Rust für Embedded-Systeme

Rust hat sich zu einem ernsthaften Kandidaten für die Systemprogrammierung gemausert. Zeit, einen Blick auf die Möglichkeiten und Fallstricke von Rust in Embedded-Systemen zu werfen.

Von Simon Jakob

Noch immer sind C und C++ dominant unter den Programmiersprachen im Embedded-Bereich. Doch seit einiger Zeit gewinnt Rust Anteile in der auf IT- und funktionale Sicherheit bedachten Landschaft der eingebetteten Systeme. Warum das so ist, lässt sich am Beispiel der Bug-Bounty-Summary-Meldung von Google erahnen, die eine Korrelation zwischen der verringerten Zahl eingegangener Vulnerabilities und dem gestiegenen Anteil von Rust in neueren Android-Versionen aufzeigt (siehe ix.de/z7qb). Was macht Rust aber nun – wie behauptet – sicherer als andere Systemprogrammiersprachen, und ist es so interessant, dass Entwickler eingebetteter Systeme einen Wechsel ins Auge fassen sollten?

Embedded-Systeme sind in Sachen Ressourcen stark begrenzt. Die Spannweite reicht von Controllern mit wenigen Kilobyte Speicher und ohne Memory Management Unit (MMU) bis hin zu leistungsstarken Systems-on-a-Chip (SoCs) auf ARM-Basis mit mehreren CPUs und Gigabyte Speicher. Oft setzt die Industrie aber einfache Controller (wie bei Motorsteuerungen) oder SoCs mit überschaubarer, aber zweckmäßiger Ausstattung ein. Eine sehr häufige Anforderung ist, dass diese Systeme zuverlässig funktionieren und dabei den industriespezifischen Safety- und Securitystandards genügen müssen. Embedded-Software-Architektinnen und -Architekten legen Systeme deshalb häufig redundant aus und achten darauf, dass sie harte Echtzeitanforderungen erfüllen.

Speichersicherheit unbedingt erforderlich

Funktionale Sicherheit erfordert Determinismus als zentralen Baustein, um Anforderungen gerecht zu werden, wie sie zum Beispiel in der für die Avionik geltenden Norm DO-178C formuliert sind. Bereits Multicore-Prozessoren einzusetzen ist kritisch, und nur langsam finden solche Systeme Einsatz in der Praxis. Die EASA beispielsweise hat erst Anfang 2022 eine offizielle Leitlinie verabschiedet, die zeigt, wie funktionale Sicherheit in solchen Systemen gewährleistet werden kann. Vor diesem Hintergrund müssen Programmierer jede Unwägbarkeit umschiffen – worunter Speicherproblematiken wie Null-Pointer-Fehler, Dangling Pointer, Buffer Overflows oder auch Race Conditions und undefiniertes Verhalten fallen. In einem robusten System ist es also notwendig, dass stets bekannt ist, was im Detail passiert. Speichersicherheit und verlässliche Nebenläufigkeit sind unabdingbar.

Embedded Rust, die Rust-Variante für eingebettete Systeme, bietet nun mehrere wesentliche Vorteile, die es für Embedded-Software-Architekten interessant machen. Zunächst einmal garantiert Rust Speichersicherheit über das Ownership-Konzept. Dies und ein effektives Datentypenmanagement bilden die Grundlage für sichere Nebenläufigkeit. Die Rust-Macher sprechen hier von furchtloser Nebenläufigkeit. Weil Embedded Rust keine Laufzeitumgebung braucht, ohne „teure“ Komponenten wie Garbage Collectors auskommt und jeder Ecke der Sprache anzumerken ist, dass sie performant sein möchte, passt sie ins Anforderungsprofil vieler eingebetteter Systeme.

Rust hat eine wachsende Community und eine eigene Arbeitsgruppe – die Rust Embedded Devices Working Group –, die sich um Embedded-Fragen kümmert. Dazu gehören eine dedizierte Webseite und die Rust-Embedded-Einsteigerdokumentation (siehe ix.de/z7qb). Die Rust Embedded Devices Working Group wirkt darauf hin, dass die Unterstützung für

iX-TRACT

- Rust ist performant und garantiert Speicher- und Threadsicherheit. Das erleichtert es Rust-Software, industriespezifische Safety- und Securitystandards einzuhalten.
- Allerdings fehlt noch eine formale Sprachspezifikation und die für garantierte Safety wichtige qualifizierte Toolchain.
- Das Ownership-Konzept verhindert Null Pointer und Dangling Pointer, macht Programmierern allerdings zusätzliche Arbeit.
- Die Rust Embedded Devices Working Group kümmert sich darum, dass Rust auf möglichst vielen Hardwarearchitekturen läuft, beispielsweise durch einen standardisierten Hardware Abstraction Layer für ARM-Controller.

viele Architekturen vorhanden ist und ausgebaut wird. Das Rust-eigene Paketsystem – Pakete heißen in Rust Crates – enthält beispielsweise einen standardisierten Hardware Abstraction Layer (HAL) für ARM-Controller, der Entwicklerinnen und Entwicklern das Leben einfacher macht. Zudem gibt es für viele Boards (etwa STM32) modular aufgebaute Crates, die alle wesentlichen Komponenten ansprechen.

Da die Sprache Rust noch jung ist, hat sie nicht den Ballast und den Umfang von C/C++. Sie besitzt objektorientierte und funktionale Merkmale und ist im Vergleich mit C/C++ zwar auch nicht leicht zu erlernen, doch benötigen Embedded-Entwickler deutlich weniger Erfahrung mit ihr. Ihre Idiome in Kombination mit der Toolchain arbeiten wie eine Art Co-Pilot in Fragen der Sicherheit mit. Und zu guter Letzt können Systemarchitekten bestehenden Code mit Rust-Code und umgekehrt erweitern. Bewährte Komponenten erhalten dadurch eine Laufzeitverlängerung.

Rusts Alleinstellungsmerkmal: Ownership

So stellt Rust Speichersicherheit her: Ownership besagt – kurz und vereinfacht gesprochen –, dass jede Speicherressource genau einer Variablen gehört. Dadurch ist stets bekannt, was im Speicher passiert. Rust verhindert so, dass Ressourcen falsch alloziert werden. In der Konsequenz bedeutet das für eingebettete Systeme, dass Nullpointer-Fehler und Dangling Pointer der Vergangenheit angehören. Wie aber funktioniert das?

Während in C der Versuch, auf einen nicht initialisierten Speicherbereich zuzugreifen, ein Programm zur Laufzeit abstürzen lassen kann, verhindert Rust bereits beim Kompilieren die Ursache des Problems. Erstens gibt es nicht den klassischen Pointer in Rust, sondern speichersichere Referenzen und darüber hinaus Smart Pointer, die aber ebenfalls dem Rust-Sicherheitskonzept unterworfen sind. Zweitens führt der Compiler eine statische Analyse durch und prüft mögliche Speicherproblematiken vor der Übersetzung, was problematisches Laufzeitverhalten verhindert.

Das fängt damit an, dass Variablen standardmäßig Quasikonstanten sind und einen Gültigkeitsbereich (Scope) haben. Sie werden erst über das Präfix `mut` veränderbar, was ein wenig an Kofferrollwagen an Flughäfen erinnert, die man erst schieben kann, wenn man den Riegel herunterdrückt.

Listing 1: Gültigkeitsbereich Variable beim Kopieren

```
fn main() {
    let x = String::from("Gültig?"); // Strings an x zuweisen
    let y = x; // Pointer auf den Heap kopieren
    println!("{}", y); // Ausgabe funktioniert hier theoretisch noch
    println!("{}", x); // hier ist x aber bereits ungültig
}
```

Listing 2: Ownership bei Funktionen

```
fn main() {
    let x = ausflugsziel();
    println!("{}", x);
}

fn ausflugsziel() -> String {
    let nach_hause = String::from("Heimkehr!");
    nach_hause // Return in Rust durch Aufruf der Variablen ohne ;
}
```

Listing 1 zeigt, was bei Kopiervorgängen mit dem Gültigkeitsbereich einer Variablen passiert. Bei solch einem Code wirft der Compiler einen Fehler aufgrund der letzten `println`-Anweisung aus. Unter der Haube weist der Compiler `x` erst den String „`Gültig?`“ zu. `x` erhält einen Eintrag auf dem Stack und der String wird im Heap angelegt. Beim Übertragungsvorgang wird aber aus Performancegründen nur die Stackinformation kopiert – wie bei einer flachen Kopie. Allerdings verliert `x` darüber hinaus nun seine Gültigkeit, was verhindert, dass ein Thread ungewollt auf Ressourcen zugreift. Außerdem: Verlässt `y` nun seinen Gültigkeitsbereich, räumt Rust den Platz im Heap automatisch auf. Während das erste `println` also noch kompilieren würde, verweigert der Compiler beim zweiten die Übersetzung.

Würden wie im Beispiel in Listing 1 die Heap-Informationen mitkopiert, wären die Kosten in Sachen Performance womöglich zu hoch. Allerdings macht Rust Ausnahmen: Ist eine Variablengröße zur Kompilierzeit bekannt, kann es sein, dass Rust sie nur auf den Stack legt – etwa bei Ganz- und Fließkommazahlen, Char und Boolean. In diesem Fall bliebe die Variable im Gültigkeitsbereich. Wenn Entwickler im oben genannten Codebeispiel `x` einfach 5 zuweisen, anstatt einen String zu erzeugen, funktioniert die Übersetzung. Dies ist eines der Beispiele, die zeigen, wie Rust zugunsten der Performance auf Einheitlichkeit verzichtet.

Analog greift das Ownership-Konzept auch an anderen Stellen, beispielsweise bei Funktionen: Übergibt Rust einen Wert, wird er in der Ausgangsfunktion ungültig. Der Wert und die Gültigkeit gehen bei Funktionen mit Rückgabewert auf Wanderschaft (Listing 2).

Veränderbare Referenzen

Embedded-Software-Architekten können Referenzen anstelle von Pointern – die im klassischen Sinne in Rust nicht existieren – verwenden, um auf Speicherbereiche zuzugreifen. Dies geschieht über das &-Präfix. Da sichergestellt ist, dass ein Wert nur einer Variablen gehört, ist auch sicher, dass Referenzen auf gültige Werte und Speicherbereiche zugreifen. Das Konzept nennt Rust Borrowing. Im Prinzip ist es wie bei einem guten Freund: Leiht man ihm ein Buch aus, kann er es

Listing 3: Eine C-Bibliothek mit dem Schlüsselwort extern aufrufen

```
#[link(name = "gruss-lib")]

extern "C" {
    fn gruesse();
}

fn main() {
    unsafe {
        gruesse();
    }
}
```

lesen. Er darf aber nicht darin Textstellen markieren, Seiten herausreißen oder ein anderes Buch zurückgeben. Wenn er dennoch etwas ins Buch hineinschreiben möchte, kann er aber darum bitten. Das funktioniert über veränderbare Referenzen und wie bei veränderbaren Variablen über das `mut`-Präfix. Allerdings darf nur eine veränderbare Referenz existieren.

Der tiefere Sinn und Nutzen

Rust – so scheint es – macht es Entwicklerinnen und Entwicklern aber umständlicher, als es sein müsste: Die Variablen sind nicht veränderbar und Rust legt beim Kopieren Steine in den Weg. Was zunächst kontraintuitiv und frustrierend gegenüber anderen Programmiersprachen erscheinen mag, hat den Sinn, Speichersicherheit unter Beibehaltung der Performance zu ermöglichen. Unter dem Strich steht letzten Endes ein Sicherheitsgewinn, den Entwickler durch Einschränkungen und Umständlichkeit erkaufen.

Die technische Erklärung ist, dass Rust an vielen Stellen unvorhersehbares Verhalten unterbindet, das zum Beispiel dadurch entstünde, dass zwei oder mehr Threads auf die gleiche veränderliche Speicherressource zugriffen und mindestens ein Thread diese veränderte. Für die weitere Absicherung über Mutexe, Arc-Typen und Semaphoren kann Rust Ressourcen und Variablen blockieren und threadsicher verwenden. Das verhindert Race Conditions und in der Folge Data Races und Deadlocks. In Konsequenz ermöglicht es sichere Nebenläufigkeit.

Die Tücken von Rust

Rust ist aber nur so sicher, wie Entwickler es letzten Endes haben möchten. Sein Sicherheitskonzept ist für Entwickler von Embedded-Systemen gerade deshalb interessant, weil der Trade-off Sicherheit gegen Umständlichkeit im Laufe der Zeit kostengünstiger wird, je mehr Erfahrungen sie mit den Sprachidiomen gewinnen. Rust lässt aber auch viel unsicheres Verhalten zu – nämlich dann, wenn ein Vorgang alternativlos ist.

Entwicklerinnen und Entwickler können unsichere Operationen mit dem Schlüsselwort `unsafe` ausführen. Auch der `panic`-Mechanismus zeugt davon, dass Rust kein Allheilmittel für jegliche Unwägbarkeit oder fehlerhafte Programmierung ist. Die Rust-Macher sehen dies jedoch nur als letzten Ausweg – schließlich sollen Entwickler schon zur Kompilierzeit das Programm so sicher wie möglich konstruieren. Das Ownership-Konzept verlangt ihnen einiges an Einarbeitung ab und die ersten Versuche, sich in der Sprache auszudrücken, erfordern ein hohes Maß an Frustrationstoleranz.

Es fehlt darüber hinaus bis heute eine formale Sprachspezifikation, und die für garantierte Safety wichtige qualifizierte Toolchain wird erst mittelfristig verfügbar sein. Ferrous Systems und AdaCore arbeiten daran. Auch die GNU Compiler Collection (GCC) lässt auf sich warten und wird Rust bestenfalls im laufenden Jahr unterstützen. In Summe: So vielversprechend Rust auch ist, in vielerlei Hinsicht steckt die Sprache noch in den Kinderschuhen.

C-Verträglichkeit mit Safety-Gewinn

Rust ist so zugeschnitten, dass es sich mit C-Code verträgt. Es kann mit Foreign Function Interfaces (FFI) C-Code aufrufen und umgekehrt. Hier ist es naturgemäß problematisch, dass das Ownership-Konzept in C nicht existiert. Rust-Code aus C aufzurufen, ergibt aber dennoch Sinn, weil in sich gekapselte Funktionen zumindest in diesem Teil der Gesamtapplikation sicher sind. Rust erlaubt, bestehenden C-Legacy-Code weiterzuverwenden, was unter anderem dann interessant ist, wenn bereits in C geschriebene Bibliotheken sicherheitszertifiziert sind und nachzertifiziert werden sollen. In einem einfachen Beispiel sieht das so aus (C-Code, bereits kompilierte Bibliothek):

```
#include <stdio.h>

void gruesse() {
    printf("Gruss aus C!\n");
}
```

Rust ruft über das Schlüsselwort `extern` die C-Bibliothek auf (Listing 3). Entwickler müssen diese in Rust einmal als Interface deklarieren, damit sie danach in `main()` verfügbar ist. Dafür ist zwingend vorher `unsafe` aufzurufen, da C aus Rust-Sicht unsicher ist. `unsafe` hebt einige normalerweise in Rust geltende Einschränkungen auf, etwa das Verbot der Dereferenzierung eines Rawpointer oder das Aufrufen einer unsicheren Funktion oder Methode. Es ist also Signalwort und eine Art Cheat für Rust gleichermaßen. Die Rust-Doku spricht hier von einer „Superpower“.

Der C-Aufruf ist jedoch nur ein einfaches Beispiel. Wollen Systemarchitekten umfangreichere C-Code-Teile übernehmen, nutzen sie das bindgen-Crate, um das Interface nicht manuell deklarieren zu müssen.

Avionik: Multicore-Processing für sicherheitskritische Bereiche

In der Avionik ist seit einigen Jahren der Einsatz von Multicore-Systemen ein heißes Eisen. Bisher ist es erst einmal gelungen, ein Multicore-System gegen den Avionics-Safety-Standard DO-178C auf höchstem Level (DAL A) zu zertifizieren. Die AMC-20-193-Richtlinie (Acceptable Means of Compliance) der Europäischen Agentur für Flugsicherheit (EASA) dient als Leitfaden für den sicheren Einsatz. Sie beschreibt Anforderungen, wie ein System aufgebaut sein muss, und welche Anforderungen Programmierer und die Programmierung erfüllen müssen. Mit Rust ist es denkbar, dass in Zukunft häufiger Multicore-Systeme eine hohe Zertifizierungsstufe erlangen, weil Rust viele Probleme der Nebenläufigkeit zur Kompilierzeit durch das Ownership-Konzept bereits entschärft.

Letzten Endes ist die Frage, ob das für ein großes Plus an Sicherheit reicht. Es ist aber ein Schritt in die richtige Richtung. Wichtiger ist jedoch: AMC 20-193 fordert, besonderen Wert auf Sicherheitsmechanismen in der Programmierung zu legen und unter anderem den Fokus auf Race Conditions zu setzen und sie zu vermeiden (AMC 20-193, Seite 6; siehe ix.de/z7qb). Das Ownership-Konzept ist dafür eine solide Grundlage. Auch deswegen ist Rust eine geeignete Programmiersprache, um künftig leistungsfähigere Hardware einsetzen zu können und Systemanbietern die Angst vor Worst-Case-Szenarien zu nehmen.

Neben der Avionik haben Systemarchitekten im Umfeld von Automotive, des Bahnwesens und der Industrie im Allgemeinen ganz ähnliche Herausforderungen, was Safety angeht. Sie alle können von den besonders strengen Anforderungen an Avionik-Systeme profitieren und deren Konzepte adaptieren. Für bodennahe Industrien ist zudem die Frage nach der Security sehr wichtig – was nicht heißen soll, dass Safety keine Rolle spielte, jedoch sind Automotive, Medizintechnik und Industrie noch viel eher von Cyberattacken betroffen. Insbesondere Automotive mit dem Konzept des Software-defined Vehicle sucht nach Lösungen für eine erhöhte Sicherheit und hat Rust deswegen schon seit einiger Zeit auf dem Schirm. Die allgemeine Sicherheitslage wird besser – um auf das eingangs erwähnte Beispiel von Google zurückzu-

kommen –, wenn nicht nur undefinierte Zustände und Laufzeitfehler weniger werden, sondern sich eben auch häufig über Speicherfehler adressierte Vektoren zum Angriff auf IT-Systeme in ihrer Anzahl verringern.

Rust mit Hypervisor und Echtzeitbetriebssystem

Embedded Rust ist für den Bare-Metal-Einsatz ausgelegt. Allerdings verträgt es sich auch mit eingebetteten Systemen mit Echtzeitbetriebssystemen und Separation-Kernel. Embedded Rust funktioniert beispielsweise mit dem Echtzeitbetriebssystem und Hypervisor PikeOS. Softwarearchitekten profitieren von einigen Features und Annehmlichkeiten, etwa dass PikeOS abgefangene `panics` zum OS-Health-Monitor leitet und verarbeitet. Dazu erhalten Systemarchitekten und Softwareentwickler die Komfortfunktionen eines Betriebssystems: Safety-zertifizierbare Dateisysteme, die Nutzung von Queuing- und Sampling-Ports, natives Threading und weitere.

PikeOS bindet das Alloc- und das Core-Crate ein und verzichtet auf die Standardbibliothek. Das Heap-Management läuft dann über die Programmierung innerhalb eines für die jeweilige Partition vorallozierten Speicherbereichs. Minimalisten oder Anhänger der Sichtweise, dass möglichst wenige Funktionen auf einem System laufen sollten, damit es sicher ist, bleiben vermutlich lieber bei Bare-Metal-Systemen und -Coding. Allerdings bietet ein Betriebssystem auch Vorteile. Im Falle des PikeOS-Separation-Kernels in der Version 5.1.3 ist das eine Securityzertifizierung auf dem hohen Niveau EAL5+; das Plus zeigt, dass unter anderem die Gefahrenanalyse auf dem höchsten Level EAL7 erreicht wurde. Zudem sind Echtzeitbetriebssysteme häufig Safety-zertifiziert, sodass der Zertifizierungsaufwand für sicherheitskritische Anwendungen geringer ausfällt.

Fazit

Rust bietet viele Neuerungen, die es ermöglichen, sicherere Systeme zu konstruieren. Rust verbindet dabei Performance mit Sicherheit bei Speicherverwaltung und Nebenläufigkeit, was Systementwicklerinnen und -entwickler durch eine steile Lernkurve und einen gewissen Grad an Umständlichkeit erkaufen. Das perfekt sichere System existiert nicht und wird vermutlich nie existieren. Dafür finden Angreifer immer wieder neue Wege, wie sie Systeme kompromittieren können. Und wo immer Menschen arbeiten, passieren Fehler. Diese Unwägbarkeiten wird es immer geben. Rust leistet aber einen profunden Beitrag, die Welt der eingebetteten Systeme und damit unsere immer digitaler werdende Welt etwas sicherer zu machen. (nb@ix.de)

Quellen

Links zur Rust-Dokumentation und zu den Quellen: ix.de/z7qb

SIMON JAKOB

ist Fachinformatiker Anwendungsentwicklung, Technikjournalist und PR-Manager. Er arbeitet als Digital Content & PR Manager für die SYSGO GmbH, eine Tochtergesellschaft von Thales.