The Embedded Rust Book (German)

Contents

1. Einleitung

Willkommen bei „Das Embedded-Rust Buch“ – einem Einführungswerk zur Verwendung der Programmiersprache Rust auf „Bare-Metal“-Embedded-Systemen, wie etwa Mikrocontrollern.

1.0.1. Für wen Embedded-Rust geeignet ist

Embedded-Rust ist für alle, die eingebettete Systeme programmieren und dabei die Vorteile der fortgeschrittenen Konzepte und Sicherheitsgarantien der Rust-Sprache nutzen möchten.

(Siehe auch Für wen ist Rust geeignet?)

1.0.2. Umfang

Die Ziele dieses Buches sind:

- Machen Sie Entwickler mit der Embedded-Rust-Entwicklung vertraut – zum Beispiel damit, wie man eine Entwicklungsumgebung einrichtet.
- Teilen Sie *aktuelle* Best Practices für den Einsatz von Rust in der Embedded-Entwicklung – zum Beispiel: Wie man Sprachfunktionen von Rust optimal nutzt, um korrektere Embedded-Software zu schreiben.
- Dient in einigen Fällen als Kochbuch – zum Beispiel: Wie kombiniere ich C und Rust in einem einzigen Projekt?

Dieses Buch ist bestrebt, so allgemein wie möglich zu bleiben; um jedoch sowohl den Lesern als auch den Autoren die Arbeit zu erleichtern, verwendet es in allen Beispielen die ARM-Cortex-M-Architektur. Dennoch setzt das Buch keine Vorkenntnisse zu dieser speziellen Architektur voraus und erläutert die entsprechenden Besonderheiten, wo immer dies erforderlich ist.

1.0.3. Für wen dieses Buch gedacht ist

Dieses Buch richtet sich an Personen mit Vorkenntnissen im Bereich Embedded-Systeme oder Rust; wir sind jedoch der Meinung, dass jeder, der sich für die Embedded-Programmierung mit Rust interessiert, von diesem Buch profitieren kann. Wer keine Vorkenntnisse mitbringt, dem empfehlen wir, den Abschnitt „Annahmen und Voraussetzungen“ zu lesen und sich entsprechendes Wissen anzueignen, um den größtmöglichen Nutzen aus dem Buch zu ziehen und das Leseerlebnis zu verbessern. Im Abschnitt „Weitere Ressourcen” finden Sie Hinweise zu Themen, in die Sie sich bei Bedarf einarbeiten können.

1.0.3.1. Annahmen und Voraussetzungen

- Sie sind sicher im Umgang mit der Programmiersprache Rust und haben bereits Rust-Anwendungen in einer Desktop-Umgebung geschrieben, ausgeführt und deren Fehler behoben. Zudem sollten Sie mit den Idiomen der 2018 Edition vertraut sein, da sich dieses Buch auf Rust 2018 konzentriert.
- Sie sind sicher in der Entwicklung und Fehlersuche bei Embedded-Systemen in einer anderen Sprache wie C, C++ oder Ada und mit Konzepten vertraut wie:
  - Cross-Kompilierung
  - Im Speicher abgebildete Peripheriegeräte
  - Interrupts
  - Gängige Schnittstellen wie I2C, SPI, seriell usw.

1.0.3.2. Weitere Ressourcen

Falls Ihnen einer der oben genannten Punkte nicht vertraut ist oder Sie mehr über ein bestimmtes, in diesem Buch erwähntes Thema erfahren möchten, könnten einige dieser Ressourcen hilfreich für Sie sein.

Thema	Ressource	Beschreibung
Rust	Rust Book	Wenn Sie sich mit Rust noch nicht sicher fühlen, empfehlen wir Ihnen dringend, dieses Buch zu lesen.
Rust, Embedded	Discovery Book	Wenn Sie sich noch nie mit eingebetteter Programmierung beschäftigt haben, ist dieses Buch möglicherweise der bessere Einstieg
Rust, Embedded	Embedded Rust Bookshelf	Hier finden Sie weitere Ressourcen, die von der Embedded Working Group von Rust bereitgestellt werden.
Rust, Embedded	Embedonomicon	Die Feinheiten der Embedded-Programmierung mit Rust.
Rust, Embedded	embedded FAQ	Häufig gestellte Fragen zu Rust im Embedded-Umfeld.
Rust, Embedded	Comprehensive Rust 🦀: Bare Metal	Unterrichtsmaterial für einen viertägigen Kurs zu Bare-Metal Rust development
Interrupts	Interrupt	-
Memory-mapped IO/Peripherals	Memory-mapped I/O	-
SPI, UART, RS232, USB, I2C, TTL	Stack Exchange about SPI, UART, and other interfaces	-

1.0.3.3. Übersetzungen

Dieses Buch wurde von großzügigen Freiwilligen übersetzt. Wenn Ihre Übersetzung hier aufgeführt werden soll, erstellen Sie bitte einen Pull Request, um sie hinzuzufügen.

- Japanisch (repository)
- Chinesisch (repository)

1.0.3.4. Wie Sie dieses Buch nutzen

Dieses Buch ist im Allgemeinen so konzipiert, dass es von Anfang bis Ende gelesen wird. Spätere Kapitel bauen auf den Konzepten früherer Kapitel auf; dabei werden bestimmte Themen in den früheren Kapiteln möglicherweise noch nicht bis ins Detail behandelt, sondern erst in einem späteren Kapitel erneut aufgegriffen.

Für den Großteil der in diesem Buch enthaltenen Beispiele wird das Entwicklungsboard STM32F3DISCOVERY von STMicroelectronics verwendet. Dieses Board basiert auf der ARM-Cortex-M-Architektur. Zwar ist die grundlegende Funktionalität bei den meisten auf dieser Architektur basierenden CPUs identisch, doch unterscheiden sich Peripheriekomponenten und andere Implementierungsdetails der Mikrocontroller je nach Hersteller – und oft sogar zwischen verschiedenen Mikrocontroller-Familien desselben Herstellers.

Aus diesem Grund empfehlen wir die Anschaffung des Entwicklungsboards STM32F3DISCOVERY, um die Beispiele in diesem Buch praktisch nachvollziehen zu können.

1.0.3.5. Beiträge zu diesem Buch

Die Arbeit an diesem Buch wird in diesem Repository koordiniert und hauptsächlich vom Ressourcen-Team vorangetrieben.

Wenn Sie Schwierigkeiten haben, den Anweisungen in diesem Buch zu folgen, oder wenn ein Abschnitt des Buches nicht klar genug oder schwer verständlich ist, handelt es sich um einen Fehler; dieser sollte im Problem-Verfolgungswerkzeug des Buches gemeldet werden.

Pull Requests, die Tippfehler korrigieren oder neue Inhalte hinzufügen, sind herzlich willkommen!

1.0.4. Wiederverwendung dieses Materials

Dieses Buch wird unter den folgenden Lizenzen vertrieben:

- Die in diesem Buch enthaltenen Codebeispiele und eigenständigen Cargo-Projekte stehen unter den Bedingungen sowohl der MIT-Lizenz als auch der Apache-Lizenz v2.0.
- Die in diesem Buch enthaltenen Texte, Bilder und Diagramme stehen unter der Lizenz Creative Commons CC-BY-SA v4.0.

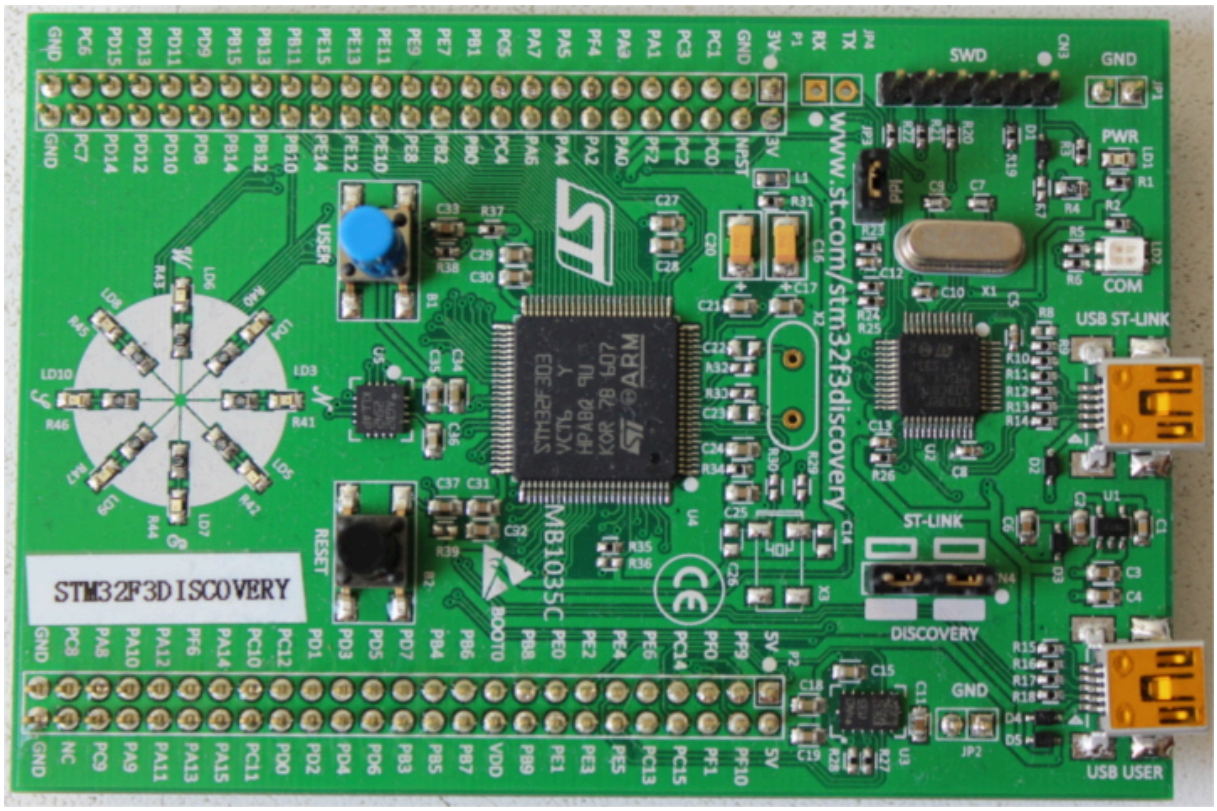
Kurz gesagt: Wenn Sie unsere Texte oder Bilder für Ihre Arbeit verwenden möchten, müssen Sie:

- Geben Sie die entsprechende Quellenangabe an (d. h. erwähnen Sie dieses Buch auf Ihrer Folie und geben Sie einen Link zur entsprechenden Seite an).
- Geben Sie einen Link zur CC-BY-SA v4.0-Lizenz an.
- Geben Sie an, ob Sie das Material in irgendeiner Weise verändert haben, und stellen Sie Änderungen an unserem Material unter derselben Lizenz zur Verfügung.

Teilen Sie uns bitte auch mit, ob Sie dieses Buch nützlich finden!

1.1. Lernen Sie Ihre Hardware kennen

Machen wir uns mit der Hardware vertraut, mit der wir arbeiten werden.



1.1.1. STM32F3DISCOVERY (die “F3”)

Was beinhaltet diese Platine?

- Ein STM32F303VCT6-Mikrocontroller. Dieser Mikrocontroller hat
  - Ein Single-Core-ARM-Cortex-M4F-Prozessor mit Hardware-Unterstützung für Gleitkommaoperationen einfacher Genauigkeit und einer maximalen Taktfrequenz von 72 MHz.
  - 256 KiB “Flash”-Speicher. (1 KiB = 10**24** bytes)
  - 48 KiB RAM.
  - Eine Vielzahl integrierter Peripherieeinheiten wie Timer, I2C, SPI und USART.
  - GPIO-Pins (General Purpose Input/Output) und andere Pin-Typen, die über die beiden Stiftleistenreihen an den Seiten der Platine zugänglich sind.
  - Einen Beschleunigungsmesser als Teil des LSM303DLHC-Chips.
  - Ein Magnetfeldstärkenmessgerät als Teil des LSM303DLHC-Chips. gekennzeichneten USB-Anschluss zugänglich ist.
- Ein Beschleunigungssensor als Teil des LSM303DLHC-Chips. die Übersetzung ist veraltet
- Ein Gyroskop als Teil des L3GD20-Chips.
- 8 Benutzer-LEDs, angeordnet in Form eines Kompasses.
- Ein zweiter Mikrocontroller: ein STM32F103. Dieser Mikrocontroller ist eigentlich Teil eines On-Board-Programmers/Debuggers und mit dem Mini-USB-Anschluss namens „USB ST-LINK“ verbunden.

Eine detailliertere Liste der Funktionen und weitere technische Daten des Boards finden Sie auf der Website von STMicroelectronics.

Ein wichtiger Hinweis: Seien Sie vorsichtig, wenn Sie externe Signale an das Board anlegen möchten. Die Pins des Mikrocontrollers STM32F303VCT6 sind für eine Nennspannung von 3,3 Volt ausgelegt. Weitere Informationen finden Sie im Abschnitt „6.2 Absolute maximum ratings“ (Absolute Grenzwerte) im Datenblatt.

1.2. Eine `no_std`-Rust-Umgebung

Der Begriff „Embedded Programming“ (Programmierung eingebetteter Systeme) umfasst ein breites Spektrum unterschiedlichster Programmierbereiche. Das Spektrum reicht von der Programmierung von 8-Bit-Mikrocontrollern (wie dem ST72325xx mit nur wenigen Kilobyte RAM und ROM bis hin zu Systemen wie dem Raspberry Pi (Modell B 3+), der über einen 32/64-Bit-Quad-Core-Prozessor (Cortex-A53 mit 1,4 GHz) und 1 GB RAM verfügt. Je nach Zielsystem und Anwendungsfall gelten bei der Programmierung unterschiedliche Einschränkungen und Rahmenbedingungen.

Es gibt zwei grundlegende Kategorien der Programmierung eingebetteter Systeme:

1.2.1. Hosted Environments

Derartige Umgebungen ähneln einer herkömmlichen PC-Umgebung. Das bedeutet, dass Ihnen eine Systemschnittstelle (z. B. POSIX) zur Verfügung steht, die Grundfunktionen (Primitive) für die Interaktion mit verschiedenen Systemkomponenten bietet – etwa Dateisystemen, Netzwerkfunktionen, Speicherverwaltung, Threads usw. Standardbibliotheken wiederum basieren in der Regel auf diesen Grundfunktionen, um ihre Funktionalität zu implementieren. Zudem können eine spezifische System-Root-Umgebung (sysroot) sowie Einschränkungen bei der RAM- oder ROM-Nutzung vorliegen; auch spezielle Hardware oder Ein-/Ausgabeschnittstellen (I/Os) können vorhanden sein. Insgesamt vermittelt die Programmierung den Eindruck, in einer auf einen speziellen Einsatzzweck ausgerichteten PC-Umgebung zu arbeiten.

1.2.2. Rein physische Umgebung

In einer rein physischen Umgebung wird vor Ihrem Programm kein Code geladen. Ohne die vom Betriebssystem bereitgestellte Software kann die Standardbibliothek nicht geladen werden. Stattdessen kann das Programm zusammen mit den verwendeten Crates ausschließlich die Hardware (Bare Metal) nutzen. Um zu verhindern, dass Rust die Standardbibliothek lädt, verwenden Sie `no_std`. Die plattformunabhängigen Teile der Standardbibliothek sind über libcore verfügbar. libcore schließt außerdem Funktionen aus, die in eingebetteten Umgebungen nicht immer erwünscht sind. Eine dieser Funktionen ist ein Speicherallokator für die dynamische Speicherverwaltung. Falls Sie diese oder andere Funktionalitäten benötigen, stehen Ihnen häufig entsprechende Crates zur Verfügung.

1.2.2.1. Die libstd-Laufzeitumgebung

Wie bereits erwähnt, erfordert die Verwendung von libstd eine gewisse Systemintegration; dies liegt jedoch nicht nur daran, dass libstd eine einheitliche Schnittstelle zu Betriebssystem-Abstraktionen bereitstellt, sondern auch daran, dass es eine Laufzeitumgebung (Runtime) mitliefert. Diese Laufzeitumgebung kümmert sich unter anderem um die Einrichtung eines Stack-Overflow-Schutzes, die Verarbeitung von Kommandozeilenargumenten sowie das Starten des Haupt-Threads, bevor die `main`-Funktion des Programms aufgerufen wird. In einer `no_std`-Umgebung steht diese Laufzeitumgebung nicht zur Verfügung.

1.2.3. Zusammenfassung

`#![no_std]` ist ein Attribut auf Crate-Ebene, das angibt, dass das Crate gegen das `core`-Crate anstatt gegen das `std`-Crate gelinkt wird. Das libcore-Crate wiederum ist eine plattformunabhängige Teilmenge des `std`-Crates, die keinerlei Annahmen über das System trifft, auf dem das Programm ausgeführt wird. Dementsprechend stellt es APIs für Sprachprimitive wie Gleitkommazahlen, Strings und Slices bereit sowie APIs, die Prozessorfunktionen wie atomare Operationen und SIMD-Befehle zugänglich machen. Es fehlen jedoch APIs für Funktionen, die eine Plattformintegration erfordern. Aufgrund dieser Eigenschaften eignen sich `no_std`- und libcore-Code für jegliche Art von Bootstrapping-Code (Stufe 0), wie etwa Bootloader, Firmware oder Kernel.

1.2.3.1. Überblick

Eigenschaft	no_std	std
heap (dynamischer Speicher)	*	✓
collections (Vec, BTreeMap, usw.)	**	✓
Schutz vor Stack-Überlauf	✘	✓
führt Initialisierungscode vor der main-Funktion aus	✘	✓
libstd verfügbar	✘	✓
libcore verfügbar	✓	✓
schreibt firmware, kernel, oder bootloader code	✓	✘

* Nur wenn Sie das `alloc`-Crate und einen geeigneten Allocator wie alloc-cortex-m verwenden.

** Nur wenn Sie das `collections`-Crate verwenden und einen globalen Standard-Allocator konfigurieren.

** HashMap und HashSet stehen aufgrund des Fehlens eines sicheren Zufallszahlengenerators nicht zur Verfügung.

1.2.4. Siehe auch

- RFC-1184

1.3. Werkzeuge

Der Umgang mit Mikrocontrollern erfordert den Einsatz verschiedener Werkzeuge, da wir es mit einer Architektur zu tun haben, die sich von der Ihres Laptops unterscheidet, und Programme auf einem *entfernten* Gerät ausführen sowie debuggen müssen.

Wir werden alle unten aufgeführten Werkzeuge verwenden. Sofern keine Mindestversion angegeben ist, sollte jede aktuelle Version funktionieren; wir haben jedoch die Versionen aufgelistet, die wir getestet haben.

- Rust 1.31, 1.31-beta oder neuer, SOWIE Unterstützung für die Kompilierung für ARM Cortex-M.
- `cargo-binutils` ~0.1.4
- `qemu-system-arm`. Getestete Version: 3.0.0
- OpenOCD >=0.8. Getestete Versionen: v0.9.0 und v0.10.0
- GDB mit ARM-Unterstützung. Version 7.12 oder neuer wird dringend empfohlen. Getestete Versionen: 7.10, 7.11, 7.12 und 8.1.
- `cargo-generate` oder `git`. Diese Werkzeuge sind optional, erleichtern es jedoch, dem Buch zu folgen.

Der folgende Text erläutert, warum wir diese Werkzeuge verwenden. Installationsanweisungen finden Sie auf der nächsten Seite.

1.3.1. `cargo-generate` ODER `git`

„Bare-Metal"-Programme sind Rust-Programme ohne Standardbibliothek (`no_std`), die Anpassungen am Linker-Vorgang erfordern, um das korrekte Speicherlayout des Programms zu gewährleisten. Dies macht zusätzliche Dateien (wie Linker-Skripte) und Einstellungen (wie Linker-Flags) erforderlich. Wir haben diese Elemente in einer Vorlage zusammengefasst, sodass Sie lediglich die noch fehlenden Angaben ergänzen müssen (etwa den Projektnamen und die Spezifikationen Ihrer Zielhardware).

Unsere Vorlage ist mit `cargo-generate` kompatibel – einem Cargo-Unterbefehl zum Erstellen neuer Cargo-Projekte auf Basis von Vorlagen. Alternativ können Sie die Vorlage auch mithilfe von `git`, `curl`, `wget` oder Ihrem Webbrowser herunterladen.

1.3.2. `cargo-binutils`

`cargo-binutils` ist eine Sammlung von Cargo-Unterbefehlen, die die Verwendung der mit der Rust-Toolchain ausgelieferten LLVM-Werkzeuge vereinfachen. Zu diesen Werkzeugen gehören die LLVM-Varianten von `objdump`, `nm` und `size`, die zur Untersuchung von Binärdateien dienen.

Der Vorteil dieser Werkzeuge gegenüber den GNU-Binutils liegt darin, dass (a) die Installation der LLVM-Werkzeuge unabhängig vom Betriebssystem stets mit demselben einzelnen Befehl (`rustup component add llvm-tools`) erfolgt und (b) Werkzeuge wie `objdump` alle von `rustc` unterstützten Architekturen abdecken – von ARM bis x86_64 –, da sie auf demselben LLVM-Backend basieren.

1.3.3. `qemu-system-arm`

QEMU ist ein Emulator. In diesem Fall verwenden wir die Variante, die ARM-Systeme vollständig emulieren kann. Wir nutzen QEMU, um Embedded-Programme auf dem Host-System auszuführen. Auf diese Weise können Sie einige Teile dieses Buches auch dann nachvollziehen, wenn Sie keine Hardware zur Hand haben!

1.4. Werkzeuge für das Debugging von Embedded Rust

1.4.1. Überblick

Das Debugging eingebetteter Systeme in Rust erfordert spezialisierte Werkzeuge, darunter Software zur Steuerung des Debugging-Prozesses, Debugger zur Überwachung und Kontrolle der Programmausführung sowie Hardware-Probes, die die Interaktion zwischen dem Host-System und dem

eingebetteten Gerät ermöglichen. Dieses Dokument stellt wesentliche Software-Werkzeuge wie Probe-rs und OpenOCD vor, die den Debugging-Prozess vereinfachen und unterstützen, sowie bekannte Debugger wie GDB und die Probe-rs-Erweiterung für Visual Studio Code. Zudem werden wichtige Hardware-Probes wie Rusty-probe, ST-Link, J-Link und MCU-Link behandelt, die für das effiziente Debugging und Programmieren eingebetteter Geräte unerlässlich sind.

1.4.2. Software, die Debugging-Werkzeuge steuert

1.4.2.1. Probe-rs

Probe-rs ist eine moderne, auf Rust ausgerichtete Software für das Debugging in Embedded-Systemen. Im Gegensatz zu OpenOCD wurde Probe-rs mit dem Fokus auf Einfachheit entwickelt und zielt darauf ab, den bei anderen Debugging-Lösungen oft hohen Konfigurationsaufwand zu reduzieren. Es unterstützt diverse Debug-Probes sowie Zielsysteme und bietet eine komfortable Schnittstelle für die Interaktion mit Embedded-Hardware. Probe-rs lässt sich direkt in die Rust-Toolchain sowie – dank einer entsprechenden Erweiterung – in Visual Studio Code integrieren, wodurch Entwickler ihren Debugging-Workflow optimieren können.

1.4.2.2. OpenOCD (Open On-Chip Debugger)

OpenOCD ist ein Open-Source-Softwaretool für das Debugging, Testen und Programmieren eingebetteter Systeme. Es stellt eine Schnittstelle zwischen dem Host-System und der eingebetteten Hardware bereit und unterstützt verschiedene Übertragungsprotokolle wie JTAG und SWD (Serial Wire Debug). OpenOCD lässt sich in GDB, einen Debugger, integrieren. Das Werkzeug ist weit verbreitet, verfügt über eine umfangreiche Dokumentation sowie eine große Community, kann jedoch – insbesondere bei kundenspezifischen Embedded-Konfigurationen – eine komplexe Einrichtung erfordern.

1.4.3. Debugger

Ein Debugger ermöglicht es Entwicklern, die Ausführung eines Programms zu untersuchen und zu steuern, um Fehler oder Bugs zu identifizieren und zu beheben. Er bietet Funktionen wie das Setzen von Haltepunkten (Breakpoints), das schrittweise Durchlaufen des Codes sowie die Überprüfung von Variablenwerten und Speicherzuständen. Debugger sind für eine gründliche Softwareentwicklung und -wartung unerlässlich, da sie Entwicklern helfen sicherzustellen, dass sich ihr Code unter verschiedenen Bedingungen wie vorgesehen verhält.

Debugger bieten folgende Möglichkeiten:
- Auf die im Speicher abgebildeten Register zugreifen.
- Haltepunkte/Überwachungspunkte setzen.
- In/aus im Speicher abgebildeten Registern lesen und schreiben.
- Erkennen, wenn die MCU aufgrund eines Debug-Ereignisses angehalten wurde.
- Die MCU-Ausführung nach dem Auftreten eines Debug-Ereignisses fortsetzen.
- Den Flash-Speicher des Mikrocontrollers löschen und beschreiben.

1.4.3.1. Probe-rs Visual Studio Code Extension

Probe-rs bietet eine Visual Studio Code-Erweiterung, die ein nahtloses Debugging-Erlebnis ohne aufwendige Einrichtung ermöglicht. Dank dieser Integration können Entwickler Rust-spezifische Funktionen wie Pretty-Printing und detaillierte Fehlermeldungen nutzen und so sicherstellen, dass ihr Debugging-Prozess optimal auf das Rust-Ökosystem abgestimmt ist.

1.4.3.2. TRACE32

TRACE32 ist eine von Lauterbach für Embedded-Systeme entwickelte professionelle Debugging- und Tracing-Lösung. Sie unterstützt eine Vielzahl von Prozessorarchitekturen, darunter ARM und RISC-V, und stellt die Verbindung zur Zielhardware über JTAG, SWD sowie diverse Trace-Schnittstellen her.

TRACE32 bietet leistungsstarke Debugging-Funktionen wie Multicore-Debugging, komplexe Breakpoints und Echtzeit-Trace-Analyse. Da das System auf Standard-ELF/DWARF-Debuginformationen basiert, ist es mit Rust-Binärdateien kompatibel, die mit herkömmlichen Werkzeugen erstellt wurden.

1.4.3.3. GDB (GNU Debugger)

GDB ist ein vielseitiges Debugging-Werkzeug, das es Entwicklern ermöglicht, den Zustand von Programmen während der Laufzeit oder nach einem Absturz zu untersuchen. Im Bereich Embedded Rust verbindet sich GDB über OpenOCD oder andere Debugging-Server mit dem Zielsystem, um mit dem Embedded-Code zu interagieren. GDB ist hochgradig konfigurierbar und unterstützt Funktionen wie Remote-Debugging, Variableninspektion und bedingte Haltepunkte. Es ist auf einer Vielzahl von Plattformen einsetzbar und bietet umfassende Unterstützung für Rust-spezifische Debugging-Anforderungen, wie etwa Pretty-Printing und die Integration in IDEs.

1.4.4. Probes

Eine Hardware-Sonde ist ein Gerät, das bei der Entwicklung und dem Debugging eingebetteter Systeme eingesetzt wird, um die Kommunikation zwischen einem Host-Computer und dem eingebetteten Zielsystem zu ermöglichen. Sie unterstützt typischerweise Protokolle wie JTAG oder SWD und ermöglicht so das Programmieren, Debuggen und Analysieren des Mikrocontrollers oder Mikroprozessors auf dem eingebetteten System. Hardware-Sonden sind für Entwickler unverzichtbar, um Haltepunkte (Breakpoints) zu setzen, den Code schrittweise auszuführen sowie Speicher und Prozessorregister zu untersuchen; dies erlaubt es ihnen, Probleme effizient in Echtzeit zu diagnostizieren und zu beheben.

1.4.4.1. Rusty-probe

Rusty-probe ist ein Open-Source-Hardware-Debugging-Interface auf USB-Basis, das für den Einsatz mit probe-rs entwickelt wurde. Die Kombination aus Rusty-Probe und probe-rs bietet Entwicklern, die mit Embedded-Rust-Anwendungen arbeiten, eine benutzerfreundliche und kostengünstige Lösung.

1.4.4.2. ST-Link

Der ST-Link ist ein weit verbreiteter Debugging- und Programmieradapter, der von STMicroelectronics primär für die Mikrocontroller-Serien STM32 und STM8 entwickelt wurde. Er unterstützt sowohl das Debugging als auch die Programmierung über JTAG- oder SWD-Schnittstellen (Serial Wire Debug). Dank der direkten Unterstützung durch das umfangreiche Angebot an Entwicklungsboards von STMicroelectronics sowie der Integration in gängige Entwicklungsumgebungen (IDEs) ist der ST-Link weit verbreitet und stellt eine komfortable Wahl für Entwickler dar, die mit STM-Mikrocontrollern arbeiten.

1.4.4.3. J-Link

J-Link, entwickelt von SEGGER Microcontroller, ist ein robuster und vielseitiger Debugger, der neben ARM auch eine breite Palette weiterer CPU-Kerne und Bausteine – wie etwa RISC-V – unterstützt. J-Link ist für seine hohe Leistungsfähigkeit und Zuverlässigkeit bekannt und unterstützt diverse Kommunikationsschnittstellen, darunter JTAG, SWD sowie Fine-Pitch-JTAG-Schnittstellen. Geschätzt wird das System zudem für seine erweiterten Funktionen, wie etwa unbegrenzte Breakpoints im Flash-Speicher, sowie für seine Kompatibilität mit einer Vielzahl von Entwicklungsumgebungen.

1.4.4.4. MCU-Link

MCU-Link ist ein von NXP Semiconductors bereitgestellter Debugging-Adapter, der auch als Programmiergerät fungiert. Er unterstützt eine Vielzahl von ARM-Cortex-Mikrocontrollern und lässt sich nahtlos in Entwicklungsumgebungen wie die MCUXpresso IDE integrieren. MCU-Link zeichnet sich besonders durch seine Vielseitigkeit und Kosteneffizienz aus und ist damit eine attraktive Lösung für Hobbyanwender, Lehrkräfte und professionelle Entwickler gleichermaßen.

1.5. Werkzeuge installieren

Diese Seite enthält betriebssystemunabhängige Installationsanleitungen für einige der Werkzeuge:

1.5.0.1. Rust-Werkzeuge

Installieren Sie rustup, indem Sie die Anweisungen unter https://rustup.rs.

HINWEIS Stellen Sie sicher, dass Sie eine Compiler-Version verwenden, die mindestens `1.31` entspricht. Der Befehl `rustc -V` sollte ein Datum ausgeben, das neuer ist als das unten angegebene.

```
$ rustc -V
rustc 1.31.1 (b6c32da9b 2018-12-18)
```

Aus Gründen der Bandbreite und der Festplattenauslastung unterstützt die Standardinstallation nur die native Kompilierung. Um die Unterstützung für die Cross-Kompilierung für die ARM-Cortex-M-Architekturen hinzuzufügen, wählen Sie eines der folgenden Kompilierungsziele aus. Für das STM32F3DISCOVERY-Board, das für die Beispiele in diesem Buch verwendet wird, verwenden Sie das Ziel `thumbv7em-none-eabihf`. Finden Sie den für Sie am besten geeigneten Cortex-M.

Cortex-M0, M0+, and M1 (ARMv6-M-Architektur):

```
rustup target add thumbv6m-none-eabi
```

Cortex-M3 (ARMv7-M-Architektur):

```
rustup target add thumbv7m-none-eabi
```

Cortex-M4 und M7 ohne Hardware-Gleitkommafunktion (ARMv7E-M-Architektur):

```
rustup target add thumbv7em-none-eabi
```

Cortex-M4F und M7F mit Hardware-Gleitkomma (ARMv7E-M-Architektur):

```
rustup target add thumbv7em-none-eabihf
```

Cortex-M23 (ARMv8-M-Architektur):

```
rustup target add thumbv8m.base-none-eabi
```

Cortex-M33 und M35P (ARMv8-M-Architektur):

```
rustup target add thumbv8m.main-none-eabi
```

Cortex-M33F und M35PF mit Hardware-Gleitkomma (ARMv8-M-Architektur):

```
rustup target add thumbv8m.main-none-eabihf
```

1.5.0.2. `cargo-binutils`

```
cargo install cargo-binutils

rustup component add llvm-tools
```

WINDOWS: Voraussetzung ist, dass die C++-Build-Tools für Visual Studio 2019 installiert sind. https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16

1.5.0.3. `cargo-generate`

Wir werden dies später verwenden, um ein Projekt aus einer Vorlage zu generieren.

```
cargo install cargo-generate
```

Hinweis: Bei einigen Linux-Distributionen (z. B. Ubuntu) müssen Sie möglicherweise vor der Installation von `cargo-generate` die Pakete `libssl-dev` und `pkg-config` installieren.

1.5.0.4. Betriebssystemspezifische Anweisungen

Befolgen Sie nun die Anweisungen für das von Ihnen verwendete Betriebssystem:

- Linux
- Windows
- macOS

1.5.1. Linux

Hier sind die Installationsbefehle für einige Linux-Distributionen.

1.5.1.1. Pakete

- Ubuntu 18.04 oder neuer / Debian Stretch oder neuer

> **HINWEIS** `gdb-multiarch` ist der GDB-Befehl, den Sie zum Debuggen Ihrer ARM-Cortex-M-Programme verwenden werden.

```
sudo apt install gdb-multiarch openocd qemu-system-arm
```

- Ubuntu 14.04 und 16.04

> **HINWEIS** `arm-none-eabi-gdb` ist der GDB-Befehl, den Sie zum Debuggen Ihrer ARM-Cortex-M-Programme verwenden werden.

```
sudo apt install gdb-arm-none-eabi openocd qemu-system-arm
```

- Fedora 27 oder neuer

```
sudo dnf install gdb openocd qemu-system-arm
```

- Arch Linux

> **HINWEIS** `arm-none-eabi-gdb` ist der GDB-Befehl, den Sie zum Debuggen von ARM-Cortex-M-Programmen verwenden.

```
sudo pacman -S arm-none-eabi-gdb qemu-system-arm openocd
```

1.5.1.2. udev-Regeln

Diese Regel ermöglicht die Nutzung von OpenOCD mit dem Discovery Board ohne Root-Rechte.

Erstellen Sie die Datei `/etc/udev/rules.d/70-st-link.rules` mit dem unten stehenden Inhalt.

```
# STM32F3DISCOVERY rev A/B - ST-LINK/V2
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", TAG+="uaccess"

# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", TAG+="uaccess"
```

Laden Sie anschließend alle udev-Regeln neu mit:

```
sudo udevadm control --reload-rules
```

Wenn Sie die Platine an Ihren Laptop angeschlossen hatten, ziehen Sie den Stecker heraus und schließen Sie ihn dann erneut an.

Sie können die Berechtigungen überprüfen, indem Sie diesen Befehl ausführen:

```
lsusb
```

Was so etwas wie

```
(..)
Bus 001 Device 018: ID 0483:374b STMicroelectronics ST-LINK/V2.1
(..)
```

anzeigen sollte.

Notieren Sie sich die Bus- und Gerätenummern. Verwenden Sie diese Nummern, um einen Pfad wie `/dev/bus/usb/<bus>/<device>` zu bilden. Verwenden Sie diesen Pfad anschließend wie folgt:

```
ls -l /dev/bus/usb/001/018
```

```
crw-------+ 1 root root 189, 17 Sep 13 12:34 /dev/bus/usb/001/018
```

```
getfacl /dev/bus/usb/001/018 | grep user
```

```
user::rw-
user:you:rw-
```

Das angehängte „+“ bei den Berechtigungen weist auf eine erweiterte Berechtigung hin. Der Befehl „getfacl” teilt dem Benutzer mit, dass er dieses Gerät verwenden darf.

Fahren Sie nun mit dem nächsten Abschnitt fort.

1.5.2. macOS

Alle Werkzeuge können mit Homebrew oder MacPorts installiert werden:

1.5.2.1. Werkzeuge mit Homebrew installieren

```
$ # GDB
$ brew install arm-none-eabi-gdb

$ # OpenOCD
$ brew install openocd

$ # QEMU
$ brew install qemu
```

> **HINWEIS** Falls OpenOCD abstürzt, müssen Sie möglicherweise die neueste Version installieren; verwenden Sie dazu:

```
$ brew install --HEAD openocd
```

1.5.2.2. Werkzeuge mit MacPorts installieren

```
$ # GDB
$ sudo port install arm-none-eabi-gcc

$ # OpenOCD
$ sudo port install openocd

$ # QEMU
$ sudo port install qemu
```

Das war’s! Gehen Sie zum nächsten Abschnitt.

1.5.3. Windows

1.5.3.1. `arm-none-eabi-gdb`

ARM stellt `.exe`-Installationsprogramme für Windows bereit. Laden Sie eines von hier herunter und befolgen Sie die Anweisungen. Aktivieren Sie kurz vor Abschluss des Installationsvorgangs die Option „Add path to environment variable“. Überprüfen Sie anschließend, ob die Werkzeuge in Ihrem `%PATH%` enthalten sind:

```
$ arm-none-eabi-gdb -v
GNU gdb (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 8.1.0.20180315-git
(..)
```

1.5.3.2. OpenOCD

Es gibt kein offizielles Binär-Release von OpenOCD für Windows, aber wenn Sie es nicht selbst kompilieren möchten, stellt das xPack-Projekt eine Binärdistribution bereit – zu finden hier. Befolgen Sie die dort aufgeführten Installationsanweisungen. Aktualisieren Sie anschließend Ihre Umgebungsvariable `%PATH%`, indem Sie den Pfad hinzufügen, unter dem die Binärdateien installiert wurden (z. B. `C:\Users\USERNAME\AppData\Roaming\xPacks\@xpack-dev-tools\openocd\0.10.0-13.1\.content\bin\`, falls Sie die einfache Installation verwendet haben).

Überprüfen Sie mit folgendem Befehl, ob OpenOCD in Ihrem `%PATH%` enthalten ist:

```
$ openocd -v
Open On-Chip Debugger 0.10.0
(..)
```

1.5.3.3. QEMU

Lade QEMU von der offiziellen Website herunter.

1.5.3.4. ST-LINK USB driver

Sie müssen außerdem diesen USB-Treiber installieren, da OpenOCD sonst nicht funktioniert. Befolgen Sie die Anweisungen des Installationsprogramms und stellen Sie sicher, dass Sie die richtige Version des Treibers (32-Bit oder 64-Bit) installieren.

Das war's! Gehen Sie zum nächsten Abschnitt.

1.5.4. Die Installation überprüfen

In diesem Abschnitt überprüfen wir, ob einige der erforderlichen Werkzeuge bzw. Treiber korrekt installiert und konfiguriert wurden.

Verbinden Sie Ihren Laptop oder PC über ein Mini-USB-Kabel mit dem Discovery-Board. Das Discovery-Board verfügt über zwei USB-Anschlüsse; verwenden Sie den mit „USB ST-LINK“ beschrifteten Anschluss, der sich mittig an der Platinenkante befindet.

Überprüfen Sie außerdem, ob die ST-LINK-Stiftleiste bestückt ist. Siehe dazu die Abbildung unten; die ST-LINK-Stiftleiste ist dort hervorgehoben.

Führen Sie nun den folgenden Befehl aus:

```
openocd -f interface/stlink.cfg -f target/stm32f3x.cfg
```

> **HINWEIS**: Ältere Versionen von OpenOCD, einschließlich der Version 0.10.0 aus dem Jahr 2017, enthalten nicht die neue (und zu bevorzugende) Datei `interface/stlink.cfg`; stattdessen müssen Sie möglicherweise `interface/stlink-v2.cfg` oder `interface/stlink-v2-1.cfg` verwenden.

Sie sollten die folgende Ausgabe erhalten und das Programm sollte die Konsole blockieren:

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use
'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might
differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.919881
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Der Inhalt stimmt möglicherweise nicht exakt überein, aber die letzte Zeile bezüglich Breakpoints und Watchpoints sollten Sie sehen. Wenn dies der Fall ist, beenden Sie den OpenOCD-Prozess und fahren Sie mit dem nächsten Abschnitt fort.

Falls du die Zeile „breakpoints“ nicht erhalten hast, versuche einen der folgenden Befehle.

```
openocd -f interface/stlink-v2.cfg -f target/stm32f3x.cfg
```

```
openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg
```

Wenn einer dieser Befehle funktioniert, bedeutet das, dass Sie eine ältere Hardware-Revision des Discovery-Boards besitzen. Das stellt kein Problem dar, aber merken Sie sich diesen Umstand, da Sie die Konfiguration später etwas anders vornehmen müssen. Sie können zum nächsten Abschnitt übergehen.

Wenn keiner der Befehle als normaler Benutzer funktioniert, versuchen Sie, sie mit Root-Rechten auszuführen (z. B. `sudo openocd ..`). Sollten die Befehle mit Root-Rechten funktionieren, überprüfen Sie, ob die udev-Regeln korrekt eingerichtet wurden.

Wenn Sie an diesem Punkt angelangt sind und OpenOCD nicht funktioniert, eröffnen Sie bitte eine Problemmeldung, und wir helfen Ihnen weiter!

2. Erste Schritte

In diesem Abschnitt führen wir Sie Schritt für Schritt durch den Prozess des Schreibens, Kompilierens, Flashen und Debuggens von Embedded-Programmen. Sie können die meisten der Beispiele ohne spezielle Hardware ausprobieren, da wir Ihnen die Grundlagen mithilfe von QEMU, einem beliebten Open-Source-Hardware-Emulator, vermitteln. Der einzige Abschnitt, in dem Hardware erforderlich ist, ist natürlich der Abschnitt Hardware, wo wir OpenOCD verwenden, um ein STM32F3DISCOVERY zu programmieren.

2.1. QEMU

QEMU (von englisch „Quick Emulator“) ist eine freie Virtualisierungssoftware, die die gesamte Hardware eines Computers emuliert und durch die dynamische Übersetzung der Prozessorinstruktionen des Gastprozessors (englisch guest) in Instruktionen für den Wirtprozessor (englisch host) eine sehr gute Ausführungsgeschwindigkeit erreicht.

URL: https://www.qemu.org/

QEMU emuliert Systeme mit den folgenden Prozessorarchitekturen:

- 68K,
- Alpha,
- ARM (32- und 64-Bit),
- CHRIS,
- HPPA,
- LatticeMico32,
- m68K bzw. Coldfire,
- MicroBlaze,
- MIPS,
- Moxie,
- Nios2,
- OpenRISC,
- PC
- Power
- PowerNV
- PowerPC (32- und 64-Bit),
- RISC-V,
- S/390,
- SH-4,
- Sparc32/64,
- TILE-Gx,
- TriCore,
- Unicore,
- x86 (x86-32 und x86-64),
- Xtensa

(Stand 2026).

Wir beginnen mit der Entwicklung eines Programms für den LM3S6965, einen ARM Cortex-M3-Mikrocontroller. Wir haben diesen als unser erstes Ziel ausgewählt, da er mit QEMU emuliert werden kann, sodass Sie sich in diesem Abschnitt nicht mit der Hardware beschäftigen müssen und wir uns auf die Werkzeuge und den Entwicklungsprozess konzentrieren können.

WICHTIG In dieser Anleitung verwenden wir den Namen „app“ als Projektnamen. Wann immer Sie das Wort „app” sehen, sollten Sie es durch den Namen ersetzen, den Sie für Ihr Projekt gewählt haben. Alternativ können Sie Ihr Projekt auch „app“ nennen und so die Ersetzungen vermeiden.

2.1.1. Erstellen eines nicht standardmäßigen Rust-Programms

Wir werden die Projektvorlage `cortex-m-quickstart` verwenden, um daraus ein neues Projekt zu erstellen. Das erstellte Projekt enthält eine Minimalanwendung: einen guten Ausgangspunkt für eine neue Embedded-Rust-Anwendung. Darüber hinaus enthält das Projekt ein Verzeichnis `examples` mit mehreren separaten Anwendungen, die einige der wichtigsten Funktionen von Embedded Rust veranschaulichen.

2.1.1.1. Verwendung von `cargo-generate`

Installieren Sie zunächst cargo-generate

```
cargo install cargo-generate
```

Erstellen Sie anschließend ein neues Projekt

```
cargo generate --git https://github.com/knurling-rs/app-template
```

```
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app
```

```
cd app
```

2.1.1.2. Verwendung von `git`

Das Repository klonen

```
git clone https://github.com/rust-embedded/cortex-m-quickstart app
cd app
```

Und fülle dann die Platzhalter in der Datei `Cargo.toml` aus

```
[package]
authors = ["{{authors}}"] # "{{authors}}" -> "John Smith"
edition = "2018"
name = "{{project-name}}" # "{{project-name}}" -> "app"
version = "0.1.0"

# ..

[[bin]]
name = "{{project-name}}" # "{{project-name}}" -> "app"
test = false
bench = false
```

2.1.1.3. Keines von beiden verwenden

Laden Sie den neuesten Snapshot der Vorlage `cortex-m-quickstart` herunter und entpacken Sie ihn.

```
curl -LO https://github.com/rust-embedded/cortex-m-quickstart/archive/master.zip
unzip master.zip
mv cortex-m-quickstart-master app
cd app
```

Oder Sie navigieren zu `cortex-m-quickstart`, klicken auf die grüne Schaltfläche „Clone or download“ und anschließend auf „Download ZIP”.

Füllen Sie anschließend die Platzhalter in der Datei `Cargo.toml` aus, wie im zweiten Teil der Version „Verwendung von `git` beschrieben.

2.1.2. Programmübersicht

Der Einfachheit halber sind hier die wichtigsten Teile des Quellcodes in `src/main.rs` aufgeführt:

```
#![no_std]
#![no_main]

use panic_halt as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {
        // Hier kommt Ihr Code hin.
    }
}
```

Dieses Programm unterscheidet sich ein wenig von einem typischen Rust-Programm, schauen wir es uns also einmal genauer an.

`#![no_std]` gibt an, dass dieses Programm *nicht* mit der Standard-Crate `std` verknüpft wird. Stattdessen wird es mit deren Teilmenge verknüpft: der Crate `core`.

`#![no_main]` gibt an, dass dieses Programm nicht die Standard-`main`- Schnittstelle verwendet, die die meisten Rust-Programme nutzen. Der Hauptgrund (kein Wortspiel beabsichtigt) für die Verwendung von `no_main` ist, dass die Nutzung der `main`-Schnittstelle im `no_std`-Kontext „Nightly“ erfordert.

`use panic_halt as _;`. Diese Crate stellt einen `panic_handler` bereit, der das Verhalten des Programms im Panikfall definiert. Wir werden darauf im Kapitel In Panik geraten des Buches näher eingehen.

`#[entry]` ist ein Attribut des `cortex-m-rt`-Crate, das dazu dient, den Einstiegspunkt des Programms zu kennzeichnen. Da wir nicht die Standard-Schnittstelle `main` verwenden, benötigen wir eine andere Möglichkeit, den Einstiegspunkt des Programms anzugeben, und das wäre `#[entry]`.

`fn main() -> !`. Unser Programm wird der *einzige* Prozess sein, der auf der Zielhardware läuft, deshalb soll es nicht beendet werden! Wir verwenden eine divergent function (das `-> !` in der Funktionssignatur), um bereits zur Kompilierungszeit sicherzustellen, dass dies auch der Fall ist.

2.1.3. Cross-Kompilierung

Zunächst benötigen wir das Speicherlayout für den Ziel-Mikrocontroller, in unserem Fall den LM3S6965. Andernfalls schlägt die Verknüpfung des Images beim Build fehl. Erstellen Sie eine Datei mit dem Namen `memory.x` im Stammverzeichnis des Projekts und fügen Sie den folgenden Inhalt ein:

```
MEMORY
{
  /* Hinweis: 1 K = 1 KiBi = 1024 bytes */
  /* TODO Passen Sie diese Speicherbereiche an das Speicherlayout Ihres Geräts an. */
  /* Diese Werte entsprechen dem LM3S6965, einem der wenigen Bausteine, die
  QEMU emulieren kann */
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}
```

```
/* Hier wird der Aufrufstapel zugewiesen. */
/* Der Stapel ist vom Typ „vollständig absteigend“. */
/* Möglicherweise möchten Sie diese Variable verwenden, um den Aufrufstapel und
  statische Variablen in verschiedenen Speicherbereichen zu lokalisieren.
  Nachstehend ist der Standardwert aufgeführt */
/* _stack_start = ORIGIN(RAM) + LENGTH(RAM); */

/* Mit diesem Symbol können Sie den Speicherort des .text-Abschnitts anpassen. */
/* Wird dies weggelassen, wird der .text-Abschnitt direkt nach dem .vector_table-
  Abschnitt platziert */
/* Dies ist nur bei Mikrocontrollern erforderlich, die bestimmte
  Konfigurationsdaten direkthinter der Vektortabelle speichern */
/* _stext = ORIGIN(FLASH) + 0x400; */

/* Beispiel für die Zuweisung nicht initialisierter Variablen zu
benutzerdefinierten RAM-Speicherplätzen. */
/* Dies setzt voraus, dass Sie oben einen Bereich namens „RAM2“ definiert und
in den Rust-Quelldateien das Attribut `#[link_section = „.ram2bss“]` zu den
Daten hinzugefügt haben, die Sie dort platzieren möchten. */
/* Beachten Sie, dass der Bereich von der Laufzeitumgebung nicht auf Null
initialisiert wird! */
/* SECTIONS {
     .ram2bss (NOLOAD) : ALIGN(4) {
       *(.ram2bss);
       . = ALIGN(4);
     } > RAM2
   } INSERT AFTER .bss;
*/
```

Der nächste Schritt besteht darin, das Programm für die Cortex-M3-Architektur *cross* zu kompilieren. Das geht ganz einfach mit dem Befehl `cargo build --target $TRIPLE`, sofern Sie wissen, wie das Kompilierungsziel (`$TRIPLE`) lauten soll. Glücklicherweise finden Sie die Antwort in der Datei `.cargo/config.toml` in der Vorlage:

```
tail -n6 .cargo/config.toml
```

```
[build]
# Waehlen Sie EINES dieser Kompilierungsziele aus
# target = "thumbv6m-none-eabi"    # Cortex-M0 and Cortex-M0+
target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4 and Cortex-M7 (no FPU)
# target = "thumbv7em-none-eabihf" # Cortex-M4F and Cortex-M7F (with FPU)
```

Für die Cross-Kompilierung für die Cortex-M3-Architektur müssen wir `thumbv7m-none-eabi` verwenden. Dieses Ziel wird bei der Installation der Rust-Werkzeuge nicht automatisch installiert. Jetzt wäre ein guter Zeitpunkt, dieses Ziel zu den Werkzeugen hinzuzufügen, falls Sie dies noch nicht getan haben:

```
rustup target add thumbv7m-none-eabi
```

Da das Kompilierungsziel `thumbv7m-none-eabi` in Ihrer Datei `.cargo/config.toml` als Standard festgelegt wurde, haben die beiden folgenden Befehle dieselbe Wirkung:

```
cargo build --target thumbv7m-none-eabi
cargo build
```

2.1.4. Überprüfen

Nun haben wir eine nicht-native ELF-Binärdatei in `target/thumbv7m-none-eabi/debug/app`. Wir können sie mit `cargo-binutils` untersuchen.

Mit `cargo-readobj` können wir die ELF-Header ausgeben, um zu überprüfen, ob es sich um eine ARM-Binärdatei handelt.

```
cargo readobj --bin app -- --file-headers
```

Bitte beachten Sie:

- `--bin app` ist eine vereinfachte Schreibweise für die Überprüfung der Binärdatei unter `target/$TRIPLE/debug/app`
- `--bin app` kompiliert die Binärdatei bei Bedarf auch (neu)

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0x0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x405
  Start of program headers:          52 (bytes into file)
  Start of section headers:          153204 (bytes into file)
  Flags:                             0x5000200
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         19
  Section header string table index: 18
```

Mit `cargo-size` lässt sich die Größe der Linker-Abschnitte der Binärdatei anzeigen.

```
cargo size --bin app --release -- -A
```

Wir verwenden `--release`, um die optimierte Version zu überprüfen.

```
app  :
section             size        addr
.vector_table       1024         0x0
.text                 92       0x400
.rodata                0       0x45c
.data                  0  0x20000000
.bss                   0  0x20000000
.debug_str          2958         0x0
.debug_loc            19         0x0
.debug_abbrev        567         0x0
.debug_info         4929         0x0
.debug_ranges         40         0x0
.debug_macinfo         1         0x0
.debug_pubnames     2035         0x0
.debug_pubtypes     1892         0x0
.ARM.attributes       46         0x0
.debug_frame         100         0x0
.debug_line          867         0x0
Total              14570
```

Eine Auffrischung zum Thema ELF-Linker-Abschnitte

- `.text` enthält die Programmbefehle
- `.rodata` enthält konstante Werte wie Zeichenfolgen
- `.data` enthält statisch zugewiesene Variablen, deren Anfangswerte *nicht* Null
- Die Datei `.bss` enthält außerdem statisch zugewiesene Variablen, deren Anfangswerte *sind* Null
- `.vector_table` ist ein *nicht* standardmäßiger Abschnitt, in dem wir den Vektor speichern (interrupt) table
- Die Abschnitte `.ARM.attributes` und `.debug_*` enthalten Metadaten und werden *nicht* beim Flashen der Binärdatei auf das Zielgerät geladen werden.

WICHTIG: ELF-Dateien enthalten Metadaten wie Debug-Informationen, sodass ihre *Größe auf der Festplatte nicht* genau den Speicherplatz widerspiegelt, den das Programm beanspruchen wird, wenn es auf ein Gerät geflasht wird. Verwenden Sie *immer* `cargo-size`, um zu überprüfen, wie groß eine Binärdatei tatsächlich ist.

Mit `cargo-objdump` lässt sich die Binärdatei disassemblieren.

```
cargo objdump --bin app --release -- --disassemble --no-show-raw-insn --print-imm-hex
```

> **HINWEIS**: Falls der obige Befehl die Fehlermeldung „Unbekanntes Befehlszeilenargument“ ausgibt, siehe den folgenden Fehlerbericht: https://github.com/rust-embedded/book/issues/269

> **HINWEIS** Diese Ausgabe kann auf Ihrem System abweichen. Neuere Versionen von rustc, LLVM und Bibliotheken können unterschiedlichen Assemblercode erzeugen. Wir haben einige der Befehle gekürzt, um den Ausschnitt kurz zu halten.

```
app:  file format ELF32-arm-little

Disassembly of section .text:
main:
     400: bl  #0x256
     404: b #-0x4 <main+0x4>

Reset:
     406: bl  #0x24e
     40a: movw  r0, #0x0
     < .. truncated any more instructions .. >

DefaultHandler_:
     656: b #-0x4 <DefaultHandler_>

UsageFault:
     657: strb  r7, [r4, #0x3]

DefaultPreInit:
     658: bx  lr

__pre_init:
     659: strb  r7, [r0, #0x1]

__nop:
     65a: bx  lr

HardFaultTrampoline:
```

```
     65c: mrs r0, msp
     660: b #-0x2 <HardFault_>

HardFault_:
     662: b #-0x4 <HardFault_>

HardFault:
     663: <unknown>
```

2.1.5. Ausführen

Als Nächstes schauen wir uns an, wie man ein eingebettetes Programm auf QEMU ausführt! Diesmal verwenden wir das `hello`-Beispiel, das tatsächlich etwas tut. Standardmäßig nutzt dieses Beispiel `[defmt]` und RTT, um Text auszugeben.

> **HINWEIS** `defmt` ist eine Abhängigkeit eines Drittanbieters (d. h. keine Kernkomponente), die im Embedded-Rust-Ökosystem weit verbreitet ist.

Um die von `defmt` im Host erzeugten Nachrichten lesen und entschlüsseln zu können, müssen wir die RTT-Transportausgabe auf „Semihosting“ umstellen. Bei Verwendung von echter Hardware erfordert dies eine Debug-Sitzung, bei Verwendung von QEMU funktioniert dies jedoch einfach so.

Stellen wir die Abhängigkeiten um:

```
cargo remove defmt-rtt
cargo add defmt-semihosting
```

Öffnen Sie `src/lib.rs` und ersetzen Sie `use defmt_rtt as _;` durch `use defmt_semihosting as _;`.

Nun können wir das Beispiel kompilieren:

```
cargo build --bin hello
```

Die ausgegebene Binärdatei befindet sich unter `target/thumbv7m-none-eabi/debug/hello`.

Um diese Binärdatei unter QEMU auszuführen, reicht in der Regel der folgende Befehl aus:

```
qemu-system-arm \
  -cpu cortex-m3 \
  -machine lm3s6965evb \
  -nographic \
  -semihosting-config enable=on,target=native \
  -kernel target/thumbv7m-none-eabi/debug/hello
```

Da wir in unserem Fall `defmt` verwenden, kann der Host die Ausgabe nicht dekodieren. Stattdessen benötigen wir ein Werkzeug von Ferrous Systems namens `qemu-run`:

```
git clone git@github.com:knurling-rs/defmt.git
cd defmt/qemu-run/
cargo run -- --machine lm3s6965evb ../qemu-rs/target/thumbv7m-none-eabi/debug/hello
```

```
Hello, world!
```

Der Befehl sollte nach der Ausgabe des Textes erfolgreich beendet werden (Exit-Code = 0). Unter *nix können Sie dies mit dem folgenden Befehl überprüfen:

```
echo $?
```

```
0
```

Schauen wir uns diesen QEMU-Befehl einmal genauer an:

- `qemu-system-arm`. Dies ist der QEMU-Emulator. Es gibt einige Varianten dieser QEMU-Binärdateien; diese hier führt eine vollständige *System*-Emulation von *ARM*-Rechnern durch, daher der Name.
- `-cpu cortex-m3`. Damit wird QEMU angewiesen, eine Cortex-M3-CPU zu emulieren. Durch die Angabe des CPU-Modells lassen sich einige Kompilierungsfehler erkennen: Wenn man beispielsweise ein Programm ausführt, das für den Cortex-M4F kompiliert wurde, der über eine Hardware-FPU verfügt, löst dies bei QEMU während der Ausführung einen Fehler aus.
- `-machine lm3s6965evb`. Damit wird QEMU angewiesen, das LM3S6965EVB zu emulieren, ein Evaluierungsboard, das einen LM3S6965-Mikrocontroller enthält.
- `-nographic`. Damit wird QEMU angewiesen, seine grafische Benutzeroberfläche nicht zu starten.
- `-semihosting-config (..)`. Damit wird QEMU angewiesen, Semihosting zu aktivieren. Semihosting ermöglicht es dem emulierten Gerät unter anderem, die Host-Ausgabe (stdout), die Host-Fehlerausgabe (stderr) und die Host-Eingabe (stdin) zu nutzen sowie Dateien auf dem Host zu erstellen.
- `-kernel $file`. Damit wird QEMU mitgeteilt, welche Binärdatei auf der emulierten Maschine geladen und ausgeführt werden soll.

Das Eintippen dieses langen QEMU-Befehls ist viel zu mühsam! Wir können einen benutzerdefinierten Runner einrichten, um den Vorgang zu vereinfachen. In der Datei `.cargo/config.toml` gibt es einen auskommentierten Runner, der QEMU aufruft; entfernen wir die Auskommentierung:

```
head -n3 .cargo/config.toml
```

```
[target.thumbv7m-none-eabi]
# Entferne den Kommentar hier, damit `cargo run` Programme auf QEMU ausfuehrt
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"
```

Dieser Runner gilt nur für das Ziel `thumbv7m-none-eabi`, das unser Standard-Kompilierungsziel ist. Nun kompiliert `cargo run` das Programm und führt es auf QEMU aus:

```
cargo ru(
  level: 2 + whole
)--release
```

```
   Compiling app v0.1.0 (file:///tmp/app)
    Finished release [optimized + debuginfo] target(s) in 0.26s
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel target/thumbv7m-none-eabi/release/
examples/hello`
Hello, world!
```

2.1.6. Fehlerbehebung (Debugging)

Das Debuggen ist für die Embedded-Entwicklung von entscheidender Bedeutung. Schauen wir uns einmal an, wie es funktioniert.

Das Debuggen eines Embedded-Geräts erfolgt *remote*, da das Programm, das wir debuggen möchten, nicht auf dem Rechner läuft, auf dem das Debugger-Programm (GDB oder LLDB) ausgeführt wird.

Beim Remote-Debugging sind ein Client und ein Server beteiligt. In einer QEMU-Umgebung ist der Client ein GDB- (oder LLDB-)Prozess und der Server der QEMU-Prozess, auf dem auch das eingebettete Programm läuft.

In diesem Abschnitt verwenden wir das bereits kompilierte Beispiel `hello`.

Der erste Schritt beim Debuggen besteht darin, QEMU im Debugging-Modus zu starten:

```
qemu-system-arm \
  -cpu cortex-m3 \
  -machine lm3s6965evb \
```

```
  -nographic \
  -semihosting-config enable=on,target=native \
  -gdb tcp::3333 \
  -S \
  -kernel target/thumbv7m-none-eabi/debug/examples/hello
```

Dieser Befehl gibt nichts auf der Konsole aus und blockiert das Terminal. Wir haben diesmal zwei zusätzliche Flags übergeben:

- `-gdb tcp::3333`. Damit wird QEMU angewiesen, auf eine GDB-Verbindung über den TCP-Port 3333 zu warten.
- `-S`. Damit wird QEMU angewiesen, die Maschine beim Start einzufrieren. Ohne diese Option hätte das Programm das Ende der Funktion `main` erreicht, bevor wir die Gelegenheit gehabt hätten, den Debugger zu starten!

Als Nächstes starten wir GDB in einem anderen Terminal und weisen es an, die Debug-Symbole des Beispiels zu laden:

```
gdb-multiarch -q target/thumbv7m-none-eabi/debug/examples/hello
```

HINWEIS: Möglicherweise benötigen Sie anstelle von `gdb-multiarch` eine andere Version von gdb, je nachdem, welche Sie im Kapitel zur Installation installiert haben. Dies könnte auch `arm-none-eabi-gdb` oder einfach nur `gdb` sein.

Anschließend stellen wir innerhalb der GDB-Shell eine Verbindung zu QEMU her, das auf dem TCP-Port 3333 auf eine Verbindung wartet.

```
target remote :3333
```

```
Remote debugging using :3333
Reset () at $REGISTRY/cortex-m-rt-0.6.1/src/lib.rs:473
473     pub unsafe extern "C" fn Reset() -> ! {
```

Sie werden feststellen, dass der Prozess angehalten wurde und der Programmzähler auf eine Funktion namens `Reset` zeigt. Das ist der Reset-Handler: das, was Cortex-M-Kerne beim Booten ausführen.

> Beachten Sie, dass gdb in manchen Konfigurationen anstelle der oben gezeigten Zeile `Reset () at $REGISTRY/cortex-m-rt-0.6.1/src/lib.rs:473` möglicherweise Warnungen wie die folgenden ausgibt:
>
> `core::num::bignum::Big32x40::mul_small () at src/libcore/num/bignum.rs:254 src/libcore/num/bignum.rs: No such file or directory.`
>
> Das ist ein bekannter Fehler. Sie können diese Warnungen getrost ignorieren, da Sie sich höchstwahrscheinlich bei `Reset()` befinden.

Dieser Reset-Handler ruft schließlich unsere Hauptfunktion auf. Lassen Sie uns den gesamten Weg dorthin mithilfe eines Haltepunkts und des Befehls `continue` überspringen. Um den Haltepunkt zu setzen, schauen wir uns zunächst mit dem Befehl `list` an, an welcher Stelle in unserem Code wir anhalten möchten.

```
list main
```

Dadurch wird der Quellcode aus der Datei „examples/hello.rs“ angezeigt.

```
6       use panic_halt as _;
7
8       use cortex_m_rt::entry;
9       use cortex_m_semihosting::{debug, hprintln};
```

```
10
11      #[entry]
12      fn main() -> ! {
13          hprintln!("Hello, world!").unwrap();
14
15          // exit QEMU
```

Wir möchten einen Haltepunkt direkt vor „Hello, world!“ setzen, das sich in Zeile 13 befindet. Dazu verwenden wir den Befehl `break`:

```
break 13
```

Wir können gdb nun mit dem Befehl `continue` anweisen, bis zu unserer Hauptfunktion weiterzulaufen:

```
continue
```

```
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main () at examples\hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```

Wir sind nun fast bei dem Code angelangt, der „Hello, world!“ ausgibt. Machen wir weiter mit dem Befehl `next`.

```
next
```

```
16          debug::exit(debug::EXIT_SUCCESS);
```

An dieser Stelle sollte auf dem Terminal, auf dem `qemu-system-arm` läuft, „Hello, world!“ angezeigt werden.

```
$ qemu-system-arm (..)
Hello, world!
```

Ein erneuter Aufruf von `next` beendet den QEMU-Prozess.

```
next
```

```
[Inferior 1 (Remote target) exited normally]
```

Sie können die GDB-Sitzung nun beenden.

```
quit
```

3. Hardware

Mittlerweile sollten Sie mit den Werkzeugen und dem Entwicklungsprozess einigermaßen vertraut sein. In diesem Abschnitt wechseln wir zur echten Hardware; der Prozess bleibt im Großen und Ganzen gleich. Tauchen wir ein.

3.0.1. Kennen Sie Ihre Hardware

Bevor wir beginnen, müssen Sie einige Eigenschaften des Zielgeräts identifizieren da diese zur Konfiguration des Projekts verwendet werden:

- Der ARM-Kern, z.B. Cortex-M3.
- Enthält der ARM-Kern eine FPU? Cortex-M4**F** und Cortex-M7**F**-Kerne tun dies.
- Wie viel Flash-Speicher und RAM hat das Zielgerät? z.B. 256 KiB von Flash und 32 KiB RAM.
- Wo werden Flash-Speicher und RAM im Adressraum abgebildet? zB RAM ist befindet sich üblicherweise an der Adresse `0x2000_0000`.

Diese Informationen finden Sie im Datenblatt oder im Referenzhandbuch Ihres/Ihrer Microcontrollers/Microcontrollerplatine.

In diesem Abschnitt verwenden wir unsere Referenzhardware, die STM32F3DISCOVERY. Diese Platine enthält einen Mikrocontroller STM32F303VCT6. Dieser Mikrocontroller verfügt über:

- Ein Cortex-M4F-Kern, der eine einzelne Präzisions-FPU enthält
- 256 KiB Flash befinden sich an der Adresse 0x0800_0000.
- 440 KiB RAM an der Adresse 0x2000_0000. (Es gibt noch eine weitere RAM-Region, aber der Einfachheit halber ignorieren wir das).

3.0.2. Konfigurieren

Wir beginnen bei Null mit einer neuen Vorlageninstanz. Zur Auffrischung siehe vorheriger Abschnitt zu QEMU, wie man das ohne `cargo-generate` macht.

```
$ cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app

$ cd app
```

Schritt Nummer eins besteht darin, ein Standardkompilierungsziel in `.cargo/config.toml` festzulegen.

```
tail -n5 .cargo/config.toml
```

```
# Waehlen Sie EINES von diesen Kompilierungszielen aus
# target = "thumbv6m-none-eabi"    # Cortex-M0 and Cortex-M0+
# target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4 and Cortex-M7 (no FPU)
target = "thumbv7em-none-eabihf" # Cortex-M4F and Cortex-M7F (with FPU)
```

Wir verwenden `thumbv7em-none-eabihf`, da es den Cortex-M4F-Kern abdeckt.

> **HINWEIS**: Wie Sie sich vielleicht aus dem vorherigen Kapitel erinnern, müssen wir alle Ziele installieren, dies ist ein neues Ziel. Vergessen Sie also nicht, die Installationsprozess `rustup target add thumbv7em-none-eabihf` für dieses Ziel auszuführen.

Der zweite Schritt besteht darin, die Speicherbereichsinformationen in die Datei `memory.x` einzugeben

```
$ cat memory.x
/* Linker script for the STM32F303VCT6 */
MEMORY
{
  /* NOTE 1 K = 1 KiBi = 1024 bytes */
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 40K
}
```

> **NOTE**: Wenn Sie aus irgendeinem Grund die Datei `memory.x` geändert haben, nachdem Sie den ersten Build für ein bestimmtes Build-Ziel erstellt haben, dann führen Sie `cargo clean` vor `cargo build` aus, da `cargo build` keine Änderungen von `memory.x` verfolgt.

Wir beginnen noch einmal mit dem Hallo-Beispiel, aber zuerst müssen wir eine kleine Änderung vornehmen.

Stellen Sie in `examples/hello.rs`sicher, daß `debug::exit()` auskommentiert oder entfernt wurde. Es wird nur zum Ausführen in QEMU verwendet.

```
#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    // exit QEMU
    // HINWEIS: Fuehren Sie dies nicht auf der Hardware aus; es kann den
    //          OpenOCD-Zustand beschaedigen.
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Sie können nun Programme mittels `cargo build` cross-kompilieren und die Binärdateien mit `cargo-binutils` untersuchen, genau wie zuvor. Das `cortex-m-rt`-Crate übernimmt die gesamte für den Betrieb des Chips erforderliche „Magie“, da erfreulicherweise so gut wie alle Cortex-M-CPUs auf die gleiche Weise booten.

```
cargo build --example hello
```

3.0.3. Fehlerbehebung (Debugging)

Das Debugging gestaltet sich etwas anders. Tatsächlich können sich bereits die ersten Schritte je nach Zielgerät unterscheiden. In diesem Abschnitt zeigen wir die Schritte, die zum Debuggen eines auf dem STM32F3DISCOVERY laufenden Programms erforderlich sind. Dies soll als Referenz dienen; für gerätespezifische Informationen zum Debugging konsultieren Sie bitte das Debugonomicon.

Wie zuvor führen wir Remote-Debugging durch, wobei der Client ein GDB-Prozess ist. Diesmal ist der Server jedoch OpenOCD.

Verbinden Sie, wie im Abschnitt “Die Installation überprüfen” beschrieben, das Discovery Board mit Ihrem Laptop/PC und prüfen Sie, ob der ST-LINK-Header ausgefüllt ist.

Führen Sie in einem Terminal `openocd` aus, um eine Verbindung zum ST-LINK auf dem Discovery-Board herzustellen. Führen Sie diesen Befehl im Stammverzeichnis der Vorlage aus; `openocd` greift dabei auf die Datei `openocd.cfg` zu, in der festgelegt ist, welche Schnittstellen- und Zieldateien verwendet werden sollen.

```
cat openocd.cfg
```

```
# Beispiel-OpenOCD-Konfiguration für das STM32F3DISCOVERY-Entwicklungsboard

# Je nach der vorliegenden Hardware-Revision muessen Sie eine dieser
# Schnittstellen auswaehlen. Es sollte jeweils nur eine Schnittstelle
# auskommentiert sein.

# Revision C (neuere Revision)
source [find interface/stlink.cfg]

# Revision A and B (aeltere Revisionen)
# source [find interface/stlink-v2.cfg]

source [find target/stm32f3x.cfg]
```

> **HINWEIS** Falls Sie im Abschnitt "Die Installation überprüfen" feststellen, dass Sie eine ältere Revision des Discovery-Boards besitzen, sollten Sie an dieser Stelle die Datei `openocd.cfg` so anpassen, dass `interface/stlink-v2.cfg` verwendet wird.

```
$ openocd
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use
'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might
differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.913879
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Starten Sie in einem weiteren Terminal GDB, ebenfalls vom Stammverzeichnis des Templates aus.

```
gdb-multiarch -q target/thumbv7em-none-eabihf/debug/examples/hello
```

HINWEIS: Wie bereits erwähnt, benötigen Sie möglicherweise anstelle von `gdb-multiarch` eine andere Version von gdb, je nachdem, welche Version Sie im Installationskapitel installiert haben. Dies könnte beispielsweise `arm-none-eabi-gdb` oder einfach `gdb` sein.

Verbinden Sie nun GDB mit OpenOCD, das auf eine TCP-Verbindung an Port 3333 wartet.

```
(gdb) target remote :3333
Remote debugging using :3333
0x00000000 in ?? ()
```

Fahren Sie nun damit fort, das Programm mithilfe des Befehls `load` auf den Mikrocontroller zu *flashen* (zu laden).

```
(gdb) load
Loading section .vector_table, size 0x400 lma 0x8000000
```

```
Loading section .text, size 0x1518 lma 0x8000400
Loading section .rodata, size 0x414 lma 0x8001918
Start address 0x08000400, load size 7468
Transfer rate: 13 KB/sec, 2489 bytes/write.
```

Das Programm ist nun geladen. Da dieses Programm Semihosting verwendet, müssen wir OpenOCD anweisen, Semihosting zu aktivieren, bevor wir einen entsprechenden Aufruf tätigen. Befehle können mithilfe des Befehls `monitor` an OpenOCD gesendet werden.

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

> Sie können sich alle OpenOCD-Befehle anzeigen lassen, indem Sie den Befehl `monitor help` aufrufen.

Wie zuvor können wir mithilfe eines Breakpoints und des Befehls `continue` direkt zu `main` springen.

```
(gdb) break main
Breakpoint 1 at 0x8000490: file examples/hello.rs, line 11.
Note: automatically using hardware breakpoints for read-only addresses.

(gdb) continue
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main_trampoline () at examples/hello.rs:11
11      #[entry]
```

> **HINWEIS** Falls GDB das Terminal blockiert, anstatt am Haltepunkt (Breakpoint) anzuhalten, nachdem Sie den oben genannten Befehl `continue` eingegeben haben, sollten Sie überprüfen, ob die Angaben zum Speicherbereich in der Datei `memory.x` für Ihr Gerät korrekt konfiguriert sind (sowohl die Startadressen *als auch* die Längen).

Springen Sie mit `step` in die main-Funktion.

```
(gdb) step
halted: PC: 0x08000496
hello::__cortex_m_rt_main () at examples/hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```

Nachdem Sie das Programm mit `next` weitergeführt haben, sollten Sie unter anderem „Hello, world!“ auf der OpenOCD-Konsole ausgegeben sehen.

```
$ openocd
(..)
Info : halted: PC: 0x08000502
Hello, world!
Info : halted: PC: 0x080004ac
Info : halted: PC: 0x080004ae
Info : halted: PC: 0x080004b0
Info : halted: PC: 0x080004b4
Info : halted: PC: 0x080004b8
Info : halted: PC: 0x080004bc
```

Die Meldung wird nur einmal angezeigt, kurz bevor das Programm in die in Zeile 19 definierte Endlosschleife eintritt: `loop {}`

Sie können GDB nun mit dem Befehl `quit` beenden.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n)
```

Das Debugging erfordert nun einige zusätzliche Schritte; daher haben wir all diese Schritte in einem einzigen GDB-Skript namens `openocd.gdb` zusammengefasst. Die Datei wurde während des Schritts `cargo generate` erstellt und sollte ohne Änderungen funktionieren. Werfen wir einen Blick darauf:

```
cat openocd.gdb
```

```
target extended-remote :3333

# print demangled symbols
set print asm-demangle on

# detect unhandled exceptions, hard faults and panics
break DefaultHandler
break HardFault
break rust_begin_unwind

monitor arm semihosting enable

load

# start the process but immediately halt the processor
stepi
```

Wenn Sie nun den Befehl `<gdb> -x openocd.gdb target/thumbv7em-none-eabihf/debug/examples/hello` ausführen, verbindet sich GDB sofort mit OpenOCD, aktiviert Semihosting, lädt das Programm und startet den Prozess.

Alternativ können Sie `<gdb> -x openocd.gdb` als benutzerdefinierten Runner einrichten, sodass `cargo run` das Programm baut *und* eine GDB-Sitzung startet. Dieser Runner ist bereits in der Datei `.cargo/config.toml` enthalten, jedoch auskommentiert.

```
head -n10 .cargo/config.toml
```

```
[target.thumbv7m-none-eabi]
# Dies auskommentieren, damit `cargo run` Programme auf QEMU ausfuehrt.
# runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
# Heben Sie die Auskommentierung einer dieser drei Optionen auf, damit
# `cargo run` eine GDB-Sitzung startet.
# Welche Option Sie waehlen sollten, haengt von Ihrem System ab.
runner = "arm-none-eabi-gdb -x openocd.gdb"
# runner = "gdb-multiarch -x openocd.gdb"
# runner = "gdb -x openocd.gdb"
```

```
$ cargo run --example hello
(..)
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x1e70 lma 0x8000400
```

```
Loading section .rodata, size 0x61c lma 0x8002270
Start address 0x800144e, load size 10380
Transfer rate: 17 KB/sec, 3460 bytes/write.
(gdb)
```

3.1. Im Speicher abgebildete Register

Bei eingebetteten Systemen stößt man mit der reinen Ausführung von Standard-Rust-Code und dem Verschieben von Daten im Arbeitsspeicher (RAM) irgendwann an Grenzen. Wenn wir Informationen in das System einspeisen oder daraus ausgeben wollen – sei es das Blinken einer LED, das Erkennen eines Tastendrucks oder die Kommunikation mit einer externen Peripheriekomponente über einen Bus –, müssen wir uns mit der Welt der Peripherieeinheiten und deren „im Speicher abgebildeten Registern“ (memory-mapped registers) befassen.

Möglicherweise stellen Sie fest, dass der für den Zugriff auf die Peripherie Ihres Mikrocontrollers erforderliche Code bereits auf einer der folgenden Ebenen implementiert wurde:

- Mikroarchitektur-Crate – Diese Art von Crate verwaltet alle nützlichen Routinen, die dem Prozessorkern gemeinsam sind, den Ihr Mikrocontroller verwendet, sowie alle Peripheriegeräte, die allen Mikrocontrollern gemeinsam sind, die diesen bestimmten Prozessorkerntyp verwenden. Das cortex-m - Crate bietet Ihnen beispielsweise Funktionen zum Aktivieren und Deaktivieren von Interrupts, die für alle Cortex-M-basierten Mikrocontroller gleich sind. Außerdem erhalten Sie Zugriff auf die „SysTick“-Peripherie, die in allen Cortex-M-basierten Mikrocontrollern enthalten ist.
- Peripheral Access Crate (PAC) – Bei dieser Art von Crate handelt es sich um einen schlanken Adapter (Wrapper) für die verschiedenen Register, die für das spezifische Mikrocontroller-Modell definiert sind, das Sie verwenden. Zum Beispiel, tm4c123x für die Tiva-C-TM4C123-Serie von Texas Instruments oder stm32f30x für die STM32F30x-Serie von STMicroelectronics. Hierbei greifen Sie direkt auf die Register zu und befolgen dabei die Betriebshinweise für die jeweilige Peripherieeinheit, wie sie im technischen Referenzhandbuch Ihres Mikrocontrollers beschrieben sind.
- HAL-Crate – Diese Crates bieten eine benutzerfreundlichere API für einen bestimmten Prozessor, häufig durch die Implementierung gängiger Traits, die in embedded-hal definiert sind. So könnte ein solches Crate beispielsweise eine `Serial`-Struktur bereitstellen, deren Konstruktor eine geeignete Kombination aus GPIO-Pins sowie eine Baudrate entgegennimmt und eine Funktion wie `write_byte` zum Senden von Daten anbietet. Weitere Informationen zu embedded-hal finden Sie im Kapitel über Portabilität.
- Board-Crate – Diese Crates gehen noch einen Schritt weiter als HAL-Crates, indem sie verschiedene Peripheriekomponenten und GPIO-Pins passend für das jeweils verwendete Entwickler-Kit oder Board vorkonfigurieren – wie zum Beispiel stm32f3-discovery für das STM32F3DISCOVERY-Board.

3.1.1. Board Crate

Eine Board-Crate ist der ideale Ausgangspunkt für den Einstieg in Embedded Rust. Sie abstrahiert auf angenehme Weise die Hardware-Details, die Anfänger in diesem Bereich oft überfordern können, und erleichtert Standardaufgaben wie das Ein- oder Ausschalten einer LED. Der bereitgestellte Funktionsumfang variiert jedoch stark von Board zu Board. Da dieses Buch hardwareunabhängig bleiben soll, werden Board-Crates hier nicht behandelt.

Wenn Sie mit dem STM32F3DISCOVERY-Board experimentieren möchten, ist es sehr empfehlenswert, sich das stm32f3-discovery-Board-Crate anzusehen; dieses bietet Funktionen, um die LEDs des Boards blinken zu lassen sowie auf den Kompass, Bluetooth und mehr zuzugreifen. Das Discovery-Buch bietet eine hervorragende Einführung in die Verwendung eines Board-Crates.

Wenn Sie jedoch an einem System arbeiten, für das es noch kein dediziertes Board-Crate gibt, oder wenn Sie Funktionen benötigen, die von vorhandenen Crates nicht abgedeckt werden, lesen Sie weiter: Wir beginnen ganz unten, bei den Mikroarchitektur-Crates.

3.1.2. Mikroarchitektur-Crate

Betrachten wir die SysTick-Peripherie, die allen Mikrocontrollern auf Cortex-M-Basis gemeinsam ist. Im cortex-m-Crate finden wir eine recht hardwarenahe API, die sich folgendermaßen verwenden lässt:

```
#![no_std]
#![no_main]
use cortex_m::peripheral::{syst, Peripherals};
use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    let mut systick = peripherals.SYST;
    systick.set_clock_source(syst::SystClkSource::Core);
    systick.set_reload(1_000);
    systick.clear_current();
    systick.enable_counter();
    while !systick.has_wrapped() {
        // Loop
    }

    loop {}
}
```

Die Funktionen der `SYST`-Struktur entsprechen weitgehend der Funktionalität, die im ARM Technical Reference Manual für diese Peripherieeinheit definiert ist. Diese API bietet keine Möglichkeit, eine Verzögerung von „X Millisekunden“ direkt umzusetzen; wir müssen dies stattdessen selbst auf einfache Weise mittels einer `while`-Schleife implementieren. Beachten Sie, dass wir erst auf die `SYST`-Struktur zugreifen können, nachdem wir `Peripherals::take()` aufgerufen haben. Dabei handelt es sich um eine spezielle Routine, die sicherstellt, dass im gesamten Programm nur eine einzige `SYST`-Instanz existiert. Weitere Informationen dazu finden Sie im Abschnitt “Peripherien”.

3.1.3. Verwendung eines Peripheral Access Crate (PAC)

Bei der Entwicklung von Embedded-Software kommen wir nicht weit, wenn wir uns auf die grundlegenden Peripheriekomponenten beschränken, die in jedem Cortex-M enthalten sind. Irgendwann müssen wir Code schreiben, der speziell auf den verwendeten Mikrocontroller zugeschnitten ist. Gehen wir in diesem Beispiel davon aus, dass wir einen Texas Instruments TM4C123 verwenden – einen soliden 80-MHz-Cortex-M4 mit 256 KiB Flash-Speicher. Um diesen Chip nutzen zu können, binden wir das tm4c123x-Crate ein.

```
#![no_std]
#![no_main]

use panic_halt as _; // panic handler

use cortex_m_rt::entry;
use tm4c123x;

#[entry]
pub fn init() -> (Delay, Leds) {
```

```
    let cp = cortex_m::Peripherals::take().unwrap();
    let p = tm4c123x::Peripherals::take().unwrap();

    let pwm = p.PWM0;
    pwm.ctl.write(|w| w.globalsync0().clear_bit());
    // Mode = 1 => Count up/down mode
    pwm._2_ctl.write(|w| w.enable().set_bit().mode().set_bit());
    pwm._2_gena.write(|w| w.actcmpau().zero().actcmpad().one());
    // 528 cycles (264 up and down) = 4 loops per video line (2112 cycles)
    pwm._2_load.write(|w| unsafe { w.load().bits(263) });
    pwm._2_cmpa.write(|w| unsafe { w.compa().bits(64) });
    pwm.enable.write(|w| w.pwm4en().set_bit());
}
```

Wir haben auf die `PWM0`-Peripherie auf genau dieselbe Weise zugegriffen wie zuvor auf die `SYST`-Peripherie, mit dem Unterschied, dass wir `tm4c123x::Peripherals::take()` aufgerufen haben. Da dieses Crate mithilfe von svd2rust automatisch generiert wurde, erwarten die Zugriffsfunktionen für unsere Registerfelder einen Closure anstelle eines numerischen Arguments. Auch wenn dies nach viel Code aussieht, kann der Rust-Compiler ihn nutzen, um eine Reihe von Prüfungen für uns durchzuführen und anschließend Maschinencode zu erzeugen, der handgeschriebenem Assemblercode sehr nahekommt! Wenn der automatisch generierte Code nicht feststellen kann, ob alle möglichen Argumente für eine bestimmte Zugriffsfunktion gültig sind (zum Beispiel, wenn die SVD das Register als 32-Bit-Register definiert, aber nicht angibt, ob bestimmte dieser 32-Bit-Werte eine besondere Bedeutung haben), wird die Funktion als `unsafe` markiert. Dies lässt sich im obigen Beispiel beim Setzen der Unterfelder `load` und `compa` mittels der Funktion `bits()` beobachten.

3.1.3.1. Lesen

Die Funktion `read()` gibt ein Objekt zurück, das schreibgeschützten Zugriff auf die verschiedenen Teilfelder dieses Registers gewährt – entsprechend der vom Hersteller für diesen Chip bereitgestellten SVD-Datei. Alle für den speziellen Rückgabetyp `R` dieses spezifischen Registers (innerhalb der jeweiligen Peripherieeinheit auf diesem Chip) verfügbaren Funktionen finden Sie in der tm4c123x-Dokumentation.

```
if pwm.ctl.read().globalsync0().is_set() {
    // Tu etwas
}
```

3.1.3.2. Schreiben

Die Funktion `write()` erwartet einen Closure mit einem einzelnen Argument. Üblicherweise bezeichnen wir dieses als `w`. Dieses Argument gewährt Lese- und Schreibzugriff auf die verschiedenen Teilfelder innerhalb des Registers, wie sie in der SVD-Datei des Herstellers für diesen Chip definiert sind. Auch hier gilt: Alle für `w` verfügbaren Funktionen – spezifisch für dieses Register, diese Peripherieeinheit und diesen Chip – finden Sie in der tm4c123x-Dokumentation. Beachten Sie, dass alle Teilfelder, die wir nicht explizit setzen, automatisch auf einen Standardwert gesetzt werden; bereits vorhandene Inhalte des Registers gehen dabei verloren.

```
pwm.ctl.write(|w| w.globalsync0().clear_bit());
```

3.1.3.3. Ändern

Wenn wir in diesem Register nur ein bestimmtes Teilfeld ändern und die übrigen Teilfelder unverändert lassen möchten, können wir die Funktion `modify` verwenden. Diese Funktion nimmt eine Closure mit zwei Argumenten entgegen – eines zum Lesen und eines zum Schreiben. Üblicherweise bezeichnen

wir diese als `r` beziehungsweise `w`. Das Argument `r` dient dazu, den aktuellen Inhalt des Registers zu betrachten, während das Argument `w` genutzt werden kann, um den Registerinhalt zu ändern.

```
pwm.ctl.modify(|r, w| w.globalsync0().clear_bit());
```

Die Funktion `modify` veranschaulicht hier eindrucksvoll die Leistungsfähigkeit von Closures. In C müssten wir den Wert zunächst in eine temporäre Variable einlesen, die entsprechenden Bits ändern und den Wert anschließend wieder zurückschreiben. Dies birgt ein erhebliches Fehlerpotenzial:

```
uint32_t temp = pwm0.ctl.read();
temp |= PWM0_CTL_GLOBALSYNC0;
pwm0.ctl.write(temp);
uint32_t temp2 = pwm0.enable.read();
temp2 |= PWM0_ENABLE_PWM4EN;
pwm0.enable.write(temp); // Uh oh! Wrong variable!
```

3.1.4. Verwendung eines HAL-Crates

Ein HAL-Crate für einen Chip funktioniert typischerweise, indem es ein spezifisches Trait für die vom PAC bereitgestellten Rohstrukturen implementiert. Oft definiert dieses Trait eine Funktion namens `constrain()` für einzelne Peripherieeinheiten oder `split()` für Komponenten wie GPIO-Ports mit mehreren Pins. Diese Funktion übernimmt die zugrundeliegende Rohstruktur der Peripherie und gibt ein neues Objekt mit einer höherwertigen API zurück. Diese API kann zudem sicherstellen, dass beispielsweise die `new`-Funktion für die serielle Schnittstelle eine Referenz auf eine `Clock`-Struktur verlangt; eine solche Struktur lässt sich nur durch den Aufruf der Funktion erzeugen, welche die PLLs konfiguriert und sämtliche Taktfrequenzen einstellt. Auf diese Weise ist es zur Kompilierzeit ausgeschlossen, ein Objekt für die serielle Schnittstelle zu erzeugen, ohne zuvor die Taktraten konfiguriert zu haben, oder dass das Objekt die Baudrate fehlerhaft in Taktzyklen umrechnet. Manche Crates definieren sogar spezielle Traits für die Zustände, die ein GPIO-Pin einnehmen kann; der Nutzer muss den Pin dann in den korrekten Zustand versetzen (etwa durch Auswahl des passenden Modus für alternative Funktionen), bevor er ihn an die Peripherieeinheit übergibt. Und das alles ohne Laufzeitkosten!

Schauen wir uns ein Beispiel an:

```
#![no_std]
#![no_main]

use panic_halt as _; // panic handler

use cortex_m_rt::entry;
use tm4c123x_hal as hal;
use tm4c123x_hal::prelude::*;
use tm4c123x_hal::serial::{NewlineMode, Serial};
use tm4c123x_hal::sysctl;

#[entry]
fn main() -> ! {
    let p = hal::Peripherals::take().unwrap();
    let cp = hal::CorePeripherals::take().unwrap();

    // Kapseln Sie die SYSCTL-Struktur in einem Objekt mit einer API auf
    // hoeherer Ebene.
    let mut sc = p.SYSCTL.constrain();
    // Waehlen Sie unsere Oszillationseinstellungen aus.
    sc.clock_setup.oscillator = sysctl::Oscillator::Main(
        sysctl::CrystalFrequency::_16mhz,
        sysctl::SystemClock::UsePll(sysctl::PllOutputFrequency::_80_00mhz),
```

```
    );
    // Konfigurieren Sie die PLL mit diesen Einstellungen.
    let clocks = sc.clock_setup.freeze();

    // Kapseln Sie die `GPIO_PORTA`-Struktur in einem Objekt mit einer API der
    // hoeheren Ebene.
    // Beachten Sie, dass es `sc.power_control` nutzen muss, um die
    // Stromversorgung der GPIO-Peripherie automatisch zu aktivieren.
    let mut porta = p.GPIO_PORTA.split(&sc.power_control);

    // Aktivieren Sie den UART.
    let uart = Serial::uart0(
        p.UART0,
        // Der Sende-Pin
        porta
            .pa1
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // Der Empfangs-Pin
        porta
            .pa0
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // Kein RTS oder CTS erforderlich
        (),
        (),
        // Die Baudrate
        115200_u32.bps(),
        // Ausgabeverarbeitung
        NewlineMode::SwapLFtoCRLF,
        // Wir benoetigen die Taktfrequenzen, um die Baudraten-Teiler zu
        // berechnen.
        &clocks,
        // Wir benoetigen dies, um die UART-Peripherie zu aktivieren.
        &sc.power_control,
    );

    loop {
        writeln!(uart, "Hello, World!\r\n").unwrap();
    }
}
```

3.2. Semihosting

Semihosting ist ein Mechanismus, der es eingebetteten Systemen ermöglicht, Ein-/Ausgabeoperationen über den eigenen Rechner (Host) durchzuführen; er wird hauptsächlich dazu genutzt, Meldungen auf der Host-Konsole zu protokollieren. Da Semihosting lediglich eine Debug-Sitzung und sonst so gut wie nichts (keine zusätzlichen Kabel!) erfordert, ist es äußerst komfortabel in der Anwendung. Der Nachteil ist jedoch die sehr geringe Geschwindigkeit: Je nach verwendetem Hardware-Debugger (z. B. ST-Link) kann jeder Schreibvorgang mehrere Millisekunden in Anspruch nehmen.

Das `cortex-m-semihosting`-Crate stellt eine API für Semihosting-Operationen auf Cortex-M-Geräten bereit. Das folgende Programm ist die Semihosting-Version von „Hello, world!“:

```
#![no_main]
#![no_std]

use panic_halt as _;
```

```
use cortex_m_rt::entry;
use cortex_m_semihosting::hprintln;

#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    loop {}
}
```

Wenn Sie dieses Programm auf der Hardware ausführen, sehen Sie die Meldung „Hello, world!“ in den OpenOCD-Protokollen.

```
$ openocd
(..)
Hello, world!
(..)
```

Sie müssen zunächst Semihosting in OpenOCD über GDB aktivieren:

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

QEMU unterstützt Semihosting-Operationen, sodass das obige Programm auch mit `qemu-system-arm` funktioniert, ohne dass eine Debug-Sitzung gestartet werden muss. Beachten Sie, dass Sie QEMU die Option `-semihosting-config` übergeben müssen, um die Semihosting-Unterstützung zu aktivieren; diese Optionen sind bereits in der Datei `.cargo/config.toml` der Vorlage enthalten.

```
$ # Dieses Programm wird das Terminal blockieren.
$ cargo run
     Running `qemu-system-arm (..)
Hello, world!
```

Es gibt auch eine `exit`-Semihosting-Operation, mit der sich der QEMU-Prozess beenden lässt. Wichtig: Verwenden Sie `debug::exit` **nicht** auf echter Hardware; diese Funktion kann Ihre OpenOCD-Sitzung beschädigen, sodass Sie keine weiteren Programme mehr debuggen können, bis Sie die Sitzung neu starten.

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
    let roses = "blue";

    if roses == "red" {
        debug::exit(debug::EXIT_SUCCESS);
    } else {
        debug::exit(debug::EXIT_FAILURE);
    }

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)

$ echo $?
1
```

Ein letzter Tipp: Du kannst das Verhalten bei einem Panic auf `exit(EXIT_FAILURE)` einstellen. Dadurch lassen sich `no_std`-Tests schreiben, die erfolgreich durchlaufen und unter QEMU ausgeführt werden können.

Praktischerweise bietet der `panic-semihosting`-Crate ein „exit"-Feature an; ist dieses aktiviert, wird `exit(EXIT_FAILURE)` aufgerufen, nachdem die Panic-Meldung auf dem `stderr` des Hosts ausgegeben wurde.

```
#![no_main]
#![no_std]

use panic_semihosting as _; // features = ["exit"]

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
    let roses = "blue";

    assert_eq!(roses, "red");

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)
panicked at 'assertion failed: `(left == right)`
  left: `"blue"`,
 right: `"red"`', examples/hello.rs:15:5

$ echo $?
1
```

HINWEIS: Um diese Funktion für `panic-semihosting` zu aktivieren, bearbeiten Sie den Abschnitt „dependencies" in Ihrer `Cargo.toml`, in dem `panic-semihosting` wie folgt angegeben ist:

```
panic-semihosting = { version = "VERSION", features = ["exit"] }
```

wobei `VERSION` für die gewünschte Version steht. Weitere Informationen zu Abhängigkeiten und Funktionen finden Sie im Abschnitt "`specifying dependencies`" des Cargo-Buchs.

3.3. In Panik geraten

Das Auslösen einer Panik ist ein wesentlicher Bestandteil der Sprache Rust. Eingebaute Operationen wie der Zugriff per Index werden zur Laufzeit auf Speichersicherheit überprüft. Erfolgt ein Zugriff außerhalb der zulässigen Grenzen, führt dies zu einer Panik.

In der Standardbibliothek ist das Verhalten bei einer Panik definiert: Der Stack des betroffenen Threads wird abgewickelt (Stack Unwinding), es sei denn, der Benutzer hat sich für einen Programmabbruch im Falle einer Panik entschieden.

In Programmen ohne Standardbibliothek ist das Verhalten bei einer Panik hingegen nicht definiert. Ein Verhalten lässt sich durch die Deklaration einer `#[panic_handler]`-Funktion festlegen. Diese Funktion muss im Abhängigkeitsgraphen des Programms genau *einmal* vorkommen und folgende Signatur aufweisen: `fn(&PanicInfo) -> !`, wobei `PanicInfo` eine Struktur ist, die Informationen über den Ort des Panic enthält.

Da das Spektrum eingebetteter Systeme von anwenderorientierten bis hin zu sicherheitskritischen Anwendungen (bei denen ein Absturz ausgeschlossen sein muss) reicht, gibt es kein universelles Verhalten im Fehlerfall („Panic"); es existieren jedoch zahlreiche gängige Ansätze. Diese verbreiteten Verhaltensweisen wurden in Crates zusammengefasst, die die Funktion `#[panic_handler]` definieren. Zu den Beispielen gehören:

- `panic-abort`. Eine Panic bewirkt die Ausführung der Abbruchanweisung.
- `panic-halt`. Ein Panic bewirkt, dass das Programm oder der aktuelle Thread anhält, indem es bzw. er in eine Endlosschleife eintritt.
- `panic-itm`. Die Panik-Meldung wird mithilfe des ITM protokolliert – einer für ARM Cortex-M spezifischen Peripheriekomponente.
- `panic-semihosting`. Die Panik-Meldung wird mithilfe der Semihosting-Technik auf dem Host protokolliert.

Möglicherweise findest du noch mehr Crates, wenn du auf crates.io nach dem Schlüsselwort `panic-handler` suchst.

Ein Programm kann eines dieser Verhalten einfach dadurch auswählen, dass es das entsprechende Crate einbindet. Dass das Verhalten im Fehlerfall (Panic-Verhalten) im Quellcode einer Anwendung als einzelne Codezeile ausgedrückt wird, ist nicht nur als Dokumentation nützlich, sondern ermöglicht es auch, dieses Verhalten je nach Kompilierungsprofil anzupassen. Zum Beispiel:

```
#![no_main]
#![no_std]

// dev-Profil: einfacheres Debuggen von Panics; man kann einen Breakpoint bei
// `rust_begin_unwind` setzen.
#[cfg(debug_assertions)]
use panic_halt as _;

// Release-Profil: Minimierung der Binaergroesse der Anwendung
#[cfg(not(debug_assertions))]
use panic_abort as _;

// ..
```

In diesem Beispiel verlinkt der Crate beim Bauen mit dem Dev-Profil (`cargo build`) auf den `panic-halt`-Crate, beim Bauen mit dem Release-Profil (`cargo build --release`) hingegen auf den `panic-abort`-Crate.

> Die `use panic_abort as _;`-Variante der `use`-Anweisung wird verwendet, um sicherzustellen, dass der `panic_abort`-Panic-Handler in die fertige ausführbare Datei aufgenommen wird, während dem Compiler gleichzeitig signalisiert wird, dass wir nichts aus diesem Crate explizit verwenden werden. Ohne die Umbenennung mittels `as _` würde der Compiler eine Warnung wegen eines ungenutzten Imports ausgeben. Gelegentlich stößt man stattdessen auf `extern crate panic_abort`; dabei handelt es sich um einen älteren Stil, der vor der Rust-Edition 2018 üblich war und heute nur noch für sogenannte „sysroot"-Crates (die gemeinsam mit Rust selbst ausgeliefert werden) – wie etwa `proc_macro`, `alloc`, `std` und `test` – verwendet werden sollte.

3.3.1. Ein Beispiel

Hier ist ein Beispiel, das versucht, auf ein Array an einer Position zuzugreifen, die über dessen Länge hinausgeht. Der Vorgang führt zu einer Panic.

```
#![no_main]
#![no_std]

use panic_semihosting as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    let xs = [0, 1, 2];
    let i = xs.len();
    let _y = xs[i]; // Zugriff ausserhalb der zulaessigen Grenzen

    loop {}
}
```

Für dieses Beispiel wurde das Verhalten `panic-semihosting` gewählt, das die Panic-Meldung mittels Semihosting auf der Host-Konsole ausgibt.

```
$ cargo run
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
panicked at 'index out of bounds: the len is 3 but the index is 4', src/main.rs:12:13
```

Sie können versuchen, das Verhalten auf `panic-halt` zu ändern, und bestätigen, dass in diesem Fall keine Meldung ausgegeben wird.

3.4. Ausnahmen (Exceptions)

Ausnahmen und Interrupts sind ein Hardwaremechanismus, mit dem der Prozessor asynchrone Ereignisse und schwerwiegende Fehler (z. B. die Ausführung einer ungültigen Anweisung) behandelt. Ausnahmen implizieren Präemption und erfordern Ausnahmebehandlungsroutinen, die als Reaktion auf das auslösende Signal ausgeführt werden.

Die `cortex-m-rt`-Crate bietet das Attribut `exception` zur Deklaration von Ausnahmebehandlungsroutinen.

```
// Ausnahmebehandlungsroutine fuer die SysTick-Ausnahme (System-Timer)
#[exception]
fn SysTick() {
    // ..
}
```

Abgesehen vom `exception`-Attribut sehen Exception-Handler wie gewöhnliche Funktionen aus, doch es gibt noch einen weiteren Unterschied: `exception`-Handler können *nicht* per Software aufgerufen werden. Bezogen auf das vorherige Beispiel würde die Anweisung `SysTick();` zu einem Kompilierfehler führen.

Dieses Verhalten ist durchaus beabsichtigt und notwendig, um eine bestimmte Eigenschaft zu gewährleisten: `static mut`-Variablen, die *innerhalb* von `exception`-Handlern deklariert werden, sind *sicher* in der Verwendung.

```
#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
```

```
    // `COUNT` wurde in den Typ `&mut u32` umgewandelt und ist sicher in der
    // Verwendung.
    *COUNT += 1;
```

Wie Ihnen vielleicht bekannt ist, führt die Verwendung von `static mut`-Variablen in einer Funktion dazu, dass diese *nicht wiedereintrittsfähig* ist. Es stellt undefiniertes Verhalten dar, eine nicht wiedereintrittsfähige Funktion direkt oder indirekt aus mehr als einem Ausnahme- oder Interrupt-Handler oder aus `main` und einem oder mehreren Ausnahme- oder Interrupt-Handlern heraus aufzurufen.

Safe Rust darf niemals zu undefiniertem Verhalten führen; daher müssen nicht-wiedereintrittsfähige Funktionen als `unsafe` gekennzeichnet werden. Dennoch habe ich gerade erwähnt, dass Ausnahme-Handler (`exception handlers`) sicher `static mut`-Variablen verwenden können. Wie ist das möglich? Dies ist möglich, da Ausnahme-Handler *nicht* per Software aufgerufen werden können und somit kein Wiedereintritt (Reentrancy) stattfinden kann. Diese Handler werden von der Hardware selbst aufgerufen, bei der davon ausgegangen wird, dass sie physisch keine Nebenläufigkeit aufweist.

Im Kontext von Ausnahme-Handlern in eingebetteten Systemen stellt das Ausbleiben gleichzeitiger Aufrufe desselben Handlers folglich sicher, dass keine Probleme mit der Wiedereintrittsfähigkeit auftreten – selbst dann nicht, wenn der Handler veränderbare statische Variablen verwendet

In einem Multicore-System, in dem mehrere Prozessorkerne gleichzeitig Code ausführen, gewinnt die Problematik der Wiedereintrittsfähigkeit (Reentrancy) erneut an Bedeutung – selbst innerhalb von Exception-Handlern. Auch wenn jeder Kern über eigene Exception-Handler verfügen mag, kann es dennoch zu Situationen kommen, in denen mehrere Kerne versuchen, denselben Exception-Handler simultan auszuführen. Um dieser Herausforderung in einer Multicore-Umgebung zu begegnen, müssen innerhalb der Exception-Handler geeignete Synchronisationsmechanismen eingesetzt werden; so wird sichergestellt, dass der Zugriff auf gemeinsam genutzte Ressourcen zwischen den Kernen korrekt koordiniert wird. Dies geschieht typischerweise durch den Einsatz von Techniken wie Locks, Semaphoren oder atomaren Operationen, um Race Conditions zu vermeiden und die Datenintegrität zu wahren.

> Beachten Sie, dass das Attribut `exception` Definitionen statischer Variablen innerhalb der Funktion umwandelt, indem es sie in `unsafe`-Blöcke einschließt und uns neue, passende Variablen des Typs `&mut` mit demselben Namen zur Verfügung stellt. Auf diese Weise können wir die Referenz mittels `*` dereferenzieren, um auf die Werte der Variablen zuzugreifen, ohne sie selbst in einen `unsafe`-Block einschließen zu müssen.

3.4.1. Ein vollständiges Beispiel

Hier ist ein Beispiel, das den System-Timer verwendet, um etwa jede Sekunde eine `SysTick`-Exception auszulösen. Der `SysTick`-Exception-Handler protokolliert in der Variablen `COUNT`, wie oft er aufgerufen wurde, und gibt anschließend den Wert von `COUNT` mittels Semihosting auf der Host-Konsole aus.

> **HINWEIS**: Sie können dieses Beispiel auf jedem Cortex-M-Gerät ausführen; es lässt sich auch unter QEMU ausführen.

```
#![deny(unsafe_code)]
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;
```

```
use cortex_m::peripheral::syst::SystClkSource;
use cortex_m_rt::{entry, exception};
use cortex_m_semihosting::{
    debug,
    hio::{self, HostStream},
};

#[entry]
fn main() -> ! {
    let p = cortex_m::Peripherals::take().unwrap();
    let mut syst = p.SYST;

    // konfiguriert den System-Timer so, dass er jede Sekunde eine SysTick-
    // Exception ausloest
    syst.set_clock_source(SystClkSource::Core);
    // Dies ist für den LM3S6965 konfiguriert, der einen Standard-CPU-Takt von
    // 12 MHz hat
    syst.set_reload(12_000_000);
    syst.clear_current();
    syst.enable_counter();
    syst.enable_interrupt();

    loop {}
}

#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
    static mut STDOUT: Option<HostStream> = None;

    *COUNT += 1;

    // Verzoegerte Initialisierung
    if STDOUT.is_none() {
        *STDOUT = hio::hstdout().ok();
    }

    if let Some(hstdout) = STDOUT.as_mut() {
        write!(hstdout, "{}", *COUNT).ok();
    }

    // WICHTIG: Lassen Sie diesen `if`-Block weg, wenn Sie das Programm auf
    //          echter Hardware ausfuehren, da sich Ihr Debugger sonst in
    //          einem inkonsistenten Zustand befinden wird.
    if *COUNT == 9 {
        // Dies beendet den QEMU-Prozess.
        debug::exit(debug::EXIT_SUCCESS);
    }
}
```

```
tail -n5 Cargo.toml
```

```
[dependencies]
cortex-m = "0.5.7"
cortex-m-rt = "0.6.3"
panic-halt = "0.2.0"
cortex-m-semihosting = "0.3.1"
```

```
$ cargo run --release
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
123456789
```

Wenn Sie dies auf dem Discovery-Board ausführen, sehen Sie die Ausgabe in der OpenOCD-Konsole. Außerdem hält das Programm *nicht* an, wenn der Zählerstand 9 erreicht.

3.4.2. Der Standard-Exception-Handler

Das `exception`-Attribut bewirkt eigentlich, dass der Standard-Exception-Handler für eine bestimmte Exception *überschrieben* wird. Wenn Sie den Handler für eine bestimmte Exception nicht überschreiben, wird sie von der Funktion `DefaultHandler` behandelt, die standardmäßig Folgendes tut:

```
fn DefaultHandler() {
    loop {}
}
```

Diese Funktion wird vom `cortex-m-rt`-Crate bereitgestellt und ist mit `#[no_mangle]` markiert, sodass Sie einen Breakpoint auf „DefaultHandler“ setzen und *unbehandelte* Exceptions abfangen können.

Es ist möglich, diesen `DefaultHandler` mithilfe des `exception`-Attributs zu überschreiben:

```
#[exception]
fn DefaultHandler(irqn: i16) {
    // benutzerdefinierter Standard-Handler
}
```

Das Argument `irqn` gibt an, welche Exception gerade bearbeitet wird. Ein negativer Wert zeigt an, dass eine Cortex-M-Exception bearbeitet wird, während ein Wert von null oder ein positiver Wert darauf hinweist, dass eine gerätespezifische Exception – auch als Interrupt bezeichnet – bearbeitet wird.

3.4.3. Der “Schwere Fehler”-Handler

Die `HardFault`-Exception nimmt eine Sonderstellung ein. Sie wird ausgelöst, wenn das Programm in einen ungültigen Zustand gerät; daher darf ihr Handler nicht zurückkehren, da dies zu undefiniertem Verhalten führen könnte. Zudem führt die Runtime-Crate einige Vorarbeiten durch, bevor der benutzerdefinierte `HardFault`-Handler aufgerufen wird, um die Fehlersuche (Debugging) zu erleichtern.

Daraus ergibt sich, dass der `HardFault`-Handler folgende Signatur aufweisen muss: `fn(&ExceptionFrame) -> !`. Das Argument des Handlers ist ein Zeiger auf Register, die beim Auftreten der Exception auf den Stack gesichert wurden. Diese Register stellen eine Momentaufnahme des Prozessorzustands zum Zeitpunkt der Auslösung der Exception dar und sind für die Diagnose eines Hard Faults hilfreich.

Hier ist ein Beispiel für eine unzulässige Operation: der Lesezugriff auf eine nicht existierende Speicheradresse.

> **HINWEIS**: Dieses Programm wird unter QEMU nicht fehlschlagen (d. h. es stürzt nicht ab), da `qemu-system-arm -machine lm3s6965evb` keine Überprüfung von Speicherzugriffen durchführt und beim Lesen von ungültigem Speicher problemlos den Wert `0` zurückgibt.

```
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;
use core::ptr;
```

```
use cortex_m_rt::{entry, exception, ExceptionFrame};
use cortex_m_semihosting::hio;

#[entry]
fn main() -> ! {
    // liest einen nicht existierenden Speicherbereich
    unsafe {
        ptr::read_volatile(0x3FFF_0000 as *const u32);
    }

    loop {}
}

#[exception]
fn HardFault(ef: &ExceptionFrame) -> ! {
    if let Ok(mut hstdout) = hio::hstdout() {
        writeln!(hstdout, "{:#?}", ef).ok();
    }

    loop {}
}
```

Der `HardFault`-Handler gibt den Wert des `ExceptionFrame` aus. Wenn Sie dies ausführen, sehen Sie auf der OpenOCD-Konsole eine Ausgabe wie diese.

```
$ openocd
(..)
ExceptionFrame {
    r0: 0x3fff0000,
    r1: 0x00000003,
    r2: 0x080032e8,
    r3: 0x00000000,
    r12: 0x00000000,
    lr: 0x080016df,
    pc: 0x080016e2,
    xpsr: 0x61000000,
}
```

Der Wert `pc` ist der Wert des Programmzählers zum Zeitpunkt der Ausnahme und verweist auf die Anweisung, die die Ausnahme ausgelöst hat.

Wenn Sie sich den Disassembler des Programms ansehen:

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
(..)
ResetTrampoline:
 8000942:       movw    r0, #0xfffe
 8000946:       movt    r0, #0x3fff
 800094a:       ldr     r0, [r0]
 800094c:       b       #-0x4 <ResetTrampoline+0xa>
```

Den Wert des Programmzählers `0x0800094a` können Sie in der Disassemblierung nachschlagen.

Sie werden sehen, dass eine Ladeoperation (`ldr r0, [r0]`) die Ausnahme verursacht hat.

Das Feld `r0` von `ExceptionFrame` gibt an, dass der Wert des Registers `r0` zu diesem Zeitpunkt `0x3fff_fffe` war.

3.5. Interrupts

Interrupts unterscheiden sich in vielerlei Hinsicht von Exceptions; ihre Funktionsweise und Verwendung sind jedoch weitgehend ähnlich, und sie werden zudem von demselben Interrupt-Controller verwaltet. Während Exceptions durch die Cortex-M-Architektur definiert sind, handelt es sich bei Interrupts – sowohl hinsichtlich der Benennung als auch der Funktionalität – stets um herstellerspezifische (und oft sogar chip-spezifische) Implementierungen.

Interrupts bieten ein hohes Maß an Flexibilität, das bei einer fortgeschrittenen Nutzung berücksichtigt werden muss. Wir werden diese Anwendungsfälle in diesem Buch zwar nicht behandeln, dennoch ist es ratsam, Folgendes zu beachten:

- Interrupts verfügen über programmierbare Prioritäten, die die Ausführungsreihenfolge ihrer Behandlungsroutinen bestimmen.
- Interrupts können geschachtelt werden und einander unterbrechen (Preemption); das heißt, die Ausführung eines Interrupt-Handlers kann durch einen anderen Interrupt mit höherer Priorität unterbrochen werden.
- Im Allgemeinen muss die Ursache für die Auslösung des Interrupts beseitigt werden, um zu verhindern, dass der Interrupt-Handler endlos erneut aufgerufen wird.

Die allgemeinen Initialisierungsschritte zur Laufzeit sind immer gleich:

- Konfigurieren Sie die Peripheriekomponente(n) so, dass Interrupt-Anforderungen zu den gewünschten Zeitpunkten ausgelöst werden.
- Legen Sie die gewünschte Priorität des Interrupt-Handlers im Interrupt-Controller fest.
- Aktivieren Sie den Interrupt-Handler im Interrupt-Controller.

Ähnlich wie bei Exceptions stellt das `cortex-m-rt`-Crate ein `interrupt`-Attribut zur Deklaration von Interrupt-Handlern bereit. Dieses Attribut steht jedoch nur zur Verfügung, wenn das Device-Feature aktiviert ist. Es ist allerdings nicht für die direkte Verwendung vorgesehen; eine solche würde zu einem Kompilierfehler führen.

Stattdessen sollten Sie die vom Device-Crate (das üblicherweise mit `svd2rust` generiert wird) bereitgestellte, re-exportierte Version des `interrupt`-Attributs verwenden. Dies stellt sicher, dass der Compiler überprüfen kann, ob der Interrupt auf dem Zielgerät tatsächlich existiert. Die Liste der verfügbaren Interrupts – sowie deren Position in der Interrupt-Vektortabelle – wird typischerweise von `svd2rust` automatisch aus einer SVD-Datei generiert.

```
rust
use lm3s6965::interrupt; // Aus dem Device-Crate zurueck-exportiertes Attribut

// Interrupt-Handler fuer den Timer2-Interrupt
#[interrupt]
fn TIMER2A() {
    // ..
    // Eindeutiger Grund fuer die generierte Interrupt-Anforderung
}
```

Interrupt-Handler ähneln – abgesehen vom Fehlen von Argumenten – gewöhnlichen Funktionen, ganz wie Exception-Handler. Aufgrund spezieller Aufrufkonventionen können sie jedoch nicht direkt von anderen Teilen der Firmware aufgerufen werden. Es ist allerdings möglich, Interrupt-Anforderungen per Software zu erzeugen, um einen Sprung zum Interrupt-Handler auszulösen.

Ähnlich wie bei Exception-Handlern ist es auch hier möglich, `static mut`-Variablen innerhalb der Interrupt-Handler zu deklarieren, um den Zustand auf *sichere* Weise zu speichern.

```
#[interrupt]
fn TIMER2A() {
    static mut COUNT: u32 = 0;

    // `COUNT` hat den Typ `&mut u32` und ist sicher zu verwenden.
    *COUNT += 1;
}
```

Für eine detailliertere Beschreibung der hier veranschaulichten Mechanismen konsultieren Sie bitte den Abschnitt zu Ausnahmen.

4. Peripheriegeräte

4.1. Was sind Peripheriegeräte?

Die meisten Mikrocontroller verfügen über mehr als nur eine CPU, einen RAM oder einen Flash-Speicher – sie enthalten Siliziumabschnitte, die für die Interaktion mit Systemen außerhalb des Mikrocontrollers sowie für die direkte und indirekte Interaktion mit ihrer Umgebung in der Welt über Sensoren, Motorsteuerungen oder menschliche Schnittstellen wie ein Display oder eine Tastatur verwendet werden. Diese Komponenten werden zusammenfassend als Peripheriegeräte bezeichnet.

Diese Peripheriegeräte sind nützlich, weil sie es einem Entwickler ermöglichen, die Verarbeitung auf sie auszulagern, sodass er sich nicht um alles in der Software kümmern muss. Ähnlich wie ein Desktop-Entwickler die Grafikverarbeitung auf eine Grafikkarte verlagern würde, können Embedded-Entwickler einige Aufgaben auf Peripheriegeräte verlagern, sodass die CPU ihre Zeit mit anderen wichtigen Dingen verbringen oder gar nichts tun kann, um Strom zu sparen.

Wenn Sie sich die Hauptplatine eines altmodischen Heimcomputers aus den 1970er oder 1980er Jahren ansehen (und tatsächlich sind die Desktop-PCs von gestern nicht so weit von den eingebetteten Systemen von heute entfernt), würden Sie Folgendes erwarten:

- Einen Prozessor
- Einen RAM-Chip
- Einen ROM-Chip
- Einen Ein-/Ausgabe-Controller

Der RAM-Chip, der ROM-Chip und der I/O-Controller (das Peripheriegerät in diesem System) wären über eine Reihe paralleler Leiterbahnen, die als „Bus“ bekannt sind, mit dem Prozessor verbunden. Dieser Bus trägt Adressinformationen, die auswählen, mit welchem Gerät auf dem Bus der Prozessor kommunizieren möchte, und einen Datenbus, der die eigentlichen Daten überträgt. In unseren eingebetteten Mikrocontrollern gelten die gleichen Prinzipien – nur dass alles auf einem einzigen Stück Silizium untergebracht ist.

Im Gegensatz zu Grafikkarten, die normalerweise über eine Software-API wie Vulkan, Metal oder OpenGL verfügen, werden Peripheriegeräte unserem Mikrocontroller über eine Hardwareschnittstelle zugänglich gemacht, die einem Teil des Speichers zugeordnet ist.

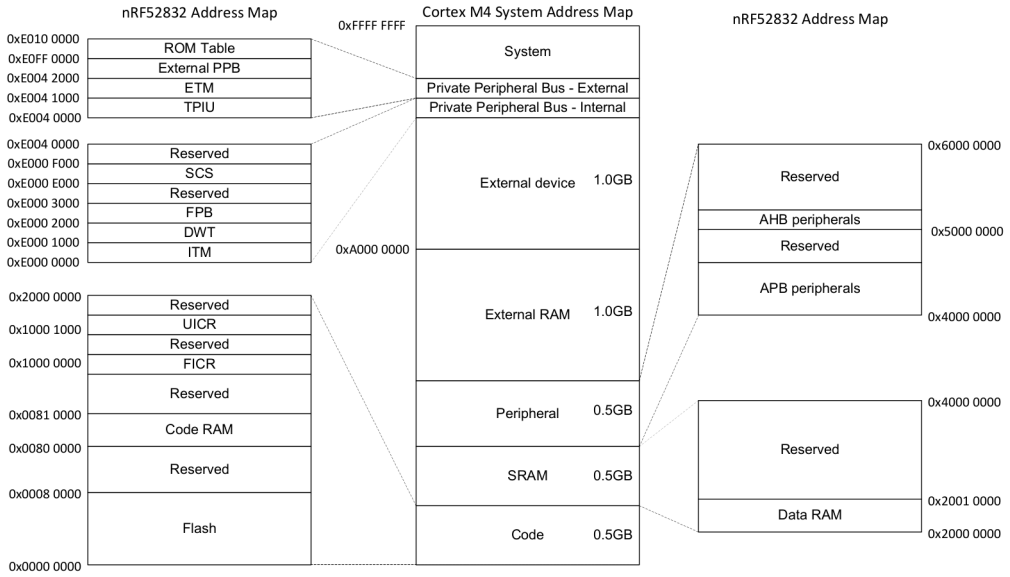
4.2. Linearer und realer Speicherraum

Bei einem Mikrocontroller kann auch das Schreiben von Daten an eine beliebige andere Adresse – etwa `0x4000_0000` oder `0x0000_0000` – ein völlig zulässiger Vorgang sein.

Auf einem Desktop-System wird der Speicherzugriff streng von der MMU (Memory Management Unit, Speicherverwaltungseinheit) kontrolliert. Diese Komponente hat zwei Hauptaufgaben: die Durchsetzung von Zugriffsberechtigungen für Speicherbereiche (um zu verhindern, dass ein Prozess den Speicher eines anderen Prozesses liest oder verändert) sowie die Abbildung von Segmenten des physischen Speichers auf virtuelle Speicherbereiche, die von der Software genutzt werden. Mikrocontroller verfügen in der Regel nicht über eine MMU; stattdessen verwendet die Software dort ausschließlich reale physische Adressen.

Obwohl 32-Bit-Mikrocontroller über einen echten, linearen Adressraum von `0x0000_0000` bis `0xFFFF_FFFF` verfügen, nutzen sie für den eigentlichen Speicher meist nur einige hundert Kilobyte dieses Bereichs. Dadurch bleibt ein beträchtlicher Teil des Adressraums ungenutzt. In früheren Kapiteln war die Rede davon, dass sich der RAM an der Adresse `0x2000_0000` befindet. Wäre unser RAM 64 KiB groß (d. h. mit einer maximalen Adresse von 0xFFFF), so entsprächen die Adressen `0x2000_0000` bis `0x2000_FFFF` diesem Speicherbereich. Wenn wir in eine Variable schreiben, die an

der Adresse `0x2000_1234` liegt, geschieht intern Folgendes: Eine Logikeinheit erkennt den oberen Teil der Adresse (in diesem Beispiel `0x2000`) und aktiviert den RAM, sodass dieser auf den unteren Teil der Adresse (in diesem Fall `0x1234`) reagieren kann. Bei einem Cortex-M-System ist zudem das Flash-ROM eingeblendet – etwa im Bereich von `0x0000_0000` bis `0x0007_FFFF` (bei einem 512-KiB-Flash-ROM). Anstatt den gesamten verbleibenden Raum zwischen diesen beiden Bereichen ungenutzt zu lassen, haben die Entwickler der Mikrocontroller die Schnittstellen für Peripheriegeräte bestimmten Speicheradressen zugeordnet. Das Ergebnis sieht dann ungefähr so aus:



Nordic nRF52832 Datasheet (pdf)

4.3. Im Speicher abgebildete Peripheriegeräte

Die Interaktion mit diesen Peripheriegeräten ist auf den ersten Blick einfach: Schreiben Sie die richtigen Daten an die richtige Adresse. Beispielsweise könnte das Senden eines 32-Bit-Worts über eine serielle Schnittstelle genauso direkt sein wie das Schreiben dieses 32-Bit-Worts an eine bestimmte Speicheradresse. Das Serial-Port-Peripheriegerät würde dann übernehmen und die Daten automatisch versenden.

Die Konfiguration dieser Peripheriegeräte funktioniert ähnlich. Anstatt eine Funktion zum Konfigurieren eines Peripheriegeräts aufzurufen, wird ein Teil des Speichers verfügbar gemacht, der als Hardware-API dient. Schreiben Sie „0x8000_0000″ in ein SPI-Frequenzkonfigurationsregister, und der SPI-Port sendet Daten mit 8 Megabit pro Sekunde. Schreiben Sie „0x0200_0000″ an dieselbe Adresse und der SPI-Port sendet Daten mit 125 Kilobit pro Sekunde. Diese Konfigurationsregister sehen in etwa so aus:

31.6.8 FREQUENCY

Address offset: 0x524

SPI frequency

Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id				A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset 0x04000000				**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Id	RW	Field	Value Id	Value	Description
A	RW	FREQUENCY			SPI master data rate
			K125	0x02000000	125 kbps
			K250	0x04000000	250 kbps
			K500	0x08000000	500 kbps
			M1	0x10000000	1 Mbps
			M2	0x20000000	2 Mbps
			M4	0x40000000	4 Mbps
			M8	0x80000000	8 Mbps

Nordic nRF52832 Datasheet (pdf)

Über diese Schnittstelle erfolgen die Interaktionen mit der Hardware, unabhängig davon, welche Sprache verwendet wird – sei es Assembler, C oder Rust.

4.4. Ein erster Versuch in Rust

4.4.1. Die Register

Betrachten wir die „SysTick“-Peripherie – einen einfachen Timer, der in jedem Cortex-M-Prozessorkern enthalten ist. Normalerweise würde man hierfür im Datenblatt des Chip-Herstellers oder im *Technical Reference Manual* nachschlagen; da dieses Beispiel jedoch für alle ARM-Cortex-M-Kerne gilt, werfen wir einen Blick in das ARM-Referenzhandbuch. Wir sehen, dass es vier Register gibt:

Offset	Name	Beschreibung	Breite
0x00	SYST_CSR	Steuerung und Status Register	32 bits
0x04	SYST_RVR	Wert-Register neu laden	32 bits
0x08	SYST_CVR	Register für den aktuellen Wert	32 bits
0x0C	SYST_CALIB	Kalibrierwert-Register	32 bits

4.4.2. Der C-Ansatz

In Rust können wir eine Sammlung von Registern auf genau dieselbe Weise darstellen wie in C – mit einem `struct`.

```
#[repr(C)]
struct SysTick {
    pub csr: u32,
    pub rvr: u32,
    pub cvr: u32,
    pub calib: u32,
}
```

Der Qualifizierer `#[repr(C)]` weist den Rust-Compiler an, diese Struktur wie ein C-Compiler anzulegen. Das ist sehr wichtig, da Rust die Neuordnung von Strukturfeldern zulässt, C jedoch nicht. Sie können sich vorstellen, wie viel Debugging wir durchführen müssten, wenn diese Felder vom Compiler stillschweigend neu angeordnet würden! Mit diesem Qualifizierer verfügen wir über unsere vier 32-Bit-Felder, die der obigen Tabelle entsprechen. Aber diese „Struktur“ allein nützt natürlich nichts – wir brauchen eine Variable.

```
let systick = 0xE000_E010 as *mut SysTick;
let time = unsafe { (*systick).cvr };
```

4.4.3. Volatile-Zugriffe

Nun gibt es bei dem oben genannten Ansatz einige Probleme.

1. Wir müssen `unsafe` verwenden, wann immer wir auf unsere Peripherie zugreifen wollen.
2. Wir haben keine Möglichkeit festzulegen, welche Register nur lesbar oder sowohl les- als auch schreibbar sind.
3. Jeder Code-Abschnitt an beliebiger Stelle Ihres Programms könnte über diese Struktur auf die Hardware zugreifen.
4. Vor allem aber funktioniert es eigentlich nicht …

Das Problem ist nun, dass Compiler clever sind. Wenn man zwei Schreibzugriffe nacheinander auf denselben Speicherbereich durchführt, kann der Compiler dies erkennen und den ersten Schreibvorgang einfach komplett überspringen. In C können wir Variablen als `volatile` kennzeichnen, um sicherzustellen, dass jeder Lese- oder Schreibvorgang wie vorgesehen ausgeführt wird. In Rust hingegen kennzeichnen wir die *Zugriffe* als volatile, nicht die Variable.

```
let systick = unsafe { &mut *(0xE000_E010 as *mut SysTick) };
let time = unsafe { core::ptr::read_volatile(&mut systick.cvr) };
```

Wir haben also eines unserer vier Probleme gelöst, aber jetzt haben wir noch mehr `unsafe`-Code! Glücklicherweise gibt es ein Crate eines Drittanbieters, das hierbei helfen kann – `volatile_register`.

```
use volatile_register::{RW, RO};

#[repr(C)]
struct SysTick {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

fn get_systick() -> &'static mut SysTick {
    unsafe { &mut *(0xE000_E010 as *mut SysTick) }
}

fn get_time() -> u32 {
    let systick = get_systick();
    systick.cvr.read()
}
```

Die volatilen Zugriffe erfolgen nun automatisch über die Methoden `read` und `write`. Schreibzugriffe sind zwar weiterhin als `unsafe` eingestuft, doch fairerweise muss man sagen, dass Hardware im Grunde aus veränderlichem Zustand besteht und der Compiler nicht wissen kann, ob diese Schreibvorgänge tatsächlich sicher sind; daher ist dies eine sinnvolle Standardeinstellung.

4.4.4. Der Rusty Wrapper

Wir müssen diese `struct` in eine API der höheren Ebene verpacken, deren Aufruf für unsere Nutzer sicher ist. Als Treiberentwickler überprüfen wir manuell die Korrektheit des `unsafe`-Codes und stellen unseren Nutzern dann eine sichere API zur Verfügung, sodass sie sich darum nicht kümmern müssen (vorausgesetzt, sie vertrauen darauf, dass wir alles richtig gemacht haben!).

Ein Beispiel hierfür wäre:

```
use volatile_register::{RW, RO};

pub struct SystemTimer {
```

```
    p: &'static mut RegisterBlock
}

#[repr(C)]
struct RegisterBlock {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

impl SystemTimer {
    pub fn new() -> SystemTimer {
        SystemTimer {
            p: unsafe { &mut *(0xE000_E010 as *mut RegisterBlock) }
        }
    }

    pub fn get_time(&self) -> u32 {
        self.p.cvr.read()
    }

    pub fn set_reload(&mut self, reload_value: u32) {
        unsafe { self.p.rvr.write(reload_value) }
    }
}

pub fn example_usage() -> String {
    let mut st = SystemTimer::new();
    st.set_reload(0x00FF_FFFF);
    format!("Time is now 0x{:08x}", st.get_time())
}
```

Das Problem bei diesem Ansatz ist nun, dass der folgende Code für den Compiler völlig zulässig ist:

```
fn thread1() {
    let mut st = SystemTimer::new();
    st.set_reload(2000);
}

fn thread2() {
    let mut st = SystemTimer::new();
    st.set_reload(1000);
}
```

Das `&mut self`-Argument der Funktion `set_reload` stellt zwar sicher, dass keine weiteren Referenzen auf *genau diese* `SystemTimer`-Struktur existieren, verhindert jedoch nicht, dass der Benutzer einen zweiten `SystemTimer` erstellt, der auf dieselbe Peripherieeinheit verweist. Auf diese Weise geschriebener Code funktioniert zwar, sofern der Autor sorgfältig genug ist, all diese „doppelten“ Treiberinstanzen zu erkennen; sobald sich der Code jedoch über mehrere Module, Treiber, Entwickler und Zeiträume hinweg erstreckt, schleichen sich derartige Fehler immer leichter ein.

4.5. Veränderbarer globaler Zustand

Leider ist Hardware im Grunde nichts anderes als veränderbarer globaler Zustand, was auf einen Rust-Entwickler sehr beängstigend wirken kann. Hardware existiert unabhängig von den Strukturen des Codes, den wir schreiben, und kann jederzeit durch die reale Welt verändert werden.

4.5.1. Wie sollten unsere Regeln sein?

Wie können wir zuverlässig mit diesen Peripheriegeräten interagieren?

1. Verwenden Sie stets `volatile`-Methoden für Lese- oder Schreibzugriffe auf Peripheriespeicher, da sich dieser jederzeit ändern kann.
2. Auf Softwareebene sollten wir in der Lage sein, eine beliebige Anzahl von Lesezugriffen auf diese Peripheriegeräte bereitzustellen.
3. Wenn eine Software Lese- und Schreibzugriff auf ein Peripheriegerät haben soll, sollte sie die einzige Referenz auf dieses Peripheriegerät besitzen.

4.5.2. Der Borrow-Checker

Die letzten beiden dieser Regeln klingen verdächtig ähnlich wie das, was der Borrow Checker bereits tut!

Stellen Sie sich vor, wir könnten die Besitzrechte an diesen Peripheriegeräten weitergeben oder unveränderliche bzw. veränderliche Referenzen darauf anbieten?

Nun, das ist möglich, aber für den Borrow Checker benötigen wir genau eine Instanz jedes Peripheriegeräts, damit Rust dies korrekt verarbeiten kann. Glücklicherweise existiert in der Hardware nur eine Instanz jedes Peripheriegeräts, aber wie können wir dies in der Struktur unseres Codes sichtbar machen?

4.6. Singletons

> In der Softwaretechnik ist das Singleton-Muster ein Entwurfsmuster, das die Instanziierung einer Klasse auf ein einziges Objekt beschränkt.
>
> *Wikipedia: Singleton-Muster*

4.6.1. Aber warum können wir nicht einfach globale Variable(n) verwenden?

Wir könnten alles als „public static“ definieren, etwa so:

```
static mut THE_SERIAL_PORT: SerialPort = SerialPort;

fn main() {
    let _ = unsafe {
        THE_SERIAL_PORT.read_speed();
    };
}
```

Dies bringt jedoch einige Probleme mit sich. Es handelt sich um eine veränderbare globale Variable, und der Zugriff auf solche Variablen ist in Rust stets „unsafe“. Zudem sind diese Variablen im gesamten Programm sichtbar, was bedeutet, dass der Borrow Checker nicht dabei helfen kann, die Referenzen und die Ownership dieser Variablen zu verfolgen.

4.6.2. Wie machen wir das in Rust?

Anstatt unser Peripheriegerät einfach als globale Variable zu definieren, könnten wir uns stattdessen dazu entschließen, eine Struktur – in diesem Fall mit dem Namen `PERIPHERALS` – zu erstellen, die für jedes unserer Peripheriegeräte ein `Option<T>` enthält.

```
struct Peripherals {
    serial: Option<SerialPort>,
}
impl Peripherals {
    fn take_serial(&mut self) -> SerialPort {
```

```
        let p = replace(&mut self.serial, None);
        p.unwrap()
    }
}
static mut PERIPHERALS: Peripherals = Peripherals {
    serial: Some(SerialPort),
};
```

Diese Struktur ermöglicht es uns, eine einzelne Instanz unserer Peripheriekomponente zu erhalten. Wenn wir versuchen, `take_serial()` mehr als einmal aufzurufen, wird unser Code eine Panic auslösen!

```
fn main() {
    let serial_1 = unsafe { PERIPHERALS.take_serial() };
    // Dies loest eine panic aus!
    // let serial_2 = unsafe { PERIPHERALS.take_serial() };
}
```

Obwohl die Interaktion mit dieser Struktur als `unsafe` gilt, benötigen wir – sobald wir über die darin enthaltene `SerialPort`-Instanz verfügen – weder `unsafe`-Code noch die `PERIPHERALS`-Struktur selbst.

Dies bringt einen geringen Laufzeit-Overhead mit sich, da wir die `SerialPort`-Struktur in eine `Option` einbetten und einmalig `take_serial()` aufrufen müssen; dieser geringe anfängliche Aufwand ermöglicht es uns jedoch, den Borrow-Checker im weiteren Verlauf des Programms voll zu nutzen.

4.6.3. Vorhandene Bibliotheksunterstützung

Obwohl wir oben unsere eigene `Peripherals`-Struktur erstellt haben, ist dies für Ihren Code nicht erforderlich; das `cortex_m`-Crate enthält ein Makro namens `singleton!()`, das diese Aufgabe für Sie übernimmt.

```
use cortex_m::singleton;

fn main() {
    // In Ordnung, wenn `main` nur einmal ausgefuehrt wird.
    let x: &'static mut bool =
        singleton!(: bool = false).unwrap();
}
```

cortex_m Dokumente

Wenn Sie zudem `cortex-m-rtic` verwenden, wird der gesamte Prozess der Definition und Bereitstellung dieser Peripheriekomponenten für Sie abstrahiert; stattdessen erhalten Sie eine `Peripherals`-Struktur, die eine Version aller von Ihnen definierten Elemente enthält, die nicht als `Option<T>` vorliegen.

```
// cortex-m-rtic v0.5.x
#[rtic::app(device = lm3s6965, peripherals = true)]
const APP: () = {
    #[init]
    fn init(cx: init::Context) {
        static mut X: u32 = 0;

        // Cortex-M-Peripherie
        let core: cortex_m::Peripherals = cx.core;

        // Geraetespezifische Peripheriegeraete
        let device: lm3s6965::Peripherals = cx.device;
```

```
    }
}
```

4.6.4. Aber warum?

Aber wie bewirken diese Singletons einen spürbaren Unterschied in der Funktionsweise unseres Rust-Codes?

```
impl SerialPort {
    const SER_PORT_SPEED_REG: *mut u32 = 0x4000_1000 as _;

    fn read_speed(
        &self // <------ Das ist wirklich, wirklich wichtig.
    ) -> u32 {
        unsafe {
            ptr::read_volatile(Self::SER_PORT_SPEED_REG)
        }
    }
}
```

Hier spielen zwei wichtige Faktoren eine Rolle:

- Da wir ein Singleton verwenden, gibt es nur eine Möglichkeit bzw.

einen Ort, um eine `SerialPort`-Struktur zu erhalten.

- Um die Methode `read_speed()` aufzurufen, benötigen wir den Besitz

einer `SerialPort`-Struktur oder eine Referenz darauf.

Zusammengenommen bedeuten diese beiden Faktoren, dass ein Zugriff auf die Hardware nur möglich ist, wenn wir die Anforderungen des Borrow-Checkers ordnungsgemäß erfüllt haben – das heißt, dass zu keinem Zeitpunkt mehrere veränderbare Referenzen auf dieselbe Hardware existieren!

```
fn main() {
    // Verweis auf `self` fehlt! Das wird nicht funktionieren.
    // SerialPort::read_speed();

    let serial_1 = unsafe { PERIPHERALS.take_serial() };

    // Sie koennen nur lesen, worauf Sie Zugriff haben
    let _ = serial_1.read_speed();
}
```

4.6.5. Behandle deine Hardware wie Daten

Da es zudem sowohl veränderbare als auch unveränderbare Referenzen gibt, lässt sich erkennen, ob eine Funktion oder Methode potenziell den Zustand der Hardware ändern könnte. Zum Beispiel:

Dies darf Hardware-Einstellungen ändern:

```
fn setup_spi_port(
    spi: &mut SpiPort,
    cs_pin: &mut GpioPin
) -> Result<()> {
    // ...
}
```

Das nicht:

```
fn read_button(gpio: &GpioPin) -> bool {
    // ...
}
```

Dies ermöglicht es uns, bereits zur **Kompilierzeit** – und nicht erst zur Laufzeit – festzulegen, ob Code Änderungen an der Hardware vornehmen darf oder nicht. Zu beachten ist, dass dies im Allgemeinen nur innerhalb einer einzelnen Anwendung funktioniert; da unsere Software für Bare-Metal-Systeme jedoch als eine einzige Anwendung kompiliert wird, stellt dies üblicherweise keine Einschränkung dar.

5. Statische Garantien

Das Typsystem von Rust verhindert Data Races zur Kompilierzeit (siehe die Traits `Send` und `Sync`). Zudem lässt sich das Typsystem nutzen, um weitere Eigenschaften bereits zur Kompilierzeit zu überprüfen, wodurch in einigen Fällen Laufzeitprüfungen überflüssig werden.

Bei Embedded-Programmen lassen sich diese *statischen Prüfungen* beispielsweise nutzen, um sicherzustellen, dass die Konfiguration von E/A-Schnittstellen korrekt erfolgt. So lässt sich etwa eine API entwerfen, bei der die Initialisierung einer seriellen Schnittstelle nur möglich ist, wenn zuvor die für diese Schnittstelle vorgesehenen Pins konfiguriert wurden.

Zudem lässt sich statisch überprüfen, ob Operationen – wie etwa das Setzen eines Pins auf den Pegel „Low“ – nur an korrekt konfigurierten Peripheriekomponenten ausgeführt werden. Der Versuch, den Ausgangszustand eines Pins zu ändern, der als „Floating Input” (Eingang mit undefiniertem Pegel) konfiguriert ist, würde beispielsweise zu einem Kompilierfehler führen.

Wie bereits im vorherigen Kapitel erläutert, lässt sich auch das Konzept des „Ownership“ (Besitzverhältnis) auf Peripheriekomponenten anwenden, um sicherzustellen, dass nur bestimmte Programmteile diese modifizieren können. Diese *Zugriffskontrolle* erleichtert das Verständnis und die Analyse der Software im Vergleich zu dem alternativen Ansatz, Peripheriekomponenten als globalen, veränderbaren Zustand zu behandeln.

5.1. Typgestützte Programmierung

Das Konzept der typestates beschreibt die Kodierung von Informationen über den aktuellen Zustand eines Objekts direkt in dessen Typ. Auch wenn dies zunächst etwas abstrakt oder kompliziert klingen mag: Wenn Sie in Rust bereits das Builder-Muster verwendet haben, sind Sie schon mit der Typestate-Programmierung in Berührung gekommen!

```
pub mod foo_module {
    #[derive(Debug)]
    pub struct Foo {
        inner: u32,
    }

    pub struct FooBuilder {
        a: u32,
        b: u32,
    }

    impl FooBuilder {
        pub fn new(starter: u32) -> Self {
            Self {
                a: starter,
                b: starter,
            }
        }

        pub fn double_a(self) -> Self {
            Self {
                a: self.a * 2,
                b: self.b,
            }
        }

        pub fn into_foo(self) -> Foo {
            Foo {
```

```
                inner: self.a + self.b,
            }
        }
    }
}

fn main() {
    let x = foo_module::FooBuilder::new(10)
        .double_a()
        .into_foo();

    println!("{:#?}", x);
}
```

In diesem Beispiel gibt es keine direkte Möglichkeit, ein `Foo`-Objekt zu erstellen. Wir müssen einen `FooBuilder` erstellen und ihn korrekt initialisieren, bevor wir das gewünschte `Foo`-Objekt erhalten können.

Dieses Minimalbeispiel kodiert zwei Zustände:
- `FooBuilder`, das einen „unkonfigurierten“ Zustand oder einen Zustand der „laufenden Konfiguration” repräsentiert
- `Foo`, das einen „konfigurierten“ oder „einsatzbereiten” Zustand repräsentiert.

5.1.1. Starke Typen

Da Rust über ein starkes Typsystem verfügt, gibt es keine einfache Möglichkeit, auf magische Weise eine Instanz von `Foo` zu erzeugen oder einen `FooBuilder` in ein `Foo` umzuwandeln, ohne die Methode `into_foo()` aufzurufen. Zudem verbraucht der Aufruf der Methode `into_foo()` die ursprüngliche `FooBuilder`-Struktur; das bedeutet, dass sie nicht ohne die Erstellung einer neuen Instanz wiederverwendet werden kann.

Dies ermöglicht es uns, die Zustände unseres Systems als Typen darzustellen und die für Zustandsübergänge erforderlichen Aktionen in jene Methoden zu integrieren, die einen Typ gegen einen anderen austauschen. Indem wir einen `FooBuilder` erstellen und diesen gegen ein `Foo`-Objekt austauschen, haben wir die Schritte einer einfachen Zustandsmaschine durchlaufen.

5.2. Peripheriegeräte als Zustandsautomaten

Die Peripherie eines Mikrocontrollers lässt sich als eine Menge von Zustandsautomaten auffassen. So ließe sich beispielsweise die Konfiguration eines vereinfachten GPIO-Pins als der folgende Zustandsbaum darstellen:
- Deaktiviert
- Aktiviert
  - Als Ausgang konfiguriert
    - Ausgang: High
    - Ausgang: Low
  - Als Eingang konfiguriert
    - Eingang: Hoher Widerstand
    - Eingang: Auf Low-Pegel gezogen
    - Eingang: Auf High-Pegel gezogen

Wenn die Peripherieeinheit im Modus `Deaktiviert` startet, müssen wir die folgenden Schritte ausführen, um in den Modus `Eingang: Hoher Widerstand` zu wechseln:
1. Deaktiviert
2. Aktiviert

3. Als Eingang konfiguriert
4. Eingang: Hoher Widerstand

Wenn wir von „Eingang: Hoher Widerstand“ zu „Eingang: Auf Low-Pegel gezogen” wechseln möchten, müssen wir die folgenden Schritte ausführen:
1. Eingang: Hoher Widerstand 2. Eingang: Auf Low-Pegel gezogen

Wenn wir einen GPIO-Pin von der Konfiguration „Eingang: Auf Low-Pegel gezogen“ auf „Ausgang: High” umstellen wollen, müssen wir ebenfalls die folgenden Schritte ausführen:
1. Eingang: Auf Low-Pegel gezogen
2. Als Eingang konfiguriert
3. Als Ausgang konfiguriert
4. Ausgang: High

5.2.1. Hardware-Darstellung

Typischerweise werden die oben aufgeführten Zustände eingestellt, indem Werte in bestimmte Register geschrieben werden, die einer GPIO-Peripherieeinheit zugeordnet sind. Definieren wir zur Veranschaulichung ein fiktives GPIO-Konfigurationsregister:

Name	Bit-Nummer	Wert	Bedeutung	Hinweise
enable	0	0	disabled	Deaktiviert den GPIO
		1	enabled	Aktiviert den GPIO
direction	1	0	input	Legt die Richtung auf „Eingang“ fest.
		1	output	Legt die Richtung auf „Ausgang“ fest.
input_mode	2..3	00	hi-z	Setzt den Eingang auf hochohmig.
		01	pull-low	Der Eingangspin wird auf Low-Pegel gezogen.
		10	pull-high	Der Eingangspin wird auf High-Pegel gezogen
		11	n/a	Ungültiger Zustand. Nicht setzen.
output_mode	4	0	set-low	Der Ausgangspin wird auf Low-Pegel gesteuert.
		1	set-high	Der Ausgangspin wird auf High-Pegel gesteuert.
input_status	5	x	in-val	0, wenn der Eingang < 1,5 V ist; 1, wenn der Eingang ≥ 1,5 V ist.

Wir *könnten* die folgende Struktur in Rust bereitstellen, um diesen GPIO zu steuern:

```
/// GPIO interface
struct GpioConfig {
    /// Von svd2rust generierte GPIO-Konfigurationsstruktur
    periph: GPIO_CONFIG,
}

impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) {
        self.periph.modify(|_r, w| {
            w.direction().set_bit(is_output)
```

```
        });
    }

    pub fn set_input_mode(&mut self, variant: InputMode) {
        self.periph.modify(|_r, w| {
            w.input_mode().variant(variant)
        });
    }

    pub fn set_output_mode(&mut self, is_high: bool) {
        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });
    }

    pub fn get_input_status(&self) -> bool {
        self.periph.read().input_status().bit_is_set()
    }
}
```

Dies würde uns jedoch erlauben, bestimmte Register zu verändern, was keinen Sinn ergibt. Was passiert beispielsweise, wenn wir das Feld `output_mode` setzen, während unser GPIO als Eingang konfiguriert ist?

Im Allgemeinen würde die Verwendung dieser Struktur es uns ermöglichen, Zustände zu erreichen, die nicht durch unsere obige Zustandsmaschine definiert sind: z. B. einen Ausgang, der auf LOW gesetzt wird, oder einen Eingang, der auf HIGH gesetzt wird. Bei mancher Hardware mag dies keine Rolle spielen. Bei anderer Hardware könnte es jedoch zu unerwartetem oder undefiniertem Verhalten führen!

Obwohl diese Schnittstelle bequem zu implementieren ist, erzwingt sie nicht die in unserer Hardware-Implementierung festgelegten Designvorgaben.

5.3. Design-Verträge

Im letzten Kapitel haben wir eine Schnittstelle geschrieben, die *keine* Design-Verträge (Design Contracts) durchsetzte. Werfen wir noch einmal einen Blick auf unser fiktives GPIO-Konfigurationsregister:

Name	Bit-Nummer	Wert	Bedeutung	Hinweise
enable	0	0	disabled	Deaktiviert den GPIO
		1	enabled	Aktiviert den GPIO
direction	1	0	input	Legt die Richtung auf „Eingang“ fest.
		1	output	Legt die Richtung auf „Ausgang“ fest.
input_mode	2..3	00	hi-z	Setzt den Eingang auf hochohmig.
		01	pull-low	Der Eingangspin wird auf Low-Pegel gezogen.
		10	pull-high	Der Eingangspin wird auf High-Pegel gezogen
		11	n/a	Ungültiger Zustand. Nicht setzen.
output_mode	4	0	set-low	Der Ausgangspin wird auf Low-Pegel gesteuert.
		1	set-high	Der Ausgangspin wird auf High-Pegel gesteuert.
input_status	5	x	in-val	0, wenn der Eingang < 1,5 V ist; 1, wenn der Eingang ≥ 1,5 V ist.

Wenn wir stattdessen den Zustand überprüfen würden, bevor wir die zugrundeliegende Hardware nutzen, und dabei unsere Entwurfsverträge zur Laufzeit durchsetzen würden, könnten wir stattdessen Code schreiben, der wie folgt aussieht:

```
/// GPIO-Schnittstelle
struct GpioConfig {
    /// Von svd2rust generierte GPIO-Konfigurationsstruktur
    periph: GPIO_CONFIG,
}

impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Muss aktiviert sein, um die Richtung festzulegen.
            return Err(());
        }

        self.periph.modify(|r, w| {
            w.direction().set_bit(is_output)
        });

        Ok(())
    }

    pub fn set_input_mode(&mut self, variant: InputMode) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Muss aktiviert sein, um den Eingabemodus festzulegen.
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // Die Richtung muss input sein.
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.input_mode().variant(variant)
        });

        Ok(())
    }

    pub fn set_output_status(&mut self, is_high: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Muss aktiviert sein, um den Ausgangsstatus festzulegen.
            return Err(());
        }

        if self.periph.read().direction().bit_is_clear() {
            // Die Richtung muss output sein.
```

```
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });

        Ok(())
    }

    pub fn get_input_status(&self) -> Result<bool, ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Muss aktiviert sein, um den Status abzurufen.
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // Die Richtung muss input sein.
            return Err(());
        }

        Ok(self.periph.read().input_status().bit_is_set())
    }
}
```

Da wir die Hardware-Beschränkungen durchsetzen müssen, führen wir letztlich zahlreiche Laufzeitüberprüfungen durch – was Zeit und Ressourcen verschwendet –, und der Code ist für Entwickler deutlich weniger angenehm zu handhaben.

5.3.1. Typzustände

Was aber, wenn wir stattdessen das Typsystem von Rust nutzen würden, um die Regeln für Zustandsübergänge durchzusetzen? Betrachten wir dieses Beispiel:

```
/// GPIO-Schnittstelle
struct GpioConfig<ENABLED, DIRECTION, MODE> {
    /// Von svd2rust generierte GPIO-Konfigurationsstruktur
    periph: GPIO_CONFIG,
    enabled: ENABLED,
    direction: DIRECTION,
    mode: MODE,
}

// Typzustaende fuer MODE in GpioConfig
struct Disabled;
struct Enabled;
struct Output;
struct Input;
struct PulledLow;
struct PulledHigh;
struct HighZ;
struct DontCare;

/// Diese Funktionen koennen an jedem GPIO-Pin verwendet werden.
impl<EN, DIR, IN_MODE> GpioConfig<EN, DIR, IN_MODE> {
    pub fn into_disabled(self) -> GpioConfig<Disabled, DontCare, DontCare> {
        self.periph.modify(|_r, w| w.enable.disabled());
        GpioConfig {
```

```
            periph: self.periph,
            enabled: Disabled,
            direction: DontCare,
            mode: DontCare,
        }
    }

    pub fn into_enabled_input(self) -> GpioConfig<Enabled, Input, HighZ> {
        self.periph.modify(|_r, w| {
            w.enable.enabled()
             .direction.input()
             .input_mode.high_z()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: HighZ,
        }
    }

    pub fn into_enabled_output(self) -> GpioConfig<Enabled, Output, DontCare> {
        self.periph.modify(|_r, w| {
            w.enable.enabled()
             .direction.output()
             .input_mode.set_high()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Output,
            mode: DontCare,
        }
    }
}

/// Diese Funktion kann fuer einen Ausgangspin verwendet werden.
impl GpioConfig<Enabled, Output, DontCare> {
    pub fn set_bit(&mut self, set_high: bool) {
        self.periph.modify(|_r, w| w.output_mode.set_bit(set_high));
    }
}

/// Diese Methoden koennen auf jedem aktivierten Eingangs-GPIO verwendet werden.
impl<IN_MODE> GpioConfig<Enabled, Input, IN_MODE> {
    pub fn bit_is_set(&self) -> bool {
        self.periph.read().input_status.bit_is_set()
    }

    pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
        self.periph.modify(|_r, w| w.input_mode().high_z());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: HighZ,
```

```
        }
    }

    pub fn into_input_pull_down(self) -> GpioConfig<Enabled, Input, PulledLow> {
        self.periph.modify(|_r, w| w.input_mode().pull_low());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledLow,
        }
    }

    pub fn into_input_pull_up(self) -> GpioConfig<Enabled, Input, PulledHigh> {
        self.periph.modify(|_r, w| w.input_mode().pull_high());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledHigh,
        }
    }
}
```

Schauen wir uns nun an, wie der Code aussehen würde, der dies verwendet:

```
/*
 * Beispiel 1: Nicht konfiguriert als High-Z-Eingang
 */
let pin: GpioConfig<Disabled, _, _> = get_gpio();

// Das ist nicht moeglich, die PIN ist nicht aktiviert!
// pin.into_input_pull_down();

// Schalten Sie den Pin nun von „unkonfiguriert“ auf einen hochohmigen Eingang
  // (High-Z-Eingang) um.
let input_pin = pin.into_enabled_input();

// Vom Pin lesen
let pin_state = input_pin.bit_is_set();

// Das ist nicht moeglich; Eingangs-Pins verfuegen nicht über diese
  // Schnittstelle!
// input_pin.set_bit(true);

/*
 * Beispiel 2: Hochohmiger Eingang (High-Z) an auf Low-Pegel gezogenen Eingang
 */
let pulled_low = input_pin.into_input_pull_down();
let pin_state = pulled_low.bit_is_set();

/*
 * Beispiel 3: Eingang auf Low gezogen, Ausgang auf High gesetzt
 */
let output_pin = pulled_low.into_enabled_output();
output_pin.set_bit(true);
```

```
// Das geht nicht, Ausgangspins verfuegen nicht über diese Schnittstelle!
// output_pin.into_input_pull_down();
```

Dies ist zweifellos eine praktische Möglichkeit, den Zustand des Pins zu speichern, aber warum sollte man es auf diese Weise tun? Warum ist das besser, als den Zustand als `enum` innerhalb unserer `GpioConfig`-Struktur zu speichern?

5.3.2. Funktionale Sicherheit zur Kompilierzeit

Da wir unsere Designvorgaben vollständig zur Kompilierzeit durchsetzen, entstehen keinerlei Laufzeitkosten. Es ist unmöglich, einen Ausgabemodus einzustellen, solange sich ein Pin im Eingabemodus befindet. Stattdessen muss man die Zustände durchlaufen, indem man den Pin zunächst in einen Ausgangspin umwandelt und anschließend den Ausgabemodus festlegt. Dadurch entfällt der Laufzeitaufwand für die Überprüfung des aktuellen Zustands vor der Ausführung einer Funktion.

Da diese Zustände zudem durch das Typsystem erzwungen werden, sind Fehler bei der Verwendung dieser Schnittstelle ausgeschlossen. Versucht der Nutzer einen unzulässigen Zustandsübergang, lässt sich der Code nicht kompilieren!

5.4. Abstraktionen ohne Kosten

Typzustände (Type States) sind zudem ein hervorragendes Beispiel für „Zero-Cost Abstractions“ – also die Möglichkeit, bestimmte Verhaltensweisen in die Kompilierzeit zu verlagern (sei es zur Ausführung oder zur Analyse). Diese Typzustände enthalten keine eigentlichen Daten, sondern dienen als Markierungen. Da sie keine Daten enthalten, besitzen sie zur Laufzeit keine tatsächliche Repräsentation im Speicher:

```
use core::mem::size_of;

let _ = size_of::<Enabled>();    // == 0
let _ = size_of::<Input>();      // == 0
let _ = size_of::<PulledHigh>(); // == 0
let _ = size_of::<GpioConfig<Enabled, Input, PulledHigh>>(); // == 0
```

5.4.1. Typen der Größe Null

```
struct Enabled;
```

Strukturen dieser Art werden als Null-Typen bezeichnet, da sie keine tatsächlichen Daten enthalten. Obwohl sich diese Typen zur Kompilierzeit „real“ verhalten – man kann sie kopieren, verschieben, Referenzen darauf erstellen usw. –, werden sie vom Optimierer vollständig entfernt.

In diesem Code-Ausschnitt:

```
pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
    self.periph.modify(|_r, w| w.input_mode().high_z());
    GpioConfig {
        periph: self.periph,
        enabled: Enabled,
        direction: Input,
        mode: HighZ,
    }
}
```

Das von uns zurückgegebene `GpioConfig`-Objekt existiert zur Laufzeit nicht. Der Aufruf dieser Funktion läuft im Wesentlichen auf einen einzigen Assembler-Befehl hinaus: das Speichern eines konstanten Registerwerts an einer bestimmten Registeradresse. Das bedeutet, dass die von uns entwickelte „Type-State“-Schnittstelle eine Abstraktion ohne Laufzeitkosten (Zero-Cost-Abstraction) darstellt – sie verbraucht weder zusätzliche CPU- oder RAM-Ressourcen noch zusätzlichen Programmspeicher

für die Zustandsverwaltung von `GpioConfig` und führt zu demselben Maschinencode wie ein direkter Registerzugriff.

5.4.2. Verschachtelung

Im Allgemeinen lassen sich diese Abstraktionen beliebig tief verschachteln. Solange es sich bei allen verwendeten Komponenten um Typen ohne Größe (Zero-Sized Types) handelt, existiert die gesamte Struktur zur Laufzeit nicht.

Bei komplexen oder tief verschachtelten Strukturen kann es mühsam sein, alle möglichen Zustandskombinationen zu definieren. In solchen Fällen lassen sich Makros verwenden, um sämtliche Implementierungen zu generieren.

6. Portabilität

In eingebetteten Systemen ist Portabilität ein sehr wichtiges Thema: Jeder Hersteller und sogar jede Produktfamilie eines einzelnen Herstellers bietet unterschiedliche Peripheriegeräte und Funktionen an, und dementsprechend variieren auch die Interaktionsmöglichkeiten mit den Peripheriegeräten.

Eine gängige Methode, diese Unterschiede auszugleichen, ist die sogenannte Hardware-Abstraktionsschicht (HAL).

> Hardware-Abstraktionen sind Sammlungen von Software-Routinen, die bestimmte plattformspezifische Details emulieren und Programmen so direkten Zugriff auf die Hardwareressourcen ermöglichen.
>
> Sie ermöglichen es Programmierern häufig, geräteunabhängige Hochleistungsanwendungen zu schreiben, indem sie standardisierte Betriebssystemaufrufe für den Hardwarezugriff bereitstellen.
>
> *Wikipedia: Hardware Abstraction Layer*

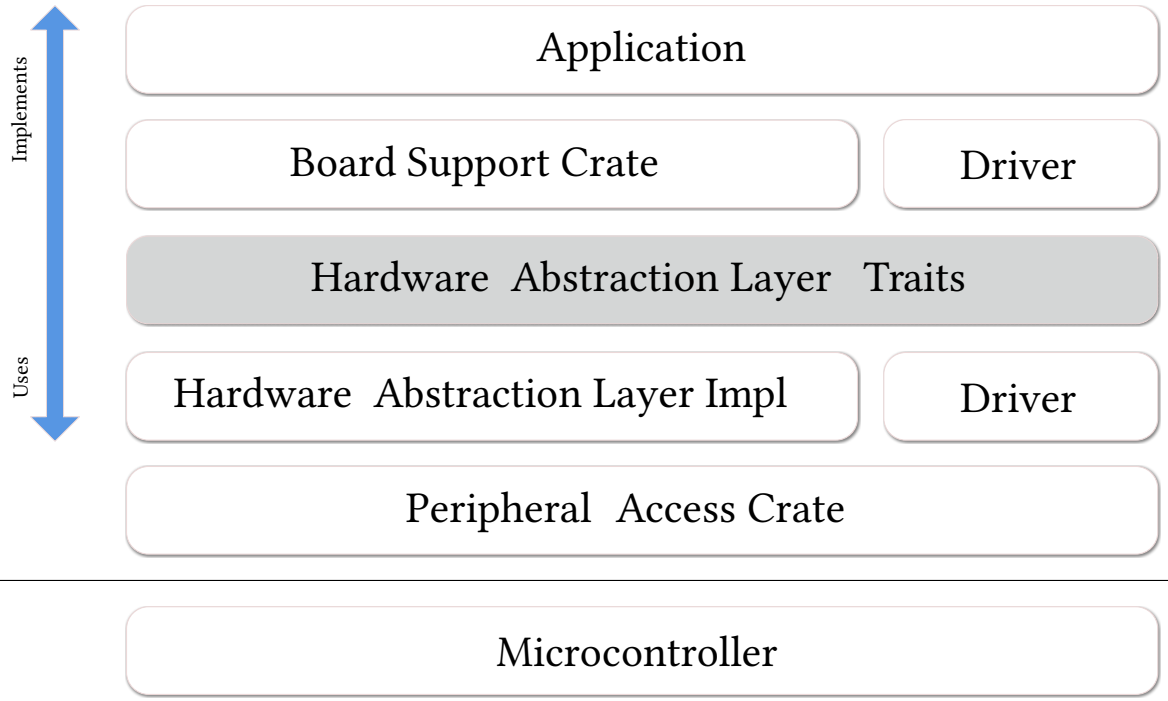
Eingebettete Systeme nehmen hier eine Sonderstellung ein, da wir es typischerweise nicht mit Betriebssystemen und vom Benutzer installierbarer Software zu tun haben, sondern mit Firmware-Images, die als Ganzes kompiliert werden – und zudem mit einer Reihe weiterer Einschränkungen konfrontiert sind. Der klassische, auf Wikipedia beschriebene Ansatz könnte zwar theoretisch funktionieren, ist aber wahrscheinlich nicht der effizienteste Weg, um Portabilität zu gewährleisten.

Wie gehen wir dabei in Rust vor? Hier kommt **embedded-hal** ins Spiel…

6.0.1. Was ist embedded-hal?

Kurz gesagt handelt es sich um eine Reihe von Merkmalen, die Implementierungsverträge zwischen **HAL-Implementierungen**, **Treibern** und **Anwendungen (oder Firmware)** definieren. Diese Verträge umfassen sowohl Fähigkeiten (d. h., wenn ein Merkmal für einen bestimmten Typ implementiert ist, stellt die **HAL-Implementierung** eine bestimmte Fähigkeit bereit) als auch Methoden (d. h., wenn ein Typ, der ein Merkmal implementiert, erstellt werden kann, ist garantiert, dass die im Merkmal spezifizierten Methoden verfügbar sind).

Ein typischer Schichtaufbau könnte wie folgt aussehen:



Einige der in **embedded-hal** definierten Traits sind:

- GPIO (Eingangs- und Ausgangspins)
- Serielle Kommunikation
- I2C
- SPI
- Timers/Countdowns
- Analog-Digital-Umsetzung

Der Hauptgrund für die Existenz der **embedded-hal**-Traits und der sie implementierenden sowie nutzenden Crates liegt darin, die Komplexität beherrschbar zu halten. Bedenkt man, dass eine Anwendung sowohl die Nutzung der Hardware-Peripherie als auch die eigentliche Anwendungslogik sowie möglicherweise Treiber für zusätzliche Hardwarekomponenten implementieren muss, wird schnell klar, dass die Wiederverwendbarkeit stark eingeschränkt ist. Mathematisch ausgedrückt: Wenn **M** die Anzahl der HAL-Implementierungen für Peripheriegeräte und **N** die Anzahl der Treiber ist, würde man – müsste man für jede Anwendung das Rad neu erfinden – bei **M** × **N** Implementierungen landen; durch die Verwendung der von den **embedded-hal**-Traits bereitgestellten API nähert sich die Implementierungskomplexität hingegen **M** + **N** an. Natürlich ergeben sich daraus weitere Vorteile, wie etwa ein geringerer Aufwand durch „Trial-and-Error“ dank wohldefinierter und sofort einsatzbereiter APIs.

6.0.2. Nutzer von embedded-hal

Wie bereits erwähnt, gibt es drei Hauptnutzer einer HAL:

6.0.2.1. HAL-Implementierung

Eine HAL-Implementierung stellt die Schnittstelle zwischen der Hardware und den Nutzern der HAL-Traits bereit. Typische Implementierungen bestehen aus drei Teilen:

- Ein oder mehrere hardwarespezifische Typen
- Funktionen zum Erstellen und Initialisieren eines solchen Typs, die häufig verschiedene Konfigurationsoptionen (Geschwindigkeit, Betriebsmodus, verwendete Pins usw.) bereitstellen.
- eine oder mehrere `trait`-Implementierungen (`impl`) von **embedded-hal**-Traits für diesen Typ

Eine solche **HAL-Implementierung** kann in verschiedenen Ausprägungen vorliegen:

- Über hardwarenahen Zugriff, z. B. über Register

- Über das Betriebssystem, z. B. durch Verwendung von `sysfs` unter Linux
- Über einen Adapter, z. B. ein Mock von Typen für Unit-Tests
- Via-Treiber für Hardware-Adapter, z. B. I2C-Multiplexer oder GPIO-Expander

6.0.2.2. Treiber

Ein Treiber implementiert eine Reihe benutzerdefinierter Funktionen für eine interne oder externe Komponente, die mit einem Peripheriegerät verbunden ist, das die embedded-hal-Traits implementiert. Typische Beispiele für solche Treiber sind verschiedene Sensoren (Temperatur-, Magnetometer-, Beschleunigungs- und Lichtsensoren), Anzeigegeräte (LED-Arrays, LCD-Displays) und Aktoren (Motoren, Sender).

Ein Treiber muss mit einer Instanz eines Typs initialisiert werden, der ein bestimmtes `Trait` von embedded-hal implementiert. Dies wird durch Trait-Bound sichergestellt. Der Treiber stellt eine eigene Typinstanz mit einem benutzerdefinierten Satz von Methoden bereit, die die Interaktion mit dem angesteuerten Gerät

6.0.2.3. Anwendung

Die Anwendung verknüpft die verschiedenen Komponenten miteinander und stellt sicher, dass die gewünschte Funktionalität erreicht wird. Bei der Portierung zwischen verschiedenen Systemen ist dies der Teil, der den größten Anpassungsaufwand erfordert, da die Anwendung die eigentliche Hardware über die HAL-Implementierung korrekt initialisieren muss – und die Initialisierung unterschiedlicher Hardware variiert, bisweilen sogar drastisch. Zudem spielen oft anwenderspezifische Entscheidungen eine wichtige Rolle: Komponenten können physisch an unterschiedliche Anschlüsse angebunden sein, Hardware-Busse erfordern mitunter externe Hardware zur Konfigurationsanpassung, oder es müssen Abwägungen bei der Nutzung interner Peripherie getroffen werden (etwa wenn mehrere Timer mit unterschiedlichen Fähigkeiten zur Verfügung stehen oder Konflikte zwischen verschiedenen Peripherieeinheiten auftreten).

7. Nebenläufigkeit

Nebenläufigkeit tritt immer dann auf, wenn verschiedene Teile Ihres Programms zu unterschiedlichen Zeiten oder in einer anderen als der vorgesehenen Reihenfolge ausgeführt werden können. Im Kontext eingebetteter Systeme umfasst dies:

- Interrupt-Handler, die immer dann ausgeführt werden, wenn der zugehörige Interrupt auftritt,
- verschiedene Formen des Multithreadings, bei denen Ihr Mikroprozessor regelmäßig zwischen Teilen Ihres Programms wechselt,
- und in einigen Systemen Mehrkern-Mikroprozessoren, bei denen jeder Kern gleichzeitig und unabhängig voneinander einen anderen Teil Ihres Programms ausführen kann.

Da viele eingebettete Programme mit Interrupts umgehen müssen, spielt Nebenläufigkeit früher oder später meist eine Rolle – und genau hier können auch viele schwer zu findende und komplexe Fehler auftreten. Glücklicherweise bietet Rust eine Reihe von Abstraktionen und Sicherheitsgarantien, die uns dabei helfen, korrekten Code zu schreiben.

7.0.1. Keine Nebenläufigkeit

Die einfachste Form der Nebenläufigkeit für ein Embedded-Programm ist der Verzicht darauf: Die Software besteht aus einer einzigen Hauptschleife, die kontinuierlich durchlaufen wird, und es kommen keinerlei Interrupts zum Einsatz. Manchmal ist genau dieser Ansatz für die vorliegende Aufgabenstellung ideal! Typischerweise liest die Schleife Eingabewerte ein, führt Berechnungen oder Verarbeitungen durch und gibt Ergebnisse aus.

```
#[entry]
fn main() {
    let peripherals = setup_peripherals();
    loop {
        let inputs = read_inputs(&peripherals);
        let outputs = process(inputs);
        write_outputs(&peripherals, outputs);
    }
}
```

Da keine Nebenläufigkeit vorliegt, müssen Sie sich keine Gedanken über die gemeinsame Datennutzung zwischen verschiedenen Programmteilen oder die Synchronisierung des Zugriffs auf Peripheriegeräte machen. Wenn ein solch einfacher Ansatz ausreicht, kann dies eine hervorragende Lösung sein.

7.0.2. Globale veränderliche Daten

Im Gegensatz zu Rust-Anwendungen außerhalb des Embedded-Bereichs haben wir meist nicht den Luxus, Speicher auf dem Heap zu reservieren und Referenzen auf diese Daten an einen neu erstellten Thread zu übergeben. Stattdessen können unsere Interrupt-Handler jederzeit aufgerufen werden und müssen wissen, wie sie auf den jeweils genutzten gemeinsamen Speicher zugreifen können. Auf unterster Ebene bedeutet dies, dass wir über *statisch reservierten*, veränderbaren Speicher verfügen müssen, auf den sowohl der Interrupt-Handler als auch der Hauptprogrammcode zugreifen können.

In Rust ist der Lese- oder Schreibzugriff auf solche `static mut`-Variablen stets als `unsafe` (unsicher) eingestuft. Ohne besondere Vorsichtsmaßnahmen könnte es nämlich zu einer sogenannten Race Condition (Wettlaufsituation) kommen: Der Zugriff auf die Variable wird mitten im Vorgang durch einen Interrupt unterbrochen, der seinerseits ebenfalls auf diese Variable zugreift.

Um zu veranschaulichen, wie dieses Verhalten subtile Fehler in Ihrem Code verursachen kann, betrachten Sie ein eingebettetes Programm, das die steigenden Flanken eines Eingangssignals innerhalb jedes Ein-Sekunden-Intervalls zählt (einen Frequenzzähler):

```
static mut COUNTER: u32 = 0;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // GEFAHR – Nicht wirklich sicher! Koennte zu Datenwettlaeufen
            //          fuehren.
            unsafe { COUNTER += 1 };
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    unsafe { COUNTER = 0; }
}
```

Sekündlich setzt der Timer-Interrupt den Zähler auf 0 zurück. Währenddessen misst die Hauptschleife kontinuierlich das Signal und erhöht den Zähler, sobald ein Wechsel von „Low“ auf „High” erkannt wird. Wir mussten das Schlüsselwort `unsafe` verwenden, um auf `COUNTER` zuzugreifen, da es als `static mut` deklariert ist; damit versichern wir dem Compiler, dass wir kein undefiniertes Verhalten verursachen. Erkennst du die Race Condition? Die Inkrementierung von `COUNTER` ist *nicht* garantiert atomar – tatsächlich wird sie auf den meisten Embedded-Plattformen in die Schritte Laden, Inkrementieren und Speichern aufgeteilt. Würde der Interrupt nach dem Laden, aber vor dem Speichern ausgelöst, ginge das Zurücksetzen auf 0 nach der Rückkehr aus dem Interrupt verloren – und wir würden für diesen Zeitraum doppelt so viele Signalwechsel zählen.

7.0.3. Kritische Abschnitte

Was können wir also gegen Data Races unternehmen? Ein einfacher Ansatz ist die Verwendung von *kritischen Abschnitten* – also Bereichen, in denen Interrupts deaktiviert sind. Indem wir den Zugriff auf `COUNTER` in der Funktion `main` in einen kritischen Abschnitt einbetten, stellen wir sicher, dass der Timer-Interrupt erst ausgelöst wird, nachdem wir das Inkrementieren von `COUNTER` abgeschlossen haben:

```
static mut COUNTER: u32 = 0;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // Ein neuer kritischer Abschnitt gewaehrleistet den
            // synchronisierten Zugriff auf COUNTER.
            cortex_m::interrupt::free(|_| {
                unsafe { COUNTER += 1 };
            });
        }
        last_state = state;
```

```
    }
}

#[interrupt]
fn timer() {
    unsafe { COUNTER = 0; }
}
```

In diesem Beispiel verwenden wir `cortex_m::interrupt::free`, doch auch andere Plattformen verfügen über ähnliche Mechanismen, um Code in einem kritischen Abschnitt auszuführen. Dies entspricht im Grunde dem Deaktivieren von Interrupts, der Ausführung von Code und dem anschließenden erneuten Aktivieren der Interrupts.

Beachten Sie, dass wir innerhalb des Timer-Interrupts keinen kritischen Abschnitt benötigten; dies hat zwei Gründe:

- Das Schreiben von 0 in `COUNTER` kann nicht von einer Race-Condition betroffen sein, da wir den Wert nicht lesen.
- Es wird ohnehin niemals vom `main`-Thread unterbrochen werden.

Wenn `COUNTER` von mehreren Interrupt-Handlern gemeinsam genutzt würde, die sich gegenseitig *unterbrechen* könnten, müsste jeder von ihnen ebenfalls einen kritischen Abschnitt verwenden.

Dies löst zwar unser unmittelbares Problem, doch wir müssen weiterhin viel unsicheren Code schreiben, dessen Verhalten wir sorgfältig analysieren müssen; zudem setzen wir möglicherweise unnötigerweise kritische Abschnitte ein. Da jeder kritische Abschnitt die Interrupt-Verarbeitung vorübergehend aussetzt, entstehen Kosten in Form von zusätzlichem Codeumfang sowie höherer Interrupt-Latenz und stärkerem Jitter (die Verarbeitung von Interrupts kann länger dauern, und die Zeit bis zur Verarbeitung schwankt stärker). Ob dies ein Problem darstellt, hängt vom jeweiligen System ab, doch im Allgemeinen sollten wir dies vermeiden.

Es ist wichtig zu beachten: Auch wenn ein kritischer Abschnitt garantiert, dass keine Interrupts ausgelöst werden, bietet er auf Mehrkernsystemen keine Exklusivitätsgarantie! Der andere Rechenkern könnte – selbst ohne Interrupts – problemlos auf denselben Speicher zugreifen wie Ihr eigener Kern. Wenn Sie mehrere Rechenkerne nutzen, benötigen Sie daher leistungsfähigere Synchronisationsmechanismen.

7.0.4. Atomarer Zugriff

Auf einigen Plattformen stehen spezielle atomare Befehle zur Verfügung, die Garantien für Lese-Modifizier-Schreib-Operationen (Read-Modify-Write) bieten. Speziell für Cortex-M gilt: `thumbv6` (Cortex-M0, Cortex-M0+) bietet lediglich atomare Lade- und Speicherbefehle, während `thumbv7` (Cortex-M3 und höher) vollständige „Compare-and-Swap“-Befehle (CAS) bereitstellt. Diese CAS-Befehle bieten eine Alternative zur drastischen Maßnahme, sämtliche Interrupts zu deaktivieren: Man kann versuchen, den Inkrementierungsvorgang durchzuführen – was meistens gelingt –, doch sollte eine Unterbrechung auftreten, wird der gesamte Vorgang automatisch erneut versucht. Diese atomaren Operationen sind auch in Mehrkernsystemen sicher.

```
use core::sync::atomic::{AtomicUsize, Ordering};

static COUNTER: AtomicUsize = AtomicUsize::new(0);

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
```

```
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // Verwenden Sie `fetch_add`, um 1 atomar zu COUNTER zu addieren.
            COUNTER.fetch_add(1, Ordering::Relaxed);
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Verwenden Sie `store`, um 0 direkt in COUNTER zu schreiben.
    COUNTER.store(0, Ordering::Relaxed)
}
```

Diesmal ist `COUNTER` eine sichere `static`-Variable. Dank des Typs `AtomicUsize` kann `COUNTER` sowohl vom Interrupt-Handler als auch vom Hauptthread aus sicher geändert werden, ohne Interrupts zu deaktivieren. Wenn möglich, ist dies die bessere Lösung – wird aber möglicherweise von Ihrer Plattform nicht unterstützt.

Hinweis zur `Ordering`: Diese beeinflusst, wie Compiler und Hardware Anweisungen neu anordnen, und hat auch Auswirkungen auf die Cache-Sichtbarkeit. Bei einer Einkernplattform ist die Option `Relaxed` ausreichend und in diesem Fall die effizienteste Wahl. Eine strengere Befehlsreihenfolge führt dazu, dass der Compiler Speicherbarrieren um die atomaren Operationen erzeugt. Je nachdem, wofür Sie atomare Operationen verwenden, benötigen Sie dies möglicherweise nicht. Die genauen Details des atomaren Modells sind komplex und werden am besten an anderer Stelle beschrieben.

Weitere Informationen zu atomaren Operationen und deren Reihenfolge finden Sie im nomicon.

7.0.5. Abstraktionen, Send und Sync

Keine der oben genannten Lösungen ist wirklich zufriedenstellend. Sie erfordern unsichere Blöcke, die sehr sorgfältig geprüft werden müssen und nicht ergonomisch sind. In Rust geht das doch bestimmt besser!

Wir können unseren Zähler in eine sichere Schnittstelle abstrahieren, die überall im Code sicher verwendet werden kann. In diesem Beispiel verwenden wir den Zähler für kritische Abschnitte, aber mit atomaren Operationen ließe sich etwas sehr Ähnliches realisieren.

```
use core::cell::UnsafeCell;
use cortex_m::interrupt;

// Unser Zaehler ist lediglich ein Wrapper um `UnsafeCell<u32>`, das Herzstueck
// der „Interior Mutability“ in Rust. Dank dieses Konzepts können wir `COUNTER`
// als `static` statt als `static mut` definieren und dennoch den Zaehlerwert
// veraendern.
struct CSCounter(UnsafeCell<u32>);

const CS_COUNTER_INIT: CSCounter = CSCounter(UnsafeCell::new(0));

impl CSCounter {
    pub fn reset(&self, _cs: &interrupt::CriticalSection) {
        // Indem wir die Uebergabe einer `CriticalSection` voraussetzen, wissen
        // wir, dass wir uns innerhalb einer `CriticalSection` befinden; daher
        // koennen wir bedenkenlos diesen `unsafe`-Block verwenden (der für den
        // Aufruf von `UnsafeCell::get` erforderlich ist).
```

```
        unsafe { *self.0.get() = 0 };
    }

    pub fn increment(&self, _cs: &interrupt::CriticalSection) {
        unsafe { *self.0.get() += 1 };
    }
}

// Erforderlich, um ein statisches CSCounter zu ermoeglichen. Siehe Erlaeuterung
// unten.
unsafe impl Sync for CSCounter {}

// COUNTER ist nicht mehr `mut`, da es „Interior Mutability“ verwendet; daher
// sind fuer den Zugriff auch keine `unsafe`-Bloecke mehr erforderlich.
static COUNTER: CSCounter = CS_COUNTER_INIT;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // No unsafe here!
            interrupt::free(|cs| COUNTER.increment(cs));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Wir muessen hier tatsaechlich einen kritischen Abschnitt betreten, nur um
    // ein gueltiges CS-Token zu erhalten, auch wenn wir wissen, dass kein
    // anderer Interrupt diesen unterbrechen koennte.
    interrupt::free(|cs| COUNTER.reset(cs));

    // Wir koennten „unsicheren“ Code (unsafe code) verwenden, um eine
    // gefaelschte CriticalSection zu erzeugen, falls wir das wirklich wollten,
    // und so den Overhead vermeiden:
    // let cs = unsafe { interrupt::CriticalSection::new() };
}
```

Wir haben unseren `unsafe`-Code in unsere sorgfältig entworfene Abstraktion verlagert; nun enthält unser Anwendungscode keine `unsafe`-Blöcke mehr.

Dieser Entwurf setzt voraus, dass die Anwendung ein `CriticalSection`-Token übergibt: Da diese Tokens nur sicher durch `interrupt::free` erzeugt werden können, stellen wir durch die Anforderung eines solchen Tokens sicher, dass wir uns innerhalb eines kritischen Abschnitts befinden, ohne die Sperre selbst implementieren zu müssen. Diese Garantie wird statisch vom Compiler gewährleistet; es entsteht also kein Laufzeit-Overhead durch `cs`. Hätten wir mehrere Zähler, könnten diese alle dasselbe `cs` verwenden, ohne dass mehrere verschachtelte kritische Abschnitte erforderlich wären.

Dies führt zu einem wichtigen Thema der Nebenläufigkeit in Rust: den Traits `Send` und `Sync`. Um das „Rust Book“ zusammenzufassen: Ein Typ ist `Send`, wenn er sicher in einen anderen Thread verschoben werden kann, während er `Sync` ist, wenn er sicher zwischen mehreren Threads geteilt werden kann.

Im Embedded-Kontext betrachten wir Interrupts als Prozesse, die in einem vom Anwendungscode getrennten Thread ausgeführt werden; daher müssen Variablen, auf die sowohl ein Interrupt als auch der Hauptcode zugreift, `Sync` implementieren.

Für die meisten Typen in Rust werden diese beiden Traits automatisch vom Compiler für dich abgeleitet. Da `CSCounter` jedoch eine `UnsafeCell` enthält, ist der Typ nicht `Sync`; folglich konnten wir keinen `static CSCounter` definieren, denn `static`-Variablen *müssen* `Sync` sein, da von mehreren Threads aus auf sie zugegriffen werden kann.

Um dem Compiler mitzuteilen, dass wir sichergestellt haben, dass der `CSCounter` tatsächlich gefahrlos zwischen Threads geteilt werden kann, implementieren wir das `Sync`-Trait explizit. Wie schon bei der früheren Verwendung kritischer Abschnitte ist dies nur auf Single-Core-Plattformen sicher; bei mehreren Kernen wäre ein deutlich höherer Aufwand erforderlich, um die Sicherheit zu gewährleisten.

7.0.6. Mutexe

Wir haben eine nützliche, speziell auf unser Zählerproblem zugeschnittene Abstraktion entwickelt; es gibt jedoch viele gängige Abstraktionen für Nebenläufigkeit.

Ein solches *Synchronisationsprimitiv* ist der Mutex (kurz für „mutual exclusion“, also gegenseitiger Ausschluss). Diese Konstrukte gewährleisten den exklusiven Zugriff auf eine Variable, wie etwa unseren Zähler. Ein Thread kann versuchen, den Mutex zu *sperren* (oder zu *erwerben*); dabei ist er entweder sofort erfolgreich, blockiert, bis die Sperre verfügbar ist, oder erhält eine Fehlermeldung, dass der Mutex nicht gesperrt werden konnte. Solange der Thread die Sperre hält, hat er Zugriff auf die geschützten Daten. Ist der Thread fertig, *entsperrt* (oder *gibt*) er den Mutex *frei*, sodass ein anderer Thread ihn sperren kann. In Rust würden wir die Freigabe üblicherweise mithilfe des `Drop`-Traits implementieren; so ist sichergestellt, dass der Mutex immer freigegeben wird, sobald er den Gültigkeitsbereich (Scope) verlässt.

Die Verwendung eines Mutex in Interrupt-Handlern kann tückisch sein: Normalerweise darf ein Interrupt-Handler nicht blockieren. Besonders fatal wäre es, wenn er blockieren würde, während er darauf wartet, dass der Haupt-Thread eine Sperre (Lock) freigibt, da dies zu einem *Deadlock* führen würde (der Haupt-Thread würde die Sperre niemals freigeben, da die Ausführung im Interrupt-Handler verbleibt). Ein Deadlock gilt nicht als „unsicher“ (unsafe): Er ist selbst in sicherem Rust möglich.

Um dieses Verhalten vollständig zu vermeiden, könnten wir einen Mutex implementieren, der für das Sperren eine kritische Sektion erfordert – genau wie in unserem Zähler-Beispiel. Solange die Dauer der kritischen Sektion der Dauer der Sperre entspricht, ist sichergestellt, dass wir exklusiven Zugriff auf die gekapselte Variable haben, ohne den Sperrstatus des Mutex explizit nachverfolgen zu müssen.

Genau dies wird uns bereits durch die `cortex_m`-Crate abgenommen! Wir hätten unseren Zähler auch unter Verwendung dieser Crate implementieren können:

```
use core::cell::Cell;
use cortex_m::interrupt::Mutex;

static COUNTER: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            interrupt::free(|cs|
```

```
                COUNTER.borrow(cs).set(COUNTER.borrow(cs).get() + 1));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Wir muessen hier noch einen kritischen Abschnitt betreten, um die
    // Mutex-Bedingung zu erfuellen.
    interrupt::free(|cs| COUNTER.borrow(cs).set(0));
}
```

Wir verwenden nun `Cell`; dieses bietet – ebenso wie sein Pendant `RefCell` – eine sichere Form der „Interior Mutability“ (internen Veränderbarkeit). Wir haben bereits `UnsafeCell` kennengelernt, die unterste Ebene der internen Veränderbarkeit in Rust: Sie ermöglicht es, mehrere veränderbare Referenzen auf den enthaltenen Wert zu erhalten, allerdings nur mittels `unsafe`-Code. Eine `Cell` ähnelt einer `UnsafeCell`, stellt jedoch eine sichere Schnittstelle bereit: Sie erlaubt lediglich das Kopieren des aktuellen Werts oder dessen Austausch, nicht jedoch den Zugriff über eine Referenz; zudem ist sie nicht `Sync` und kann daher nicht zwischen Threads geteilt werden. Aufgrund dieser Einschränkungen ist ihre Verwendung sicher, allerdings lässt sie sich nicht direkt in einer `static`-Variablen einsetzen, da `static`-Elemente die Eigenschaft `Sync` erfüllen müssen.

Warum funktioniert das obige Beispiel? Der `Mutex<T>` implementiert `Sync` für jedes `T`, das `Send` ist – wie beispielsweise eine `Cell`. Dies ist sicher möglich, da der Zugriff auf den Inhalt nur während eines kritischen Abschnitts gewährt wird. Dadurch erhalten wir einen sicheren Zähler ohne jeglichen unsicheren Code!

Das ist ideal für einfache Typen wie den `u32` unseres Zählers. Was aber ist mit komplexeren Typen, die nicht `Copy` sind? Ein sehr häufiges Beispiel in eingebetteten Systemen ist eine Peripheriestruktur, die in der Regel nicht `Copy` ist. Für diesen Fall können wir `RefCell` verwenden.

7.0.7. Gemeinsame Nutzung von Peripheriegeräten

Mit `svd2rust` und ähnlichen Abstraktionen erzeugte Device-Crates ermöglichen einen sicheren Zugriff auf Peripheriekomponenten, indem sie sicherstellen, dass zu jedem Zeitpunkt nur eine einzige Instanz der entsprechenden Peripherie-Struktur existiert. Dies gewährleistet zwar die Sicherheit, erschwert jedoch den gleichzeitigen Zugriff auf eine Peripheriekomponente sowohl aus dem Haupt-Thread als auch aus einem Interrupt-Handler heraus.

Um den Zugriff auf Peripheriekomponenten sicher zu teilen, können wir den bereits bekannten `Mutex` verwenden. Zudem benötigen wir `RefCell`; dieses stellt mittels einer Laufzeitprüfung sicher, dass jeweils nur eine einzige Referenz auf eine Peripheriekomponente ausgegeben wird. Dies ist zwar mit einem höheren Overhead verbunden als bei einem einfachen `Cell`, doch da wir Referenzen anstatt Kopien weitergeben, müssen wir sicherstellen, dass zu jedem Zeitpunkt nur eine einzige Referenz existiert.

Schließlich müssen wir auch berücksichtigen, wie die Peripheriekomponente in die gemeinsam genutzte Variable übertragen werden kann, nachdem sie im Hauptcode initialisiert wurde. Hierfür können wir den Typ `Option` verwenden, der zunächst mit `None` initialisiert und später auf die Instanz der Peripheriekomponente gesetzt wird.

```
use core::cell::RefCell;
use cortex_m::interrupt::{self, Mutex};
use stm32f4::stm32f405;
```

```rust
static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    // Die Peripherie-Singletons abrufen und konfigurieren.
    // Dieses Beispiel stammt aus einer mit svd2rust erstellten Crate, die
    // meisten Crates für eingebettete Geräte sind jedoch ähnlich.
    let dp = stm32f405::Peripherals::take().unwrap();
    let gpioa = &dp.GPIOA;

    // Eine Art Konfigurationsfunktion.
    // Gehen wir davon aus, dass es PA0 als Eingang und PA1 als Ausgang
    // konfiguriert.
    configure_gpio(gpioa);

    // Speichere GPIOA im Mutex und verschiebe es dabei.
    interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
    // Wir koennen `gpioa` oder `dp.GPIOA` nicht mehr verwenden, sondern
    // muessen stattdessen ueber den Mutex darauf zugreifen.

    // Achten Sie darauf, den Interrupt erst nach dem Setzen von MY_GPIO zu
    // aktivieren:
    // Andernfalls koennte der Interrupt ausgeloest werden, solange MY_GPIO
    // noch None enthaelt, was – wie implementiert (mit `unwrap()`) – zu einer
    // Panic fuehren wuerde.
    set_timer_1hz();
    let mut last_state = false;
    loop {
        // Wir lesen den Status nun als digitalen Eingang ueber den Mutex aus.
        let state = interrupt::free(|cs| {
            let gpioa = MY_GPIO.borrow(cs).borrow();
            gpioa.as_ref().unwrap().idr.read().idr0().bit_is_set()
        });

        if state && !last_state {
            // Setze PA1 auf High, wenn an PA0 eine steigende Flanke erkannt
            // wurde.
            interrupt::free(|cs| {
                let gpioa = MY_GPIO.borrow(cs).borrow();
                gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
            });
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Diesmal setzen wir im Interrupt lediglich PA0 zurueck.
    interrupt::free(|cs| {
        // Wir koennen `unwrap()` verwenden, da wir wissen, dass der Interrupt
        // erst aktiviert wurde, nachdem `MY_GPIO` gesetzt war; andernfalls
        // muessten wir den moeglichen Fall eines `None`-Werts behandeln.
        let gpioa = MY_GPIO.borrow(cs).borrow();
```

```
        gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().clear_bit());
    });
}
```

Das ist eine ganze Menge, die man erst einmal verarbeiten muss – schauen wir uns also die wichtigen Zeilen genauer an.

```
static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =
    Mutex::new(RefCell::new(None));
```

Unsere gemeinsam genutzte Variable ist nun ein `Mutex`, der einen `RefCell` umschließt, welcher wiederum ein `Option` enthält. Der `Mutex` stellt sicher, dass der Zugriff nur innerhalb eines kritischen Abschnitts erfolgt, und macht die Variable dadurch `Sync` – auch wenn ein einfacher `RefCell` dies nicht wäre. Der `RefCell` ermöglicht uns „Interior Mutability“ (interne Veränderbarkeit) bei Verwendung von Referenzen, was wir für unseren `GPIOA` benötigen. Das `Option` erlaubt es uns, die Variable zunächst leer zu initialisieren und den eigentlichen Wert erst später hineinzubewegen. Da wir nicht statisch, sondern nur zur Laufzeit auf das Peripherie-Singleton zugreifen können, ist dieses Vorgehen erforderlich.

```
interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
```

Innerhalb eines kritischen Abschnitts können wir `borrow()` auf dem Mutex aufrufen, was uns eine Referenz auf die `RefCell` liefert. Anschließend rufen wir `replace()` auf, um unseren neuen Wert in die `RefCell` zu verschieben.

```
interrupt::free(|cs| {
    let gpioa = MY_GPIO.borrow(cs).borrow();
    gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
});
```

Schließlich verwenden wir `MY_GPIO` auf sichere und nebenläufige Weise. Der kritische Abschnitt verhindert wie üblich das Auslösen von Interrupts und ermöglicht es uns, den Mutex auszuleihen. Die `RefCell` liefert uns dann ein `&Option<GPIOA>` und überwacht die Dauer der Ausleihe; sobald die Referenz ihren Gültigkeitsbereich verlässt, wird der Status der `RefCell` aktualisiert, um anzuzeigen, dass sie nicht mehr ausgeliehen ist.

Da wir das `GPIOA`-Objekt nicht aus dem `&Option` herausverschieben können, müssen wir es mittels `as_ref()` in ein `&Option<&GPIOA>` umwandeln. Dieses können wir schließlich mit `unwrap()` auflösen, um das `&GPIOA` zu erhalten, das uns den Zugriff zur Modifikation der Peripherieeinheit ermöglicht.

Wenn wir eine veränderbare Referenz auf eine gemeinsam genutzte Ressource benötigen, sollten stattdessen `borrow_mut` und `deref_mut` verwendet werden. Der folgende Code zeigt ein Beispiel unter Verwendung des TIM2-Timers.

```
use core::cell::RefCell;
use core::ops::DerefMut;
use cortex_m::interrupt::{self, Mutex};
use cortex_m::asm::wfi;
use stm32f4::stm32f405;

static G_TIM: Mutex<RefCell<Option<Timer<stm32::TIM2>>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    let mut cp = cm::Peripherals::take().unwrap();
    let dp = stm32f405::Peripherals::take().unwrap();
```

```
    // Eine Art Timer-Konfigurationsfunktion.
    // Angenommen, sie konfiguriert den TIM2-Timer sowie dessen NVIC-Interrupt
    // und startet schließlich den Timer.
    let tim = configure_timer_interrupt(&mut cp, dp);

    interrupt::free(|cs| {
        G_TIM.borrow(cs).replace(Some(tim));
    });

    loop {
        wfi();
    }
}

#[interrupt]
fn timer() {
    interrupt::free(|cs| {
        if let Some(ref mut tim)) =  G_TIM.borrow(cs).borrow_mut().deref_mut() {
            tim.start(1.hz());
        }
    });
}
```

Puh! Das ist zwar sicher, aber auch etwas unhandlich. Können wir sonst noch etwas tun?

7.0.8. RTIC

Eine Alternative ist das RTIC-Framework (kurz für *Real Time Interrupt-driven Concurrency*). Es erzwingt statische Prioritäten und überwacht Zugriffe auf `static mut`-Variablen („Ressourcen“), um statisch sicherzustellen, dass auf gemeinsam genutzte Ressourcen stets sicher zugegriffen wird – ohne den Overhead, der durch das ständige Betreten kritischer Abschnitte oder die Verwendung von Referenzzählung (wie bei `RefCell`) entstünde. Dies bietet eine Reihe von Vorteilen, wie etwa die Garantie von Deadlock-Freiheit sowie einen extrem geringen Zeit- und Speicheraufwand.

RTIC bietet einen asynchronen Executor, sodass Ihre Software-Tasks als `async`-Funktionen implementiert sind; darin können Sie neben herkömmlichen synchronen APIs auch `async`-APIs nutzen.

Das Framework umfasst zudem Funktionen wie Message Passing – was den Bedarf an explizit gemeinsam genutztem Zustand (Shared State) verringert – sowie die Möglichkeit, Tasks für einen bestimmten Zeitpunkt zu planen, womit sich beispielsweise periodische Aufgaben realisieren lassen. Weitere Informationen finden Sie in der Dokumentation!

7.0.9. Embassy

Embassy ist ein Ökosystem von Bibliotheken, die sich auf die Nutzung der in Rust enthaltenen `async`/`await`-Syntax für Nebenläufigkeit konzentrieren. Das Herzstück von Embassy ist sein asynchroner Executor, der die gängigsten MCU-Architekturen unterstützt.

Embassy verfolgt zudem einen „Batteries-included“-Ansatz und bietet viele weitere Komponenten an, zum Beispiel:

- Zeit-Bibliothek
- Verschiedene HAL-Bibliotheken, die auch Unterstützung für die Zeitbibliothek bieten.
- embassy-sync für Synchronisationsprimitive

Weitere Informationen finden Sie auf der Website und im Buch.

7.0.10. Echtzeitbetriebssysteme

Ein weiteres verbreitetes Modell für Nebenläufigkeit in eingebetteten Systemen ist das Echtzeitbetriebssystem (RTOS). Während diese in Rust bislang weniger stark erforscht sind, kommen sie in der klassischen Entwicklung eingebetteter Systeme häufig zum Einsatz. Zu den Open-Source-Beispielen zählen FreeRTOS und ChibiOS. Solche Echtzeitbetriebssysteme unterstützen die Ausführung mehrerer Anwendungsthreads, zwischen denen die CPU wechselt – entweder wenn die Threads die Kontrolle freiwillig abgeben (kooperatives Multitasking) oder gesteuert durch Timer bzw. Interrupts (präemptives Multitasking). Typischerweise stellen RTOS Mechanismen wie Mutexe und andere Synchronisationsprimitive bereit und interagieren häufig mit Hardwarefunktionen wie DMA-Controllern.

Zum jetzigen Zeitpunkt gibt es noch nicht viele Beispiele für Rust-RTOS, auf die man verweisen könnte; es handelt sich jedoch um ein interessantes Gebiet – bleiben Sie also dran!

7.0.11. Mehrere Kerne

Es wird immer üblicher, eingebettete Prozessoren mit zwei oder mehr Kernen auszustatten, was die Parallelverarbeitung zusätzlich verkompliziert. Alle Beispiele mit kritischen Abschnitten (einschließlich `cortex_m::interrupt::Mutex`) gehen davon aus, dass der einzige weitere Ausführungsthread der Interrupt-Thread ist. Auf einem Mehrkernsystem trifft dies jedoch nicht mehr zu. Stattdessen benötigen wir Synchronisierungsprimitive, die speziell für Mehrkernsysteme (auch SMP, für symmetrisches Multiprocessing, genannt) entwickelt wurden.

Diese verwenden typischerweise die bereits erwähnten atomaren Befehle, da das Verarbeitungssystem die Atomarität über alle Kerne hinweg gewährleistet.

Eine detaillierte Behandlung dieser Themen würde den Rahmen dieses Buches sprengen, die allgemeinen Muster sind jedoch dieselben wie im Einzelkernfall.

8. Sammlungen (Collections)

Irgendwann wirst du in deinem Programm dynamische Datenstrukturen (auch bekannt als Collections) verwenden wollen. Die Standardbibliothek (`std`) stellt eine Reihe gängiger Collections bereit: `Vec`, `String`, `HashMap` usw. Alle in `std` implementierten Collections nutzen einen globalen dynamischen Speicherverwalter (den sogenannten Heap).

Da `core` definitionsgemäß frei von Speicherzuweisungen ist, stehen diese Implementierungen dort nicht zur Verfügung; sie sind jedoch im `alloc`-Crate zu finden, das mit dem Compiler ausgeliefert wird.

Wenn Sie Sammlungen benötigen, ist eine auf dem Heap alloziierte Implementierung nicht Ihre einzige Option. Sie können auch Sammlungen mit *fester Kapazität* verwenden; eine solche Implementierung findet sich im `heapless`-Crate.

In diesem Abschnitt werden wir diese beiden Implementierungen untersuchen und vergleichen.

8.0.1. Verwendung von `alloc`

Die `alloc`-Crate ist in der Standard-Rust-Distribution enthalten. Um das Crate zu importieren, können Sie es direkt per `use` einbinden, *ohne* es als Abhängigkeit in Ihrer `Cargo.toml`-Datei zu deklarieren.

```
#![feature(alloc)]

extern crate alloc;

use alloc::vec::Vec;
```

Um eine beliebige Collection nutzen zu können, müssen Sie zunächst das Attribut `global_allocator` verwenden, um den globalen Allocator zu deklarieren, den Ihr Programm einsetzen soll. Der gewählte Allocator muss zwingend das Trait `GlobalAlloc` implementieren.

Der Vollständigkeit halber und um diesen Abschnitt so in sich abgeschlossen wie möglich zu halten, implementieren wir einen einfachen Bump-Pointer-Allokator und verwenden diesen als globalen Allokator. Wir empfehlen Ihnen jedoch *dringend*, in Ihrem Programm stattdessen einen praxiserprobten Allokator von crates.io zu verwenden.

```
// Implementierung eines Bump-Pointer-Allocators

use core::alloc::{GlobalAlloc, Layout};
use core::cell::UnsafeCell;
use core::ptr;

use cortex_m::interrupt;

// Bump-Pointer-Allocator fuer Systeme mit *einem* Rechenkern
struct BumpPointerAlloc {
    head: UnsafeCell<usize>,
    end: usize,
}

unsafe impl Sync for BumpPointerAlloc {}

unsafe impl GlobalAlloc for BumpPointerAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // `interrupt::free` ist ein kritischer Abschnitt, der die sichere
        // Verwendung unseres Allocators aus Interrupts heraus ermoeglicht.
        interrupt::free(|_| {
            let head = self.head.get();
```

```
            let size = layout.size();
            let align = layout.align();
            let align_mask = !(align - 1);

            // Verschiebe den Startpunkt zur naechsten Ausrichtungsgrenze.
            let start = (*head + align - 1) & align_mask;

            if start + size > self.end {
                // ein Nullzeiger signalisiert einen „Out of Memory“-Zustand
                ptr::null_mut()
            } else {
                *head = start + size;
                start as *mut u8
            }
        })
    }

    unsafe fn dealloc(&self, _: *mut u8, _: Layout) {
        // Dieser Allokator gibt niemals Speicher frei.
    }
}

// Deklaration des globalen Speicher-Allocators
// HINWEIS: Der Benutzer muss sicherstellen, dass der Speicherbereich
// `[0x2000_0100, 0x2000_0200]` nicht von anderen Teilen des Programms
// verwendet wird.
#[global_allocator]
static HEAP: BumpPointerAlloc = BumpPointerAlloc {
    head: UnsafeCell::new(0x2000_0100),
    end: 0x2000_0200,
};
```

Neben der Auswahl eines globalen Allocators muss der Benutzer auch festlegen, wie mit „Out of Memory“ (OOM)-Fehlern umgegangen wird; hierfür wird das *instabile* Attribut `alloc_error_handler` verwendet.

```
#![feature(alloc_error_handler)]

use cortex_m::asm;

#[alloc_error_handler]
fn on_oom(_layout: Layout) -> ! {
    asm::bkpt();

    loop {}
}
```

Sobald alles vorhanden ist, kann der Benutzer endlich die Sammlungen in „alloc“ verwenden.

```
#[entry]
fn main() -> ! {
    let mut xs = Vec::new();

    xs.push(42);
    assert!(xs.pop(), Some(42));

    loop {
        // ..
```

```
    }
}
```

Wenn Sie die Sammlungen aus dem `std`-Crate bereits verwendet haben, werden Ihnen diese vertraut vorkommen, da es sich um exakt dieselbe Implementierung handelt.

8.0.2. Verwendung von `heapless`

`heapless` erfordert keine Einrichtung, da seine Datenstrukturen nicht von einem globalen Speicher-Allocator abhängen. Binde die Datenstrukturen einfach per `use` ein und instanziiere sie:

```
// heapless version: v0.4.x
use heapless::Vec;
use heapless::consts::*;

#[entry]
fn main() -> ! {
    let mut xs: Vec<_, U8> = Vec::new();

    xs.push(42).unwrap();
    assert_eq!(xs.pop(), Some(42));
    loop {}
}
```

Sie werden zwei Unterschiede zwischen diesen Sammlungen und denen in `alloc` feststellen.

Erstens muss die Kapazität der Sammlung im Voraus deklariert werden. `heapless`-Sammlungen werden nie neu allokiert und haben eine feste Kapazität; diese Kapazität ist Teil der Typsignatur der Sammlung. In diesem Fall haben wir deklariert, dass `xs` eine Kapazität von 8 Elementen hat, d. h. der Vektor kann maximal 8 Elemente aufnehmen. Dies wird durch das `u8` (siehe `typenum`) in der Typsignatur angezeigt.

Zweitens geben die Methode `push` sowie viele weitere Methoden ein `Result` zurück. Da `heapless`-Datenstrukturen über eine feste Kapazität verfügen, können alle Operationen, die Elemente in die Struktur einfügen, potenziell fehlschlagen. Die API trägt diesem Umstand Rechnung, indem sie ein `Result` liefert, das anzeigt, ob die Operation erfolgreich war oder nicht. Im Gegensatz dazu passen `alloc`-Datenstrukturen ihre Kapazität durch eine erneute Speicherzuweisung auf dem Heap an.

Seit Version v0.4.x speichern alle `heapless`-Datenstrukturen ihre Elemente direkt „inline“. Das bedeutet, dass eine Operation wie `let x = heapless::Vec::new();` die Datenstruktur auf dem Stack anlegt; es ist jedoch auch möglich, sie in einer `static`-Variablen oder sogar auf dem Heap (`Box<Vec<_, _>>`) zu speichern.

8.0.3. Abwägungen

Berücksichtigen Sie diese Punkte, wenn Sie zwischen auf dem Heap alloziierten, verschiebbaren Sammlungen und Sammlungen mit fester Kapazität wählen.

8.0.3.1. „Out of Memory“ und Fehlerbehandlung

Bei Heap-Allokationen ist ein „Out of Memory“ (OOM) – also ein Speichermangel – immer möglich; er kann überall dort auftreten, wo eine Collection wachsen muss: So können beispielsweise alle Aufrufe von `alloc::Vec.push` potenziell zu einer OOM-Situation führen. Folglich können manche Operationen *implizit* fehlschlagen. Einige `alloc`-Collections bieten `try_reserve`-Methoden an, mit denen sich potenzielle OOM-Situationen beim Vergrößern der Collection überprüfen lassen; man muss diese Methoden jedoch aktiv einsetzen.

Wenn man ausschließlich `heapless`-Collections verwendet und keinen Speicher-Allocator für andere Zwecke nutzt, ist eine OOM-Situation ausgeschlossen. Stattdessen muss man im Einzelfall damit umgehen, wenn die Kapazität einer Collection erschöpft ist. Das bedeutet, man muss *alle* `Result`-Werte behandeln, die von Methoden wie `Vec.push` zurückgegeben werden.

Fehler durch Speichermangel (OOM) können schwieriger zu debuggen sein als etwa das Aufrufen von `unwrap` auf allen `Result`-Werten von `heapless::Vec.push`, da die Stelle, an der der Fehler sichtbar wird, nicht unbedingt mit der Ursache des Problems übereinstimmen muss. So kann beispielsweise selbst `vec.reserve(1)` ein OOM auslösen, wenn der Allocator nahezu erschöpft ist, weil eine andere Collection Speicher verloren hat (Speicherlecks sind auch in „Safe Rust“ möglich).

8.0.3.2. Speichernutzung

Die Einschätzung des Speicherverbrauchs von auf dem Heap allozierten Collections ist schwierig, da sich die Kapazität langlebiger Collections zur Laufzeit ändern kann. Manche Operationen führen möglicherweise zu einer impliziten Neuzuweisung des Speichers, wodurch der Speicherbedarf steigt; andere Collections bieten Methoden wie `shrink_to_fit` an, die den genutzten Speicher potenziell verringern können – wobei letztlich der Allocator entscheidet, ob die Speicherzuweisung tatsächlich reduziert wird. Zudem muss der Allocator unter Umständen mit Speicherfragmentierung umgehen, was den *scheinbaren* Speicherverbrauch erhöhen kann.

Verwendet man hingegen ausschließlich Collections mit fester Kapazität, legt die meisten davon in `static`-Variablen ab und legt eine maximale Größe für den Aufruf-Stack (Call Stack) fest, so erkennt der Linker, wenn mehr Speicher beansprucht wird, als physisch verfügbar ist.

Darüber hinaus werden auf dem Stack alloziierte Collections mit fester Kapazität durch das Flag `-Z emit-stack-sizes` erfasst; das bedeutet, dass Werkzeuge zur Analyse der Stack-Nutzung (wie `stack-sizes`) diese in ihre Auswertung einbeziehen.

Sammlungen mit fester Kapazität können jedoch *nicht* verkleinert werden, was zu niedrigeren Auslastungsgraden (dem Verhältnis zwischen der Größe der Sammlung und ihrer Kapazität) führen kann, als sie bei Sammlungen mit veränderbarer Kapazität möglich sind.

8.0.3.3. Maximale Ausführungszeit (WCET)

Wenn Sie zeitkritische oder Echtzeitanwendungen entwickeln, ist Ihnen die Worst-Case-Ausführungszeit (WCET) der verschiedenen Programmteile wahrscheinlich sehr wichtig.

Da `alloc`-Collections Speicher neu allokieren können, umfasst die WCET von Operationen, die die Collection vergrößern, auch die Zeit für die Neuallokation. Diese hängt wiederum von der *Laufzeitkapazität* der Collection ab. Daher ist es schwierig, die WCET beispielsweise der Operation `alloc::Vec.push` zu bestimmen, da sie sowohl vom verwendeten Allokator als auch von dessen Laufzeitkapazität abhängt.

Collections mit fester Kapazität hingegen allokieren nie Speicher neu, sodass alle Operationen eine vorhersehbare Ausführungszeit haben. Beispielsweise wird `heapless::Vec.push` in konstanter Zeit ausgeführt.

8.0.3.4. Benutzerfreundlichkeit

Für `alloc` muss ein globaler Allocator eingerichtet werden, für `heapless` hingegen nicht. Allerdings erfordert `heapless`, dass man bei der Instanziierung jeder Collection deren Kapazität festlegt.

Die `alloc`-API dürfte so gut wie jedem Rust-Entwickler vertraut sein. Die `heapless`-API orientiert sich zwar eng an der `alloc`-API, unterscheidet sich jedoch aufgrund der expliziten Fehlerbehandlung – manche Entwickler empfinden diese explizite Fehlerbehandlung womöglich als übertrieben oder zu umständlich.

9. Design-Muster

Dieses Kapitel zielt darauf ab, verschiedene nützliche Entwurfsmuster für Embedded-Rust zusammenzustellen.

9.1. HAL-Design-Muster

Hierbei handelt es sich um eine Sammlung bewährter und empfohlener Muster für die Implementierung von Hardware-Abstraktionsschichten (HALs) für Mikrocontroller in Rust. Diese Muster sind als Ergänzung zu den bestehenden Rust-API-Richtlinien für die Entwicklung von HALs für Mikrocontroller gedacht. Checkliste

- Benennung
- Interoperabilität
- Vorhersehbarkeit
- GPIO

9.1.1. Checkliste HAL-Design-Muster

- **Benennung** (*Crate richtet sich nach den Namenskonventionen von Rust*)
  - ☐ Das Crate trägt einen passenden Namen. (C-CRATE-NAME)
- **Interoperabilität** (*das Crate harmoniert gut mit der Funktionalität anderer Bibliotheken*)
  - ☐ Wrapper-Typen stellen eine Destruktor-Methode bereit. (C-FREE)
  - ☐ HALs exportieren ihr Registerzugriffs-Crate erneut. (C-REEXPORT-PAC)
  - ☐ Typen implementieren die `embedded-hal`-Traits. (C-HAL-TRAITS)
- **Vorhersehbarkeit** (*Das Crate ermöglicht gut lesbaren Code, der sich so verhält, wie er aussieht*)
  - ☐ Anstelle von Extension Traits werden Konstruktoren verwendet. (C-CTOR)
- **GPIO-Schnittstellen** (*GPIO-Schnittstellen folgen einem einheitlichen Muster*)
  - ☐ Pin-Typen haben standardmäßig die Größe Null. (C-ZST-PIN)
  - ☐ Pin-Typen stellen Methoden zum Löschen von Pins und Ports bereit. (C-ERASED-PIN)
  - ☐ Der Pin-Zustand sollte als Typparameter kodiert werden. (C-PIN-STATE)

9.1.2. Benennung

9.1.2.1. Das Crate trägt einen passenden Namen (C-CRATE-NAME)

HAL-Crates sollten nach dem Chip oder der Chip-Familie benannt werden, die sie unterstützen sollen. Ihr Name sollte auf `-hal` enden, um sie von Registerzugriffs-Crates zu unterscheiden. Der Name sollte keine Unterstriche enthalten (stattdessen sind Bindestriche zu verwenden).

9.1.3. Interoperabilität

9.1.3.1. Wrapper-Typen stellen eine Destruktor-Methode bereit (C-FREE)

Jeder vom HAL bereitgestellte Wrapper-Typ, der nicht `Copy` implementiert, sollte eine `free`-Methode anbieten; diese verbraucht den Wrapper und gibt das rohe Peripheriegerät (sowie gegebenenfalls weitere Objekte), aus dem er erstellt wurde, wieder frei.

Die Methode sollte das Peripheriegerät bei Bedarf deaktivieren und zurücksetzen. Ein Aufruf von `new` mit dem durch `free` zurückgegebenen rohen Peripheriegerät darf nicht aufgrund eines unerwarteten Zustands des Geräts fehlschlagen.

Falls für die Konstruktion des HAL-Typs weitere Objekte erforderlich sind, die nicht `Copy` implementieren (zum Beispiel I/O-Pins), sollten auch diese Objekte durch `free` freigegeben und zurückgegeben werden. In diesem Fall sollte `free` ein Tupel zurückgeben.

Zum Beispiel:

```
# pub struct TIMER0;
pub struct Timer(TIMER0);

impl Timer {
    pub fn new(periph: TIMER0) -> Self {
        Self(periph)
    }

    pub fn free(self) -> TIMER0 {
        self.0
    }
}
```

9.1.3.2. HALs exportieren ihr Registerzugriffs-Crate erneut (C-REEXPORT-PAC)

HALs können auf der Basis von mit svd2rust generierten PACs oder anderen Crates implementiert werden, die einen direkten Registerzugriff ermöglichen. HALs sollten das Crate für den Registerzugriff, auf dem sie aufbauen, stets an der Wurzel ihres eigenen Crates erneut exportieren (re-exportieren).

Ein PAC sollte unter dem Namen `pac` re-exportiert werden – unabhängig vom eigentlichen Namen des Crates –, da bereits der Name der HAL verdeutlichen sollte, auf welches PAC zugegriffen wird.

9.1.3.3. Typen implementieren die `embedded-hal`-Traits (C-HAL-TRAITS)

Vom HAL bereitgestellte Typen sollten alle anwendbaren Traits implementieren, die vom `embedded-hal`-Crate bereitgestellt werden.

Für denselben Typ können mehrere Traits implementiert werden.

9.1.4. Vorhersehbarkeit

9.1.4.1. Anstelle von Extension Traits werden Konstruktoren verwendet (C-CTOR)

Alle Peripheriegeräte, denen die HAL Funktionalität hinzufügt, sollten in einen neuen Typ gekapselt werden – selbst dann, wenn für diese Funktionalität keine zusätzlichen Felder erforderlich sind.

Die Implementierung von Extension Traits für das rohe Peripheriegerät (Raw Peripheral) sollte vermieden werden.

9.1.4.2. Methoden werden dort, wo es angebracht ist, mit `#[inline]` versehen (C-INLINE)

Der Rust-Compiler führt standardmäßig kein vollständiges Inlining über Crate-Grenzen hinweg durch. Da eingebettete Anwendungen empfindlich auf unerwartete Vergrößerungen des Codes reagieren, sollte `#[inline]` verwendet werden, um den Compiler wie folgt zu steuern:

- Alle „kleinen“ Funktionen sollten mit `#[inline]` gekennzeichnet werden. Was als „klein” gilt, ist subjektiv; im Allgemeinen zählen jedoch alle Funktionen dazu, die voraussichtlich zu einer Sequenz von nur wenigen Maschinenbefehlen kompiliert werden.
- Funktionen, die sehr wahrscheinlich konstante Werte als Parameter erhalten, sollten mit `#[inline]` gekennzeichnet werden. Dies ermöglicht es dem Compiler, selbst komplexe Initialisierungslogik zur Kompilierzeit zu berechnen, sofern die Eingabewerte der Funktion bekannt sind.

9.1.5. Empfehlungen für GPIO-Schnittstellen

9.1.5.1. Pin-Typen haben standardmäßig die Größe Null (C-ZST-PIN)

Die von der HAL bereitgestellten GPIO-Schnittstellen sollten für jeden Pin jeder Schnittstelle oder jedes Ports einen dedizierten Datentyp der Größe Null bereitstellen. Dies führt zu einer kostenlosen GPIO-Abstraktion, wenn alle Pinbelegungen statisch bekannt sind.

Jede GPIO-Schnittstelle oder jeder Port sollte eine `split`-Methode implementieren, die eine Struktur mit allen Pins zurückgibt.

Beispiel:

```
pub struct PA0;
pub struct PA1;
// ...

pub struct PortA;

impl PortA {
    pub fn split(self) -> PortAPins {
        PortAPins {
            pa0: PA0,
            pa1: PA1,
            // ...
        }
    }
}

pub struct PortAPins {
    pub pa0: PA0,
    pub pa1: PA1,
    // ...
}
```

9.1.5.2. Pin-Typen stellen Methoden zum Löschen von Pins und Ports bereit (C-ERASED-PIN)

Pins sollten Methoden zur Typeliminierung (Type Erasure) bereitstellen, die ihre Eigenschaften von der Kompilierzeit in die Laufzeit verlagern und so mehr Flexibilität in Anwendungen ermöglichen.

Beispiel:

```
/// Port A, pin 0.
pub struct PA0;

impl PA0 {
    pub fn erase_pin(self) -> PA {
        PA { pin: 0 }
    }
}

/// A pin on port A.
pub struct PA {
    /// The pin number.
    pin: u8,
}

impl PA {
    pub fn erase_port(self) -> Pin {
        Pin {
            port: Port::A,
            pin: self.pin,
        }
    }
}
```

```
pub struct Pin {
    port: Port,
    pin: u8,
    // (Diese Felder koennen gepackt werden, um den Speicherbedarf zu
    // verringern.)
}

enum Port {
    A,
    B,
    C,
    D,
}
```

9.1.5.3. Der Pin-Zustand sollte als Typparameter kodiert werden (C-PIN-STATE)

Je nach Chip oder Chipfamilie können Pins als Eingang oder Ausgang mit unterschiedlichen Eigenschaften konfiguriert werden. Dieser Zustand sollte im Typsystem abgebildet werden, um eine Verwendung der Pins in einem falschen Zustand zu verhindern.

Auch zusätzliche, chip-spezifische Zustände (z. B. die Treiberstärke) lassen sich auf diese Weise mittels zusätzlicher Typparameter kodieren.

Methoden zur Änderung des Pin-Zustands sollten als `into_input` und `into_output` bereitgestellt werden.

Zusätzlich sollten Methoden wie `with_{input,output}_state` angeboten werden, die einen Pin vorübergehend in einen anderen Zustand versetzen, ohne ihn zu verschieben (d. h. ohne Eigentumsübergang).

Für jeden Pin-Typ sollten die folgenden Methoden bereitgestellt werden (das heißt, sowohl „erased“ als auch „non-erased” Pin-Typen müssen dieselbe API bieten):

- `pub fn into_input<N: InputState>(self, input: N) -> Pin<N>`
- `pub fn into_output<N: OutputState>(self, output: N) -> Pin<N>`
- ```
  pub fn with_input_state<N: InputState, R>(
      &mut self,
      input: N,
      f: impl FnOnce(&mut PA1<N>) -> R,
  ) -> R
  ```
- ```
  pub fn with_output_state<N: OutputState, R>(
      &mut self,
      output: N,
      f: impl FnOnce(&mut PA1<N>) -> R,
  ) -> R
  ```

Der Pin-Zustand sollte durch „Sealed Traits“ eingeschränkt sein. Nutzer des HAL sollten keinen eigenen Zustand hinzufügen müssen. Die Traits können HAL-spezifische Methoden bereitstellen, die für die Implementierung der Pin-Zustands-API erforderlich sind.

Beispiel:

```
# use std::marker::PhantomData;
mod sealed {
    pub trait Sealed {}
}

pub trait PinState: sealed::Sealed {}
pub trait OutputState: sealed::Sealed {}
```

```rust
pub trait InputState: sealed::Sealed {
    // ...
}

pub struct Output<S: OutputState> {
    _p: PhantomData<S>,
}

impl<S: OutputState> PinState for Output<S> {}
impl<S: OutputState> sealed::Sealed for Output<S> {}

pub struct PushPull;
pub struct OpenDrain;

impl OutputState for PushPull {}
impl OutputState for OpenDrain {}
impl sealed::Sealed for PushPull {}
impl sealed::Sealed for OpenDrain {}

pub struct Input<S: InputState> {
    _p: PhantomData<S>,
}

impl<S: InputState> PinState for Input<S> {}
impl<S: InputState> sealed::Sealed for Input<S> {}

pub struct Floating;
pub struct PullUp;
pub struct PullDown;

impl InputState for Floating {}
impl InputState for PullUp {}
impl InputState for PullDown {}
impl sealed::Sealed for Floating {}
impl sealed::Sealed for PullUp {}
impl sealed::Sealed for PullDown {}

pub struct PA1<S: PinState> {
    _p: PhantomData<S>,
}

impl<S: PinState> PA1<S> {
    pub fn into_input<N: InputState>(self, input: N) -> PA1<Input<N>> {
        todo!()
    }

    pub fn into_output<N: OutputState>(self, output: N) -> PA1<Output<N>> {
        todo!()
    }

    pub fn with_input_state<N: InputState, R>(
        &mut self,
        input: N,
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
```

```
    }

    pub fn with_output_state<N: OutputState, R>(
        &mut self,
        output: N,
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
    }
}

// Dasselbe gilt fuer `PA` und `Pin` sowie andere Pin-Typen.
```

10. Tipps für Embedded-C-Entwickler

Dieses Kapitel versammelt eine Reihe von Tipps, die für erfahrene Embedded-C-Entwickler nützlich sein können, die mit der Programmierung in Rust beginnen möchten. Dabei wird insbesondere hervorgehoben, wie sich Dinge, die man aus C bereits kennt, in Rust unterscheiden.

10.0.1. Präprozessor

In der Embedded-C-Programmierung ist es sehr üblich, den Präprozessor für eine Vielzahl von Zwecken zu nutzen, wie zum Beispiel:

- Auswahl von Codeblöcken zur Kompilierzeit mittels `#ifdef`
- Zur Kompilierzeit festgelegte Array-Größen und Berechnungen
- Makros zur Vereinfachung häufiger Muster (um den Overhead von Funktionsaufrufen zu vermeiden)

In Rust gibt es keinen Präprozessor, weshalb viele dieser Anwendungsfälle anders gelöst werden. Im weiteren Verlauf dieses Abschnitts behandeln wir verschiedene Alternativen zur Verwendung des Präprozessors.

10.0.1.1. Code-Auswahl zur Kompilierzeit

Das Äquivalent zu `#ifdef ... #endif` in Rust sind Cargo-Features. Diese sind etwas formaler als der C-Präprozessor: Alle möglichen Features werden pro Crate explizit aufgelistet und können entweder aktiviert oder deaktiviert sein. Features werden eingeschaltet, sobald man ein Crate als Abhängigkeit angibt; sie sind zudem additiv: Wenn irgendein Crate im Abhängigkeitsbaum ein Feature für ein anderes Crate aktiviert, ist dieses Feature für alle Nutzer jenes Crates aktiviert.

Angenommen, du hast ein Crate, das eine Bibliothek von Grundbausteinen für die Signalverarbeitung bereitstellt. Jeder dieser Bausteine könnte zusätzliche Kompilierzeit beanspruchen oder eine umfangreiche Tabelle von Konstanten definieren, die du gerne vermeiden möchtest. Du könntest für jede Komponente in deiner `Cargo.toml` ein Cargo-Feature definieren:

```
[features]
FIR = []
IIR = []
```

Verwenden Sie dann in Ihrem Code `#[cfg(feature="FIR")]`, um zu steuern, was einbezogen wird.

```
/// In deiner lib.rs auf oberster Ebene

#[cfg(feature="FIR")]
pub mod fir;

#[cfg(feature="IIR")]
pub mod iir;
```

Sie können Codeblöcke analog dazu nur dann einfügen, wenn eine Funktion *nicht* aktiviert ist oder wenn eine beliebige Kombination von Funktionen aktiviert oder deaktiviert ist.

Darüber hinaus bietet Rust eine Reihe automatisch gesetzter Bedingungen, die Sie verwenden können, beispielsweise `target_arch`, um je nach Architektur unterschiedlichen Code auszuwählen. Ausführliche Informationen zur bedingten Kompilierung finden Sie im Kapitel Bedingte Kompilierung der Rust-Referenz.

Die bedingte Kompilierung bezieht sich nur auf die unmittelbar folgende Anweisung oder den folgenden Block. Kann im aktuellen Gültigkeitsbereich kein Block verwendet werden, muss das `cfg`-Attribut mehrfach eingesetzt werden. Es sei darauf hingewiesen, dass es meist besser ist, den gesamten Code einzubeziehen und den Compiler bei der Optimierung nicht mehr benötigten Code („Dead Code“)

entfernen zu lassen: Dies ist für Sie und Ihre Nutzer einfacher, und der Compiler leistet im Allgemeinen gute Arbeit beim Entfernen von ungenutztem Code.

10.0.1.2. Größen und Berechnungen zur Kompilierzeit

Rust unterstützt `const fn` – Funktionen, die garantiert zur Kompilierzeit ausgewertet werden können und sich daher überall dort einsetzen lassen, wo Konstanten erforderlich sind, etwa bei der Größe von Arrays. Dies lässt sich mit den zuvor genannten Funktionen kombinieren, zum Beispiel:

```
const fn array_size() -> usize {
    #[cfg(feature="use_more_ram")]
    { 1024 }
    #[cfg(not(feature="use_more_ram"))]
    { 128 }
}

static BUF: [u32; array_size()] = [0u32; array_size()];
```

Diese Funktionen sind seit Version 1.31 in Stable Rust verfügbar, weshalb die Dokumentation noch spärlich ist. Auch der für `const fn` verfügbare Funktionsumfang ist zum Zeitpunkt der Erstellung dieses Textes stark eingeschränkt; es ist jedoch zu erwarten, dass die in einer `const fn` zulässigen Operationen in künftigen Rust-Versionen erweitert werden.

10.0.1.3. Makros

Rust bietet ein äußerst leistungsfähiges Makrosystem. Während der C-Präprozessor fast direkt auf dem Text des Quellcodes arbeitet, operiert das Rust-Makrosystem auf einer höheren Ebene. Es gibt zwei Arten von Rust-Makros: *Macros by Example* (Makros nach Muster) und *prozedurale Makros*. Erstere sind einfacher und weiter verbreitet; sie sehen wie Funktionsaufrufe aus und können zu einem vollständigen Ausdruck, einer Anweisung, einem Element oder einem Muster expandiert werden. Prozedurale Makros sind komplexer, ermöglichen jedoch äußerst leistungsfähige Erweiterungen der Sprache Rust: Sie können beliebige Rust-Syntax in neue Rust-Syntax umwandeln.

Wenn Sie normalerweise ein C-Präprozessor-Makro verwenden würden, sollten Sie im Allgemeinen prüfen, ob stattdessen ein „Macro-by-Example“ (Makro-durch-Beispiel) die Aufgabe erfüllen kann. Solche Makros lassen sich in Ihrem Crate definieren und sowohl intern verwenden als auch für andere Nutzer exportieren. Beachten Sie jedoch, dass sie zu vollständigen Ausdrücken, Anweisungen, Elementen (Items) oder Mustern expandieren müssen; bestimmte Anwendungsfälle von C-Präprozessor-Makros funktionieren daher nicht – etwa Makros, die nur einen Teil eines Variablennamens oder unvollständige Listenelemente erzeugen.

Wie bei Cargo-Features lohnt es sich auch hier zu überlegen, ob das Makro überhaupt erforderlich ist. Oftmals ist eine gewöhnliche Funktion leichter verständlich und wird vom Compiler zu demselben Maschinencode expandiert (geinlined) wie ein Makro. Die Attribute `#[inline]` und `#[inline(always)]`-Attribute bieten Ihnen zusätzliche Kontrolle über diesen Vorgang, wobei jedoch Vorsicht geboten ist: Der Compiler inlined Funktionen aus demselben Crate ohnehin automatisch, wenn dies sinnvoll ist; ein erzwungenes Inlining in ungeeigneten Fällen kann die Leistung sogar verschlechtern.

Eine Erläuterung des gesamten Rust-Makrosystems würde den Rahmen dieser Tipps-Seite sprengen; für alle Einzelheiten sei daher auf die Rust-Dokumentation verwiesen.

10.0.2. Build System

Die meisten Rust-Crates werden mit Cargo erstellt (auch wenn dies nicht zwingend erforderlich ist). Cargo löst dabei viele der schwierigen Probleme, die mit herkömmlichen Build-Systemen verbunden sind. Dennoch kann es sinnvoll sein, den Build-Prozess individuell anzupassen. Hierfür stellt Cargo

`build.rs`-Skripte bereit. Dabei handelt es sich um Rust-Skripte, die bei Bedarf mit dem Cargo-Build-System interagieren können.

Zu den häufigen Anwendungsfällen für Build-Skripte gehören:

- Informationen zum Build-Zeitpunkt bereitstellen, zum Beispiel durch das statische Einbetten des Build-Datums oder des Git-Commit-Hashs in die ausführbare Datei.
- Linker-Skripte zur Build-Zeit generieren, abhängig von ausgewählten Funktionen oder anderer Logik.
- die Cargo-Build-Konfiguration ändern
- Zusätzliche statische Bibliotheken für den Linkvorgang hinzufügen

Derzeit gibt es keine Unterstützung für Post-Build-Skripte, wie man sie üblicherweise für Aufgaben wie die automatische Erstellung von Binärdateien aus den Build-Objekten oder die Ausgabe von Build-Informationen verwendet hat.

10.0.2.1. Cross-Kompilierung

Die Verwendung von Cargo als Build-System vereinfacht auch die Cross-Kompilierung. In den meisten Fällen genügt es, Cargo die Option `--target thumbv6m-none-eabi` mitzugeben und die entsprechende ausführbare Datei unter `target/thumbv6m-none-eabi/debug/myapp` zu finden.

Für Plattformen, die nicht nativ von Rust unterstützt werden, müssen Sie `libcore` für das jeweilige Zielsystem selbst kompilieren. Auf solchen Plattformen lässt sich Xargo als Ersatz für Cargo verwenden, da es `libcore` automatisch für Sie erstellt.

10.0.3. Iteratoren vs. Array-Zugriff

In C sind Sie es wahrscheinlich gewohnt, direkt über den Index auf Arrays zuzugreifen:

```
int16_t arr[16];
int i;
for(i=0; i<sizeof(arr)/sizeof(arr[0]); i++) {
    process(arr[i]);
}
```

In Rust ist dies ein Anti-Muster: Indexzugriffe können langsamer sein (da eine Bereichsprüfung erforderlich ist) und verschiedene Compiler-Optimierungen verhindern. Dieser Unterschied ist wichtig und sollte wiederholt werden: Rust prüft bei manueller Array-Indizierung auf Zugriffe außerhalb der Grenzen, um Speichersicherheit zu gewährleisten, während C problemlos auf Bereiche außerhalb des Arrays zugreift.

Verwenden Sie stattdessen Iteratoren.

```
let arr = [0u16; 16];
for element in arr.iter() {
    process(*element);
}
```

Iteratoren bieten eine Vielzahl leistungsstarker Funktionen, die Sie in C manuell implementieren müssten, wie z. B. Verkettung, Zipping, Aufzählung, Minimum- und Maximumsuche, Summierung und vieles mehr. Iteratormethoden lassen sich ebenfalls verketten, was zu sehr lesbarem Code für die Datenverarbeitung führt.

Weitere Informationen finden Sie unter Iteratoren im Buch und Iterator-Dokumentation.

10.0.4. Referenzen vs. Zeiger

In Rust gibt es zwar Zeiger (sogenannte *Raw Pointer*), diese werden jedoch nur unter bestimmten Umständen verwendet, da ihre Dereferenzierung stets als `unsafe` gilt – Rust kann nämlich nicht die üblichen Garantien darüber geben, was sich hinter dem Zeiger befindet.

Meistens verwenden wir stattdessen *Referenzen* (gekennzeichnet durch das Symbol &) oder *veränderbare Referenzen* (gekennzeichnet durch `&mut`). Referenzen verhalten sich ähnlich wie Zeiger, da sie dereferenziert werden können, um auf die zugrundeliegenden Werte zuzugreifen; sie sind jedoch ein wesentlicher Bestandteil des Ownership-Systems von Rust: Rust stellt strikt sicher, dass zu jedem Zeitpunkt entweder nur eine veränderbare Referenz *oder* mehrere unveränderbare Referenzen auf denselben Wert existieren dürfen.

In der Praxis bedeutet dies, dass man genauer abwägen muss, ob man veränderbaren Zugriff auf Daten benötigt: Während in C Veränderbarkeit der Standard ist und `const` explizit angegeben werden muss, verhält es sich in Rust genau umgekehrt.

Eine Situation, in der man dennoch „Raw Pointer“ (rohe Zeiger) verwenden könnte, ist die direkte Interaktion mit Hardware (zum Beispiel beim Schreiben eines Zeigers auf einen Puffer in ein DMA-Peripherieregister); zudem kommen sie im Hintergrund bei allen Crates für den Peripheriezugriff zum Einsatz, um das Lesen und Schreiben speicherabgebildeter Register (memory-mapped registers) zu ermöglichen.

10.0.5. Volatiler (unsicherer) Zugriff

In C können einzelne Variablen mit `volatile` gekennzeichnet werden, um dem Compiler mitzuteilen, dass sich ihr Wert zwischen den Zugriffen ändern kann. Volatile Variablen werden häufig in eingebetteten Systemen für im Speicher abgebildete Register verwendet.

In Rust verwenden wir anstelle der Kennzeichnung einer Variable als `volatile` spezielle Methoden für den Zugriff auf unsichere Daten: `core::ptr::read_volatile` und `core::ptr::write_volatile`. Diese Methoden akzeptieren einen `*const T` oder einen `*mut T` (Rohzeiger, wie oben beschrieben) und führen einen flüchtigen Lese- bzw. Schreibvorgang durch.

In C könnten Sie zum Beispiel schreiben:

```
volatile bool signalled = false;

void ISR() {
    // Signalisieren, dass der Interrupt aufgetreten ist
    signalled = true;
}

void driver() {
    while(true) {
        // Schlafen bis zum Signal
        while(!signalled) { WFI(); }
        // Signalisierten Indikator zuruecksetzen
        signalled = false;
        // Fuehren Sie eine Aufgabe aus, die auf den Interrupt gewartet hat
        run_task();
    }
}
```

Das Äquivalent in Rust würde bei jedem Zugriff volatile Methoden verwenden:

```
static mut SIGNALLED: bool = false;
```

```
#[interrupt]
fn ISR() {
    // Signalisieren, dass eine Unterbrechung aufgetreten ist
    // (Im realen Code sollten Sie eine hoeherwertige primitive Datenstruktur,
    // wie z. B. einen atomaren Datentyp, in Betracht ziehen).
    unsafe { core::ptr::write_volatile(&mut SIGNALLED, true) };
}

fn driver() {
    loop {
        // Schlafen bis zum Signal
        while unsafe { !core::ptr::read_volatile(&SIGNALLED) } {}
        // Signalisierten Indikator zuruecksetzen
        unsafe { core::ptr::write_volatile(&mut SIGNALLED, false) };
        // Fuehren Sie eine Aufgabe aus, die auf den Interrupt gewartet hat
        run_task();
    }
}
```

Am Codebeispiel sind einige Dinge erwähnenswert:

- Wir können `&mut SIGNALLED` an die Funktion übergeben, die `*mut T` erwartet, da `&mut T` automatisch in `*mut T` umgewandelt wird (und das Gleiche gilt für `*const T`).
- Für die Methoden `read_volatile` und `write_volatile` benötigen wir `unsafe`-Blöcke, da es sich um `unsafe`-Funktionen handelt. Es liegt in der Verantwortung des Programmierers, für eine sichere Verwendung zu sorgen; weitere Einzelheiten sind der Dokumentation der jeweiligen Methoden zu entnehmen.

Es ist selten erforderlich, diese Funktionen direkt im eigenen Code zu verwenden, da sie üblicherweise von höherwertigen Bibliotheken für Sie übernommen werden. Bei speicherabgebildeten Peripheriekomponenten implementieren die entsprechenden „Peripheral Access Crates“ den volatilen Zugriff automatisch, während für Nebenläufigkeits-Primitive bessere Abstraktionen zur Verfügung stehen (siehe das Kapitel Nebenläufigkeit).

10.0.6. Gepackte und ausgerichtete Datentypen

In der Embedded-Programmierung mit C ist es üblich, dem Compiler vorzugeben, dass eine Variable eine bestimmte Ausrichtung (Alignment) aufweisen oder eine Struktur „gepackt“ (packed) statt ausgerichtet sein muss – meist, um spezifische Hardware- oder Protokollanforderungen zu erfüllen.

In Rust wird dies über das `repr`-Attribut an einer `struct` oder `union` gesteuert. Die Standarddarstellung macht keine Zusagen über das Speicherlayout und sollte daher nicht für Code verwendet werden, der mit Hardware oder C interagiert. Der Compiler kann die Reihenfolge der Strukturmitglieder ändern oder Füllbytes (Padding) einfügen, und das Verhalten kann sich in zukünftigen Rust-Versionen ändern.

```
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7ffecb3511d0 0x7ffecb3511d4 0x7ffecb3511d2
```

```
// Die Reihenfolge der Noten wurde auf x, z, y geaendert, um die Packdichte zu
// verbessern.
```

Um Layouts zu gewährleisten, die mit C interoperabel sind, verwenden Sie `repr(C)`:

```
#[repr(C)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7fffd0d84c60 0x7fffd0d84c62 0x7fffd0d84c64
// Die Reihenfolge bleibt erhalten und das Layout veraendert sich im Laufe der
// Zeit nicht.
// `z` ist auf Zwei-Byte-Grenzen ausgerichtet, sodass sich zwischen `y` und `z`
// ein Padding-Byte befindet.
```

Um eine kompakte Darstellung zu gewährleisten, verwenden Sie `repr(packed)`:

```
#[repr(packed)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    // Referenzen muessen stets korrekt ausgerichtet sein; um also die
    // Adressen der Struct-Felder zu ueberpruefen, verwenden wir
    // `std::ptr::addr_of!()`, um einen Rohzeiger (Raw Pointer) zu erhalten,
    // anstatt einfach `&v.x` auszugeben.
    let px = std::ptr::addr_of!(v.x);
    let py = std::ptr::addr_of!(v.y);
    let pz = std::ptr::addr_of!(v.z);
    println!("{:p} {:p} {:p}", px, py, pz);
}

// 0x7ffd33598490 0x7ffd33598492 0x7ffd33598493
// Zwischen `y` und `z` wurde kein Padding eingefuegt, daher ist `z` nun
// nicht ausgerichtet.
```

Beachten Sie, dass die Verwendung von `repr(packed)` die Ausrichtung des Typs ebenfalls auf `1` setzt.

Um eine bestimmte Ausrichtung festzulegen, verwenden Sie schließlich `repr(align(n))`, wobei `n` die Anzahl der auszurichtenden Bytes ist (und eine Zweierpotenz sein muss).

```
#[repr(C)]
#[repr(align(4096))]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
```

```
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    let u = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
    println!("{:p} {:p} {:p}", &u.x, &u.y, &u.z);
}

// 0x7ffec909a000 0x7ffec909a002 0x7ffec909a004
// 0x7ffec909b000 0x7ffec909b002 0x7ffec909b004
// Die beiden Instanzen `u` und `v` wurden an 4096-Byte-Grenzen ausgerichtet,
// was an den `000` am Ende ihrer Adressen erkennbar ist.
```

Beachten Sie, dass wir `repr(C)` mit `repr(align(n))` kombinieren können, um ein ausgerichtetes und C-kompatibles Layout zu erhalten. Die Kombination von `repr(align(n))` mit `repr(packed)` ist nicht zulässig, da `repr(packed)` die Ausrichtung auf `1` setzt. Ebenso ist es nicht zulässig, dass ein `repr(packed)`-Typ einen `repr(align(n))`-Typ enthält.

Weitere Details zu Typ-Layouts finden Sie im Kapitel Typ-Layout der Rust-Referenz.

10.0.7. Weitere Ressourcen

- In diesem Buch:
  - Ein bisschen C zu Ihrem Rust
  - Ein bisschen Rust zu Ihrem C
- Die Rust-Embedded-FAQs
- Rust-Pointer für C-Programmierer
- Früher habe ich Pointer verwendet – und jetzt?

11. Interoperabilität

Die Interoperabilität zwischen Rust- und C-Code hängt stets von der Umwandlung von Daten zwischen den beiden Sprachen ab. Hierfür gibt es in der Standardbibliothek (`stdlib`) ein spezielles Modul namens `std::ffi`.

`std::ffi` stellt Typdefinitionen für primitive C-Datentypen bereit, wie etwa `char`, `int` und `long`. Zudem bietet es Hilfsmittel zur Konvertierung komplexerer Typen wie Strings, indem es sowohl `&str` als auch `String` auf C-Typen abbildet, die sich einfacher und sicherer handhaben lassen.

Seit Rust 1.30 stehen die Funktionalitäten von `std::ffi` wahlweise in `core::ffi` oder `alloc::ffi` zur Verfügung, je nachdem, ob Speicherreservierungen erforderlich sind oder nicht. Auch die Crates `cty` und `cstr_core` bieten vergleichbare Funktionalitäten an.

Rusttyp	Dazwischenliegend	C-Typ
`String`	`CString`	`char *`
`&str`	`CStr`	`const char *`
`()`	`c_void`	`void`
u32 or u64		

, `c_uint`, `unsigned int`, tr((en: [etc], de: [usw.], ja: [他],)), [...], [...],))

Ein Wert eines primitiven C-Datentyps kann als Wert des entsprechenden Rust-Datentyps verwendet werden und umgekehrt, da der erstere lediglich ein Typalias des letzteren ist. Beispielsweise lässt sich der folgende Code auf Plattformen kompilieren, auf denen `unsigned int` 32 Bit lang ist.

```
fn foo(num: u32) {
    let c_num: c_uint = num;
    let r_num: u32 = c_num;
}
```

11.0.1. Interoperabilität mit anderen Build-Systemen

Eine häufige Voraussetzung für die Einbindung von Rust in Embedded-Projekte ist die Kombination von Cargo mit Ihrem bestehenden Build-System, beispielsweise Make oder CMake.

Wir sammeln Beispiele und Anwendungsfälle hierzu in unserem Problem-Verfolgungswerkzeug unter Issue #61.

11.0.2. Interoperabilität mit RTOS

Die Integration von Rust in ein RTOS wie FreeRTOS oder ChibiOS befindet sich noch in der Entwicklung; insbesondere der Aufruf von RTOS-Funktionen aus Rust heraus kann sich als schwierig erweisen.

Derzeit unterstützen die folgenden Projekte öffentlich die Interoperabilität zwischen Rust und RTOS:

- Zephyr-Projekt

Wir sammeln hierfür Beispiele und Anwendungsfälle in unserem Problem-Verfolgungswerkzeug unter Issue #62.

11.1. Ein bisschen C zu Ihrem Rust

Die Verwendung von C oder C++ innerhalb eines Rust-Projekts umfasst zwei wesentliche Aspekte:

- Einkapselung der bereitgestellten C-API für die Verwendung mit Rust
- Erstellen Ihres C- oder C++-Codes zur Integration mit dem Rust-Code

Da C++ über keine stabile ABI verfügt, auf die der Rust-Compiler abzielen könnte, wird empfohlen, bei der Kombination von Rust mit C oder C++ die `C`-ABI zu verwenden.

11.1.1. Definition der Schnittstelle

Bevor C- oder C++-Code aus Rust heraus verwendet werden kann, muss (in Rust) definiert werden, welche Datentypen und Funktionssignaturen im verlinkten Code vorhanden sind. In C oder C++ würde man eine Header-Datei (`.h` oder `.hpp`) einbinden, die diese Daten definiert. In Rust ist es erforderlich, diese Definitionen entweder manuell nach Rust zu übertragen oder ein Werkzeug zu ihrer Generierung zu verwenden.

Zunächst behandeln wir die manuelle Übertragung dieser Definitionen von C/C++ nach Rust.

11.1.1.1. Einbinden von C-Funktionen und -Datentypen

Typischerweise stellen in C oder C++ geschriebene Bibliotheken eine Header-Datei bereit, die alle in öffentlichen Schnittstellen verwendeten Typen und Funktionen definiert. Eine Beispieldatei könnte wie folgt aussehen:

```c
/* File: cool.h */
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c, CoolStruct* cs);
```

Nach Rust übertragen, sähe diese Schnittstelle folgendermaßen aus:

```rust
/* File: cool_bindings.rs */
#[repr(C)]
pub struct CoolStruct {
    pub x: cty::c_int,
    pub y: cty::c_int,
}

extern "C" {
    pub fn cool_function(
        i: cty::c_int,
        c: cty::c_char,
        cs: *mut CoolStruct
    );
}
```

Schauen wir uns diese Definition Schritt für Schritt an, um die einzelnen Bestandteile zu erläutern.

```rust
#[repr(C)]
pub struct CoolStruct { ... }
```

Standardmäßig garantiert Rust weder die Reihenfolge noch das Padding oder die Größe der in einer `struct` enthaltenen Daten. Um die Kompatibilität mit C-Code zu gewährleisten, verwenden wir das Attribut `#[repr(C)]`; dieses weist den Rust-Compiler an, für die Anordnung der Daten innerhalb der Struktur stets dieselben Regeln wie C anzuwenden.

```rust
pub x: cty::c_int,
pub y: cty::c_int,
```

Aufgrund der Flexibilität, wie C oder C++ ein „int“ oder „char” definiert, wird empfohlen, in „cty“ definierte primitive Datentypen zu verwenden, die Typen von C auf Typen in Rust abbilden.

```rust
extern "C" { pub fn cool_function( ... ); }
```

Diese Anweisung definiert die Signatur einer Funktion namens `cool_function`, die die C-ABI verwendet. Da die Signatur ohne den Funktionsrumpf definiert wird, muss die eigentliche Funktionsdefinition an anderer Stelle bereitgestellt oder aus einer statischen Bibliothek in die endgültige Bibliothek bzw. das fertige Binärprogramm eingebunden werden.

```
    i: cty::c_int,
    c: cty::c_char,
    cs: *mut CoolStruct
```

Ähnlich wie bei unserem obigen Datentyp definieren wir die Datentypen der Funktionsargumente mithilfe von C-kompatiblen Definitionen. Der Übersichtlichkeit halber behalten wir zudem die ursprünglichen Argumentnamen bei.

Hier begegnet uns ein neuer Typ: `*mut CoolStruct`. Da C das Konzept der Rust-Referenzen (die etwa so aussehen: `&mut CoolStruct`) nicht kennt, verwenden wir stattdessen einen sogenannten „Raw Pointer" (Rohzeiger). Da das Dereferenzieren dieses Zeigers als `unsafe` gilt und es sich tatsächlich um einen `null`-Zeiger handeln kann, ist bei der Interaktion mit C- oder C++-Code besondere Sorgfalt geboten, um die für Rust typischen Garantien zu wahren.

11.1.1.2. Automatische Generierung der Schnittstelle

Anstatt diese Schnittstellen manuell zu erstellen – was mühsam und fehleranfällig sein kann –, gibt es ein Werkzeug namens bindgen, das diese Konvertierungen automatisch durchführt. Hinweise zur Verwendung von bindgen finden Sie im bindgen-Benutzerhandbuch; der typische Ablauf sieht jedoch folgendermaßen aus:

1. Sammeln Sie alle C- oder C++-Header, die Schnittstellen oder Datentypen definieren, die Sie mit Rust verwenden möchten.
2. Erstelle eine Datei namens `bindings.h`, die mittels `#include "..."` jede der Dateien einbindet, die du in Schritt eins zusammengetragen hast.
3. Übergeben Sie diese `bindings.h`-Datei zusammen mit den für die Kompilierung Ihres Codes verwendeten Flags an `bindgen`. Tipp: Verwenden Sie `Builder.ctypes_prefix("cty")` / `--ctypes-prefix=cty` und `Builder.use_core()` / `--use-core`, um den generierten Code `#![no_std]`-kompatibel zu machen.
4. `bindgen` gibt den generierten Rust-Code direkt im Terminal aus. Diese Ausgabe lässt sich in eine Datei Ihres Projekts umleiten, beispielsweise `bindings.rs`. Sie können diese Datei in Ihrem Rust-Projekt verwenden, um mit C/C++-Code zu interagieren, der als externe Bibliothek kompiliert und gelinkt wurde. Tipp: Vergessen Sie nicht, das `cty`-Crate zu verwenden, falls die Typen in den generierten Bindings das Präfix `cty` aufweisen.

11.1.2. Erstellen Ihres C/C++-Codes

Da der Rust-Compiler nicht direkt weiß, wie man C- oder C++-Code (oder Code einer anderen Sprache mit C-Schnittstelle) kompiliert, muss der Nicht-Rust-Code vorab kompiliert werden.

Bei Embedded-Projekten bedeutet dies meist, dass der C/C++-Code zu einem statischen Archiv (z. B. `cool-library.a`) kompiliert wird, welches dann im abschließenden Link-Schritt mit dem Rust-Code zusammengeführt werden kann.

Wenn die gewünschte Bibliothek bereits als statisches Archiv vorliegt, ist eine erneute Kompilierung des Codes nicht erforderlich. Es genügt, die bereitgestellte Header-Datei für die Schnittstelle wie oben beschrieben umzuwandeln und das statische Archiv beim Kompilieren bzw. Linken einzubinden.

Liegt der Code als Quellcode-Projekt vor, muss der C/C++-Code in eine statische Bibliothek kompiliert werden. Dies kann entweder durch Aufruf des vorhandenen Build-Systems (z. B. `make`, `CMake` usw.)

oder durch Portierung der erforderlichen Kompilierungsschritte auf das sogenannte `cc`-Crate erfolgen. Für beide Vorgehensweisen ist die Verwendung eines `build.rs`-Skripts erforderlich.

11.1.2.1. Rust-`build.rs`-Build-Skripte

Ein `build.rs`-Skript ist eine in Rust-Syntax verfasste Datei, die auf dem Kompilierrechner ausgeführt wird – und zwar *nachdem* die Abhängigkeiten Ihres Projekts erstellt wurden, aber *bevor* Ihr Projekt selbst kompiliert wird.

Die vollständige Referenz finden Sie hier. `build.rs`-Skripte eignen sich beispielsweise zur Code-Generierung (etwa mittels bindgen), zum Aufruf externer Build-Systeme wie `Make` oder zur direkten Kompilierung von C/C++-Code unter Verwendung des `cc`-Crates.

11.1.2.2. Auslösen externer Build-Systeme

Bei Projekten, die komplexe externe Projekte oder Build-Systeme einbinden, ist es oft am einfachsten, `std::process::Command` zu verwenden, um andere Build-Systeme aufzurufen (sogenanntes „Shelling-out“). Dabei navigieren Sie über relative Pfade, führen einen festen Befehl aus (wie etwa `make library`) und kopieren anschließend die erzeugte statische Bibliothek an den entsprechenden Ort im `target`-Build-Verzeichnis.

Auch wenn Ihre Crate für eine eingebettete Plattform ohne Standardbibliothek (`no_std`) gedacht ist, wird die `build.rs` ausschließlich auf dem Rechner ausgeführt, der die Crate kompiliert. Das bedeutet, dass Sie beliebige Rust-Crates verwenden können, die auf Ihrem Kompilier-Host lauffähig sind.

11.1.2.3. Kompilieren von C/C++-Code mit dem `cc`-Crate

Bei Projekten mit geringen Abhängigkeiten oder überschaubarer Komplexität – oder wenn es schwierig ist, das Build-System so anzupassen, dass eine statische Bibliothek (statt einer fertigen Binärdatei oder eines ausführbaren Programms) erzeugt wird – kann es einfacher sein, stattdessen das cc-Crate zu verwenden; dieses bietet eine idiomatische Rust-Schnittstelle zu dem vom Host bereitgestellten Compiler.

Im einfachsten Fall, bei dem eine einzelne C-Datei als Abhängigkeit für eine statische Bibliothek kompiliert wird, sähe ein Beispiel für ein `build.rs`-Skript, das das cc-Crate verwendet, folgendermaßen aus:

```
fn main() {
    cc::Build::new()
        .file("src/foo.c")
        .compile("foo");
}
```

Die Datei `build.rs` befindet sich im Wurzelverzeichnis des Pakets. `cargo build` kompiliert und führt sie dann vor dem eigentlichen Build-Vorgang des Pakets aus. Dabei wird ein statisches Archiv namens `libfoo.a` erstellt und im Verzeichnis `target` abgelegt.

11.2. Ein bisschen Rust zu Ihrem C

Die Verwendung von Rust-Code in einem C- oder C++-Projekt besteht größtenteils aus zwei Teilen.

- Erstellung einer C-kompatiblen API in Rust
- Einbindung Ihres Rust-Projekts in ein externes Build-System

Abgesehen von `cargo` und `meson` bieten die meisten Build-Systeme keine native Rust-Unterstützung. Daher fährst du höchstwahrscheinlich am besten damit, einfach `cargo` für die Kompilierung deines Crates und aller Abhängigkeiten zu verwenden.

11.2.1. Ein Projekt einrichten

Erstellen Sie wie gewohnt ein neues `cargo`-Projekt.

Es gibt Flags, mit denen man `cargo` anweisen kann, eine Systembibliothek anstelle des üblichen Rust-Ziels zu erstellen. Auf diese Weise können Sie auch einen abweichenden Namen für die Bibliothek festlegen, falls dieser sich von dem des restlichen Crates unterscheiden soll.

```
[lib]
name = "your_crate"
crate-type = ["cdylib"]      # Erstellt eine dynamische Bibliothek
# crate-type = ["staticlib"] # Erstellt eine statische Bibliothek
```

11.2.2. Erstellung einer C-API

Da C++ über keine stabile ABI verfügt, auf die der Rust-Compiler abzielen könnte, nutzen wir C für die Interoperabilität zwischen verschiedenen Sprachen. Dies gilt auch für die Verwendung von Rust innerhalb von C- und C++-Code.

11.2.2.1. `#[no_mangle]`

Der Rust-Compiler verarbeitet Symbolnamen anders als Linker für nativen Code erwarten. Daher muss jeder Funktion, die Rust zur Verwendung außerhalb von Rust exportiert, mitgeteilt werden, dass der Compiler sie nicht verändern soll.

11.2.2.2. `extern "C"`

Standardmäßig verwendet jede in Rust geschriebene Funktion die Rust-ABI (die ebenfalls nicht stabilisiert ist). Beim Erstellen von nach außen gerichteten FFI-APIs muss der Compiler daher angewiesen werden, die System-ABI zu verwenden.

Je nach Plattform möchten Sie möglicherweise eine bestimmte ABI-Version anvisieren; diese sind hier dokumentiert.

Wenn man diese Teile zusammensetzt, erhält man eine Funktion, die ungefähr so aussieht.

```
#[no_mangle]
pub extern "C" fn rust_function() {

}
```

Genau wie bei der Verwendung von C-Code in Ihrem Rust-Projekt müssen Sie nun Daten in eine Form umwandeln – und aus dieser zurück –, die der Rest der Anwendung versteht.

11.2.3. Verknüpfung und übergeordneter Projektkontext

Damit ist die eine Hälfte des Problems gelöst. Wie verwendet man das nun?

Das hängt stark von Ihrem Projekt bzw. Ihrem Build-System ab.

`cargo` erstellt – je nach Plattform und Einstellungen – eine Datei namens `my_lib.so`, `my_lib.dll` oder `my_lib.a`. Diese Bibliothek kann einfach von Ihrem Build-System eingebunden (gelinkt) werden.

Um jedoch eine Rust-Funktion aus C heraus aufzurufen, ist eine Header-Datei erforderlich, in der die Funktionssignaturen deklariert werden.

Für jede Funktion in Ihrer Rust-FFI-API muss eine entsprechende Deklaration im Header vorhanden sein.

```rust
#[no_mangle]
pub extern "C" fn rust_function() {}
```

würde dann werden

```
void rust_function();
```

usw.

Es gibt ein Werkzeug zur Automatisierung dieses Prozesses namens cbindgen, das Ihren Rust-Code analysiert und daraus Header-Dateien für Ihre C- und C++-Projekte generiert.

An diesem Punkt ist die Verwendung der Rust-Funktionen aus C heraus so einfach, wie den Header einzubinden und die Funktionen aufzurufen!

```
#include "my-rust-project.h"
rust_function();
```

12. Unsortierte Themen

12.1. Optimierungen: Der Kompromiss zwischen Geschwindigkeit und Größe

Jeder möchte, dass sein Programm extrem schnell und extrem klein ist, doch meist lassen sich nicht beide Eigenschaften zugleich verwirklichen. Dieser Abschnitt behandelt die verschiedenen Optimierungsstufen von `rustc` und deren Auswirkungen auf die Ausführungszeit sowie die Größe der Binärdatei eines Programms.

12.1.1. Keine Optimierungen

Dies ist die Standardeinstellung. Wenn Sie `cargo build` aufrufen, wird das Entwicklungs-Profil (auch bekannt als `dev`-Profil) verwendet. Da dieses Profil für das Debugging optimiert ist, werden Debug-Informationen aktiviert, jedoch *keinerlei* Optimierungen vorgenommen; es wird also `-C opt-level=0` verwendet.

Zumindest bei der Bare-Metal-Entwicklung verursachen Debug-Informationen keine Kosten im Sinne von Speicherplatzbedarf im Flash- oder ROM-Speicher. Daher empfehlen wir, Debug-Informationen auch im Release-Profil zu aktivieren – standardmäßig sind sie dort deaktiviert. Dies ermöglicht Ihnen die Verwendung von Breakpoints beim Debuggen von Release-Builds.

```
[profile.release]
# Symbole sind praktisch und vergroessern die Dateigroesse im Flash nicht
debug = true
```

Der Verzicht auf Optimierungen ist ideal für das Debugging, da sich das schrittweise Durchlaufen des Codes anfühlt, als würde man das Programm Anweisung für Anweisung ausführen; zudem lassen sich Stack-Variablen und Funktionsargumente in GDB per `print` ausgeben. Bei optimiertem Code führt der Versuch, Variablen auszugeben, hingegen zur Meldung `$0 = <value optimized out>`.

Der größte Nachteil des `dev`-Profils besteht darin, dass die resultierende Binärdatei sehr groß und langsam ist. Meist wiegt vor allem die Größe schwer, da nicht optimierte Binärdateien Dutzende KiB an Flash-Speicher belegen können – Speicherplatz, über den das Zielgerät womöglich gar nicht verfügt. Die Folge: Die nicht optimierte Binärdatei passt nicht auf das Gerät!

Gibt es eine Möglichkeit, kleinere und dennoch für das Debugging geeignete Binärdateien zu erhalten? Ja, es gibt einen Trick.

12.1.1.1. Optimierung der Abhängigkeiten

Cargo bietet mit `Profil-Überschreibung` eine Funktion, mit der Sie den Optimierungsgrad von Abhängigkeiten überschreiben können. So lassen sich alle Abhängigkeiten hinsichtlich ihrer Größe optimieren, während die oberste Crate unoptimiert und debuggfreundlich bleibt.

Beachten Sie, dass generischer Code manchmal zusammen mit der Crate optimiert werden kann, in der er instanziiert wird, anstatt mit der Crate, in der er definiert ist. Wenn Sie in Ihrer Anwendung eine Instanz einer generischen Struktur erstellen und feststellen, dass diese Code mit großem Speicherbedarf einbindet, kann es sein, dass eine Erhöhung des Optimierungsgrads der relevanten Abhängigkeiten keine Wirkung zeigt.

Hier ist ein Beispiel:

```
# Cargo.toml
[package]
name = "app"
# ..
```

```
[profile.dev.package."*"] # +
opt-level = "z" # +
```

Ohne die Überschreibung:

```
$ cargo size --bin app -- -A
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 9060   0x8000400
.rodata               1708   0x8002780
.data                    0  0x20000000
.bss                     4  0x20000000
```

Mit der Überschreibung

```
$ cargo size --bin app -- -A
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 3490   0x8000400
.rodata               1100   0x80011c0
.data                    0  0x20000000
.bss                     4  0x20000000
```

Das bedeutet eine Verringerung des Flash-Speicherverbrauchs um 6 KiB, ohne die Debug-Fähigkeit des Haupt-Crates zu beeinträchtigen. Wenn man in eine Abhängigkeit hineinspringt, erscheinen zwar wieder die Meldungen `<value optimized out>`, doch in der Regel möchte man das Haupt-Crate debuggen und nicht die Abhängigkeiten. Sollte es dennoch erforderlich sein, eine Abhängigkeit zu debuggen, lässt sich die Funktion `profile-overrides` nutzen, um diese spezifische Abhängigkeit von der Optimierung auszunehmen. Siehe dazu das folgende Beispiel:

```
# ..

# Optimiere das `cortex-m-rt`-Crate nicht
[profile.dev.package.cortex-m-rt] # +
opt-level = 0 # +

# optimieren Sie jedoch alle anderen Abhaengigkeiten
[profile.dev.package."*"]
codegen-units = 1 # better optimizations
opt-level = "z"
```

Jetzt sind das Top-Level-Crate und `cortex-m-rt` debuggerfreundlich!

12.1.2. Auf Geschwindigkeit optimieren

Seit dem 18.09.2018 unterstützt `rustc` drei Stufen der Geschwindigkeitsoptimierung: `opt-level = 1`, `2` und `3`. Beim Ausführen von `cargo build --release` wird das Release-Profil verwendet, das standardmäßig auf `opt-level = 3` eingestellt ist.

Sowohl `opt-level = 2` als auch `3` optimieren auf Geschwindigkeit zulasten der Binärgröße; allerdings führt Stufe `3` mehr Vektorisierung und Inlining durch als Stufe `2`. Insbesondere ist zu beobachten, dass LLVM bei einem `opt-level` von 2 oder höher Schleifen entrollt („Loop Unrolling“). Das Entrollen von Schleifen ist recht kostspielig im Hinblick auf den Flash-/ROM-Speicherbedarf (z. B. Anstieg von 26 auf 194 Bytes für eine Schleife zum Nullsetzen eines Arrays), kann aber unter geeigneten Bedingungen (etwa bei einer ausreichend hohen Anzahl von Iterationen) die Ausführungszeit halbieren.

Derzeit gibt es keine Möglichkeit, das Entrollen von Schleifen bei `opt-level = 2` und `3` zu deaktivieren; wenn Sie sich diesen Speicheraufwand also nicht leisten können, sollten Sie Ihr Programm stattdessen auf eine geringe Größe hin optimieren.

12.1.3. Nach Größe optimieren

Seit dem 18.09.2018 unterstützt `rustc` zwei Stufen zur Größenoptimierung: `opt-level = "s"` und `"z"`. Diese Bezeichnungen wurden von Clang/LLVM übernommen und sind nicht besonders aussagekräftig; allerdings soll das `"z"` signalisieren, dass damit kleinere Binärdateien erzeugt werden als mit `"s"`.

Wenn Ihre Release-Binärdateien hinsichtlich der Größe optimiert werden sollen, ändern Sie die Einstellung `profile.release.opt-level` in der Datei `Cargo.toml` wie unten dargestellt.

```
[profile.release]
# oder "z"
opt-level = "s"
```

Diese beiden Optimierungsstufen senken den Inline-Schwellenwert von LLVM erheblich – einen Kennwert, der darüber entscheidet, ob eine Funktion „geinlined“ (direkt an der Aufrufstelle eingebettet) wird oder nicht. Eines der Prinzipien von Rust sind „Zero-Cost-Abstractions” (Abstraktionen ohne Laufzeitkosten); diese nutzen häufig sogenannte „Newtypes“ und kleine Funktionen zur Wahrung von Invarianten (z. B. Funktionen wie `deref` oder `as_ref`, die einen inneren Wert ausleihen). Ein niedriger Inline-Schwellenwert kann daher dazu führen, dass LLVM Optimierungsmöglichkeiten verpasst (etwa das Entfernen von nicht erreichbaren Programmzweigen oder das Inlinen von Closure-Aufrufen).

Bei der Optimierung auf eine geringe Binärgröße kann es sinnvoll sein, den Inline-Schwellenwert zu erhöhen und zu prüfen, ob sich dies auf die Größe der Binärdatei auswirkt. Die empfohlene Methode zur Änderung dieses Schwellenwerts besteht darin, das Flag `-C inline-threshold` zu den übrigen `rustflags` in der Datei `.cargo/config.toml` hinzuzufügen.

```
# .cargo/config.toml
# Dies setzt voraus, dass Sie die cortex-m-quickstart-Vorlage verwenden
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = [
  # ..
  "-C", "inline-threshold=123", # +
]
```

Welchen Wert verwenden? Ab Version 1.29.0 gelten für die verschiedenen Optimierungsstufen folgende Inline-Schwellenwerte:

- `opt-level = 3` verwendet 275
- `opt-level = 2` verwendet 225
- `opt-level = "s"` verwendet 75
- `opt-level = "z"` verwendet 25

Du solltest 225 und 275 ausprobieren, wenn du auf die Größe optimierst.

12.2. Mathematische Funktionen mit `#[no_std]` nutzen

Wenn Sie mathematische Operationen durchführen möchten – wie etwa die Berechnung der Quadratwurzel aus der Exponentialfunktion einer Zahl – und Ihnen die vollständige Standardbibliothek zur Verfügung steht, könnte Ihr Code folgendermaßen aussehen:

```
//! Einige mathematische Funktionen mit Standardunterstützung verfügbar

fn main() {
    let float: f32 = 4.82832;
    let floored_float = float.floor();
```

```
    let sqrt_of_four = floored_float.sqrt();

    let sinus_of_four = floored_float.sin();

    let exponential_of_four = floored_float.exp();
    println!("Floored test float {} to {}", float, floored_float);
    println!("The square root of {} is {}", floored_float, sqrt_of_four);
    println!("The sinus of four is {}", sinus_of_four);
    println!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
}
```

Ohne Unterstützung durch die Standardbibliothek stehen diese Funktionen nicht zur Verfügung. Stattdessen kann ein externes Crate wie `libm` verwendet werden. Der Beispielcode sähe dann folgendermaßen aus:

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::{debug, hprintln};
use libm::{exp, floorf, sin, sqrtf};

#[entry]
fn main() -> ! {
    let float = 4.82832;
    let floored_float = floorf(float);

    let sqrt_of_four = sqrtf(floored_float);

    let sinus_of_four = sin(floored_float.into());

    let exponential_of_four = exp(floored_float.into());
    hprintln!("Floored test float {} to {}", float, floored_float).unwrap();
    hprintln!("The square root of {} is {}", floored_float, sqrt_of_four).unwrap();
    hprintln!("The sinus of four is {}", sinus_of_four).unwrap();
    hprintln!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
    .unwrap();
    // exit QEMU
    // HINWEIS: Fuehren Sie dies nicht auf der Hardware aus; es kann den
    //          OpenOCD-Zustand beschaedigen.
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Wenn Sie komplexere Operationen wie DSP-Signalverarbeitung oder fortgeschrittene lineare Algebra auf Ihrem Mikrocontroller durchführen müssen, könnten Ihnen die folgenden Crates weiterhelfen.

- CMSIS DSP library binding

- constgebra
- micromath
- microfft
- nalgebra

A. Anhang A: Glossar

Das Embedded-Ökosystem ist geprägt von einer Vielzahl unterschiedlicher Protokolle, Hardwarekomponenten und herstellerspezifischer Elemente, für die jeweils eigene Begriffe und Abkürzungen verwendet werden. Dieses Glossar soll diese auflisten und Hinweise zum besseren Verständnis geben.

BSP

Ein „Board Support Crate“ stellt eine auf eine bestimmte Platine zugeschnittene Schnittstelle auf hoher Ebene bereit. Es hängt in der Regel von einem HAL-Crate ab. Eine ausführlichere Beschreibung finden Sie auf der Seite zu den im Speicher abgebildeten Registern oder für einen umfassenderen Überblick sehen Sie sich dieses Video an.

FPU

Gleitkommaeinheit. Ein „Mathematikprozessor“, der ausschließlich Operationen mit Gleitkommazahlen ausführt.

HAL

Ein „Hardware Abstraction Layer“-Crate bietet eine entwicklerfreundliche Schnittstelle zu den Funktionen und Peripheriegeräten eines Mikrocontrollers. Es wird in der Regel auf Basis eines Peripheral Access Crate (PAC) implementiert. Möglicherweise implementiert es auch Traits aus dem `embedded-hal`-Crate. Eine ausführlichere Beschreibung finden Sie auf der Seite zu den im Speicher abgebildeten Registern oder für einen umfassenderen Überblick in diesem Video.

I2C

Wird manchmal auch als „I²C“ oder „Inter-IC” bezeichnet. Es handelt sich um ein Protokoll für die Hardware-Kommunikation innerhalb eines einzelnen integrierten Schaltkreises. Weitere Informationen finden Sie hier.

PAC

Ein „Peripheral Access Crate“ ermöglicht den Zugriff auf die Peripheriegeräte eines Mikrocontrollers. Es handelt sich um eines der Crates auf niedrigerer Ebene, das in der Regel direkt aus der bereitgestellten SVD generiert wird, häufig unter Verwendung von svd2rust. Die Hardware-Abstraktionsschicht hängt in der Regel von diesem Crate ab. Eine ausführlichere Beschreibung finden Sie auf der Seite zu den im Speicher abgebildeten Registern oder für einen umfassenderen Überblick in diesem Video.

SPI

Serial Peripheral Interface. Weitere Informationen finden Sie hier.

SVD

„System View Description“ ist ein XML-Dateiformat, das dazu dient, die Programmierersicht eines Mikrocontrollers zu beschreiben. Weitere Informationen hierzu finden Sie auf der ARM CMSIS-Dokumentationsseite.

UART

Universeller asynchroner Empfänger-Sender. Weitere Informationen finden Sie hier.

USART

Universeller synchroner und asynchroner Empfänger-Sender. Weitere Informationen finden Sie hier.