

Bachelorarbeit

Entwurf und Implementierung einer
hochperformanten, serverbasierten
Kommunikationsplattform für Sensordaten
im Umfeld des automatisierten Fahrens in Rust

Michael Watzko

Sommersemester 2018
14.02.2018 - 22.06.2018

Erstprüfer: Prof. Dr. rer. nat. Dipl.-Inform. Manfred Dausmann
Zweitprüfer: ... Hannes Todenhagen



Firma: IT Designers GmbH
Betreuer: Dipl. Ing. (FH) Kevin Erath M.Sc.

Sperrvermerk

U SHALL NOT PASS

Ehrenwörtliche Erklärung

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

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

Esslingen, den 22. Februar 2018

Michael Watzko

Danksagungen

„Alle Zitate aus dem Internet sind wahr!“

Albert Einstein

„Rust is a vampire language, it does not reflect at all!“

<https://www.youtube.com/watch?v=-Tj8Q12DaEQ>

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Projektkontext	2
1.2.1	Ablauf	3
1.2.2	Sicherheit	3
1.3	Zielsetzung	3
1.4	Aufbau der Arbeit	4
2	Die Programmiersprache Rust	5
2.1	Geschichte	5
2.2	Anwendungsgebiet	6
2.3	Aufbau eines Projektverzeichnisses	7
2.3.1	Klassisch	7
2.3.2	Als Crate	7
2.4	Hello World	8
2.5	Standardbibliothek	9
2.5.1	core	9
2.5.2	std	10
2.6	Alles hat einen Rückgabewert	10
2.7	use mod pub	10
2.8	Scope / Memory Management, Lebenszeit	10
2.9	Eigentümer- und Verleihprinzip	11
2.10	Rust als funktionale Programmiersprache	13
2.11	Rust als Objekt-Orientierte Programmiersprache	13
2.12	Versprechen von Rust	13
2.12.1	Sichere Nebenläufigkeit	13
2.12.2	Zero Cost Abstraction	13
2.12.3	Kein undefiniertes Verhalten	13
2.12.4	Keine vergessene Null-Pointer Prüfung	13
2.12.5	Keine vergessene Fehlerprüfung	14
2.12.6	No dangling pointer	15
2.13	Einbinden von Bibliotheken	15
2.14	Kernfeatures	18
2.15	Schwächen	18
2.16	Performance Fallstricke	18

2.17 Beispiele von Verwendung von Rust	18
3 Stand der Technik (c++ Version)	20
3.1 Hochperformant -> parallel?	20
3.2 Serverbasierte Kommunikationsplattform	20
3.3 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?	20
3.4 ASN.1	20
3.5 PER	20
3.6 MEC-View Server und Umgebung	20
4 Anforderungen	21
4.1 Funktionale Anforderungen	21
4.2 Nichtfunktionale Anforderungen	21
4.3 Kein Protobuf weil	21
5 Systemanalyse	22
5.1 Systemkontextdiagramm	22
5.2 Schnittstellenanalyse	22
5.3 C++ Referenzsystem	22
6 Systementwurf	23
6.1 Änderungen bedingt durch Rust	23
7 Implementierung	24
8 Auswertung	25
9 Zusammenfassung und Fazit	I
Literatur	II
Glossary	IV
Abkürzungsverzeichnis	V
Abbildungsverzeichnis	VI

1 Einleitung

1.1 Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Tesla Autos einen allgemeinen Bekanntheitsgrad erreicht. Um ein Auto selbstständig fahren lassen zu können, müssen erst viele Hürden gemeistert werden, zum Beispiel das Spur halten, auch bei fehlenden Fahrspurmarkierungen, das Interpretieren von Stoppschildern und navigieren durch komplexen Kreuzungen. **TODO: ref tesla.com?**

Bevor das Auto Entscheidungen treffen kann, muss es zuallererst ein Modell seines Umfelds erstellen oder zur Verfügung gestellt bekommen. Aber vielleicht kann ein Auto nicht immer selbstständig genügend Informationen zu seinem Umfeld sammeln? **TODO: (huhuhu Server implied huhuhu) TODO: fix 404**

1.2 Projektkontext

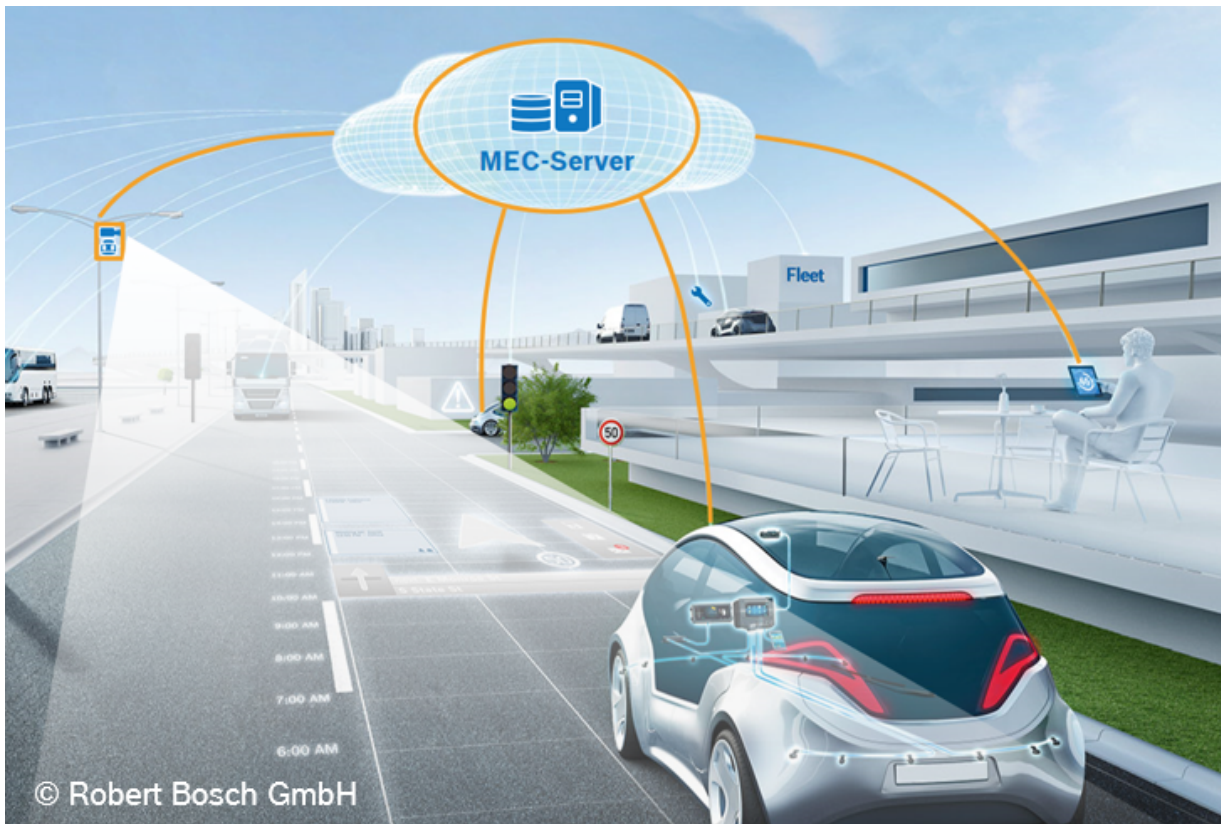


Abbildung 1.1: MEC-View Schaubild der Robert Bosch GmbH [MEC]

Quelle: https://www.uni-due.de/~hp0309/images/Schaubild_BoschStyle_V2.png

Diese Abschlussarbeit befasst sich mit dem MEC-Server, der Teil des MEC-View Forschungsprojekt ist. Das MEC-View Projekt wird durch das BMWi gefördert und befasst sich mit der Thematik autonom fahrender Fahrzeuge. Es soll erforscht werden ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist um in eine Vorfahrtsstraße autonom einzufahren.

Das Forschungsprojekt ist dabei ein Zusammenschluss verschiedener Unternehmen:

Unternehmen	Aufgabenbereich
Bosch	Hochautomatisiertes Fahrzeug
Osram	„Intelligente“ Infrastruktursensoren
Nokia	5G Mobilfunk, Mobile Edge Computing (MEC)
Universität Ulm	Sensordatenfusion, Prädiktion
IT-Designers Gruppe	MEC-Server Architektur Mikroskopische Verkehrsanalyse Verhaltensanalyse
TomTom	Hochgenaue statische und dynamische Karten
Daimler	Fahrstrategien Verhaltensanalyse Streckenfreigabe
Universität Duisburg	Mikroskopisch, stochastische Simulationsmodelle

1.2.1 Ablauf

Externe Sensoren übermitteln erkannte Fahrzeuge via Mobilfunk an einen [MEC-Server](#), der direkt am Empfängerfunkmast angeschlossen ist. **TODO: platform, vm?** Nachdem die erkannten Fahrzeuge der verschiedenen Sensoren zusammengeführt wurden (Fusions-Algorithmus), sollen sie an das autonom fahrende Fahrzeug über Mobilfunk übermittelt werden. Somit erhält das Fahrzeug bereits im Voraus Einsicht über eventuelle Möglichkeiten in die Vorfahrtsstraße einzufahren und könnte deshalb beispielsweise die Geschwindigkeit anpassen. Zudem sollen bei unübersichtlichen Kreuzungen somit zuverlässiger andere Verkehrsteilnehmer erkannt werden.

1.2.2 Sicherheit

TODO: überhaupt relevant? Da bei Fehlern möglicherweise andere Verkehrsteilnehmer zu Schaden kommen können, müssen diverse Sicherheitsrichtlinien beachtet werden. Die Industrienorm ISO 26262 beschreibt dabei verschiedene Vorgehensweisen, unter anderem eine [FBA](#), Risikoabschätzung durch Einstufung nach [SILs](#) und beschreibt Gegenmaßnahmen.

1.3 Zielsetzung

Das Ziel ist es, eine alternative Implementierung des [MEC-View Servers](#) in Rust zu schaffen. Durch die Garantien ([Abschnitt 2.12](#)) von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert wird und somit eine fehlertolerantere und sicherere Implementation geschaffen werden kann.

TODO: ? Für eine bessere Wartbarkeit und Nachvollziehbarkeit soll die Implementation in Ihrer **TODO: Struktur/**Architektur der C++ Implementation ähneln.

1.4 Aufbau der Arbeit

Diese Arbeit ist im wesentlichen in die folgenden Themengebiete aufgeteilt: Grundlagen, Anforderungs- und Systemanalyse, Systementwurf und Implementation und Auswertung.

Im Themengebiet Grundlagen sollen wesentliche Bestandteile dieser Arbeit erläutert und erklärt werden. Hierzu zählt zum einen die Programmiersprache Rust in ihrer Entstehungsgeschichte **TODO: ref**, Garantien **TODO: ref** und Sprachfeatures **TODO: ref**, zum anderen die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen **TODO: ref** und dem Systemkontext in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext in dem das System betrieben werden soll genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt sowie eine Übersicht von Systemen mit denen interagiert wird.

Das Themengebiet Systementwurf und Implementation befasst sich mit dem theoretischen und praktischen Lösen der im vorherigen Kapitel aufgestellten Anforderungen. Aufgrund der Tatsache, dass es sich hierbei um eine alternative Implementation handelt, wird zur bestehenden C++ Implementation Bezug genommen. Auf architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt. **TODO: michael.write_more();**

2 Die Programmiersprache Rust

TODO: quotation besser verpacken

„[...]Leute, die [...] sichere Programmierung haben wollen, [...] können das bei Rust haben, ohne die [von D] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen.“ [Lei17]

„It’s not bad programmers, it’s that C is a hostile language“ [Qui, S. 54]

„I’m thinking that C is actively hostile to writing and maintaining reliable code“ [Qui, S. 129]

„[...] Rust makes it safe, and provides nice tools“ [Qui, S. 130]

„Rust hilft beim Fehlervermeiden“ [Grü17]

„Rust is [...] a language that cares about very tight control“ [fgi17]

Rust ist eine Programmiersprache, die versucht performant – und daher durch Abstraktionen mit keinem zusätzlichen „Kosten“ **TODO: ref zero cost abstractions** – sichere Programmierung zu ermöglichen. Ziel ist eine **TODO: Systemprogrammiersprache**, die sicher **TODO: cite chapter** und performant ist, ohne eine Laufzeit ausgeführt werden kann und **TODO: ergonomisch nutzbar** ist. Verschiedene Fehlerquellen – wie „dangling pointers“, „double free“ oder „memory leaks“ **TODO: ref** – werden durch Abstraktionen und mit Hilfe des Compilers verhindert. Anders als Programmiersprachen, die dies mit Hilfe einer Laufzeit ermöglichen (zbsp. Java oder C#), wird dies in Rust durch eine statische Lebenszeitanalyse (2.8) und einem Eigentümerprinzip (2.9) bei der Compilation gewährleistet.

TODO: unused orly rust [Bla15]

2.1 Geschichte

In 2006 [Rusa] begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Code zu schreiben. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“. [Sch13]

Auch Federico Mena-Quintero – Mitbegründer des Gnome projekts **TODO: cite** <https://people.gnome.org/~federico/> **or so** – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [Grü17]. In Vorträgen **TODO: nix mehrzahl?** vermittelt er seither, wie Bibliotheken durch Implementationen in Rust ersetzt werden können [Qui].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, als einfache Tests und die Kernprinzipien demonstriert werden konnten. Die Entwicklung findet dabei öffentlich einsehbar auf GitHub unter <https://github.com/rust-lang/rust> statt und wird dabei nicht ausschließlich von Mozilla Angestellten koordiniert. Die Stabilität des Compilers trotz hoher Flexibilität während der Entwicklung wird durch Unterscheidung von drei Veröffentlichungskanälen – release, stable und nightly – in Kombination mit automatisierten Tests **TODO: ref?** gewährleistet. [Rusa]

TODO: wo anders? make more text, make better text Bei Rust geht es also in vielerlei Hinsicht darum, Fehlerquellen aus anderen Programmiersprachen (C, C++) zu unterbinden, aber gleichzeitig genau so schnell Ausführen zu können. **TODO: hobbyprojekt, mozilla, open-source, Entwicklung auf GitHub - jeder kann sich beteiligen, test(coverage), automatisierte builds, stable/beta/nightly**

2.2 Anwendungsgebiet

Das Ziel von Rust ist es, das Designen und Implementieren von sicheren, nebenläufig und auch praktisch tauglichen Systemen möglich zu machen [Rusa]. **TODO: intro paragraph**

Da Rust den LLVM¹-Compiler nutzt, erbt Rust auch eine große Anzahl der Zielplattformen die LLVM unterstützt. Die Zielplattformen sind in drei Stufen unterteilt, bei denen verschieden stark ausgeprägte Garantien vergeben sind. Es wird zwischen

- „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64),
- „Stufe 2: Compiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und
- „Stufe 3“ (u. a. Thumb)

unterschieden [Rusb]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [atu]).

¹ Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Compiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplattformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb].

2.3 Aufbau eines Projektverzeichnis

2.3.1 Klassisch

```

1 src/
2 |-- main.rs
3 |-- functionality.rs
4 |-- module/
5     |-- mod.rs
6     |-- functionality.rs
7     |-- submodule/
8         |-- mod.rs
9         |-- functionality.rs

```

Listing 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager Cargo ([Unterabschnitt 2.3.2](#)) eine solche Benennung als Standardkonvention erwartet, kann bei manueller Nutzung des Compilers auch ein anderer Name für die Quelldatei vergeben werden.

Der Compiler startet in der Wurzeldatei und lädt weitere Module, die durch `mod module;` gekennzeichnet sind (ähnlich `#include "module.h"` in C/C++). Ein Modul kann dabei eine weitere Quelldatei oder ein ganzes Verzeichnis sein. Ein Verzeichnis wird aber nur als

gültiges Modul interpretiert, wenn sich eine *mod.rs* Datei darin befindet.

Wie bereits angedeutet, wird in Rust nicht eine „Klasse“, Datenstruktur oder Aufzählung pro Datei erwartet (**TODO: wie das bei Java der Fall ist**), sondern eine Quelldatei entspricht einem Modul. Dies umfasst in vielen Fällen wenige aber mehrere Datenstrukturen, zugehörige Aufzählung und Fehlertypen.

2.3.2 Als Crate

TODO: text is shit Im Gegensatz zu einem klassischen Aufbau ([Unterabschnitt 2.3.1](#)) wird von der Rust **TODO: Gemeinschaft** das Werkzeug „Cargo“ (dt. Fracht/Ladung **TODO: .**) angeboten. Mit Cargo können ähnlich wie zum Beispiel mit Maven **TODO: cite?** in Java, Abhängigkeiten zu anderen Bibliotheken verwaltet werden. Ein Cargo Projekt wird dabei als „Crate“ (dt. Kiste/Kasten **TODO: .**) bezeichnet. Eine offizielle Verzeichnis befindet sich auf <https://crates.io/>. Von <https://crates.io/> werden standardmäßig Abhängigkeiten nachgeladen. Jeder kann neue Bibliotheken hochladen/veröffentlichen, für den Namen gilt dabei „first come, first serve“.

```

1 crate/
2 |-- Cargo.toml
3 |-- src/
4     |-- ...

```

Listing 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

TODO: text blow is shit Eine Crate kann entweder als Ziel ein ausführbares Programm oder eine Bibliothek haben. Davon abhängig wird die Wurzeldatei für ein ausführbares Programm in *src/main.rs* und für eine Bibliothek in *src/lib.rs* erwartet. Mit dem erzeuge einer Crate () `cargo --bin meineCrate` bzw. `cargo --lib meineBib` wird auch gleichzeitig [git](#)² für das Verzeichnis initialisiert.

Projekte als Crates zu **TODO: benutzen** ist der empfohlene Standard **TODO: cite**.

TODO: dependencies

TODO: Cargo init -bin <name>

TODO: missing .gitignore / .git mention / git altogether

TODO: Cargo.toml

TODO: [crates.io], Anzahl Pakete

TODO: Compilierablauf, downloaden, compilieren von crates

2.4 Hello World

```

1 fn main() {
2     println!("Hello World");
3 }

```

Listing 2.3: „Hello World“ in Rust

TODO: let, optionaler datentyp, macros, generics, () statt void

TODO: official format/naming convetion, use, function, macro

TODO: Variables, Structs, Enums, Traits

TODO: type safety langauge

TODO: Rust -> MIR -> assembler

TODO: MIR/assemblerbeispiele?

[[Jim17](#)]

TODO: pattern matching

TODO: formatierung

² (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005. **TODO: cite**

2.5 Standardbibliothek

Die Rust Community ist darum bemüht, die Standardbibliothek sehr leichtgewichtig zu halten. Sie wird selbst als eine Crate (siehe [Unterabschnitt 2.3.2](#)) zur Verfügung gestellt, auf die standardmäßige Abhängigkeit besteht. Für die Verwendung von Rust im Embedded Bereich, kann diese Abhängigkeit, die für Microcontroller sehr umfangreich ist, durch `#![no_std]` unterbunden werden. Daraufhin sind nur noch die in der `core` Crate zur Verfügung gestellten Sprachkonstrukte verwendbar.

2.5.1 core

Die in der `core` Crate zur Verfügung gestellten Sprachkonstrukte sind im wesentlichen die üblichen Verdächtigen: `bool` für boolische Ausdrücke; `char` für ein einzelnes Unicode Zeichen; `str` für eine Zeichenkette; `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, (bald `u128`, `i128` [TODO: cite](#)) und `usize`, `isize` für ganze Zahlen; `f32`, `f64` für Fließkommazahlen in einfacher und zweifacher Präzision; Arrays und Slices [\[Rusc\]](#).

Ganzzahlige primitive Datentypen mit `u` beginnend sind vorzeichenlos und mit `i` beginnend sind vorzeichenbehaftet, gefolgt mit der Anzahl der Bits die der Datentyp groß ist. [TODO: shit sentence](#) Die einzige Ausnahme bildet der Datentyp `usize` bzw `isize`, da dieser immer so groß ist, wie die Architektur der Zielplattform (X86 -> 32 Bit, X86_64 -> 64 Bit). Ein Anwendungsfall von `usize` ist dabei die Indexierung eines Arrays oder einer Slice ([TODO: siehe nächster paragraph?](#)), da der Index hierfür niemals negativ und niemals größer sein kann, wie die Architektur der Zielplattform darstellen kann [TODO: erwähnen?: größer könnte man garnicht adressieren](#).

Durch dieses Schema bei der Bezeichnung der Datentypen wird eine Verwirrung wie zum Beispiel in C unterbunden, wo die primitiven Datentypen (`short`, `int`, `long`, ..) keine definierte Größe haben, sondern dies abhängig vom eingesetzten Compiler und der Zielplattform ist [\[DD13, S. 187\]](#). Erst ab C99 wurden zusätzliche, aber optionale, ganzzahlige Datentypen mit bestimmter Größe definiert [\[GD14, S. 141\]](#).

Konstanten können eindeutig einem Datentyp zugewiesen werden, indem dieser angehängt wird. `4711u16` ist somit vom Datentyp `u16`. Des weiteren dürfen Ziffern durch beliebiges setzen von `_` getrennt werden, um die Lesbarkeit zu erhöhen: `1_000_000_f32`. Eine Schreibweise in Binär (`0b0000_1000_u8`), in Hexadezimal (`0xFF_08_u16`) oder Oktal (`0o64_u8`) ist auch möglich. Konstante Zeichen und Zeichenketten können auch als Bytes (`b'b'` entspricht `u8` und `b"abc"` entspricht `&[u8]`) [TODO: hinterlegt](#) werden.

Arrays haben immer eine zur Compilezeit bekannte Größe und Initialisierungswert (siehe [Unterabschnitt 2.12.3](#)). Dynamische Arrays gibt es nicht, da diese zu oft Fehlerquellen seien [TODO: cite!](#) (Abhilfe: `Vec<_>`, siehe [Unterabschnitt 2.5.2](#)). Die Notation ist `[<Füllwert>; <Größe>]`. `[0_u8; 128]` steht also für ein 128 Byte langes Byte Array, das mit 0-en vom Datentyp `u8` gefüllt ist.

„Slices“ (dt. Scheibe/Stück) bezeichnet Rust Referenzen auf Arrays, die auch nur Teilbereiche umfassen können. Die Größe einer Slice wird dabei mit der Referenz auf den Startwert gespeichert **TODO: explain Fat-Pointer?** und bei Funktionsaufrufen übergeben. Ein zusätzlicher Parameter für die Größe eines Buffers, wie in C üblich, ist somit unnötig. Die Notation ähnelt die eines Arrays, aber ohne Größenspezifikation: `[<Datentyp>]`. Eine Slice kann von einem Array oder einer anderen Slice erzeugt werden, dabei wird der Start- und Endindex des Teilbereiches angegeben. Falls kein Start- oder Endindex angegeben wurde, wird das jeweilige Limit übernommen (0, max) **TODO: shit text:** `let slice : &[u8] = &array[..8];`

Rust kennt `null` (-Pointer) nicht, bietet aber in `core` bereits `Option<_>` als Ersatz an. Für die Fehlerbehandlung wird nicht auf ein Exception-Handling zurückgegriffen, sondern ein eigener Datentyp angeboten, der entweder den Rückgabewert enthält, oder aber einen Fehler: `Result<_, _>`. Die Funktionsweise und die Vorteile von `Option<_>` und `Result<_, _>` wird in [Unterabschnitt 2.12.4](#) genauer erklärt.

2.5.2 std

Die Crate `std` erweitert `core` um viele **TODO: collections etc.** `Vec<_>` `HashMap<_, _>` `String` `Box` **TODO: heap dinge**

TODO: core, datatypes, arrays slices, no null „billion dollar mistake“

TODO: std, Vec, str, String, no_std für embedded

TODO: println!, writeln! formatting

<https://www.youtube.com/watch?v=-Tj8Q12DaEQ>

TODO: static type system with local type inference

2.6 Alles hat einen Rückgabewert

TODO: () ??

2.7 use mod pub

2.8 Scope / Memory Management, Lebenszeit

Rust benutzt ein „statisches, automatisches Speicher Management – keinen Garbage Collector“ [\[Gil17\]](#). Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Scopes **TODO: ?** ermittelt. Durch diese statische Analysen wird

eine Variable durch eine automatisierte Anweisung, die der Compiler einfügt, freigeben. Dies gilt auch für Variablen auf dem Heap. Ein Manuelles `free(*void)` wie in C/C++ üblich entfällt.

```

1 fn main() { // neuer Scope
2     let mut a = Box::new(5); // 5 kommt auf den Heap
3     { // neuer Scope
4         let b = Box::new(10); // 10 kommt auch auf den Heap
5         *a += *b; // a ist nun 15
6     } // Lebenszeit von b zuende, Speicher wird freigeben
7     println!("a: {}", a); // Ausgabe: "a: 15"
8 } // Lebenszeit von a zuende, Speicher wird freigegeben

```

Listing 2.4: Scope von Variablen

Als Alternative kann eine Variable oder Datenstruktur auch vorzeitig durch Aufruf von `std::mem::drop(_)` freigeben werden. Die optionalen Implementation von `std::ops::Drop` `TODO: trait? ref?` kommt der Implementation des Destruktors aus C++ gleich.

„static automatic memory management“ - no garbage collection [Gil17] `TODO: static automatic memory management no garbage collection`

`TODO: while compiling, does not compile on error / unprovable code, trait Drop`

`TODO: autodrop, auto file close`

2.9 Eigentümer- und Verleihprinzip

Bereits 2003 beschreibt Bruce Powel Douglass im Buch „Real-Time Design Patterns“, dass „passive“ Objekte ihre Arbeit nur in dem `TODO: Thread-Kontext` ihres „aktiven“ Eigentümers tätigen sollen [Dou03, S. 204]. In dem beschriebenen „Concurrency Pattern“ wird eine klare Zuordnung getätigt, welche Objekte welchem anderen Objekt als Eigentümern zugeordnet sind, um eine sicherere Nebenläufigkeit zu schaffen `TODO: shit`.

Diese Philosophie setzt Rust direkt in der Sprache um, so darf eine Variable immer nur einen Eigentümer haben. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer für eine Variable, kann diese auch ausgeliehen werden; entweder exklusiv mit sowohl Lese- als auch Schreiberlaubnis, oder mehrfache mit nur Leseerlaubnis.

Die Philosophie „jede Variable hat einen Eigentümer“ ist nicht neu `TODO: active -> Eigentümer` [Dou03, S. 204] `TODO: cite!`, aber Rust setzt dies in der Sprache um. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer für eine Variable, kann diese auch

ausgeliehen werden; einmal mit einer exklusiven Schreiberlaubnis, oder mehrfache mit nur Leserechten. **TODO: mutability fehlt** Da eine Schreiberlaubnis immer exklusiv ist, werden **TODO: Data-Races** unterbunden; da dies nur bei mehreren Parteien mit gleichzeitigem Schreib- und Leserecht auftreten kann.

Die Garantie nur einen exklusiven Schreibzugriff oder mehrere Lesezugriffe auf eine Variable zu haben, wird durch die statische Lebenszeitanalyse garantiert ([Abschnitt 2.8](#)).

TODO: Split example, explain more

```

1 fn main() {
2     let mut a = Box::new(1.0_f32); // Eigentümer der neuen
3                                     // Heap-Variable ist a
4
5     {
6         let b = &a; // a wird an b mit Lesezugriff verliehen
7         let c = &a; // a wird an c mit Lesezugriff verliehen
8
9         println!("a: {}", a); // "a: 1"
10        println!("b: {}", b); // "b: 1"
11        println!("c: {}", c); // "c: 1"
12
13        // let d = &mut a; // Nicht erlaubt: Es existieren
14                        // verliehene Lesezugriffe
15
16        // *a = 7_f32; // Nicht erlaubt: Es existieren
17                    // verliehene Lesezugriffe
18
19    } // Ende von b und c, a nicht mehr verliehen
20
21    {
22        let e = &mut a; // Leihe a mit Schreiberlaubnis
23        **e = 9_f32;    // Setze Inhalt von a
24
25        // println!("a: {}", a); // Nicht erlaubt: exklusiver
26                                // Zugriff an e verliehen
27
28        println!("e: {}", e); // "e: 9"
29
30    } // Ende von e, a nicht mehr verliehen
31
32    println!("a: {}", a); // "a: 9"
33    let f = a; // Neuer Eigentümer der Heap-Variable ist f
34    // *a = 12.5_f32; // Nicht erlaubt: Nicht mehr Eigentümer
35    // *f = 12.5_f32; // Nicht erlaubt: f nicht änderlich

```

```
36     println!("f: {}", f); // "f: 9"  
37 }
```

Listing 2.5: Eigentümer und Referenzen von Variablen

2.10 Rust als funktionale Programmiersprache

TODO: functional programming -> no global state, no exceptions, find literature
TODO: prove via code

2.11 Rust als Objekt-Orientierte Programmiersprache

TODO: trait
TODO: prove via design patterns, a few? from faq:: Is Rust object oriented?
It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but not everything, and not always using the same abstraction you're accustomed to.

2.12 Versprechen von Rust

2.12.1 Sichere Nebenläufigkeit

TODO: Send, Sync, No dataraces weil Ownership [Abschnitt 2.9](#), Channel, Mutex, Rw-Lock

2.12.2 Zero Cost Abstraction

2.12.3 Kein undefiniertes Verhalten

TODO: auch: no uninitialized usage
TODO: ref oreilly

2.12.4 Keine vergessene Null-Pointer Prüfung

TODO: explain option

2.12.5 Keine vergessene Fehlerprüfung

```
1 #include <stdio.h>
2
3 void main(void) {
4     FILE *file = fopen("private.key", "w");
5     fputs("42", file);
6 }
```

Listing 2.6: Negativbeispiel: Fehlende Fehlerprüfung in C

In Listing 2.6 sind mindestens zwei Fehler versteckt, die aber keinen Compileabbruch auslösen, sondern sich zur Laufzeit zeigen können. Der erste Fehler ist eine fehlende Überprüfung des Rückgabewertes von `fopen` in Zeile 4, da dieser `null` ist, falls das Öffnen der Datei fehlgeschlagen ist. Der Versuch in die Datei zu schreiben in Zeile 5 kann daraufhin in einen Speicherzugriffsfehler Resultieren und das Programm abstürzen lassen. **TODO: vergleiche Java/C++(+++) exceptions (vergessen von fehlerbehandlung in c++(+++) trotzdem möglich)**

In Rust wird weder eine Ausnahme geworfen, noch ein Rückgabewert zurück gegeben, der ohne Prüfung verwendet werden kann:

```
1 use std::fs::File;
2 use std::io::Write;
3
4 fn main() {
5     match File::open("private.key") {
6         Err(e) => println!("Fehler aufgetreten: {}", e),
7         Ok(mut file) => {
8             let _ = write!(file, "42");
9         }
10    }
11 }
```

Listing 2.7: Positivbeispiel: Keine fehlende Fehlerprüfung in Rust

Der Rückgabewert von `File::open("private.key")` in Zeile 5 von Listing 2.7 ist vom Typ `Result<File, Error>`. Auf den eigentlichen Rückgabewert `File` kann nicht ohne eine Fehlerprüfung zugegriffen werden, da dies `Result` verhindert. Eine Fehlerprüfung

kann wie in Zeile 5 mit einem `match` passieren, oder auch mit anderen Funktionen wie `.unwrap()`, `.unwrap_or()` ... <https://doc.rust-lang.org/std/result/enum.Result.html> die dann aber eine `panic!` `TODO: ref` auslösen, falls ein Fehler vorliegt – somit wird ein undefiniertes Verhalten unterbunden `TODO: ref`.

`TODO: Der zweite Fehler...?` Durch die Lebenszeitanalyse `TODO: ref` in Rust ist der Scope der `File` Variable bekannt, weshalb in dem Rust Beispiel in [Listing 2.7](#) die Datei auch wieder ordnungsgemäß geschlossen, während dies im C Beispiel in [Listing 2.6](#) nicht der Fall ist.

`TODO: explain result`

2.12.6 No dangling pointer

`TODO: src` <https://www.youtube.com/watch?v=d1uraoHM8Gg>

2.13 Einbinden von Bibliotheken

Externe Datentypen

Rust bietet durch das [Foreign Function Interface](#)³ die Möglichkeit, andere (System-)Bibliotheken einzubinden. Entsprechende Strukturen und Funktionen werden durch einen `extern`-Block oder im Falle von Strukturen stattdessen optional mit einem `#[repr(C)]` gekennzeichnet.

In einem Beispiel, soll die Nutzung von [Foreign Function Interface](#) demonstriert werden.

```

1 typedef struct PositionOffset {
2     long position_north;
3     long position_east;
4     long *std_dev_position_north; // OPTIONAL
5     long *std_dev_position_east;  // OPTIONAL
6
7     // ...
8 } PositionOffset_t;
```

Listing 2.8: Ausschnitt von „PositionOffset“ `TODO: ref` `mecview lib` in C, autgen ASN

³ Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer anderen Programmiersprache geschrieben wurden. [[Wik18](#)]

Die Struktur in [Listing 2.8](#) muss zur Nutzung in Rust zuerst bekannt gemacht werden. Dabei gibt es mehrere Möglichkeiten:

1. Falls der Aufbau der Struktur nicht von Bedeutung ist, kann es ausreichen, den Datentyp lediglich bekannt zu machen: `#[repr(C)] struct PositionOffset;`
2. Der Aufbau ist wie bei [Punkt 1](#) unbedeutend, es soll aber ausdrücklich auf einen externen Datentyp hingewiesen werden: `extern { type PositionOffset; }` [[Ruse](#)] ([TODO: nightly](#))
3. Der Inhalt der Struktur ist von Bedeutung, da darauf zugegriffen werden soll oder in Rust eine Instanz erzeugbar sein soll. In diesem Fall muss die Struktur komplett wiedergegeben werden:

```

1 use std::os::raw::c_long;
2
3 #[repr(C)]
4 pub struct PositionOffset {
5     pub position_north: c_long,
6     pub position_east: c_long,
7     pub std_dev_position_north: *mut c_long,
8     pub std_dev_position_east: *mut c_long,
9     // ...
10 }
```

Listing 2.9: Ausschnitt von „PositionOffset“ [TODO: ref mecview lib](#) in Rust

In [Listing 2.9](#) ist die Struktur „PositionOffset“ definiert, die durch das Attribut `#repr(C)` wie eine C-Struktur im Speicher organisiert wird. Somit ist sie kompatibel zu der C-Struktur aus [Listing 2.8](#).

Wenn auf eine C-Struktur zugegriffen wird, sollten auch, wie in [Listing 2.9](#) zu sehen, spezielle Datentypen (`c_long`, `c_void`, `c_char`, ...) verwendet werden, um die Kompatibilität mit verschiedenen Systemen und C-Compilern zu wahren. [TODO: u32 immer 32bit, aber int nicht immer gleich \(Beispiel!?\) -> Probleme](#)

Ein C-Pointer `*long` wird in Rust „Raw-Pointer“ genannt und entweder `*mut c_long` oder `*const c_long` geschrieben. Der Unterschied ist wie zwischen `&mut c_long` und `&c_long` und dient dem [TODO: Rusttypsystem!? ref!?](#) zur Unterscheidung [TODO: Erzwingt im Besitz von entsprechender Mutability zu sein](#), während es für die C-Seite keinen Unterschied macht [[Rusd](#)]:

Referenz in Rust	Raw-Pointer in Rust	C-Pointer
<code>&mut c_long</code>	<code>*mut c_long</code>	<code>long*</code>
<code>&c_long</code>	<code>*const c_long</code>	<code>long*</code>

Abbildung 2.1: Vergleich Rust Raw-Pointer und Referenz zu C-Pointer

Externer Funktionsaufruf

Während eine Struktur, die eine externe Struktur wiedergibt, sich optional in einem `extern {}` Block befinden kann, ist es zwingend, eine externe Funktionen darin bekannt zu machen:

```

1 use std::os::raw::c_void;
2
3 #[link(name = "messages", kind = "static")]
4 extern {
5     type asn_TYPE_descriptor_s;
6     type asn_enc_rval_t;
7
8     fn uper_encode_to_buffer(
9         type_descriptor: *const asn_TYPE_descriptor_s,
10        struct_ptr: *const c_void,
11        buffer: *mut c_void,
12        buffer_size: usize,
13    ) -> asn_enc_rval_t;
14 }
```

Listing 2.10: Externe Funktionsdefinition der ASN.1 Funktion zum Enkodieren

Wie in Listing 2.10 zu sehen ist, können auch `extern {}` Blöcke mit Attributen versehen werden. Zwingend ist bei der Verwendung eines `#[link(..)]` Attributes der Name der Bibliothek, auf die sich der im `extern {}` Block stehende Code bezieht. Optional kann auch wie in Listing 2.10 die Art der **TODO: Linkung** (dylib, static) angegeben werden.

Die Art der Definition einer externen Funktion unterscheidet sich nicht von einer normalen Funktionsdefinition. Es sollten aber, wie in Abschnitt 2.13 beschrieben, zu C bzw. der externen Sprache kompatiblen Datentypen verwendet werden.

2.14 Kernfeatures

<https://www.youtube.com/watch?v=d1uraoHM8Gg>

TODO: no need for a runtime, all static analytics

TODO: memory safety

TODO: data-race freedom

TODO: active community

TODO: concurrency: no undefined behavior

TODO: ffi binding Foreign Function Interface

TODO: zero cost abstraction

TODO: package manager: cargo

<https://www.youtube.com/watch?v=-Tj8Q12DaEQ>

TODO: static type system with local type inference

TODO: explicit notion of mutability

TODO: zero-cost abstraction *(do not introduce new cost through implementation of abstraction)

TODO: errors are values not exceptions TODO: no null

TODO: static automatic memory management no garbage collection

TODO: often compared to GO and D (44min)

2.15 Schwächen

<https://www.youtube.com/watch?v=-Tj8Q12DaEQ>

TODO: compile-times

TODO: Rust is a vampire language, it does not reflect at all!

TODO: depending on the field -> majority of libraries?

2.16 Performance Fallstricke

TODO: [Llo]

2.17 Beispiele von Verwendung von Rust

TODO: firefox

<https://www.youtube.com/watch?v=-Tj8Q12DaEQ>

TODO: GTK binding heavily to rust

TODO: unstable TODO: ffi

3 Stand der Technik (c++ Version)

3.1 Hochperformant -> parallel?

3.2 Serverbasierte Kommunikationsplattform

3.3 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?

3.4 ASN.1

3.5 PER

3.6 MEC-View Server und Umgebung

4 Anforderungen

4.1 Funktionale Anforderungen

4.2 Nichtfunktionale Anforderungen

4.3 Kein Protobuf weil

5 Systemanalyse

5.1 Systemkontextdiagramm

5.2 Schnittstellenanalyse

5.3 C++ Referenzsystem

TODO: Design Pattern, Gamma et al, four important aspects
TODO: Real Time Design Patterns Buch: Ab Seite 141, verschiedene Systempatterns, microkernel [Dou03, S. 151]?
channel architektur pattern [Dou03, S. 167]?

TODO: hard real-time [Dou03, S. 75]

TODO: soft real-time [Dou03, S. 76]

TODO: Message Queuing Pattern [Dou03, S. 207]

6 Systementwurf

6.1 Änderungen bedingt durch Rust

7 Implementierung

TODO: Schwierigkeiten: FFI binding, manuell -> meh, also generieren

8 Auswertung

9 Zusammenfassung und Fazit

Literatur

- [atu] aturon. GitHub. Tracking issue for 128-bit integer support (RFC 1504). Englisch. URL: <https://github.com/rust-lang/rust/issues/35118#issuecomment-278078118> (besucht am 19.02.2018).
- [Bla15] Jim Blandy. Why Rust? Trustworthy, Concurrent System Programming. Englisch. 2015. URL: <http://www.oreilly.com/programming/free/files/why-rust.pdf> (besucht am 01.06.2017).
- [DD13] P.J. Deitel und H. Deitel. C for Programmers with an Introduction to C11. Deitel Developer Series. Pearson Education, 2013. ISBN: 9780133462074.
- [Dou03] B.P. Douglass. Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems. Addison-Wesley object technology series Bd. 1. Addison-Wesley, 2003. ISBN: 9780201699562.
- [fgi17] fgilcher. Subreddit Rust. fgilcher kommentiert. Englisch. 3. Nov. 2017. URL: https://www.reddit.com/r/rust/comments/7amv58/just_started_learning_rust_and_was_wondering_does/dpb9qew/ (besucht am 14.02.2018).
- [Gil17] Florian Gilcher. GOTO 2017. Why is Rust Successful? Englisch. 6. Dez. 2017. URL: <https://www.youtube.com/watch?v=-Tj8Q12DaEQ> (besucht am 21.02.2018).
- [GD14] J. Goll und M. Dausmann. C als erste Programmiersprache: Mit den Konzepten von C11. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014. ISBN: 9783834822710.
- [Grü17] Sebastian Grüner. „C ist eine feindselige Sprache“. Der Mitbegründer des Gnome-Projekts Interviewt. Deutsch. 22. Juni 2017. URL: <https://www.golem.de/news/rust-c-ist-eine-feindselige-sprache-1707-129196.html> (besucht am 14.02.2018).
- [Jim17] Jason Orendorff Jim Blandy. Programming Rust. Fast, Safe Systems Development. O'Reilly Media, Dez. 2017. ISBN: 1491927283.
- [Lei17] Felix von Leitner. Fefes Blog. D soll Teil von gcc werden. Deutsch. 22. Juni 2017. URL: <https://blog.fefe.de/?ts=a7b51cac> (besucht am 14.02.2018).
- [Llo] Llogiq. Llogiq on stuff. Rust Performance Pitfalls. Englisch. URL: <https://llogiq.github.io/2017/06/01/perf-pitfalls.html> (besucht am 14.02.2018).
- [LLVa] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Overview. Englisch. URL: <https://llvm.org/> (besucht am 19.02.2018).
- [LLVb] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Features. Englisch. URL: <https://llvm.org/Features.html> (besucht am 19.02.2018).

- [MEC] MEC-View. MEC-View. Deutsch. URL: <http://mec-view.de/> (besucht am 19.02.2018).
- [Qui] Federico Mena Quintero. Replacing C library code with Rust. What I learned with librsvg. Englisch. URL: <https://people.gnome.org/~federico/blog/docs/fmq-porting-c-to-rust.pdf> (besucht am 14.02.2018).
- [Rusa] Rust. The Rust Programming Language. Englisch. URL: <https://www.rust-lang.org/en-US/faq.html> (besucht am 16.02.2018).
- [Rusb] Rust. The Rust Programming Language. Rust Platform Support. Englisch. URL: <https://forge.rust-lang.org/platform-support.html> (besucht am 19.02.2018).
- [Rusc] Rust-Lang/Book. The Rust Programming Language. Primitive Types. Englisch. URL: <https://doc.rust-lang.org/book/first-edition/primitive-types.html> (besucht am 21.02.2018).
- [Rusd] Rust-Lang/Book. The Rust Programming Language. Unsafe Rust. Englisch. URL: <https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html#dereferencing-a-raw-pointer> (besucht am 20.02.2018).
- [Ruse] Rust-Lang/RFCs. GitHub. Tracking issue for RFC 1861: Extern types. Englisch. URL: <https://github.com/rust-lang/rust/issues/43467> (besucht am 20.02.2018).
- [Sch13] Julia Schmidt. Graydon Hoare im Interview zur Programmiersprache Rust. Deutsch. 12. Juli 2013. URL: <https://www.heise.de/-1916345> (besucht am 16.02.2018).
- [Wik17] Wikipedia. LLVM — Wikipedia, Die freie Enzyklopädie. 2017.
- [Wik18] Wikipedia. Foreign function interface — Wikipedia, The Free Encyclopedia. 2018.

Glossar

Foreign Function Interface Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [Wik18] . 15, 18

git (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005. **TODO: cite** . 8

LLVM Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplattformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb]. . 6

Abkürzungsverzeichnis

BMWi Bundesministerium für Wirtschaft und Energie. [2](#)

FBA Fehler Baum Analysen. [3](#)

MEC Mobile Edge Computing basierte Objekterkennung für hoch- und vollautomatisches Fahren. [2](#), [3](#)

SIL Safety Integrity Level. [3](#)

Abbildungsverzeichnis

1.1	MEC-View Schaubild der Robert Bosch GmbH [MEC]	2
2.1	Vergleich Rust Raw-Pointer und Referenz zu C-Pointer	17