Steuerung einer mobilen Roboterplattform mit einem vorgegebenen eingebetteten System und Echtzeitbetriebssystem

Diplomarbeit

des Studiengangs Elektrotechnik, Fachrichtung Automatisierungstechnik im Fachbereich EMT der FH Bonn-Rhein-Sieg

Christoph Probst

Matrikelnummer: 9006285

Erstprüfer:	Prof. Dr. Wolfgang Joppich
Zweitprüfer:	Prof. Dr. Marco Winzker

Bearbeitungszeitraum: 15. Mai 2008 bis 15. August 2008

Bonn, August 2008

Kurzfassung

Das Roboter-Baukastensystem ProfiBot vom Fraunhofer Institut IAIS wird in Zusammenarbeit mit der Hochschule für Technik und Wirtschaft (HTW-Saarbrücken) und der Firma HighTec EDV-Systeme GmbH aus Saarbrücken zu einer mobilen Roboterplattform für die Ausbildung an Hochschulen weiterentwickelt. In dieser Diplomarbeit wird das vorgegebene eingebettete System und ein Echtzeitbetriebssystem der Firma HighTec EDV-Systeme GmbH benutzt, um die Regelung der Motoren und der Fahrzeugbewegungen zu implementieren. Ein Benutzer kann die mobile Roboterplattform mithilfe einer Schnittstelle zur Anwendungsprogrammierung (API) auf Basis von physikalischen Größen ansteuern und aktuelle Zustände abfragen. Um die mobile Roboterplattform flexibel benutzen zu können, werden mit einer zusätzlichen Elektronik digitale und analoge Ein- und Ausgänge des eingebetteten Systems auf die Anwendungen der Robotik angepasst. Neben einem Programmstartschalter und Status-LEDs können vier Schaltleisten zur Kollisionserkennung und analoge Sensoren mit Einheitssignal angeschlossen werden. Zuletzt wird die Reglerstruktur kalibriert und getestet.

Abstract

In cooperation with HighTec EDV-Systeme GmbH and Hochschule für Technik und Wirtschaft in Saarbrücken Fraunhofer IAIS advances the robot assembly kit ProfiBot to a mobile robot platform. It shall be used in university context. In this thesis a specified embedded system and real time operating system (both from HighTec EDV-Systeme GmbH) is used, to implement the motor control and the movement of the vehicle. The user should be able to control the mobile robot platform with an application programmer interface (API). The API needs input as physical values and returns physical values as output. To use the mobile robot platform flexibly for extended applications, some analogue and digital input and output ports are adapted to the embedded system by an additional circuit board. It is possible to connect a program start switch, state LEDs, four bumpers to detect an obstacles and analogue sensors with standard signal. Finally the controller will be calibrated and tested.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Mobile Roboterfahrzeuge werden zunehmend in unterschiedlichen Bereichen eingesetzt. Neben industriellen Anwendungen gibt es Modelle, die als Unterhaltungsmedium oder als Lehrgegenstand genutzt werden. Die meisten dieser mobilen Roboterfahrzeuge sind spezielle Lösungen für ein Problem. Damit sind auch die Softwaremodule und Programmierschnittstellen, falls vorhanden, an die spezielle Lösung angepasst. Für die Lehre sind solche Roboterfahrzeuge eher ungeeignet. Um Anwenderprogramme auf unterschiedlichen Roboterfahrzeugen auszuführen ist eine einheitliche Schnittstelle nötig. In der vorliegenden Arbeit werden für eine mobile Roboterplattform mit vorgegebenem eingebetteten System und Echtzeitbetriebssystem eine Regelstruktur für die Motoren und ein Schnittstellenkonzept entwickelt und getestet, um eine einheitliche Ansteuerung von mobilen Roboterfahrzeugen zu ermöglichen.

1.1 Motivation

1.1.1 Ausbildungsrobotik

Das Fraunhofer-Institut IAIS (Institut für **I**ntelligente **A**nalyse und **I**nformations**s**ysteme) entwickelt im Bereich Ausbildungsrobotik Robotersysteme, die dazu dienen, anhand ihrer Bau- und Funktionsweise zu lernen. Robotersysteme sind interessant für die Ausbildung, weil mit ihnen fachübergreifende Lerninhalte in einem interdisziplinären Umfeld vermittelt werden können. Mechatroniker lernen z.B. die Programmierung und Steuerung und Informatiker bekommen Einblick in Elektrik und Mechanik. In diesem Zusammenhang reicht es jedoch nicht aus, einzelne Teilsysteme zu betrachten, sondern es ist notwendig, das Gesamtsystem zu verstehen. Die eingesetzten Robotersysteme sind oft nicht sehr innovativ. Dafür ist ihre Technik so transparent, dass die Grundlagen vieler Fachbereiche leicht zu verstehen sind. Zwei Beispiele für Roboter in der Ausbildung sind im Folgenden aufgeführt.

1. Roberta: Mädchen erobern Roboter

   Roberta ist ein Ausbildungskonzept, in dem mithilfe von LEGO-MINDSTORMS®-Robotern Mädchen für Technik begeistert werden. Anhand von Begleitbüchern können aus LEGO-Bausätzen Roboter aufgebaut und programmiert werden. Die Programmierschnittstelle erlaubt es, neben einfach und schnell zu verwirklichenden Programmen auch komplexere Probleme aus der Robotik zu behandeln. Obwohl sich Roberta an Schülerinnen und Schüler richtet, haben auch einige Hochschulen Interesse an der Lernweise gefunden, z.B. für die Einführung in das Thema „eingebettete Systeme“.

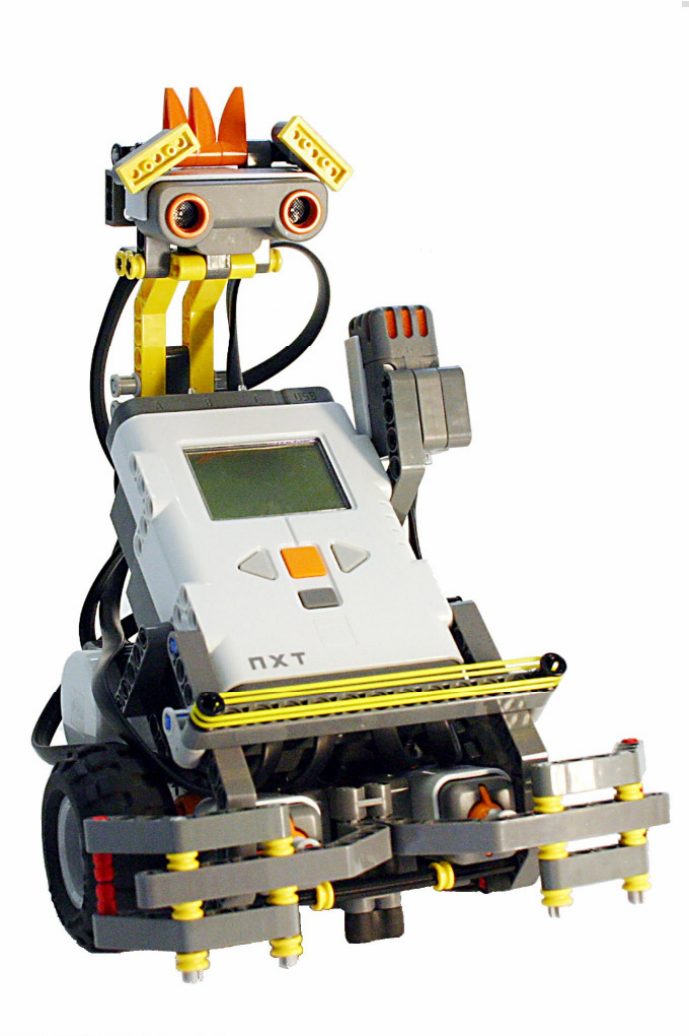


Abbildung 1: Roberta NXT-Roboter (Foto: IAIS)

2. ProfiBot: mechatronisches Baukastensystem für die Berufsausbildung

Im Projekt ProfiBot wird ein Roboter-Baukastensystem inklusive Lehr- und Lernmaterial für die berufliche Mechatronikausbildung entwickelt. Ziel dieses Baukastensystems ist es, möglichst viele Lernfelder der Mechatronikausbildung in einem Lernsystem zu vereinen. Den Auszubildenden ist es möglich, im Rahmen ihrer Ausbildung den Roboter nach und nach aufzubauen, Teile dafür selber anzufertigen und dabei berufliche Handlungsfähigkeit zu erlangen. Eine Aufgabensammlung für die Ausbilder unterstützt diese mit Ideen und Hintergrundinformationen. Ist das Gerät vollständig aufgebaut, stellt es ein programmierbares mobiles Robotersystem dar. Mithilfe von vorgefertigten Modulen in der signalflussorientierten Programmierumgebung ICONNECT von der Firma MICRO-EPSILON können die Auszubildenden sehr schnell einfache Programme für den ProfiBot zusammenstellen. Der ProfiBot als programmierbare Roboterplattform wird auch von einigen Hochschulen als Lernplattform benutzt

Abbildung 2: Baukastensystem ProfiBot (Foto: IAIS)

Da die Ausbildungsrobotik viele fachliche Bereiche abdeckt und interdisziplinäre Arbeit fördert, werden Roboter als Lernmedium immer interessanter. Viele Roboter auf dem Markt sind jedoch Lösungen für spezielle Probleme und daher für die Lehre eher ungeeignet.

1.1.2 Die Weiterentwicklung des ProfiBot

Das Fraunhofer-Institut IAIS arbeitet im Projekt ProfiBot mit verschieden Partnern aus der Ausbildungspraxis zusammen, die mit ihren Erfahrungen der Qualitätssteigerung des Produktes beitragen können. Auf den regelmäßigen Partnertreffen stellte sich nach und nach heraus, dass Änderungen bzw. Erweiterungen am Baukastensystem ProfiBot sinnvoll sind. Der Steuerlaptop stellt zwar eine gut bedienbare Plattform für das Steuerprogramm dar, ist aber recht teuer, nimmt viel Platz in Anspruch und hat eine getrennte und begrenzte Stromversorgung. Zudem kamen einige Partner nicht mit dem graphischen Programmiersystem ICONNECT zurecht und wollten lieber in bekannten Programmiersprachen wie C oder C++ programmieren.

Ein weiterer Schwachpunkt ist das Antriebssystem. Im ProfiBot sind zwei Motoren mit Getriebe und Encoder der Firma maxon motor verbaut, die mit einem vom Fraunhofer

Institut IAIS entwickelten Motorkontroller, dem „TMC 2000“, angesteuert werden. Die Motoren und der Motorkontroller sind sehr teure Komponenten. Die Leistung der Motoren wird nur mäßig ausgeschöpft, da zum einen die Nennspannung der Motoren bei 30 V, die Betriebsspannung aber nur bei nominell 24 V liegt, und zum anderen die Software des TMC 2000 aus Sicherheitsgründen die Ansteuerung der Motoren drosselt. Das hat zur Folge, dass der Roboter nur geringe Geschwindigkeiten erreicht und die Antriebseinheit recht teuer und schwer ist.

In Zusammenarbeit mit der Hochschule für Technik und Wirtschaft (HTW) in Saarbrücken und der Firma HighTec EDV-Systeme GmbH in Saarbrücken entstand die Idee, eine dem ProfiBot ähnliche Roboterplattform zu bauen, die mit einem eingebetteten System und einem verbesserten Antriebssystem ausgestattet ist. Grundlage hierfür sind einige Produkte der Firma HighTec EDV-Systeme GmbH. Dazu gehört ein Golf-Caddy, der einen drucksensitiven Griff besitzt und den Golfspieler beim Ziehen des Golf-Caddys durch zwei Elektroantriebe unterstützt. Ein Echtzeitbetriebsystem, das ebenfalls von der Firma HighTec EDV-Systeme GmbH entwickelt wurde, steuert die Motoren an, liest die Positionen der Motoren ein und reagiert auf die Sensoren am Griff. Mit den Komponenten aus dem Golf-Caddy, den Motoren, dem eingebetteten System und dem Echtzeitbetriebssystem, soll im Rahmen dieser Arbeit versucht werden, den ProfiBot so zu verbessern, dass er in der Hochschulausbildung effizient und effektiv eingesetzt werden kann.

1.1.3 Ziele der Arbeit

In Vorarbeit zu dieser Diplomarbeit wurde, wie in Kapitel 2.6.1 genauer beschrieben, auf Basis des ProfiBot-Baukastensystems eine mobile Roboterplattform mit dem Namen „HighTecBot“ aufgebaut, die mit oben genannten Komponenten der Firma HighTec EDV-Systeme GmbH ausgestattet wurde. Damit steht eine leicht aufzubauende und programmierbare Plattform zur Verfügung. Da es sich nicht um eine spezielle Lösung für ein Problem aus der Robotik handelt, sondern um ein System, das auf verschiedenste Weise eingesetzt und erweitert werden kann, muss die Programmierung ebenso flexibel wie einfach möglich sein. Insbesondere für die Ausbildungsrobotik ist es wichtig, einfach zu bedienende Schnittstellen zur Verfügung zu stellen. Hilfreich ist es, wenn man die Vorgaben an die Roboterplattform auf physikalischen Größen basierend formulieren kann und Rückmeldungen in den gleichen Größen erhält.

Ein Ziel dieser Arbeit ist es, die vorhandene Software so zu erweitern, dass eine Schnittstelle zur Anwendungsprogrammierung (**a**pplication **p**rogramming **i**nterface, kurz: API) innerhalb des eingebetteten Systems zur Verfügung gestellt wird, die die mobile Roboterplattform geregelt nach physikalischen Vorgaben bewegt. Grundlage dieser Schnittstelle ist die im Golf-Caddy verwendete Ansteuerung der Motoren, die es erlaubt, eine PWM (**P**uls**w**eiten**m**odulation) für jeden Motor vorzugeben und die Motorpositionen einzulesen. Die Schnittstelle soll so allgemein sein, dass Anwenderprogramme, die diese Schnittstelle benutzen, ohne Anpassungen auf andere Roboterfahrzeuge übertragbar sind.

Ein weiteres Ziel ist es, durch eine zusätzliche Hardware die Möglichkeit zu eröffnen, analoge Sensoren und einen Programm-Start-Schalter einfach an das vorgegebene eingebettete System des HighTecBot anzuschließen.

Das Ergebnis dieser Arbeit ist kein fertiges Produkt für die Hochschulausbildung, sondern ein wichtiger Schritt hin zu einer standardisierten Ansteuerung von mobilen Roboterfahrzeugen. Zudem ist der Prototyp des HighTecBot soweit entwickelt, dass eine leichte Bedienung des Roboters und eine einfache Programmierung der Bewegungsabläufe möglich werden.

1.2 Anforderungen

Aus den im Abschnitt 1.1.3 beschriebenen Zielen dieser Arbeit können folgende Anforderungen abgeleitet werden.

- Einbinden der Robotersteuerung als Modul ins Echtzeitbetriebsystem

  Existiert die Robotersteuerung als ein Modul, das in das vorgegebene Echtzeitbetriebsystem eingebunden werden kann, ist eine einfache Erweiterung des vorhandenen Betriebssystems möglich. Die Robotersteuerung kann wahlweise hinzugenommen oder weggelassen werden. Die Software bleibt flexibel.

- Schnittstellenfunktionen als API zur Verfügung stellen

  Ein Benutzer soll aus einer Auswahl von Funktionen auswählen können, um Vorgaben an die mobile Roboterplattform zu übergeben, oder Istwerte zu bekommen. Die Schnittstelle vereinfacht den Datenaustausch zwischen dem Steueralgorithmus des Benutzers und der internen Regelung der Robotersteuerung.

- Vorgabe der Bewegungen anhand physikalischer Größen

  Um die Bedienung der mobilen Roboterplattform auch durch Personen zu ermöglichen, die nicht alle Einzelheiten und Besonderheiten kennen, soll die Schnittstelle möglichst allgemeinverständlich gehalten sein. Sinnvoll sind physikalische Größen, da auch die Bewegungen anderer mobiler Roboterplattformen in den gleichen Größen beschrieben werden können. Die Schnittstelle ist damit auf andere Roboter übertragbar. Benutzerprogramme können auf andere Systeme übertragen bzw. von anderen Systemen übernommen werden.

- Sicherstellen der Vorgabe durch Regelungen

  Die von einem Benutzer eingebenen Vorgaben sollen so genau wie nötig ausgeführt werden. Dazu sind Regler nötig, die den Istwert an den Sollwert angleichen können.

- Istwerte sollen abfragbar sein

  Damit einem Benutzer immer bekannt ist, in welchem Zustand sich die mobile Roboterplattform befindet, sollen alle vorgebbaren Werte auch abfragbar sein. Das von einem Benutzer implementierte Steuerprogramm könnte dann z.B. als übergeordneter Regler arbeiten oder die Werte visualisieren.

- Maximalwerte sollen einstellbar sein

  In der Ausbildung kann die mobile Roboterplattform für verschiedene Zwecke eingesetzt werden. Um das zu ermöglichen, sollen Begrenzungen der Maximalwerte von dem Benutzer vorgegeben werden können. Damit wird die Schnittstelle flexibel einsetzbar.

- Geradeausfahrt soll möglich sein

  Die Geradeausfahrt eines Fahrzeugs wird von fahrzeugspezifischen und äußeren Faktoren beeinflusst. Kann man diese Faktoren durch Softwarelösungen kompensieren, spart man sich aufwändige Kalibrierungsfahrten.

- Weitere Sensoren sollen eingebunden werden können

  HighTecBot ist in seiner jetzigen Form ein erweiterbarer Baukasten. Dies soll sich nicht nur im mechanischen Aufbau, sondern auch in der Erweiterung mit Sensoren äußern. Das vorgegebene eingebettete System bietet dazu die Möglichkeit.

- Starten und Stoppen des Programms durch einen Schalter

  Ein komfortables Bedienen der mobilen Roboterplattform äußert sich unter anderem in einem Starten und Stoppen des Steuerprogramms durch einen externen Schalter. Dazu gehört eine Statusanzeige des Programms, mit der evtl. auch Fehler sichtbar gemacht werden können. Schalter und Statusanzeige müssen eingebunden werden.

1.3 Strukturierung der Arbeit

Insgesamt besteht die vorliegende Diplomarbeit aus vier Kapiteln. Nachfolgend wird kurz beschrieben, wie die Arbeit aufgebaut und strukturiert ist.

Im Einleitungsteil wird beginnend von der Problematik, das die meisten im handel erhältlichen Roboter spezielle Lösungen sind, über die Ausbildungsrobotik auf ein spezielles Projekt im Fraunhofer-Institut IAIS hingeführt. Aus der Entstehung und Weiterentwicklung des Projektes ProfiBot zur mobilen Roboterplattform HighTecBot lassen sich Ziele bestimmen, die einen wichtigen Schritt in der Entwicklung der Ausbildungsrobotik für den Hochschulbereich darstellen. Die zum Erreichen der Ziele abgeleiteten Anforderungen sollen in dieser Arbeit umgesetzt werden.

In einem Grundlagenteil wird generelles Hintergrundwissen angesprochen, das zu dieser Arbeit nötig ist, außerdem werden die vorgegebene Hardware und Software vorgestellt. Der HighTecBot, der als Programmierplattform dient, ist ebenfalls eine wichtige zu besprechende Grundlage.

Im Hauptteil „Entwicklung und Test“ werden zunächst die Anforderungen an die zu entwickelnde Hardware gestellt, die zur Verfügung stehenden Schnittstellen analysiert und nach und nach die nötigen Entwicklungsschritte hin zur fertigen Elektronik dargestellt. Nach der Einbindung der Hardware in die Software wird ein Konzept für die zu programmierende Schnittstelle definiert. Anschließend werden die Besonderheiten und Unterebenen der Robotersteuerung beschrieben. Tests sollen zeigen, dass die Schnittstelle wie gewünscht funktioniert.

Im letzten Kapitel wird die Arbeit zusammengefasst und das Ergebnis besprochen. Mögliche Verbesserungen und sinnvolle Weiterentwicklungen sowie der mögliche Einsatz der mobilen Roboterplattform werden aufgezeigt.

1.4 Inhalt der Begleit-CD

Auf der Begleit-CD befinden sich folgende Inhalte:

Im Ordner „Ausarbeitung“:

- Diese Diplomarbeit im PDF-Format

Im Ordner „Material“:

- diverse Datenblätter und technische Beschreibungen
- Bilder des HighTecBot und andere Bilder
- Schaltplan und Layout der Konverterplatine erstellt mit dem Programm „Eagle“ von CadSoft Computer GmbH

Im Ordner „Quellcode“:

- nachladbare Robotersteuerung-Task als Projekt von Codeblocks
- nachladbare Anwender-Task als Projekt von Codeblocks
- Programmbibliothek für die Schnittstellenkommunikation
- API-Dokumentation, erstellt mit dem Programm „doxygen“ im HTML-Format

1.5 Danksagung

Diese Diplomarbeit wurde von Herr Prof. Dr. Wolfgang Joppich vom Fachbereich Elektrotechnik, Maschinenbau, Technikjournalismus an der FH Bonn-Rhein-Sieg betreut und am Fraunhofer Institut für Intelligente Analyse und Informationssysteme geschrieben. Dabei gab es eine enge Zusammenarbeit zwischen dem Fraunhofer Institut IAIS, der Firma HighTec EDV-Systeme GmbH und der Hochschule für Wirtschaft und Technik in Saarbrücken.

Ich danke Herrn Prof. Dr. Wolfgang Joppich für die Überlassung des Themas, seine Anregungen und sein Interesse an der Arbeit. Herrn Prof. Dr. Marco Winker danke ich, dass er als Zweitprüfer zur Verfügung stand.

Mein besonderer Dank geht an die Gruppe Ausbildungsrobotik am Fraunhofer Institut IAIS, insbesondere an Frau Ulrike Petersen, Herrn Josef Börding, Herrn Björn Flintrop und Herrn Christoph Brauers, die mir mit vielen hilfreichen Tipps und Anregungen zur Seite standen.

Weiterhin danke ich der Firma HighTec EDV-Systeme, die mir die Hardware zur Verfügung gestellt hat und mir die Gelegenheit gab, mich in das eingebettete System und Echtzeitbetriebssystem einzuarbeiten. Hier gilt mein besonderer Dank Herrn Horst Lehser, Herrn Willi Theiß, Herrn Stefan Hittinger, Herrn Mario Copelli und weiteren Mitarbeitern, die mir zur Seite standen.

Ebenfalls danke ich Frau Prof. Dr. Martina Lehser und ihren Studenten, die bereit waren, die entwickelte Software zu testen und weiter zu verwenden.

2 Grundlagen

Die Erstellung einer Applikationsschnittstelle (API) erfordert nicht nur Kenntnisse in der Programmierung. Da das Umfeld aus Sensoren und Aktuatoren besteht, muss auch deren Funktionsweise verstanden und einbezogen werden. Zudem müssen die Eigenschaften der vorgegebenen Komponenten für diese Arbeit beachtet werden. Auf das Umfeld, die vorgegebenen Komponenten und die in der Vorarbeit erstellte mobile Roboterplattform HighTecBot wird in diesem Kapitel eingegangen.

2.1 Eingebettete Systeme

Ist ein informationsverarbeitendes System in ein Produkt oder ein größeres Gesamtsystem integriert, spricht man von einem eingebetteten System (engl. embedded system). Ein PC unterscheidet sich unter anderem von einem eingebetteten System darin, dass er mehrere standardisierte Ein- und Ausgabegeräte hat und somit eine allgemeine Lösung darstellt. Eingebettete Systeme sind häufig spezielle Lösungen für spezielle Probleme. Sie bestehen immer aus einem oder mehreren Prozessoren, einem Speicher für das Programm und je nach Aufgabe aus einer oder mehreren Schnittstellen nach außen. Durch eine optimale Anpassung an ihre Aufgabe können häufig teure Komponenten wie z.B. Eingabegeräte und Bildschirme eingespart werden. Prozessor, Speicher und Schnittstellen sind an das Einsatzgebiet angepasst. Dadurch entsteht trotz der speziellen Lösung ein kostengünstiges System. Ein eingebettetes System in einer Waschmaschine benötigt z.B. keine Festplatte oder einen hochauflösenden Bildschirm. Eingebettete Systeme im Automobil lösen Teilaufgaben wie z.B. Bordmonitor, Motorsteuerung oder Klimaanlage und kommunizieren über Datenbusse miteinander. Dabei ergänzen sie sich in ihren Funktionalitäten, so dass ein Gesamtsystem entsteht. Eingebettete Systeme müssen verlässlich und effizient sein und je nach Anforderung hohe Echtzeitbedingungen einhalten. Typischerweise sind sie reaktive Systeme, die kontinuierlich in einer Geschwindigkeit mit ihrer Umgebung interagieren, die von der Umwelt vorgegeben wird [19] (S. 4).

Für Entwicklungsaufgaben sind auf dem Markt eingebettete Systeme erhältlich, die neben leistungsstarken Mikrokontrollern eine Reihe von Schnittstellen zur Verfügung stellen, so genannte „evulation-boards“. Verwendet man solche Systeme zur Motorregelung, sind oft die mitgelieferten Leistungsendstufen auf das eingebettete System angepasst.

2.2 Echtzeitbetriebssysteme

Ein System reagiert in Echtzeit, wenn es innerhalb einer vorher definierten endlichen Zeit korrekt auf ein Ereignis reagiert. Man unterscheidet zwischen harter Echtzeit und weicher Echtzeit. Bei harter Echtzeit wird im Gegensatz zu weicher Echtzeit garantiert, dass innerhalb einer vorher definierten Zeit auf das Ereignis korrekt reagiert wird. Demzufolge kann es bei harter Echtzeit bei Verletzen der zeitlichen Frist zu ernsthaften Fehlfunktionen bis hin zum Totalausfall kommen. Bei weicher Echtzeit führt ein Verletzen der zeitlichen Frist zu einer Minderung der Leistungsfähigkeit [1] (S. 338f).

Ein Echtzeitbetriebssystem ist in der Lage, neben der Speicher- und Prozessverwaltung auch in Echtzeit Ereignisse zu verwalten. Dies stellt je nach Aufgabe hohe Anforderungen an das System und die Anwendungen.

Die digitale Regelung und Positionierung von permanenterregten Synchronmotoren erfordert schnelle Prozessoren, die in harter Echtzeit arbeiten. Für Stromregelungen sind Reaktionszeiten im µs-Bereich erforderlich, für Drehzahlregelungen im ms-Bereich. Solche Regelungen werden daher oft ohne Betriebssystem als einzige Anwendung auf einem Prozessor implementiert. Mit steigender Rechenleistung der Prozessoren und der Verbreitung von Mehrprozessorsystemen können zunehmend neben Regelalgorithmen auch andere Anwendungen ausgeführt werden. Hier ergeben sich aber Probleme mit der Speicherverwaltung und dem Scheduling der Anwendungen. Daher werden Betriebssysteme benötigt, die echtzeitfähig sind, also neben Speicherverwaltung und Anwendungsverwaltung eine ausreichend schnelle Reaktionszeit auf Ereignisse besitzen.

2.3 Sensorlose Positionserfassung

Unter Positionserfassung bei Motoren versteht man die Positionsdetektion der rotierenden Welle relativ zum feststehenden Gehäuse. Dazu werden zusätzliche Sensoren verwendet, die z.B. Markierungen auf der Welle zählen. Methoden zur sensorlosen Positionserfassung bei Motoren haben gegenüber den sensorbehafteten Methoden einige Vorteile. Neben Kosten, Platz und Gewichtsersparnis verringert sich auch der Wartungsaufwand. Zusätzliche Bauteile wie z.B. ein Resolver, Encoder oder Hallsensor entfallen. Daher werden solche Verfahren oft dort eingesetzt, wo hohe Dynamik, hohe Positioniergenauigkeit sowie Gewicht- und Platzminimierung vom Motor gefordert werden. Auch in der Robotik werden diese Forderungen gestellt. Zurzeit gibt es mehrere bekannte Verfahren zur sensorlosen Positionsermittlung bei Motoren.

Wird ein Dreiphasenmotor nur mit zwei Phasen angesteuert, kann an der dritten Phase über die aktuelle Induktion die Rotorlage ermittelt werden. Nachteilig ist dabei, dass die Position nur bei Bewegung der Rotorwelle ermittelt werden kann. Im Stillstand ist keine Auswertung möglich. Zudem kann der Motor nur mit zwei Drittel seiner Leistung betrieben werden, da nur zwei Phasen Strom führen.

Eine weitere Methode, die Position des Rotors sensorlos zu ermitteln, besteht darin, den Strom in allen drei Phasen zu messen. Dazu ist ein Mess-Shunt (niederohmiger Nebenschlusswiderstand) in mindestens zwei der Phasen nötig. Zudem wird die aktuelle Spannung benötigt. Ist die Rotorlage vor der Messung bekannt, kann aus diesen Werten die aktuelle Position ermittelt werden [23] (S. 293ff). Der Nachteil dieser Methode ist, dass Mess-Shunts zur Strommessung benötigt werden und dass die vorherige Position bekannt sein muss.

Das in dieser Arbeit vorgegebene und angewendete Verfahren zur sensorlosen Rotorlagebestimmung wurde von Dr. Strothmann, einem der Geschäftsführer der Firma HighTec EDV-Systeme GmbH entwickelt. Es nennt sich virtuHall [24]. Der Name wird von „**virtu**eller **Hall**sensor“ abgeleitet. Das daraus entstandene Patent hält die Firma Elmos Semiconductor AG [8]. VirtuHall nutzt eine Spannung, die am Sternpunkt detektiert werden kann und setzt sie in Bezug zu den aktuellen Schaltzuständen der Umrichterbrücke. Für diese Art der Positionserfassung muss ebenfalls die Position des Rotors vor der Messung bekannt sein. Diese wird in einer Initialisierungsphase durch einen Puls erkannt, der so genannten Nord-Süd-Erkennung. Der Vorteil dieser Methode liegt darin, dass die Rotorlage auch bei nicht angesteuertem Motor erkannt werden kann. Nachteilig ist, dass der Sternpunkt aus dem Motorgehäuse herausgeführt sein muss. Die virtuHall-Funktionalität ist als Software in einer Task im Basissystem des eingebetteten Systems eingebunden. Weiteres hierzu wird im Abschnitt 2.6.2 „Software“ erläutert.

2.4 Digitaler Regelalgorithmus

Wie im Abschnitt 1.1 bereits angedeutet, kann die mobile Roboterplattform HighTecBot über eine PWM-Vorgabe angesteuert und die Positionen der Räder ermittelt werden. Um eine vorgegebene physikalische Größe tatsächlich zu erreichen, wird eine Regelung mit Ausgleich benötigt. Diese wird als digitaler Regelalgorithmus implementiert. Der Unterschied zu einer analogen Regelung besteht darin, dass die digitale Stellgröße in eine analoge Stellgröße und die analogen Istwerte in digitale Istwerte umgewandelt werden müssen. Die Analog-Digitalwandlung geschieht dabei mit Abtast-Haltegliedern, die in der mathematischen Beschreibung eines digitalen Regelkreises nach Große und Schorn [10] (S. 335f) zu einer Totzeit führen. Um eine digitale Regelung trotzdem wie eine analoge Regelung behandeln zu können, muss sie ein quasikontinuierliches Verhalten aufweisen. Dies ist der Fall, wenn die Abtastfrequenz deutlich höher liegt als die Abarbeitungszeit des Regelalgorithmus. In diesem Fall kann die Totzeit der Abtast-Halteglieder vernachlässigt und der Regelkreis mit den aus analogen Regelungen bekannten Parametrierungsregeln eingestellt werden [18] (S. 471ff).

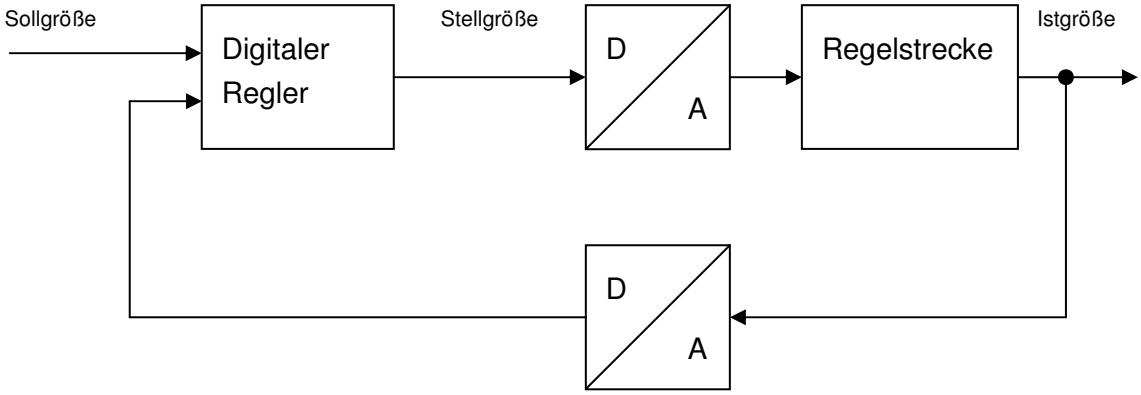


Abbildung 3: Komponenten eines digitalen Regelkreises

Bei einer digitalen Regelung muss gewährleistet sein, dass die Abtastung des Istwertes immer zu einem korrekten Ergebnis führt. Dies ist der Fall, wenn die Zeit zwischen den Abtastvorgängen bekannt ist und einbezogen wird. Es gibt zwei Möglichkeiten dies umzusetzen [5] (S. 395ff).

1. Zeiterfassung zwischen zwei Abtastpunkten.

   Hierbei wird jedes Datum mit einem Zeitstempel versehen. Nach einem Abtastvorgang wird ein Timer gestartet, der bei einem erneuten Abtastvorgang ausgelesen und neu gestartet wird. Damit kann bei jeder Istwertauswertung mithilfe des Zeitstempels der korrekte Wert ermittelt werden. Diese Methode erlaubt es, variable Programmlaufzeiten zwischen den Abtastzeitpunkten zu realisieren. Wird die Zeit zwischen zwei Abtastpunkten zu groß, kann es z.B. durch Timerüberlauf zu Fehlern kommen. In einem kritischen Zeitraum zwischen dem Auslesen des Timers und dem Abtasten darf es zu keiner variablen Laufzeit des Programms kommen (z.B. durch Interrupts), da diese Zeit nicht durch den Timer erfasst werden kann und es zu Folgefehlern kommt.

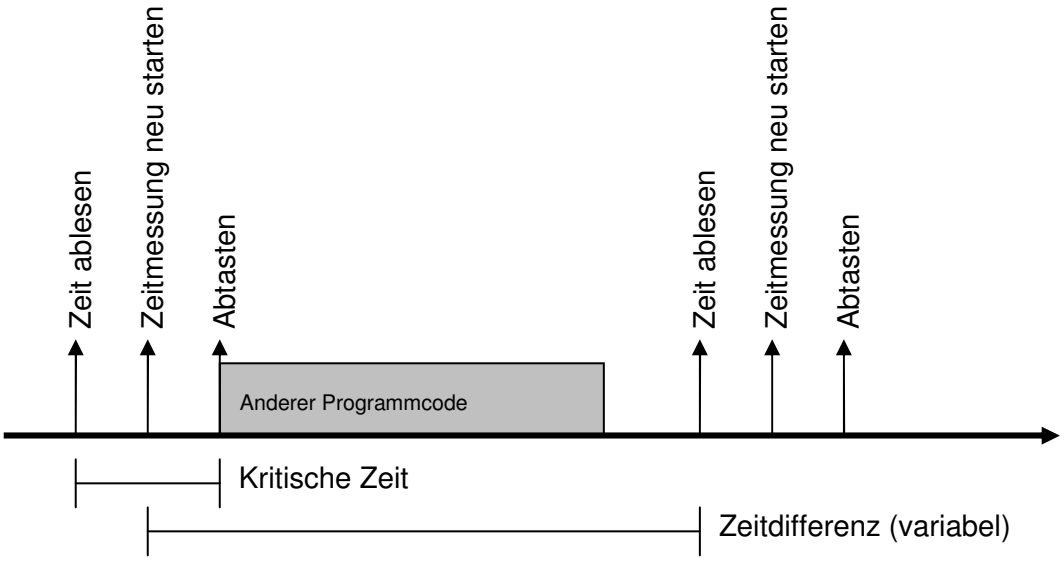


Abbildung 4: Istwerterfassung mit variabler Zeiteinteilung

2. Abtastung zu festen Zeiten

Diese Methode setzt voraus, dass eine bekannte periodische Zeit genau eingehalten werden kann. Nach dem periodischen Signal wird sofort der Istwert abgetastet. Die kritische Zeit liegt zwischen einem periodischen Signal und dem Abtasten. Da es keine variable Zeit zu verrechnen gibt, ist die Auswertung der Istwerte einfach. Jedoch muss sichergestellt werden, dass die periodische Zeit genau eingehalten werden kann, da es ansonsten zu Fehlern kommt. Dies kann von einem Echtzeitbetriebsystem unterstützt werden.

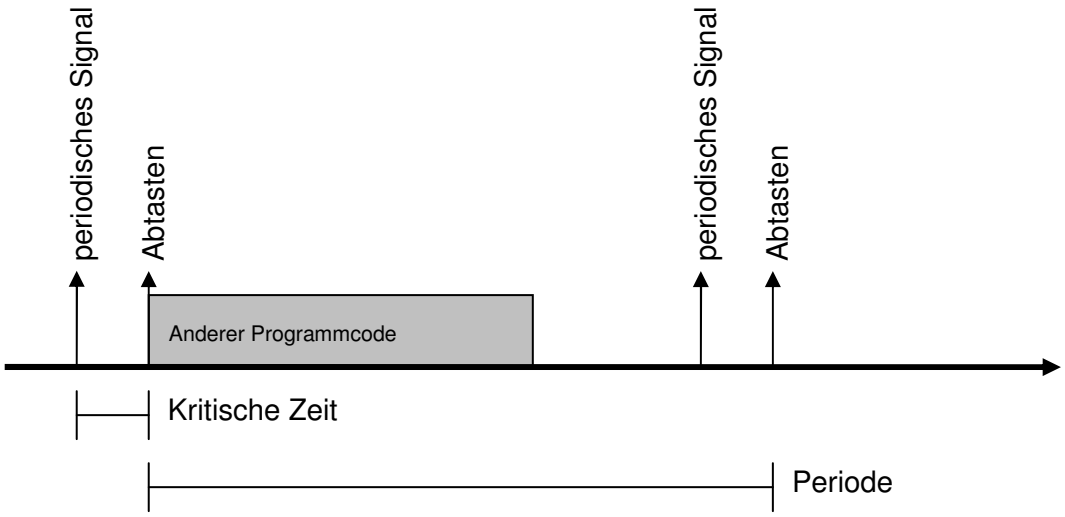


Abbildung 5: Istwerterfassung mit periodischem Signal

Elektrische Antriebe, die eine hohe Positioniergenauigkeit und Dynamik aufweisen müssen, werden häufig mit einer Kaskadenstruktur geregelt. Dabei sind mehrere Regelkreise ineinander verschachtelt. In einem unteren Kreis wird das Drehmoment geregelt. Ein darüber angeordneter Kreis regelt die Winkelgeschwindigkeit, bzw. die Drehzahl. Optional kann in einem weiteren überlagerten Kreis die Position geregelt werden. Vorraussetzung für eine solche Kaskade ist, dass der jeweils unterlagerte Kreis schneller arbeitet als der überlagerte Regelkreis und dass die entsprechenden Istwerte verfügbar sind. Der Vorteil einer Kaskadenstruktur ist, dass Störungen bereits in den unteren Kreisen ausgeregelt werden, ohne dass sie sich in den überlagerten Kreisen auswirken. In den unterlagerten Kreisen kann eine Begrenzung der Werte z.B. des Drehmomentes vorgenommen werden, um den Motor vor Schäden durch Überlast zu schützen [18] (S. 701f).

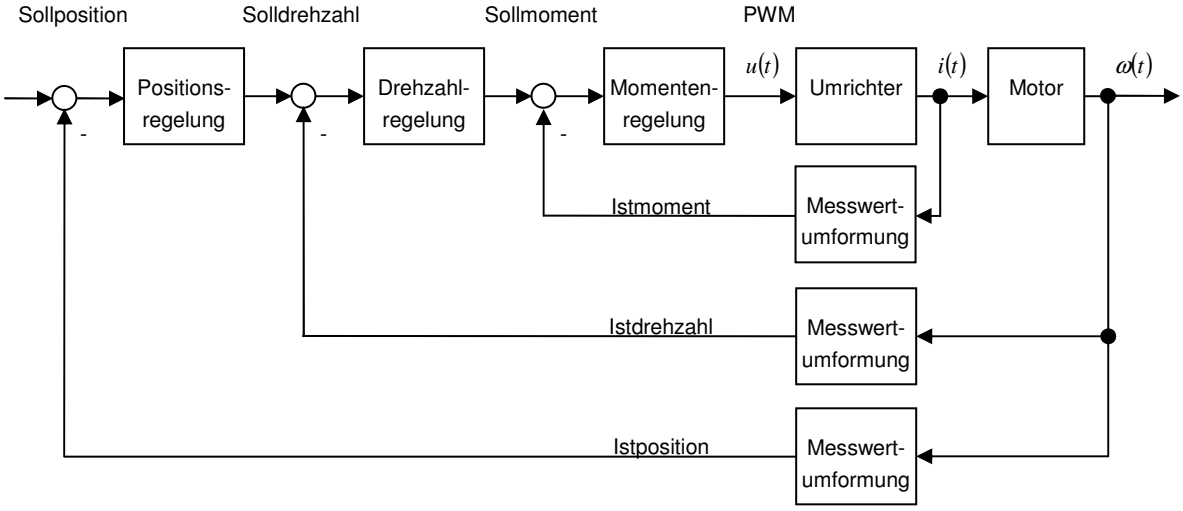


Abbildung 6: Schema einer Kaskadenregelung

Ein in der Praxis häufig eingesetzter Regler ist der PID-Regler. PID steht für Proportionalteil, Integralteil und Differentialteil. Ein PID-Regler setzt sich aus allen drei Reglerteilen zusammen. Je nach Anforderung können aber auch Reglerteile weggelassen werden, z.B. bei einem PI- oder P-Regler. Allen Reglern ist gemein, dass sie anhand der Regeldifferenz (Differenz zwischen Soll- und Istwert) den Sollwert beeinflussen.

Ein proportionaler Regler multipliziert die Regeldifferenz mit einem entsprechenden Faktor, so dass der Istwert erreicht wird. Er ist nicht in der Lage eine bleibende Regeldifferenz auszugleichen.

Ein integraler Regler integriert ständig die Regeldifferenz auf. Das Ergebnis beeinflusst die Stellgröße so, dass der gewünschte Istwert erreicht wird. Ein integraler Regler ist in der Lage eine bleibende Regeldifferenz auszugleichen.

Ein differenzialer Regler erfasst die Änderung der Regeldifferenz. Diese ist im Augenblick des Aufschaltens der Störgröße sehr groß, weswegen sofort eine starke Veränderung der Stellgröße erreicht wird. Man benutzt D-Reglerteile, um die Geschwindigkeit des Ausregelvorganges zu verbessern. Jedoch destabilisiert dieser das System, so dass es zum Schwingen kommen kann. Bei verrauschten Eingangssignalen ist ein D-Reglerteil daher ungünstig. Dörrscheidt und Latzel beschreiben die Reglerteile in [7] (S. 118ff).

2.5 Vorgegebene Komponenten

2.5.1 Das eingebettete System EasyRun-TC1796

Für diese Arbeit ist das eingebettete System EasyRun-TC1796 der Firma HighTec EDV-Systeme GmbH vorgegeben. Dieses System besteht aus zwei Teilen, einem Mikrokontroller-Board und einer Leistungsendstufe. Die Leistungsendstufe ist auf die Motoren angepasst, die in Abschnitt 2.5.3 „Permanenterregte Synchronmotoren“ beschrieben sind. Die Betriebsspannung liegt bei 4,5 – 28,5 V. Daher kann das Board mit der nominellen Versorgungsspannung der beiden Akkus des HighTecBot von 24 V betrieben werden. Die interne Spannung auf dem Board beträgt 3,3 V. Das Board stellt unter anderem folgende Schnittstellen zur Verfügung:

- Ethernet
- USB 2.0
- RS232
- LIN-Bus
- 4 x CAN-Bus
- JTAG

Zudem kann eine SD-Speicherkarte auf dem Board eingeschoben werden. Zusätzlich sind 10 frei konfigurierbare digitale Ein- und Ausgänge und 8 analoge Eingänge mit einer Auflösung von 12 Bit verfügbar [13].

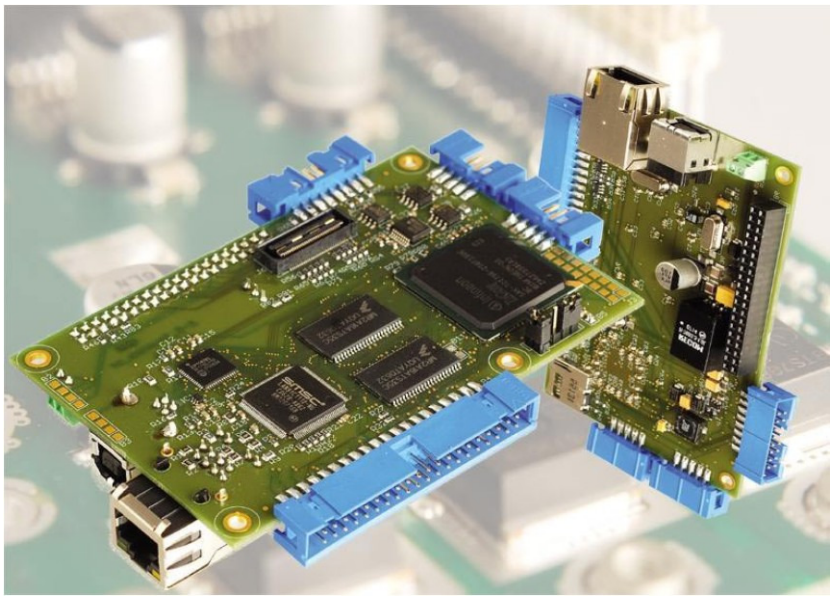


Abbildung 7: EasyRun-TC1796 der Fa. HighTec EDV-Systeme GmbH [16]

Auf dem Board ist der Mikrokontroller TriCore 1796 von der Firma Infineon Technologies AG eingebaut [14]. Er verfügt über einen 2 MB großen programmierbaren Flash-Speicher und einen 150 MHz 32 Bit Prozessor. Neben einer Memory Protection Unit, die hardwareseitige Speicherschutzmechanismen enthält, ist auch eine Floating Point Unit vorhanden, mit der Gleitkommazahlen verrechnet werden können. Der Datentyp `double` wird durch die Hardware nicht unterstützt. Beim Programmieren und Kompilieren des Programmcodes für diesen Prozessor ist darauf zu achten, dass alle Gleitkommazahlen im Datentyp `float` vorliegen, um eine effiziente Programmausführung zu erhalten. Der Datentyp `double` wird durch eine Emulationsbibliothek des Compilers behandelt.

2.5.2 Echtzeitbetriebssystem PXROS-HR

In dieser Arbeit wird das Echtzeitbetriebssystem PXROS-HR (portable extendible real-time operating system – High Reliability) der Fa. HighTec EDV-Systeme GmbH verwendet. Weiterführende Informationen finden sich im „Quickstart“ [11] und „Users Guide“ [12] für PXROS. Wie in Kapitel 2.6.2 beschrieben, ist dieses Betriebssystem für die vorliegende Arbeit vorgegeben. Es ist mit dem eingebetteten System EasyRun-TC1796, das in Abschnitt 2.5.1 dargestellt wurde, bereits mehrfach in der Praxis eingesetzt worden.

In PXROS-HR werden prioritätsbasiert eigenständige Programmteile, die so genannten Tasks, verwaltet. Jeder Task erfüllt bestimmte Aufgaben. Es gibt einen Task für die Kommunikation über TCP/IP, einen Task für die virtuHall-Funktionalitäten, Tasks, die das Debuggen erleichtern usw. Der Betriebssystemkern verwaltet die verschiedenen Tasks gemäß ihrer Priorität und weist ihnen Prozessorzeit zu. Dabei wird ein prioritätsbasiertes verdrängendes Scheduling angewandt [11] (S.27f). Wird ein höher priorisierter Task aktiv, dann wird ein niedriger priorisierter Task unterbrochen und kann erst dann wieder aktiv werden, wenn er der wartende Task mit der höchsten Priorität ist. Das Betriebsystem selber hat dabei eine höhere Priorität als die Tasks und kann, um die Echtzeit zu gewähren, nur von Hardware-Interrupts unterbrochen werden. Die Interrupts sind ebenfalls mit unterschiedlichen Prioritäten versehen. Die folgende Grafik verdeutlicht die Prioritäten der Programmteile.

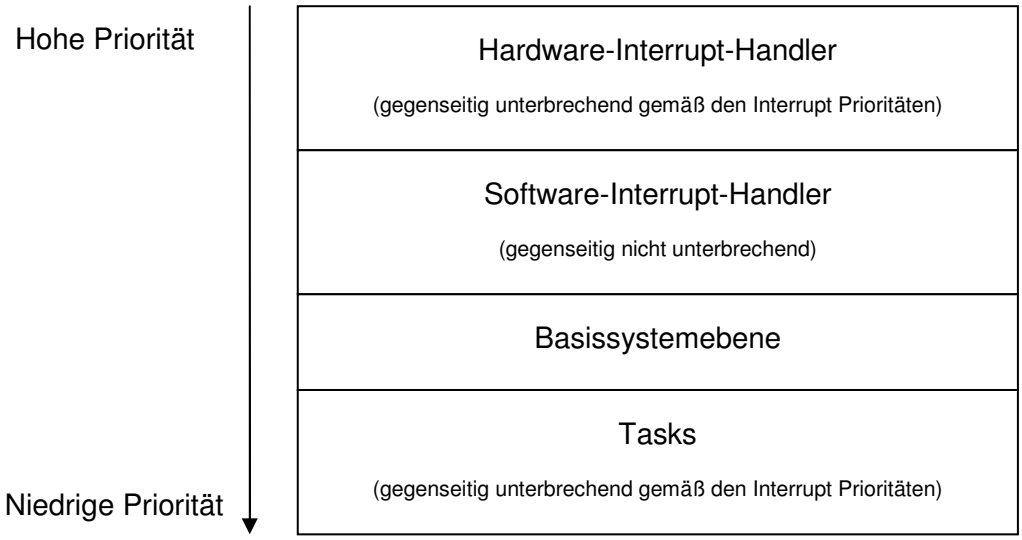


Abbildung 8: Prioritäten bei PXROS-HR [12]

Ein Task kann drei verschiedene Zustände einnehmen. Ist er im Zustand „wartend“, wartet er auf externe Ereignisse oder Nachrichten. Trifft ein solches Ereignis bzw. eine Nachricht ein, wechselt er in den Zustand „bereit“. Ist er der Task mit der höchsten Priorität in diesem Zustand, wechselt er in den Zustand „aktiv“ und bekommt vom Betriebssystem Prozessorzeit zugewiesen. Der Task wird abgearbeitet, bis ein höher priorisierter Task „bereit“ wird oder er seinen Programmteil fertig abgearbeitet hat. Dann haben andere Tasks die Möglichkeit, den Prozessor zu belegen.

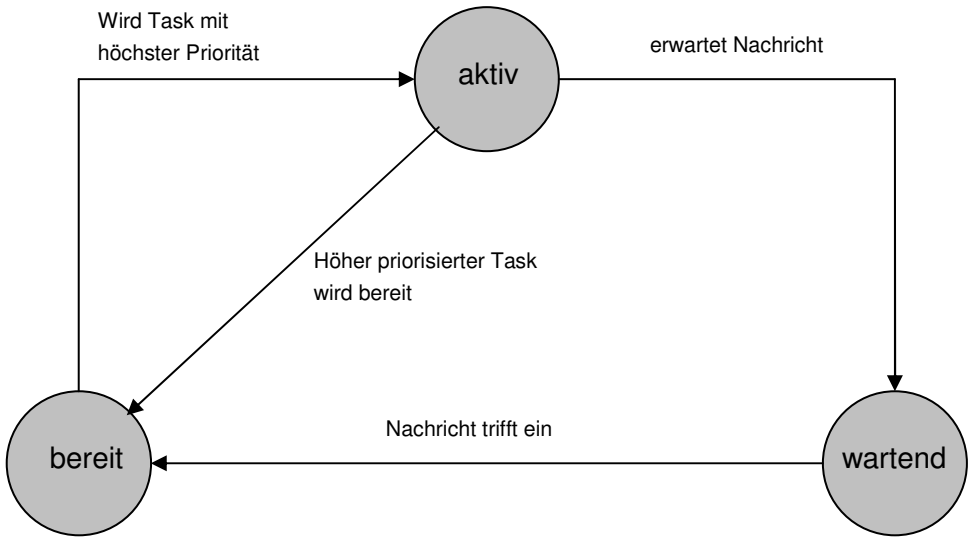


Abbildung 9: Prioritätsbasiertes verdrängendes Scheduling der Tasks [12]

Neben der Verwaltung der Tasks werden auch der Speicher und die Ressourcen, wie z.B. IO-Ports, vom Betriebssystem verwaltet. Wie in Kapitel 2.5.1 erwähnt, wird das eingebettete System EasyRun-TC1796 verwendet, das mit dem Prozessor TriCore 1796 von Infineon arbeitet. Dieser Prozessor besitzt eine Memory Protection Unit, die es erlaubt, den Speicher in bestimmten Betriebsmodi zu schützen. Mit diesen Modi arbeitet PXROS-HR. Es gibt drei verschiedene Modi. Den „Supervisor Mode“, in dem ein Benutzer auf alle Ressourcen des Prozessors Zugriff hat, den „User Mode 1“, in dem ein Benutzer nur eingeschränkten Zugriff auf Ressourcen hat und den „User Mode 0“, in dem ein Benutzer keinen Zugriff auf Ressourcen hat. Als Anwendungsprogrammierer arbeitet man immer im User Mode 0, da alle benötigten Ressourcen vom Basissystem bereitgestellt werden. Damit ist es einem Anwender nicht möglich, durch fehlerhafte Programmteile Schäden am Basissystem zu verursachen. Allenfalls der eigene Programmteil kann durch Fehler beeinträchtigt werden.

Der Datenaustausch zwischen den Tasks erfolgt über Nachrichten, die sich die Tasks gegenseitig zuschicken können, oder über so genannte Handler, die ein Ereignis auslösen, sobald ein Hardware- oder Software-Interrupt ausgelöst wurde. Die Bedeutung eines Ereignisses kann zwischen Empfänger und Sender beliebig definiert werden.

Zurzeit besteht das Betriebsystem PXROS-HR aus einem Basissystem, das mit beliebigen Zusatzfunktionen erweitert werden kann, indem ein Task mit entsprechender Funktionalität hinzugefügt wird. Dazu wird der nichtflüchtige Flash-Speicher des EasyRun-TC1796 verwendet. Die wichtigsten verwendeten Tasks sind:

- TCP für Ethernet-Kommunikation (Task nachladen, debuggen)
- PX-View für die visuelle Ausgabe des Laufzeitverhaltens
- PX-logger für die visuelle Ausgabe beliebiger Variablen
- virtuHall-API für die Ansteuerung der beiden Motoren

Anwender können bis zu zwei nachladbare Tasks in einen flüchtigen RAM-Speicher laden. Diese Tasks haben den gleichen Aufbau wie die Tasks im Basissystem, befinden sich aber im User Mode 0 und damit in einem gekapselten Bereich. Sie werden vom Betriebssystem genau so behandelt wie Tasks im Basissystem.

Mit dem Betriebssystem PXROS-HR wird bereits eine Motorsteuerung in einem Golf-Caddy realisiert. Der Golf-Caddy besitzt einen drucksensitiven Griff, der erkennt, ob und wie stark der Golf-Caddy gezogen wird. Zwei Elektromotoren unterstützen die Bewegung soweit, dass ein Golfspieler kaum Mühe hat den Golf-Caddy zu bewegen. Dazu wird die virtuHall-API des Betriebssystems genutzt, die bereits eine Ansteuerung der Motoren zulässt.

2.5.3 Permanenterregte Synchronmotoren

Ein einfacher permanenterregter Synchronmotor besteht aus einem Rotor mit einem Permanentmagneten und einem Stator mit Wicklungen. In den Wicklungen wird ein frequenzabhängiges Drehfeld erzeugt, dem die Permanentmagneten auf dem Rotor folgen. Es entsteht eine Drehbewegung. Durch moderne Umrichtertechnik kann das Drehfeld in seiner Frequenz und somit die Drehzahl des Motors gesteuert werden. Bei hoher Belastung des Motors stellt sich ein Verdrehwinkel des Rotors zum Drehfeld ein. Wird dieser Winkel zu groß, fällt das mittlere Moment am Rotor auf Null und der Motor bleibt stehen. Da die Frequenz durch den Umrichter veränderbar ist, kann der Motor in diesem Fall wieder angefahren werden.

Vor ca. 20 Jahren wurden permanenterregte Synchronmotoren überwiegend als Generatoren in Kraftwerken eingesetzt. Da die Umrichtertechnik zu dieser Zeit noch nicht so weit fortgeschritten war wie heute, liefen diese Motoren immer mit einer festen netzfrequenzabhängigen Drehzahl. Heute können diese Motoren vielseitiger eingesetzt werden, da durch die Frequenzumrichter beliebige Drehzahlen angefahren werden können. Die Anzahl der Spulen und Polpaare muss nicht mehr der Drehzahl angepasst werden, sondern kann auf Leistung und Wirkungsgrad hin optimiert werden. Permanenterregte Synchronmotoren sind wartungsarm und haben einen hohen Wirkungsgrad. Einsatzgebiete sind z.B. Bahnantriebe, Werkzeugmaschinen und Roboterantriebe.

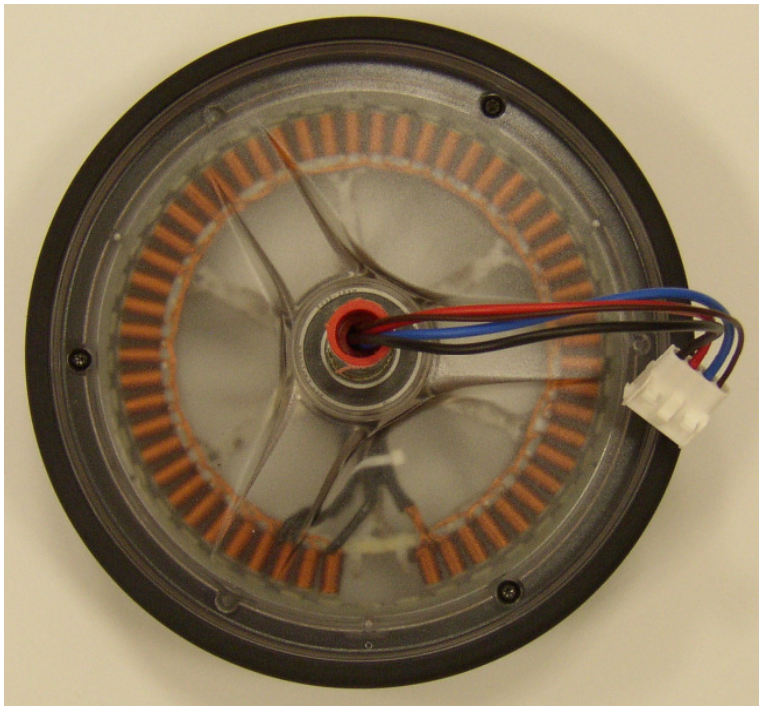


Abbildung 10: Rad, Motor und Sensor in einem Bauteil

Für diese Arbeit steht eine besondere Bauform eines permanenterregten Synchronmotors zur Verfügung. Der Stator liegt innen und ist mit 48 Spulen in Sternschaltung bestückt. Der Stator ist fest mit der Radwelle verbunden. Der Rotor liegt außen und bildet die Lauffläche des Rades. Auf seiner Innenseite sind 40 Permanentmagneten angebracht. Die Welle des Rades, die als Hohlwelle ausgeführt ist, wird in einem Lagerblock festgeklemmt und ist statisch mit dem Rahmen des Roboters verbunden. Das einzige bewegliche Teil ist der Rotor, der zugleich Reifen des Rades ist. Für den elektrischen Anschluss des Motors sind die drei Phasen und der Sternpunkt in der Hohlwelle nach außen geführt. Die drei Phasen dienen zur Ansteuerung des Motors. Das Sternpunktsignal wird benötigt, um mithilfe des virtuHall-Verfahrens, das in Kapitel 2.3 beschrieben ist, die Position des Rotors zu ermitteln.

Elektrische Leistung	130 W
Maximale Drehzahl	Ca. 95 U/min
Raddurchmesser	250 mm
Polpaarzahl	20
Wicklungsart	Sternwicklung

Tabelle 1: Technische Daten eines Motors

2.6 Die mobile Roboterplattform HighTecBot

2.6.1 Hardware

Auf der Grundlage des ProfiBot-Baukastensystems wurde als Vorarbeit zu dieser Diplomarbeit im Rapid-Prototyping-Verfahren der in Kapitel 1.1.2 erwähnte HighTecBot aufgebaut. Ziel war es dabei, das Baukastenprinzip beizubehalten, um Änderungen am Rahmen einfach und schnell vornehmen und weitere Sensoren einfach montieren zu können. Zudem wurden überwiegend die gleichen Profillängen und Schraubentypen wie beim ProfiBot-Baukasten verwendet. Es wurde auf einen einfachen, Material und Gewicht sparenden Aufbau geachtet.

Der Rahmen wurde an die größeren Räder der Fa. HighTec EDV-Systeme GmbH angepasst. Die dadurch entstehende größere Bodenfreiheit erfordert etwas größere bzw. höhere Lenkrollen. Die Schublade, die beim ProfiBot für den Steuerlaptop benötigt wurde, entfällt. Durch den gewonnenen Platz wird kein zusätzlicher Sensor-Aufbaurahmen benötigt. Die Halterungen für den Motorkontroller und die Motoren, die Lagerblöcke und die Ketten entfallen ebenfalls. Anders als beim ProfiBot gibt es vier Schaltleisten, die auf einer einheitlichen Höhe von 150 mm angebracht sind und alle Seiten des Fahrzeugs abdecken. Die Schaltleisten werden nicht mithilfe einer Auswertelektronik, sondern mit einer zusätzlichen Platine ausgewertet. Die Entwicklung ist Teil dieser Arbeit und wird in Kapitel 3.1 beschrieben. Beide Baukastensysteme verfügen über die gleiche Stromversorgungseinheit und einen Not-Aus-Schalter.

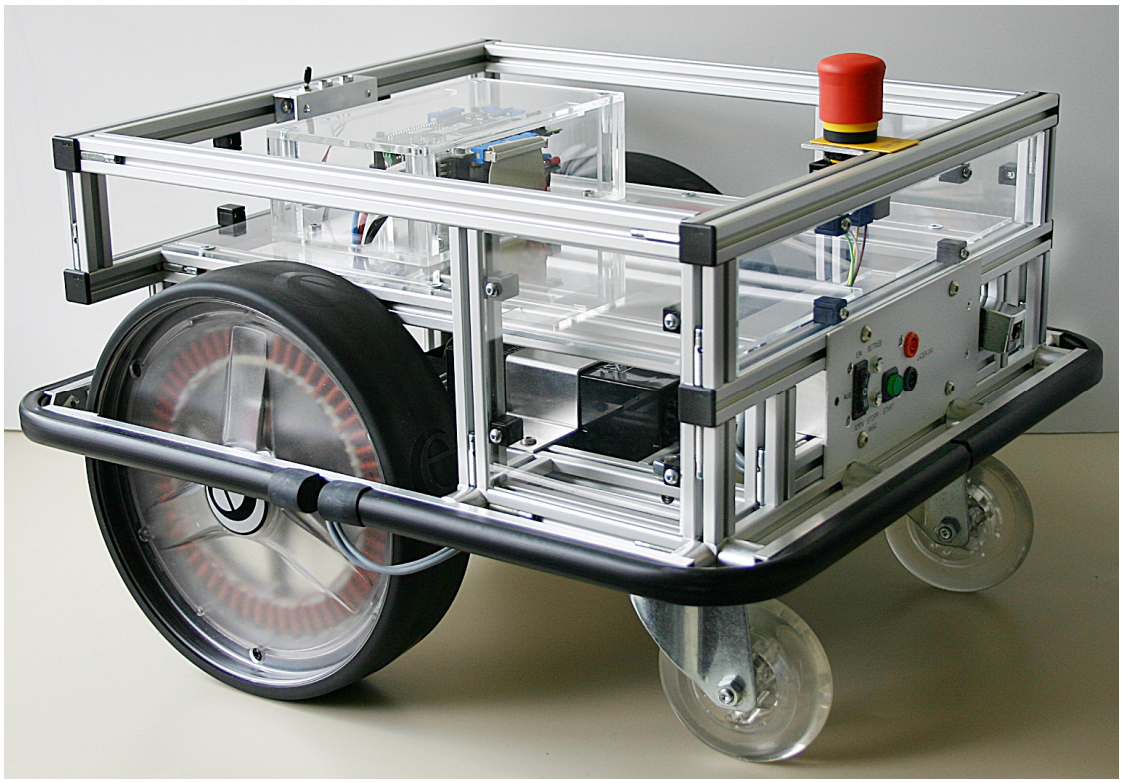


Abbildung 11: Die mobile Roboterplattform HighTecBot (Foto: IAIS)

Als weitere Hardware steht das Kontrollerboard EasyRun-TC1796 mit angefügter Leistungsendstufe für zwei Motoren und zwei Radnabenmotoren als permanenterregte Drehstrom-Synchronmotoren in Sternwicklung der Fa. HighTec EDV-Systeme GmbH zur Verfügung. Diese Komponenten werden im Kapitel 2.5.1 „Das eingebettete System EasyRun-TC1796“ und 2.5.3 „Permanenterregte Synchronmotoren“ vorgestellt und beschrieben.

2.6.2 Software

Um eine Applikationsschnittstelle zu entwickeln, sind verschiedene Tools notwendig. Für diese Arbeit wurde die Entwicklungsumgebung Codeblocks verwendet [6]. Codeblocks ist eine frei verfügbare Entwicklungsumgebung, die auf verschiedenen Betriebssystemen ausgeführt werden kann. Die Fa. HighTec EDV-Systeme GmbH hat Codeblocks durch das Einbinden von nützlichen Tools für ihre Anwendung erweitert. Mit dem Debugger „TriCoreInsight“ können zur Laufzeit des Basissystems bis zu zwei Tasks nachgeladen und im laufenden Betrieb getestet werden.

Für das eingebettete System gibt es bereits das Echtzeitbetriebssystem PXROS-HR, das mit einer bestimmten Konfiguration an Tasks bereitgestellt wurde. Folgende wichtige Tasks sind im Basissystem bereits eingebunden:

- PxLogger
- PxView
- Bootloader
- TCP/IP-Kommunikation
- virtuHall-API

PxLogger ist eine Erweiterung des „Osqoop“ open source Ozilloskops für PCs [20]. Es wurde von der Firma HighTec EDV-Systeme GmbH für PXROS-HR angepasst, damit es als eigener Task im Basissystem integriert ist. Mit ihm können alle Variablen zur Laufzeit des Programms aufgezeichnet und angezeigt werden. Die aufgezeichneten Daten werden in einer Datei gespeichert. Für die Benutzung des Oszilloskops ist eine Verbindung über Ethernet nötig.

PxView ermöglicht es, einen Puffer auszuwerten, in dem Daten über die Laufzeit einzelner Funktionen hinterlegt werden. Damit kann zur Laufzeit eines Programms überprüft werden, wie gut Echtzeitbedingungen eingehalten werden. Zudem werden sämtliche Taskwechsel sichtbar, die das Betriebssystem durchführt.

Der **Bootloader** ist ein Programmteil, der es ermöglicht, in einer Bootsequenz nach dem Einschalten das Basissystem zu erneuern. Damit können Programmteile in den Flash-Speicher des Systems geladen werden. Die von TriCoreInsight nachgeladenen Tasks können dann über Taskkommunikation auf die Programmteile im Basissystem zugreifen.

Mithilfe der **TCP/IP-Kommunikation** ist es möglich, Tasks nachzuladen oder die Programme PxLogger und PxView zu verwenden. TCP/IP ist daher zurzeit die wichtigste Schnittstelle des eingebetteten Systems.

Die **virtuHall-API** ermöglicht eine einfache Ansteuerung beider Motoren mithilfe einer PWM-Vorgabe. Dabei werden die Motoren nur angesteuert. Es findet keine Regelung hinsichtlich der Drehzahl statt. Zudem ist es möglich, Motorparameter wie z.B. Position, Phasenlage, Phasenstrom und Spannung abzufragen. Jede aufgerufene Funktion der virtuHall-API hat eine Intertaskkommunikation zur Folge. Alle Funktionen der virtuHall-API haben das Präfix „vh“, um die Identifikation zu erleichtern. Die Rückgabe von Parametern erfolgt nicht über den Returnwert der Funktion, sondern durch Übergabeparameter auf Zeiger, in der Syntax am Stern (*) erkennbar. Der Returnwert der Funktion dient als Fehlercode. Gibt die Funktion eine Null zurück, konnte sie erfolgreich abgearbeitet werden. Eine Zahl größer Null kann einem Fehlercode zugeordnet werden.

Die wichtigsten Funktionen der virtuHall-API sind im Folgenden aufgelistet:

Funktion	**Übergabeparameter**	**Kurzbeschreibung**
`vhSetPower()`	**`motor_id_t`** `id`, **`short`** `pwm`, **`short*`** `speed`	Setzt für den Motor mit der Identifikation `id` die PWM-Vorgabe und gibt im Parameter `speed` einen undefinierten Wert der Winkelgeschwindigkeit zurück.
`vhGetPhase()`	**`motor_id_t`** `id`, **`unsigned short*`** `phase`	Gibt die Phaseninformation des Motors mit der Identifikation `id` zurück. Pro Polpaar entsteht eine elektrische Phase, die mit 12 Bit aufgelöst ist.
`vhGetCVT()`	**`motor_id_t`** `id`, **`short*`** `current1`, **`short*`** `current2`, **`short*`** `current3`, **`short*`** `voltage`	Gibt den Phasenstrom der drei Phasen des Motors mit der Identifikation `id` und die Betriebsspannung zurück.
`vhSetPortType()`	**`motor_id_t`** `id`, **`short`** `port`, **`short`** `direction`, **`short`** `pullup`, **`short`** `interrupt`	Konfiguriert den I/O-Pin mit der Nummer `port`. Die Ports entsprechen den Nummern 0 – 7 und den Pins 23 - 30 auf dem 40-poligen Stecker des Boards.

`vhSetPortIO()`	**`motor_id_t`** `id`, **`unsigned short`** `value`	Im Parameter `value` wird eine Bitmaske übergeben. Die acht niederwertigsten Bits entsprechen den Portnummern 0 – 7 und geben an, wie der Ausgang gesetzt wird.
`vhGetPortIO()`	**`motor_id_t`** `id`, **`unsigned short*`** `value`	Im Parameter `value` wird eine Bitmaske übergeben. Die acht niederwertigsten Bits entsprechen den Portnummern 0 – 7 und haben den Wert der Eingänge.
`vhGetAnalogPort()`	**`motor_id_t`** `id`, **`unsigned short*`** `result`	Der Parameter `result` muss ein Feld mindestens der Größe acht sein. In dem Feld werden die acht Werte der Analogen Eingänge 0 – 7 übergeben.

Tabelle 2: Die wichtigsten virtuHall-Funktionen

Mit der Funktion `vhSetPower()` kann die PWM eines Motors auf einen vorgegebenen Wert gesetzt werden. Der Motor wird solange mit dem Wert angesteuert, bis eine andere Vorgabe erfolgt. Neben der PWM-Vorgabe gibt die Funktion den Wert `speed` zurück, der eine undefinierte Auskunft über die aktuelle Geschwindigkeit des Motors gibt. Damit können mit einem Funktionsaufruf alle relevanten Werte für die Regelung eines Motors übergeben werden. Mit weiteren Funktionen kann auf die Ein und Ausgabeschnittstelle des Mikrokontrollers zugegriffen werden (I/O-Ports), um Werte einzulesen oder auszugeben.

2.6.3 Vergleich der Baukastensysteme

Zusammenfassend werden die wichtigsten Veränderungen vom ProfiBot zum HighTecBot in einer Tabelle dargestellt. Das Baukastensystem HighTecBot steht in der beschriebenen Weise als Grundlage für diese Arbeit zur Verfügung.

	ProfiBot	**HighTecBot**
Steuerplattform	Laptop	Eingebettetes System
Steuerprogramm	Graphisches Programm als Signalgraph	Programmteil als eigenständige Task in C
Motorkontroller	TMC 2000	Im eingebetteten System integriert
Antriebseinheit	maxon motor mit Getriebe	Radnabenmotor ohne Getriebe
Positionserfassung der Räder	Encoder von maxon motor	virtuHall Verfahren, kein Sensor nötig
Schaltleisten	Zwei, vorne und hinten, benötigt Auswertelektronik	Vier, an jeder Ecke eine, benötigt keine Auswertelektronik

Tabelle 3: Vergleich ProfiBot und HighTecBot anhand der wichtigsten Veränderungen

Die wichtigsten Eigenschaften des HighTecBot sind die einfache Systemarchitektur bestehend aus Stromversorgung, eingebettetem System und sensorlosen Antriebsrädern, Reduzierung auf eine Stromversorgung und deutliche Kosten- und Gewichtsersparnis. Durch den einfachen Aufbau ist die Plattform für die berufliche Mechatronik ausbildung uninteressanter geworden. Dagegen bietet die Möglichkeit der hardwarenahen und praxisorientierten Programmierung den Einsatz als Lehrobjekt in Hochschulen an.

3 Entwicklung und Test

In diesem Kapitel wird beschrieben, wie die Ziele der Arbeit und die damit verbundenen Anforderungen umgesetzt werden. Die Ziele lassen sich in zwei Teilziele aufteilen. Die zu entwickelnde Hardware muss die Möglichkeit bieten, weitere Sensoren, die Schaltleisten, einen Schalter für den Programmstart und Status-LEDs anzuschließen. Die Software soll so erweitert werden, dass eine Schnittstelle zur Anwendungsprogrammierung (API) zur Verfügung steht, mit der der Roboter gesteuert werden kann. Die Vorgaben sollen in physikalischen Größen eingegeben werden können. Beim Erreichen beider Ziele sind die Eigenheiten der vorgegebenen Komponenten zu berücksichtigen.

3.1 Entwicklung der Hardware

Nachladbare Tasks eines Benutzers werden vom Betriebssystem sofort eingebunden und gemäß ihrer Priorität ausgeführt. Lädt man einen Task mit einem Steuerprogramm nach, wird dieses ausgeführt und je nach Programm kann sich die mobile Roboterplattform sofort bewegen. Möglicherweise ist sie zu diesem Zeitpunkt noch über ein Kabel mit dem Entwicklerrechner verbunden. Ein Stoppen des Programms ist nur über das Stoppen des Tasks vom Rechner aus oder über ein programminternes Ereignis möglich. Es kann passieren, dass die mobile Roboterplattform das Kabel mitschleppt und dadurch Schäden verursacht. Eine solche Handhabung ist unkomfortabel und gefährlich. Daher wurde die Anforderung gestellt, dass ein Steuerprogramm mithilfe eines Schalters gestartet und gestoppt werden kann. In diesem Zusammenhang wäre es auch nützlich zu erkennen, in welchem Betriebszustand sich das Programm gerade befindet.

Als einfache Kollisionssensoren sind am HighTecBot vier Schaltleisten vorhanden, wie sie auch beim ProfiBot eingesetzt werden. Diese sollen mit dem eingebetteten System EasyRun-TC1796 ausgewertet werden können.

Will man die mobile Roboterplattform autonom fahren lassen, sind weitere Sensoren, z.B. Abstandsensoren nötig. Einige dieser Sensoren könnten nach entsprechender Anpassung an die vorhandenen analogen Eingänge des EasyRun-TC1796 angeschlossen werden. Damit ist eine flexiblere Verwendung möglich.

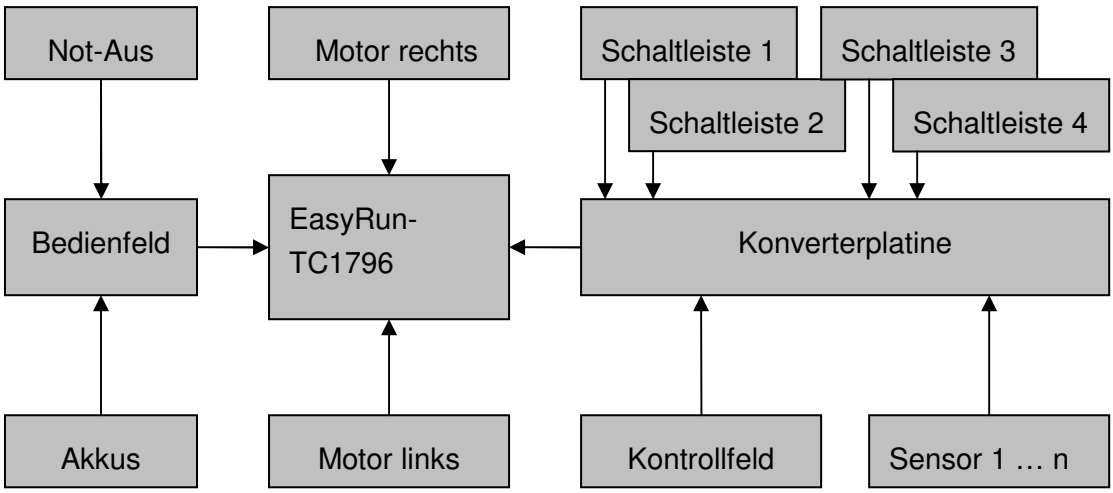


Abbildung 12: Grobes Schema der Verschaltung

Diese Anforderungen erfordern die Entwicklung einer zusätzlichen Elektronikkomponente, die die gewünschten Informationen und Signale auf die Schnittstelle des EasyRun-TC1796 anpasst. Dazu wird eine geeignete Konverterplatine entwickelt, die im Folgenden beschrieben ist.

3.1.1 Anforderungen an die Konverterplatine

Die Konverterplatine soll drei Aufgaben erfüllen:

1. Ein Schalter und LEDs sollen angeschlossen werden können. Die Signalspannungen des Schalters müssen vom EasyRun-TC1796 als „High“ bzw. „Low“-Pegel verarbeitet werden können. Der geringe Maximalstrom pro Ausgangspin des Mikrokontrollers für die LEDs darf nicht überschritten werden. Daher ist ein Treiber nötig, der einen höheren Strom zur Verfügung stellt. Vom eingebetteten System EasyRun-TC1796 lässt sich keine geeignete Versorgungsspannung abgreifen. Daher ist das Bereitstellen einer geeigneten Versorgungsspannung auf der Konverterplatine nötig. Die Schalter und LEDs werden auf einem externen Kontrollfeld zusammengefasst, damit sie gut erreichbar angebracht werden können.
2. Die Schaltleisten sollen ausgewertet werden können. Eine Schaltleiste besteht aus einer Kontaktschiene, die mit einem Abschlusswiderstand abgeschlossen wird. Es können damit drei Zustände detektiert werden: ausgelöst, nicht ausgelöst und Kabelbruch. Da drei Zustände nicht von einem digitalen Eingang ausgewertet werden können, aber genug analoge Eingänge zur Verfügung stehen, werden die Schaltleisten so geschaltet, dass sie von einem analogen Eingang ausgewertet werden können.

3. Weitere Sensoren für autonomes Fahren sind oft z.B. Abstandsensoren. Diese geben ein analoges Signal aus. Um möglichst viele Typen und Fabrikate anschließen zu können, ist die Anpassung auf das Einheitssignal 0 – 10 V sinnvoll. Neben einer Schutzbeschaltung gegen Überspannung sind Spannungsteiler nötig, die das Eingangssignal auf den Spannungspegel von 3,3 V des EasyRun-TC1796 anpassen.

Zusätzlich zu diesen Anforderungen soll die Konverterplatine möglichst kostengünstig sein, einfach gehalten werden und kleine Abmessungen haben. Aufgrund der recht niedrigen Frequenzen bis maximal 1 kHz sind keine speziellen EMV-Maßnahmen (EMV = **E**lektro**m**agnetische **V**erträglichkeit) erforderlich. Die Platine soll allenfalls vor fremden EMV-Einflüssen soweit geschützt sein, dass Messsignale möglichst unverfälscht bleiben. Die Konverterplatine soll direkt an die Spannung des Bordnetzes von 24 V angeschlossen werden können. Da die Möglichkeit besteht, eine RS232 Verbindung zu nutzen, soll diese Schnittstelle auch herausgeführt werden. Diese könnte auch zur Verwendung von ProfiBot-Steuerprogrammen genutzt werden.

Werden keine Schaltleisten an die Konverterplatine angeschlossen, sind vier analoge Eingänge unbenutzt. Durch einen Jumper könnten diese vier Eingänge zu den analogen Sensoreingängen hinzugeschaltet werden. Damit werden Ressourcen besser genutzt.

In einer weiteren Entwicklungsstufe des ProfiBot-Bedienfeldes wird dieses immer weiter an Sicherheitsanforderungen angepasst. Ein zukünftiger Schritt wird sein, die Schaltleisten am Bedienfeld auszuwerten und über eine Kommunikationsschnittstelle der Motorsteuerung die Schaltzustände mitzuteilen. Ein Schaltungsteil auf der Konverterplatine sieht eine solche Schnittstelle vor. Der aktuelle Entwicklungstand des Bedienfeldes ist noch nicht so weit fortgeschritten, dass dieser Teil in der vorliegenden Arbeit vertieft werden kann. Dieser Schaltungsteil wird nicht weiter besprochen.

Die Komponente, die den Schalter und die LEDs beinhaltet, nennt sich Kontrollfeld. Es ist ein Modul, das extern von der Konverterplatine angebracht werden kann.

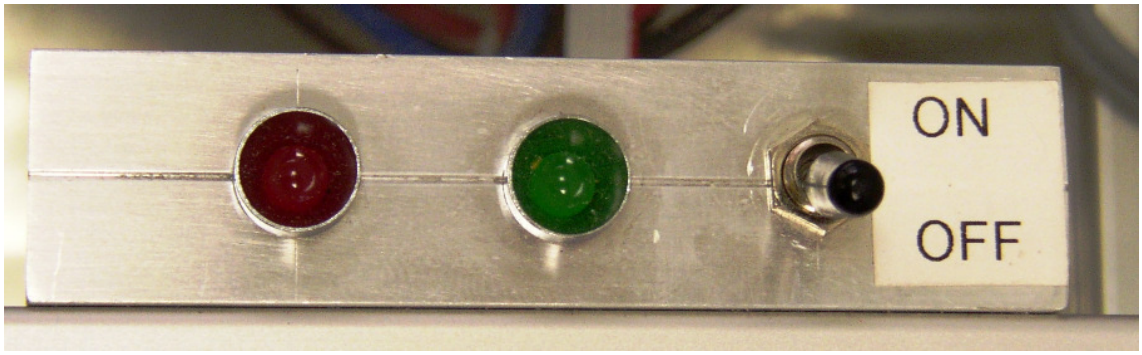


Abbildung 13: Kontrollfeld mit Schalter und LEDs

3.1.2 Schnittstellen der Konverterplatine

Einige Schnittstellen können entsprechend ihrer Verwendung angepasst werden.

Zusätzliche analoge Sensoren von unterschiedlichen Herstellern haben oft verschiedene Stecker. Um möglichst viele Sensoren anzuschließen, wurde auf eine Steckverbindung verzichtet und eine Klemmleiste mit Schraubklemmen verwendet. Neben den Eingängen für die Signale können an der Klemmleiste Masse und die Versorgungsspannung von 3,3 V abgegriffen werden.

Die Schaltleisten haben vorgegebene Stecker. Damit diese ohne Umbau verwendet werden können, wurden die entsprechenden Buchsen auf die Konverterplatine gebaut. Die Schaltleisten können dann direkt eingesteckt werden.

Da die LEDs und der Schalter nicht auf der Platine, sondern gut erreichbar auf einem externen Kontrollfeld sein sollen, ist hier eine weitere Schnittstelle nötig. Für das Anstecken des Kontrollfeldes, sowie der Kommunikation mit dem Bedienteil ist ein universeller Pfostenstecker im Rastermaß vorgesehen.

Eine weitere Schnittstelle ist zum Bordnetz von 24 V für die Stromversorgung nötig. Hier wird ein zweipoliger Schraubklemmverbinder eingebaut. Folgendes Schema und die Tabelle zeigen die Schnittstellen und ihre geplanten Steckverbinder.

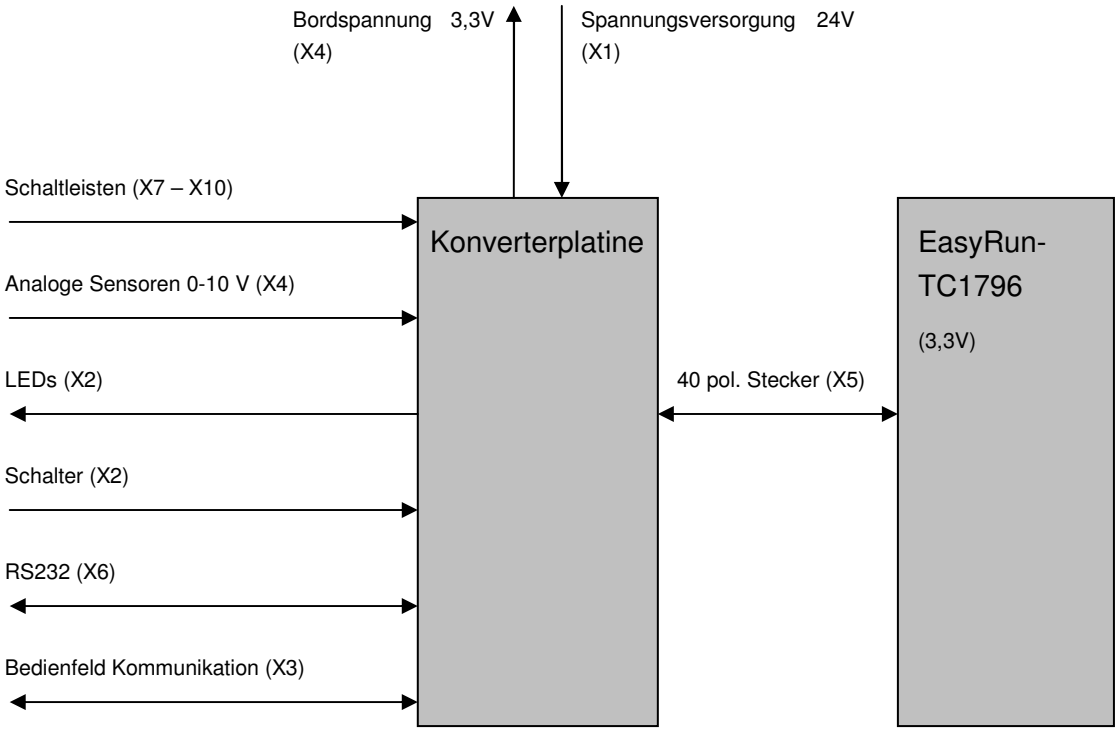


Abbildung 14: Schema der Schnittstellen der Konverterplatine

Steckerkürzel	Stecker	Funktion
X1	2-pol. Schraubklemme	Spannungsversorgung 24 V
X2	6-pol. Pfostenstecker	Schnittstelle LEDs und Schalter
X3	6-pol. Pfostenstecker	Schnittstelle zum Bedienfeld
X4	10-pol. Schraubklemme	8 analoge Sensoreingänge, Masse, 3,3 V Versorgungsspannung
X5	40-pol. Pfostenstecker	Schnittstelle zum EasyRun-TC1796
X6	9-pol. Sub-D Stecker	RS232 Schnittstelle
X7-X10	2-pol. Stecker	Steckplätze für vier Schaltleisten

Tabelle 4: Übersicht der Schnittstellen

Die Schnittstelle zum eingebetteten System EasyRun-TC1796 stellt eine 40-polige Steckverbindung dar. Neben CAN-Bus und RS232-Verbindung sind auch einige I/O-Ports des Mikrokontrollers herausgeführt. Einige Pins sind mit Masse belegt. Die interne Spannung des EasyRun-TC1796 von 3,3 V wird nicht herausgeführt. Eine komplette Beschreibung der Schnittstellen des EasyRun-TC1796 findet sich in [13]. Von der virtuHall-API aus sind folgende Pins anzusprechen:

Digitale Ports		Analoge Ports	
Pin	**Port**	**Pin**	**Port**
21	Measure-Port 0	33	Analog-Port 0
22	Measure-Port 1	34	Analog-Port 1
23	I/O-Port 0	35	Analog-Port 2
24	I/O-Port 1	36	Analog-Port 3
25	I/O-Port 2	37	Analog-Port 4
26	I/O-Port 3	38	Analog-Port 5
27	I/O-Port 4	39	Analog-Port 6
28	I/O-Port 5	40	Analog-Port 7
29	I/O-Port 6		
30	I/O-Port 7		

Tabelle 5: Verwendbare I/O-Ports des EasyRun-TC1796

3.1.3 Entwicklung des Schaltplans der Konverterplatine

Schaltplan und Layout werden mit der Studentenversion des Elektronik-CAD-Programms „Eagle“, Version 4.16r1 von der CadSoft Computer GmbH hergestellt [4]. Die im Schaltplan benutzten Bauteile sind mit CAD-Daten verknüpft, so dass Größe und Pinbelegung im Layout sofort verfügbar sind. Die meisten auf dem Markt erhältlichen Bauteile sind in einer Datenbank dem Programm beigefügt oder an entsprechenden Stellen über das Internet erhältlich. Aus dem Layout können Masken für die Herstellung geätzter Platinen erzeugt werden. Damit steht ein vielseitiges Tool zur Platinenentwicklung zur Verfügung. Der mit Eagle erstellte Schaltplan und das Layout liegen als Projekt für Eagle auf der Begleit-CD bei.

3.1.3.1 Spannungsversorgung, Schalter und LEDs

Da die 40-polige Schnittstelle zum EasyRun-TC1796 keine belastbare Spannung von 3,3 V bereitstellt, muss eine eigene Spannungsversorgung eingefügt werden. Im Vorfeld muss geklärt werden, welche Leistung die Konverterplatine aufnimmt, um die Spannungsversorgung zu dimensionieren. Die Verbraucher sind:

- zwei LEDs
- ein Schalter, der High-Pegel schalten soll
- vier Schaltleisten mit 8,2 kOhm Abschlusswiderstand

Der aufgenommene Strom bewegt sich im mA-Bereich bis maximal 50 mA. Um Sicherheit und die Möglichkeit zu bieten, weitere Verbraucher anzuschließen, wurde ein Festspannungsregler mit einem maximalen Ausgangsstrom von 150 mA gewählt. Da die Betriebsspannung des HighTecBot aufgrund der Batterien zwischen 22 und 28 V schwanken kann, sollte der Festspannungsregler eine Eingangsspannung von bis zu 30 V verarbeiten können. Ein solches Bauteil ist der Festspannungsregler LT 1121 CZ-3.3 von Linear Technology [17]. An den Eingang wird ein Entstörkondensator von 10 nF und an den Ausgang ein Glättungskondensator von 10µF geschaltet. Damit steht eine kurzschlussfeste, verpolungssichere und leistungsstarke Variante der Spannungsversorgung zur Verfügung.

Als LEDs werden zwei Standard-LEDs verwendet. Da ein Ausgang des EasyRun-TC1796 maximal 4 mA Strom liefert, werden die LEDs mithilfe von jeweils einem Transistor in Emitterschaltung angesteuert. Der Schalter soll je nach Schaltstellung einen Low- oder High-Pegel an einem Pin erzeugen. Hier ist lediglich ein Pullup-Widerstand notwendig. Daraus ergibt sich folgender Schaltplan:

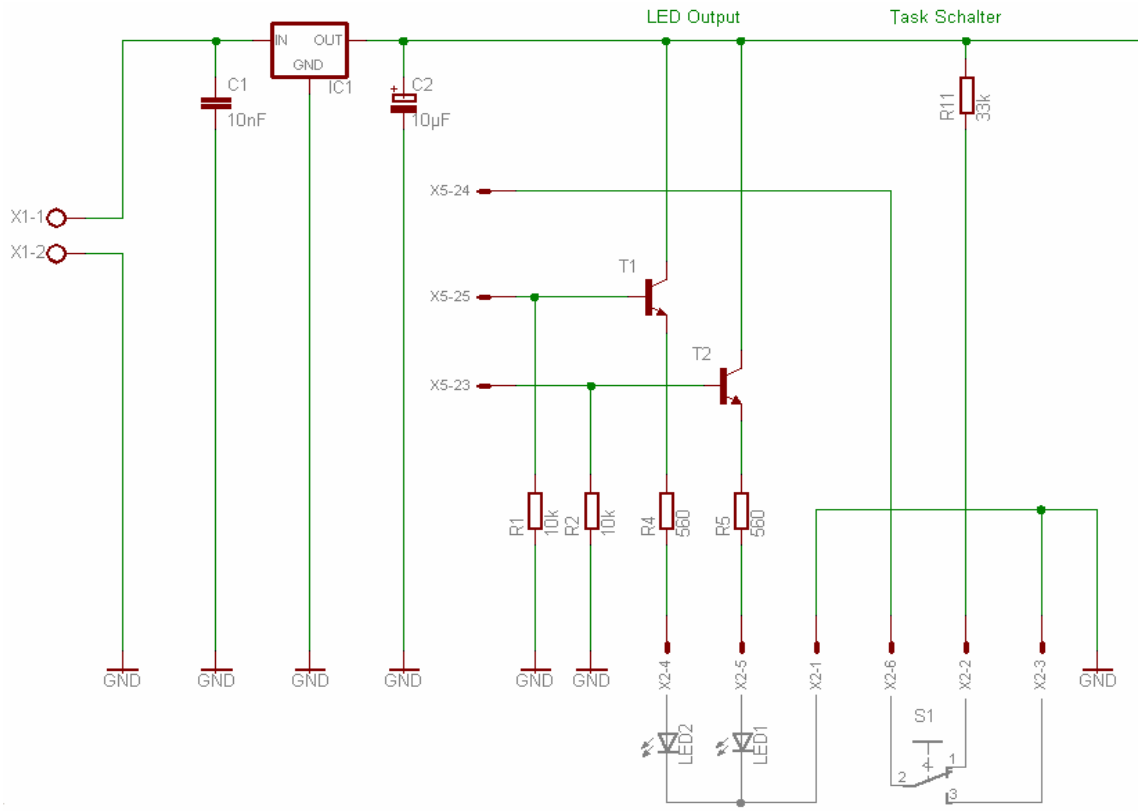


Abbildung 15: Schaltplan der Spannungsversorgung

Die grau eingezeichneten LEDs und der Schalter sind nicht auf der Konverterplatine implementiert, sondern im externen Kontrollfeld. X1 ist die Schraubklemme, an die die Bordspannung von 24 V angelegt wird. X5 ist der 40-polige Stecker mit den entsprechenden Pins. Die Verbindung zu den LEDs und dem Schalter stellt X2 her.

3.1.3.2 Schaltleisten und analoge Eingänge

Die Schaltleisten arbeiten mit ihrem Abschlusswiderstand als Spannungsteiler. Der Abschlusswiderstand und ein zusätzlicher Widerstand erzeugen den dritten Schaltzustand. Dieser zusätzliche Widerstand wird auf der Konverterplatine eingebaut und verbindet ein Ende der Schaltleiste mit der Versorgungsspannung. Das andere Ende der Schaltleiste liegt auf Masse. Wird keine Kollision detektiert, liegt zwischen den beiden Widerständen der Pegel des Spannungsteilers an. Bei einer Kollision wird der Abschlusswiderstand kurzgeschlossen und so ein Low-Pegel erzeugt. Bei einem Kabelbruch fällt der Abschlusswiderstand weg und ein High-Pegel liegt am Pin an. Diese drei Zustände werden an einem analogen Eingang detektiert. Für vier Schaltleisten werden vier analoge Eingänge belegt. Dieselben vier analogen Eingänge können mit vier Jumpern zu Eingängen für die Sensorauswertung umgeschaltet werden. Die Sensorsignale liegen als Einheitssignal von 0 – 10 V an. Um sie für die analogen Eingänge anzupassen wird jeweils ein Spannungsteiler auf 3,3 V verwendet. Damit diese Spannung nicht überschritten wird, ist eine Zenerdiode parallel geschaltet. Die Widerstände sind so dimensioniert, dass maximal ein Strom von 1 mA fließt.

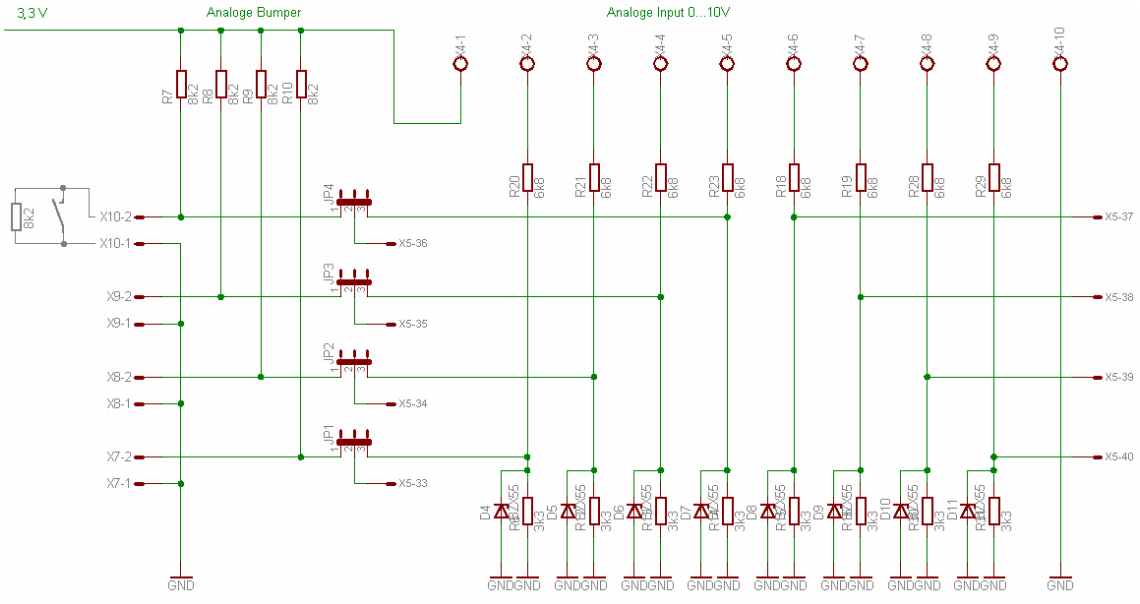


Abbildung 16: Schaltplan der analogen Eingänge

An X10 ist exemplarisch in grau eine Schaltleiste eingezeichnet, um die Funktion zu verdeutlichen. Der Abschlusswiderstand dieser Schaltleiste bildet mit R7 einen Spannungsteiler, der über Jumper 4 an Pin 36 ausgewertet wird. Alternativ können die vier für die Schaltleisten verwendeten Pins durch die andere Jumperstellung als Analog-Port 0 – 3 benutzt werden. Die Analog-Ports 4 – 7 sind immer als Sensoreingänge verfügbar. Die Sensoren können an X4 Pin 2 – 9 der Klemmleiste angeschlossen werden. Pin 1 der Klemmleiste stellt 3,3 V Versorgungsspannung zu Verfügung und Pin 10 Masse.

3.1.3.3 RS232-Verbindung und Bedienfeldkommunikation

In diesem Teil der Schaltung werden die Pins für RxD (receive data, Daten empfangen) und TxD (transmit data, Daten senden) der RS232-Verbindung auf einen Sub-D-Stecker geführt. Diese Verbindung dient als Schnittstelle für eine spätere Bus-Kommunikation, die in dieser Arbeit nicht weiter vertieft wird. Der Vollständigkeit halber werden sowohl die RS232-Anbindung wie auch die Kommunikation mit dem Bedienteil als Schaltplan dargestellt. Wichtig ist dabei, dass eine direkte Masseverbindung zwischen EasyRun-TC1796 und Konverterplatine besteht, um den analogen Sensoren und den Analog-Digital-Wandlern eine gleiche Basis für ihre Auswertung zu geben.

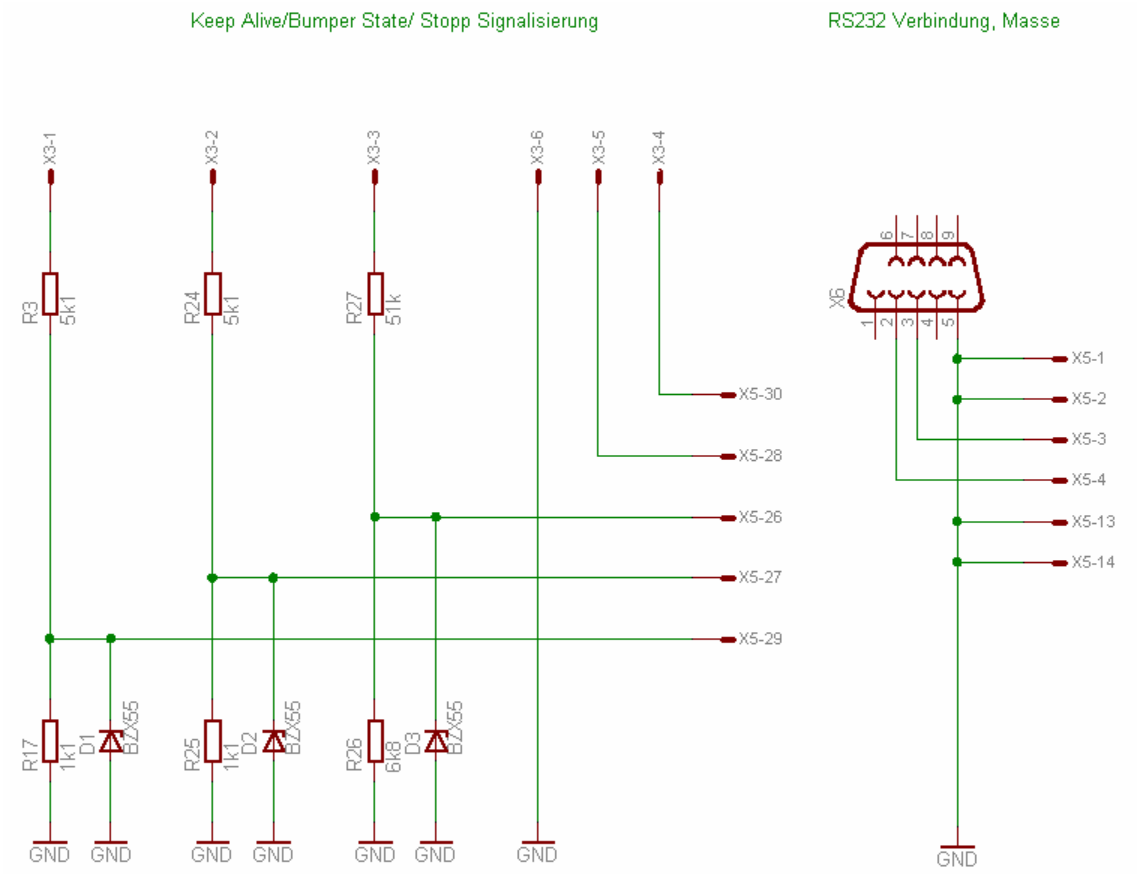


Abbildung 17: Schaltplan der RS232-Verbindung

X3 stellt die Verbindung zum Bedienfeld her. Die RS232-Verbindung wird mit dem Stecker X6 realisiert. Die Pins 1, 2, 13 und 14 des 40-poligen Steckers X5 werden mit Masse verbunden.

3.1.4 Umsetzung des Schaltplans

In einer ersten Version wurde die Schaltung auf einer Lochrasterplatine umgesetzt, um die Funktionalität zu prüfen und leicht Änderungen vornehmen zu können. Nachdem alle Schaltungsteile funktionierten, wurde ein Layout für eine geätzte Platine entwickelt. In einem Layout werden die Bauteile und Leitungsverbindungen so positioniert, dass die Platine real hergestellt werden kann. Bevor die Bauteile und Leiterbahnen platziert werden, wird festgelegt, wie die mechanischen Eigenschaften der Platine sein müssen, insbesondere die Befestigungspunkte. Die Konverterplatine muss keine großen mechanischen Kräfte aufnehmen und ist in einem Kunststoffgehäuse untergebracht. Vier Befestigungspunkte, jeweils einer an einer Ecke, sind daher ausreichend. Am Rand der Platine und um die Befestigungspunkte wird ein Sperrbereich für Leiterbahnen und Bauteile definiert, damit beim Bohren der Löcher keine Leiterbahn beschädigt wird und beim späteren Anbringen der Schrauben genug Platz ist. Nun können Bauteile und Leiterbahnen platziert werden.

Die Stecker wurden so angeordnet, dass sie größtenteils am Rand der Platine liegen und sich aufgrund ihrer Baugröße nicht behindern. Aus dem Schaltplan ist ersichtlich, dass sich viele Leiterbahnen kreuzen werden, nicht zuletzt, weil in der Nähe des 40-poligen Steckers viele Leitungen zusammen kommen. Daher werden Ober- und Unterseite (Top- und Bottom-Layer) der Platine benutzt. Nachdem alle Bauteile durch Leitungen verbunden sind, kann die Massefläche auf Ober- und Unterseite erzeugt werden. Diese schützt die Platine vor EMV-Einflüssen und verhindert starke EMV-Emissionen. Um eine gute Verbindung der beiden Masseflächen zu erreichen, werden zusätzliche Durchkontaktierungen angebracht.

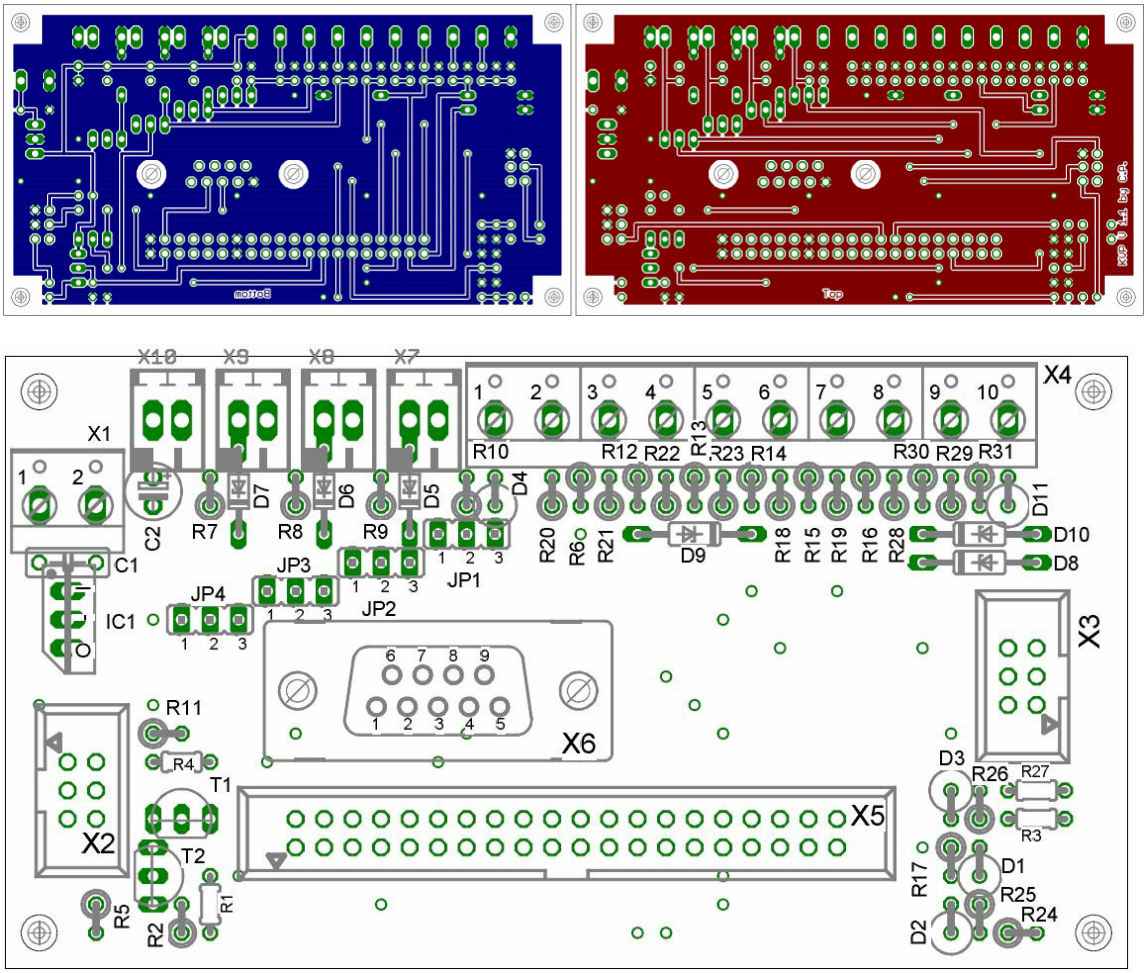


Abbildung 18: Bottom- und Top-Layer und Bauteilplatzierung der Konverterplatine

In Abbildung 18 ist die Platzierung und Verdrahtung der Bauteile zu sehen. Die blauen Linien stellen die Leitungsverbindungen auf der Unterseite der Platine dar und die roten Linien die Verbindungen auf der Oberseite. Die blaue und rote Fläche ist die Massefläche. Die Stecker X1, X4 und X7 bis X10 sind am oberen Rand der Platine angebracht, Stecker X2 und X3 befinden sich am linken bzw. rechten Rand. Die recht hohe Spannung von 24 V beschränkt sich ausschließlich auf die obere linke Ecke und ist durch Masse vom Rest der Platine getrennt. Die Jumper JP1 bis JP4 sind einfach den Steckern X7 bis X10 zuzuordnen. Ist ein Jumper auf der linken Seite eingesteckt, ist die Schaltleiste aktiv, ist er auf der rechten Seite eingesteckt, ist der Sensoreingang

aktiv. Die Abmessungen betragen 100 mm mal 55 mm. Kleinere Abmessungen können erreicht werden, wenn ein kleineres Rastermaß oder SMD-Bauweise verwendet wird. Doch je enger die Stecker aneinanderrücken, desto schwieriger gestaltet sich die Bedienung.

3.1.5 Die Konverterplatine

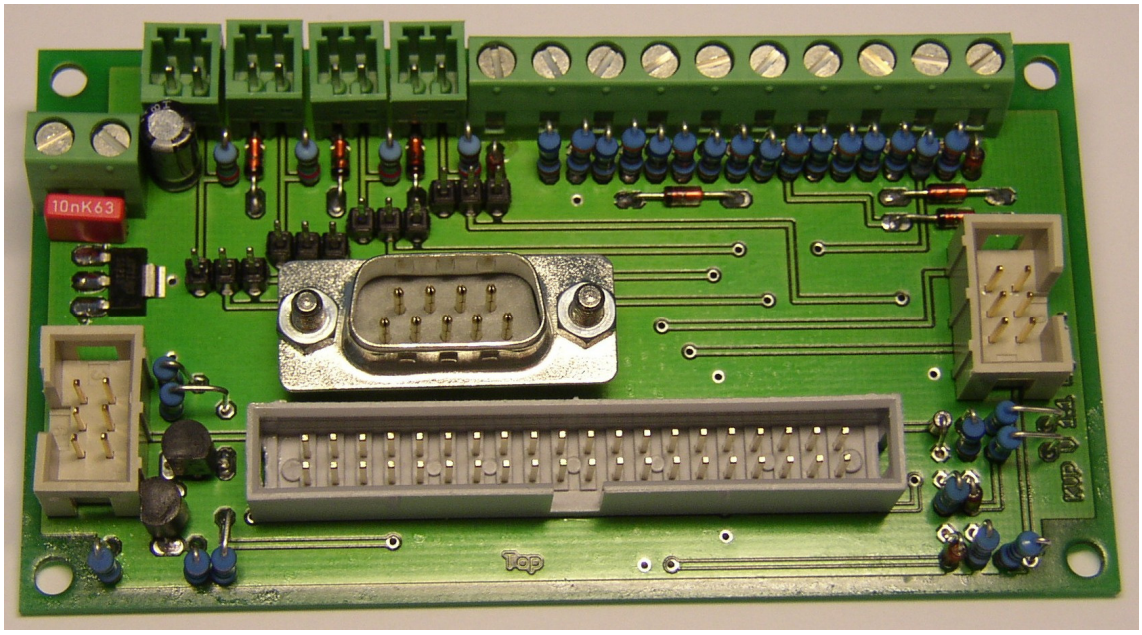


Abbildung 19: Fertige Konverterplatine

Mithilfe der Konverterplatine können nun einfach ein Kontrollfeld mit Schalter und LEDs sowie weitere Sensoren wie Schaltleisten oder analoge Sensoren an einer universellen Schraubklemme an das eingebettete System EasyRun-TC1796 angebunden werden. Damit ist die Ausrichtung des EasyRun-TC1796s auf den Einsatz in der Robotik gegeben.

3.1.6 Einbindung in die Software

Im Abschnitt 2.6.2 wurden bereits die relevanten Funktionen der virtuHall-API aufgelistet, mit denen die digitalen und analogen Ein- und Ausgänge angesprochen werden können. Mit `vhSetPortType()` muss in einer Initialisierungsphase festgelegt werden, wie ein digitaler Port benutzt wird. Der Port kann als Eingang oder als Ausgang sowie als Pullup, Pulldown oder kein Pullup konfiguriert werden. Einen Eingang als Interrupt zu verwenden, steht in der derzeitigen Version noch nicht zur Verfügung.

Die Übergabe der digitalen Informationen an den Ports erfolgt über ein Bitfeld. Das LSB (less significant bit) steht dabei für Port 0 und das achte Bit für Port 7. Bitfelder werden sowohl für Eingangsports als auch für Ausgangsports benutzt. Die I/O-Ports können mit den beiden Funktionen `vhSetPortIO()`gesetzt und `vhGetPortIO()`

ausgelesen werden. Um die I/O-Ports flexibel zu nutzen, werden die internen Funktionsblöcke `set_Output(int port, int value)` und `get_Input(int port, int* value)` geschrieben. Im Parameter `port` muss die Nummer des digitalen Ports eingetragen werden und im Parameter `value` eine 1 oder 0 bzw. `ON` oder `OFF`, je nachdem, wie der Ausgang gesetzt werden soll. Die jeweiligen Portnummern sind in Tabelle 5 aufgelistet.

Codeausschnitt zum Setzen eines Ausgangs:

```
//Ein- und Ausgabezustände definieren
#define ON    1
#define OFF   0

//Definieren einer Variablen für das Bitfeld
static unsigned int pin_output = 0;

static int set_Output(int port, int value)
{
     //Definieren einer Bitmaske
     unsigned short mask = 1;
     //Variable für den Rückgabeparameter
     int ret_val = NO_ERROR;

     //Wenn "High" ausgegeben werden soll:
     if(value == ON)
     {
          //Das Bitfeld an der Stelle "port" auf "High" setzen
          pin_output |= (mask << port);
          //Das Bitfeld der virtuHall-API übergeben
          vhSetPortIO(MOTOR_1, pin_output);
     }
     //Wenn "Low" ausgegeben werden soll:
     else if(value == OFF)
     {
          //Das Bitfeld an der Stelle "port" auf "Low" setzen
          mask = mask << port;
          mask = ~ mask;
          pin_output = pin_output & mask;
          //Das Bitfeld der virtuHall-API übergeben
          vhSetPortIO(MOTOR_1, pin_output);
     }
     else
          //Bei einer ungültigen Eingabe einen Fehler erzeugen
          ret_val = INPUT_ERROR;

     return ret_val;
}
```

Die Behandlung der Fehlercodes in der Variable `ret_val` wird im Abschnitt 3.2.5 beschrieben.

Beim Einlesen von Eingängen ist die Vorgehensweise ähnlich. Das Bitfeld wird von der virtuHall-API ausgelesen und an der entsprechenden Stelle auf den Zustand hin geprüft.

Codeausschnitt zum Einlesen eines Eingangs:

```
//Definieren einer Variablen für das Bitfeld
static unsigned int pin_input = 0;


static int get_Input(int port, int* value)
{
     //Definieren einer Bitmaske
     unsigned short mask = 1;
     //Variable für den Rückgabeparameter
     int ret_val = NO_ERROR;

     //Das Bitfeld von der virtuHall-API auslesen
     vhGetPortIO(MOTOR_1, &pin_input);

     //Die Bitmaske auf die auszulesende Stelle vorbereiten
     mask <<= port;

     //prüfen, welchen Zustand die auszulesende Stelle hat
     if((pin_input & mask) == 0)
          *value = 0;
     else if((pin_input & mask) >= 1)
          *value = 1;
     else
          //In einem binären System kommt dieser Fall normalerweise
          //nicht vor
          ret_val = SYSTEM_ERROR;

    return ret_val;
}
```

In Verbindung mit der Konverterplatine haben die einzelnen Ports bestimmte Funktionen. Diese sind in folgender Tabelle aufgelistet:

Port	**Stecker**	**Pin**	**Funktion**
Measure-Port 1	-	-	Keine
Measure-Port 2	-	-	Keine
I/O-Port 0	X2	5	Grüne LED
I/O-Port 1	X2	6	Schalter
I/O-Port 2	X2	4	Rote LED
I/O-Port 3	X3	3	Bedienfeldkommunikation
I/O-Port 4	X3	2	Bedienfeldkommunikation
I/O-Port 5	X3	5	Bedienfeldkommunikation
I/O-Port 6	X3	1	Bedienfeldkommunikation
I/O-Port 7	X3	4	Bedienfeldkommunikation
Analog-Port 0	X7, (X4)	2, (2)	Schaltleiste (analoger Sensor)
Analog-Port 1	X8, (X4)	2, (3)	Schaltleiste (analoger Sensor)
Analog-Port 2	X9, (X4)	2, (4)	Schaltleiste (analoger Sensor)
Analog-Port 3	X10, (X4)	2, (5)	Schaltleiste (analoger Sensor)
Analog-Port 4	X4	6	Analoger Sensor
Analog-Port 5	X4	7	Analoger Sensor
Analog-Port 6	X4	8	Analoger Sensor
Analog-Port 7	X4	9	Analoger Sensor

Tabelle 6:Zuweisung der I/O-Ports an die Konverterplatine

3.2 Entwicklung der Software

An die Software wurden im Vorfeld einige Anforderungen gestellt. Als Teil des Basissystems von PXROS-HR soll in so genannten nachladbaren Tasks auf die Funktionalitäten der Robotersteuerung über eine Robotersteuerung-API zugegriffen werden können. Die Robotersteuerung selber soll ebenfalls ein Task sein, der sich in das Echtzeitverhalten des Betriebsystems einfügt und anderen Anwendungen Prozessorzeit übriglässt. Um die Robotersteuerung individuell zu halten, müssen Maximalwerte einstellbar sein, die den gesamten Wertebereich bis zur physikalischen Maximalgrenze abdecken. Sollwerte sollen physikalischen Größen entsprechen. Sämtliche Istwerte sollen jederzeit erfassbar sein und ebenfalls physikalischen Größen entsprechen. Letztlich soll eine Regelung dafür sorgen, dass die mobile Roboterplattform geradeaus fährt und Sollwerte erreicht werden.

Die Robotersteuerung wurde als eigenständiger Task entwickelt, die als Modul in das Basissystem mit eingebunden werden kann. In der Systemarchitektur von PXROS-HR macht es keinen Unterschied, ob der Task nachladbar in den RAM-Speicher oder als Modul des Basissystems in den Flash-Speicher geladen wird. Voraussetzung für das Nachladen weiterer Tasks ist, dass es einen Zeitraum gibt, in der der Task in den Betriebszustand „bereit“ wechselt, um Tasks mit geringerer Priorität die Möglichkeit zu geben, ausgeführt zu werden. Die Priorität der Robotersteuerung muss zwischen der Priorität der virtuHall-API und den Prioritäten der Anwender-Tasks liegen.

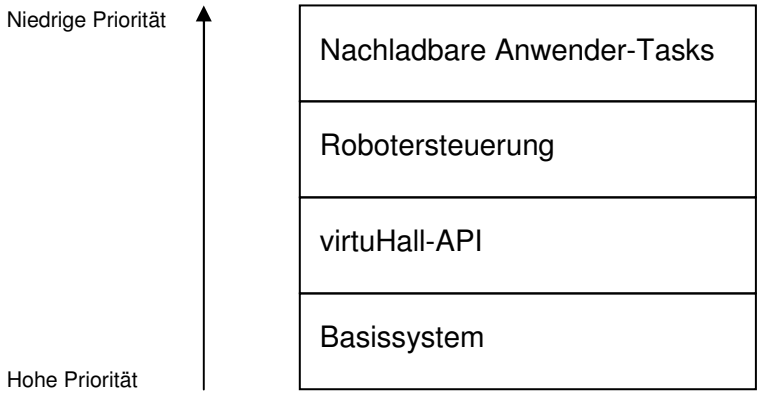


Abbildung 20: Hierarchie der Tasks durch Prioritäten

Neben der richtigen Einbindung in PXROS-HR wurde bei der Programmierung darauf geachtet, dass alle nötigen Maximalwerte eingestellt und abgefragt werden können. Siehe hierzu Abschnitt 3.2.6. Die Anforderung, alle Istwerte jederzeit zu erfassen, wurde in dem Konzept der Robotersteuerung verarbeitet. Dieses wird in Abschnitt 3.2.1 und 3.2.4 näher erläutert.

Die Geradeausfahrt hängt sehr stark von äußeren Einflüssen ab, die ohne zusätzliche Sensorik am Roboter nicht ausgeglichen werden können [2]. Mögliche Einflüsse sind:

- Unebener Boden, Bodenwellen
- Schlupf zwischen Rad und Untergrund
- Unterschiedliche Raddurchmesser
- Ungenau erfasster Radabstand

Durch eine Regelung der Geradeausfahrt kann erreicht werden, dass beide Motoren die gleiche Anzahl an Inkrementen weiterbewegt werden. Mithilfe einer Kalibrierung können zudem ungleiche Raddurchmesser und ungenau erfasster Radabstand kompensiert werden. Fehler, die durch Einflüsse aus der Umgebung entstehen, werden hier nicht von Sensoren erfasst und kompensiert. Damit kann die Geradeausfahrt der mobilen Roboterplattform zurzeit nur bedingt erfüllt werden.

3.2.1 Das Konzept der API

Als Robotersteuerung-API wird die Schnittstelle zu den Benutzern verstanden. Sie soll so allgemein sein, dass ein Anwenderprogramm, das diese Schnittstelle benutzt, ohne Anpassungen auf andere mobile Roboterfahrzeuge mit der gleichen Schnittstelle übertragbar ist. Möglich ist dies, indem die Schnittstelle auf physikalischen Größen basiert. Mithilfe dieser Schnittstelle kann auf bestimmte Funktionen der unteren Ebenen zugegriffen werden. Die Robotersteuerung wurde konzeptionell in zwei aufeinander aufbauenden Ebenen unterteilt.

In einer unteren Ebene, die direkt auf der virtuHall-API aufbaut, wird die Drehzahlregelung der einzelnen Räder realisiert. Jedes Rad wird von einem eigenen Regler geregelt, bekommt eigene Eingangswerte und gibt eigene Ausgangswerte aus. Jedes Rad entspricht einem eigenständigen Modul. Alle Räder können von der Robotersteuerung-API aus angesteuert werden. Diese Ebene wird Antriebsteuerung genannt, da die Antriebsmotoren direkt angesprochen werden. Damit lässt sich der Roboter bereits navigieren. Bei dieser Art der Navigation kann anhand der Ansteuerung keine direkte Aussage über Geschwindigkeit oder Verdrehwinkel des Fahrzeuges gemacht werden.

Dafür gibt es eine weitere Ebene, die auf der Drehzahlregelung aufbaut. Diese Ebene nennt sich Fahrzeugsteuerung. Hier werden nicht mehr die einzelnen Räder des Roboters betrachtet, sondern der Roboter als Fahrzeug insgesamt. Demnach kann das Fahrzeug einen beliebigen Antrieb besitzen. Als Eingangsgrößen erhält diese Ebene z.B. eine Lineargeschwindigkeit oder eine Rotationsgeschwindigkeit. Diese wird dann über die Drehzahl der einzelnen Räder umgesetzt. In dieser Ebene, die nicht mehr direkt auf die virtuHall-API zugreift, ist es möglich, einen direkten Bezug zwischen Soll- und Ist-Geschwindigkeit des Fahrzeuges herzustellen.

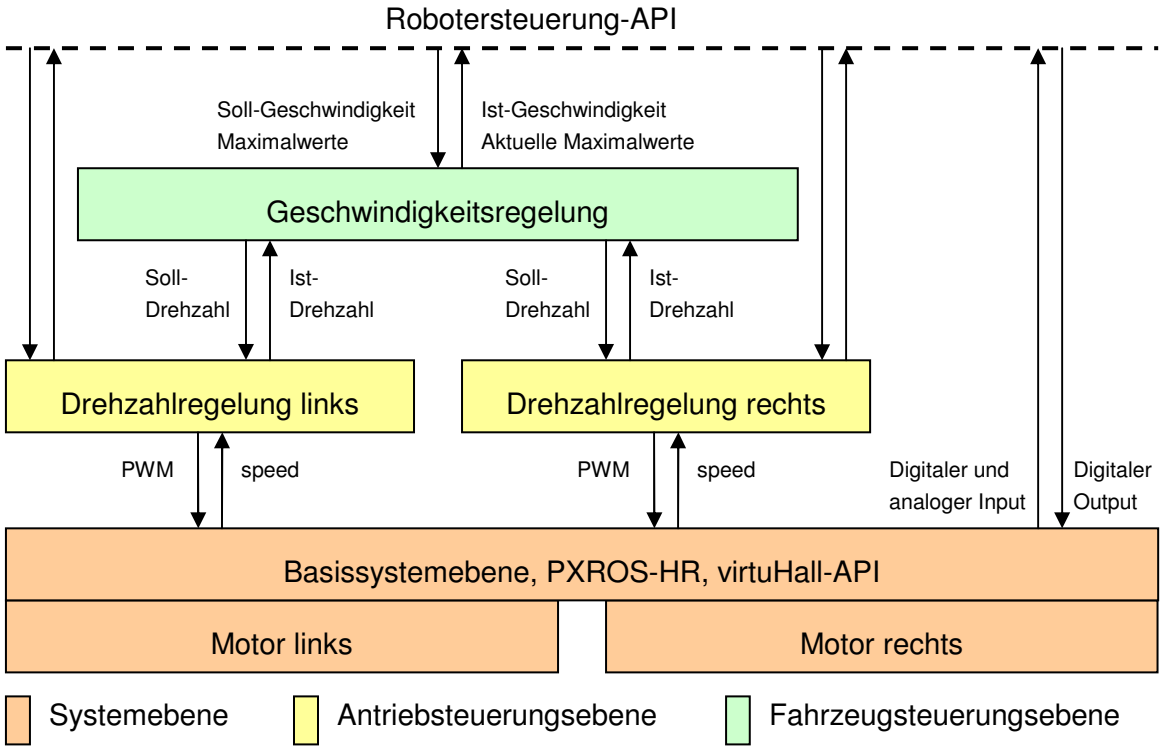


Abbildung 21: Schema der Ebenen der Robotersteuerung-API

Die Robotersteuerung-API stellt die Schnittstelle zu den Benutzern dar. Antriebsteuerung und Fahrzeugsteuerung sind zwei aufeinander aufbauende Ebenen, die zusammen Robotersteuerung genannt werden. Die Parameterübergabe zwischen der Robotersteuerung-API, der Fahrzeugsteuerung und der Antriebsteuerung basiert auf physikalischen Größen. Die Drehzahl liegt in rpm (engl. rotations per minute, Umdrehungen pro Minute) vor und die Geschwindigkeiten und Beschleunigungen werden in Millimeter pro Sekunde (mm/s) bzw. Millimeter pro Sekunde Quadrat (mm/s²) angegeben.

3.2.2 Schnittstellenfunktionen

Damit der Benutzer der API einfach auf die Funktionalitäten der Robotersteuerung zugreifen kann, gibt es eine Reihe ausführlich beschriebener Schnittstellenfunktionen, die einfach zu verstehen und zu bedienen sind. Mit diesen können Sollwerte eingestellt und Istwerte abgefragt werden. Die Namen der Schnittstellenfunktionen beinhalten keine Abkürzungen und sind so gegliedert, dass sie anhand ihrer Syntax zugeordnet werden können. Die Benutzer können intuitiv nicht bekannte Funktionen aufrufen. Alle Funktionen, die einen Sollwert an die Robotersteuerung übergeben, beginnen mit „`Set`“. Den Istwert bekommt man mit einer Funktion, die mit „`Get`“ beginnt, sonst aber die gleiche Syntax hat. Z.B. kann man eine Soll-Geschwindigkeit mit `htbSetVelocity()` vorgegeben und die Ist-Geschwindigkeit mit `htbGetVelocity()` abfragen.

Die Schnittstellenfunktionen lassen sich in drei Gruppen aufteilen: Antriebsteuerung, Fahrzeugsteuerung und allgemeine Funktionen. Die Funktionen der Antriebsteuerung haben in ihrem Namensaufbau nach den Wörtern `Set` und `Get` immer das Wort `Motor` stehen. Handelt es sich um einen Maximalwert, ist das Wort `Max` vorhanden. Weiß der Benutzer z.B. nicht, wie man die maximale Drehzahl eines Rades abfragt kann er sich die Syntax einer Funktion herleiten. Alle Funktionen beginnen mit `htb`. Anschließend kommt `Get`, das Wort, mit dem Istwerte abgefragt werden. Da es sich um die Antriebsteuerung handelt, kommt nun das Wort `motor`. Für den Maximalwert benötigt man das Wort `max` und schließlich die physikalische Größe `RPM`. Zusammengesetzt ergibt sich die Funktion: `htbGetMotorMaxRPM()`. Möchte man die aktuelle Drehzahl erfassen, lässt man das Wort `max` einfach weg.

In der Gruppe der Funktionen für die Fahrzeugsteuerung ist das gleiche Prinzip angewendet. Alle Sollwerte können als Istwerte abgefragt werden. Zusätzlich kann auch die aktuelle Beschleunigung und Verzögerung abgefragt werden.

Die Gruppe der allgemeinen Funktionen besteht aus zwei Schnittstellenfunktionen. Diese sind zum Umschalten eines Betriebsmodus gedacht (siehe hierzu Abschnitt 3.2.4). Alle Schnittstellenfunktionen sind im Anhang A „Übersicht der Schnittstellenfunktionen“ detailliert aufgelistet. Zudem befindet sich auf der Begleit-CD eine Dokumentation der Schnittstelle im html-Format. Diese wurde mit dem Programm doxygen [26] erstellt.

3.2.3 Struktur der Robotersteuerung

Insgesamt besteht die Robotersteuerung aus zwei Header-Files und einem Modul, das als ein Task in PXROS-HR eingebunden werden kann. Ein zusätzliches Modul sorgt dafür, dass der Task so gekapselt ist, dass er zur Laufzeit nachladbar ist. Die Komponenten sind im Einzelnen:

Name des Moduls	**Funktion**
`htb.h`	Hier sind die Prototypen der Funktionen und die Fehlercodes aufgelistet, die ein Benutzer ansprechen kann bzw. erhält.
`htbintern.h`	Hier werden alle internen Werte definiert wie z.B. Reglerparameter, physikalische Maximalwerte und Funktionsprototypen.
`main.c`	Das Hauptmodul. Es führt Funktionsaufrufe von außen, die Berechnung der internen und externen Istwerte und die Regelalgorithmen aus.
`taskmain.c`	Kapselt `main.c` so, dass es ein nachladbarer Task für PXROS-HR ist.

Tabelle 7: Programmteile der Robotersteuerung

Der Programmteil `main.c` ist der Kern der Robotersteuerung. Er ist in verschiedene Teile untergliedert, die den beiden Ebenen Antriebsteuerung und Fahrzeugsteuerung zuzuordnen sind. Das Modul `main.c` ist wie folgt gegliedert:

- Einbinden von Header-Files
- Deklarierung und Initialisierung von modulglobalen Variablen
- Hauptroutine „`taskmain`“
    - Deklarierung und Initialisierung von lokalen Variablen
    - Initialisierung der Kommunikation zur virtuHall-API
    - Initialisierung des Zeitmanagements
    - Hauptschleife
    - Schalter- und LED-Behandlung
    - Warten auf Vorgaben von Schnittstellenfunktionen
    - Abarbeiten der Drehzahlerfassung und Regelung
        - Abarbeiten der Geschwindigkeitserfassung und Regelung
- Funktionsblöcke der internen Funktionen
    - Drehzahlregelung und Erfassung
    - Geschwindigkeitserfassung
    - Linear- und Rotationsgeschwindigkeitsregelung
    - Routinen für die I/O-Ports
    - Routine für die Initialisierung
    - Systemfunktionen
- Funktionsblöcke der Schnittstellenfunktionen
    - Antriebsteuerung
    - Fahrzeugsteuerung
    - Allgemeine Schnittstellenfunktionen

Einzelne Punkte werden in späteren Kapiteln ausführlich behandelt. Der gesamte Quellcode liegt auf der Begleit-CD-ROM bei.

Innerhalb von `main.c` werden Daten über modulglobale Variablen ausgetauscht. Einige der Funktionen, z.B. der Drehzahlregler `set_RPM()`, benötigen so viele Daten, dass eine übersichtliche Parameterübergabe über die Funktion kaum möglich ist. Ein Zusammenfassen der Variablen in einem Feld (array) würde die Lesbarkeit des Programmcodes weiter einschränken. Auf modulglobale Variablen kann hingegen ohne großen Aufwand von den Schnittstellenfunktionen aus zugegriffen werden. Damit ist ein einfaches Setzen und Abfragen der Ist- und Sollwerte möglich. Mithilfe einer objektorientierten Programmierung könnte der Datenaustausch zwischen den Funktionen bzw. Methoden besser gelöst werden.

Die Schnittstellenkommunikation zwischen zwei Tasks wird über Nachrichten realisiert. Eine Nachricht besteht im Kern aus einem struct-Element, das als Inhalt den Namen der Funktion und die Parameter inklusive Rückgabewert enthält. In einer Programmbibliothek wird der Funktionsname dem Funktionsaufruf zugeordnet.

Ruft ein Anwender in seinem Task eine Funktion aus der Robotersteuerung auf, wird folgender Ablauf abgearbeitet:

1. Der Aufruf bewirkt das Generieren des Funktionsnamens.
2. Der Funktionsname und die Übergabeparameter werden in das Nachrichtenelement geschrieben.
3. Die Nachricht wird an den Robotersteuerung-Task gesendet und auf Antwort gewartet. Der Anwender-Task geht in den Zustand „wartend“ über.
4. Der Robotersteuerung-Task empfängt die Nachricht, und ruft die Funktion, die er dem Namen zuordnet, mit den Übergabeparametern auf.
5. Die Funktion wird in der Robotersteuerung ausgeführt und übergibt die Ergebnisse wieder dem Nachrichtenelement.
6. Die Nachricht wird zurückgesendet und der Robotersteuerung-Task geht in den Zustand „wartend“ zurück.
7. Der Anwender-Task empfängt die Nachricht, geht in den Zustand „bereit“ über und schreibt, nachdem er „aktiv“ geworden ist, die Übergabeparameter in die vom Anwender aufgerufene Funktion.

Dieser Ablauf wird bei jedem Funktionsaufruf einer Robotersteuerung-Funktion in dem Anwender-Task durchgeführt.

Die Funktionalität dieser Taskkommunikation ist in weiteren Programmmodulen implementiert. Sie wird hier nicht näher aufgeführt, da sie für diese Arbeit eine untergeordnete Rolle spielt. Der Quellcode existiert bereits bei der Fa. HighTec EDV-Systeme GmbH und wurde für die hier benötigte Taskkommunikation übernommen. Der Quellcode dazu ist auf der Begleit-CD zu finden.

3.2.4 Verschiedene Benutzermodi

Von der Robotersteuerung-API können beide Ebenen angesprochen werden. Damit ist eine Ansteuerung der einzelnen Räder (Antriebsteuerung) oder eine Ansteuerung des gesamten Fahrzeugs (Fahrzeugsteuerung) möglich. Um die Ebenen während der Ansteuerung zu trennen, wurden vier mögliche Modi definiert:

1. Antriebsteuerung (im Quellcode: `SINGLE_MOTOR` Mode)

   Nur die Steuerfunktionen der Antriebsteuerung sind aktiv. Alle Steuerfunktionen der Fahrzeugsteuerung werden nicht ausgeführt. Istwerte der Fahrzeugsteuerung können immer noch abgefragt werden.

2. Fahrzeugsteuerung (im Quellcode: `VEHICLE` Mode)

   Nur die Steuerfunktionen der Fahrzeugsteuerung sind aktiv. Alle Steuerfunktionen der Antriebsteuerung werden nicht ausgeführt. Wie im Modus Antriebsteuerung können die Istwerte der anderen Ebene ebenfalls erfasst werden.

3. Entwicklermodus (im Quellcode: `DEVELOPEMENT` Mode)

   In einem Entwicklungsmodus können beide Ebenen gleichzeitig betrieben werden. Dies kann aber zu Problemen führen, wenn zeitgleich eine Drehzahl und eine Geschwindigkeit vorgegeben werden.

4. Deaktiviert (im Quellcode: `OFF` Mode)

   Dieser Modus stellt sicher, dass von der Robotersteuerung-API keine Steuerbefehle an die Motoren gelangen. Istwerte beider Ebenen können nach wie vor erfasst werden. Damit kann die mobile Roboterplattform als reiner Sensor genutzt werden.

Die Modi können mit der Funktion `htbSetMode(int mode)` eingestellt und mit der Funktion `htbGetMode(int* mode)` abgefragt werden. Diese beiden Funktionen sind in allen Modi aufrufbar.

3.2.5 Übergabeparameter und Rückgabewerte

Der Aufruf der Funktionen der Robotersteuerung-API ähnelt dem der virtuHall-API. Allen Funktionen ist das Präfix „htb“ vorangesetzt. „htb“ soll die Zuordnung der Funktionen zu „HighTecBot“ erleichtern. Die Ist- und Sollwerte werden mithilfe der Übergabeparameter ausgetauscht. Istwerte werden dabei mit einem Zeiger übergeben. Der Rückgabewert einer Funktion stellt einen Fehlercode dar. Damit entspricht die Handhabung der Funktionen der Robotersteuerung API der virtuHall-API. Mögliche Rückgabewerte einer Funktion sind:

1. NO_ERROR

   Dieser Wert wird immer dann zurückgegeben, wenn die Funktion ohne erkennbare Fehler ausgeführt wurde. Alle Funktionen können diesen Wert zurückgeben. Damit kann ein Benutzer eine Fehlerbehandlung durchführen, indem der Rückgabewert der aufgerufenen Funktion auf Ungleichheit mit 0 überprüft wird.

   Codebeispiel:

   ```
   if(htbSetVelocity(500))
        {Fehlerbehandlung;}
   ```

   Kehrt `htbSetVelocity()` mit einem Rückgabewert ungleich Null zurück, wird die Fehlerbehandlung ausgeführt. Diese kann die nachfolgend aufgelisteten Fehlercodes noch auswerten.

2. SYSTEM_ERROR

   Dieser Wert wird immer dann zurückgegeben, wenn ein Fehler mit dem Basissystem auftritt, z.B. wenn die Funktion `vhSetPower()` mit einem Wert ungleich Null zurückkehrt. Ein Anwender kann diesen Fehler im Normalfall nicht beheben.

3. MODE_ERROR

   Wird eine Funktion aufgerufen, die im derzeit aktiven Modus deaktiviert ist, wird dieser Fehler zurückgegeben. Ein Benutzer muss sicherstellen, dass er sich im richtigen Modus befindet und ggf. wechseln.

4. INPUT_ERROR

   Dieser Wert wird zurückgegeben, wenn ein ungültiger Wert übergeben wird. Z.B. nimmt die Funktion `htbSetMotorRelative()` nur Werte zwischen -100 und +100 entgegen. Werte außerhalb dieser Grenzen verursachen den Fehler.

5. REG_ERROR

   Der Drehzahlregler führt während seiner Laufzeit eine Plausibilitätsprüfung durch (siehe Abschnitt 3.3.6). Wird eine Regelabweichung erkannt, die in einem definierten Zeitraum nicht auszuregeln ist, wird dieser Fehler gesetzt. Damit kann z.B. eine Überlast oder ein Hindernis erkannt werden.

Eine Liste aller aufrufbaren Funktionen mit Parameterbeschreibung und Rückgabewerten der Robotersteuerung-API befindet sich im Anhang A „Übersicht der Schnittstellenfunktionen“.

3.2.6 Benutzerdefinierte Maximalwerte

Um die Robotersteuerung an eine beliebige Plattform oder eine spezielle Aufgabe anzupassen, können von einem Benutzer beliebige Maximalwerte vorgegeben werden. Diese werden von den physikalischen Maximalwerten begrenzt. Auf der Ebene der Antriebsteuerung kann die Drehzahl der Motoren von einem Benutzer begrenzt werden, indem die Funktion `htbSetMotorMaxRPM(unsigned short id, int max_rpm)` aufgerufen wird. Intern wird dann die maximale Drehzahl dieses Motors auf den vorgegebenen Wert begrenzt. Möchte man die physikalische Grenze wieder einstellen, kann dem Parameter `max_rpm` der Wert `DEFAULT` übergeben werden. Alle Maximalwerte werden in einer Initialisierungsphase auf den physikalischen Maximalwert eingestellt. Der aktuelle maximale Wert kann mit `htbGetMotorMaxRPM(unsigned short id, int* max_rpm)` abgefragt werden.

Auf der Ebene der Fahrzeugsteuerung können die Maximalwerte ähnlich wie bei der Antriebsteuerung mit „`Set`“ und „`Get`“ eingestellt und abgefragt werden. Ein Zurücksetzen mit `DEFAULT` ist ebenfalls möglich. Folgende Maximalwerte können in der Fahrzeugsteuerung verändert werden:

Maximalwert	Beschreibung
acceleration	Die Beschleunigung des Fahrzeugs in der Linearbewegung
deceleration	Die Verzögerung des Fahrzeugs in der Linearbewegung
rotation acceleration	Die Winkelbeschleunigung des Fahrzeugs
rotation deceleration	Die Winkelverzögerung des Fahrzeugs
forward velocity	Lineargeschwindigkeit vorwärts
backward velocity	Lineargeschwindigkeit rückwärts
rotation forward velocity	Winkelgeschwindigkeit im Uhrzeigersinn
rotation backward velocity	Winkelgeschwindigkeit gegen den Uhrzeigersinn

Tabelle 8: Einstellbare Maximalwerte der Fahrzeugsteuerung

Die Maximalwerte ergeben sich jeweils aus den Bewegungen vorwärts und rückwärts sowie Drehungen rechts und links herum und deren Beschleunigungen und Verzögerungen.

Das Setzen der benutzerdefinierten Maximalwerte wird in den Schnittstellenfunktionen der API ausgeführt. Hier wird der interne Maximalwert in den jeweiligen globalen Variablen verändert. Die interne Funktion `checkInput(int* input, int max_value, int min_value, int mode)` prüft dabei, ob der Eingabewert

korrekt ist und erzeugt ggf. einen Fehler. Überschreitet die Eingabe die physikalische Grenze, wird der Fehler `INPUT_ERROR` ausgegeben und der Wert mithilfe der übergebenen Adresse auf die physikalische Grenze gesetzt. Das ist für den Fall wichtig, wenn ein Benutzer keine Fehlerbehandlung durchführt. Ist die Eingabe kleiner Null, wird ebenfalls der Fehler `INPUT_ERROR` ausgegeben. Bleibt der Eingabewert im gültigen Bereich, wird der Wert auf den die Adresse zeigt unverändert gelassen. Als Maximalwert kann immer nur eine positive Zahl eingegeben werden. Zusätzlich wird überprüft, ob der korrekte Modus für die gewünschte Aktion eingestellt ist. Siehe hierzu die Erläuterungen im Abschnitt 3.2.4.

Codeausschnitt der Funktion `checkInput()`:

```
//Initialisieren der globalen Variablen
static int mode_int;

int checkInput(int* input, int max_value, int min_value, int mode)
{
     //Initialisieren des Rückgabewertes
    int ret_val = NO_ERROR;

    //den korrekten Modus prüfen
    if((mode_int == mode) || (mode_int == DEVELOPEMENT))
    {
        //bei DEFAULT auf den physikalischen Maximalwert setzen
        if(*input == DEFAULT)
            *input = max_value;
        //obere Grenze prüfen
        else if(*input > max_value)
        {
            *input = max_value;
            ret_val = INPUT_ERROR;
        }
        //untere Grenze prüfen
        else if(*input < min_value)
        {
            *input = min_value;
            ret_val = INPUT_ERROR;
        }
    }
    else//bei falschem Modus Fehlerausgabe
        ret_val = MODE_ERROR;

    return ret_val;
}
```

Der Übergabeparameter `input` ist die Adresse des zu prüfenden Wertes. `max_value` und `min_value` stellen die obere und untere Grenze dar, die der Wert von `input` nicht über- bzw. unterschreiten darf. Der Parameter `mode` wird zur Prüfung des Betriebsmodus benötigt, die ebenfalls in dieser Funktion durchgeführt wird.

3.2.7 Die Robotersteuerung als Reglerstruktur

Aus regelungstechnischer Sicht bildet jede der beiden Ebenen ein Modul, in dem zum einen der Regelalgorithmus ausgeführt wird, zum anderen die Eingangs- und Ausgangsgrößen auf die physikalische Entsprechung in der Ebene umgerechnet werden. Beispielsweise benötigt die Regelung der Geschwindigkeit eine Vorgabe in m/s. Diese Vorgabe wird ausgeregelt und ein Istwert in m/s zurückgegeben. Die Ebene der Drehzahlregelung wird dabei nicht mit einem Wert in m/s gespeist, sondern mit einem umgerechneten Wert in U/min. Ein Rückgabewert wird auch in dieser Einheit entgegengenommen.

Das Konzept der zwei aufeinander aufbauenden Ebenen in der Robotersteuerung-API ergibt einen Regler mit Kaskadenstruktur. Eine Drehzahlregelung wird von einer Geschwindigkeitsregelung überlagert. Bedingung dafür ist, dass die Drehzahlregelung deutlich schneller arbeitet als die Geschwindigkeitsregelung und dass Drehzahl und Geschwindigkeit als Messgrößen vorliegen. Beide Bedingungen sind gegeben, da die Abarbeitungszeit durch periodische Ereignisse im Betriebsystems entsprechend eingestellt werden kann (siehe Abschnitt 3.2.8) und die Drehzahl der einzelnen Räder sowie die Geschwindigkeit des Gesamtfahrzeugs aus Übergabewerten der virtuHall-API errechnet werden können.

Damit wird folgende Reglerstruktur realisiert. Die detaillierte Struktur wird in Abschnitt 3.4.3 behandelt.

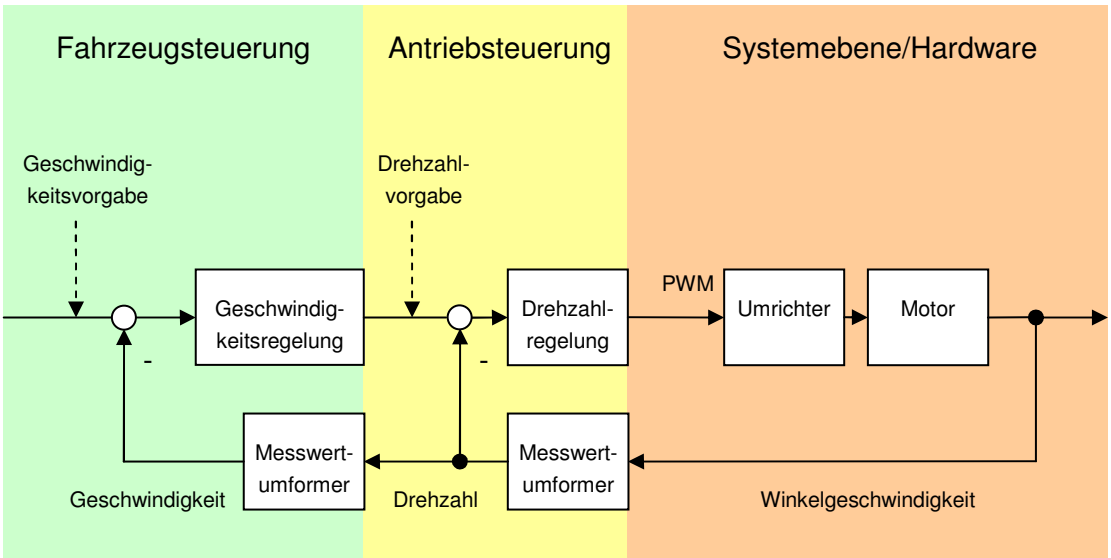


Abbildung 22: Schema der Reglerstruktur der Robotersteuerung

Eine Drehmomentregelung könnte der Drehzahlregelung unterlagert werden, um die Kraft am Rad zu regeln und zu begrenzen. Darauf muss an dieser Stelle aus zwei Gründen verzichtet werden.

1. Eine Momentregelung erfordert eine präzise Erfassung des aktuellen Momentes in Echtzeit. Da das Drehmoment eines Elektromotors nach Fuest [9] (S. 113ff) direkt vom Strom abhängt, kann es innerhalb kürzester Zeit auf ein Maximum ansteigen. Der nötige digitale Regelalgorithmus müsste deutlich öfter abgearbeitet werden als es in der Ebene möglich ist, in der die Robotersteuerung implementiert ist. Eine Momentregelung ist daher nur in einer hardwarenäheren Ebene möglich.
2. Wie in Punkt 1. erwähnt, muss für eine Momentregelung der aktuelle Strom in allen drei Phasen erfasst werden. Die Funktion `vhGetCVT()` liefert den Strom in allen drei Phasen des Motors. Eine Auswertung des Stroms bringt nur unzureichend genaue Messwerte. Da die Leistungselektronik für einen Maximalstrom von bis zu 45 A ausgelegt ist, löst der 12 Bit A/D-Wandler seinen Ausgabewert bis zum Maximalstrom auf. Werden die Motoren maximal belastet, fließt ein Strom von 5,2 A, was nur einem kleinen Teilbereich des A/D-Wandlers entspricht. Eine Stromänderung von 0,1 A ist unter Berücksichtigung des Messfehlers des A/D-Wandlers und des Rauschens kaum wahrnehmbar, da sie aufgrund der Auflösung unter neun Schritten liegt. Damit ist eine sensible Begrenzung oder Regelung mit dieser Messmethode nicht möglich.

3.2.8 Sicherstellen der harten Echtzeit

Wie in Abschnitt 2.4 beschrieben, benötigt ein digitaler Regelalgorithmus einen verwertbaren Istwert in Bezug auf die Zeit. Dazu ist es nötig, entweder einen Zeitstempel für jedes Datum zu erfassen oder eine bekannte periodische Zeit genau einzuhalten. Wird der Istwert einem Zeitstempel zugeordnet, kann zwischen den Abtastschritten Programmcode variabler Länge ausgeführt werden. Jedoch ist es bei dieser Methode nötig, die Zeit zwischen den Abtastschritten genau zu messen. In der Mikrokontrollertechnik ist dies mit einem Timer möglich, der z.B. im Mikrosekundentakt zählt. Dieser Timer kann bei jedem Abtastschritt ausgelesen und neu gestartet werden.

PXROS-HR bedient sich solcher Timer, um eine eigene variable Zeiteinheit, die PxTicks zu erzeugen. Diese sind jedoch nicht im unteren µs-Bereich skalierbar, da hierdurch eine zu große Prozessorlast für die Zeitbehandlung entstehen würde. Ein PxTick entspricht in der derzeitigen Konfiguration genau einer Millisekunde. Da von einem nachladbaren Task aus kein Zugriff auf einen Hardware-Timer möglich ist, ist eine ausreichend genaue Zeitmessung mit der Methode des Zeitstempels nicht möglich.

Da das Betriebssystem PXROS-HR die PxTicks genau zur Verfügung stellt, lässt sich ein periodisches Ereignis nutzen, mit dem die Abtastung der Istwerte erfolgen kann. Eine interne Uhr in PXROS-HR läuft im Takt der PxTicks. Mit der Funktion `PxPeRequest()` wird ein periodisches Ereignis definiert, das immer nach einer definierten Anzahl von PxTicks ausgelöst wird. Dieser Software-Interrupt hat eine hohe Priorität, damit die Zeit immer eingehalten werden kann (siehe Abschnitt 2.5.2). Mit der Funktion `PxAwaitEvents()` wird auf die Auslösung des Zeitereignisses gewartet. Ist es ausgelöst, werden Abtastvorgang und Regelalgorithmus ausgeführt. Ist es nicht ausgelöst, geht der Task in den Zustand „wartend“ über. Andere, niedriger priorisierte Tasks (z.B. die Anwender-Task) können ausgeführt werden, bis das Ereignis ausgelöst wird. Damit ist die Anforderung aus Kapitel 1.2 erfüllt, dass der Task nicht den Prozessor blockiert und sich somit in das Echtzeitbetriebsystem einbinden lässt.

Das Abfragen des Ereignisses kann auch auf andere Weise geschehen. Mit `PxResetEvents()` wird nur überprüft, ob ein bestimmtes Ereignis eingetreten ist, aber nicht gewartet. Ist ein Ereignis eingetreten, wird es von der Funktion zurückgesetzt und die Funktion liefert einen Wert ungleich Null zurück.

Für die Robotersteuerung ist es nötig, eine Kaskadenstruktur aufzubauen, wie es in Abschnitt 3.2.7 „Die Robotersteuerung als Reglerstruktur“ beschrieben wird. Dazu sind zwei verschiedene zeitliche Zyklen notwendig. Die unterlagerte Drehzahlregelung muss deutlich schneller arbeiten als die überlagerte Geschwindigkeitsregelung. Um dies umzusetzen, werden zwei verschiedene periodische Ereignisse benötigt. Eine Abschätzung der maximalen Periodendauer zur Sicherstellung einer korrekten und sauberen Regelung, die das Nyquist-Kriterium erfüllt, ist in [18] (S. 468f) aufgeführt. Nach [10] (S. 337f) muss die Abtastfrequenz mindestens doppelt so groß sein wie die abzutastende Frequenz (Shannon-Theorem). In dieser Arbeit wird auf eine saubere Regelung geachtet. Daher werden die Regelzyklen möglichst schnell ausgelegt. Das Ereignis für die Drehzahlregelung wird alle zwei Millisekunden ausgelöst und das Ereignis für die Geschwindigkeitsregelung alle acht Millisekunden. Damit arbeitet der unterlagerte Regelkreis der Drehzahlregelung viermal so schnell wie der überlagerte Regelkreis. Außerdem sind die Bedingungen für das korrekte Erfassen des Signals erfüllt.

Nachfolgend ist der Code dargestellt, der zum Initialisieren und Nutzen von periodischen Ereignissen in der Robotersteuerung implementiert wird. Die beiden verschiedenen Ebenen Drehzahl- und Geschwindigkeitsregelung sind an den Kürzeln („_RPM“ - Rotation per Minute und „_VEL“ - Velocity) erkennbar.

Codeausschnitt Zeitbehandlung:

```
//Events definieren. Das 10. und 11. Bit wird hierfür genutzt.
#define   EVENT_RPM   (1<<10)
#define   EVENT_VEL   (1<<11)

//legt die Anzahl der PxTicks fest, nach der ein Ereignis ausgelöst
//wird. Ein PxTick entspricht einer Millisekunde
#define   PE_TIME_RPM   2
#define   PE_TIME_VEL   8

//Variablen für die Events definieren
Pxpe_t    pe_rpm;
Pxpe_t    pe_vel;

//Die Events mit der Zeit und dem Bit für das Event initialisieren
pe_rpm = PxPeRequest(PXOpoolTaskdefault, PE_TIME_RPM, EVENT_RPM);
pe_vel = PePeRequest(PXOpoolTaskdefault, PE_TIME_VEL, EVENT_VEL);

//Die Events starten. Ab hier werden periodische Ereignisse ausgelöst.
PxPeStart(pe_rpm);
PxPeStart(pe_vel);

while(1)//Hauptschleife
{
     //Auf das Event EVENT_RPM Warten und ggf. handeln.
     if(PxAwaitEvents(EVENT_RPM) == EVENT_RPM)
     {
          //Das Event zurücksetzen, damit es erneut ausgelöst werden
          //kann.
          PxClearModebits(PXTmodeDisableAborts);

          // Drehzahlregelung
          Abtasten_des_Istwertes_Drehzahl;
          Regelung_der_Drehzahl;

          //Prüfen, ob das zweite Event EVENT_VEL auch ausgelöst wurde
          if(PxResetEvents(EVENT_VEL) == EVENT_VEL)
          {
               //Ein Zurücksetzen des Ereignisses entfällt

               //Geschwindigkeitsregelung
               Errechnen_der_Geschwindigkeiten;
               Regelung_der_Geschwindigkeiten;
          }
     }
}//Ende Hauptschleife
```

Antriebsteuerung/Drehzahlregelung

Fahrzeugsteuerung/Geschwindigkeitsregelung

Anstatt der Funktion `PxAwaitEvents()` wird im Quellcode die Funktion `PxMsgReceive_EvWait()` benutzt. Der Unterschied besteht darin, dass nicht nur auf ein Ereignis gewartet wird, sondern auch auf das Eintreffen einer Nachricht von einem anderen Task. Das ist nötig, um die in Abschnitt 3.2.3 beschriebene Struktur der Kommunikation zu realisieren. Um den Code, der die harte Echtzeit sicherstellt, überschaubar zu halten, wurde im Beispiel die Funktion `PxAwaitEvents()` verwendet.

Die Einhaltung der periodischen Zeit ist solange gegeben, wie höher priorisierte Tasks nicht die Rechenzeit des Prozessors voll belegen. Um sicherzustellen, dass genügend Zeit für die Abarbeitung des eigenen Tasks bleibt, wurde das zeitliche Verhalten mit dem Programm PxView untersucht.

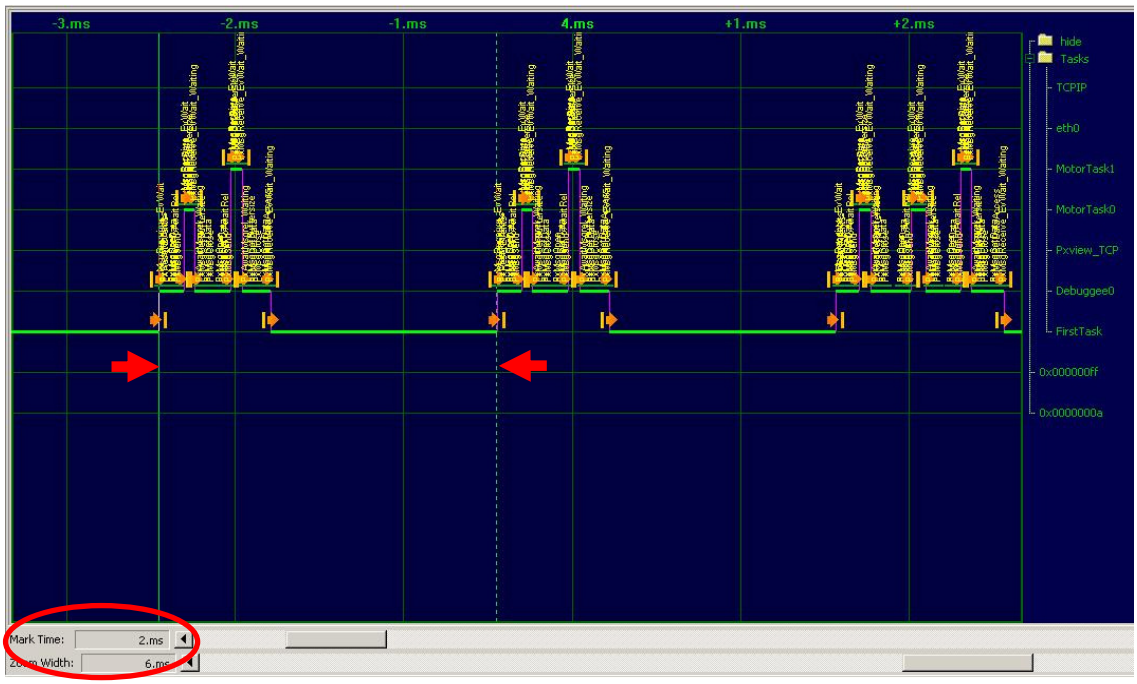


Abbildung 23: Zeitverhalten von PXROS-HR

Die grüne Linie ist die Prozessorzeit, die ein Task belegt. Die gelben Schriftzüge zeigen Systemfunktionen an, die zu genau diesem Zeitpunkt aufgerufen werden. Die zugehörige Zeitskala ist oben aufgetragen. Rechts sind die beteiligten Tasks aufgelistet, darunter „Motor Task0“ und „Motor Task1“, die beide zur virtuHall-API gehören. Der Task „Debuggee0“ ist der gerade untersuchte Task. Die „FirstTask“ führt die Leerlaufzeit aus. Links, mit einem roten Pfeil markiert, wechselt die Taskzuteilung gerade zur untersuchten Robotersteuerung-Task. Innerhalb einer Abarbeitungsphase wird jeweils einmal zur „Motor Task0“ und zur „Motor Task1“ gewechselt. Das ist der Aufruf der Funktion `vhSetPower()`, der jeweils eine Intertaskkommunikation zur Folge hat. Nachdem ein Zyklus vorbei ist, wird in den Leerlaufprozess gewechselt. Nach exakt zwei Millisekunden, mit dem rechten roten Pfeil markiert, beginnt ein neuer Zyklus. Die zeitliche Periode kann eingehalten werden. Es ist sogar reichlich Puffer vorhanden. Ein vom Benutzer der API erstellter nachladbarer Task hat eine niedrigere Priorität als die

Robotersteuerung-Task. Niedriger priorisierte Tasks können die Zykluszeit von zwei Millisekunden nicht beeinflussen, da sie bei Eintreten eines periodischen Ereignisses unterbrochen werden.

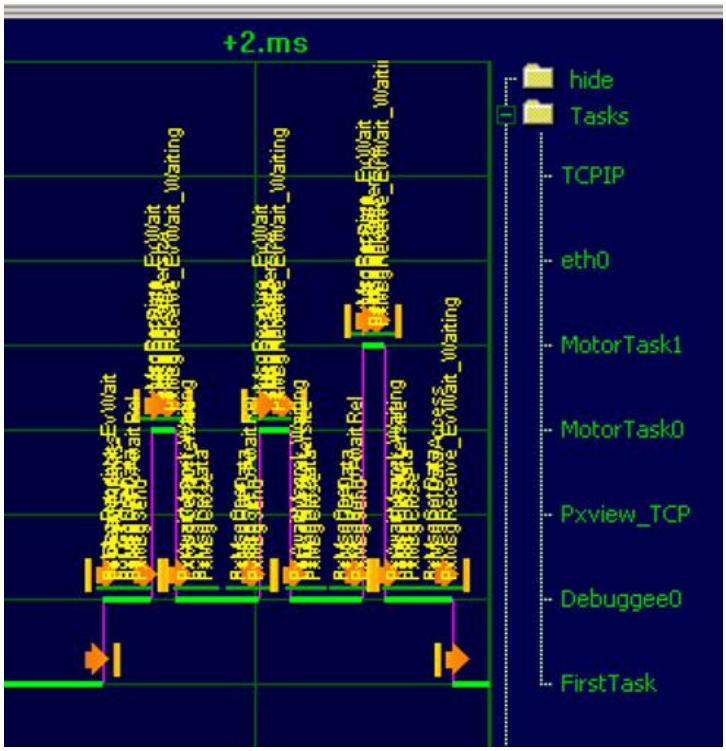


Abbildung 24: Zyklusfigur der Fahrzeugsteuerung (Ausschnitt aus PxView)

Am rechten Rand ist eine Zyklusfigur mit drei Intertaskkommunikationen zu sehen. Das ist der Zyklus der Fahrzeugsteuerung, der alle acht Millisekunden abgearbeitet wird. Die dritte Intertaskkommunikation (die erste in der Zyklusfigur) kommt durch die Abfrage des Schalters zustande.

Ein Zyklus der Antriebsteuerung dauert 665 µs und ein Zyklus der Fahrzeugsteuerung, der die Antriebsteuerung beinhaltet, 995 µs. Davon wird ein Großteil der Zeit für die Intertaskkommunikation und die Abarbeitung des Auftrages an die virtuHall-API benötigt. Die Kommunikation mit der Motortask und die Abarbeitung eines Auftrages dauert etwa 225 µs. Eine schnellere Kommunikation und Abarbeitung würde erreicht, wenn die Regelung nicht als nachladbarer Task im RAM-Speicher, sondern als Teil des Basissystems im vielfach schnelleren Flash-Speicher ausgeführt würde. Zudem ist das Basissystem momentan nicht auf Zeit optimiert und die Mitführung der Statistiken für PxView benötigt zusätzliche Ausführungszeit. Daher ist bei der Programmierung darauf geachtet worden, möglichst wenige Taskwechsel zu verursachen.

Wird die Robotersteuerung als API genutzt, muss eine nachladbare Anwender-Task existieren (siehe Abschnitt 3.2). Da diese aber eine geringere Priorität besitzt als der Task der Robotersteuerung, kann die geforderte Echtzeit eingehalten werden. Die Anwender-Task wird nach Ablauf der zwei Millisekunden unterbrochen und die Robotersteuerung ausgeführt.

3.3 Antriebsteuerung

3.3.1 Allgemeine Strukturen

Auf der Ebene der Antriebsteuerung können die Antriebsräder einzeln angesprochen werden. Damit lässt sich der Roboter in alle Richtungen bewegen. Die Vorgabe des Sollwertes kann auf zwei verschiedene Arten getätigt werden. Der physikalische Bezug wird mit einer Vorgabe in Umdrehungen pro Minute hergestellt. Dabei kann eine Maximaldrehzahl, die von der Bauart der Motoren bestimmt wird, nicht überschritten werden. Die physikalische Maximaldrehzahl ist daher für Anwender unveränderbar, kann aber abgefragt werden. Mit einer relativen Vorgabe kann ein Motor prozentual in beide Richtungen angetrieben werden. Dementsprechend sind Werte zwischen -100 … +100 Prozent möglich. Diese Werte werden linear auf den zulässigen Bereich bis zur Maximaldrehzahl abgebildet.

Da für alle Antriebsräder die gleichen Berechnungen durchzuführen sind, existiert der Code nur einmal. Jede modulglobale Variable der Antriebsteuerung ist ein Feld der Länge der Anzahl der Motoren. Mit jedem Durchführen der Berechnungen wird die Motorenidentifikation mit übergeben, mit der dann die individuellen Werte jedes Motors in dem entsprechenden Feld einer Variablen gespeichert werden können. Obwohl die Berechnungen vom gleichen Code ausgeführt werden, können die Motoren durch die Maximalwerte und radspezifischen Werte verschieden parametrisiert werden.

Die Motoridentifikation ist bei der vorgegebenen mobilen Roboterplattform auf zwei verschiedene Arten möglich. Übergibt man die Identifikation `MOTOR_1` oder `MOTOR_2`, wird der jeweilige Motor in seiner natürlichen Richtung angetrieben. Betrachtet man einen Motor von der Hohlwelle aus, ist das eine Bewegung im Uhrzeigersinn. Da die Motoren an der mobilen Roboterplattform als Differenzialantrieb angebracht sind, würde sie sich bei einer Ansteuerung der beiden Motoren mit einem gleichen positiven Wert auf der Stelle drehen. Um in der Antriebsteuerung die mobile Roboterplattform als Fahrzeug zu sehen, können die Motoren auch mit der Identifikation `MOTOR_RIGHT` und `MOTOR_LEFT` angesteuert werden. Dabei wird das rechte Rad in der Antriebsteuerung invertiert, damit bei einer gleichen positiven Ansteuerung der Räder das Fahrzeug vorwärts fährt und nicht auf der Stelle dreht.

3.3.2 Istwerterfassung

Für die Erfassung der aktuellen Drehzahl bietet die virtuHall-API zwei verschiedene Möglichkeiten, die es zu untersuchen gilt.

Eine Möglichkeit der Drehzahlerfassung ist den Parameter `speed` der Funktion `vhSetPower(`**`motor_id_t`** `id, `**`short`** `pwm, `**`short*`** `speed)` auszuwerten. Vorteilhaft ist, dass mit nur einem Funktionsaufruf Sollwert vorgegeben und Istwert abgefragt werden können. Da jeder Funktionsaufruf der virtuHall-API eine Intertaskkommunikation im Betriebssystem zur Folge hat, kann mit dieser Methode Re-

chenzeit gespart werden. Nachteilig ist, dass bei dieser Vorgehensweise ein zeitlicher Verzug in Länge der periodischen Abarbeitungszeit `PE_TIME_RPM` entsteht, da der abgefragte Istwert erst am Ende des Zyklus zur Verfügung steht, gespeichert und zu Beginn eines neuen Zyklus verarbeitet wird. Zudem ist der Parameter `speed` undefiniert. Zwar kann durch eine Umrechnung die aktuelle Drehzahl bereitgestellt werden, diese ist aber nicht aus bekannten physikalischen Größen errechnet.

Die Alternative dazu ist eine eigene Geschwindigkeitsabfrage über die Funktion `vhGetPhase(`**`motor_id_t`** `id,` **`unsigned short*`** `phase)`. Hierbei ist vorteilhaft, dass über die Anzahl der Polpaare der Motoren ein direkter physikalischer Bezug zwischen der Änderung der Phase und der Drehzahl des Rades hergestellt werden kann.

i: Anzahl der Inkremente pro Umdrehung

p: Polpaarzahl des Motors

n: Auflösung eines A/D-Wandlers

$$i = p \cdot n \qquad i = 20 \cdot 2048 = 40960$$

Die Drehzahl lässt sich aufgrund der hohen Auflösung des A/D-Wandlers und der hohen Polpaarzahl sehr genau erfassen. Pro Umdrehung hat das Rad 40960 Inkremente. Nachteilig ist, dass dadurch der errechnete Wert für die Drehzahl mit starkem Rauschen behaftet ist. Um das Signal zu glätten, ist ein Filter nötig, der aber einen zeitlichen Verzug mit sich bringt. Ungünstig ist auch, dass ein Aufruf einer virtuHall-Funktion immer eine Intertaskkommunikation zur Folge hat und damit wesentlich mehr Rechenzeit für eine Periode benötigt wird. Zum Vergleich sind beide Signale ungefiltert dargestellt.

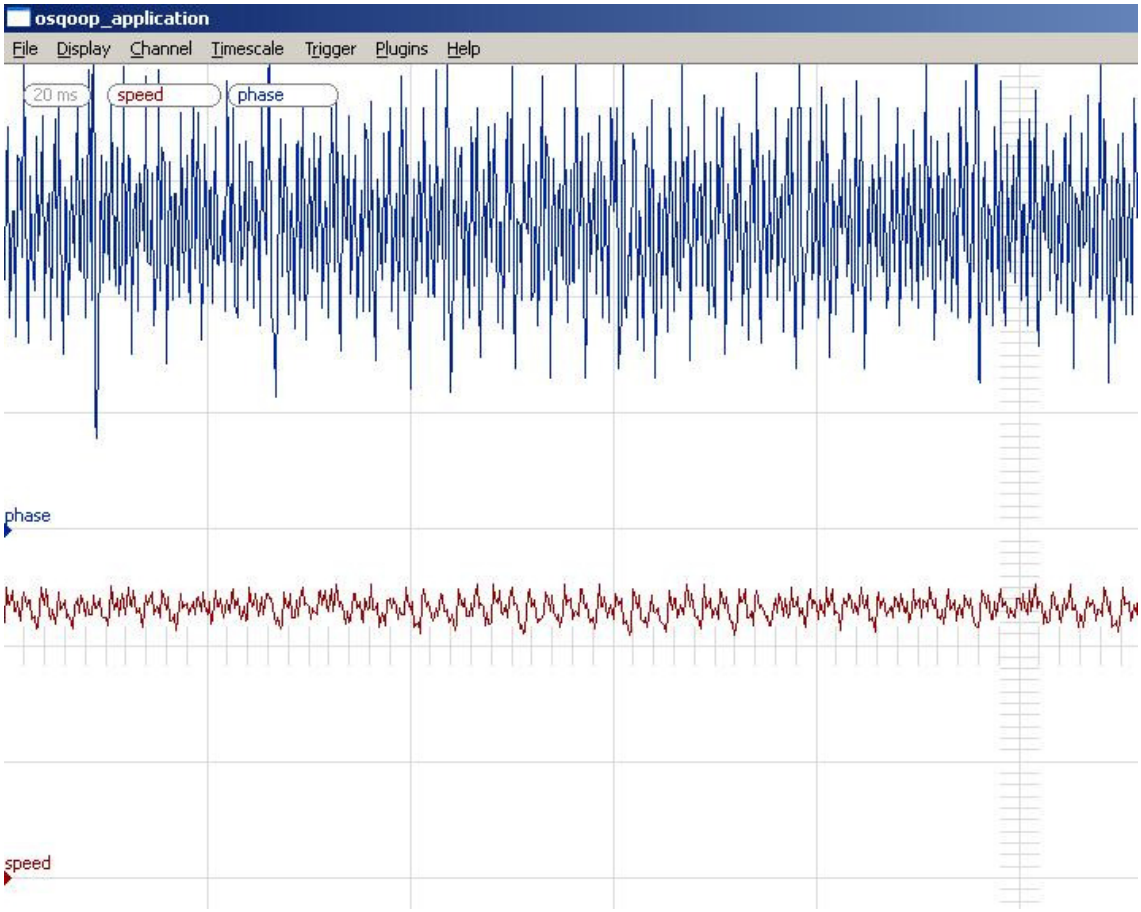


Abbildung 25: Vergleich zwischen "speed" und "phase"

Bei gleicher Skalierung im digitalen Oszilloskop ist das Rauschen beim Signal „phase“ deutlich stärker. Der Filter, der nötig ist, um das Signal zu glätten, bringt eine ähnlich lange Verzögerungszeit mit sich wie der zeitliche Verzug, der entsteht, wenn die Drehzahl mit dem Parameter `speed` erfasst wird. Jedoch kann bei Verwendung der Funktion `vhSetPower()` Rechenzeit gespart werden, da weniger Intertaskkommunikationen nötig sind. Daher wurde die Drehzahlerfassung über die Funktion `vhSetPower()` vorgenommen und die Verzögerungszeit der Länge der Abarbeitungszeit `PE_TIME_RPM` in Kauf genommen.

Der Übergabewert `speed` der Funktion `vhSetPower()` soll nun zur Drehzahlerfassung der Räder herangezogen werden, um die Drehzahl der Räder in Umdrehungen pro Minute zu messen. Der Wert von `speed` muss dazu umgerechnet werden. Da über den Wert `speed` und seine Zuordnung zu physikalischen Größen nichts bekannt ist, muss er zunächst analysiert werden. Mit dem Programm PxLogger können Variablen mitgeloggt und visuell dargestellt und damit analysiert werden. Dazu wurde ein Motor mit 50 % angesteuert und der Wert von `speed` aufgezeichnet.

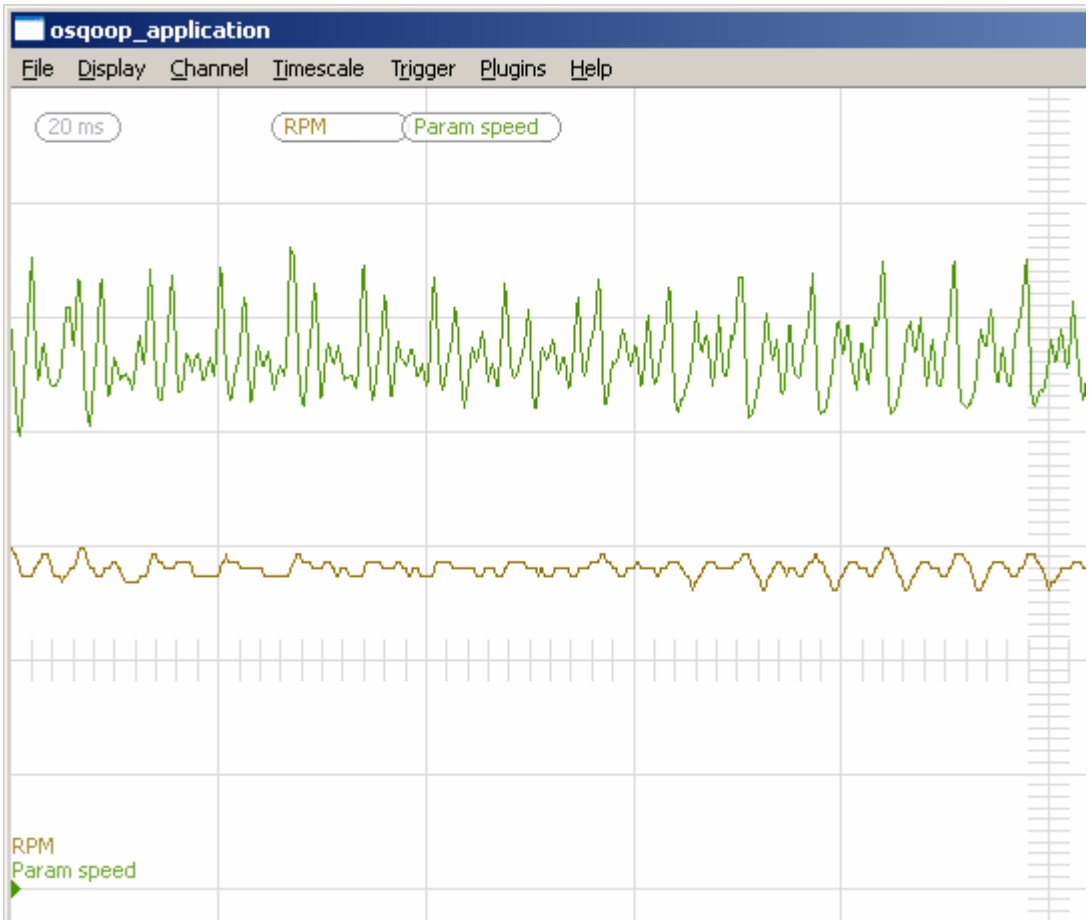


Abbildung 26: Rückgabewert "Param speed" und gefilterte "RPM"

Man erkennt, dass die Aufzeichnung von „Param speed“ verrauscht ist. Bei der Auswertung der mitgeschriebenen Werte konnte festgestellt werden, dass `speed` einen Durchschnittswert von 400 annimmt. Die tatsächliche Drehzahl des Rades beträgt 48 U/min. Damit ergibt sich für diesen Betriebspunkt ein Faktor von:

$$Faktor = \frac{speed}{Drehzahl} = \frac{400}{48} = 8{,}3\overline{3}$$

Diese Auswertung wurde auch für andere Betriebspunkte durchgeführt. Dabei stellte sich heraus, dass der gleiche Zusammenhang zwischen dem Wert von `speed` und der Drehzahl über den gesamten Wertebereich besteht. Zur Umrechnung von `speed` auf Drehzahl kann demnach ein proportionaler Faktor von 8,33 eingesetzt werden.

Um das Rauschen zu minimieren wurde ein variabler Filter eingesetzt, der bei jedem Schritt den Mittelwert aus den letzten zehn Werten bildet. Im folgenden Codeausschnitt sind die Variablen als Feld der Länge der Anzahl der Motoren definiert. Mit der Variablen `id` wird die Berechnung für den jeweiligen Motor durchgeführt. Der Codeausschnitt wird für jeden Motor einmal zyklisch durchlaufen.

Codeausschnitt des Filters:

```
//Anzahl der gespeicherten Werte zur Mittelwertbildung
#define RPM_FILTER   10
//Anzahl der Motoren
#define MOTORS       2
//Umrechnungsfaktor definieren
#define CALC_SPEED   8.333f

//Modulglobale Variablen:

//Feld, das die letzten x Werte für alle Motoren speichert
static float  middle[MOTORS][RPM_FILTER]   = {{0},{0}};
//Zähler zum Befüllen des Feldes
static int    filter_inkrement[MOTORS]     = {0};
//Variable für die aktuelle gefilterte Drehzahl
static float  current_rpm[MOTORS]          = {0};


//Zyklisch abzuarbeitender Codeabschmitt:

//Lokale Variablen
float  raw_rpm;
float  rpm;
int    i;

//Faktorisierung des Wertes von speed zur ungefilterten rpm
raw_rpm = speed / CALC_SPEED;

//Befüllen des Feldes
middle[id][filter_inkrement[id]] = raw_rpm;
//Zähler zum Befüllen des Feldes hochzählen und ggf. zurücksetzen
filter_inkrement[id]++;
if(filter_inkrement[id] >= RPM_FILTER)
     filter_inkrement[id] = 0;

//Summe aus allen gespeicherten Werten in rpm bilden
for(i = 0; i < RPM_FILTER; i++)
   rpm += middle[id][i];
//Mittelwert bilden
current_rpm[id] = rpm / RPM_FILTER;
```

Eine Mittelwertbildung bringt aus regelungstechnischer Sicht eine Totzeit mit sich. Diese sollte möglichst klein gehalten werden, um eine hohe Regelgüte zu erreichen. Das Ergebnis ist ein Kompromiss zwischen guter Filterung des Eingangssignals und möglichst kleiner Totzeit. In Abbildung 26 ist die gefilterte Drehzahl „RPM“ aus dem Wert `speed` zu sehen. Die Verbesserung des Signals ist deutlich sichtbar.

3.3.3 Mathematische Beschreibung der Strecke

Ein Kernelement der Antriebsteuerung ist die Drehzahlregelung. Ein vorgegebener Sollwert soll möglichst schnell und ohne großes Überschwingen erreicht werden. Im Vorfeld der Implementierung des Reglers wurde eine Sprungantwort des Systems aufgenommen, um festzustellen, um welche Art Strecke es sich handelt. Hierzu wurde das Programm PxLogger benutzt. Ein Motor wurde mit einer Drehzahlvorgabe von 50 U/min angesteuert und die resultierende Istdrehzahl aufgezeichnet.

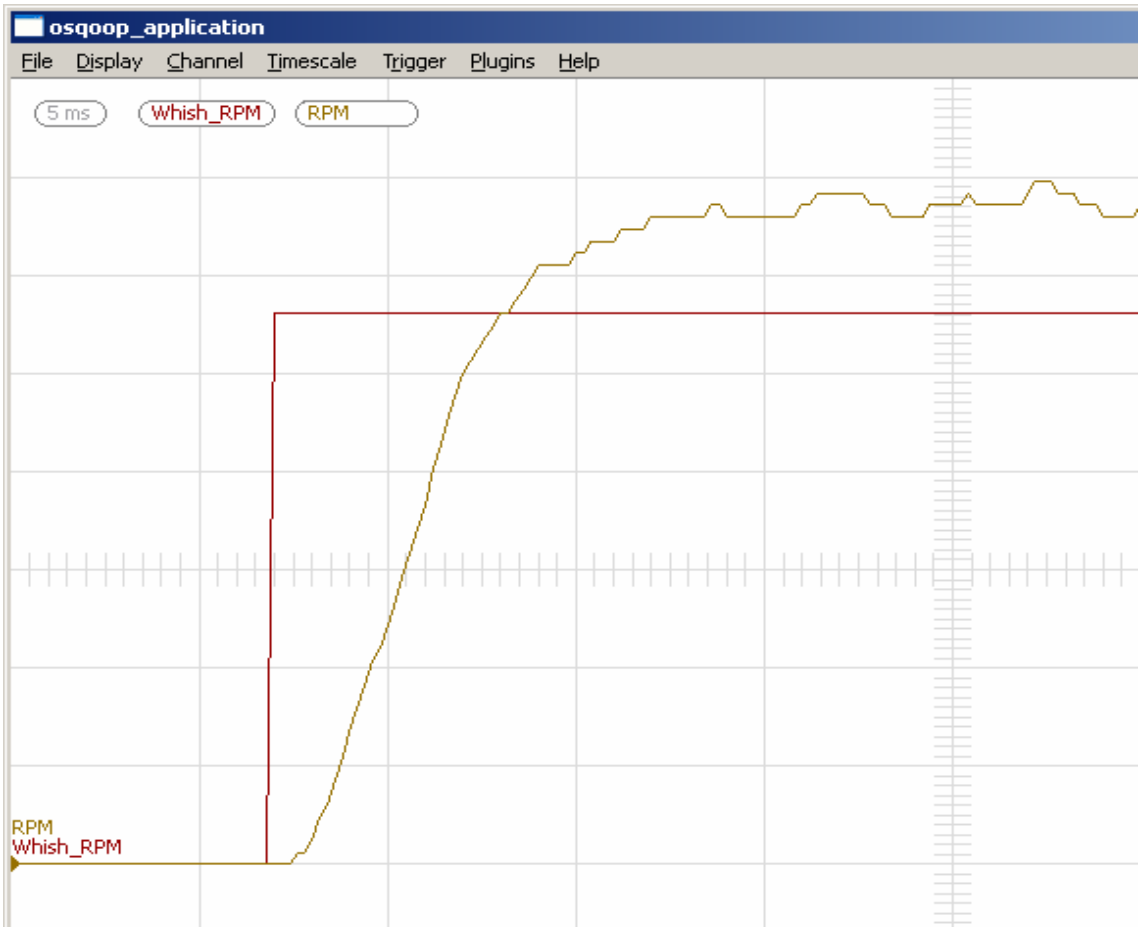


Abbildung 27: Sprungantwort eines Motors

Die Aufzeichnung in Abbildung 27 kann nun ausgewertet werden. Da die Istdrehzahl nach dem Aufschalten der Solldrehzahl nicht sofort ansteigt, ist eine kleine Totzeit vorhanden. Diese kann durch den Zeitverzug beim Abarbeiten des Algorithmus und durch den Umrichter und dessen Leistungstransistoren verursacht werden. Es ist auch zu erkennen, dass die Istdrehzahl nach der Totzeit nicht sofort steil, sondern verzögert ansteigt. Damit handelt es sich um ein System höherer Ordnung [25] (S. 210). Der Istwert nähert sich im weiteren Verlauf einem Endwert an, um den er aufgrund des Rauschens oszilliert. Das zeigt, dass die Strecke mit Ausgleich ist [25] (S. 211f). Sie verhält sich nicht wie ein reiner Integralteil. Mithilfe der Methode der Zeitprozentkennwerte

kann die Ordnung des Systems ermittelt werden. Die genaue Vorgehensweise dazu beschreiben Dörrscheidt und Latzel in [7] (S. 157ff). Sie ist im Anhang B „Ausführliche Berechnung des Drehzahlreglers“ aufgeführt. Aus der Auswertung konnte folgende mathematische Beschreibung für die Strecke abgeleitet werden:

$$G(s) = \frac{K_s}{(1 + sT_M)^n} = \frac{1{,}2}{(1 + s0{,}0144s)^3}$$

Es handelt sich um eine Strecke 3. Ordnung mit einem Proportionalbeiwert von 1,2 und einer Zeitkonstante von 0,0144 s. Diese Werte können später für die Parametrisierung des Reglers genutzt werden.

3.3.4 Implementierung des Regelalgorithmus

Nachdem die Strecke bekannt ist, kann der entsprechende Regler dazu entworfen werden. Da die Solldrehzahl später genau ausgeregelt werden soll, kommt ein P-Regler nicht in Frage, da dieser ohne Ausgleich arbeitet, also eine bleibende Regeldifferenz hinterlassen kann. Ein zusätzlicher I-Anteil würde die Regeldifferenz beseitigen. Ein PI-Regler wäre demnach denkbar. Da das Eingangssignal verrauscht ist, würde sich ein D-Anteil destabilisierend auswirken und wird nicht verwendet. Beim Entwurf dieses Reglers wird die Drehzahlregelung als einschleifiger Regelkreis gesehen und nicht als Teil einer Kaskadenregelung.

Um einen flexiblen Regelalgorithmus zu erhalten, wird ein PID-Regler so implementiert, dass P-, I- und D-Anteil abgeschaltet werden können. Im Programmcode sind daher alle Teile des Reglers voneinander getrennt. Dies ist möglich, wenn man die Nachstellzeit T_n und die Vorhaltezeit T_v in Faktoren für die einzelnen Kanäle konvertiert [22].

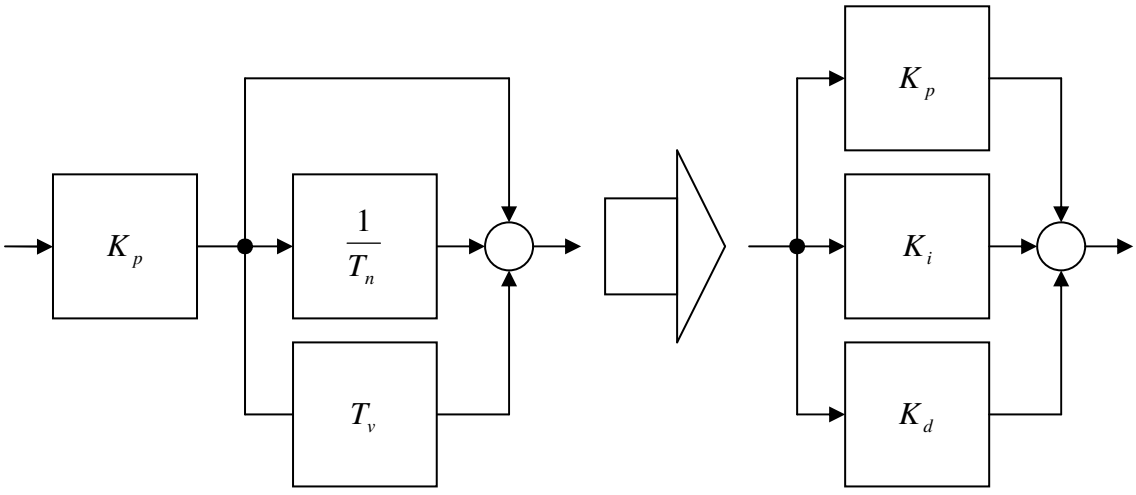


Abbildung 28: Konvertierung der Reglerteile in einzelne Kanäle

Mathematisch ergibt sich:

$$K_i = \frac{K_p}{T_n} \qquad K_d = K_p \cdot T_v$$

Um einen Kanal zu deaktivieren, reicht es aus, wenn ein Benutzer für T_n oder T_v Null eingibt. Eine Abfrage vor dem entsprechenden Programmcode setzt dies dann um. Wird für T_n eine Null eingegeben, wird mithilfe einer Abfrage K_i auf Null gesetzt, da eine Division durch Null nicht erlaubt ist.

Nachfolgend ist der Programmcode aufgeführt, der den digitalen PID-Regler umsetzt [5] (S. 395ff). Wie bei der Istwerterfassung speichern die modulglobalen Variablen in einem Feld der Länge der Anzahl der Motoren die jeweiligen Daten. Mit jedem Zyklus wird die Routine mit der Identifikation `id` für jeden Motor aufgerufen. Alle spezifischen Parameter haben das Kürzel `_rpm`, um sie von den Geschwindigkeitsreglern zu unterscheiden, die im Abschnitt 3.4.4 behandelt werden.

```
//Parameter des PID-Reglers (D-Anteil deaktiviert)
#define   KP_RPM    0.77f
#fefine   TN_RPM    0.026f
#define   TV_RPM    0.0f

//Modulglobale Variablen
static float rpm_integral[MOTORS] = {0};   //Speicher Integralteil
static float rpm_temp_diff[MOTORS] = {0};  //Speicher Differentialteil
static float kp_rpm = 0;
static float ki_rpm = 0;
static float kd_rpm = 0;

//Initialisierung der Parameter kp, ki und kd
kp_rpm = KP_RPM;

if(TN_RPM > 0)
    ki_rpm = KP_RPM/TN_RPM;
else   //Division durch Null nicht erlaubt
    ki_rpm = 0;

kd_rpm = KP_RPM * TV_RPM;


//Zyklisch abzuarbeitender Programmcode:

//Lokale Variablen
float differential = 0;   //Zwischenspeicher Differential
float delta_rpm = 0;      //Speicher Regelungsdifferenz
float new_rpm = 0;        //Speicher neue Drehzahl
```

```
//PID-Regler, Berechnung der Regeldifferenz
delta_rpm = wish_rpm[id] - (current_rpm[id]);

//Berechnung des Integralteils
if(ki_rpm > 0)
{
     rpm_integral[id] = rpm_integral[id] +
          ((ki_rpm * delta_rpm * PE_TIME_RPM) / 1000);

//Anti-Wind-Up begrenzt den Integralteil
if(rpm_integral[id] > (2 * max_motor_rpm[id]))
     rpm_integral[id] = (2 * max_motor_rpm[id]);
else if(rpm_integral[id] < (2 * -max_motor_rpm[id]))
     rpm_integral[id] = (2 * -max_motor_rpm[id]);

}
else   //Integralteil deaktiviert
     rpm_integral[id] = 0;

//Berechnung des Differenzialteils
if(kd_rpm > 0)
{
     differential =(kd_rpm * (1000 / PE_TIME_RPM) *
            (delta_rpm - rpm_temp_diff[id]));
      rpm_temp_diff[id] = delta_rpm;
}
else   //Diffentialteil deaktiviert
     differential = 0;

//Summieren von P-, I- und D-Teil
new_rpm = (kp_rpm * delta_rpm) + rpm_integral[id] + differential;

//Begrenzen der neuen Drehzahl auf die zulässigen Maximalwerte
if(new_rpm > max_motor_rpm[id])
     new_rpm = max_motor_rpm[id];
else if(new_rpm < -max_motor_rpm[id])
      new_rpm = -max_motor_rpm[id];
```

Die Zeiten T_n und T_v liegen in Sekunden vor, der Regelalgorithmus wird jedoch in der periodischen Zeit `PE_TIME_RPM` abgearbeitet. Dies macht eine zusätzliche Umrechnung der Werte im Integral- und Differentialteil notwendig.

3.3.5 Parametrisierung des Reglers

Um die passenden Parameter zu ermitteln, gibt es einige Möglichkeiten:

- Empirisches Einstellen
- Einstellregeln anwenden
- Berechnung der Parameter

Da sowohl eine Sprungantwort als auch die mathematische Beschreibung der Strecke vorliegen, können alle genannten Möglichkeiten angewendet werden. Die ausführliche Berechnung der Parameter befindet sich im Anhang B „Ausführliche Berechnung des Drehzahlreglers“. Die Vorgehensweise dazu kann in [3] nachgelesen werden.

Um die passenden Parameter zu ermitteln, wurden alle drei genannten Methoden benutzt. Zunächst wurde mittels Einstellregeln und Berechnung der Parameter auf eine Überschwingweite von 10% ein erster Parametersatz erstellt. Dieser wurde dann durch empirisches Einstellen weiter optimiert. Die verwendeten Parameter sind:

Proportionalbeiwert: $K_p = 0{,}77$

Nachstellzeit: $T_n = 0{,}026\,\mathrm{s}$

Vorhaltezeit: $T_v = 0\,\mathrm{s}$

Das Ergebnis lässt sich mit PxLogger sichtbar machen. Dazu wurde ein Motor mit 100 % angesteuert und Ist- und Solldrehzahl aufgezeichnet.

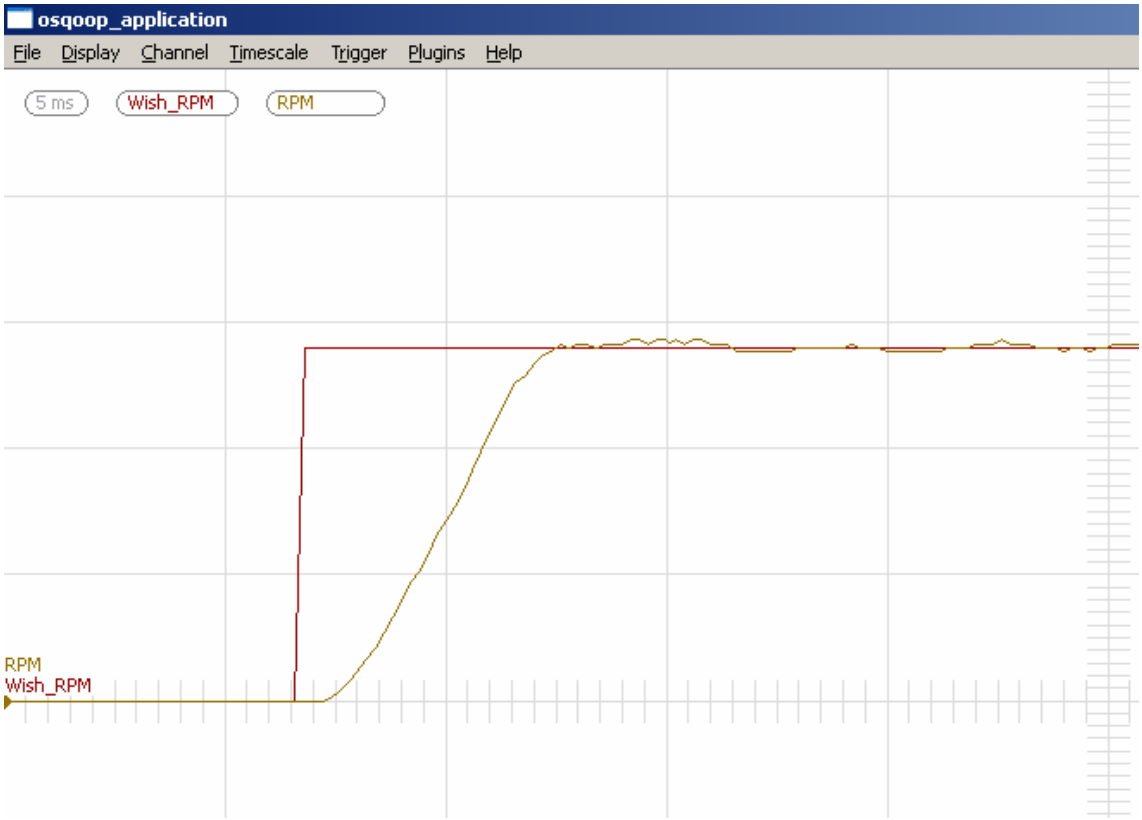


Abbildung 29: Soll- und Istwert der parametrisierten Drehzahlregelung

Die Istgröße „RPM“ wird mit geringem Überschwingen ohne bleibende Regeldifferenz ausgeregelt. Die Anregelzeit beträgt ca. 60 ms. Würde man die Parameter so verändern, dass ein größeres Überschwingen stattfindet, könnte die Anregelzeit verkleinert werden. Tests haben aber gezeigt, dass die mobile Roboterplattform dann zum Ruckeln neigt und sich evtl. aufschwingen kann. Mit einem PID-Regler wäre ebenfalls eine schnellere Ausregelung zu erreichen. Der D-Anteil würde jedoch dazu führen, dass bei kleinen Störungen sofort ein starkes Gegensteuern eintritt und bei verrauschtem Signal eine eher destabilisierende Wirkung zustande kommt. Als eine weitergehende Lösung wäre z.B. ein Zustandsregler zu implementieren, der erweiterte Möglichkeiten zur Parametrisierung bietet.

3.3.6 Plausibilitätsprüfung des Reglers

Wird eine mobile Roboterplattform in Verkehr gebracht, muss sie einem hohen Sicherheitsstandard genügen. Zu Sicherheitsmaßnahmen, die in der Software implementiert werden können, gehören unter anderem Drehmomentüberwachung und -begrenzung, Abschalten bei Überlast und Plausibilitätsprüfungen der eigenen Parameter. Maßnahmen, die Strom und Momentüberwachung voraussetzen, können aus den bereits genannten Gründen (Abschnitt 3.2.7) nicht umgesetzt werden. Eine Plausibilitätsprüfung des Reglers ist jedoch möglich.

Im Normalfall kann ein gut eingestellter Regler die vorgegebene Sollgröße innerhalb einer bestimmten Zeitspanne einstellen. Störungen, die z.B. durch einen Lastwechsel zustande kommen, werden ebenfalls innerhalb einer bestimmten Zeit wieder ausgeglichen. In Regelgrößen ausgedrückt, wird der Istwert dem Sollwert innerhalb eines Zeitrahmens angeglichen. Ist dies nicht der Fall, liegt ein Fehler vor und das System sollte abschalten. Fährt die mobile Roboterplattform z.B. gegen ein Hindernis, können sich die Räder gar nicht oder nur verzögert weiterdrehen. Die Istdrehzahl fällt ab. Der Regler wird versuchen, diese Störung auszuregeln. Schafft er dies nicht innerhalb einer bestimmten Zeit, muss angenommen werden, dass es sich nicht um einen Lastwechsel während der Fahrt, sondern um ein Hindernis handelt. Die Antriebe werden dann abgeschaltet und der Fehlercode `REG_ERROR` erzeugt. Die Istdrehzahl wird auf Plausibilität im Vergleich zur Solldrehzahl geprüft.

Diese Funktionalität ist als Programmcode im Funktionsblock des Drehzahlreglers implementiert, damit jedes Rad einzeln überwacht wird. Dazu wird in jedem Zyklus die Regelabweichung zwischen Soll- und Istwert errechnet. Überschreitet diese Abweichung über einen definierten Zeitraum einen Wert, kommt es zur Abschaltung der Antriebe.

Codeausschnitt zur Plausibilitätsprüfung:

```
//Definition der maximalen Abweichung (Grenzwert)
#define EMERGENCY_FILTER    25
//Definition der Zeit. Ein Zyklus ist PE_TIME_RPM lang, also 2 ms
#define EMERGENCY_CYCLES    100

//Modulglobale Variablen
static int error_rpm[MOTORS] = {0};   //Speichert die Abweichung
static int em_cycles[MOTORS] = {0};   //Zähler für die Zyklen

//Errechnen der Abweichung zwischen Ist- und Sollwert
error_rpm[id] = wish_rpm[id] - current_rpm[id];

//Unterdrückt die Überprüfung beim Abbremsen
if(((wish_rpm[id] > 0) && (error_rpm[id] > 0)) ||
     ((wish_rpm[id] < 0) && (error_rpm[id] < 0)))
```

```
{
     //Prüft eine einmalige Überschreitung des Grenzwertes
     if((error_rpm[id] > EMERGENCY_FILTER) ||
     (error_rpm[id] < -EMERGENCY_FILTER))
     {
          //Inkrementiert den Zähler
          em_cycles[id]++;
          //Prüft, ob die Abweichung länger als die definierte
          //Zeit anhält
          if(em_cycles[id] >= EMERGENCY_CYCLES)
          {
               //Die Motoren werden direkt abgeschaltet
               vhSetPower(MOTOR_1, 0, &speed);
               vhSetPower(MOTOR_2, 0, &speed);
               //Der Funktionsblock kehrt mit einem Fehlercode zurück
               ret_val = REG_ERROR;
          }
     }
     else      //Bei keiner Abweichung wird der Zähler zurückgesetzt
          em_cycles[id] = 0;
}
```

Beim Abbremsen der mobilen Roboterplattform kann es dazu kommen, dass die Masse des Fahrzeugs eine Regelabweichung über die definierte Zeitgrenze hinaus erzeugt und ein Fehler ausgelöst wird, ohne dass ein Hindernis vorhanden ist. Daher wird die Überprüfung beim Abbremsvorgang unterdrückt. Dies behindert die Plausibilitätsprüfung nur unwesentlich. Zwei Fälle sind denkbar:

1. Die mobile Roboterplattform bremst auf Null ab oder kehrt ihre Bewegungsrichtung um und gerät gegen ein Hindernis. Das Fahrzeug bleibt in absehbarer Zeit sowieso stehen bzw. fährt vom Hindernis weg. Wäre die Plausibilitätsprüfung aktiv und könnte das Hindernis erkennen, wäre die Folge ein Abschalten der Motoren. Diesen Zustand nimmt das Fahrzeug aber gerade sowieso schon ein, bzw. entfernt sich vom Hindernis. Dieser Fall stellt daher kein Problem dar.
2. Die mobile Roboterplattform bremst auf eine niedrigere Geschwindigkeit in gleicher Bewegungsrichtung ab und gerät während des Bremsvorgangs gegen ein Hindernis. Nach dem Auftreffen auf das Hindernis fällt die Istdrehzahl unter die Solldrehzahl ab. Dieser neue Zustand entspricht nicht mehr dem Abbremsen und die Plausibilitätsprüfung wird aktiv. Die Antriebe werden abgeschaltet.

Problematisch kann es werden, wenn der Wert von `EMERGENCY_FILTER`, also der Grenzwert, zu hoch eingestellt ist. Im derzeitigen Stand beträgt er 25 U/min. Bremst das Fahrzeug soweit ab, dass die Räder langsamer als 25 U/min drehen, und gerät gegen ein Hindernis, fällt die Istdrehzahl auf Null ab. Der Differenzwert ist dann kleiner als 25 U/min und wird nicht erkannt, da er sich innerhalb der Toleranz von `EMERGENCY_FILTER` befindet. Die Räder drehen sich in diesem Fall weiter, ohne dass das Hindernis erkannt wird. Jedoch ist die Drehzahl recht gering. Trotzdem kann in einer Vielzahl von Fällen die Plausibilitätsprüfung einen Fehler erkennen und abschalten. Dies bedeutet ein deutliches Plus an Sicherheit.

3.3.7 Sonstiges zur Antriebsteuerung

Zu Beginn des Funktionsblocks der Drehzahlregelung wird der Eingangswert der Solldrehzahl auf die maximal zulässige Drehzahl begrenzt, um eine Überreaktion des Reglers zu vermeiden. Wird z.B. eine Drehzahl von 500 U/min durch eine interne Berechnung des Geschwindigkeitsreglers vorgegeben, würde der Regler ohne die Begrenzung auf diese Drehzahl ausregeln, obwohl das Rad physikalisch viel niedriger begrenzt ist. Damit würde ein falsches Reglerverhalten entstehen, was mit der vorherigen Begrenzung vermieden wird. Der Programmcode entspricht dem der Begrenzung auf die maximale Drehzahl und wird daher nicht noch mal aufgeführt. Der gesamte Programmcode ist auf der Begleit-CD beigefügt.

Einige Tests des Drehzahlreglers haben gezeigt, dass im Stillstand der mobilen Roboterplattform ein ständiges Ansteuern der Motoren stattfindet, um diese auf der gegenwärtigen Position zu halten. Bemerkbar ist dieses Verhalten durch leise stetige Geräusche in den Motoren. Es kann mit PxLogger sichtbar gemacht werden. Da die Motoren eine hohe Positionsauflösung haben, rauscht der Wert von `speed` im Stillstand. Das versucht der Regler auszuregeln und steuert die Motoren leicht an. Um diese Unruhe zu verhindern, wird die Vorgabe an die PWM explizit auf Null gesetzt, wenn die Ausregelung sich nur auf den Stillstand bezieht. Dies wird durch die Überprüfung von Grenzwert und Sollwert erreicht.

Codeausschnitt:

```
//Definiert den Grenzwert
#define OFF_RPM    2

//Befindet sich die Ausregelung im Genzwert und ist die Vorgabe Null,
//wird die Ausregelung ebenfalls auf Null gesetzt
if(((new_rpm < OFF_RPM) && (new_rpm > -OFF_RPM)) &&
     (wish_rpm[id] == 0))
     new_rpm = 0;
```

3.4 Fahrzeugsteuerung

3.4.1 Allgemeines

Die Fahrzeugsteuerung beschreibt eine zweite Abstraktionsebene, die ihre Schnittstelle nach unten nicht zur virtuHall-API, sondern zur Antriebsteuerung hat. Sie bezieht die Istwerte in U/min aus der Antriebsteuerung und gibt Sollwerte in U/min an die Antriebsteuerung zurück. Die Ebene der Fahrzeugsteuerung kann, wie auch die Antriebsteuerung, direkt vom Benutzer in physikalischen Größen angesprochen werden. Neben Geschwindigkeiten können Maximalwerte vorgegeben und ausgelesen werden. Sie ist eine eigenständige Ebene, die auf die Antriebsteuerung aufsetzt.

Grundsätzlich besteht die Fahrzeugsteuerung aus zwei Teilen, der Lineargeschwindigkeitsregelung und der Rotationsgeschwindigkeitsregelung. Beide können gleichermaßen Einfluss auf die Vorgabewerte der unteren Ebene nehmen. Um eine gleichzeitige Linear- und Rotationsbewegung auszuführen, müssen beide Bewegungsarten überlagert werden können. Das ist nur dann ohne großen Aufwand möglich, wenn für Linear- und Rotationsbewegung der gleiche Bezugspunkt im Fahrzeug betrachtet wird. Bei der vorliegenden mobilen Roboterplattform ist das der Mittelpunkt auf der virtuellen Achse zwischen den Antriebsrädern. Um diesen Punkt kann sich das Fahrzeug auf der Stelle drehen und dieser Punkt kann ebenso einer linearen Bewegung folgen.

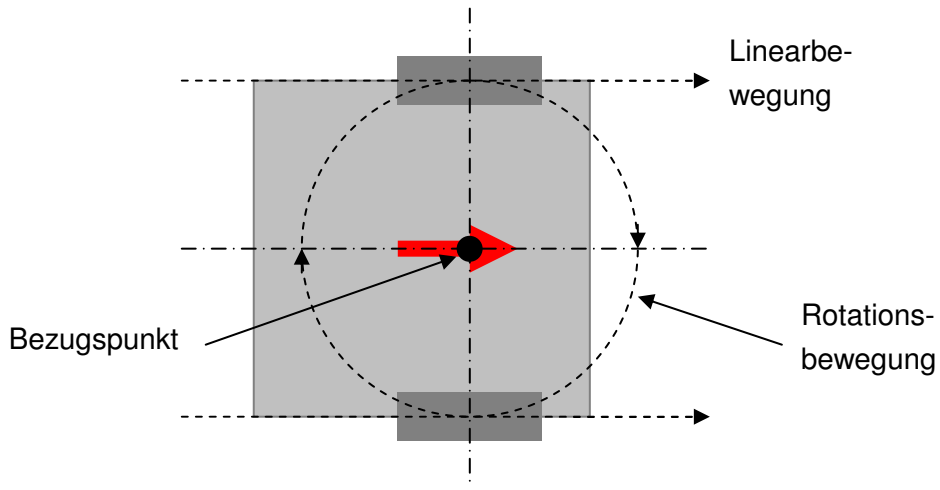


Abbildung 30: Definition des Bezugpunktes

3.4.2 Messwertumwandlung

Geometrisch gesehen handelt es sich um ein System 2. Ordnung, das sich demnach auf einer Ebene bewegen kann. Die relative Position des Fahrzeuges ergibt sich aus der polaren Beschreibung über die Strecke und den Ausrichtungswinkel. Beide Größen können aus dem zurückgelegten Weg beider Räder umgerechnet werden [2]. Dazu werden die aktuelle Drehzahl der Räder, deren Umfang und der Abstand der Räder zueinander benötigt.

RPM : aktuelle Drehzahl eines Rades

$d_{L/R}$: Durchmesser eines Rades

Δt : Zeitabschnitt der Messung (Abtastrate)

$S_{L/R}$: Zurückgelegte Strecke eines Rades

$$\Delta S_{L/R} = \frac{RPM_{L/R} \cdot \pi \cdot d_{L/R}}{\Delta t}$$

Pro Zeitabschnitt legt ein Rad die Strecke ΔS_L bzw. ΔS_R zurück. Dabei ist zu beachten, dass die Umdrehungen pro Minute vorliegen, der Zeitabschnitt in einem Programm aber im Millisekundenbereich liegt. Hier ist eine entsprechende Umrechnung nötig.

Aus den Wegabschnitten $\Delta S_{L/R}$ der einzelnen Räder kann auf die Bewegung des Bezugspunktes geschlossen werden. Lineare Verschiebung und Winkel ergeben sich aus:

θ : Winkel der Ausrichtung

S : Verschiebung des Bezugspunktes

b : Radabstand

$$\Delta S = \frac{\Delta S_R + \Delta S_L}{2}$$

$$\Delta \theta = \frac{\Delta S_R - \Delta S_L}{b}$$

In jedem Zeitabschnitt Δt bewegt sich der Bezugspunkt um die Strecke ΔS und den Winkel $\Delta \theta$. Summiert man diese Werte auf, kann die absolut gefahrene Strecke und der absolute Ausrichtungswinkel berechnet werden. Setzt man die Werte ΔS und $\Delta \theta$ in Bezug zum Zeitabschnitt Δt, erhält man Linear- und Winkelgeschwindigkeit des Fahrzeugs im Bezugspunkt. Aus der Geschwindigkeitsänderung pro Zeitabschnitt kann die Beschleunigung und Verzögerung berechnet werden. Daraus entstehen alle relevanten Istwerte für die Fahrzeugsteuerung und für die Überprüfung der Maximalwerte. Bei Beschleunigung und Verzögerung ist zu beachten, dass eine negative Beschleunigung eine positive Verzögerung bedeutet. Diese Berechnungen sind im internen Funktionsblock `GetMovingValues()` zusammengefasst.

Codeausschnitt der Messewertumwandlung:

```
static int GetMovingValues( int*   vel_int,
                            float* rot_vel_int,
                            float* angle_int,
                            int*   dis_int,
                            long*  acc_int,
                            long*  rot_acc_int)
{
     //Initialisierung der lokalen Variablen
     float delta_S_L = 0;        //zurückgelegter Weg linkes Rad
     float delta_S_R = 0;        //zurückgelegter Weg rechtes Rad
     float delta_S = 0;          //zurückgelegter Weg Bezugspunkt
     float delta_theta = 0;      //Winkel in Radiant

     //Berechnung der Wege des linken und rechten Rades
     delta_S_R = (current_rpm[MOTOR_RIGHT] * PI * PE_TIME_VEL *
       W_DIA_R * (-1)) / 60000;
     delta_S_L = (current_rpm[MOTOR_LEFT] * PI * PE_TIME_VEL *
       W_DIA_L) / 60000;
```

Die Variable `current_rpm` beinhaltet den Istwert der Drehzahl des entsprechenden Rades. Die Durchmesser der Räder werden durch `W_DIA_R` und `W_DIA_L` eingebracht. Genau wie die Konstante `PI` werden die Werte der Raddurchmesser im Header-File `htbintern.h` festgesetzt. Die Zeit `PE_TIME_VEL` bildet mit dem Wert `60000` einen Faktor für die Umrechnung von Umdrehungen pro Minute auf die Zykluszeit der Fahrzeugsteuerung. Die Invertierung beim rechten Motor ist aus in Abschnitt 3.3.1 genannten Gründen nötig.

```
     //Bewegung des Bezugspunktes berechnen
     delta_S = (delta_S_L + delta_S_R)/2;
     delta_theta = (delta_S_L - delta_S_R)/(W_DIS);
     //Umrechnen von Radiant in Grad
     delta_theta = (delta_theta * 360)/(2 * PI);

     //Ausgabeparameter berechnen:

     //Geschwindigkeit und Winkelgeschwindigkeit
     *vel_int = (delta_S * 1000)/PE_TIME_VEL;
     *rot_vel_int = (delta_theta * 1000)/PE_TIME_VEL;
     //Zurückgelegte Strecke und Verdrehwinkel
     *dis_int += delta_S;
     *angle_int += delta_theta;

     //Berechnen der Linear- und Winkelbeschleunigung
     *acc_int = ((*vel_int - vel_old) * 1000)/PE_TIME_VEL;
     *rot_acc_int = ((*rot_vel_int - rot_vel_old) * 1000)/PE_TIME_VEL;
     //Neuen Wert speichern
     vel_old = *vel_int;
     rot_vel_old = *rot_vel_int;
```

Die Beschleunigung entspricht der Ableitung der Geschwindigkeit. Der Programmcode ist ähnlich dem Differentialteil des digitalen Regelalgorithmus. Auch hier ist eine Umrechnung zwischen der Zykluszeit und der physikalischen Einheit nötig.

```
    //Beschleunigung invertieren, wenn die Geschwindigkeit negativ ist
    if(*vel_int < 0)
        *acc_int = -*acc_int;
    if(*rot_vel_int < 0)
        *rot_acc_int = -*rot_acc_int;

    return NO_ERROR;
}
```

Gefahrene Strecke und Winkel ergeben eine polare Beschreibung der Fahrzeugposition in einer Ebene. Indem man die Wegabschnitte in ein kartesisches Koordinatensystem umrechnet und addiert, erhält man die relative Bewegung des Fahrzeugs in x- bzw. y-Richtung. Solche Berechnungen gehören konzeptionell in eine Ebene über der Fahrzeugsteuerung, da es sich bereits um die Positionierung in einer Ebene handelt, und sind nicht in der Fahrzeugsteuerung implementiert.

3.4.3 Entwurf der Linear- und Rotationsgeschwindigkeitsregelung

In Abschnitt 3.2.7 wurde beschrieben, dass es sich bei der Robotersteuerung um eine Kaskadenregelung handelt. Der unterlagerte Regelkreis bildet die Drehzahlregelung. Diese wurde jedoch in Abschnitt 3.3.4 als einschleifiger Regelkreis implementiert, da bei Benutzung der Antriebsteuerung die Geschwindigkeitsregelung nicht benutzt wird und so keine Kaskade zustande kommt. Das ist anders, wenn Drehzahl- und Geschwindigkeitsregler eine Kaskadenstruktur bilden sollen. Damit muss der unterlagerte Drehzahlregler angepasst werden. Als PI-Regler würde er zwar für ein genaues Ausregeln der Drehzahl sorgen, aber die Möglichkeit verwehren, die Geschwindigkeitsregelung als PI-Regler zu gestalten. Reuter und Zacher [21] (S. 117 u. S. 124) schreiben, dass eine Regelung mit zwei Integralanteilen instabil wirkt. Es muss demnach eine Struktur geschaffen werden, die nach Möglichkeit nur einen I-Anteil enthält.

Eine Lösung dafür ist, den unterlagerten Drehzahlregler als P-Regler ohne Ausgleich und den überlagerten Geschwindigkeitsregler als PI-Regler mit Ausgleich zu gestalten. Die Geschwindigkeit wird sich dann auf den Sollwert einstellen, die Drehzahl stellt sich ebenfalls entsprechend dem Sollwert der Geschwindigkeit ein. Dabei kann es durchaus zu einer bleibenden Regelabweichung zwischen Solldrehzahl und Istdrehzahl kommen. Das spielt aber keine Rolle, da es Ziel ist, die Vorgabe der Geschwindigkeit ohne Regelabweichung einzustellen.

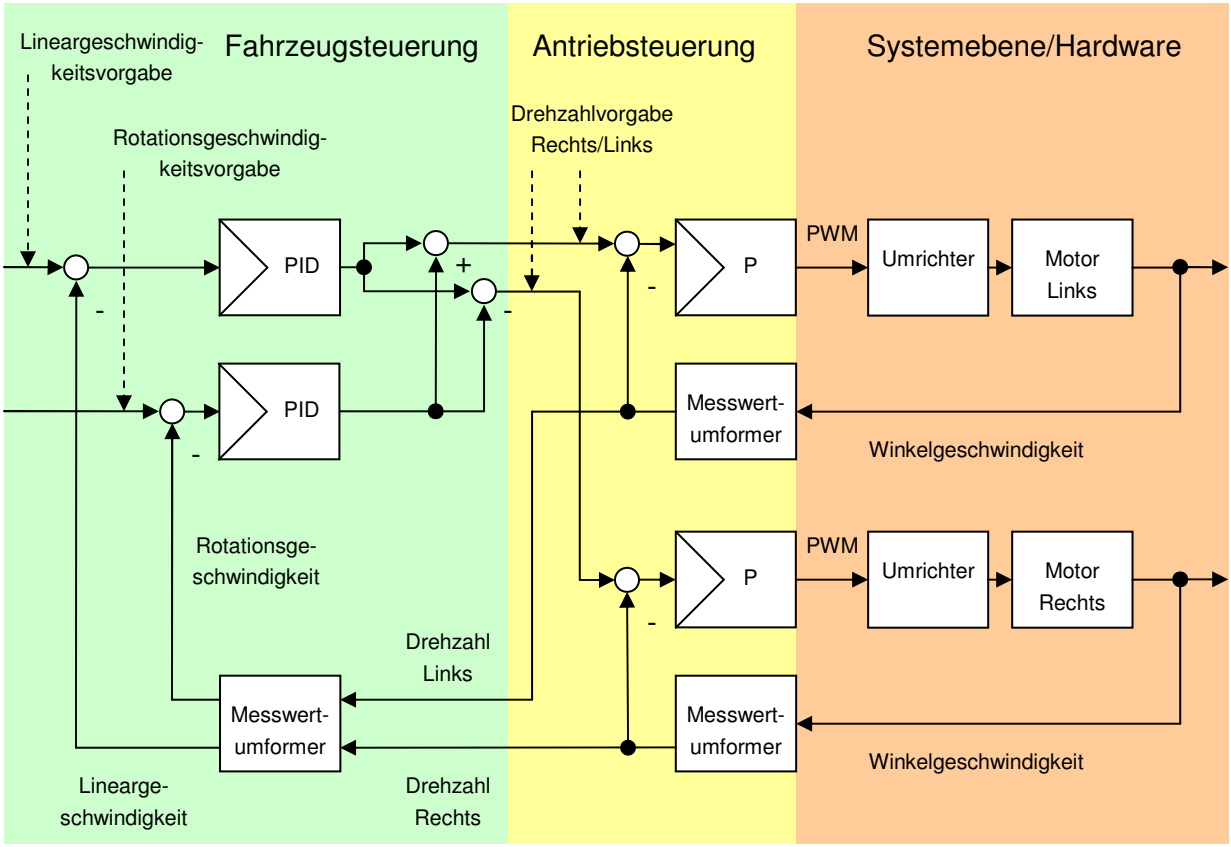


Abbildung 31: Reglerstruktur der Fahrzeugsteuerung

In Abbildung 31 ist deutlich die Überlagerung von Linear- und Rotationsgeschwindigkeit durch Parallelität erkennbar. Während die Stellgröße der Lineargeschwindigkeit für beide Räder positiv ist, addiert sich die Stellgröße der Rotationsgeschwindigkeit positiv bzw. negativ hinzu, um die Drehbewegung des Fahrzeugs zu erwirken. Da eine positive Winkeländerung eine Drehung im Uhrzeigersinn zur Folge haben soll, wird die Vorgabe der Rotationsgeschwindigkeit am rechten Rad subtrahiert und am linken Rad addiert.

3.4.4 Implementierung des Regelalgorithmus

Der unterlagerte Regelkreis der Drehzahlregelung ist bereits in der Ebene der Antriebsteuerung implementiert. Hier wird anstatt eines PI-Reglers ein P-Regler verwendet. Der unterlagerte Kreis wird daher so parametriert, dass I-Anteil und D-Anteil abgeschaltet sind. Das erfolgt mit der Vorgabe des Wertes Null. Die neuen Werte werden den Variablen `kp`, `ki`, und `kd` zu Beginn der Geschwindigkeitsregelung zugeordnet.

Die Regelalgorithmen für Linear- und Rotationsgeschwindigkeitsregler entsprechen dem des Drehzahlreglers. Die Variablen sind mit `vel` (eng. velocity, Geschwindigkeit) und `rot_vel` (eng. rotation velocity, Rotationsgeschwindigkeit) gekennzeichnet. Beide Regler besitzen im Integralteil eine Anti-Wind-Up-Maßnahme und begrenzen sowohl die Sollwerte als auch die Stellgrößen auf die eingestellten Maximalwerte. Der ausführliche Programmcode ist auf der Begleit-CD beigefügt.

3.4.5 Parametrisieren der Regler

Will man eine Kaskadenstruktur parametrisieren, muss zunächst der unterlagerte Regelkreis eingestellt werden. Im vorliegenden Reglerentwurf ist das ein P-Regler. Der Proportionalbeiwert K_p wird anhand der Regelabweichung eingestellt. Die genauen Berechnungen finden sich in Anhang C „Ausführliche Berechnung der Geschwindigkeitsregler“. Das Ergebnis ist:

$$K_p = 1{,}35$$

Damit lässt sich eine Sprungantwort für die Parametrisierung der Lineargeschwindigkeit aufzeichnen.

Abbildung 32: Sprungantwort der Lineargeschwindigkeit

In Abbildung 32 sind Soll- und Istdrehzahl („Wish_RPM“ und „RPM“) eines Rades mit P-Regler zu sehen. Der Proportionalbeiwert des P-Reglers wurde so eingestellt, dass bei einer Ansteuerung mit maximaler Geschwindigkeit die maximale Drehzahl ohne große Regeldifferenz erreicht wird. Das ist nötig, um später den gesamten Drehzahlbereich für die Regelung der Geschwindigkeit ausnutzen zu können.

Zusätzlich ist in Abbildung 32 die Soll- und Ist-Geschwindigkeit als Sprungantwort dargestellt. Die Sprünge des Wertes „Velocity“ beim Anstieg kommen durch die zeitliche Aufteilung der Kaskade zustande. „Velocity“ wird nur alle `PE_TIME_VEL`, also alle acht Millisekunden aktualisiert, während der Wert „RPM“ alle zwei Millisekunden ausgege-

ben wird. Das Signal zeigt in etwa den Verlauf eines Abtast-Haltegliedes. Aus der Sprungantwort kann nun eine mathematische Beschreibung abgeleitet werden, die den unterlagerten Regelkreis und die Regelstrecke zusammenfasst. Die Berechnungen finden sich ebenfalls im Anhang C „Ausführliche Berechnung der Geschwindigkeitsregler“. Die mathematische Beschreibung der Regelstrecke ist:

$$G(s) = \frac{K_s}{(1 + sT_M)^n} = \frac{1}{(1 + s0{,}0052s)^7}$$

Der unterlagerte Abschnitt verhält sich wie ein System 7. Ordnung mit einem Proportionalbeiwert von 1 und einer Zeitkonstante von 0,0052 s. Mit dieser Beschreibung lassen sich die Parameter der Regelung für die Lineargeschwindigkeit ermitteln. Aus den im Anhang C „Ausführliche Berechnung der Geschwindigkeitsregler“ angefügten Berechnungen und Einstellregeln ergeben sich folgende Werte:

Proportionalbeiwert: $K_p = 0{,}45$

Nachstellzeit: $T_n = 0{,}016\,\mathrm{s}$

Vorhaltezeit: $T_v = 0\,\mathrm{s}$

Das Ergebnis kann mit PxLogger aufgezeichnet werden. Dazu wurde der Lineargeschwindigkeitsregelung die Maximalgeschwindigkeit vorgegeben.

Abbildung 33: Ausregelvorgang der Lineargeschwindigkeitsregelung

Obwohl die unterlagerte Drehzahlregelung ohne Ausgleich arbeitet und eine Regeldifferenz bleibt, kann die überlagerte Lineargeschwindigkeitsregelung den Sollwert genau einstellen. Die Anregelzeit beträgt ca. 70 ms. Damit ist die Lineargeschwindigkeitsregelung parametrisiert.

Der überlagerte Regelkreis für die Rotationsgeschwindigkeit arbeitet parallel zur Lineargeschwindigkeitsregelung. Die Rotationsgeschwindigkeitsregelung ist für die Geradeausfahrt der mobilen Roboterplattform verantwortlich. Lautet die Vorgabe Null Grad pro Sekunde, sorgt die Regelung dafür, dass rechtes und linkes Rad exakt die gleiche Anzahl an Inkrementen weiterbewegt werden und das Fahrzeug somit geradeaus fährt. Äußere Einflüsse z.B. Schlupf an den Rädern können auf diese Weise nicht berücksichtigt werden. Dafür sind weitere Programmteile, z.B. eine Schlupfdetektion oder zusätzliche Sensoren, erforderlich. Die Vorgehensweise zur Parametrisierung ist die gleiche wie beim Lineargeschwindigkeitsregler. Eine Sprungantwort des unterlagerten Kreises liefert die gleichen Ergebnisse, da sich die mathematische Beschreibung nicht ändert. Die Ermittlung der Parameter für die Rotationsgeschwindigkeitsregelung finden sich im Anhang C „Ausführliche Berechnung der Geschwindigkeitsregler“. Für die Rotationsgeschwindigkeitsregelung wird folgender Parametersatz verwendet:

Proportionalbeiwert: $K_p = 0{,}40$

Nachstellzeit: $T_n = 0{,}016\,\mathrm{s}$

Vorhaltezeit: $T_v = 0\,\mathrm{s}$

Die Aufzeichnung mit PxLogger zeigt, dass auch die Rotationsregelung ohne Regelabweichung arbeitet. Hier beträgt die Anregelzeit ca. 60 ms.

Abbildung 34: Ausregelvorgang der Rotationsgeschwindigkeitsregelung

3.4.6 Überlagerung von Linear- und Rotationsgeschwindigkeit

Damit die mobile Roboterplattform einen Bogen fahren kann, müssen Linear- und Rotationsgeschwindigkeit gleichzeitig ausgeregelt werden können. Mit der vorliegenden Reglerstruktur ist dies durch die Parallelität der Geschwindigkeitsregler möglich. Wird z.B. das Fahrzeug mit 25 % seiner möglichen Lineargeschwindigkeit positiv und gleichzeitig mit 10 % seiner möglichen Rotationsgeschwindigkeit im Uhrzeigersinn angesteuert, dreht sich das linke Rad mit 35 % und das rechte Rad mit 15 %. Das Fahrzeug fährt einen Rechtsbogen.

Jedoch stößt die Regelung schnell an die physikalischen Grenzen der maximalen Raddrehzahl. Bewegt sich das Fahrzeug mit der Hälfte seiner Geschwindigkeit vorwärts und dreht es gleichzeitig mit der Hälfte seiner Rotationsgeschwindigkeit, steht ein Rad still und das andere dreht sich mit voller Leistung. Die Regelung ist an ihrer Grenze angelangt, da das Fahrzeug weder schneller vorwärts fahren noch schneller drehen könnte. Beide zusätzlichen Bewegungen würden ein Rad mit mehr als 100 % ansteuern. Durch die Begrenzung des Sollwertes in der Regelung tritt dieser Fall nicht ein. Das Übersteuern führt dazu, dass eine Regelung das stillstehende Rad solange anzusteuern versucht, bis der Sollwert erreicht ist. Das erzeugt eine Störgröße in der anderen Regelung, die diese auszugleichen versucht. Es stellt sich ein Mittelwert ein. Keine der beiden Regelungen kann ihren Sollwert erreichen. Linear- und Rotationsgeschwindigkeit verlangsamen sich.

Um einen solchen Fall zu verhindern könnte z.B. eine Plausibilitätsprüfung ähnlich wie bei der Drehzahlregelung implementiert werden, die dann z.B. einen Fehler meldet, oder das Fahrzeug sogar abschaltet. Auch mit der Priorisierung einer Geschwindigkeit kann der Fehler verhindert werden.

3.4.7 Beschleunigungsbegrenzung

Wird einem Regler ein Sollwert vorgegeben, beginnt die Ausregelung auf diesen Wert umgehend. Bei einer Geschwindigkeitsänderung von Null auf die maximale Geschwindigkeit dauert dies ohne Last 70 ms. Fährt das Fahrzeug auf dem Boden, drehen die Räder sofort durch. Daher ist eine Beschleunigungsbegrenzung nötig, die so eingestellt werden kann, dass die Räder beim Anfahren nicht durchdrehen. Das gleiche gilt für das Abbremsen. Die Funktionalität entspricht einer Rampe, die mit physikalischen Werten, der Beschleunigung oder der Verzögerung, eingestellt werden kann. Die Istwerte werden aus der Geschwindigkeitsdifferenz errechnet (Abschnitt 3.4.2 Messwertumwandlung). Messungen mit PxLogger haben gezeigt, dass diese Werte sehr stark schwanken, besonders beim Anfahren und Abbremsen des Fahrzeugs. Eine Regelung auf eine feste Beschleunigung oder Verzögerung ist daher ausgeschlossen. In dieser Arbeit wird eine Beschleunigungsbegrenzung implementiert, die nicht vom Istwert ausgeht, sondern vom Sollwert abhängt. Wird eine Geschwindigkeit als Sollwert

vorgegeben, wird die dadurch verursachte Geschwindigkeitsänderung mit der maximal erlaubten Beschleunigung oder Verzögerung verglichen. Übersteigt die Änderung die Maximalwerte, wird der Sollwert in jedem Schritt solange begrenzt, bis die Differenz nicht mehr zu groß ist.

Dabei ist zu beachten, dass bei positiver und negativer Geschwindigkeit die Beschleunigung und Verzögerung auf den Stillstand bezogen sind, also jeweils umgekehrt sind. Fährt das Fahrzeug mit positiver Geschwindigkeit, bedeutet ein numerisches Abfallen der Geschwindigkeit eine Verzögerung, weil es langsamer fährt. Fährt es mit negativer Geschwindigkeit, bedeutet ein numerisches Abfallen des Wertes eine Beschleunigung, weil das Fahrzeug schneller rückwärts fährt. Diese beiden Fälle müssen unterschieden werden. Zudem bedeutet eine negative Beschleunigung eine positive Verzögerung.

Codeausschnitt Beschleunigungs- und Verzögerungsbegrenzung der Lineargeschwindigkeit:

```
//Festlegen der physikalischen Maximalwerte
#define MAX_ACC    10000     //mm/s²
#define MAX_DEC    10000     //mm/s²

//Initialisierung der modulglobalen Variablen
static int    max_acc     = MAX_ACC;
static int    max_dec     = MAX_DEC;
static int    vel_store   = 0; //Zwischenspeicher für nächsten Schritt

//Periodisch abzuarbeitender Code:

//Abfrage, ob die Geschwindigkeit positiv ist
if(vel >= 0)
{
     //Abfrage, ob die Differenz der Geschwindigkeiten größer als
     //die erlaubte Beschleunigung ist, und Geschwindigkeit begrenzen
     if((velocity - vel_store) > ((max_acc * PE_TIME_VEL)/1000))
          velocity = vel_store + (max_acc * PE_TIME_VEL)/1000;
     //Abfrage, ob die Differenz der Geschwindigkeiten kleiner als
     //die erlaubte Verzögerung ist, und Geschwindigkeit begrenzen
     else if((velocity - vel_store) < ((-max_dec * PE_TIME_VEL)/1000))
          velocity = vel_store - (max_dec * PE_TIME_VEL)/1000;
}
//Bei negativer Geschwindigkeit
else if(vel < 0)
{
     //Abfrage, ob die Differenz der Geschwindigkeiten kleiner als
     //die erlaubte Beschleunigung ist, und Geschwindigkeit begrenzen
     if((velocity - vel_store) < ((-max_acc * PE_TIME_VEL)/1000))
          velocity = vel_store - (max_acc * PE_TIME_VEL)/1000;
     //Abfrage, ob die Differenz der Geschwindigkeiten kleiner als
     //die erlaubte Verzögerung ist, und Geschwindigkeit begrenzen
     else if((velocity - vel_store) > ((max_dec * PE_TIME_VEL)/1000))
          velocity = vel_store + (max_dec * PE_TIME_VEL)/1000;
}

//Den neu eingestellten Wert für den nächsten Schritt speichern
vel_store = velocity;
```

Die physikalische Obergrenze der Beschleunigung und Verzögerung `MAX_ACC` und `MAX_DEC` ist sehr hoch gewählt, um ein schnelles Abbremsen zu ermöglichen, auch wenn dabei die Räder rutschen würden. Eine weitergehende Entwicklung wäre an dieser Stelle Schlupf zu detektieren, um immer nur so stark zu bremsen, dass die Räder sich noch drehen, vergleichbar mit einer Anti-Schlupf-Regelung (ASR) in Kraftfahrzeugen.

Eine Beschleunigungsbegrenzung wurde auch für die Rotationsbeschleunigung implementiert. Um die Auswirkung zu ermitteln, wird ein Anfahrvorgang mit PxLogger aufgezeichnet. Es wird eine Geschwindigkeit von 1000 mm/s vorgegeben und die Beschleunigung auf 1000 mm/s² begrenzt. Rechnerisch sollte das Fahrzeug die gewünschte Geschwindigkeit in einer Sekunde erreichen.

Abbildung 35: Positive Rampe der Geschwindigkeit

Tatsächlich benötigt das Fahrzeug geringfügig länger zum Erreichen der vorgegebenen Geschwindigkeit. Diese Verzögerung kommt durch die Totzeiten des Algorithmus und der Leistungsendstufe zustande. Man erkennt, dass der Signalverlauf von „Velocity“ recht wellig ist. Eine Erklärung hierfür ist, dass die Messung ohne Last aufgenommen wurde. Da für eine Auswertung mit PxLogger eine Verbindung über ein Ethernetkabel nötig ist, konnte die Messung mit Last, also auf dem Untergrund nicht ohne weiteres durchgeführt werden. Lässt man die Welligkeit außer Acht, ist die Rampe eine gleichmäßige Beschleunigung auf die Sollgeschwindigkeit.

3.5 Kalibrierung und Test des Reglers

3.5.1 Kalibrierung

Aufgrund von Produktionsunterschieden und Montageungenauigkeiten können die Parameter Raddurchmesser und Radabstand bei jedem Bausatz unterschiedlich sein. Das führt dann zu Problemen, wenn ein Sollwert genau erreicht werden soll, z.B. eine genaue Geschwindigkeit oder Strecke. Faktoren, die den Raddurchmesser und den Radabstand beeinflussen, können durch eine Kalibrierung größtenteils ausgeglichen werden.

Zum Kalibrieren muss man die tatsächlich zurückgelegte Strecke und den Winkel mit den berechneten Werten vergleichen und gegebenenfalls mit korrigierten Werten angleichen. Um die tatsächlich zurückgelegte Strecke zu messen, wurde ein Steuerprogramm geschrieben, dass dem Roboter eine geringe Geschwindigkeit vorgibt, solange er einen Punkt, z.B. 2000 mm, noch nicht erreicht hat. An diesem Punkt stoppt der Roboter und man kann den Abstand zwischen Start- und Stopppunkt messen. Besteht eine Differenz zwischen den Werten, kann der Durchmesser der Räder `W_DIA_L` und `W_DIA_R` in dem Header-File `htbintern.h` angepasst werden. Die Drehbewegung kann man in ähnlicher Weise kalibrieren. Besteht eine Differenz zwischen tatsächlichem und berechnetem Winkel, muss der Wert für den Radabstand `W_DIS` angepasst werden. Die Genauigkeit, mit der der Roboter navigiert, kann bis auf wenige Millimeter genau kalibriert werden.

Der Ablauf der Kalibrierung kann mithilfe eines vorgegebenen Umfeldes und weiterer Sensoren automatisiert werden. Z.B. könnten zusätzliche Sensoren Markierungen auf dem Boden, die eine bekannte Entfernung zueinander haben, detektieren, mit der gemessenen Entfernung am Rad abgleichen und ggf. Werte korrigieren.

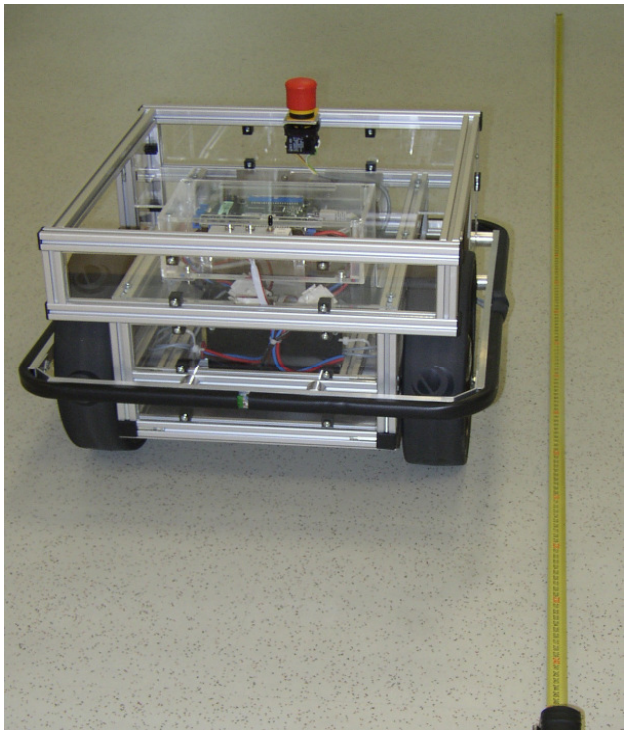


Abbildung 36: Der HighTecBot bei einer Kalibrierungsfahrt

3.5.2 Fahrt auf einer Schräge

Um zu testen, wie die Regler auf Störgrößen reagieren, muss der Roboter wechselnden Lasten ausgesetzt sein. Das ist dann der Fall, wenn er z.B. eine Schräge hinauf oder hinunter fährt. Ein Testprogramm wurde so geschrieben, dass dem Roboter eine gleichmäßige Geschwindigkeit von 1000 mm/s vorgegeben wird. Fährt er von einer ebenen Fläche aus eine Schräge hoch, sollte er diese Geschwindigkeit beibehalten.

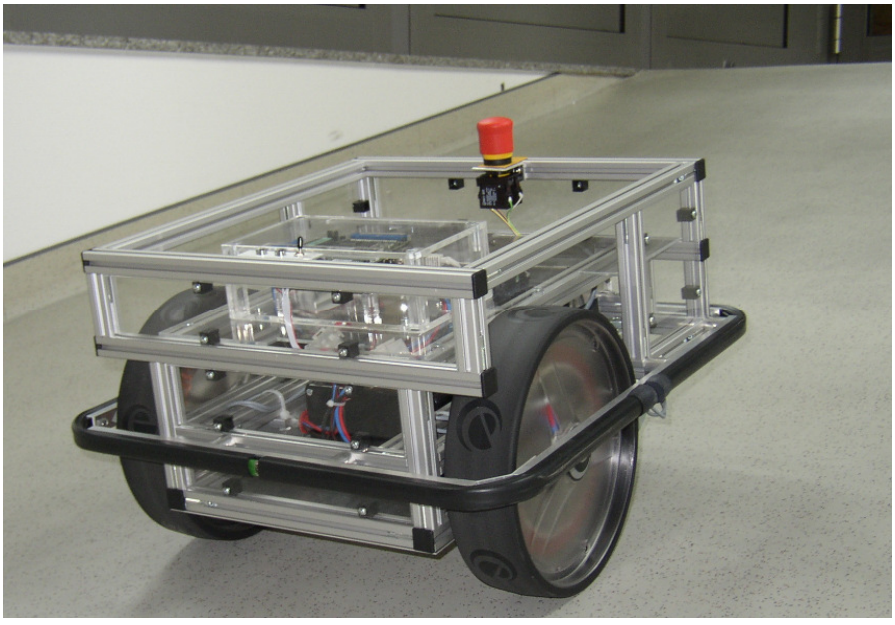


Abbildung 37: Fahrt auf einer Schräge

Mit dem Programm PxLogger wurde während dieser Fahrt Soll- und Istgeschwindigkeit sowie Soll- und Istdrehzahl des linken Rades aufgenommen.

Abbildung 38: Reglerverhalten bei Lastwechsel

Bei der linken roten Linie beginnt der Roboter die Fahrt und beschleunigt mit 1000 mm/s² auf die Sollgeschwindigkeit von 1000 mm/s. Zur Verdeutlichung ist die Beschleunigungsrampe in der Grafik mit einer schwarzen Linie markiert. Der Roboter erreicht die Sollgeschwindigkeit und der Regler sorgt für dessen Einhaltung. Bei der zweiten roten Linie erreicht das Fahrzeug die Schräge. Es entsteht eine Abweichung der Geschwindigkeit. Um die gleiche Geschwindigkeit zu behalten, muss eine stärkere PWM-Ansteuerung erfolgen. Das ist an dem roten Signalverlauf „Wish RPM“ zu erkennen. Die tatsächliche Drehzahl und damit die Fahrzeuggeschwindigkeit bleibt gleich und kann beibehalten werden. Der Regler ist in der Lage, auf Lastwechsel zu reagieren und Störgrößen auszugleichen.

4 Zusammenfassung und Ausblick

In dieser Arbeit wurden die Software und die Hardware einer mobilen Roboterplattform soweit an ein vorgegebenes eingebettetes System und Echtzeitbetriebssystem angepasst, dass eine flexible und einfache Verwendung der Plattform möglich wird. Die vor Beginn der Arbeit gestellten Anforderungen sind größtenteils umgesetzt worden.

Um ein Steuerprogramm ohne Eingriff in den Programmcode zu starten und seinen Betriebszustand zu sehen, ist ein Kontrollfeld mit einem Schalter und zwei Leuchtdioden angebracht worden. Passend zum eingebetteten System wurde eine Konverterplatine entwickelt, die es erlaubt, das Kontrollfeld, Schaltleisten zur Kollisionserkennung und analoge Sensoren mit Einheitssignal anzuschließen. Die Konverterplatine passt die Eingangs- und Ausgangssignale an die Schnittstelle zum eingebetteten System an. Die Anforderung, ein Programmstartschalter und weitere Sensoren für eine flexible Verwendung der mobilen Roboterplattform anschließen zu können, wurde erfüllt.

Für die Ansteuerung der beiden Motoren wurde ein Steuer- und Regelungskonzept entwickelt und umgesetzt. Es zielt auf die Bereitstellung einer Schnittstelle zur Anwendungsprogrammierung (API) ab, mit der die mobile Roboterplattform mithilfe von Vorgaben in physikalischen Größen über Schnittstellenfunktionen angesteuert werden kann. Neben Vorgaben der Sollwerte kann ein Benutzer auch Parameter in physikalischen Größen (Istwerte) abfragen, um jederzeit den aktuellen Zustand der mobilen Roboterplattform zu erfassen. Damit sind die Anforderungen, Schnittstellenfunktionen als API zur Verfügung zu stellen, die Vorgabe der Bewegungen anhand physikalischer Größen zu geben und Istwerte abfragen zu können, erfüllt.

Das Steuerungskonzept sieht zwei Ebenen vor, die aufeinander aufbauen und getrennt voneinander betrieben werden können. In der Antriebsteuerung können die Motoren einzeln prozentual oder mit einer Drehzahlvorgabe bewegt werden. Damit kann der Roboter bereits gesteuert werden. Für das Erreichen der Sollwerte sorgt eine PI-Regelung. Auf die Antriebsteuerung baut die Fahrzeugsteuerung auf. In dieser Ebene wird die mobile Roboterplattform als Fahrzeug betrachtet. Das Fahrzeug kann sich mit einer vorgegebenen Lineargeschwindigkeit vorwärts und rückwärts bewegen und mit einer Rotationsgeschwindigkeit drehen. Beide Geschwindigkeiten lassen sich überlagern, um z.B. einen Bogen zu fahren. Auch hier wird das Erreichen der Sollwerte durch jeweils einen PI-Regler sichergestellt. Durch die Regelung der Rotationsgeschwindigkeit kann die Anforderung der Geradeausfahrt nur bedingt erfüllt werden, da eine Kalibrierung nötig ist und äußere Einflusse bisher unberücksichtigt bleiben. Die Anforderung, dass Maximalwerte einstellbar sein sollen, ist erfüllt, da die Begrenzung aller Drehzahlen, Geschwindigkeiten und Beschleunigungen mit benutzerdefinierten Werten über Schnittstellenfunktionen möglich ist. Damit können verschiedene Programme oder Roboterplattformen betrieben und verschiedene Betriebszustände realisiert werden.

Zuletzt wurde die mobile Roboterplattform manuell kalibriert und anschließend an einer Schräge getestet. Dabei konnte festgestellt werden, dass die Regler wie gefordert in der Lage sind, auf Lastwechsel zu reagieren und die vorgegebenen physikalischen Größen einzuhalten.

Der Programmcode wurde als eigenständiger nachladbarer Task für das Echtzeitbetriebssystem PXROS-HR geschrieben. Damit ist die entwickelte Schnittstelle (API) ein Modul, das sowohl als nachladbarer Task, als auch als Teil des Basissystems nutzbar ist. Das erfüllt die Anforderung, dass die Robotersteuerung als Modul in das Echtzeitbetriebssystem eingebunden werden kann.

Mit der vorliegenden Arbeit ist es gelungen, eine funktionsfähige und modulare Schnittstelle für ein vorgegebenes eingebettetes System und Echtzeitbetriebsystem zu entwickeln und die Anforderungen größtenteils zu erfüllen. Das eingebettete System wurde durch eine zusätzliche Elektronik so erweitert, dass es deutlich besser für den Einsatz in der Ausbildungsrobotik geeignet ist. An einigen Stellen in der Arbeit wurden bessere Lösungen gefunden als ursprünglich geplant. Im folgenden Ausblick wird auf weitere Verbesserungsmöglichkeiten und den zukünftigen Einsatz eingegangen.

Die für diese Arbeit berechneten und verwendeten Reglerparameter sind in genau einem Punkt optimal. Ändern sich die Bedingungen, stellen sie allenfalls eine Annäherung dar. In diesem Zusammenhang könnte der Regler so gestaltet werden, dass er adaptive oder lernende Algorithmen verwenden und als Zustandsregler erweitert werden kann. Ein Zustandsregler erhöht zwar den Rechenaufwand, bietet aber im Zusammenhang mit adaptiven und lernenden Algorithmen die Möglichkeit, für jeden Betriebspunkt der mobilen Roboterplattform selbstständig nahezu optimale Reglerparameter zu finden und zu verwenden.

Fehlerzustände in einer Software führen zu falschen Berechnungen oder zum Ausfall bis hin zum Stoppen des Programms. Um dem frühzeitig entgegenzuwirken, könnten weitere Plausibilitätsprüfungen implementiert werden. Die vorhandene Plausibilitätsprüfung der Drehzahlregelung lässt sich z.B. für den Bremsfall erweitern. Bei der Überlagerung von Linear- und Rotationsgeschwindigkeit ist ebenfalls eine Prüfung möglich, um zu erkennen, ob der Regler an seinen Grenzen angelangt ist. Eine Priorisierung einer der beiden Geschwindigkeiten vermeidet die Fehlfunktion der Reglerstruktur.

Navigiert die mobile Roboterplattform nur über die Positionserfassung der Räder, kommt es bei schlechter Bodenhaftung zu erheblichen Fehlern. Mit einer elektronischen Schlupfkontrolle, ähnlich wie beim Automobil (ASR, Anti-Schlupf-Regelung), kann ein solcher Fehler minimiert werden.

Da die Bereitstellung der Robotersteuerung-API nur ein Schritt in der Entwicklung des HighTecBot darstellt, sind weitere Entwicklungen und Verbesserungen denkbar und notwendig. Mit leichteren Akkus könnte deutlich Gewicht eingespart werden. Die derzeit verwendete Leistungsendstufe für die beiden Motoren wird von der Firma HighTec EDV-Systeme GmbH weiterentwickelt. Die nächste Version soll durch Verwendung anderer Leistungstransistoren die Effizienz der Ansteuerung der Motoren und damit ihre Leistung steigern und die Totzeit deutlich verringern. Mit daran angepassten Reglerparametern könnte sich die mobile Roboterplattform noch wesentlich agiler verhalten.

Die Programmstruktur der Robotersteuerung besteht konzeptionell aus zwei Ebenen. Der Quellcode ist in einem Hauptmodul implementiert, das zudem die Daten intern über modulglobale Variablen austauscht. Die Strukturierung und der Datenaustausch könnten durch Objektorientierte Programmierung übersichtlicher gestaltet werden.

HighTecBot ist zurzeit ein Prototyp, der für den Einsatz in der Hochschulausbildung weiterentwickelt werden soll. Dafür ist es sinnvoll, die Schnittstellen am eingebetteten System zur Kommunikation mit anderen Geräten zu nutzen. Ein Steuerprogramm auf einem solchen Gerät kann auf die in dieser Arbeit geschaffene Schnittstelle zugreifen. Denkbar sind Feldbussysteme aus der Industrie, LAN- und WLAN-Verbindungen. Eine graphische Oberfläche könnte das Eingeben der Steuerbefehle erleichtern und gleichzeitig Istwerte ausgeben. Die Einsatzgebiete sind aufgrund der standardisierten Schnittstelle vielfältig.

Literaturverzeichnis

[1] Beierlein, Thomas; Hagenbruch, Olaf: Taschenbuch Mikroprozessortechnik. 3. Auflage, Carl Hanser Verlag, München, Wien 2004

[2] Bertram, Torsten; Odometrie und Probabilistisches Bewegungsmodell. Vorlesungsskript SS 2007, Universität Dortmund 2007

[3] Bunzemeier, Andreas: Regelungstechnik, der geschlossene Kreis. Vorlesungsskript SS 2007, FH Bonn-Rhein-Sieg, Sankt Augustin 2007

[4] CadSoft Computer GmbH: Eagle. Version 4.16r1. www.cadsoft.de - zugegriffen am 06/2005

[5] Caspi Paul, Maler Oded: From Control Loops to Real-Time Programs. In: Handbook of Networked and Embedded Control Systems, S. 395 - 418. Birkhäuser Verlag, Boston 2005

[6] Code::Blocks: The open source, cross platform, free C++ IDE. www.codeblocks.org - zugegriffen am 28.07.2008

[7] Dörrscheidt, Frank; Latzel, Wolfgang: Grundlagen der Regelungstechnik. B.G.Teubner, Stuttgart 1993

[8] Elmos Semiconductor AG: virtuHall. www.elmos.de/produkte/know-how/virtuhallR.html - zugegriffen am 31.07.2008

[9] Fuest, Klaus; Döring, Peter: Elektrische Maschinen und Antriebe. Vieweg-Verlag, Wiesbaden 2004

[10] Große, Norbert; Schorn, Wolfgang: Taschenbuch der praktischen Regelungstechnik. Carl Hanser Verlag, München, Wien 2006

[11] HighTec EDV-Systeme GmbH: PXROS Quickstart. Version 1.2, internes Papier

[12] HighTec EDV-Systeme GmbH: PXROS User's Guide. Version 1.2, internes Papier

[13] Hittinger, Stefan: Technical Discription TC-Control 2, V1.0.1. Firma HighTec EDV-Systeme GmbH, Saarbrücken, 12.03.2008, internes Papier

[14] Infineon Technologies AG: Data Sheet, V1.0, TC 1796, 32-Bit Single-Chip Microcontroller TriCore. Infineon Technologies AG, München 2008

[15] Latzel, Wolfgang: Einstellregeln für vorgegebene Überschwingweiten. In: at-Automatisierungstechnik, 41 (1993) Heft 4, S. 103-113

[16] Lehser, Martina: Roboter-Baukastensystem für die Ausbildungs-Robotik an Hochschulen. Produktblatt, Stand 21.02.2008

[17] Linear Technology: LT1121-3.3 Micropower Low Dropout Regulators. www.linear.com/pc/productDetail.jsp?navID=H0,C3,P1369#descriptionSection. - zugegriffen am 05.04.2008

[18] Lutz, Holger; Wendt Wolfgang: Taschenbuch der Regelungstechnik. 7. Auflage, Harri Deutsch Verlag, Frankfurt 2007

[19] Marwedel, Peter: Eingebettete Systeme. Springer-Verlag, Berlin, Heidelberg 2007

[20] Osqoop: The Open Oscillscope. http://lsn.unige.ch/osqoop/ - zugegriffen am 24.07.2008

[21] Reuter, Manfred; Zacher, Serge: Regelungstechnik für Ingenieure. Vieweg-Verlag, Wiesbaden 2004

[22] Roboternetz: Digitaler Regler. www.roboternetz.de/wissen/index.php/Regelungstechnik. - zugegriffen am 17.06.2008

[23] Stölting, Hans-Dieter; Kallenbach, Eberhard: Handbuch Elektrische Kleinantriebe. 3. Auflage, Carl Hanser Verlag, München, Wien 2006

[24] Strothmann, Rolf: Motoren als Aktoren und Sensoren. http://www.elektroniknet.de/fileadmin/freigaben/termine/kfz2007/praesentationen/Strothmann.pdf - zugegriffen am 20 07.2008

[25] Tröster Fritz: Steuerungs- und Regelungstechnik für Ingenieure. Oldenbourg Wissenschaftsverlag GmbH, München 2005

[26] Van Heesch, Dimitri: doxygen, Source code documentation generator tool. www.stack.nl/~dimitri/doxygen - zugegriffen am 30.07.2008

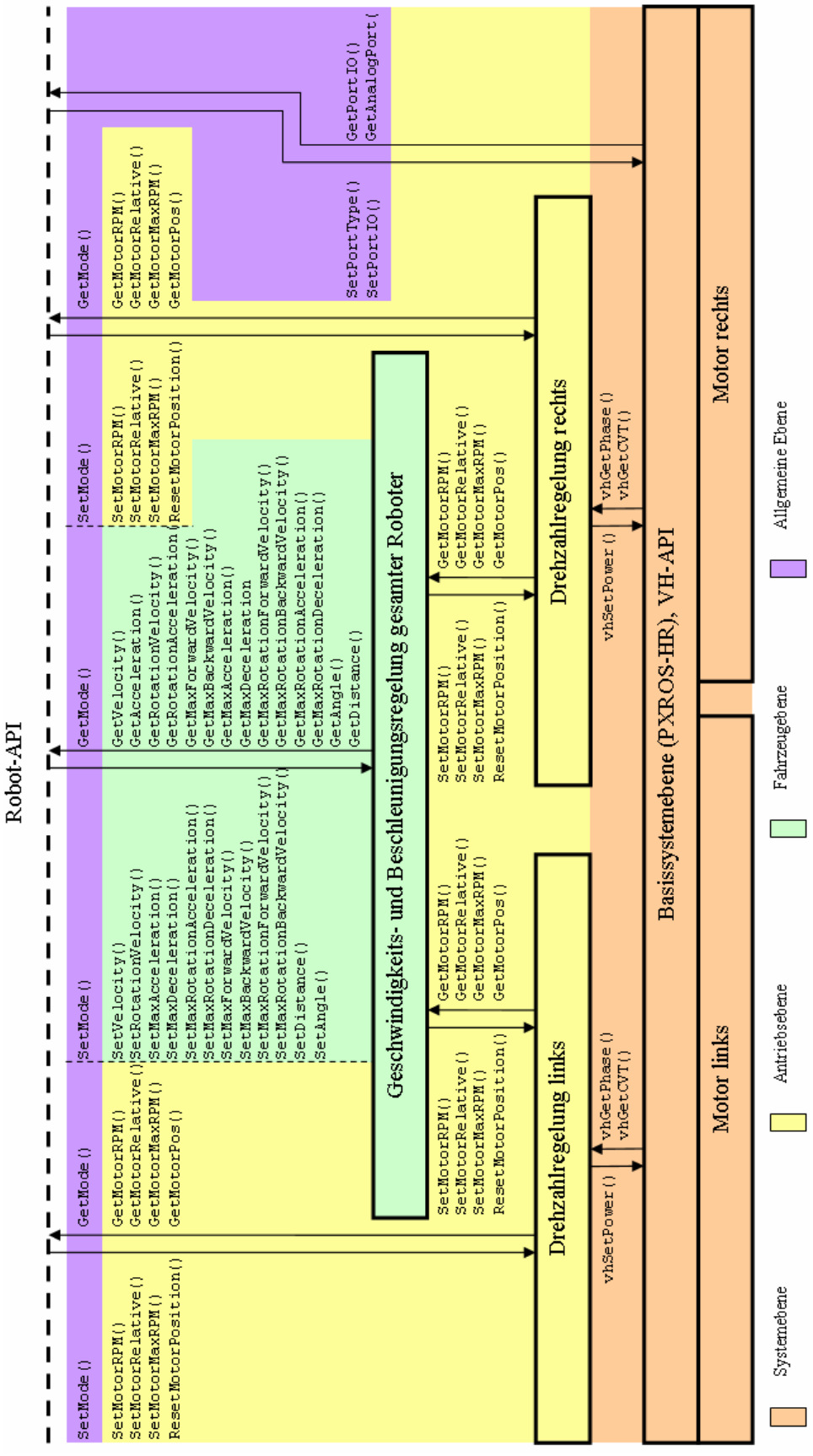
Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum

Unterschrift

Anhang A Übersicht der Schnittstellenfunktionen



A.1 Schnittstellenfunktionen der Antriebsteuerung

Funktion:	`int htbSetMotorRPM(unsigned short id, int wish_rpm)`
Beschreibung:	Die Umdrehungsgeschwindigkeit eines Motors kann in Umdrehungen pro Minute gesetzt werden.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
wish_rpm:	Die gewünschte Umdrehungsgeschwindigkeit in Umdrehungen pro Minute. Die Vorgabe wird durch eine Regelung erreicht. Ist der Eingabewert höher als die physikalisch maximal erreichbare Drehzahl, wird ein Fehler ausgegeben.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Mit `MOTOR_1` und `MOTOR_2` wird keine Invertierung vorgenommen, um z.B. bei zwei positiven Werten ein Fahrzeug geradeaus zu bewegen. Das Fahrzeug würde sich auf der Stelle drehen. Mit `MOTOR_RIGHT` und `MOTOR_LEFT` wird automatisch ein Motor invertiert, so dass das Roboterfahrzeug vorwärts bzw. rückwärts fährt. Die Motornummern werden als `#define` im Header-File `htb.h` definiert.

Funktion:	`int htbSetMotorRelative(unsigned short id, int relative_speed)`
Beschreibung:	Die Umdrehungsgeschwindigkeit eines Motors wird relativ zur Maximalumdrehungsgeschwindigkeit gesetzt.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
relative_speed:	Die gewünschte relative Umdrehungsgeschwindigkeit. Gültige Werte: Ganzzahlen zwischen -100 und +100
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Wird die Maximalgeschwindigkeit durch `SetMotorMaxRPM()` heruntergesetzt, ändert sich auch die relative Abbildung der Umdrehungsgeschwindigkeit auf den neuen Wertebereich. Ein Wert von 100 entspricht immer der maximalen Umdrehungsgeschwindigkeit.

Funktion:	`int htbSetMotorMaxRPM(unsigned short id, int max_rpm)`
Beschreibung:	Hiermit kann die aktuelle maximale Umdrehungsgeschwindigkeit eines Motors angepasst werden. Eine höhere Umdrehungsgeschwindigkeit als die physikalisch erreichbare ist nicht möglich.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
max_rpm:	Die neue maximale Umdrehungsgeschwindigkeit.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der eingegebene Wert größer als die physikalisch maximale Umdrehungsgeschwindigkeit, wird ein Fehler ausgegeben.

Funktion:	`int htbGetMotorRPM(unsigned short id, int* rpm)`
Beschreibung:	Die aktuelle Umdrehungsgeschwindigkeit eines Motors kann in Umdrehungen pro Minute abgefragt werden.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
rpm:	Die aktuellen Umdrehungen pro Minute.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMotorRelative(unsigned short id, int* relative_speed)`
Beschreibung:	Die aktuelle relative Umdrehungsgeschwindigkeit eines Motors wird abgefragt.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
relative_speed:	Die aktuelle relative Umdrehungsgeschwindigkeit als ein Wert zwischen -100 und +100.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMotorMaxRPM(unsigned short id, int max_rpm)`
Beschreibung:	Die aktuelle maximale Umdrehungsgeschwindigkeit eines Motors wird abgefragt.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
max_rpm:	Die aktuelle maximale Umdrehungsgeschwindigkeit des Motors
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMotorPosition(unsigned short id, int* position)`
Beschreibung:	Nach dem Einschalten des Systems ist die Position Null. Der Wert `position` wird je nach Fahrtrichtung inkrementiert oder dekrementiert.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
position:	Die Position des Motors.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbResetMotorPosition(unsigned short id)`
Beschreibung:	Die Motorposition kann zurückgesetzt werden. Die aktuelle Motorposition wird als Position Null angenommen.
id:	Die Motornummer als: `MOTOR_1, MOTOR_2, …, MOTOR_X, MOTOR_RIGHT, MOTOR_LEFT`
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

A.2 Schnittstellenfunktionen der Fahrzeugsteuerung

Funktion:	`int htbSetVelocity(int velocity)`
Beschreibung:	Die Geschwindigkeit des Fahrzeugs kann eingestellt werden.
velocity:	Die Geschwindigkeit des Fahrzeugs in mm/s. Positive Werte für vorwärts, negative Werte für rückwärts.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert größer als die aktuelle maximale Geschwindigkeit, wird ein Fehler ausgegeben. Zum Beschleunigen wird die aktuell eingestellte maximale Beschleunigung verwendet.

Funktion:	`int htbSetRotationVelocity(int rotation_velocity)`
Beschreibung:	Die Winkelgeschwindigkeit des Fahrzeugs kann eingestellt werden.
rotation_velocity:	Die Winkelgeschwindigkeit des Fahrzeugs in Grad/s. Positive Werte für „im Uhrzeigersinn“, negative Werte für „gegen den Uhrzeigersinn“.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert größer als die aktuelle maximale Winkelgeschwindigkeit, wird ein Fehler ausgegeben. Zum Beschleunigen wird die aktuell eingestellte maximale Winkelbeschleunigung verwendet.

Funktion:	`int htbSetMaxAcceleration(int acceleration)`
Beschreibung:	Setzt die maximale Beschleunigung. Diese kann nicht höher als die physikalisch maximale mögliche Beschleunigung gewählt werden.
acceleration:	Die Beschleunigung in mm/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert höher als die physikalische maximal mögliche Beschleunigung, wird ein Fehler zurückgegeben.

Funktion:	`int htbSetMaxDeceleration(int deceleration)`
Beschreibung:	Setzt die maximale Verzögerung. Diese kann nicht höher als die physikalisch maximal mögliche Verzögerung gewählt werden.
deceleration:	Die Verzögerung in mm/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert höher als die physikalische maximal mögliche Verzögerung, wird ein Fehler zurückgegeben.

Funktion:	`int htbSetMaxRotationAcceleration(int acceleration)`
Beschreibung:	Setzt die maximale Winkelbeschleunigung. Diese kann nicht höher als die physikalisch maximal mögliche Winkelbeschleunigung gewählt werden.
acceleration:	Die Winkelbeschleunigung in Grad/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert höher als die physikalische maximal mögliche Winkelbeschleunigung, wird ein Fehler zurückgegeben.

Funktion:	`int htbSetMaxRotationDeceleration(int deceleration)`
Beschreibung:	Setzt die maximale Winkelverzögerung. Diese kann nicht höher als die physikalisch maximal mögliche Winkelverzögerung gewählt werden.
deceleration:	Die Winkelverzögerung in Grad/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Ist der Eingabewert höher als die physikalische maximal mögliche Winkelverzögerung, wird ein Fehler zurückgegeben.

Funktion:	`int htbSetMaxForwardVelocity(int f_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Geschwindigkeit vorwärts wird gesetzt.
f_velocity:	Die aktuelle maximal mögliche Geschwindigkeit vorwärts in mm/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbSetMaxBackwardVelocity(int b_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Geschwindigkeit rückwärts wird gesetzt.
b_velocity:	Die aktuelle maximal mögliche Geschwindigkeit rückwärts in mm/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbSetMaxRotationForwardVelocity(int f_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Winkelgeschwindigkeit im Uhrzeigersinn wird gesetzt.
f_velocity:	Die aktuelle maximal mögliche Winkelgeschwindigkeit in Grad/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbSetMaxRotationBackwardVelocity(int b_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Winkelgeschwindigkeit gegen den Uhrzeigersinn wird gesetzt.
b_velocity:	Die aktuelle maximal mögliche Winkelgeschwindigkeit in Grad/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetVelocity(int* velocity)`
Beschreibung:	Die Geschwindigkeit des Fahrzeugs kann abgefragt werden.
velocity:	Die Geschwindigkeit des Fahrzeugs in mm/s. Positive Werte für vorwärts, negative Werte für rückwärts.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetRotationVelocity(int* rotation_velocity)`
Beschreibung:	Die Winkelgeschwindigkeit des Fahrzeugs kann abgefragt werden.
rotation_velocity:	Die Winkelgeschwindigkeit des Fahrzeugs in Grad/s. Positive Werte für „im Uhrzeigersinn“, negative Werte für „gegen den Uhrzeigersinn“.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetAcceleration(int* acceleration)`
Beschreibung:	Die aktuelle Beschleunigung des Fahrzeugs kann abgefragt werden.
acceleration:	Die Beschleunigung des Fahrzeugs in mm/s². Positive Werte für Beschleunigung, negative Werte für Verzögerung.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetRotationAcceleration(int* acceleration)`
Beschreibung:	Die aktuelle Winkelbeschleunigung des Fahrzeugs kann abgefragt werden.
acceleration:	Die Winkelbeschleunigung des Fahrzeugs in Grad/s². Positive Werte für Beschleunigung, negative Werte für Verzögerung.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxAcceleration(int* acceleration)`
Beschreibung:	Die eingestellte maximal mögliche Beschleunigung wird abgefragt.
acceleration:	Die Beschleunigung in mm/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxDeceleration(int* deceleration)`
Beschreibung:	Die eingestellte maximal mögliche Verzögerung wird abgefragt.
deceleration:	Die Verzögerung in mm/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxRotationAcceleration(int* acceleration)`
Beschreibung:	Die eingestellte maximal mögliche Winkelbeschleunigung wird abgefragt.
acceleration:	Die Winkelbeschleunigung in Grad/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxRotationDeceleration(int* deceleration)`
Beschreibung:	Die eingestellte maximal mögliche Winkelverzögerung wird abgefragt.
deceleration:	Die Winkelverzögerung in Grad/s². Nur positive Werte möglich.
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxForwardVelocity(int* f_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Geschwindigkeit vorwärts wird ausgegeben.
f_velocity:	Die aktuelle maximal mögliche Geschwindigkeit vorwärts in mm/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxBackwardVelocity(int* b_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Geschwindigkeit rückwärts wird ausgegeben.
b_velocity:	Die aktuelle maximal mögliche Geschwindigkeit rückwärts in mm/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxRotationForwardVelocity(int* f_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Winkelgeschwindigkeit im Uhrzeigersinn wird ausgegeben.
f_velocity:	Die aktuelle maximal mögliche Winkelgeschwindigkeit in Grad/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetMaxRotationBackwardVelocity(int* b_velocity)`
Beschreibung:	Die aktuelle maximal mögliche Winkelgeschwindigkeit gegen den Uhrzeigersinn wird ausgegeben.
b_velocity:	Die aktuelle maximal mögliche Winkelgeschwindigkeit in Grad/s
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int GetDistance(int* distance)`
Beschreibung:	Gibt die gefahrene Entfernung vom Startpunkt aus zurück.
distance:	Die Entfernung in mm
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Funktion:	`int htbGetAngle(int* angle)`
Beschreibung:	Der aktuelle Verdrehwinkel des Roboters kann abgefragt werden. Nach dem Einschalten ist der Wert Null und verändert sich, wenn das Fahrzeug sich dreht oder eine Kurve fährt.
angle:	Der Verdrehwinkel in Grad
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

A.3 Allgemeine Schnittstellenfunktionen

Funktion:	`int SetMode(short mode)`
Beschreibung:	Der Benutzungsmodus kann verändert werden. Im Antriebsmodus sind die Motoren einzeln bedienbar und die Funktionen des Fahrzeugmodus gesperrt. Im Fahrzeugmodus sind die Funktionen des Antriebsmodus gesperrt. Im Entwicklungsmodus sind beide Modi möglich.
mode:	Gültige Modi sind: `VEHICLE`, `SINGLE_MOTOR`, `DEVELOPEMENT`, `OFF`
Rückgabewert:	`NO_ERROR` bei OK, Fehlercode bei Fehler
Besonderheiten:	Im Entwicklungsmodus ist darauf zu achten, dass sowohl Funktionen des Antriebsmodus als auch Funktionen des Fahrzeugmodus die Motoren gleichzeitig ansprechen können. Dies kann zu Fehlern führen.

Funktion:	`int GetMode(short mode)`
Beschreibung:	Der aktuell aktive Benutzungsmodus wird abgefragt.
mode:	Der aktuell aktive Benutzungsmodus. Möglich ist: `VEHICLE`, `SINGLE_MOTOR`, `DEVELOPEMENT`, `OFF`
Rückgabewert:	`NO_ERROR`
Besonderheiten:	keine

Anhang B Ausführliche Berechnung des Drehzahlreglers

B.1 Mathematische Beschreibung der Strecke

Mithilfe des Verfahrens der Zeitprozentkennwerte kann eine Sprungantwort der Strecke so analysiert werden, dass ein mathematisches Modell daraus hergeleitet werden kann. Aus der Analyse in Abschnitt 3.3.3 geht hervor, dass es sich um ein System höherer Ordnung mit Ausgleich handelt. Danach richtet sich die weitere Vorgehensweise. Voraussetzung ist, dass Soll- und Istwert bekannt sind und es eine Zeitskalierung gibt. PxLogger zeichnet die Werte graphisch über die Zeit auf und loggt sie gleichzeitig in einer Datei mit, so dass alle Werte bekannt sind.

Abbildung 39: Auswertung der Sprungantwort der Drehzahl

Aus der Sprungantwort werden folgende Werte abgelesen:

$\Delta x = 59{,}8\%$ Differenz des Istwertes (55 U/min)

$\Delta y = 50\%$ Differenz des Sollwertes (46 U/min)

$t_{10} = 15ms$ Zeit, nach der der Istwert auf 10 % angestiegen ist

$t_{50} = 40ms$ Zeit, nach der der Istwert auf 50 % angestiegen ist

$t_{90} = 72ms$ Zeit, nach der der Istwert auf 90 % angestiegen ist

Formal wird eine Strecke höherer Ordnung wie folgt beschrieben [3] (S. 128):

$$x(t) = K_s \cdot \Delta y \left[1 - e^{\frac{t}{T_M}} \cdot \sum_{v=0}^{n-1} \frac{1}{v!} \cdot \left(\frac{t}{T_M} \right)^v \right] \quad => \quad G(s) = \frac{K_s}{(1 + sT_M)^n}$$

Wobei K_s der Proportionalbeiwert, T_M die Zeitkonstante und n die Ordnung der Strecke ist.
Ermitteln der Ordnung anhand der Tabelle der Zeitprozentkennwerte [7] (S. 158):

n	α_{10}	α_{50}	α_{90}	μ_A
1	9,491	1,443	0,434	0,046
2	1,880	0,596	0,257	0,137
3	0,907	0,374	0,188	0,207
4	0,573	0,272	0,150	0,261
5	0,411	0,214	0,125	0,304
6	0,317	0,176	0,108	0,340
7	0,257	0,150	0,095	0,370
8	0,215	0,130	0,085	0,396
9	0,184	0,115	0,077	0,418
10	0,161	0,103	0,070	0,438

Tabelle 9: Tabelle der Zeitprozentkennwerte

$$\mu = \frac{t_{10}}{t_{90}} = \frac{15ms}{72ms} = 0{,}208 \quad => \quad n = 3$$

Berechnung von K_s:

$$K_s = \frac{\Delta x}{\Delta y} = \frac{59{,}8\%}{50\%} = 1{,}196$$

Berechnung von T_M:

$$T_M = \frac{1}{3} \cdot (t_{10} \cdot \alpha_{10} + t_{50} \cdot \alpha_{50} + t_{90} \cdot \alpha_{90})$$

$$T_M = \frac{1}{3} \cdot (17ms \cdot 0{,}907 + 40ms \cdot 0{,}374 + 78ms \cdot 0{,}188) = 14{,}4ms = 0{,}0144s$$

Daraus ergibt sich die mathematische Beschreibung der Strecke:

$$G(s) = \frac{1{,}2}{(1 + s0{,}0144s)^3}$$

B.2 Einstellverfahren nach Latzel

Beim Einstellverfahren nach Latzel [15] geht man anhand einer Tabelle vor, aus der man durch minimale Umrechnungen die Werte für die Nachstellzeit T_n und den Proportionalbeiwert K_p errechnet.

	PI-Regler		PID-T_1-Regler *)			
n	T_n / T_M	$K_P \cdot K_S$	T_n / T_M	T_v / T_M	T_d / T_M	$K_P \cdot K_S$
2	1,55	1,650	PID-T_1-Regler nicht sinnvoll			
3	1,96	0,884	2,47	0,66	0,13	2,543
4	2,30	0,656	2,92	0,84	0,17	1,461
5	2,59	0,540	3,31	0,99	0,20	1,109
6	2,86	0,468	3,66	1,13	0,23	0,914
7	3,10	0,417	3,97	1,25	0,25	0,782
8	3,32	0,379	4,27	1,36	0,27	0,689
9	3,53	0,349	4,54	1,47	0,29	0,617
10	3,73	0,325	4,80	1,57	0,31	0,559

Tabelle 10: Einstellwerte nach Latzel [15]

Für einen PI-Regler und einer Strecke 3. Ordnung gilt dann:

$$\frac{T_n}{T_M} = 1{,}96 \qquad T_n = T_M \cdot 1{,}96 \qquad T_n = 0{,}0282s$$

$$K_p \cdot K_s = 0{,}884 \qquad K_p = \frac{0{,}884}{K_s} \qquad K_p = 0{,}74$$

B.3 Berechnung der Parameter

Alternativ zu den Einstellregeln nach Latzel können die Parameter auch berechnet werden. Die Strecke besteht in der vorliegenden Beschreibung aus drei gleichen Zeitgliedern. Damit kann die Methode der Betragsanpassung angewandt werden. Dabei wird die Kreisfrequenz ϖ_{03} errechnet, bei der der Amplitudengang um 3 dB gegenüber dem Wert $\varpi = 0$ abgesunken ist. Die Nachstellzeit des PI-Reglers wird nun so ausgelegt, dass der 3 dB Abfall der Streckenkreisfrequenz durch den entsprechenden Anstieg des Reglers kompensiert wird [3] (S. 100). Mathematisch beschrieben:

$$\left|G_s(j\varpi_{03})\right|_{dB} - \left|G_s(j0)\right|_{dB} = -3dB$$

Im linearen Bereich:

$$-3dB \mathrel{\hat{=}} \frac{1}{\sqrt{2}} \qquad => \qquad \frac{\left|G_s(j\varpi_{03})\right|}{\left|G_s(j0)\right|} = \frac{1}{\sqrt{2}}$$

Die vorhandene Strecke einsetzen und Betrag auflösen:

$$\frac{K_s}{\left(1+\varpi_{03}{}^2 T_M{}^2\right)^3} \cdot \frac{1}{K_s} = \frac{1}{\sqrt{2}}$$

Umstellen nach ϖ_{03}:

$$\varpi_{03} = \frac{1}{T_M}\sqrt{\sqrt[3]{2}-1} \qquad \varpi_{03} = 35{,}4s^{-1}$$

$$T_n = \frac{1}{\varpi_{03}} = \frac{1}{35{,}4s^{-1}} = 0{,}0282s$$

Der Proportionalbeiwert soll nun so gewählt werden, dass der Regler 10 % überschwingt. Dazu wird eine Faustformel angewandt, die besagt, dass Phasenreserve in Grad plus Überschwingen in Prozent 70 ergeben soll. Demnach muss die Phasenreserve auf 60° eingestellt werden. Da die Stabilitätsgrenze eines Systems nach dem Nyquist-Kriterium bei -180° liegt, wird der Winkel auf -120° eingestellt.

Vom offenen Regelkreis aus kann der Phasenverzug in Grad ermittelt werden:

$$G_o(j\varpi) = \frac{K_p(1+sT_n)}{sT_n} \cdot \frac{K_s}{(1+sT_M)^3}$$

$$G_o(j\varpi_d) = -90° + \arctan(\varpi_d T_n) - 3 \cdot \arctan(\varpi_d T_M) = -120°$$

Da die Auflösung von ϖ_d sehr aufwändig ist, wurde ϖ_d numerisch ermittelt. Als Ergebnis wurde für einen Winkel von $-119{,}86°$ $\varpi_d = 30s^{-1}$ berechnet.

Aus dem Einheitskreis des Nyquist-Kriteriums leitet sich her, dass der Betrag des offenen Regelkreises eins sein muss, um K_p für diese Phasenreserve einzustellen [25] (S. 304ff).

$$\left|G_o\left(j\varpi_d\right)\right| \stackrel{!}{=} 1$$

Vorliegende Beschreibung des Reglers einsetzen und Betrag auflösen:

$$1 = \frac{K_p \cdot K_s \cdot \sqrt{1+\varpi_d{}^2 T_n{}^2}}{\varpi_d T_n \left(\sqrt{1+\varpi_d{}^2 T_M{}^2}\right)^3}$$

Nach K_p umstellen:

$$K_p = \frac{\varpi_d T_n \left(\sqrt{1+\varpi_d{}^2 T_M{}^2}\right)^3}{K_s \cdot \sqrt{1+\varpi_d{}^2 T_n{}^2}} \qquad K_p = 0{,}696$$

Es wird deutlich, dass mit den Einstellregeln nach Latzel und mit der Berechnung der Parameter in etwa die gleichen Werte ermittelt werden können. Nachdem diese Werte im Programm eingetragen wurden, konnte durch empirisches Einstellen ein optimaler Parametersatz gefunden werden.

Proportionalbeiwert: $K_p = 0{,}77$

Nachstellzeit: $T_n = 0{,}026\,\mathrm{s}$

Vorhaltezeit: $T_v = 0\,\mathrm{s}$

Anhang C Ausführliche Berechnung der Geschwindigkeitsregler

C.1 Unterlagerter Regelkreis

Für die unterlagerte Drehzahlregelung soll ein P-Regler parametrisiert werden. Der Proportionalbeiwert K_p muss so eingestellt werden, dass bei maximaler Ansteuerung die maximale Drehzahl erreicht werden kann. Bei den ersten Versuchen mit K_p-Werten zwischen 0,5 und 2 stellte sich heraus, dass vor dem Erreichen der Maximaldrehzahl ein Übersteuern der Räder sichtbar wurde. Dieses Problem wird behoben, indem der Istwert durch die Variable `divide` vorgeteilt wird. Der Wert, der für K_p empirisch eingestellt wurde, beträgt:

$$K_p = 1{,}35$$

Das Ergebnis ist in Abbildung 32 auf Seite 81 zu sehen.

C.2 Mathematische Beschreibung der Strecke (unterlagerter Kreis und Strecke)

Der überlagerte Regelkreis soll als PI-Regler parametrisiert werden. Die Vorgehensweise ist die gleiche wie beim Entwurf des Drehzahlreglers. Die Strecke hat die gleiche mathematische Beschreibung mit einer anderen Ordnung und anderen Werten. Um diese zu ermitteln, wurde die Sprungantwort der Geschwindigkeit aufgenommen und ausgewertet. Es wurde die Maximalgeschwindigkeit vorgegeben. Dadurch, dass der unterlagerte P-Regler genau auf die Maximaldrehzahl eingestellt wurde, bleibt eine bleibende Regeldifferenz aus. Um das Signal besser auswerten zu können, wurde in das stufige Signal eine Annäherung an den realen Verlauf interpoliert.

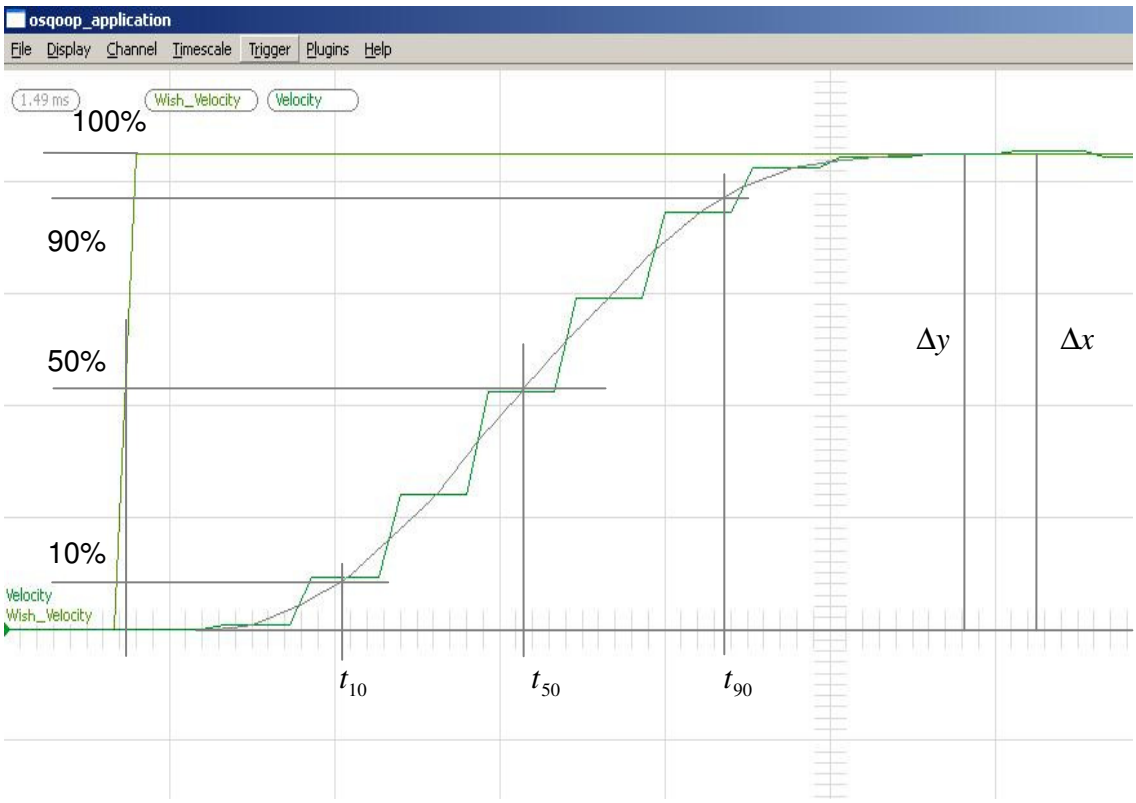


Abbildung 40: Auswertung der Sprungantwort der Lineargeschwindigkeit

Aus der Sprungantwort werden folgende Werte abgelesen:

$\Delta x = 100\%$	Differenz des Istwertes (1397 mm/s)
$\Delta y = 100\%$	Differenz des Sollwertes (1397 mm/s)
$t_{10} = 20ms$	Zeit, nach der der Istwert auf 10 % angestiegen ist
$t_{50} = 36ms$	Zeit, nach der der Istwert auf 50 % angestiegen ist
$t_{90} = 54ms$	Zeit, nach der der Istwert auf 90 % angestiegen ist

Formal wird eine Strecke höherer Ordnung wie folgt beschrieben [3] (S. 128):

$$x(t) = K_s \cdot \Delta y \left[1 - e^{\frac{t}{T_M}} \cdot \sum_{v=0}^{n-1} \frac{1}{v!} \cdot \left(\frac{t}{T_M} \right)^v \right] \quad \Rightarrow \quad G(s) = \frac{K_s}{(1 + sT_M)^n}$$

Wobei K_s der Proportionalbeiwert, T_M die Zeitkonstante und n die Ordnung der Strecke ist.

Ermitteln der Ordnung anhand der Tabelle der Zeitprozentkennwerte. Die Tabelle ist bereits in Anhang B „Ausführliche Berechnung des Drehzahlreglers“ aufgeführt (Tabelle 9).

$$\mu = \frac{t_{10}}{t_{90}} = \frac{20ms}{54ms} = 0{,}370 \qquad => \qquad n = 7$$

Berechnung von K_s:

$$K_s = \frac{\Delta x}{\Delta y} = \frac{100\%}{100\%} = 1$$

Berechnung von T_M:

$$T_M = \frac{1}{3} \cdot \left(t_{10} \cdot \alpha_{10} + t_{50} \cdot \alpha_{50} + t_{90} \cdot \alpha_{90}\right)$$

$$T_M = \frac{1}{3} \cdot \left(20ms \cdot 0{,}257 + 36ms \cdot 0{,}150 + 54ms \cdot 0{,}095\right) = 5{,}2ms = 0{,}0052s$$

Daraus ergibt sich die mathematische Beschreibung der Strecke:

$$G(s) = \frac{1}{\left(1 + s0{,}0052s\right)^7}$$

C.3 Parametrisierung überlagerter Regelkreis nach Latzel

Da die Vorgehensweise die gleiche wie für den Drehzahlregler ist, wird sie nicht näher ausgeführt. Die Ergebnisse sind:

$$\frac{T_n}{T_M} = 3{,}1 \qquad T_n = T_M \cdot 3{,}1 \qquad T_n = 0{,}016s$$

$$K_p \cdot K_s = 0{,}417 \qquad K_p = \frac{0{,}417}{K_s} \qquad K_p = 0{,}417$$

C.4 Berechnung der Parameter des überlagerten Regelkreises

Auch hier wird die gleiche Vorgehensweise wie für den Drehzahlregler angewandt. Dieser Zusammenhang wird umgesetzt:

$$\frac{|G_s(j\varpi_{03})|}{|G_s(j0)|} = \frac{1}{\sqrt{2}}$$

Die vorhandene Strecke einsetzen und Betrag auflösen:

$$\frac{K_s}{\left(1+\varpi_{03}{}^2 T_M{}^2\right)^7} \cdot \frac{1}{K_s} = \frac{1}{\sqrt{2}}$$

Umstellen nach ϖ_{03}:

$$\varpi_{03} = \frac{1}{T_M}\sqrt{\sqrt[7]{2}-1} \qquad \varpi_{03} = 62s^{-1}$$

$$T_n = \frac{1}{\varpi_{03}} = \frac{1}{62s^{-1}} = 0{,}016s$$

Der Proportionalbeiwert soll nun so gewählt werden, dass der Regler 10 % überschwingt.

Vom offenen Regelkreis aus kann der Phasenverzug in Grad ermittelt werden:

$$G_o(j\varpi) = \frac{K_p(1+sT_n)}{sT_n} \cdot \frac{K_s}{(1+sT_M)^7}$$

$$G_o(j\varpi_d) = -90° + \arctan(\varpi_d T_n) - 7 \cdot \arctan(\varpi_d T_M) = -120°$$

Da die Auflösung von ϖ_d sehr aufwändig ist, wurde ϖ_d numerisch ermittelt. Als Ergebnis wurde für einen Winkel von $-120{,}05°$ $\varpi_d = 25s^{-1}$ berechnet.

Aus dem Einheitskreis des Nyquist-Kriteriums leitet sich her, dass der Betrag des offenen Regelkreises 1 sein muss, um K_p für diese Phasenreserve einzustellen [25] (S. 304ff).

$$|G_o(j\varpi_d)| \overset{!}{=} 1$$

Vorliegende Beschreibung des Reglers einsetzen und Betrag auflösen:

$$1 = \frac{K_p \cdot K_s \cdot \sqrt{1+\varpi_d{}^2 T_n{}^2}}{\varpi_d T_n \left(\sqrt{1+\varpi_d{}^2 T_M{}^2}\right)^7}$$

Nach K_p umstellen:

$$K_p = \frac{\varpi_d T_n \left(\sqrt{1+\varpi_d{}^2 T_M{}^2}\right)^7}{K_s \cdot \sqrt{1+\varpi_d{}^2 T_n{}^2}} \qquad K_p = 0{,}394$$

Es wird deutlich, dass mit den Einstellregeln nach Latzel und mit der Berechnung der Parameter in etwa die gleichen Werte ermittelt werden können. Nachdem diese Werte im Programm eingetragen wurden, konnte durch empirisches Einstellen ein optimaler Parametersatz gefunden werden.

Proportionalbeiwert: $K_p = 0{,}45$

Nachstellzeit: $T_n = 0{,}016\,\mathrm{s}$

Vorhaltezeit: $T_v = 0\,\mathrm{s}$

C.5 Rotationsgeschwindigkeitsregelung

Für die Rotationsgeschwindigkeitsregelung kann die gleiche mathematische Beschreibung verwendet werden. Dies belegt ein Auswerten einer Sprungantwort für die Rotationsgeschwindigkeit.

Abbildung 41: Analyse der Sprungantwort der Rotationsgeschwindigkeit

Aus der Sprungantwort werden folgende Werte abgelesen:

$\Delta x = 102\%$ Differenz des Istwertes (385 °/s)

$\Delta y = 100\%$ Differenz des Sollwertes (377 °/s)

$t_{10} = 20ms$ Zeit, nach der der Istwert auf 10 % angestiegen ist

$t_{50} = 37ms$ Zeit, nach der der Istwert auf 50 % angestiegen ist

$t_{90} = 55ms$ Zeit, nach der der Istwert auf 90 % angestiegen ist

Formal wird eine Strecke höherer Ordnung wie folgt beschrieben [3] (S. 128):

$$x(t) = K_s \cdot \Delta y \left[1 - e^{\frac{t}{T_M}} \cdot \sum_{v=0}^{n-1} \frac{1}{v!} \cdot \left(\frac{t}{T_M} \right)^v \right] \quad \Rightarrow \quad G(s) = \frac{K_s}{(1 + sT_M)^n}$$

Wobei K_s der Proportionalbeiwert, T_M die Zeitkonstante und n die Ordnung der Strecke ist.

Ermitteln der Ordnung anhand der Tabelle der Zeitprozentkennwerte. Die Tabelle ist in Anhang B „Ausführliche Berechnung des Drehzahlreglers“ bereits verwendet worden (Tabelle 9).

$$\mu = \frac{t_{10}}{t_{90}} = \frac{20ms}{55ms} = 0{,}364 \quad \Rightarrow \quad n = 7$$

Berechnung von K_s:

$$K_s = \frac{\Delta x}{\Delta y} = \frac{102\%}{100\%} = 1{,}02$$

Berechnung von T_M:

$$T_M = \frac{1}{3} \cdot (t_{10} \cdot \alpha_{10} + t_{50} \cdot \alpha_{50} + t_{90} \cdot \alpha_{90})$$

$$T_M = \frac{1}{3} \cdot (20ms \cdot 0{,}257 + 37ms \cdot 0{,}150 + 55ms \cdot 0{,}095) = 5{,}3ms = 0{,}0053s$$

Daraus ergibt sich die mathematische Beschreibung der Strecke:

$$G(s) = \frac{1}{(1 + s0{,}0053s)^7}$$

Da sich die Werte der Sprungantworten der Linear- und Rotationsgeschwindigkeit gleichen, wurden die Ergebnisse der Lineargeschwindigkeit verwendet und durch empirisches Einstellen optimiert. Daraus ergab sich folgender Parametersatz:

Proportionalbeiwert: $K_p = 0{,}40$

Nachstellzeit: $T_n = 0{,}016\,\text{s}$

Vorhaltezeit: $T_v = 0\,\text{s}$