

Vorstellung

Tiago Manczak

Michael Schury

<https://github.com/datenzauberer>

Rust Meetup Augsburg



Diese Präsentation findet ihr unter: <https://github.com/rust-augsburg/2024-04-20-linux-tag-rust-picow-workshop>

Warum Rust und insbesondere Embedded Rust?

- Sicherheit, insb. Speichersicherheit (Ownership, Borrowing, Lifetimes))
- Vermeidung von Laufzeitfehlern
- Zero-Cost-Abstractions
- Portabilität und Cross-Compiling
- Einheitliches Ökosystem
 - Cargo als Paket Manager
 - Dependency Management
 - Test Framework
 - Dokumentationserstellung
 - Plattformunabhängige Entwicklung

Warum Raspberry Pi Pico WH ?

<https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html#raspberry-pi-pico-w-and-pico-wh>

Entscheidung für Pico Family

- **Preis-Leistungs-Verhältnis:** viele Peripherals zu einem günstigen Preis
- Professionelle Entwicklungsumgebung
- Umfassende Dokumentation

Entscheidung für Model Pico W

- Preisvorteil
- Wi-Fi-Fähigkeit

Embedded Grundlagen

- **Board vs. Microcontroller**
 - **Board:** Pico WH, Pico W, Pico H
 - **Microcontroller:** RP2040
- **Cross-Compilation**
- **Debug Probe**

Pico

Mikrocontroller: RP2040

- RP steht für Raspberry
- 2 Anzahl der Prozessoren
- 0 ARM-Architektur M0+
- 4 Speicher: 264KB SRAM

W-Modell: Infineon CYW43439

- **Eingebautes WLAN:** 802.11 b/g/n
- **Bluetooth:** Unterstützung für BLE

Nützliche Links:

- [rp2040-datasheet.pdf](#)
- [getting-started-with-pico.pdf](#)
- <https://datasheets.raspberrypi.com>
- <https://www.raspberrypi.com/documentation/microcontrollers>

Picoprobe setup

Two Raspberry Pi Picos are used. PicoA operates as a debug probe, while PicoB serves as the production probe (the target hosting your code). Details in [Getting Started with Pico - Appendix A: Using Picoprobe](#).

Flash debug probe (PicoA)

Download the firmware **debugprobe_on_pico.uf2** from:

<https://github.com/raspberrypi/debugprobe/releases>

ATTENTION: **debugprobe_on_pico.uf2** is needed !!

(Alternatively build the debugprobe from source code as described "Getting started with Pico - Build and flash picoprobe")

Boot the Raspberry Pi PicoA with the BOOTSEL button pressed and copy the firmware, e.g.

```
sudo cp ~/Downloads/debugprobe_on_pico.uf2 /media/michael/RPI-RP2
```

Picoprobe Wiring

Debug Wiring (SWD and UART bridge)

PicoA		PicoB	
Pin	Description	Pin	Description
38	GND	Debug2	GND
4	GP2	Debug1	SWCLK

PicoA		PicoB	
5	GP3	Debug3	SWDIO
6	GP4/UART1 TX	2	GP1/UART0 RX
7	GP5/UART1 RX	1	GP1/UART0 RX

Note: DebugPins are numbered from left to right when the USB connector is facing up.

Power Supply (Optional)

One advantage of using a Pico as a debug probe is that no separate power supply for the PicoB is needed:

PicoA		PicoB	
Pin	Description	Pin	Description
39	VSYS	39	VSYS

Of course It's also possible to provide the power the PicoB via USB.

Testing Picoprobe

Test the connection to the probe

```
probe-rs list
```

This should return a output like:

```
The following debug probes were found:
[0]: Debugprobe on Pico (CMSIS-DAP) (VID: 2e8a, PID: 000c, Serial:
E661A4D417595929, CmsisDap)
```

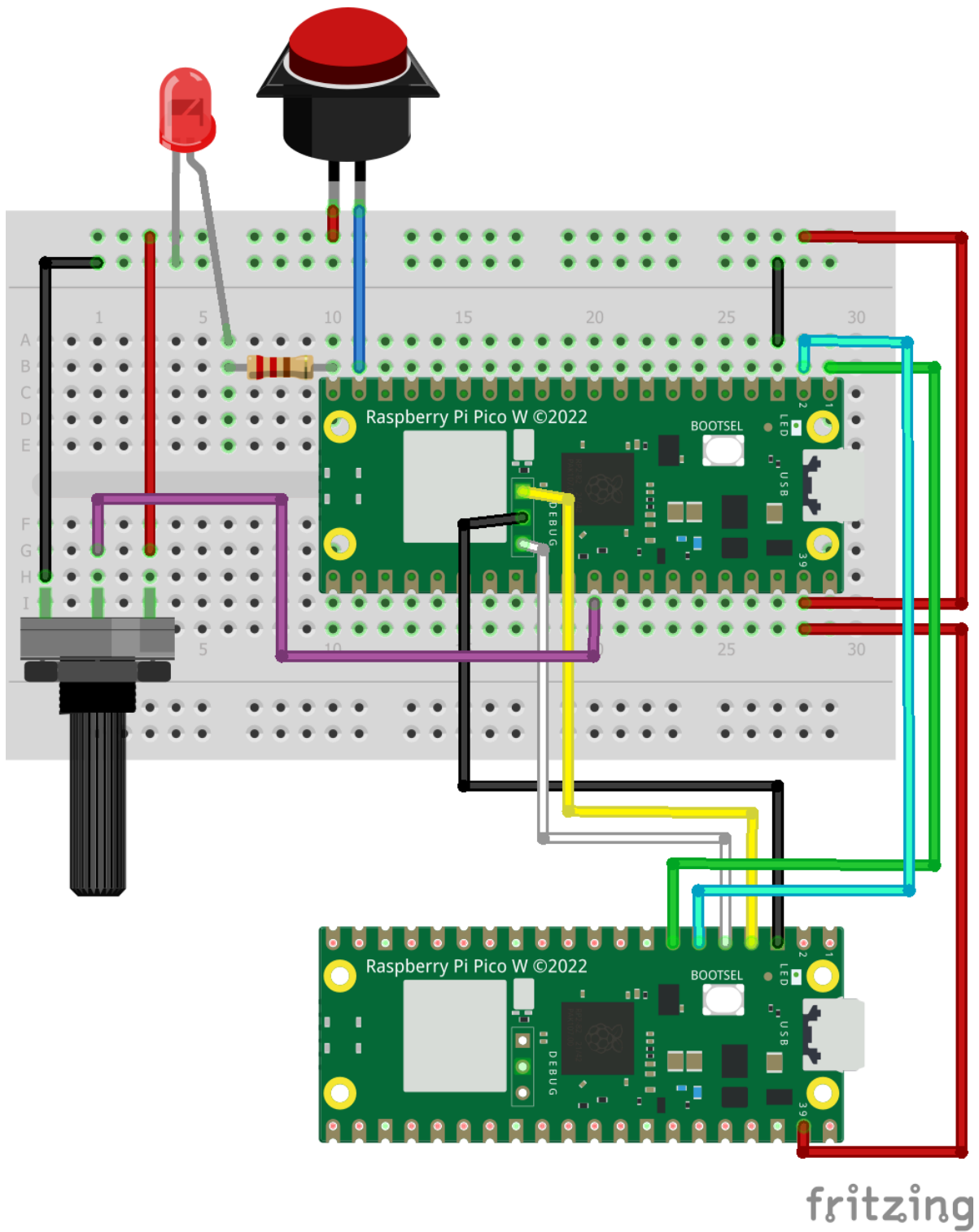
Now test with flashing a simple application.

Beschreibung des Hardware Aufbaus

Stückliste

- Pico W (als Debug Probe)
- Pico W (als Target)
- LED
- Taster
- Widerstand 670 Ohm
- Potentiometer 470 KOhm
- Female-Female-cable (zur Verbindung mit Debug Probe: 3 DebugPins)
- Female-Male-cable (zur Verbindung mit Debug Probe: UART und Power)

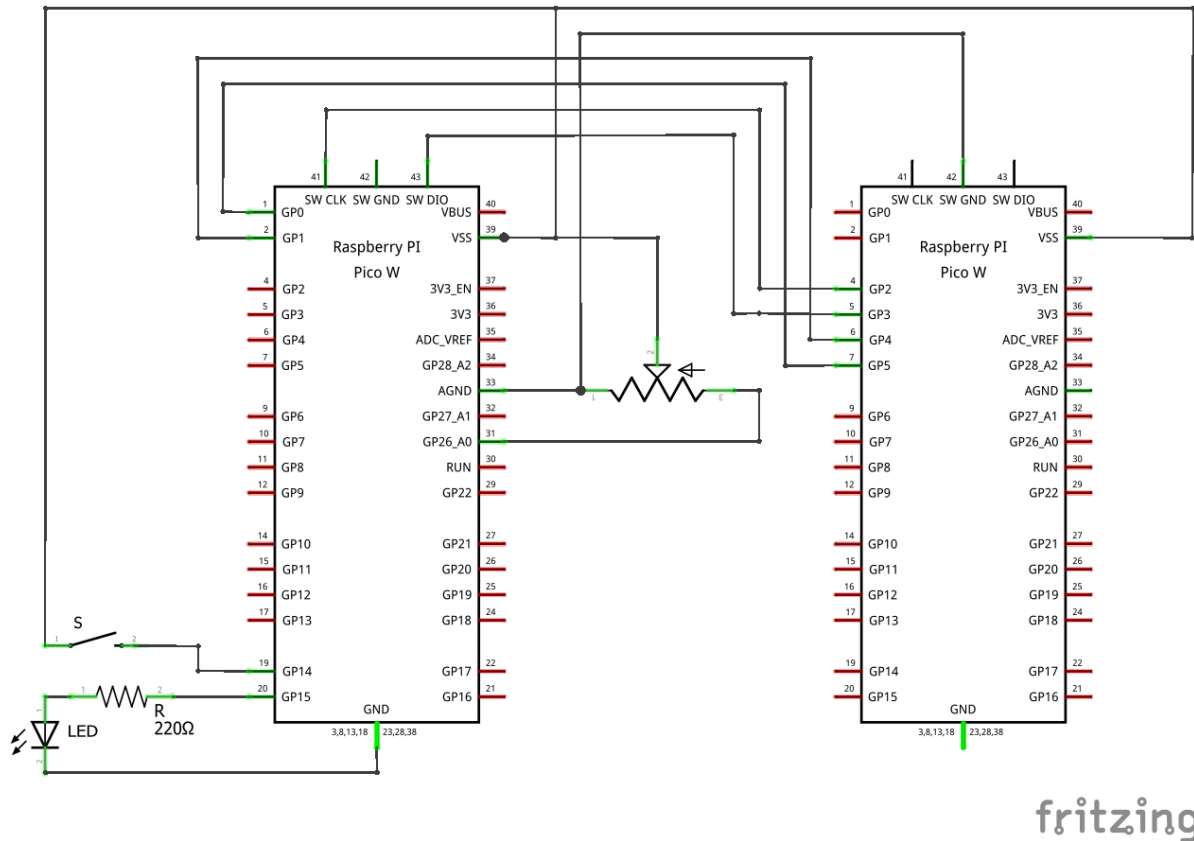
Steckbrettaufbau



Belegte PINs von Target Pico:

GP14 Taster
GP15 LED
GP26 / ADC0 Potentiometer

Schaltplan



fritzing

Steckbrettverdrahtung

Für DebugPico sind die Pins entsprechend der Nummerierung auf dem Pico-Board angeordnet. Für die linke und rechte Seite des Steckbretts folgt die Pin-Nummerierung der jeweiligen Steckbrett Nummerierung

Steckbrett Linke Seite

Quelle	Ziel
+	Rechts 2
1	DebugPico 7
2	DebugPico 6
3	- (left)
19	Button 1
+	Button 2
20	Widerstand 1
25	Widerstand 2
25	LED Anode
-	LED Kathode

Steckbrett Rechte Seite

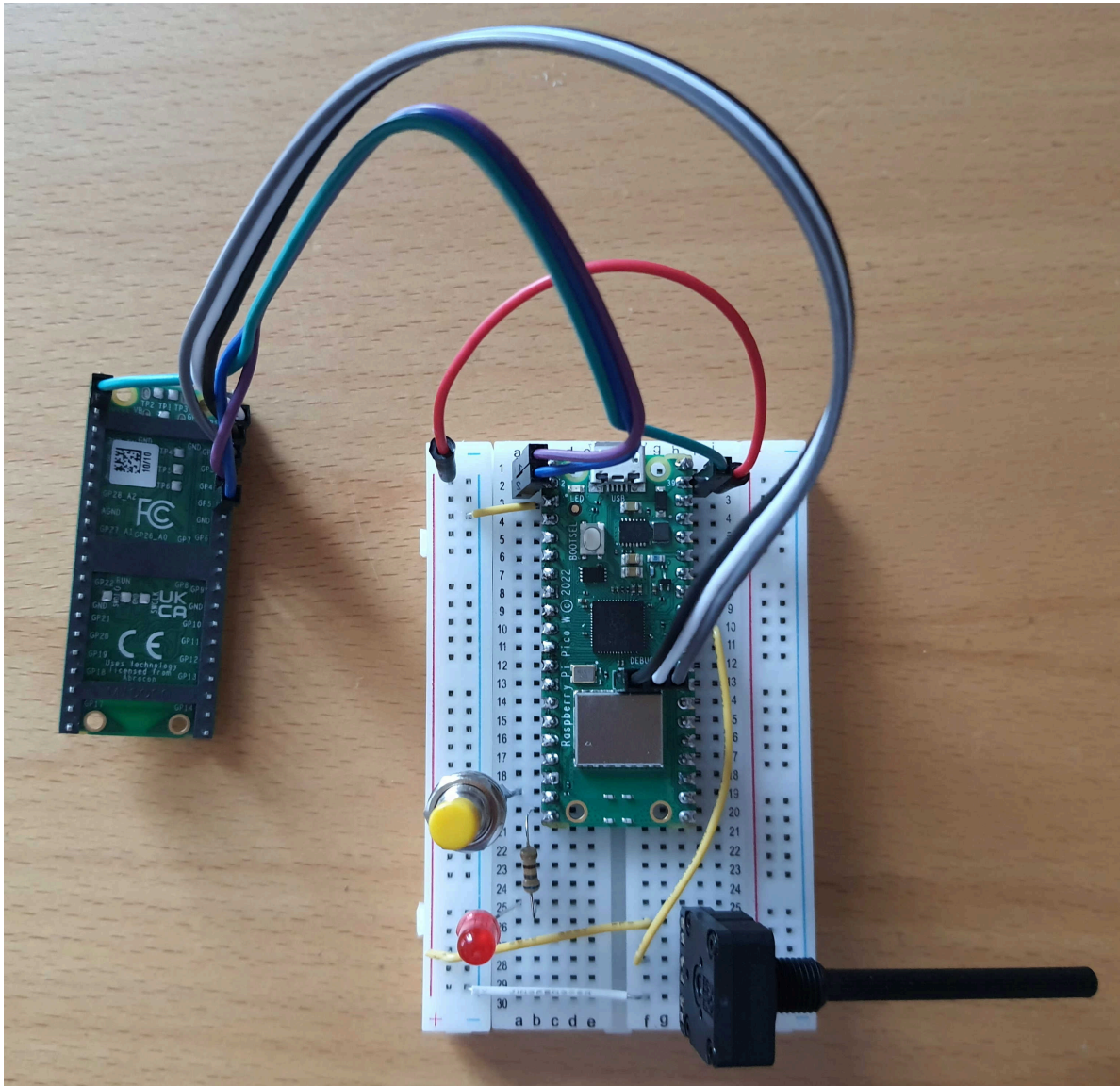
Quelle	Ziel
2	DebugPico 39
2	+ (links)
26	+ (links)
28	10
30	- (links)
26	Potentiometer 1
28	Potentiometer 2
30	Potentiometer 3

DebugPins

Die Pins auf dem Debug Probe sind von links aufsteigend nummeriert (USB Stecker zeigt nach oben).

Quelle	Ziel
1	DebucPico 4
2	DebucPico 3
3	DebucPico 5

Bild Steckbrettaufbau



Setup Allgemein

Um sicherzustellen, dass wir im Workshop direkt loslegen können, wäre es super, wenn ihr das Setup der Entwicklungsumgebung schon vorab erledigen könntet. Vielen Dank!.

Im Folgenden ist die Installation unter Ubuntu 22.04 LTS exemplarisch beschrieben. Solltet ihr ein anderes Betriebssystem nutzen, lest bitte die entsprechenden Installationsanleitungen.

Zur Information für Neugierige: Im Workshop werden wir zwei Raspberry Pi Pico verwenden. Einer wird als 'Debug-Probe' und der andere als 'Target' verwendet (näheres unter "[Debug Probe](#)"). Auch wenn's ohne Debug-Probe klappt, wird's auf Dauer echt lästig, weil man den Pico vor jedem Flashen immer wieder in den BOOTSEL-Modus bringen muss.

Setup der Entwicklungsumgebung

Die folgende Beschreibung des Setups setzt voraus, dass Ubuntu 22.04 LTS verwendet wird.

Ubuntu 22.04 einrichten

Bitte stellt sicher, dass Ihre Ubuntu 22.04-Installation wie unten beschrieben eingerichtet ist.

```
sudo apt update
sudo apt install -y git
# curl wird für die Installation von rust benötigt
sudo apt install -y curl
# libudev-dev wird für die Fehlersuche benötigt
sudo apt install -y libudev-dev

# für Cargo-Generierung:
sudo apt install -y build-essential
sudo apt install -y pkg-config libssl-dev
```

Visual Studio Code (VSCode)

Lade VS Code von der offiziellen Website herunter:

<https://code.visualstudio.com/download>

Installiere es, z.B.:

```
sudo apt install ~/Downloads/code_1.87.2-1709912201_amd64.deb
```

Installiere die VS Code Erweiterungen (über die Kommandozeile):

```
# rust development
code --install-extension rust-lang.rust-analyzer
# debug rust code
code --install-extension vadimcn.vscode-lldb # auf macOS/Linux
#code --install-extension ms-vscode.cpptools # unter Windows
# für die Fehlersuche mit probe-rs:
code --install-extension probe-rs.probe-rs-debugger
```

Starte VS Code:

- Starte VS Code aus dem Terminal mit `code` oder über das Anwendungsmenü.

Rust installieren

Folge den offiziellen Rust-Installationsanweisungen, um Rust zu installieren, einschließlich des Compilers (rustc) und des Paketmanagers (cargo):

<https://www.rust-lang.org/tools/install>

Führen den folgenden Installationsbefehl aus (mit einer "1 Standard Installation").

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Neustart der Shell oder setzen der Umgebungsvariablen wie in der Terminalausgabe angegeben (z.B. für Bash: `source $HOME/.cargo/env`).

Überprüfe Rust-Installation

```
rustc --version
```

Dieser Befehl liefert die Version des Rust-Compilers, `rustc`, zurückgeben, welcher aktuell auf Ihrem System verwendet wird. Dies ist zum Beispiel `rustc 1.77.1 (7cf61ebde 2024-03-27)`, wobei die Versionsnummer und das Veröffentlichungsdatum angegeben werden.

Cargo ist der Paketmanager und das Build-System von Rust. Um zu überprüfen, ob Cargo korrekt installiert ist und um seine Version zu sehen:

```
cargo --version
```

Embedded Rust Development Dependencies

Installiere `cargo-generate`:

```
cargo install cargo-generate
```


Wenn die Installation aufgrund fehlender Abhängigkeiten fehlschlägt, die benötigten Pakete wie in [Ubuntu 22.04 setup](#) beschrieben installieren.

Folge den Installationsanweisungen [rp-rs/rp-hal](#) *Getting Started*:

```
rustup self update
rustup update stable
rustup target add thumbv6m-none-eabi
cargo install elf2uf2-rs --locked
cargo install probe-rs --features cli --locked
cargo install flip-link
```

Wenn die Installation aufgrund fehlender Abhängigkeiten fehlschlägt, die benötigten Pakete wie in [Ubuntu 22.04 setup](#) beschrieben installieren.

ACHTUNG: Aktualisiere die `/etc/udev/rules.d` (wie in der [Probe.rs Dokumentation](#) beschrieben):

```
curl -o ~/Downloads/69-probe-rs.rules https://probe.rs/files/69-probe-rs.rules
sudo cp ~/Downloads/69-probe-rs.rules /etc/udev/rules.d
sudo udevadm control --reload
sudo udevadm trigger
```


Dokumentationserstellung

Zur Erstellung dieser Dokumentation bitte folgende Tools installieren:

```
cargo install mdbook
cargo install mdbook-toc
cargo install mdbook-pdf
```

Minimale Rust Grundlagen

```
fn main() {  
    // Immutable Variable mit expliziter Typangabe  
    let _logical: bool = true; // Variablen sind standardmäßig immutable  
  
    // Mutable Variable - nur mutable Variablen können geändert werden  
    let mut mutable: i32 = 12; // Mutable `i32`  
  
    // Call by Reference: Übergeben der Variable `mutable` an eine Funktion  
    modify_value(&mut mutable); // die Funktion benötigt einen mutable ref  
  
    // Ausgabe des veränderten Wertes  
    println!("Modified value: {}", mutable);  
  
    show_value(&mutable);  
    //demo_for_loop();  
}  
  
// Funktion, die einen mutable reference auf eine i32 annimmt und den Wert  
fn modify_value(value: &mut i32) {  
    *value += 9; // Dereferenzierung und Änderung des Wertes  
}  
  
// Funktion, die einen reference verarbeitet  
fn show_value(value: &i32) {  
    // Ausgabe des referenzieren Wertes  
    // Dereferenzierungsoperator * ist nicht notwendig  
    println!("Show value: {}", *value);  
}  
  
fn demo_for_loop() {  
    for i in 0..3 {  
        println!("Simple delay with for_loop: {i}");  
    }  
}
```



Obiger Code nutzt <https://play.rust-lang.org/> zum Ausführen.

Links

[Rust by Example](#) zum Nachschlagen

[Rust Book](#) zum Lernen

[Rustlings](#) zum Üben

[crates.io](#)

[docs.rs](#)

Rust Embedded

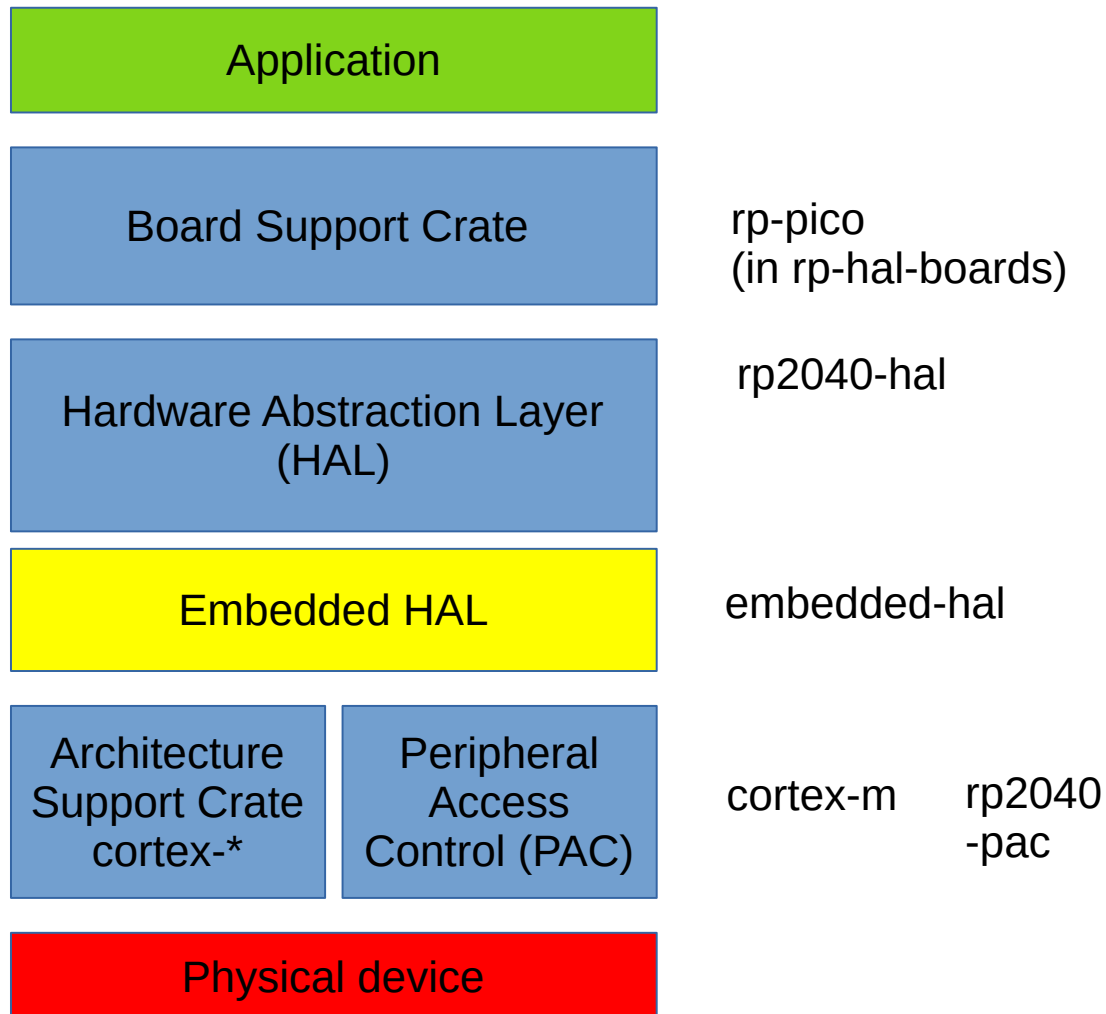
Herausforderungen

- `no_std`-Umgebung
- Cross-Compilierung
- Debugging
- Logging

Möglichkeiten in Rust

- `rp-pico`
 - einfach(er) zu lernen
 - Sammlung von crates
- `embassy`
 - async-basiert
 - schwieriger zu lernen
 - unterstützt Wi-Fi (und interne LED)

Rust Embedded Architektur



Links:

- `rp-pico`: (in Github)
 - <https://crates.io/crates/rp-pico> enthält auch Links zu den Beispielen
 - ist in <https://github.com/rp-rs/rp-hal-boards>
- `rp2040-hal`:

- <https://crates.io/crates/rp2040-hal> enthält keine(!) Links zu Beispielen
 - https://docs.rs/rp2040-hal/latest/rp2040_hal/
 - <https://github.com/rp-rs/rp-hal/tree/main/rp2040-hal>
 - <https://github.com/rp-rs/rp-hal/tree/main/rp2040-hal/examples> guter Einstiegspunkt
- embedded-hal:
 - <https://github.com/rust-embedded/embedded-hal> enthält "Import info: `embedded-hal v1.0` is now released!"
 - <https://blog.rust-embedded.org/embedded-hal-v1/> Focus on drivers

We've removed traits that were found to not be usable for generic drivers (most notably timers).

Details siehe: [migrating-from-0.2-to-1.0](#)

Projektaufbau

```
.
├── .cargo
│   └── config.toml
├── Cargo.toml
├── Embed.toml
├── memory.x
├── README.md
├── src
│   └── main.rs
└── .vscode
    └── settings.json
```

- `.cargo/config` specifies runner, build target
- `Embed.toml` configures `probe-rs`, `gdb`, `rtt`
- `memory.x` describes where the RAM and FLASH are and their sizes
- `build.rs` ensures `memory.x` is in out directory
- `.vscode/settings.json` configures debugger

Projekterstellung

Wir legen ein Musterprojekt namens `rust-pico-linuxtag` an, starten es und debuggen es ein bisschen.

Testaufbau und Debugging

Projekt anlegen via shell:

Nutze den folgenden Befehl in der Shell, um das Projekt zu erstellen:

```
cargo generate https://github.com/datenzauberer/rp2040-project-template
# offizielles project-template verwendet embedded-hal v0.2 (statt 1.0)
# cargo generate https://github.com/rp-rs/rp2040-project-template
```

Wähle `rust-pico-linuxtag` als Projektnamen und `probe-rs` für das Flashen.
Der Output sollte etwa so aussehen:

```
🔧 project-name: rust-pico-linuxtag ...
🔧 Generating template ...
? 🧑 Which flashing method do you intend to use? ›
> probe-rs
  elf2uf2-rs
  custom
  none

✓ 🧑 Which flashing method do you intend to use? • probe-rs
🔧 Moving generated files into: `../rust-pico-linuxtag`...
🔧 Initializing a fresh Git repository
✨ Done! New project created ../rust-pico-linuxtag
```


Code anpassung in src/main.rs

Öffne das Projekt in VSCode: , z.B. via Shell:

```
code ./rust-pico-linuxtag
```

Ändere in `src/main.rs` den LED-Pin:

```
let mut led_pin = pins.gpio15.into_push_pull_output();
```

Und dann starten wir:

```
cargo run
```

Debugging

Anpassung der `launch.json` für das neue Binary

Nun `launch.json` anpassen, ersetze `rp2040-project-template` mit `rust-pico-linuxtag`:

```
        "programBinary": "target/thumbv6m-none-eabi/debug/rust-pico-linuxtag",
```

Den Namen des Binaries kannst Du auch mittels Shell ermittelt: `ls -l target/thumbv6m-none-eabi/debug/`

Debugger in VSCode starten mit `Ctrl-Shift-D` und zum Starten auf den grünen Pfeil klicken.

Projektanalyse

Mit `cargo tree` werden die Projektabhängigkeiten angezeigt.

Nützliche Links

Hier ein paar nützliche Links für die Arbeit mit dem Pico unter Rust:

- **Board Support Package:** [RP2040 HAL Boards GitHub](#)
- **HAL:** [rp2040-hal auf docs.rs](#)
- Board Support Package: <https://github.com/rp-rs/rp-hal-boards/tree/main/boards/rp-pico/examples>
- HAL: https://docs.rs/rp2040-hal/latest/rp2040_hal

Um die Repos zu klonen, geh in Repo-Verzeichnis:

```
git clone https://github.com/rp-rs/rp-hal-boards
git clone https://github.com/rp-rs/rp-hal
```

```
# Beispiele sind unter examples zu finden:
ls ./rp-hal-boards/boards/rp-pico/examples
ls ./rp-hal/rp2040-hal/examples
```

General Purpose Input/Output (GPIO)

Auf unserem Steckbrett wird GPIO15 für die LED, GPIO14 für den Taster verwendet.

Allgemeine Erklärung zu GPIOs: [elektronik-kompendium.de GPIO](http://elektronik-kompendium.de/gpio)

Information zur Verwendung von GPIOs mit Rust: https://docs.rs/rp2040-hal/latest/rp2040_hal/gpio/index.html

Ansteuern der LED:

```
let mut led_pin = pins.gpio15.into_push_pull_output();
```

Beispiele

Beispiel1: blinky mit clock delay

Bereits geklont. Betrachte die Implementierung der Verzögerung:

```
delay.delay_ms(500);
```

Beispiel2: blinky mit Rust Schleife

Beschreibung

Verwende zur Verzögerung einer Rust Schleife.

Beispiel3: LED leuchtet nur wenn Taster gedrückt ist

Falls ihr schon früher die Lösung habt dürft ihr gerne kreativ werden und Euch selbst Aufgaben ausdenken.

Eine Variante hierzu: Taster als Schalter: Durch Betätigen des Tasters wird die LED eingeschaltet, bei erneutem Drücken ausgeschaltet.

Pulse-width modulation (PWM)

PWM lt. elektronik-kompendium.de

Beispiel4: LED mit PWM ansteuern

Wichtige Information: LED ist auf GPIO15. Hierzu muss aus dem Dateblatt die Slice und die Channel ermittelt werden. Ansonsten kann das `rp2040-hal` Beispiel verwendet werden.

Analog-to-digital converter (ADC)

Nützliche Links:

elektronik-kompodium.de: ADC

rohm.de: ad-da-converters

Beispiele

Beispiel5: Temperature sensor in free-running mode

siehe: https://docs.rs/rp2040-hal/0.10.0/rp2040_hal/adc/index.html#free-running-mode-with-fifo

Beispiel6: AnalogWert von Potentiometer lesen

Wie oben nur wird der `ADC0` gelesen:

```
// Configure GPIO26 as an ADC input
let mut adc_pin_0 =
rp2040_hal::adc::AdcPin::new(pins.gpio26).unwrap();
```

Interrupt Requests (IRQ)

elektronik-kompodium.de Pico IRQs

Beispiel7: blinky mit Interrupt gesteuerten Timer

Abschluss

Fragen und Antworten

Weiteren Praxis-Themen einholen

Folgende Vorschläge (unsererseits):

- Memory Game
- RTC
- PWM
- PIO
- I2C
- SPI
- Multicore-Programming
- Async (Embassy)
- Wi-fi
- Bluetooth

Habt Ihr Vorschläge und Ideen ??

Nächste Schritte



Rust Augsburg Meetup 16.05.2024