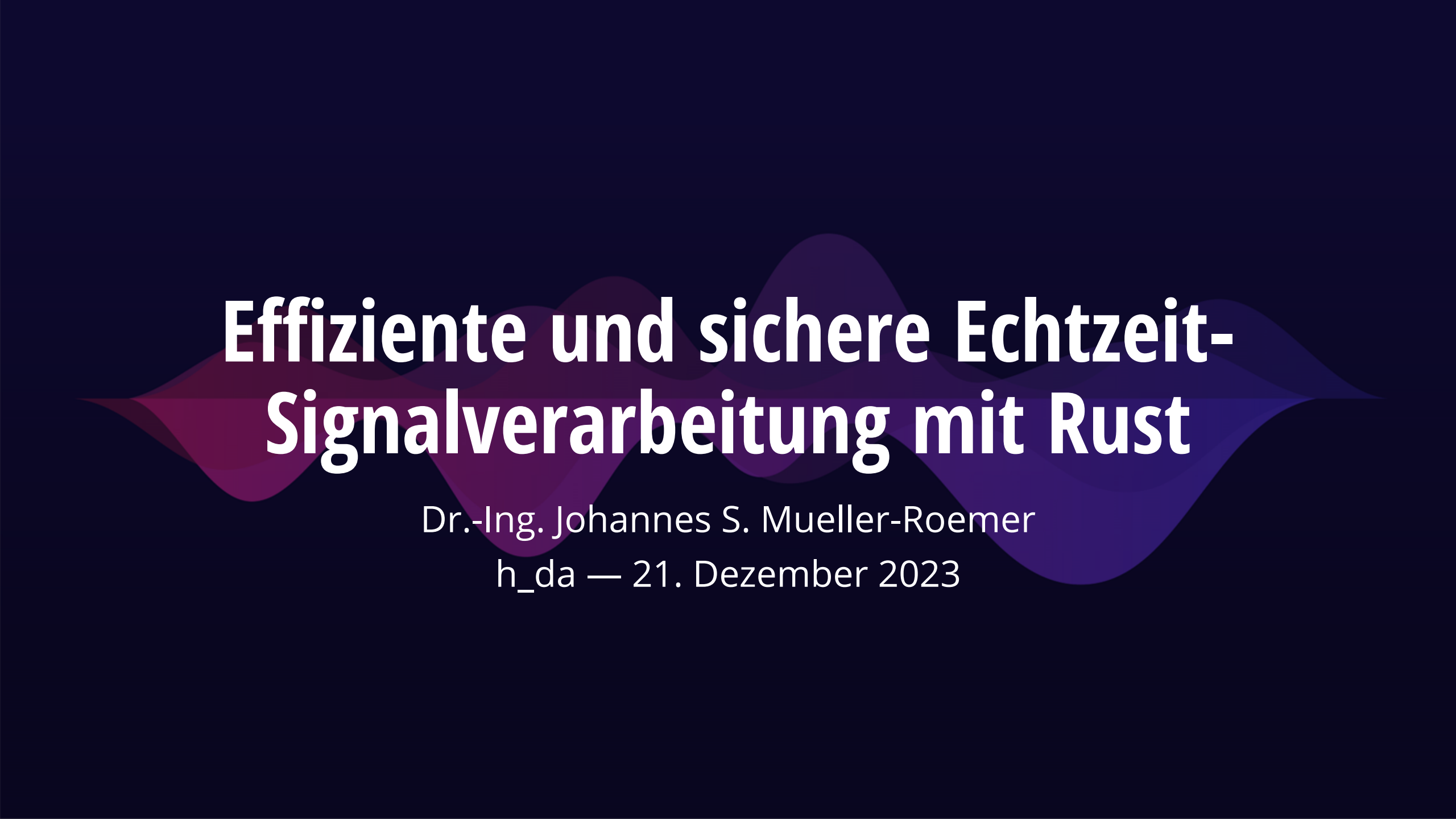


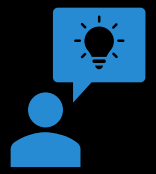
Effiziente und sichere Echtzeit-Signalverarbeitung mit Rust

Dr.-Ing. Johannes S. Mueller-Roemer

h_da — 21. Dezember 2023

Überblick

Lernziele



Motivation und Begriffe

DSP mit Rust

Ausblick

Lernziele

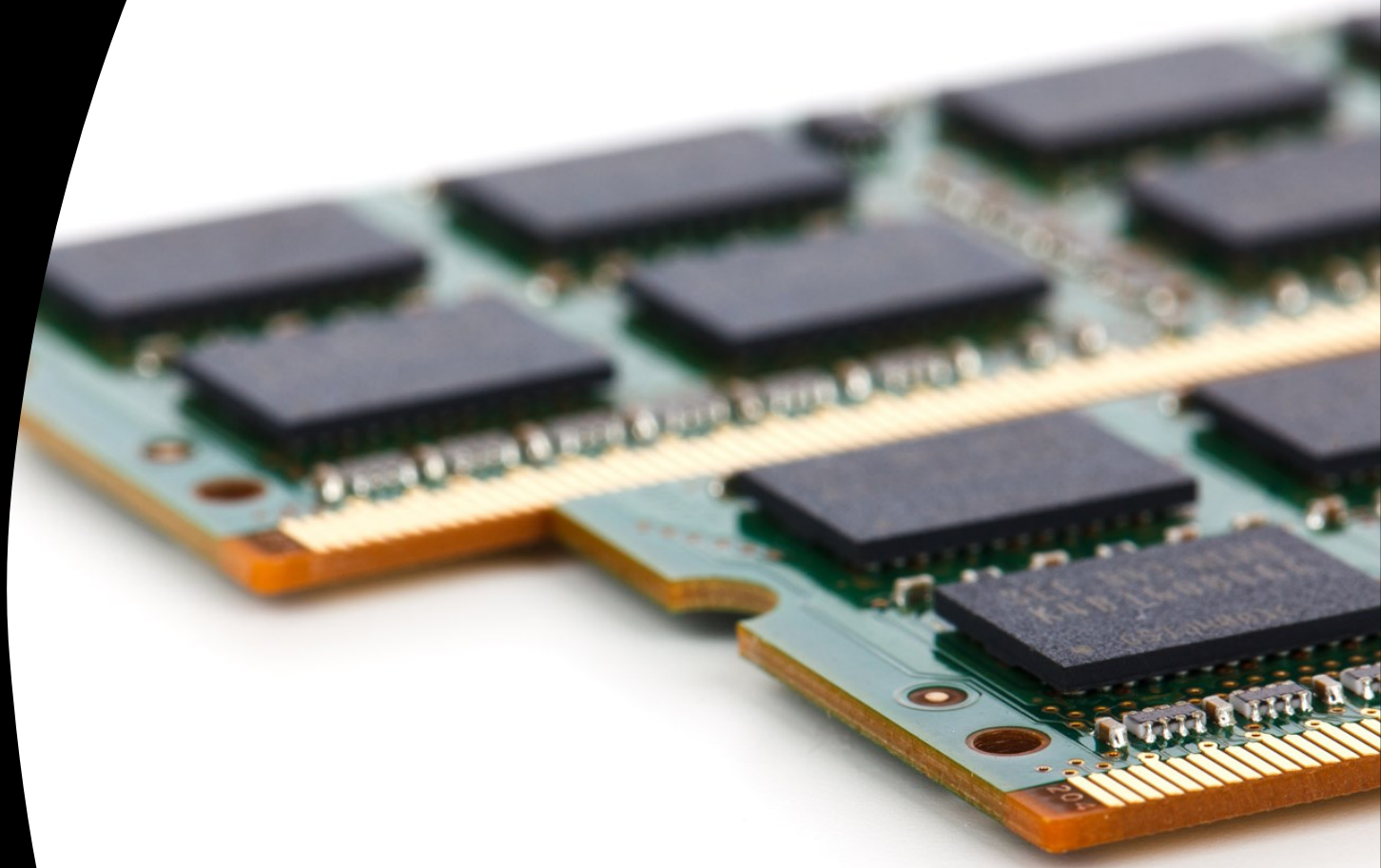
Lernziele

Wichtige Begriffe und Definitionen kennen

Wissen was zu beachten ist, um konsistente, niedrige Latenzen zu erreichen

Verstehen welche Probleme Rust löst und welche nicht

Motivation und Begriffe



DSP

- *Digitale Signalverarbeitung (DSP)* mit niedriger, vorhersagbarer Latenz notwendig für…
  - Audio-Signalverarbeitung
    - insb. für Live-Einsatz
  - Software-Defined Radio (SDR)
    - insb. für bidirektionale Kommunikation
  - (Akustisches) Beamforming
  - u.v.m.

© Electro-Harmonix

Echtzeit

- Bei Anwendungen die eine niedrige, vorhersagbare Latenz benötigen, spricht man von *Echtzeit-Systemen* (engl. *real-time systems*)
- Diese kann man weiter unterteilen in…
  - *Hard real-time* systems
    - Systeme, bei denen jedes Verfehlen einer Deadline ein Fehler ist und zu Schäden an Mensch oder Maschine führen kann, z. B. ABS
  - *Soft real-time* systems
    - Systeme bei denen gelegentliches Verfehlen einer Deadline nur die Qualität senkt
  - Weitere, detailliertere Unterteilung möglich (firm real-time etc.)

BRANDT, S. A., ET AL. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. *RTSS 2003. 24th IEEE Real-Time Systems Symposium*, 2003

Safety?

- Im Kontext von Rust ist oft von „safe“ und „unsafe“ die Rede
- „Safety“ = Sicherheit, aber welcher Art?
- Funktionale Sicherheit im Sinne von ISO 26262/ASIL oder DO-178C/DAL?
  - Nein, nur durch entsprechende Prozesse und Dokumentation (QM, Validierung, Tests) erreichbar
- Cybersecurity und Sicherheit vor Cyber-Angriffen?
  - Nein, während Rust manche relevante Problemklassen eliminiert, schützt es nicht vor Fehlern wie der Nutzung schwacher Hash-Funktionen, dem Speichern von Passwörtern im Klartext usw.

Safety

- Rust wurde entwickelt mit dem Ziel Speicherzugriffs- und Synchronisationsfehler sowie zu verhindern…
  - …ohne dabei *Garbage Collection (GC)* nutzen zu müssen oder Effizienz einzubüßen
  - …oder auf low-level Zugriff verzichten zu müssen → `unsafe`
- Speicherzugriffsfehler machen bis zu 60–75% der sicherheitsrelevanten Fehler aus
  - Der in Rust verwendete Ansatz sie zu verhindern ist formal bewiesen

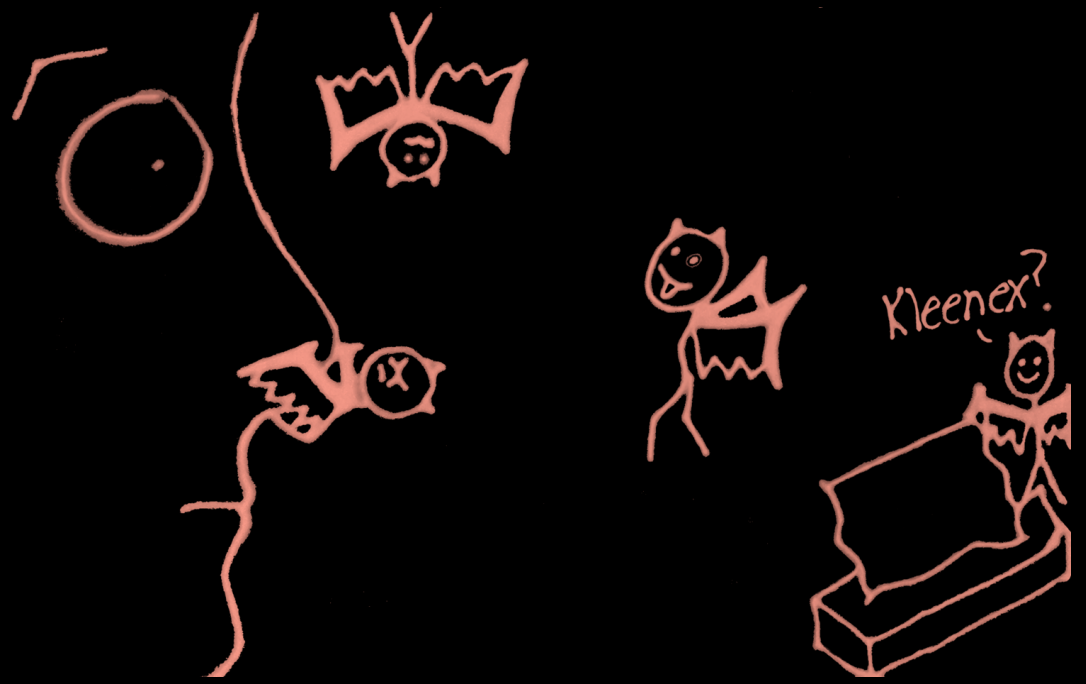
GAYNOR, A. What science can tell us about C and C++'s security. 2020
JUNG, R. ET AL. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages 2 (POPL)*, 2018

Rust für DSP auf eingebetteten Systemen

- Warum Rust für Echtzeit-DSP auf eingebetteten Systemen verwenden?
  - Speichersichere Sprachen gibt es viele
  - Funktionale Programme sind oft trivial parallelisierbar
- Wegen der Echtzeit-Anforderung sind Sprachen mit GC problematisch
  - GC läuft zu schlecht vorhersagbaren Zeitpunkten
- Funktionale Sprachen müssen i. d. R. sog. „Boxing“ nutzen
  - Allokation aller Variablen auf dem Heap
  - Kann nicht immer durch Optimierung korrigiert werden

Rust für DSP auf eingebetteten Systemen

- Bisher daher meist Nutzung von C oder C++
- Rust erlaubt die gleiche Effizienz ohne Risiko von…
  - Speicherzugriffsfehlern (use after free, double delete, …)
  - „Nasal Demons“ (*Undefined Behavior*)



Rust für DSP auf eingebetteten Systemen

- Rust setzt wie C++ auf *zero-cost Abstractions*
  - Abstraktionen die **nach Optimierung** kein Overhead **zur Laufzeit** haben
- Z. B., statt Register direkt per Adresse anzusprechen…

```
pub const PTR: *const gpioa::RegisterBlock = 0x4002_0000 as *const _;
```

- …können Register in Pins zerlegt werden und sichergestellt werden, dass keine Doppelbelegung oder *Data Races* auftreten, und dass sie korrekt konfiguriert sind (in/out, …)

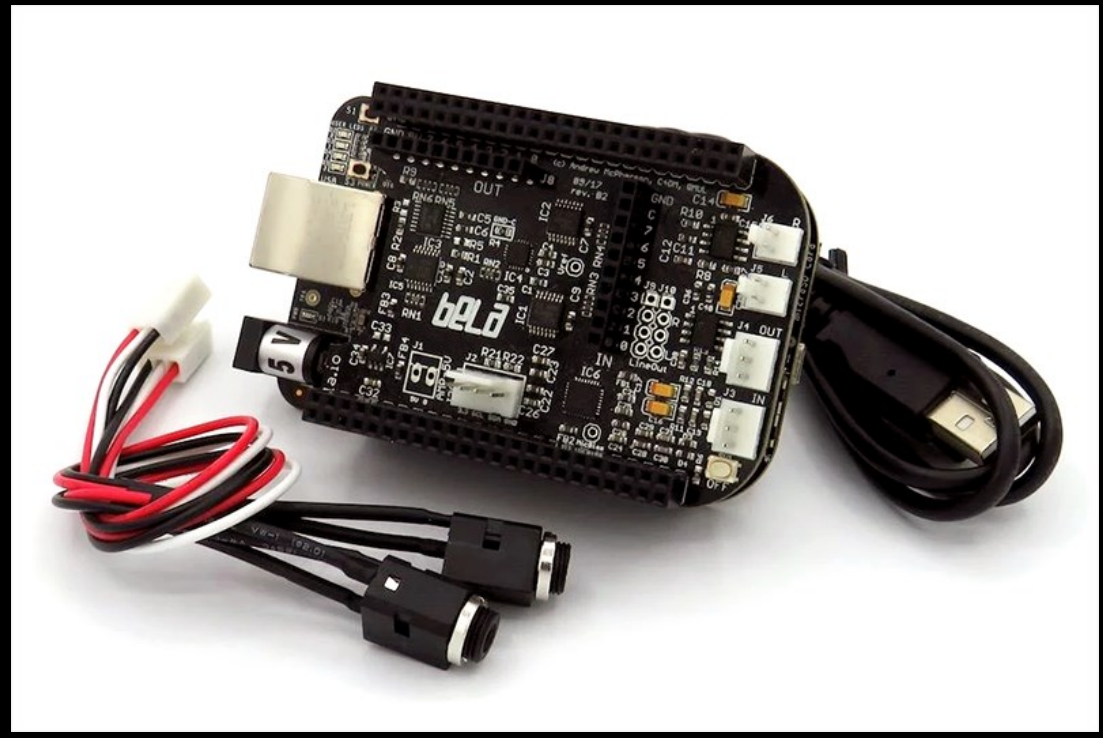
```
let com = serial::Port::new(pa2, pa3, USART2, &clocks, 9600.bps()).ok()?;
```

DSP mit Rust


left shift
let q13 = u32x4::from_cast(i32x4::from_cast(x << 1) >> 31);
// no wrapping negation on u32x4, so cast to i32x4 for negation
let xm = ((u32x4::from_cast(-i32x4::from_cast(x)) & q13)
| (x & !q13))
& 0x7fff_ffff;
// reciprocal multiplication is exact for powers of two
let xmf = f32x4::from_cast(xm) * (0x4000_0000 as f32).recip();
// no f32x4::powi
let ret = 1.5 * xmf - 0.5 * (xmf * xmf * xmf);
// no f32x4::copysign
((u32x4::from_bits(ret) & 0x7fff_ffff) | (x & 0x8000_0000))
.into_bits()
}
#[cfg(test)]
mod tests {
use approx::assert_abs_diff_eq;
use packed_simd::*;
fn reference_phasors() -> [u32; 12] {
let scale = (1u64 << 32) as f64;
let scale = |f| (scale * f) as u32;
[
scale(0.0), // 0
scale(0.125), // τ/8
scale(0.25), // τ/4
// 3τ/8

Entwicklungsumgebung

- BeagleBone® Black
  - 1GHz ARM® Cortex-A8
  - NEON FPU
  - 2×32-bit PRUs
- Bela I/O-Board
  - Je 2 16-bit DΣ-ADC/DACs
- Embedded Linux mit RTOS-Erweiterungen
  - UI auf Linux-Threads
  - IO auf RTOS-Threads



Echtzeit in der Praxis

- Bela unterstützt 2–128 Sample Blöcke
  - 45 µs bei 44100 kHz!
- Nur möglich durch RTOS-Threads mit höherer Priorität als Kernel-Threads
  - System-Funktionen wie Allokation oder Mutexes führen zu Priority Inversion!
- C-API erfordert *Foreign Function Interface (FFI)*

```rust
/// C-compatible trampoline function to call render function
extern "C" fn render_trampoline<Application, Constructor>(
    context: *mut bela_sys::BelaContext,
    user_data: *mut c_void,
) where
    Application: BelaApplication,
{
    let user_data = unsafe {
        &mut *(user_data as *mut UserData<
            Application, Constructor
        >)
    };
    if let UserData::Application(user_data) = user_data {
        // NOTE: cannot use catch_unwind safely here, as it
        // returns a boxed error (allocation in RT thread)
        let mut context = unsafe {
            Context::<RenderTag>::new(context)
        };
        user_data.render(&mut context);
    };
}
```

Echtzeit in der Praxis

- Begrenzung von `unsafe` auf begrenzte Code-Abschnitte
- Kann Rust auch vor Nutzung von Syscalls schützen?
- Nicht direkt, aber…
  - Man kann Echtzeit- und UI-Code in Crates trennen
  - `#[no_std]` in Echtzeit-Crate nutzen, wie auf µC
  - Großer Teil von `std` in `core` erhalten
  - Erhöht Wiederverwendbarkeit

```rust
#![no_std]

use core::simd::prelude::*;
use micromath::F32Ext;

// […]

pub fn render_sample(&mut self) -> f32 {
    // […]

    let signal = partial_signals
        .into_iter()
        .zip(gains)
        .map(|(partial, gain)| partial * f32x4::splat(gain))
        .sum::<f32x4>()
        .sum();

    // soft clipping
    let signal = signal.max(-1.0).min(1.0);
    let signal = 1.5 * signal - 0.5 * signal.powi(3);

    signal
}
```

Ausblick

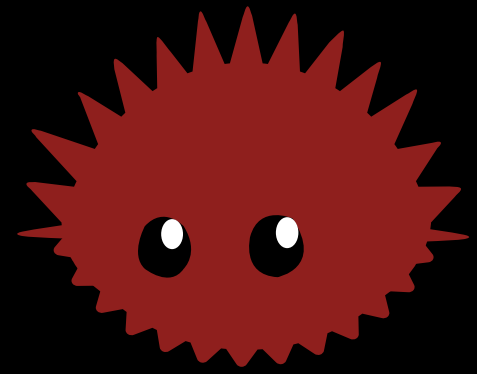
Ausblick

- Rust auf µC
  - Nutzung von `#[no_std]` und `unsafe`
  - HAL- und PAL-Crates
- Rust auf Linux mit RTOS-Erweiterungen
  - Effekte wie Priority Inversion
  - FFIs
- Praktische Signalverarbeitung auf modernen Prozessoren
  - Parallelität (Threads, SIMD und GPU)
  - Eigenheiten von DSP mit Gleitkomma- und Festkommaarithmetik

Wer mehr wissen will…

Wer mehr wissen will…

- Learn Rust
  https://www.rust-lang.org/learn
  - The Rust Programming Language
  - Rustlings
  - Rust by Example
  - Embedded Book
  - (The Rustonomicon)
- Real-Time Interrupt-driven Concurrency (RTIC)
  https://rtic.rs/1/book/en/
- knurling-rs
  https://github.com/knurling-rs



Wer mehr wissen will…

- Digitale Signalverarbeitung
  - Lyons, R. G. Understanding Digital Signal Processing. Pearson, 2010
  - Pirkle, W. C. Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 With DSP Theory. Routledge, 2019
- Bela
  - Embedded-Linux Plattform mit Xenomai RTOS-Erweiterungen für Audio DSP
    https://bela.io/

Dr.-Ing. Johannes S. Mueller-Roemer

h_da — 21. Dezember 2023

Folien