

# Bachelorarbeit

Institut für Informatik  
Fachbereich Informatik und Mathematik

## *Entwurf einer Steuerung für ein selbstbalancierendes Fahrrad*

Jonah Zander  
Matrikel-Nr.: 7345074

Prüfer:

Prof. Dr.-Ing. Lars Hedrich  
Apl. Prof. Dr. Mathias Pacher

Betreuer:

Prof. Dr.-Ing. Lars Hedrich

Abgabedatum:

31.08.2023

Frankfurt am Main, 31. August 2023

**Bitte dieses Formular zusammen mit der Abschlussarbeit abgeben!**

## **Erklärung zur Abschlussarbeit**

**gemäß § 35, Abs. 16 der Ordnung für den Bachelorstudiengang Informatik  
vom 17. Juni 2019:**

Hiermit erkläre ich

Zander, Jonah  
(Nachname, Vorname)

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ich bestätige außerdem, dass die vorliegende Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass alle eingereichten schriftlichen gebundenen Versionen meiner vorliegenden Bachelorarbeit mit der digital eingereichten elektronischen Version meiner Bachelorarbeit übereinstimmen.

Frankfurt am Main, den 31.08.2023

Jonah Zander

Unterschrift der/des Studierenden

# **Zusammenfassung**

Auf Basis einer gegebenen Hardware-Plattform wird in dieser Arbeit eine Software-Steuerung für ein selbstfahrendes Fahrrad entwickelt. Die Steuerung folgt einem Multi-Thread Ansatz, für das Auslesen und Ansteuern der Sensoren und Aktoren unter Debian-Linux. Unter Verwendung eines echtzeitfähigen Mikrocontrollers wird die zeitkritische Auswertung eines Sensors bearbeitet. Logging und graphische Auswertung der Protokolldaten werden implementiert. Die Hardware-Plattform wird um eine Sicherheitsmaßnahme erweitert. Details zu Fähigkeiten der Plattform werden beschrieben. Zuletzt wird eine erfolgreiche Testfahrt detailliert betrachtet.

# Aufgabenstellung

Am Ende dieses Projektes soll sich ein selbstbalancierendes Fahrrad auf ebenem Boden für mindestens 60 Sekunden bewegen können, ohne dass dabei die am Fahrrad angebrachten Stützräder den Boden berühren.

Bei dem Entwurf eines Systems ist es oft sehr schwer, verschiedene Sensoren und Aktoren zu verbinden. Die Sensoren müssen richtig und rechtzeitig ausgelesen, die Werte gegebenenfalls überprüft und verarbeitet werden. Die Aktoren müssen angesteuert werden, sich zuverlässig verhalten und mit Sicherheitsmaßnahmen versehen werden, sodass sie sich selbst nicht beschädigen und keine Gefahr darstellen. All das macht es herausfordernd, ein zuverlässiges System zu entwerfen und zu entwickeln.

Das Ziel dieser Arbeit ist es, eine Steuerung für ein selbstbalancierendes Fahrrad zu erarbeiten. Dazu ist ein Fahrrad mit einem Lenkungsmotor, Hinterradmotor, Sensorik und Mikrokontroller (BeagleBone Black) gegeben, auf welchem Debian läuft. Durch die zu erstellende Software werden die Sensoren ausgelesen und Aktoren angesteuert. Diese Arbeit baut auf vorherigen Arbeiten auf, welche eine grundlegende Struktur für dieses Projekt bieten [Ranz] [Kara].

Der auf dem BeagleBone laufende Code soll übersichtlich und erweiterbar gestaltet sein. Die Sensor- und Aktor-Werte sollen protokolliert und in verständlicher Weise dargestellt werden können.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Stand der Technik . . . . .	1
<b>2 Fahrrad-Plattform</b>	<b>3</b>
2.1 Mikrokontroller . . . . .	4
2.1.1 BeagleBone Black (BBB) . . . . .	4
2.1.2 Programmable Realtime Unit (PRU) . . . . .	6
2.2 Sensoren . . . . .	6
2.2.1 BNO055 . . . . .	6
2.2.2 MD49 . . . . .	7
2.2.3 BTS7960 . . . . .	8
2.2.4 Empfänger/Fernbedienung . . . . .	9
2.3 Aktoren . . . . .	9
2.3.1 EMG49 . . . . .	9
2.3.2 Hinterradnabenmotor (HNM) . . . . .	10
2.4 Energie . . . . .	11
2.5 Sicherheit . . . . .	11
2.6 Fähigkeiten des Gesamtsystems . . . . .	11
2.7 Fehlende Funktionalität . . . . .	12
<b>3 Methodik</b>	<b>13</b>
3.1 Programmiersprache . . . . .	13
3.2 Anforderungen an den Code . . . . .	13
3.3 PRU . . . . .	14
3.3.1 Geschwindigkeits- und Distanzmessung . . . . .	15
3.4 Modularisierung in C . . . . .	18
3.4.1 I2C . . . . .	19
3.4.2 BNO055 . . . . .	19
3.4.3 MD49 . . . . .	20
3.4.4 IO . . . . .	21
3.4.5 BTS7960 . . . . .	23
3.4.6 Fernbedienung . . . . .	24
3.4.7 PID Kontroller . . . . .	26

3.4.8 Autobike . . . . .	27
3.5 Logging . . . . .	31
<b>4 Tests und Evaluation</b>	<b>32</b>
4.1 Lenkmotor-Test . . . . .	32
4.1.1 Lenkwinkel . . . . .	38
4.1.2 Hysterese . . . . .	39
4.2 Hinterradnabenmotor Test . . . . .	40
4.3 Gesamttest des Fahrrads . . . . .	44
<b>5 Zusammenfassung und Ausblick</b>	<b>56</b>
<b>Literaturverzeichnis</b>	<b>58</b>
<b>A Anhang</b>	<b>59</b>
A.1 Kompilieren des Codes . . . . .	59
A.2 Erstellen von Diagrammen mittels plot.py . . . . .	59
A.3 USB-WLAN-Adapter . . . . .	60
A.4 Samba . . . . .	62

# 1 Einleitung

In den letzten Jahren haben sich Geräte, welche in irgendeiner Form die Balance halten sollen, großer Beliebtheit erfreut. So z.B. Drohnen, welche sich mittels der meist vier Propeller in der Luft halten können oder sogenannte ‚Hoverboards‘, welche zur Fortbewegung benutzt werden, indem sich eine Person darauf stellt, nach vorne lehnt und das Hoverboard beschleunigt, damit die Person nicht fällt. In beiden Fällen müssen die Geräte auf äußere Veränderungen reagieren. Bei den Hoverboards muss das Gerät die Beschleunigung anpassen, falls sich die stehende Person über den Schwerpunkt hinaus lehnt und bei Drohnen müssen diese auch noch stabil fliegen, selbst wenn ein starker Wind auf eine Seite der Drohne wirkt. Diese Veränderungen müssen rechtzeitig erfolgen.

Ähnliche Herausforderungen finden sich ebenfalls bei der Entwicklung eines selbstfahrenden Fahrrads.

## 1.1 Stand der Technik

Im Bereich des Designs und der Analyse von selbstfahrenden Fahrrädern gibt es einige Arbeiten. Diese befassen sich jedoch meist mit der Stabilisierung durch ein Reaktionsrad wie z.B. bei Nitheesh et al. welche in ihrer Arbeit ein mathematisches Modell dazu aufgestellt haben [kuot].

Tamaldin et al. zählen verschiedene Arten ein Fahrrad zu steuern auf [Taot17]. Sie beschreiben ebenfalls ein Reaktionsrad als geeignet, da dieser Ansatz ein großes Drehmoment liefern und das Fahrrad sogar im Stillstand stabilisieren könne. Die Vorteile einer Balanceregelung mittels Lenkradsteuerung seien dagegen ein geringeres Gewicht und Energieverbrauch.

S.H. Park und S.Y. Yi haben ein Motorrad-Modell getestet welches 2 Reaktionsräder verwendet, um die Balance zu halten [PaYi20].

Im Bereich der Motorräder gibt es von verschiedenen Herstellern Konzepte, welche das Gleichgewicht halten sollen. Der Fokus liegt hier jedoch auch auf dem Halten des Gleichgewichts im Stillstand. Der Honda Riding Assist 2.0 kann z.B. das Hinterrad nach rechts und links ausschwenken, um durch diese Gewichtsverlagerung das Gleichgewicht wieder herzustellen [Baum22]. Ein ähnliches Konzept gibt es auch von Yamaha [Yama17].

Für den Ansatz einer Balanceregelung nur durch Lenkradsteuerung findet sich wenig. Hier haben andere Studenten, welche bereits an dieser Fahrrad-Plattform gearbeitet haben, wichtige Vorarbeit geleistet.

Anna Ranz hat 2021 einen wesentlichen Beitrag in der Wahl der Hardware-Komponenten des hier verwendeten Fahrrads geleistet, welche sich möglichst aus Standard-Bauteilen zusammensetzen sollten, die an ein normales Fahrrad montiert wurden. Weiterhin hat sie in ihrer Arbeit unter anderem ein mathematisches Modell des Fahrrads betrachtet [Ranz].

Durch Amine Karabila wurden 2022 Komponenten der Schaltung in LTSpice übertragen und die Funktionalität der Plattform um eine Ansteuerung des Hinterradmotors erweitert [Kara]. Grundlagen für die Kommunikation einzelner Komponenten untereinander wurden durch beide geschaffen.

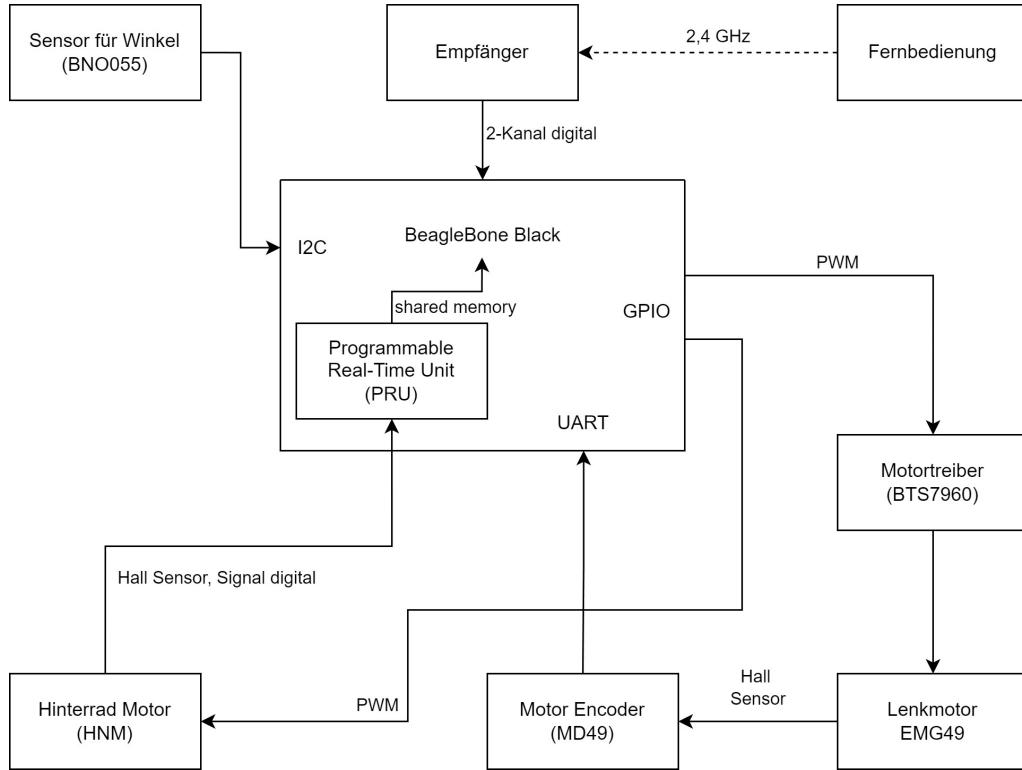
## Aufbau dieser Arbeit

Diese Arbeit betrachtet zuerst die einzelnen Komponenten des Fahrrads und geht danach genauer ins Detail, wie die einzelnen Komponenten angesprochen werden. Diese werden mit geeigneter Softwaresteuerung kombiniert und Fahrten mit diesem System betrachtet. Das Problem eines selbstbalancierenden Fahrrads wird aus einer praktischen Sicht bearbeitet.

## 2 Fahrrad-Plattform

Um eine Lenkradsteuerung für die Balanceregelung zu testen werden Motoren benötigt, die sowohl das Hinterrad, als auch das Lenkrad antreiben. Ebenfalls erforderlich sind Sensoren, die Informationen über die Rotation der Motoren liefern, sowie ein absoluter Lagesensor, um die Ausrichtung des Fahrrads zu bestimmen. Die Daten von jedem dieser Sensoren müssen verarbeitet werden, um geeignete Anweisungen an die Motoren zu senden. Diese Aufgabe wird von einem Einplatinencomputer übernommen, der auch die erfassten Werte aufzeichnet. Zusätzlich kommt eine Fernbedienung zum Einsatz, um dem Fahrrad eine Richtung vorgeben zu können und die Durchführung von Tests zu erleichtern.

Die Abbildung 2.1 zeigt die verschiedenen Komponenten, die in diesem Projekt verwendet werden. Anschließend werden die Fähigkeiten und die Verwendung dieser Komponenten erläutert.



**Abbildung 2.1:** Übersicht über die essentiellen Bestandteile des selbstbalancierenden Fahrrads. Batterie und Level-Shifter wurden ausgelassen. Pfeile repräsentieren Flussrichtung wichtiger Daten für die Steuerung.

## 2.1 Mikrokontroller

### 2.1.1 BeagleBone Black (BBB)

#### 2.1.1.1 Zweck

Der auf dem Board laufende C-Code bildet das Zentrum des Projekts. Es werden die Eingaben der Sensorik kombiniert und die passenden Befehle an die Aktoren gesendet.

#### 2.1.1.2 Allgemeine Informationen

Der BBB ist eine Entwicklungsplattform mit einem AM355x 1GHz ARM Cortex-A8 Prozessor mit 512MB DDR3 RAM und außerdem zwei Programmable Real-Time Units 32-bit Mikrocontrollern (siehe: 2.1.2). Eine Verbindung zum BeagleBone Black kann über Ethernet oder USB erfolgen. Das Board kann mit 5 Volt über ein USB-Kabel oder ein Netzteil versorgt werden.

Die für dieses Projekt wichtigsten Kommunikationsformen, welche der BBB bietet sind die I/O-Pins, Analog-Inputs, PWM-Output, SPI, UART Serial Ports und I2C [BF]. Weitere Funktionalitäten, die in diesem Projekt ungenutzt bleiben, sind z.B. das Controller Area Network, Multichannel Audio Serial Port, Touchscreen und LCD Controller.

**PWM-Output:** Der AM355x hat insgesamt acht PWM-Outputs. Jeweils zwei durch drei eHRPWM Module und jeweils einen durch die zwei eCAP Module. Manche dieser PWM Ausgänge können an mehrere Pins des BeagleBone gelegt werden.

**UART:** Der Sitara Prozessor hat insgesamt 6 UART Module, wobei nur 4 durch die BBB Pins genutzt werden können. Mögliche Baud Raten für die Kommunikation reichen von 300 bps bis zu 3.6864 Mbps [TI19, Kapitel 19]. Die Module haben Software und Hardware flow control. Die Zahl der zu übertragenen und empfangenen Bits kann zwischen 5-8 eingestellt werden mit optionaler parity Bit Generierung/Erkennung sowie die Wahl zwischen 1, 1,5 und 2 Stop-Bits.

**I2C:** Auf dem Board befinden sich 3 I2C-Module, wobei nur 2 für den Nutzer zugänglich sind, da einer für den internen HDMI Kontrollbus verwendet wird. Diese Module unterstützen 7 oder 10 Bit Adressierung und eine Bus-Frequenz von bis zu 400 kHz.

**Analog-Inputs:** Dem Nutzer stehen 7 analoge Inputs zur Verfügung, wobei diese mit Vorsicht zu verwenden sind, da sie Teil des Touchscreen Controllers sind und für eine Spannung von 0-1,8V ausgelegt sind.

**GPIO:** Der BeagleBone Black hat insgesamt 4 GPIO Module mit jeweils 32 Input/Output Pins, jedoch sind nicht alle mit den BeagleBone Header Pins verbunden. Sie können Interrupt Requests auslösen, haben ein Keyboard Interface mit Entprellung und können genutzt werden für Daten Input und Output. Die zulässige Spannung an diesen Pins ist 0-3,3V und es können je nach Pin maximal 4-6 mA bereitgestellt werden und höchstens 8 mA abfließen. Es besteht auch die Möglichkeit Pull-Up oder Pull-Down Widerstände für die einzelnen Pins zu aktivieren.

### 2.1.1.3 Projektspezifische Informationen

Das Betriebssystem auf dem BBB ist eine Debian GNU/Linux Version 10 (buster) Distribution. Sie ist auf einer SD Karte installiert um das Programm sowie Systemeinstellungen als gänze kopieren zu können. Zur Kommunikation dient der Ethernet Port, über welchen man sich mittels SSH mit dem Board verbinden kann. Es ist außerdem ein USB-WLAN-Adapter am Beaglebone angebracht, über welchen man sich ebenfalls verbinden kann. Eine weitere bereits vorinstallierte Möglichkeit das Board zu verwenden ist mittels der web-basierten IDE Cloud9, welche automatisch durch das Board gestartet wird.

### 2.1.2 Programmable Realtime Unit (PRU)

#### 2.1.2.1 Zweck

Die Ausführungszeit von Programmen auf PRU's ist deterministisch weshalb sie sich besonders gut für Zeit sensible Aufgaben eignen. Sie bieten einen einfachen Befehlssatz und können so durch Entwickler genutzt werden um Funktionalitäten eingebetteter Systeme zu erweitern. Auf dem BBB befinden sich davon zwei 32-Bit Einheiten. Einer der beiden PRU's wird verwendet um die Geschwindigkeit des Hinterradnabenmotors, sowie die zurückgelegte Distanz zu bestimmen.

#### 2.1.2.2 Allgemeine Informationen

Die PRU's laufen mit einer Taktrate von 200Mhz und verfügen jeweils über 8kB Programm- und Datenspeicher. Die PRU's haben einen geteilten 12kB Speicher. Sie haben enhanced GPIO welche auch an bestimmte Pins des BBB weitergeleitet werden können. Sie können außerdem auf alle Ressourcen des SoC durch den Interface/OCP Master Port zugreifen [TI19, Seite 199] und können somit Speicher mit dem Linux-Host teilen. Die PRU's haben einen geteilten Interrupt Controller, welcher bis zu 64 Input-Events unterstützt.

## 2.2 Sensoren

### 2.2.1 BNO055

#### 2.2.1.1 Zweck

Der BNO055 ist ein absoluter Orientierungssensor [Bosc21]. Dieser wird benutzt um die Neigung des Fahrrads zu bestimmen und anhand dessen die Lenkung zu steuern.

#### 2.2.1.2 Allgemeine Informationen

Der BNO055 beinhaltet einen Beschleunigungssensor, Gyroskop und Magnetometer. Die Daten dieser Sensoren werden durch einen Algorithmus kombiniert und können in verschiedenen Formen zur Verfügung gestellt werden, z.B. als Quaternion, Euler-Winkel, linearer Beschleunigungs- oder Gravitationsvektor. Auch ein Temperatursensor ist vorhanden um temperaturbedingte Veränderungen der Sensordaten auszugleichen. Je nachdem in welchem Modus der Sensor betrieben wird, kann es sein, dass z.B. die Daten des Gyroskops ignoriert werden. Es wird außerdem eine automatische Kalibration aller Sensoren im Hintergrund durchgeführt [Bosc21, page 24]. Der Sensor kann mittels I2C oder UART kommunizieren. Der IMU-Betriebsmodus ist schnell, wodurch die Sensordaten mit einer Frequenz

von 100Hz berechnet werden können. Die Magnetometer Daten werden in diesem Modus jedoch nicht beachtet.

### 2.2.1.3 Projektspezifische Informationen

Der in diesem Projekt genutzte BNO055 ist auf einer Breakout-Platine angebracht. Der BeagleBone Black kommuniziert mit dem Sensor über I2C. Einige Funktionen zur Kommunikation mit diesem Chip sind bereits durch die Vorgänger umgesetzt worden, wurden jedoch größtenteils noch einmal überarbeitet um Korrekturen vorzunehmen. So wurden Wartezeiten um den Betriebsmodus des Sensors zu ändern angepasst, um unerwartetes Verhalten auszuschließen, des weiteren wurde die Interpretation der Sensordaten im Code angepasst, da fälschlicherweise angenommen wurde, dass es sich um Winkel im Bogenmaß handelte, wobei sie im Gradmaß gegeben wurden.

Es gibt außerdem wichtige Hinweise die im Datenblatt genannt werden, welche auch für dieses Projekt von Bedeutung sind:

Der Fusion Algorithmus nutzt die Daten des Beschleunigungssensors um den Drift des Gyroskops auszugleichen. Wenn der Sensor in Bewegung ist, werden die Daten des Beschleunigungssensors jedoch temporär ignoriert. Dadurch kann der Neigungswinkel, falls sich der Sensor über eine lange Zeit hin bewegt oder langanhaltende Vibration erfährt, abdriften [Bosc21, Seite 24, Notiz 2].

Der BNO055 wurde eigentlich zur Verfolgung von menschlichen Bewegungen entwickelt. Falls der Sensor über längere Zeit stark beschleunigt wird, könnte dies dazu führen, dass die Richtung der Beschleunigung fälschlicherweise als Gravitationsvektor interpretiert wird [Bosc21, Seite 24, Notiz 3].

In den durchgeführten Tests gab es bisher keine Probleme mit einem Drift des Sensors oder einen Fehler durch schnelle Beschleunigungen. Jedoch könnten diese Hinweise für zukünftige Arbeiten wichtig sein, falls die Beschleunigung des Fahrrads geändert werden oder das Fahrrad über eine unebene Kieselstrecke oder Kopfsteinpflaster fahren soll, welche für viele Vibrationen sorgen könnte. Unklar ist, nach welcher Zeit solche Effekte zu beobachten wären.

## 2.2.2 MD49

### 2.2.2.1 Zweck

Der MD49 ist ein 24 Volt 5 Ampere H-Brücken Motortreiber. Er wird dazu genutzt die Position des Motors für die Lenkradsteuerung EMG49 (siehe: 2.3.1) zu überprüfen.

### 2.2.2.2 Allgemeine Informationen

Der MD49 kann zwei Motoren entweder einzeln oder gleichzeitig kontrollieren. Der Input der Hall Sensoren wird verwendet um Zählerstände bereit zu stellen, welche Aufschluss über Rotationsdistanz sowie Richtung der Rotation geben. Das Board verfügt über Kurzschluss- und Überspannungsschutz, wobei Fehler durch ein Error-Byte analysiert werden können. Der MD49 kann über UART mit einer Baud Rate von 9600 oder auch 38400 angesprochen werden [REb].

### 2.2.2.3 Projektspezifische Informationen

Die Ansteuerung des Lenkungsmotors hat sich, trotz der vielseitigen Funktionen des MD49 in den vorherigen Arbeiten als zu langsam erwiesen, wodurch diese jetzt durch den BTS7960 (siehe Abschnitt 2.2.3) geschieht. Der MD49 wird also momentan lediglich zur Erfassung der Encoder-Werte genutzt. Es wurde bereits ein möglicher Codeansatz zur Kommunikation mit dem MD49 über UART implementiert, welcher jedoch noch einmal als eigenständiger Code ausgelagert und neu implementiert wurde um die Struktur zu verbessern und auch um anders mit fehlerhafter Kommunikation umzugehen. Da der Encoder-Wert ein wichtiger Bestandteil der Steuerung ist, stellt sich die Frage, wie schnell dieser mit dem Beaglebone Black abgefragt werden kann. Unter einfachen Testbedingungen konnte eine Abfragefrequenz von 241Hz erreicht werden.

## 2.2.3 BTS7960

### 2.2.3.1 Zweck

Zwei BTS7960 auf einer Breakout-Platine werden dazu genutzt den Lenkungsmotor (siehe Abschnitt 2.3.1) für das Fahrrad zu steuern. Das Lenkrad ist über einen Riemen mit dem Motor verbunden.

### 2.2.3.2 Allgemeine Informationen

Die zulässige Eingangsspannung für den Motor liegt bei 6-27V mit einem maximalen Strom von 43A. Der Controller wird mit einer Spannung zwischen 3,3-5V betrieben und hat Unter- sowie Überspannungsschutzmechanismen. Die Drehrichtung und Geschwindigkeit des Motors wird mittels der beiden PWM-Signale RPWM bzw. LPWM gesteuert.

### 2.2.4 Empfänger/Fernbedienung

#### 2.2.4.1 Zweck

Um eine einfache Eingabe für Bewegungsrichtung und Geschwindigkeit des Fahrrads zu haben, wurde eine Fernbedienung und ein Empfänger am Fahrrad angebracht. Es wurde die Carson 500500104 Fernbedienung und Empfänger Kombination verwendet da diese klein und kostengünstig ist.

#### 2.2.4.2 Allgemeine Informationen

Diese Fernbedienung verfügt über zwei stufenlos regelbare Kanäle und einen schaltbaren Kanal. Es gibt einen Knopf, welcher die Laufrichtung der Steuerung umkehren kann und es gibt für Gas und Lenkung auch eine Trimmfunktion. Der Sender wird mit vier AAA Batterien betrieben und der Empfänger kann mit bis zu 11V versorgt werden. Im Idealfall hat diese eine Reichweite von 150m und nutzt Sendetechnik mit 2,4GHz. Durch die Verwendung von 2,4GHz wird keine lange Antenne benötigt. Die Nutzung eines Kanals regeln Sender und Empfänger von alleine und durch gleiche Codierung der beiden ist eine Störung durch einen fremden Sender nahezu ausgeschlossen [CARS23].

#### 2.2.4.3 Projektspezifische Informationen

Da der Empfänger ein PWM Signal ausgibt und dieses nur schwierig durch den BeagleBone Black analysiert werden kann, wurde hier ein einfacher Tiefpass-Filter verwendet um das PWM Signal als eine variable Spannung mittels der Analog-Eingänge des Boards auslesen zu können.

## 2.3 Aktoren

### 2.3.1 EMG49

#### 2.3.1.1 Zweck

Der EMG49 ist ein Motor welcher zur Bewegung des Fahrradlenkers verwendet wird. Er ist über einen Riemen mit dem Lenkrad verbunden.

#### 2.3.1.2 Allgemeine Informationen

Der EMG49 arbeitet mit 24 Volt und verfügt über 2 Hall-Effekt Sensoren und einem Übersetzungsgetriebe mit einem Verhältnis von 49:1. Die Hall-Effekt Sensoren können direkt an den MD49 (siehe Abschnitt 2.2.2) angeschlossen werden und werden durch ihn

mit der passenden Spannung versorgt. Das Drehmoment, welches durch diesen Motor bereit gestellt werden kann, beträgt 16 kg/cm. Der Motor kann eine maximale Rotationsgeschwindigkeit von 122 rpm erreichen. Für eine volle Rotation des Motors zählt der Encoder 980 Schritte [REa].

### 2.3.1.3 Projektspezifische Informationen

Für die Kontrollsleife dieses Motors ist es interessant die maximal mögliche Veränderung des Encoder-Zählerwerts, welcher durch den MD49 ausgelesen wird, zu kennen.

$$\begin{aligned}\text{Max Encoder-Wert pro Sekunde} &= \frac{122 \text{ rpm}}{60 \text{ s}} \times 980 \text{ Pulse pro Umdrehung} \\ &= 1992, \bar{6}\end{aligned}$$

## 2.3.2 Hinterradnabenmotor (HNM)

### 2.3.2.1 Zweck

Der Hinterradnabenmotor ist Teil eines E-Bike-Selbstumbau-Sets und treibt das Fahrrad an.

### 2.3.2.2 Allgemeine Informationen

Der Motor dieses Sets ist ein 48 Volt 1500W Motor und ist anstelle des normalen Fahrrad Hinterrades verbaut. Dieses Set enthält außerdem den Motorkontroller, LCD Display mit Eingabefunktionen, einen Pedalsensor (zur Erkennung ob der Benutzer in die Pedale tritt), sowie Gas- und Bremshebel.

Der HNM hat auch einen Hall-Sensor. Eine volle Umdrehung des Hinterrads lässt die Spannung an der Hall-Effekt Sensor Leitung 46 mal zwischen 0 und 5V wechseln, wobei der Reifen einen Durchmesser von ungefähr 63cm hat.

### 2.3.2.3 Projektspezifische Informationen

In diesem Projekt wurde das Set so angepasst, dass der BeagleBone Black den Motor mittels eines PWM-Signals ansteuern kann. Auch wird der Pedalsensor nicht benutzt. So kann der Motor sehr einfach mit dem C-Code auf dem BBB angesteuert werden. Die Erfassung der Geschwindigkeit wird auf dem Display des E-Bike Sets zwar angezeigt, kann aber nicht separat durch den Kontroller ausgegeben werden. Um die Geschwindigkeit des Motors und auch die zurückgelegte Distanz zu erfassen, werden die Daten eines der beiden Hall-Effekt Sensoren an die PRU weitergegeben. Es wird lediglich ein Hall-Sensor

verwendet, da eine Rückwärtsbewegung des Fahrrads nicht vorgesehen ist und dadurch eine Bestimmung der Laufrichtung entfällt. Momentan ist eine bremsende Wirkung des Motors nicht im Code implementiert.

## 2.4 Energie

Den Strom für dieses Fahrrad liefern zwei 24V Lithium-Ionen-Batterien. Die Batterien haben jeweils eine Kapazität von 10,4Ah. Der Lenkmotor wird mittels einer dieser Batterien mit 24V betrieben. Der Hinterradmotor wird durch die Reihenschaltung beider Batterien mit 48V betrieben. Der BeagleBone Black wird mittels eines 24V zu 5V Spannungswandlers betrieben. Weitere Komponenten wie die Sensoren werden über die 5V oder 3,3V Pins des BeagleBone Black versorgt.

## 2.5 Sicherheit

Das Fahrrad besitzt zwei höhenverstellbare Stützräder, welche vor Stürzen und Beschädigungen der Elektronik schützen. Der Code läuft auf einer SD-Karte, von welcher es ein Image gibt, um diese bei Beschädigung austauschen zu können. Des weiteren befinden sich auf dem Fahrrad zwei Not-Schalter, einer um die Lenkung auszuschalten und einer für die gesamte Elektronik. Es befindet sich eine Leine am Fahrrad, welche an einem Zug-Schalter angebracht ist um bei einem Ruck der aufsichtsführenden Person den Hinterradmotor direkt auszuschalten. Bei Empfangsverlust der Fernbedienung führt dies automatisch zu Programmabbruch. Es gab in der Vergangenheit Probleme im Code, welche dafür gesorgt haben, dass sich der Enable und PWM-Wert an der Lenkung nicht verändert haben, wodurch sich das Lenkrad so stark gedreht hat, dass die Bremsleitung abgerissen ist. Um diesem vorzubeugen wurde eine ‚Reißleine‘ angebracht, welche bei Verbindungsabbruch die Enable Pins des BTS7960 2.2.3 auf GND zieht und somit hardwareseitig die Rotation des Lenkrads abschaltet. Die maximalen Geschwindigkeiten für Hinterrad und Lenkungsmotor können im Code durch Anpassung der jeweiligen Controller Werte begrenzt werden.

## 2.6 Fähigkeiten des Gesamtsystems

Dieses Gesamtsystem kann also den Neigungswinkel und auch die Richtung des Fahrrads bestimmen. Es kann die Stellung des Lenkrads und die Geschwindigkeit des Hinterradnabenmotors ermittelt werden. Das Fahrrad kann beschleunigt und das Lenkrad bewegt

werden. Es kann mittels einer Fernbedienung gesteuert und mit Not-Ausschaltern abgeschaltet werden. Somit kann mit diesem System eine Steuerung entworfen werden, welche das Fahrrad fortbewegen und balancieren lässt.

## 2.7 Fehlende Funktionalität

Das Fahrrad lässt sich nicht mittels seiner Komponenten bremsen und muss entweder von Hand aufgehalten werden oder ausrollen. Der Antriebsmotor könnte das Fahrrad auch bremsen, diese Funktion ist aber nicht umgesetzt. Der Magnetometer, den der BNO055 bietet, wird nicht verwendet und könnte für bessere räumliche Orientierung genutzt werden.

# 3 Methodik

## 3.1 Programmiersprache

Die Software ist in der C Version geschrieben, die durch den GCC Compiler gegeben ist. Der Code wird aktuell mit GCC Version (Debian 8.3.0-6) 8.3.0 kompiliert. C zeichnet sich vor allem durch seine Effizienz und Performanz aus und wurde deshalb auch für dieses Projekt gewählt. Der Code verbraucht wenig Speicherplatz und zeitintensive Operationen wie das Kompilieren zur Laufzeit oder Garbage Collection gibt es in C nicht. Speicher lässt sich direkt mit Adressen und Pointern manipulieren, wodurch der Programmierer viel Kontrolle über die Hardware hat. All das macht C zu einer idealen Programmiersprache für hardwarenahe und schnelle Programme, wie sie bei zeitsensitiven Anwendungen wie einem selbstbalancierenden Fahrrad benötigt werden. Für das Kompilieren des Codes wird ein Makefile verwendet, welches mit CMake erstellt wurde. CMake vereinfacht den Build-Vorgang und sorgt dafür, dass Änderungen an Dateien leichter vorgenommen werden können. Weitere Informationen zur Verwendung von CMake sind im Anhang unter A.1 gegeben.

Um Code für die PRU's schreiben zu können, werden die Texas Instruments Code Generation Tools verwendet. Sie beinhalten unter anderem den Compiler, welcher nach dem C89 oder C99 Standard geschriebenen C-Source-Code zu Assembly übersetzt, sowie einen Assembler und einen Linker. Interessant anzumerken ist dabei, dass sich Assembler-Code für die PRU nur in Zusammenhang mit einem C-Programm schreiben lässt, welches als Container für den Assembler Code fungiert.

## 3.2 Anforderungen an den Code

Der Code wurde, wie in C vorgesehen, in einzelnen Übersetzungseinheiten implementiert. Die Aufteilung wurde so gewählt, dass sich logische Blöcke ergeben, die aufeinander aufbauen. Das erhöht die Wartbarkeit und vermindert Code-Wiederholungen.

### 3.3 PRU

Im Folgenden soll die Programmierung der PRU mit dem Beaglebone erläutert werden. In diesem Projekt wurden die PRU Code Generation Tools von Texas Instruments in der Version 2.1.5 verwendet. Die Version kann mittels `clpru --compiler-revision` abgefragt werden. Um Code für die PRU zu schreiben, muss zunächst ein Ordner erstellt werden, in welchem die Dateien `resource_table_empty.h` und `AM335x_PRU.cmd` abgelegt werden müssen. Als Makefile wurde eines verwendet, welches durch Texas Instruments mit den Code Generation Tools bereitgestellt wurde und durch Derek Molly angepasst wurde. Dieses Makefile übernimmt die Codeübersetzung und auch die Installation des Codes auf der PRU.

Um die PRU zu starten oder zu stoppen muss zuerst in das Verzeichnis `/sys/class/remoteproc/remoteproc1` gewechselt werden. Wobei `remoteproc1` für PRU 0 ist und `remoteproc2` für PRU 1 vorgesehen ist. In diesem Verzeichnis kann mittels `cat state` abgefragt werden, ob die PRU gerade läuft (`running`) oder abgeschaltet (`offline`) ist. Mittels `echo "start" > state` und `echo "stop" > state` wird die Ausführung des PRU Codes dann entweder gestartet oder abgebrochen. Damit die GPIO-Pins der PRU ausgelesen werden können, müssen diese zuvor mittels `config-pin` aktiviert werden.

Um den ausführbaren Code zu generieren, muss im Arbeitsverzeichnis `make` aufgerufen werden. Mit `sudo make install_PRU0` kann der Code auf die PRU geladen werden. Die Make-Datei geht hier davon aus, dass die jeweilige PRU bereits läuft. Der Prozess schlägt fehl, falls sich diese im ausgeschalteten Zustand befindet. In diesem Fall kann die PRU wie vorher beschrieben in den ausführenden Zustand versetzt werden, um das Laden des Codes zu erlauben.

Während des Testens ist es hilfreich die Registerwerte der PRU's abfragen zu können. Hierzu kann in das Verzeichnis `/sys/kernel/debug/remoteproc/remoteproc1` gewechselt werden und mit `sudo cat regs` werden die Inhalte der PRU-Register im Terminal angezeigt. Dies kann nur erfolgen, wenn die Ausführung des Codes auf der PRU gestoppt ist. Mit dem Assembler-Befehl `HALT` wird die Ausführung durch den Code selbst gestoppt. `Devmem2` ist ein weiteres hilfreiches Werkzeug, mit welchem der Wert einer Speicheradresse im Terminal angezeigt werden kann. Mit `sudo ./devmem2 0x4a300000` kann z.B. das Datum an der ersten Stelle im RAM der PRU0 angezeigt werden, wie in Abbildung 3.1 zu sehen ist.

### 3.3 PRU

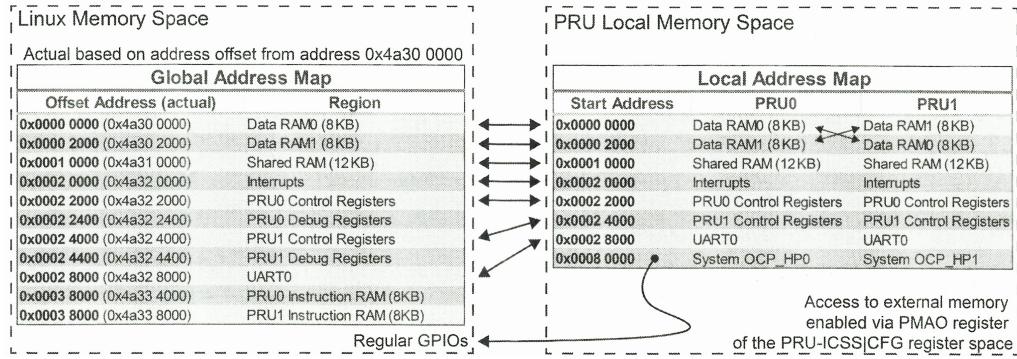


Abbildung 3.1: Memory-Map der PRU erstellt von Derek Molly [Moll19, Seite 692]

#### 3.3.1 Geschwindigkeits- und Distanzmessung

Wenn sich das Rad dreht und ein Magnet an dem Hall-Effekt-Sensor vorbeifährt, führt das zu einer Veränderung des digitalen Ausgangssignals. Das Signal wechselt zwischen den Spannungspegeln 0V und 3,3V. Die Aufgabe des Codes auf der PRU ist es, die Anzahl dieser Flanken in einer bestimmten Zeit und absolut zu zählen und diese dem ARM-Prozessor zugänglich zu machen. Damit kann die zurückgelegte Strecke und die Geschwindigkeit des Rads bestimmt werden. Diese Aufgabe kann nicht durch den BBB übernommen werden, da das System unter Debian nicht echtzeitfähig ist.

Folgender C Code dient lediglich als Container für den Assembler Code:

Listing 3.1: pru\_bike\_speed.c

```

1 #include <stdint.h>
2 #include <pru_cfg.h>
3 #include "resource_table_empty.h"
4
5 extern void start(void); //start is defined inside the
                           assembler code
6
7 void main(void)
8 {
9     start(); //call to the assembler function.
10 }
```

**Listing 3.2:** pru\_bike\_speed.asm

```

1 .cdecls "pru_bike_speed.c"
2 .clink
3 .global start
4
5 start:
6 LDI32 r3, 0x00000000
7 LDI32 r15, 0x02625a00 ; 0.2 s
8 LDI32 r27, 0x00022000 ; PRU0 control register address
9 LDI32 r17, 0x00000000
10 LDI32 r11, 0x00000000
11 LDI32 r10, 0x00000000
12 LDI32 r13, 0x00008000 ; GPIO pin selection
13 LBBO &r0, r27, 0, 4 ; load control register
14
15 pulsecount:
16 SBBO &r11, r3, 0, 4 ; current pulse count to memory
17 ADD r17, r17, r11
18 SBBO &r22, r3, 4, 4
19 SBBO &r17, r3, 8, 4 ; total count to memory
20 LDI32 r11, 0x00000000 ; reset counter
21 LDI32 r22, 0x00000000 ; reset counter
22
23 CLR r0, r0, 3
24 SBBO &r0, r27, 0, 4 ; disable cycle counter
25 SBBO &r3, r27, 0xc, 4 ; reset cycle counter.
26 SET r0, r0, 3
27 SBBO &r0, r27, 0, 4 ; enable cycle counter
28
29 checkpulse:
30 ADD r22, r22, 1
31 QBEQ nopulse, r10, r31 ; check GPIO change
32 ADD r11, r11, 1 ; inc pulsecount
33 XOR r10, r10, r13
34
35 nopulse:
36 LBBO &r14, r27, 0xc, 4 ; store cycle count
37 QBGE pulsecount, r15, r14 ; new pulsecount if 0,2s elapsed
38 JMP checkpulse

```

- Zeilen 5-13: Es werden Startwerte in die Register geladen.
  - Register 15 entscheidet über welchen Zeitraum die Werte gemessen werden, bevor sie in den Speicher geschrieben werden. Hier  $2625a00_{\text{hex}} = 40000000_{\text{dec}}$  Takte bei 200MHz für einen neuen Wert alle 0,2 Sekunden.

- Im Register 0 wird der Wert des PRU0 Kontrollregisters gespeichert (s.u.).
- In Register 27 befindet sich die Adresse des PRU0 Kontrollregisters.
- Register 13 bestimmt den Eingangs-GPIO-Pin, der abgefragt werden soll.
- Zeilen 16-21: Hier werden die für den ARM Prozessor interessanten Werte in den Speicher geschrieben.
  - Zeile 16: In die Adresse 0x4a300000 wird die Anzahl der Pulse in 0,2s aus Register 11 geschrieben.
  - Zeile 18: In die Adresse 0x4a300004 wird die Anzahl der Schleifendurchgänge geschrieben um die Ausführungszeit zu messen.
  - Zeile 19: In die Adresse 0x4a300008 wird die gesamt Anzahl der Impulse geschrieben.
- Zeilen 23-27: Hier wird der Taktzähler zurückgesetzt. Zuerst muss der Zähler abgeschaltet werden, indem ein Bit im Steuerregister gesetzt wird. Der eigentliche Zähler wird mit einem Schreibbefehl in das Zählregister zurückgesetzt, und zuletzt wird der Zähler durch Setzen des Bits im Steuerregister wieder aktiviert.
- Zeilen 29-38: Es wird überprüft, ob sich der Wert des GPIO-Pins verändert hat und je nachdem der Zähler inkrementiert. Zum Schluss wird dann überprüft wie viel Zeit vergangen ist, und falls die eingestellte Zeit vorbei ist, werden die Werte gespeichert (siehe Zeilen 16-23), wonach das Zählen fortgesetzt wird.

Damit der Code ausgeführt werden kann, muss der hier verwendete GPIO-Pin aktiviert werden. Dies kann mittels des config-pin Tools mit `config-pin p8.15 pruin` erreicht werden.

#### 3.3.1.1 Genauigkeit dieser Messungen

Die Geschwindigkeit sollte möglichst schnell und akkurat gemessen werden können. Hier ergibt sich ein Konflikt für langsame Geschwindigkeiten. Es wird 0,2s als Messintervall gewählt. Mit einem Raddurchmesser von ca. 0,629m und gezählten 46 Pulsen pro Umdrehung kann abgeschätzt werden, wie genau diese Geschwindigkeitsmessungen werden können. Für eine geringe Geschwindigkeit von 1m/s müssten innerhalb von 0,2s ungefähr 5 Pulse gemessen werden.

$$\frac{1 \frac{m}{s} * 0,2s}{\frac{0,629m*\pi}{46}} \approx 4,66$$

und für eine sehr hohe Geschwindigkeit von 15m/s

$$\frac{15 \frac{m}{s} * 0,2s}{\frac{0,629m*\pi}{46}} \approx 69,84$$

ungefähr 70. Ein Messdurchgang der PRU braucht 8 Taktzyklen, es könnten also 25M Pulse durch den Hall-Effekt-Sensor gemessen werden.

Der Fehler zwischen zwei Messungen

$$\frac{\frac{0,629m*\pi}{46}*(n+1)}{0,2s} - \frac{\frac{0,629m*\pi}{46}*n}{0,2s} = \frac{0,629m*\pi}{46*0,2s*2} \approx \pm 0,11 \frac{m}{s}$$

beträgt ca. 0,11m/s. Diese Abweichung könnte verringert werden durch ein kleineres Hinterrad oder durch längere Messzeiten. Für das Fahrrad ist diese Auflösung jedoch ausreichend, wobei eine längere Verzögerung zwischen Messwerten schlecht für Kontrollsleifen sein könnte.

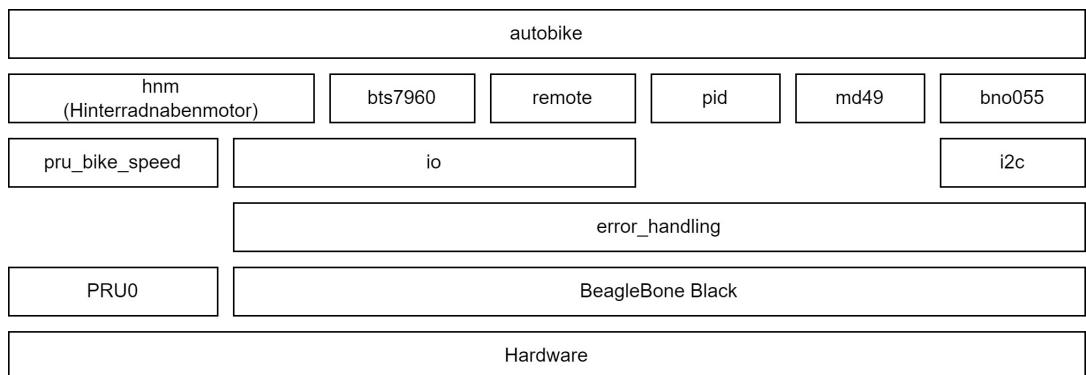
### 3.3.1.2 Erweiterbarkeit der Messung

Da zwei Hall-Effekt-Sensoren vorhanden sind, könnten auch beide ausgelesen werden, um z.B. eine Rückwärtsbewegung des Fahrradreifens zu bestimmen oder die Genauigkeit von Messungen zu erhöhen. Falls alternativ das System robuster gestalten werden sollte, könnte die andere PRU genutzt werden, um den zweiten Sensor auszulesen, um so die berechneten Daten untereinander abzugleichen und Fehler oder ein Versagen des Systems zu erkennen.

## 3.4 Modularisierung in C

Der Programmcode wurde in logisch zusammenhängende Übersetzungseinheiten aufgeteilt. Hier werden die selbstgeschriebenen Einheiten und wichtigsten darin enthaltenen Funktionen und deren Gebrauchsweise beschrieben.

Das gesamte Projekt befindet sich auf dem BeagleBone im Verzeichnis `/var/lib/cloud9/fahrradprojekt/autobike/`.



**Abbildung 3.2:** Übersicht über die einzelnen Übersetzungseinheiten im C Programm. Jeder Block ist jeweils von den darunterliegenden abhängig.

### 3.4.1 I2C

Dieses Modul wird für die Kommunikation mit dem BNO055 Sensor benötigt. Es bietet einfache Funktionalitäten, um mit Geräten über die i2c-Schnittstelle zu interagieren.

- `i2c_fd_t i2c_device_open(char* bus, long address)`

Mit dieser Funktion kann die Verbindung zu einem Gerät geöffnet werden. Dabei kann als Bus entweder "/dev/i2c-1" oder "/dev/i2c-2" verwendet werden. Die Adresse des Geräts hängt von dem i2c-Gerät ab. Der BNO055 hat z.B. entweder die Adresse 28<sub>hex</sub> oder 29<sub>hex</sub>, je nachdem ob ein Jumper gesetzt wird.

- `void i2c_device_close(i2c_fd_t fd);`

Diese Funktion schließt diese Verbindung wieder.

- `void i2c_device_write(i2c_fd_t fd, const uint8_t *data, int num_bytes);`

Hiermit können Daten an das i2c-Gerät geschickt werden. Eine Liste an 8 bit langen unsigned Integern dient dabei als Behälter für die zu sendenen Bytes. Das erste Element der Liste ist jedoch die Registeradresse des ersten Bytes und jedes weitere Byte wird an jede folgende Adresse geschickt. Die Funktion benötigt außerdem die Länge dieser Liste.

- `void i2c_device_read(i2c_fd_t fd, uint8_t *data, int num_bytes);`

Um Daten von einem i2c-fähigen Gerät zu empfangen, wird diese Funktion verwendet. Auch hier muss eine uint8-Liste angelegt werden, wobei das erste Element dieser Liste angibt, von welcher Startadresse aus gelesen werden soll. Falls das Gerät Burst-Reads unterstützt, kann die Liste länger als 2 Einträge sein.

### 3.4.2 BNO055

Dieses Modul implementiert einige Funktionen, um die Kommunikation zwischen dem BeagleBone Black und dem Sensor zu vereinfachen. Dabei werden die Funktionen des i2c-Moduls genutzt.

- `i2c_fd_t bno055_device_open(char* bus, long address, char *i2c_pin1, char *i2c_pin2);`

Öffnet die Verbindung mit dem BNO055. Hier müssen Geräteadresse und Beagle-Bone Black i2c-Bus angegeben werden, sowie die Pins für die i2c-Kommunikation, damit der Code sie mit Hilfe des config-pin Tools richtig konfigurieren kann.

- `void bno055_set_page0(i2c_fd_t fd);`

Da der BNO055 viele Register besitzt sind diese in 2 logische Seiten geteilt. Auf Seite 1 befinden sich Register zur Einstellung der Sensoren. Seite 0 befasst sich mit

generellen Konfigurationen und Ausgabedaten. In diesem Projekt werden lediglich Einstellungen auf der Seite 0 vorgenommen.

- `uint8_t bno055_get_mode(i2c_fd_t fd);`

Mit dieser Funktion wird der Betriebsmodus des BNO055 ausgegeben. Um den Betriebsmodus auslesen zu können muss sich der Sensor bereits auf Seite 0 befinden.

- `void bno055_set_mode(i2c_fd_t fd, uint8_t mode);`

Hiermit wird der Betriebsmodus des Sensors eingestellt. Die Funktion wählt dazu selbstständig zuerst Seite 0 des BNO055 an. Mögliche Modi sind in [Bosc21, Table 3-5] dargestellt. In diesem Projekt wird nur der Inertial Measurement Unit Modus (8<sub>hex</sub>) verwendet.

- `i2c_fd_t bno055_soft_reset(i2c_fd_t fd);`

Diese Funktion setzt das RST\_SYS-Bit im SYS\_TRIGGER-Register und löst somit einen Reset aus. Diese Funktion beinhaltet außerdem einen Wartebefehl, da sich gezeigt hat, dass der BNO055 mindestens 0,55 (!) Sekunden benötigt um wieder Messwerte liefern zu können.

- `void bno055_get_euler(i2c_fd_t fd, f_euler_degrees_t *euler, float angle_unit_factor);`

Hiermit werden Heading, Roll und Pitch als Winkel ausgelesen. Sie werden in einem struct gespeichert, dessen Adresse als Argument übergeben werden muss. Beim Auslesen dieser Werte werden die zugehörigen Register mittels Burst-Read gelesen, wodurch sichergestellt ist, dass sich alle Messwerte auf denselben Zeitpunkt beziehen. Um die Winkel im Gradmaß zu erhalten, müssen zuerst jeweils 2 Byte für jeden Wert als signed Integer kombiniert und danach noch durch einen Winkelfaktor geteilt werden. Dieser ist entweder 16 für das Gradmaß oder 900 für das Bogenmaß.

#### 3.4.2.1 Anmerkungen

Diese Code-Einheit könnte so angepasst werden, dass zur Kommunikation die serielle Schnittstelle UART anstelle von i2c verwendet wird. Der Sensor bietet beide Möglichkeiten an. Mit UART könnte eine schnellere Kommunikation erfolgen, aber da die kombinierten Sensorwerte lediglich mit 100Hz erneuert werden, würde dies im aktuellen Fall keinen Geschwindigkeitsvorteil ergeben. i2c wurde gewählt, da hier am Bus leicht ein weiterer Sensor angeschlossen und damit das System erweitert werden könnte. Dies könnte zusätzliche Daten zur Verfügung stellen oder die Ausfallsicherheit durch einen zweiten Sensor erhöhen.

#### 3.4.3 MD49

Die Funktionen in diesem Modul befassen sich mit der Kommunikation zwischen dem BeagleBone Black und dem MD49. Diese Kommunikation findet über UART statt. Für UART

wurde kein eigenes Modul angelegt, da die Kommunikation über UART in C größtenteils durch das Standard-Modul `termios` implementiert ist.

- `uart_fd_t md49_device_open(char* port, char *uart_pin1, char *uart_pin2);`

Mit dieser Funktion wird die Kommunikation mit dem MD49 gestartet. Wie bei i2c muss hier der Port für die Kommunikation (im Projekt `"/dev/ttyO4"`) angegeben werden und welche Pins dabei genutzt werden sollen. Andere Kommunikationsparameter wie die Anzahl der Stop- und Parity-Bits, sowie die Baudrate von 38400 Baud werden in dieser Funktion gesetzt. Sollten sich diese Kommunikationsparameter ändern, wenn z.B. mit einem Jumper die Baudrate auf 9600 reduziert wird, müssen sie in dieser Funktion geändert werden.

- `void md49_device_write(uart_fd_t fd, uint8_t command);`

Diese Funktion übernimmt das Senden von Befehlen die genau ein Byte lang sind. Sie werden automatisch, wie von der Spezifikation verlangt, mit einem 0 Byte eingeleitet. Befehle, welche 2 Bytes benötigen, werden in diesem Projekt nicht verwendet. Diese wären z.B. das Setzen von Geschwindigkeit oder Beschleunigung, was in diesem Projekt jedoch durch den BTS7960 ausgeführt wird.

- `void md49_device_read(uart_fd_t fd, uint8_t *data, int data_size);`

Antworten des Boards können mit dieser Funktion gelesen werden. Die Länge der Antworten reicht von 0 bis 8 Byte. Die Anzahl an Antwortbytes muss mit angegeben werden, sowie eine Liste an `uint8`, welche als Behälter für die einzelnen Bytes dient.

- `void md49_device_reset(uart_fd_t fd);`

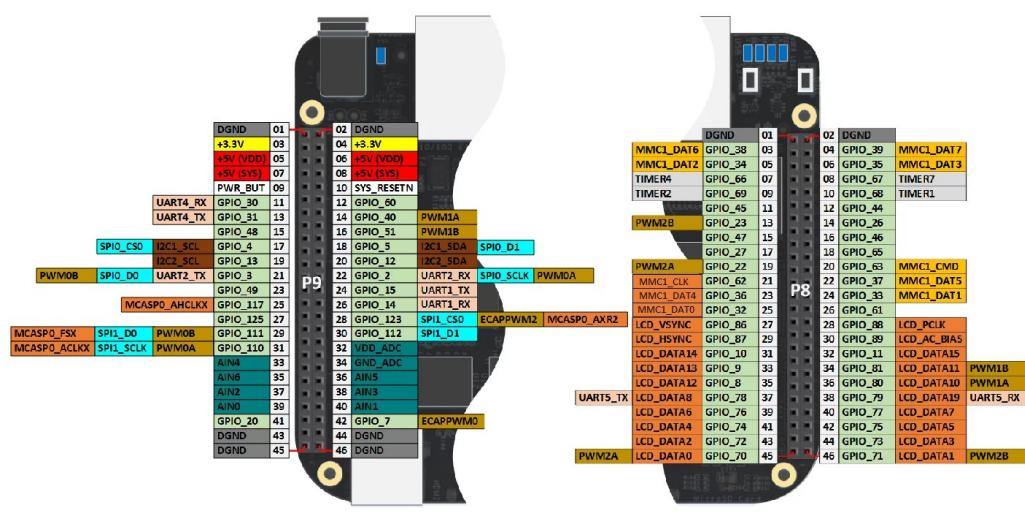
Das Zurücksetzen der Encoder Werte erfolgt durch diesen Befehl.

- `int32_t md49_get_encoder1(uart_fd_t fd);`

Die am häufigsten verwendete Funktion dieses Moduls sendet einen Befehl an den MD49, um den Wert des Encoders auszulesen. Der Encoder-Wert setzt sich aus 4 Bytes zusammen (das höchstwertige Byte zuerst) und wird als vorzeichenbehafteter Integer interpretiert und von der Funktion zurückgegeben. Falls die Hall-Effekt-Sensoren am Eingang des zweiten Encoders angeschlossen würden, müsste die Funktion dementsprechend angepasst werden.

#### 3.4.4 IO

Dieses Modul befasst sich mit GPIO, Analog-Eingängen und PWM-Ausgängen des BeagleBone.



**Abbildung 3.3:** Übersicht über die Funktionen und Benennungen der Pins des BeagleBone Black.  
Quelle: <https://microcontrollerslab.com/wp-content/uploads/2019/12/Beaglebone-Black-pinout-diagram.png>

#### 3.4.4.1 PWM-Ausgänge

- void pwm\_open(pwm\_fd\_t \*fd, char \*pwm\_chip, char \*pwm\_channel,  
char \*pwm\_pin, int pwm\_period);

Wie in der Abbildung 3.3 zu sehen gibt es acht verschiedene PWM Pins (ockerfarben). Der PWM-Chip und Channel, durch welchen das Signal erzeugt werden soll, müssen in dieser Funktion angegeben werden. Außerdem wird der Pin, durch den das Signal nach außen geführt wird, bestimmt. PWM0A kann z.B. über den P9.22 oder P9.31 Pin verwendet werden. Anstelle der Frequenz wird die Periodendauer in Nanosekunden angegeben. Für eine Frequenz von 10kHz wird also ein Integer vom Wert 100000 gegeben. Mit diesem Befehl wird die Generierung eines Signals auch gestartet, aber das Tastverhältnis liegt noch bei 0%.

- `void pwm_disable(pwm_fd_t *fd);`

Mit diesem Befehl wird das Tastverhältnis auf 0% gesetzt und die Generierung des PWM Signals gestoppt.

- void pwm\_enable(pwm\_fd\_t \*fd);

Hier wird mit der Generierung des Signals fortgefahren. Meist ist das Tastverhältnis jedoch noch 0%, da dieses mit `pwm.disable` so gesetzt wird.

- void pwm\_set\_duty\_cycle(pwm\_fd\_t \*fd, int duty\_cycle);

Diese Funktion wird genutzt um das Tastverhältnis zu setzen. Dies erfolgt durch die Angabe an Nanosekunden, in der das Signal auf High steht. Ist diese Zahl größer als die Periodendauer oder genau die Periodendauer, entspricht dies einem Tastverhältnis von 100%.

### 3.4.4.2 GPIO

- `gpio_fd_t gpio_open(char *gpio_pin, char *gpio_direction);`  
Ein Pin kann mittels des `gpio_direction` Parameter entweder als "in" oder als "out" Pin konfiguriert werden. Die Angabe, welcher Pin genutzt wird, unterscheidet sich von der des PWM-Pins. Für die Aktivierung eines GPIO-Pins werden die in der Abbildung 3.3 hellgrünen Nummern der GPIO Pins als String gegeben.
- `void gpio_enable(gpio_fd_t *fd);`  
Falls es sich um einen OUT-Pin handelt wird das Signal an diesem auf HIGH gesetzt.
- `void gpio_disable(gpio_fd_t *fd);`  
Falls es sich um einen OUT-Pin handelt wird das Signal an diesem auf LOW gesetzt.
- `int gpio_in_read(gpio_fd_t *fd);`  
Falls es sich um einen IN-Pin handelt wird der Wert, welcher an diesem anliegt, entweder als 1 (HIGH) oder 0 (LOW) ausgelesen.

### 3.4.4.3 Analog-Eingänge

- `adc_fd_t adc_open(char * adc_pin);`  
Um einen Analog-Eingang benutzen zu können, muss er mittels dieser Funktion ausgewählt werden.  
Bei den in der Abbildung 3.3 dunkelgrün markierten Pins handelt es sich um die Analog-Pins. Der `adc_pin` Parameter wählt dabei den jeweiligen Eingang aus. Dabei kann von "0" bis "6" Jeder Pin gewählt werden.
- `int adc_get_raw(adc_fd_t *fd);`  
Diese Funktion gibt einen Wert als Integer mit einer Genauigkeit von 12 Bit zurück.

## 3.4.5 BTS7960

Dieses Modul verwendet viele der Funktionen des IO-Moduls und übernimmt die Steuerung des BTS7960 Boards. Dieses Board benötigt für die Steuerung des Lenkmotors 2 PWM Signale und 2 Enable Signale.

- `void bts7960_device_open(bts7960_fd_t *fd, char *left_enable_pin, char *right_enable_pin, char *left_pwm_chip, char *left_pwm_channel, char *left_pwm_pin, char *right_pwm_chip, char *right_pwm_channel, char *right_pwm_pin);`  
Um das Board benutzen zu können müssen alle Parameter für die PWM und GPIO Pins gesetzt werden. Da im Projekt die GPIO enable Pins immer gleichzeitig an- oder ausgeschaltet werden, können für den right- und den left\_enable\_pin derselbe Pin angegeben werden.

- `void bts7960_turn(bts7960_fd_t *fd, int speed);`

Diese Funktion bekommt einen Wert zwischen -100000 und 100000. Dieser Wert gibt die Geschwindigkeit und Richtung der Rotation an. Ein positiver Wert bewegt den Lenker nach rechts und ein negativer nach links. Falls sich die Rotationsrichtung verändert, werden die entsprechenden PWM-Pins so geschaltet, dass nur entweder der linke oder der rechte PWM-Wert aktiviert ist.

### 3.4.5.1 Anmerkung

Diese Funktion wird in einer Schleife zusammen mit den Encoder Werten verwendet, welche der MD49 liefert. Es ist vorgekommen, dass der MD49 keinen neuen Encoder Wert geliefert hat. Aufgrund der fehlenden Echtzeitfähigkeit des Betriebssystems hat sich auch ein softwareseitiger Ansatz, welcher die Zeit der letzten Messung überwacht, als unzureichend erwiesen. Deshalb wurde eine „Reifleine“ am Fahrrad angebracht, welche bei einem Fehler das GPIO Enable-Signal auf GND zieht und die Rotation somit hardwareseitig stoppt.

### 3.4.6 Fernbedienung

Dieses Modul nutzt die Analog-Eingänge, welche durch das IO-Modul nutzbar gemacht werden. Durch die Nutzung von RC-Tiefpass Filtern um die PWM-Signale in Spannungen zu wandeln, geht Auflösung verloren und das Signal verspätet sich. Die Aufgabe dieses Moduls ist diese Spannung zu messen und die Stellung des Lenkrads und des Gashebels der Fernbedienung auszugeben.

- `remote_fd_t remote_open(char* adc_pin_1, char* adc_pin_2);`

Es müssen zu Beginn wie in Analog-Eingänge 3.4.4.3 beschrieben, die zu nutzenden Eingangspins angegeben werden.

- `int remote_get_steering(remote_fd_t *fd);`

Das Ergebnis dieser Funktion ist eine Zahl, welche angibt, ob die Fernbedienung überhaupt verbunden ist und ob das Lenkrad nach rechts, links oder gar nicht eingeschlagen ist. Dabei steht 0 für gerade, 1 für links und 2 für rechts. -1 steht hier dafür, dass die Fernbedienung nicht verbunden ist. Die Grenzwerte an denen z.B. eine Linksneigung gegenüber der geraden Fahrt zu erkennen ist, müssen hier eingestellt werden.

- `int remote_get_throttle(remote_fd_t *fd);`

Diese Funktion arbeitet auf gleiche Weise, wobei 0 für keine Beschleunigung, 1 für vorwärts und 2 für rückwärts steht.

### 3.4.6.1 Anmerkung

Durch die Verwendung einer PRU um die aktive Zeit des PWM-Signals zu messen, könnte die Genauigkeit der Fernbedienung gesteigert werden. Im Moment wird eine höhere Genauigkeit jedoch nicht benötigt.

### 3.4.6.2 Hinterradnabenmotor

Das Modul hnm.c befasst sich mit der Ansteuerung des Hinterradnabenmotors, sowie dem Auslesen der Rotationsgeschwindigkeit des Motors. Es ist funktionell in diese zwei Teile geteilt, welche unabhängig von einander genutzt werden können.

Die Ansteuerung des Motors erfolgt mittels PWM, wodurch auch hier die Funktionalität auf der des IO-Moduls beruht:

- `void hnm_device_open(hnm_fd_t *fd, char *pwm_chip,  
char *pwm_channel, char *pwm_pin);`

Es muss die Angabe des verwendeten PWM-Pins und Chips erfolgen wie in 3.4.4.1 PWM-Ausgänge beschrieben. Der Hinterradnabenmotor verlangt eine Taktfrequenz von 10kHz. Diese wird durch diese Funktion standardmäßig gesetzt.

- `void hnm_disable(hnm_fd_t *fd); und void hnm_enable(hnm_fd_t *fd);`  
Unabhängig vom Setzen des Tastverhältnisses kann die Signalgenerierung deaktiviert und aktiviert werden. Bei der Deaktivierung wird automatisch das Tastverhältnis auf 0% gesetzt, damit es bei unvorsichtiger Aktivierung nicht zu Problemen kommt.
- `void hnm_set_pwm(hnm_fd_t *fd, int speed);`  
Es kann ein Geschwindigkeitswert von 0 (keine Bewegung) bis 100000 (maximale Geschwindigkeit) gesetzt werden. Bei einem Wert von weniger als 30000 bewegt sich das Rad jedoch nicht, da die dann anliegende Spannung nicht groß genug ist.

Die folgenden Funktionen sind abhängig von dem für die PRU geschriebenen Code.

- `void hnm_enable_speed_reading(hnm_fd_t *fd);`

Diese Funktion ist sehr auf den PRU Code zugeschnitten. Der PRU-Input-Pin wird aktiviert und die PRU0 wird gestartet. Dafür muss sich der für die Geschwindigkeitsmessung zuständige Code bereits auf der PRU befinden. Es wird außerdem Speicher der PRU0 in den virtuellen Speicher des Prozesses gemappt, um auf diesen zugreifen zu können. Dazu müssen die Zieladressen, in welcher der Zählerstand für die Geschwindigkeit und die zurückgelegte Distanz des Hinterrades gespeichert werden, bekannt sein.

- `float hnm_speed(hnm_fd_t *fd);`

Mit dem Raddurchmesser, dem Zählintervall und dem Zählerstand wird in dieser

Funktion auf den Speicher zugegriffen, der Wert als unsigned long interpretiert und dann die Geschwindigkeit in m/s zurückgegeben.

- `float hnm_distance(hnm_fd_t *fd);`

Die Distanz wird analog zur Geschwindigkeit ermittelt und zurückgegeben.

Voreingestellte Parameter sind:

- Durchmesser des Hinterrads.
- Anzahl Pulse die durch den Hall-Effekt Sensor bei einer Umdrehung gesendet werden.
- Die verwendete PRU und der verwendete PRU-Input-Pin.
- Die Speicheradressen der Zählwerte für Geschwindigkeit und Distanz.
- Das Zeitintervall in dem der Geschwindigkeitswert aufgenommen wurde.

Falls sich diese Werte verändern, müssen sie im Code entsprechend angepasst werden und das Projekt neu kompiliert werden.

#### 3.4.7 PID Kontroller

Sowohl der Hinterradnabenmotor als auch der Lenkmotor werden mittels eines PID-Kontrollers gesteuert. Da sich die Zeiten zwischen den Berechnungen der Kontrollerwerte nicht voraussagen lassen, ist der Messzeitpunkt ein sehr wichtiger Bestandteil dieses Kontrollers.

- `pid_val_t pid_open(float Kp, float Ki, float Kd, float output_min, float output_max, float integral_min, float integral_max);`

Mit dieser Funktion wird der Regler mit seinen Startvariablen initialisiert.

- $K_p$ ,  $K_i$ ,  $K_d$  sind die Konstanten, für den Proportional-, Integral- und Ableitungsterm.
- $output\_min$ ,  $output\_max$  begrenzen den Ausgabewert des Reglers.
- $integral\_min$ ,  $integral\_max$  begrenzen die möglichen Werte des Integralterms.

- `float pid_output(pid_val_t *pid, float setpoint, float process_variable, long long current_time);`

Für die Berechnung des nächsten Steuerwerts muss der Zeiger des `pid_val_t` structs übergeben werden. Außerdem muss der Soll- und Ist-Wert der zu steuernden Einheit übergeben werden, sowie der Zeitpunkt zu welchem diese Werte ermittelt wurden. Der Zeitwert ist in Mikrosekunden seit Programmstart anzugeben.

Bei der Berechnung des Proportionalterms wird der Fehlerterm ( $Fehlerterm = Sollwert - Istwert$ ), mit dem  $K_p$  Faktor multipliziert.

Für die Berechnung des Integralterms wird zum bestehenden Integralterm der Wert einer Iteration hinzugefügt. Der Wert einer Iteration wird in jedem Durchlauf durch die Multiplikation des  $K_i$  Faktors mit dem Fehlerterm und der Zeitdifferenz zwischen dem aktuellen und dem letzten Durchlauf berechnet. Danach wird überprüft ob der Integralterm innerhalb der gesetzten Begrenzungen liegt und falls nötig angepasst.

Für die Berechnung des Ableitungsterms wird eine Liste an Fehlerterminen, Ableitungstermen und deren Berechnungszeiten gespeichert. Die Länge dieser Listen kann mittels des `HISTORY_LEN` Makro im Code verändert werden. Da der Ableitungsterm anfällig gegenüber schnellen Veränderungen ist, werden diese Listen verwendet um die Veränderung des Fehlerterms zwischen dem aktuellen und dem ältesten Eintrag in der Fehlerterm-Liste zu berechnen. Zusätzlich wird der Durchschnitt aus den letzten so bestimmten Werten berechnet, um die Anfälligkeit gegenüber schnellen Veränderungen weiter zu vermindern.

#### 3.4.8 Autobike

Die `autobike.c` Implementierungsdatei wird genutzt um das Fahrrad selbstbalancierend fahren zu lassen. Sie nutzt alle beschriebenen Code-Einheiten. Die Abhängigkeiten der einzelnen Einheiten lassen sich in Abbildung 3.2 sehen, deren Aufgaben in Abbildung 3.4. Es werden drei Regelschleifen gleichzeitig ausgeführt. Eine für die Steuerung des Hinterradmotors, eine für den Lenkmotor und eine für den Neigungswinkel des Fahrrads. Diese laufen in einzelnen Threads. Für die Steuerung des Hinterradmotors wird außerdem das Programm auf der PRU ausgeführt.

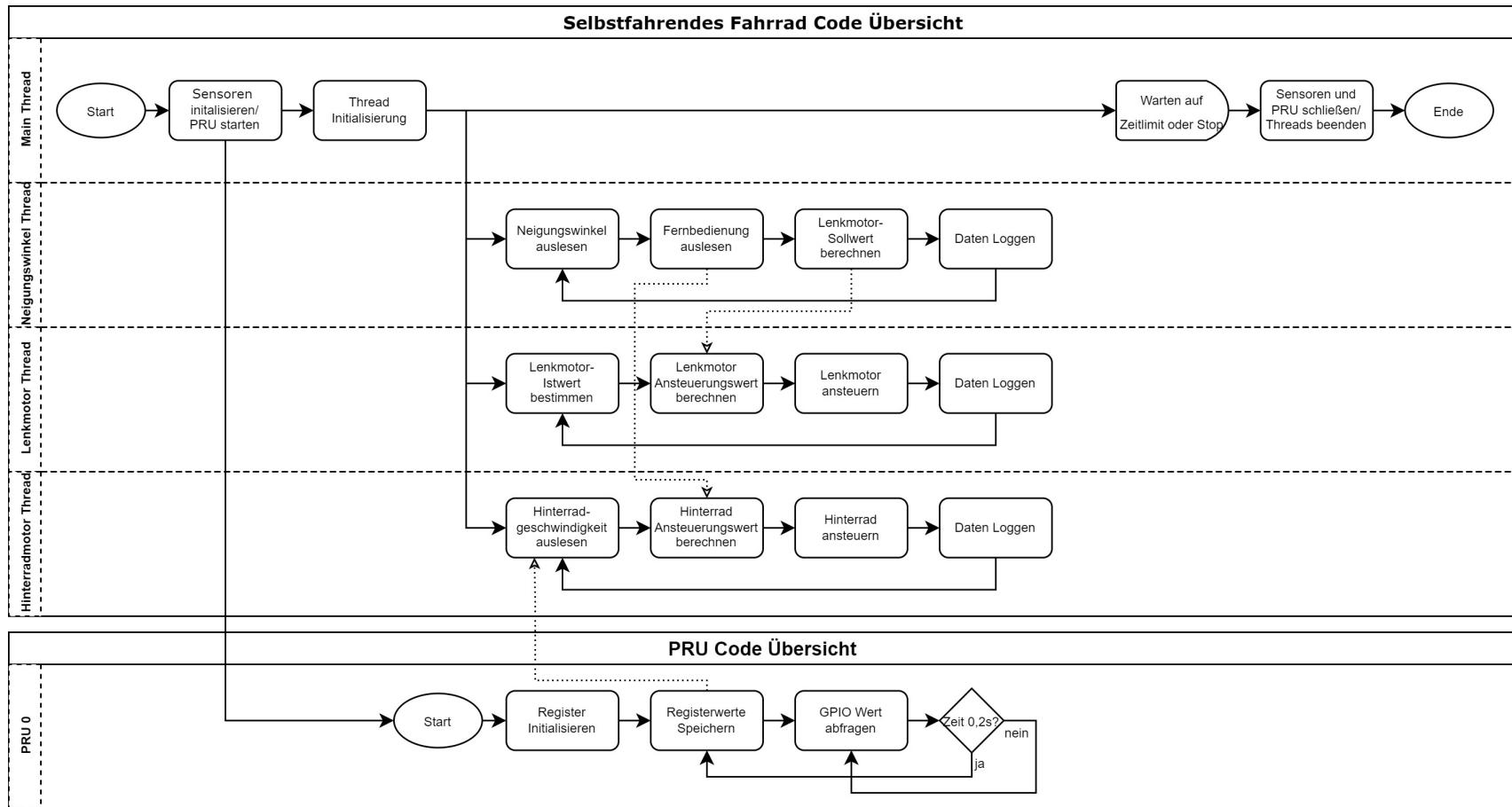
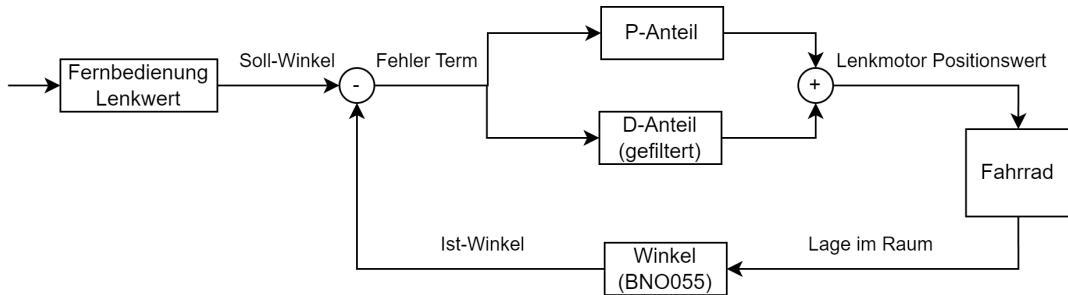


Abbildung 3.4: Übersicht über die Grundlegenden Aufgaben der Software. Die gepunkteten Linien zeigen die Datenabhängigkeiten zwischen den Threads und der PRU.

Obwohl der BeagleBone Black nur einen Ein-Kern-Prozessor besitzt, werden für die Ausführung der einzelnen Regelschleifen POSIX-Threads benutzt. Es wäre auch denkbar einen Codeansatz zu wählen, in welchem Schleifen unterschiedlich oft durchlaufen werden, um eine „parallele“ Ausführung des Codes zu erreichen. Es wurde jedoch der Thread-Ansatz gewählt, da die Struktur des Codes so deutlich einfacher lesbar und erweiterbar ist. Außerdem müssten in Schleifen ebenfalls Ausführungszeiten überwacht werden.

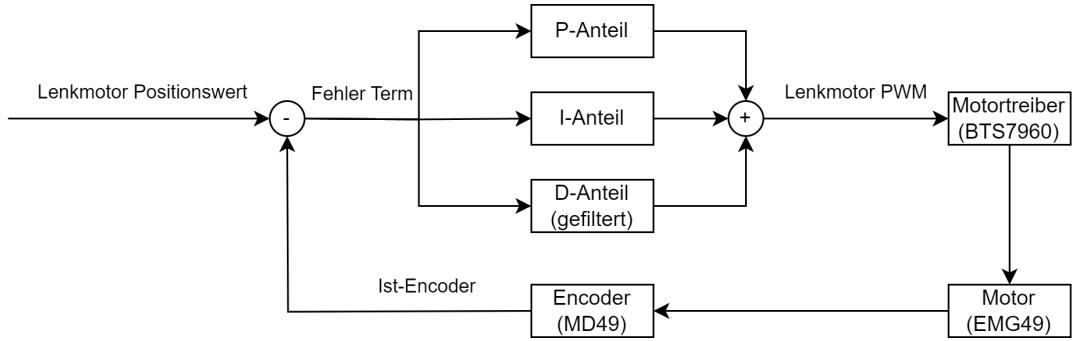
In jeder dieser Threads wird die `usleep()` Funktion verwendet, um die Ausführungs frequenz des jeweiligen Threads anzupassen.

Im Neigungswinkel-Thread werden die Steuerungswerte der Fernbedienung ausgelesen. Der Geschwindigkeits-Wert wird dem Hinterradmotor-Thread unter Verwendung eines MUTEX bereitgestellt. Der Stand der Fernbedienung wird bereits in diesem Thread ausgelesen, da der Lenkwert wichtig für die Bestimmung des Lenkmotor-Wertes ist. Der Lenkwert gibt den Soll-Winkel für die Fahrt des Fahrrads an, also ob sich das Fahrrad nach rechts, links oder geradeaus bewegen soll. Mittels eines PD-Reglers und des aktuellen Zeitstempels wird der neue Steuerwert berechnet, welcher dem Lenkmotor-Thread ebenfalls durch ein MUTEX zur Verfügung gestellt wird. Da der Neigungswinkel durch den BNO055 mit einer Frequenz von 100 Hz aktualisiert wird, soll diese Kontrollsleife mit ähnlicher Frequenz durchlaufen werden:



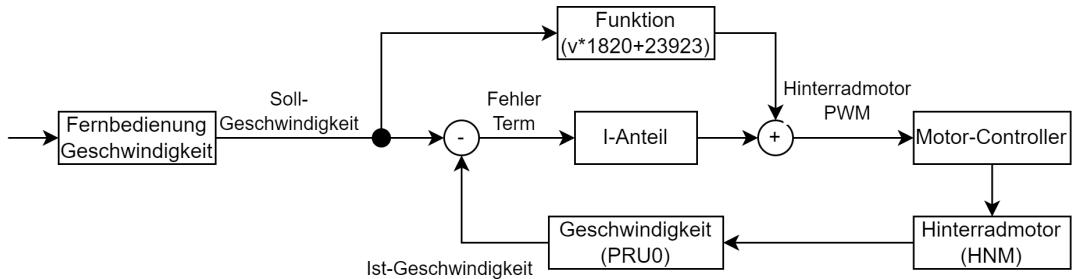
**Abbildung 3.5:** Übersicht über die Kontrollsleife für den Neigungswinkel.

Im Lenkmotor-Thread wird der so berechnete Soll-Lenkmotor-Encoder-Wert, angesteuert. Hierzu wird ebenfalls ein PID-Kontroller verwendet. Der Ist-Wert des Encoders wird durch den MD49 bereitgestellt. Dieser Regler läuft aktuell mit einer Frequenz von ungefähr 160Hz.



**Abbildung 3.6:** Übersicht über die Kontrollsleife für den Lenkmotor.

Der Thread für den Hinterradmotor erhält den Soll-Wert für die Geschwindigkeit durch den Geschwindigkeitswert der Fernbedienung, welcher durch den Neigungswinkel-Thread ausgelesen wird. Der Ist-Wert der Geschwindigkeit wird durch die PRU bestimmt. Mit dem Soll-Wert des Hinterradmotors wird ein PWM-Wert berechnet, welcher den Motor ansteuert. Um die Ansteuerung genauer zu machen, wird ein zusätzlicher I-Anteil auf diesen PWM-Wert hinzu addiert. Ein purer PID-Kontroller ist für diesen Motor nicht geeignet, da der Motor zu langsam ist. Für diesen Thread ist eine Ausführungs frequenz von 6Hz ausreichend.



**Abbildung 3.7:** Übersicht über die Kontrollsleife für den Hinterradmotor.

Threads, welche nicht in der Abbildung 3.4 aufgenommen sind, sind ein Print-Thread und ein Watchdog-Thread.

Der Print-Thread gibt in der Konsole einige Daten des Fahrrads aus, mit welchen bereits zur Laufzeit einige Parameter überwacht werden können. Da dies mit einer geringen Frequenz geschehen kann, gibt es hierfür einen eigenen Thread.

Der Watchdog-Thread wurde eingerichtet, da es bei der Kommunikation über UART und i2c dazu kommen kann, dass der BeagleBone für eine unbegrenzte Zeit auf eine Nachricht der jeweiligen Komponente wartet. Dieses Verfahren bietet keine Garantie, dass eine lange Wartezeit auch tatsächlich als Fehler gewertet wird. Da der Watchdog-Thread selbst nicht auf einem Echtzeitfähigem-System befindet. Hier wäre eine Implementation auf der PRU

sinnvoll. Dennoch ist die aktuelle Implementation als POSIX-Thread bereits hilfreich und hat in Tests die Programmausführung ab und zu aufgrund von Fehlern gestoppt. Dies war z.B. durch ein loses Kabel geschehen.

## 3.5 Logging

In den einzelnen Threads des autobike Programms werden die wichtigsten Daten für den jeweiligen Thread aufgenommen. Sie werden in Form einer .csv Datei gespeichert und sind benannt nach dem Namen des dazugehörigen Threads und dem Ausführungszeitpunkt. Das Erstellen der Logging-Dateien kann mittels des Makro `#define LOGGING` eingeschaltet werden. Die durch das Programm geschriebenen .csv Dateien werden im Ordner `autobike/logging/logging_files/` abgelegt.

Um die Daten automatisiert zu visualisieren wurde ein Script in Python (Version 3.7.3) geschrieben, welches mittels gnuplot (Version 5.2 patchlevel 6) die Datenpunkte visualisiert. In der Datei `plot.py` sind die Befehle für die Erstellung der jeweiligen Diagramme gespeichert.

Die erstellten Diagramme werden als .png Dateien im `autobike/logging/plots/` Ordner abgespeichert und die visualisierten .csv Dateien werden anschließend im `autobike/logging/logging_files_visualized` Ordner abgelegt.

Logging-Dateien und Diagramme sind in der Datei `.gitignore` aufgeführt, wodurch sie beim Commit nicht in die Versionskontrolle aufgenommen werden.

Informationen für das Erstellen der Diagramme befinden sich im Anhang A.2.

# **4 Tests und Evaluation**

Um das Testen des Fahrrads zu vereinfachen wird eine WLAN Verbindung über ein USB-Adapter zum BeagleBone hergestellt. Die Konfiguration des Adapters ist im Anhang A.3 beschrieben. Um die Logging Dateien einfacher über das Netzwerk verwalten zu können wurde auf dem Board außerdem Samba installiert. Informationen darüber, wie diese Software installiert und verwendet wird sind im Anhang unter A.4 zu finden.

## **4.1 Lenkmotor-Test**

Um den PID-Regler des Lenkmotors zu untersuchen und Parameter ( $K_p$ ,  $K_i$ ,  $K_d$ ) zu ermitteln, die gute Ergebnisse liefern, wurde der Test „steering\_pid\_test“ entwickelt. Dieser Test umfasst den md49-Thread und den main-Thread. Der Code ähnelt stark dem des „autobike“-Codes. Anstelle der Veränderung der Soll-Variable für die Lenkradstellung durch den bno055-Thread, geschieht dies nun durch eine Sequenz im Main-Thread. Die Tests werden durchgeführt und die aufgezeichneten Logging-Daten werden wie im „autobike“-Test mit dem Plot-Tool visualisiert.

Die folgenden Graphen wurden mit den folgenden Werten erstellt:

**Listing 4.1: PID-Werte**

```
1 steering_pid = pid_open(850.0, 300.0, 30.0, -100000, 100000,  
-20000, 20000);
```

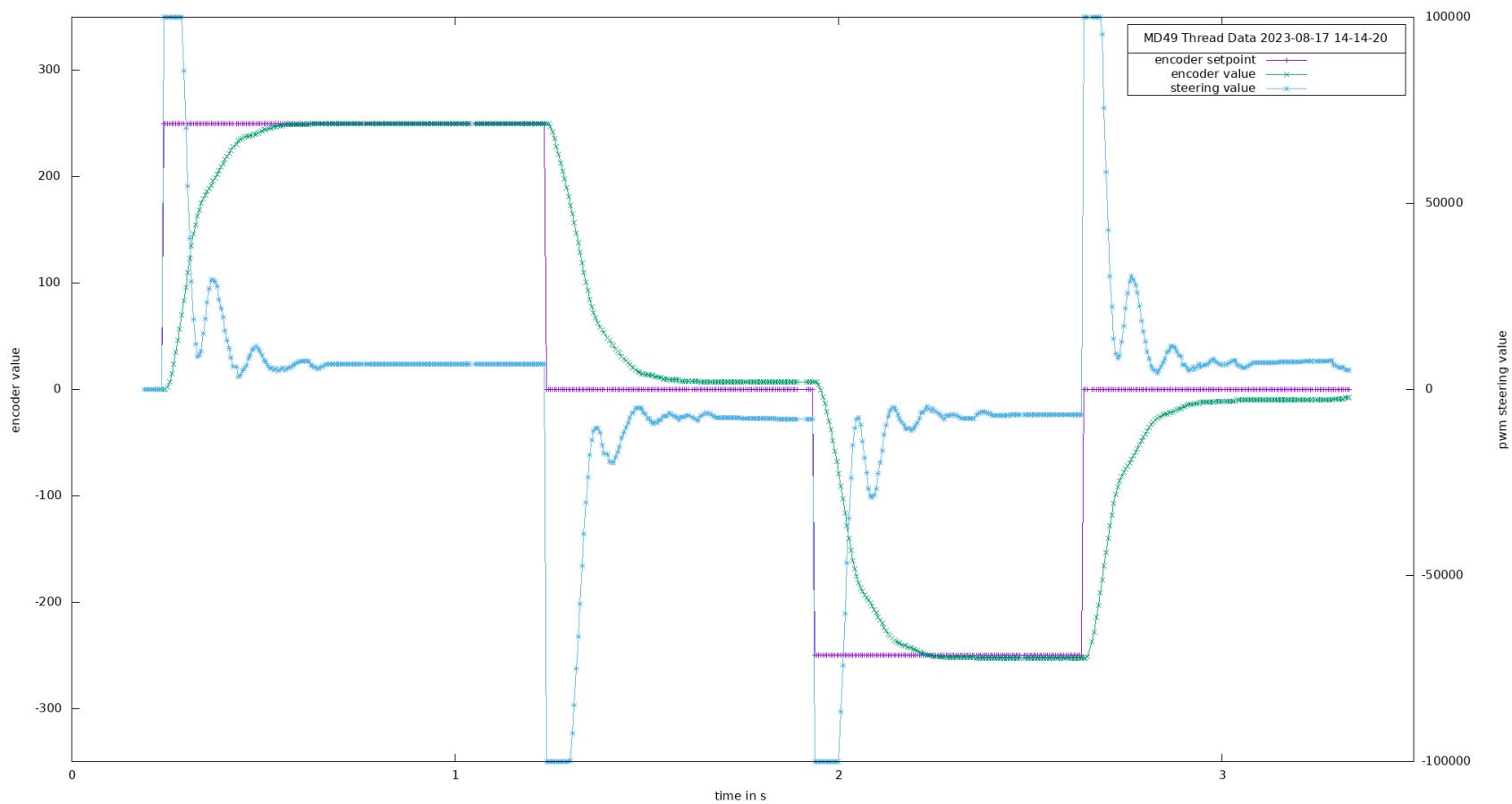
**Listing 4.2: Befehlsfolge 1**

```
1 md49_setpoint = 0;  
2 usleep(50000);  
3 md49_setpoint = 250;  
4 sleep(1);  
5 md49_setpoint = 0;  
6 usleep(700000);  
7 md49_setpoint = -250;  
8 usleep(700000);  
9 md49_setpoint = 0;  
10 usleep(700000);
```

**Listing 4.3: Befehlsfolge 2**

```
1 for (md49_setpoint=0; md49_setpoint<250; md49_setpoint++){  
    usleep(1000);}  
2 for (md49_setpoint; md49_setpoint>-250; md49_setpoint--){  
    usleep(1000);}  
3 for (md49_setpoint; md49_setpoint<0; md49_setpoint++){usleep  
(1000);}
```

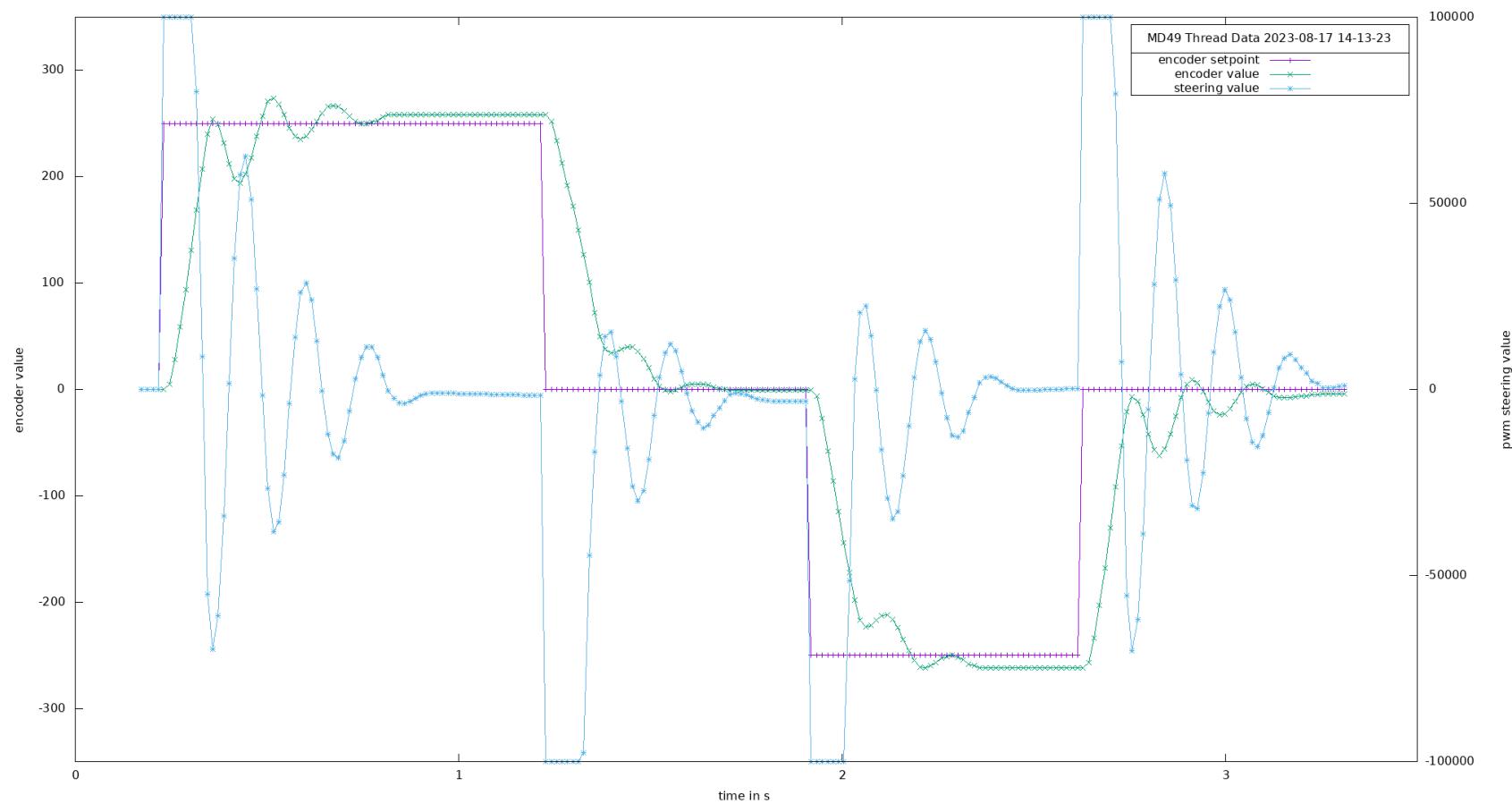
## 4.1 Lenkmotor-Test



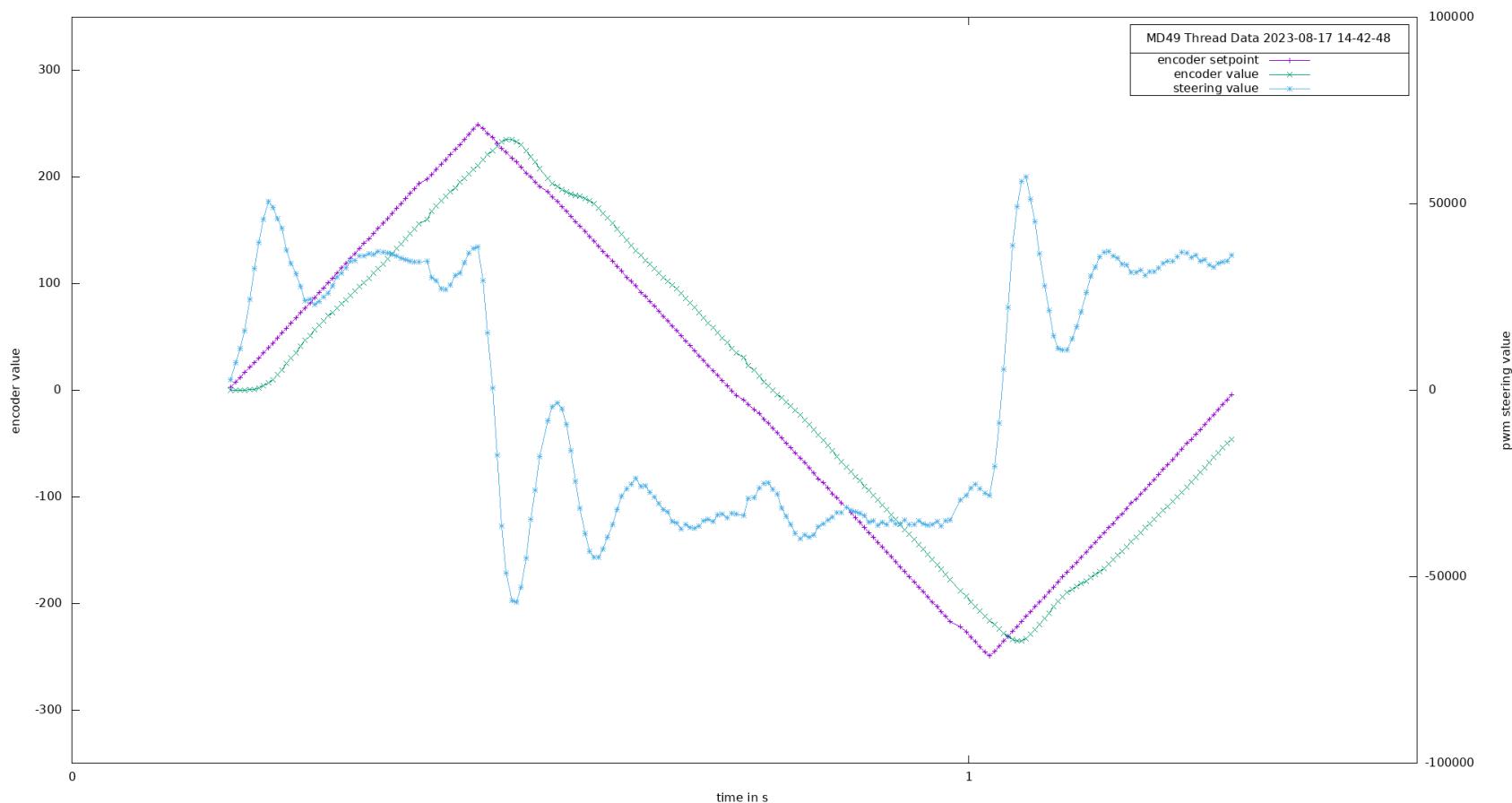
**Abbildung 4.1:** Lenkmotor Test ohne Last und geringer Schleifenverzögerung. Erstellt mit Befehlsfolge aus Listing 4.2. Durchschnittliche Zeit zwischen zwei Werten: 5279,7 $\mu$ s (189,4Hz). Standardabweichung: 1085,9 $\mu$ s.

## 4.1 Lenkmotor-Test

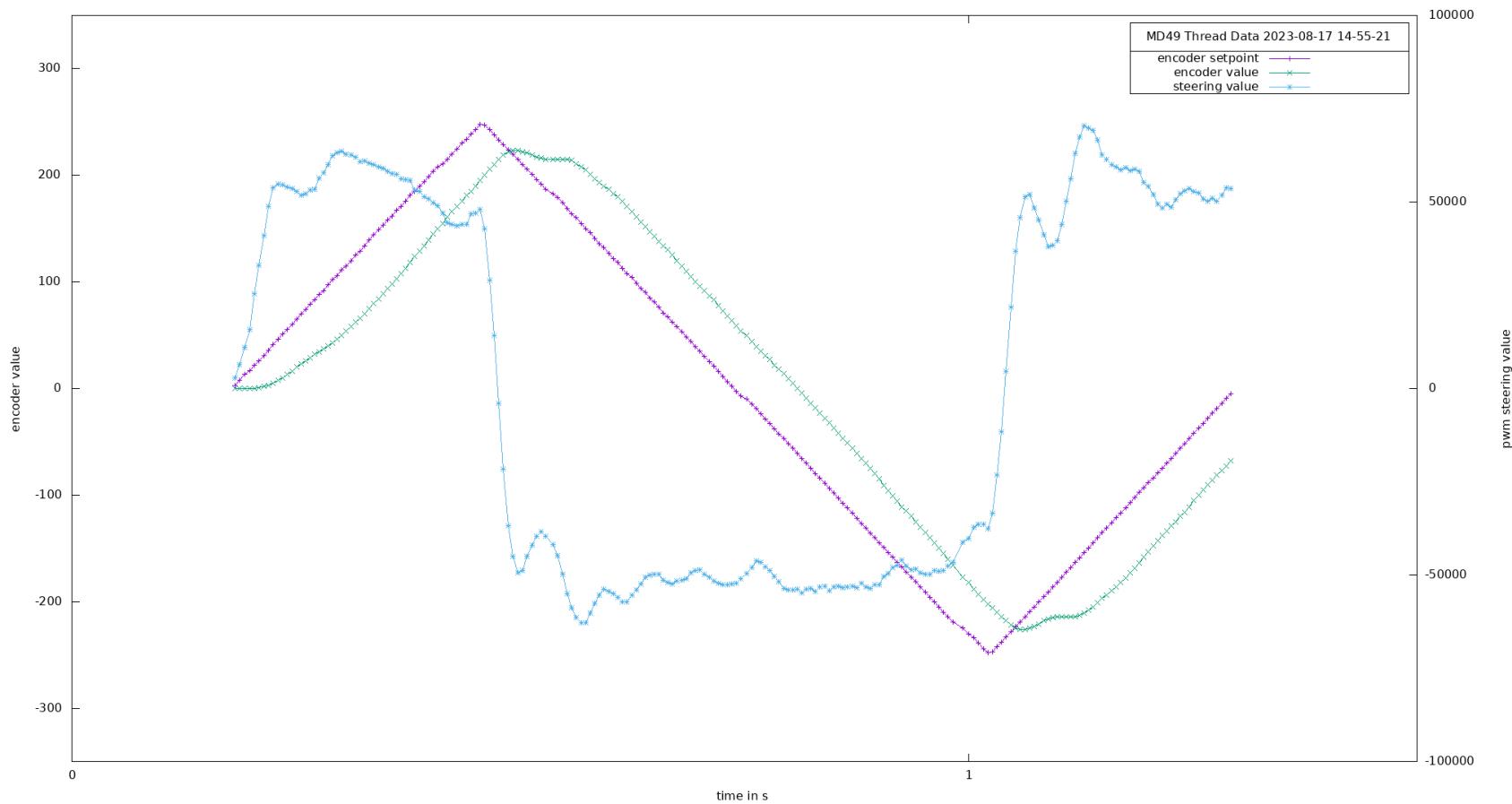
35



**Abbildung 4.2:** Lenkmotor Test ohne Last und hoher Schleifenverzögerung. Erstellt mit Befehlsfolge aus Listing 4.2. Durchschnittliche Zeit zwischen zwei Werten:  $14194,0\mu\text{s}$  (70,5Hz). Standardabweichung:  $195,8\mu\text{s}$ .



**Abbildung 4.3:** Lenkmotor Test ohne Last. Erstellt mit Befehlsfolge aus Listing 4.3. Durchschnittliche Zeit zwischen zwei Werten:  $5218,8\mu\text{s}$  (191,6Hz). Standardabweichung:  $658,1\mu\text{s}$ .



**Abbildung 4.4:** Lenkmotor Test mit Last. Erstellt mit Befehlsfolge aus Listing 4.3. Durchschnittliche Zeit zwischen zwei Werten:  $5188,7\mu\text{s}$  (192,7Hz). Standardabweichung:  $522,1\mu\text{s}$ .

Abbildungen 4.1 und 4.2 wurden beide mit der Befehlsfolge aus Listing 4.2 erstellt und ohne Verbindung zwischen Lenkrad und Motor. Da sich die anzustrebenden Lenkwerte stark verändern, sollte dieser Test vermieden werden, falls Motor und Lenkrad über einen Riemen verbunden sind, da dies zu hoher Belastung auf den Motor führen könnte. In den Abbildungen steht die lila Linie für den angestrebten Encoder-Wert und die grüne Linie für den tatsächlichen Encoder-Wert. Die hellblaue Linie repräsentiert den Wert, den der Regler als Ansteuerungswert an den Lenkmotor sendet.

In Abbildung 4.1 zeigt sich die Funktion des Reglers. Die durchschnittliche Zeit zwischen zwei Werten beträgt hier 5279 µs (mit einem Minimum von 4952 µs und einem Maximum von 25398 µs). In Abbildung 4.2 wurde die Wartezeit zwischen den Schleifendurchläufen erhöht, was zu einem Durchschnitt von 14194 µs (mit einem Minimum von 14020 µs und einem Maximum von 15529 µs) zwischen zwei Werten führt.

Es zeigt sich, dass der PID-Regler bei bestimmten Frequenzen am besten funktioniert, was wohl auf den D-Anteil zurückzuführen ist. Dieser D-Anteil kann aufgrund der Mittelwertbildung und der Betrachtung vergangener Werte nicht schnell genug auf Änderungen im Fehler reagieren.

Abbildungen 4.3 und 4.4 zeigen die Werte des Reglers bei Verwendung der Befehlsfolge in Listing 4.3. Dabei wird der anzustrebende Encoder-Wert kontinuierlich verändert. Der Unterschied zwischen den beiden Abbildungen besteht darin, dass in Abbildung 4.3 der Lenkmotor nicht mit dem Lenkrad verbunden ist, während er es in Abbildung 4.4 ist.

Ohne die zusätzliche Last des Lenkrads beträgt der durchschnittliche Fehler (Unterschied zwischen Soll- und Ist-Wert) 33, mit einem Maximum von 42. Bei Verwendung der Last, die durch das Lenkrad entsteht, ist eine deutliche Verschiebung der grünen Kurve nach rechts zu erkennen. Hier beträgt der durchschnittliche Fehler 51, mit einem Maximum von 67.

In beiden Abbildungen ist außerdem zu erkennen, dass die Ausführung dieser Kontrollsleife zu unregelmäßigen Zeiten erfolgt, da beispielsweise die lila Linie im Übergang zwischen dem Encoder Wert 250 bis -250 linear verlaufen müsste. Hier sind jedoch kleine Verzögerungen erkennbar.

### 4.1.1 Lenkwinkel

Um den Zusammenhang zwischen dem Encoder-Wert und dem Lenkwinkel zu bestimmen, wurden zwei Testreihen aufgenommen. Das Lenkrad wurde gerade ausgerichtet und der Encoder zurückgesetzt. Dann wurde das Lenkrad in einen 90°-Winkel zum Rest des Fahrrads in linker Richtung gedreht, und der Encoder-Wert aufgenommen. Danach wurde das Lenkrad im 90°-Winkel zum Fahrrad in die entgegengesetzte Richtung gebracht, und dieser Wert aufgenommen. Dies wurde fünf mal wiederholt. Dies analog für die andere Seite ebenso durchgeführt. Ergebnisse siehe Abbildung 4.5. Durch die Bildung eines

## 4.1 Lenkmotor-Test

---

Durchschnitts aller Werte wurde bestimmt, dass ein Schritt des Encoders ungefähr  $0,26^\circ$  Veränderung des Lenkwinkels entspricht.

Links dann Rechts			Rechts dann Links		
Durchgang	Encoder Wert Rechts	Encoder Wert Links	Durchgang	Encoder Wert Rechts	Encoder Wert Links
1	335	-358	1	369	-359
2	362	-323	2	342	-364
3	358	-338	3	328	-363
4	350	-344	4	329	-369
5	347	-336	5	325	-372

Durchschnitt		Durchschnitt	
350,4	-339,8	338,6	-365,4

**Abbildung 4.5:** Testreihen zu Lenkradstellung und Encoder-Werte.

### 4.1.2 Hysterese



(a) Zurücksetzen des Encoder-Wertes. (b) Lenkerstellung nach Auslenken und Zurücklenken des Lenkrads.

**Abbildung 4.6:** Test zur Hysterese durch die Verbindung von Motor und Lenkrad über einen Riemen.

Der Riemen zwischen Lenkrad und Motor verursacht Fehler. In diesem Versuch wurde das Lenkrad zunächst nach rechts gesteuert und anschließend in die mittlere Position zurückgeführt. Hier wurde der Encoder-Wert auf 0 zurückgesetzt (zu sehen in Abbildung 4.6a). Daraufhin wurde das Lenkrad vollständig nach links eingeschlagen und solange zurückgedreht, bis der Encoder-Wert wieder 0 erreichte. Wie in Abbildung 4.6b zu erkennen, unterscheidet sich die Stellung des Lenkrads trotz gleichem Encoder-Wert deutlich.

Es macht einen Unterschied aus welcher Richtung der gewünschte Punkt angesteuert wird.

## 4.2 Hinterradnabenmotor Test

Um das Hinterrad anzusteuern wurden zuerst PWM-Testwerte an den Motor geschickt und die Geschwindigkeit aufgezeichnet. Das Rad hatte dabei keinen Kontakt mit dem Boden.

PWM [ns]	Geschw. [m/s]
20000	0
22000	0
24000	0
26000	0,7
28000	1,8
30000	3,1
32000	4,4
34000	5,7
36000	7,2
38000	8,3
40000	9,6
42000	10,5
44000	11,3
46000	12,2
48000	13,1
50000	13,7
52000	14,4

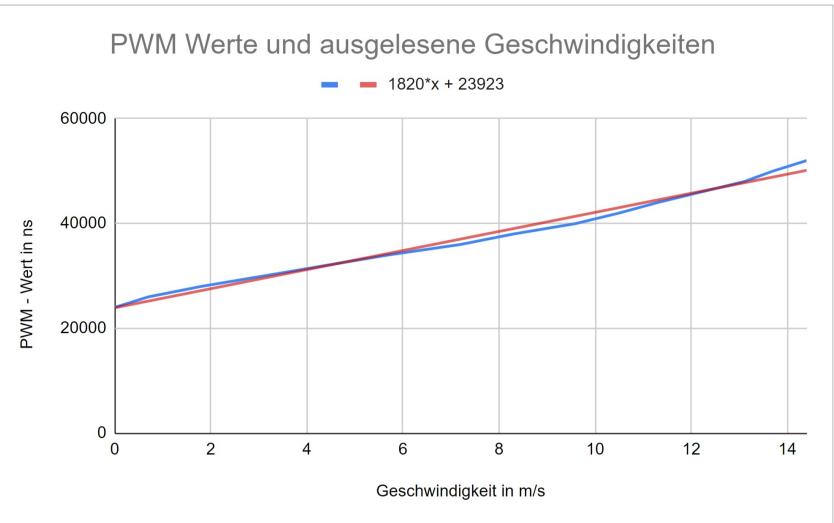


Abbildung 4.7: PWM-Werte und die dabei aufgezeichnete Geschwindigkeit.

Die Funktion der linearen Trendlinie wurde genutzt um den benötigten PWM-Wert für eine gegebene Geschwindigkeit zu berechnen. Mit diesem Ansatz wurde mit der Befehlsfolge aus Listing 4.6 die Abbildung 4.8 erstellt.

Listing 4.4: Kontroller 1 Hinterradnabenmotor

```

1 if (setpoint_speed >= 1.9){
2     new_pwm_value = (int)(1820 * setpoint_speed + 23923);
3 } else{
4     new_pwm_value = 0;
5 }
```

Die grüne Linie ist der Soll-Wert und die lila Linie ist der Ist-Wert. Man erkennt, dass einige Geschwindigkeiten, wie z.B. 5 m/s, gut erreicht werden, während andere nicht so präzise sind. Zudem gab es in diesem Test keine veränderliche Last am Hinterrad.

In Abbildung 4.9 wurde der Motor auf die gleiche Weise betrieben. In Abbildung 4.10 wurde dieser Test erneut durchgeführt, allerdings wurde die Hinterradbremse des Fahrrads

leicht gehalten, um ein Schleifen und somit eine Last zu simulieren. Zwischen Sekunde 3 und 4 wurde die Bremse zu stark gedrückt, aber zwischen Sekunden 5-10 lässt sich gut erkennen, dass der Fehler aufgrund der zusätzlichen Last größer ist als ohne.

Um diesen Fehler bei verschiedenen Geschwindigkeiten und insbesondere bei Last zu minimieren, wurde dieser Regler um den I-Anteil des PID-Kontroller-Moduls erweitert. Dadurch wird der Fehler durch Hinzufügen eines weiteren Steuerwerts verringert. Dies ist in Abbildung 4.11 zu sehen. Der entsprechende Code sieht folgendermaßen aus:

**Listing 4.5: Kontroller 2 Hinterradnabenmotor**

```

1 if (setpoint_speed >= 1.9){
2     new_pwm_value = (int)(1820 * setpoint_speed + 23923);
3 } else{
4     new_pwm_value = 0;
5 }
6
7 pid_hnm_value = (int)pid_output(&hnm_pid, setpoint_speed,
8                                 speed_value, ms_since_start);
9 new_pwm_value += pid_hnm_value;
10 if (new_pwm_value < 0){
11     new_pwm_value = 0;
12 } else if (new_pwm_value > 55000){
13     new_pwm_value = 55000;
14 }
```

Die Zeilen 9-13 sorgen dafür, dass der neue Steuerwert gültige Werte annimmt.

Da dieser Korrekturwert relativ langsam zum bestehenden Wert hinzugefügt wird, dauert es eine gewisse Zeit, bis der angesteuerte Wert erreicht ist. Zudem übersteuert dieses Verfahren zunächst die angestrebte Geschwindigkeit. Eine Reduzierung dieser Übersteuerung sollte durch Hinzufügen eines D-Anteils erreicht werden können, dies wurde jedoch nicht getestet.

## 4.2 Hinterradnabenmotor Test

---

**Listing 4.6:** Befehlsfolge 1

```

1 setpoint_speed = 0.0 f;
2 usleep(500000);
3 setpoint_speed = 2.0 f;
4 sleep(6);
5 setpoint_speed = 5.0 f;
6 sleep(6);
7 setpoint_speed = 2.0 f;
8 sleep(10);
9 setpoint_speed = 0.0 f;
10 sleep(5);

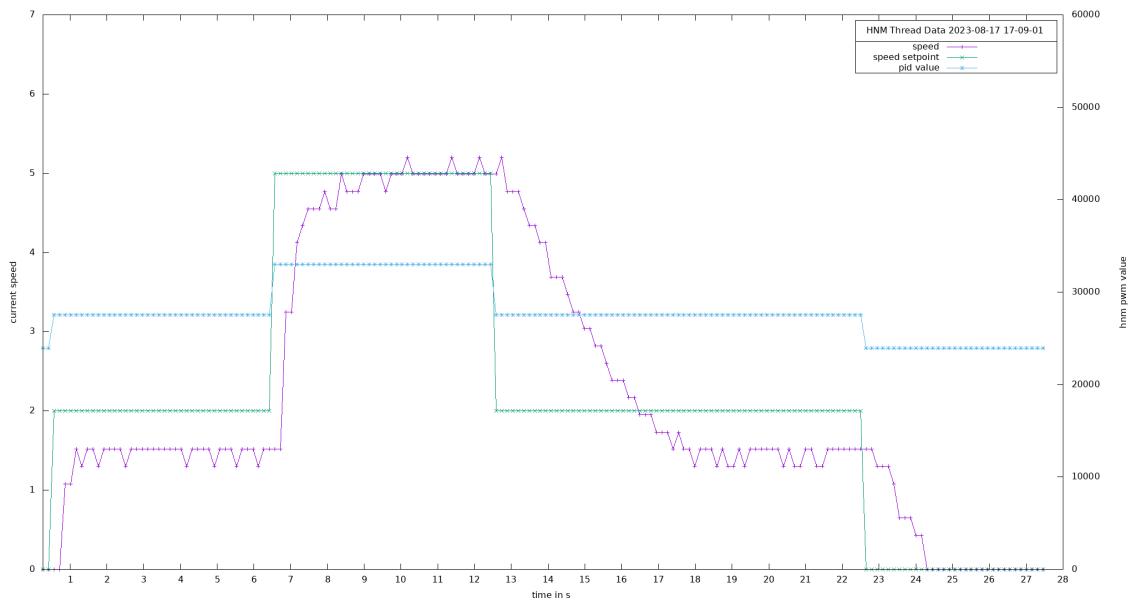
```

**Listing 4.7:** Befehlsfolge 2

```

1 setpoint_speed = 0.0 f;
2 usleep(500000);
3 setpoint_speed = 3.3 f;
4 sleep(10);
5 setpoint_speed = 0.0 f;
6 sleep(5);

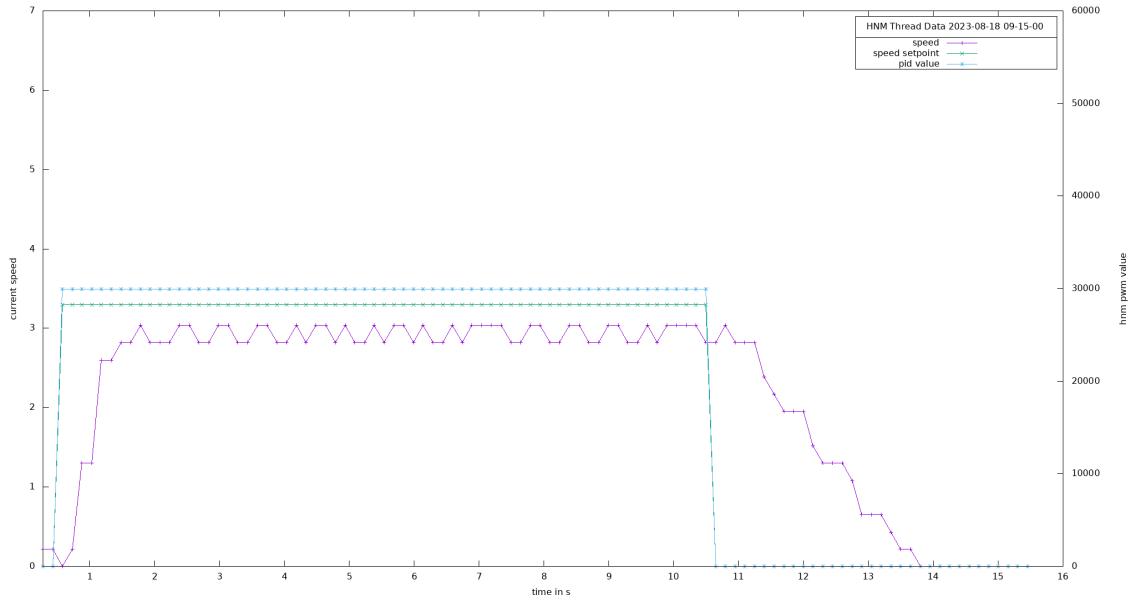
```



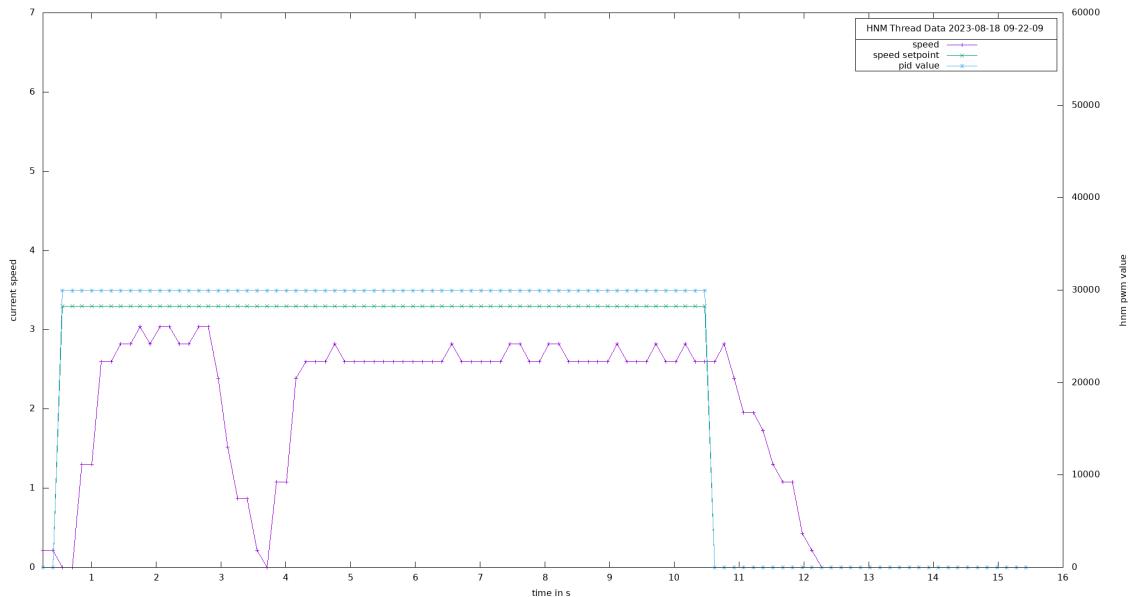
**Abbildung 4.8:** Regler basierend auf linearer Funktion. Ausführung der Befehlsfolge in Listing 4.6. Durchschnittliche Zeit zwischen zwei Werten:  $150222,4\mu s$  ( $6,7Hz$ ). Standardabweichung:  $54,7\mu s$ .

## 4.2 Hinterradnabenmotor Test

---



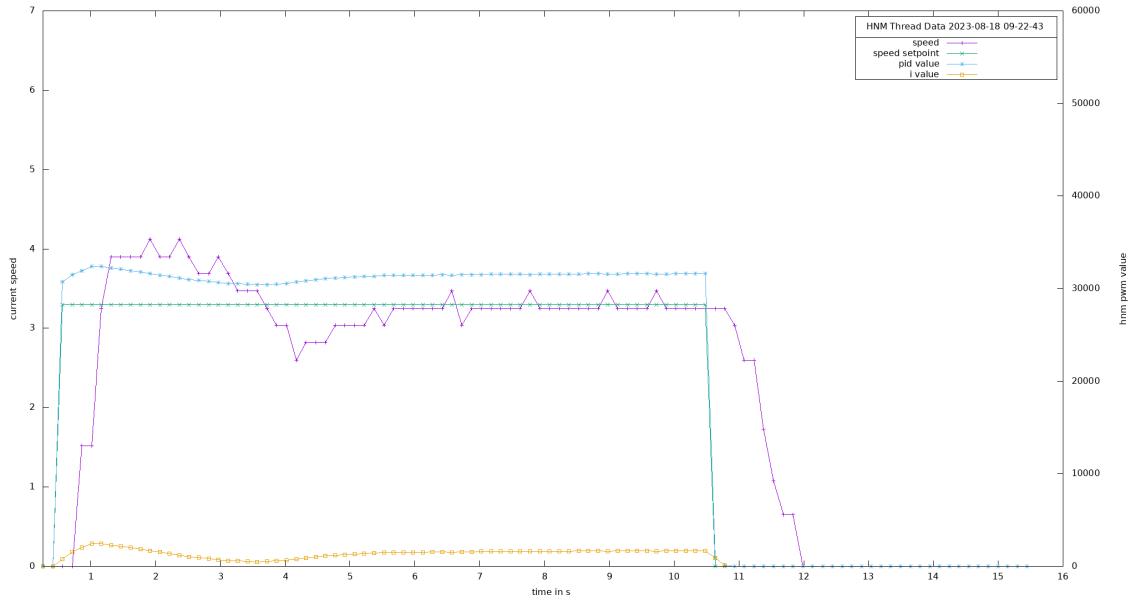
**Abbildung 4.9:** Regler basierend auf linearer Funktion. Ausführung der Befehlsfolge in Listing 4.7. Durchschnittliche Zeit zwischen zwei Werten: 150214,0 $\mu$ s (6,7Hz). Standardabweichung: 49,9 $\mu$ s.



**Abbildung 4.10:** Regler basierend auf linearer Funktion. Ausführung der Befehlsfolge in Listing 4.7 mit leichtem Schleifen der Bremse. (Zwischen Sekunde 3 und 4 wurde diese zu stark angezogen.) Durchschnittliche Zeit zwischen zwei Werten: 1510228,5 $\mu$ s (6,7Hz). Standardabweichung: 97,8 $\mu$ s.

### 4.3 Gesamttest des Fahrrads

---



**Abbildung 4.11:** Regler basierend auf linearer Funktion mit zusätzlichem I-Anteil. Ausführung der Befehlsfolge in Listing 4.7 mit leichtem Schleifen der Bremse. Zielwert wird nun gut erreicht. Durchschnittliche Zeit zwischen zwei Werten: 150309,6 $\mu$ s (6,7Hz). Standardabweichung: 679,5 $\mu$ s.

### 4.3 Gesamttest des Fahrrads

Die Ausführung des Gesamttests erfolgte auf einem alten Flugplatz. Dieser war sehr eben, breit und hatte nur wenige Risse bzw. Löcher. Die Fahrt wurde mit einer Geschwindigkeit von 3,1 m/s getestet, damit eine Person mit dem Fahrrad mitlaufen konnte, um dieses bei Fehlfunktion zu stoppen. Das Programm `autobike.c` wurde hierfür verwendet.

#### 4.3 Gesamttest des Fahrrads

---



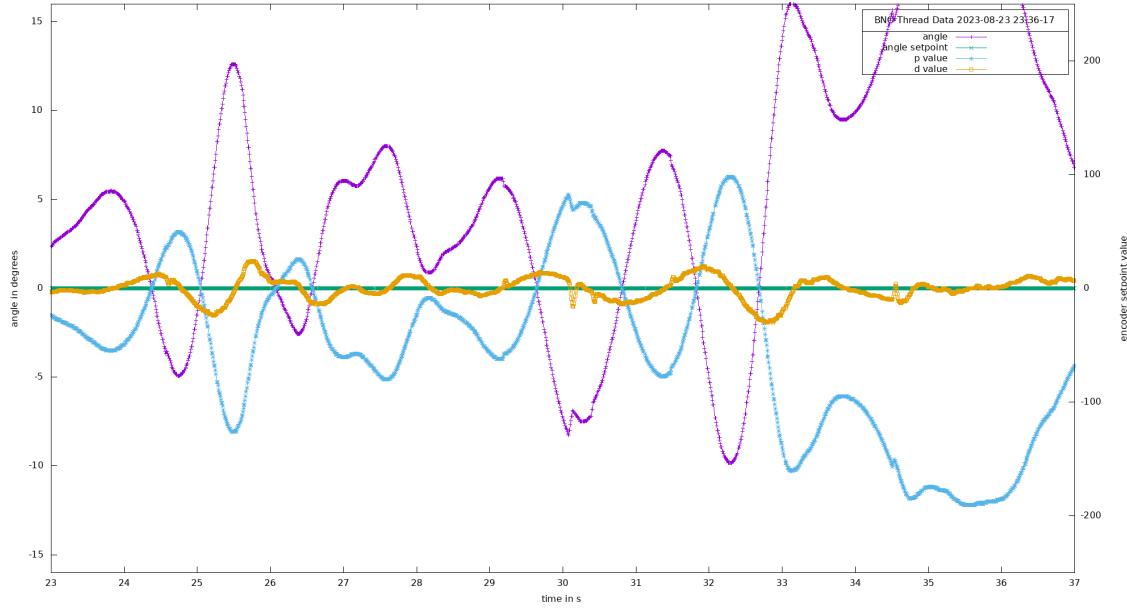
**Abbildung 4.12:** Bild der Teststrecke und der Versuchsdurchführung.

Die Regelwerte für Hinterrad und Lenkmotor wurden aus den vorherigen Versuchen übernommen.

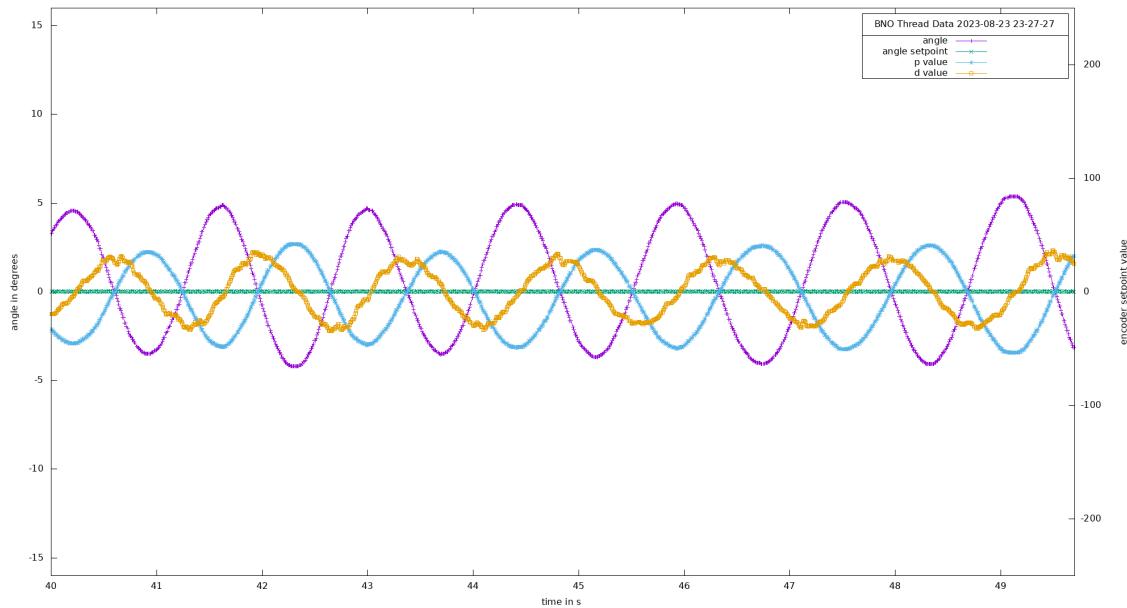
Der Effekt verschiedener Konstanten für den Ableitungsterm des PD-Reglers wurden untersucht. Die folgenden Abbildungen sind Auszüge aus verschiedenen Testfahrten, in denen das Fahrrad die Zielgeschwindigkeit erreicht hat und das Fahrrad möglichst gerade gefahren ist.

### 4.3 Gesamttest des Fahrrads

---



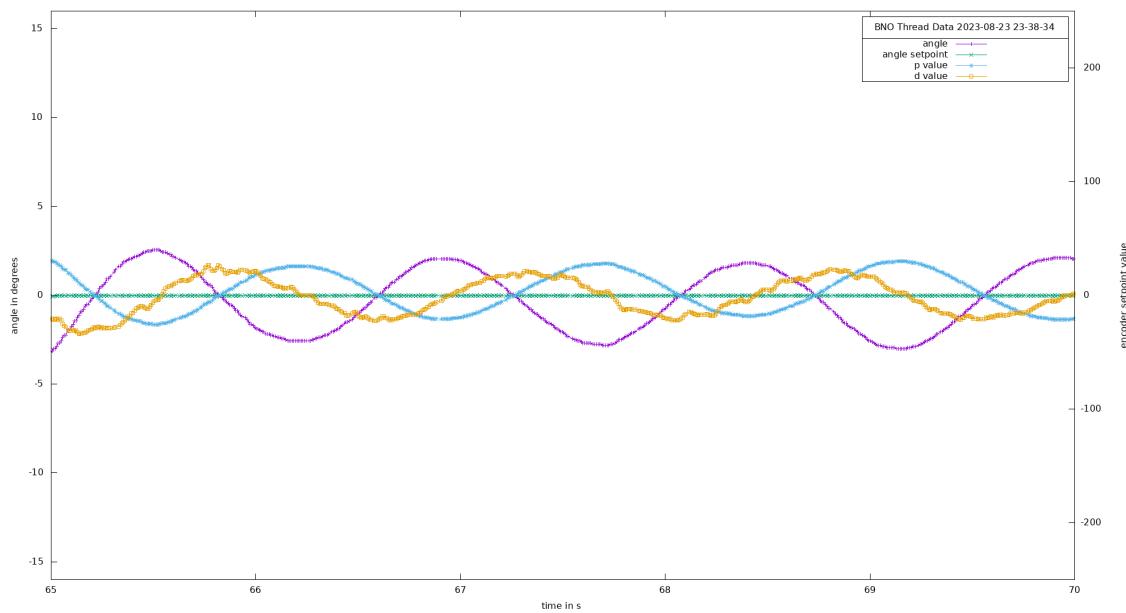
**Abbildung 4.13:** D-Wert von 0,6. Das Fahrrad konnte nicht balancieren.



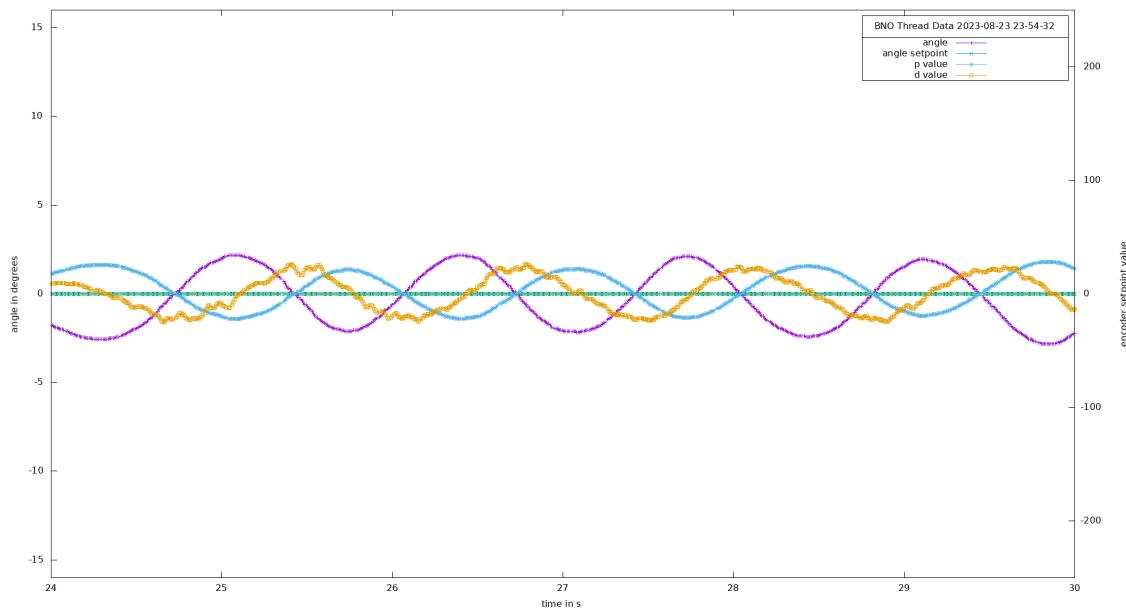
**Abbildung 4.14:** D-Wert von 1,6 mit einem durchschnittlichen Fehler von  $2,86^\circ$  pro Sekunde.

#### 4.3 Gesamttest des Fahrrads

---



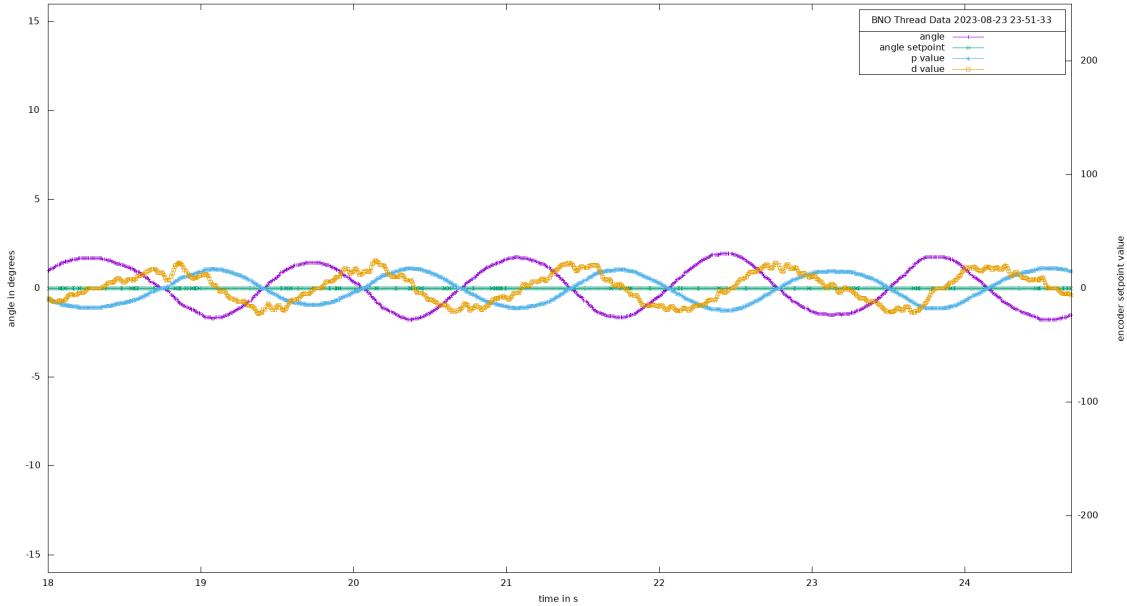
**Abbildung 4.15:** D-Wert von 2,0 mit einem durchschnittlichen Fehler von  $1,58^\circ$  pro Sekunde.



**Abbildung 4.16:** D-Wert von 2,2 ein durchschnittlicher Fehler von  $1,50^\circ$  pro Sekunde.

### 4.3 Gesamttest des Fahrrads

---



**Abbildung 4.17:** D-Wert von 2,4 ein durchschnittlicher Fehler von  $1,1^\circ$  pro Sekunde.

Die Güte der Regler wurden errechnet, indem der absolute Fehler zwischen dem Soll- und Ist-Wert auf-integriert wurde. Um diesen Fehler zu normalisieren wurde dieser anschließend durch die in dem Test vergangene Zeit geteilt. Es zeigt sich, dass mit dem gegebenen P-Wert ein D-Wert benötigt wird, damit das Fahrrad balancieren kann. Obwohl der Fehler in dieser Testreihe bei einem D-Wert von 2,4 am geringsten ist, wurde ein D-Wert von 2,2 präferiert. Dies liegt daran, dass für das Lenken der Soll-Winkel des Reglers verändert wird. Mit dem D-Wert von 2,4 hat das Fahrrad zu stark auf diese Veränderung des Soll-Winkels durch Einlenken reagiert. Um dem entgegen zu wirken könnte das Lenken verlangsamt werden, dies war jedoch in den weitergehenden Tests nicht erwünscht, damit engere Kurven gefahren werden konnten.

Für den nächsten Test sollte eine längere Fahrt durchgeführt werden, in welcher sowohl das gerade Fahren, sowie das Fahren von Kurven getestet werden sollte. Die Nutzung der Fernbedienung wurde so konfiguriert, dass sie den Soll-Winkel verändert, jedoch wurde dieser Winkel selbst nach dem Zurücklenken der Fernsteuerung in die neutrale Stellung beibehalten. Dadurch konnte die Fernbedienung genutzt werden um den Soll-Winkel des PD-Reglers zu trimmen oder das Fahrrad zu lenken.

In diesem Test wurde das Fahrrad zunächst möglichst gerade auf der Strecke gehalten, wonach eine starke Rechtskurve folgt, um das Fahrrad zu wenden und in die entgegengesetzte Richtung fahren zu lassen.

Bis Sekunde 8 wurde zuerst die Funktion des Fahrrads überwacht und dieses dann auf die Zielgeschwindigkeit gebracht und losgelassen. Es fuhr bis Sekunde 39 auf einer recht

geraden Linie und fuhr zwischen Sekunde 39-47 eine starke Rechtskurve, um zu wenden. Danach wurden noch weitere Kurven mit größerem Lenkradius gefahren, bis sich das Fahrrad wieder am Start des Tests befand.

Einstellungen aus Listing 4.3 wurden in dieser Testfahrt verwendet:

**Listing 4.8: PID-Werte aus autobike.c**

```

1 //PID Angle to Steering
2 #define ANGLE_PID_KP 10.0 f
3 #define ANGLE_PID_KI 0.0 f
4 #define ANGLE_PID_KD 2.2 f
5 #define ANGLE_PID_OUTPUT_MIN -150.0 f
6 #define ANGLE_PID_OUTPUT_MAX 150.0 f
7 #define ANGLE_PID_INTEGRAL_MIN -60.0 f
8 #define ANGLE_PID_INTEGRAL_MAX 60.0 f
9
10 #define STEERING_MAX_ANGLE_OFFSET 14.0 f
11 #define STEERING_ANGLE_OFFSET_STEPP 0.05 f
12 #define STEERING_MAX_OFFSET 0.0 f
13 #define STEERING_OFFSET_STEPP 0.0 f
14
15 #define BNO_THREAD_WAIT 5000
16
17 //PID Steering PID
18 #define MD49_PID_KP 850.0 f
19 #define MD49_PID_KI 300.0 f
20 #define MD49_PID_KD 30.0 f
21 #define MD49_PID_OUTPUT_MIN -100000.0 f
22 #define MD49_PID_OUTPUT_MAX 100000.0 f
23 #define MD49_PID_INTEGRAL_MIN -20000.0 f
24 #define MD49_PID_INTEGRAL_MAX 20000.0 f
25
26 #define MD49_THREAD_WAIT 1000
27
28 //PID HNM
29 #define HNM_PID_KP 0.0 f
30 #define HNM_PID_KI 1600.0 f
31 #define HNM_PID_KD 0.0 f
32 #define HNM_PID_OUTPUT_MIN -20000.0 f
33 #define HNM_PID_OUTPUT_MAX 20000.0 f
34 #define HNM_PID_INTEGRAL_MIN -1000.0 f

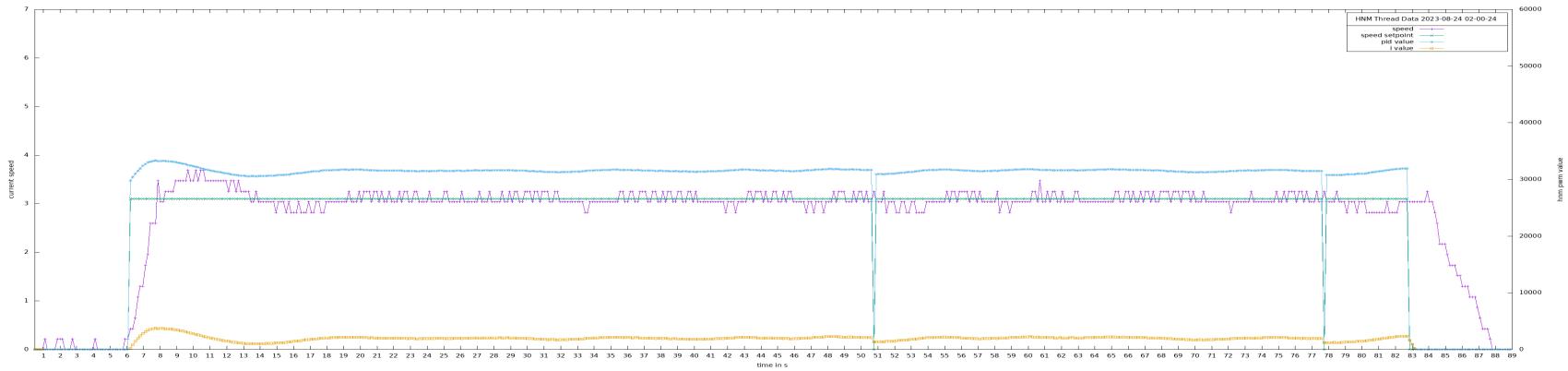
```

#### 4.3 Gesamttest des Fahrrads

---

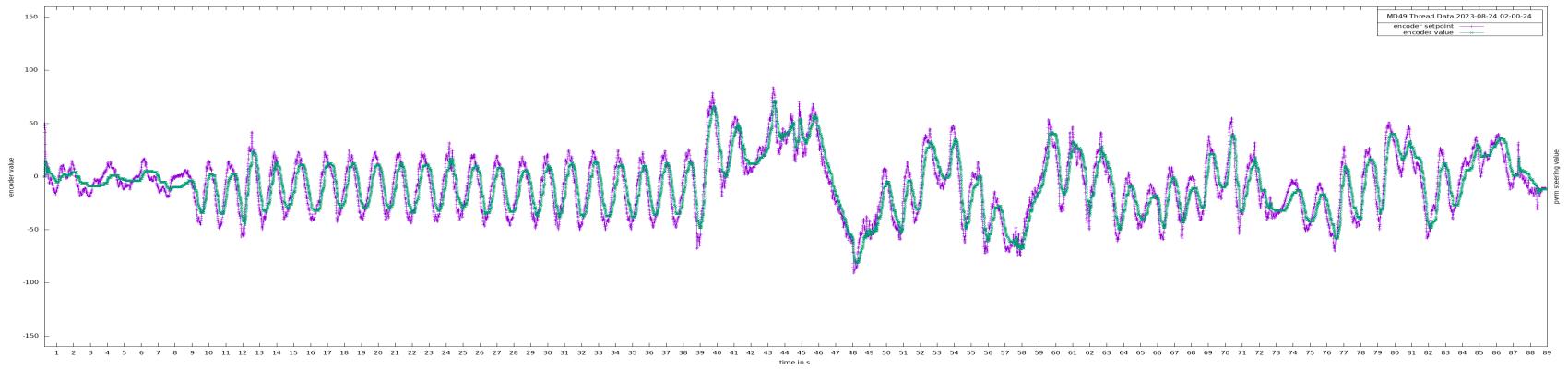
```
35 #define HNM_PID_INTEGRAL_MAX          10000.0 f
36
37 #define HNM_PWM_MULTIPLIER             1820
38 #define HNM_PWM_ADDITION               23923
39
40 #define HNM_REMOTE_SPEED              3.1 f
41
42 #define HNM_THREAD_WAIT                150000
```

### 4.3 Gesamttest des Fahrrads



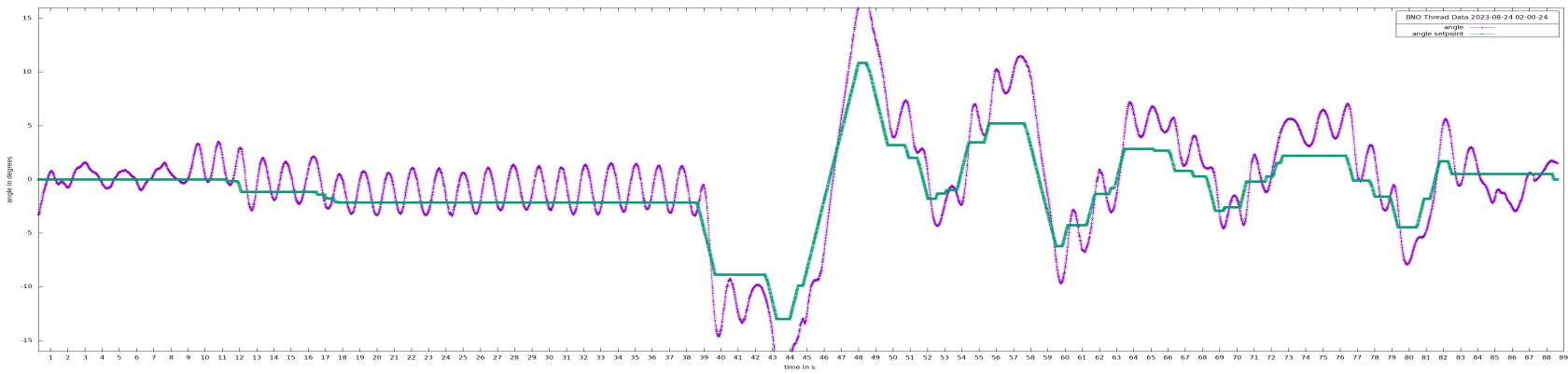
**Abbildung 4.18:** *autobike.c* Test. Hinterradmotor-Thread. (Die zwei kurzen Spitzen in Sekunde 51 und 77 geschehen aufgrund der Ungenauigkeit der Fernbedienung) Mit einer durchschnittlichen Zeit zwischen 2 Messwerten von  $150552,8\mu\text{s}$  ( $6,6\text{Hz}$ ) und einer Standardabweichung von  $1235,1\mu\text{s}$

51



**Abbildung 4.19:** *autobike.c* Test. Lenkmotor-Thread. Mit einer durchschnittlichen Zeit zwischen 2 Messwerten von  $5422,5\mu\text{s}$  ( $184,4\text{Hz}$ ) und einer Standardabweichung von  $1092,8\mu\text{s}$

### 4.3 Gesamttest des Fahrrads



**Abbildung 4.20:** Neigungswinkel-Thread. Mit einer durchschnittlichen Zeit zwischen 2 Messwerten von  $7811,8\mu\text{s}$  (128,0Hz) und einer Standardabweichung von  $1591,0\mu\text{s}$ . Bis Sekunde 39 gerade Fahrt, danach in Kurven.

Abbildung 4.18 zeigt den Regler des Hinterrads. Für jede Sekunde beträgt der Fehler des Reglers 0,04m/s. Der Regler hat einen geringen Fehler und erreicht annähernd die Zielgeschwindigkeit. Die zwei kurzen Spitzen in Sekunde 51 und 77 der Abbildung lassen sich auf die Geschwindigkeitsvorgabe der Fernbedienung zurückführen. Da die Signalmessung durch das Messen von Spannung anstelle des PWM-Wertes für den Lenkwert ungenau ist (siehe 3.4.6).

Der Fehler des Lenkmotor-Reglers in Abbildung 4.19 ist 11,34 Schritte pro Sekunde, welches einem Lenkwinkel-Fehler von ungefähr  $2,9^\circ$  pro Sekunde entspricht.

Von Sekunde 8 bis zum Fahrtsende hat der Regler für den Neigungswinkel in Abbildung 4.20 einen Fehler von  $2,09^\circ$  pro Sekunde. Auffallend ist dass der gesamte Fehler oberhalb der grünen Linie  $116,83^\circ$  ist, und nur  $51,94^\circ$  unterhalb.

Dieser auffällige Unterschied lässt sich folgendermaßen begründen: Um eine gerade Linie zu fahren wurde der Soll-Winkel mit der Fernbedienung während des Tests angepasst. Für eine gerade Fahrt des Fahrrads ist meist ein gewisser Soll-Winkel nötig, welcher nicht  $0^\circ$  ist. Dies hat sich in weiteren Test-Fahrten bestätigt. Dafür gibt es mehrere Gründe. Die Lenkradstellung wird für jede Programmausführung wieder zurück gesetzt, diese wird jedoch durch den Encoder-Wert des Lenkmotors bestimmt und stellt nicht die absolute Stellung des Lenkrads dar. Die Lenkradstellung ist also recht ungenau und selbst bei einem Encoder-Wert von 0 nicht unbedingt gerade (siehe 4.1.2). Weiterhin ist es so, dass der Sensor für den Neigungswinkel des Fahrrads mittels Pins auf einem Steckbrett angebracht ist. Dadurch kann der Winkel, in dem der Sensor angebracht ist, variieren. Weitere Gründe, für ein unbeabsichtigtes Abdriften der Fahrtrichtung könnte die Gewichtsverteilung der Komponenten am Fahrrad sein, da z.B. der Lenkmotor leicht versetzt zum Fahrrad angebracht ist oder auch die Unebenheit der Fahrbahn selbst.

Dass der Fehler des Lenkwinkels bei dieser Fahrt bei  $2,09^\circ$  und nicht wie in Abbildung 4.16  $1,50^\circ$  ist, lässt sich ebenfalls begründen: Durch die Veränderung des Soll-Winkels kann das Fahrrad auch Kurven fahren, ohne dass die Stützräder aufsetzen. Um eine Balance zu erreichen und um Kurven fahren zu können ein PD-Regler ausreichend.

Wenn der Fehler-Term 0 wird und der Winkel keine Veränderung erfährt, ist die Stellung des Lenkrads gerade. Wird ein Soll-Neigungswinkel von  $8^\circ$  für eine Kurvenfahrt angegeben, wird der Ist-Wert immer um einen höheren Winkel z.B.  $11^\circ$  oszillieren. Da das Fahrrad sonst für den  $8^\circ$  Neigungswinkel ein gerades Lenkrad vorgäbe. Um diesen Fehler zu minimieren könnte ein I-Anteil eingestellt werden. Möglicherweise könnte auch ein zum Soll-Winkel Proportionaler Offset der Lenkerstellung helfen. Dies wurde jedoch nicht getestet.

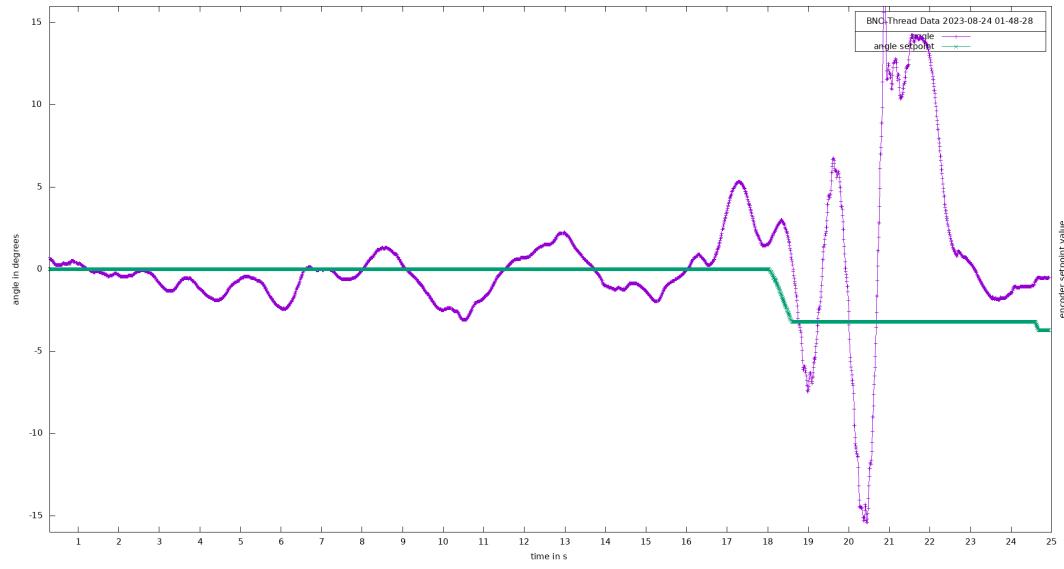
Der Füllstand der Reifen hat einen Einfluss auf die Fahrt des Fahrrads. Mit vollen Reifen hat das Vorderrad schnell angefangen zu springen. Dies wurde durch die schnelle Veränderung im Winkel durch den Lenkwinkel-PD-Kontroller verstärkt und führte da-

#### 4.3 Gesamttest des Fahrrads

---

zu, dass das Fahrrad umgefallen ist. Mit reduziertem Reifendruck ist dieser Effekt nicht aufgetreten und das Fahrrad konnte die Balance halten.

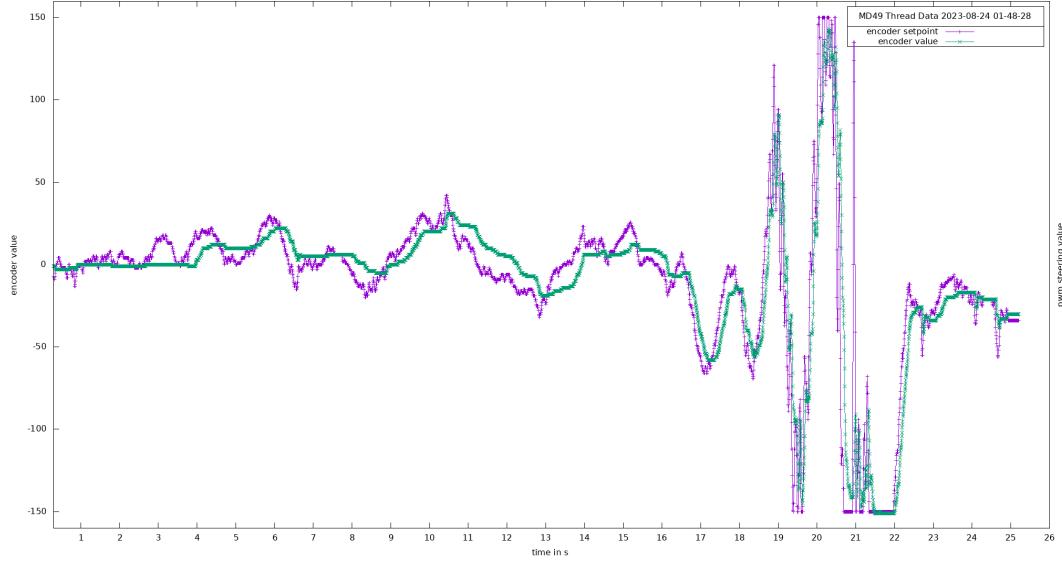
Ein ähnlicher Effekt zeigt sich, wenn die Geschwindigkeit des Fahrrads auf 5m/s erhöht wird. Die schnelle Veränderung des Winkels sorgt dafür, dass das Lenkrad zu stark schwingt. Dieser einfache PD-Regler ist nicht ausreichend um eine variable Geschwindigkeit des Fahrrads zuzulassen und es müsste mit einem erweiterten Steuerungsansatz gearbeitet werden.



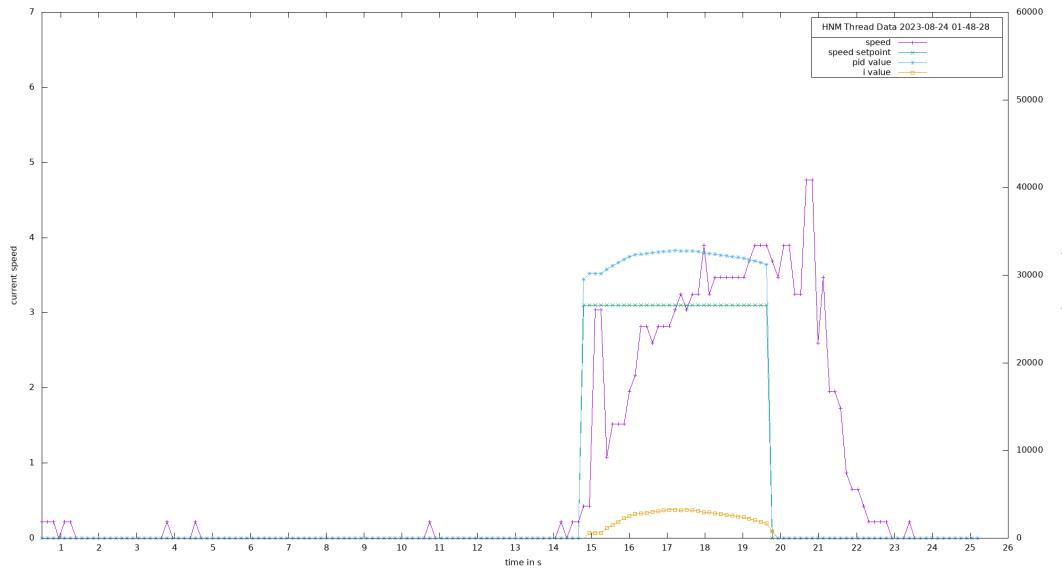
**Abbildung 4.21:** *autobike.c Test. Lenkwinkel-Thread. Aufschwingen des Fahrrads bei vollen Reifen.*

### 4.3 Gesamttest des Fahrrads

---



**Abbildung 4.22:** *autobike.c Test. Lenkmotor-Thread. Volle Reifen. Die Vorgabe des Lenkwerts wird annähernd erreicht, ist jedoch nicht zielführend für eine balancierte Fahrt.*



**Abbildung 4.23:** *autobike.c Test. Hinterradmotor-Thread. Volle Reifen. Bis Sekunde 14 befindet sich das Fahrrad im Stillstand. Ab Sekunde 15 wird es für eine Kurze Zeit beschleunigt.*

## 5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde gezeigt, dass ein Fahrrad mit Lenkradsteuerung selbstbalancierend fortbewegt werden kann. Dabei sind sogar gesteuerte Richtungswechsel möglich, welche in dieser Arbeit mittels Fernbedienung vorgegeben wurden.

Die einzelnen Komponenten der Fahrrad-Plattform und eine mögliche Methodik für die Steuerung wurden vorgestellt. Zuletzt wurde eine erfolgreiche Testfahrt gezeigt. Das Ziel einer balancierten Fahrt von mindestens 60 Sekunden wurde erreicht ohne dass es erkennbare Schwierigkeiten gegeben hätte, die einer Verlängerung der Fahrt im Wege gestanden hätten. Diese Arbeit hat sich auf die Fahrt bei einer konstanten Geschwindigkeit und auf ebenem Boden beschränkt.

Hiermit wurde eine Grundlage geschaffen, auf der in Zukunft relativ einfach weitere Steueralgorithmen implementiert und getestet werden können.

### Ausblick

Weiterentwicklungen, welche mit dieser Plattform umsetzbar und sinnvoll sein könnten, sind beispielsweise eine geschwindigkeitsabhängige Steuerung, oder in Bezug auf ein assistierendes Fahren auch eine gewichtsabhängige Steuerung, da sich Reifendruck und Last durch Fahrten mit und ohne Fahrer stark unterscheiden. Ein weiterer interessanter Aspekt, welcher mit diesem Fahrrad genauer untersucht werden könnte, ist der Effekt von Gewichtsverlagerung durch eine Person oder Transportlast auf die Lenkung.

Ein echtzeitfähiger Watchdog welcher die Kommunikation der einzelnen Threads überwacht würde die Sicherheit weiter erhöhen.

Es könnte die Übertragbarkeit des Systems auf ein kleineres (Modell-)Fahrrad untersucht werden, da dadurch der Platzbedarf für weitere Entwicklungen und zum Testen deutlich verringert werden könnte. Dies könnte ebenfalls die Gefahr eines Unfalls minimieren.

Es lässt sich leicht vorstellen, wie die Funktionalität auch mit einer Kamera oder GPS-Tracking erweitert werden könnte. Ein balancierendes Fahrrad könnte es Älteren oder Menschen, mit Problemen das Gleichgewicht zu halten ermöglichen, wieder sicherer Fahrrad zu fahren. Es könnte auch, wie bei Autos, eine Hinderniserkennung eingebaut werden.

Der Effekt eines zusätzlichen Reaktionsrads könnte untersucht werden.

---

Mit einem automatisch ausfahrenden Ständer könnte das Fahrrad auch von selbst stehen bleiben oder von alleine an eine bestimmte Stelle bestellt oder geparkt werden.

# Literaturverzeichnis

- [Baum22] U. Baumann, *Umfallen unmöglich dank Selbststabilisierung*, 2022
- [BF] BeagleBoard.org-Foundation, *System Reference Manual for the BeagleBone Black*, url: <https://docs.beagleboard.org/latest/boards/beaglebone/black/ch01.html>, letzter Zugriff: 23. Juli 2023
- [Bosc21] Bosch, *Data sheet, BNO055, Intelligent 9-axis absolute orientation sensor*, 2021
- [CARS23] CARSON, *Reflex Wheel Start 2.4GHz Digital Proportional Radio Control System*, 2023
- [Kara] A. Karabila, *Auf dem Weg zum selbstfahrenden Fahrrad: Analyse und Implementierung einer Regelung der Balance (und der Geschwindigkeit)*
- [kuot] G. N. kumar et al., *Design and Analysis of a Self-balancing Bicycle Model*
- [Moll19] D. Molly, *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux, Second Edition*, John Wiley & Sons, Inc., Indianapolis, Indiana, 2019
- [PaYi20] S.-H. Park, S.-Y. Yi, *Active Balancing Control for Unmanned Bicycle Using Scissored-pair Control Moment Gyroscope*, International Journal of Control, Automation and Systems, Bd. 18, Nr. 1, S. 217–224, 2020
- [Ranz] A. Ranz, *How to make a bicycle (not) fall - implementing a steering control for a self-balancing bicycle*
- [REa] Robot-Electronics, *EMG49, mounting bracket and wheel specification*
- [REb] Robot-Electronics, *MD49 - Dual 24 Volt 5 Amp H Bridge Motor Drive*
- [Taot17] N. Tamaldin et al., *Design of a Self-balancing Bicycle*, in *Proceedings of Mechanical Engineering Research Day 2017*, S. 160–161, 2017
- [TI19] Texas-Instruments, *AM335x and AMIC110 Sitara Processors, Technical Reference Manual*, 2019
- [Yama17] Yamaha, *Yamaha motor concept model Motoroid*, 2017

# A Anhang

## A.1 Kompilieren des Codes

Zur Erstellung des Makefiles für dieses Projekt wird CMake in der Version 3.13.4 verwendet. CMake ist ein Open-Source-Build-System welche das Kompilieren von Projekten vereinfacht. Dazu wird eine `CMakeLists.txt` Datei durch den Benutzer erstellt, welche Abhängigkeiten, Informationen über Projektstruktur, Linker-, Kompiler-Optionen und Build-Ziele enthalten kann. Die Verwendung von CMake vereinfacht das Kompilieren von Projekten mit mehreren Implementierungsdateien erheblich, da nur die geänderten Dateien neu kompiliert und die von diesen Dateien abhängigen ausführbaren neu gelinkt werden.

Um eine neue ausführbare Datei zu erstellen, muss in der Datei `CMakeLists.txt` der Befehl `add_executable()` verwendet werden. An erster Stelle sollte der Name der Zielfile stehen, gefolgt von den Implementierungsdateien ohne Kommata. Nach dem Speichern der Datei reicht es aus, `make` auszuführen, um die neue Datei zu erstellen. Alternativ kann mit `make [dateiname]` nur eine bestimmte Datei neu kompiliert werden.

## A.2 Erstellen von Diagrammen mittels `plot.py`

Um die Visualisierung der Logging Dateien zu starten, wird im Terminal im Verzeichnis `autobike/logging/tools/` der Befehl `python3 plot.py` ausgeführt. Dieser erzeugt automatisch die Diagramme. Falls nur bestimmte Daten der Datei geplottet werden sollen kann dies mit `python3 plot.py start_sekunden end_sekunden [x_pixel]` erreicht werden. Um beispielsweise einen Plot zu erstellen, der 1080 pixel breit ist und die Daten von Sekunde 14 bis 22 umfasst muss `python3 plot.py 14 22 1080` aufgerufen werden. Die Anzahl an Pixeln ist dabei optional und kann auch weggelassen werden. Wenn ein Diagramm ohne Anfangs- oder Endzeitpunkt erstellt werden soll, kann das Argument `start_sekunden` oder `end_sekunden` durch den leeren String `""` ersetzt werden.

### A.3 USB-WLAN-Adapter

Für die Durchführung der Testfahrten ist es hilfreich, eine Verbindung über WLAN mit dem Fahrrad herzustellen. Da das Einrichten des Adapter nicht trivial ist, soll sie im Folgenden beschrieben werden:

1. Um herauszufinden, ob der USB-WLAN-Adapter als Zugangspunkt konfiguriert werden kann, kann im Terminal der Befehl `sudo iw list | grep "Supported interface modes" -A 8` ausgeführt werden. Falls in der Liste der Verfügbaren Modi „AP“ steht, kann mit den nächsten Schritten fortgefahren werden.
2. Als nächstes müssen die benötigten Softwarepakete mit `sudo apt-get install hostapd dnsmasq` heruntergeladen werden.
3. Die Netzwerk-Schnittstelle wird in `/etc/network/interfaces` bearbeitet. Am Ende dieser Datei müssen diese Zeilen eingefügt werden:

```
auto wlan0
iface wlan0 inet static
    address 192.168.42.1
    netmask 255.255.255.0
    network 192.168.42.0
    broadcast 192.168.42.255
```

`wlan0` kann mit dem Namen der Netzwerk-Schnittstelle ausgetauscht werden.

4. Um den Netzwerkdienst neu zu starten wird  
`sudo systemctl restart networking.service` verwendet.
5. In `/etc/hostapd/hostapd.conf` müssen die folgenden Zeilen hinzugefügt werden.

```
interface=wlan0
driver=nl80211
ssid=AccessPointName
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=AccessPointPassword
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
```

```
rsn_pairwise=CCMP
```

Hier muss möglicherweise der Name der Netzwerk-Schnittstelle angepasst werden. AccessPointName und AccesPointPassword müssen mit dem gewünschten Namen und Passwort ausgetauscht werden.

6. Als nächstes muss der DHCP-Server eingerichtet werden. Dazu wird /etc/dnsmasq.conf geöffnet und folgende Zeilen hinzugefügt.

```
interface=wlan0
dhcp-range
    =192.168.42.100,192.168.42.199,255.255.255.0,12h
```

7. Wenn der BeagleBone Black neu gestartet wird, kann der Zugangspunkt mit folgenden Befehlen gestartet werden:

```
sudo systemctl start hostapd.service
sudo systemctl start dnsmasq.service
```

## A.4 Samba

Um die Dateien, die sich auf dem BeagleBone befinden leichter verwalten zu können, wurde die Open-Source-Software Samba (Version 4.9.5-Debian) installiert. Mit ihr ist es möglich über das Netzwerk auf freigegebene Dateien zuzugreifen.

1. Samba kann mittels den folgenden Befehlen installiert werden.

```
sudo apt-get update  
sudo apt-get install samba
```

2. In der Datei `/etc/samba/smb.conf` können Ordner hinzugefügt werden, um sie freizugeben. Um z.B. das Root Verzeichnis freizugeben, können die folgenden Zeilen in der Konfigurationsdatei eingefügt werden.

```
[root]  
comment = Root file system  
path = /  
browsable = yes  
read only = no  
valid users = debian  
create mask = 660  
directory mask = 770  
force user = debian  
force group = debian
```

3. Benutzer können mittels `sudo smbpasswd -a benutzername` hinzugefügt werden.
4. Mit `sudo systemctl restart smbd.service nmbd.service` wird der Samba-Dienst neu gestartet.
5. Jetzt kann beispielsweise von einem Windows-Rechner über `\BeagleBone_IP` auf das freigegebene Verzeichnis zugegriffen werden.