteuerung vorgesehen sind, kommen sie allerdings in einzelnen Ticks, die jeweils einen Schritt von einstellbarer Größe in eine bestimmte Richtung bedeuten. Der so erhaltene Sollwert wird nun mit dem Istwert zu einer Positionsdifferenz zwischen dem angestrebten Ziel und der aktuellen Position verrechnet. Nun nehmen sich verschieden Regelroutinen (beschrieben

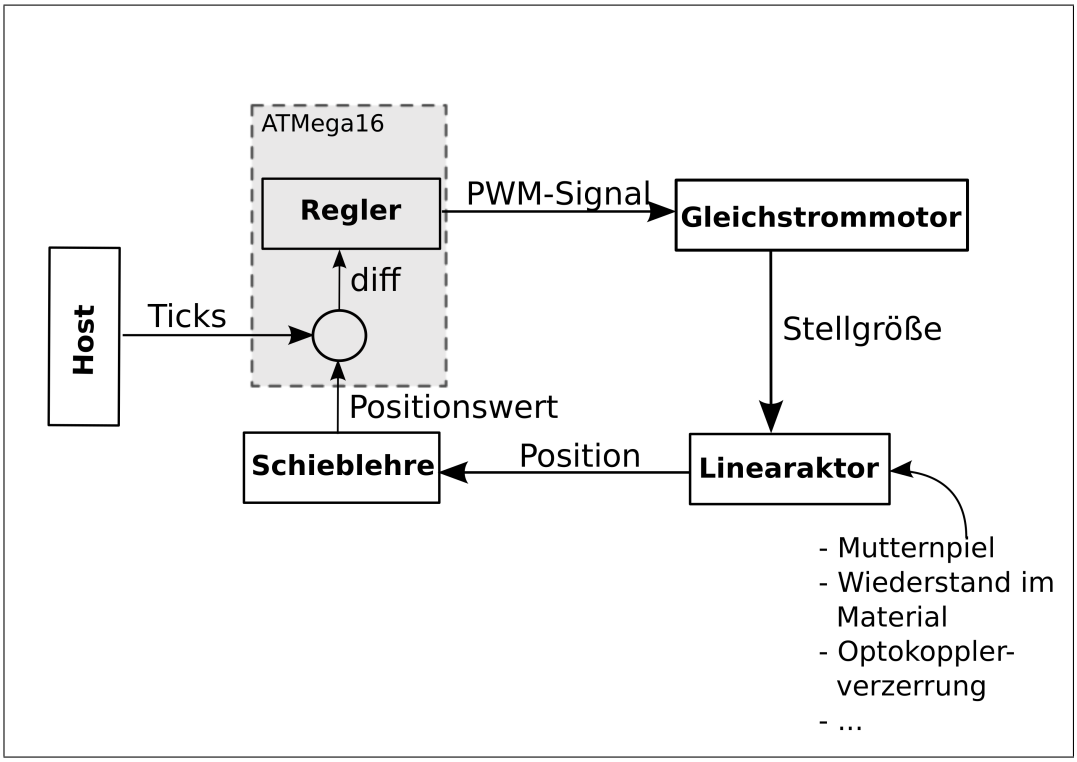


Abbildung 3.2: Der Regelkreis im Speziellem

in Kapitel 6 "Regelalgorithmen") diesen Differenzwert oder teilweise auch die einzelnen Soll- und Istwerte und errechnen die gewünschte Geschwindigkeit. Da das Stellglied ein Gleichstrommotor ist, kann das Signal nicht einfach als Taktmotorsignal weitergegeben werden, so wie es über den Host in das System hineingelangt ist. Stattdessen müssen die Regelroutinen eine pulsweitenmodulierte Steuergröße erzeugen, die der Motor wie ein Analogsignal wahrnimmt. Dieser Motor treibt nun eine Gewindestange an, welche den oberen Teil der Linearführung bewegt, womit der Kreis geschlossen wäre.
Das Problem dieses System liegt darin, dass sich die Führungsschiene nicht hundertprozentig an die Stellgröße hält. Durch Spiel im Gewindetrieb oder durch mechanischen Widerstand in der linearen Bewegung spielt eine nicht vernachlässigbare Störung mit, wodurch ein Regelkreis notwendig wird. Genaueres dazu im Kapitel 2 "Motivation".

## 3.3 PID-Regler

Ein PID-Regler ist die Reihenschaltung von drei Reglern, um eine möglichst schnelle und genaue Regelung zu erhalten: dem P, I und D-Regler. PID-Regler haben sich in der Praxis sehr bewährt und können sowohl elektrotechnisch als auch in der Software implementiert werden. In diesem Kapitel wird nur auf die Software Regler eingegangen, da diese auch in dieser Arbeit genutzt werden.

### 3.3.1 P-Regler

Der P-Regler ist der einfachste Regler. Er beeinflusst die Stellgröße "p"roportional zur Abweichung von Soll- und Istwert. Je größer die Abweichung, desto stärker muss nachgeregelt werden. Er sorgt dafür dass der Istwert möglichst schnell auf dem Sollwert gebracht wird. Mathematisch lässt sich dies wie folgt darstellen:

$$y(t) = Kp * e(t)$$

hierbei ist $y$ die Stellgröße und $e$ die Regelabweichung abhängig vom Zeitpunkt $t$. $Kp$ ist eine systemabhängige Konstante, die empirisch festzustellen ist. Im Quellcode sähe solch ein Regler folgendermaßen aus:

```
y = Kp * e;
```

Das Makel des P-Reglers besteht darin, dass er sehr schnell "übersteuert" und dadurch zu schwingen anfängt, wodurch er sehr lange braucht bis er den Sollwert stabil erreicht hat. Im schlimmsten Fall kann es sogar passieren, dass es zur Resonanz kommt und die Schwingung sich immer weiter aufschaukelt, wodurch die Abweichungen nur noch größer werden, wie in Abbildung 3.3 zu erkennen. [Was13]

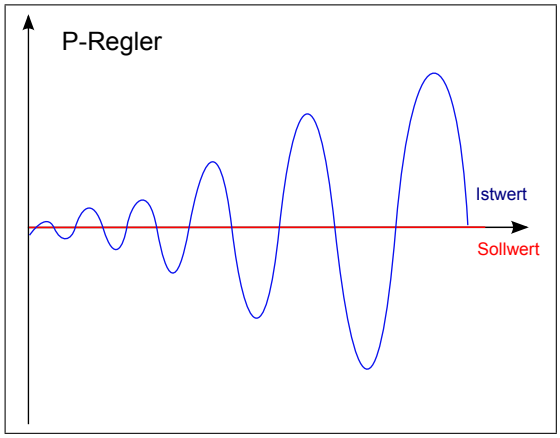


Abbildung 3.3: Schwingung eines P-Reglers

### 3.3.2 PD-Regler

Um dem im Kapitel 3.3.1 beschriebenen Schwingen entgegen zu wirken, nutzt man zusätzlich noch den "d"ifferenzierenden D-Regler. Er versucht den Istwert auf einen stabilen Wert zu bringen, unabhängig von der tatsächlichen Abweichung. Hierzu betrachtet der D-Regler lediglich die Änderung der Abweichung und agiert proportional zu dieser. Ein D-Regler macht alleinstehend keinen Sinn und man findet ihn daher nur in Verbindung mit einem P-Regler vor:

$$y(t) = Kp * e(t) + Kd * \frac{\delta e(t)}{\delta t}$$

Hierbei ist $Kd$ wieder eine systemabhängige Konstante. Implementieren könnte man dies so:

```
y = Kp * e + Kd * (e - ealt)/dt;
ealt = e;
```

Hierbei steht $dt$ für die Abtastzeit. Im Quellcode der vorliegenden Arbeit findet man diesen Divisor nicht, da die Abtastzeit konstant bleibt und $dt$ somit auch eine Konstante ist. Diese kann direkt mit $Kd$ verrechnet werden, um zur Laufzeit einen Rechenschritt und somit Zeit zu sparen. [Was13]

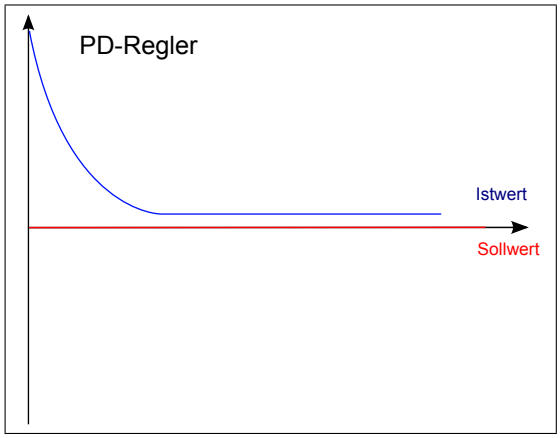


Abbildung 3.4: Verlauf eines PD-Reglers

Nun wurde erreicht, dass der Sollwert möglichst schnell auf einen stabilen Wert fährt, der im Idealfall dem Sollwert entspricht. Probleme treten nur auf, wenn es einen systembedingten Bias, eine konstante Verzerrung, gibt. Dieser lässt sich mit einem PD-Regler nicht beseitigen. Dieses Fehlverhalten erkennt man in Abbildung 3.4.

### 3.3.3 PID-Regler

Der I-Regler ändert den Stellwert abhängig von den auf“i"ntegrierten Abweichungen. Je länger eine Abweichung besteht, desto stärker wird also dagegen angegangen. Dies führt dazu, dass jede Differenz ausgeglichen wird, allerdings recht langsam im Vergleich zu den anderen Reglern. [Was13]

$$y(t) = Ki * \int e(t)\delta t$$

bzw. in Software implementiert:

```
esum = esum + e;
y = Ki * esum * dt;
```

Wie beim PD-Regler kann man auch hier das $dt$ bei einer konstanten Abtastzeit direkt mit $Ki$ verrechnen.
Um keine I-Regler spezifischen Geschwindigkeitseinbüßungen zu haben, kombiniert man nun den I-Regler mit einem PD-Regler, wodurch man das bestmögliche Ergebnis erreicht. Der PID-Regler bringt den Istwert schnell und genau auf den Sollwert:

$$y(t) = Kp * e(t) + Kd * \frac{\delta e(t)}{\delta t} + Ki * \int e(t)\delta t$$

bzw.

```
esum = esum + e;
y = Kp * e;                    // P
y += Ki * esum * dt;           // I
y += Kd * (e - ealt)/dt;       // D
ealt = e;
```

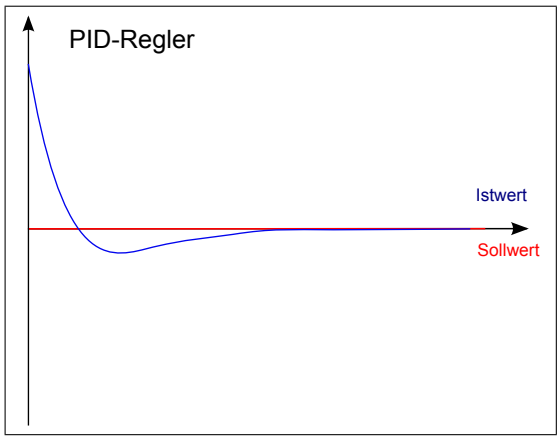


Abbildung 3.5: Verlauf eines PID-Reglers

# Kapitel 4

# Die Hardware

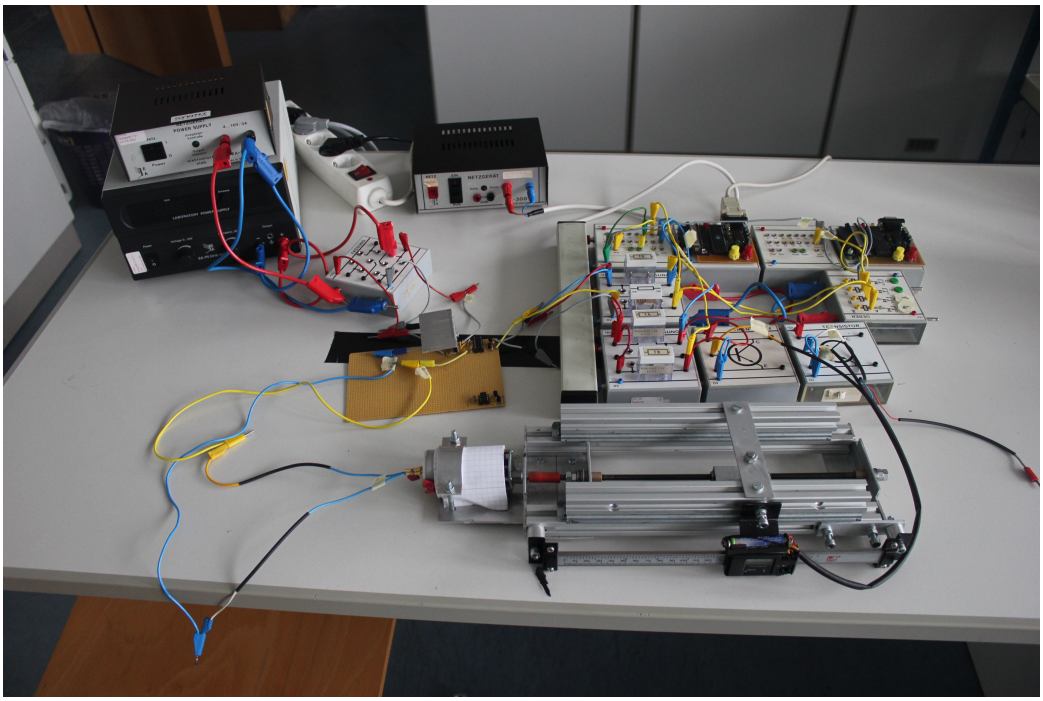

Abbildung 4.1: Der Versuchsaufbau

Die Hardware besteht im Grunde aus drei Komponenten: dem Linearantrieb, dem Messschieber sowie dem ATmega16. Damit diese jedoch miteinander kommunizieren können ist noch einiges an Peripherie hinzuzufügen. Hauptursache hierfür waren die unterschiedlichen Spannungsniveaus mit denen die einzelnen Bauteile arbeiten sowie die auftretenden Induktionsspitzen durch den Elektro-

motor. Der Versuchsaufbau (Abbildung 4.1) enthält noch einen weiteren Mikrocontroller mit angebundenem Bedienpanel, der aber lediglich zur Simulation des Host-PCs zum Testen und Verifizieren dient. Im folgenden wird genauer auf die Bauteile und ihre Kommunikation eingegangen.

## 4.1 Der Messschieber

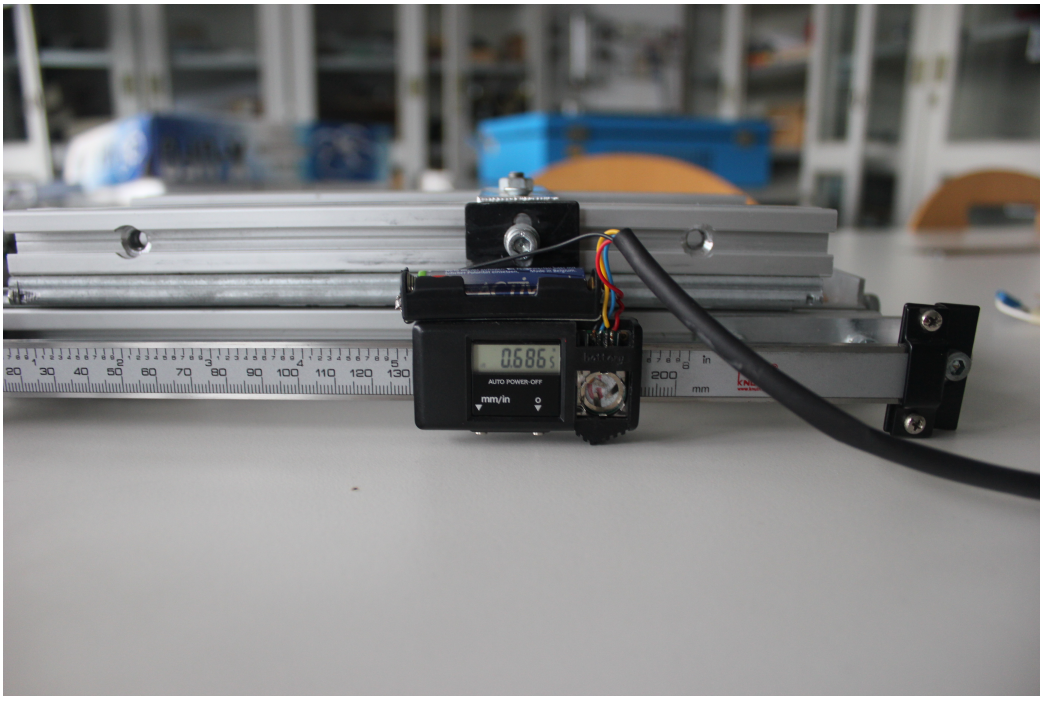


Abbildung 4.2: Der Messschieber

### 4.1.1 Die Kommunikation

Für den Versuchsaufbau wurde ein günstiger digitaler Messschieber verwendet, der so an den Linearantrieb angebracht wurde, dass er die Verschiebungen zwischen den beiden Schienen misst (Abbildung 4.2. Der Messschieber besitzt zwei Tasten, eine für den Wechsel zwischen Zoll und Millimeter, die andere um die aktuelle Position als Null zu definieren. Neben der digitalen Anzeige verfügt er noch über einen digitalen Ausgang, über dem er seine aktuelle Position periodisch sendet. Hierfür wird ein Protokoll verwendet, welches in Kapitel 4.1.2

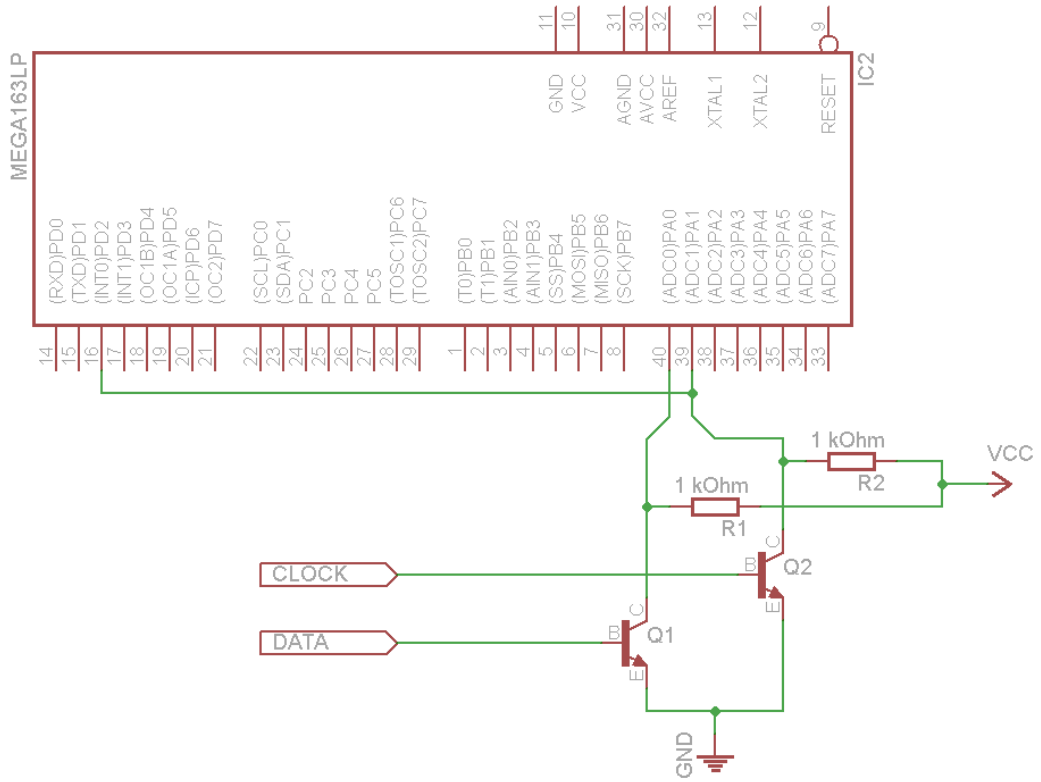


Abbildung 4.3: Die Messchieber-Anbindung

näher beschrieben wird. Versorgt wird die Schieblehre (nach einem kleinem Umbau) von einer AAA-Batterie, was bedeutet, dass die interne Logik auf 1,5 Volt arbeitet. Der ATmega wird allerdings mit 5 Volt betrieben und die Grenze zwischen high und low-Pegel eines Input-Pins liegt bei circa 2 bis 2,5 Volt. Dies bedeutet, dass man das Signal der Schieblehre nicht direkt auslesen kann.
Der Ausgangsport der Schieblehre besteht aus vier Pins: Die Masse, die mit der Masse des Controllers verbunden werden muss. Die Versorgungsspannung, an die die externe Batterie angeschlossen wurde und die für die Konfiguration benötigt wird (vgl. Kapitel 4.1.3). Sowie die clock- und die data-Leitung über die die Positionswerte übermittelt werden. Die Dauer eines übertragenen Bits misst gerade mal 13$\mu$s, was bedeutet das der benötigte Signalverstärker sehr schnell sein muss. Realisiert wurden diese Signalverstärker mit jeweils einem Transistor, dessen Basis, an die die jeweilige Leitung gekoppelt wurde, schon bei kleineren Spannungswerten einen high-Wert annimmt. Der Kollektor wird sowohl mit dem

Eingangspin des ATmega's verbunden als auch mit den 5Volt Versorgungsspannung. Der Emitter wird jeweils gegen Masse geschaltet. Diese Schaltung bewirkt, dass auf dem Eingangs-Pin im Normalfall stets 5 Volt anliegen, lediglich wenn an der clock- beziehungsweise data-Leitung ein high-Signal anliegt schaltet der Transistor durch und zieht damit den Atmel-Pin gegen Masse. Damit in diesem Fall kein Kurzschluss entsteht, muss jeweils noch ein Widerstand zwischengeschaltet werden. Das gesamte Schaltnetz für die Schieblehrensignalübertragung ist in Abbildung 4.3 zu sehen. Die Schaltung bewirkt nun zwar, dass das Signal invertiert ist, was aber kein Problem darstellt, da der Controller dies einfach wieder zurück-invertieren kann. Damit der ATmega nicht ständig aktiv an der Leitung horchen muss, ob ein Signal ankommt, wurde der clock-Eingang noch mit dem Pin für einen externen Interrupt verbunden. Sobald nun ein Signal ankommt wird die Interrupt-Routine ausgelöst, die den Übertragenen Wert ausließt, Genaueres dazu findet sich in Kapitel 7.2.

### 4.1.2 Das Protokoll

Im Normalfall sendet die Schieblehre drei mal die Sekunde ihre Position über den digitalen Ausgang. Mit einer gewissen Tastenkombination, die in Kapitel 4.1.3 beschrieben wird, kann man dies auf 50 Hertz erhöhen. Jedes Signal ist ein Paket aus zwei 24-Bit-Zahlen, und hat insgesamt eine Länge von 850$\mu$s. Im Gegensatz zu vielen anderen Messschiebern handelt es sich hierbei nicht exakt um die Zahl die auch auf dem Display dargestellt wird. Ändert man mit einer Taste die Displaywerte von Zoll zu Millimeter, so hat dies keinerlei Einfluss auf die übertragenen Werte.
Mit der ersten Zahl wird eine absolute Position übertragen, unabhängig vom letzten drücken der Zero-Taste. Der Messschieber hat allerdings keinen fest definierten Nullpunkt, dieser wird vielmehr bei jedem kompletten Ausschalten neu definiert. Für den Regelkreis dieser Arbeit genügt dies jedoch, da nur relative Bewegungen gemessen werden und keine absoluten Positionen bekannt sein müssen. Die zweite Zahl, die direkt im Anschluss übertragen wird, ist dagegen relativ zu der Position, an der das letzte mal die Nullstell-Taste gedrückt wurde.
Die Einheit in der die Werte übertragen werden ist 1/20480 Zoll. Umgerechnet sind dies circa 806,3 Schritte pro Millimeter, die theoretisch eine Auflösung von 0,00124mm ergeben. Die letzten Bits flackern allerdings sehr stark, sodass diese

hohe Auflösung praktisch nicht gewährleistet werden kann. In diesem Projekt werden daher einfach die untersten drei Bits ignoriert, wodurch noch eine Genauigkeit von $1/20480Zoll * 2^3 = 1/2560Zoll = 1/100,787mm$ bleibt und damit etwas über der gewünschten Genauigkeit von $1/100mm$ liegt.
In der Abbildung 4.4 ist der schematische Ablauf des Protokolls dargestellt und

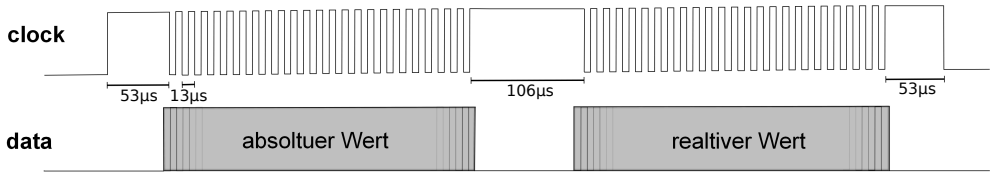


Abbildung 4.4: Der Schematische Ablauf des Messchieber-Protokolls

in Abbildung 4.5 erkennt man eine tatsächliche Übertragung, die am Oszilloskop aufgenommen wurde. Wird ein Paket gesendet, so wird die clock-Leitung zuerst als Präambel 53$\mu$s auf high gesetzt. Im Anschluss kommen 23 weitere Pulse mit einer Periodendauer von 13$\mu$s. Die 24 übertragenen Bits werden auf der Datenleitung, synchron zu den fallenden Flanken der clock-Leitung, gelesen. Zwischen den beiden übertragenen Werten bleibt die clock-Leitung 106$\mu$s auf high, bevor die zweite Zahl übertragen wird, diesmal aber im zweier Komplement. Zum Abschluss folgt noch ein 53$\mu$s langes Bit auf der clock-Leitung, bevor beide Leitungen wieder auf ihren default-low-Wert fallen. [MÏ3]

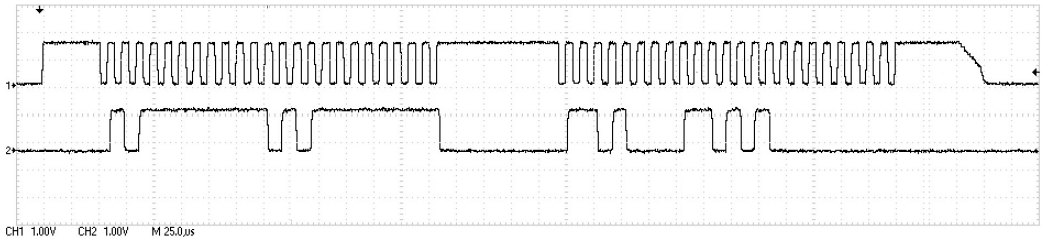


Abbildung 4.5: Das Messschieber-Signal am Oszilloskop (1=clock, 2=data)

### 4.1.3 Die Zustände

Der Messschieber hat intern fünf verschiedene Zustände, in denen er sich befinden kann, zu erkennen in Abbildung 4.6. Schaltet man die Schieblehre ein, so

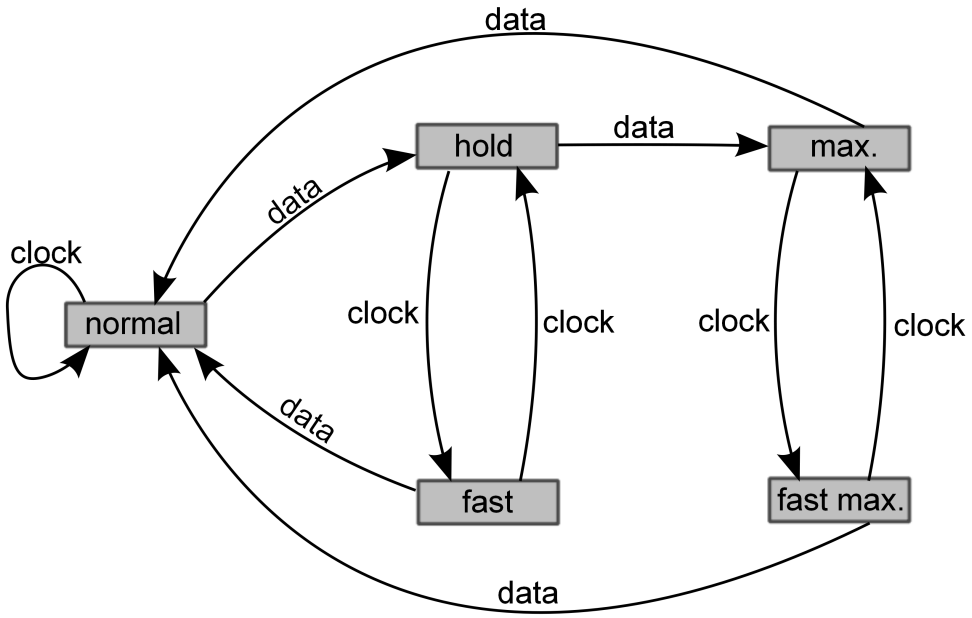


Abbildung 4.6: Die zustände der Schieblehre

befindet er sich im “normalen” Modus, indem er drei mal in der Sekunde seinen Positionswert aktualisiert. Bringt man ihn in den “hold”-Modus, so hält er den aktuellen Wert, unabhängig von jeglichen Bewegungen. Aus diesem Modus kann man in den “fast”-Modus wechseln. Dieser ähnelt dem normalen Modus, allerdings aktualisiert er den Positionswert nun mit 50Hz. Er ist die Konfiguration, in der sich die Schieblehre für das Auslesen des Linearantriebs in diesem Projekt befinden sollte. Des Weiteren gibt es noch zwei Konfigurationen (“max.” und “fast max.”), in denen der die Schieblehre den höchsten angefahrenen Wert anzeigt, entweder mit 3Hz oder mit 50Hz. Man wechselt zwischen den einzelnen Modi, indem man entweder an die clock- oder an die data-Leitung einen high-Pegel von 1,5V anlegt. Alternativ zum high-Pegel an der clock-Leitung kann man auch die Zero-Taste drücken. Für einen high-Pegel an der data-Leitung ist kein Taster angebracht, wodurch man nicht ohne weiteres zwischen den Zuständen wechseln kann. Mit welcher Kombination man in welchen Zustand kommt, erkennt man in Abbildung 4.6. ‘data’ steht hierbei für das Anlegen eines high-Pegels an die data-Leitung und analog steht ‘clock’ für einen high-Pegel an der clock-Leitung.[MÏ3]

Über den digitalen Ausgang werden keinerlei Informationen gegeben in welchem Zustand sich die Schieblehre gerade befindet, wodurch es für den Mikrocontroller schwer ist, die passende Einstellung vorzunehmen. Dies ist aber auch nicht nötig, da die einmal eingestellte Konfiguration so lange bestehen bleibt, bis die Schieblehre ganz ausgeschaltet wird. Das bedeutet, dass lediglich nach einem Batteriewechsel darauf geachtet werden muss, dass die Schieblehre wieder in den "fast"-Modus gebracht wird. Dieser Modus ist zu erkennen an einem kleinen "F" auf dem Display.

## 4.2 Der Antrieb

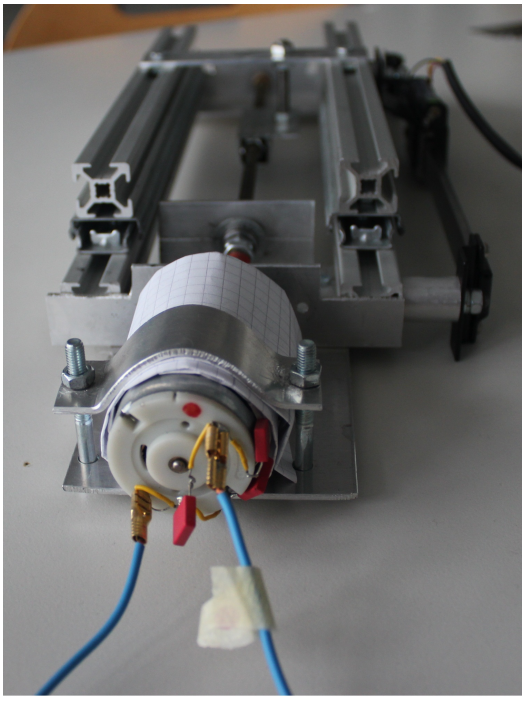

Abbildung 4.7: der Gleichstrommotor an der Linearführung

### 4.2.1 Das Signal

Der Linearantrieb wird durch einen Gleichstrommotor angetrieben. Die Geschwindigkeitsregelung geschieht über die Regulierung der Höhe der Versorgungsspan-

nung. Alternativ ist es aber auch möglich, ein pulsweitenmoduliertes Signal anzulegen. Der Motor nimmt dies als analoges Signal wahr. Da es mit dem ATmega16 ohne weiteres möglich ist, ein PWM-Signal zu erzeugen, ist dies auch die Wahl in dieser Arbeit. Neben der Geschwindigkeit muss auch die Drehrichtung des Motors gesteuert werden. Soll der Linearantrieb vorwärts fahren, so liegt auf dem einen Motoreingang das PWM-Signal an, der andere Eingang wird gegen Masse geschaltet. Ist auf dem PWM-Eingang nun ein high-Pegel, so besteht eine Spannungsdifferenz von "Vs" zwischen Masse- und PWM-Eingang, der Strom "fließt" also vom PWM- zum Masse-Pin. Soll der Motor nun in die andere Richtung drehen, so wird auf dem Masse-Pin ein high-Pegel angelegt und das PWM-Signal wird invertiert. Dies führt nun dazu, dass wenn ein low-Pegel am PWM-Eingang anliegt, eine Spannungsdifferenz von -"Vs" zwischen Masse- und PWM-Pin liegt, der Strom "fließt" also andersrum, wodurch der Linearantrieb rückwärts fährt. Alternativ könnte man für diese Problemstellung auch eine H-Brückenschaltung einsetzen.

### 4.2.2 Der Optokoppler

Ein großes Problem des Versuchsaufbaus war, dass der Gleichstrommotor sehr starke Induktionsspannungen auf den Leitungen verursachte, die vor allem beim Anlaufen oder Abbremsen des Motors so stark waren, dass der Messschieber dadurch fehlerhafte Werte lieferte. Auch nach Dämpfung der Induktionsspitzen durch Kondensatoren zwischen den beiden Eingangsleitungen des Motors sowie jeweils ein Kondensator zwischen der Eingangleitung und dem Motorgehäuse, blieben die Induktionsspitzen noch zu hoch für den empfindlichen Messschieber. Als einziger Ausweg blieb die Trennung der Massen des Mikrocontrollers und der Schieblehre von der Masse des Motors, sodass die Induktionsspitzen nicht auf dem ganzen System verteilt werden. Hierzu dient ein Optokoppler. Ein Optokoppler wandelt ein digitales Signal mittels einer LED in ein Lichtsignal um, welches von einer Photozelle erkannt und wieder in ein digitales Signal mit getrennter Masse umgewandelt wird. Dadurch sind die beiden Massen komplett voneinander getrennt und die Signalweitergabe nur in eine Richtung möglich, sodass kein Störsignal über die Leitungen zurückfließen kann. Beim Erstellen des Schaltnetzes muss noch bedacht werden, dass der Optokoppler das Signal invertiert, weswegen man einen Transistor vorschalten sollte, um wieder das richtige

Signal zu erhalten (siehe Abbildung 4.10).
Ein weiteres sehr wichtiges Detail bei der Trennung der Massen, ist die Entkopplung des Motorgehäuses von dem Gehäuse des Messschiebers, da beide jeweils mit ihren Massen gekoppelt sind. Im Normalfall sitzen beide Gehäuse direkt an der metallernen Linearschiene, wodurch auch ihre Massen direkt miteinander verbunden sind. Ein einfaches doppelt gefaltetes Stück Papier um das Motorgehäuse genügt schon, um diese Verbindung zu trennen, zu erkennen in der Abbildung 4.7.

#### 4.2.2.1 Die Optokopplerverzerrung

Für den Versuchsaufbau wurde der Optokoppler "6N138" verwendet ( [opt13] ), da er schnell genug für das PWM-Signal schaltet. Vorherige Versuche mit einem anderem Koppler schlugen fehl, da eine fallende Flanke zwar sehr schnell, quasi direkt weitergegeben wird, so hat die steigende Flanke doch eine gewisse Verzögerung. Auch bei dem "6N138" ist dies zu erkennen. In Abbildung 4.8 erkennt man das in den Optokoppler eingehende Signal (unten) und wie es aus dem Optokoppler wieder herauskommt (mitte), mit deutlich langsamer ansteigender

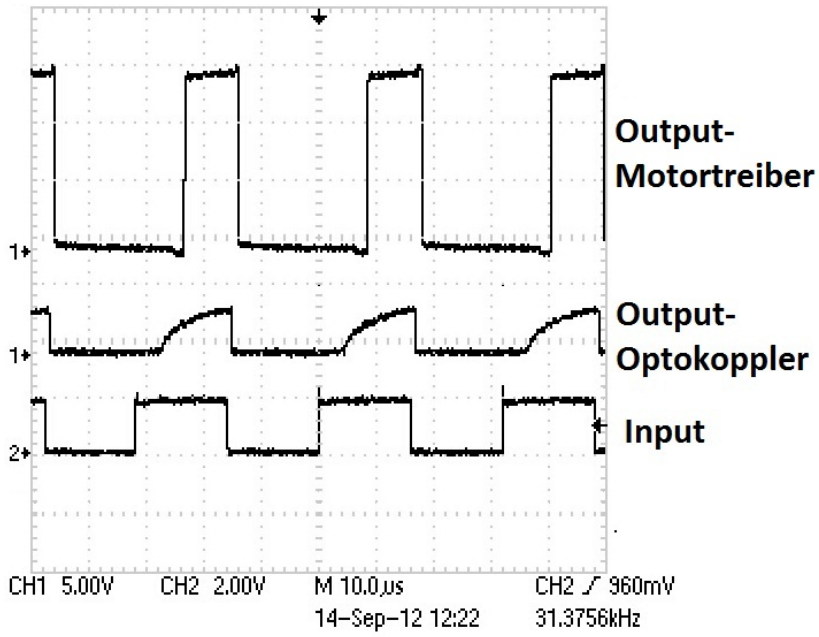


Abbildung 4.8: Verzerrung durch den Optokoppler

Flanke. Die oberste Spur der Abbildung zeigt das Ausgangssignal des hinter dem Koppler geschalteten Motortreibers, welches durch die langsam ansteigende Flanke des Treiber-Eingangsignals erst verzögert auf high wechselt, wodurch das PWM-Signal ein wenig vermindert wird. Dies ist aber nicht weiter schlimm, man muss es nur in der Software bedenken und dem Fehler dort entgegenwirken.

### 4.2.3 Der Motortreiber

Der Versorgungsspannung für den Motor (Vs) kann man bis zu 15 Volt hochdrehen und beim Anlaufen oder Richtungswechsel benötigt dieser bis zu einem Ampere. Dies bedeutet, dass ein solcher Motor nicht direkt an den Mikrocontroller angeschlossen werden kann, es muss ein Treiberbaustein zwischengeschaltet werden. In der vorliegenden Arbeit wurde dazu der L298 (genauer ein L298N [l2913]) genommen. Dieser Baustein sorgt dafür, dass ein eingehendes 5 Volt Signal auf ein Signal mit bis zu 50 Volt und bis zu 3 Ampere übersetzt wird, abhängig von der Eingangsspannung (Vs). In dem Versuchsaufbau lag diese bei 15 Volt, da der Motor für höhere Spannungen nicht ausgelegt ist. Den internen Aufbau des Treiberbaustein sieht man in Grafik 4.9.

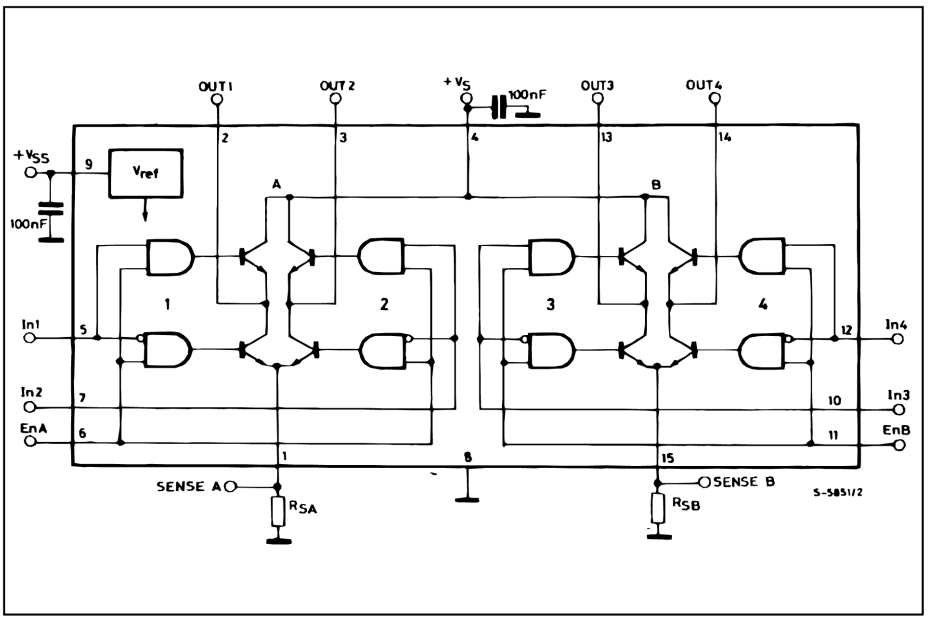


Abbildung 4.9: Aufbau eines L298 Motortreibers [l2913]

### 4.2.4 Das Schaltnetz

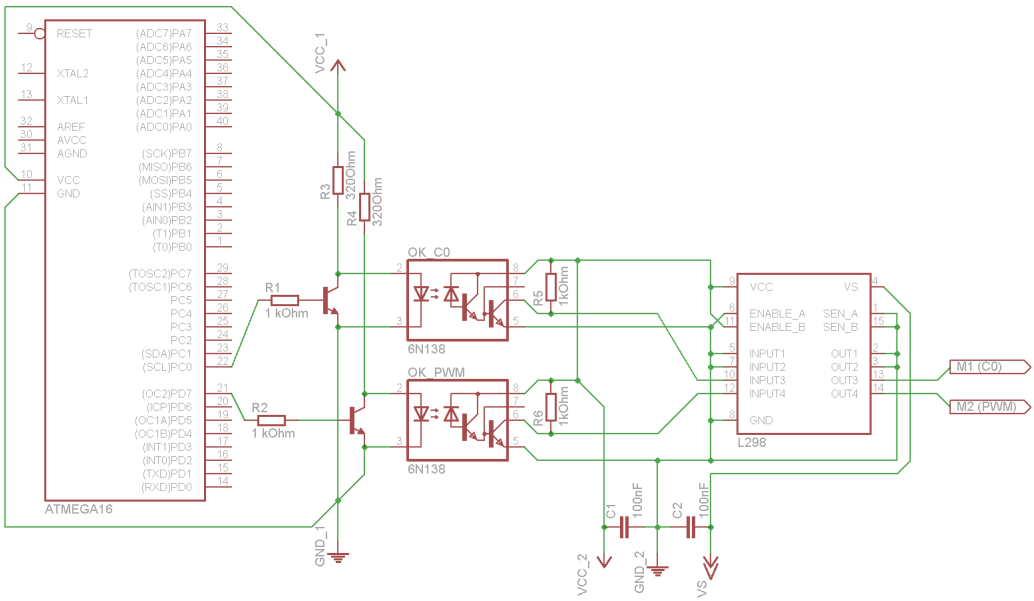


Abbildung 4.10: Die Motor-Anbindung

Man erkennt in Abbildung 4.10, dass der Aufbau des Schaltnetzes für die Motoransteuerung symmetrisch ist. Der obere Teil dient der Übertragung zur Richtungssteuerung und der untere Teil ist für die Übermittlung des pulsweitenmodulierten Signals zuständig. Die zwei Ausgänge, die für die Motorsteuerung benötigt werden sind der Pin PC0, über den die Drehrichtung des Motors gesteuert wird, sowie der Pin PD7, da an ihm das Pulsweitenmodulat anliegt. Beide Signale steuern, über einen Transistor zur Invertierung (siehe Kapitel 4.2.2), die Diode des Optokopplers, welcher das Signal weiterreicht an die Eingänge INPUT3 und INPUT4 des Motortreibers. Dieser erhöht das Signal auf die an Vs anliegende Spannung, in diesem Fall sind dies 15 Volt. Die Ausgänge OUTPUT3 und OUTPUT4 des Treiberbausteins führen direkt an den Gleichstrommotor. Am Treiber muss der Pin ENABLE_B gesetzt werden, um damit die Stränge drei und vier zu aktivieren, die Stränge eins und zwei würden mit ENABLE_A aktiviert. Im vorliegenden Fall wurden aber die Stränge drei und vier wegen ihrer Pinbelegung bevorzugt, um die Erstellung der Platine zu erleichtern. In einem erweiterten Versuchsaufbau zum Vergleich des Gleichstrommotors mit einem Schrittmotor wurden die Stränge eins und zwei ebenfalls genutzt. Durch sie lief das invertierte

Signal der anderen beiden Stränge, um somit den Schrittmotor zu steuern.
Zu erkennen ist, dass die von den Optokopplern linke und rechte Hälfte jeweils eine eigen Masse, GND_1 und GND_2, sowie eine eigen Versorgungsspannung der Logik, VCC_1 und VCC_2, besitzen. Dadurch bleiben die Induktionsspitzen des Motors lediglich in der rechten Hälfte des Aufbaus und stören weder den ATmega noch den Messschieber.

## 4.3 Das gesamte Schaltnetz

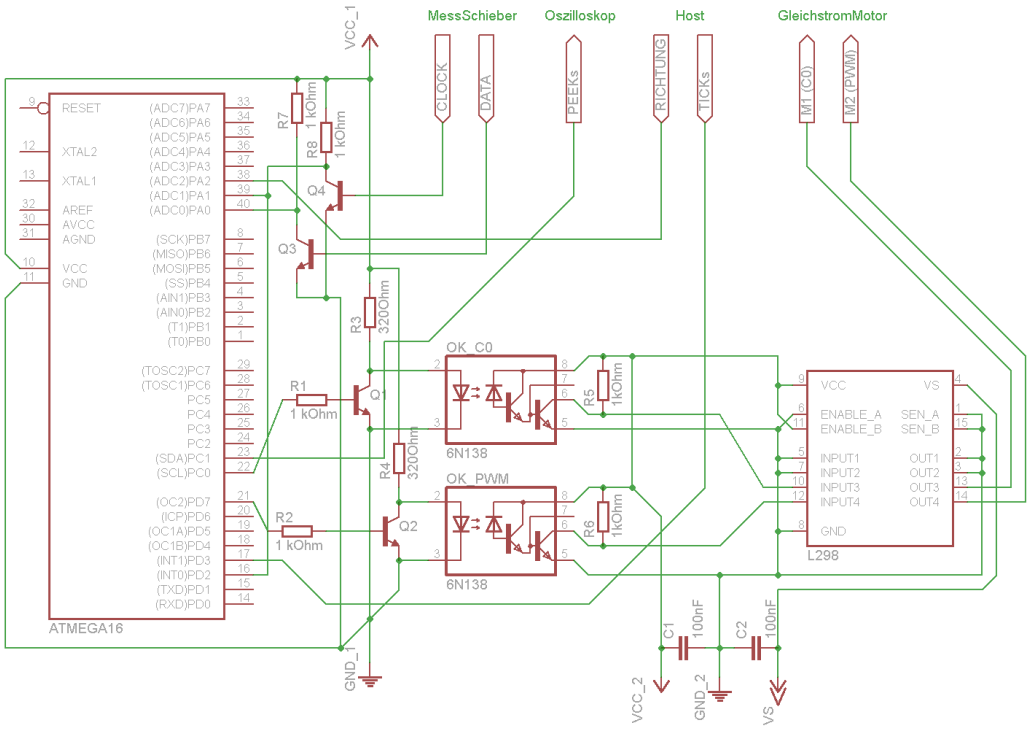


Abbildung 4.11: das komplette Schaltnetz

Betrachtet man das komplette Schaltnetz in Abbildung 4.11, so erkennt man, dass der größte Teil aus der Anbindung der Schieblehre und des Linearantriebs besteht, beschrieben in Kapitel 4.1.1 und Kapitel 4.2.4. Hinzu kommen noch zwei Leitungen, die vom Host kommen. Die Leitung, über die die Ticks gesendet werden, geht an Pin PD3 um dort bei jedem Tick externe Interrupts auslösen zu können. Die zweite Leitung geht an Pin PA2, über diese wird die Information über

das Vorzeichen des aktuell reinkommenden Ticks übermittelt. Außerdem wird vom Pin PC1 noch ein Ausgang an ein Oszilloskop geleitet, es dient lediglich dem Debugging und der Verifikation.

# Kapitel 5

# Software-Architektur

## 5.1 Modularität

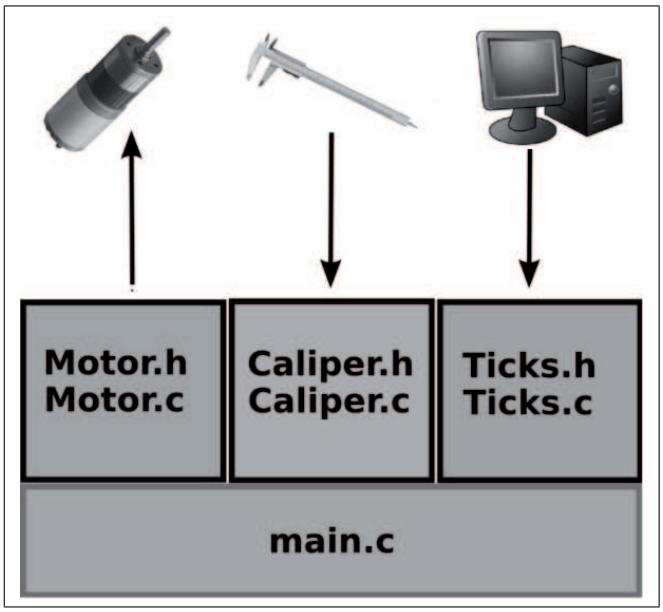


Abbildung 5.1: Die einzelnen Module und ihre Funktionalität

Die Software, die auf dem ATmega16 läuft wurde modular aufgebaut, um es möglichst übersichtlich zu halten, damit es möglichst gut wartbar und an Änderungen anpassbar ist. Das System wurde aufgeteilt in je ein separates Modul für das Auslesen der Schieblehre, eines für die Handhabung der Inputs des Hosts

sowie eines für die Motorsteuerung (siehe Abbildung 5.1). Ebenso gibt es noch ein Hauptmodul in dem die main-Routine läuft sowie ein Uart-Modul zur Evaluation und zum Debugging, welches in der finalen Version nicht mehr benötigt wird.
Die Modularität hat den Vorteil, dass es nun sehr einfach wird einzelne Komponenten der Hardware auszutauschen. So kann zum Beispiel eine neue Schieblehre eingesetzt werden, welche ein anderes Protokoll verwendet, oder den Motor zu Vergleichszwecken (siehe Kapitel 2) durch einen Schrittmotor ersetzt werden und es muss dazu lediglich ein einzelnes Modul angepasst bzw. ersetzt werden.

## 5.2 Parallelität

Die Software muss in der Lage sein mit hoher Frequenz den Motor zu regeln und gleichzeitig jedes Signal der Schieblehre zu empfangen. Das Regeln nimmt durch regelmäßige Gleitkommarechnungen relativ viel Zeit in Anspruch und die einige hundert Mikrosekunden langen Schieblehrensignale werden mit einer Frequenz von 50 Hertz gesendet. Ebenso darf die Software keinen einzigen Tick vom Host verpassen, welche mit bis zu 500 Hertz gesendet werden. Um dies zu gewährleisten wäre eine parallele Arbeitsweise auf mehreren Threads sinnvoll, die sich allerdings auf einem ATmega nicht so ohne weiteres implementieren lässt, daher wurde hier eine alternative Vorgehensweise gewählt.
In der main-loop laufen nacheinander mehrere Funktionen, die für das Regeln der Stellgröße und für das direkte Ansteuern des Motors zuständig sind. Wenn nun ein Signal von der Schieblehre oder ein Steuer-Tick vom Host eintrifft, so wird durch die richtige Verdrahtung ein externer Interrupt INT0 bzw. INT1 ausgelöst. Dadurch wird die main-Schleife verlassen und das Programm springt in die passende Interrupt Service Routine. Die Routine zum Verarbeiten der Ticks ist sehr kurz, da sie nur schnell überprüfen muss, welches Vorzeichen der Tick hat und dies abspeichert, danach kann der programpointer direkt wieder an die vorherige Stelle zurückspringen. Die Routine zum Auslesen des Schieblehren-Signals ist allerdings etwas problematischer, da ein solches Signal fast eine ganze Millisekunde dauert und man das Programm nicht zu jederzeit problemlos so lange unterbrechen kann. Es kommt hierbei zu Fehlern, wenn während dieser Zeit noch weitere Ticks eintreffen, da normalerweise andere Interrupts blockiert werden, solange sich das Programm in einer Interruptroutine befindet. Dies wür-

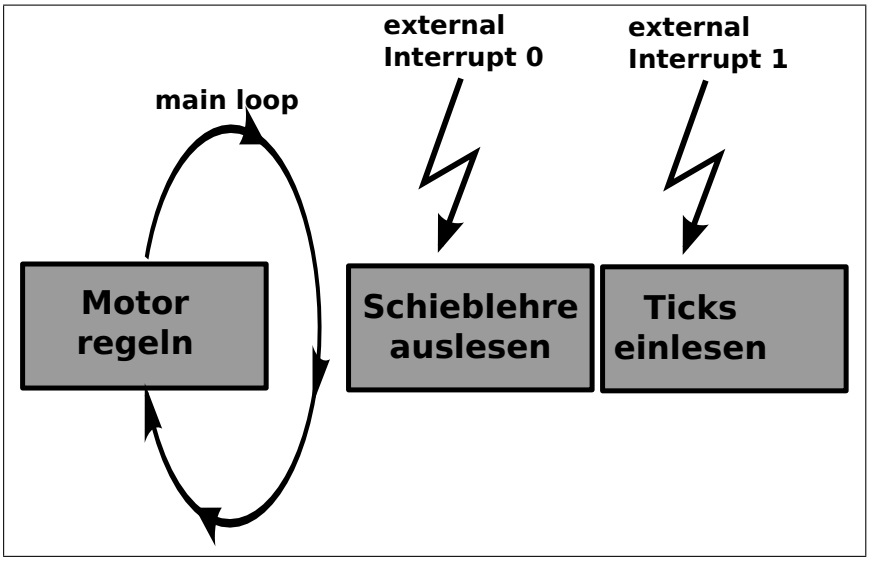


Abbildung 5.2: Parallel ablaufende „Threads“

de bedeuten, dass in einer ungünstigen Situation einige Ticks nicht wahrgenommen würden. Daher ist hier darauf zu achten andere Interrupts manuell zuzulassen. Des Weiteren muss darauf geachtet werden, dass die Dauer einer Interrupt Service Routine der Ticks kürzer ist als die Dauer eines Bits des Schieblehrensignals, damit keines dieser Bits „verschluckt“wird. Daher befinden sich in der Tick-Auslese-Routine keinerlei Berechnungen, lediglich ein Vermerk, dass ein solcher Tick angekommen ist, damit dies in der main-Schleife bearbeitet werden kann.
Des Weiteren existieren noch weitere Interrupt Service Routinen, welche allerdings sehr kurz sind und für deren Bedeutung zu verstehen man detailreicher in den Code schauen müsste. Daher wird in diesem Kapitel nicht genauer auf diese eingegangen, mehr dazu im Kapitel 7.

### 5.2.1 Alternative Signalauslese-Routine

Im finalen Code startet die Schieblehrensignal-Interrupt-Routine sobald ein Signal eintrifft und wird erst beendet, nachdem dieses komplett angekommen ist. Eine gute Alternative wäre es, durch jedes einzelne Bit einen Interrupt auslösen zu lassen, in dessen Routine das einzelne Bit gelesen wird und welche direkt danach wieder beendet wird. Dadurch hat man anstatt einer einzigen langen Routi-

ne, eine Vielzahl sehr kurzer Routinen, wodurch eine Unterbrechbarkeit gegeben wäre und noch weniger Zeit in Anspruch genommen würde. Tests haben allerdings gezeigt, dass dies in dem vorliegenden Fall nicht funktioniert, da die Bitfrequenz der Schieblehre enorm hoch ist und das Umswitchen in die ISR(Interrupt Service Routine) relativ zu dieser zu lange dauert. In den meisten Fällen funktioniert dies zwar, allerdings wird ab und zu doch ein Bit „übersprungen“, wodurch das ganze Programm nicht mehr richtig funktionierte. Würde man jedoch eine Schieblehre mit deutlich geringerer Frequenz verwenden, wie es in den ersten Tests auch der Fall war, so wäre dieses Vorgehen die elegantere Variante.

## 5.3 Zustandsautomat

Aufgabe des Systems ist es, die Linearführung sowohl auf einer Stelle zu halten und bei externen Bewegungen die Position gegebenenfalls nachzukorrigieren, als auch bei eintreffenden „Ticks“vom Host in der passenden Geschwindigkeit zu fahren. Sobald keine Ticks mehr vom Host kommen, ist es außerdem notwendig möglichst schnell den Zielpunkt zu erreichen, ohne über das Ziel hinauszufahren und ohne in einen Schwingungszustand zu gelangen. Da dies drei sehr unterschiedliche Szenarien sind, die unterschiedliche Regelverfahren bedürfen, ist es angebracht die Motorregelung als Zustandsautomaten zu implementieren.
Der Zustand **HOLD** ist dafür zuständig, die Linearführung an der aktuellen Position zu halten. Verändert sich diese Position durch äußere Einflüsse, so muss der Regler möglichst schnell wieder auf die angestrebte Position zurückfahren. In dieser Position verbleibt das System solange, bis die ersten Ticks vom Host eintreffen. Nun wird in den **RUN**-Zustand gewechselt, in dem die zeitlichen Abstände der eintreffenden Ticks analysiert werden und die Schieblehre mit passender Geschwindigkeit gefahren wird. Das System bleibt in diesem Zustand solange wie weitere Ticks eintreffen. Bleiben diese Ticks für eine gewisse Zeitspanne aus, so wird in den Zustand **STOPPAGE** gewechselt, in dem die Zielposition möglichst optimal angefahren wird. Ist das Ziel erreicht, wechselt das System automatisch wieder in den Hold-Zustand und wartet dort auf weitere Ticks. Dieser Kreislauf ist in Abbildung 5.3 graphisch dargestellt.

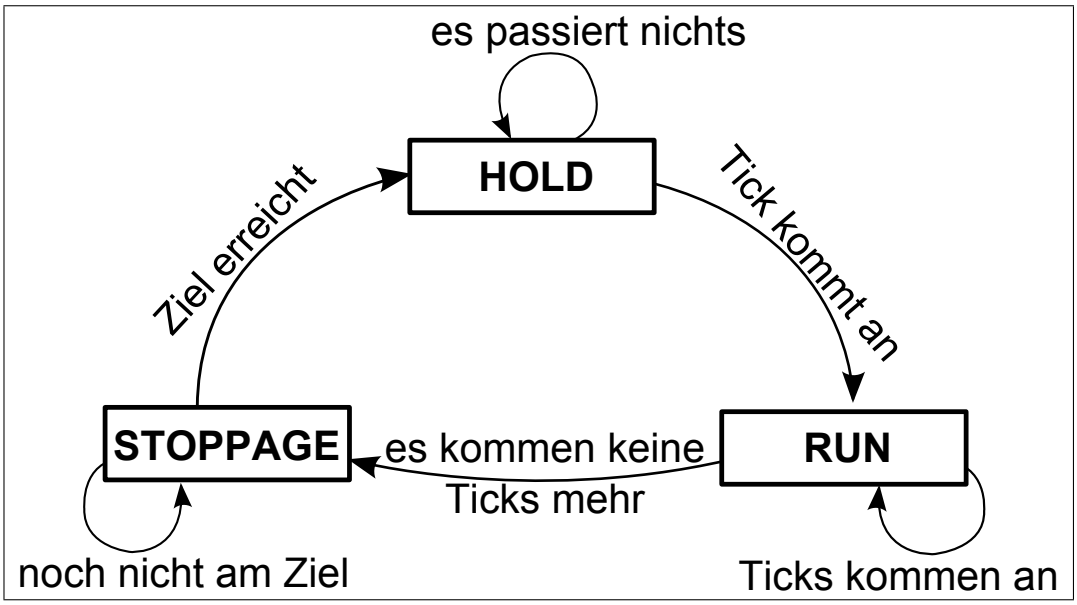


Abbildung 5.3: Statemachine der Motorregelung

# Kapitel 6

# Regelalgorithmen

Wie in Kapitel 5.3 beschrieben, werden für die Motorsteuerung drei verschiedene Regelalgorithmen benötigt, einer für jeden Zustand der statemachine. Im folgenden Kapitel werden die einzelnen Zustände und ihre spezifischen Regelalgorithmen genauer beschrieben.

## 6.1 Regelung im HOLD-Zustand

Ziel des Haltezustandes ist es, die Linearführung auf einer gegebenen Position zu halten. Die Problematik besteht hierbei darin, dass nur sehr kleine Positionsdifferenzen ausgeglichen werden müssen, was bedeutet dass der Motor sehr langsam drehen muss, da das System ansonsten sehr schnell in einen Schwingungszustand gerät. Nun kann man aber das Anfahrverhalten eines Gleichstrommotors mit der physikalischen Reibung vergleichen: Haftreibung ist größer als Gleitreibung. Dies bedeutet, dass der Motor eine gewisse Spannung benötigt, um überhaupt erst anzufahren. Beginnt er sich jedoch zu drehen, so tut er dies annähernd direkt mit erhöhter Geschwindigkeit. Ist er einmal am drehen, könnte man den Motor zwar runterdrosseln auf eine langsamere Geschwindigkeit, allerdings ist er bis dahin schon über das Ziel hinaus gefahren. Hierdurch ist es fast unmöglich sehr kurze Strecken geregelt mit sehr langsamer Geschwindigkeit zu fahren, was hier allerdings benötigt würde.
Die Lösung des Problems besteht darin, dem Motor die volle Spannung zu geben, diese aber nach einer sehr kurzen Zeitspanne wieder komplett zu nehmen, sodass er nur kurze Spannungsschübe erhält. Er fährt jedes mal kurz an, stoppt

jedoch sofort wieder. Wendet man dies periodisch an, so ist es möglich den Motor noch langsamer fahren zu lassen, als es eine konstante minimale Spannung zulässt. Im Pseudocode sieht dies folgendermaßen aus:

```
while(is != goal)
{
    drive_towards_goal(v_maximal);
    delay(2ms);
    stop();
    delay(30ms);
}
```

Die *2ms* und *30ms* wurden hierbei empirisch bestimmt und sind Motorabhängig. Es wurde auch versucht ein P-Regler zu implementieren und die Wartezeit abhängig von der Entfernung der aktuellen Position (*is*) bis zum Ziel (*goal*) zu setzen. Um bei größeren Abständen trotzdem nicht zu schnell zu werden und in Schwingungen zu geraten, war die Wartezeit nicht linear, antiproportional zu der Entfernung, sondern antiproportional zur Wurzel der halben Entfernung. Dies brachte in den meisten Situationen zwar auch ein gutes Ergebnis und war bei größeren Position-Ziel-Differenzen schneller, allerdings war es nicht möglich ein Schwingen komplett zu vermeiden. Da die Schwingungsvermeidung eine wesentlich höhere Priorität hat als die Geschwindigkeit, wurde dieser Lösungsansatz wieder verworfen.

## 6.2 Regelung im RUN-Zustand

### 6.2.1 Grundgeschwindigkeitsberechnung

Sobald die ersten Ticks vom Host eintreffen wechselt die Motorregelung in den RUN-Zustand. Ziel ist es nun die Linearführung mit genau der Geschwindigkeit zu fahren, die die Ticks angeben. Weil jeder Tick immer eine Verschiebung des Ziels um eine gewisse Strecke bedeutet, ist die zu fahrende Geschwindigkeit abhängig von der Zeitspanne zwischen den Ticks. Diese Zeitspanne kann allerdings schwanken kann. Aus diesem Grund ist es notwendig, die benötigte Geschwindigkeit nach jedem eintreffenden Tick neu zu berechnen.

Zu fahrende Geschwindigkeit [inch/s]:

$$v = \frac{s}{t}$$

Strecke eines Ticks [inch]:

$$s = \frac{1}{TicksPerInch}$$

Zeit zwischen den Ticks [s]:

$$t = \frac{FCPU}{TickPeriode * TimerPrescaler}$$

Einzustellender PWM-Wert:

$$pwm = v * \frac{PWM_{max}}{V_{max}}$$

Daraus resultiert die Formel:

$$pwm = \frac{FCPU * PWM_{max}}{TicksPerInch * TimerPrescaler * V_{max}} * \frac{1}{TickPeriode}$$

hierbei ist:
- $\frac{FCPU}{TimerPrescaler}$ die Taktfrequenz des Timers, der die Abstände zwischen den ticks misst
- $PWM_{max}$ der maximale Wert für die Pulsweitenmodulation, bei einer 8-Bit-Modulation also 255
- $\frac{1}{TicksPerInch}$ ist die Schrittweite die ein Tick darstellt
- $V_{max}$ ist die gemessene Geschwindigkeit die die Linearführung bei voller Pulsweite fährt
- $TickPeriod$ ist die Anzahl der Takte zwischen zwei Ticks

Man erkennt, dass der erste Teil des Produkts nur aus Konstanten besteht. Lediglich die $TickPeriod$ variiert und muss zur Laufzeit verrechnet werden.

### 6.2.2 PD-Regler

Führe man nun mit dieser berechneten Geschwindigkeit ( $v_{berechnet}$), so würden der Soll- und der Ist-Wert trotzdem sehr schnell auseinander driften, da noch viele nicht berechenbare Einflüsse wirken. Daher ist die Verwendung eines Reglers

notwendig. Dieser Regler regelt aber im Gegensatz zu dem HOLD-Regler nicht die Position, sondern die Geschwindigkeit. Hierbei kommt ein PD-Regler, ähnlich wie in Kapitel 3.3.2 beschrieben, zum Einsatz. Durch diesen wird schnellstmöglich die gewünschte Geschwindigkeit erreicht ohne in einen Schwingungszustand zu gelangen. Außerdem kann schnell auf Änderungen reagiert werden. In dem Regler wurde allerdings die Zeitkomponente ($dt$) weggelassen, da Tests gezeigt haben, dass diese sehr stabil auf dem selben Wert bleibt, wodurch man sie einfach in die Konstante $Kd$ mit einrechnen kann. Auf diese Weise spart man sich das Messen der Ausführungszeit eines Schleifendurchgangs sowie performancelastige float-Berechnungen. Daraus ergibt sich nun folgende Formel als Regler:

$$e\_alt = e$$

$$e = soll - ist$$

$$v = v_{berechnet} + [Kp * e] + [Kd * (e - e\_alt)]$$

### 6.2.3 Distanz

Nun besteht noch die Problematik, dass Ticks ohne Voranmeldung eintreffen und man theoretisch sofort mit der passenden Geschwindigkeit fahren müsste. Dies ist nicht möglich, da zuerst die Geschwindigkeit berechnet werden muss und der Motor auch nur relativ langsam anfahren kann. Dadurch entsteht direkt am Anfang schon eine recht große Differenz zwischen Soll- und Ist-Wert. Dies wird versucht zu vermeiden, indem beim ersten Tick für kurze Zeit mit voller Pulsweite gefahren wird, um die Motorträgheit beim Starten zu kompensieren, aber selbst dies reicht nicht aus, um schnell genug aufzuholen. Dieses eigentliche Problem wird sich daher nun zu Nutzen gemacht, da beim plötzlichen Abbremsen am Ende einer zu fahrenden Strecke ein ähnliches Problem besteht: Hier ist der Motor auf hohen Drehzahlen und muss plötzlich schlagartig stehen bleiben. Dies würde im Normalfall bedeuten, dass die Linearführung über das Ziel hinaus fährt. Aus diesem Grund wird gar nicht versucht während einer Fahrt mit der Linearschiene genau da zu sein, wo es die Ticks eigentlich gerade vorgeben, sondern immer mit einem gewissen Abstand zurück zu bleiben. Genau der Abstand, der am Anfang durch den trägen Start verloren ging, beziehungsweise der Abstand der am Ende benötigt wird, um rechtzeitig abbremsen zu können, ohne zu weit

zu fahren. Dieser Abstand wird auch in jedem Schleifendurchlauf neu berechnet, da er proportional zur gefahrenen Geschwindigkeit ist. Daraus ergibt sich nun ein neuer Regler:

$$distance = v\_alt * K\_distance$$

$$e\_alt = e$$

$$e = soll - ist$$

$$v = v_{berechnet} + [Kp * (e - distance)] + [Kd * (e - e\_alt)]$$

Da ausschließlich der P-Regler für die Anfahrt auf das Ziel zuständig ist, muss nur hier dieser Abstand ($distance$) einfließen. Der D-Regler verhindert lediglich die Entstehung des Schwingens unabhängig des absoluten Ziels. Die Konstante $K\_distance$ wurde empirisch bestimmt und liegt hier bei $1/2$. Sie kann nach Belieben noch etwas verändert werden, je nachdem welche Prioritäten man setzt.
In Abbildung 6.1 erkennt man nun den Ablauf einer gefahrenen Strecke. Abgetragen ist hier die Differenz zwischen Soll- und Ist-Wert entlang der y-Achse zur Zeit entlang der x-Achse. Man erkennt, dass anfangs schnell eine gewisse Differenz zwischen Soll- und Ist-Wert entsteht, die möglichst konstant eingehalten wird. Die Differenz ist anfangs doch etwas größer als der geplante Abstand, sodass es zu einer leichten Schwingung kommt, die aber sehr schnell behoben wird. Man erkennt in dem Diagramm ebenso, dass dies genau der Abstand ist, der auch für den Abbremsvorgang benötigt wird.

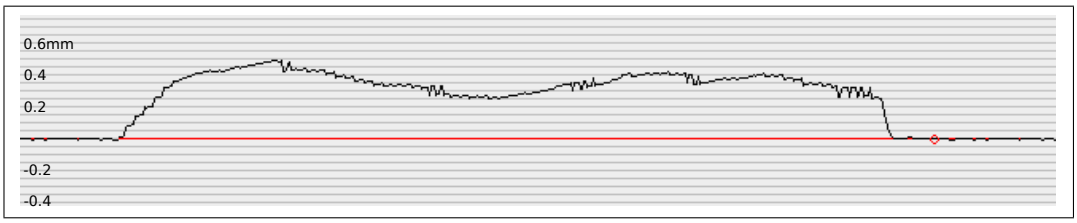


Abbildung 6.1: Zeit-Distanz-Diagramm mit geplantem Abstand

### 6.2.4 I-Regler

Warum wird hier kein I-Regler verwendet? Der I-Regler ist dafür zuständig eine systembestimmte Abweichung zum Sollwert zu eliminieren (siehe Kapitel 3.3.3). Wie im vorherigen Kapitel beschrieben wird aber eine solche Abweichung sogar

gewollt hervorgerufen. Der I-Regler würde zwar sicherstellen, dass diese Abweichung genauer eingehalten wird, allerdings hat eine minimale Verschiebung dieser Abweichung kaum Auswirkungen. Würde man diesen I-Regler trotzdem implementieren, so könnte man auf die Zeitmessung der Schleifendurchläufe ($dt$) nicht verzichten und in jedem Durchlauf würden zeitaufwändige Gleitkommaberechnungen stattfinden. Praktische Versuche haben gezeigt, dass die Frequenz der Schleifendurchläufe hierdurch nur ein Drittel so groß wird, wodurch der Regler nur noch langsamer reagieren kann. Außerdem hat die Praxis gezeigt, dass der I-Regler kaum positive Effekte hervorruft. Insgesamt überwiegen also die Nachteile gegenüber den Vorteilen, weshalb hier nur ein PD-Regler und kein PID-Regler verwendet wird.

### 6.2.5 Grenzüberschreitung

Worauf bei der Geschwindigkeitsregelung noch zu achten wäre, ist dass man die Geschwindigkeit (also die Pulsweite) nicht unter einen gewissen Wert senkt, da der Motor sonst zu wenig Strom bekommt und stehen bleibt, obwohl der Mikrocontroller davon ausgeht er würde langsam drehen. Wenn die Ticks so langsam kommen, dass die Geschwindigkeit unter den Minimalwert geraten würde, dann wird direkt wieder in den HOLD-Zustand gewechselt, da dieser solche langsamen Bewegungen besser regeln kann. Ebenso darf die Pulsweite nicht über den maximalen Wert ($PWM\_max$ = 255) gelangen. Dies wird vor jedem Einstellen der Puslweite überprüft. Gegebenenfalls wird der Wert einfach zurückgesetzt auf die Maximalgeschwindigkeit. In diesem Fall ist der Motor einfach zu langsam und dies kann eine Software nicht ausgleichen.

## 6.3 Regelung im STOPPAGE-Zustand

Wenn sich das System im RUN-Zustand befindet, aber für eine gegebene Zeit keine Ticks mehr eingetroffen sind, so wird in den STOPPAGE-Zustand gewechselt. Hier wird nun die Linearführung möglichst schnell auf den genauen Zielpunkt gebracht. Die Schwierigkeit hierbei besteht darin, dass der Motor aus voller Fahrt abgebremst werden muss und das Ziel punktgenau angefahren werden soll. Dabei darf die Schiene unter keinen Umständen über das Ziel hinausfahren. Bedenkt man, dass die Linearführung in eine Platinenfräse eingebaut wird, so würde dies

bedeuten, dass die Fräse weiter fährt als sie soll und somit mehr gefräst wird als geplant, wodurch eine Platine gegebenenfalls unbrauchbar würde. Daher lag hier die oberste Priorität darin, nicht über das Ziel hinaus zu fahren. Hierdurch wird außerdem automatisch vermieden, dass die Linearführung um den Zielpunkt herum schwingt, welches ein sehr ungünstiger Zustand ist, aus dem man nur sehr schwer wieder heraus kommt.

### 6.3.1 Bremsimpuls

Ähnlich wie bei der RUN-Regelung wird auch hier beim Eintreten des Zustandes ein kurzer starker Impuls mit voller Pulwseite auf den Motor gegeben (vgl. Kapitel 6.2.3).

```
if( erster loop durchlauf im STOPPAGE-Zustand )
{
    drive_towards_v(v_maximal);
    delay(2ms);
    stop();
}
```

Dieser Impuls entgegen der Fahrrichtung soll den Motor, der sich bis zu diesem Zeitpunkt noch in voller Fahrt befand, möglichst stark abbremsen, um nun mit geringer Geschwindigkeit den Zielpunkt anzufahren. Dieser kurze Impuls reicht nicht aus, um den Motor vollkommen zum stehen zu bringen. Dies ist aber auch nicht nötig, da die in Kapitel 6.2.3 beschriebene Distanz nun wieder aufgeholt werden muss.

### 6.3.2 Zielanfahrt

Nachdem der kurze Bremsimpuls gegeben wurde kommt es nun darauf an das Ziel genau anzufahren. Hierbei ist darauf zu achten, dass der Motor nicht zu schnell dreht, da man sonst sehr schnell über das Ziel hinausfährt. Diese restliche zu fahrende Strecke beträgt selbst bei höchster Geschwindigkeit lediglich circa 0.3 Millimeter. Der Regler muss die Geschwindigkeit also möglichst schnell drosseln, darf aber auch nicht komplett abbremsen, da das Ziel möglichst schnell angefahren werden soll.

Der Ansatz hierzu ähnelt einem reinen D-Regler. Das Ziel wird mit der Geschwindigkeit $v_{min}$ angefahren, die allerdings abhängig von der tatsächlich gefahrenen Geschwindigkeit noch verringert wird. Hierbei ist $v_{min}$ die minimale Pulsweite, mit der der Motor gerade noch so dreht. Dadurch dass sich der Motor allerdings schon in voller Fahrt befindet, ist die tatsächliche Geschwindigkeit $v_{real}$ selbst bei einer geringen angelegten Pulsweite höher.
Der sich nun ergebende Regler sieht wie folgt aus:

$$e\_alt = e$$

$$e = soll - ist$$

$$\begin{aligned} v &= v_{min} - Kd'_{stop} * v_{real} \\ &= v_{min} - Kd'_{stop} * C * (|e\_alt| - |e|) \\ &= v_{min} - Kd_{stop} * (|e\_alt| - |e|) \end{aligned}$$

Anschließend muss noch darauf geachtet werden, dass sich die Geschwindigkeit im Intervall [0,$v_{min}$] befindet und dem ausgerechneten $v$ das passende Vorzeichen gegeben wird. $Kd_{stop}$ wurde hierbei wieder empirisch ermittelt.
Ist der Abstand zwischen Soll- und Ist-Wert gering genug, so wird in den HOLD-Zustand gewechselt. Wichtig für den HOLD-Regler war nun auch, dass das System nun nicht in einen Schwingungszustand gerät, falls die Geschwindigkeit doch noch zu hoch ist. Aus diesem Grund wurde der HOLD-Regler sehr stark schwingungs-entgegenwirkend implementiert, sodass dieser Fall nicht mehr eintreten kann. Dadurch ist er je nachdem zwar etwas langsamer, aber wie bereits erwähnt liegt die höchste Priorität darin, nicht über das Ziel hinaus zu fahren, was im Schwingfall passieren würde. Daher wird in Kauf genommen, dass das letzte Stück etwas langsamer angefahren wird.

## 6.4 Übersicht

In Abbildung 6.2 sind die drei Zustände in einem Zeit-Distanz-Diagramm eingezeichnet. Man erkennt, dass der STOPPAGE-Zustand nur sehr kurz bestehen bleibt und dennoch ist er der wichtigste Zustand mit der längsten Entwicklungszeit, da es in diesem darauf ankommt sowohl schnell als auch genau zu sein. Im Gegensatz dazu muss der Regler im HOLD-Zustand zwar auch sehr genau sein, aber nicht so schnell, da hier nur minimale Änderungen behoben werden müssen. Der Regler im RUN-Zustand muss zwar sehr schnell reagieren, um keine zu große Distanz aufzubauen, allerdings muss dieser dann auch nicht hundertprozentig genau regeln.


Abbildung 6.2: Zustände im Zeit-Distanz-Diagramm

# Kapitel 7

# Implementierung

Nachdem in den vorherigen Kapiteln bereits die Algorithmen zur Regelung und zur Interaktion mit der Hardware in der Theorie besprochen wurden, wird im folgenden Kapitel auf die genaue Implementation in C eingegangen.

## 7.1 Main

Das Main Modul (Anhang B.1) hat selber kaum Funktionalität, es dient vielmehr als Verbindungsglied der anderen drei Module, wie in Kapitel 5.1 beschrieben. In diesem Modul werden unter anderem die globalen Variablen deklariert, damit jedes Modul darauf Zugriff hat.

```
23 volatile signed short is, goal, v, diff, old_diff, distance, new_ticks_arrived;
24 volatile signed char state, direction, new_is;
25 volatile unsigned short c_v, tick_Period;
```

Quellcode 7.1: "main.c"

**is** und **goal** sind hierbei der Ist- und der Soll- Wert, **v** gibt die aktuelle Geschwindigkeit an, **diff** und **old_diff** sind die aktuelle und die alte Differenz zwischen dem Ist- und dem Soll-Wert, **distance** beschreibt die in Kapitel 6.2.3 beschrieben Distanz und in **new_ticks_arrived** ist gespeichert ob und wie viele Ticks seit dem letzten Schleifendurchlauf eingetroffen sind. Die Variable **state** beinhaltet den Wert der Zustandsautomaten und kann **HOLD**, **RUN** oder **STOPPAGE** annehmen. **direction** gibt an in welche Richtung der Motor drehen soll, **UP**=1 oder **DOWN**=-1. **c_v** ist eine Konstante zur Berechnung der Geschwindigkeit, die in Kapitel 6.2 genauer beschrieben wird, und in **tick_Period** wird die Zeit zwi-

schen zwei Ticks gespeichert. Da diese Variablen alle auch von den Interrupt Service Routinen benötigt werden, welchen man keine Parameter übergeben kann, ist es notwendig sie als globale Variablen zu deklarieren.
Außerdem befindet sich in dem Main-Modul die main-Methode, die so kurz wie möglich gehalten wurde.

```
28  int main(void)
29  {
30          initUART();
31          sei();
32          initMotor();
33          initCaliper();
34          initTicks();
35
36          while(1){
37
38                  switch (state){
39                          case STOPPAGE: stoppage(); break;
40                          case RUN: run(); break;
41                          case HOLD: hold(); break;
42                  }
43
44                  UART_Send_fast(goal-is);
45          }
46
47  }
```

Quellcode 7.2: "main.c"

Beim Starten werden lediglich die Initialisierungsmethoden der einzelnen Module aufgerufen. Danach startet die main-loop, in der abhängig vom Status der passende Regelalgorithmus aufgerufen wird (Zeile 38 - 42). Die Initialisierung des UART-Moduls und die Übermittlung der Differenz des Soll- und Ist-Wertes in Zeile 44 ist lediglich für die Kalibrierung und Validierung wichtig.

## 7.2 Messschieber

In diesem Kapitel geht es um das Auslesen der Messchieberdaten mittels dem Caliper-Modul, bestehend aus der *Caliper.h* (Anhang B.2) und *Caliper.c* (Anhang B.3).

### 7.2.1 Initialisierung

Wie in Kapitel 4.1.1 beschrieben, wird die Daten-Leitung des Messchiebers an Pin PA0 angeschlossen und die clock-Leitung an Pin PA1 sowie an Pin PD2, da hierbüber die externen Interrupts ausgelöst werden.

```
11 #define CLOCK    (~PINA & 0x02)
12 #define DATA     (PINA & 0x01)
```

Quellcode 7.3: "Caliper.h"

Da die Signale durch die Transistoren invertiert werden, müssten diese in der Software wieder zurück invertiert werden. Bei dem Datensignal kann man hierauf allerdings verzichten, weil dieses im Zweierkomplement übertragen wird. Da bei dem Einlesen der übertragenen Zahl die letzten Bits weggerundet werden (siehe Kapitel 4.1.2), reicht das Invertieren hier tatsächlich, um aus dem Zweierkomplement den richtigen Wert zu erlangen.

```
4  void initCaliper(){
5
6          DDRA=0x00;
7          PORTA=0xFF;
8
9          // erstes Packet Überspringen
10         while(!CLOCK);
11         _delay_ms(1);
12
13         // enable external Interrupt
14         MCUCR |=  (1<<ISC01);
15         GICR |= (1<< INT0);
16
17         // erster is Wert messen
18         for(int i = 0; i<30; i++)
19                 _delay_ms(1);
20
21 }
```

Quellcode 7.4: "Caliper.c"

Bei der Initialisierung werden zuerst die passenden Pins als Input-Pins gesetzt (Zeile 6,7). Da das Eintreffen eines Datenpaketes über den externen Interrupt an Pin INT0 (PD2) erkannt wird, wird dieser in Zeile 15 aktiviert. Das Signal liegt in invertierter Form am ATmega an, daher muss der Interrupt bei einer fallenden Flanke ausgelöst werden (Zeile 14). Bevor der Interrupt aber aktiviert wird, sollte noch gewährleistet sein, dass nicht genau in dem Moment ein Datenpaket an der Leitung anliegt und sofort ein Interrupt ausgelöst wird. Damit könnte der Controller nicht erkennen, an welchem Bit des Paketes er sich gerade befindet. Er würde daher trotzdem warten bis er alle 2 mal 24 Bits gehört hat und würde dadurch nicht nur falsche Werte auslesen sondern durch das aktive Warten alle anderen Methoden blockieren. Hier ist eine Synchronisation sehr wichtig. Um diese durchzuführen wartet man bis ein Paket ankommt. Dieses wird übersprungen, danach erst wird der Interrupt aktiviert. Da ein solches Datenpaket 850$\mu$s (s. Kapitel 4.1.2) dauert, wird nach dem Eintreffen eine Millisekunde gewartet (s.

Zeile 10,11). Sobald der Interrupt aktiv ist beginnt dieser, parallel zur Initialisierungsmethode (s. Kapitel 5.2), den Messschieber auszulesen. Daher wird in dieser noch 30ms gewartet (s. Zeile 18,19) bevor die Methode verlassen wird. Da die Schieblehrensignale mit 50Hz eintreffen, wird in dieser Zeit garantiert, dass ein Positions-Wert eingelesen wurde, welcher die Initialisierungsmethode des nächsten Moduls benötigt.
Die 30ms können hier nicht direkt als Parameter der Methode **_delay_ms()** übergeben werden, da man im Datenblatt den Maximalwert von

$$delay\_ms_{max} = \frac{262.14[ms]}{F\_CPU[MHz]} = \frac{262.14}{16} ms = 16.38ms$$

vorfindet [atm13]. Daher ist in Zeile 18, wie auch an anderen Stellen des Programms, eine separate for-Schleife nötig.

### 7.2.2 Messung trifft ein

Trifft ein Datenpaket einer Messung der Schieblehre ein, so wird am Pin INT0, an dem die clock-Leitung anliegt, eine fallende Flanke erkannt. Sofort wird ein Interrupt ausgelöst und der Program Counter springt in die passende Interrupt Service Routine (ISR).

```
24  ISR(INT0_vect)
25  {
26          // nur Interrupt INT0 sperren
27          GICR &= ~(1<< INT0);
28          sei();
29
30          short val = 0;
31
32          // 3Bits wegrunden
33          for(int i=0;i<3;i++)
34          {
35                  while(!CLOCK);
36                  while(CLOCK);
37                  //nop
38          }
39
40          // lese is-Wert
41          for(int i=0;i<16;i++)
42          {
43                  while(!CLOCK);
44                  while(CLOCK);
45                  val |= (DATA) << i;
46          }
47          is = val;
48          new_is = TRUE;
```

```
49
50          // 5 restliche Bits + 24 zweiter Block + 1 Stopbit
51          for(int i=0;i<30;i++)
52          {
53                  while(!CLOCK);
54                  while(CLOCK);
55                  //nop
56          }
57
58          // INT0 wieder freigeben
59          GIFR |= (1<< INTF0);
60          GICR |= (1<< INT0);
61  }
```

Quellcode 7.5: "Caliper.c"

Normalerweise werden andere Interrupts für den Zeitraum, indem sich das Programm in einer ISR befindet, deaktiviert. Lediglich der zuletzt aufgetretene Interrupt wird gespeichert und die Service Routine abgearbeitet sobald die aktuelle ISR zu Ende gelaufen ist. Da die ISR zum Auslesen eines Datenpaketes allerdings mit 850$\mu$s sehr lange dauert, würden dadurch sehr oft eintreffende Ticks, welche ebenfalls über Interrupts wahrgenommen werden, übersehen. Wie schon in Kapitel 5.2 beschrieben, ist es daher wichtig andere Interrupts trotzdem zuzulassen. Dies geschieht mit dem Befehl **sei()** (s. Zeile 28). Vorher müssen aber die INT0-Interrupts deaktiviert (s. Zeile 27) werden, damit diese ISR nicht ein zweites mal aufgerufen wird während sie schon aktiv ist. Dies würde normalerweise passieren, da bei jedem eintreffenden Bit eine fallende Flanke erkannt würde. Nun darf am Ende der ISR nicht vergessen werden diese wieder zu aktivieren. Zur Sicherheit werden eventuell doch gespeicherte INT0-Interrupts, die in der Zwischenzeit aufgetreten sind, gelöscht (s. Zeilen 59, 60).
Zum Auslesen der einzelnen Bits wird jeweils bei einer fallenden Flanke der clock-Leitung der Wert der Datenleitung ausgelesen. Das Warten auf die fallende Flanke geschieht hierbei aktiv mittels zweier while-Schleifen. Es werden allerdings die ersten drei Bits übersprungen (Zeile 32 - 38), die folgenden 16 Bits werden abgespeichert und als Ist-Wert gesetzt (Zeile 41 - 48). Die restlichen 30 Bits können wiederum übersprungen werden (Zeile 50 - 56). Wieso diese Bits nicht gespeichert werden müssen, wird in Kapitel 9.1 genauer erklärt.

## 7.3 Ticks

Für das Erfassen der vom Host eingehenden Ticks ist das Ticks-Modul, bestehend aus *Ticks.h* (Anhang B.4) und *Ticks.c* (Anhang B.5), zuständig.

### 7.3.1 Initialisierung

Sowohl die Erfassung der eingehenden Ticks, als auch das Erkennen, dass keine Ticks mehr gesendet werden, geschieht mittels Interrupt Routinen. Daher ist es zu Beginn notwendig diese zu initialisieren und zu aktivieren.

```
25 void initTicks();
26 ISR(INT1_vect);
27 ISR(TIMER1_OVF_vect);
28 ISR(TIMER1_COMPA_vect);
```

Quellcode 7.6: "Ticks.h"

Zuerst wird der externe Interrupt am Pin Int1 für die steigende Flanke, durch Setzen der Interrupt-Sense-Control-Bits (ISC) im MikroControllerUnit-Control-Register (MCUCR), initialisiert. Im General-Interrupt-Control-Register (GICR) wird dieser dann aktiviert (Zeile 21, 22). [atm13]
Der 16-Bit Timer1 ist dafür zuständig die Zeit zwischen den ankommenden Ticks zu messen. Initial wird hierfür sein Zähler (TCNT1) auf 0 gesetzt (Zeile 5) und der Prescaler wird eingestellt. Zur leichteren Konfiguration wird dies mittels einer switch-Anweisung über eine #define geregelt (Zeile 13 - 19). Hier wurde der Wert auf 64 festgelegt, wodurch der Konstante Teil der Gleichung aus Kapitel 6.2 noch in ein *unsigned short* passt. Somit kann auf teure long-Berechnungen verzichtet werden, aber gleichzeitig bleibt noch eine möglichst hohe Auflösung des Timers bestehen.

```
4  void initTicks(){
5
6          // zeitmessung zwischen Ticks mit Timer1
7          tick_Period = 0;
8          TCNT1 = 0;
9
10         new_ticks_arrived = FALSE;
11         goal = is;
12
13         switch(TIMER1_PRESCALER){
14                 case 1: TCCR1B |= (1<<CS10); break;
15                 case 8: TCCR1B |= (1<<CS11); break;
16                 case 64: TCCR1B |= ((1<<CS11)|(1<<CS10)); break;
17                 case 256: TCCR1B |= (1<<CS12); break;
18                 case 1024: TCCR1B |= ((1<<CS12)|(1<<CS10)); break;
19         }
20
21         MCUCR |= (1<<ISC11)|(1<<ISC10);  // The rising edge of INT1
22         GICR |= (1<< INT1);                      // enable external interrupt 1
23
24         TIMSK |= ((1<<TOIE1)     // TimerOverflowInterruptEnable
25                         |(1<<OCIE1A)); // OnCompareInterruptEnable
```

Quellcode 7.7: "Ticks.c"

Der Timer wird nicht nur für die Zeitmessung zwischen den Ticks für die Geschwindigkeitsberechnung benötigt, sondern auch um zu Erkennen, ob längere Zeit keine Ticks mehr angekommen sind und somit vom Zustand RUN in den Zustand STOPPING gewechselt werden soll. Hierfür werden die zwei Interrupts Timer-Overflow- und On-Compare-Interrupt aktiviert (Zeile 24, 25).
Außerdem wird in der Initialisierung noch der Soll-Wert mit dem Ist-Wert gleichgesetzt. Der Ist-Wert ist die Position an der die Linearführung startet und das Gleichsetzen des Soll-Wertes sorgt dafür, dass diese Position gehalten wird, solange keine weiteren Befehle eintreffen.

### 7.3.2 Tickerfassung

Erreicht ein Tick den Mikrocontroller, so sorgt dieser für eine steigende Flanke am Int1-Pin, welche einen Interrupt auslöst und das Programm in die INT1-ISR leitet.
Da dieser Interrupt auch während dem Auslesen der Schieblehre passieren kann, muss die Service-Routine kürzer sein als ein Bit der Messung, damit kein Bit „verschluckt“ wird. Daher wurde hier, sowie in den beiden ISR aus Kapitel 7.3.3, darauf geachtet, die Routinen möglichst kurz zu halten.

```
30  // tick kommt an
31  ISR(INT1_vect)
32  {
33          // TickTimer auslesen
34          tick_Period = TCNT1;
35          TCNT1 = 0;
36
37          OCR1A = tick_Period+C_STOP;
38
39          if(PINA & 0x04)
40                  new_ticks_arrived--;
41          else
42                  new_ticks_arrived++;
43  }
```

Quellcode 7.8: "Ticks.c"

Wird die ISR ausgelöst, so wird eingangs der Timer1 ausgelesen und für die nächste Zeitmessung wieder auf 0 gesetzt, damit die gemessene Zeit möglichst genau ist. Diese Zeit wird für die Geschwindigkeitsberechnung benötigt. Die maximale Frequenz mit der die Ticks ankommen ist 500Hz, dies bedeutet, dass die $tick\_Period$ einen minimalen Wert von $tick\_Period = \frac{F\_CPU[Hz]/Prescaler}{TickFrequenz[Hz]} = \frac{16000000Hz/64}{500Hz} = 500$ erreicht. Dies lässt erkennen, dass die Timerauflösung hoch

genug ist. Andererseits ist die minimale Frequenz, bis zu der der Timer mitzählen kann, ohne dass ein Overflow-Interrupt ausgelöst wird $f = \frac{F\_CPU[Hz]/Prescaler}{timer.maxvalue} = \frac{16000000/64}{2^{16}} = 3,8147Hz$. Dies würde eine Geschwindigkeit von circa 0,038 mm/s bedeuten, mit der nicht zu rechnen ist. Falls trotzdem mit einer solch langsamen Geschwindigkeit gefahren wird, so würde der Overflow-Interrupt dafür sorgen, dass in den HOLD-Zustand gewechselt wird, für den eine solch geringe Geschwindigkeit kein Problem darstellt.
Nachdem die $tick\_period$ gemessen wurde, wird auf diese eine Konstante (hier 100) addiert und dies als Output-Compare-Register (OCR)-Wert des Timer1 genommen. Diese Konstante bestimmt den Zeitraum, der nach dem letzten Tick gewartet wird, bevor mit dem Ende einer Ticksequenz gerechnet wird ( vgl. Kapitel 7.3.3)
Bevor die Interrupt Service Routine wieder verlassen wird, muss noch kontrolliert werden, ob der angekommene Tick "positiv" oder "negativ" ist, er also eine Sollwertverschiebung nach vorne oder nach hinten bedeutet. Dies erkennt man an PINA3. Liegen hier 5 Volt an, so ist der Tick negativ, andernfalls ist er positiv (Zeile 39 - 42). Diese Erkenntnis wird nun allerdings nicht direkt in der Sollwertvariable ($goal$) vermerkt, sondern in $new\_ticks\_arrived$ zwischengespeichert, da an diesem Wert erkannt werden soll, ob die Geschwindigkeitsberechnung neu durchgeführt werden muss.

### 7.3.3 Endpunkt einer Strecke

Da es kein Signal gibt, das signalisiert, ob die Linearführung nach einer gefahrenen Strecke stoppen soll, kann dies nur daran erkannt werden, dass keine weiteren Ticks mehr ankommen. Um dies möglichst schnell zu realisieren wird angenommen, dass zwischen einem Tick $T_n$ und dem nächsten Tick $T_{n+1}$ genauso viel Zeit liegt, wie zwischen letzten Ticke $T_n$ und dem Tick $T_{n-1}$ davor, plus einer gewissen Toleranz $delta$. Es wird also davon ausgegangen, dass sich die vorgegebene Geschwindigkeit nur langsam ändert. Ist nach einem Tick bereits die Zeit $Zeit(T_n) - Zeit(T_{n-1}) + delta$ vergangen, ohne dass ein weiterer Tick eingetroffen ist, so wird vermutet, dass auch in naher Zukunft kein weiterer Tick eintritt. Anschließend wird in den STOPPAGE-Zustand gewechselt, um auf dem Sollwert stehen zu bleiben. Die Zeit $t(T_n) - t(T_{n-1})$ steht in der $tick\_Period$, welche addiert mit dem $delta$ ( =: $C\_STOP$) als Compare-Wert des OnCompare-

Interrupts genommen wird (vgl. Quellcode 7.8, Zeile 37). In diesem Fall wurde für das $delta$ der Wert 100 ermittelt, wodurch eine Toleranz von $C_STOP * fracPrescalerF\_CPU[Hz] = 100 * \frac{64}{16000000Hz} = 4 * 10^{-5}s$ zustande kommt, bevor der OnCompare-Interrupt (Zeile 46 - 55) ausgelöst wird und damit in den STOPPAGE-Zustand gewechselt wird.

```
46  ISR(TIMER1_OVF_vect)
47  {
48          if(state == RUN)
49          {
50                  stop();
51                  state = STOPPAGE;
52                  distance = 0;
53          }
54
55  }
56
57  ISR(TIMER1_COMPA_vect)
58  {
59          if(state == RUN)
60          {
61                  stop();
62                  state = STOPPAGE;
63                  distance = 0;
64          }
65  }
```

Quellcode 7.9: "Ticks.c"

Neben dem OnCompare-Interrupt ist außerdem noch der Overflow-Interrupt aktiviert, welcher ausgelöst wird, sobald der Timer1 ”überläuft”, wie in Kapitel 7.3.2 bereits erwähnt. Der Inhalt dieser Routine (Zeile 57 - 65) ist der gleiche wie in der OnCompare-Routine. Es wurde hierfür allerdings keine separate Methode geschrieben, die in beiden aufgerufen würde, obwohl es Programmierrichtlinien gebieten. Hier wurde eher darauf geachtet die ISRs sehr kurz zu halten, wie in Kapitel 7.3.2 bereits erklärt, und daher auf einen weiteren Methodenaufruf verzichtet.
Da die Interrupts ständig ausgelöst werden, wenn sich das System im HOLD-Zustand befindet und keine Ticks ankommen, wird hier darauf geachtet, dass die Funktionalität nur ausgelöst wird, wenn sich das System im RUN-Zustand befindet (Zeile 48 und 59).

## 7.4 Motorsteuerung

Die Ansteuerung des Gleichstrommotors mittels PWM sowie die Regelung übernimmt das Motor-Modul, bestehend aus *Motor.h* (Anhang B.6) und *Motor.c* (Anhang B.7).
Das Modul besteht neben der Initialisierung und der direkten Motoransteuerung hauptsächlich aus drei Regelalgorithmen, je einen pro Zustand, wobei sich die Regelung im RUN-Zustand aufteilt in die Grundgeschwindigkeitsberechnung und die eigentliche Regelung.

### 7.4.1 Initialisierung

Wie in Kapitel 6.2.1 beschrieben, besteht die Geschwindigkeitsberechnung aus einer Konstanten multipliziert mit der Zeit zwischen den Ticks. Da die Berechnung (Zeile 6) dieser Konstante ($c\_v$) recht aufwändig und zeitintensiv ist, wird diese nur einmal initial berechnet und abgespeichert, sodass die Methode zu Grundgeschwindigkeitsberechnung lediglich diesen Wert auslesen muss. Da eine Berechnung mit long- oder float Werten auf dem ATmega sehr viel länger dauert als eine short-Berechnung, wird hier ebenfalls darauf geachtet, dass der Wert in eine unsigned short-Variable passt, er also zwischen 0 und $2^{16} - 1 = 65536$ liegt. Da der errechnete Wert allerdings 80330 beträgt, wird dieser halbiert, um ihn abspeichern zu können, und dort wo er genutzt wird, wird er wieder verdoppelt.

```
4  void initMotor(){
5
6          c_v = (unsigned short)((float)VMAX*F_CPU/2/(V_MAX_INCHPERSEC * TIMER1_PRESCALER * TICKSPERINCH));
7
8          // Outout
9          DDRC = 0xFF;
10         DDRD |= 0x80;
11
12         // PWM mit Timer2
13         TCCR2 = (1<<WGM20)              //      Phase Correct
14                         |(1<<COM21)      //      nicht invertiert
15                         |(1<<CS22);      //      prescaler 32 -> 1/4MHz
16
17         //Initialwerte festlegen
18         diff = 0;
19         old_diff = 0;
20         v = 0;
21         drive(0);
22         distance = 0;
23         state = HOLD;
24 }
```

Quellcode 7.10: "Motor.c"

Ebenso werden in der init-Methode die Pins (PC0 ,PD7), die zum Motor führen, als Ausgangspins deklariert (Zeilen 9,10) sowie die Pulsweite zur Motorsteuerung darauf eingestellt, dass sie phasenkorrekt, nicht invertiert und mit einem Prescaler von 32, also mit $F\_CPU/Prescaler = 16MHz/32 = 500kHz$, läuft (Zeile 12 - 15). Dieser Wert ist so gewählt, dass die Frequenz gering genug für den hier etwas zu langsamen Optokoppler (vgl. Kapitel 4.2.2.1) aber trotzdem möglichst hoch ist, um eine flüssige Motorsteuerung zu gewährleisten.
Anschließend werden noch Variablen initialisiert, die für die Regelung wichtig sind, und der Zustand auf HOLD gesetzt.

### 7.4.2 Motoransteuerung

Der Motor wird über die beiden Pins PD7 und PC0 gesteuert, die über Optokoppler und Treiberbaustein direkt mit den Anschlüssen des Motors verbunden sind (siehe Kapitel 4.2). Zum Vorwärtsfahren bleibt PC0 auf low-Pegel, dieser Anschluss dient nun als Masse-Pol des Motors. Mit dem anderen Pin PD7 wird nun geregelt, ob sich der Motor dreht oder nicht, beziehungsweise regelt die Pulsweite, die an PD7 anliegt, die Drehgeschwindigkeit. Um eine Drehung in die entgegengesetzte Richtung zu bewirken, wird an dem Pin PC0 ein high-Pegel angelegt und die Pulsweitenmodulation invertiert. Dies bewirkt, dass im Stillstandsfall beide Anschlüsse high-Pegel haben, es also keine Spannungsdifferenz gibt und somit der Motor stehen bleibt. Wird an PD7 und damit auch an dem zugehörigen Motoranschluss ein low-Pegel angelegt, so besteht wiederum eine Spannungsdifferenz, allerdings mit umgekehrten Vorzeichen, der Motor dreht also andersrum. Genau dieses Verhalten wurde in den beiden Methoden *forwards* (ab Zeile 27) und *backwards* (ab Zeile 34) implementiert, wobei $OCR2$ das Register ist, welches die Pulsweitenmodulation regelt. Im $OCR2$-Register steht die Anzahl der high-Pegel Anteile von insgesamt 255 (da 8-Bit PWM) Teilen, das heißt der Pin ist zu $\frac{OCR2}{255}$stel der Zeit high. Das Signal zu invertieren bedeutet high- und low-Pegelzeiten zu tauschen. Hierzu genügt es, den Wert (*speed*) vom Maximalwert (*255*) abzuziehen, da gilt : $1 - \frac{speed}{255} = \frac{255}{255} - \frac{speed}{255} = \frac{255 - speed}{255}$.

```
26 // faehrt mit Geschwindigkeit(speed [0,255]) vorwaerts
27 void forwards(unsigned char speed)
28 {
29         PORTC &= 0xfe;
30         OCR2 = speed;
31 }
32
33 // faehrt mit Geschwindigkeit(speed [0,255]) rueckwaerts
34 void backwards(unsigned short speed)
35 {
36         PORTC |= 0x01;
37         OCR2 = 255-speed;
38 }
```

Quellcode 7.11: "Motor.c"

Da es für die Regelalgorithmen aufwändig ist zwischen Vorwärts und Rückwärts zu unterscheiden, sie aber ohne Probleme im negativen Bereich arbeiten können, ist es noch wünschenswert eine Wrapper-Methode (*drive*, ab Zeile 41) zur Motoransteuerung zu haben, welche Werte von -255 bis +255 entgegennimmt und diese passend an *forwards* und *backwarts* weiterleitet. Hier wird außerdem darauf geachtet, dass der Eingangswert im Betrag nicht größer 255 ist, und beim Rückwärtsfahren wird noch eine Konstante $C\_DIFF$ addiert. Dies ist notwendig, da der Motor in diese Richtung langsamer dreht als in die andere Richtung. Bedingt ist dies einerseits durch die Optokopplerverzerrung (Kapitel 4.2.2.1), andererseits haben Tests aber auch gezeigt, dass der hier verwendete Motor bei gleicher Spannung abhängig von der Drehrichtung unterschiedliche Geschwindigkeiten hat.

```
40 // fährt mit Geschwindigkeit(speed [-255,255])
41 void drive(signed short speed)
42 {
43         if(speed >= 0)
44                 forwards( (speed>=VMAX) ? VMAX : (unsigned char)speed );
45         else{
46                 speed = abs(speed);
47                 speed = (speed <= C_DIFF) ? 0 : ((speed >= VMAX+C_DIFF) ? VMAX : speed - C_DIFF);
48                 backwards(speed);
49         }
50 }
```

Quellcode 7.12: "Motor.c"

Um den Motor anzuhalten würde es genügen $drive(0)$ auszuführen. Da es in diesem Zusammenhang aber auch immer erwünscht ist, danach im HOLD-Zustand zu sein, und man die Grundgeschwindigkeit auf 0 setzen möchte, wurde hierfür noch eine separate Methode realisiert: $stop()$ (ab Zeile 53).

```
52 // Anhalten
53 void stop()
54 {
55         PORTC = 0x00;
56         OCR2 = 0;
57         v = 0;
58         state = HOLD;
59 }
```

Quellcode 7.13: "Motor.c"

### 7.4.3 HOLD-Regler

Wie in Kapitel 6.1 zum HOLD-Regelalgorithmus beschrieben, führt dieser Regler kurze Impulse in Sollwert($goal$)-Richtung aus, falls sich die Schieblehre nicht auf der gewünschten Position befindet (Zeile 132, 134).
Die Länge des Impulses sowie die Länge der Zeit zwischen den Impulsen wird hierbei über die Konstanten $C\_HOLD\_DRIVE$ und $C\_HOLD\_STOP$ geregelt, welche die Werte 2 und 30 haben und damit die Millisekundenanzahl angeben. Wie in Kapitel 7.2.1 bereits erklärt, ist auch hier eine for-Schleife (Zeile 145, 146) nötig, da $\_delay\_ms()$ lediglich maximal 16ms verzögert. [atm13]
Befindet sich die Schieblehre auf der richtigen Position, so sorgt dieser Algorithmus lediglich dafür, dass der Motor stehen bleibt (Zeile 149).

```
129 void hold()
130 {
131
132         diff = (goal-is);
133
134         if(diff != 0)
135         {
136                 // kurzer Impuls
137                 if(is < goal)
138                         drive(VMAX);
139                 else
140                         drive(-VMAX);
141                 _delay_ms(C_HOLD_DRIVE);
142
143                 // stop
144                 forwards(0);
145                 for(int i = 0; i< C_HOLD_STOP; i++)
146                         _delay_ms(1);
147         }
148         // steht richtig
149         else forwards(0);
150
151 }
```

Quellcode 7.14: "Motor.c"

### 7.4.4 RUN-Regler

Die $run$-Methode realisiert den Regler, welcher in Kapitel 6.2 beschrieben wurde.

#### 7.4.4.1 Regelung

Da die Berechnung der Grundgeschwindigkeit lediglich von der Zeit zwischen den Ticks abhängt, lohnt es nur diese Berechnung auszuführen, wenn neue Ticks angekommen sind (Zeile 156, 157). Wie genau diese Berechnung abläuft, wird im nächsten Kapitel beschrieben.

```
154 void run()
155 {
156         if(new_ticks_arrived != 0)
157                 calculate_v();
158
159         old_diff = diff;
160         diff = (goal-is);
161
162         float v_regler;
163         v_regler = C_P * (diff-distance);       // P-Regler
164         v_regler += C_D * (diff - old_diff);// D-Regler
165
166         drive(v + (short)v_regler);
167
168 }
```

Quellcode 7.15: "Motor.c"

Wurde die Grundgeschwindigkeit errechnet, so wird auf diese der Regel-Wert $v\_regler$ addiert. Dieser Wert berechnet sich exakt so, wie in den Kapiteln 6.2.2 und 6.2.3 beschrieben.

#### 7.4.4.2 Grundgeschwindigkeitsberechnung

Da die Methode zur Grundgeschwindigkeitsberechnung immer dann aufgerufen wird, wenn neue Ticks angekommen sind, ist es ihre erste Aufgabe, diese Ticks-Anzahl mit dem Sollwert zu verrechnen und festzulegen in welche Richtung gefahren werden muss (Zeile 65 - 68).
Befindet sich das System nun noch im HOLD-Zustand, so wird die Geschwindigkeit auf den höchsten Wert gesetzt, um der Trägheit des Motors entgegenzuwirken (Zeile 71 - 75), wie bereits in Kapitel 6.2.3 beschrieben. Nun kann außerdem noch keine Geschwindigkeit berechnet werden, da diese abhängig ist vom zeitlichen Abstand zweier Ticks und es in diesem Fall noch keinen Vorgängertick gibt.

```
63  void calculate_v()
64  {
65          // goal anpassen
66          direction = (new_ticks_arrived > 0) ? UP : DOWN;
67          goal += new_ticks_arrived;
68          new_ticks_arrived = 0;
69
70          // erster Tick
71          if(state != RUN)
72          {
73                  v = VMAX * direction;
74                  state = RUN;
75          }
76          // sonstige ticks
77          else {
78                  short v2 = (c_v/tick_Period*2);
79
80                  if(v2<VMIN)
81                          state = HOLD;
82                  else
83                          v = v2 * direction;
84          }
85
86          // Distanz anpassen
87          distance = (v / C_DISTANCE);
88          old_diff = 0;
89  }
```

Quellcode 7.16: "Motor.c"

Ist dies allerdings nicht der erste Tick, so kann gemäß Kapitel 6.2.1 die Grundgeschwindigkeit berechnet werden. Bei der Berechnung darf nicht vergessen werden die Geschwindigkeit zu verdoppeln, da $c\_v$ nur halb so groß ist, wie es eigentlich sein sollte (vgl. Kapitel 7.4.1). Nun wird noch getestet ob die errechnete Geschwindigkeit nicht geringer ist, als der Minimalwert. Unter diesem würde sich der Motor nicht bewegen, weil die Spannung zu gering wäre. In diesem Fall würde der RUN-Regler nicht funktionieren und der HOLD-Regler wird stattdessen eingesetzt (Zeilen 80, 81).
Abschließend wird noch die Distanz aus Kapitel 6.2.3 an die neue Grundgeschwindigkeit angepasst (Zeile 87) und die alte Differenz zwischen Ist- und Soll-wert nullgesetzt, da diese abhängig von der Distanz und somit nicht mehr gültig ist.

### 7.4.5 STOPPAGE-Regler

Der STOPPAGE-Regler erkennt seinen ersten Schleifendurchlauf daran, dass die $direction$ noch gesetzt ist. Daraufhin gibt er den kurzen starken Impuls entgegen der Fahrtrichtung, wie in Kapitel 6.3.1 beschrieben, um schon mal etwas Schwung aus der Linearführung zu nehmen (Zeile 94 - 100).

```
92  void stoppage()
93  {
94          // starker Gegenimpuls am Anfang
95          if(direction != 0){
96                  drive((signed short)-VMAX * direction);
97                  _delay_ms(C_STOP_IMPULS);
98                  forwards(0);
99                  direction = 0;
100         }
101
102         // alte Differenz
103         old_diff = diff;
104         while(new_is==FALSE);
105         new_is = FALSE;
106         diff = (goal-is);
107
108         // am Ziel angekommen -> HOLD
109         if((abs(diff) <= C_STOP_DISTANCE))
110                 stop();
111         // ... noch nicht -> Regler
112         else {
113                 signed short v_regler = C_BRAKE * (abs(old_diff) - abs(diff));
114
115                 if (v_regler > VMIN)
116                         v_regler = VMIN;
117                 else if (v_regler < 0)
118                         v_regler = 0;
119
120                 if(is < goal)
121                         drive((signed short)(VMIN) - v_regler);
122                 else
123                         drive(v_regler - VMIN);
124         }
125
126 }
```

Quellcode 7.17: "Motor.c"

Im Gegensatz zu den anderen Regelalgorithmen ist dieser nur dann sinnvoll, wenn ein Unterschied zwischen der alten und der neuen Differenz zwischen Soll- und Ist-Wert besteht. Daher muss dieser Regler warten bis ein neuer Ist-Wert ermittelt wurde, da ansonsten $old\_diff$ und $diff$ gleich wären (Zeile 102 - 106).
Ist nun die Differenz zwischen $goal$ und $is$ geringer als ein gewisser Schwellwert $C\_STOP\_DISTANCE$ (hier 3, also $\frac{3}{TICKSPERINCH} = \frac{3}{2560} inch \approx 0.03mm$), sich die Linearführung also sehr nah an ihrem Ziel befindet, so wird in den HOLD-Zustand gewechselt.
Ist dies noch nicht der Fall so wird der Regler gemäß Kapitel 6.3.2 angewendet (Zeile 113 - 123). Hierbei wird zusätzlich darauf geachtet, dass $v\_regler$ im Bereich zwischen 0 und VMIN liegt, um Nebeneffekte zu vermeiden.

# Kapitel 8

# Portabilität und Flexibilität

Das System wurde darauf ausgelegt, dass es nicht nur für diesen einen konkreten Versuchsaufbau brauchbar ist. Man soll die Schieblehre, den Motor oder das Protokoll zum Host ohne großen Aufwand austauschen können.

## 8.1 Hostwechsel

Als Eingangssignal vom Host wird ein schrittmotorähnliches Ansteuerungssignal erwartet. Das Einzige was sich hier ändern könnte, wäre die Größe der Schritte. Dies würde für den Regler bedeuten, dass sich die Sollwertverschiebung pro Tick ändert. Der Wert der sich hier ändern würde, ist die Konstante $TICKSPERINCH$, welche in *Motor.h* definiert wird. In dieser steht die Schrittgröße in $[\frac{1}{Inch}]$. Ändert man diese, so muss darauf geachtet werden, dass der Wert bei der $c\_v$ Berechnung (Quellcode 7.10, Zeile 6) weiterhin in eine short-Variable passt. Gegebenenfalls muss hier das Teilen durch Zwei modifiziert werden.

## 8.2 Schieblehrenwechsel

Wird die Schieblehre ausgetauscht, so ändert sich sehr wahrscheinlich auch das Protokoll zu Datenübertragung. In diesem Fall müsste die komplette Auslesemethode (Quellcode 7.5) neu geschrieben werden. Außerdem müsste in der Initialisierung darauf geachtet werden, ob es genügt 1ms und später 30ms zu warten, um ein Datenpaket zu überspringen, beziehungsweise auf eins zu warten (Quell-

code 7.4).
Ändert sich das Protokoll insofern, dass sich die Bitfrequenz enorm verringert, so ist außerdem zu überlegen den alternativen Auslesealgorithmus zu nutzen, der in Kapitel 5.2.1 beschrieben wird.

## 8.3 Motorenwechsel

Soll der Motor gewechselt werden, so muss kein neuer Quellcode geschrieben werden wie bei dem Schieblehrenaustausch, es müssen lediglich einige Konstanten angepasst werden, vorausgesetzt es handelt sich wieder um einen Gleichstrommotor. Dennoch wird ein Motorwechsel komplizierter, da viele Parameter empirisch ermittelt und aufeinander abgestimmt werden müssen.
Um diese Parameter zu ermitteln, benötigt man ein UART-Terminal mit integriertem Plotter, ähnlich dem, welcher die Grafiken wie zum Beispiel Abbildung 6.1 erstellt hat. Auf diesem lässt man sich die Differenz zwischen Soll- und Ist-Wert anzeigen, indem man diesen in der main-loop mittels der $UART\_Send\_fast$-Methode versendet, wie im Quellcode 7.2 Zeile 44 zu erkennen ist.
Zur Bestimmung von $V\_MAX\_INCHPERSEC$ und $C\_DIFF$ deaktiviert man zuerst alle Regelalgorithmen. Als Startwert für $V\_MAX\_INCHPERSEC$ nimmt man die maximale Geschwindigkeit des Linearantriebes, $C\_DIFF$ setzt man auf 0. Nun lässt man eine Testsequenz von Ticks mit konstanter Frequenz und positivem Vorzeichen ablaufen (z.B. 500 Ticks mit 250 Hz). Auf der UART-Ausgabe erkennt man nun wie sehr die Linearführung „hinterherhinkt“. Ist $V\_MAX\_INCHPERSEC$ zu klein gewählt, so wird der Linearaktor diesen Abstand durch zu schnelles Fahren wieder aufholen und ggf. sogar den Sollwert „überholen“. Ist der Wert jedoch zu groß gewählt, so wird sich die Soll-Ist-Differenz immer weiter vergrößern. Mit diesem Wissen muss nun $V\_MAX\_INCHPERSEC$ eingestellt werden, sodass die Differenz nach einer kurzen Startphase einen konstanten Wert beibehält. Hierbei ist es egal wie groß die Differenz ist.
$C\_DIFF$ gleicht den Geschwindigkeitsunterschied des Fahrens in entgegengesetzter Richtung bei betragsmäßig gleicher Ansteuerung aus. Um diesen Wert zu ermitteln wendet man die gleiche Testfrequenz an, allerdings mit negativem Vorzeichen. Im Idealfall sollte auf dem Terminal dasselbe Bild zu sehen sein, an der x-Achse gespiegelt. Sollte dies nicht der Fall sein, so variiert man $C\_DIFF$, sodass nach kurzer Startphase wieder eine konstante Differenz zu sehen ist.

Nun können nacheinander die Regelalgorithmen wieder zugeschaltet werden und mittels der Testfrequenz kontrollieren, ob an den jeweiligen Parametern, welche in Kapitel 6 beschrieben wurden, etwas verändert werden muss. Sie sollten jedoch relativ allgemeingültig sein und bedürfen lediglich kleinen Veränderungen.
Da die Regelalgorithmen speziell für Gleichstrommotoren entwickelt wurden, ließe sich hier ohne komplett neue Regler keine andere Motorenart einsetzen.

# Kapitel 9

# Auswertung

Ziel des Projektes ist es den Schrittmotor eines Linearantriebes durch einen geregelten Gleichstrommotor zu ersetzen und damit das Spiel aus der Linearführung zu kompensieren. Zu Kontrollieren ist nun, ob das Spiel wirklich ausgeglichen wird (Kapitel 9.3), ob sich die Soll-Ist-Differenz während einer Teststrecke nicht wesentlich erhöht (Kapitel 9.2) und ob die gewünschte Genauigkeit von 0,01 mm eingehalten werden kann (Kapitel 9.1). Außerdem sollte beachtet werden wie sich die Maximalgeschwindigkeit des Linearaktors verändert (Kapitel 9.4).
Um dies möglichst einfach zu beobachten wurde ein kleines Java-Programm entwickelt, welches per UART Werte entgegennimmt und diese in Echtzeit plottet. Ein Diagramm dieses Plotters ist zum Beispiel Abbildung 6.1 oder 9.3. Mit diesem Programm können viele Werte kontrolliert und deren zeitlichen Verlauf beobachtet werden. Am sinnvollsten ist hier die Differenz zwischen Soll- ($goal$) und Ist- ($is$) Wert. Da diese Werte in jedem Schleifendurchlauf gesendet werden, bedeutet dies, dass die Skalierung der Zeit-Achse (x) nicht konstant ist, weil die Schleifendurchlaufsdauer je nach Zustand variiert.

## 9.1 Auflösung

Der alte Schrittmotor hatte 100 Schritte pro Umdrehung und die benutzte Gewindestange hat eine Umdrehung pro Millimeter. Daraus ergab sich also eine Genauigkeit von $\frac{1}{100}mm$, welche mit dem neuen System möglichst eingehalten werden soll. Wie bereits in Kapitel 4.1.2 beschrieben übermittelt das Schieblehrenprotokoll einen Werte mit einer Auflösung von 1/20480 Zoll, welche aber aufgrund

des Flackerns der letzten Bits auf 1/2560 Zoll = $\frac{1}{100,787}mm$ verringert wurde. In dieser Einheit rechnet das System nun auch weiter, da das Umrechnen in eine SI-Einheit eine aufwändige zeitintensive Gleitkommarechnung wäre, die man so einsparen kann. Daher ist es allerdings auch notwendig den Host so einzustellen, dass ein Tick eine Bewegung um $\frac{1}{100,787}mm$ bedeutet. Die gewünschte Auflösung von 0.01 mm wird sogar minimal überboten wird.
Durch das Rechnen in einer solch kleinen Einheit, könnte es passieren, dass die Größe der Variablenspeicher nicht ausreicht, um die komplette Teststrecke abzudecken. Die Soll- und Ist-Werte (*goal*, *is*) werden als signed short Typen gespeichert. Sie decken also eine Strecke von $-2^{15} * \frac{1}{2560}$ Inch = -12,8 Inch bis +12.8 Inch ab. Da die Linearführung eine Gesamtlänge von 8 Inch hat, kommt die Speichergröße hier auch im schlimmsten Fall nicht an ihre Grenzen.
Man könnte die Auflösung noch etwas erhöhen, indem man, statt dem Weglassen der letzten drei Bits, ein Mittelwert über diese ermittelt. Dies hätte aber die Nachteile, dass eine gewisse unerwünschte Trägheit in das System gebracht würde, dass die Soll- und Istwerte nun länger als 16 Bit wären und dass in jedem Schleifendurchlauf teure Gleitkommaberechnungen ausgeführt werden müssten, wodurch die Performance merkbar leiden würde. Außerdem wäre dies nur sinnvoll, solange die Linearführung still steht. Sobald sie eine Geschwindigkeit von mehr als einem Hundertstelmillimeter pro Messung erreicht, würde die Mittelwertsberechnung keinen Vorteil mehr liefern. Aus diesem Grund wurde hier die einfachere und ausreichende Variante gewählt, die letzten drei Bits einfach abzuschneiden.

## 9.2 Soll-Ist-Differenz

Um das alte System mit dem neuen System zu vergleichen, wurde wieder ein Schrittmotor eingesetzt, wodurch die Elektronik am Treiberbaustein sowie der Code leicht geändert werden musste. Die geänderten Software-Module Ticks, Motor und main befinden sich im Anhang C.
Vergleicht man nun das Diagramm des UART-Terminals einer Testsequenz auf dem alten System mit ungeregeltem Schrittmotor (Abbildung 9.1) mit dem Diagramm, welches mit selber Testsequenz auf dem neuen System mit geregeltem Gleichstrommmotor (Abbildung 9.2) aufgenommen wurde, so erkennt man auf dem ersten Blick eine deutliche Verschlechterung.

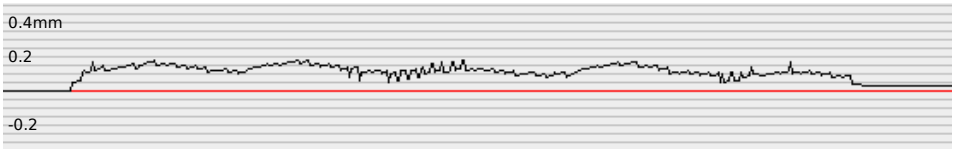


Abbildung 9.1: altes System (Schrittmotor)

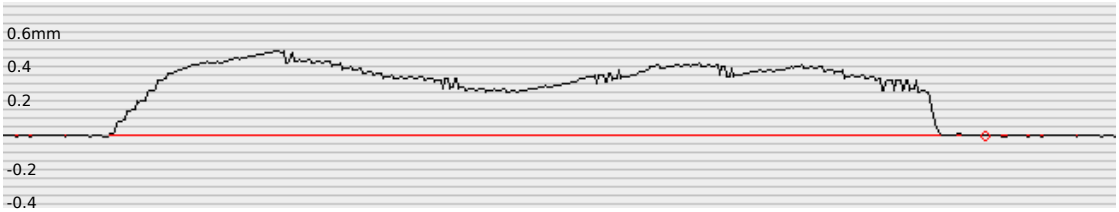


Abbildung 9.2: neues System (Gleichstrommotor)

Während der Fahrt ist die Differenz zwischen Soll- und Ist-Wert wesentlich größer, was bedeutet, dass das neue System stärker hinterher hängt. Dies liegt daran, dass ein Gleichstrommotor träger ist als ein Schrittmotor und nur relativ langsam anfährt. In diesem Beispiel ist die Distanz nie größer als $d = 50 * \frac{1}{100.878}mm$ bei einer Tickfrequenz von 250Hz, welche zu einer Geschwindigkeit von $v = f * \delta s = 250Hz * \frac{1}{100.878}mm$ führt. Hieraus ergibt sich eine zeitliche Verzögerung von $t = d/v = \frac{1}{5}Sekunde$, die die Linearführung hinter dem Sollwert hinterher liegt. Dies sind durchaus akzeptable Werte, die toleriert werden können, auch wenn es in dieser Hinsicht dem alten System gegenüber eine kleine Verschlechterung ist.

## 9.3 Spielausgleich

Betrachtet man ein Diagramm des Plotters mit angeschlossenem ungeregeltem Schrittmotor, so erkennt man, dass die Führung nach einer gefahrenen Strecke oft nicht genau den Sollwert trifft. Ersichtlich wird dies daran, dass die Soll-Ist-Differenz nach der Testsequenz nicht mehr auf Null zurück geht.

Ist der Linearaktor vorher in die selbe Richtung gefahren wie bei der Teststrecke, so ist die neu entstandene Abweichung nicht sehr groß (Abbildung 9.3).

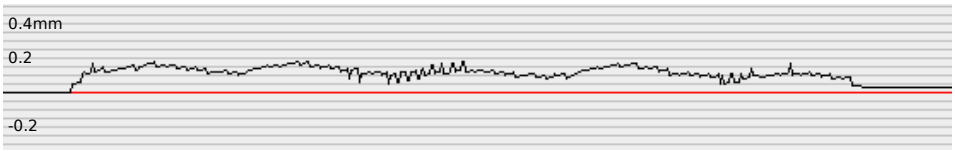


Abbildung 9.3: Schrittmotor, ohne Richtungswechsel

Fuhr die Schieblehre zuvor allerdings in die entgegengesetzte Richtung, so ist eine starke Abweichung zu erkennen (Abbildung 9.4).

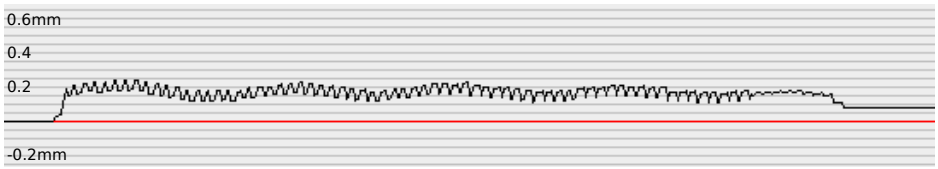


Abbildung 9.4: Schrittmotor, mit Richtungswechsel, langsam

Erhöht man nun noch die Frequenz der Ticks und damit die Geschwindigkeit, so kann die Abweichung eine inakzeptable Größe erlangen (Abbildung 9.5).

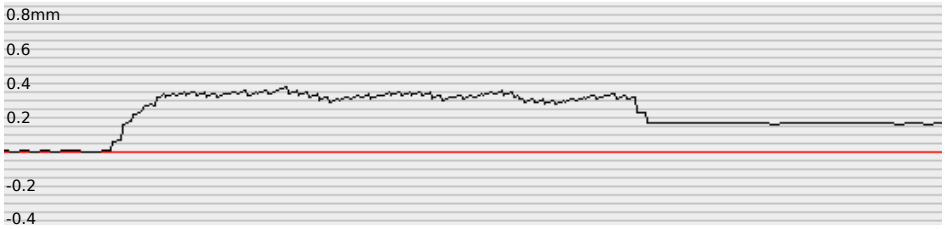


Abbildung 9.5: Schrittmotor, mit Richtungswechsel, schnell

Betrachtet man dies nun mit dem neuen System mit geregeltem Gleichstrommotor, so erkennt man, dass dieser Fehler mithilfe der Regler komplett beseitigt wurde. Die Linearführung befindet sich nach einer Fahrt immer genau auf dem Sollwert (Abbildung 9.6).

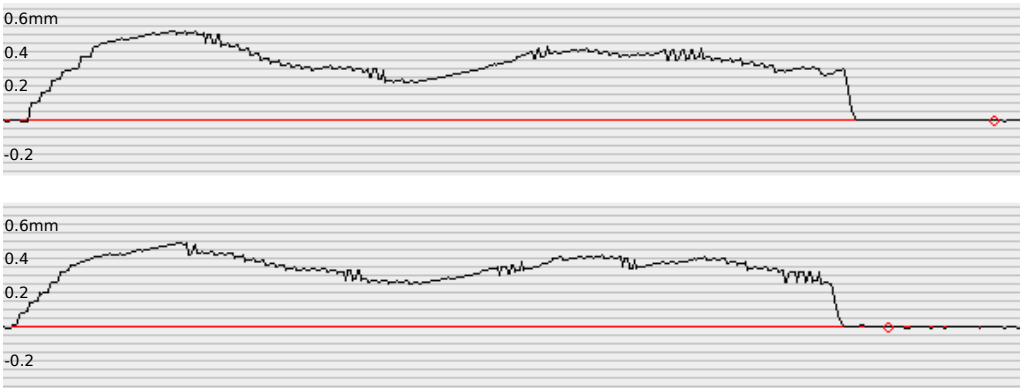


Abbildung 9.6: geregelter Gleichstrommotor

Selbst wenn nun die Linearführung durch äußere Einflüsse bewegt werden würde, würde sich das System wieder an die richtige Stelle fahren. Außerdem sorgt der STOPPAGE-Regler (Kapitel 6.3) dafür, dass das System nicht über das Ziel hinaus fährt, wie in Abbildung 9.7 zu sehen ist. Hier wurde dieser Regler zu Anschauungszwecken deaktiviert. Zusätzlich sorgt der HOLD-Regler (Kapi-

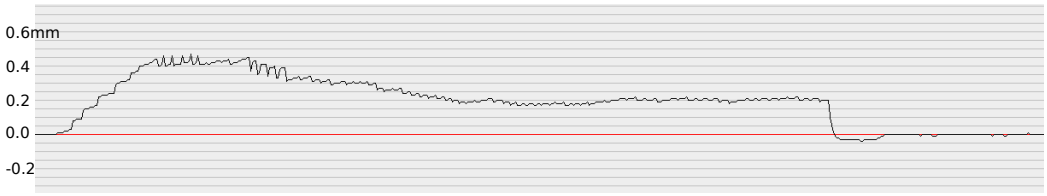


Abbildung 9.7: über das Ziel hinaus fahren

tel 6.1) dafür, dass die angefahrene Zielposition schnell gehalten wird und das System nicht zu schwingen anfängt. Für Abbildung 9.8 wurde auch dieser Regler deaktiviert.

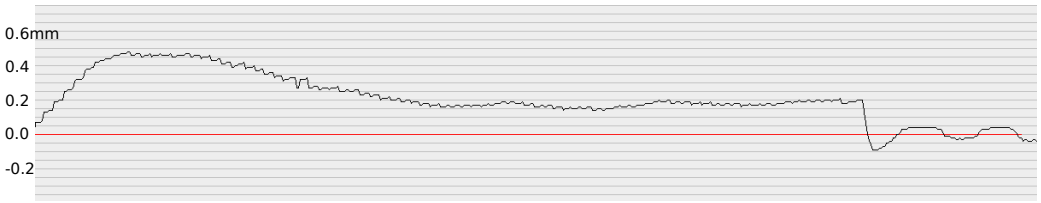


Abbildung 9.8: Schwingendes System

Zusammenfassend kann man sagen, dass das Spiel wie gewünscht komplett eliminiert wird.

## 9.4 Maximalgeschwindigkeit

Durch den Einsatz des Gleichstrommotors ist es nun möglich die Schiene mit einer Geschwindigkeit zu steuern, die doppelt so hoch ist wie in dem vorherigen System mit Schrittmotor. Die Maximalgeschwindigkeit wird bei $500 Ticks/sec$ erreicht und beträgt somit $500 Ticks/sec * \frac{1}{2560} Zoll/Tick = 0,1953 Zoll/sec \approx 4,961 mm/sec$. Der Schrittmotor schaffte hingegen nur maximal 250 Schritte pro Sekunde, was mit der Schrittweite von $\frac{1}{100}$Umdrehung und einer Umdrehung pro Millimeter zu einer Geschwindigkeit von $2,5 mm/sec$ führt. Der Gleichstrommotor hat in Tests ohne Belastung auch eine höhere Geschwindigkeit von etwa 5,3 mm/sec erreicht. Dennoch ist von einer höheren Tickfrequenz als 500 abzuraten, damit der Regler bei größeren Widerständen im Material noch die Möglichkeit hat mehr Spannung anzulegen und damit trotzdem die gewünschte Geschwindigkeit zu erreichen.

# Kapitel 10

# Schluss

Diese Arbeit hat sich damit befasst einen Linearantrieb, durch Verwendung eines neuen Motors und durch die Aufstellung eines Regelkreises, zu verbessern. Durch die Ersetzung des Schrittmotors durch einen Gleichstrommotor sind einige Komplikationen aufgetaucht, die es zu beheben galt. Allerdings ist das Ergebnis ein Linearaktor mit deutlich höherer Geschwindigkeit und stärkerem Antrieb. Betrachtet man nun die Ergebnisse aus Kapitel 9, so erkennt man, dass dieser Vorteil in einem geregelten System ohne nennenswerte Nachteile zu erreichen ist. Die Auflösung von etwa $\frac{1}{100}mm$ ist in etwa gleich geblieben, hat sich sogar minimal erhöht. Auch der erstrebte Ausgleich des Umkehrspiels funktioniert wie gewünscht. Selbst bei einem Spiel in der Gewindespindel oder bei widerstandsfähigem zu fräsendem Material wird der Zielpunkt genau angefahren. Der einzige Nachteil ist die Trägheit, welche das System durch den neuen Motor und das schwingungsfreie Regeln erhält. Durch den geschickten Einsatz einer geplanten Soll-Ist-Differenz während der Fahrt, wurde jedoch auch dieses Problem für den vorliegenden Verwendungszweck in einer Platinenfräse bestmöglich minimiert. Zusammenfassend hat diese Arbeit gezeigt, dass der Einsatz eines Regelkreises an einem Linearaktor eine durchaus vorzeigbare Alternative zu teuren Führungen stellt.

# Anhang A

# Bilder

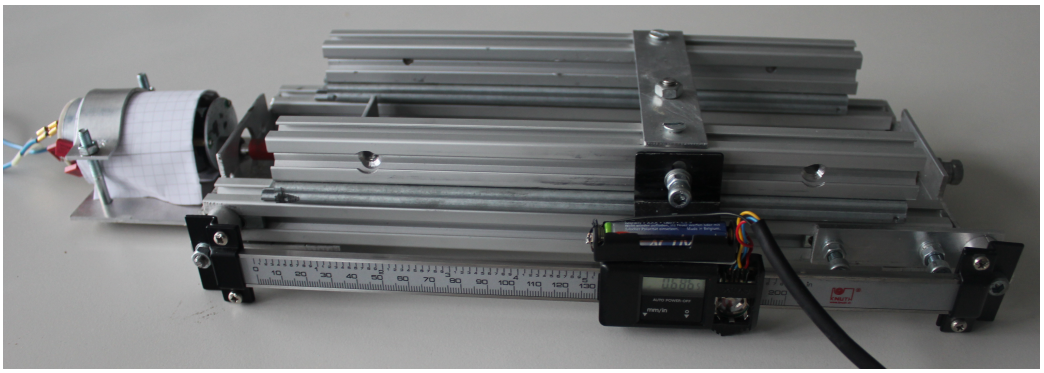

Abbildung A.1: Der Linearaktor - Fronalansicht

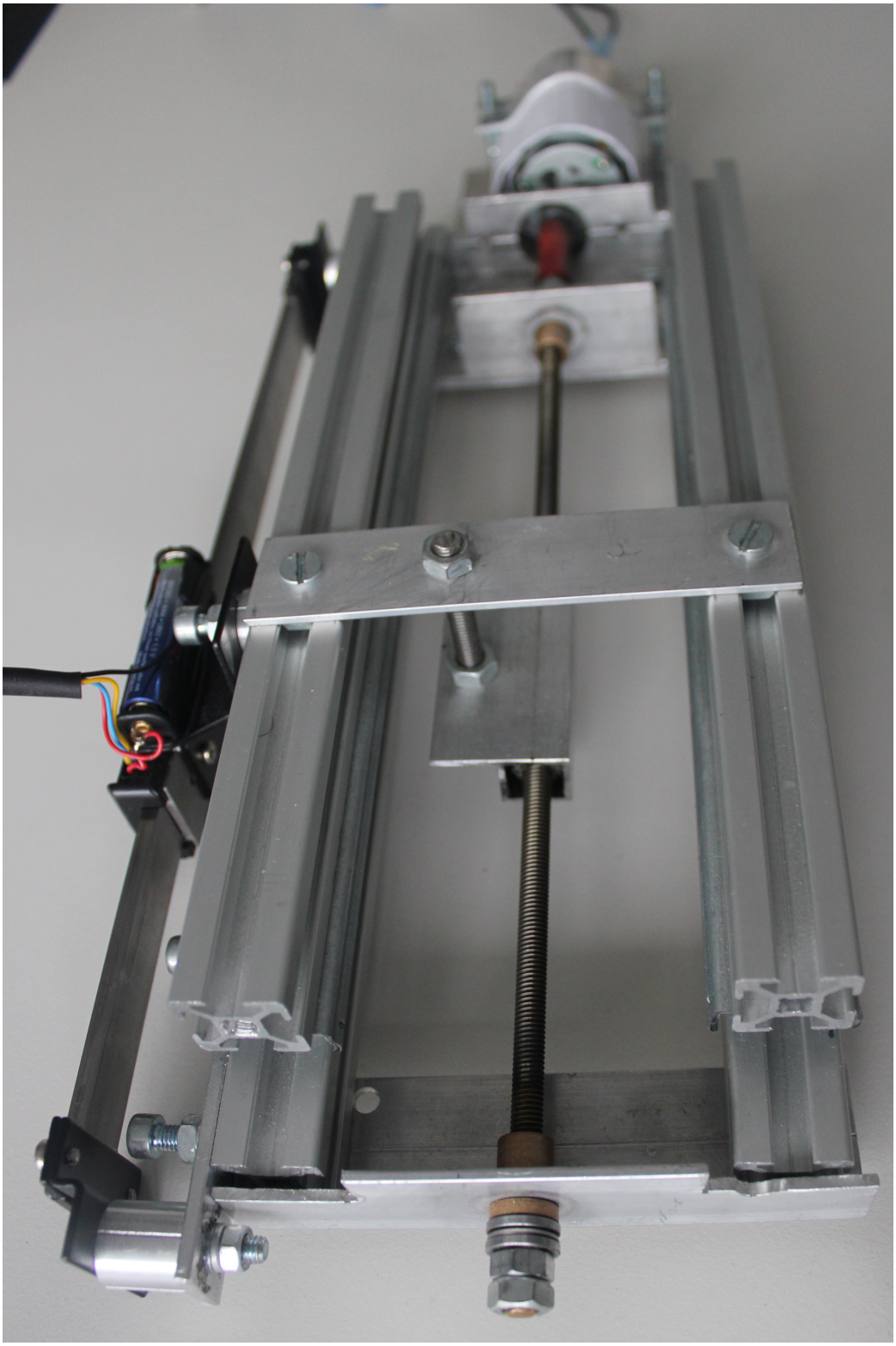

Abbildung A.2: Der Linearaktor - Seitenansicht

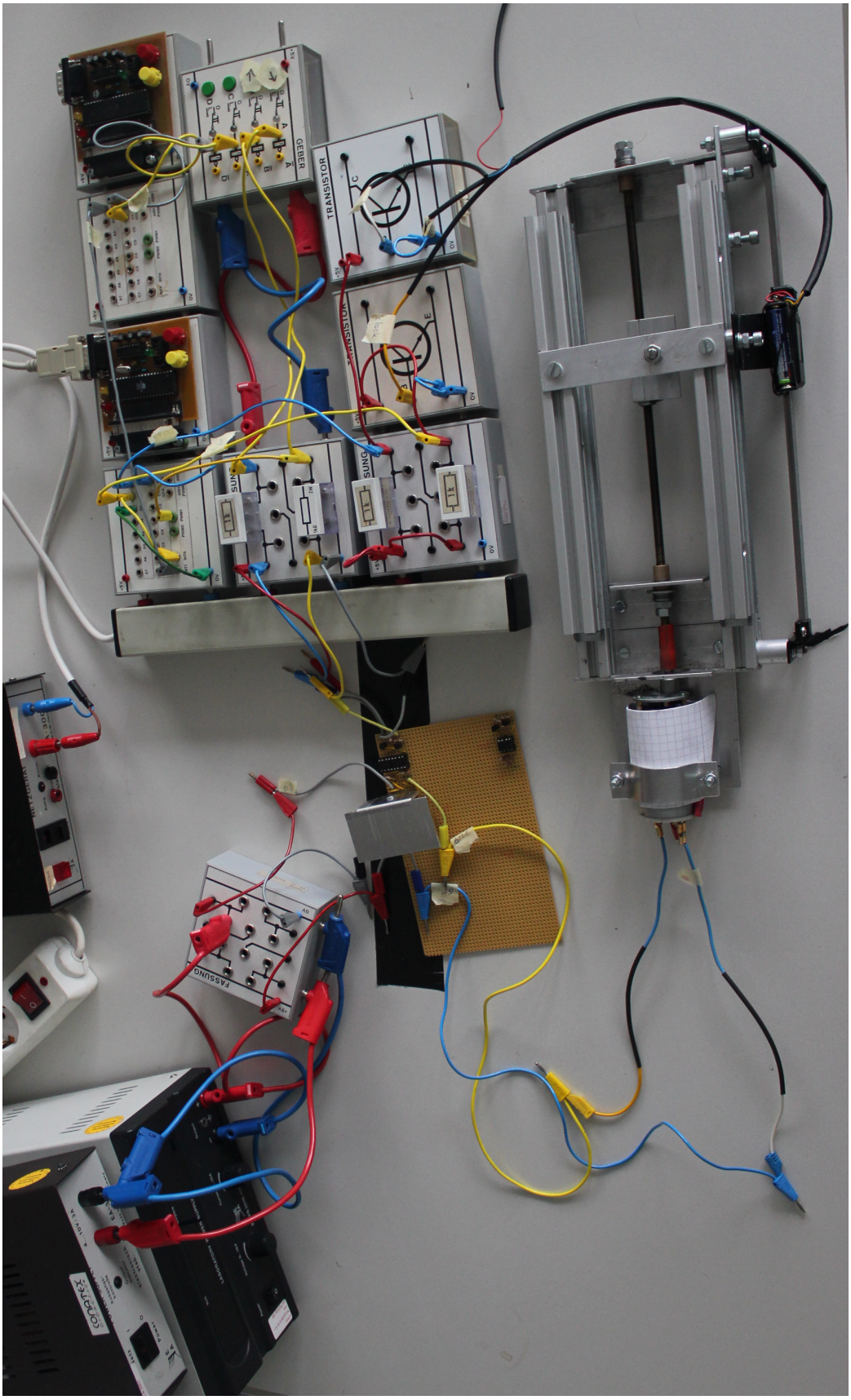


Abbildung A.3: Übersicht über den Versuchsaufbau

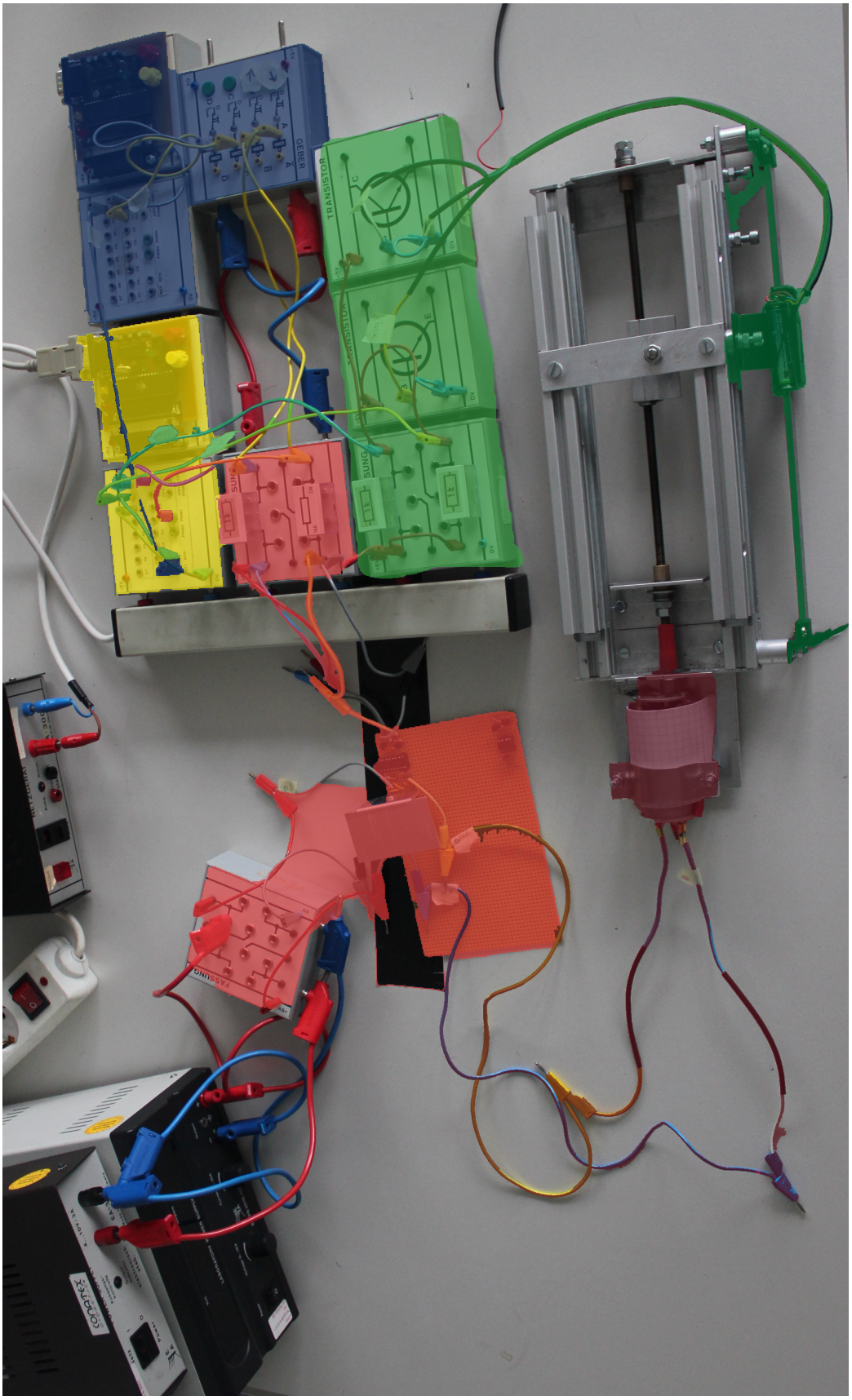

Abbildung A.4: eignefärbte Übersicht über den Versuchsaufbau
Grün = Schieblehre, Blau = HostSimulator, Rot = Motor, Gelb = Mikrocontroller

# Anhang B

# Quellcode

# B.1 main.c

```
1  /*
2   * BachelorArbeit.c
3   *
4   * Created: 05 - 10.2012
5   *  Author: Benedikt Jöbgen
6   */
7
8  #define F_CPU   16000000
9
10 #include <avr/io.h>
11 #include <avr/interrupt.h>
12 #include <stdlib.h>
13
14 #include "Motor.h"
15 #include "Caliper.h"
16 #include "Ticks.h"
17 #include "Uart.h"
18
19 #define HOLD            0
20 #define RUN                     1
21 #define STOPPAGE        2
22
23 volatile signed short is, goal, v, diff, old_diff, distance, new_ticks_arrived;
24 volatile signed char state, direction, new_is;
25 volatile unsigned short c_v, tick_Period;
26
27 ////////////////////////  MAIN  ////////////////////////////////////////
28 int main(void)
29 {
30         initUART();
31         sei();
32         initMotor();
33         initCaliper();
34         initTicks();
35
36         while(1){
37
38                 switch (state){
39                         case STOPPAGE: stoppage(); break;
40                         case RUN: run(); break;
41                         case HOLD: hold(); break;
42                 }
43
44                 UART_Send_fast(goal-is);
45         }
46
47 }
48 ////////////////////////////////////////////////////////////////////main//
```

## B.2 Caliper.h

```
1  #ifndef CALIPER_H
2  #define CALIPER_H
3
4  #define F_CPU 16000000
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <stdlib.h>
9  #include <util/delay.h>
10
11 #define CLOCK    (~PINA & 0x02)
12 #define DATA     (PINA & 0x01)
13
14 #define FALSE    0
15 #define TRUE     1
16
17 #define PEEK_ON                  PORTC |= 2;
18 #define PEEK_OFF                 PORTC &= ~0x02;
19 #define PEEK                     {PEEK_ON; _delay_us(1); PEEK_OFF;}
20
21 extern volatile signed short is;
22 extern volatile signed char new_is;
23
24 void initCaliper();
25 ISR(INT0_vect);
26
27  #endif /* CALIPER_H_ */
```

## B.3 Caliper.c

```
1  #include "Caliper.h"
2
3  //////// SCHIEBLEHRE ////////////////////
4  void initCaliper(){
5
6          DDRA=0x00;
7          PORTA=0xFF;
8
9          // erstes Packet Überspringen
10         while(!CLOCK);
11         _delay_ms(1);
12
13         // enable external Interrupt
14         MCUCR |=  (1<<ISC01);
15         GICR |= (1<< INT0);
16
17         // erster is Wert messen
18         for(int i = 0; i<30; i++)
19                 _delay_ms(1);
20
21 }
22
23 // Caliper ISR
24 ISR(INT0_vect)
25 {
26         // nur Interrupt INT0 sperren
27         GICR &= ~(1<< INT0);
28         sei();
29
30         short val = 0;
31
32         // 3Bits wegrunden
33         for(int i=0;i<3;i++)
34         {
35                 while(!CLOCK);
36                 while(CLOCK);
37                 //nop
38         }
39
40         // lese is-Wert
41         for(int i=0;i<16;i++)
42         {
43                 while(!CLOCK);
44                 while(CLOCK);
45                 val |= (DATA) << i;
46         }
47         is = val;
48         new_is = TRUE;
49
50         // 5 restliche Bits + 24 zweiter Block + 1 Stopbit
51         for(int i=0;i<30;i++)
52         {
53                 while(!CLOCK);
54                 while(CLOCK);
55                 //nop
56         }
57
58         // INT0 wieder freigeben
59         GIFR |= (1<< INTF0);
60         GICR |= (1<< INT0);
61 }
62 ////////////////////////////// Schieblehre //
```

## B.4 Ticks.h

```
1  #ifndef TICKS_H
2  #define TICKS_H
3
4  #define F_CPU 16000000
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <stdlib.h>
9  #include <util/delay.h>
10
11 #define HOLD            0
12 #define RUN                     1
13 #define STOPPAGE        2
14
15 #define FALSE   0
16 #define TRUE    1
17
18 #define TIMER1_PRESCALER 64
19 #define C_STOP  100
20
21 extern volatile signed short is, goal, diff, distance, new_ticks_arrived;
22 extern volatile signed char state;
23 extern volatile unsigned short tick_Period;
24
25 void initTicks();
26 ISR(INT1_vect);
27 ISR(TIMER1_OVF_vect);
28 ISR(TIMER1_COMPA_vect);
29
30  #endif /* TICKS_H_ */
```

## B.5 Ticks.c

```
1  #include "Ticks.h"
2
3  //////// TICKS ////////////////////
4  void initTicks(){
5
6          // zeitmessung zwischen Ticks mit Timer1
7          tick_Period = 0;
8          TCNT1 = 0;
9
10         new_ticks_arrived = FALSE;
11         goal = is;
12
13         switch(TIMER1_PRESCALER){
14                 case 1: TCCR1B |= (1<<CS10); break;
15                 case 8: TCCR1B |= (1<<CS11); break;
16                 case 64: TCCR1B |= ((1<<CS11)|(1<<CS10)); break;
17                 case 256: TCCR1B |= (1<<CS12); break;
18                 case 1024: TCCR1B |= ((1<<CS12)|(1<<CS10)); break;
19         }
20
21         MCUCR |= (1<<ISC11)|(1<<ISC10);  // The rising edge of INT1
22         GICR |= (1<< INT1);                      // enable external interrupt 1
23
24         TIMSK |= ((1<<TOIE1)     // TimerOverflowInterruptEnable
25                         |(1<<OCIE1A)); // OnCompareInterruptEnable
26
27
28 }
29
30 // tick kommt an
31 ISR(INT1_vect)
32 {
33         // TickTimer auslesen
34         tick_Period = TCNT1;
35         TCNT1 = 0;
36
37         OCR1A = tick_Period+C_STOP;
38
39         if(PINA & 0x04)
40                 new_ticks_arrived--;
41         else
42                 new_ticks_arrived++;
43 }
44
45 // zähler zwischen ticks -> kam schon länger kein tick mehr
46 ISR(TIMER1_OVF_vect)
47 {
48         if(state == RUN)
49         {
50                 stop();
51                 state = STOPPAGE;
52                 distance = 0;
53         }
54
55 }
56
57 ISR(TIMER1_COMPA_vect)
58 {
59         if(state == RUN)
60         {
61                 stop();
62                 state = STOPPAGE;
63                 distance = 0;
64         }
```

```
65 }
66 //////////////////////////// ticks //
```

## B.6 Motor.h

```
1  #ifndef MOTOR_H
2  #define MOTOR_H
3
4  #define F_CPU 16000000
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <stdlib.h>
9  #include <util/delay.h>
10 #include <math.h>
11
12 #define TICKSPERINCH              2560.0F
13 #define V_MAX_INCHPERSEC          0.31
14
15 #define VMAX 255
16 #define VMIN 45
17
18 #define TIMER1_PRESCALER 64
19
20 #define FALSE   0
21 #define TRUE    1
22
23 #define DOWN    -1      // für direction
24 #define UP              1
25
26 #define HOLD            0       // für state
27 #define RUN                     1
28 #define STOPPAGE        2
29
30 #define C_DISTANCE              2       // distance = v / C_DISTANCE
31 #define C_P                             0.9F// regeln beim run
32 #define C_D                             0.5F//  ''
33 #define C_BRAKE                 10      // abbremsfaktor bei stoppage
34 #define C_STOP_DISTANCE 3       // distanz ab der von stoppage in hold geswitcht wird
35 #define C_STOP_IMPULS   5       // starker Stopimpuls [ms]
36 #define C_HOLD_DRIVE    2       // Fahrimpuls [ms]
37 #define C_HOLD_STOP             30      // Stopimpuls [ms]
38
39 #define C_DIFF                  20      //differenzkonstante zwischen den richtungen
40
41 extern volatile signed short is, goal, v, diff, old_diff, distance, new_ticks_arrived;
42 extern volatile signed char state, direction, new_is;
43 extern volatile unsigned short c_v, tick_Period;
44
45 void initMotor();
46 void forwards(unsigned char speed);
47 void backwards(unsigned short speed);
48 void drive(short speed);
49 void calculate_v();
50 void stoppage();
51 void hold();
52 void run();
53
54  #endif /* MOTOR_H */
```

# B.7 Motor.c

```
1  #include "Motor.h"
2
3  //////// MOTORSTEUERUNG ////////////////////
4  void initMotor(){
5
6          c_v = (unsigned short)((float)VMAX*F_CPU/2/(V_MAX_INCHPERSEC * TIMER1_PRESCALER * TICKSPERINCH));
7
8          // Outout
9          DDRC = 0xFF;
10         DDRD |= 0x80;
11
12         // PWM mit Timer2
13         TCCR2 = (1<<WGM20)              //      Phase Correct
14                         |(1<<COM21)             //      nicht invertiert
15                         |(1<<CS22);             //      prescaler 32 -> 1/4MHz
16
17         //Initialwerte festlegen
18         diff = 0;
19         old_diff = 0;
20         v = 0;
21         drive(0);
22         distance = 0;
23         state = HOLD;
24 }
25
26 // faehrt mit Geschwindigkeit(speed [0,255]) vorwaerts
27 void forwards(unsigned char speed)
28 {
29         PORTC &= 0xfe;
30         OCR2 = speed;
31 }
32
33 // faehrt mit Geschwindigkeit(speed [0,255]) rueckwaerts
34 void backwards(unsigned short speed)
35 {
36         PORTC |= 0x01;
37         OCR2 = 255-speed;
38 }
39
40 // fährt mit Geschwindigkeit(speed [-255,255])
41 void drive(signed short speed)
42 {
43         if(speed >= 0)
44                 forwards( (speed>=VMAX) ? VMAX : (unsigned char)speed );
45         else{
46                 speed = abs(speed);
47                 speed = (speed <= C_DIFF) ? 0 : ((speed >= VMAX+C_DIFF) ? VMAX : speed - C_DIFF);
48                 backwards(speed);
49         }
50 }
51
52 // Anhalten
53 void stop()
54 {
55         PORTC = 0x00;
56         OCR2 = 0;
57         v = 0;
58         state = HOLD;
59 }
60
61 // berechnet zu fahrende Geschwindigkeit
62 // ungeregelt, nur abhängig der zeitlichen Tickabstände
63 void calculate_v()
64 {
```

```
65          // goal anpassen
66          direction = (new_ticks_arrived > 0) ? UP : DOWN ;
67          goal += new_ticks_arrived;
68          new_ticks_arrived = 0;
69
70          // erster Tick
71          if(state != RUN)
72          {
73                  v = VMAX * direction;
74                  state = RUN;
75          }
76          // sonstige ticks
77          else {
78                  short v2 = (c_v/tick_Period*2);
79
80                  if(v2<VMIN)
81                          state = HOLD;
82                  else
83                          v = v2 * direction;
84          }
85
86          // Distanz anpassen
87          distance = (v / C_DISTANCE);
88          old_diff = 0;
89  }
90
91  // Stoppage-Regler
92  void stoppage()
93  {
94          // starker Gegenimpuls am Anfang
95          if(direction != 0){
96                  drive((signed short)-VMAX * direction);
97                  _delay_ms(C_STOP_IMPULS);
98                  forwards(0);
99                  direction = 0;
100         }
101
102         // alte Differenz
103         old_diff = diff;
104         while(new_is==FALSE);
105         new_is = FALSE;
106         diff = (goal-is);
107
108         // am Ziel angekommen -> HOLD
109         if((abs(diff) <= C_STOP_DISTANCE))
110                 stop();
111         // ... noch nicht -> Regler
112         else {
113                 signed short v_regler = C_BRAKE * (abs(old_diff) - abs(diff));
114
115                 if (v_regler > VMIN)
116                         v_regler = VMIN;
117                 else if (v_regler < 0)
118                         v_regler = 0;
119
120                 if(is < goal)
121                         drive((signed short)(VMIN) - v_regler);
122                 else
123                         drive(v_regler - VMIN);
124         }
125
126 }
127
128 // HOLD_Regler
129 void hold()
130 {
131
```

```
132         diff = (goal-is);
133
134         if(diff != 0)
135         {
136                 // kurzer Impuls
137                 if(is < goal)
138                         drive(VMAX);
139                 else
140                         drive(-VMAX);
141                 _delay_ms(C_HOLD_DRIVE);
142
143                 // stop
144                 forwards(0);
145                 for(int i = 0; i< C_HOLD_STOP; i++)
146                         _delay_ms(1);
147         }
148         // steht richtig
149         else forwards(0);
150
151 }
152
153 // RUN-Regler
154 void run()
155 {
156         if(new_ticks_arrived != 0)
157                 calculate_v();
158
159         old_diff = diff;
160         diff = (goal-is);
161
162         float v_regler;
163         v_regler = C_P * (diff-distance);       // P-Regler
164         v_regler += C_D * (diff - old_diff);// D-Regler
165
166         drive(v + (short)v_regler);
167
168 }
169
170 //////////////////////////// motor //
```

## B.8 Uart.h

```
1  #ifndef UART_H
2  #define UART_H
3
4  #define F_CPU 16000000
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <stdlib.h>
9
10 #define BAUD    38400
11
12 void init_UART();
13 void UART_Send(char x);
14 void UART_Send_fast(char x);
15
16
17  #endif /* UART_H */
```

## B.9 Uart.c

```
1  #include "Uart.h"
2
3  //////// UART ////////////////////
4
5  void initUART()
6  {
7          unsigned int ubrr = F_CPU/16/BAUD-1 ;
8
9          UCSRB = (1<<TXEN);
10         UCSRC = (1<<URSEL)|(3<<UCSZ0);
11         UBRRH = (unsigned char)(ubrr>>8);
12         UBRRL = (unsigned char)ubrr;
13 }
14
15 // sendet garantiert, aber langsam
16 void UART_Send(char x)
17 {
18         while(!(UCSRA & (1<<UDRE)));
19                 UDR = x;
20 }
21
22 // garantiert senden nicht, aber schnell
23 void UART_Send_fast(char x)
24 {
25         if(UCSRA & (1<<UDRE))
26                 UDR = x;
27 }
28 ////////////////////////////// uart //
```

# Anhang C

# Quellcode Schrittmotor

Im Folgenden werden nur die Module aufgeführt, welche sich von den original Modulen unterscheiden. Die Module Uart und Caliper bleiben unverändert.

# C.1 main_Schrittmotor.c

```
1  /*
2   * main_Schrittmotor.c
3   *
4   * Created: 12.2012
5   *  Author: Benedikt Jöbgen
6   */
7
8  #define F_CPU   16000000
9
10 #include <avr/io.h>
11 #include <avr/interrupt.h>
12 #include <stdlib.h>
13
14 #include "Schrittmotor.h"
15 #include "Schieber.h"
16 #include "Ticks_Schrittmotor.h"
17 #include "Uart.h"
18
19 volatile signed short is, goal, v, diff, olddiff, distance, new_ticks_arrived;
20 volatile signed char direction, newis, ticked;
21 volatile unsigned short c_v, tick_Period;
22
23 ////////////////////////  MAIN  ////////////////////////////////////////
24 int main(void)
25 {
26
27         UART_Init();
28         sei();
29         initSchrittmotor();
30
31         for(int i = 0; i<100; i++)
32                         _delay_ms(10);
33
34         initSchieber();
35         initTicks();
36
37         while(1){
38
39                 if(ticked>0){
40                         schritt_vor();
41                         goal += ticked;
42                         ticked = 0;
43                 }
44                 else if(ticked<0){
45                         schritt_rueck();
46                         goal += ticked;
47                         ticked = 0;
48                 }
49                 UART_Send_fast(goal-is);
50         }
51 }
52 ////////////////////////////////////////////////////////////////////main//
```

## C.2 Schrittmotor.h

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  volatile unsigned char step; // state des steps
5
6  void initSchrittmotor();
7  void schritt_vor();
8  void schritt_rueck();
```

## C.3 Schrittmotor.c

```
1  #include "Schrittmotor.h"
2
3  void initSchrittmotor(){
4          DDRC=0xFF;
5          step = 0;
6          PORTC = 20;
7  }
8
9  void schritt_vor(){
10         step = (step+3)%4;
11         switch(step){
12                         case 0: PORTC = 0;      break;
13                         case 1: PORTC = 1;      break;
14                         case 2: PORTC = 5;      break;
15                         case 3: PORTC = 4;      break;
16                 }
17 }
18
19 void schritt_rueck(){
20         step = (step+1)%4;
21         switch(step){
22                         case 0: PORTC = 0;      break;
23                         case 1: PORTC = 1;      break;
24                         case 2: PORTC = 5;      break;
25                         case 3: PORTC = 4;      break;
26                 }
27 }
```

## C.4 Ticks_Schrittmotor.h

```
1  #ifndef TICKS_H
2  #define TICKS_H
3
4  #define F_CPU 16000000
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <stdlib.h>
9  #include <util/delay.h>
10
11 #include "Schrittmotor.h"
12
13 #define HOLD              0
14 #define RUN                       1
15 #define STOPPAGE          2
16
17 #define FALSE    0
18 #define TRUE     1
19
20 #define TIMER1_PRESCALER 64
21 #define VMAX 255
22
23 extern volatile signed short is, goal, diff, olddiff, distance, new_ticks_arrived;
24 extern volatile signed char state, ticked;
25 extern volatile unsigned short tick_Period;
26
27 void initTicks();
28 ISR(INT1_vect);
29 ISR(TIMER1_OVF_vect);
30 ISR(TIMER1_COMPA_vect);
31
32
33  #endif /* TICKS_H_ */
```

## C.5 Ticks_Schrittmotor.c

```
1  #include "Ticks_Schrittmotor.h"
2
3  ////////////////TICKS//////////////////////////////////////////////////////////
4  void initTicks()
5  {
6          goal = is;
7          ticked = 0; // anzahl der angekommen ticks (mit vorzeichen)
8
9          MCUCR |= (1<<ISC11)|(1<<ISC10);          //The rising edge of INT1
10         GICR |= (1<< INT1);                          //enable external interrupt 1
11 }
12
13 // tick kommt an
14 ISR(INT1_vect)
15 {
16         if(PINA & 0x04){
17                 ticked++;}
18         else{
19                 ticked--;}
20 }
21 ////////////////////////////////////////////////////////////////////////ticks//
```

# Literaturverzeichnis


[atm13] *ATmega16 datasheet*. http://www.atmel.com/Images/doc2466.pdf, 2013.

[l2913] *L298 datasheet - dual full-bridge driver*. http://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf, 2013.

[MÏ13] Nick Müller. Protocols of digital scales. URL: http://www.yadro.de/digital-scale/protocol.html , 2013.

[opt13] *6N138 datasheet - 8pin dip high speed split darlinghton photocoupler*. http://www.promelec.ru/pdf/6N138EverL.pdf, 2013.

[Was13] Waste. Regelungstechnik. URL: http://www.rn-wissen.de/index.php/Regelungstechnik , 2013.