Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Automatische Fehlererkennung mit dem Dynamixel V2.0 Protokoll

Daniel Djahangir

6djahang@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 6803168

Erstgutachter: Dr. Andreas Mäder
Zweitgutachter: Marc Bestmann

Abgabe: 09.2020

Inhaltsverzeichnis

1 Einleitung

Beim Bauen und Entwickeln von Robotern kann es oft zu Hardwareproblemen kommen. Deswegen werden von den meisten Firmen, welche Hardware zur Verfügung stellen auch Diagnosetools zusätzlich und meist kostenfrei beim Kauf mitgeliefert. Leider bieten diese Programme nur externe Möglichkeiten an, welche nicht direkt in das eigene Programm integrierbar sind. Das bedeutet, dass immer dann, wenn ein Hardwareproblem vermutet wird erst zum Tool gewechselt werden muss, um dieses zu bestätigen. Hinzu kommt, dass es vor allem in der Robotik nur Software für die Verarbeitung der Pakete, also der Kommunikation und Überwachung von Komponenten gibt. Wenn aber ein Motor nicht mehr funktionsfähig ist, lässt sich ohne weitere Kenntnisse nicht genau erkennen, wie der Fehler zustande gekommen ist und sich beheben lässt. Es liegt leider nicht im Interessengebiet von den meisten dieser Hersteller, die Motoren selber reparieren zu können und in eine umfangreiche Fehlererkennung zu investieren. Unter den Dynamixel-Serien (Motoren des Robo-Cups, welche in dieser Arbeit behandelt werden) befinden sich sehr vielseitige Motoren, welche sowohl genau aber auch kraftvoll arbeiten. In diesem Kapitel möchte ich das Ziel dieser Arbeit erklären und geeignete, bereits vorhandene Analysemethoden vorstellen. Außerdem beschäftige ich mich mit der Funktionsweise der Kommunikation des Dynamixel-Protokolls, um eine ausreichende Grundlage für die Analyse zu erreichen.

1.1 Zielsetzung

Ziel dieser Bachelorarbeit ist es Merkmale und Eigenschaften mit dem Dynamixel Protokoll V2 fehlerhafter Dynamixel-Motoren zu finden und dann zu untersuchen. Falls sich defekte Motoren unterscheiden lassen, sollen diese durch ein Programm erkannt werden. Das Programm soll Gebrauch im RoboCup-Team der Universität Hamburg finden und bestenfalls während eines laufenden Roboters die Fehler erkennen. Eine Software die manuell und bei Bedarf ausgeführt werden kann, um den Bus zu untersuchen, ist aber auch möglich. Neben der Fehleranalyse ist es unter anderem auch die Aufgabe des Programms das Zeitverhalten der Motoren zu analysieren. Es soll herausgefunden werden, wie lange funktionstüchtige aber auch fehlerhafte Motoren zum Antworten brauchen. Dynamixel-Motoren können mit einer künstlichen Verzögerung versehen werden, welche die Kommunikation über den Bus verbessern sollen. Diese langsam eingestellten Motoren, aber auch fehlerhafte Motoren, die aus anderen Gründen langsam antworten, sollen erkannt werden. Vermutlich sind nicht alle Fehler erkennbar, klar unterscheidbar

oder können einem Motor zugewiesen werden.

1.2 Grundlagen des Dynamixel Protokolls

Das Dynamixel Protokoll [Rob] kommuniziert mit Paketen. Jedes dieser Pakete besitzt einen Header bestehend aus drei Bytes. Der Header markiert den Anfang eines Pakets und ist für die Erkennung von Paketen und Trennung von Paketinhalten wichtig. Dieser wird beim Dynamixel von einem leeren Reserved-Byte gefolgt, der momentan noch keinen Nutzen hat, aber bei neuen Protokollversionen vielleicht eine Rolle zugewiesen bekommt.

Header1	Header2	Header3	Reserved
0xFF	0xFF	0xFD	0x00

Tabelle 1.1: Dynamixel v2 Paketanfang

Danach folgt die Paket ID. Jede an dem Bus verbundene Komponente hat eine eigene ID, um Motoren voneinander unterscheiden und diese gezielt ansteuern zu können. Hierfür wird ebenfalls ein Byte zur Verfügung gestellt, wobei aber nur die 253 Werte (0x00 bis 0xFC) für IDs genutzt werden können. Um eine mehrfache Nutzung von dem Header und dem Escapecharacter des Stuffings zu vermeiden, werden 0xFD und 0xFF nicht benutzt. 0xFE wird hierbei als die Broadcast ID bezeichnet und richtet sich an alle am Bus angeschlossenen Komponenten. Jetzt wird in dem Paket noch die Länge für den restlichen Paketrumpf in Byte angegeben. Dies ist zwar im Allgemeinen nicht zwingend notwendig, kann aber die Verarbeitung beschleunigen, wenn es unterschiedlich lange Pakete geben kann, was bei Dynamixel der Fall ist. Da der restliche Rumpf nur noch aus der Instruktion, den Parametern und zwei (CRC-)Prüfbytes besteht, entspricht die Länge der Anzahl der Parametern + 3 (2 CRC-bytes und 1 Instruktionsbyte). Mit den Prüfbytes wird eine zyklische Redundanzprüfung (Cyclic redundancy check) durchgeführt (siehe 1.2.1). Für die Angabe der Länge werden genau wie für die Prüfsumme auch 2 Bytes gebraucht. Ein Beispielpaket würde z.B. so aussehen:

Header1	Header2	Header3	Reserved	ID	LEN1	LEN2	INST	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x03	0x00	0x01	0x19	0x4E

Tabelle 1.2: Dynamixel v2 Instruktionspaket

Dieses Paket entspricht einem Ping zu einem Motor mit der ID=1. Bei einem Ping wird unabhängig von dem Eintrag des Status Return Level-Registers des jeweiligen Motors versucht, ein Statuspaket zu senden. In dem Status Return Level-Register wird angegeben, ob Statuspakete nie nur bei Read-Instructions oder immer gesendet werden sollen.

Man kann also die Statuspakete auch ganz ausstellen, um nur noch halb so viele Pakete zu verarbeiten und den Master zu entlasten.

Header1	Header2	Header3	Reserved	ID	LEN1	LEN2	INST	ERR	P1	P2	P3	CRC1	CRC2
0xFF	0xFF	0xFD	0x00	0x01	0x07	0x00	0x55	0x00	0x06	0x04	0x26	0x65	0x5D

Tabelle 1.3: Dynamixel v2 Statuspaket

Dies ist ein Statuspaket als Antwort auf den oben gesendeten Ping. Statuspakete haben einen ähnlichen Aufbau wie Instruktionspakete. Nach der Längenangabe wird auch eine Instruktion angegeben, jedoch liegt der Wert des Instruktionsbytes immer bei 0x55, um das Paket als Statuspaket zu kennzeichnen. Das nächste Feld gibt an, wie die Instruktion verlaufen ist. Dieses Errorbyte enthält ein Alertbit (MSB) gefolgt von der Error Number (7 bits).

Das Alertbit verweist auf ein Hardwareproblem. Ist dieses auf 1 gesetzt, kann man in dem Hardware Error Status-Register des Motors nachschauen, was für Probleme vorliegen. In diesem Register gibt die Anzahl der Einsen die Anzahl der erkannten Probleme wieder. Je nachdem an welcher Stelle sich eine 1 befindet, ist ein anderes Problem erkannt worden. Dabei können die Motoren aber nur die fünf Problematiken erkennen, die in der Tabelle aufgelistet sind. Es werden deshalb auch nur maximal fünf Bits des Bytes aktiv.

Byte	Fehler
0	Input Voltage Error
1	Unused
2	Overheating Error
3	Motor Encoder Error
4	Electrical Shock Error
5	Overload Error
6	Unused
7	Unused

Tabelle 1.4: Hardware Error Status-Register Referenztabelle

Treten mehrere Fehler auf, kann es sein, dass der Motor selbstständig auf die Bits im Hardware Error Status-Register reagiert. Es gibt z.B. das Register „Shutdown“, in dem ein Code vordefiniert ist, sich aber auch bei den meisten Motoren ändern lässt und über dieses eingestellt werden kann, bei welchen fehlerhaften Bits der Motor heruntergefahren werden soll. Die Bits im Hardware Error Status-Register werden gesetzt, wenn bestimmte Limits überschritten werden. Diese Limits sind auch vorgespeichert, lassen sich aber auch ändern, obwohl Robotis mahnt, dies nicht unbedacht zu tun, da der Motor eventuell beschädigt werden könnte. Ähnlich lässt sich z.B. das Limit der Spannung und Tempe-

ratur setzen. Wird das Limit jeweils überschritten, setzt der Motor das dazugehörige Bit im Error Status-Register (0 oder 2).
Findet ein Fehler bei der Verarbeitung eines Paketes statt, gibt es einen Errorcode in den nachfolgenden 7 bits (Error Number). Die Error Number hat nicht direkt etwas mit dem Alertbit zu tun. Die Verarbeitungsfehler werden durch falsche Instruktionspakete hervorgerufen. Gibt man z.B. unbekannte Befehlsnummern an, oder sind die Überprüfungsbits des CRC falsch, dann wird der Befund in diesen 7 bits codiert. Alle sieben Fehlercodes sind in folgender Tabelle aufgelistet:

Fehlercode	Fehlerbezeichnung	Beschreibung
0x01	Result Fail	Motor kann Paket nicht verarbeiten.
0x02	Instruction Error	Undefinierte Instruktion
0x03	CRC Error	CRC ist nicht erfolgreich.
0x04	Data Range Error	Die zu schreibenden Daten befinden sich nicht in dem für das Register vorgesehenem Bereich.
0x05	Data Length Error	Die zu schreibenden Daten besitzen nicht die für das Register passende Länge.
0x06	Data Limit Error	Die zu schreibenden Daten übertreffen das Limit
0x07	Access Error	Die zu lesenden/schreibenden Register sind lese-/schreibegeschützt.

Tabelle 1.5: Error Number Referenztabelle

Diese von Dynamixel bereitgestellte Fehlererkennung funktioniert aber nur, wenn die Motoren erreichbar sind. Für meine Arbeit reicht es nicht aus diese Fehlercodes zu benutzen, sondern es müssen eigene Methoden zur Erkennung erarbeitet werden.

1.2.1 Zyklische Redundanzprüfung (CRC)

Beim CRC-Prüfverfahren wird an die Bitfolge der Nachricht ein Divisionsrest angefügt. Dieser Divisionsrest entspricht der binären Polynomendivision der Nachricht mit einem gewählten Generatorpolynom der Länge l. Die Nachricht wird vor der Division um $l-1$ Stellen logisch nach links verschoben. Ersetzt man die $l-1$ angefügten Stellen mit dem berechneten Rest, ergibt sich nun bei der Division der Nachricht in Polynomdarstellung mit dem Generatorpolynom kein Rest mehr.

Wird die Nachricht beim Sendevorgang verfälscht, verändert sich in den meisten Fällen auch der Divisionsrest von Nachricht und Generatorpolynom. Es kann aber auch vorkommen, dass sich die fehlerhafte Nachricht zu einem neuen Vielfachen von dem Generatorpolynom verändert hat und deshalb fälschlicherweise nicht als Fehler erkannt wird, da ebenfalls kein Rest beim Teilen entsteht.

Der Rest beim CRC lässt sich durch arithmetische Operationen sehr effizient berechnen. Durch das Prüfverfahren können Einbit-, Bündelfehler, Fehler, deren Polynomdarstellung einen kleineren Grad als das Generatorpolynom haben und Fehler mit ungeraden fehlerhaften Bits erkannt werden. Leider lässt sich dann aber nicht mehr der Fehler korrigieren.

1.2.2 RS-485 und TLL

Die Mastersysteme, welche ich bei der Analyse verwenden werde (USB2Dynamixel und U2D2_INT), unterstützen zwei Industriestandards für die Datenübertragung. Diese lassen sich durch einen Hebel verstellen. Die Motoren nutzen dann entweder RS-485 oder TLL. Die meisten Motoren z.B. der MX-Serie von Dynamixel kennzeichnen den Standard des Motors durch ein R oder T hinter der Seriennummer. RS-485 benutzt ein Leitungspaar und invertiert den Spannungspegel auf der zweiten Leitung. Durch die Differenz lässt sich dann wieder das ursprüngliche Datensignal konstruieren. Im Gegensatz zu TLL sind höhere Baudraten und größere Distanzen möglich, da Differenzsignale eine Möglichkeit sind, um Signalstörungen wie Gleichtaktstörungen zu beseitigen [SKS90]. Gleichtaktstörungen sind Störungen, die auf besonders großen Entfernungen bemerkbar sind. Dabei können Störfaktoren die Spannungen der Leitungen beeinflussen und so ein ungenaues Ergebnis erzeugen. Sendet man aber den invertierten Spannungspegel mit, lässt sich bestimmen, wie stark die Störung ist, beziehungsweise wie die ursprünglichen Daten ausgesehen haben müssen, wenn man davon ausgeht, dass beide Leiter unter ungefähr der gleichen Störung beeinträchtigt waren. TTL existiert schon seit den 60er Jahren und ist einer der kompatibelsten, billigsten und verfügbarsten Standards, welcher aber keinerlei störungstolerierende Mechanismen aufweist.

1.2.3 Byte-Stuffing

Beim Verschicken von Paketen auf einem seriellen Bus muss erkannt werden, wo ein Paket endet und das nächste anfängt, da es vorkommen kann, dass im Payload gleiche Bits wie im Header des Paketes gebraucht werden. Das kann zu einer fehlerhaften Paketgliederung führen, sodass der Empfänger entweder nur einen Teil des Paketes als ein solches erkennt, oder aber den Inhalt des ganzen Paketes in zwei oder mehrere Pakete aufteilt. Damit sich der Inhalt des Payloads wirklich eindeutig von dem Header unterscheiden lässt, wird beim Byte-Stuffing vor jeder im Payload vorkommenden Bitsequenz, die dem Header ähnelt, ein vordefiniertes Escape-Zeichen angehängt. Das Escape-Zeichen ist auch ein Byte. Dieses kann dann später beim Analysieren des Empfängers wieder herausgelesen und entfernt werden. Das Escape-Zeichen selber darf auch nicht im Payload vorkommen, um zu verhindern, dass dieses gelöscht wird. Diese Problematik kann gelöst werden, indem ein weiteres Escape-Zeichen vor diesem (im Payload vorkommenden Escape-Zeichen) angebracht wird. Jetzt weiß der Empfänger, dass nach jedem Escape-Zeichen ein Byte folgt, welches entweder dem Flag oder dem Escape-Zeichen selber äh-

nelt, aber nicht als ein solches behandelt werden darf.
Im Folgenden wird das Byte-Stuffing noch an einem kleinen Beispiel veranschaulicht, wobei das D für ein beliebiges Datenbyte (außer dem Escape- oder Flagzeichen) steht. Folgende Bytesequenz möchte übertragen werden:

FLAG	D	ESC	ESC	D	D	D	D

Tabelle 1.6: Nachricht ohne Byte-Stuffing

Aus den oben genannten Regeln wird dann ein Paket erzeugt:

FLAG	HEADER	ESC	FLAG	D	ESC	ESC	ESC	ESC	D	D
D	D	TRAILER	FLAG							

Tabelle 1.7: Nachricht mit Byte-Stuffing

Der Empfänger macht einen ähnlichen Vorgang rückwärts. Er liest solange den Bus bis ein Flag erscheint. Dies signalisiert den Start eines Pakets. Im HEADER und TRAILER befinden sich Metainformationen zu dem Paket u.a. Prüfbits, Sender und Fehlercodes. Wenn ein Escape-Zeichen erscheint, dann löscht der Empfänger dieses und behandelt das nächste Byte als Datenbyte. Also wird das nächste Byte weder gelöscht noch als Ende des Pakets angesehen.[CB99]

1.3 Referenzarbeiten

Bei meiner Literaturrecherche bin ich auf FTA und MBD aufmerksam geworden [LGTL85, SMW05]. Dies sind zwei Verfahren für die Fehlererkennung.

Bei der Fault Tree Analysis (FTA) wird ein Baum erzeugt, welcher in der Wurzel den Fehler enthält. Die Wurzel wird dann in mehrere Events unterteilt, welche Ursachen für den Wurzelfehler sind. Diese Events werden logisch verknüpft, um zu einem übergeordnetem Event oder dem Wurzelevent zusammengeführt zu werden. Unter Nutzung von FTAs müssen bei neuen Erkenntnissen nur leichte Abänderungen in der Baumstruktur erzeugt oder der Funktionsumfang erweitert werden. Die Grundstruktur des Programms änderts sich nicht. Durch die deduktive Fehlererkennung wird ersichtlich, welche boolesche Verknüpfung von Events zu welchem Fehler führen. Zuvor unersichtliche Abhängigkeiten von Fehlern können durch Vergleiche von FTAs im Entwicklungsprozess entdeckt werden. Leider muss für eine gute FTA ein umfangreiches Wissen über das zu untersuchende System gesammelt werden. Der FTA-Baum wird sich also während der Entwicklung höchstwahrscheinlich mehrmals ändern und abhängig vom Projekt sehr

schnell wachsen. Zusätzlich erwarten die Verknüpfungen der Events in den FTA-Ästen immer boolesche Antworten. Für Events, welche nur mit einer gewissen Wahrscheinlichkeit vorhergesagt werden können, bietet der klassische FTA also keine Lösung.

Model Based Disgnosis (MBD) ist eine Form der Analyse, bei der kein Wissen der genauen Ursachen eines Fehler erforderlich ist. Lediglich die objektiven Äußerungen der Messergebnisse wie z.B. Spannung, Messzeit etc. und der aktuelle Zustand des Systems können Rückschlüsse auf einen Fehler mit möglicher Ursache liefern. Für die Fehleranalyse wird dann die Diskrepanz des erwarteten und des gemessenen Zustands des Systems verglichen.

Die MBD sollte ursprünglich als Alternative für Systemanalysen ohne Expertenwissen dienen. MBD reagiert sehr felxibel auf neue Fehler. Allerdings gibt es keine Prioritäten in den Gewichtungen und Genauigkeiten der Messungen. Ob das Modell überhaupt der Norm entspricht, lässt sich leider auch nicht feststellen.

2 Analyse

In der Analyse werden zuerst die Dynamixel-API und deren wichtigste Bestandteile eingeführt und die für die Arbeit relevanten Schnittstellen (Leseoperationen) verglichen. Danach werden zuerst die Daten und anschließend die Antwortzeiten und Spannungen aller Motoren untersucht.

2.1 Vorgehen

Um die Hardwarefehler zu erkennen, nutze ich das Dynamixel-SDK. Das SDK gibt nur Verarbeitungsfehler bei Paketen an, muss also noch erweitert werden. Erst wenn sich die genauen Pakete auslesen lassen, lässt sich ein guter Überblick über alle Komponenten gewinnen. Danach werde ich die Rohdaten des Busses einmal ohne, und dann jeweils mit den fehlerhaften Motoren messen. Durch das künstliche Hervorrufen von Fehlern lässt sich hoffentlich erkennen, ob und wie sich diese äußern. Auf Basis dieser Diagnose kann nun ein Programm entwickelt werden, welches diese Äußerungen berücksichtigt. Dabei sind nur so viele Äußerungen zu beachten, bis alle Fehler klar unterscheidbar sind. Je spezieller die Abfragen bei der Fehlererkennung werden, umso sicherer wird zwar die Erkennung der Fehler, aber gleichzeitig fliegen alle Motoren mit ähnlichen Fehlern aus der Analyse, die nur leicht von den Charakteristiken abweichen. Es ist also wichtig, ein angepasstes Maß an Konkretisierung bei der Fehlererkennung zu finden. Für die Erkennung der langsam antwortenden Motoren, lässt sich das Return Delay-Register einfach auslesen, aber trotzdem werde ich auch das Zeitverhalten für jeden Motor, Fehler und Lesevorgang messen, um u.a. festzustellen, ob noch andere Verzögerungen wann und wo auftreten können. Im weiteren Verlauf der Arbeit werde ich Funktionsnamen der API oder aus eigener Entwicklung **fett** und Inhalte dieser Funktionen *kursiv* hervorheben.

2.2 Aufbau

Für die Analyse habe ich die meiste Zeit den USB2DYNAMIXEL-Adapter genutzt, mit dem Dynamixel direkt vom PC aus genutzt werden kann. Die Verbindung von dem USB2DYNAMIXEL zu dem PC läuft über USB, während sich die andere Seite dann über RS232, RS485 oder TTL mit der Dynamixel Hardware verbinden lässt. Die Motoren brauchen aber zusätzlich noch eine Stromzufuhr, welche sinnvollerweise entweder vor dem ersten oder hinter dem letzten Motor erfolgt. Der Anschluss der Stromzufuhr spielt eine entscheidende Rolle beim Orten von Wackelkontakten oder komplett losen Kabeln. Ich

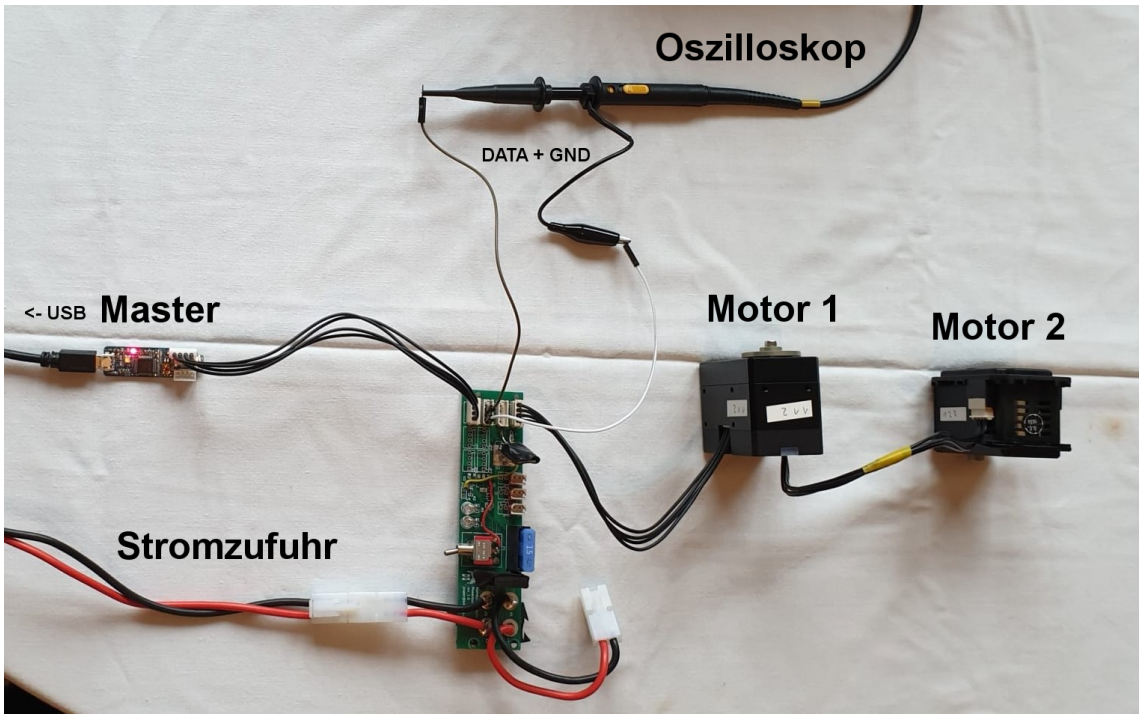


Abbildung 2.1: Aufbau für die Analyse: Zwei oder mehr Motoren werden über RS-485 an den Bus angeschlossen. Der U2D2_INT (Master) ist per USB mit dem Rechner, aber auch wie die Motoren mit dem Bus verbunden. Ein Oszilloskop misst die Spannung auf dem Bus (siehe Messspitze auf dem Bild). Eine Platine mit RS-485 Anschlüssen, einer Stromzufuhr und einem Schalter, um diese zu aktivieren, befindet sich in der Mitte des Bildes.

werde deshalb in der weiteren Analyse one-sided als Kürzel für einen Stromanschluss am letzten Motor (im Falle der Abbildung: Motor 2) und double-sided für einen Anschluss vor dem ersten Motor vom Adapter aus gesehen, nutzen (wie in der Abbildung dargestellt).

2.3 Grundlegende Struktur der API

In der API findet man zunächst eine (robotis_def.py) Datei mit grundlegenden, vordefinierten Werten (Fehlercodes, Instruktionscodes) und arithmetischen Operationen.

Als nächstes gibt es den Port-Handler (port_handler.py), in dem u.a. der Port geöffnet, geschlossen und Timeouts gesetzt werden können. Dieser ist essenziell, um Kommunikation zwischen Master und Motoren überhaupt zu ermöglichen.

Der Packet-Handler (packet_handler.py) ist für das Empfangen und Senden von Paketen zuständig. Dieser gibt in Abhängigkeit der Protokollversion (1.0 oder 2.0) ein anderes Objekt zurück. Es gibt also noch zwei Klassen entsprechend für die beiden Protokolle.

Ich werde ausschließlich den Packet-Handler 2.0 nutzen. Der Hauptunterschied der beiden Versionen besteht in dem größeren Funktionsumfang der neueren Version (u.a. der Kompatibilität mit den neuen Sync- und Bulk-Klassen) sowie der hinzugekommenen PID-Steuerungen, die sehr präzise Bewegungen ermöglichen.

Daneben gibt es noch vier weitere Klassen, die Bulk-, Sync Reads und Bulk-, Sync Writes erleichtern sollen (group_bulk_read.py, group_bulk_write, group_sync_read.py, group_sync_write). Diese vier Klassen benutzen jeweils den Port- und Packet Handler, lesen damit dann die Daten von dem Bus ein und aktualisieren damit zu jedem Motor dann den zugehörigen Eintrag in einem Dictionary. Die Daten können dann natürlich auf Verfügbarkeit überprüft und ausgelesen werden. Außerdem wird das Instruktionspaket nur einmal erzeugt. Falls neue Motoren hinzugefügt werden sollen, muss das Instruktionspaket nur einmal aktualisiert werden.

Der Port- und Packet-Handler sowie alle vier Klassen für die Bulk- und Sync Operationen werden dann in die Initialisierungsdatei (__init__.py) importiert.

2.4 Der Packet-Handler

Der Packet-Handler ist ein wichtiger Bestandteil für die Fehlererkennung. Ich habe in der folgenden Grafik die Funktionsabhängikeiten des Packet-Handlers visualisiert, um einen besseren Überblick der Funktionalitäten und möglichen Kandidaten für die Fehlererkennung zu bekommen.

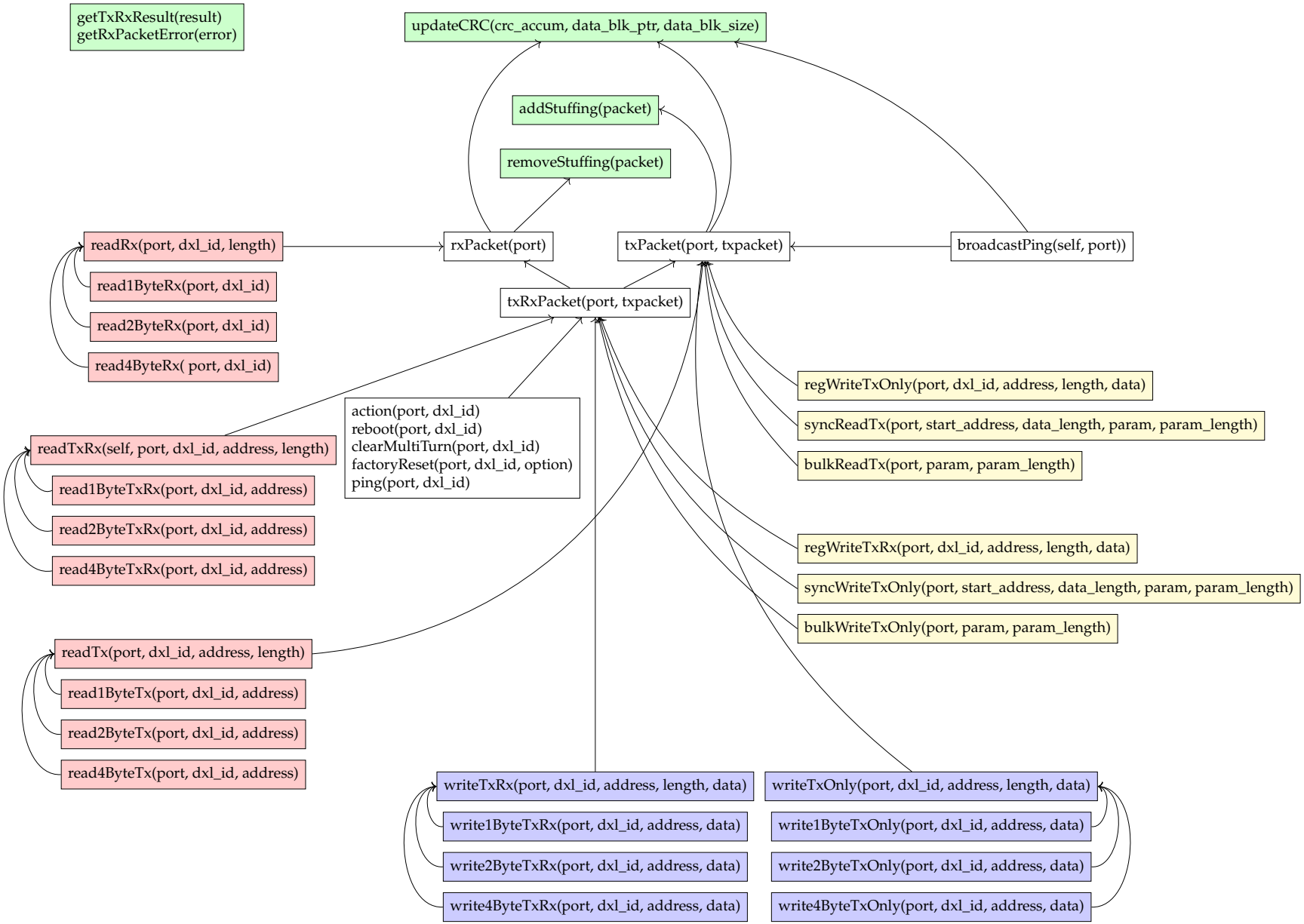


Abbildung 2.2: Übersicht der Funktionen und Funktionsabhängikeiten des Packet-Handlers

Grün gekennzeichnet sind die grundlegenden Funktionen der Paketverarbeitung (CRC, Stuffing und Ergebnis- und Fehlercodes). Diese werden von den drei Funktionen **rxPacket**, **txPacket** und **txRxPacket** genutzt. Auf diesen Funktionen basieren alle anderen Lese- und Schreibfunktionen.

txPacket versendet ein Paket, während **rxPacket** ein Paket von dem Bus einliest. Da für den meisten Gebrauch die Motoren so eingestellt sind (Status Return Level-Register), dass sie auf jegliche Anfragen auch antworten, ist es sinnvoll beide Funktionen in **txRxPacket** zu vereinen. **txRxPacket** sendet also ein Paket und wartet dann auf eine Antwort.

Dabei empfängt **rxPacket** zwar alle Daten, verarbeitet aber nur Pakete die gültig sind. Ist ein Paket nicht komplett oder enthält Fehler, werden nur die Fehlercodes in Strings verarbeitet, die dann ausgegeben werden können.

BroadcastPing und **rxPacket** sind die einzigen Funktionen, welche Daten lesen. Da der **BroadcastPing** nur den Port und keine ID braucht, eignet er sich gut zum Suchen von Motoren. Er ist im Gegensatz zu **rxPacket** keine interne Methode, welche nur eine Teilfunktionalität für übergeordnete Methoden liefert, sondern wird als öffentliche Schnittstelle der API angeboten. Eine weitere Unterteilung in der Grafik erfolgt noch in rot für Lese- und blau für Schreibfunktionen. Diese Funktionen lesen oder schreiben auf 1,2 oder 4 Bytes des RAM oder nicht flüchtigem EEPROM eines Motors. Gelb markiert sind die drei spezielleren Funktionalitäten **regWrite**, **syncWrite** und **bulkWrite**. Sync- und Bulkoperationen werden genauer im Abschnitt 2.4.1 erklärt. **RegWrite** ist eine Schreiboperation, die nicht direkt ausgeführt wird, sondern zwischengespeichert und dann durch die Anweisung **aktion** ausgeführt werden kann. Dies soll eine bessere Synchronisation der Schreiboperation erlauben, hat für meine Arbeit aber jetzt keinen speziellen Nutzen gefunden. Die Abkürzungen **tx** oder **rx** stehen dabei auch hier wieder für das Senden (transmit) oder Empfangen (receive) von Paketen. Um den Sende- und Empfangsprozess besser zu verstehen, nehmen wir den **BroadcastPing** genauer unter die Lupe:

```
1  data_list = {}
2
3  STATUS_LENGTH = 14
4
5  rx_length = 0
6  wait_length = STATUS_LENGTH * MAX_ID
7
8  txpacket = [0] * 10
9  rxpacket = []
10
11 tx_time_per_byte = (1000.0 / port.getBaudRate()) *10.0;
12
13 txpacket[PKT_ID] = BROADCAST_ID
14 txpacket[PKT_LENGTH_L] = 3
15 txpacket[PKT_LENGTH_H] = 0
```

```
txpacket[PKT_INSTRUCTION] = INST_PING
```

Listing 2.1: Initialisierung im Broadcastping

Zu Beginn wird in dem Broadcastping eine leere Liste initialisiert (*data_list*). Diese soll später den Inhalt der ganzen Statuspakete aller Motoren enthalten. *STATUS_LENGTH* ist eine Konstante, welche die Länge eines erwarteten Statuspakets angibt (HEADER0, HEADER1, HEADER2, RESERVED, ID, LENGTH_L, LENGTH_H, INST, ERROR, PARAM1, PARAM2, PARAM3, CRC16_L, CRC16_H). Durch die drei Parameter des Statuspakets kommt man dann auf eine Gesamtlänge von 14 Bytes. Die ersten beiden Parameter geben die Modelnummer an und in dem letzten wird die Version der Firmware angegeben.

rx_packet ist ein Array, welches alle empfangenen Bytes enthält. *rx_length* gibt die Länge dieses Arrays an. Die Länge wird sich während des Programms mehrmals ändern, da aus dem Array auch die verarbeiteten Bytes wieder gelöscht werden.

Die *wait_length* gibt an, wie viele Bytes erwartungsgemäß empfangen werden. Dieser entspricht der Länge eines Pakets multipliziert mit der höchstmöglichen ID (252).

Jetzt wird das *txpacket*, also das Paket zum Senden des Pings vorbereitet. Die variablen Paketinhalte, also *PKT_ID*, *PKT_LENGTH_L*, *PKT_LENGTH_H* und *PKT_INSTRUCTION* werden dann gesetzt und mit **txPacket** verschickt. Beim Aufruf von **txPacket** werden noch Stuffing- und CRC-Bytes eingefügt. Man beachte, dass beim Broadcastping zwar nur ein Ping, aber mit der BROADCAST_ID gesendet wird. So reagiert jeder Motor auf das Paket.

```
result = self.txPacket(port, txpacket)
if result != COMM_SUCCESS:
    port.is_using = False
    return data_list, result
#set rx timeout
#port.setPacketTimeout(wait_length * 1)
port.setPacketTimeoutMillis((wait_length * tx_time_per_byte) +
    (3.0 * MAX_ID) + 16.0);
```

Listing 2.2: Übertragen des Broadcastpings

txPacket gibt das Ergebnis des Sendeprozesses zurück. Das Ergebnis liefert Meldungen über Fehler, die beim Sendevorgang passieren können z.B. wenn der Port schon gebraucht wird oder das Paket zu groß zum Versenden ist.

Falls also das Paket nicht erfolgreich versendet werden konnte, wird der Port geschlossen und die leere *data_list* zusammen mit dem beim Senden entstandenen Fehler ausgegeben.

Das Timeout für die Pakete wird auf die Anzahl der erwarteten Bytes, multipliziert mit der Zeit pro Byte gesetzt, um möglichst alle Pakete zu empfangen.

```
while True:
    rxpacket += port.readPort(wait_length - rx_length)
    rx_length = len(rxpacket)

    if port.isPacketTimeout():  # or rx_length >= wait_length
        break

port.is_using = False

if rx_length == 0:
    return data_list, COMM_RX_TIMEOUT



while True:
    if rx_length < STATUS_LENGTH:
        return data_list, COMM_RX_CORRUPT

    # find packet header
    for idx in range(0, rx_length - 2):
        if (rxpacket[idx] == 0xFF and rxpacket[idx + 1] == 0xFF and
            rxpacket[idx + 2] == 0xFD):
            break

    if idx == 0:  # found at the beginning of the packet
        # verify CRC16
        crc = DXL_MAKEWORD(rxpacket[STATUS_LENGTH - 2],
            rxpacket[STATUS_LENGTH - 1])

        if self.updateCRC(0, rxpacket, STATUS_LENGTH - 2) == crc:
            result = COMM_SUCCESS

            data_list[rxpacket[PKT_ID]] = [
                DXL_MAKEWORD(rxpacket[PKT_PARAMETER0 + 1],
                    rxpacket[PKT_PARAMETER0 + 2]),
                rxpacket[PKT_PARAMETER0 + 3]]

            del rxpacket[0: STATUS_LENGTH]
            rx_length = rx_length - STATUS_LENGTH

            if rx_length == 0:
                return data_list, result

        else:
            result = COMM_RX_CORRUPT

            # remove header (0xFF 0xFF 0xFD)
            del rxpacket[0: 3]
            rx_length = rx_length - 3

```

```
51      else:
52          # remove unnecessary packets
53          del rxpacket[0: idx]
54          rx_length = rx_length - idx
55
56  # FIXME: unreachable code
57  return data_list, result
```

Listing 2.3: Empfangen und Analysieren der Antworten des Broadcastpings

In der nächsten while-Schleife werden solange Bytes am Port gesammelt, bis das *Packet-Timeout* erreicht wurde. Wir warten in dieser Schleife also immer gleich lang auf eine Antwort. Solange das PacketTimeout noch nicht erreicht wurde, wird auf die noch fehlende Länge (Differenz zwischen erwarteter Länge der Antwort und jetziger Länge der Antwort) der Antwort gewartet.

Nach der Auslese wird der Port wieder freigegeben und es kann mit der Verarbeitung von *rxpacket* weitergehen.

Dann wird überprüft, ob Pakete angekommen sind. Wenn ja, werden diese in der nächsten While-Schleife versucht erkannt zu werden.

Falls der übrig geliebene Rest in dem *rx_packet* kürzer als 14 Bytes lang ist, kann dies kein gültiges Paket mehr sein, wird ignoriert und die *data_list* zusammen mit dem Fehler ausgegeben.

Falls *rxpacket* also mindestens 14 Bytes enthält, wird nach dem Header (0xFFFFFD) gesucht. Wird dieser nicht gefunden oder befindet sich nicht an erster Stelle, wird alles bis dorthin gelöscht. Beim Nichtfinden würde dann das ganze Array gelöscht werden.

Wenn man einen gültigen Header vor sich liegen hat, werden die Bytes an den Stellen 12 und 13 wo die Paritätsbits für die zyklische Redundanzprüfung (CRC) liegen sollten, überprüft, indem die Parität der vorherigen Bytes neu berechnet und verglichen wird.

Wenn der Vergleich erfolgreich war, werden aus dem Paket die Parameter herausgelesen und an *data_list* angehängt. Dann wird das gefundene Paket aus *rxpacket* gelöscht und falls dies das letzte Paket aus *rxpacket* war, terminiert die Schleife. Falls nicht, geht es wieder mit dem Rest in *rxpacket* weiter.

Da *result* nur eine Meldung zur Zeit tragen kann, ist es gut möglich, dass sich während der Verarbeitung von *rxpacket* die Meldung von *result* mehrmals zwischen *COMM_SUCCESS* und *COMM_RX_CORRUPT* ändern kann. Es wird nur die Meldung des letzten Pakets ausgegeben. Dies lässt sich auch überprüfen, indem ein funktionstüchtiger und ein Motor mit beschädigten Paketen (z.B. durch Wackelkontakt) an den Bus angeschlossen werden. Vertauscht man die IDs in dem Bus, ändert sich auch der Inhalt von *result*, da die Lesereihenfolge der Motoren im Broadcastping von der ID abhängt (siehe 2.4.2).

2.4.1 Sync Read und Bulk Read

Wie in 2.3 schon angemerkt, stößt man in der API auf zwei Instruktionen namens Sync- und Bulk Read. Sync- und Bulk Read sind beide sehr ähnlich. Grundsätzlich soll von

mehreren Motoren gleichzeitig gelesen werden. Beim Sync Read werden von mehreren Motoren gleiche Bytes ausgelesen. Dazu wird in den Parametern jede ID spezifiziert, die angesprochen werden, und dann noch der Bereich der Daten, der aus den Motoren gelesen werden soll. Die Dauer eines Sync Reads ist deswegen von der Länge des Bereichs und von der Anzahl der IDs abhängig.

Beim Bulk Read wird auch von mehreren Motoren gelesen, aber es kann einzeln für jede ID der Lesebereich genau spezifiziert werden. Für die Fehlererkennung kann Sync Read interessant sein, wenn später in Echtzeit Statuspakete von allen Motoren gebraucht werden.[Rob]

2.4.2 Broadcast- vs Direktping

Wie in der Einleitung zum Dynamixelprotokoll schon erwähnt, kann man statt einem gezielten Ping zu einer einzelnen ID auch einen Broadcastping durchführen. Hierzu ändert man einfach die ID zu der Broadcast-ID 254 (0xFE) um. Tatsächlich wird bei dieser ID aber nicht nur die Reichweite vergrößert, sondern auch eine Wartefunktion innerhalb des Motors genutzt. Damit also nicht jeder Motor gleichzeitig den Bus benutzt, wird hier für die $n = 252$ möglich anzuschließenden Motoren eine Zeitspanne bereitgestellt, in der jeweils immer ein Motor den Bus nutzen kann. Der Motor mit der ID i schreibt dann mit der konstanten Wartezeit d im Zeitraum von $(i-1) * d$ bis $i * d$. Auch wenn nur wenige Motoren angeschlossen sind, dauert ein Broadcastping also immer $n * d$, da ja die Anzahl und IDs der Motoren unwissend sind. Man findet auch in dem Quellcode für das Warten auf die Statuspakete folgenden Ausdruck:

```
wait_length = STATUS_LENGTH * MAX_ID
...
tx_time_per_byte = (1000.0 / port.getBaudRate()) *10.0;
...
port.setPacketTimeoutMillis(
(wait_length * tx_time_per_byte) + (3.0 * MAX_ID) + 16.0
);
```

Listing 2.4: Dauer eines Broadcastpings

Bei einer Baudrate von 2000000 und einer *MAX_ID* von 252 erhält man eine erwartete Dauer von 789,64 ms, welche durch eine selbst durchgeführte Messung bestätigt werden konnte. Dies macht einen Broadcastping zu einer äußerst kostspieligen Funktion.

Über die Dauer des direkten Pings findet man:

```
if txpacket[PKT_INSTRUCTION] == INST_READ:
            port.setPacketTimeout(
            DXL_MAKEWORD(txpacket[PKT_PARAMETER0 + 2],
            txpacket[PKT_PARAMETER0 + 3]) + 11)
        else:
```

```
6             port.setPacketTimeout(11)
```

Listing 2.5: Dauer eines direkten Pings

Da bei einem direkten Ping *txpacket[PKT_INSTRUCTION] = INST_PING* gesetzt wird, werden hier also nur 11 Bytes gewartet, was $11 \cdot (\frac{1}{200}) + 16 * 2 + 2 = 34,055ms$ entspricht. Auch dies lässt sich in etwa bestätigen ($34,325ms$). Natürlich tritt dieser Timeout nur bei fehlerhaften Pings auf. Kommt das Paket früher an, ist die gesamte Dauer deutlich geringer ($\varnothing 0,6725ms$).

Im Durchschnitt dauert also der Broadcastping in jedem Fall mehr als 23 mal länger als ein direkter Ping. Der Broadcastping dauert dabei unabhängig von Erfolg oder Misserfolg immer gleich lang. Wenn aber der direkte Ping erfolgreich ist, ist dieser sogar mehr als 1000-mal schneller als der Broadcastping. Für zusätzliche Zeitmessungen siehe 2.7.

2.5 Fehlertypen

Im Folgenden betrachte ich jeden beobachteten Fehler. Am Anfang werden, falls möglich, die durch den Fehler erzeugten Daten markiert und die Äußerungen beschrieben. Danach wird dann auf die Eigenschaften eingegangen und erläutert, ob dieser Fehler erkannt werden kann.

2.5.1 Dauerhafter Verbindungsverlust

Erscheinungsbild

One-sided: [0], [0,0], [0, 0...,0]
Double-sided: [255, 255, 253, 0, 107, 7, 0, 85, 0, 65, 1, 44, 93, 231]

Erkennbar (one-sided) im Broadcastping ist eine getrennte Verbindung durch eine Anzahl an leeren Bytes, die beim Mastersystem ankommen, während double-sided dann einfach nur die Motoren fehlen, die durch das lose Kabel nicht mehr mit dem Bus verbunden sind. Double-sided kann aber natürlich auch eine Null erreichen, wenn das erste Kabel von dem Adapter nicht mit dem ersten Motor verbunden ist.

Eigenschaften und Identifizierung

Normalerweise wird nur eine Null durch einen Broadcastping (one-sided) empfangen. Es können aber auch zwei oder mehr ankommen, wenn bei Motoren oder dem Netzteil die Spannung schwankt. Dieser Effekt kann auch selbst reproduziert werden, indem während des Broadcastpings lose Kabel an den Bus angeschlossen werden (mehr zu den Ursachen im Abschnitt 2.6).

Ein Verbindungsverlust lässt sich also one-sided erkennen, wenn nur Nullen und keine anderen Daten den Master erreichen. Double-sided gibt es in dem Fall eines Verbindungsverlusts kein Signal mehr zu allen Motoren ab dieser Verluststelle. Kennt man also die Anzahl der Motoren, kann man bestimmen, welches Kabel nicht verbunden ist, indem diese Anzahl mit der Anzahl der aktuell erreichbaren Motoren verglichen wird.

2.5.2 Wackelkontakt

Erscheinungsbild

[0, 0, 0, 0, 0, 0, 0, 12, 0, 222, 0, 255, 255, 253, 0, 1, 7, 0, 85, 0, 55, 1, 42, 154, 192, 0]

Ein Wackelkontakt lässt sich entweder durch unzuordbare Daten erkennen, die neben den vollständigen Paketen noch auf dem Bus liegen, oder durch einen dauerhaften Verbindungsverlust, welcher zeitlich unregelmäßig auftritt.

Eigenschaften und Identifizierung

Wenn sich genau in dem Moment, wo der Master den Bus liest, das Kabel gerade löst oder wieder verbindet, können durch das hohe elektrische Potential Entladungen in Form von kleinen Blitzen über den Bus laufen. Wenn das passiert, enstehen dadurch Daten auf dem Bus, die nicht zu Paketen gehören, oder aus denen man irgendeinen anderen inhaltlichen Schluss auf die Daten der Pakete ziehen kann.

Ein, wie in der rechten Grafik der Abbildung 2.3 durch einen Wackelkontakt zerstörtes Paket, lässt sich im Nachhinein durch CRC überprüfen und erkennen. Allgemein fällt in beiden Abbildungen auf, dass der Wackelkontakt bezogen auf den Byteinhalt sehr stark zwischen 0 und 255 schwankt. Erwiesenermaßen gelingt einem Motor mit einem Wackelkontakt hin und wieder ein beabsichtigtes und somit korrektes Byte, welches aber unnütz ist, da es nur ein Teil von einem Paket ist. Das 16te Byte der rechten Grafik der Abbildung 2.3 ist ein solches Beispiel. Es liefert ein Paketbyte (128), welches Teil des gültigen Pakets des zweiten Motors ist.

Durch die Abbildung 2.4 bekommt man einen Eindruck, wie häufig bestimmte Paketlängen bei Wackelkontakten auftreten können. Jetzt kann man erkennen, wie der Wackelkontakt die Längen der Pakete beeinflusst und dass Paketlängen über 60 Bytes bei dieser Messung eine sehr geringe Auftrittswahrscheinlichkeit haben. Da das gültige Statuspaket 14 Bytes beträgt, sind zwei und drei Motoren erst ab einer Gesamtlänge von 28 ($2 \cdot 14$ Bytes) und 42 ($3 \cdot 14$ Bytes) erkennbar.

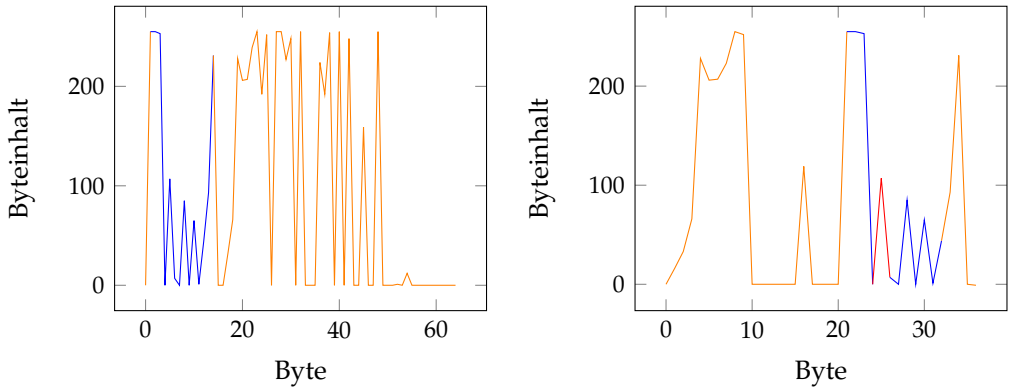


Abbildung 2.3: Zwischen zwei Motoren (double-sided) wird künstlich ein Wackelkontakt (orange gekennzeichnet) erzeugt. Blau markiert ist ein Datenpaket von dem näher am Master liegenden Motor. In der linken Grafik ist das Paket unverletzt, während auf der rechten Seite das Datenpaket durch den Wackelkontakt zerstört wurde (rot gekennzeichnet).

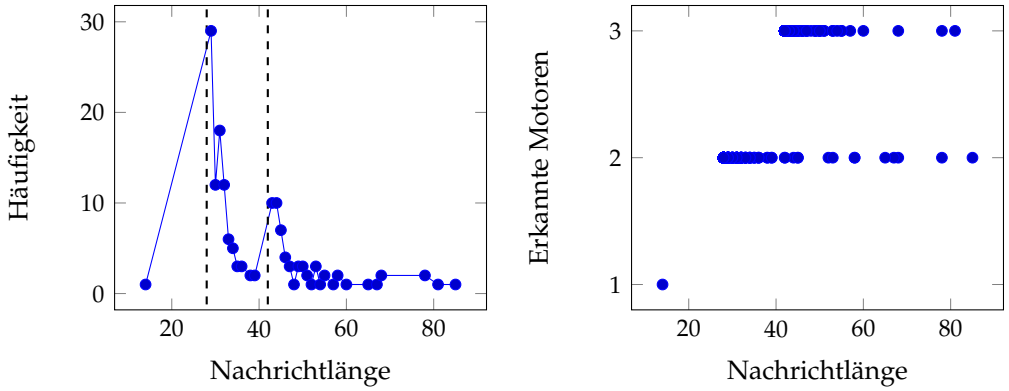


Abbildung 2.4: Bei über 1000 Pings wurden Längen hervorgerufener Wackelkontakte gemessen. Die Wackelkontakte wurden zwischen den letzten beiden Motoren in einer Reihe (double-sided) von drei Motoren künstlich verursacht. Unverfälschte Pakete mit Längen 28 und 42 wurden nur markiert und die Werte dann aus den Daten genommen, da dies die Werte für exakt 2 und 3 erkannte Motoren sind. Auf der linken Grafik erkennt man die Häufigkeiten bestimmter Nachrichtlängen, während auf der rechten die Anzahl der erkannten Motoren in Abhängigkeit der Nachrichtenlängen zu sehen ist.

Für den Versuch in der Abbildung 2.4 ergeben sich 39,01 Bytes für den Durchschnitt und eine Standardabweichung von 11,85 Bytes.

Grundsätzlich gilt, dass ein Wackelkontakt immer, wenn er im richtigen Moment auftritt, Pakete auf dem Bus zerstören kann. Geht allerdings die Spannungsquelle für den Bus vom Master aus, ist die Wahrscheinlichkeit, dass die Motoren, welche näher am Master liegen die Pakete noch erfolgreich durchbringen können sehr hoch (siehe 2.4). Dies bietet eine Möglichkeit den Wackelkontakt nicht nur zu erkennen, sondern auch zu orten.

In dieser Arbeit werde ich versuchen den Wackelkontakt auf beide Arten zu erkennen. Für den ersten Fall muss eine Zeitkomponente mit in das Programm einfließen. Man kann also unmöglich aus nur einem Statuspaket ablesen, ob ein Wackelkontakt vorliegt, wenn dieses nur die Information über eine erfolgreiche oder getrennte Verbindung trägt. Hier wäre es besser eine der effizienteren Alternativen (direkten Ping oder Sync Read) zu verwenden, da diese Analyse während der Laufzeit erfolgen muss. Während dieser Analyse wird dann evaluiert, ob und wie viele Pakete fehlerhaft sind. Wenn kein Paket ankommt, ist der Motor nicht verbunden oder defekt. Kommen alle unversehrt an, scheint es kein Übertragungsproblem zu geben. Wird nur auf einen Teil der Anfragen geantwortet, entsteht der erste Verdacht auf einen Wackelkontakt. Teilt man die Anzahl der Erfolgreichen durch die Gesamtanzahl, erhält man eine Fehlerrate, die angibt welcher Teil der gesamten Pakete erfolgreich war. Jetzt müssen nur noch passende Schwellwerte für die Erkennung gefunden werden, sodass einzelne gängige Paketverluste nicht direkt als Fehler klassifiziert werden.

Allerdings reicht es nicht aus zu wissen, wie viele Pakete nicht angekommen sind, sonst würde immer, wenn ein Kabel komplett entfernt werden würde, nach gewisser Zeit ein Wackelkontakt erkannt werden. Die Streuung spielt auch eine Rolle. Liegen die erfolglosen Pakete also sehr gleichmäßig zerstreut, statt auf einen Punkt gehäuft, ist die Wahrscheinlichkeit größer, dass ein Wackelkontakt vorliegt. Sie sind dann statistisch unabhängiger von den Paketen davor. One-sided oder double-sided unterscheiden sich hier sehr ähnlich wie bei einem dauerhaften Verbindungsverlust. Double-sided lässt sich der Wackelkontakt also orten.

2.5.3 Nicht sendender Motor

Erscheinungsbild

[255, 255, 253, 0, 1, 7, 0, 85, 0, 55, 1, 42, 154, 192, 255, 255, 253, 0, 5, 7, 0, 85, 0, 55, 1, 42, 130, 128]

Anhand der Messdaten des Busses alleine lässt sich keine Aussage über die Existenz eines solchen Motors treffen, denn ein solch angeschlossenen Motor verändert nicht die Busdaten.

Eigenschaften und Identifizierung

Ein Motor, der gar nichts mehr sendet, aber noch Teil des Bussystems ist und Daten von anderen Motoren über sich laufen lässt, kann nicht ohne weiteres erkannt werden. Kennt man jedoch die Anzahl der angeschlossenen Motoren, kann man durch das Ausschlussverfahren bestimmen, welche Motoren sich nicht melden. Diese Motoren werden bei der Erkennung, wenn überhaupt also nur indirekt bestimmt werden können, da sie sich nicht auf dem Bus äußern. Leider ist ein nicht sendender Motor ein großes Problem für die Erkennung eines dauerhaften Verbindungsverlusts (double-sided). Wenn beispielsweise der erste von fünf angeschlossenen Motoren auf einem Bus nicht antwortet, wobei das Kabel zwischen dem dritten und vierten Motor lose oder defekt ist, würden drei Motoren nicht erkannt werden. Es wäre aber nicht möglich einen der nicht erkannten Motoren als nicht verbunden oder nicht sendend zuzuordnen. Würde man die Reihenfolge der Motoren im Bus kennen, könnte zumindest an bestimmten Stellen sicher ausgeschlossen werden, dass es sich um einen dauerhaften Verbindungsverlust handelt.

2.5.4 Rhythmischer Störer

Erscheinungsbild

[0, 255, 255, 253, 0, 1, 7, 0, 85, 0, 55, 1, 42, 154, 192, 0, 255, 255, 253, 0, 5, 7, 0, 85, 0, 55, 1, 42, 130, 128, 0]

Ein rhythmischer Störer ist ein Motor der rhythmisch einzelne Nullbytes erzeugt. Dieser Motor ähnelt grundsätzlich einem nicht sendenden Motor. Zusätzlich findet man aber vor und hinter jedem Paket eines Broadcastpings eine Null.

Eigenschaften und Identifizierung

Das Problem hier ist von elektrotechnischer Natur. Misst man die Spannung auf dem Bus, wenn ein solcher Motor angeschlossen ist, findet man eine deutlich niedrigere Spannung als ohne diesen vor. Mit rhythmisch sind also lediglich die regelmäßigen Abstände der Nullen auf dem Bus gemeint, es beschreibt nicht das Verhalten dieses Fehlers. Der Fehler kann erkannt werden, indem die Busdaten von vorne nach hinten durchgeparst werden. Findet man zu Beginn genau eine Null, so kann der Fehler bestätigt werden, indem nach genau jedem erfolgreichen Paket eine weitere Null gefunden wird.

Dabei muss ein solches Problem nicht durch einen Motor erzeugt werden, sondern kann auch durch eine ungünstige Stromversorgung oder andere Bus-Komponenten entstehen. Ein permanenter Störer (siehe 2.5.5) kann keine Daten mehr auf dem Bus erzeugen, wenn dieser bereits rhythmisch gestört ist. Die Kommunikation zwischen Master und den anderen Motoren kann also ironischerweise durch einen rhythmischen Störer verbessert werden. Natürlich fehlen dann aber Anhaltspunkte für die Erkennung des

verdeckten (permanenten) Störers. Wie der Fehler erzeugt wird und mit dem permanenten Störer reagiert, wird im Abschnitt 2.6 (Spannungsveränderungen durch Störer) genauer erklärt.

2.5.5 Permanenter Störer

Erscheinungsbild

[...12, 6, 0, 12, 2, 4, 198, 0, 0, 255, 255, 253, 0, 1, 7, 0, 85, 0, 55, 1, 42, 154, 192, 0, 0, 0, 0, 0, 96...]

Dieser störende Motor schreibt unabhängig von einem Instruktionpaket trotzdem Daten auf den Bus. Es fällt außerdem auf, dass er dies ununterbrochen bis zum Timeout des Broadcastpings tut (3528 Bytes).

Eigenschaften und Identifizierung

Auch der permanente Störer löst Spannungsprobleme auf dem Bus aus (siehe 2.6). Im folgenden Graphen erkennt man, wie die ersten 500 Bytes eines permanenten Störers aussehen können. Blau markiert ist ein von einem weiteren funktionstüchtigen Motor erzeugtes, unzerstörtes Statuspaket. Durch die Grafik wird bemerkbar, dass der permanente Störer sich wiederholende, sehr ähnliche Chunks an Bytes über den kompletten Broadcastping sendet.

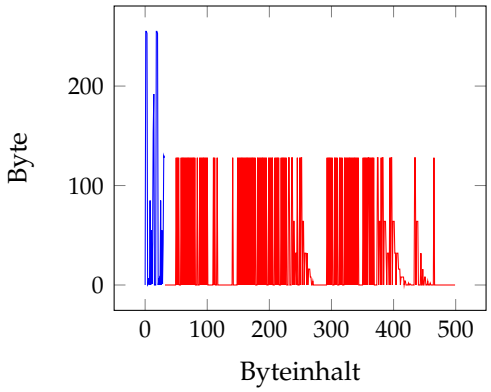


Abbildung 2.5: Die ersten 500 Bytes der durch einen permanenten Störer beeinflussten Busdaten. Neben dem permanenten Störer (rot markiert) befindet sich noch ein gültiges Statuspaket auf dem Bus (blau markiert).

Leider stören Motoren mit diesem Fehler die Kommunikation zwischen anderen Motoren und dem Master. Alle Pakete, welche auf dem Bus liegen, können durch diese Daten zerstört werden. Meine Messungen ergeben, dass bei einem direkten Ping im Durchschnitt 48 von 1000 (4,8 %) Paketen beschädigt sind. Im Gegensatz dazu ist die Wahrscheinlichkeit bei einem Broadcastping, dass die Pakete eines einzelnen Motors nicht vollständig ankommen, etwa 55%. Je mehr Motoren angeschlossen sind, desto höher ist

die Wahrscheinlichkeit, dass der permanente Störer die Daten eines beliebigen Paketes zerstört. Für den folgenden Graphen wurden jeweils 1000 Broadcastpings durchgeführt und dann 1, 2, 3 und 4 funktionsfähige Motoren hinzugeschaltet, um zu überprüfen, wie der Störer die Pakete beeinflusst:

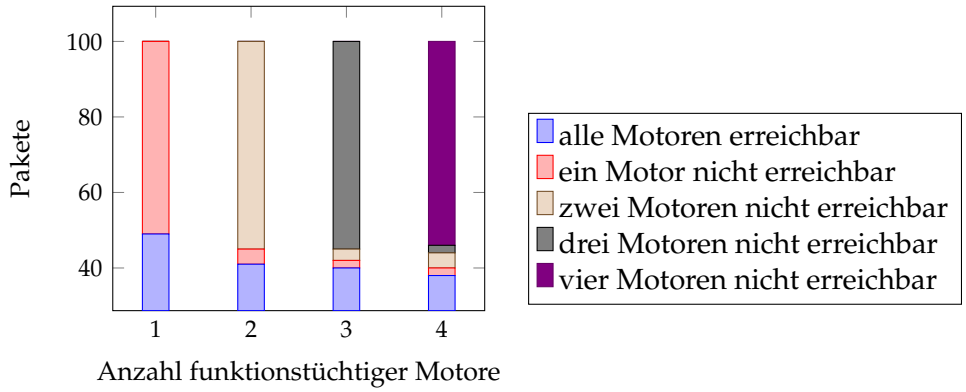


Abbildung 2.6: Fehlerrate von durch einen permanenten Störer beeinflussten Pakete funktionsfähiger Motoren

Erkannt werden kann dieser Fehler, indem die Länge des gesamten Paketes mit der Anzahl der erkannten, funktionstüchtigen Motoren verglichen wird. Alle überzähligen Daten sind Indizien für einen Fehler. Um zwischen permanentem Störer und Wackelkontakt zu unterscheiden, muss aber noch eine weitere Bedingung miteinspielen. Leider lässt sich die Spannung beim Empfangen der Daten nicht direkt über den Master messen, sonst würde man bei einem permanenten Störer eine niedrigere Spannung als bei einem Wackelkontakt feststellen können (siehe 2.6). Eine Klassifizierung kann aber über die Länge der Daten stattfinden: Ein Wackelkontakt kann in demselben Zeitfenster nicht so viele Daten auf den Bus legen wie ein permanenter Störer. Wie bereits im Abschnitt 2.5.2 getestet, konnte ich keinen künstlichen Wackelkontakt mit einer Länge von 100 oder mehr Bytes erzeugen, während der permanente Störer immer bis zum Timeout sendet (3528 Bytes).

Ist ein Kabel in Bewegung wie bei einem Wackelkontakt, ändert sich das elektrische Potenzial ständig. Für eine große Datenmenge müssten die Kabel bei einem Wackelkontakt konstant sehr nahe beieinander sein, damit der Widerstand zwischen den Kabeln sehr gering ist und dann sehr viele Entladungen in einem kleinen Zeitraum passieren können. Dass sich also ein permanenter Störer fälschlicherweise als Wackelkontakt herausstellt, ist sehr unwahrscheinlich, auch bei einer Standardabweichung von 11,85. Man muss bedenken, dass der Roboter immer in Bewegung ist, und nicht nur eine Messung zur festen Klassifizierung von Fehlern führt.

2.6 Spannungsveränderungen durch Störer

Um herauszufinden, wie die Störer funktionieren, wie sie funktionstüchtige Motoren und sich untereinander beeinflussen, werden die Spannungen unterschiedlicher Motoren einzeln und in Kombination gemessen.

Motor	Vpp	Vp+	Vp-	Mean
FM	600-800mV	2,5-3V	2-2,5	2,6V
FMB	5,2-6V	4,7-6,5V	300-500mV	-
PS	400mV	1,12V	560mV	814mV
RS	440mV	2,56V	2,12V	2,35V

Motorkombination	Vpp	Vp+	Vp-
FMB + PS	4-5V	4-5V	-40-440mV
FMB + RS	2-3V	2-3V	-40-440mV
FMB + PS + PS	4V	2,5-3,5V	-120-440mV
FMB + RS + RS	<2V	<2V	<-40mV
FMB + RS + PS	<2V	<2V	<-40mV

Tabelle 2.1: Spannungen unterschiedlicher Motoren und Motorkombinationen (siehe Glossar für Abkürzungen)

Der permanente Störer setzt die ursprünglichen 2,4V der Grundspannung auf 814mV herunter. Dies tut er auch beim Senden und Empfangen von Paketen über den Bus. Dabei fällt auch die Spannung während des Sendens des Pakets von 5-6V auf 4-5V ab, was aber noch keinen Einfluss auf die Paketdaten hat. Zwischen jedem Paket liegt die Spannung aber dann so tief, dass sich für den Master diese nicht mehr in dem Lesebereich befindet. Er versucht dann, falls gerade kein Paketbyte zum Lesen vorhanden ist, die Grundspannung als Wert zu interpretieren. Deswegen erreichen die an dem Master ankommenden Daten auch immer die Maximallänge, wenn ein permanenter Störer angeschlossen ist.

Beim rhythmischen Störer fällt auf, dass die Spannung der Pakete noch tiefer abfällt. Ein Motor sendet dann nur noch mit halb so großer Spannung auf etwa 2-3V. Verbindet man zusätzlich einen permanenten oder rhythmischen Störer, kann sich die Spannung noch tiefer senken, wenn dieser durch noch einen stärkeren Strom das Netz belastet. Bei zwei angeschlossenen rhythmischen Störern fällt die Spannung unter 2,5V und es lassen sich unter der Verwendung von nur einem Datensignal (TLL) schon keine Pakete mehr zu den Motoren durchbringen und diese antworten nicht mehr. Ob sie das Instruktionspaket nicht mehr erkennen, oder die Motoren zu wenig Betriebsspannung zum Senden haben, ist noch unklar.

Es ist offensichtlich, dass sich bei Störern die Spannung auf dem Bus verändert. Mei-

stens lassen sich dann die Pakete noch richtig interpretieren, aber die Spannung zwischen diesen kann den Master beeinträchtigen. Ob und wie der Master diese dann interpretiert, hängt von dessen Widerständen und der Größe des Lesebereichs ab. Z.B. bietet der U2D2_INT noch einen DIP-Switch, welcher bei Bedarf umgeschaltet werden kann. Durch das Umschalten wird ein 120Ω Widerstand zwischengeschaltet, welcher beide Störer ausblenden lässt. Allerdings ist für den U2D2_INT generell kein permanenter Störer erkennbar. Für ihn sind alle Störer rhythmisch. Zwei Störer an unterschiedlich angeschlossenen Mastern können also unterschiedlich interpretiert werden.

Zuletzt muss ich betonen, dass auch nicht jeder Störer jedem anderen derselben Art gleicht. Obwohl sie natürlich dieselben Busdaten auf demselben Master produzieren, können diese durch unterschiedlichste Probleme verursacht werden. Deshalb kann man nicht generell vorhersagen, welche Fehler was für Auswirkungen in Kombination mit anderen Fehlern auf den Bus haben, solange nicht genau bekannt ist, wie der spezifische Fehler hervorgerufen wird.

2.7 Antwortzeiten

Abhängig von dem Motorchip, dem Controller (Master) aber auch vielen kleinen Faktoren, wie Spannungsänderungen, Reihenfolge im Bussystem und Länge des Instruktions- und Statuspakete, kann es Verzögerungen von Antworten der jeweiligen Motoren auf die Anfragen geben. Zusätzlich besitzen die meisten neueren Motoren von Dynamixel ein Return Delay Time-Register. Hier kann manuell eine Verzögerung von 0 bis maximal 508 μs eingestellt werden. Laut dem Dynamixel-Handbuch ist dies die Verzögerung zwischen Ankunft und Absenden des Paketes. Das Return Delay Time-Register erlaubt dem Master des Bussystems genug Zeit, um von Tx (Transmitter) auf Rx (Receiver) umzustellen. Dies kann nämlich je nach Leistung und Beschäftigung des Controllers etwas schwanken. Um herauszufinden wie stark der Einfluss bestimmter messbarer Faktoren ist, werden unterschiedliche Motoren unter diesen Bedingungen getestet und miteinader verglichen. Interessant ist hier in erster Linie das Return Delay Time -Register, das im laufenden Programm erkannt und überprüft werden soll. Trotzdem werden noch die Antwortzeiten unterschiedlicher Fehler, Motorenreihen und Paketlängen gemessen, um einen besseren Überblick der Einflussfaktoren zu bekommen. Wie bereits in dem Paper [BGZ19] beschrieben, muss der USB-Latency-Timer hierfür heruntergesetzt werden, um ungedrosselte Messwerte zu erhalten. Für die erste Messung (Antwortzeiten unterschiedlicher Paketlängen) konnte ich noch den alten USB2Dynamixel verwenden, wobei ich für die restlichen Messungen auf den neueren U2D2_INT umsteigen musste.

Misst man die Antwortzeiten unterschiedlicher funktionstüchtiger Motoren und Paketlängen, dann lässt sich kein Unterschied feststellen, was nicht überraschend ist, da bei einer Baudrate von 2.000.000 theoretisch zwischen einem und drei gesendeten Bytes

nur ein Unterschied von etwa $1,5\mu s$ liegt. Außerdem benutzen alle Motoren denselben Chip. Leider reagiert der U2D2_INT bei einem USB-Latency-Timer von $0ms$ nicht mehr im Gegensatz zu dem alten USB2Dynamixel, weshalb der zweite und dritte Graph mit einer Latenz von $1ms$ gemessen wurde.

Die Streuung der Antwortzeiten der beiden Diagramme hängt sehr stark von dem Controller und der Sendefrequenz der Motoren ab. Jeder Motorchip arbeitet mit einer bestimmten Frequenz. Erreicht ein Instruktionspaket den Motor im ungünstigen Augenblick, kann die Verarbeitung erst im nächsten Zyklus erfolgen. Dies erkennt man gut in der ersten Grafik. In der zweiten hingegen erkennt man viel präziesere Häufungen auf bestimmten Werten. Dies hängt mit der USB-Latenz zusammen. Die Werte liegen nach der ersten Millisekunde am Bus schon bereit und können direkt aufgenommen werden. Abhängig von der Baudrate, können sie sich dann nur noch um Baudzyklen verschieben ($1\mu s$), und sind nicht von der Sendefrequenz des Motors abhängig.

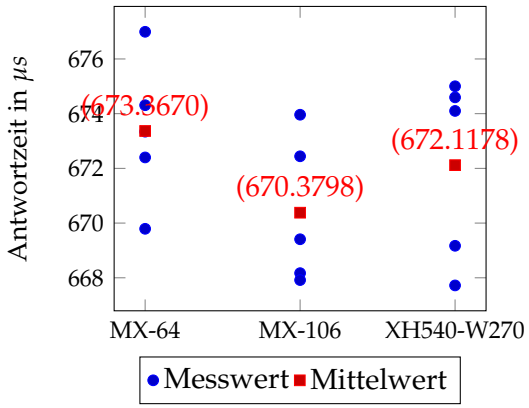


Abbildung 2.7: Antwortzeiten unterschiedlicher Motoren bei einer USB-Latenz von 0ms

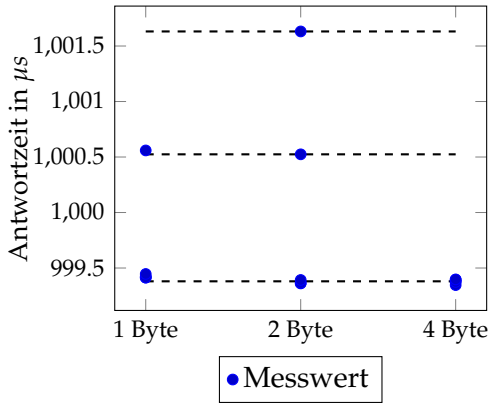


Abbildung 2.8: Antwortzeiten unterschiedlicher Paketlängen bei einer USB-Latenz von 1ms

Jetzt messen wir den Einfluss des Delay Time-Registers auf die Messzeit. Normalerweise würde man hier eine annähernd proportionale Abhängigkeit erwarten. Allerdings

wurde der USB Latency Timer des U2D2_INT auf eine Millisekunde gestellt. Bei einem Delay Time-Registereintrag von 254 (508μs) kann der Kommunikationsprozess nicht mehr in einer Millisekunde stattfinden, wodurch der Master dann am Ende der zweiten Millisekunde auf die Daten reagieren kann. Deshalb findet dort im Graphen ein Sprung statt.

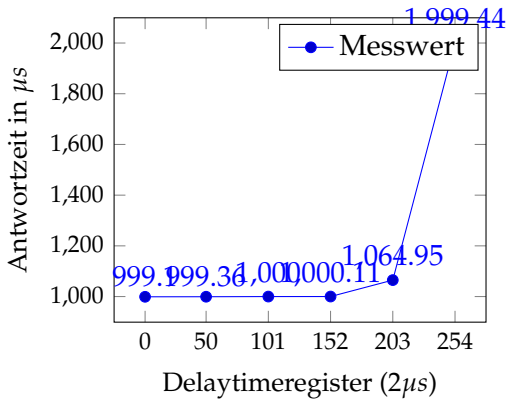


Abbildung 2.9: Antwortzeiten eines Motors mit unterschiedlichen Delay Time-Register einträgen bei einer USB-Latenz von 1ms

3 Entwicklung

Um die Busdaten nun als Fehler zu klassifizieren, benutzen wir einerseits denselben modifizierten BroadcastPing von der Analyse, um am Anfang nach Motoren zu suchen und zusätzlich aber dann double-sided einen modifizierten Sync Read, der im Gegensatz zum normalen Sync Read den kompletten Businhalt ausgibt. One-sided reicht ein normaler Read. Der Inhalt der Busdaten ist entscheidend für die Fehlererkennung. Um den Wackelkontakt über erfolgreiche und nicht erfolgreiche Pakete (also über einen Zeitraum) zu erkennen, werden stochastische Mittel eingesetzt.

Steinbauer et al. [SMW05] haben als Lösungsansätze für eine Fehlererkennung die Fehlerbaumanalyse und modellbasierte Diagnose vorgestellt. Die Fehlerbaumanalyse versucht einen Fehler durch eine boolsche Verknüpfung von mehreren Events zu erkennen. Dabei gibt es mehrere Typen von Events, die so weit es geht immer tiefer zerlegt werden können.

Bei der modellbasierte Diagnose wird hingegen die Norm auf mehrere Arten gemessen und dann untersucht, wie stark die gemessenen Werte von der Norm abweichen. Dabei kann die Norm und unterschiedliche Fehlerkategorien durch Bereiche gekennzeichnet werden.

Ich habe mich entschieden einen Bottom-Up Approach der FTA zu nutzen. Dabei liegen in der Wurzel die Busdaten, welche durch Abfragen in jeweilige Fehler kategorisiert werden können (Äste). Dies liegt sehr nah an dem wirklich vorliegenden Ausgangszustand des Problems in welchem die Busdaten vorliegen und reduziert in diesem Fall die Anzahl der Fehlerbäume, da alle Fehler direkt über die Daten klassifiziert werden müssen. Aus diesem Grund würden die meisten Fehlerbäume ohnehin nur aus einem Ast bestehen. [SMW05, LGTL85]

3.1 Aufbau

Das Programm besteht aus einem **Detector**, welcher die Schnittstelle zur Fehlerkennung bereitstellt. Der **Detector** bietet Methoden zum Setzen von Einstellungen, das Initialisieren und Suchen von Motoren sowie das Lesen und Verarbeiten von neuen Daten. Da das Lesen hier davon abhängt, ob der Anschluss double-sided oder one-sided ist, werden hier je nach Bedarf andere Leser initialisiert, um die gewünschten Daten zu liefern. Die Fehler können am **Detector** durch eine Funktion ausgegeben werden. Bei jedem neuen

Lesevorgang werden die alten Fehler wieder gelöscht, aber über eine einstellbare Dauer rückblickende Verbindungsprobleme für die Analyse auf Wackelkontakte gespeichert.

3.2 Paketanalyse

```
    def scanAndInit(self):
        if self.refMotors:
            x = 0
            while len(self._data_set) < self.refMotors
            and x < self._samplesize:
                self.getData()
                x += 1
        else:
            for x in range(self._samplesize):
                self.getData()
        if self._data_set:
                self.reader.init(self._data_set)




    def getData(self):
        dxl_data, result = self.pkth.broadcastPingRaw(self.ph)
        data, errors = self.pkth.investigateData(dxl_data,
            self.recSingleAsErr)

        for datum in data:
            self._data_set.add(datum)
        for error in errors:
            self._errors_set.add(error)
```

Listing 3.1: Initialisierung des Detectors

In **scanAndInit** wird solange nach Motoren gesucht, bis entweder alle Motoren gefunden worden sind oder die Maximal-Suchdauer erreicht wurde. Dies hängt natürlich davon ab, ob der Nutzer sich am Anfang die Zeit nimmt, die Anzahl der Motoren in dem Parameter *refMotors* mitzugeben. Gibt man die Anzahl als Parameter mit, so wird wahrscheinlich die Motorensuche etwas erleichtert, da schon vor der Maximal-Suchdauer alle Motoren gefunden werden können. Außerdem kann nach der Suche überprüft werden, ob alle Motoren gefunden wurden. Aber auch wenn die Anzahl nicht mitgegeben wurde, kann man während des Betriebs die momentan erkannte Motoranzahl manuell überprüfen. Alle erkannten Fehler und Motoren werden dann in einer Menge gespeichert.

3.3 Untersuchungsvorgang eines Datenpakets

Kommt ein Datenpaket an, muss dieses unabhängig, ob bei der Suche am Anfang oder bei der weiterführenden Analyse danach, überprüft werden. Es kann natürlich auch vorkommen, dass gar kein Motor bei der Suche erkannt wurde. Deshalb wäre es gut auch bei einer erfolglosen Suche aufgefallene Fehler zu sammeln und nicht mit der Analyse erst nach dem Suchvorgang zu beginnen. Der folgende Baum soll verdeutlichen, welche Fehler (außer der zeitlichen Analyse, welche nochmal extra behandelt wird) sich aus möglichen Daten erkennen lassen können.

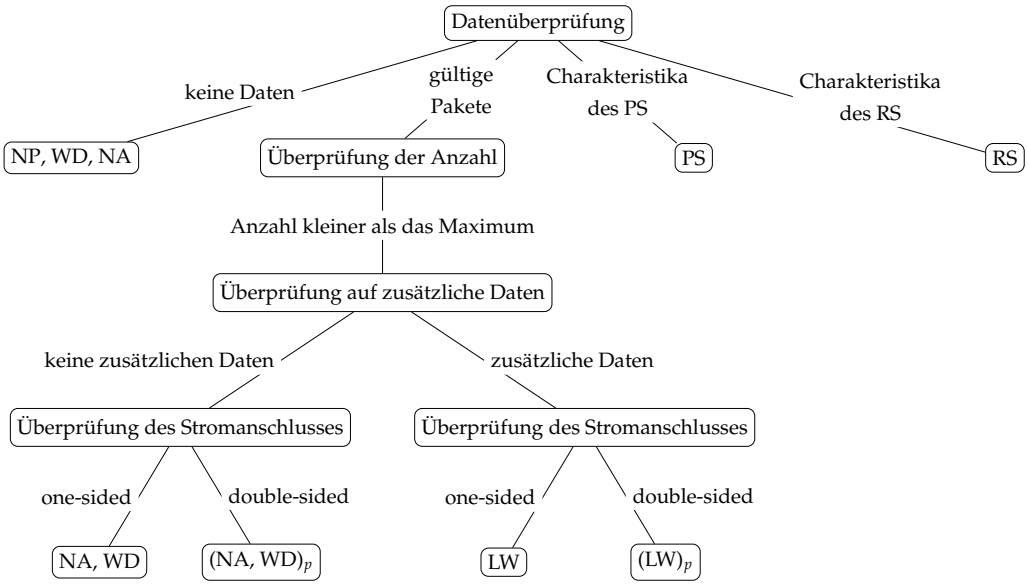


Abbildung 3.1: Fehlerklassifizierungsbaum (siehe Glossar für Abkürzungen)

In der folgenden Funktion **investigateData** werden die Busdaten des Broadcastpings analysiert. Sie setzen damit erfolgreich die Datenüberprüfungen der ersten und dritten Ebene des Baums um. NP, WD und NA können leider noch nicht auseinandergehalten werden. Deshalb fasse ich sie in dem Fehlercode *PH_LOST_SIGNAL* zusammen. Die Untersuchung der Pakete der Reader findet auf ähnliche Weise im Sync Read oder Ping statt.

```
1     def investigateData(self, packet, recSingleAsErr):
2         counter_null = 0
3         STATUS_LENGTH = 14
4         hasSingJam = 0
5         length = len(packet)
6         result = []
7         success = []
8
```

```
        if self.checkZero(packet):
            result.append(PH_LOST_SIGNAL)
            return success, result

        elif length==3528:
            result.append(PH_PERM_JAMMER)

        while len(packet)>= STATUS_LENGTH:
            for idx in range(0, length - 2):
                if packet[idx] == 0:
                    if counter_null == 0:
                        counter_null = 1
                    continue
                elif packet[idx] == 0xFF and packet[idx + 1]//
                == 0xFF and packet[idx + 2] == 0xFD:
                    break
            # verify CRC16
            if idx == 0:
                crc = DXL_MAKEWORD(packet[STATUS_LENGTH - 2],
                packet[STATUS_LENGTH - 1])
                if self.updateCRC(0, packet, STATUS_LENGTH - 2) == crc:
                    success.append(packet[PKT_ID])
                    del packet[0: STATUS_LENGTH]
                    if counter_null == 1:
                        counter_null = 0
                    else:
                        counter_null = -1
            else:
        del packet[0: idx]
        if (self.checkZero(packet) and counter_null == 0
            and recSingleAsErr):
            hasSingJam = 1
            result.append(PH_SING_JAMMER)

        if (length > STATUS_LENGTH * len(success)
            + hasSingJam * (len(success) + 1)):
            result.append(PH_LOOSE_WIRE)
        return success, result
```

Listing 3.2: Untersuchen der Busdaten bei einem Broadcastping

3.4 Datenstreifen und Reader

Nach der Suche und Initialisierung können jetzt während der Laufzeit des Roboters relativ effizient mit einem Sync Read (double-sided) oder sogar nur einem Ping (one-sided) die notwendigen Daten gelesen werden. Je nachdem wie die Spannungsquelle angeschlossen ist, können unterschiedliche Reader verwendet werden. Reader geben an, wie die Daten eingelesen und interpretiert werden sollen. Speichert man diese Daten in

einer Warteschlange mit vordefinierter Größe, kann man in diesem Zeitraum neben **investigateData** auch nach Veränderungen Ausschau halten, die auf einen Wackelkontakt hinweisen. Die folgende Funktion ist Bestandteil eines Readers (double-sided), versucht also auch die Stelle an der das Problem liegt, zu erkennen.

```
def basicRead(self):
        self._errors.clear()
        dxl_comm_result = self.groupSyncReadRaw.txRxPacket()
        self._errors.update(self.groupSyncReadRaw.getErrors())
        if dxl_comm_result != COMM_SUCCESS:
            print("%s" % self.packetHandler
            .getTxRxResult((dxl_comm_result)))

        self.amount = 0
        for id in self._data_set:
            dxl_getdata_result = self.groupSyncReadRaw
            .isAvailable(id, self.ADDR_PRO_PRESENT_POSITION,
            self.LEN_PRO_PRESENT_POSITION)
            if dxl_getdata_result:
                self.amount += 1
                data = self.groupSyncReadRaw
                .getData(id, self.ADDR_PRO_PRESENT_POSITION,
                self.LEN_PRO_PRESENT_POSITION)

        if self.Initialized:

            if self.amount_counter[self.sampledata[0]] > 1:
                self.amount_counter[self.sampledata[0]] -= 1
            else:
                del self.amount_counter[self.sampledata[0]]

            if self.rem_counter > 0:
                self.rem_counter -= 1

            if self.rem_counter == 0:
                self.distances.pop(0)
                if  self.distances:
                    self.rem_counter = self.distances[0]
                else:
                    self.rem_counter = -1



        self.sampledata.append(self.amount)
        if self.amount in self.amount_counter:
            self.amount_counter[self.amount] += 1
        else:
```

```
43              self.amount_counter[self.amount] = 1
44
45          if self.amount < len(self._data_set):
46              self._errors.add(PH_Wire_DISC)
47              self.distances.append(self.distance)
48              self.distance = 1
49              if self.rem_counter == -1:
50                  self.rem_counter = self.distances[0]
51          else:
52              if self.distance >=  self._samplesize:
53                  self.distance =  self._samplesize
54              else:
55                  self.distance += 1
```

Listing 3.3: Beispielimplementation der Lesefunktion eines Readers

Um Rechenzeit zu sparen, sollte nicht nach jedem Sync Read über die Warteschlange der Daten iteriert, sondern nur die Änderungen (das erste und das letzte Element) betrachtet werden. Für die Wackelkontakterkennung werden die Abstände der verlorenen Pakete sowie die Anzahl gebraucht. Soll zusätzlich noch die Position bestimmt werden (nur double-sided möglich), wird zusätzlich noch das Minimum und Maximum der erkannten Motoren sowie die beiden meist auftretenden Häufigkeiten der erkannten Motoren in dem Zeitfenster ausgegeben, damit der Nutzer schnellstmöglich den Wackelkontakt beheben kann.

3.5 Überprüfung auf einen Wackelkontakt

```
1   def readDataStripe(self):
2         self._errors_set.clear()
3
4         self.reader.readDataStripe()
5         self._errors_set.update(self.reader.getErrors())
6
7         distances = self.reader.getDistances()
8         rate = len(distances)/float(self._samplesize)
9
10        if rate > self._rate_bootom_threshold and
11        rate < self._rate_top_threshold and
12        self.checkDistrib(distances) > self._dist_threshold:
13            self._errors_set.add(rd.PH_LOOSE_WIRE2)
```

Listing 3.4: Nutzung des Readers im Detector

Nachdem der Reader die Abstände der fehlerhaften Pakete und die Fehlerrate berechnet hat, kann der Detector dann Vergleiche mit Schwellenwerten durchführen. Wenn die Daten die notwendigen Charakteristiken für einen Wackelkontakt haben, wird der Fehler der Fehlermenge hinzugefügt.

3.6 Stochastische Verteilung beim Wackelkontakt

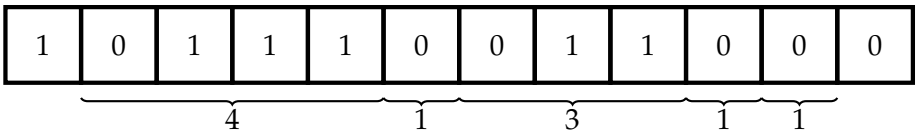
```
1    def checkDistrib(self, distances):
2        if len(distances) > 0:
3            actual = sum(distances)/float((len(distances)))
4            best = (self._samplesize-1)/float(len(distances))
5            return (actual-1)/(best-1)
6        else:
7            return 0
```

Listing 3.5: Stochastische Verteilung eines Wackelkontakts

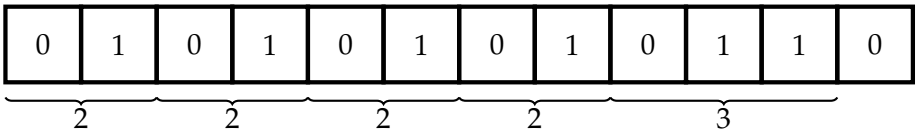
Neben dem Verhältnis von erfolglosen und erfolgreichen Paketen, welches schon in **checkLooseWire** überprüft wurde, spielt auch die Verteilung eine Rolle. Zwar kommt ein Wackelkontakt nicht immer in regelmäßigen Abständen vor, aber eine starke Abhängigkeit, wie z.B. eine Folge von Einsen direkt gefolgt von einer Folge von Nullen kann eher auf ein komplett verlorenes Signal deuten. Hierbei muss das Zeitintervall zwischen Paketen beachtet werden. Für ein Zeitintervall ist es optimal die Bewegungsdauer der Motoren des Roboters als Maß zu nehmen, da ein Wackelkontakt meist zwischen Bewegungen des Roboters auftritt. **checkDistrib** wird von **checkLooseWire** genutzt, und soll anhand der Anzahl der Pings und den Stellen der Fehlschläge die Verteilung berechnen. Hierfür wird die stochastische Abhängigkeit von einzelnen Fehlschlägen zueinander berechnet, indem über den Abstand gemittelt wird und mit dem größtmöglichen Abstand verglichen wird.

Beispiel

Sei $N = 12$ die Gesamtlänge des Arrays, in dem die Erfolglosen als Nullen und die Erfolgreichen Pings als Einsen gekennzeichnet werden. Die Anzahlen an Nullen bezeichnen wir als n.



Jetzt wird über die Abstände gemittelt: $\epsilon = \frac{4+1+3+1+1}{5} = 2$. Der größtmögliche, durchschnittliche Abstand lässt sich durch $\hat{\epsilon} = \frac{N-1}{n-1}$ berechnen. In diesem Fall würde sich $\frac{12-1}{6-1} = 2.75$ ergeben. Dies lässt sich auch nochmal manuell überprüfen. Eine mögliche Gleichverteilung mit optimalem Abstand sähe dann so aus:



Dies wird dann durch $\epsilon = \frac{2+2+2+3}{5} = \frac{N-1}{n-1} = 2.75 = \hat{\epsilon}$ bestätigt. Daraus ergibt sich für $\frac{\epsilon}{\hat{\epsilon}}$:

$$\frac{\epsilon}{\hat{\epsilon}} = \frac{2}{2,75} = 0,\overline{72}$$

und als normalisierter Wert ($\frac{\epsilon-c}{\hat{\epsilon}-c}$), wobei zum Normalisieren der minimale Abstand c, welcher durch ϵ erreicht werden kann, benutzt wird:

$$\frac{\epsilon - c}{\hat{\epsilon} - c} = \frac{2-1}{2,75-1} = 0,57142$$

.

Hier spielen vor allem zwei Faktoren eine Rolle. Einerseits das Verhältnis von erfolgreichen zu erfolglosen Paketen und die Verteilung der Pakete. Sind die Pakete z.B. am Anfang erfolglos und dann plötzlich erfolgreich, muss dies nicht zwingend ein Wackelkontakt sein.

3.7 Schnittstelle und Nutzung der Fehlererkennung

Um die API zu nutzen, muss also dann nur noch an dem Detector zu Beginn der Bus mit **scanAndInit** gelesen und dann während des Betriebs wiederholt **readDataStripe** gefolgt von **giveFeedback** aufgerufen werden. **scanAndInit** muss erfolgreich Motoren gefunden haben, um später von diesen die Daten (**readDataStripe**) zu lesen. **giveFeedback** sollte nach der Suche am Anfang und immer zwischen jedem **readDataStripe** erfolgen, um alle Suchergebnisse auszuschöpfen.

4 Auswertung

Auswerten und einschätzen lässt sich die für die Fehlererkennung benötigte Rechenzeit. Zusätzlich kann man noch mit den Schwellenwerten des Wackelkontakts experimentieren.

Wie bereits beschrieben, liest das Programm per Funktionsaufruf neue Daten ein. Die Frequenz der aufeinanderfolgenden Funktionsaufrufe wird als Datenstreifenleserate bezeichnet, die ich im Folgenden untersuche. Eine allgemein perfekte Frequenz der Datenstreifenleserate gibt es vermutlich nicht. Es ist aber ratsam sich den Bewegungen des Roboters anzupassen, da sich Kabel vermutlich mit den Bewegungen der Robotergelenke lösen oder wieder verbinden.

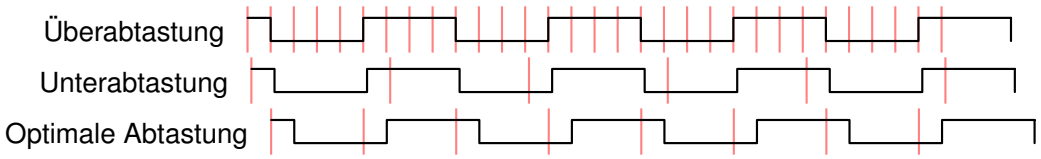


Abbildung 4.1: Abtastung eines Wackelkontakts

Die Frequenz muss also dem Roboter entsprechend angepasst werden, sollte aber etwa im Bereich von $\frac{1}{5}$ bis $20Hz$ liegen, um größtmögliches Feedback auf einen Wackelkontakt zu bekommen. Setzt man die Frequenz zu hoch, wird unnötig oft gelesen und dies schlägt sich negativ auf die Performance aus. Setzt man sie hingegen zu niedrig, werden zu wenig Daten gesammelt, um den richtigen Verlauf des Wackelkontakts zu erkennen, was sich schlecht auf die Fehleranalyse auswirkt. Eine niedrigere Frequenz reagiert später auf Fehler. Eine höhere Frequenz braucht eine größere Datenmenge (Stichprobengröße), da das Zeitintervall zwischen den Messungen immer kleiner wird, und somit einen insgesamt kleineren Zeitabschnitt betrachtet, als mit einer niedrigeren Frequenz und selben Stichprobengröße. Ändert man die Frequenz, ist es deshalb ratsam gleichermaßen die Stichprobengröße anzupassen.

Für die Klassifiziereung von Wackelkontakten gibt es theoretisch auch drei Parameter mit denen sich experimentieren lässt. Die ersten beiden (**_rate_bootom_threshold** und **_rate_top_threshold**) geben die untere und obere Grenze für das Verhältnis von nicht erreichten Paketen zur gesamten Stichprobengröße an (siehe 3.6). Es ist sinnvoll, vor allem bei TLL die untere Schranke etwas anzuheben, da immer gängige Paketverluste unab-

hängig von defekten Motoren erwartet werden müssen.

Auch **_dist_threshold**, also der Wert, mit der die stochastische Verteilung überprüft wird, sollte etwas größer als Null sein, um zu gewährleisten, dass nicht immer wenn die Stromzufuhr unterbrochen wird, ein Wackelkontakt erkannt wird.

Aus der folgenden Abbildung entnimmt man, dass sich die beanspruchte Rechenzeit zwischen einem **BroadcastPing** der API und der **broadcastPingRaw**-Funktion nur um 13ms unterscheidet. Die **scanAndInit** Funktion des Detectors benutzt den **broadcastPingRaw** und fügt dessen Ergebnisse, Fehlercodes und gefundene Motoren jeweils einer Menge hinzu. **scanAndInit** führt dabei mehrfach den **broadcastPingRaw** aus und dauert deswegen um ein Vielfaches länger als dieser. Der **broadcastPing** wurde zwar für die Fehlererkennung in den **broadcastPingRaw** und **investigateData** aufgeteilt, aber letzendlich wurde hier nur der Sendeprozess von der Paketanalyse getrennt, und die Paketanalyse befindet sich in derselben linearen Komplexitätsklasse wie die Paketgliederung vom **broadcastPing**. Ähnlich verhält es sich mit der ursprünglichen Sync Read-Klasse und der modifizierten Sync ReadRaw-Klasse, welche von der Funktion **readDataStripe** genutzt wird. Hier beeinflusst die Rohdatenanalyse nur geringfügig die Dauer und die CPU-Auslastung während der Programmschleife.

Lesefunktion	Dauer	Standardabweichung
broadcastPing	789,76ms	$16,08\mu s$
broadcastPingRaw	789,771 ms	$29,707\mu s$
investigateData	$1,99\mu s$	$981ns$
Sync Read	$953\mu s$	$14,229\mu s$
readDataSpripe	$965\mu s$	$10,679\mu s$

Tabelle 4.1: Antwortzeiten essentieller Lese- und Analysefunktionen bei einer USB-Latenz von $1ms$ und einer Probengröße von 1000

Lesefunktion	CPU-Auslastung	Standardabweichung
Sync Read	8,376%	25,9425%
readDataSpripe	8,842%	27,100%

Tabelle 4.2: Vergleich der CPU-Auslastung des Sync Reads und der readDataSpripe-Funktion während einer Dauerschleife bei einer USB-Latenz von $1ms$ und einer Probengröße von 10000

5 Fazit

In diesem Kapitel werden die wichtigsten Ergebnisse zusammengefasst, die Schwierigkeiten dieser Arbeit erklärt und zuletzt Komponenten und Methoden diskutiert, welche in Zukunft noch implementiert werden können, um die Fehleranalyse zu verbessern.

5.1 Zusammenfassung

Das Dynamixel Protokoll arbeitet mit Paketen, die zwischen den Motoren und dem Master über einen Bus hin- und hergesendet werden können. Dabei lässt sich jeder Motor genau über eine ID identifizieren. Die Pakete lassen sich in Instruktionspakete, welche ausschließlich von dem Master gesendet werden und in Statuspakete, welche ausschließlich von den Motoren gesendet werden, unterteilen. Schaut man sich die API für das Dynamixel Protoll V2 genauer an, findet man mehrere Funktionen, die für die Fehlererkennung interessant sein könnten.

Der Broadcastping führt eine Suche durch, welche ungefähr 0.8 Sekunden dauert und während dieser Zeit auf jegliche Antworten aller Motoren mit der vorher spezifizierten Baudrate wartet. Dieser Ping ist sehr praktisch, um am Anfang der Fehlererkennung nach Motoren zu suchen.

Während der Roboter Fußball spielt, kann sich dieser keinen Broadcastping leisten. Hier bediene ich mich dem Sync Read, einer Anweisung, die mehrere Motoren gleichzeitig mit nur einem Instruktionspaket ansteuern kann.

Bei den entdeckten Fehlern handelt es sich um Störer, die Spannungen und somit Daten auf dem Bus verändern und Motoren, die nicht kommunizieren aber dennoch den Bus nicht unterbrechen. Dann kann es noch vorkommen, dass ein Kabel nicht verbunden ist. Hierbei spielt der Anschluss der Stromquelle auf dem Bus eine Rolle. Wenn er von derselben Seite des Adapters fließt, kann man noch feststellen welche Motoren bis zur Stelle des Problems funktionieren. Wenn mehrere Verbindungsverluste oder zusätzliche Daten auf dem Bus durch eine schlechte Verbindung (Extremfall über die Luft durch einen Lichtbogen) erzeugt werden, liegt ein Wackelkontakt vor.

Bei dem zeitlichen Verhalten von den Motoren ist aufgefallen, dass die USB-Latenz die Antwortzeiten reguliert. Die Antwortzeiten von Motoren hängen im Wesentlichen von dem Master, der Baudrate, dem Return Delay TIme-Register, der USB-Latenz und dem Motorchip ab.

Beim Entwickeln des Programms habe ich mich auf eine Struktur geeinigt, welche einen Detector und mehrere Reader berücksichtigt. Reader sind für das Lesen und Erkennen von speziellen Fehlern zuständig, während sich am Detector die Schnittstelle für den Nutzer befindet. Der Detector liest dann mit dem Reader immer Daten ein und aktualisiert damit die gefundenen Probleme. Ein Wackelkontakt kann neben Entladungen auf dem Bus auch durch den zeitlichen Verlauf von erreichbaren und unerreichbaren Motoren erkannt werden. Hierfür werden über einen Zeitraum Daten gesammelt und dann das Verhältnis und die Verteilung bestimmt, um über einen Wackelkontakt zu entscheiden.

Die Datenstreifenleserate für die Wackelkontakterkennung muss an die rhythmischen Bewegungen des Roboters angepasst werden, da Wackelkontakte mit dessen Bewegungen entstehen können.

Durch den effizienten Umgang mit den Datenstreifen der Sync ReadRaw-Klasse verändert sich insbesondere die Dauer und die CPU-Auslastung der **readDataStripe** nur geringfügig.

5.2 Problematiken und Schwierigkeiten

Einige Probleme, wie die Unterscheidung von nicht antwortenden Motoren und nicht verbundenen Motoren oder auch eine unterstützende Klassifizierung durch Echtzeit-Spannungsmessungen konnten aus mehreren Gründen nicht weiter untersucht oder entwickelt werden. Einerseits fehlten genug fehlerhafte Motoren oder überhaupt Motoren eines zuvor gemessenen Fehlertyps. Das liegt daran, dass sich die Menge der kaputten Motoren im RoboCup-Team verändert. Motoren werden repariert und können dann natürlich nicht mehr untersucht werden. Außerdem musste nach einem Defekt eines alten USB2DYNAMIXEL ein neuer U2D2_INT als Master benutzt werden, welcher auf beide Störer nicht mehr reagiert hat. Der Vorteil ist natürlich, dass dann die Störer nicht mehr stören können. Der Nachteil ist, dass angefangene Messungen nicht auf dieselbe Art zuende geführt werden können, aber auch, dass höchstwahrscheinlich die Störer auch nicht mehr erkannt werden können.

5.3 Ausblick

In dieser Arbeit wurde zwar eine Lösung entworfen, um den Fehlerbehebungsprozess der Motoren zu beschleunigen, aber nicht besonders auf die Ursachen der Fehler hin-, sondern nur mit den Äußerungen, die notwendig waren, um ein Programm zu entwerfen und die Fehler zu klassifizieren gearbeitet. Können mehrere Fehler dieselben Ursachen haben? Lassen sich manche Fehler vorbeugen oder sind sie Alterungserscheinigungen, die nach ungewisser Zeit auftreten? Welche Fehler wurde noch nicht von dieser Arbeit

erfasst? Ich habe in dieser Arbeit nur mit den fehlerhaften Motoren gearbeitet, die sich im Labor angesammelt hatten. Es wäre trotzdem möglich, dass diese Fehlermenge von der kompletten abweicht. Die Einbindung eines abstrakteren Analyseverfahrens würde hilfreich sein im Falle, dass Fehler nicht durch den modifizierten FTA erkannt werden. Würde man die Reihenfolge der Motoren im Bus ohne Angabe durch Parameter erkennen können, wäre dadurch eine bessere Klassifizierung zwischen nicht sendendem Motor und dauerhaftem Verbindungsverlust möglich. Außerdem könnten dann nicht nur die Positionen der Wackelkontakte sondern die genauen Motoren für die Analyse ausfindig gemacht werden. Ein besseres Verständnis bestimmter Fehler wie der rhythmische oder permanente Störer ermöglichen eine noch bessere und eindeutigere Klassifizierung dieser.

Ob in Zukunft die Analyse auf Hardwareprobleme leichter oder schwerer wird, lässt sich schwer sagen. Es hängt stark davon ab, welches Interesse die Hersteller daran haben, dass die Käufer auch selber in der Lage sind die Hardware zu analysieren und zu reparieren. Die Fehleranalyse wird jedoch immer wichtiger, da die Hardware der Roboter komplexer und wartungsbedürftiger wird. Je mehr Zeit durch automatische Analyse eingespart werden kann, umso seltener wird ein Entwicklerteam aus einem für die Entwicklung des Roboters entscheidenden Arbeitsprozess herausgerissen.

6 Literaturverzeichnis

[BGZ19] BESTMANN, Marc ; GÜLDENSTEIN, Jasper ; ZHANG, Jianwei: High-frequency multi bus servo and sensor communication using the Dynamixel protocol. In: *Robot World Cup* Springer, 2019, S. 16–29

[CB99] CHESHIRE, Stuart ; BAKER, Mary: Consistent overhead byte stuffing. In: *IEEE/ACM Transactions on networking* 7 (1999), Nr. 2, S. 159–172

[LGTL85] LEE, Wen-Shing ; GROSH, Doris L. ; TILLMAN, Frank A. ; LIE, Chang H.: Fault Tree Analysis, Methods, and Applications A Review. In: *IEEE transactions on reliability* 34 (1985), Nr. 3, S. 194–203

[McA94] MCAULEY, Anthony J.: Weighted sum codes for error detection and their comparison with existing codes. In: *IEEE/ACM Transactions On Networking* 2 (1994), Nr. 1, S. 16–22

[Rob] ROBOTIS: *Protocol 2.0*, `https://emanual.robotis.com/docs/en/dxl/protocol2/`, Abruf: 2020-06-07

[SKS90] SCHWAB, Adolf J. ; KÜRNER, Wolfgang ; SCHWAB, Adolf J.: *Elektromagnetische Verträglichkeit*. Bd. 4. Springer, 1990

[SMW05] STEINBAUER, Gerald ; MÖRTH, Martin ; WOTAWA, Franz: Real-time diagnosis and repair of faults of robot control software. In: *Robot Soccer World Cup* Springer, 2005, S. 13–23

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

$_p$ Positionserkennung möglich. 31

FM Funktionstüchtiger Motor. 25

FMB Funktionstüchtiger Motor im Betrieb (Senden und Empfangen). 25

LW Wackelkontakt (loose Wire). 31

NA Nicht antwortender Motor (Not Answering). 31

NP Keine Stromzufuhr (No Power). 31

PS Permanenter Störer. 25, 31

RS Rhythmischer Störer. 25, 31

WD Loses Kabel (Wire Disconnect). 31

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____