IT DESIGNERS GRUPPE

Hochschule Esslingen
University of Applied Sciences

Bachelorarbeit

Analyse der Eignung der Programmiersprache Rust zur Entwicklung eines Kommunikationsstacks für eingebettete Systeme

Fakultät Informationstechnik

Studiengang Angewandte Informatik

Sommersemester 2021/2022

Alexander Hübener

Zeitraum: 01.09.2021 - 31.01.2022
Prüfer: Prof. Dr. Rer. Nat. Markus Enzweiler
Zweitprüfer: Dipl.-Ing.(FH) Alexander Renz

Firma: IT-Designers GmbH, Esslingen am Neckar
Betreuer: Alexander Renz

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 21.01.2022 

Unterschrift

Danksagung

Ich möchte mich bei allen bedanken, die mich während meines Studiums und der Bachelorarbeit unterstützt haben. Besonders bei meinen Eltern, die mir den Weg zum Studium geebnet haben.

Zuerst möchte ich Pavel Kirienko danken, der der Gründer des UAVCAN Projekts ist und mich in das Projekt zur Weiterentwicklung der UAVCAN Rust Bibliothek mit aufgenommen hat. Zudem möchte ich mich bei David Lenfesty für die Zusammenarbeit an der Rust Bibliothek bedanken.

Des Weiteren möchte ich meinem Betreuer Alexander Renz meinen Dank aussprechen. Er ist mir in Meetings mit Rat zur Seite gestanden und hat mir mit seiner Erfahrung gute Anregungen für die Arbeit gegeben.

Zudem möchte ich mich bei meiner Mutter und meinen Freunden Timothy Nas, Andreas Lautner und Jakob Guhl für das Korrekturlesen bedanken.

Schlussendlich geht der Dank noch an meinen Professor Markus Enzweiler. Mit Meetings herrschte ein reger Austausch, was mir ein sicheres Gefühl gegeben hat, dass die Arbeit in die richtige Richtung geht. Aufkommende Fragen konnten immer gestellt und es konnte mit einer schnellen Antwort gerechnet werden.

„Wenn das Problem nicht verstanden wurde, ist man in jeder Programmiersprache stumm.“

– Axel Schwab

Kurzfassung

In Pkws, Lkws oder fliegenden Verkehrsmitteln kommunizieren einzelne Untersysteme über Bussysteme miteinander. Die einzelnen Systeme sind meist eingebettete Systeme und bestehen aus Mikrocontrollern zum Auslesen von Sensoren und zur Steuerung von Aktoren. Eingebettete Systeme besitzen begrenzte Ressourcen, dazu zählen CPU-Leistung und ein begrenzter Speicherplatz.

Für die Entwicklung einer Software für derartige Systeme ist die Wahl der Programmiersprache essentiell. Die Sprache soll die Ressourcen für eine effiziente Programmausführung bestmöglich ausnutzen und dennoch den Programmierer mit beispielsweise einer automatisierten Speicherverwaltung unterstützen. Speicherzugriffsfehler können in kritischen Anwendungsgebieten verheerende Folgen mit sich bringen.

In dieser Arbeit soll die Sprache Rust anhand einer Implementation des Kommunikationsstacks des UAVCAN Standards auf die Eignung im Bereich der eingebetteten Systeme analysiert werden. Für das theoretische Verständnis werden hierzu Anforderungen bei eingebetteten System herausgearbeitet, Sprachfunktionalitäten der Sprache Rust analysiert und die Funktionsweise des Kommunikationsstacks zusammengefasst.

Während der Arbeit wird eine vorhandene Rust-Implementation des UAVCAN Standards auf ein eingebettetes System portiert. Hierbei werden die Stärken von Rust für ein solches Vorgehen aufgezeigt. Für die Evaluation der Implementation wird eine Benchmark-Suite entwickelt.

Schlussendlich werden Zeit- und Speicher-Messungen mit der UAVCAN Bibliothek durchgeführt. Dabei werden Software-Only Messungen und Messungen eines Hardwareaufbaus durchgeführt. Die Messergebnisse sollen zur Analyse der Eignung der Sprache hergenommen werden.

Schlagworte: Rust, Eingebettete Systeme, Kommunikationsstack, Eignungsanalyse

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

BLE Bluetooth Low Energy

CAN Controller Area Network

CiA CAN in Automation

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver Transmitter

RTOS Real-time operating system

HMI Human-Machine-Interface

ETA Estimated Time of Arrival

IP Internet Protocol

UDP User Datagram Protocol

DSDL Data Structure Description Language

FFI Fremdfunktionsschnittstelle

CAN FD Controller Area Network Flexible Data-Rate

HAL Hardwareabstraktionsschicht

DWT Data Watchpoint Trigger

GCC GNU-C-Compiler

CRC Cyclic Redundancy Check

CPU Central Processing Unit

IDE Integrated Development Environment

TLSF Two-Level Segregate Fit

RTT Real-Time Transfer

ROM Read-Only Memory

RAM Random-Access Memory

1 Einleitung

1.1 Motivation

Pkws, Lkws oder fliegende Verkehrsmittel, seien diese bemannt oder unbemannt, benötigen Sensoren und Aktoren zur Steuerung, Navigation und Stabilisierung. Aufgrund steigender Anforderung an Verarbeitungsleistung werden kleine abgeschlossene Systeme eingesetzt, welche untereinander mit Hilfe von Netzwerktechnik Daten austauschen. Ein System kann hierbei aus einem Mikrocontroller und angeschlossener Peripherie bestehen. Die eingesetzten Mikrocontroller haben meist nur eingeschränkte Ressourcen bezüglich Speicher und CPU-Leistung. Demnach ist es wichtig, dass die verwendete Software für diese Einschränkungen optimiert wird und somit effizienter ausgeführt werden kann.

Die verwendete Programmiersprache soll den Programmierer in diesem Umfeld vor Ausführungsfehlern bewahren, gleichzeitig aber auch eine speicher- und ausführungseffiziente Firmware gewährleisten. In dieser Arbeit wird hierfür die Programmiersprache Rust für eine Kommunikationsstack-Implementation auf einem eingebetteten System angewandt und evaluiert.

1.2 Rahmenbedingungen

Bei dieser Arbeit sollen eingebettete Systeme als Entwicklungsumgebung betrachtet werden. Ein eingebettetes System kann verschiedenst ausgelegt werden. Dabei können die Leistung und die verfügbaren Ressourcen der Steuereinheit des Systems stark variieren, sowie die Systemgrenzen unterschiedlich definiert werden.

In dieser Arbeit soll unter einem eingebetteten System allgemein ein System verstanden werden, das Mikrocontroller mit den folgenden Spezifikationen implementiert:

- FLASH $\leq 512\,\mathrm{kB}$
- RAM $\leq 128\,\mathrm{kB}$
- CPU Kern(e) $= 1$
- CPU Takt $\leq 170\,\mathrm{MHz}$

Des Weiteren werden diese Spezifikationen durch im Laufe der Arbeit verwendeter Testhardware zusätzlich beschränkt.

Als abgegrenztes System soll ein Mikrocontroller und dessen Peripherie definiert sein. Das System soll durch ein Bussystem auf ein verteiltes System erweitert werden, bei welchem die einzelnen Systeme miteinander kommunizieren können.

Ergebnisse dieser Arbeit können auch auf Systeme mit mehr Ressourcen bezogen werden. Für eingebettete Systeme, bei welchen diese eingeschränkt sind, sind die Analysen umso mehr von Bedeutung.

1.3 Aufgabenstellung

Anhand dieser Arbeit soll die Eignung der Programmiersprache Rust für die Entwicklung eines Kommunikationsstacks auf eingebetteten Systemen analysiert werden.

Im ersten Abschnitt soll eine theoretische Auseinandersetzung mit den für die Zielsetzung relevanter Themen erfolgen. Dazu wird ein Überblick über eingebettete Systeme und daraus folgende Anforderungen für die eingebettete Systemprogrammierung gegeben. Anschließend soll ein Einblick in die Programmiersprache Rust und deren Alleinstellungsmerkmale geschaffen werden. Als Nächstes werden die Spezifikationen des CAN-Bus aufgezeigt.

In einem weiteren Schritt soll der UAVCAN Standard als Kommunikationsstack theoretisch analysiert werden. Hierzu werden dessen Grundfunktionalitäten herausgearbeitet und APIs der Rust- und der Referenz-Implementation in Arduino gegenübergestellt.

Durch einen praktischen Teil der Arbeit soll die Entwicklung mit Rust auf eingebetteten Systemen erprobt werden. Hierbei wird die Open-Source entwickelte Rust Bibliothek des UAVCAN Standards auf ein derartiges System portiert und ein Testprogramm zur Funktionsevaluierung der Änderungen geschrieben.

Schlussendlich soll die Rust-Implementation des UAVCAN Standards anhand des Softwarequalitätsmerkmals der Leistungsfähigkeit evaluiert werden. Hierfür sollen Zeit- und Speichermessungen durchgeführt werden. Die Messwerte sollen den Werten, die bei der Arduino-Implementation ermittelt werden, gegenübergestellt werden.

Mithilfe der theoretischen Betrachtungen der Programmiersprache Rust, der Analyse der Implementations API und dem praktischen Teil mit der Programmierung und Portierung auf eingebettete Systeme und der Durchführung von Leistungsmessungen zur Evaluation der Rust-Implementation, soll ein Fazit gezogen werden, ob die Sprache Rust sich für die Entwicklung eines Kommunikationsstacks auf eingebetteten Systemen eignet.

2 Theoretische Auseinandersetzung

In diesem Kapitel sollen die für die Ausarbeitung wichtigen Themen theoretisch betrachtet und analysiert werden. Dazu wird ein Überblick über eingebettete Systeme gegeben und Anforderungen für die Softwareentwicklung auf diesen herausgearbeitet. Anschließend wird die Programmiersprache Rust betrachtet. Dabei werden Funktionen analysiert, die markant für die Sprache sind. Folgend wird ein Überblick über den *Controller Area Network* (CAN)-Bus und das CAN-Protokoll gegeben.

2.1 Eingebettete Systeme

Ein eingebettetes System (engl. Embedded System) ist ein Computersystem, das in einem technischen System eingesetzt wird, um dieses zu regeln, zu steuern oder zu überwachen [13, 14]. Eingebettete Systeme haben demzufolge einen weitläufigen Einsatzbereich [11, 25].

Im Jahr 2009 sind laut [2, 14] 98 % der produzierten *Central Processing Unit* (CPU)s in eingebetteten Systemen verwendet worden. Der hohe Anteil solcher Systeme macht deutlich, dass es wichtig ist, die Standards in diesem Bereich voranzutreiben und damit das Entwickeln zu erleichtern.

Werden Allzweck-Computer und eingebettete Systeme nebeneinander betrachtet, ergeben sich Unterschiede, auf die im weiteren Verlauf eingegangen wird.

Ein Allzweck-Computer ist für verschiedene Aufgaben konzipiert. Das System hat keine Beschränkungen hinsichtlich der Energieaufnahme, des Formfaktors und passt sich preislich an den Konsummarkt an. Der Benutzer hat eine Schnittstelle in Form einer Maus und einer Tastatur, über welche er mit dem System interagieren kann [39]. Außerdem gibt es eine visuelle Darstellung über einen Monitor zur Überprüfung der Systemantwort [39].

Ein eingebettetes System hingegen wird meistens zur Erfüllung einer Aufgabe entwickelt [11] und entsprechend variiert die Komplexität [14]. Zudem kann ein eingebettetes System unabhängig arbeiten oder als Teil eines größeren Systems eingesetzt werden [14]. Bei der Entwicklung müssen jedoch Aspekte wie das Einhalten von Energieverbrauchswerten [25], Kosten und Formfaktoren berücksichtigt werden.

Zur Ausführung der Aufgabe des Systems werden Mikrocontroller verwendet [14]. Dies sind kleine und günstige Microcomputer [47]. Die Komponenten eines Mikrocontrollers sind auf einem Chip implementiert, dadurch wird es auch als System auf einem Chip (SoC)

bezeichnet [47]. Dazu gehören eine CPU, ein Speicherbereich, ein Interface zum möglichen Debuggen der Ausführung und eine Peripherieeinheit. Über angebrachte Pins bekommt der Mikrocontroller Strom, erhält Eingangssignale und kann Ausgangssignale herausgeben.

Ein Mikrocontroller wird anhand der CPU Leistung und des verfügbaren Speichers entsprechend der Anforderung auf das eingebettete System ausgewählt. Deshalb sind die verfügbaren Ressourcen für die Programmierung eingeschränkt [11].

2.1.1 Softwareerstellung für einen Mikrocontroller

Wird die Softwareerstellung für Betriebssysteme betrachtet, werden die Programme basierend auf bereitgestellten Betriebssystem-Bibliotheken erstellt. Diese abstrahieren die Hardware verschiedener Hersteller. Für einen Mikrocontroller gibt es hierfür eine *Hardwareabstraktionsschicht* (HAL). Diese bildet eine Schicht zwischen der Hersteller spezifischen Hardware und dem Programmcode [15].

Auf einem Mikrocontroller wird grundsätzlich nur ein Programm ausgeführt. Dieses ist für die Konfiguration des Mikrocontrollers zuständig, welche zu Beginn im Programm durch HAL Funktionsaufrufe umgesetzt wird.

Die Software wird auf einem Allzweck-Computer entwickelt. Durch die Cross-Kompilierung kann eine ausführbare Datei für das Zielsystem erstellt werden, die Firmware [10]. Diese muss zur Ausführung auf den Mikrocontroller übertragen und in die vorgesehenen Speicherbereiche geschrieben werden. Nach einem Reset des Mikrocontrollers wird die neue Firmware ausgeführt.

Zur Übertragung der Firmware gibt es verschiedene Möglichkeiten. Eine davon ist zum Beispiel das Verwenden eines *in-circuit* Debuggers und Programmierers [17]. Dies ist beispielsweise auf das in der Arbeit verwendete Entwicklungsboard Nucleo-G431RB, das in Kapitel 4.2 näher beschrieben wird, integriert.

Zum Debuggen von Software auf Mikrocontrollern werden verschiedenste Techniken verwendet, da auch angesteuerte Hardware überprüft werden muss. Dazu zählen zum Beispiel ein angeschlossenes Bussystem zur Übermittlung von Nachrichten oder ein analoger Ausgang.

Zu den Techniken, zur Überprüfung auf richtige Hardwarefunktionalität, gehören zum Beispiel das Verwenden eines Oszilloskops oder eines Logik Analysers [51].

Des Weiteren kann zur Untersuchung der Software auf Fehler der *in-circuit* Debugger und Programmierer verwendet werden. Damit kann auf die Debug-Einheit des Mikrocontrollers zugegriffen werden und somit Haltepunkte gesetzt, einzelne Instruktionen des Programms ausgeführt und Werte von definierten Variablen im Code überprüft werden [17].

Anforderungen

Laut [27] hat die Software für eingebettete Systeme die Anforderung einer langen Lebenszeit, dafür muss diese gepflegt und gewartet werden. Hierfür wird eine entsprechend gute Codequalität und eine verständliche Softwarearchitektur gefordert [27].

Speziell bei Forderungen einer kurzen Latenzzeit und eines hohen Durchsatzes an Reaktionen auf Ereignisse im System, muss gewährleistet sein, dass Programme schnellstmöglich ausgeführt werden können. Demzufolge muss die Software die verfügbaren Ressourcen effizient einsetzen. Es ist wichtig, dass die auszuführenden Aufgaben so wenig wie möglich CPU-Zyklen benötigen und dass zeitintensive Operationen möglichst gezielt eingesetzt werden.

[27] beschreibt die Verwendung von dynamischem Speicher als sehr zeitintensiv, da diese einige CPU-Zyklen benötigt. Bei Nichtverwendung des dynamischen Speichers muss Speicher basierend auf Worst-Case Berechnungen statisch belegt werden. Das kann zu einer ineffizienten Verwendung des verfügbaren Speichers führen [27].

Folglich muss bei der Entwicklung von Software für eingebettete Systeme zum einen auf die Ressourcen geachtet werden, wie zum Beispiel verfügbare Energie, Speicher und Leistung, zum anderen auf die Anforderungen an das System. Dazu gehören Reaktionszeit, lange Lebenszeit und die Zuverlässigkeit, also eine hohe Wahrscheinlichkeit der fehlerfreien Funktion.

2.2 Programmiersprache Rust

Im vorherigen Kapitel wurden Anforderungen an ein eingebettetes System und an die dafür geschriebene Software dargestellt. Rust soll eine Programmiersprache sein, welche gut geeignet ist, die folgenden Anforderungen abzudecken:

Zuverlässigkeit beim Verwenden des Speichers durch eingeführte Regeln und Funktionen, welche zur Kompilierzeit überprüft werden, gewährleisten,

Reaktionszeit und Leistung durch einen Compiler, der effizienten Maschinencode generiert,

lange Lebenszeit einer Software durch die Anwendung von Software Entwurfsmustern, ohne größeren oder ineffizienten Code zu generieren, was zu einer guten Codequalität führen soll,

verschiedenste Zielsysteme abdecken durch gute Werkzeuge, die den Programmierer hierbei unterstützen den Quellcode zu kompilieren.

In diesem Kapitel wird zunächst auf die Geschichte der Sprache eingegangen, da die Sprache noch nicht sehr lange existiert. Danach wird auf die Entwicklungsumgebung eingegangen. Daraufhin wird ein Überblick über die Funktionen der Sprache zur Gewährleistung von Speichersicherheit gegeben. Folgend werden weitere Funktionen, die es speziell bei Rust gibt, betrachtet. Bevor zuletzt auf den Rust Compiler eingegangen wird.

2.2.1 Geschichte

Die Programmiersprache Rust wurde ab 2006 von Graydon Hoare als ein Hobbyprojekt entwickelt [34]. Hoare betreute die Sprache drei Jahre und führte grundlegende Techniken ein. Ursprünglich kommt Graydon Hoare aus dem Bereich System- und Desktop-Programmierung, wo er mit der Sprache C++ arbeitete. So nannte er als Grund, wieso er mit dem Rust Projekt begonnen hat, dass C++ „[..] ziemlich fehlerträchtig“ sei [40]. Als Beispiel gab er die Speichersicherheit an, welche nicht immer gewährleistet sei [40].

Rust entwickelte sich von Beginn an empirisch. So verfolgt die Sprache immer dieselben Ziele, aber die Funktionalitäten änderten sich im Laufe der Jahre [20].

[20] teilt die Entwicklungsjahre der Sprache Rust in verschiedene Epochen ein. Die erste Epoche wird als „The Personal Years“ bezeichnet. Dabei wurden Konzepte etabliert, wie zum Beispiel Speichersicherheit, die in Kapitel 2.2.4 behandelt wird, die Kontrolle über das Verändern von Variablen, siehe *Unveränderbar von Beginn* und dass Regeln der Sprache auch gebrochen werden können, siehe *Unsicheres Rust*.

Ab 2009 hat sich die Firma Mozilla finanziell an dem Projekt beteiligt, da zu dieser Zeit die Grundprinzipien der Sprache feststanden und grundlegende Tests funktionierten [34]. Nun war die Sprache in der sogenannten „The Graydon Years“ Epoche angelangt, in der das Team um die Programmiersprache wuchs [20].

Als Nächstes wurde das Typsystem erarbeitet [20]. Durch das Typsystem wurden Implementationen unabhängiger von den Basiskonzepten der Sprache, woraufhin einige Funktionalitäten in Bibliotheken ausgelagert worden sind [20]. Im gleichen Schritt wurde das Werkzeug *cargo*[35] etabliert und die Bibliotheks-Registration *crates.io*[8] aufgesetzt.

Im Mai 2015 wurde in „The Release Years“ die Rust 1.0 Version veröffentlicht [20]. [20] meint, dass die Weiterentwicklung nicht nur die Sprache selbst betrifft, sondern auch das Ecosystem, die Werkzeuge, die Stabilität der Sprache und die Community.

Laut [20] befindet sich Rust momentan in der Epoche „The Production Years“, in welcher die Sprache in Projekten verwendet werden soll. Laut [43] ergaben Umfragen 2021, dass Rust schon sechs Jahre hintereinander, also seit der Version 1.0, die „meist geliebte Programmiersprache“ unter Entwicklern sei.

2.2.2 Entwicklung der Sprache

Die Entwicklung wird von etlichen, auf der ganzen Welt verteilten Entwicklern vorangetrieben. Diese kommen aus Gebieten wie der C++ Entwicklung, Skriptsprachen und funktionaler Programmierung. Es gibt das Team, das sich um die Vision und Prioritäten der Sprache kümmert. Außerdem gibt es mehrere kleinere Teams, die einzelne Bereiche weiterentwickeln [20].

Ein Team beschäftigt sich zum Beispiel mit dem *Embedded* Bereich [33]. Hierfür gibt es viele Materialien, welche öffentlich unter https://docs.rust-embedded.org/ zur Verfügung stehen. Dadurch hat jeder Entwickler einen barrierefreien Einstieg in die Sprache und kann so an der Sprache mitarbeiten.

In Rust wird neue Funktionalität durch RFC[1]s eingeführt. Diese Funktionsanfragen können auf GitHub eingesehen werden. Ein RFC soll grundlegende Änderungen beschreiben und über Implementierungsarten mit den jeweiligen Rust Teams abstimmen. Dadurch soll die Sprache konsistent und einheitlich weiterentwickelt werden [37].

Rust hat einen genau definierten Update Zyklus. Laut [36] und wie auch [20] erwähnt hat, soll die Sprache Rust trotz Updates zu jeder Zeit stabil sein und trotzdem gleichzeitig weiterentwickelt werden können. Damit sich die Entwickler auf Änderungen einstellen können, gibt es drei Veröffentlichungs-Kanäle: *stable*, *beta* und *nightly*. Auf dem *nightly* Kanal wird jede Nacht eine neue Version veröffentlicht. Diese hat somit die neuesten Funktionen, Änderungen sind jedoch noch wenig getestet. Der *beta* Kanal ist zum Testen von der jeweils nächsten stabilen Version. Alle sechs Wochen erscheint auf dem *stable* Kanal eine neue Version, die als stabil gilt und die aktuelle Version der Sprache Rust repräsentiert [36].

Dadurch, dass die Sprache kontinuierlich weiterentwickelt wird und immer wieder neue

[1] RFC bedeutet ausgeschrieben Request for Comment. Mit einem RFC werden neue Funktionalitäten mit der Community besprochen. Dazu gehören u.a. die Relevanz, der Einfluss auf aktuelle Bestandteile der Sprache und die Speichersicherheit.

Versionen erscheinen, ändert sich auch die Version während dieser Arbeit. Dennoch wurde versucht, durchzuführende Tests und Messungen mit derselben Version zu machen, damit Vergleiche konsistent bleiben.

2.2.3 Entwicklungsumgebung

Für das Verwalten der Entwicklungsumgebung gibt es das Werkzeug *rustup*. Mit Hilfe dessen können installierte Targets und Werkzeugketten auf dem Computer verwaltet werden. Außerdem lässt sich vereinfacht ein Update der Version von Rust und aller dazugehöriger Komponenten durchführen.

Speziell für das Cross-Kompilieren ist das Installieren und Verwalten von Targets und Werkzeugketten wichtig und soll am Beispiel des Targets *armv7-unknown-linux-gnueabihf*, welches die Zielplattform der verwendeten Mikrocontroller (Kapitel 4.2) beschreibt, gezeigt werden.

Mit dem Werkzeug *rustup* können verfügbare Targets aufgelistet werden (keine vollständige Liste):

```
$ rustup target list
arm-unknown-linux-gnueabihf
x86_64-apple-ios
x86_64-unknown-linux-gnu
armv7-unknown-linux-gnueabihf
```

Listing (2.1): Auflisten von verfügbaren Zielplattformen.

Anschließend kann das gewünschte Target installiert werden:

```
rustup install armv7-unknown-linux-gnueabihf
```

Listing (2.2): Installation einer Zielplattform.

Neben *rustup* gibt es die Paketverwaltungssoftware *cargo*. Diese verwaltet verwendete Pakete in einem Projekt. In der Projektdatei, welche *Cargo.toml* genannt wird, können Abhängigkeiten definiert werden. In Listing 2.3 ist der Inhalt einer solchen gezeigt.

So kann für jede Abhängigkeit speziell die Version, wie dies in Zeile 8 zu sehen ist, angegeben werden und es können weitere Metadaten hinzugefügt werden.

```
[package]
authors = ["Autor <Autor@gmail.com>"]
name = "Beispiel"
edition = "2021"
version = "0.1.0"
```

```
6
7 [dependencies]
8 cortex-m = "0.6.0"
9 cortex-m-rt = "0.6.10"
```

Listing (2.3): Beispiel Inhalt einer Cargo.toml Datei

Die Pakete werden mit *cargo* von *crates.io* heruntergeladen. Dies ist die Bibliotheksregistrierung für Rust Bibliotheken. Dort kann jeder Entwickler Bibliotheken, welche in Rust als *crate* bezeichnet werden, hochladen und dementsprechend publizieren.

Bibliotheken können auch direkt von GitHub geladen werden oder aus einem lokalen Ordner eingebunden werden.

Des Weiteren vereinfacht *cargo* das Bauen und Ausführen eines jeden Projekts. Hierbei wird der Rust Compiler mit den benötigten Compiler-Flags indirekt aufgerufen und Abhängigkeiten werden mit eingebunden.

2.2.4 Speicherverwaltung

Bei der Speicherverwaltung in Programmen wird grundsätzlich zwischen dem Stapelspeicher (Stack) und dem dynamischen Speicher (Heap) unterschieden. Der Stack wird vom Compiler zum Speichern lokaler Variablen von Funktionen verwendet. Die Größe der Variablen muss bei der Kompilierung bekannt sein. Auf den Stapelspeicher können Objekte entweder gelegt oder das zuletzt daraufgelegte heruntergenommen werden. Die Aufgabe des Compilers ist die Verwaltung dieses Speichers.

Auf dem Heap werden zur Laufzeit variabel große Datenmengen gespeichert und wieder freigegeben. Programmiersprachen haben verschiedene Ansätze für die Verwaltung des dynamischen Speichers, vor allem dessen erneute Freigabe. Zum Beispiel wird in Java ein sogenannter Garbage-Collector[1] eingesetzt, welcher den Speicher verwalten soll [26]. In C muss der Programmierer selbst den nicht mehr benötigten Speicher freigeben.

In Rust kann der Compiler durch Regeln und Funktionalitäten bereits bei der Kompilierung herausfinden, wann Variablen nicht mehr verwendet und demnach Speicher automatisch, ohne Überprüfung zur Laufzeit, erneut freigegeben werden kann.

Eigentümerschaft (Ownership)

Die Eigentümerschaft ist laut [36] das Alleinstellungsmerkmal von Rust und soll Speichersicherheit garantieren.

[1] Die Runtime des Programms verwaltet zyklisch den Speicher und gibt nicht mehr benötigten Speicher wieder frei [26].

Die Eigentümerschaft besagt, dass ein Wert eine Variable hat, die sein Eigentümer ist. Des Weiteren darf es zur selben Zeit immer nur einen Eigentümer geben. Die Lebenszeit des Wertes erstreckt sich über die Lebenszeit des Eigentümers [36].

Ein Wert kann seinen Eigentümer wechseln, was in Rust als eine *move* Operation bezeichnet wird [36]. Der bisherige Eigentümer verliert dabei den Wert. Damit die Eigentümerschaft jedoch nicht aufgegeben werden muss, kann der Wert kopiert und einem neuen Eigentümer zugewiesen werden. Dies führt dazu, dass die Werte voneinander unabhängig werden.

In Listing 2.6 ist das Übertragen von Eigentum des Wertes vom Typ *Large* in Zeile 5 gezeigt. Hierbei muss darauf hingewiesen werden, dass *Large* nicht kopiert werden kann, wie das der Fall bei skalaren Typen in Rust ist. Der Wert wird als Erstes der Variable *a*, welche nun der Eigentümer von *Large* ist, zugewiesen. In der darauffolgenden Zeile wird der Wert von *a* an *b* übergeben. Nun ist der Eigentümer des Wertes die Variable *b*. In Zeile 7 soll die Variable *a* im Terminal angezeigt werden.

Wird dieses Programm nun kompiliert, zeigt der Compiler eine Fehlermeldung, die in Verbindung mit Zeile 7 steht. Ohne eine Ausführung des Programms wird darauf hingewiesen, dass der Wert von *a* zu *b* übertragen wurde. Der genaue Fehler kann in Listing 2.5 nachgelesen werden. Dabei wird explizit darauf hingewiesen, dass der Typ *Large* nicht kopiert werden kann, somit eine *move* Operation stattfindet und *a* keine Gültigkeit mehr hat.

```
1 #[derive(Debug, Default)]
2 struct Large {/* discard */}
3
4 fn main() {
5     let a = Large::default();
6     let b = a;
7     println!("var: {}", a); // error
8 }
```

Listing (2.4): Darstellung der Eigentümerschaft in Rust.

```
1 error[E0382]: borrow of moved value: `a`
2   --> src/main.rs:7:23
3    |
4 5  |     let a = Large::default();
5    |         - move occurs because `a` has type `Large`, which does
       not implement the `Copy` trait
6 6  |     let b = a;
7    |             - value moved here
8 7  |     println!("var: {:?}", a); // error
9    |                           ^ value borrowed here after move
```

Listing (2.5): Fehlermeldung bei Verwendung einer zuvor ausgeliehenen Variable.

Durch das Zusammenspiel von Eigentümerschaft und Compiler ist genau definiert, welche Variable der Eigentümer eines Wertes ist und dass keine ungewollte Kopie entsteht.

Lebenszeit einer Variable

Nachdem durch die Eigentümerschaft klar definiert ist, dass ein Wert nur einen Besitzer und demnach auch dessen Lebenszeit hat, muss nun betrachtet werden, wie lange die Variable Gültigkeit hat.

Laut [36] sollen Gültigkeitsbereiche, im Englischen Scope, sowohl für die Eigentümerschaft als auch das Ausleihen von Variablen und deren Lebenszeiten wichtig sein.

Ein Bereich wird durch geschweifte Klammern definiert. Es können weitere Scopes im Inneren eines bestehenden aufgemacht werden. Dabei bleibt die Gültigkeit von äußeren Variablen in die geschachtelten Bereiche erhalten. Umgekehrt verlieren Variablen eines inneren Bereichs jedoch an Gültigkeit [36]. Bei einem Zugriff auf äußere Variablen wird gleichermaßen die Eigentümerschaft angewandt und die Werte entweder kopiert oder neu zugewiesen.

Ist eine Variable in einem Scope definiert und ist dieser zu Ende, endet die die Lebenszeit der Variable und wird zerstört [36]. In Listing 2.6 ist ein solches Szenario gezeigt. Der äußere Scope wird in Zeile 1 geöffnet und der innere in Zeile 3. Der Wert der Variable *außen* wird in Zeile 4 im inneren Bereich der Variable *innen* durch eine *move* Operation zugewiesen. Somit ist *außen* nun ungültig. Wenn der innere Scope in Zeile 5 zu Ende ist, ist auch die Lebenszeit von *innen* zu Ende und deren Wert wird zerstört.

```
1 {
2     let außen = Large::default();
3     {
4         let innen = außen; // Besitzer wechseln
5     } // zerstören von innen
6 }
```

Listing (2.6): Bereiche und Lebenszeiten in Rust.

Referenzen

Durch die bereits erläuterte Eigentümerschaft, können Werte entweder verschoben oder kopiert werden. Damit diese Einschränkung aufgelockert werden kann, gibt es in Rust Referenzen [36].

Durch eine Referenz kann ein Wert einer Variable vorübergehend ausgeliehen werden, was in Rust als *borrowing* bezeichnet wird. Hierfür soll das Beispiel aus dem Kapitel Eigentümerschaft modifiziert werden. Diese Änderung ist in Listing 2.7 in Zeile 6 zu sehen. Nun wird eine Referenz auf die Variable a erstellt, welche in b gespeichert wird. Soll nun

in Zeile 7 die Variable *a* angezeigt werden, wird kein Fehler beim Kompilieren angezeigt. Der Eigentümer des Wertes hat sich nicht geändert und somit ist *a* weiterhin valide.

```
1 #[derive(Debug, Default)]
2 struct Large {/* discard */}
3
4 fn main() {
5     let a = Large::default();
6     let b = &a;
7     println!("var: {}", a); // ok
8 }
```

Listing (2.7): Beispiel für Referenzen in Rust.

Referenzen sind zunächst unveränderbar, siehe *Unveränderbar von Beginn*. Soll eine Referenz erstellt werden, über welche der referenzierte Wert verändert werden kann, muss dies explizit durch *mut* gekennzeichnet werden.

Für unveränderliche und veränderliche Referenzen auf dieselbe Variable gilt die Regel, dass es unbegrenzt viele unveränderliche Referenzen gleichzeitig geben darf, aber nur immer eine veränderliche [36]. Dies soll verhindern, dass die Variable während des Lesens verändert wird, und verhindert somit Data Races[1].

2.2.5 Sprachfunktionalität

Unveränderbar von Beginn

In C/C++ kann der Wert einer Variablen immer verändert werden, außer sie ist explizit mit *const* markiert. In Rust hingegen, sind alle Variablen zu Beginn unveränderlich. Der Programmierer muss eine Variable explizit als veränderlich markieren, um deren Wert zu ändern.

Durch diese Funktionalität kann bei erneutem Lesen des Codes erkannt werden, welche Variablen im Verlauf der Ausführung verändert werden [36]. In Listing 2.8 ist gezeigt, wie eine unveränderliche und eine veränderliche Variable deklariert wird.

```
1 let s = 12; // Nicht veränderbar
2 let mut m = 234; // veränderbar
```

Listing (2.8): Deklaration einer unveränderlichen und veränderlichen Variable.

Eine Modifikation einer unveränderlichen Variable wird in Rust während der Kompilierung überprüft [36]. Dabei wird durch eine Kompiliermeldung mitgeteilt, wenn eine Variable

[1] Tritt auf, wenn mindestens zwei Threads gleichzeitig auf dieselbe Variable zugreifen und einer versucht die Variable zu ändern. [7]

verändert wird, nachdem sie nicht explizit als veränderlich deklariert wurde. Eine solche Meldung ist in Listing 2.9 zu sehen.

```
error[E0384]: cannot assign twice to immutable variable `s`
--> src/main.rs:3:3
  |
1 |   let s = 23;
  |       -
  |       |
  |       first assignment to `s`
  |       help: consider making this binding mutable: `mut s`
2 |   s = 612;
  |   ^^^^^^^ cannot assign twice to immutable variable
```

Listing (2.9): Fehlermeldung beim Verändern einer unveränderlichen Variable.

Aus der Kompilierwarnung ist herauszulesen, dass die Variable *s* nicht explizit veränderlich ist, aber danach zu *612* verändert werden soll. Außerdem wird in Zeile 8 vorgeschlagen, die Variable *s* als *mut* zu deklarieren.

Unsicheres Rust

Die Speichersicherheit, welche Rust garantiert, kommt von einer statischen Programmanalyse und einem konservativen Compiler [36].

Dieser weist den Programmcode, für den zu wenig Information vorhanden ist, in Form einer Fehlermeldung schnell ab [36]. Demzufolge kann Code, welcher vom Programmierer als „sicher“ gesehen wird, aber nicht den Regeln vom Rust entspricht, nicht kompiliert werden.

Außerdem kann im „sicheren“ Rust auf keine Hardwarefunktionen zugegriffen werden, da diese laut [36] als unsicher gelten. Auch Aufrufe von in C geschriebenen Funktionen, wie es in *Fremdfunktionsschnittstelle von Rust* beschrieben wird, werden als unsicher deklariert.

Vermeintlich unsicherer Code wird in Rust mit einem *unsafe* Block gekennzeichnet [36], womit die Verantwortung für dessen Sicherheit dem Programmierer übergeben wird. Laut [36] muss der Code in solchen Blöcken nicht wirklich unsicher sein, sondern kann schlicht nicht vom Compiler auf Sicherheit überprüft werden.

Jedoch bleiben bei der Kennzeichnung mit *unsafe* die bereits angeführten Mechanismen wie Unveränderbar von Beginn und die Eigentümerschaft erhalten [36].

Eine weitere Anwendung der *unsafe* Blöcke besteht bei der hardwarenahen Programmierung, wobei Zeiger für den Speicherzugriff verwendet werden, was in Rust als unsicher gilt. Die Gültigkeit des Zeigers kann durch keinen Mechanismus durch den Compiler geprüft werden [36]. Dies kann dazu führen, dass der Zeiger auf einen nicht gültigen Speicher-

bereich verweist und dadurch unvorhersehbares Verhalten auftritt. Dies kann eintreffen, wenn der Zeiger von dem Programmierer selbst erstellt oder modifiziert wird.

Fremdfunktionsschnittstelle von Rust

In Rust ist es möglich, externe Funktionen, die in anderen Programmiersprachen geschrieben sind, aufzurufen. Hierfür wird das Interface als *Fremdfunktionsschnittstelle* (FFI) bezeichnet [9]. Derzeit wird hauptsächlich das Aufrufen von C Funktionen unterstützt [9].

Um eine externe Funktion aufzurufen, muss diese in Rust definiert werden. Dies kann sich wie eine Header-Datei in C vorgestellt werden [9]. Wie in Listing 2.10 in Zeile 8 zu sehen ist, muss hierfür ein Funktionskopf der Funktion definiert werden.

Die Funktion *call_function()* kann somit wie eine normale Funktion in Rust aufgerufen werden. Zu beachten ist, dass dieser Aufruf in einem *unsafe* Block gemacht werden muss, da der Compiler keine weiteren Informationen über die Funktion, wie zum Beispiel Lebenszeiten der Übergabeparameter, hat.

```
1 #[repr(C)]
2 pub struct AnyCStruct {
3     pub x: c_int,
4     pub y: c_int,
5 }
6
7 // Funktionsdefinition
8 pub extern "C" fn call_function(cs: *mut AnyCStruct);
```

Listing (2.10): Rust FFI Interface Funktionsdefinition.

2.2.6 Compiler

Rustc ist der Compiler von Rust. Zur Erstellung von ausführbaren Dateien wird *LLVM*[1] verwendet [38].

Rustc ist, genauso wie *CLANG*, ein Frontend, das den Rust Quellcode in *LLVM* Sprache übersetzt. Diese Zwischensprache wird als *LLVM IR* bezeichnet. *LLVM core* generiert daraus Maschinencode [102]. Dieser Ablauf wird in der Grafik Abbildung 2.1 dargestellt. Dabei ist die Kompilierung mit *LLVM* in zwei hintereinander ablaufende Prozesse aufgeteilt. Der erste optimiert den Code und der darauffolgende generiert schlussendlich den Maschinencode.

[1] LLVM ist eine Ansammlung modularer und wiederverwendbarer Compiler und Toolchain Technologien [102].

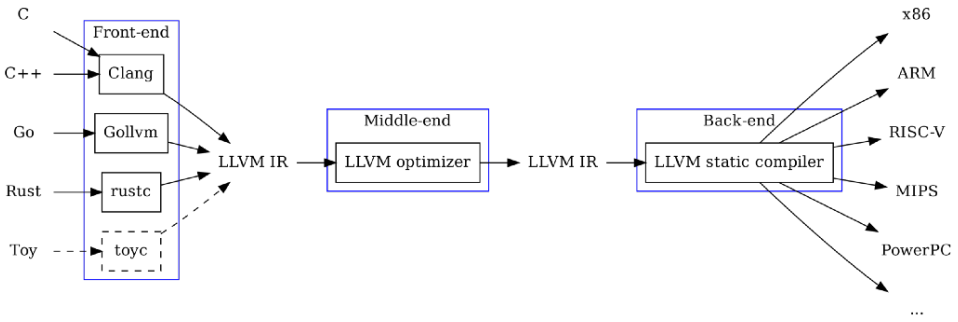

Abbildung (2.1): LLVM zur Verwendung als Compiler-Backend. [50]

Im Folgenden werden Marker erklärt, welche bei der Kompilierung zur Konfiguration des Compilers angegeben werden können. Dadurch soll entweder eine schnellere Kompilierung, eine kleinere kompilierte Binärdatei oder ein schnell ausführbares Programm erreicht werden.

Über den Marker *opt-level* können bei *rustc* verschiedene Optimierungsstufen aktiviert werden [35]. Als Wert kann eine Zahl zwischen null bis drei oder die Charaktere *s* und *z* angegeben werden [35]. Wird eine Zahl im genannten Bereich angegeben, optimiert der Rust Compiler den Quellcode, um eine bessere Ausführungsgeschwindigkeit zu erzielen.

Ist das *opt-level* gleich null, wird keine Optimierung vorgenommen. Je höher die Optimierungsstufe ist, desto mehr Optimierungen wendet der Compiler an und dementsprechend länger benötigt auch die Kompilierung [38]. Eine Optimierung kann zum einen Funktionseinbettung sein, zum anderen *loop vectorization*, was eine *LLVM* Funktion ist und Schleifen optimieren soll [24, 38].

Zudem wird bei eingebetteten Systemen häufig eine möglichst geringe Firmwaregröße benötigt. Hierfür kann entweder der Charakter *s* oder *z* angegeben werden. Diese Optionen fordern *rustc* auf, eine kleinere Binärdatei zu erstellen [38]. Zudem soll *z* die *loop vectorization*, welche den Quellcode auf Kosten größerer Binärdateien optimiert [24], ausstellen [38].

Ein weiterer Marker zur besseren Codegenerierung ist *codegen-units* [38]. Für diesen Marker ist ein Zahlenwert ab eins erlaubt, der die Anzahl an Kompilier-Einheiten angibt. Je größer der Wert ist, desto parallelisierter kann der Compiler den Code kompilieren. Dies kann jedoch zu einem langsamer ausführbaren Programm führen [38]. Wird der Wert auf eine Kompilier-Einheit gesetzt, muss der Compiler den Quellcode in einem Stück optimieren [38].

Zuletzt soll *lto*[38] betrachtet werden, das zusätzliche Optimierung aktivieren soll. Wird dieser Marker auf *true* gesetzt, führt der Compiler eine Optimierung für alle verwendeten Bibliotheken durch [38].

2.3 Controller Area Network (CAN)

Der *Controller Area Network* (CAN) Bus wurde 1983 von der Firma Bosch entwickelt [19, 21]. Die grundlegende Idee dahinter war, den Kabelbaum, vor allem in Auto, zu verkürzen [21]. Heute findet der CAN-Bus nicht mehr nur im Auto Anwendung, sondern auch in weiteren Bereichen wie der Medizintechnik, der Automatisierungstechnik [19, 21] und der Luftfahrt [4].

Im Folgenden soll zuerst auf *Classic-CAN* eingegangen werden. In Kapitel 2.3.5 werden Modifikationen für *Controller Area Network Flexible Data-Rate* (CAN FD) angeführt.

2.3.1 Systemübersicht

CAN ist ein Multimaster-System [19]. Das bedeutet, dass es keine zentrale Steuereinheit gibt. Die einzelnen Kommunikationspartner werden als Knoten bezeichnet [19]. Jeder Knoten kann zu jeder Zeit Nachrichten senden. Aus diesem Grund ist ein Arbitrierungsverfahren erforderlich [19].

Die Knoten in einem CAN-Bus haben keine Adressen [19]. Jede Nachricht hat eine Identifikationsnummer [19], anhand derer, die für dir Knoten bestimmten Nachrichten herausgefiltert werden. Die Kommunikationsbeziehung der Knoten basiert auf dem Producer-Consumer-Modell[1] [19]. In Abbildung 2.2 is ein Aufbau aus drei Knoten dargestellt. Jeder Knoten ist mit den jeweils anderen Knoten über eine Stichleitung am CAN-Bus angeschlossen.

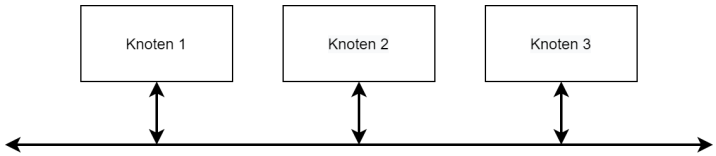

Abbildung (2.2): Schema eines CAN-Bus mit angeschlossenen Knoten.

Die räumliche Ausdehnung des CAN-Bus ist taktratenabhängig [19, 21]. Dies bedeutet, werden höhere Taktraten benötigt, muss die Leitungslänge verkürzt werden. So beträgt zum Beispiel die Taktrate von *High-Speed* CAN bis zu 1MBit [21], wobei eine maximale Leitungslänge von 50 Meter zulässig ist. In Tabelle 2.1 ist eine Übersicht über die Taktraten und die dazu maximalen Leitungslängen aufgelistet.

2.3.2 Physikalischer Aufbau

Für den physikalischen Aufbau soll die 2-Draht Differentialsignal-Übertragung[19] betrachtet werden. Der Bus besteht aus zwei Leitungen. Diese werden als *CAN high* und *CAN*

[1] Bei dem Producer-Consumer-Modell hat jede Nachricht einen Erzeuger und kann von einem oder mehreren Teilnehmern genutzt werden (Consumer). Dabei erreicht jede Nachricht jeden Knoten. [19]

CAN Variante	Taktrate	Ausdehnung
High-Speed	1 Mbit/s	40 m
	500 kbit/s	130 m
	250 kbit/s	270 m
Low-Speed	50 kbit/s	1,3 km
	10 kbit/s	6,7 km

Tabelle (2.1): Taktraten von CAN und die zugehörige maximale Ausdehnung des Bus. [21]

low bezeichnet und werden am jeweiligen Ende mit einem 120 Ω Widerstand verbunden [19].

Ein Knoten kann aus einem Mikrocontroller, der einen CAN-Controller[1] implementiert hat, bestehen. Mit dem CAN-Transceiver wird der Knoten an das physikalische Übertragungsmedium angeschlossen. Der CAN-Transceiver wandelt bidirektional die Signale zwischen dem Mikrocontroller und dem CAN-Bus um. Außerdem trennt dieser den CAN-Controller und den CAN-Bus galvanisch voneinander, sodass der Controller bei Überspannungen nicht zerstört wird [49].

Der Buspegel ist in Abbildung 2.3 dargestellt. Das übertragene Bit ist 1, wenn *CAN high* und *CAN low* dieselbe Spannung haben. Dies wird als rezessiver Pegel bezeichnet [19]. Eine 0 wird übertragen, wenn eine Differenz zwischen den Leitungen herrscht, was als dominanter Pegel bezeichnet wird [19]. In Abbildung 2.3 ist die Differenz U_{diff} 2 V.

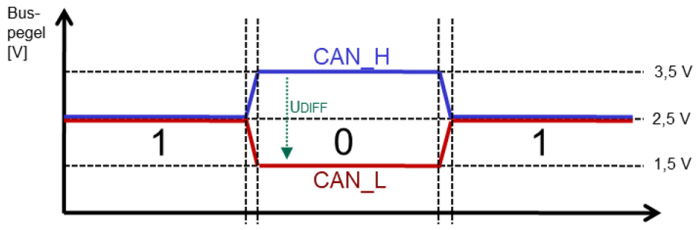

Abbildung (2.3): Pegel auf dem CAN-Bus. [19]

2.3.3 Aufbau von CAN-Nachrichten

Classic-CAN wird in zwei Formate eingeteilt:

- Standard-CAN (2.0 A)

[1] "Der CAN-Controller erbringt die vom CAN-Protokoll vorgeschriebenen Kommunikationsfunktionen, und entlastet dadurch weitestgehend den Host."[49]

- Extended-CAN (2.0 B)

Hauptsächlich werden diese Formate nach der Länge der verwendeten Identifikationsnummer unterschieden, welche in Kapitel 2.3.4 beschrieben wird.

In Abbildung 2.4 ist eine *Extended-CAN* Nachricht zu sehen. Das erste Bit ist immer *0*, darauf folgt die Identifikationsnummer. Diese ist bei *Extended-CAN* in zwei Teile aufgeteilt, damit die Formate untereinander kompatibel sind. Ein Bit vor der zweiten Hälfte der Identifikationsnummer signalisiert, dass es sich um eine erweiterte Nummer handelt. Im *DATA*-Feld befinden sich die Daten der Nachricht. Dieses Feld ist in seiner Länge, mit maximal 8 Byte, variabel. Die Länge wird durch das *DLC*-Feld angegeben [19].

Eine *Extended-CAN*-Nachricht hat eine Gesamtlänge l von mindestens 67 bit bis maximal 131 bit. Die Länge ist abhängig von der Anzahl p an *DATA*-Bytes und kann exakt mit $l = 67\,\text{bit} + p * 8\,\text{bit}\,\text{B}^{-1}$ berechnet werden [19].

Beim Transport auf dem CAN-Bus dürfen nicht mehr als fünf aufeinanderfolgende Bits gleich sein. Deshalb werden nach dem Erstellen einer Nachricht sogenannte *Stuffing*-Bits eingefügt, die jeweils eine solche Folge mit einer *1* oder *0* unterbrechen [19]. Die Bits werden beim Empfangen der Nachrichten ignoriert [19]. Laut [19] kann die Länge der CAN-Nachrichten auf dem Bus mit $l_{bus} \approx l * 1.25$ angenähert werden, wobei 25 % *Stuffing*-Bits angenommen werden.

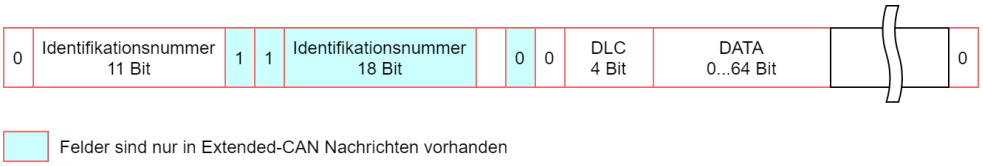

Abbildung (2.4): Aufbau einer Extended-CAN Nachricht. [19]

2.3.4 CAN-Identifikationsnummer

Die Identifikationsnummer soll zur Erkennung des Nachrichtentyps für die Knoten dienen. Bei *Standard-CAN* ist diese 11 Bit und bei *Extended-CAN* 29 Bit lang [19]. Somit können bei erst genanntem 2032 und bei letzterem 536 Millionen verschiedene Nachrichtentypen identifiziert werden [19].

Des Weiteren soll diese zur Busarbitrierung dienen. Da bei einem CAN-Bus alle Knoten an derselben Leitung angeschlossen sind, wird durch die Festlegung der Nachrichtennummer ebenso die Priorität der Nachricht festgelegt. Dabei ist die niedrigere Zahl immer dominierend.

Die Arbitrierung läuft wie folgt ab. Gibt ein Knoten seine Nachricht auf den Bus, überprüft dieser, ob der Pegel mit den gesendeten Bits übereinstimmt. Im Falle, dass zwei Teilnehmer gleichzeitig senden, wird derjenige Teilnehmer, der eine 1 sendet bemerken, dass der Pegel

sich nicht anpasst, da die 0 des anderen dominiert. Nun bricht dieser Knoten das Senden ab und wiederholt es zu einem späteren Zeitpunkt [19].

2.3.5 CAN FD

CAN FD ist laut [19] im Jahr 2012 als Erweiterung für das CAN-Protokoll vorgestellt worden. Der Grund hierfür war, größere Datenmengen verarbeiten und höhere Datenraten erzielen zu können. Außerdem soll es zu *Classic-CAN* kompatibel sein [19].

Bei CAN FD kann sowohl das *Standard-CAN* als auch das *Extended-CAN* Format verwendet werden. Das Datenfeld wird auf bis zu 64 Byte erweitert [19]. Das *DLC*-Feld behält die Länge von 4 bit, dabei ist die Datenlänge in Stufen über eine Tabelle wählbar [19]. Über 8 Byte wächst die verwendbare Länge bis 24 Byte in 4er-Schritten und danach in 8 Byte und 16 Byte Schritten.

Damit die Datenraten des *Classic-CAN* eingehalten werden können, gibt es zwei Übertragungsraten beim Senden. Mit der Rate des *Classic-CAN* werden Bits für Identifikationsnummer, Marker und für das Ende der Nachricht übertragen. Das *DLC*-Feld und das *DATA*-Feld können mit einer Rate von bis zu 8 Mbit/ sec gesendet werden [19]. Dies muss vorab bei allen Knoten konfiguriert werden [19].

3 UAVCAN

3.1 Entwicklung

UAVCAN[6] ist ein Projekt, das UAVCAN Standard spezifiziert. Auf der offiziellen Website[6] wird explizit darauf hingewiesen, dass der Standard auf unbegrenzte Zeit ohne Lizenzen verwendet werden kann. Der Standard wird öffentlich über das UAVCAN Forum, Software-Repositories und Ressourcen, welche über die Website zur Verfügung stehen, entwickelt [48].

Das UAVCAN Konsortium ist nicht gewinnorientiert, sondern möchte den Standard unterstützen, die Technologie weiter standardisieren und das Ökosystem erweitern [6]. Diesem Konsortium kann man kostenlos beitreten und dabei Vorteile bei der Verwendung in Projekten nutzen [6].

Aktuell ist der Standard in der Version *1.0-beta* [48]. Zuvor gab es eine experimentelle *UAVCAN v0* Version [41]. Der neuere Standard hat das Kommunikationsprotokoll vereinfacht und außerdem Design Probleme behoben [41]. Dadurch, dass die alte Version von Herstellern noch verwendet wird, wird diese weiterhin gewartet, aber nicht weiterentwickelt [41].

Der UAVCAN Standard kann mit Hilfe der Spezifikation selbst implementiert werden. Dies soll in 1000 logischen Codezeilen umsetzbar sein [6]. Außerdem gibt es Open-Source Implementationen des Standards, welche unter MIT[1] lizenziert sind.

Die Tabelle 3.1 soll die zum Zeitpunkt dieser Arbeit verfügbaren und auf der Website referenzierten Implementationen aufzeigen.

3.2 UAVCAN Standard

UAVCAN ist ein offenes, leichtgewichtiges Protokoll, welches zur zuverlässigen, deterministischen Kommunikation im Bereich der Fahrzeugtechnik, der Luft- und Raumfahrttechnik und im Robotik Bereich entwickelt wird [48].

[1] Die MIT Lizenz soll jeden Entwickler dazu berechtigen, die Software zu benutzen, zu bearbeiten, weiterzuverteilen und zu verkaufen. Ebenso können neue Lizenzen hinzugefügt werden. Dabei muss zu jeder Zeit der MIT Lizenztext beinhaltet sein. [23]

Bibliothek	Unterstützte Transportarten	Programmier-sprache	Einsatzgebiet	Entwicklungs-stadium
Libcanard	UAVCAN/ CAN	C11	Eingebettete Systeme	Released
PyUAVCAN	Jegliche	Python	HMI, Entwick-lung	Released
Libuavcan	Jegliche	C++11	Eingebettete Systeme	in Bearbeitung, ETA 2021Q4

Tabelle (3.1): Eine Auflistung der aktuell verfügbaren UAVCAN Implementationen. [6]

Das Protokoll unterstützt das Publisher-Subscriber[1]-Kommunikationsmodell, sowie das Aufrufen von Prozeduren auf anderen Geräten.

In UAVCAN ist der Broker, welcher die Nachrichten beim Publisher-Subscriber-Modell verteilt, dezentralisiert und wird mit einer Interface-Beschreibungssprache spezifiziert [48].

3.2.1 Fähigkeiten

Der UAVCAN Standard definiert Fähigkeiten und Standardgrenzen, welche durch die Implementation gewährleistet werden sollen. Im Folgenden sind diese in Stichpunkten aufgelistet.

- Es soll mindestens[2] 128 identifizierbare Teilnehmer im System geben können [48].
- UAVAN unterstützt unbegrenzt viele Datentypen, welche selbst erstellt und mit Versionen versehen werden können [48].
- Zum Nachrichtenaustausch können 8192 Nachrichtentyp-Identifikationsnummern vergeben werden [48].
- Für den Aufruf von externen Funktionen anderer Teilnehmer sind 512 Identifikationsnummern verfügbar [48].
- Es sollen mindestens[2] acht Prioritäten-Level für Nachrichten unterstützt werden [48].
- In der Applikationsschicht sollen verschiedene Funktionen des Protokolls implementiert sein, wie z.B. Zeitsynchronisation [48].
- Der Standard definiert keine Anforderungen für die physikalische Übertragung [48].

[1] Bei dem Publisher-Subscriber-Modell wird jede Nachricht von einem Teilnehmer, auch Publisher, über das Kommunikationssystem an einen oder mehrere Teilnehmer, die Subscriber, versendet [19].
[2] Die genaue Zahl ist abhängig von dem verwendeten Transportprotokoll .

3.2.2 Protokollaufbau

Der UAVCAN Standard teilt den Kommunikationsstack in drei Schichten ein [41].

- Applikationsschicht
- Präsentationsschicht
- Transportschicht

Dabei ist die Transportschicht der Hardware am naheliegendsten und die Applikationsschicht steht dem Programmierer für höhere Schichten zur Verfügung. Die Präsentationsschicht ist zwischen diesen beiden Schichten einzuordnen [48], zu sehen in Abbildung 3.1.

Applikationsschicht

Dadurch, dass UAVCAN für verschiedene Anwendungszwecke einsetzbar sein soll, werden durch den Standard grundlegende Funktionen für die Applikationsschicht definiert. Dazu gehören folgende Funktionen [41]:

- Diagnostikfunktionen
- Überwachung von Knoten
- Konfiguration des Systems

Weitere Funktionen sollen für den jeweiligen Einsatzbereich vom Benutzer individuell spezifiziert werden [41].

Präsentationsschicht

Die Präsentationsschicht soll eine typisierte Präsentation der Nachrichten in der Applikationsschicht ermöglichen. Dies wird durch Deserialisierung der Daten aus dem Transport erreicht. Wird eine Nachricht aus der Applikationsschicht erstellt, soll diese serialisiert werden.

Transportschicht

Die Transportschicht soll die Daten über ein wählbares Transportprotokoll verschicken [41]. Die Funktionalität des Transports soll an die Hardware angepasst sein [41]. Ein Beispiel hierfür ist das Filtern von Paketen.

Ein CAN-Controller, wie er in Kapitel 2.3.2 beschrieben wird, hat eine Filterung auf CAN Identifikationsnummern implementiert. Diese Hardwareimplementierung soll vom jeweiligen Transport genutzt werden. So wird Laufzeit eingespart, welche benötigt wird, um Pakete in der Software zu analysieren [41].

Während dieser Arbeit sind für UAVCAN zwei Transportprotokolle vorgesehen. Dies ist zum einen der Transport über CAN-Bus, was als *UAVCAN/CAN* bezeichnet wird und zum anderen der Transport über IP/UDP, *UAVCAN/UDP* genannt. Dies ist in Abbildung 3.1 in der Transportschicht abgebildet.

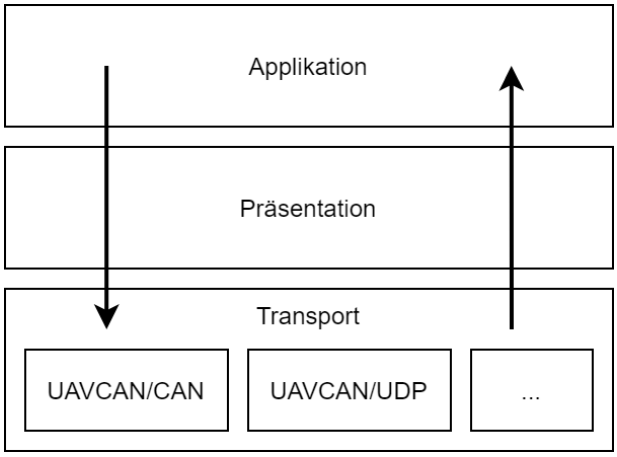

Abbildung (3.1): Die Schichten des UAVCAN Stacks.

Im Folgenden wird genauer auf *UAVCAN/CAN* eingegangen, da dies der genutzte Transport für die Implementation in der Arbeit ist.

3.2.3 Grundkonzepte beim Datenaustausch

In einem UAVCAN Netzwerk hat laut [48] nahezu jeder Knoten eine eindeutige Identifikationsnummer, auch *node*-ID genannt. Wie in Kapitel 3.2.1 aufgeführt, soll diese bis mindestens 127 gehen. Dies ist genauer durch das Transportprotokoll begrenzt. Bei CAN ist die *node*-ID durch die Größe der CAN-Identifikationsnummer, welche in Kapitel 2.3.4 beschrieben wird, eingeschränkt.

In Abbildung 3.2 ist dargestellt, wie die CAN-Identifikationsnummer bei UAVCAN verwendet wird. Dabei wird an erster Stelle die Priorität der Nachricht festgelegt. Die *node*-ID wird durch 7 Bit an letzter Stelle angegeben. Dies erlaubt die Mindestanzahl von 128 identifizierbaren Teilnehmern im System.

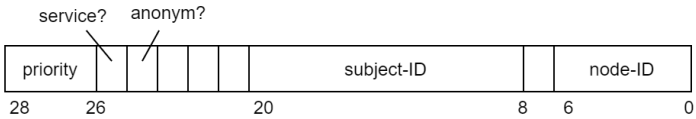

Abbildung (3.2): Felder der CAN-Identifikationsnummer in UAVCAN.

Ein Knoten kann auch als anonym initialisiert werden, wobei dieser dann keine *node*-ID erhält [48]. Wenn ein Knoten ohne Identifikationsnummer initialisiert wird, kann er

bei aktivem Netzwerk durch ein *plug-and-play* Verfahren eine *node*-ID bekommen. Diese Funktion wird durch das *anonym* Bit in der CAN-Identifikationsnummer repräsentiert, was in Abbildung 3.2 zu sehen ist. In dieser Arbeit wird nicht weiter darauf eingegangen.

In UAVCAN gibt es zwei Mechanismen für die Kommunikation.

Nachrichten Verteilen Datenaustausch mit der Beziehung eins zu vielen, was mit dem Publisher-Subscriber-Modell implementiert ist [48].

Externer Funktionsaufruf Eins zu eins Interaktion zwischen Knoten nach dem Request/-Response Prinzip [48]. Dadurch ist das *Client-Server-Modell*[1] umgesetzt. Wird bei UAVCAN als *service* bezeichnet.

Im Weiteren soll hauptsächlich der Fokus auf das klassische Nachrichten-Verteilen gelegt werden, da dies die Hauptfunktion von UAVCAN ist [48]. Die Funktionsaufrufe externer Funktionen haben eine ähnliche Implementation. Unterschiede bestehen hauptsächlich in Namenskonventionen und dem Ablauf der Kommunikation. Bei dem CAN Transport wird dies durch das *service* Bit gekennzeichnet, siehe Abbildung 3.2.

Eine Nachricht ist durch eine *subject*-ID[2] gekennzeichnet. Diese steht vor der *node*-ID, siehe Abbildung 3.2, und hat eine Länge von 13 Bit, was zu den 8192 identifizierbaren Nachrichtentypen aus Kapitel 3.2.1 führt.

3.2.4 Senden/ Empfangen von Übertragungen

Bei UAVCAN wird der Austausch von Nachrichten auch als Übertragung bezeichnet. Die Übertragung beinhaltet die Daten der Nachricht, einen Zeitstempel für deren Erstellung und eine *subject*-ID.

Bei einem Nachrichtenaustausch können unterschiedlich viele Bytes übertragen werden. Da das verwendete Transportprotokoll jedoch nur eine bestimmte Byte-Anzahl pro Datenrahmen zulässt, muss die Übertragung segmentiert werden. Dies geschieht bei UAVCAN in der Transportschicht. Dabei wird zwischen zwei Szenarien unterschieden. Passen die Daten beim Transport in einen Datenrahmen, wird dies als Einzelrahmenübertragung bezeichnet, wenn mehrere Rahmen benötigt werden, ist dies eine Vielrahmenübertragung.

In Abbildung 3.3 sind diese zwei Fälle und deren Rahmenlayouts dargestellt. Hierbei wird ein Transport über *Classic*-CAN, wobei die maximale Datenlänge 8 Byte beträgt, betrachtet. Die linke Seite veranschaulicht die Übertragung bei einem Rahmen. Dies ist dann möglich, wenn die Daten 7 Byte oder weniger lang sind. Das achte Byte wird dabei als Ende-Byte verwendet, welches zusätzliche Informationen, wie die Transport-ID und Transportsteuerung-Bits, beinhaltet.

[1] Bei dem *Client-Server-Modell* fragt ein Client bei einem Server Informationen oder einen Dienst an. Der Server liefert daraufhin die Daten oder stellt den angeforderten Dienst zur Verfügung. [19]

[2] In der Spezifikation wird die *subject*-ID auch als *port*-ID bezeichnet. *port*-ID ist dabei der Überbegriff von *subject*-ID und *service*-ID, welche für Funktionsaufrufe verwendet wird.

Auf der rechten Seite in Abbildung 3.3 ist eine Vielrahmenübertragung dargestellt. Die Datenübertragung umfasst in diesem Fall 19 Byte und wird beim Senden in drei CAN-Rahmen aufgeteilt.

Jedes dieser Segmente hat ein Ende-Byte. Das Ende-Byte beinhaltet Bits zur Transportflusssteuerung. Eine Kombination zweier Bits lässt auf die Position eines Rahmens schließen. Ist bei einem Rahmen das erste Bit gesetzt, ist dies der Start der Übertragung und ist das jeweils andere Bit gesetzt, ist dies der letzte Rahmen. Bei dazwischenliegenden Datenrahmen sind diese Bits 0. Ein weiteres Bit wird pro Rahmen umgeschaltet, um zu signalisieren, dass kein Rahmen verloren gegangen ist.

Zusätzlich wird bei einer Vielrahmenübertragung eine CRC Prüfsumme über die zu sendenden Bytes berechnet und im letzten Rahmen eingefügt. Dies ist durch die gelben Bytes veranschaulicht.

Beim Empfangen der Übertragung wird das Ende-Byte überprüft und die Informationen über den Rahmen herausgenommen. Rahmen mit derselben Transport-ID gehören zusammen. Das Ende-Byte signalisiert den Start und das Ende einer Übertragung. Außerdem wird erneut die CRC Prüfsumme über die empfangenen Bytes berechnet und mit der gesendeten Prüfsumme abgeglichen.

Übertragungen haben, wie erwähnt, einen Zeitstempel für deren Erstellung. Dadurch kann überprüft werden, ob die einzelnen Rahmen bei einer Vielrahmenübertragung in der erforderlichen Zeit gesendet werden. Wird ein Rahmen zu spät gesendet, bricht die Übertragung ab. Auch der Knoten, welcher die Rahmen empfängt, überprüft diese Zeit und bricht bei Überschreitung das Empfangen ab.

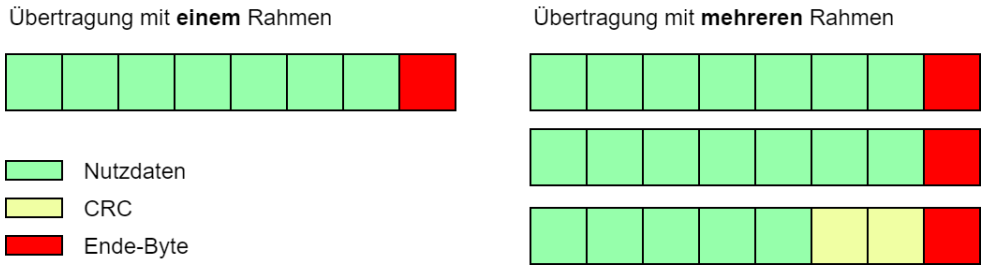

Abbildung (3.3): UAVCAN Rahmen-Layout in Bytes für Übertragungen.

Beim Empfangen von Übertragungen muss die Bibliothek zwischen empfangenen Rahmen selektieren. Dies basiert auf der *subject*-ID, welche abonniert wird.

Außerdem müssen bei Vielrahmenübertragungen die bereits empfangenen Bytes zwischengespeichert werden. Des Weiteren muss dabei zwischen verschiedenen Übertragungen unterschieden werden.

Hierfür ist ein Datenmodell in Abbildung 3.4 zu sehen. Für eine Übertragung wird eine Session definiert. Diese kann durch *subject*-ID, *node*-ID und Transport-ID eindeutig

identifiziert werden. Ein Rahmen kann dadurch einer entsprechenden Session zugewiesen werden.

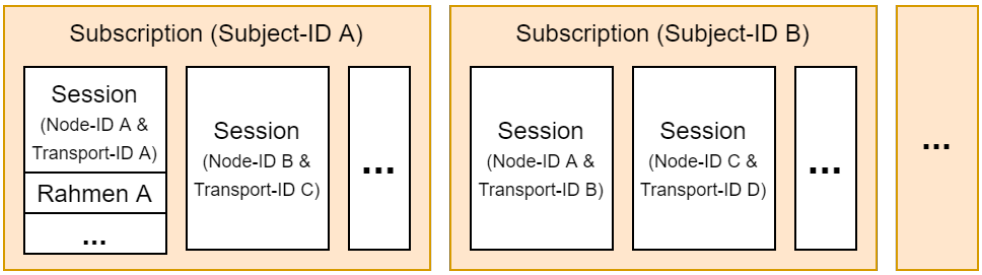

Abbildung (3.4): Modell für das dynamische Empfangen von Übertragungen.

3.3 API der Bibliotheken (C++/Rust)

In diesem Abschnitt wird ein Überblick über die API der UAVCAN Bibliotheken zum einen in C++/Arduino und zum anderen in Rust gegeben. Außerdem wird jeweils die Verwendung des dynamischen Speichers betrachtet, da dies bei den späteren Zeitmessungen und Speichermessungen mitberücksichtigt werden soll.

3.3.1 C++/Arduino API

In dieser Arbeit ist mit Arduino- und C++-Implementation dieselbe Bibliothek gemeint. Hierbei handelt es sich hauptsächlich um eine in C geschriebene UAVCAN-Implementation. Diese ist mit einer C++ API umhüllt. Diese Implementation ist für die Arduino-Plattform geschrieben. Die API ist in Listing 3.1 in Form eines Beispielaufbaus zu sehen.

Für die Implementation müssen zwei Rückruffunktionen definiert und implementiert werden. Welche in diesem Fall als *transmit_func()* und *on_data_receive()* bezeichnet werden. Dabei wird *transmit_func()* für jeden Rahmen aufgerufen, welcher auf den CAN-Bus gegeben werden soll. Hingegen wird *on_data_receive()* aufgerufen, wenn eine komplette UAVCAN Übertragung eingegangen ist. Dadurch ist gewährleistet, dass der Programmierer das jeweilige weitere Vorgehen beim Senden und Empfangen implementieren kann.

In Zeile 7 wird ein UAVCAN Knoten mit der ID 13 und der Empfangs-Rückruffunktion als globale Variable initialisiert. Des Weiteren wird in Zeile 10 einmalig die *subject*-ID *PORT_ID* abonniert. Folglich soll bei jedem Erhalt einer Nachricht dieser *subject*-ID die Funktion *on_data_receive()* aufgerufen werden.

In der Hauptschleife, welche die Funktion *loop()* darstellt, wird im ersten Bereich eine Nachricht versendet. Dazu wird der Nachrichten-Typ erstellt und an die UAVCAN-Implementation mit *publish()* übergeben. Intern wird die Nachricht in die zu sendenden Rahmen zerlegt, siehe Kapitel 3.2.4, und in einer Liste zwischengespeichert. In Zeile 25

wird die entsprechende Sendefunktion für die zuvor zwischengespeicherten Rahmen aufgerufen.

Die *publish()* Funktion wird bei späteren Messungen nicht verwendet. Stattdessen wird die Funktion *enqueueTransfer()* aufgerufen. Diese ist normalerweise privat, wird aber hierfür als öffentlich markiert. Sie erlaubt den Transport von Bytes und verwendet keine Datentypen.

Im zweiten Teil des Hauptprogramms werden die Rahmen empfangen. Dabei wird jeder Rahmen mit der Funktion *onCanFrameReceived()* der UAVCAN Bibliothek übergeben. Sobald eine komplette Übertragung eingegangen ist und die *subject*-ID dieser mit einer abonnierten ID übereinstimmt, wird die Rückruffunktion *on_data_receive()* aufgerufen.

```
1  // Sende-Callback
2  bool transmit_func(CanardFrame const &frame);
3  // Empfangs-Callback
4  void on_data_receive(CanardTransfer const &transfer, ArduinoUAVCAN &
       uavcan);
5
6  // global static
7  ArduinoUAVCAN uc(13, transmit_func);
8
9  void setup() {
10     uc.subscribe<Bit_1_0<PORT_ID>>(on_received);
11 }
12
13 void loop() {
14     // Senden
15     if (send) {
16         // Nachricht erstellen
17         SynchronizedTimestamp_1_0<PORT_ID> time;
18         time.data = uavcan_time_SynchronizedTimestamp_1_0 { 1234 };
19
20         // Nachricht an Bibliothek übergeben
21         uc.publish(time);
22     }
23
24     // alle zu sendenden CAN Rahmen senden
25     while (uc.transmitCanFrame()) {}
26
27     // Empfangen
28     if (received_on_line) {
29         // CAN HAL Rahmen in CanardFrame Typ konvertieren
30         CanardFrame frame;
31         // setzen von Rahmen Infos
32
33         // Empfangen Rahmen der Bibliothek übergeben
```

```
34          uc.onCanFrameReceived(frame);
35      }
36  }
```

Listing (3.1): Übersicht über die C++/Arduino API.

Anmerkungen zur dynamischen Speichernutzung (Arduino)

Die Verwendung und der Ablauf des dynamischen Speichers ist für das spätere Messen von Ausführungszeiten wichtig, da das Anfordern von Speicher und dessen erneute Freigabe Zeit benötigt. Durch die folgenden Anmerkungen können die Messungen besser zwischen den Bibliotheken in Bezug gesetzt werden.

Die Arduino-Implementation verwendet beim Senden und Empfangen von Rahmen, sowie beim Abonnieren von *subject*-IDs dynamischen Speicher. Dieses Abonnieren wird in der Regel zu Beginn der Ausführung vorgenommen und spielt für den weiteren Ausführungsverlauf von Senden und Empfangen keine wesentliche Rolle.

Wenn die Funktion *publish()* in Zeile 21 aufgerufen und die Übertragung in der Bibliothek in einzelne Rahmen aufgeteilt wird, wird jeder Rahmen in einer Liste gespeichert. Für jede zu speichernde Struktur wird Speicher angefordert. Beim Senden der Rahmen in Zeile 25 wird der Speicher wieder nacheinander freigegeben. Dafür wird für diese Spanne, in Abhängigkeit der Rahmenanzahl, Speicher allokiert.

In Abbildung 3.5 ist der Ablauf des Anforderns und Wiederfreigebens von Speicher beim Empfangen von Rahmen dargestellt. Das Modell für diesen Vorgang ist in Kapitel 3.2.4 geschildert. Es wird einmalig für das Managen einer neuen Session Speicher allokiert. Dies tritt ein, wenn eine Übertragung mit einer *subject*-ID und *node*-ID empfangen wird. Anschließend wird für die Spanne bis zur vollständigen Übertragung Speicher für diese angefordert und danach wieder freigegeben. Ein Worst-Case ergibt sich beim Empfangen der ersten Übertragung einer Session. Der Best-Case ergibt sich bei darauffolgenden Übertragungen für diese Session.

3.3.2 Rust API

Die Rust-API verändert sich während dieser Arbeit. Die Änderungen werden während der Portierung auf ein eingebettetes System in Kapitel 4.1 und im Laufe der Messungen in Kapitel 5.3.2 durch Ausführ-Optimierungen vorgenommen.

In Listing 3.2 ist der Stand der Rust-API zum Ende der Arbeit hin zu sehen, da sich dieser während der Arbeit nicht wesentlich verändert.

Für die Rust-Implementation wird eine extern definierte Uhr benötigt. Diese wird in Zeile 3 definiert. Außerdem wird ein *SessionManager* gebraucht. Dieser beinhaltet die Verwaltung

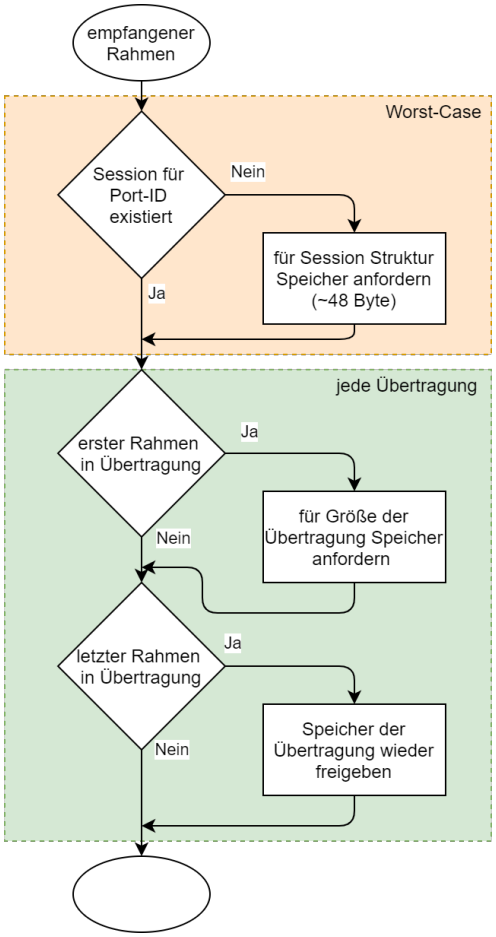

Abbildung (3.5): Dynamischer Speicher allokieren/freigeben beim Empfangen von Rahmen.

des dynamischen Speichers und wird speziell in Kapitel 5.3.2 angepasst. Zudem wird noch die *subject*-ID *PORT_ID* abonniert.

Der Knoten wird schlussendlich in Zeile 14 mit dem *SessionManager* und der zuvor initialisierten Uhr erstellt.

Das Senden einer Übertragung wird in Listing 3.2 ab Zeile 18 dargestellt. Die Übertragung wird durch eine *Transfer* Struktur erstellt und anschließend ein Iterator durch die UAVCAN Bibliothek erzeugt. Diese Hilfsstruktur erlaubt es in der folgenden *while*-Schleife die einzelnen zu sendenden Rahmen abzufragen und auf den CAN-Bus zu geben.

In Zeile 28 bei dem Empfangen von CAN-Rahmen, wird der darauffolgende Code ausgeführt. Dabei wird die CAN-Nachricht in einen Rahmen der UAVCAN Bibliothek umgewandelt und mithilfe der Funktion *try_receive_frame()* verarbeitet. Ein Rahmen wird dann angenommen, wenn die entsprechende *subject*-ID zuvor abonniert wurde. Ist eine Übertragung vollständig angekommen, gibt die genannte Funktion die Übertragung als Rückgabewert zurück.

```rust
fn main() {
    // Initialisieren der Applikations-Uhr
    let clock = StmClock::new();

    let mut session_manager = HeapBasedSessionManager::<CanMetadata>::new();
    session_manager.subscribe(Subscription::new(
        TransferKind::Message,
        PORT_ID,
        7, // Maximale Größe der Übertragungen (Byte)
        Milliseconds(500)
    ));

    // Initialisieren des Knotens für CAN Transport
    let mut node = Node::<_, Can, _>::new(Some(13), session_manager, clock);

    loop {
        // Senden
        if send {
            // Übertragung erstellen
            let transfer = Transfer::new(...);

            // Übertragung in CAN Rahmen zerlegen
            let frame_iter = node.transmit(&transfer).unwrap();
            while Some(frame) = frame_iter.next() {}
        }

        // Empfangen
        if received_on_line {
            // CAN HAL Rahmen in UavcanFrame Typ konvertieren
            let mut uavcan_frame = UavcanFrame::new(...);

            // Rahmen verarbeiten
            if let Some(transfer) = node.try_receive_frame(uavcan_frame) {}
        }
    }
}
```

Listing (3.2): Übersicht über die Rust API.

Anmerkungen zur dynamischen Speichernutzung (Rust)

Wie bereits in Kapitel 3.3.1 aufgeführt, ist die dynamische Speichernutzung für die Analyse der Messergebnisse wichtig.

Im Gegensatz zur C++-Implementation, verwendet die Rust-Implementation nur beim Abonnieren von *subject*-IDs und beim Empfangen von Übertragungen dynamischen Speicher. Beim Senden kommt diese mit statisch allokiertem Speicher aus.

Wie bereits erwähnt, hat sich während der Zeitmessungen die Verwendung der dynamischen Speicherverwaltung beim Empfangen verändert. Deshalb wird im Folgenden dieses Verhalten zum Ende der Arbeit beschrieben.

Für eine Übertragung, für diese es noch keine Session mit der jeweiligen *session*-ID und *node*-ID gibt, wird eine neue Session erstellt. Dabei wird nicht zwischen Speicher für Managen und Übertragen unterschieden, wie das der Fall bei der Arduino-Implementation ist. Es wird Speicher in der Größe für beides zusammen angefordert. Eine Session bleibt solange bestehen, bis die Ausführung beendet ist. Dieses Verhalten weicht von der Arduino-Implementation insofern ab, dass diese nach einer erfolgreichen Übertragung den Speicher für die Übertragung wieder freigibt.

4 Praktische Anwendung von Rust

In diesem Kapitel wird nun die zur Zeit der Arbeit in Entwicklung stehende Rust-Implementation des UAVCAN Standards auf ein eingebettetes System portiert. Zudem wird ein Programm entwickelt, das grundlegende Funktionen beinhaltet, damit dieses auf einem Mikrocontroller ausgeführt werden und als Basis für weitere Programme, wie zum Beispiel eines Testprogramms zur Evaluation der Portierung, dienen kann. Anschließend wird die Architektur einer entwickelten Benchmark-Suite vorgestellt.

4.1 Portierung der Bibliothek auf eingebettete Systemumgebungen

Die Implementation ist vor dieser Arbeit ausschließlich für den Einsatz auf Systemen mit dem Betriebssystem Linux vorgesehen. Für das Verwenden der Bibliothek auf einem Mikrocontroller muss diese ausschließlich Abhängigkeiten zu den darauf verfügbaren Bibliotheken und Funktionalitäten besitzen.

In Rust wird eine Bibliothek für eingebettete Systeme, genauer gesagt für Umgebungen, bei welchen die Standardbibliothek von Rust nicht verfügbar ist, mit dem Attribut *no_std* markiert. Damit wird verhindert, dass die Standardbibliothek in das Programm hineingelinkt wird.

Für die Portierung der Rust Bibliothek müssen zwei größere Gebiete betrachtet werden. Zum einen die Speicherverwaltung und zum anderen die Erfassung der Systemzeit auf eingebetteten Systemen.

1. In der UAVCAN Implementation für Betriebssysteme wird ein sogenannter *SessionManager* als *Trait*[1] implementiert. Dieser wird verwendet, um die Speichernutzung beim Empfangen von Übertragungen zu abstrahieren. Für Betriebssysteme gibt es eine Implementation dieses *SessionManager*s, der die Standardbibliotheks-Container verwendet. Dazu zählen die *Vec*[2] und *HashMap*[3] Strukturen.

   Für die Verwendung auf einem eingebetteten System wird nun ein angepasster *Ses-*

[1] Soll dem Rust Compiler Funktionalität eines Typs mitteilen, die dieser besitzt und mit anderen teilen kann. Ist gleichzusetzen mit einem Interface [36].
[2] Ein Array, das dynamisch zur Laufzeit unter der Verwendung des Heap Speichers wachsen kann [29].
[3] Implementation einer Hashtabelle zur Speicherung von Daten welche durch Schlüsselwörter indiziert werden können [29].

sionManager implementiert. Dieser hat ausschließlich Abhängigkeiten zu den *core*[1] und *alloc*[2] Bibliotheken. Hieraus werden die gleichnamigen Strukturen, wie aus der Standardbibliothek, verwendet. Außerdem wird hierbei weiterhin ein Heap Speicher eingesetzt, der vom Programmierer in der Software implementiert werden muss. Die gesamte Implementationslogik wird vom *SessionManager* für Betriebssysteme übernommen.

Der angepasste *SessionManager* kann auf allen Systemen, mit oder ohne Betriebssystem, verwendet werden, solange ein globaler Allokator definiert ist. Ein solcher Allokator wird bei Umgebungen, bei denen die Standardbibliothek verwendet wird, automatisch implementiert.

2. Des Weiteren muss die UAVCAN Bibliothek die Möglichkeit haben, die aktuelle Zeit des Systems zu erfassen. Dies wird auf einem Betriebssystem durch die Standardbibliothek abstrahiert. In Rust wird ein monotoner Zeitstempel mit *std::Instant::now()* abgerufen. Auf einem eingebetteten System steht eine solche Implementation, ohne weitere Bibliotheken, nicht zur Verfügung. Deshalb wird eine generische Implementation gewählt, bei dieser der Programmierer der UAVCAN Bibliothek eine Uhr übergeben muss.

   Für die Implementation wird die Bibliothek *embedded-time* verwendet, die *Traits* für eine solche Uhr und weitere Strukturen zur Nutzung von zeitabhängigen Implementationen implementiert.

   Strukturen der UAVCAN Bibliothek, die Systemzeit benötigen, haben nun einen zusätzlichen generischen Typparameter. In Listing 4.1 ist dies am Beispiel der *Node* Struktur gezeigt. Der Typname *C* wird in Zeile 4 auf Typen, die das *Trait Clock* implementieren, beschränkt. Dadurch kann bei der Erstellung der *Node* eine für das System angepasste Uhr übergeben werden, siehe in Listing 3.2 Zeile 14.

   Nun kann die aktuelle Systemzeit durch den Aufruf der Funktion *try_now()*, die durch das *Trait Clock* implementiert werden muss, auf der generischen Uhr abgefragt werden, siehe Zeile 15.

```
1  struct Node<..., C>
2  where
3      // beschränkt Typ C auf Typen, die Clock implementieren
4      C: embedded_time::Clock
5  {
6      ...,
7      clock: C
8  }
9
10 impl<..., C> Node<C>
```

[1] Ist eine abhängigkeitslose Grundbibliothek der Rust Standardbibliothek, die Grundtypen von Rust beinhaltet [32].
[2] Beinhaltet Strukturen zur dynamischen Speicherverwaltung [31].

```
where
    C: embedded_time::Clock
{
    fn act_timepoint(&self) -> embedded_time::Instant<C> {
        self.clock.try_now().unwrap()
    }
}
```

Listing (4.1): Definition und Implementation der Node Struktur mit einer generischen Uhr.

Für den Fall, dass die Bibliothek auf einem System mit Betriebssystem verwendet wird, wird eine Uhr, die das *Trait Clock* implementiert und in der Funktion *try_now()* die Funktion *std::Instant::now()* aufruft, implementiert. Diese Uhr ist jedoch nur verfügbar, wenn die UAVCAN Bibliothek speziell für Betriebssysteme mit der Bibliotheksfunktionalität *std* kompiliert wird.

Diese Implementation erlaubt es, dass die Bibliothek sowohl auf Systemen mit Betriebssystem weiterverwendet, als auch angepasst auf anderen Systemen verwendet werden kann.

4.2 Verwendete Hardware

Die in der Arbeit verwendete Hardware besteht aus zwei *NUCLEO-G431RB* Entwicklungsboards. Diese sind jeweils mit einem CAN-Transceiver, welche in Kapitel 2.3.2 beschrieben wurden, an die Bus Leitungen angeschlossen. Der Bus ist mit zwei 120 Ω Widerständen terminiert. Abbildung 4.1 veranschaulicht diesen Aufbau.

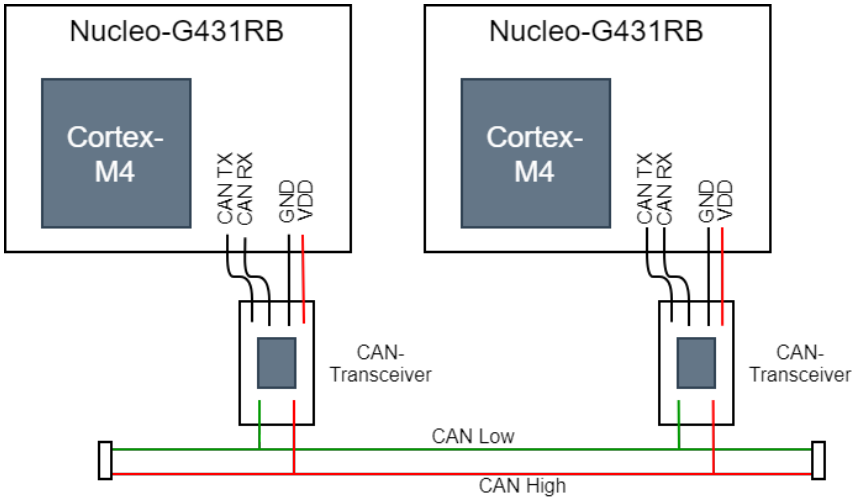

Abbildung (4.1): Hardwareaufbau bestehend aus zwei Mikrocontrollern und dem CAN-Bus.

NUCLEO-G431RB

Das *NUCLEO-G431RB* stellt eine einfache Möglichkeit dar, prototypisch auf der *STM32G4* Mikrocontroller Serie zu entwickeln. Das Board legt die Pins des Controllers auf einen ARDUINO Uno V3 Verbinder und auf einen ST morpho Verbinder [42]. Zum Programmieren wird keine externe Programmiersonde benötigt, da ein STLINK-V3E Debugger/ Programmierer auf dem Board integriert ist.

Die CPU auf dem Board ist ein Arm Cortex-M4, der maximal mit 170 MHz takten kann [42]. Es sind 128 KiB an Flash-Speicher und 32 KiB an RAM verfügbar [42].

4.3 Erstellung einer ausführbaren Software für Mikrocontroller

Es wird eine in Rust geschriebene Software erstellt, die auf den Mikrocontrollern ausführbar ist. Dies soll ermöglichen, die Portierung der UAVCAN Bibliothek zu testen und im Bezug auf eingebettete Systeme weiterzuentwickeln. Außerdem soll der Grundaufbau des Programms für die späteren Messungen verwendet werden.

Zur Erstellung werden Bibliotheken verwendet, die von der Community entwickelt werden. Diese sind mit entsprechender Verwendung in Tabelle 4.1 aufgeführt. Außerdem ist die Plattform, auf der die Bibliothek verfügbar ist und deren verwendete Version mit aufgeführt. Mit dieser Version kann nachvollzogen werden, welche Funktionalität bei der Entwicklung verfügbar ist und verhindert ungewollte Versionssprünge.

Die aufgeführten Bibliotheken ermöglichen eine Abstrahierung über die verwendete Hardware. Dabei kann die Konfiguration des Mikrocontrollers mit dem HAL anhand von Funktionen und Interfaces vorgenommen werden. Somit müssen nicht einzelne Registerbits verändert werden, da dies zu möglichen Fehlern führen kann.

Durch die Logging Bibliotheken in Tabelle 4.1 kann die Ausführung der Software auf dem Mikrocontroller überwacht werden, wenn der Controller über ein Debugger an einem Host-System angeschlossen ist. Logs können in Form von Strings auf dem Host-System angezeigt werden.

4.3.1 Probleme bei der Entwicklung

Die *NUCLEO-G431RB* Boards implementieren ein CAN Interface, das zu Beginn der Arbeit durch keine Rust Bibliothek unterstützt wird. Daher muss eine Lösung gefunden werden, um die CAN Peripherie zu konfigurieren und Nachrichten über den Bus zu senden.

Hierfür wird zuerst die Konfiguration durch das Ändern einzelner Registerbits betrachtet. Nach einigen Anläufen stellt sich dies als ein fehleranfälliges Vorgehen heraus. Daher wird entschieden, den vom Hersteller in C entwickelten HAL zu verwenden.

Mit diesem HAL wird die Konfiguration des Boards und der Peripherien durchgeführt.

Bibliothek	Verwendung	Anwendungsfall	Verfügbarkeit	Version
cortex-m	abstrahiert Funktionalität von Cortex-M Prozessoren	Verwendbar auf Cortex-M Prozessoren	crates.io	0.6.0
cortex-m-rt	implementiert benötigte Rust Funktionen, zum Beispiel der Einstiegspunkt des Programms, die bei einer *no_std* Umgebung manuell eingefügt werden müssen	Verwendbar auf Cortex-M Prozessoren	crates.io	0.6.10
stm32g4xx-hal	HAL für die STM32G4 Mikrocontroller Serie	STM32G4 Mikrocontroller Serie	crates.io	0.0.1[1]
defmt	zum Loggen von Zuständen und Werten	eingebettete Systeme	crates.io	0.2.0
defmt-rtt	implementiert einen globalen Logger in die Software, der defmt Logs über RTT an einen Host sendet	Mikrocontroller mit RTT Logging	crates.io	0.2.0

Tabelle (4.1): Verwendete Bibliotheken für die Entwicklung der ausführbaren Software.

Damit die Funktionen des HALs in Rust aufgerufen werden können, wird die Fremdfunktionsschnittstelle verwendet, erläutert im Kapitel *Fremdfunktionsschnittstelle von Rust*.

Die Fremdfunktionsschnittstelle wird dabei nicht manuell geschrieben, sondern mit der Bibliothek *bindgen* automatisch generiert. Dabei müssen die C Header Dateien angegeben werden. Entsprechend wird eine Rust Bibliothek mit den verfügbaren Funktionsköpfen und C Strukturen generiert. Diese Bibliothek kann schlussendlich als Abhängigkeit in der zu erstellenden Software eingebunden werden. Zur Kompilierung wird der C HAL als statisch gelinkte Bibliothek angegeben.

In Rust können nun die in C geschriebenen Funktionen aufgerufen werden. Der Aufruf muss jedoch, wie in Kapitel *Fremdfunktionsschnittstelle von Rust* beschrieben, in einem *unsafe* Block gemacht werden. Der *unsafe* Block sollte jedoch nur gezielt und an Stellen, bei welchen direkt die Hardware angesprochen wird, in einem Rust Programm verwendet werden. Aus diesem Grund ist der nächste denkbare Schritt eine API um die unsicheren Funktionsaufrufe herum zu entwerfen, die unsichere Zustände der HAL API verhindert.

In der Arbeit wurde dies nicht weiter verfolgt, da zu diesem Zeitpunkt der Rust HAL eine Unterstützung für das CAN Interface auf dem *NUCLEO-G431RB* Board erhielt.

4.4 Entwicklung einer Benchmark-Suite für ein eingebettetes System

Die Entwicklung der UAVCAN Rust Bibliothek ist nach dieser Arbeit nicht abgeschlossen. Deshalb wird eine Benchmark-Suite entwickelt, mithilfe derer automatisch Softwarezeitmessungen auf einem eingebetteten System ausgeführt werden können. Somit kann die Entwicklung weiterhin evaluiert und das Erfassen der Zeiten in den Entwicklungsprozess eingebaut werden.

In Rust gibt es die Möglichkeit, in Umgebungen, bei denen die Standardbibliothek verwendet werden kann, Zeitmessungen von Code direkt in Tests einzubinden. Recherchen ergaben, dass es für eingebettete Systeme noch keinen Standard gibt. Somit muss eine eigene Implementation entwickelt werden.

Die Benchmark-Suite soll auf verschiedenen Hardwarearchitekturen, mit oder ohne Betriebssystem, ausgeführt werden können. Des Weiteren soll es möglich sein, weitere API Zeitmessungen hinzuzufügen und die verschiedenen Messungen automatisiert an einem Stück durchzuführen. Die gemessenen Zeiten sollen sich den Zeiten annähern, die einen realen Einsatz der Bibliothek abdecken.

Die Benchmark-Suite soll aus drei Komponenten bestehen, die im Folgenden aufgelistet sind:

1. Testbench
2. Suite
3. Hardware spezifischer Teil

Die Testbench, im Deutschen für Prüfstand, bietet die Möglichkeit, Ausführzeiten der Funktionsaufrufe zu messen und auszuwerten.

Die Suite soll generisch die zu messenden Szenarien darstellen. Generisch bedeutet hierbei, dass die zu messenden Funktionen unabhängig von Hardware spezifischem Code, das heißt verschiedenster Speicherverwaltung oder verschiedenster Zeitimplementationen, ausgeführt werden können. Hauptsächlich wird dabei in *no_std* und *std* Kontext unterschieden.

Schließlich deckt die dritte Komponente den hardwarespezifischen Teil ab. Nachdem die ersten zwei Komponenten Bibliotheken sind, ist diese Dritte nach der Kompilierung eine ausführbare Datei, die auf das entsprechende Testsystem geladen wird.

4.4.1 Testbench

Für die Testbench wird der Ansatz aus der Bibliothek *Liar*[28] verwendet. Diese ist öffentlich auf GitHub zugänglich und hat das Ziel, Benchmarks für *no_std* Umgebungen zu ermöglichen. Aufgrund dessen, dass der Code seit mehreren Jahren nicht weiterentwickelt wird und es für die, bei dieser Arbeit gestellten Anforderungen, an Funktionalität fehlt, wurde entschieden, Teile der Implementation zu übernehmen und zu erweitern.

In Abbildung 4.2 ist der Klassenaufbau der Testbench dargestellt. Der *Bencher* ist der Haupteinstiegspunkt. Mit der Funktion *bench()* kann eine neue Zeitmessung erfasst werden. Hierbei wird eine Referenz an eine Funktion übergeben, in der die Messung durchgeführt und als Messeinheit bezeichnet wird. Durch Angabe eines Namens kann nach der Ausführung das Ergebnis entsprechend identifiziert werden. In der Messeinheit wird mit *run()* oder *run_with_watch()*, die als Erweiterung hinzugefügt wird, eine Messung ausgeführt. Der Unterschied bei den zwei Funktionen besteht darin, dass *run_with_watch()* eine Stoppuhr bereitstellt, die es erlaubt, an bestimmten Stellen im Programm die Zeit zu starten, zu pausieren und zu stoppen. Bei *run()* wird die gesamte Aufrufzeit der Funktion gemessen.

Der *Bencher* verwendet intern einen *Runner*, der die Messungen durchführt. Ebenso wie der *Bencher*, besitzt dieser gleichnamige Funktionen, die jeweils zur Ausführung aufgerufen werden. Bei deren Aufruf führen diese die Messeinheit in einer Schleife aus. Die Anzahl der Iterationen kann durch ein *Const-Generic*[1] Argument angegeben werden. Die gemessenen Zeiten werden im Parameter *data* des *Benchers* zwischengespeichert.

Sind die Messungen über ein Szenario komplett, wird das Ergebnis in Form von *Samples* vom *Bencher* zurückgegeben. Durch das Aufrufen der Funktion *get_summary()* in *Samples*, die zu der *Liar* Implementation hinzugefügt wird, wird das Ergebnis statistisch ausgewertet. Dazu gehören der Minimal- und Maximalwert, sowie der arithmetische Mittelwert und der Median. Auf diese Werte kann durch öffentlich markierte Felder zugegriffen werden.

Die Stoppuhr wird ebenso zur Bibliothek hinzugefügt. Die Struktur in Abbildung 4.2 *Watch* hat die öffentlichen Funktionen *start()* und *stop()*, die in der Testeinheit aufgerufen werden, um die Messung an einem Punkt zu starten und an einem anderen zu stoppen. Soll die Messung pausieren, wird die *stop()* Funktion und beim erneuten Starten die Funktion *start()* aufgerufen. Für die Testbench Bibliothek intern kann die Stoppuhr mit *reset()* zurückgesetzt und mit *get_elapsed()* die gestoppte Zeit abgefragt werden.

Die Verwendung der Testbench Bibliothek ist in Listing 4.2 dargestellt. Hierbei wird in Zeile 3 der Bencher mit einer Uhr für die Zeitmessungen initialisiert. In Zeile 6 wird eine Messung mit der *bench_test* Einheit aus Listing 4.3 durchgeführt. Das Ergebnis der Messung wird in der Variable *result* gespeichert.

[1] Ein Const-Generic Argument erlaubt es einem Typ, neben Typ- und Lebenszeit-Argumenten, auch einen konstanten Wert zu übergeben. Ein Beispiel dafür ist die Angabe der Länge eines Arrays. [30]

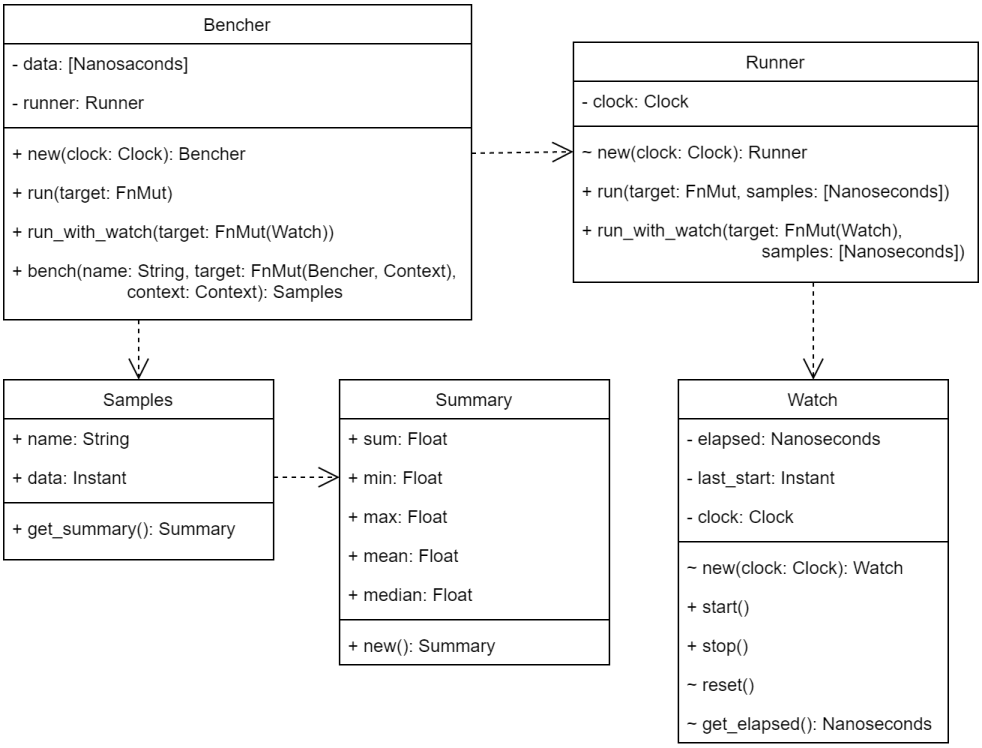

Abbildung (4.2): Klassendiagramm der Testbench-Implementation.

```
fn main() {
    let clock = StmClock::new();
    let bencher = Bencher::new(clock);
    let context = UavcanContext::new();

    let result = bencher.bench("a test", &mut bench_test, &mut
        context);
}
```

Listing (4.2): Testbench Bibliothek Anwendungsbeispiel.

4.4.2 Suite

In der *Suite* werden die zu messenden Szenarien definiert. Ein Szenario ist zum Beispiel das Senden einer Übertragung unter bestimmten Bedingungen und mit einer bestimmten Datenlänge. Der Aufbau eines solchen ist in Listing 4.3 gezeigt.

Zur Definition wird der Funktion *run_with_watch()* in Listing 4.3 in Zeile 5 eine anonyme Funktion[1] übergeben, die vom *Runner* in einer Schleife aufgerufen wird. In der anonymen

[1] Eine anonyme Funktion wird in Rust auch als *Closure* bezeichnet. Die Funktion kann einer Variablen zugewiesen werden und somit an verschiedenen Punkten aufgerufen werden. Außerdem kann diese

Funktion werden eine Messung und zusätzliche Operationen, die für jeden Messdurchgang erforderlich sind, durchgeführt. In diesem Fall in Zeile 5 wird die Laufvariable i nach jeder Messung inkrementiert.

Die Messeinheit besteht aus den Eingangs- und Ausgangs-Variablen und aus der zu messenden Funktion. Die Eingangsvariablen müssen an eine *black_box()* Funktion übergeben werden. Diese gibt dieselbe Variable unverändert wieder zurück. Dies ist in Listing 4.3 Zeile 7 zu sehen. Das Ergebnis der zu messenden Funktion wird ebenfalls einer *black_box()* übergeben.

Diese Vorgehensweise wurde anhand von Tests ermittelt. Testversuche ergaben, dass der Rust-Compiler nicht verwendete Variablen und Funktionsaufrufe wegoptimiert. Aus diesem Grund muss das Ergebnis mit einem *black_box()* Funktionsaufruf verwendet werden. Des Weiteren müssen die Eingangsvariablen ebenfalls mit einem *black_box()* Funktionsaufruf übergeben werden. Denn sind diese über die Iterationen konstant, führt der Rust-Compiler weitere ungewollte Optimierungen durch. Dies bedeutet zum Beispiel bei konstanten Variablen eine einmalige Auswertung einer Funktion und das Verwenden desselben Ergebnisses bei weiteren Iterationen.

Die Funktion *black_box()* wird in Rust auch als Einheitsfunktion bezeichnet [29]. Der Compiler behandelt diese Funktion bei der Kompilierung maximal pessimistisch [29]. Sprich, bei der Kompilierung soll dieser davon ausgehen, dass der übergebene Parameter in einem anderen Kontext verwendet wird und demnach nicht wegoptimiert werden darf[1].

Durch den *context* parameter in dem Funktionskopf in Listing 4.3 kann auf Kontext zugegriffen werden, der Hardware abhängig initialisiert werden muss. Diese Initialisierung kann nicht in der Funktion *bench_test()* gemacht werden, da die Messszenarien generisch über verschiedene Hardware implementiert werden.

```
1  fn bench_test(bencher: &mut Bencher, context: &mut Context) {
2      // Initialisieren von Variablen für die Messung
3      let mut i = 0;
4      // Die Messung mit dem Bencher durchführen
5      bencher.run_with_watch(|watch| {
6          // Eingabe definieren
7          let input = black_box(i);
8
9          // Messung starten
10         watch.start();
11         let output = test_func(input);
12         // Messung stoppen
```

Variablen aus der Umgebung der Funktion erfassen. [36]

[1] Genauer steht in der Dokumentation, "[...]a Rust compiler is encouraged to assume that black_box can use dummy in any possible valid way that Rust code is allowed to without introducing undefined behavior in the calling code."[29]. Des Weiteren wird darauf hingewiesen, dass die Funktion auf einem *best-effort* Prinzip basiert und demnach unterschiedlich die Optimierung auf verschiedenen Plattformen stoppt [29].

```
13          watch.stop();
14
15          // Ergebnis verwerten
16          black_box(output);
17          i = i.wrapping_add(1);
18      })
19  }
```

Listing (4.3): Aufbau eines Tests in der Suite.

In der *Suite* werden außerdem die auszuführenden Messszenarien in einer *run()* Funktion definiert. Anschließend werden die Messungen in Hardware spezifischer Software durch einen Aufruf dieser ausgeführt.

4.4.3 Hardware spezifischer Teil

Im Gegensatz zu den vorherigen zwei Komponenten, wird diese zu einer ausführbaren Datei kompiliert, die auf den Mikrocontroller geflasht und ausgeführt wird. In diesem Programm werden hauptsächlich der Mikrocontroller konfiguriert, sowie Hardware abhängige Strukturen für die Suite und die Testbench erstellt. Speziell für eingebettete Systeme, wird außerdem ein globaler Heap für die Speicherverwaltung initialisiert.

Wenn alle benötigten Komponenten erstellt sind, wird die Funktion *run()* aus der Suite aufgerufen, die dann die definierten Benchmarks ausführt.

Da das Ziel der Messungen ist, vergleichbar zu den Einzelmessungen zu sein, muss der Compiler die Messeinheiten bestmöglich optimieren. Tests ergaben, dass bei einer Optimierung des gesamten Systems auf Geschwindigkeit, also aller Bibliotheken und des Hauptprogramms, die Bench-Ergebnisse variieren. Dies basiert auf den Optimierungen, die der Compiler trotz Verwendung der *black_box()* Funktion vornimmt. Weitere Tests zeigten, dass, wenn die erstellte Bibliothek-Testbench auf eine kleinere Binärdatei Größe optimiert wird[1], die Messergebnisse konstanter messbar sind.

Das kompilierte Programm wird mit dem Werkzeug *probe-run* auf den Mikrocontroller gespielt und ausgeführt. Außerdem werden die Testergebnisse über das *Real-Time Transfer* (RTT) Protokoll vom Board an das Host-System geschickt und auf dem Terminal angezeigt.

[1] Dies kann erzielt werden, wenn in der Cargo.toml Datei der Eintrag [profile.<profile>.package.<crate-name>] hinzugefügt und darunter das Optimierungslevel speziell für diese Bibliothek mit *opt-level* angegeben wird. [35]

5 Evaluierung der UAVCAN-Implementation in Rust

Die Rust-Implementation soll nach Kriterien der Softwarequalität beurteilt werden. Softwarequalität wird durch Merkmale, die die Software erfüllen muss, definiert. Im ISO-25010 Softwareproduktqualitätsbaum ist die Leistungsfähigkeit eines dieser Merkmale und wird als Ausführungsmerkmal kategorisiert [44].

Auch McCall listet Qualitätsfaktoren für Software auf. Wie in [44] ist für ihn die Effizienz ebenfalls ein Faktor [5] und ist mit der Leistungsfähigkeit gleichgesetzt. Die Effizienz misst den Verarbeitungsaufwand, welcher zur Ausführung einer Funktion nötig ist [5].

In [22] wird die Effizienz so interpretiert, dass sie den Umfang darstellt, in welchem eine Leistung mit minimalem Ressourcenverbrauch erbracht wird. Als Kriterien für die Effizienz werden Ausführungs- und Speichereffizienz genannt [22].

Die UAVCAN-Implementation in Rust soll basierend auf derer Leistungsfähigkeit evaluiert werden. Da von der Spezifikation keine einzuhaltenden Leistungswerte vorliegen, wird als Richtwert für die Evaluierung eine in C++ implementierte UAVCAN Bibliothek hergenommen. Dabei muss berücksichtigt werden, dass zum einen die jeweilige Umsetzung des Standards, zum anderen die verwendete Programmiersprache Einfluss auf die Messungen haben.

Da das Ziel der Arbeit ist, das Potential von Rust für die UAVCAN Spezifikation aufzuzeigen, konzentrieren sich die Messungen auf verschiedene Implementationsweisen und auf sprachliche Aspekte. Es wird herausgearbeitet, welchen Einfluss die Sprache auf die Leistung der Implementationsumsetzung hat.

Im Folgenden wird zuerst auf Messmethoden eingegangen, welche anschließend für das Messen von Ausführungszeit und Speicherverbrauch verwendet werden. Zu beachten ist, dass die Messreihen nur Momentaufnahmen des aktuellen Projektstands sind, da die Rust-Implementation im Laufe dieser Arbeit und darüber hinaus weiterentwickelt wird.

5.1 Messmethodik

In [16] werden Metriken und Techniken zur Messung von Software-Leistung aufgelistet und beschrieben, mit welchen Leistungsfähigkeit von Software gemessen wird. Im Folgenden sind diese zusammengefasst aufgeführt.

5.1.1 Verfügbare Metriken

Das **Performance-Profiling** misst die Zeit in einem System auf einer Funktion-für-Funktion Basis [16]. Die Zeiten einzelner Funktionsaufrufe werden aufgezeichnet. So kann der Entwickler problematische Bereiche identifizieren, in denen laut der Analyse viel Zeit bei der Ausführung benötigt wird, und diese Bereiche optimieren [16].

Bei dem **A-B Timing** werden in der Software zwei Punkte bestimmt, welche Start- und Endpunkt einer Messung definieren [16]. Dieser Bereich kann mehrere Funktionsaufrufe umfassen. Durch diese Art der Messung wird überprüft, ob geforderte Ausführungszeiten eingehalten werden [16].

Bei einer weiteren Methode wird die **Antwort auf externe Events gemessen**. Ein Beispiel hierfür ist eine CAN-Nachricht, welche ein externes Event darstellt, auf welche die Software reagieren soll. Die Zeit zwischen Event (Eingehen der Nachricht) und der Reaktion des Systems wird gemessen. Dies ist laut [16] eine wichtige Metrik für Echtzeitsysteme, bei welcher die Hardware und Software für geforderte Ausführungszeiten evaluiert werden sollen.

5.1.2 Verfügbare Techniken und Tooling

Im Buch [3] wird bei Laufzeitanalysatoren zwischen statischen und dynamischen Messverfahren unterschieden. Bei einer statischen Messung wird das Programm zur Messung nicht ausgeführt. Der Analysator soll lediglich Engpässe an der Codebasis finden. Zugleich wird darauf hingewiesen, dass das statische Analysieren aufgrund komplexer Systeme sehr schwer ist [3]. Deshalb werden in der Praxis dynamische Verfahren verwendet [3]. Dabei wird das Programm auf der entsprechenden Hardware ausgeführt. [16] führt dazu folgende Techniken auf:

Logik Analyser Sind in Form von externer Hardware verfügbar und sollen zum Debuggen von digitalen Schaltungen verwendet werden. Damit kann verifiziert werden, dass auf dem Bus ein korrekter Datenaustausch stattfindet. Durch die Verwendung von zusätzlicher Hardware können Antwortzeiten hochpräzise aufgenommen werden. [16]

Software-Only Leistungsfähigkeits-Überwachung Dazu zählt das Stack-Sampling. Hierbei wird periodisch ein Abbild des Stacks gemacht, damit können Schlüsse auf die Programmausführung und auf den Speicherverbrauch gezogen werden. Eine weitere Möglichkeit ist die Source-Code Instrumentierung. Bei dieser Technik werden Funktionsaufrufe in den Source-Code eingefügt. [16]

Hardware-Assisted Leistungsfähigkeits-Überwachung Hierbei wird zusätzliche Hardware zur präziseren Messung eingesetzt, sodass der Source-Code nicht modifiziert werden muss [16].

5.1.3 Einheitliche Messumgebungen schaffen

Damit die Messwerte der Zeit- und Speichermessungen vergleichbar sind, ist es wichtig, Messungen nahezu unter denselben Bedingungen durchzuführen.

Aufgrund dessen werden die Messungen auf denselben Mikrocontrollern ausgeführt. Als Transport wird das Extended-CAN-Bus Protokoll verwendet. Wie in Kapitel 2.3.3 erläutert, beschränkt dies die maximale Datenmenge auf 8 Byte pro Transportrahmen.

Die Compiler sollen den Quellcode mit hoher Geschwindigkeits-Optimierung kompilieren. In C++ wird der *GNU-C-Compiler* (GCC), in der Version *9.2.1 [arm-none-eabi-9]*, verwendet. Dabei wird der Marker *-O3* gesetzt. Für den *rustc* Compiler, in der Version *1.57.0-nightly (2021-09-29)*, werden die Optionen *opt-level=3*, *lto=true* und *codegen-units=1* angegeben.

Dadurch, dass der Rust-Implementation während der Arbeit Funktionen fehlen, die die Arduino Bibliothek unterstützt, wird eine modifizierte Schnittstelle der Arduino-Bibliothek verwendet. Hierbei wird auf das Verwenden von Datentypen der UAVCAN Spezifikation verzichtet.

5.2 Zeitmessungen von Bibliotheksfunktionen

Bei diesen Zeitmessungen wird die Technik *Software-Only* Leistungsfähigkeits-Überwachung verwendet, siehe Kapitel 5.1.2. Dabei wird die Source-Code Instrumentierung angewandt. Hierbei soll ausschließlich die Ausführungszeit für Funktionsaufrufe der Bibliothek analysiert werden.

Für die jeweiligen Messungen wird kurz erläutert, welche Funktionsaufrufe gemessen werden, welche Szenarien die Messung beeinflussen und welche Erkenntnisse aus diesen gezogen werden. Manche Messergebnisse führen dazu, dass Optimierungen in der Implementation durchgeführt werden.

5.2.1 Grundüberlegungen zu den Messszenarien

Die Messergebnisse der Zeitmessungen variieren basierend auf dem CPU-Takt der Mikrocontroller. Aus diesem Grund werden die Mikrocontroller für die jeweiligen Messungen auf einen Takt von 170 MHz konfiguriert.

Für die Messungen in den jeweiligen Sprachen werden dieselben Nachrichten verwendet. Das heißt, die gesendeten und zu empfangenden Daten sollen gleich sein. Dabei ist besonders wichtig, dass diese Nachrichten dieselbe Länge haben. Es werden mit den Implementationen jeweils verschiedene Messreihen mit unterschiedlichen Datenlängen durchgeführt. Dies ist wichtig, da eine Nachricht, die zu lang für einen Transportrahmen ist, aufgeteilt wird, wie in Kapitel 3.2.4 beschrieben. Die Überlegung hierbei ist, dass die Implementationen, abhängig von den Datenlängen, unterschiedlich viel Zeit für die Verarbeitung benötigen. Somit kann zusätzlich die Skalierbarkeit bei größeren Übertragungen evaluiert werden.

Für eine Messreihe wird die Länge der Übertragung so variiert, dass die resultierende Datenlänge in Schritten von 8 Byte inkrementiert wird, das der Datenlänge eines Transportrahmens von CAN entspricht. Hierbei muss beachtet werden, dass beim UAVCAN Protokoll zusätzliche Bytes hinzugefügt werden, siehe Kapitel 3.2.4. Damit die Datenlänge p einer Übertragung in volle Übertragungsrahmen mit der Länge m passt, kann diese mit Gleichung 5.1 berechnet werden. f steht für die Anzahl der Rahmen.

$$p(f,m) = \begin{cases} m-1 & : f = 1 \\ (f-1)*(m-1)+(m-3) & : f > 1 \end{cases} \quad f \in \{1,2,3,4,...\}, m \in \{\geq 8\} \tag{5.1}$$

Für einen Transport mit *Classic*-CAN resultiert die Gleichung 5.2.

$$p(f) = \begin{cases} 7 & : f = 1 \\ (f-1)*7+5 & : f > 1 \end{cases} \quad f \in \{1,2,3,4,...\} \tag{5.2}$$

Für eine Messung werden mehrere Messergebnisse aufgezeichnet, über die der Mittelwert für das Endergebnis berechnet wird. Dies soll mögliche Messschwankungen, die durch Hardware auftreten können, verhindern. Jedoch sollten die Messergebnisse relativ konstant messbar sein, da kein Betriebssystem, das mehrere Programme ausführen muss, verwendet wird. Denkbar mögliche Ursachen für schwankende Messergebnisse sind Interrupts oder auf Implementationslogik basierende Ereignisse der UAVCAN Bibliotheken.

Eine Messung über ein Szenario wird in Intervallen durchgeführt. Dabei wird beim Senden und beim Empfangen mit einem Abstand von einer Sekunde eine Übertragung gesendet, beziehungsweise eine vollständige Übertragung empfangen.

Gemessen wird nun die Zeit, welche die jeweilige Implementation für das Senden beziehungsweise das Empfangen von einem bis f Rahmen in Einerschritten benötigt.

5.2.2 Benötigte Softwarekomponenten für die Messungen

Für die Messung von Ausführungszeiten auf dem Mikrocontroller wird eine monotone Uhr benötigt. Diese soll hochpräzise sein. Im besten Fall soll diese eine Auflösung von einem CPU-Takt des Controllers haben. Dies macht es möglich, Funktionen zu messen, die wenig CPU Zyklen benötigen.

Die Uhr soll durch Aufrufen einer Funktion die aktuelle Systemzeit bezogen auf den Programmstart liefern. Zudem soll das System ermöglichen, die verstrichene Zeit zwischen zwei oder mehreren Punkten im Programm zu messen. Aufgrund dessen, dass Uhren auf einem Mikrocontroller meistens auf einem Zählregister basieren, darf ein Zählerüberlauf das Messergebnis nicht beeinflussen.

Umsetzung in Rust

Das in Kapitel 4.3 erwähnte HAL crate verfügt über eine Stoppuhr, die die aufgeführten Anforderungen besitzt und als *MonoTimer* bezeichnet wird.

Der Timer basiert auf der *Data Watchpoint Trigger* (DWT)[1] Einheit. Diese integriert einen Zähler, der mit einem zuvor konfigurierten Takt durch die Hardware inkrementiert wird und dessen Register 32 bit[1] weit ist. Standardmäßig wird der Taktgeber der DWT Einheit durch den HAL auf die gleiche Frequenz wie der CPU-Takt gesetzt. Durch Aufrufen von *DWT::get_cycle_count()* wird der aktuelle Zählerstand zurückgegeben. Wenn der Zähler überläuft, also den maximalen Wert von 2^{32} erreicht, startet dieser wieder von 0 [1].

Entspricht der Takt des Zählers dem Takt der CPU, können Zeitspannen von bis zu 25 sec gemessen werden, bevor dieser überläuft. Dies kann mit Gleichung 5.3 berechnet werden.

$$\frac{1}{170\ \text{MHz}} * 2^{32} = \ 25\ \text{Sekunden} \tag{5.3}$$

Eine Messung wird mit t_1 als Startpunkt und mit t_2 als Ende der Messung definiert. Die verstrichene Zeitspanne wird mit Gleichung 5.4 berechnet.

$$\Delta t = t_2 - t_1 \tag{5.4}$$

Damit Zeitspannen gemessen werden können, die vor einem Überlauf starten und danach enden, werden die Zeitpunkte umlaufend subtrahiert. Dies wird in Listing 5.1 in Rust Code gezeigt. Die Zeitpunkte werden dabei von dem vorzeichenlosen Datentyp in den jeweiligen vorzeichenbehafteten Datentyp konvertiert. Nach der Subtraktion wird das Ergebnis erneut als vorzeichenlos interpretiert. Dieses Verhalten kann in Rust mit der Funktion *wrapping_sub()* erzielt werden.

```
let t1 = 250u8;
let t2 = 10u8;
let verstrichen = (t2 as i8 - t1 as i8) as u8;
assert_eq!(verstrichen, t2.wrapping_sub(t1));
```

Listing (5.1): Umlaufendes subtrahieren mit Typkonvertierung und mit der wrapping_sub() Funktion.

In Listing 5.2 wird gezeigt, wie eine Messung mit *MonoTimer* durchgeführt werden kann. Dabei wird zuerst eine Stoppuhr erstellt. Diese bekommt als Argumente die DWT Peripherie und eine Referenz zu den System-Uhren. Diese werden für die Umrechnung der Zählerschritte (Ticks) in eine richtige Zeiteinheit benötigt. Danach wird der Startpunkt festgelegt. Nach dem Messbereich werden die benötigten Ticks abgefragt. Intern verwendet die Methode *elapsed()* die Funktion *wrapping_sub()*. Anschließend werden diese in Mikrosekunden umgerechnet.

```
    let uhr = MonoTimer::new(cp.DWT, cp.DCB, &rcc.clocks);
    let start = uhr.now();
    {
        // Messbereich
    }
    let verstrichene_ticks = start.elapsed();
    let mikros= uhr.frequency().duration(verstrichene_ticks).0;
```

Listing (5.2): Einsatz des MonoTimers in Rust.

Aufgrund der geforderten Möglichkeit zur Unterbrechung von Messungen und da der *MonoTimer* dies nicht unterstützt, wird eine Stoppuhr basierend auf der zuvor beschriebenen Funktionalität implementiert. Diese erlaubt es, mit den Funktionen *start()* und *stop()* die Messung zu starten, zu pausieren und zu stoppen.

Umsetzung in C++

Um in der C++-Implementation die Ausführungszeiten zu messen, wird ebenso eine Stoppuhr implementiert.

Eine erste Überlegung dabei ist, die Funktion *micros()* des Arduino-Framework für die aktuelle Systemzeit zu verwenden. Die Funktion gibt die Zeit in einer Auflösung von Mikrosekunden wieder. Nach einer Analyse der Arduino-Implementation stellt sich jedoch heraus, dass die Systemzeit mit einem Zähler, welcher durch das Interrupt *SysTick* inkrementiert wird, im HAL umgesetzt ist. Werden nun Interrupts durch die Software vorübergehend ausgestellt, kann der Zähler abweichen.

Um genaue Messungen durchführen zu können, muss der Zähler der *Data Watchpoint Trigger* (DWT) Einheit, genauso wie in Rust, verwendet werden. Dies bietet die Möglichkeit in Nanosekunden zu messen. Außerdem ist eine solche Implementation unabhängig von Interrupts. Dadurch sind auch die Messergebnisse besser vergleichbar.

5.2.3 Ablauf der Zeitmessungen mit Messpunkten

In diesem Abschnitt wird der Ablauf von Funktionsaufrufen zwischen definierten Messpunkten betrachtet. In Kapitel 3.3 sind die verwendeten Funktionsnamen der APIs der jeweiligen Sprache anhand von Code erklärt.

Die Messabläufe sind für das Verständnis und für das Nachvollziehen der Messergebnisse im darauffolgenden Kapitel wichtig. Die Messpunkte werden so gelegt, dass entweder ein Sende- oder ein Empfangsvorgang gemessen wird.

Arduino/C++

Der Ablauf des Sendevorgangs bei der Arduino-Implementation ist in Abbildung 5.1 dargestellt.

Für das Senden wird zuerst ein Übertrag erstellt. Dabei werden die zu sendenden Daten mit der Länge nach Gleichung 5.1 generiert. Dieser Vorgang wird nicht mitgemessen, da dies nicht Teil der Bibliothek ist und bei der Arduino- und der Rust-Implementation jeweils derselbe Vorgang ist.

Nachdem die Übertragung erstellt ist, wird nun die Messung gestartet, siehe Abbildung 5.1 grüner Block.

Im Messbereich wird die Übertragung mit der Funktion *enqueueTransfer* der UAVCAN Bibliothek übergeben. Daraufhin wird die Funktion *transmitCanFrame* und folglich die Rückruffunktion *transmit_func* aus der Bibliothek für die Anzahl f an zu sendenden Rahmen aufgerufen. Die Rückruffunktion ist mit leerem Funktionskörper definiert, sodass ausschließlich die Bibliotheksfunktionen gemessen werden. Die Zeitmessung wird gestoppt,

wenn sich keine weiteren Rahmen in der Warteschlange befinden, siehe Abbildung 5.1 roter Block.

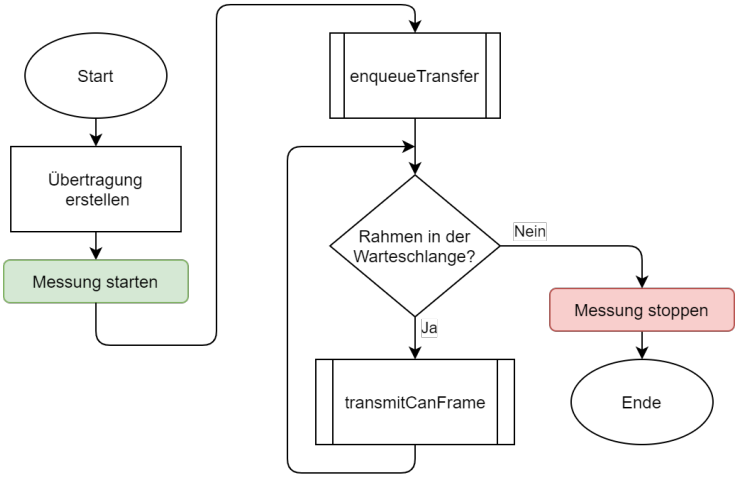

Abbildung (5.1): Ablaufdiagramm der Messungen des Sendens der UAVCAN-Implementation in C++.

Der Ablauf zum Messen des Empfangsvorgangs ist in Abbildung 5.2 dargestellt.

Hierbei wird außerhalb des Messbereichs ein Array mit den zu empfangenden Rahmen erstellt. Der Dateninhalt wird mit passendem Ende-Byte und CRC-Bytes, siehe Kapitel 3.2.4, erzeugt.

Der Messbereich beginnt mit dem Empfangen des ersten Rahmens, siehe Abbildung 5.2 grüner Block. Dieser wird mit der Funktion *onCanFrameReceived* an die UAVCAN Bibliothek übergeben.

Müssen weitere Rahmen empfangen werden, wird die Messung zunächst pausiert, siehe Abbildung 5.2 gelber Block, und vor dem nächsten Empfangen wieder gestartet. Die Messung wird pausiert, da bei der Messimplementation der $n-te$ Rahmen dynamisch generiert wird. Die Funktion *onCanFrameReceived* wird für die Anzahl f an zu empfangenden Rahmen erneut aufgerufen.

Ist die Übertragung komplett empfangen, wird die definierte Rückruffunktion *on_data_receive* aus der Bibliothek aufgerufen, in der die Messung dann gestoppt wird. Dieser Vorgang ist in Abbildung 5.2 im gestrichelten Kasten dargestellt.

Nach dem Stoppen der Messung kehrt die *onCanFrameReceived* trotz vollständigem Empfang zurück und der Ablauf wird beendet, da keine weiteren Rahmen vorhanden sind.

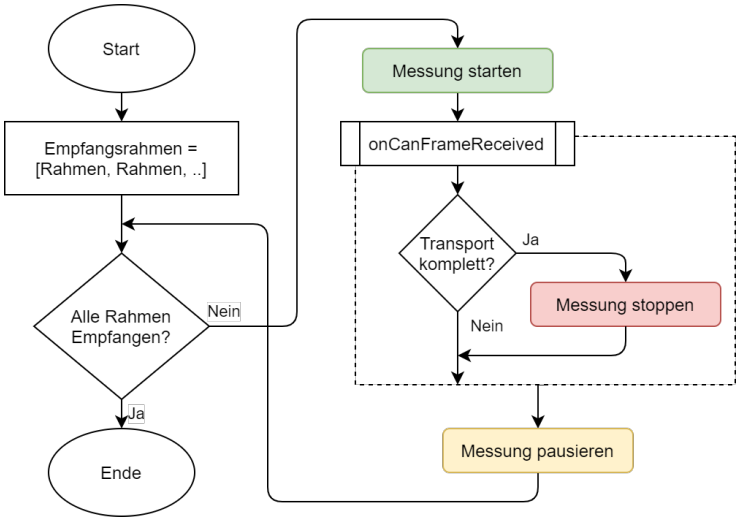

Abbildung (5.2): Ablaufdiagramm der Messungen des Empfanges der UAVCAN-Implementation in C++.

Rust

Der Ablauf der Messungen des Sendevorgangs in Rust ist in Abbildung 5.3 dargestellt. Entsprechend wie bei Arduino, wird zuerst die zu sendende Übertragung generiert. Diese enthält ebenso eine, mit Gleichung 5.1 berechnete, Datenlänge.

Daraufhin wird die Aufrufzeit der Funktion *transmit* gemessen. Diese liefert einen Iterator, das heißt eine Struktur, welche die Übertragung mit dem Aufrufen von *next* in Rahmen aufteilt. Diese Messung ist in Abbildung 5.3 mit der Abfolge „Messung starten“, Funktionsaufruf, „Messung stoppen“ dargestellt.

Folgend werden in einer Schleife mit der Funktion *next* die zu sendenden Rahmen abgefragt, bis der Iterator keine weiteren Elemente zurückgibt. Die Messung wird für jedes einzelne abzufragende Element gestartet und danach pausiert, sodass die Messungen genau die Bibliotheksaufrufe abdecken.

Ist die Übertragung vollständig verarbeitet, wird die Messung schlussendlich gestoppt, veranschaulicht in Abbildung 5.3 roter Block.

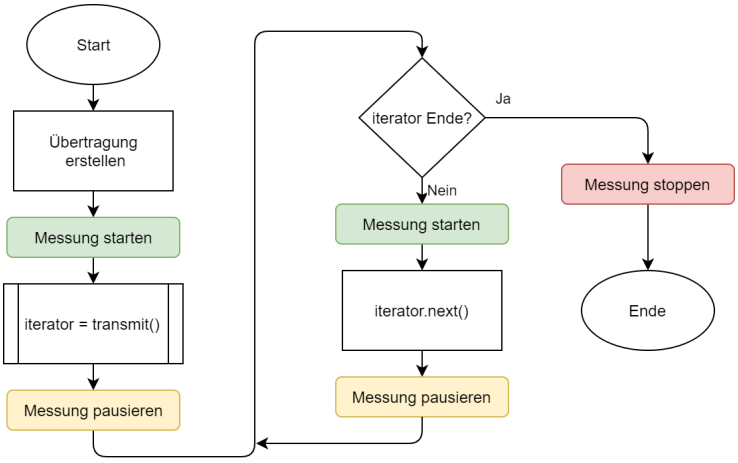

Abbildung (5.3): Ablaufdiagramm der Messungen des Sendens der UAVCAN-Implementation in Rust.

Beim Messen des Empfangsvorgangs in Rust wird ebenfalls wie bei Arduino ein Array mit Rahmen generiert. Dies ist der erste Schritt in Abbildung 5.4.

Daraufhin wird die Messung für das Empfangen des ersten Rahmens gestartet. Wenn die Funktion *try_receive_frame* zurückkehrt, wird die Messung pausiert. Dieser Ablauf wird in einer Schleife fortgeführt, bis die Übertragung komplett empfangen ist und die Funktion *try_receive_frame* diese zurückgibt.

Die Messung wird gestoppt, wenn alle Rahmen verarbeitet sind, siehe Abbildung 5.4 roter Block.

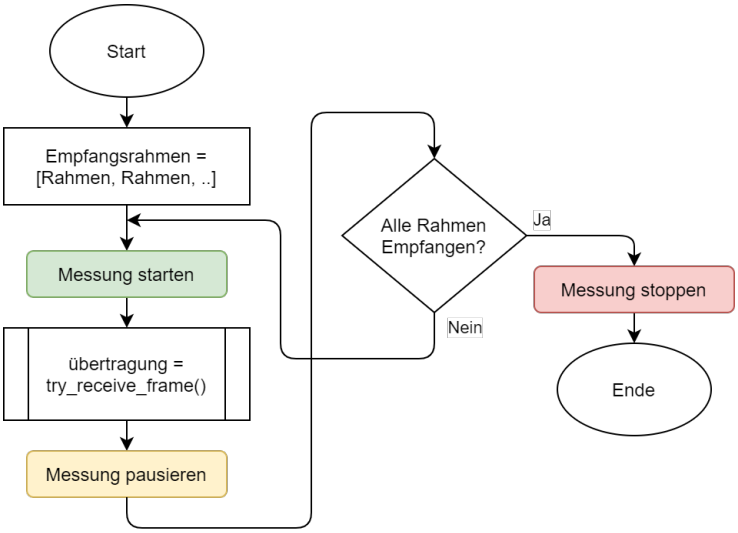

Abbildung (5.4): Ablaufdiagramm der Messungen des Empfangens der UAVCAN-Implementation in Rust

5.3 Auswertung der Zeitmessungen

In den nun folgenden Abschnitten werden zuerst die Messungen für die jeweilige Arduino- und Rust-Implementation anhand von Schaubildern erklärt und dargestellt. Anschließend werden die Messwerte zueinander in Bezug gesetzt.

Die Messwerte werden abhängig von der für den Transport benötigten Rahmenanzahl f ausgewertet und in Mikrosekunden angegeben. Zwischen den Messpunkten ist die Zeit linear interpoliert, damit Beziehungen zwischen den Messreihen besser ersichtlich sind.

Die Messkurven sind farblich gezeichnet, wobei Kurven mit derselben Farbe in Beziehung zueinander stehen.

5.3.1 Arduino Referenzmessungen

Senden von Übertragungen

Die Messverläufe für den Sendevorgang sind in Abbildung 5.5 auf der linken Seite dargestellt. Erste Messungen sind grün gezeichnet. Diese zeigen die Ausführzeiten der Bibliothek mit der integrierten Heap-Implementation. Die grün gestrichelte Linie stellt die Zeit dar, die initial für das Übergeben der Übertragung mit der Funktion *enqueueTransfer* an die Bibliothek benötigt wird. Somit ist die Differenz beider Kurven die Zeit, die zum Senden der Rahmen benötigt wird.

Um bessere Referenzmesswerte zu erhalten, wird ebenso wie bei der Rust-Implementation der Heap-Algorithmus TLSF verwendet. Diese Änderung führt zu den rot dargestellten Kurven in Abbildung 5.5. Hierbei stellt die rot gestrichelte Kurve erneut die Zeit des Funktionsaufrufs von *enqueueTransfer* dar.

Die in gelb dargestellten Werte sollen eine berechnete Näherung darstellen, bei der kein dynamischer Speicher verwendet wird, da in Rust beim Senden komplett auf den Heap verzichtet wird. Die Näherung wird durch Gleichung 5.5 berechnet, wobei $\vec{t}_{gesamt}$ die mit dem TLSF Algorithmus gemessenen Zeiten sind.

Die Zeiten für $\vec{t}_{heapalloc/free}$ in Gleichung 5.5 werden durch Tests ermittelt. Eine Analyse hat ergeben, dass pro Rahmen eine Struktur von 40 Byte gespeichert wird. Demnach wird die Zeit $t_i(f)$ für i Rahmen mit $f * 40\,\mathrm{B}$ berechnet, wobei f die Rahmenanzahl ist.

$$\vec{t}_{ohneheap} = \vec{t}_{gesamt} - \vec{t}_{heapalloc/free} \tag{5.5}$$

Empfangen von Rahmen

Für den Empfangsvorgang werden zuerst Messungen mit dem in der Arduino Bibliothek implementierten Heap durchgeführt. Dies veranschaulicht die grüne Linie in Abbildung 5.5

auf der rechten Seite. Des Weiteren werden weitere Messungen mit dem TLSF Algorithmus durchgeführt, die in rot dargestellt sind, damit die Messergebnisse mit der Rust-Implementation vergleichbar werden.

Zu sehen ist, dass die Verwendung des TLSF Algorithmus eine Verbesserung der Empfangszeit zu Folge hat. Die Verbesserung ist nicht so stark wie beim Senden. Dies basiert darauf, dass bei einem Empfangsvorgang einmalig dynamischer Speicher angefordert und freigegeben wird. Demnach ist die eingesparte Zeit durch den Heap Algorithmus relativ gering.

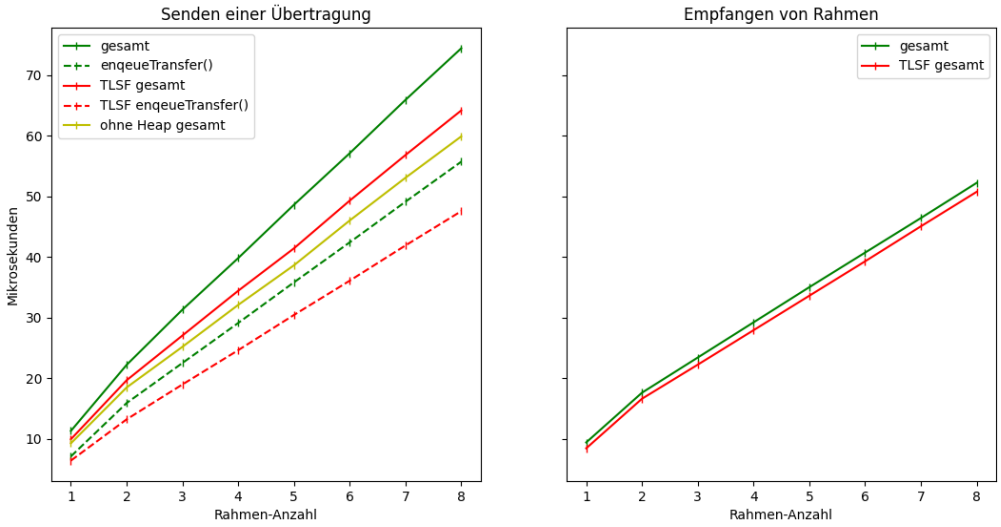

Abbildung (5.5): Messungen der Arduino-Implementation.

5.3.2 Rust-Implementations Messungen

Entsprechend werden nun bei der Rust-Implementation dieselben Messreihen ausgewertet, mit dem Unterschied, dass die Rahmen-Anzahl bis zu 16 Rahmen beträgt. Dies soll es ermöglichen, das Verhalten bei größeren Übertragungen auf Abweichungen zu untersuchen.

Senden von Übertragungen

In Abbildung 5.6 auf der linken Seite sind die Messpunkte der Messung des Sendens aufgetragen. Für dieses Szenario wird nur eine Messreihe durchgeführt. Beim Versuch von Verbesserungen an der Programmlogik für eine optimierte Ausführungszeit konnten keine signifikanten Ersparnisse erzielt werden.

Bei der Messung der Übertragung eines Rahmens fällt auf, dass diese Zeit nicht zum linearen Verlauf der weiteren Messungen in der Messreihe passt. Dies lässt sich damit erklären, dass für diesen Fall keine CRC-Prüfsumme berechnet wird und demnach diese

Zeit wegfällt. Dieses Verhalten ist auch bei der Arduino-Implementation festzustellen. Jedoch lässt es sich in Rust besser zu der CRC-Prüfsumme hin eingrenzen, da der Speicher beim Senden in Rust nur statisch verwendet und demnach dieser Faktor ausgeschlossen wird. Anhand von Gleichung 5.6 lässt sich die wegfallende Zeit t_c für die Berechnung der Prüfsumme annähern und liegt bei den Messungen etwa bei 1,665 µs.

$$\backsim t_c(n) = t_2 - \frac{\sum_{i=3}^{n} t_i - t_{i-1}}{n-1} \tag{5.6}$$

wobei:

t_i : Zeit für das Senden von i Rahmen
n : Anzahl ausgewerteter Messungen

Empfangen von Rahmen

Der rechte Graph in Abbildung 5.6 zeigt den Messverlauf der Messungen für das Empfangen von Rahmen. Dabei sind die Messungen, unterteilt in Worst- und Best-Case, die durch das Speicherbeziehen für die zu empfangenden Nachrichten zustandekommen.

Die erste Messreihe ist grün dargestellt. Dabei handelt es sich um Zeiten, die über den ursprünglichen Stand der Bibliothek gemessen werden und zur ersten Orientierung dienen, wie viel Zeit die Rust-Implementation im Vergleich zur Arduino-Implementation benötigt. Dabei wird festgestellt, dass die Werte der Messergebnisse bei Rust größer sind, vor allem beim Empfangen mehrerer Rahmen, als die in C++. Daher wird die Rust-Implementation auf Optimierungsmöglichkeiten untersucht. Die durchgeführten Optimierungen sind im Folgenden aufgelistet.

- Die erste Optimierung hängt mit der Verwendung des dynamischen Speichers zusammen. Kurzgefasst wurde vor der Optimierung für eine neue Session ein *Vec* der Rust Bibliothek mit einer Standardkapazität initialisiert. Ein Testversuch ergab, dass diese gleich Null ist. Wird diese Kapazität während des Empfangens neuer Rahmen überschritten, muss neuer Speicher angefordert werden [29]. Die Größe der zu empfangenden Nachricht ist jedoch bekannt.

  Zur Optimierung wird demnach beim Empfangen einer neuen Session der Puffer mit einer Standardkapazität versehen. Diese hängt von der Länge der zu empfangenden Daten ab und kann abhängig von der Rahmen-Anzahl mit Gleichung 5.1 berechnet werden. Diese Größe muss wie in Listing 3.2 in Zeile 9 beim Abonnieren einer *subject*-ID angegeben werden.

  Abbildung 5.6 zeigt die Messverläufe nach der Optimierung, in rot dargestellt. Nun sind die Messverläufe linear und steigen nicht so stark wie zuvor, vor allem beim Empfangen von mehr als einem Rahmen. Die Abweichung bei vier Rahmen der

vorherigen Messung muss demnach mit der dynamischen Erweiterung des Empfang-Puffers zu tun haben. Der flache Verlauf lässt darauf schließen, dass pro empfangener Rahmen Zeit eingespart wird.

- Nach der signifikanten Verbesserung fällt bei der Implementation auf, dass bei Abschluss einer Session der Speicher für diese erneut freigeben wird. Eine Session wird geschlossen, wenn deren angegebene Ablaufzeit dieser überschritten oder eine Nachricht komplett empfangen ist.

  Somit muss für das Empfangen einer neuen Nachricht mit derselben *subject*-ID der Speicher wieder allokiert werden. Zum einen ist dieses Verhalten von Vorteil bei selten empfangen Nachrichten, da der Speicher nicht belegt bleibt. Zum anderen wird Zeit benötigt, die bei frequent empfangenen Nachrichten einer *subject*-ID eingespart werden kann.

  Ein Kompromiss zwischen Speicherverbrauch und Zeiteinsparung kann sein, dass der Programmierer pro *subject*-ID angibt, ob die Session wiederverwendet oder immer wieder freigegeben werden soll. Eine weitere Möglichkeit ist, dass die Vorgehensweise basierend auf der Länge der zu empfangenden Nachrichten gewählt wird. Damit dieses automatisierte Verhalten nachvollziehbar ist, muss es für den Benutzer ersichtlich sein, wann welches Verhalten auftritt. Eine hybride Lösung ist ebenfalls denkbar. Die Bibliothek trifft die Entscheidung, jedoch hat der Benutzer die Möglichkeit, Einfluss zu nehmen und das Verhalten statisch anzugeben.

  Folglich wird die Implementation, die möglichst viel Zeit einspart, weiter verfolgt. Dabei wird eine Session und somit der Empfangs-Puffer wiederverwendet. Daraufhin folgen Zeitmessungen, die einen Worst- und einen Best-Case haben. Der Worst-Case tritt immer bei der ersten zu empfangenden Nachricht auf, was in Abbildung 5.6 als blau gestrichelte Linie zu sehen ist. Der Best-Case hat dabei konstant einen Offset von der Zeit, die für das Beschaffen des Speichers benötigt wird.

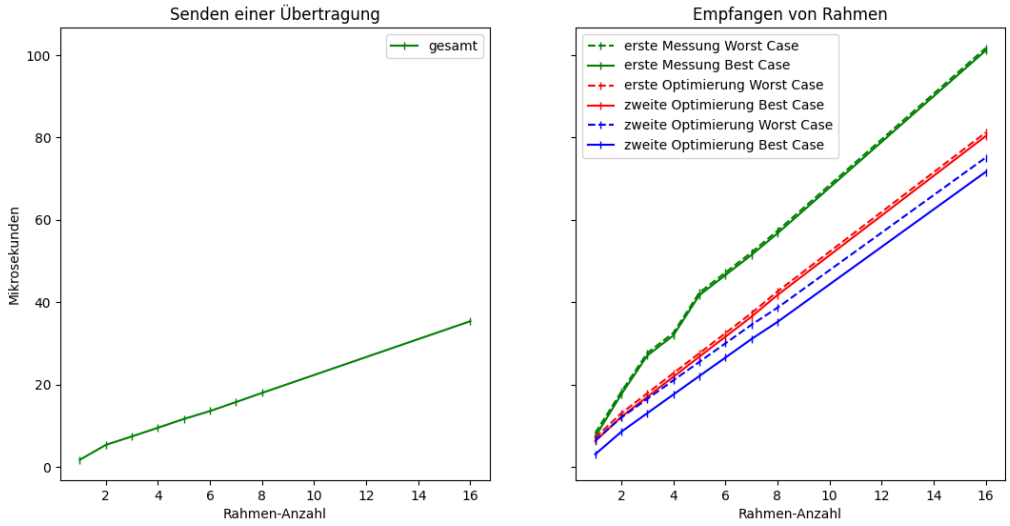

Abbildung (5.6): Messungen der Rust-Implementation.

5.3.3 Vergleich der Rust und Arduino Messungen

Für den Vergleich der Messungen werden die kürzesten Zeiten der jeweiligen Implementation übernommen. In Abbildung 5.7 sind jeweils die durch Optimierungen erzielten Messergebnisse mit den durchgezogenen Linien dargestellt.

Beim Senden von Übertragungen werden bei der Arduino-Implementation die Messungen mit dem TLSF Algorithmus verwendet. Diese sind in Abbildung 5.7 durch die roten Kreuz-Markierungen auf der linken Seite zu sehen. Die durchgezogene rote Linie zeigt die berechneten Zeiten ohne Heap. Für das Senden in Rust wird die Kurve aus Abbildung 5.6 übernommen und entsprechend in Abbildung 5.7 grün dargestellt.

Zum Vergleich der Messergebnisse beim Empfangen von Rahmen werden von der Arduino-Implementation die TLSF Messungen aus Abbildung 5.5 verwendet und von der Rust-Implementation diese aus Abbildung 5.6, die über die letzte Optimierung gemessen wurden.

Sowohl beim Senden als auch beim Empfangen von Rahmen, weist die Kurve einen Knick bei einer Anzahl von zwei Rahmen auf. Dies erklärt sich dadurch, dass bei der Übertragung von mehreren Rahmen die Prüfsumme berechnet werden muss. Diese Zeit fällt bei der Übertragung von nur einem Rahmen weg.

Zur besseren Übersicht sind in Tabelle 5.1 signifikante Zeitdifferenzen aufgeführt, die aus Abbildung 5.7 entnommen sind. Dabei ist zu beachten, dass die Rust-Implementation jeweils die Implementation ist, die weniger Zeit benötigt und somit die Differenzen mit $\Delta t = t_{arduino} - t_{rust}$ berechnet sind.

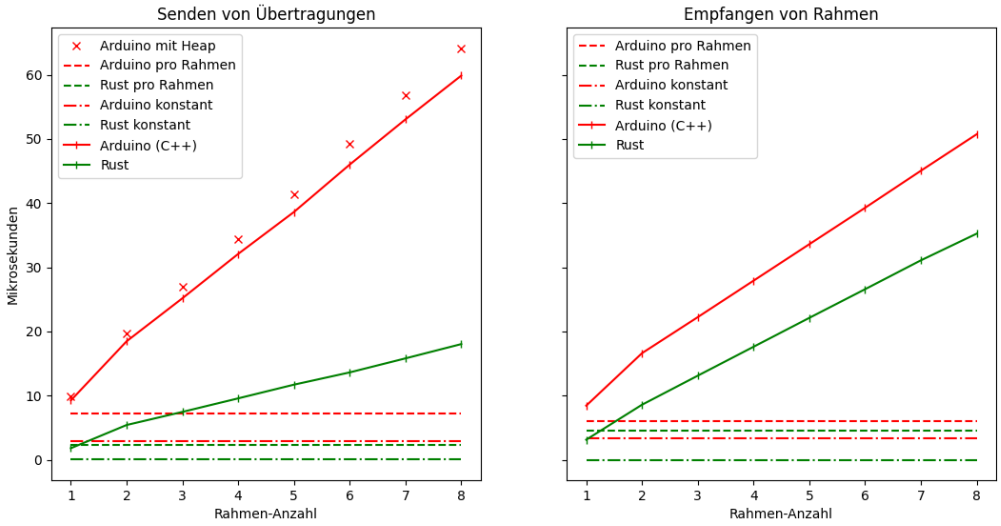

Abbildung (5.7): Vergleich der Arduino/Rust Messungen.

Kategorie	Δt Senden [µs]	Δt Empfangen [µs]
ein Rahmen	7,455	5,306
8 Rahmen	41,847	15,491
pro Rahmen	4,913	1,455
initial	2,818	3,410

Tabelle (5.1): Zeitdifferenz zwischen den Implementationen.

Nachdem zuvor auf die Zeitmessungen und auf API Aufrufe während den Messungen eingegangen wurde, können die Zeitmessungen in eine initiale Zeit und in eine Zeit pro Rahmen rechnerisch unterteilt werden. Dabei wird beim Senden immer initial ein Funktionsaufruf für eine Übertragung gemacht und weitere Funktionsaufrufe pro Rahmen. Dieses Muster ist zwar beim Empfangen bei den API Aufrufen nicht direkt zu erkennen, ist aber auch hier vorhanden. Dabei werden die Rahmen einzeln empfangen, wodurch die Zeit pro Rahmen zustande kommt. Ist die ganze Übertragung empfangen, wird diese mit den empfangenen Rahmen vervollständigt, das die initiale Zeit darstellt.

Hierbei kann $t_{proRahmen}$ mit der Steigung ab zwei Rahmen berechnet werden und $t_{initial}$ mit Gleichung 5.7. Diese Gleichung kann angewandt werden, da der Verlauf ab 2 Rahmen sowohl beim Senden als auch beim Empfangen nahezu linear ist.

$$t_{initial} = t_2 - (2 * t_{proRahmen}) \tag{5.7}$$

Hierbei ist auffällig, dass $t_{initial}$ bei der Rust-Implementation beim Senden und Empfangen nahezu Null ist, siehe Abbildung 5.7. Dies schließt darauf, dass die Rust-Implementation beim Senden die Übertragung nicht einmalig in Rahmen teilt, sondern jeden Rahmen dynamisch generiert. Beim Empfangen wird jeder empfangene Rahmen direkt in die vollständige Übertragung eingesetzt. Weiter fällt auf, dass die Rust-Implementation dennoch weniger Zeit pro Rahmen als die Arduino-Implementation benötigt. Das lässt darauf schließen, dass der Rust Compiler besser auf die Ausführungszeit optimiert.

Zusammengefasst kann gesagt werden, dass die bessere Ausführungszeit der Rust-Implementation auf eine Implementation zurückzuführen ist, bei der die Rechenzeit effizienter eingesetzt wird. Das heißt, es wird beim Senden auf dynamischen Speicher verzichtet, wobei der Nutzer dennoch die API effizient, ohne Programmieraufwand, verwenden kann.

Da beim Senden und Empfangen von Übertragungen mit einem Rahmen das Vorgehen nahezu dasselbe ist, sind die unterschiedlichen Messergebnisse einer besseren Kompilierung von Rust zuzuordnen. Diese wirkt sich dann entsprechend weiter bei größeren Übertragungen mit mehreren Rahmen aus, da dabei die Logik von einem Rahmen mehrmals ausgeführt wird.

5.4 Gegenüberstellung der Messergebnisse der entwickelten Benchmark-Suite und der API Messungen

Nun werden die API Messungen in der entwickelten Benchmark-Suite (Kapitel 4.4) durchgeführt. Daraus wird die Funktion und die Richtigkeit der Benchmark-Suite evaluiert.

Nach der Ausführung der Benchmark-Suite ist die in Listing 5.3 aufgelistete Ausgabe auf dem Terminal zu sehen. Für jede Messung ist das entsprechende Label mitangegeben. Außerdem wird der Median der 100 durchgeführten Messungen in ns/Iteration ausgegeben. Die letzte Spalte gibt jeweils die Abweichung zum Median an und wird mit $max(\vec{t}) - min(\vec{t})$ berechnet. Diese Angaben sind der in der Rust Standardbibliothek implementierten Bench-Funktionalität nachempfunden.

Ein solcher Durchlauf dauert ohne Kompilierungs- und Flash-Zeit 2,26 sec, wie in Listing 5.3 in Zeile 20 ausgegeben wird. Diese Zeit ist abhängig von der Anzahl der zu überprüfenden Messeinheiten und derer Ausführungszeit. Mit dem Windows Terminal Werkzeug *Measure-Commands* wird eine Gesamtausführungszeit von 65,14 sec gemessen. Dazu zählen die bereits erwähnte Ausführungszeit von 2,26 sec, die vom Compiler ausgegebene Kompilierzeit von 57,14 sec und sonstige Zeit, wie zum Beispiel die Zeit für den Übertrag der Firmware auf den Mikrocontroller.

```
1  running benches ...
2
3  function: bench_send_01    1_558 ns/iter (+/- 24)
4  function: bench_send_02    4_394 ns/iter (+/- 600)
5  function: bench_send_03    5_658 ns/iter (+/- 689)
6  function: bench_send_04    6_994 ns/iter (+/- 600)
7  function: bench_send_05    8_294 ns/iter (+/- 606)
8  function: bench_send_06    9_652 ns/iter (+/- 600)
9  function: bench_send_07    10_917 ns/iter (+/- 688)
10 function: bench_send_08    12_305 ns/iter (+/- 611)
11 function: bench_receive_01    3_176 ns/iter (+/- 3_294)
12 function: bench_receive_02    8_423 ns/iter (+/- 3_023)
13 function: bench_receive_03    12_887 ns/iter (+/- 3_241)
14 function: bench_receive_04    17_293 ns/iter (+/- 2_794)
15 function: bench_receive_05    21_698 ns/iter (+/- 3_253)
16 function: bench_receive_06    26_127 ns/iter (+/- 3_235)
17 function: bench_receive_07    30_555 ns/iter (+/- 3_919)
18 function: bench_receive_08    34_961 ns/iter (+/- 3_141)
19 ------------------------
20 finished - took 2.261s
```

Listing (5.3): Messergebnisdarstellung der Bench-Suite.

Während der Erstellung der Benchmark-Suite gab es Bedenken, dass die Zeitmessungen durch unterschiedlichste Optimierungen des Compilers nicht mit Messwerten einer reali-

tätsnahen Applikation übereinstimmen. Deshalb werden die Messergebnisse mit den Ergebnissen aus Kapitel 5.3 verglichen.

Hierzu werden in Abbildung 5.8 die Messwerte in grün dargestellt. Als Referenz sind die Messwerte der Arduino-Implementation in rot veranschaulicht, wobei dies für die Auswertung der Benchmark-Suite keine Relevanz hat. Es ist zu erkennen, dass beim Empfangen die Messwerte der Benchmark-Suite und die der Einzelmessungen nahezu übereinstimmen. Folglich repräsentieren die Benchmark-Suite-Messwerte die Ausführungszeiten einer realitätsnahen Anwendung sehr gut.

Beim Senden von Übertragungen stimmen die Ausführungen der Benchmark-Suite und die API Messungen aus Kapitel 5.3.3 bei einem Rahmen überein. Bei mehreren Rahmen entsteht ein immer größer werdender Unterschied. Dabei steigen bei der Benchmark-Suite die Messwerte weniger an als bei den Einzelmessungen. Dies lässt darauf schließen, dass bei dieser das Erstellen eines Rahmens weniger Zeit benötigt. Da die Messwerte dennoch linear verlaufen, kann berechnet werden, dass die Benchmark-Suite pro Rahmen 0,08 µs weniger Rechenzeit benötigt und sich somit bei 8 Rahmen ein Unterschied von 5,13 µs ergibt.

Die Abweichungen der Messwerte beim Empfangen werden auf den Worst-Case, der auch bei den Einzelmessungen gemessen wird, zurückgeführt. Jedoch sind die Abweichungen beim Senden von den Übertragungen nicht auf eine Logik der Bibliothek zurückzuführen. Durch weiteres Analysieren der Messwerte einzelner Iterationen konnte festgestellt werden, dass diese Abweichungen zufällig auftreten und kein richtiges Muster aufweisen. Die einzige Gemeinsamkeit ist, dass diese rund 600 ns betragen. Tests ergeben, dass die Abweichungen schlussendlich auf Interrupts durch die Implementation der Uhr zurückzuführen sind und zufällig auf manche Messungen fallen.

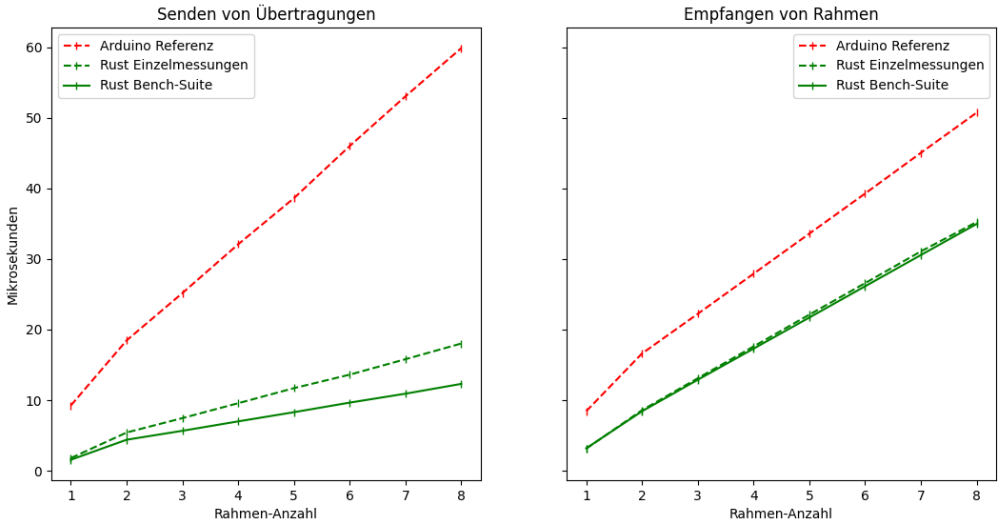

Abbildung (5.8): CAN-Nachrichten auf dem Bus.

Insgesamt kann gesagt werden, dass die Messergebnisse der Benchmark-Suite gut mit denen der API Messungen übereinstimmen. Dennoch führen kleine Optimierungsunterschiede zu anderen Ausführungszeiten. Die Benchmark-Suite kann Änderungen zu vorherigen Bibliotheksversionen hinsichtlich einer Verbesserung oder Verschlechterung in Bezug auf die Ausführungszeit relativ evaluieren. Des Weiteren konnte durch Tests ermittelt werden, dass die Messergebnisse bei Hinzufügen weiterer Messeinheiten gleichbleiben und nicht durch verschiedene Optimierungen variieren.

5.5 Zeitmessungen eines Echo-Knotens

Nun sollen Messungen an einem Szenario durchgeführt werden, bei dem ein CAN-Bus eingebunden ist. Dabei werden zwei Nucleo-G431RB Boards verwendet. Das eine Board soll einen Mess-Knoten darstellen und das andere einen Echo-Knoten. Die Knoten kommunizieren über *Classic*-CAN. Als Kommunikations-API dienen die UAVCAN-Implementationen.

Der Mess-Knoten basiert bei allen Messungen auf einer in Rust geschriebenen Software. Demzufolge unterscheiden sich die Messungen nur in der Ausführzeit des Echo-Knotens. Ein Nachteil dieser Herangehensweise ist, dass es keine Messergebnisse für einen gesamten, in C++ geschriebenen Aufbau gibt.

Das Szenario sieht wie folgt aus. Zuerst legt der Mess-Knoten variabel lange Daten in Form einer UAVCAN Übertragung auf den CAN-Bus. Der Echo-Knoten hat die *subject*-ID, auf welche die Übertragung gesendet wird, abonniert und empfängt die Datenrahmen, die durch die UAVCAN Bibliothek erneut zu einer Übertragung zusammengesetzt werden. Durch Kopieren einzelner Bytes wird die Übertragung kopiert, durch die UAVCAN Bibliothek erneut in Rahmen des Transports zerlegt und auf den CAN-Bus gegeben. Diese wird vom Mess-Knoten erneut empfangen. Über diesen gesamten Ablauf hinweg, also ab dem Verarbeiten zum Senden im Mess-Knoten bis zum erneuten kompletten Empfangen im Mess-Knoten, wird t_{ges} gemessen.

Zunächst soll die Zusammensetzung von t_{ges} theoretisch betrachtet werden. Danach werden die tatsächlichen Messungen des Hardwareaufbaus ausgewertet.

5.5.1 Theoretische Betrachtung der Zeitmessungen

In Abbildung 5.9 ist die Kommunikation zwischen dem Mess-Knoten und dem in Rust geschriebenen Echo-Knoten dargestellt. Bei diesem Modell wird die Zeit, die für das Kopieren der Übertragung im Echo-Knoten benötigt wird, weggelassen und es werden nur die Kommunikation und die Bibliotheksaufrufe der UAVCAN Bibliothek berücksichtigt. Bei dem veranschaulichten Vorgang werden Übertragungen verwendet, die nicht in einen Rahmen passen und demnach in mehreren CAN-Nachrichten gesendet werden müssen.

Wie zuvor erwähnt erstreckt sich die Zeit t_{ges} über das Senden und bis zum Empfangen im Mess-Knoten. In der weiteren Betrachtung wird diese Zeit in kleinere Zeitabschnitte eingeteilt.

Zuerst werden die Daten im Mess-Knoten mithilfe der UAVCAN Bibliothek verarbeitet. Dabei wird die Funktion *transmit* aufgerufen. Diese erstellt den Iterator, über diesen die Nachricht aufgeteilt wird. Durch den Aufruf von *next* wird der erste Rahmen abgerufen und anschließend über den verwendeten HAL auf den Bus gegeben. Sobald der Bus in der Zeit t_1 wieder frei ist, wird der nächste Rahmen gesendet. Dieser Vorgang wird n mal wiederholt, bis alle Rahmen versendet sind.

Der Echo-Knoten empfängt den ersten Rahmen nach t_1, nachdem dieser losgeschickt wur-

de. Daraufhin werden die Rahmen parallel gesendet und im Echo-Knoten verarbeitet. Dieser Sende-Empfangsvorgang dauert die Zeit t_2.

Die im Echo-Knoten benötigte Zeit setzt sich aus der Verarbeitung des zuletzt gesendeten Rahmens, aus dem Kopieren der Übertragung und aus dem erneuten Verarbeiten durch die UAVCAN-Implementation fürs Senden über den CAN-Bus zusammen.

Die Zeit t_1 wird nach Gleichung 5.8 berechnet. Diese Zeit hängt von der Länge der CAN-Nachricht und der Geschwindigkeit t_{proBit} des CAN-Bus ab. Die Länge der CAN-Nachricht setzt sich hierbei aus den Nutzdaten p in Bytes und 67 bit zusätzlicher CAN-Nachrichten Felder zusammen, siehe Kapitel 2.3.3.

$$t_1(p, t_{proBit}, n_{stuffingBits}) = ((p * 8\,\text{bit}\,\text{B}^{-1} + 67\,\text{bit}) + n_{stuffingBits}) * t_{proBit} \tag{5.8}$$

Wobei:

p	: Länge der Daten in einer CAN-Nachricht	[B]
t_{proBit}	: Zeit pro Bit auf dem CAN-Bus	[µs]
$n_{stuffingBits}$	: Aufgefüllte Bits in einer CAN-Nachricht	[bit]

Für die Messungen und den aktuellen Hardwareaufbau sollen folgende Werte verwendet werden:

p	$= 8\,\text{B}$
t_{proBit}	$= 1\,\text{µs}$
$n_{stuffingBits}$	$\approx$ min: 0 bit; Messungen : 7,75 bit

dies führt zu den folgenden Werten für Gleichung 5.8:

$$t_{1\_min} = 131\,\text{µs}$$
$$t_{1\_messungen} = 138{,}75\,\text{µs}$$

Die Empfangszeit t_2 wird nach Gleichung 5.9 berechnet und hängt von t_1 und der Anzahl f der empfangenen Rahmen ab. Die Annahme, dass t_2 nicht von weiteren Variablen abhängig ist, besteht solange die Zeit $t_{try\_receive} \leq t_1$ und dadurch der Empfangsknoten direkt bereit ist, die jeweils nächste Nachricht zu empfangen. Außerdem dürfen keine weiteren CAN-Knoten gleichzeitig Nachrichten auf dem Bus schicken, sodass keine Kollisionen auftreten.

$$t_2(n) = t_1 * f \tag{5.9}$$

Wobei:

f	: Anzahl der Rahmen für eine Übertragung

Die Zeit im Messknoten wird durch Gleichung 5.10 angenähert. Dabei werden die Zeiten der aufgerufenen Funktionen mit einberechnet und jeweils als $t_{<Funcktionsname>}$ bezeichnet.

$$\begin{aligned} t_{mess} &= t_{transmit} + t_{next} + t_{try\_receive} \\ t_{mess} &= 0\,\mu s + 2{,}2\,\mu s + 4{,}6\,\mu s \\ t_{mess} &= 6{,}8\,\mu s \end{aligned} \tag{5.10}$$

Durch Gleichung 5.11 kann t_{ges} berechnet werden. Bei dem Rust Knoten kann t_{mess} mit der Zeit, die im Echo-Knoten benötigt wird, gleichgesetzt werden. Für den in Arduino geschriebenen Echo-Knoten muss diese Zeit einzeln berechnet werden.

$$t_{ges\_rust} = (t_{mess} + t_2) * 2 \tag{5.11}$$

Ein theoretischer Wert für die Echo-Knoten soll mithilfe der Messergebnisse aus Kapitel 5.3.3 für 8 Rahmen berechnet werden.

Dafür ergeben sich für den in Rust implementierten Echo-Knoten folgende Zeiten:

$$\begin{aligned} t_{ges\_rust\_min} &= (6{,}8\,\mu s + 1048\,\mu s) * 2 \\ t_{ges\_rust\_min} &= 2109{,}6\,\mu s \\ t_{ges\_rust\_messungen} &= (6{,}8\,\mu s + 1110\,\mu s) * 2 \\ t_{ges\_rust\_messungen} &= 2233{,}6\,\mu s \end{aligned}$$

Die minimale Umlaufzeit ergibt sich, wenn in der CAN-Nachricht keine extra Bits eingefügt werden. Für eine bessere Näherung an die eigentlichen Messungen, wird die durch Tests durchschnittliche ermittelte Anzahl an extra Bits (7 bit/Nachricht) verwendet.

Für eine allgemeinere Näherung, siehe Kapitel 2.3.3, wird mit 25 % an *Stuffing*-Bits gerechnet. Demzufolge würde eine t_{ges_rust} Zeit von 2633,6 µs resultieren.

Für das Messen des in Arduino implementierten Echo-Knotens kann die in Abbildung 5.9 gezeigte Kommunikation nahezu identisch übernommen werden, mit dem Unterschied, dass beim Echo-Knoten andere API Aufrufe angegeben werden müssen. Anstatt *try_receive*, *transmit* und *next* sind dies *onCanFrameReceive*, *publish* und *transmitCanFrame*.

Aufgrund dieser Anpassungen wird t_{ges} in diesem Fall durch Gleichung 5.12 berechnet.

Der Wert $t_{ges\_arduino\_messungen}$ wird mit $t_{1\_messungen}$ ermittelt.

$$\begin{aligned} t_{echo} &= t_{onCanFrameReceive} + t_{publish} + t_{transmitCanFram} \\ t_{ges\_arduino} &= t_{mess} + t_{echo} + t_2 * 2 \end{aligned} \tag{5.12}$$

$$\begin{aligned} t_{echo} &= 65\,\mu s \\ t_{ges\_arduino} &= 2291{,}8\,\mu s \end{aligned}$$

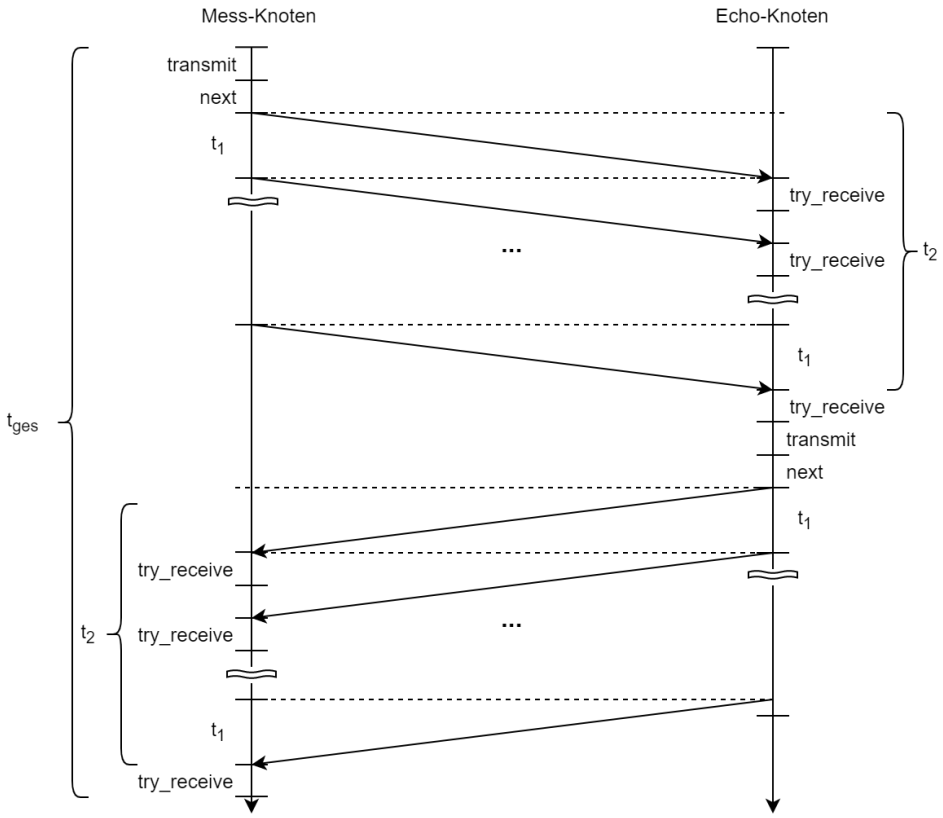

Abbildung (5.9): Kommunikation zweier Knoten über CAN (Rust Echo-Knoten).

5.5.2 Messergebnisse

Nachdem im vorherigen Kapitel die Messwerte theoretisch ermittelt wurden, werden die Messungen nun anhand des Hardwareaufbaus durchgeführt. Hierfür wird t_{ges} für einen, vier und acht Rahmen gemessen. Die Ergebnisse sind in Abbildung 5.10 auf der linken Seite dargestellt. Die grüne Kurve zeigt die Messungen für den in Rust geschriebenen Echo-Knoten und die rote die Messungen für den Echo-Knoten in Arduino.

Bei den Messungen ist aufgefallen, dass die Werte einer Messreihe zwischen Iterationen Abweichungen aufweisen. Bei den Implementationen in Rust und C++ sind die Abweichungen nahezu dieselben. Diese liegen bei einem Rahmen bei 4 µs und bei acht Rahmen bei 22 µs. Bei den Messungen von t_{ges} können diese Abweichungen vernachlässigt werden, weshalb die in Abbildung 5.10 dargestellten Messergebnisse über 100 Messungen gemittelt sind.

Wie bereits bei der theoretischen Berechnung von t_{ges}, ergab sich nun auch bei der Analyse mit dem Logic Analyser, dass die Kommunikation auf dem CAN-Bus die meiste Zeit benötigt. Dies sind im Mittel 97,7 % bei den Messungen mit dem Rust Echo-Knoten und 93 % mit dem Arduino Echo-Knoten.

Die Zeitdifferenz zwischen t_{ges} des Echo-Knotens in Rust und des in C++ beträgt bei einem Rahmen 32,58 µs und bei acht Rahmen 43,16 µs.

Im rechten Diagramm in Abbildung 5.10 sind die Latenzzeiten des Echo-Knotens aufgetragen. Diese werden durch den Logic Analyser ermittelt. Dabei wird die Pause auf dem Bus gemessen, dargestellt in Abbildung 5.11. Ebenso ist in der Grafik der Datenaustausch auf dem CAN-Bus dargestellt, wobei die Kommunikation in drei Phasen eingeteilt wird. Dies ist zum einen die Phase, in der die Nachrichten vom Mess-Knoten gesendet werden und zum anderen die Antwortphase des Echo-Knotens. Diese Zeit entspricht jeweils t_2 bei der theoretischen Betrachtung.

Die dritte Phase ist die Pause auf dem Bus, bei welcher kein Nachrichtenaustausch stattfindet und somit mit der Verarbeitungszeit im Echo-Knoten gleichzustellen ist. Die Zeit erstreckt sich über $t_{transmit} + t_{next} + t_{try\_receive}$ beim Echo-Knoten in Rust. Beim Echo-Knoten in C++ sind dies die entsprechenden Funktionen der Arduino API. Wie bereits erwähnt, werden bei der theoretischen Betrachtung Operationen im Echo-Knoten, wie zum Beispiel das Kopieren der Daten nach dem Empfang, nicht einbezogen.

Die gemessenen Verarbeitungszeiten der Echo-Knoten sind über Iterationen einer Messung konstant. Bei den Ergebnissen fällt auf, dass die Zeit, die der Arduino Echo-Knoten benötigt, beim Empfang mehreren Rahmen stark ansteigt. Die Rust-Implementation weist hingegen keinen starken Anstieg auf. Bei acht Rahmen benötigt der in Rust geschriebene Echo-Knoten mit 9 µs um den Faktor 7,43 kürzer als der in C++ geschriebene Knoten. Dies ist hauptsächlich durch die Implementation der Sendelogik erklärbar. Beim Arduino Echo-Knoten wird die zu sendende Übertragung zuerst komplett in einzelne Rahmen aufgeteilt und erst danach der erste Rahmen gesendet. In Rust hingegen wird jeweils nur der als nächstes zu sendende Rahmen erzeugt.

Schlussendlich benötigt der Rust Echo-Knoten zur Verarbeitung weniger Zeit als der Arduino Echo-Knoten. Die Unterschiede können auf die API Messungen aus Kapitel 5.2 bezogen werden.

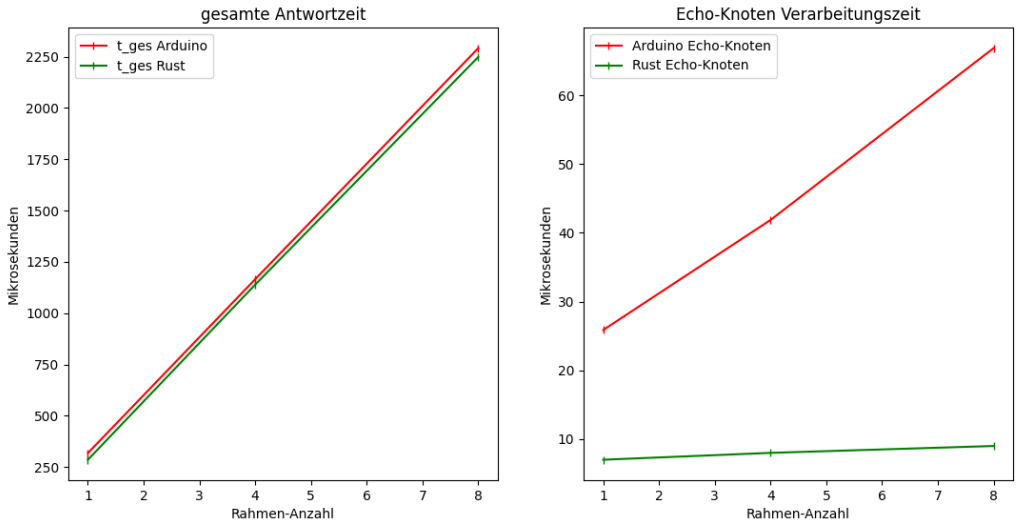

Abbildung (5.10): Messergebnisse der Echo-Knoten Implementationen.

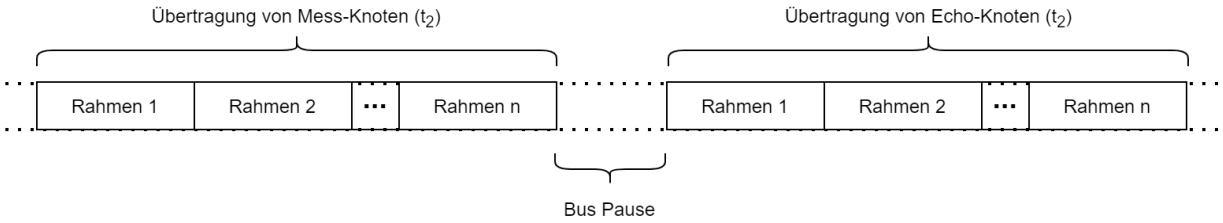

Abbildung (5.11): CAN-Nachrichten auf dem Bus.

Die Messergebnisse werden nun zur Übersicht den API Aufrufmessungen in Tabelle 5.2 gegenübergestellt.

Bei Rust fällt bei $t_{ges\_messungen}$ eine Differenz zwischen den theoretischen und der gemessenen Werte auf. Hierzu werden folgende Überlegungen gemacht:

1. Da die theoretischen Betrachtungen auf den API Messungen basieren, kann die Differenz aufgrund unterschiedlicher Optimierung auftreten.
2. Des Weiteren könnte die Zeitdifferenz durch zusätzliche Operationen in Verbindung mit CAN zustandekommen.
3. Außerdem kann es sein, dass der theoretische Messwert bei Arduino nicht exakt und ebenso wie bei Rust kleiner als die gemessene Zeit ist.

Die plausibelste Erklärung der Zeitdifferenz basiert auf der ersten und der letzten Überlegung. Aufgrund unterschiedlichster Faktoren, wie zum Beispiel der Codegröße und der

Unterschiede in der Ablauflogik, optimiert der Rust-Compiler das Programm unterschiedlich. Des Weiteren können die errechneten Werte bei der Arduino-Implementation aufgrund vorheriger Messfehler falsch berechnet worden sein.

Im Folgenden wird nicht weiter auf diese Differenz eingegangen. Der Fokus wird näher auf die benötigte Zeit im Echo-Knoten gelegt.

Die Zeiten t_{echo} der theoretisch berechneten Werte und der zugehörigen Messungen beider Implementationen liegen nah zusammen. Der Unterschied beträgt jeweils ungefähr 2 µs, was durch das Kopieren vor dem erneuten Senden der Daten und durch weitere benötigte Operationen erklärbar ist.

Zusammengefasst kann gesagt werden, dass sich die theoretischen Betrachtungen gut den Hardwaremessungen annähern. Dass bei den theoretischen Betrachtungen die Messwerte der API Messungen miteinbezogen werden, zeigt wiederum die Richtigkeit dieser. Dennoch ist zu beachten, vor allem bei der Gesamtumlaufzeit, dass zum Beispiel die Anzahl der Stuffing-Bits bei den Rechnungen aus den Analysen mit dem Logic Analyser entnommen werden.

Zeit	Rust [µs]		Arduino [µs]	
	theoretisch	gemessen	theoretisch	gemessen
$t_{ges\_messungen}$	2233,6 µs	2248,34 µs	2291,8 µs	2291,5 µs
t_{echo}	6,8 µs	8,9 µs	65 µs	67 µs

Tabelle (5.2): Theoretisch errechnete und gemessene Zeiten bei acht Rahmen im Vergleich.

5.6 Messen von Speicherverbrauch

In diesem Kapitel wird der Speicherverbrauch ausgewertet. Dabei wird der statische Speicherverbrauch, der nach der Kompilierung eines Programmes für den Mikrocontroller feststeht, betrachtet. Außerdem wird der Speicher während der Ausführung des Programmes überwacht. Hierbei wird der Fokus auf den Stack- und den Heap-Bereich gelegt.

Für die folgenden Messungen wird der jeweils in Rust und C++ geschriebene Echo-Knoten aus Kapitel 5.5 verwendet. Außerdem sollen die Compiler den Code auf Ausführungsgeschwindigkeit optimieren.

5.6.1 Statischer Speicherverbrauch

Mithilfe eines Linker Skriptes werden die Speicherbereiche in einem Mikrocontroller festgelegt. Dieses wird an den jeweiligen Mikrocontroller und dessen Speicher angepasst. Dabei werden Speicherbereiche, die zum einen während der Ausführung veränderlich (RAM) und zum anderen unveränderlich (ROM) sind, unterschieden.

Weiterhin wird das Layout in Segmente unterteilt. Dies sind zum einen stets benötigte Segmente wie *.text*, *.bss* und *.data* [45], zum anderen können weitere Segmente für spezielle Variablen definiert werden. Im Folgenden soll kurz der Inhalt der aufgeführten Segmente und deren Speicherbereich erläutert werden.

.text Dieses Segment wird im ROM Bereich auf dem Mikrocontroller gespeichert und beinhaltet die auszuführenden Instruktionen und Konstanten [45].

.rodata Beinhaltet statische, unveränderliche Werte und befindet sich deshalb im ROM Speicherbereich.

.bss Ausgeschrieben *block starting symbol* Segment und beinhaltet statische und globale Variablen, welchen im Code kein Wert zugewiesen wird. Das Segment befindet sich im RAM [45].

.data Hierbei handelt es sich um statische und globale Variablen, die mit einem bestimmten Wert initialisiert werden. Dieses Segment ist ebenfalls im RAM [45].

In Abbildung 5.12 ist der ROM und RAM Bereich, speziell für die in dieser Arbeit verwendeten Mikrocontroller, siehe Kapitel 4.2, dargestellt. Diese Layouts werden durch Ausgabe der Speicherbereiche der kompilierten Firmware ermittelt. Der gelbe und grüne Bereich zusammen ergeben den RAM.

Die Segmente im RAM werden bei Rust und Arduino unterschiedlich platziert. Bei Arduino stehen diese direkt am Beginn und bei Rust am Ende des RAMs. Dabei muss in Abbildung 5.12 der untere Bereich der Grafik als Beginn gesehen werden, da dort der Speicher mit der Adresse *0x0000 0000* beginnt.

In Rust wird das umgekehrte Speicherlayout durch den Linker *flip-link*[18] erzielt, der zur Kompilierung angegeben wird. Dabei werden die statischen RAM Segmente oberhalb

des Stacks positioniert. Das resultierende Layout soll verhindern, dass der Stack in die im RAM definierten Segmente wächst [18]. Wie in Abbildung 5.12 auf der linken Seite zu sehen ist, wächst der Stack somit gegen den ROM Bereich. Da dieser aber während der Ausführung unveränderlich ist, führt ein Überlauf zu einem Fehler.

Für den verwendeten Heap im Programm wird ein zusätzliches Segment im RAM erzeugt, das dessen Speicherbereich von den restlichen Variablen abtrennt, was eine eindeutige Speicheranalyse zur Folge hat. Dieses Segment ist in Abbildung 5.12 im grünen Bereich als *Heap Bereich* dargestellt.

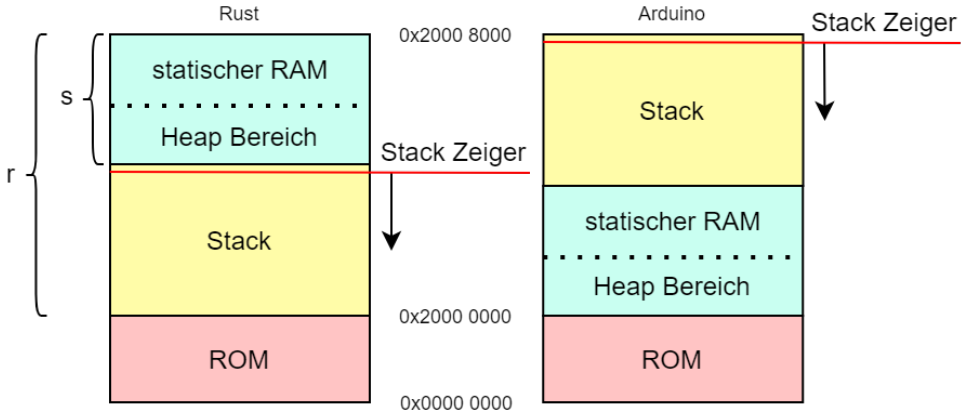

Abbildung (5.12): Speicherlayout auf dem Mikrocontroller.

Die Größen der Segmente, die durch die Kompilierung der in Rust und C++ geschriebenen Programme erzeugt werden, werden anschließend gegenübergestellt. Diese können mit dem Linux Befehl *size* ausgegeben werden. Tabelle 5.3 zeigt die Segmente, aufgeteilt in ROM und RAM.

Als verfügbarer Speicher wird der des bereits angeführten Mikrocontrollers betrachtet. Dieser hat 128 KiB an ROM Speicher zur Verfügung. Somit belegt die in Rust geschriebene Firmware 12,18 % und die in C++ geschriebene Firmware 28,74 %.

Bei den Segmentgrößen *.text* und *.rodata* gibt es Unterschiede. Das Segment *.text* ist bei Arduino um den Faktor 2,3 größer als bei Rust. Analysen ergaben, dass zum einen bei Arduino Funktionen hineingelinkt werden, die nicht zur Ausführung benötigt werden. Zum anderen bettet Rust, wenn möglich, Funktionen in übergeordnete Funktionen ein, was eine komplexere Optimierung zulässt.

Das *.rodata* Segment ist bei Arduino um den Faktor 3,4 größer als bei Rust. Hierbei zeigte eine Analyse, dass bei C++ Error Strings, sowie Pin Mappings und Funktionsnamen in diesem Segment enthalten sind. Bei Rust hingegen sind es vergleichsweise kurze Error Strings. Aufgrund dessen ist die Größe abhängig von der Implementation, insbesonders von der Verwendung von statischen Zeichenketten.

Im Weiteren soll die Auslastung im RAM Bereich betrachtet werden. Wie in Abbildung 5.12 dargestellt, werden im RAM das *.heap* Segment und die weiteren statischen Segmente aus Tabelle 5.3 platziert. Die Größe dieses Bereichs soll mit s bezeichnet werden.

Die Segmente *.data* und *.bss* sollen hierbei zusammen betrachtet werden. Nachdem Rust für statische und globale Variablen insgesamt 292 Byte verwendet, wird bei Arduino 4648 Byte benötigt. Hierbei muss erwähnt werden, dass eine Analyse der TLSF Implementation in C++ ergeben hat, dass diese bei der Initialisierung 3236 Byte benötigt, die im *.bss* Segment liegen. In Rust ist der TLSF Algorithmus speichereffizienter implementiert.

Das Segment *.heap* ist jeweils nahezu gleich groß. Bei der Arduino Software wird außerdem ein *.__user_heap_stack* Segment eingebaut, das nach [45] von der ANSI C Bibliothek definiert wird, um den Operator *new* zu verwenden.

Resultierend wird bei Rust 13,43 % und bei Arduino 31,37 % des RAMs durch die Bereiche belegt.

Für den Stack, der gelb markierte Bereich in Abbildung 5.12, bleibt eine Größe von $s_{stack} = r - s$, wobei r die Gesamtgröße des RAMs ist. Entsprechend ist bei Rust 28 368 Byte und bei Arduino 22 488 Byte Speicher frei.

Speicherbereich	Segment	Rust [B]	Arduino [B]
ROM	.vector_table | .isr_vector	472	472
	.text	14152	32576
	.rodata	1340	4576
	weitere	-	40
gesamt		15964	37664
RAM	.data	0	304
	.bss	292	4344
	.heap	4108	4096
	.__user_heap_stack	-	1536
gesamt		4400	10280

Tabelle (5.3): Segment-Größen der kompilierten Firmware.

5.6.2 Erstellung eines Werkzeuges zum Messen von Speicherauslastung

Nun wird die Speicherauslastung während der Ausführung der jeweiligen Firmware betrachtet. Hierfür wird ein Werkzeug erstellt, das auf einem Host-System ausgeführt wird.

Dieses soll die jeweilige Firmware auf den Mikrocontroller aufspielen und im Anschluss die Auslastung des Stack Speicherbereichs in Intervallen messen.

Das Werkzeug wird in der Programmiersprache Rust entwickelt und verwendet die Bibliothek *probe-rs*. Diese beinhaltet Funktionalität zum einen zur Übertragung der Firmware auf den Controller und zum anderen zur Steuerung und Überwachung des Mikrocontrollers.

Die Überprüfung der Auslastung des Speichers wird wie folgt umgesetzt. Zuerst soll der Speicher mit einem Muster beschrieben werden. Im Anschluss wird der Speicher ausgelesen und auf das Muster überprüft. Ist das Muster nicht mehr vorhanden, ist davon auszugehen, dass der Speicher beschrieben ist. Des Weiteren wird der aktuelle Stack Zeiger aus den CPU Registern ausgelesen. Dadurch kann eine Annahme der aktuellen Speicherauslastung im Stack Bereich getroffen werden.

Bei der Ausführung des Werkzeugs werden folgende Schritte abgearbeitet:

1. Die jeweilige Firmware einlesen.
2. Die eingelesene Firmware analysieren und Informationen, wie zum Beispiel die Position des Stack-Zeigers zu Beginn der Ausführung, auslesen.
3. Eine Verbindung zum Mikrocontroller aufbauen und anschließend die CPU anhalten.
4. Den Speicher mit dem zuvor erwähnten Muster beschreiben.
5. Die Firmware aufspielen.
6. Den Mikrocontroller zurücksetzen und starten.
7. In einer Schleife den Speicher auf das Muster untersuchen.
8. Nach einer variablen Zeit den Vorgang abbrechen und das Werkzeug beenden.

Für jede Speicherüberprüfung in der Schleife werden folgende Informationen gesammelt und berechnet:

Stack-Zeiger Versatz Der Versatz, den der Stack-Zeiger relativ zu seinem Start Punkt hat.

Gesamtauslastung Bereich in Bytes, der bei der kompletten Ausführung mindestens einmal beschrieben wird.

Regionen Bereiche, die fortlaufend beschrieben sind. Eine Bereichsgrenze wird gesetzt, sobald n Bytes nicht überschrieben werden. Demzufolge können mehrere Regionen in der Gesamtauslastung entstehen. Der Start der zuerst auftretenden Region und das Ende der letzten sind die Grenzen der Gesamtauslastung.

Instruktions-Zeiger Dieser gibt an, an welcher Stelle im Programm die Messung durchgeführt wird.

Am Ende der Ausführung werden die einzelnen Messungen in einer Datei gespeichert.

Um den Speicherverbrauch genauer zu analysieren, wird das Programm auf dem Mikrocontroller bis zu einer bestimmten Stelle ausgeführt. Hierzu wird anstelle der Schleife, siehe Ausführungsschritt sieben, ein Abbruchpunkt im Mikrocontroller gesetzt. Erreicht der Mikrocontroller diese Stelle, hält dieser an. Das Werkzeug liest die aufgelisteten Werte aus und zeigt diese an.

Eine andere Herangehensweise ist die Überwachung von Instruktion zu Instruktion. Dabei lässt der Benutzer das Programm Schritt für Schritt ablaufen und erhält jedes Mal die ausgelesenen Werte. Nach Testversuchen ergaben sich Schwierigkeiten darin. Zum einen muss jeder Schleifendurchgang, sobald eine Schleife von mehreren Iterationen auftritt, durchgeklickt werden. Zum anderen wird eine aufgerufene Interruptroutine kontinuierlich ausgeführt, ohne erneut an die auszuführende Stelle im Programm zu springen. Ein Ansatz kann eine hybride Lösung sein, bei der das Programm bis zu einer gewissen Stelle ausgeführt wird und der Benutzer die Möglichkeit hat, weitere Instruktionen einzeln auszuführen.

5.6.3 Messungen von Speicherauslastung zur Laufzeit

Im Folgenden wird zuerst der Stack und die CPU Register mit dem entwickelten Analyse Werkzeug ausgelesen. Anschließend wird zusätzlich die Heap Implementation überwacht, womit der dynamische Speicherverbrauch ausgelesen werden kann.

Stack

Die Stack-Auslastung wird in Byte gemessen. Dabei werden die beschriebenen Bytes im Stack vom Start des Programms an gezählt. Hierbei werden Bytes, welche möglicherweise erneut verwendet werden, als weiterhin belegt gezählt. Somit wird der Wert der Stack-Auslastung monoton steigen.

Die Messung wird in vier Programmablaufabschnitten durchgeführt. Diese sind im Folgenden aufgelistet:

Controller Startup beinhaltet Funktionsaufrufe, wie das Zurücksetzen des Controllers und Initialisieren von statischen Variablen. Diese Funktionen sind für den Programmierer nicht sichtbar.

Config & Setup In diesem Abschnitt wird das Board, sowie die Strukturen für den UAVCAN Knoten, konfiguriert.

Hauptschleife Leerlauf Der Echo-Knoten befindet sich hierbei in der Hauptschleife des Programms und wartet auf eine Übertragung auf dem CAN-Bus.

Hauptschleife Last Der Echo-Knoten befindet sich erneut in der Hauptschleife und empfängt, verarbeitet und sendet kontinuierlich Übertragungen.

In Abbildung 5.13 sind die Messergebnisse für Rust in grün und Arduino in rot dargestellt. Im ersten Abschnitt werden bei der C++ Firmware direkt mehr Bytes als bei der Rust Implementation belegt. Dies kommt daher, dass Arduino die Variablen des *.bss* Segments initialisiert.

Im nächsten Abschnitt ist die Anzahl der belegten Bytes ungefähr gleich. Dies ist jedoch Zufall.

In der Hauptschleife ist zu erkennen, dass die für Arduino geschriebene Software um den Faktor 3,5 weniger Byte belegt, als die in Rust geschriebene Software.

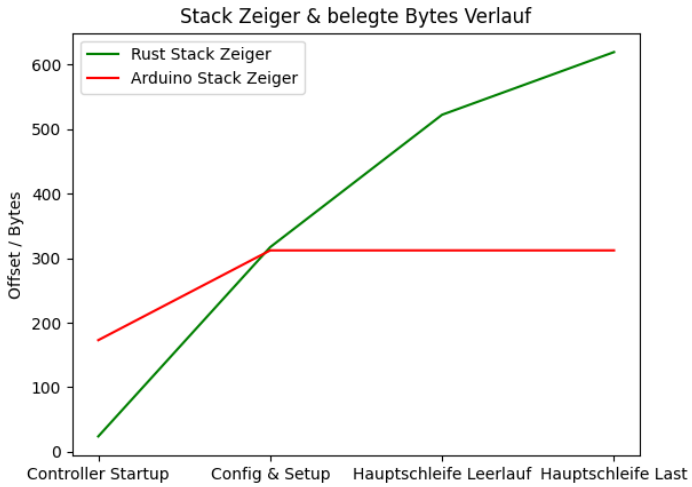

Abbildung (5.13): Anzahl maximal benutzter Bytes im Stack.

Um den Verlauf deutlicher analysieren zu können, muss Abbildung 5.14 betrachtet werden. In dieser Grafik wird der Abstand des Stack Zeigers zur Startadresse während der Hauptschleife unter Last dargestellt.

Bei der in C++ geschriebenen Firmware oszilliert der Stack Zeiger zwischen einem Abstand von 6 und 110 Byte, was eine Differenz von 104 Byte ergibt. Bei der in Rust geschriebenen Firmware tritt diese Oszillation im Bereich von 535 und 615 Byte auf, mit einer Differenz von 80 Byte.

Der Stack Zeiger des Rust Programms bewegt sich dabei dauerhaft in einem höheren Bereich als bei Arduino. Dies lässt sich nach einer Analyse des Assembler Codes auf die Funktionsschachtelung zurückführen. Dadurch, dass bei Rust die Konfiguration und die zusätzlichen Initialisierungsschritte in der *main* Funktion vorgesehen und durchgeführt werden, werden entsprechend für diese Funktion Bytes für lokale Variablen benötigt. Springt das Programm nun in die Hauptschleife, wird auf die letzte Position des Stack Zeigers aufgebaut.

Bei Arduino hingegen wird der Konfigurationsschritt in einer externen Funktion durchgeführt. Dadurch kann der Stack Zeiger beim Zurückkehren wieder auf seine alte Position gesetzt werden. Beim Durchlaufen der Hauptschleife wird mit einem geringeren Abstand als bei Rust gestartet und bei jeder Iteration wird der Stack Zeiger für die jeweiligen Funktionen verschoben und danach erneut auf die Ausgangsposition gesetzt.

Im Gegensatz zu Rust sind Variablen in C++, die in mehreren Funktionen, wie zum Beispiel der Konfigurationsfunktion und der Hauptschleife vorkommen, als global definiert. Demzufolge müssen diese nicht als lokale Variablen in Funktionen initialisiert und zwischen diesen übergeben werden.

Die etwas geringere Oszillation in Rust wird damit erklärt, dass der Rust Compiler, wenn möglich, Funktionen in nächsthöhere Funktionen einbettet. In diesem Fall läuft das Programm in einer Schleife in der *logic* Funktion ab. Diese beinhaltet kaum Funktionsaufrufe. Folglich kommen wenige Kontextwechsel auf dem Stack vor. Sobald jedoch eine Funktion aufgerufen wird, muss der komplette lokale Speicherbedarf für diese vorgesehen werden. Somit springt der Stack Zeiger zu einem größeren Abstand als zuvor. Dieses Verhalten kann beobachtet werden, wenn der Verlauf des Stack Zeigers vom Programmabschnitt „Zurücksetzen des Boards“ bis zum Abschnitt „Hauptschleife“, aufgezeichnet wird.

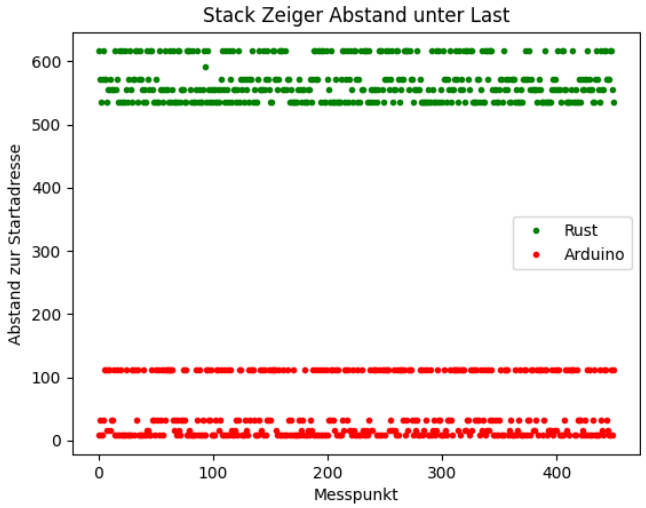

Abbildung (5.14): Abstand des Stack Zeigers bei der Programmausführung.

Mit dem Wissen über das Verhalten des Stack Zeigers kann nun auch der höhere maximale Speicherbedarf aus Abbildung 5.13 bei Rust erklärt werden. Da der Stack Zeiger bei Rust durch die Schachtelung der Funktionen mehr inkrementiert als dekrementiert wird, ist auch die Gesamtspeicherauslastung höher als bei C++.

Heap

Beim Messen von der Heap-Auslastung werden die Funktionsaufrufe zum Allokieren und Befreien von Speicherbereich überwacht. Dabei wird der Speicherbedarf der jeweiligen Operation aufgezeichnet. Hierbei soll nicht die exakte, vom verwendeten Allokations-Algorithmus abhängige Speicherauslastung gemessen werden, sondern nur die der UAVCAN Bibliotheks-Implementationen.

In Tabelle 5.4 sind die gemessenen Speichervorgänge aufgelistet und aufgeteilt in Speicherbedarfsgröße, in die Häufigkeit der Allokation und den Zeitpunkt, an dem der Speicherbereich wieder freigegeben wird. Das Auftreten der einzelnen Allokationen ist implementationsabhängig. Die Größe der zu allokierenden Bereiche hängt von den verwendeten Collektions ab, die in Rust aus der *alloc* Bibliothek und in C++ von der Standardbibliothek verwendet werden, sowie von den Verwaltungsstrukturen der jeweiligen Implementation.

Zur Ermittlung der Speicherbedarfsgrößen wird das Programm zurückgesetzt und ab diesem Zeitpunkt gemessen. Zuerst wird der Konfigurationsprozess durchlaufen, bis der Programmablauf in der Dauerschleife zum Empfangen und Senden von Übertragungen verweilt.

Im Konfigurationsschritt werden die in den zwei ersten Zeilen in Tabelle 5.4 aufgeführten Speicherbereiche angefordert und frühestens bei Beendung des Programmes wieder freigegeben. Bei der Rust-Implementation wird die Übertragungs-ID nicht von der Bibliothek verwaltet, weshalb dieser Vorgang wegfällt. Resultierend wird im Konfigurationsschritt 96 Byte bei Rust und 608 Byte bei Arduino belegt.

Zum Senden und Empfangen von Übertragung werden die Speicherbereiche aus Zeile 3 bis 5 belegt. Wie zuvor in Kapitel 3.3.2 analysiert, wird einmalig Speicher für das Erstellen einer Session und deren Empfangs-Puffer belegt. Diese Bereiche werden in der in Rust geschriebenen Software erst wieder bei Beendung des Programms freigegeben. Bei Arduino wird genauso wie bei Rust ein Puffer zum Empfangen von Übertragungen allokiert. Dieser wird jedoch nach vollständigem Empfangen erneut freigegeben und wird deshalb k mal erneut allokiert, wobei k die Anzahl von empfangenen Übertragungen ist. Beim in C++ implementierten Programm wird beim Senden ausschließlich dynamischer Speicher verwendet. Dabei werden acht einzelne Bereiche für eine Übertragung belegt und beim Senden des jeweiligen i-ten Rahmen wieder freigegeben. Diese Speicherallokation und Deallokation wird k mal im Programmablauf wiederholt.

Resultierend ergibt sich bei der Rust-Implementation eine maximale Heap Auslastung von 490 Byte, die nach dem ersten Empfangen eines Rahmens während der Programmausführung konstant bleibt. Bei der Arduino-Implementation ist der Heap maximal mit einer Anzahl von 928 Byte ausgelastet. Diese alterniert während der Hauptschleife zwischen den 608 Byte nach der Konfiguration und dem Maximalwert. Die maximale Auslastung bei Arduino ist demnach nahezu doppelt so hoch wie die der Rust-Implementation.

		Rust	Arduino
Abonnieren von einer *subject*-ID	Größe [B]	96	576
	Häufigkeit	1	1
	Deallokieren	1	1
Übertragungs-ID Verwaltung	Größe [B]	-	32
	Häufigkeit	-	1
	Deallokieren	-	1
Session Erstellung	Größe [B]	340	-
	Häufigkeit	1	-
	Deallokieren	1	-
Empfangen von Rahmen	Größe [B]	54	54
	Häufigkeit	1	k
	Deallokieren	1	vollständiger Empfang
Senden von Übertragungen	Größe [B]	-	$8 * 40$
	Häufigkeit	-	k
	Deallokieren	-	Senden von Rahmen i

Tabelle (5.4): Heap Auslastung.

5.6.4 Überblick über die Gesamtspeicherauslastung im RAM

Nach dem Überblick über die statische RAM Auslastung, die Stack Auslastung und die Heap Auslastung werden die einzelnen Bereiche zusammengefügt. Daraus ergibt sich die Gesamtspeicherauslastung, dargestellt in Abbildung 5.15.

Die statische Auslastung ergibt sich hierbei aus den Linker Segmenten (Tabelle 5.3). Dabei wird das *.heap* Segment ausgelassen, da dies bereits durch die dynamische Heap Auslastung miteinbezogen wird. In Abbildung 5.15 ist zu sehen, dass bei der in C++ geschriebenen Firmware der statische Bereich einen sehr großen Anteil ausmacht und somit die Kurven initial höher verlaufen als bei Rust.

Dann wird in Abbildung 5.15 die Heap Auslastung zum statischen Teil addiert, was durch die gestrichelt-gepunktete Kurve dargestellt ist. Dieser Anteil ist bei der Arduino-Implementation ebenso mehr als bei der Rust-Implementation.

Zuletzt wird die dynamische Stack Auslastung addiert. Dies macht bei der Rust-Implementation einen größeren Anteil aus.

Resultierend ergibt sich eine maximale Gesamtauslastung bei Rust von 1410 Byte und bei Arduino von 7420 Byte. Dies sind bei der Arduino-Implementation um den Faktor 5,2

mehr. Somit ist bei Rust noch 95,70 % und bei Arduino 77,36 % an RAM des verwendeten Mikrocontrollers frei.

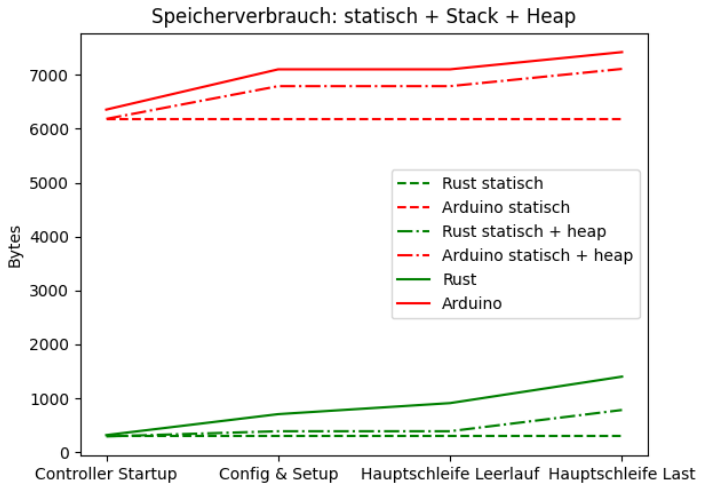

Abbildung (5.15): Gesamtspeicherauslastung im RAM.

6 Auswertung und Ausblick

6.1 Zusammenfassung und Auswertung der Arbeit

In dieser Arbeite wurden zuerst signifikante Merkmale der Programmiersprache Rust aufgeführt. Danach wurde der in Rust geschriebene UAVCAN Protokollstack auf ein eingebettetes System portiert. Hierbei ist es gelungen, die vorgenommenen Änderungen durch Push Anfragen auf GitHub in den aktuellen Stand der Bibliothek einzubinden. Daraufhin konnte eine auf dem UAVCAN Protokoll basierende Kommunikation zwischen zwei Mikrocontrollern, der eine mit einer in Rust, der andere mit einer in C++ geschriebenen Software aufgebaut werden. Des Weiteren wurde die Rust-Implementation anhand des Softwarequalitätsmerkmals Leistungsfähigkeit und dessen Kriterien Ausführungs- und Speichereffizienz evaluiert. Dafür wurden Messreihen an den in Rust und C++ geschriebenen UAVCAN Bibliotheken durchgeführt und anschließend die sich resultierenden Messwerte gegenübergestellt. Es wurde versucht, Differenzen der Messergebnisse auf die Implementationslogik und die verwendete Sprache zu beziehen. Während den Zeitmessungen konnten erfolgreich Optimierungen an der Rust-Implementation durchgeführt werden, die ebenfalls in das GitHub Projekt aufgenommen wurden. Für die weitere Entwicklung der Bibliothek konnte eine Möglichkeit gefunden werden, die Zeitmessungen für fortlaufende Versionen durchzuführen und weiterhin die Ausführungseffizienz zu evaluieren. Zusätzlich wurden Speichermessungen durchgeführt, wobei die statische Speicherauslastung nach der Kompilierung des Programms und die dynamische Verwendung des Heaps und Stacks gemessen wurden. Hierbei konnten markante Unterschiede in der Speichernutzung festgestellt werden, die zum einen auf die Implementationslogik der Bibliotheken und zum anderen auf die Kompilierung der Sprache zurückzuführen sind.

Bei der Rust-Implementierung des Kommunikationsstacks des UAVCAN Projekts konnten markante Änderungen durchgeführt werden, die die Entwicklung der Bibliothek weiter voranbringen. Somit wurde die Bibliothek auf *no_std* Umgebungen umgestellt. Hierzu wurden Abhängigkeiten zu der Rust Standardbibliothek entfernt. Stattdessen werden nun die Grundbibliotheken *core* und *alloc* verwendet. Im Zuge dessen wurde eine generische Zeitimplementation für die Bibliothek gefunden, bei dieser der Programmierer eine an das System angepasste Zeitquelle angeben muss.

Für das Testen der Änderungen der Bibliothek konnte ein Testprogramm in Rust erstellt werden, das auf eingebetteten Systemen ausführbar ist. Die anfänglichen Schwierigkeiten aufgrund des Fehlens einer CAN-Implementation für die Boards, konnten durch die Fremd-

funktionsschnittstelle von Rust umgangen werden. Durch die Community-Arbeit während der Arbeit konnte schlussendlich eine Rust-Implementation eines HALs verwendet werden. Das Testprogramm konfiguriert mit Hilfe des HALs die Uhren und die verwendete Peripherie, wie zum Beispiel den CAN-Controller der Mikrocontroller. Die Firmware konnte erfolgreich auf die Controller übertragen und darauf ausgeführt werden.

Mit dem Testprogramm und einer einfachen Implementation der portierten UAVCAN Bibliothek konnte eine Verbindung über den CAN-Bus zwischen zwei Mikrocontrollern aufgebaut werden und somit die Richtigkeit der Änderungen während der Portierung evaluiert werden.

Es wurde zudem eine Benchmark-Suite in Rust aufgebaut, die die Entwicklung der Bibliothek unterstützen kann. Hierbei konnte eine Möglichkeit gefunden werden, das System derartig generisch aufzubauen, dass die Messungen auf verschiedenen Zielsystemen, wie zum Beispiel eingebettete Systeme, durchgeführt werden können. Im Laufe der Arbeit konnten in Kapitel 5.4 die Messergebnisse der Benchmark-Suite auf ihre Richtigkeit überprüft werden. Zum Ende der Arbeit wird die Benchmark-Suite noch nicht aktiv in der Entwicklung eingesetzt.

In Kapitel 5 wurden erfolgreich Zeit- und Speichermessungen durchgeführt. Somit konnte die UAVCAN Bibliothek anhand der Leistungsfähigkeit des ISO- 25010 Softwareproduktqualitätsbaums evaluiert werden. Bei den Zeitmessungen wurden Software-Only Messungen durchgeführt, um die API Funktionsaufrufe zu messen. Hierbei wurden die genauen Messabläufe in Flussdiagrammen aufgezeigt und auf Einflüsse der Bibliotheks-Implementationslogiken, wie zum Beispiel das Verwenden des Heaps, eingegangen. Als Referenz zu den Rust-Implementations Messungen wurde eine in C++ geschriebene UAVCAN-Implementation gemessen. Somit konnten die Messergebnisse der Rust Bibliothek eingeordnet und auf Vor- und Nachteile der Implementation eingegangen werden.

Die Messungen haben gezeigt, dass die Rust Bibliothek bei der Ausführungsgeschwindigkeit auf dem Stand der Arduino-Implementation ist. Die in Rust geschriebene Bibliothek konnte sogar um einiges bessere Messergebnisse erzielen. Die Zeitersparnisse basieren zum einen auf den verschiedenen Implementationen, wobei die Verwendung des dynamischen Speichers auch ausschlaggebend ist, zum anderen auf der Sprache Rust und einer besseren Optimierung des Rust Compilers. Zudem konnten Optimierungen an der UAVCAN Rust Bibliothek vorgenommen werden, die die Ausführungszeiten weiter verbessert haben.

Des Weiteren wurden Zeitmessungen eines Szenarios mit dem CAN-Bus durchgeführt. Dabei wurde ein Mikrocontroller als Echo-Knoten eingesetzt, der empfangene Daten kopiert und zurücksendet. Für Referenzmessungen wurde die Software für den Echo-Knoten zusätzlich in C++ geschrieben.

Für dieses Szenario wurde die Kommunikation zuerst theoretisch betrachtet und mögliche Messwerte basierend auf den vorherigen Software-Only Messungen und auf theoretischen Längen einer CAN-Nachricht berechnet. Anschließend wurden die gemessenen Werte den theoretisch berechneten Werten gegenübergestellt. Insgesamt fiel auf, dass der Nachrichtenaustausch auf dem CAN-Bus die meiste Zeit in Anspruch nimmt und somit die

Ausführungszeiten der Software nahezu vernachlässigt werden können. Daher wurde zusätzlich mit einem Logic-Analyser die Latenzzeit des Echo-Knotens gemessen. Hierbei ist der Unterschied zwischen der Rust- und der Arduino-Implementation als Referenz deutlich sichtbar geworden. Die Rust-Implementation benötigt aufgrund der Umsetzung der Funktionen in der Bibliothek deutlich weniger Zeit, was auf eine gute Umsetzung für dieses Szenario schließen lässt.

Während der Speicherauslastungs-Messungen wurde der Echo-Knoten in Rust und in C++ sowohl nach der Kompilierung, als auch bei der Ausführung analysiert. Dies führte zu einem Vergleich zwischen den Implementationen der UAVCAN Bibliothek und der verwendeten Programmiersprachen. Insgesamt benötigt die Rust-Implementation um einiges weniger Heap Speicher bei der Ausführung. Zudem sind die Segmente im Speicher der Rust Firmware deutlich kleiner. Dies wurde auf weniger globale Variablen und auf eine speichereffizientere Kompilierung zurückgeführt. Die Stack-Auslastung war bei Rust höher als bei C++. Dies beruht darauf, dass in Rust viele Funktionen in übergeordnete Funktionen eingebettet werden. Somit werden die Funktionen größer und benötigen mehr lokalen Speicher. Der Vorteil davon ist, dass dabei weniger Aufrufstapelwechsel gemacht werden müssen und somit weniger Instruktionen benötigt werden.

Insgesamt wurde ersichtlich, dass der in Rust geschriebene Echo-Knoten deutlich weniger statischen und dynamischen Speicher benötigt. Dies liegt zum einen an der Implementation der UAVCAN Bibliothek und zum anderen an der Programmiersprache und wie die Compiler den Quellcode kompilieren.

In der Arbeit konnten gute Erkenntnisse aus den Messungen hinsichtlich der Fragestellung gezogen werden, ob sich Rust für die Entwicklung eines Kommunikationsstacks auf eingebetteten Systemen eignet. Es gab kaum Messergebnisse, die nicht entweder der Umsetzung in der Bibliothek oder der Sprachen zugeordnet werden konnten. Somit sind die Messungen in sich abgeschlossen.

Zudem konnten die Änderungen an der UAVCAN Rust Bibliothek vollständig in die aktuelle Entwicklung eingearbeitet und somit das Entwicklungsziel, die Portierung auf ein eingebettetes System, zu einem erfolgreichen Abschluss gebracht werden.

6.2 Fazit

Die Arbeit hat gezeigt, dass die Programmiersprache Rust zur Entwicklung des Kommunikationsstack des UAVCAN Projekts auf einem eingebetteten System verwendet werden kann.

Die zur Verfügung stehenden Werkzeuge zur Entwicklung mit der Sprache Rust erlauben einen unkomplizierten Einstieg in die Programmierung. Die kontinuierliche Weiterentwicklung der Sprache durch die Rust Teams und das ausgeklügelte Update-System führen zu einer schnellen Implementation von neuen Sprachfunktionen und zu einer stabilen Sprache. Es besteht die Möglichkeit, einer effizienten Programmierung durch Verwendung von

Bibliotheken, die die Rust-Community Open-Source entwickelt. Eine gute Dokumentation und Anwendungsbeispiele von Bibliotheken vereinfachen die Entwicklung mit Rust.

Bei der Entwicklung von Software für eingebettete Systeme ist der Support der Rust Community durchaus nützlich. Vor der Entwicklung mit der Sprache auf einem Mikrocontroller muss der Support für die Zielarchitektur geprüft werden. Es sind bereits etliche Unterstützungs-Bibliotheken vorhanden. Bei neu erschienen Boards kann in Rust entweder auf den Support durch die Community abgewartet, auf eigene Implementationen eines HALs gesetzt oder die Support-Bibliothek des Herstellers, meistens in C geschrieben, über die Fremdfunktionsschnittstelle der Sprache Rust eingebunden werden. Aufgrund der aufgezählten Möglichkeiten und der schnellen Community Entwicklung gibt es kaum Support-Probleme.

Die analysierten Softwareanforderungen für eingebettete Systeme aus Kapitel 2.1 kann Rust gänzlich abdecken. Das Typsystem, das Programmier-Muster und Abstraktionen über Funktionalität zulässt und die Möglichkeit bietet, den Rust Code in Module zu strukturieren, macht die entwickelte Software wartbar und verständlich. Hilfreich ist außerdem eine einfache Verwaltung der Abhängigkeiten eines Programmes mit Angaben von Bibliotheks-Versionen.

Rust bietet zudem die Möglichkeit, die vorhanden Ressourcen eines Systems gezielt und effizient auszunutzen. Ein bedeutender Faktor hierbei ist, dass ein Rust Programm zu Maschineninstruktionen kompiliert wird. Somit muss der Quellcode nicht zur Laufzeit ausgewertet werden und kann bei der Kompilierung bestmöglich auf Geschwindigkeit und Speicherverbrauch optimiert werden.

Außerdem erlaubt die Sprache neben Abstraktionsmöglichkeiten auch den direkten Speicherzugriff mit Zeigern. Dies ermöglicht eine gezielte Speicherverwaltung auf Systemen mit einem geringen Speicherbereich.

Ein weiterer Vorteil bei Rust ist die statische Analyse des Programms bei der Kompilierung. Fehler bei der Speicherverwaltung können durch eingeführte Programmierregeln, wie zum Beispiel der Eigentümerschaft, vor einer Ausführung aufgezeigt werden. Dies ist besonders vorteilhaft bei der Entwicklung für eingebettete Systeme. Die besagten Fehler können bereits vor einer Übertragung auf den Mikrocontroller und der Ausführung der Software behoben werden. Außerdem veranlasst das Einhalten der Programmierregeln eine umfänglichere Planung der Softwarearchitektur und Programmlogik, das einerseits eine längere Planung der Software bedeutet, andererseits zu einem verständlichem Code führt. Die Speichersicherheit wird somit erzielt.

Die Entwicklung mit der Programmiersprache Rust gestaltet sich als eine Zusammenarbeit mit dem Rust Compiler. Dieser ermittelt Fehler im Programm und weist den Programmierer mit verständlichen Fehlermeldungen darauf hin. Zusätzlich führt der Compiler Möglichkeiten zur Fehlerbehebung an.

Bei der Portierung der UAVCAN Rust Bibliothek auf ein eingebettetes System ist die Bedeutung der Modularität und der generischen Programmierung der Sprache ersicht-

lich geworden. Somit können Änderungen am Programmcode gezielt angegangen werden. Abhängigkeiten zu Betriebssystemen lassen sich ohne großen Mehraufwand zu speziellen Implementationen, die für eingebettete Systeme nötig sind, überführen. Dies ermöglicht nahezu dieselbe Entwicklung auf eingebetteten Systemen und Systemen mit Betriebssystem.

Die Zeit- und Speichermessungen haben gezeigt, dass Rust keine Nachteile für die Ausführung der Bibliothek mit sich bringt. Nach Optimierungen an der Speicherverwaltung und der Bibliothekslogik hat die Rust-Implementation bei allen Messungen gleich gute, wenn nicht bessere Ergebnisse im Vergleich zur Arduino-Implementation erzielen können. Die Messwerte ergaben, dass die in Rust geschriebene Software ebenso wie die in C++ geschriebene Software für den eingebetteten Bereich in Richtung begrenzter CPU Leistung sowie begrenzter Speicher einsetzbar ist.

Bei den Messungen war die Grenze zwischen den Einflussfaktoren der Umsetzung des UAVCAN Standards und der verwendeten Programmiersprache nicht immer eindeutig. Dennoch wurde eine bestmögliche Separierung mithilfe weiterer Information basierend auf Analysen und Tests versucht.

Schlussendlich kann gesagt werden, dass die Kombination von der Sprache Rust und der Implementation in der Sprache aufgrund der guten Messergebnisse, des ausgeprägten Ökosystems und guter Sprachfunktionalitäten auf eingebetteten Systemen weiterverfolgt werden soll.

6.3 Ausblick

Die Entwicklung der Rust UAVCAN Bibliothek ist zum Ende der Arbeit nicht abgeschlossen. Daher wäre es von Vorteil, die entwickelte Benchmark-Suite in den Entwicklungsprozess mit einzubauen, um zu überprüfen, ob Änderungen an der Bibliothek Einfluss auf die Ausführungszeit haben. Des Weiteren könnte durch die Ausführung der Benchmark-Suite auf verschiedenen Plattformen herausgefunden werden, ob geforderte Zeiten eingehalten werden können.

Zudem können weitere Szenarien zu der Benchmark-Suite hinzugefügt werden. Dazu kann zum Beispiel das Messen von Senden und Empfangen von Übertragungen in höheren Schichten des Schichtenmodells gehören. Des Weiteren können Szenarien hinzugefügt werden, bei denen mehrere Übertragungen gleichzeitig empfangen werden. Außerdem können weitere Transportprotokolle, wie zum Beispiel CAN FD und UDP, evaluiert werden.

Für die weiteren Messungen muss jedoch die Rust UAVCAN Bibliothek weiterentwickelt werden. Dazu gehört das Ausbauen der höheren Schichten, wie Repräsentationsschicht und Applikationsschicht. Außerdem muss der Support für weitere Transportprotokolle erweitert werden.

Zudem muss weiterhin die Entwicklung mit der Sprache Rust auf eingebetteten Systemen

in der Industrie beobachtet werden. Rust wird von großen Tech-Konzernen wie Microsoft, Google, Amazon, Dropbox und Facebook bereits zunehmend eingesetzt [52]. Jedoch fordern laut [12] missions- und sicherheitskritische Industrien einen sehr langfristigen Support, Qualifizierungspakete, branchenspezifische Werkzeuge, Verifikationshilfen, wie zum Beispiel der Verifikation des Rust-Compilers auf einen korrekten, erzeugten Maschinencode, Schulungen und branchenspezifische Beratung bezüglich der Sprache. Ferrous Systems, ein Unternehmen, das jeglichen Support, wie zum Beispiel Schulungen für die Programmiersprache bereit stellt [46], gab hierfür im Februar 2021 das Projekt Ferrocene[12] bekannt. Zur Zeit dieser Arbeit und darüber hinaus wird in diesem daran gearbeitet, diese Anforderungen für die Entwicklung in sicherheitskritischen Bereichen abzudecken. Das langfristige Ziel dabei ist, den Status-Quo der Softwarequalität und Korrektheit in diesen Bereichen mit der Verwendung der Sprache Rust zu verbessern [12].

Literatur

[1] arm. *ARMv7-M Architecture Reference Manual*. DDI0403E.e. URL: https://developer.arm.com/documentation/ddi0403/ee (besucht am 05. 11. 2021).

[2] Michael Barr. *Real men program in C*. 1. Aug. 2009. URL: https://www.embedded.com/real-men-program-in-c/ (besucht am 16. 12. 2021).

[3] Juliane T. Benra und Wolfgang A. Halang. *Software-Entwicklung für Echtzeitsysteme*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-01596-0.

[4] Arne Brehmer und Rick Lotoczky. „CAN based protocols in avionics“. In: *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*. IEEE, 5. Okt. 2014. DOI: 10.1109/dasc.2014.6979561.

[5] Joseph P. Cavano und James A. McCall. „A framework for the measurement of software quality“. In: ACM Press, 1. Jan. 1978. DOI: 10.1145/800283.811113.

[6] UAVCAN Consortium. *UAVCAN*. 1. Jan. 2021. URL: https://uavcan.org/ (besucht am 29. 09. 2021).

[7] Oracle Corporation. *What is a Data Race?* 1. Jan. 2010. URL: https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html (besucht am 28. 11. 2021).

[8] Team CratesIo. *The Rust community's crate registry*. URL: https://crates.io/ (besucht am 26. 11. 2021).

[9] Rust Embedded. *The Embedded Rust Book*. 6. Okt. 2021. URL: https://docs.rust-embedded.org/book/interoperability/c-with-rust.html (besucht am 07. 10. 2021).

[10] *Embedded software*. 11. Okt. 2021. URL: https://en.wikipedia.org/wiki/Embedded_software (besucht am 16. 12. 2021).

[11] *Embedded system*. 7. Dez. 2021. URL: https://en.wikipedia.org/wiki/Embedded_system (besucht am 16. 12. 2021).

[12] Florian. *Ferrocene Part 3: The Road to Rust in mission- and safety-critical*. 24. Feb. 2021. URL: https://ferrous-systems.com/blog/ferrocene-update-three-the-road/ (besucht am 19. 01. 2022).

[13] Sebastian Gerstl. *Grundlagen des Embedded Software Engineering*. 11. Aug. 2019. URL: https://www.embedded-software-engineering.de/grundlagen-des-embedded-software-engineering-a-855039/ (besucht am 29. 11. 2021).

[14] Balaji Gunasekaran. *9 Essential Microcontroller Peripherals Explained*. 7. Apr. 2020. URL: https://embeddedinventor.com/9-essential-microcontroller-peripherals-explained/ (besucht am 16. 12. 2021).

[15] *Hardware abstraction*. 17. Nov. 2021. URL: https://en.wikipedia.org/wiki/Hardware_abstraction (besucht am 30. 11. 2021).

[16] Nat Hillary. „Measuring Performance for Real-Time Systems". English. Website of the company. 1. Nov. 2005. URL: https://www.nxp.com/docs/en/white-paper/CWPERFORMWP.pdf (besucht am 02. 11. 2021).

[17] *In-Circuit Debuggers (Programmer-Debuggers)*. © 2021 Microchip Technology, Inc. URL: https://microchipdeveloper.com/tls0101:in-circuit-debuggers (besucht am 30. 11. 2021).

[18] japaric. *flip-link*. Hrsg. von BriocheBerlin. 7. Sep. 2020. URL: https://github.com/knurling-rs/flip-link/commits/main/README.md (besucht am 06. 12. 2021).

[19] Prof. Reinhard Keller. „Weitere Systeme Embedded Communication". 1. März 2021.

[20] Steve Klabnik. *The Story of Rust*. 8. Aug. 2016. URL: http://steveklabnik.github.io/history-of-rust/ (besucht am 27. 09. 2021).

[21] Sebastian Kosch. „CAN". 23. Okt. 2007. URL: https://ess.cs.tu-dortmund.de/Teaching/PGs/autolab/ausarbeitungen/Kosch-CAN-Ausarbeitung.pdf (besucht am 28. 09. 2021).

[22] Clara Lange. „Softwarequalitätsmodelle". Forschungsber. Technische Universität München, 1. Jan. 2015. URL: https://wwwmatthes.in.tum.de/file/6ikz2i550193/sebis-Public-Website/-/Proseminar/Lange-Qualitaetsmodelle-Ausarbeitung.pdf (besucht am 02. 11. 2021).

[23] MIT Licence. *The MIT License (MIT)*. URL: https://mit-license.org/ (besucht am 29. 09. 2021).

[24] LLVM-Team. *Auto-Vectorization in LLVM*. LLVM Compiler Infrastructure. 30. Nov. 2021. URL: https://llvm.org/docs/Vectorizers.html (besucht am 30. 11. 2021).

[25] A. Munir, S. Ranka und A. Gordon-Ross. „High-Performance Energy-Efficient Multicore Embedded Computing". In: *IEEE Transactions on Parallel and Distributed Systems* 23.4 (29. Juli 2011), S. 684–700. DOI: 10.1109/tpds.2011.214.

[26] Oracle. *Java Garbage Collection Basics*. Oracle. URL: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (besucht am 25. 11. 2021).

[27] Lazaros Papadopoulos u. a. „Interrelations between Software Quality Metrics, Performance and Energy Consumption in Embedded Applications". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. ACM, 2018. DOI: 10.1145/3207719.3207736.

[28] Joe Ranweiler. *Liar*. 4. Apr. 2018. URL: https://github.com/ranweiler/liar (besucht am 23. 12. 2021).

[29] *Rust Dokumentation*. Version 1.56.1. 1. Nov. 2021. URL: https://doc.rust-lang.org/stable/std/ (besucht am 16. 11. 2021).

[30] Rust Team. *Const generics MVP hits beta!* 26. Feb. 2021. URL: https://blog.rust-lang.org/2021/02/26/const-generics-mvp-beta.html (besucht am 19. 11. 2021).

[31] Rust Team. *Crate alloc*. URL: https://doc.rust-lang.org/alloc/ (besucht am 15. 12. 2021).

[32] Rust Team. *Crate core*. URL: https://doc.rust-lang.org/core/ (besucht am 15. 12. 2021).

[33] Rust Team. *Embedded devices*. URL: https://www.rust-lang.org/what/embedded (besucht am 23. 11. 2021).

[34] Rust Team. *Frequently Asked Questions*. 6. Dez. 2018. URL: https://prev.rust-lang.org/en-US/faq.html (besucht am 27. 09. 2021).

[35] Rust Team. *The Cargo Book*. 22. Okt. 2021. URL: https://doc.rust-lang.org/cargo/ (besucht am 26. 11. 2021).

[36] Rust Team. *The Rust Programming Language*. URL: https://doc.rust-lang.org/stable/book/ (besucht am 27. 09. 2021).

[37] Rust Team. *The Rust RFC Book. Rust RFCs - Active RFC List*. Hrsg. von Eric Huss. 29. Sep. 2021. URL: https://rust-lang.github.io/rfcs/ (besucht am 27. 09. 2021).

[38] Rust Team. *The rustc book*. Englisch. URL: https://doc.rust-lang.org/rustc/what-is-rustc.html (besucht am 05. 11. 2021).

[39] Supriya Saxena. *Difference between Computer and Embedded System*. 16. Juni 2020. URL: https://www.geeksforgeeks.org/difference-between-computer-and-embedded-system/ (besucht am 16. 12. 2021).

[40] Julia Schmidt. *Graydon Hoare im Interview zur Programmiersprache Rust*. 12. Juli 2013. URL: https://www.heise.de/developer/artikel/Graydon-Hoare-im-Interview-zur-Programmiersprache-Rust-1916345.html (besucht am 27. 09. 2021).

[41] Pavel Kirienko Scott Dixon. *The UAVCAN Guide*. 13. Juli 2020. URL: https://forum.uavcan.org/t/the-uavcan-guide/778 (besucht am 29. 09. 2021).

[42] ST. *STM32G4 Nucleo-64 boards (MB1367). User manual*. 4. Aufl. STMicroelectronics, 1. Feb. 2021, S. 44.

[43] Stackoverflow. *2021 Developer Servey. Programming, scripting, and markup languages*. 1. Jan. 2021. URL: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-language-love-dread (besucht am 26. 11. 2021).

[44] Gernot Starke. *Question C-1-2: What are quality goals (aka quality attributes)? | arc42 FAQ*. 1. Jan. 2019. URL: https://faq.arc42.org/questions/C-1-2/ (besucht am 22. 11. 2021).

[45] Erich Styger. *text, data and bss: Code and Data Size Explained*. 14. Apr. 2013. URL: https://mcuoneclipse.com/2013/04/14/text-data-and-bss-code-and-data-size-explained/ (besucht am 06. 12. 2021).

[46] Ferrous Systems. *Committed to Rustfor your peace of mind*. URL: https://ferrous-systems.com/ (besucht am 19. 01. 2022).

[47] John Teel. *Introduction to Microcontrollers. Discover what a microcontroller is, how it differs from a microprocessor, and the critical specifications you need to understand in order to pick the best one for your specific product application.* 30. Juni 2021. URL: https://predictabledesigns.com/introduction-to-microcontrollers/ (besucht am 16. 12. 2021).

[48] Team UAVCAN Development. *UAVCAN Specification v1.0-beta*. 1. Apr. 2021. URL: https://uavcan.org/specification/UAVCAN_Specification_v1.0-beta.pdf.

[49] Vector Informatik GmbH. *Einführung in CAN (DE) | CAN-Controller*. 22. Sep. 2021. URL: https://elearning.vector.com/mod/page/view.php?id=40 (besucht am 28. 09. 2021).

[50] Aseem Wangoo. *What's LLVM?* 22. Dez. 2020. URL: https://medium.datadriveninvestor.com/whats-llvm-4c0c3ed43a72 (besucht am 20. 12. 2021).

[51] *What is the difference between a Logic Analyzer and an Oscilloscope?* saleae. URL: https://blog.saleae.com/what-is-the-difference-between-a-logic-analyzer-and-an-oscilloscope/ (besucht am 30. 11. 2021).

[52] Philipp Zapf-Schramm. *Saar-Informatiker beweist Sicherheit der Programmiersprache Rust – und wird dafür international preisgekrönt*. 15. Juli 2021. URL: https://nachrichten.idw-online.de/2021/07/15/saar-informatiker-beweist-sicherheit-der-programmiersprache-rust-und-wird-dafuer-international-preisgekroent/ (besucht am 19. 01. 2022).