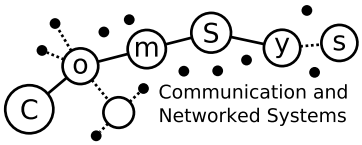
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Bachelorarbeit

Evaluierung der Sprache Rust zur Programmierung von Mikrocontrollern

Tom Heimbrodt

Matr. 217205

Betreuer: Prof. Dr. rer. nat. Mesut Güneş
Betreuender Assistent: M. Sc. Marian Buschsieweke

Institut für Intelligente Kooperierende System, Otto-von-Guericke-Universität Magdeburg

28. Mai 2019

Zusammenfassung

Zusammenfassung

Eingebettete Systeme werden heutzutage vor allem in den Sprachen C und C++ programmiert. In den letzten Jahren haben sich jedoch mehrere Alternativen entwickelt, von denen eine die Programmiersprache Rust ist. Deren Eignung im Kontext der Mikrocontrollerprogrammierung wurde im Zuge dieser Arbeit anhand der Kriterien Hardware-Unterstützung, Geschwindigkeit, Zuverlässigkeit und Wartbarkeit, sowie Produktivität überprüft. Die Ergebnisse zeigen, dass Rust bei der Unterstützung von Hardware und im Bereich der Geschwindigkeit C unterlegen ist, jedoch dafür in anderen Punkten zu bevorzugen wäre. Gerade bei der Zuverlässigkeit überzeugt die Sprache durch ihr einzigartiges Speichermodell, welches viele schwerwiegende Fehlerquellen schon zur Übersetzungszeit ausschließen kann. Dadurch könnte Rust eine geeignete Wahl bei der Programmierung von Mikrocontrollern sein und ihre Verbreitung in diesem Bereich in Zukunft zunehmen.

Abstract

Today, embedded systems are mainly programmed using the languages C and C++. However, during the last years multiple alternatives have been developed, among these the programming language Rust. Its suitability with respect to microcontroller software-development was evaluated using the criteria hardware support, performance, reliability and maintainability, as well as productivity. The result of this thesis is that Rust performs worse than C at hardware support and performance, but seems superior considering other aspects. It convinced especially at reliability, thanks to its unique memory model. This model is capable of eliminating many types of critical errors at compile time. In consequence Rust could be a suitable choice for programming microcontrollers and its usage in this field might increase in future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Akronyme

API Application Programming Interface.

CPU Central Processing Unit.

FFI Foreign Function Interface.

GPIO General-Purpose Input/Output.

HAL Hardware Abstraction Layer.

IDE Integrated Development Environment.
IETF Internet Engineering Task Force.
IoT Internet of Things.
ISA Instruction Set Architecture.
ISR Interrupt Service Routine.
I²C Inter-Integrated Circuit.

JIT-Compiler Just-in-Time Compiler.
JSON JavaScript Object Notation.
JVM Java Virtual Machine.

RAM Random Access Memory.
REPL Read-Eval-Print-Loop.
RFC Request for Comments.
ROM Read Only Memory.

SPI Serial Peripheral Interface.

USART Universal Synchronous and Asynchronous Serial Receiver and Transmitter.
USB Universal Serial Bus.

Glossar

Boilerplate Code Überflüssiger Code, der nicht direkt dazu beiträgt die Funktionalität umzusetzen, jedoch z. B. von der Programmiersprache erzwungen wird.

Borrow-Checker Ein Teil des Rust-Compilers welcher sicherstellt, dass Regeln, die die Veränderbarkeit und Lebensdauer von Referenzen betreffen, eingehalten werden.

Crate Eine Bibliothek, die von Rust-Code genutzt werden kann. Üblicherweise auf der Webseite `crates.io` veröffentlicht.

Dynamic Typing Eine Form eines Typ-Systems, bei eine Variable Werte beliebigen Typs halten kann.

Foreign Function Interface Schnittstellen, mit denen in anderen Programmiersprachen geschriebene Bibliotheken eingebunden werden können.

General-Purpose Input/Output Allgemeine (digitale) Ein- und Ausgänge des Mikrocontrollers.

Interior Mutability Ein Konzept welches die Veränderbarkeit von Daten nach außen kapselt, sodass Manipulationen auf einem eigentlich nur lesbaren Objekt durchgeführt werden können.

Interrupt Service Routine Methoden die zu bestimmten Zeitpunkten, wie einem Signaleingang, den Programmfluss unterbrechen und ausgeführt werden.

Lifetime-Specifier Anmerkung im Quelltext, welche die Lebensdauer des Objekts angibt. Überlicherweise mit den ersten Kleinbuchstaben des Alphabets bezeichnet.

LoRaWAN Funkstandard zur Übertragung kleiner Datenmengen mit wenig Energieaufwand über relativ große Entfernungen.

Memory Ownership Ein Programmierkonzept, in welchem jeder Wert bzw. jede Objektinstanz einem Besitzer zugeordnet ist.

Monade Ein Typkonzept aus der funktionalen Programmierung durch das Werte gekapselt werden und welches linear kombiniert werden kann.

Panic Ein üblicherweise fataler Laufzeitfehler in der Programmiersprache Rust, der den Abbruch der Ausführung zur Folge hat.

Polymorphismus Ein Konzept aus der objektorientierten Software-Enwicklung, durch das Objekte unterschiedlichen Typs, aber mit ähnlichen Merkmalen zusammengefasst werden können.

Race Conditions Ein undefiniertes oder nicht-deterministisches Verhalten, dass auftritt, wenn mehrere Ausführungsstränge konkurrierend auf Daten oder Resourcen zugreifen.

Read-Eval-Print-Loop Eine interaktive Laufzeitumgebung, in der Ausdrücke, ähnlich einer Shell, zeilenweise eingegeben und ausgewertet werden.

schwebende Zeiger (dangling Pointer) Zeiger auf Speicherregionen, die bereits freigegeben wurden.

Toolchain Zusammenfassung aller Werkzeuge, die zum Erstellen, Flashen und Testen von Programmen genutzt werden.

Tracing Garbage Collection Eine Form der automatischen Speicherverwaltung zur Laufzeit (Garbage Collector). Die Zugriffsmöglichkeiten auf jedes Objektes werden verfolgt, um es an geeigneter Stelle freigeben zu können.

Trait Eine Form der Schnittstellendefinition in Rust, vergleichbar mit Interfaces oder Type-Classes in anderen Sprachen.

Universal Synchronous and Asynchronous Serial Receiver and Transmitter Serielle Kommunikationsschnittstelle für Mikrocontroller, die sowohl asynchron, als auch synchron, also mit zusätzlichem Taktsignal, betrieben werden kann.

KAPITEL 1

Einleitung

In der allgemeinen Softwareentwicklung spielen die Sprachen C und C++ oder gar Assembly eine zunehmens geringere Rolle und werden durch modernere Sprachen verdrängt [1][2][3]. Im Gegensatz dazu ist bei der Entwicklung eingebetteter Systeme dieser Trend nur langsam zu beobachten [4][5]. Da sich C und C++ hier seit Jahren etabliert haben, mag es zunächst auch wenig profitabel erscheinen eine alternative Sprache zu nutzen. Im Gegensatz zu herkömmlichen PCs, bei denen beinahe unbegrenzt Speicher und Rechenleistung zur Verfügung stehen, ist die Ressourcennutzung auf Mikrocontrollern ein kritisches Thema. So muss oft auf eine komplexe Abstraktionsschicht verzichtet werden, die viele der modernen und aufstrebenden Sprachen nutzen.

Zu diesen hilfreichen, aber während der Ausführung zusätzlichen Aufwand benötigenden Funktionen zählen beispielsweise Interpreter bzw. Just-in-Time Compiler (JIT-Compiler), Tracing Garbage Collection, Dynamic Typing und Out-Of-Bounds-, sowie Null-Zeiger-Überprüfungen zur Laufzeit. Viele der in der konventionellen Softwareentwicklung am meisten genutzten Sprachen, zu denen unter anderem Java, C♯, Javascript und Python zählen, bedienen sich fast aller dieser Funktionen. Sie gehen also davon aus, dass der Gewinn an Zuverlässigkeit und Produktivität bei der Entwicklung stärker wiegt, als der damit verbundene Mehraufwand zur Laufzeit.

1.1 Motivation

Wäre es möglich einige dieser Funktionen auch auf die Programmierung eingebetteter Systeme zu übertragen oder gleichwertige Alternativen zu ihnen zu finden, so wäre es denkbar, dass hier ebenfalls von positiven Effekten profitiert werden könnte. Diese sind insbesondere wünschenswert, da der Bereich der eingebetteten Systeme ein nicht vernachlässigbarer Zweig der Informatik und Elektrotechnik ist, dessen Umsatz jährlich ansteigt [6][7].

Zu den möglichen positiven Effekten zählt unter anderem eine gesteigerte Produktivität der Entwickler. Viele moderne Sprachen bieten beispielsweise eine automatische Speicherverwaltung, sodass die Aufgabe, Objekte zum richtigen Zeitpunkt freizugeben, systematisch und fehlerfrei übernommen wird. Diese fällt also nicht mehr an den Programmierer. Zusätzlich entfällt die Zeit, die benötigt wird, falls schwer zu findende oder zu beseitigende Probleme

beim Zugriff oder der Freigabe von Speicherregionen auftreten.

Weiterhin bieten viele dieser Sprachen zusätzliche Konstrukte, die zwar in Sprachen wie C auch realisierbar sind, jedoch meist mühevoll und sehr langatmig nachgestellt oder eigenhändig für einen bestimmten Verwendungszweck implementiert werden müssen. Zu diesen zählen neben vielen weiteren Iteratoren, Vererbung, Schnittstellen (Interfaces, Traits oder Type Classes), andere objektorientierte Merkmale, sowie Lambda-Ausdrücke bzw. anonyme Funktionen.

Diese ermöglichen meist nicht nur eine höhere Produktivität beim Erstellen von Programmen, sondern durch die Reduktion von Boilerplate Code und durch die bessere Strukturierung des Programmcodes durch oben genannte Mittel eine höhere Wart-, Erweiter- und Wiederverwendbarkeit.

Schließlich kann durch eine automatische Speicherverwaltung die Fehleranfälligkeit von Programmen reduziert werden. Dazu trägt ebenfalls eine bessere Unterstützung für einheitliche und sinnvolle Behandlung von Laufzeitfehlern bei, wie Exceptions in Java und C♯ oder die Result-Struktur in Rust. Dies ist für viele eingebettete Systeme von höchster Wichtigkeit, da diese teils Jahre ununterbrochen laufen, schwer zu warten und auszutauschen sind oder gar sicherheitskritische Vorgänge steuern oder überwachen.

1.2 Überblick einiger ausgewählter Programmiersprachen

Aus einigen dieser genannten Sprachen haben sich Ableger entwickelt, die für leistungsfähigere Mikrocontroller kompiliert werden können. Hier kann beispielsweise das Projekt MicroPython genannt werden, welches in Abschnitt 1.3 näher vorgestellt wird. Dies hat den Vorteil, dass mit der selben Sprache sowohl allgemeine Softwaresysteme, als auch Steuerungen eingebetteter Geräte entworfen werden können. So könnten Ressourcen beider Bereiche eventuell zum Vorteil aller kombiniert werden. Da die Kosten für eingebettete Systeme dazu stetig sinken [8], während ihre Leistungsfähigkeit weiter ansteigt [9], zeigen diese Sprachen ein nicht zu vernachlässigendes Zukunftspotenzial.

Daneben scheint es, abseits von C/C++, kaum Sprachen zu geben, die explizit einen Fokus auf Hardwarenähe oder gar Mikrocontroller legen. Eine der wenigen bekannten und moderneren Sprachen auf diesem Gebiet ist Rust, die zwar als Allzweck-Sprache entworfen wurde, aber dabei auch einen Fokus auf die Programmierung eingebetteter Geräte legt [10].

Im Folgenden sollen ein paar ausgewählte Sprachen und ihre Eckdaten vorgestellt, sowie ihr Potenzial im Kontext der Mikrocontroller-Programmierung kurz eingeschätzt werden. Dabei handelt es sich um eine oberflächliche Betrachtung, die lediglich verdeutlichen soll, wieso die Programmiersprache Rust, neben den vielen alternativen, nicht beachteten Sprachen, als Thema dieser Arbeit behandelt wird.

1.2.1 C

Zunächst hat sich die Programmiersprache C [11] schon seit Jahren in der Entwicklung eingebetteter Systeme etabliert [4]. Sie kann auf jedem praxisrelevanten Gerät eingesetzt werden, auch weil die meisten herstellerspezifischen Entwicklungswerkzeuge auf C aufbauen. Darüber hinaus ist das Binärformat Cs die Grundlage für Foreign Function Interfaces

(FFIs) anderer Sprachen, um Implementierungen unterschiedlicher Programmiersprachen zu verbinden.

Durch den produktiven Einsatz haben sich jedoch auch einige Schwach- und Kritikpunkte offenbart. Einer der schwerwiegendsten davon ist die manuelle Verwaltung und Manipulation von Speicheradressen und -regionen. Auf das Dereferenzieren von schwebenden Zeigern (dangling Pointer) oder von Zeigern auf nicht allozierte Speicherbereiche, zum Beispiel durch einen Buffer-Overflow, lassen sich viele Abstürze oder sogar Sicherheitslücken zurückführen [12][13]. Dazu kommen Speicherleaks durch nicht freigegebene Speicherbereiche. Durch Programme wie Valgrind [14] können diese zwar zur Laufzeit erkannt werden, einen garantierten Schutz vor solchen Fehlern kann bei der Sprache C jedoch nie gewährleistet werden.

Zusätzlich ist der Funktionsumfang der Sprache trotz stetiger Weiterentwicklung und Anpassung des C-Standards, wie durch `ISO/IEC 9899:2018` (C18) [15], relativ begrenzt, was die Umsetzung von objektorientierten oder funktionalen Paradigmen nur über Umwege möglich macht.

1.2.2 C++

C++ [16] ist größtenteils eine Erweiterung des C-Standards um die begrenzte Funktionalität auszubessern. Insbesondere liegt ein Fokus der Sprache auf der Möglichkeit zur objektorientierten Programmierung [17][18].

Da hier ebenfalls eine manuelle Speicherverwaltung erfolgen muss, bleiben viele der Kritikpunkte an C bestehen. Diese wurden allerdings teilweise in Überarbeitungen am C++-Standard adressiert. So wurden mit der Version C++11, die Smart-Pointer `unique_ptr`, `shared_ptr` und `weak_ptr` Teil der Standardbibliothek [19]. Diese ermöglichen bei entsprechender Verwendung das automatische Freigeben des referenzierten Speichers, sofern der Zeiger selbst freigegeben wird. Letzteres kann beispielsweise durch Deklaration als Stack-Variable umgesetzt werden.

Die Fülle und Komplexität der Sprachfunktionen C++'s kann teilweise auch als negativ betrachtet werden. So haben viele modernere objektorientierte Programmiersprachen, die sich wie C♯ und Java nach C++ entwickelt haben, Konzepte vereinfacht oder weiterentwickelt. Beispielsweise wurden in diesen mehrfache Vererbung durch Schnittstellen, sowie Turing-vollständige [20] Templates durch generische Typen oder Funktionen ersetzt. Letztere erlauben u. a. keine Verallgemeinerung über Ganzzahlen, Template Specialization oder beliebig viele Argumente (Variadic Templates).

In der Mikrocontrollerprogrammierung wird C++ ebenfalls als bewährte Sprache betrachtet und ist deshalb auf den meisten Plattformen nutzbar [4]. Dies liegt unter anderem daran, dass Hardware-Abstraktionen, welche für C geschrieben wurden, auch problemlos aus C++ nutzbar sind. Es wird also lediglich ein C++-fähiger Compiler benötigt.

1.2.3 Java

Java [21] ist eine im Vergleich zu den beiden zuvor vorgestellten Sprachen relativ moderne Programmiersprache, die ebenfalls einen Fokus auf objektorientierte Programmierung

legt. Sie ist eine der ersten erfolgreichen Sprachen, die statt direkt zu ausführbarem Code zu kompilieren zunächst einen platformunabhängigen Bytecode nutzt. Dies setzt voraus, dass auf der Zielplattform eine Implementierung der Java Virtual Machine (JVM) installiert ist, die diesen Code dann interpretiert oder zur Laufzeit in Maschinencode kompiliert (JIT-Compiler). In der JVM ist ebenfalls ein Garbage Collector enthalten, der zusammen mit Überprüfungen zur Laufzeit und der Abwesenheit von Zeigern die zuvor genannten Fehlerquellen aus C/C++ ausschließt.

Die Möglichkeit der Programmierung eines Mikrocontrollers mit Java scheint also erstrebenswert. Allerdings stellt die Implementierung der JVM inklusive JIT-Compiler und Garbage Collector sehr hohe Anforderungen an die Hardware, die viele Mikrocontroller nicht erfüllen können.

Außerdem werden an einige eingebettete Systeme Echtzeitanforderungen gestellt. Diese sind durch die Nutzung eines Garbage Collectors schwieriger zu erfüllen, da dieser zu bestimmten Zeiten die Programmausführung pausieren muss, um nicht mehr benötigte Objekte zu finden und freizugeben. In einigen Fällen könnten ebenfalls jegliche Heap-Allokationen unerwünscht sein. Da diese jedoch nicht explizit, sondern bei der Instanziierung eines Objektes durchgeführt werden, wäre dieser Anwendungsfall auch schwierig in Java umzusetzen.

1.2.4 Python

Eine weitere interessante Programmiersprache ist Python [22]. Diese ermöglicht zwar ebenfalls objektorientierte Programmierkonstrukte, legt aber einen Fokus auf Einfachheit und Produktivität. Dies kann beispielsweise an der dynamischen Typisierung oder an der simplen Syntax, die unter anderem die Einrückung des Codes auswertet, festgemacht werden.

Als Vertreter der Skriptsprachen können Python-Implementierungen aus Quelltexten interpretiert werden. Dieser kann allerdings auch, ähnlich wie Java, zu Bytecode kompiliert werden. Analog zu den für Java genannten Gründen können auch hier keine Speicherfehler auftreten.

Python ist ebenfalls stark in der allgemeinen Softwareentwicklung verbreitet [2] und scheint als Sprache auch das Potenzial zur Implementierung von Steuerungen auf Mikrocontrollern zu besitzen. Die Möglichkeit Skripte direkt, eventuell in einer Read-Eval-Print-Loop (REPL) auszuführen scheint sehr hilfreich beim Entwickeln und Testen von Schnittstellen für Peripherie, wie Sensoren und Aktoren. Allerdings können die selben Probleme angemerkt werden, wie schon bei Java. Dabei fielen noch die zusätzlichen Anforderungen für das Entwickeln eines Interpreters und Ausführen einer REPL ins Gewicht.

1.2.5 Haskell

Haskell ist ein Vertreter der funktionalen Programmiersprachen [23][24]. Dadurch wird ein Fokus auf pure Funktionen ohne Seiteneffekte und innere Zustände gelenkt, deren Komplexität bei vielen imperativen und objektorientierten Sprachen zu Fehlerquellen führen kann. So müsste theoretisch die Funktionsweise jeder Methode bei jedem möglichen Zustand getestet werden.

Die zur Mikrocontrollerprogrammierung oft benötigte Interaktion mit Hardwarebausteinen

lässt sich jedoch schwer durch pure Funktionen beschreiben. Diese könnten wie andere I/O-Operationen in Haskell durch Monaden modelliert und mit der `do`-Notation vergleichsweise zu imperativen Code definiert werden.

Diese relativ komplexe Abstraktionsschicht erfordert jedoch eine steile Lernkurve für Programmierer, die noch nicht mit funktionaler Programmierung vertraut sind. Die grundsätzlichen Anforderungen an Mikrocontroller sind dagegen relativ gering. Allerdings können sich beim Ausführen funktionaler Programme starke Geschwindigkeits- oder Speicherprobleme zeigen, deren Optimierung sehr aufwändig werden kann. Dies liegt daran, dass viele in funktionalen Sprachen geschriebene Algorithmen in Anlehnung an mathematische Betrachtungen oft zunächst die gesamte Lösungsmenge in Betracht ziehen und diese anschließend nach tatsächlich korrekten Lösungen filtern.

1.2.6 Rust

Rust ist eine sehr junge Programmiersprache, die, wie auf ihrer Homepage zu lesen ist, einen Fokus auf Geschwindigkeit (Performance), Zuverlässigkeit (Reliability) und Produktivität (Productivity) setzt [10]. In der Sprache sind sowohl imperative und objektorientierte, als auch einige funktionale Einflüsse zu erkennen.

Rust scheint als eine der ersten verbreiteten Programmiersprachen Speichersicherheit nicht durch einen Garbage Collector, sondern durch das Prinzip des Speicherbesitzes (Memory Ownership) zu lösen. Dieser kann schon zur Übersetzungszeit ausgewertet werden. Darüber hinaus lässt sich dieses Prinzip nicht nur auf die Speicherverwaltung, sondern auch auf Probleme anwenden, die durch Multithreading entstehen. Dies ist im Kontext der Mikrocontrollerprogrammierung z. B. für Interrupt Service Routines (ISRs) relevant und wird von vielen anderen Sprachen nicht oder nur am Rande adressiert.

Falls diese Versprechen sich bewahrheiten und als praxistauglich herausstellen, scheint Rust also durch seinen sicheren und dennoch ressourcenschonenden Entwurf prädestiniert für den Einsatz auf Mikrocontrollern. Daher wurde beschlossen die Einsatzmöglichkeiten und Eignung Rusts in der eingebetteten Programmierung in dieser Arbeit näher zu betrachten und auszuwerten.

1.3 Verwandte Arbeiten

Zunächst gibt es einige Ansätze unterschiedliche Programmiersprachen – abseits von C bzw. C++ – zur Entwicklung auf Mikrocontrollern zu nutzen. Von diesen möchte ich hier im Folgenden MicroPython und eLua vorstellen.

MicroPython [25] ist ein Projekt, welches darauf abzielt, die bereits im vorherigen Abschnitt vorgestellte Sprache Python im Bereich der eingebetteten Systeme nutzen zu können. Dazu bieten die Entwickler ebenfalls eigene Platinen an, auf denen die Implementierung ausgeführt werden kann. Alternativ lassen sich jedoch auch andere, leistungsfähigere Mikrocontroller wie z. B. ein ESP32 damit steuern. Kleinere Geräte werden jedoch nicht unterstützt, da, wie in meiner vorherigen Analyse beschrieben, eine relativ umfangreiche Laufzeitbibliothek benötigt wird. So müssen laut der Homepage mindestens 256 KiB Flash-Speicher und 16 KiB RAM vorhanden sein. MicroPython kann insbesondere für Sensoraufgaben im

Bereich des Internets der Dinge (IoT) eingesetzt werden, wie Gregory anhand eines LoRaWAN-fähigen Temperatursensors [26] oder Kodali und Mahesh mit einer Ansteuerung eines DHT-Sensors über einen ESP8266 [27] zeigen.

Einen weiteren Versuch einer Sprachimplementierung für Mikrocontroller bildet das eLua Projekt [28], welches die Sprache Lua auf diesen lauffähig macht. Neben einem Interpreter für eingebettete Systeme ist es ebenfalls Ziel des Projektes zusätzliche Schnittstellen bereitzustellen, um die Hardwarekomponenten der Geräte sinnvoll anzusteuern. Lua ist, wie Python, eine Skriptsprache und besitzt daher ähnliche Vor- und Nachteile: Es ist auf der einen Seite sehr entwicklerfreundlich und fehlerresistent, opfert für diese Eigenschaften jedoch zumeist niedrige Hardwareanforderungen. So ist ebenfalls im eLua-Wiki zu lesen, dass mindestens 256 KiB ROM- und 64 KiB RAM-Speicher empfohlen werden, auch wenn diese für bestimmte Einsatzgebiete durch Optimierungen reduziert werden könnten [29].

Bei der Betrachtung dieser beiden Projekte scheint also der Trend in Richtung Skriptsprachen auf leistungsfähigeren Mikrocontrollern zu gehen. Sie zeigen zudem das deutliche Vorhandensein an einem Interesse an Sprachen abseits von C/C++. Auch wenn die Menge an verfügbarem Speicher und die Geschwindigkeit vieler Mikrocontroller in den vergangenen Jahren zugenommen haben, ließen diese Projekte dennoch eine Lücke in Bereich kleinerer Chips zurück. Darüber hinaus müssen die Implementierungen der Laufzeitumgebungen weiterhin mit herkömmlichen Sprachen durchgeführt werden. Ob Rust diese Lücke füllen kann, wird sich im Verlauf dieser Arbeit zeigen.

Die hier vorgestellte Erwägung zu ihrer Nutzung auf eingebetteten Systemen ist dabei nicht gänzlich neu. Zum einen ist Rust, dank der Arbeit vieler Mitglieder der Community, schon heute auf mehreren Geräten lauffähig und dank der Implementierung einiger Hardware-Abstraktionen auch sinnvoll nutzbar. Zum anderen sind diese Möglichkeiten aufgrund guter Dokumentation und Anleitungen einfach ausprobierbar. Diese finden sich beispielsweise in den Büchern „The Embedded Rust Book“ [30], „Discovery“ [31] oder „The Embedonomicon“ [32], sowie in der Dokumentation der einzelnen Bibliotheken.

Auch finden sich verschiedene wissenschaftliche Arbeiten, die sich mit dem Einsatz Rusts beschäftigen. Zum Beispiel stellen Rivera, Lindner und Lindgren in ihrer Arbeit „Heapless: Dynamic Data Structures without Dynamic Heap Allocator for Rust“[33] die Implementierung einiger Datenstrukturen, wie Listen, Zeichenketten oder Ringbuffer ohne Heapallokationen vor. Diese wurden in Rust implementiert und sind für das Einbinden aus anderem Rust-Code auf Mikrocontrollern konzipiert.

Darüber hinaus gibt es Ansätze Betriebssysteme für Mikrocontroller in der Sprache zu schreiben. Eines dieser Projekte ist Tock [34], laut eigener Beschreibung ein eingebettetes Betriebssystem, dass konzipiert wurde, um mehrere, sich gegenseitig nicht vertrauende Anwendungen gleichzeitig auf speicherarmen und energiesparenden Mikrocontrollern laufen zu lassen. (Im Original: An embedded operating system designed for running multiple concurrent, mutually distrustful applications on low-memory and low-power microcontrollers.) Dabei stellt es sich als einsetzbar beim Implementieren von Sensornetzwerken oder sicherheitskritischen Geräten, wie beispielsweise USB-Authentifizierungs-Sticks vor. Es kann auch genutzt werden, um IoT-Controller und Wearables, d.h. tragbare eingebettete Systemen wie Smartwatches umzusetzen.

Darüber hinaus hat Heldring untersucht, ob sich Rust zur Entwicklung eines echtzeitfähigen

Betriebssystems für eingebettete Systeme eignet [35]. Dafür hat er einen Scheduler in Rust implementiert und ihn mit der originalen C-Version verglichen, wobei er eine grundlegende Eignung Rusts im Bereich der Echtzeitsysteme festgestellt hat. Da solche Anforderungen an viele eingebettete Systeme, die z. B. Steuerungsaufgaben erfüllen sollen, gestellt werden, sind in diesem Bereich meist Skriptsprachen mit ungewissen Ausführungszeiten im Nachteil.

Eingebettete Betriebssysteme könnten jedoch auch die Nutzung von C attraktiver gestalten. Diese könnten beispielsweise Teile der Speicherverwaltung oder die Ansteuerung von Hardwarekomponenten im Hintergrund übernehmen und so die Entwicklung von Anwendungen vereinfachen. Hier könnte als Vertreter das Projekt tinyOS [36] genannt werden, welches überwiegend in nesC, einem Dialekt der Sprache C programmiert wurde. Es legt besonderen Fokus auf Geräte zur kabellosen Übertragung, z. B. für Sensornetzwerke, intelligente Gebäude und Messgeräte, sowie andere Anwendungen im IoT.

KAPITEL 2

Angefertigte Implementierungen und durchgeführte Experimente

Damit die Eignung Rusts als Programmiersprache für Mikrocontroller fundiert eingeschätzt werden kann, wurden zunächst einige Fragestellungen und Kriterien aufgestellt, die durch die angefertigten Implementierungen beantwortet bzw. eingeschätzt werden sollen.

Zunächst muss überprüft werden, ob es überhaupt möglich ist, nicht-triviale Programme für eingebettete Systeme in Rust zu entwickeln. Dies beinhaltet nicht nur das Ausführen des kompilierten Codes auf der entsprechenden CPU-Architektur, sondern ebenfalls die unverzichtbare Anforderung sinnvoll die verwendete Hardware anzusteuern. Zu diesen Funktionen zählen unter anderen die Konfiguration von General-Purpose Input/Output (GPIO), Digital-Analog- und Analog-Digital-Umwandler, Timer, Interrupts und Kommunikationsschnittstellen, wie beispielsweise USART. Darüber hinaus sollte die Sprache es ebenfalls ermöglichen, Steuerungen für Mikrocontroller mit sehr begrenzten Ressourcen, insbesondere die Menge an flüchtigen und nicht-flüchtigen Speicher betreffend, umzusetzen.

Die folgenden Kriterien basieren auf der Annahme, dass diese Frage größtenteils positiv beantwortet werden kann, da ansonsten der produktive Einsatz Rusts in der Mikrocontroller-Programmierung zum aktuellen Zeitpunkt ohnehin ausgeschlossen wäre.

Als weiteres Kriterium sollte betrachtet werden, wie produktiv mit Rust entwickelt werden kann. Da andere Sprachen schon seit vielen Jahren erfolgreich im Einsatz sind, wird nur die theoretische Möglichkeit Rust einzusetzen wenige Entwickler dazu anregen die Sprache zu wechseln bzw. eine neue, noch unvertraute Sprache einzusetzen. Ein entscheidender Vorteil wäre es allerdings, wenn es möglich wäre, schneller und effektiver Programme vergleichbarer Qualität zu erstellen. Zu diesem Punkt zählen unter anderem der Nutzen von vorhandenen Sprachfeatures, aber auch die Ausgereiftheit und Bedienfreundlichkeit der verwendeten Toolchain, sowie die Vermeidung und das Finden von Fehlern. Zu Beginn fließt hier außerdem der Aufwand ein, den die Einrichtung der Toolchain und das Aneignen der Sprache benötigen. In einigen Anwendungsfällen sind hier auch weitere Faktoren, wie die Dauer der Ausführung des Compilers zu beachten.

Schließlich muss für eine Bewertung zusätzlich die Effizienz beachtet werden, mit der in Rust geschriebene und kompilierte Algorithmen ausgeführt werden. Hier läge der theoretische Ide-

alwert bei speziell auf die CPU optimiertem Assembly, praktisch ist es allerdings sinnvoll die Messwerte mit in C geschriebenen Programmen zu vergleichen. Dies folgt zum einen aus der hoher Verbreitung der Programmiersprache C in der Mikrocontroller-Programmierung, zum anderen aus dem Grund, dass C als eine sehr hardwarenahe Sprache betrachtet werden kann, die ohne tiefe Abstraktionsschichten auskommt. Deshalb sollte hinreichend optimierter C-Code einem optimalen Ergebnis in vielen Fällen sehr nah kommen. Natürlich spielen in diesen Bereich auch die Optimierungsfunktionen des verwendeten Compilers eine nicht zu vernachlässigende Rolle.

Zuletzt muss untersucht werden, ob es zusätzliche Aspekte der Programmiersprache gibt, die den Einsatz generell oder in bestimmten Anforderungsgebieten attraktiver machen. Auf den ersten Blick fällt hier beispielsweise die Überprüfung bestimmter Arten von Speicherfehlern zur Übersetzungszeit auf. Dies könnte sowohl allgemein von Vorteil sein, aber insbesondere auch in Einsatzgebieten in denen eine maximale Ausfallsicherheit gewünscht wird ein relevanter Grund sein, Rust zum entwickeln eines Programms zu nutzen.

Um diese Kriterien bewerten zu können wurden im Zuge dieser Bachelorarbeit die nachfolgend beschriebenen Implementierungen angefertigt.

Dabei schien es zunächst sinnvoll ein Programm zu entwerfen, dass eine grundlegende Aufgabenstellung erfüllt, um die Eignung Rusts selbstständig an einem Projekt nachfolziehen zu können. Dafür wurde eine Messung der Umgebungstemperatur und -feuchtigkeit und die Darstellung dieser auf einer Sieben-Segment-Anzeige umgesetzt. Diese wurde sowohl in Rust als auch in C als Referenzimplementierung geschrieben, wie ich im nachfolgenden Unterkapitel 2.1 beschreiben werde.

Da dieses Programm jedoch nicht dafür geeignet ist, die Geschwindigkeitsunterschiede zur Laufzeit einzuschätzen, wurden anschließend einige Algorithmen speziell für diesen Zweck implementiert und ihre Ausführungszeiten gemessen. Dabei wurde der Fokus auf unterschiedliche Aspekte, wie Array-Iteration, Rekursion, numerische Berechnungen usw. gelegt. Analog zur vorherigen Implementierung wurden diese sowohl in Rust als auch in C umgesetzt. Sie werden im Kapitel 2.2 näher erläutert.

Die folgenden Beispiele wurden mit dem Rust-Compiler `rustc` der Nightly-Version 1.33.0 kompiliert, bzw. mit `arm-none-eabi-gcc` in Version 8.3.0 für den C-Code.

2.1 Temperatur- und Feuchtigkeitsmessung

Um die Vor- und Nachteile von Rust auf eingebetteten Systemen praktisch beurteilen zu können, sollte zunächst eine beispielhafte Aufgabenstellung bearbeitet werden. Diese bestand darin, die aktuelle Umgebungstemperatur und -feuchtigkeit zu messen und diese auf einer Anzeige darzustellen. Diese Anforderung wurde ausgewählt, da sie auf der einen Seite relativ simpel und ohne größere Vorarbeit umsetzbar ist, auf der anderen Seite allerdings die grundlegenden Ein- und Ausgabemechanismen des Mikrocontrollers abdeckt. Darüber hinaus besitzt sie eine gewisse Praxisrelevanz. Man könnte diese Steuerung beispielsweise sinnvoll in einen digitalen Wecker oder einen Kühlschrank integrieren.

Weitergehende Funktionen des Mikrocontrollers, wie Schnittstellen über USB, SPI, I²C usw. oder Analog-Digital-Konvertierung wurden hingegen nicht primär berücksichtigt. Die-

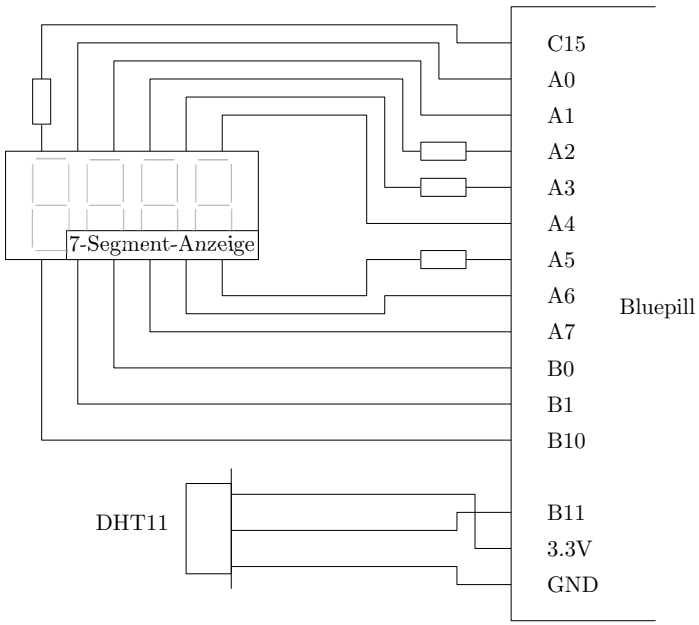


Abbildung 2.1: Schematischer Aufbau der Steuerung zur Temperatur- und Feuchtigkeitsmessung.

se würden allerdings ohnehin über die Register des Mikrocontrollers konfiguriert, deren Funktionsweise schon durch die GPIO-Ansteuerung getestet wird. Alternativ wäre die Ansteuerungen dieser erweiterten Funktionen durch eine abstrahierende Bibliothek denkbar. Deren Design würde sich allerdings auch nur zu einem kleinen Teil nach der verwendeten Programmiersprache richten. Darüber hinaus gibt es hier insbesondere für C mehrere Bibliotheken mit unterschiedlichen Abstraktionsgraden und Zielstellungen, wie die STM32 Low-Layer APIs oder STM32 Cube Hardware Abstraction Layer (HAL) [37].

Um die Anforderung umzusetzen wurde das STM32-F1 Entwicklungsboard Bluepill genutzt. Zum Messen der Temperatur und der Feuchtigkeit wurde zudem ein DHT11 Sensor und zum Anzeigen der Werte eine vierstellige Sieben-Segment-Anzeige verwendet. Der schematische Aufbau dazu ist in Abbildung 2.1 dargestellt. Neben einer Implementierung in Rust wurde eine Referenzimplementierung in C angefertigt, da die von STM bereitgestellten Bibliotheken für die Entwicklung mit C konzipiert sind. Diese werden deshalb über C-Header-Dateien eingebunden. Für die Implementierung in Rust wurden insbesondere die Crates `stm32f1xx-hal` [38] und `cortex-m` [39] benutzt. Das Gegenstück in C wurde mit Hilfe der STM32 Low Level Bibliothek [40] geschrieben, die Teil der STM32 Cube HAL [37] vom Hersteller STMicroelectronics ist. Die Ansteuerung des Sensors und der Anzeige wurden selbstständig implementiert, ohne vorhandenen Code einzubinden.

2.2 Algorithmen zur Performanceevaluierung

Im weiteren Verlauf wurden einige Algorithmen umgesetzt, deren Ausführungszeit anschließend gemessen wurde. Da diese Tests nicht in einen generellen Vergleich zwischen

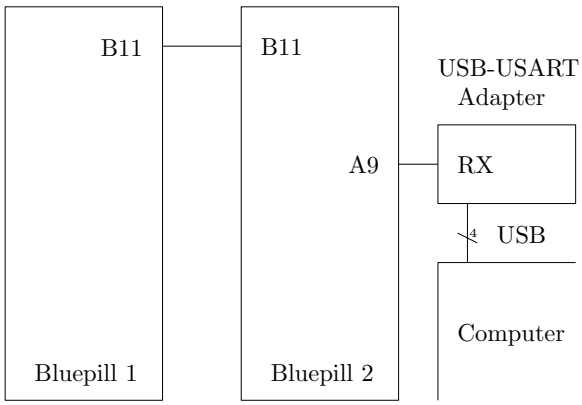


Abbildung 2.2: Schematischer Aufbau der Messungen zur Geschwindigkeits-Evaluierung.

C und Rust ausufern sollen und CPU-, aber auch Speicher- und Bus-Architekturen zwischen Desktop-Computern und Mikrocontrollern große Unterschiede aufweisen, wurden sie ebenfalls auf dem STM32-F1 als beispielhaften Vertreter der Mikrocontroller ausgeführt.

Dies stellte die Umsetzung allerdings vor die Herausforderung, die benötigten Zeiten möglichst genau und vor allem fair und systematisch zu messen. Um dieses Problem zu lösen, wurde ein zweites Bluepill-Board genutzt, über einen Pin mit dem Testgerät verbunden und über eine serielle Schnittstelle (USART) an einen Desktop-Computer angeschlossen. Der einzige Mehraufwand, der dabei auf dem den Algorithmus ausführenden Board das Ergebnis beeinträchtigen könnte, ist das Setzen des Pins während der Messung und Rücksetzen des Pins bei der Fertigstellung. Dies sollte allerdings nur wenige Taktzyklen benötigen, also bei einer entsprechend langen Messung vernachlässigbar klein sein. Darüber hinaus weißt das zweite Board auch nur eine kleine Mess-Ungenauigkeit auf, da hier lediglich kontinuierlich der Pin überprüft und ein Zähler hochgezählt werden muss. Diese Daten werden während der Pause, in der der Pin auf `LOW` steht, in ein Array geschrieben und erst nach Abschluss aller Messungen über die serielle Schnittstelle übertragen. Der genauere Aufbau ist Abbildung 2.2 zu entnehmen.

Um zu zeigen, dass die genannten Annahmen korrekt sind, wurde im ersten Versuch eine Reihe von 99 Messungen durchgeführt und während der Auswertung in einem Box-Plot (Abbildung 3.1) dargestellt, sodass die Größe der maximalen Abweichung vom Median ersichtlich ist.

Dieser Test wurde ebenfalls auf wesentliche Merkmale beschränkt. So wurden keinerlei externe Bibliotheken oder dynamisch allozierter Speicher genutzt, da die Konzentration auf Laufzeitunterschiede zwischen den Sprachen Rust und C gelenkt wurde. Dadurch müssen die Architektur, das Design und die Schnittstellen dieser Bibliotheken nicht berücksichtigt werden.

Bei der Implementierung der Algorithmen wurde, um die Werte vergleichbarer zu machen, darauf geachtet, dass alle Quelltexte sehr ähnlich gehalten und keine Geschwindigkeitsbeeinträchtigenden Abweichungen vorhanden sind. Jedoch sollten naheliegende Sprachfunktionen wie in praktischen Anwendungen genutzt werden. Dazu zählt zum Beispiel das Ersetzen von `for`-Schleifen aus C-Code mit Iterator-basierten Schleifen in Rust. Außerdem wurden hier zur Fehlerbehandlung die Typen `Option<T>` und `Result<T, E>` genutzt,

während bei C-Code meist nur besondere Werte wie -1 zurückgegeben wurden. Diese Anforderungen machten es sehr schwierig bereits fertig gestellte Algorithmen von Webseiten wie zum Beispiel Rosetta Code [41] zu nutzen, da hier die Implementierungen größtenteils unabhängig von einander angefertigt wurden und zum Teil stark von einander abweichen.

Die zuvor genannten Prinzipien gelten für alle folgenden Algorithmen sofern nicht explizit anders angegeben.

2.2.1 Größter gemeinsamer Teiler

Die erste Umsetzung, die getestet wurde, ist die Ermittlung des größten gemeinsamen Teilers mit dem Euklidischen Algorithmus. Dies ist ein relativ simpler Algorithmus, der nur drei Variablen und als einzigen arithmetischen Operator Modulo benötigt und entweder rekursiv oder iterativ implementiert werden kann. Daher eignet er sich sehr gut als minimales Beispiel.

Für den Benchmark wurde der größte gemeinsame Teiler der Zahlen 366437447 und 836492442 ermittelt. Da diese Berechnung jedoch sehr schnell durchgeführt wird, wurde sie 10.000 mal wiederholt.

2.2.2 Arraydurchläufe

Des Weiteren wurde eine Methode geschrieben, die jedes Element eines Arrays liest und anschließend den größten dieser Werte zurückgibt. Dieser Test ist relevant, da in vielen Algorithmen Arrays oder vergleichbare Strukturen wie Listen oder Vektoren benötigt werden. Hier sind ebenfalls schon nach theoretischen Überlegungen Geschwindigkeitsunterschiede zu erwarten, da in Rust Laufzeitüberprüfungen bei der Indizierung von Arrays durchgeführt werden, die in C nicht vorhanden sind. Dafür kann bei Out-of-Bounds-Fehlern in Rust eine Panic ausgelöst werden, während bei C-Code undefiniertes Verhalten auftritt.

Außerdem würde sich bei einer ideomatischen Rust-Implementierung die Verwendung von Iteratoren anbieten, um den Quellcode lesbarer zu gestalten. Da diese jedoch eine weitere Abstraktionsschicht darstellen, wurde es für sinnvoll erachtet zu überprüfen, wie sich diese auf die Geschwindigkeit des Programms auswirkt.

Um zu überprüfen, wie sich die Dauer der Ausführung in Abhängigkeit zur Array-Größe verhält, wurden nur die ersten η Elemente durchlaufen. Der dafür verwendete Speicherbereich blieb jedoch der selbe, das heißt nur ein Teil des erstellten Arrays wurde iteriert. Für jede Größe η wurde der Algorithmus 1.000 mal wiederholt.

2.2.3 Primzahlenberechnung

Ein weiterer Test berechnet alle Primzahlen, die im Intervall zwischen 2 und einer oberen Schranke, hier als η bezeichnet liegen. Dazu wird jede Zahl ϑ im genannten Intervall überprüft, indem nach einem Teiler in der Menge

$$\left[2, \sqrt{\vartheta}\right] \cap \Big(\{2,3\} \cup \{6x-1 \mid \forall x \in \mathbb{N}\} \cup \{6x+1 \mid \forall x \in \mathbb{N}\}\Big)$$

gesucht wird.

Dafür wurde sowohl eine C- als auch eine ideomatische Rust-Implementierung angefertigt und in Abhängigkeit der Schranke η getestet. Da sich hier auf Anhieb ein großer Geschwindigkeitsunterschied zum Nachteil Rusts abzeichnete wurde ebenfalls eine Implementierung entwickelt, die eine möglichst originalgetreue Übersetzung der C-Version ist. Der größte Unterschied zur vorherigen Variante ist dabei, dass keine Iteratoren genutzt werden.

2.2.4 Numerische Integration

Da numerische Berechnungen für einige auf Mikrocontrollern ausgeführte Steuerungen unerlässlich sind, wurde ebenfalls die numerische Integration einer Funktion umgesetzt. Dabei wurde der Einfachheit halber die Trapez-Methode zur Berechnung des Integrals der Funktion $f(x) = x^2$ im Intervall von 0 bis 1 verwendet. Die Anzahl der verwendeten Stützpunkte wurde dabei variiert, um dadurch auftretende Unterschiede erkennbar zu machen.

2.2.5 Dijkstras-Algorithmus

Als Beispiel für einen etwas komplexeren Algorithmus wurde zuletzt Dijkstras Algorithmus implementiert, der in einem gewichteten und in diesem Fall ebenfalls gerichteten Graphen von einem Startknoten die kürzesten Pfade zu allen anderen Knoten berechnet.

Der für das Experiment verwendete Graph mit der Menge an Knoten V und Kanten E, sowie den Kantengewichten $\mu : E \to \mathbb{R}^+$ ist wie folgt definiert.

$$
\begin{aligned}
V =& \{v_0, v_1, v_2, \dots, v_{|V|-1}\} \\
E =& \{(v_i, v_{i+1}) \mid \forall i \in \mathbb{N}_0 \cap [0, |V| - 2]\} \\
&\cup \left\{ (v_{3i}, v_{3i+3}) \;\middle|\; \forall i \in \mathbb{N}_0 \cap \left[0, \frac{|V|-4}{3}\right] \right\} \\
&\cup \left\{ (v_{5i+1}, v_{5i+7}) \;\middle|\; \forall i \in \mathbb{N}_0 \cap \left[0, \frac{|V|-8}{5}\right] \right\}
\end{aligned}
$$

$$
\forall (v_i, v_j) \in E : \; \mu((v_i, v_j)) = \begin{cases} 1 & \text{falls } i+1 = j \\ 2 & \text{falls } i+3 = j \\ 3{,}5 & \text{falls } i+6 = j \end{cases}
$$

Die Anzahl an Knoten wurde, wie schon bei vorherigen Tests, variiert. Der Startknoten ist in jedem Fall v_0.

KAPITEL 3

Ergebnisse

In diesem Kapitel sollen nun die gesammelten Daten interpretiert und ausgewertet werden. Schließlich wird zusammen mit weiteren theoretischen Betrachtungen ein fundiertes abschließendes Ergebnis verfasst.

Zuletzt werden die erhaltenen Erkenntnisse noch einmal zusammengefasst und ein Ausblick auf weitere Aspekte geworfen, deren Auswertung nicht mehr im Rahmen dieser Arbeit möglich war.

3.1 Auswertung

Die erstellten Implementierungen haben gezeigt, dass es zumindest in einigen Fällen möglich ist, sinnvolle Steuerungen für Mikrocontroller in Rust zu entwickeln. In diesem Abschnitt soll daher mit Blick auf unterschiedliche Merkmale, die schon im Kapitel 2 genannt wurden, überprüft werden, ob es sich als gleichwertige oder gar besser Alternative zur Programmierung in C eignet.

3.1.1 Lauffähigkeit auf unterschiedlichen Klassen von Mikrocontrollern

Da die erstellten Steuerungen nur für einen einzigen Mikrocontroller geschrieben wurden, es allerdings eine große Vielfalt an unterschiedlichsten eingebetteten Systemen gibt, soll hier überprüft werden, auf wie vielen dieser in Rust geschriebene Programme ausgeführt werden können. Damit dies systematisch erfolgen kann, muss zunächst eine Klassifizierung der zu betrachtenden Mikrocontroller-Systeme durchgeführt werden. Diese ist durch zwei Kriterien möglich. Zum einen kann nach der verwendeten Instruction Set Architecture (ISA) klassifiziert werden, zum anderen nach der Menge an zur Verfügung stehendem Speicher.

Klassifizierung nach CPU-Architektur

Zu den verschiedenen CPU-Architekturen zählen neben anderen `ARM` bzw. `Thumb`, `AVR`, `Xtensa`, `PIC` bzw. `MIPS` und `STM/ST` [4]. Voraussetzung dafür, dass eine Architektur von

Rust unterstützt wird, ist das Vorhandensein eines entsprechenden Backends im Compiler-Framework `LLVM` [42], welches der Rust-Compiler `rustc` nutzt. Die von `LLVM` unterstützten und durch `rustc` nutzbaren Backends sind zu diesem Zeitpunkt `AArch64`, `ARM`, `Hexagon`, `MIPS`, `MSP430`, `NVIDIA PTX`, `PowerPC`, `RISC-V`, `Sparc`, `SystemZ`, `Thumb` und `x86`. Diese können durch den Befehl `cargo objdump -- -version` aufgelistet werden [42]. Der Übersichtlichkeit halber wurden hier Variationen und Erweiterungen, wie `x86-64` nicht separat aufgelistet und virtuelle Architekturen, wie Web-Assembly ausgelassen.

Damit für diese Architekturen auch produktiv entwickelt werden kann, müssen die entsprechenden Backends auch in Rusts Toolchain und den Paketmanager `cargo` integriert werden. So können Bibliotheken, insbesondere `core`, eine minimale Form der Standardbibliothek für Systeme ohne Betriebssystem, in das Programm integriert werden. Die hier unterstützten Architekturen sind `AArch64`, `ARM`, `x86`, `MIPS`, `PowerPC`, `RISC-V`, `SystemZ`, `Sparc` und `Thumb`. Sie können mittels `rustup target list` aufgelistet werden. Die Liste ist wie zuvor zusammengefasst.

Von praktischer Bedeutung für die Mikrocontroller-Programmierung sind hier lediglich `Thumb` und `ARM`, sowie für etwas leistungsfähigere eingebettete Geräte `AArch64` und `MIPS`. Für Letztere wird ein Betriebssystem wie Linux vorausgesetzt. Daneben gibt es Bestrebungen das Entwickeln von Programmen für `AVR`-Systeme zu ermöglichen [43]. Auch für die `Xtensa`-Architektur wird an einem `LLVM`-Backend gearbeitet [44]. Diese könnten also in Zukunft ebenfalls von der Rust-Toolchain unterstützt werden.

Nicht unterstützte CPU-Architekturen können auch über Umwege mit Rust programmiert werden. Zum Beispiel gibt es Bemühungen Rust auf ESP-Geräten (`Xtensa`-Architektur) laufen zu lassen, in dem zunächst zu C-Code übersetzt wird, welcher dann mit `gcc` kompiliert werden kann [45].

Abschließend kann hier also festgehalten werden, dass bei der Unterstützung von ISAs noch deutliche Defizite bei Rust vorhanden sind. Dieser Nachteil wird wahrscheinlich kurz- und langfristig durch die Unterstützung weiterer Architekturen verringert werden. Anzunehmen Rust könnte die vielseitigen Einsatzmöglichkeiten von C erreichen wäre jedoch illusorisch.

Klassifizierung nach Größe des Speichers

Für die Klassifizierung von Systemen nach der Menge an verfügbarem Speicher kann das Kapitel 3 des RFC 7228 der Internet Engineering Task Force (IETF) [46] genutzt werden. Danach werden drei unterschiedliche Klassen an beschränkten Systemen unterschieden, die in Tabelle 3.1 ersichtlich sind sind. Dem in dieser Arbeit verwendeten STM32 im Bluepill stehen 20 KiB RAM und 64 KiB Flash zur Verfügung [47], womit es als Klasse 2 Gerät eingeordnet werden kann.

Im Folgenden wird die Größe der Binärdateien des implementierten Algorithmus zur Anzeige der Temperatur und Feuchtigkeit analysiert und in Tabelle 3.2 aufgeschlüsselt. Die beiden größten und damit relevanten Sektionen `.text` und `.rodata` sind unveränderbar und können im Flash-Speicher gelagert werden, sofern dieser wie bei vielen Mikrocontrollern

Klassifizierung des Geräts	Datenspeicher (RAM)	Programmspeicher (Flash)
Klasse 1	$\ll$ 10 KiB	$\ll$ 100 KiB
Klasse 2	$\sim$ 10 KiB	$\sim$ 100 KiB
Klasse 3	$\sim$ 50 KiB	$\sim$ 250 KiB

Tabelle 3.1: Klassifizierung von Mikrocontrollern nach Speicherkapazitäten

Name des Segments	Größe in der C Implementierung	Größe in der Rust Implementierung
.text	1552	14492
.rodata	96	7840
.data	0	0
.bss	0	4
.vector_table	0	304
.init	4	0
.fini	4	0

Tabelle 3.2: Größe einzelner Segmente der erstellten DHT-Implementierung.

Speicher-adressierbar ist. Hier zeigt sich der Vorteil der in der C-Version genutzten Low-Level Bibliothek, da diese Umsetzung nur ca. $1,6$ KiB ROM benötigt. Obwohl ersichtlich ist, dass die selbe Implementierung in Rust deutlich mehr Flash-Speicher benötigt, ist diese mit ca. 22 KiB noch relativ klein und könnte vielleicht sogar auf einige leistungsstärkere Klasse 1 Geräte geschrieben werden.

Dies muss jedoch nicht bedeuten, dass Rust für die meisten Mikrocontroller der Klasse 1 gänzlich ungeeignet ist. Die hier genutzte Version ist zum einen im Optimierungs-Level 3 für eine möglichst hohe Geschwindigkeit kompiliert und kann durch die Optimierung mit Fokus auf Binärgröße noch ein wenig verkleinert werden. Zum anderen wäre es für Geräte mit sehr geringem Speicher denkbar, die Standardbibliothek und Hardware-Crates auf den nötigsten Teil, ähnlich der verwendeten Low-Level Bibliothek in C zu optimieren. Dies würde jedoch zunächst einen deutlichen Mehraufwand bedeuten und soll in dieser Arbeit nicht näher betrachtet werden, da dann auch fraglich wäre, ob andere Vorteile gegenüber C, die aus der Standardbibiliothek resultieren, nicht ebenfalls entfernt würden.

Hier soll nun abschließend festgestellt werden, dass sich Rust bezogen auf die Speicheranforderungen für viele insbesondere Klasse 2 und 3 Geräte eignet. Für sehr eingeschränkte Systeme ist eine spezielle Betrachtung und Optimierung unausweichlich, dies träfe jedoch auch auf C-Code zu, wenn auch nicht im selben Ausmaß.

3.1.2 Produktivität

Wie schon zuvor in Kapitel 2 erläutert, ist nach der prinzipiellen Eignung zur Programmierung von Mikrocontrollern die Produktivität, mit der in Rust Programme geschrieben werden können, ein wichtiger Faktor, welcher zum Erfolg oder Misserfolg beitragen könnte. Diese ist leider sehr schwer einzuschätzen, da sie nicht an klaren Kriterien definiert werden kann, sondern vor allem durch praktische Experimente ermittelt werden müsste. Darüber

hinaus schwankt die Produktivität auch von Entwickler zu Entwickler und wird zusätzlich durch Faktoren, wie Vorkenntnisse und Erfahrung, aber auch persönliche Präferenzen des Programmierers beeinflusst [48]. Dennoch sollen hier ein paar Indizien und Argumente aufgezeigt und deren möglicher Einfluss auf die Produktivität begründet werden.

Zunächst soll als eher oberflächlicher Indikator angemerkt werden, dass Rust in der Stack Overflow Developer Survey 2019 im vierten Jahr in Folge als am meisten geliebte Sprache (im Orignal: „most loved language“) angegeben wurde [49]. Die Beliebtheit einer Sprache lässt sich zwar nicht direkt auf deren Produktivität zurückführen, jedoch scheint es naheliegend, dass Entwickler keine Sprachen mögen, in denen das Entwickeln von Algorithmen umständlich und zeitaufwändig ist. Darüber hinaus werden sie ebenfalls nicht gerne Sprachen nutzen in denen sie Mengen an Boilerplate Code schreiben müssen oder in denen wichtige, in anderen Sprachen vorhandene Sprachkonstrukte fehlen. Die Umkehrung dieser Argumentation ist zwar nicht notwendiger Weise korrekt, aber durchaus vorstellbar.

Nachfolgend sollen noch weitere Aspekte beleuchtet werden, die fundiertere Schlussfolgerungen zulassen sollten.

Einarbeitungszeit in die Sprache

Grade bei einer modernen und noch recht wenig verbreiteten Sprache wie Rust, sind die Hürden zur Einarbeitung nicht zu vernachlässigen. Sofern die Entscheidung getroffen wird, ein Projekt ganz oder teilweise in Rust umzusetzen, muss sich jeder jetzt oder in der Zukunft in diesem Bereich Beteiligte zumindest grundlegendes Wissen zur Sprache aneignen.

Dies wird zunächst dadurch erschwert, dass Rust in einigen Bereichen eine abweichende Syntax zu den verbreiteten C-ähnlichen Sprachen hat, zu denen unter anderem auch C++, objective-C, C♯, Java, JavaScript und D gehören. Dies ist beispielsweise am Vergleich einer Funktionsdeklaration zwischen C und Rust im Quellcode 3.1 und 3.2 erkennbar.

```
1 fn double(x: i32) -> i32 {
2   2 * x
3 }
```

Quellcode 3.1: Funktionsdeklaration in Rust

```
1 int double(int x)
2 {
3   return 2 * x;
4 }
```

Quellcode 3.2: Funktionsdeklaration in C

Auf der anderen Seite sind viele der hinter der Syntax liegenden Konzepte sehr ähnlich zu eventuell schon bekannten Sprachen. Dazu können Funktionsdeklarationen, Klassen bzw. Strukturen, Zeiger und Referenzen, Iteratoren und viele weitere zählen. Darüber hinaus

führt Rust jedoch auch Elemente ein, die nur sehr wenige, vor allem moderne Programmiersprachen besitzen, oder die dort erst nachträglich hinzugefügt wurden. Zu diesen zählen zum Beispiel Tupel, `match`-Ausdrücke oder die Integration von funktionalen Elementen, wie die Verkettung von Fehlern oder das Manipulieren von Variablen mit Funktionen höherer Ordnung, zu denen beispielsweise `and_then`, `or_else`, `map`, `filter` oder `fold` zählen.

Ein Alleinstellungsmerkmal Rusts ist darüber hinaus der sogenannte Borrow-Checker, der die Integrität von Referenzen zur Übersetzungszeit sicherstellt. Zum einen muss dieser feststellen können, dass Referenzen nicht länger gespeichert werden, als das Objekt, welches sie referenzieren. In einfachen Fällen kann dies ohne zusätzlichen Aufwand seitens des Programmierers durchgeführt werden, in anderen werden zusätzliche Anmerkungen benötigt. Zu diesen Fällen zählen insbesondere Strukturen, die Referenzen beinhalten, wie in Quellcode 3.3 zu sehen ist.

```
1 struct ReferenzText<'a> {
2     text: &'a str,
3 }
4
5 impl<'a> ReferenzText<'a> {
6     pub fn ausgabe(&self) {
7         println!("{}", self.text);
8     }
9 }
```

Quellcode 3.3: Strukturdefinition mit Referenz

Hier habe ich eine Struktur implementiert, die eine Referenz auf eine Zeichenkette beinhaltet. Deutlich zu erkennen ist, das nicht nur an der Referenz `&str` selbst, sondern auch bei der Strukturdefinition und bei jedem Block zur Implementierung von Funktionen, ein sogenannter Lifetime-Specifier `'a` angegeben werden muss. Ebenfalls zu erkennen ist, dass in der Ausgabe-Funktion auf eine explizite Angabe der Lebenszeit verzichtet werden kann. Dies ist jedoch auch durch `fn ausgabe(&'a self)` oder `fn ausgabe<'b>(&'b self)` möglich. Im ersten Fall hat die Referenz auf das Objekt die gleiche Lebenszeit, wie die im Objekt enthaltene Referenz. Im zweiten wird eine andere, neu definierte genutzt. Da das Objekt ohnehin nicht länger existieren kann, als die in ihm enthaltene Referenz, wäre, wenn überhaupt, die erste Version vorzuziehen, da sie einfacher zu lesen ist.

An diesen Ausführungen lässt sich schon erkennen, dass Anmerkungen zur Lebenszeit grade für Anfänger nicht immer leicht zu überschauen sind.

Darüber hinaus hat der Borrow-Checker noch eine zweite Aufgabe. Er muss sicherstellen, dass zu jedem Wert entweder höchstens eine Referenz mit Schreibzugriff existiert oder beliebig viele lesende. In Rust sind Variablen und Referenzen standardmäßig nur lesbar, und müssen, um manipuliert zu werden, mit dem Schlüsselwort `mut` markiert werden. Damit Daten auch von unterschiedlichen Programmteilen verändert werden können, wird sogenannte Interior Mutability [50] verwendet. Hierfür können normalerweise Konstrukte wie `Rc<RefCell<T>>` bzw. `Arc<Mutex<T>>` oder `Arc<RwLock<T>>` für Single- bzw. Multithreading verwendet werden. Der äußere Typ bezeichnet hierbei einen Referenzenzähler

(Reference Counter bzw. Atomic Reference Counter), der innere ermöglicht das manipulieren der Daten über ein Objekt, welches nicht mit `mut` gekennzeichnet werden muss. Die Referenzenzähler können hierbei mit dem Smart Pointer `shared_ptr` aus der C++ Standardbibliothek vergleichen werden. Ein analoges Konzept zu `RefCell<T>` gibt es allerdings nicht, da in C keine ähnliche Regeln existieren, die dies erfordern würden.

Diese Konstrukte könnten für Anfänger schnell komplex werden, in der Entwicklung für Mikrocontroller sind sie jedoch gar nicht erst verwendbar. Dies liegt daran, dass der Wert innerhalb des Referenzen-Zählers logischerweise auf dem Heap gespeichert werden muss und dynamische Speicherallokationen nicht teil der minimalen Standardbibliothek `core` zur Mikrocontrollerentwicklung sind. Dies bedeutet, dass es nur möglich ist komplexere Daten, die über atomische Variablen hinaus reichen, aus verschiedenen Programmteilen zu verändern, falls Alternativen betrachtet werden. Dazu zählt entweder das Nutzen eventueller anderer Bibliotheken oder das eigenhändige Entwickeln von vergleichbaren Lösungen mittels `unsafe`-Code, in dem klassische Zeiger dereferenziert werden können. Dies würde den Programmierer schon beim Planen der Softwarearchitektur dazu zwingen diese Probleme zu berücksichtigen oder es wäre nötig sie im nachhinein entsprechend anzupassen.

Es muss also festgehalten werden, dass zur Einarbeitung in Rust selbst für in anderen Sprachen erfahrene Programmierer einige Hürden überwunden werden müssen. Hingegen kann die relativ simple Sprache C recht einfach erlernt werden und wirkt dadurch zunächst deutlich attraktiver. Aber auch hieraus können sich Probleme entwickeln. In Rust wird der Programmierer gezwungen sich mit dem Speichermanagement seines Programms auseinanderzusetzen und klare hierarchische Strukturen zu schaffen. Dagegen kann ein unerfahrener C-Programmierer relativ schnell scheinbar funktionierenden Code schreiben, der jedoch in einigen Fällen Probleme verursacht. Zu diesen könnten Speicherleaks oder das Dereferenzieren von Zeigern auf zerstörte Objekte zählen. Zusätzlich kann das unbeachtete Verwenden der selben Daten aus unterschiedlichen Programmteilen die Integrität dieser Daten gefährden. Ein Beispiel für letztere Situation im Kontext der Mikrocontrollerprogrammierung wäre das Ausführen einer ISA, die auf Daten zugreift, die vom Hauptprogramm zuvor nur teilweise geschrieben werden konnten, da es durch den Interrupt unterbrochen wurde.

Ausgereiftheit der Toolchain

Ein weiterer wichtiger Aspekt, der sich auf die Produktivität der Entwickler auswirken kann, ist die Funktionsweise und der Umfang der verwendeten Toolchain. Hier gibt es einige Punkte zu beachten, zu denen Buildwerkzeuge und -skripte, Editoren bzw. Integrierte Entwicklungsumgebungen (IDEs), Debugger und Bibliotheken, vor allem zur Ansteuerung der Hardware, gehören.

Der erste Punkt wird in Rust durch das Programm `cargo` abgedeckt [51]. Dabei handelt es sich sowohl um ein Werkzeug zum Erstellen der Binärdateien, als auch zum Verwalten der benötigten Bibliotheken, in Rust Crates genannt. Zusätzlich ermöglicht es das Testen mittels Unit Tests oder Benchmarken von Code.

Da es hier zur Einschätzung kaum klar definier- und anschließend bewertbare Kriterien gibt, soll im Folgenden kurz mein persönlicher Eindruck über `cargo` angerissen werden. Während der Entwicklung der Implementierungen und auch bei vorherigen Projekten mit Rust sind keine negativen Punkte über `cargo` aufgefallen. Die Einbindung von Bibliothe-

ken funktioniert einwandfrei und kann zum Beispiel über die Seite `crates.io`, über ein Git-Repository oder einfach über Dateipfade angegeben werden. Diese werden dann heruntergeladen und mit dem eigenen Quellcode kompiliert. Da fast alle Rust-Bibliotheken `cargo` als Buildsystem nutzen, können keine Probleme mit inkompatiblen Build-Skripten auftreten, wie es in C möglich wäre. Darüber hinaus können separat Compiler-Argumente im Debug- und Release-Modus angegeben werden, die z. B. zur Auswahl des Optimierungslevels genutzt werden können. Die Erstellung von Binärdateien für andere Plattformen, insbesondere Mikrocontroller, ließ sich ebenfalls problemlos durchführen. Die extra hierfür angepasste Implementierung `xargo` [52] habe ich nicht nutzen müssen, da diese Funktionen mittlerweile in `cargo` integriert wurden.

Dies zeigt auch schon den einzigen Punkt, der hier als negativ angemerkt werden sollte. Durch die noch anhaltend schnelle und stetige Weiterentwicklung der Sprache und Werkzeuge entstehen schnell veraltete Dokumentationen oder nicht mehr nutzbare Anleitungen in Blog-Einträgen oder Ähnlichem. Darüber hinaus kann Verwirrung entstehen, zu welchem Zweck Programme wie `xargo` nun existieren und ob diese noch benötigt werden, wenn ein Teil der Funktionalität in `cargo` integriert wurde. Schließlich entstehen dadurch auch viele Dopplungen von Bibliotheken mit unterschiedlicher Aktualität, die auf der Seite `crates.io` vorhanden sind. Dies ist teilweise dem Umstand geschuldet, dass aus Kompatibilitätgründen keine bereits erstellten Bibliotheken wieder entfernt werden können. Auch beruhen diese Bibliotheken selber auf unterschiedlichen Versionen anderer Crates, sodass jede referenzierte Version heruntergeladen und kompiliert werden muss.

Als Editoren für Rust-Code können eine Vielzahl an unterschiedlichen Programmen verwendet werden. Zwar gibt es keine IDE für Rust, allerdings können durch viele Plugins für Editoren grundlegende Funktion, wie Syntax Highlighting, Code-Vervollständigung, das Suchen von Implementierungen oder die Integration eines Debuggers genutzt werden [53]. Dieser Teil kann also als annähernd vergleichbar zu C gelten, auch wenn eine umfangreichere Menge an C-Editoren verfügbar ist.

Zum Debugging von in Rust geschrieben Steuerungen kann der auch für C-Programme genutzte Debugger `gdb` [54] verwendet werden. Dieser wurde erfolgreich im Zusammenspiel mit OpenOCD [55] zum Testen der Implementierungen auf dem Mikrocontroller genutzt. Hier ergeben sich also weder Vor- noch Nachteile gegenüber C.

Als letztes soll noch auf die Möglichkeiten zur Steuerung der Hardwarekomponenten des Mikrocontrollers eingegangen werden. Da sich diese von Gerät zu Gerät unterscheiden, beziehen sich die nachfolgenden Betrachtungen auf das zur Implementierung genutzte Crate `stm32f1xx-hal` [38] für den STM32F1. Dieses implementiert allerdings auch Schnittstellen, die in der hardware-agnostischen `embedded-hal` [56] Bibliothek definiert sind.

Das Crate hält sich zunächst sehr nah an den vorhandenen Konfigurationsregistern. Dabei wird jedes Register als Objekt seines eigenen Typs betrachtet. Dieses Konzept wird konsequent weitergeführt, in dem der Status eines GPIO-Pins in seinem Typ kodiert wird. So gibt es beispielsweise die Typen `PA1<Output<PushPull>>`, `PA1<Output<OpenDrain>>`, `PA1<Analog>`, `PA1<Input<PullUp>` usw. Dies ist zum einen hilfreich, da beispielsweise bei einer Funktionsdefinition zur Übersetzungszeit sichergestellt werden kann, dass der übergebene Pin im korrekten Zustand konfiguriert ist. Auf der anderen Seite werden aber auch Probleme aufgeworfen. Zum Beispiel gibt es zwar Schnittstellen wie `InputPin` und

`OutputPin`, die das Lesen bzw. Setzen eines Pins abstrahieren, zur Kommunikation mit dem zuvor verwendeten DHT-Sensor ist jedoch zwischenzeitlich eine Umkonfiguration zwischen In- und Output erforderlich. Dies erschwert es eine Funktion zu schreiben, die die Kommunikation mit einem solchen Sensor unabhängig vom verwendeten Pin implementiert. In diesem Fall wurde dafür während der Umsetzung eine eigene Schnittstelle `DHTPin` definiert, die sowohl In-, als auch Output ermöglicht und für den gewünschten Pin implementiert. Dies ist jedoch auch nur eine teilweise akzeptable Lösung, da die Implementierung dieser Schnittstelle ca. 80 Zeilen Quellcode umfasst und für jeden damit verwendeten Pin kopiert werden müsste. Darüber hinaus wird zum Wechsel zwischen In- und Output-Pin zwingend eine Referenz mit Schreibzugriff auf das entsprechende Konfigurationsregister benötigt. Sofern diese in die Implementierung des DHT-Pins verschoben wird, kann auf Grund Rusts Regeln für Referenzen keine weitere existieren. Daraus folgt, dass zu einem späteren Zeitpunkt kein anderer Pin mehr mit diesem Register umkonfiguriert werden kann. Hier fehlt also eine logische Trennung zwischen einzelnen Pins, die die Hardware so nicht erlaubt. Die einzige offensichtliche Lösung dieses Dilemmas besteht darin, herkömmliche Zeiger zu nutzen und in `unsafe`-Regionen zu dereferenzieren. Da das selbe Register dadurch allerdings von unterschiedlichen Stellen manipuliert werden kann, kann Rusts Borrow-Checker nicht mehr für die Integrität dieses Registers garantieren. Der entscheidene Vorteil, den Rust hier allerdings gegenüber C hat, ist, dass diese Garantien nur für die Variablen, auf die von dem Code innerhalb der `unsafe`-Blöcke zugegriffenen wird, aufgehoben werden. So kann die Aufmerksamkeit des Entwicklers konsequent auf mögliche Problemstellen gelenkt werden, während diese Fehler in C prinzipiell jederzeit auftreten könnten.

Am Rande soll hier noch erwähnt werden, dass auch die Einbindung von Inline Assembly in Rust mit dem Makro `asm!` möglich ist [57]. Diese Funktion ist jedoch noch nicht in der stabilen Version nutzbar. Inline Assembly ist in einigen Fällen nötig, in denen spezielle Programmteile von Hand optimiert werden sollen oder die Arbeit mit Registern nötig ist, die nicht durch eine Speicheradresse angesprochen werden können.

Einfluss einiger Sprachfunktionalitäten

Natürlich können auch weitergehende Funktionalitäten der Sprache einen positiven Einfluss auf die Produktivität haben. Hier greift Rust viele Konzepte von etablierten Programmiersprachen auf und bietet sowohl Möglichkeiten zur objektorientierten Programmierung als auch Sprachelemente einiger funktionaler Sprachen.

Zu ersteren zählt beispielsweise das Erstellen von Klassen, in Rust Strukturen genannt, und die Abstraktion von Funktionalitäten durch Schnittstellen, in Rust Traits genannt, die von bestimmten Klassen implementiert werden können. Durch diese kann außerdem Polymorphismus umgesetzt werden. Zusätzlich können Funktionen, Strukturen und Traits durch generische Typargumente generalisiert werden. Hingegen ist eine direkte Möglichkeit zur Vererbung nicht gegeben.

Zu den funktionalen Elementen zählt der Umgang mit Iteratoren, die durch Funktionen wie `map`, `filter`, `take_while`, `fold` oder `zip` verändert werden können. Diese sind beispielsweise mit der selben Semantik in Haskell zu finden. Dazu erinnern Features, wie Pattern Matching mit `match` oder das einfache Zusammenfassen mehrerer Werte zu Tupeln ebenfalls stark an funktionale Programmiersprachen. Außerdem sind Variablen in Rust

standardmäßig unveränderbar und müssen für eine imperative Programmierweise mit dem Schlüsselwort `mut` definiert werden.

Über diese Möglichkeiten hinaus können in Rust Makros definiert werden, die über bloße Textersetzung, wie in Cs Präprozessor, hinaus gehen. So können sie für unzählige hilfreiche Funktionen verwendet werden, die auch eventuelle Limitierungen der Sprache umgehen können. Beispielsweise soll hier das Makro `vec!` aus der Standardbibliothek vorgestellt werden, dass es ermöglicht Vektoren aus Literalen wie `vec![1, 2, 3]` zu erzeugen. Darüber hinaus werden Makros zur Ausgabe auf der Konsole oder zum Logging, beispielsweise mit `println!` bzw. `warn!` oder `trace!` des Crates `log` [58] verwendet. Als Makros erlauben sie das zusätzliche Formatieren von beliebigen Anzahlen an Argumenten, ähnlich zu `printf` in C. Diese können in Rust nicht als normale Funktion definiert werden, da die Sprache keine variadischen Funktion, also Funktionen mit einer beliebigen Anzahl an Argumenten unterstützt. Makros können aber auch für komplexere Sachverhalte genutzt werden, wie das Makro `object!` aus dem Crate `json` [59] zeigt. Dieses ermöglicht es auf einfache Weise JSON-Objekte im Quellcode zu definieren, ohne sie zur Laufzeit aus einem String parsen zu müssen. Eine solche Vorgehensweise wäre in C nur durch das wiederholte Ausführen mehrerer Funktionen möglich.

3.1.3 Zuverlässigkeit und Wartbarkeit

Ein weiterer Punkt, der Einfluss darauf haben könnte, wie attraktiv Rust zur Programmierung von Mikrocontrollern wirkt, wäre eine größere Zuverlässig der Implementierungen gewährleisten zu können. Da Mikrocontroller in vielen eingebetteten Systemen zum Einsatz kommen, die zum Teil ununterbrochen laufen, ist eine größtmögliche Ausfallsicherheit wünschenswert. Sie können ebenfalls an Orten eingesetzt werden, in denen ein Austausch oder erneutes Flashen einer korrigierten Software kaum rentabel ist. Zusätzlich werden einige dieser Systeme gar in sicherheitskritischen Bereichen wie dem Verkehr oder zur Steuerung von Maschinen eingesetzt, in denen eine Störung unter allen Umständen vermieden werden muss.

Daneben ist es bei der Softwareentwicklung immer möglich, dass unvorhergesehene Fehler auftreten. Von daher wäre auch eine Hilfe bei der Fehlersuche und -behebung von positivem Einfluss. Schließlich kann es ebenfalls möglich sein, dass bestehender Quellcode für neue Anforderungen oder zur Unterstützung anderer Geräte angepasst werden soll.

In den anschließenden drei Abschnitten sollen daher die Wart- und Erweiterbarkeit, sowie die Möglichkeiten zur Fehlerbehandlung und -vermeidung von Rust-Quellcode beleuchtet werden.

Wart- und Erweiterbarkeit

Wie effizient Quellcode angepasst oder erweitert werden kann hängt in erster Linie wohl eher mit der erstellten Softwarearchitektur und den Fähigkeiten, sowie der Sorgfältigkeit des Programmierers zusammen, als mit der verwendeten Programmiersprache. Daher wird in diesem Punkt das Verwenden von Rust nur eine untergeordnete Rolle spielen. Dennoch soll hier kurz auf zwei Punkte eingegangen werden.

Zum einen sind Anpassungen und Refactorings des Quellcodes aufwändiger und komplexer, sobald Änderungen an Struktur der Speicherverwaltung vorgenommen werden. Hiermit ist beispielsweise das Verwenden von Referenzen anstatt Kopien von Werten gemeint. Dazu zählen außerdem Änderungen an der Lebensdauer von Objekten, z. B. falls ein solches länger verwendet oder von einem anderen Teil des Codes wieder freigegeben werden soll.

Hier zeigt sich ein Vorteil Rusts, indem Speicherfehler, zum Beispiel durch das Benutzen von Referenzen auf gelöschte Objekte, zur Übersetzungszeit ausgeschlossen werden können. Dies vereinfacht es Programmierern Veränderungen an solchen Programmteilen vorzunehmen, die besonders komplex sind bzw. vor einiger Zeit oder von einem anderen Entwickler angefertigt wurden. Vermeintlich versteckte Fehlerquellen können hier durch den Compiler aufgedeckt werden.

Auf der anderen Seite ist es dafür folglich auch nötig, dass der Compiler die Lebenszeiten der Objekte und Referenzen nachvollziehen kann. Dafür müssen an einigen Stellen Anmerkungen im Code erfolgen, die teilweise recht umständlich ausfallen können. Hier sei als Beispiel eine Struktur vorstellbar, die unter anderem als Attribut ein Objekt speichert. Zur Optimierung soll nun, eventuell sogar nur zu Testzwecken, dieses Objekt durch eine Referenz auf selbiges, welches jetzt an einem anderen Ort gespeichert wird, ausgetauscht werden. Falls die Struktur vorher keine Referenz beinhaltete, wird nun eine explizite Anmerkung der Lebenszeit für sie erfordert. Diese muss an vielen unterschiedlichen Stellen im Quellcode hinzugefügt werden, z. B. an ihrer Definition selbst, an dem `impl`-Block zur Definition von Methoden auf der Struktur und an jeder ihrer Trait-Implementierungen, wie teilweise im Quellcode 3.3 zu erkennen ist. Dieses Beispiel behandelt zwar nur einen Randfall, zeigt aber, dass bei der Verwendung von Rust manchmal kleinere Ärgernisse auftreten können.

Fehlerbehandlung

Nun sollen die Möglichkeiten zur Fehlerbehandlung bei der Programmierung mit Rust aufgezeigt werden.

Anders als Enumerationen in C können solche in Rust je nach Variante unterschiedliche zusätzliche Variablen speichern [60]. Dies ermöglicht die Implementierung der generischen Typen `Option<T>` und `Result<T,E>`, die in Rust der Fehlerbehandlung dienen. Dabei zeigt ersterer die Möglichkeit zur Abwesenheit eines Wertes, während der zweite bei einem aufgetretenen Fehler einen zusätzlichen z. B. zur genaueren Spezifikation der Ursache speichert. Diese Art der Fehlerbehandlung orientiert sich stark an vielen, aus funktionalen Sprachen bewährten Typen. Hier seien als Beispiel Haskells Typen `Maybe` und `Either` oder Scalas `Option`, `Either` oder `Try` genannt. Andere Sprachen, die ähnliche Konzepte nutzen sind F♯ oder OCalm.

Wenn nun eine Funktion eine dieser beiden Enumerationen als Rückgabewert besitzt, um anzudeuten, dass sie fehlschlagen kann, wird der Programmierer dazu gezwungen eine Behandlung des möglichen Fehlers durchzuführen. Der eventuell enthaltene, gewünschte Rückgabewert kann also nicht einfach implizit aus `Option<T>` oder `Result<T,E>` entnommen werden. Die simpelste Form der Behandlung, die der Programmierer durchführen kann, ist das Ausführen der Funktion `unwrap` auf dem erhaltenen Wert. Dies führt im Falle des Fehlers zum Auftreten einer sogenannten Panic, welche für gewöhnlich zum Abbrechen der Ausführung des Programms führt. Daher sollte diese Möglichkeit nur bei einem ohnehin

kritischen Fehlschlag genutzt werden. Eventuell könnte sie auch für Prototyping verwendet werden, in dem zunächst keine Randfälle beachtet werden sollen.

Eine korrekte Fehlerbehandlung kann durch unterschiedliche Arten durchgeführt werden, wie im Quellcode 3.4 anhand einer Divisions-Funktion dargestellt ist. Diese kann fehlschlagen, sofern der Divisor null ist. Zum einen ist es möglich durch einen `match`-Ausdruck die Varianten der Enumeration aufzuschlüsseln, wie in Zeile 9 bis 16 demonstriert wurde. Falls die Art des aufgetretenen Fehler irrelevant ist, kann auch eine `if let`-Anweisung wie in den Zeilen 18 bis 24 verwendet werden. Diese kann also als ein vereinfachter `match`-Ausdruck mit nur einem Arm aufgefasst werden.

Neben diesen Varianten bietet Rust auch noch Möglichkeiten zur Fehlerfortpflanzung (Error-Propagation). Hierbei können mehrere Funktion mit den selben Fehlertypen, jedoch nicht zwingend den gleichen Ergebnistypen, nacheinander ausgeführt werden. Dies kann solange geschehen bis ein Fehler auftritt oder die Aufgabe erfolgreich abgeschlossen wurde. Bei abweichenden Fehlertypen können diese durch die Funktion `map_err` in einander umgewandelt werden. Zum einen können die aufzurufenden Funktionen, in Anlehnung an funktionale Sprachen, durch wiederholtes aufrufen von `and_then` in anonymen Funktionen ausgeführt werden, wie in Zeilen 26 bis 33 zu sehen ist. Zum anderen existiert dafür eine Kurzschreibweise, die den `?`-Operator nutzt. Dieser kann wie in den Zeilen 35 bis 46 angewendet werden, wenn die Funktion ebenfalls ein `Result<T,E>` mit dem gleichen Fehlertypen zurückgibt. Der von der verschachtelten Funktion zurückgegebene Wert kann dann im weiteren Verlauf so verwendet werden als wäre ein Fehlschlag nicht möglich. Tritt dieser jedoch auf, so wird er von der Funktion frühzeitig zurückgegeben. Dieser Operator kann zur Zeit in der stabilen Rust Version nur für die Typen `Result<T,E>` und `Option<T>` genutzt werden, es gibt aber Bemühungen diese Funktionalität durch einen Trait zu generalisieren, wie in Rusts RFC 1859 nachzulesen ist [61].

Schließlich gibt es auch Entwicklungen, die es ermöglichen sollen den `?`-Operator nicht nur in Funktionen mit entsprechendem Rückgabewert nutzen zu können. Diese, in Rusts RFC 2388 festgehaltenen Überlegungen [61], zeigen starke Parallelen zu einer `do`-Notation, wie sie beispielsweise in Haskell üblich ist.

Im Gegensatz dazu gibt es in der Programmiersprache C keine einheitliche Möglichkeit zur Behandlung von Laufzeitfehlern. Da die Entwicklung von Strukturen dafür dem Entwickler überlassen ist, können viele zueinander inkompatible Lösungen entstehen. Da Programmierer außerdem nicht dazu gezwungen werden, sich mit der Möglichkeit eines Fehlschlags zu beschäftigen könnten einige oder gar viele Randfälle nicht beachtet werden und im Produktiveinsatz zu Störungen führen.

Fehlervermeidung

Neben den Konstrukten zur sinnvollen Behandlung von Laufzeitfehlern hilft Rust auch beim Vermeiden dieser durch Überprüfungen, die schon zur Übersetzungszeit stattfinden. Neben den Typ-Überprüfungen die alle Sprachen mit statischer Typisierung ermöglichen, können durch Rusts Design weitere Arten von Fehlern festgestellt werden.

```
fn division(x: i32, y: i32) -> Result<i32, &'static str> {
  if y == 0 {
    Err("Division durch null.")
  } else {
    Ok(x / y)
  }
}

fn behandlung_mit_match(x: i32, y: i32) {
  match division(x, y) {
    Ok(ergebnis) =>
      println!("Das Ergebnis lautet {}.", ergebnis),
    Err(fehler) =>
      println!("Die Operation ist fehlgeschlagen: {}", fehler),
  }
}

fn behandlung_mit_if_let(x: i32, y: i32) {
  if let Ok(ergebnis) = division(x, y) {
    println!("Das Ergebnis lautet {}." ergebnis);
  } else {
    println!("Die Operation ist fehlgeschlagen.");
  }
}

fn fehlerfortpflanzung_funktional(x: i32, y: i32)
  -> Result<i32, &'static str> {

  division(x, y)
    .and_then(|ergebnis| {
      division(ergebnis, y)
    })
}

fn fehlerfortpflanzung_fragezeichen_operator(x: i32, y: i32)
  -> Result<i32, &'static str> {

  let ergebnis = division(x, y)?;
  division(ergebnis, y)
}
```

Quellcode 3.4: Möglichkeiten zur Fehlerbehandlung in Rust

Rust nutzt eine Form der automatischen Speicherverwaltung, sodass ein manuelles Allozieren und Freigeben von Speicher in den meisten Fällen nicht nötig ist. Um dies nicht nur für Objekte auf dem Stack, sondern auch auf dem Heap zu ermöglichen, führt Rust das Konzept des Speicherbesitzes (Memory Ownership) ein. Jedes Objekt wird demnach genau von einem anderen oder einem Scope, z. B. in einer Funktion, besessen. Falls dessen Existenz endet, kann es ebenfalls freigegeben werden. Diese Funktionalität kann nicht nur zum Freigeben von Speicher, sondern auch durch das Implementieren eines Destruktors für Ressourcen genutzt werden. Dieser wird in Rust durch den `Drop`-Trait definiert. Beispielsweise kann so automatisch eine Datei geschlossen werden oder, grade im Kontext der Mikrocontrollerprogrammierung, Peripherie zurückgesetzt oder Kommunikationsschnittstellen sinnvoll beendet werden. Dies funktioniert besonders gut im Zusammenspiel mit den zuvor aufgezeigten Möglichkeiten zur Fehlerbehandlung, da so Programme sinnvoll auf Randbedingungen reagieren und gleichzeitig sichergestellt wird, dass in jedem möglichen Szenario wichtige Ressourcen wieder freigegeben werden.

Das Prinzip des Speicherbesitzes ist kein gänzlich neues Konzept, sondern ähnelt der Verwendung des C++ Smart-Pointers `unique_ptr` [19]. Im Gegensatz zu diesem wird dieses Prinzip in Rust jedoch für alle Objekte verwendet und muss nicht manuell vom Programmierer eingesetzt werden.

Daneben gibt es außerdem die schon zuvor angerissenen Garantien für Referenzen. In sicherem Rust, also ohne die Verwendung von `unsafe`-Blöcken, kann es zur Laufzeit nicht zu Fehlern durch das Referenzieren freigegebener Objekte oder durch Race Conditions kommen. Letztere können auch auf Mikrocontrollern mit nur einer Ausführungseinheit auftreten, in dem der Programmablauf zwischenzeitlich durch Interrupts unterbrochen wird. Da diese Problemklassen in Sprachen wie C ohne spezielle Vorkehrungen sehr leicht auftreten können und es fast unmöglich ist sicherzugehen, dass diese in einem solchen Programm nicht vorhanden sind, überzeugt Rust hier mit einem klaren Vorteil.

Eine weitere Schwachstelle, die in C bzw. C++ zu undefiniertem Verhalten, bis hin zu Sicherheitslücken führen kann, sind Buffer-Overflows. Diese können zwar in Rust nicht zur Übersetzungszeit vermieden werden, allerdings wird in Rust bei jeder Array-Indizierung eine Überprüfung auf Out-of-Bounds-Zugriffe durchgeführt. Dies führt zwar immer noch zum auslösen einer Panic und damit zum Absturz des Programms bei fehlerhaften Zugriffen, diese können jedoch nicht länger von einem Angreifer zum auslesen vertraulicher Informationen oder Ähnlichem genutzt werden. Bei der Dereferenzierung von Zeigern in `unsafe`-Code gilt diese Garantie, sowie die zuvor genannten, nicht mehr.

Neben diesen Aspekten seien hier noch einmal die erweiterten Möglichkeiten Rusts Typ-Systems erwähnt, durch die es der Hardware-Abstraktionsschicht möglich war, den Konfigurationszustand eines Pins in seinem Typen zu codieren. Für Pins als Eingang ist der Input-Pin Trait implementiert, analog dazu sind Ausgänge umgesetzt. Je nach Konfiguration sind also bei der Programmierung unterschiedliche Schnittstellen und Methoden zugänglich. Dieses Konzept garantiert folglich die korrekte Konfiguration beim Nutzen eines Pins und ist so selbst in Sprachen mit generischen Typen, wie beispielsweise C++, nur schwer umsetzbar.

3.1.4 Geschwindigkeit

Zuletzt soll die Geschwindigkeit von Implementierungen ausgewertet werden, die mit Rust geschrieben wurden und mit den Umsetzungen in C verglichen werden.

Zur Darstellung der jeweils 99 Messwerte des ersten Algorithmus wurde ein Box-Plot angefertigt, welcher in Abbildung 3.1 zu finden ist. Es ist ersichtlich, dass die in C geschriebenen Versionen beinahe eine identische Ausführungszeit zeigen. Dagegen benötigt die rekursive Version in Rust deutlich länger als die iterative, welche etwas effizienter als die C-Version ausgeführt wird. Wie dem Diagramm entnehmbar ist, ist die Differenz zwischen dem Maximum und dem Minimum der einzelnen Messreihen sehr gering, daher kann in diesem und den folgenden Versuchen auf eine detaillierte statistische Auswertung verzichtet werden.

Bei der Iteration über ein Array, welche in Abbildung 3.2 dargestellt ist, zeigt sich ein geringer Geschwindigkeitsnachteil bei der Nutzung von Rust. Werden für diese Iteratoren genutzt, wächst der Unterschied noch weiter an.

Ähnlich verhält es sich bei der Berechnung von Primzahlen, wie in Abbildung 3.3 zu sehen. Hier zeigt die ideomatische Rust-Implementierung eine deutlich verlängerte Ausführungszeit. Wird der Code in einem C-ähnlichen Stil geschrieben, nähert sich diese der Dauer der

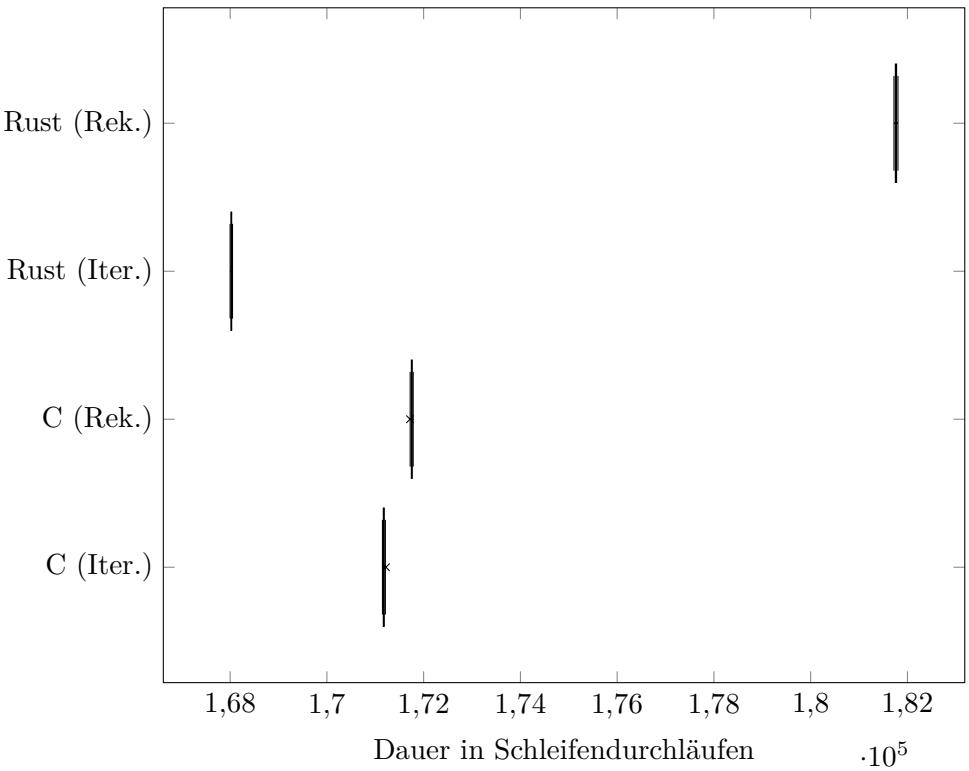


Abbildung 3.1: Box-Plot Darstellung der Dauer, die verschiedene Implementierungen zur Berechnung des größten gemeinsamen Teilers der Zahlen 366437447 und 836492442 benötigen. Für jede Programmiersprache wurde der Algorithmus iterativ und rekursiv getestet und pro Messwert 10.000 mal ausgeführt.

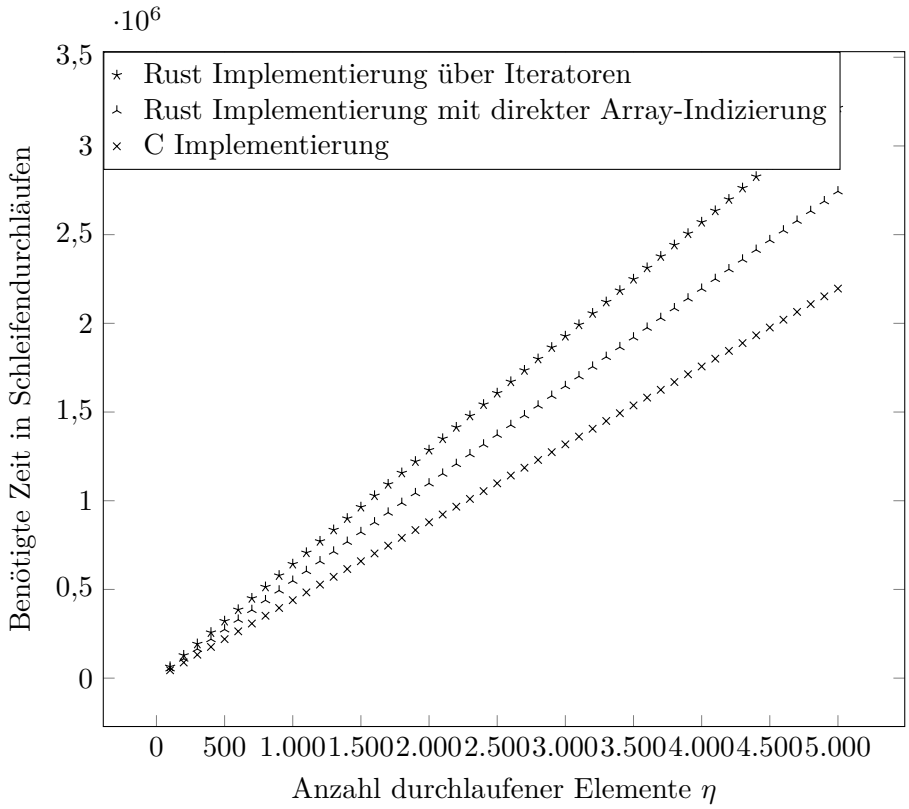


Abbildung 3.2: Benötigte Zeit zum Iterieren von Elementen eines Arrays in Abhängigkeit der Anzahl η durchlaufener Elemente. Alle entsprechenden Einträge wurden seriell abgerufen, um deren Maximum zu ermitteln. Dafür wurde entweder direkte Indizierung mit einem Zähler oder das Durchlaufen eines Iterators genutzt.

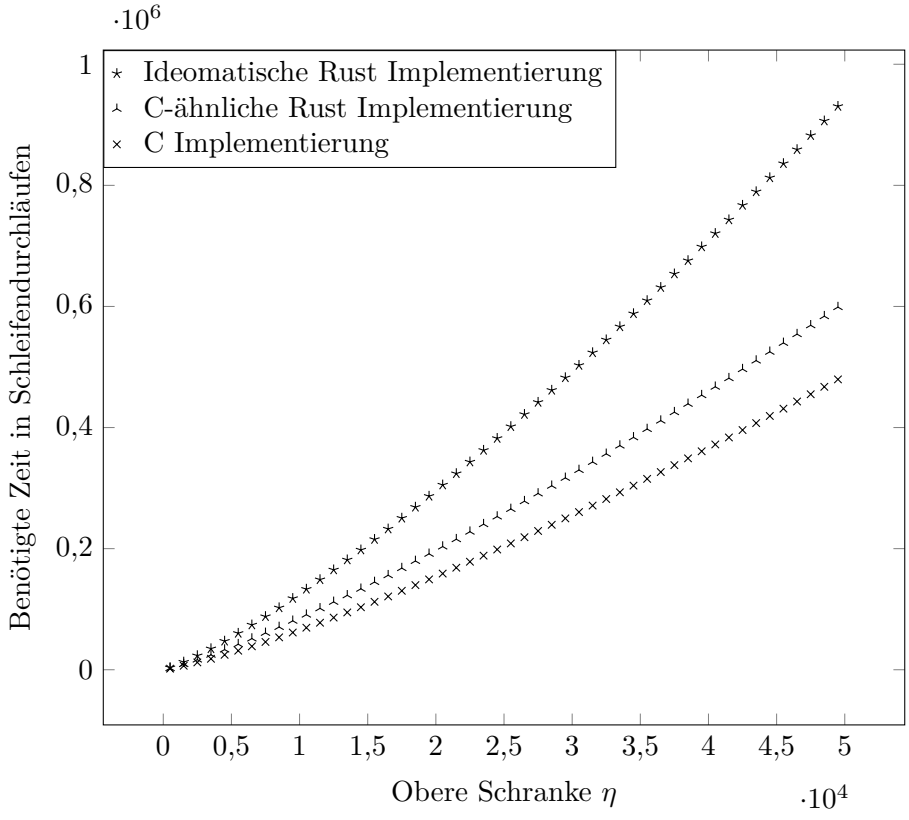


Abbildung 3.3: Darstellung der Dauer verschiedener Implementierungen, um alle Primzahlen von 2 bis zu einer oberen Schranke η in Abhängigkeit dieser Schranke zu berechnen.

Algorithmus	Dauer der C-Version	Dauer der Rust-Version	Relative Dauer der Rust Version
GGT (Iter.)	171176	168023	98.16%
GGT (Rek.)	171754	181764	105.83%
Arraydurchläufe	2195932	3214423	146.38%
Integrierung	371233	431463	116.22%
Primzahlenberechnung	479534	930541	194.05%
Dijkstras Alg.	864279	550959	63.75%

Tabelle 3.3: Zusammenfassung der Geschwindigkeitsunterschiede zwischen Rust und C. Die relative Dauer bezieht sich auf die entsprechende C-Implementierung. Es wurden immer die Durchführungen mit der längsten Ausführungszeit ausgewertet und für den Algorithmus zur Berechnung des größten gemeinsamen Teilers der Median genutzt. Es wurde immer die ideomatische Version ausgewertet.

originalen C-Version an, ist jedoch noch immer ein wenig langsamer.

Die Messungen der Umsetzung der numerischen Integration zeigen nur einen kleinen Geschwindigkeitsunterschied, den C knapp für sich gewinnen kann. Dies kann in Abbildung 3.4 nachvollzogen werden.

Im Gegensatz dazu wir bei der Darstellung der Ausführungszeiten Dijkstras Algorithmus in Abbildung 3.5 deutlich, dass hier die in Rust geschriebene Variante deutlich effizienter ausgeführt wird.

Um hier eine detaillierte Auswertung der Einflüsse bestimmter Sprachfeatures anzufertigen, müsste die Menge an getesteten Implementierungen deutlich höher liegen. Zudem müssten sie, sofern möglich, mit unterschiedlichen Compilern übersetzt werden, z. B. bei der C-Version sowohl mit `gcc` als auch `clang`. Daher soll hier, in Übereinstimmung mit dem Ziel dieser Bachelorarbeit, nur ein grober Überblick ermöglicht werden, der einen Einblick in praktisch zu erwartende Ergebnisse liefert.

Bei der Betrachtung der relativen Unterschiede in Tabelle 3.3 wird deutlich dass keine Sprache konzeptionell langsamer ist. In den meisten Fällen muss bei der Nutzung von Rust ohne spezielle Optimierungen ein geringer Geschwindigkeitsverlust akzeptiert werden. In anderen Fällen, wie bei der ideomatischen Version der Primzahlenberechnung, kann Rust jedoch auch um Faktor zwei langsamer sein. Dafür kann es ebenfalls Fälle geben in denen Rusts Compiler bessere Optimierungen durchführt als `gcc`, wie durch die Implementierung Dijkstras Algorithmus belegt ist.

Da dies selbstständig angefertigte Implementierungen sind, ist natürlich nicht auszuschließen, dass sich in ihnen Fehler befinden, die einen Einfluss auf die Geschwindigkeit nehmen. Einen solchen Fehler, der bei einer Kontrolle aufgefallen ist und berichtigt wurde, befand sich zunächst in der C-Implementierung des Primzahlentests. In der `for`-Schleife die alle möglichen Teiler bis zur Wurzel der Zahl durchtestet, wurde die Wurzelfunktion direkt innerhalb der Abbruchbedingung der Schleife aufgerufen. Obwohl das Ergebnis konstant war, hat der Compiler diese Aufrufe nicht optimieren können, sodass die Ausführungszeit deutlich zu lang ausfiel. Wenn man bedenkt dass diese Fehlerquelle in der Rust-Version durch

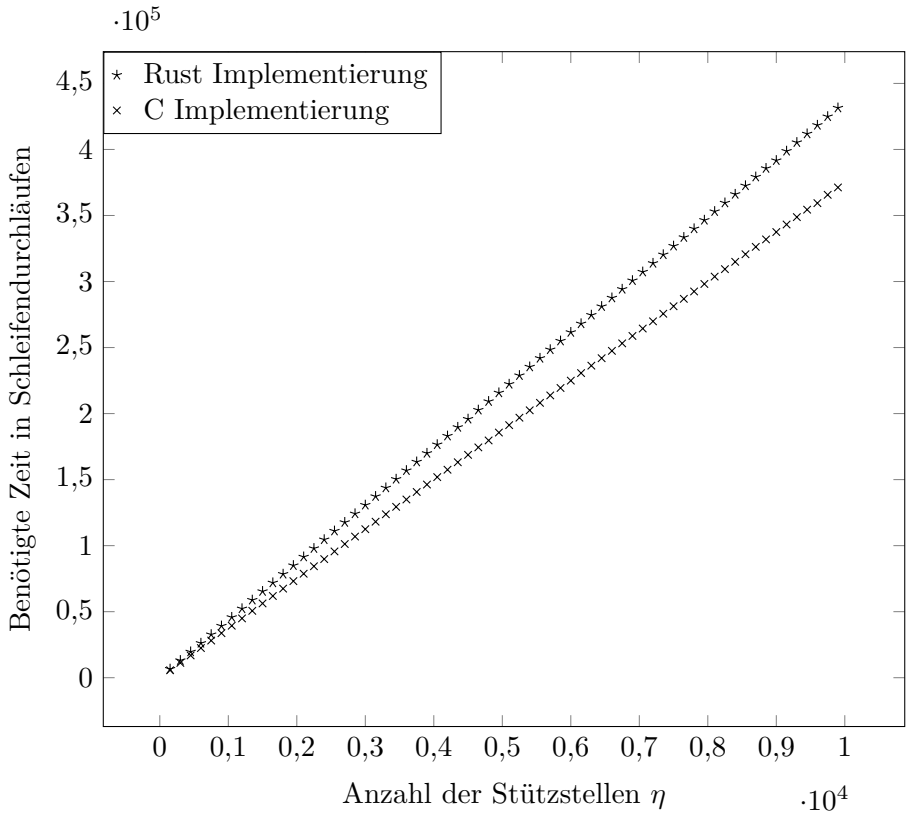


Abbildung 3.4: Darstellung der Dauer verschiedener Implementierungen, um numerisch das Integral über $f(x) = x^2$ im Interval $[0, 1]$ mit Hilfe der Trapezregel bei η Stützstellen in Abhängigkeit von η zu berechnen.

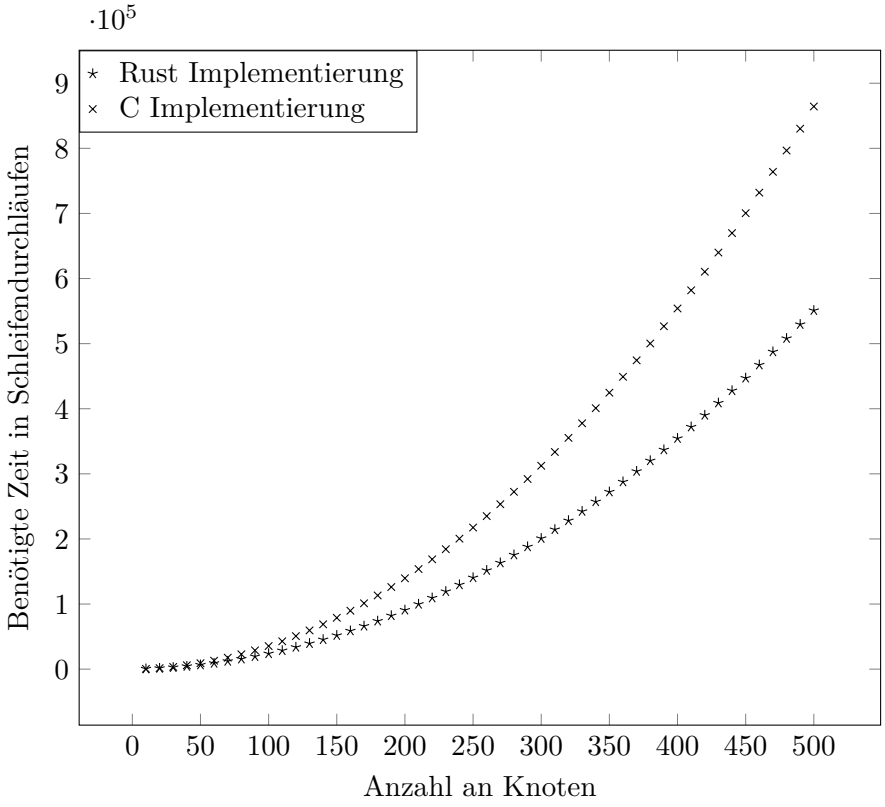


Abbildung 3.5: Dauer der Ausführung der Implementierungen von Dijkstras Algorithmus in C und Rust in Abhängigkeit von der Anzahl an Knoten.

die Verwendung von Iteratoren ausgeschlossen wird, kann dies ebenfalls als Vorteil Rusts gewertet werden, obwohl durch ihre Nutzung ein allgemeiner, aber kleinerer Geschwindigkeitsnachteil entsteht.

Darüber hinaus sind die Implementierungen der Rust Bibliotheken oft noch jung, sodass es Stellen geben kann, die noch nicht ausreichend optimiert wurden. Beispielsweise gibt es eine Problemmeldung, in der die Geschwindigkeit der von mir genutzten `step_by` Methode von Reihen als zu gering eingestuft wird [62]. Solche Meldungen könnten in Zukunft noch behoben werden, wodurch die negativen Geschwindigkeits-Einflüsse Rusts noch weiter verringert werden könnten.

Meiner Meinung nach sind die Nachteile Rusts in diesem Abschnitt, unter anderem aufgrund der soeben aufgeführten Punkte, sowie der ohnehin hohen Schwankungen, eher gering. Falls bestimmte Geschwindigkeitskriterien erfüllt werden müssen, so kann in Rust ebenfalls eine C-ähnliche Programmierung genutzt werden. Daneben können besonders kritische Stellen auch in C oder Inline Assembly geschrieben werden, ohne für den Rest des Programms auf andere Vorteile Rusts verzichten zu müssen.

3.2 Zusammenfassung

In dieser Arbeit wurde nach einer kurzen Betrachtung alternativer Programmiersprachen und Laufzeitumgebungen, die Eignung der Sprache Rust zur Entwicklung auf Mikrocontrollern anhand der Kriterien Hardwareunterstützung, Produktivität, Geschwindigkeit und Zuverlässigkeit, sowie Wartbarkeit untersucht. Zur Bewertung hinsichtlich einiger Kriterien wurden dafür unterschiedliche eigene Implementierungen angefertigt.

Dabei wurden die folgenden Ergebnisse ermittelt:

- Die Hardwareunterstützung lässt sich als insgesamt gut, aber ausbaufähig einstufen. So werden nur wenige relevante ISAs vom Compiler unterstützt, die sich jedoch mit der Zeit ausweiten könnten. Im Gegensatz zu Sprachen mit Laufzeitumgebungen lassen sich jedoch auch ohne besondere Optimierungen problemlos Systeme mit nicht mehr als 32 KiB Flash-Speicher programmieren.
- Die Produktivität scheint zwiegespalten. Zum einen bieten nützliche Sprachkonstrukte und Einflüsse unterschiedlicher Programmierparadigmen einen deutlichen Vorteil gegenüber relativ beschränkten Sprachen wie C. Auf der anderen Seite erhöhen Konzepte, wie Lebenszeiten und Regeln für Referenzen, mit teilweise einzigartiger Syntax und Semantik, das Risiko für Hürden zum Erlernen der Sprache und gelegentlichen Ärgernissen in der täglichen Verwendung.
- Wird die Ausführungsdauer mit C verglichen, so müssen in vielen Fällen kleine Abstriche in Kauf genommen werden. Je nach Algorithmus kann diese aber auch deutlich besser oder schlechter ausfallen. Zur zielgerichteten Optimierung kann ein C-ähnlicher Stil oder Inline Assembly genutzt werden.
- Bei der Zuverlässigkeit und Wartbarkeit kann Rust dagegen mit einer Glanzleistung punkten. So werden ganze Fehlerklassen zur Übersetzungszeit ausgeschlossen, es gibt Standardkonzepte zur einfachen Auswertung von anderen Laufzeitfehlern und keine Möglichkeit zum Auftreten von undefiniertem Verhalten innerhalb sicheren Rust

Codes. Sollten die dafür nötigen Beschränkungen an das Programm zu knapp werden, wie es grade in der eingebetteten Programmierung wahrscheinlich ist, lassen sich diese kritischen Bereiche sauber vom Rest des Quellcodes trennen, ohne für diesen Garantien opfern zu müssen.

3.3 Fazit

Insgesamt sehe ich für die Sprache Rust deutliches Potential eine breite Verwendung in der Mikrocontrollerprogrammierung zu finden. Durch ihr einzigartiges Speicherkonzept hebt sie sich von anderen Alternativen ab, sowohl in positiver, als auch negativer Hinsicht. Die meisten der negativen Punkte lassen sich allerdings entkräftigen, wenn man bedenkt, dass hierdurch Potenzial für fehlerfreiere und sicherere Anwendungen geschaffen wird. Diese Eigenschaften werden sich im ökonomischen Bereich durch geringen Arbeitsaufwand bei der Beseitigung von Problemen und zufriedenere Kunden wiederspiegeln.

Außerdem können die Schwierigkeiten Rusts dem Entwickler insbesondere dann hinderlich werden, wenn er keine klare und hierarchische Speicherstruktur entwirft. Genau an diesem Punkt scheint zwar die Codierung in C einfacher, allerdings ist grade hier das Risiko für Fehlerquellen enorm hoch. Damit ein Programm in Rust kompiliert, wird der Programmierer dazu gezwungen sich mit der Speicherstruktur seines Quellcodes intensiv auseinanderzusetzen. Dies sollte grade in C/C++ ebenfalls zutreffen, jedoch wird hier eine solche Überprüfung nahezu unmöglich.

3.4 Ausblick

Bestimmte Einzelfallentscheidungen, ob ein Projekt mit Rust umgesetzt werden soll, müssen natürlich separat den vorliegenden Anforderungen und Umständen entsprechend getroffen werden. Diese Arbeit sollte also vor allem dazu dienen Rust als potenziellen Kandidaten für diese Entscheidung in Betracht zu ziehen.

Auch die Auswertung ist hier nur als Gesamtüberblick zu verstehen. Die praktischen Eindrücke stammen vor allem aus dem Test mit einem einzigen Mikrocontroller, der nur schwer die Fülle an Variationen von Systemen wiederspiegeln kann, die in der Praxis anzutreffen sind. Über die Unterschiede zur Geschwindigkeit von Rust-Implementierungen, sowie die Produktivität der Entwickler gegenüber anderen Sprachen ließen sich, darüber hinaus, jeweils eigene Arbeiten anfertigen, die den genauen Einfluss bestimmter Sprachfunktionen und Designentscheidungen beurteilen könnten.

Schließlich sind diese Bewertungen nicht allgemeingültig, da sich um die Sprachen Ökosysteme bilden, die stetig in Bewegung sind. Dies trifft zur Zeit vor allem auf Rust zu, sodass Änderungen an Bibliotheken oder die Unterstützung zusätzlicher Systeme denkbar sind. Des Weiteren sind weitergehende Optimierungen und Anpassungen der Sprache möglich. Aber auch C ist keine tote Sprache, sondern wird durch neue Standardisierungen ergänzt. Änderungen an Bibliotheken o. ä. sind hier natürlich ebenfalls denkbar.

Literaturverzeichnis

[1] IEEE Spectrum. Interactive: The Top Programming Languages 2018. URL: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. [Online; zugegriffen am 23.05.2019].

[2] TIOBE. TIOBE Index for May 2019. URL: https://www.tiobe.com/tiobe-index/. [Online; zugegriffen am 23.05.2019].

[3] Ben Frederickson. Ranking Programming Languages by GitHub Users. URL: https://www.benfrederickson.com/ranking-programming-languages-by-github-users/. [Online; zugegriffen am 24.05.2019].

[4] Aspencore. 2017 Embedded Markets Study. URL: https://m.eet.com/media/1246048/2017-embedded-market-study.pdf. [Online; zugegriffen am 23.05.2019].

[5] Eclipse IoT Working Group, AGILE IoT, IEEE, and Open Mobile Alliance. IoT Developer Survey 2018. URL: https://www.slideshare.net/kartben/iot-developer-survey-2018. [Online; zugegriffen am 23.05.2019].

[6] Research Report Insights. Embedded Systems Market to Register High Revenue Growth at 6.1% CAGR During Forecast by 2024. URL: https://www.pr-inside.com/embedded-systems-market-to-register-high-revenue-g-r4752665.htm. [Online; zugegriffen am 27.05.2019].

[7] Crystal Market Research. Embedded Systems Market by Product and Application - Global Industry Analysis and Forecast to 2022. URL: https://www.crystalmarketresearch.com/report/embedded-systems-market. [Online; zugegriffen am 27.05.2019].

[8] Gordon Bell. Moore's Law evolved the PC industry; Bell's Law disrupted it with players, phones, and tablets: New Platforms, tools, and sevices. Technical report, January 2014.

[9] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3):46–54, March 2011.

[10] Rust Team. Webseite der Programmiersprache Rust. URL: https://www.rust-lang.org/. [Online; zugegriffen am 23.05.2019].

[11] Brian Kernighan and Dennis Ritchie. The C Programming Language. 2. Auflage. 1988. ISBN: 0131103709.

[12] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to

Dangling Pointers. URL: http://www.fort-knox.org/files/freesentry.pdf. [Online; zugegriffen am 26.05.2019].

[13] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving Software Security with a C Pointer Analysis. URL: https://suif.stanford.edu/papers/icse05.pdf. [Online; zugegriffen am 26.05.2019].

[14] Valgrind Developers. Webseite des Projektes Valgrind. URL: http://www.valgrind.org/. [Online; zugegriffen am 24.05.2019].

[15] International Organisation for Standardization and International Electrotechnical Commission. ISO/IEC 9899:2018. URL: https://www.iso.org/standard/74528.html. [Online; zugegriffen am 24.05.2019].

[16] Standard C++ Foundation. Webseite der Programmiersprache C++. URL: https://isocpp.org/. [Online; zugegriffen am 24.05.2019].

[17] Charles Wallace. The Semantics of the C++ Programming Language. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.6692&rep=rep1&type=pdf, 1993. [Online; zugegriffen am 24.05.2019].

[18] Bjarne Stroustrup. Why C++ is not just an Object-Oriented Programming Language. URL: https://www.win.tue.nl/~evink/education/avp/pdf/c++-not-just-an-oopl.pdf. [Online; zugegriffen am 24.05.2019].

[19] Bjarne Stroustrup. C++11 - the new ISO C++ standard. URL: http://www.stroustrup.com/C++11FAQ.html. [Online; zugegriffen am 24.05.2019].

[20] Todd L. Veldhuizen. C++ Templates are Turing Complete. URL: https://web.archive.org/web/20060208052020/http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf. [Online; zugegriffen am 24.05.2019].

[21] Oracle Corporation. Java Documentation. URL: https://docs.oracle.com/en/java/index.html. [Online; zugegriffen am 24.05.2019].

[22] Python Software Foundation. Webseite der Programmiersprache Python. URL: https://www.python.org/. [Online; zugegriffen am 24.05.2019].

[23] Haskell.org Infrasturcture. Webseite der Programmiersprache Haskell. URL: https://www.haskell.org/. [Online; zugegriffen am 24.05.2019].

[24] Miran Lipovača. Learn You a Haskell for Great Good! ISBN: 978-1-59327-283-8. URL: http://learnyouahaskell.com/.

[25] George Robotics Limited. Webseite des Projektes MicroPython. URL: https://micropython.org/. [Online; zugegriffen am 20.05.2019].

[26] Timothy C Gregory. Experimentation with Inexpensive Internet of Things (IoT) Modules: A Thermometer Using LoRaWAN. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/1061722.pdf. [Online; zugegriffen am 20.05.2019].

[27] Ravi Kishore Kodali and Kopulwar Shishir Mahesh. Low Cost Ambient Monitoring using ESP8266. 2016. National Institute of Technology, Warangal. DOI: 10.1109/IC3I.2016.7918788.

[28] eLua Project. Webseite des Projektes eLua. URL: http://www.eluaproject.net/. [Online; zugegriffen am 28.05.2019].

[29] eLua Project. eLua Frequently Asked Questions. URL: http://wiki.eluaproject.net/FAQ. [Online; zugegriffen am 28.05.2019].

[30] Rust Embedded Devices Working Group. The Embedded Rust Book. URL: https://docs.rust-embedded.org/book/index.html. [Online; zugegriffen am 21.05.2019].

[31] Rust Embedded Devices Working Group. Discovery. URL: https://docs.rust-embedded.org/discovery/index.html. [Online; zugegriffen am 21.05.2019].

[32] Rust Embedded Devices Working Group. The Embedonomicon. URL: https://docs.rust-embedded.org/embedonomicon/index.html. [Online; zugegriffen am 21.05.2019].

[33] Jorge Aparicio Rivera, Marcus Lindner, and Per Lindgren. Heapless: Dynamic Data Structures without Dynamic Heap Allocator for Rust. Technical Report 10.1109/INDIN.2018.8472097, Luleå University of Technology, 2018.

[34] Tock Embedded OS. Webseite des Projektes Tock. URL: https://www.tockos.org/. [Online; zugegriffen am 21.05.2019].

[35] Wicher Heldring. An RTOS for embedded systems in Rust. University of Amsterdam. URL: https://esc.fnwi.uva.nl/thesis/centraal/files/f155044980.pdf, 2018. [Online; zugegriffen am 21.05.2019].

[36] TinyOS. Webseite des Projektes tinyOS. URL: http://tinyos.net/. [Online; zugegriffen am 21.05.2019].

[37] STMicroelectronics. STM32CubeF1. URL: https://www.st.com/en/embedded-software/stm32cubef1.html. [Online; zugegriffen am 26.05.2019].

[38] Danial Egger and Jorge Aparicio. Crate: stm32f1xx-hal. URL: https://crates.io/crates/stm32f1xx-hal. [Online; zugegriffen am 26.05.2019].

[39] Jorge Aparicio and The Cortex-M Team. Crate: cortex-m. URL: https://crates.io/crates/cortex-m. [Online; zugegriffen am 26.05.2019].

[40] George Kontsevich. Low Level API - for STM32F1 chips. URL: https://github.com/geokon-gh/stm32f1-ll/. [Online; zugegriffen am 26.05.2019].

[41] Mike Mol. Webseite des Projektes Rosetta Code. URL: https://rosettacode.org/. [Online; zugegriffen am 26.05.2019].

[42] Rust Embedded Devices Working Group. Frequently Asked Questions. URL: https://docs.rust-embedded.org/faq.html. [Online; zugegriffen am 26.05.2019].

[43] The AVR-Rust project. A fork of the Rust programming language with AVR support. URL: https://github.com/avr-rust/rust. [Online; zugegriffen am 26.05.2019].

[44] Espressif Systems. Work-in-progress version of LLVM for Xtensa. URL: https://github.com/espressif/llvm-xtensa. [Online; zugegriffen am 26.05.2019].

[45] Eitan Mosenkis. Script for installing/running toolchain for building ESP8266 firmware in Rust. URL: https://github.com/emosenkis/esp-rs. [Online; zugegriffen am 26.05.2019].

[46] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. IETF RFC 7228. URL: https://tools.ietf.org/html/rfc7228. [Online; zugegriffen am 26.05.2019].

[47] STM32duino Wiki. Blue Pill. URL: https://wiki.stm32duino.com/index.php?title=Blue_Pill. [Online; zugegriffen am 28.05.2019].

[48] Zahra Karimi, Ahmad Baraani-Dastjerdi, Nasser Ghasem-Aghaee, and Stefan Wagner. Links between the Personalities, Styles and Performance in Computer Programming. 2016. Journal of Systems and Software. 111: 228-241. DOI: 10.1016/j.jss.2015.09.011.

[49] Stack Exchange Inc. Stack Overflow Developer Survey 2019. URL: https://insights.stackoverflow.com/survey/2019. [Online; zugegriffen am 23.05.2019].

[50] Nicholas Matsakis and Aaron Turon. The Rust Programming Language. 2. Auflage. ISBN: 9781593278281; URL: https://doc.rust-lang.org/1.30.0/book/second-edition/. [Online; zugegriffen am 26.05.2019].

[51] Rust Team. The Cargo Book. URL: https://doc.rust-lang.org/cargo/index.html. [Online; zugegriffen am 26.05.2019].

[52] Jorge Aparicio. Webseite des Tools xargo. URL: https://github.com/japaric/xargo. [Online; zugegriffen am 26.05.2019].

[53] Manuel Hoffmann. Are we (I)DE yet? URL: https://areweideyet.com/. [Online; zugegriffen am 26.05.2019].

[54] The GDB Developers. GDB: The GNU Project Debugger. URL: https://www.gnu.org/software/gdb/. [Online; zugegriffen am 26.05.2019].

[55] Dominic Rath. Webseite des Projektes OpenOCD. URL: http://openocd.org/. [Online; zugegriffen am 26.05.2019].

[56] Jonathan Pallant and Jorge Aparicio. Crate: embedded-hal. URL: https://crates.io/crates/embedded-hal. [Online; zugegriffen am 26.05.2019].

[57] Rust Team. The Unstable Book. URL: https://doc.rust-lang.org/unstable-book/the-unstable-book.html. [Online; zugegriffen am 26.05.2019].

[58] Rust Lang Nursey Team. Crate: log. URL: https://crates.io/crates/log. [Online; zugegriffen am 27.05.2019].

[59] Maciej Hirsz. Crate: json. URL: https://crates.io/crates/json. [Online; zugegriffen am 27.05.2019].

[60] Steve Klabnik and Carol Nichols. The Rust Programming Language. Edition 2018. URL: https://doc.rust-lang.org/stable/book/. [Online; zugegriffen am 26.05.2019].

[61] Rust Team. The Rust RFC Book. URL: https://rust-lang.github.io/rfcs/. [Online; zugegriffen am 26.05.2019].

[62] Github-Nutzer wooleyra. Iter with step_by(2) performs slowly. URL: https://github.com/rust-lang/rust/issues/59281. [Online; zugegriffen am 26.05.2019].

Hiermit versichere ich, dass die vorliegende Arbeit mit dem Titel *Evaluierung der Sprache Rust zur Programmierung von Mikrocontrollern* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Magdeburg, 28. Mai 2019

(Tom Heimbrodt)