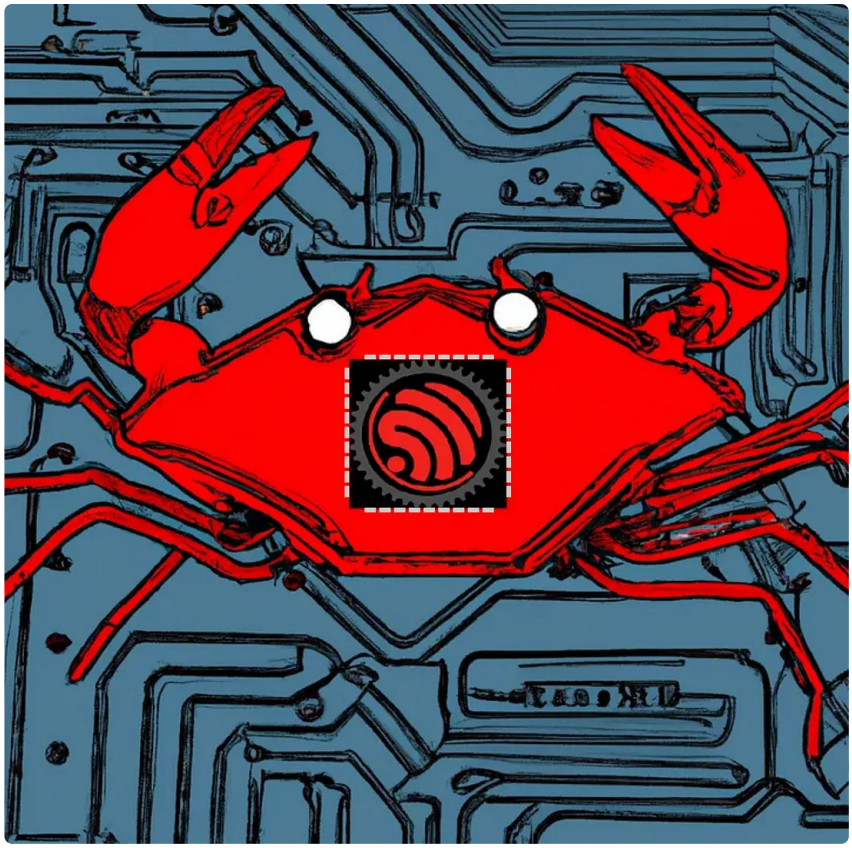
Rust + Embedded

een krachtig ontwikkelduo

Juraj Sadel (Espressif)

Met zijn focus op geheugen- en threadveiligheid is Rust een populaire taal voor het maken van betrouwbare en veilige software. Maar is Rust een slimme oplossing voor embedded toepassingen? Dit artikel laat zien dat Rust veel voordelen biedt ten opzichte van traditionele embedded ontwikkeltalen zoals C en C++, waaronder geheugenveiligheid, ondersteuning voor gelijktijdigheid en betere prestaties.



Bron: gegenereerd door DALL-E.

Rust is de populairste nieuwe taal ter wereld geworden, met elk jaar meer geïnteresseerden. De focus ligt op geheugen- en threadveiligheid met als doel betrouwbare en veilige software te maken. De ondersteuning van Rust voor gelijktijdigheid (concurrency) en parallel programmeren is vooral relevant voor embedded ontwikkeling, waar efficiënt gebruik van resources cruciaal is.

Het begin van Rust

Het oorspronkelijke idee voor een programmeertaal Rust is bij toeval ontstaan. In 2006 keerde Graydon Hoare terug naar zijn appartement in Vancouver, waar de lift weer eens niet werkte door een softwarebug. Hoare woonde op de 21ste verdieping en terwijl hij de trap opliep, begon hij te denken: “Wij computeraars kunnen niet eens liftsoftware maken die niet crasht!”. Dit bracht hem ertoe een nieuwe programmeertaal te ontwerpen. Hij hoopte dat het mogelijk zou zijn om kleine, snelle code te schrijven zonder geheugenbugs [1]. Als je geïnteresseerd bent in een meer gedetailleerde en technische geschiedenis van Rust, bezoek dan [2] en [3].

Bijna 18 jaar later is Rust de populairste nieuwe taal ter wereld geworden, met elk jaar meer geïnteresseerden. In het eerste kwartaal van 2020 waren er ongeveer 600.000 Rust-ontwikkelaars en in het eerste kwartaal van 2022 steeg het aantal naar 2,2 miljoen [4]. Techgiganten zoals Mozilla, Dropbox, Cloudflare, Discord, Facebook (Meta), Microsoft en anderen gebruiken Rust in hun codebase. In de afgelopen zes jaar bleef Rust de meest ‘geliefde’ programmeertaal [5].

Embedded ontwikkeling

Embedded ontwikkeling is niet zo populair als web- of desktopontwikkeling, en dit zijn een paar redenen waarom:

- Hardwarebeperkingen: de embedded systemen hebben waarschijnlijk beperkte hardware-resources, zoals prestaties en geheugen. Dit kan het moeilijker maken om software voor deze systemen te ontwikkelen.
- Beperkte en/of nichemarkt: de embedded markt is beperkter dan die voor web- en desktoptoepassingen, en dit kan het financieel minder interessant maken voor ontwikkelaars die gespecialiseerd zijn in embedded programmeren.
- Gespecialiseerde low-level kennis: gedegen kennis van de concrete hardware en low-level programmeertalen is noodzakelijk bij embedded ontwikkeling.
- Langere ontwikkelicycli: het ontwikkelen van software voor embedded systemen kan langer duren dan het ontwikkelen van software voor web- of desktopapplicaties, omdat de code moet worden getest en geoptimaliseerd voor de specifieke hardware-vereisten.
- Low-level programmeertalen: deze talen, zoals assembler of C, bieden weinig abstractie voor de ontwikkelaar en geven directe toegang tot hardware-resources en geheugen, wat zal leiden tot geheugenbugs.

Dit zijn maar een paar redenen waarom en hoe embedded ontwikkeling uniek is en niet zo bekend en lucratief voor jonge programmeurs als webontwikkeling. Als je gewend bent aan de meest gebruikte en moderne programmeertalen zoals Python, JavaScript of C# waar je niet elke processorcyclus en elke kilobyte in het geheugen hoeft te tellen, dan is het een ingrijpende verandering om met embedded ontwikkeling te beginnen. Het kan erg ontmoedigend zijn om in de embedded wereld te stappen, niet alleen voor beginners, maar ook voor ervaren web-/desktop-/mobiele ontwikkelaars. Daarom is het erg interessant en noodzakelijk om een moderne programmeertaal te hebben voor embedded ontwikkeling.

Waarom Rust?

Rust is een moderne en relatief jonge taal met een focus op geheugen- en threadveiligheid, met als doel betrouwbare en veilige software te maken. Daarnaast is de ondersteuning van Rust voor gelijktijdigheid en parallel programmeren bijzonder relevant voor embedded ontwikkeling, waar efficiënt gebruik van resources cruciaal is. De groeiende populariteit en het ecosysteem van Rust maken het een aantrekkelijke optie voor ontwikkelaars, vooral voor degenen die op zoek zijn naar een moderne taal die zowel efficiënt als veilig is. Dit zijn de belangrijkste redenen waarom Rust een steeds populairdere keuze wordt, niet alleen voor embedded ontwikkeling, maar vooral voor projecten waarbij veiligheid, beveiliging en betrouwbaarheid hoog in het vaandel staan.

De ondersteuning van Rust voor concurrency en parallellisme is vooral relevant voor embedded ontwikkeling, waar efficiënt gebruik van resources cruciaal is.

Voordelen (vergeleken met C/C++)

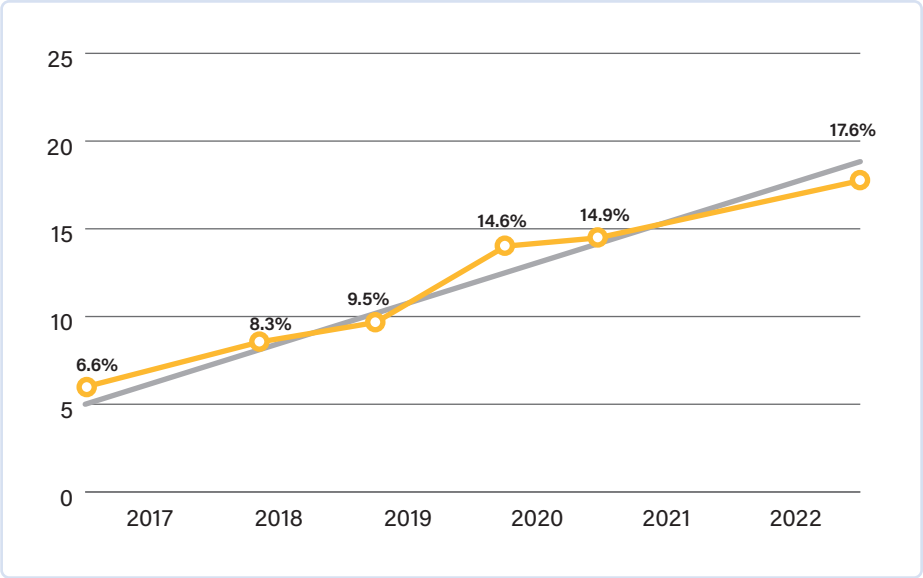
Laten we eens kijken naar de belangrijkste voordelen van Rust.

- Geheugenveiligheid: Rust biedt sterke garanties voor geheugenveiligheid via het eigendoms- en leensysteem, wat erg nuttig is bij het voorkomen van veel voorkomende geheugengerelateerde bugs zoals null-pointer verwijzingen of buffer overflows. Met andere woorden, Rust garandeert geheugenveiligheid tijdens het compileren door middel van het ownership en borrowing systeem. Dit is vooral belangrijk bij embedded ontwikkeling, waar geheugen-/resourcebeperkingen het moeilijker kunnen maken om zulke bugs te detecteren en op te lossen.
- Concurrency (gelijktijdigheid): Rust biedt uitstekende ondersteuning voor zero-cost abstracties en veilige concurrency en multithreading, met een ingebouwde `async/await` syntaxis en een krachtig type-systeem dat veel voorkomende concurrency-bugs zoals data races voorkomt. Dit kan het makkelijker maken om veilige en efficiënte parallelle code te schrijven, niet alleen in embedded systemen.
- Prestaties: Rust is ontworpen voor hoge prestaties en kan op dat gebied wedijveren met C en C++, terwijl het nog steeds de veiligheid van het geheugen garandeert en ondersteuning voor gelijktijdigheid biedt.
- Leesbaarheid: de syntaxis van Rust is ontworpen om leesbaarder en minder foutgevoelig te zijn dan C en C++, met functies zoals patroonherkenning, type-inferentie en functionele programmeerconstructies. Dit kan het makkelijker maken om code te schrijven en te onderhouden, vooral voor grotere en complexere projecten.
- Groeiend ecosysteem: Rust heeft een groeiend ecosysteem van bibliotheken (crates), tools en resources voor (niet alleen) embedded ontwikkeling, wat het makkelijker kan maken om met Rust aan de slag te gaan en de nodige ondersteuning en resources voor een bepaald project te vinden.
- Packet manager en build system: de Rust-distributie bevat een officiële tool genaamd Cargo, die wordt gebruikt om het build-, test- en publicatieproces te automatiseren, samen met het maken van een nieuw project en het beheren van de afhankelijkheden.

Nadelen (vergeleken met C/C++)

Aan de andere kant is Rust geen perfecte taal en heeft het ook enkele nadelen ten opzichte van andere programmeertalen (niet alleen ten opzichte van C en C++).

- Leercurve: Rust heeft een steilere leercurve dan veel andere programmeertalen, waaronder C. De unieke functies, zoals het al genoemde ownership en borrowing, kunnen enige tijd kosten om te begrijpen en eraan te wennen en vormen daarom een grotere uitdaging om met Rust aan de slag te gaan.
- Compilatietijd: het geavanceerde type-systeem en de borrow checker van Rust kunnen resulteren in langere compilatietijden in vergelijking met andere talen, vooral bij grote projecten.
- Tooling: hoewel het ecosysteem van Rust snel groeit, heeft het misschien nog niet hetzelfde niveau van tool-ondersteuning als meer gevestigde



Figuur 1. Het groeiende percentage ontwikkelaars dat met Rust wil werken (bron: Yalantis [4]).

programmeertalen. C en C++ bestaan bijvoorbeeld al tientallen jaren en hebben een enorme codebase. Dit kan het moeilijker maken om de juiste tools voor een bepaald project te vinden en te gebruiken.

- Gebrek aan low-level controle: de veiligheidsfuncties van Rust kunnen soms de low-level controle beperken tot C en C++. Dit kan het uitdagender maken om bepaalde low-level optimalisaties uit te voeren of direct met hardware te communiceren, maar het is mogelijk.
- Omvang van de community: Rust is nog steeds een relatief nieuwe programmeertaal vergeleken met meer gevestigde talen als C en C++, wat betekent dat het een kleinere community van ontwikkelaars en bijdragers heeft, en minder resources, bibliotheken en tools.

Over het algemeen biedt Rust veel voordelen ten opzichte van traditionele embedded ontwikkeltalen zoals C en C++, waaronder geheugenveiligheid, ondersteuning voor concurrency, prestaties, leesbaarheid van de code en een groeiend ecosysteem. Hierdoor wordt Rust een steeds populairdere keuze voor embedded ontwikkeling, vooral voor projecten waar veiligheid, beveiliging en betrouwbaarheid de hoogste prioriteit hebben. De nadelen van Rust in vergelijking met C en C++ hebben te maken met het feit dat Rust een relatief nieuwe taal is, en met de unieke mogelijkheden. Veel ontwikkelaars vinden echter dat de voordelen van Rust het een aantrekkelijke keuze maken voor bepaalde projecten.

Hoe kan Rust draaien?

Er zijn verschillende manieren om op Rust gebaseerde firmware te draaien, afhankelijk van de omgeving en de vereisten van de toepassing. De op Rust gebaseerde firmware kan meestal in twee modi gebruikt worden: *hosted-environment* of *bare-metal*. Laten we daar eens naar kijken.

Wat is hosted-environment?

In Rust komt de *hosted-environment* dicht in de buurt van een normale PC-omgeving [6], wat betekent dat je de beschikking krijgt over een besturingssysteem. Met het besturingssysteem is het mogelijk om de Rust-standaardbibliotheek (*std*) te bouwen. [7] De *std* verwijst naar de standaardbibliotheek, die kan worden gezien als een verzameling modules en typen die bij elke installatie van Rust worden meegeleverd. De *std* biedt een set van meerdere functionaliteiten voor het bouwen van Rust-programma's, waaronder datastructuren, netwerken, mutexen en andere synchronisatieprimitieven, invoer/uitvoer en meer.
Met de *hosted-environment* benadering kun je de functionaliteit van het op C gebaseerde development framework ESP-IDF [8] gebruiken, omdat het een *newlib*-omgeving [9] biedt die 'krachtig' genoeg is om de Rust-standaardbibliotheek bovenop te bouwen. Met andere woorden, met de *hosted-environment* (soms gewoon *std* genoemd) benadering, gebruiken we de ESP-IDF als een besturingssysteem en bouwen we de Rust-applicatie daar bovenop. Op deze manier kunnen we gebruik maken van alle standaardbibliotheekfuncties die hierboven zijn genoemd en kunnen we ook al C-functionaliteit implementeren vanuit de ESP-IDF API.
In **listing 1** kun je zien hoe een blinky-voorbeeld [10] dat draait bovenop ESP-IDF (FreeRTOS) eruit kan zien. Meer voorbeelden zijn te vinden in *esp-idf-hal* [11].

Waarom je hosted environment zou gebruiken

- Uitgebreide functionaliteit: als je embedded systeem veel functionaliteit vereist, zoals ondersteuning voor netwerkprotocollen, file I/O of complexe datastructuren, dan zul je waarschijnlijk de hosted-environment aanpak willen gebruiken omdat *std*-bibliotheken een breed scala aan functionaliteiten bieden die gebruikt kunnen worden om relatief snel en efficiënt complexe applicaties te bouwen.
- Overdraagbaarheid: de *std*-crate biedt een gestandaardiseerde set API's die gebruikt kunnen worden op verschillende platformen en architecturen, waardoor het gemakkelijker wordt om code te schrijven die overdraagbaar en herbruikbaar is.
- Snelle ontwikkeling: de *std*-crate biedt een rijke set aan functionaliteit die gebruikt kan worden om snel en efficiënt applicaties te bouwen, zonder zorgen over low-level details.

Wat is bare-metal?

Bare-metal betekent dat we geen besturingssysteem hebben om mee te werken. Wanneer een Rust-programma gecompileerd is met het `no_std` attribuut, betekent dit dat het programma geen toegang heeft tot bepaalde mogelijkheden. Dit betekent niet noodzakelijkerwijs dat je geen netwerken of complexe datastructuren kunt gebruiken met `no_std`. Je kunt zonder `std` alles doen wat je met `std` kunt doen, maar het is complexer en uitdagender. `no_std` programma's vertrouwen op een set kernfuncties van de taal die beschikbaar zijn in alle Rust-omgevingen, bijvoorbeeld datatypes, controlestructuren of low-level geheugenbeheer. Deze aanpak is handig voor embedded programmeren waar geheugengebruik vaak beperkt is en low-level controle over hardware nodig is.
In **listing 2** zie je een mogelijk bare-metal blinky-voorbeeld [12] (dus zonder besturingssysteem). Meer voorbeelden zijn te vinden in *esp-hal* [13].

Waarom je bare-metal zou gebruiken

- Kleine geheugen-footprint: als je embedded systeem beperkte resources heeft en een kleine geheugen-footprint moet hebben, zul je waarschijnlijk *bare-metal* willen gebruiken, omdat *std*-functies het uiteindelijke binaire bestand veel groter maken en meer compilatietijd nodig hebben.
- Directe controle over de hardware: als je embedded systeem meer directe controle over de hardware vereist, zoals low-level apparaatdrivers of toegang tot gespecialiseerde hardwarefuncties, dan zul jewaarschijnlijk *bare-metal* willen gebruiken omdat *std* abstracties toevoegt die het moeilijker kunnen maken om direct met de hardware te communiceren.
- Real-time beperkingen of tijdkritische toepassingen: als je embedded systeem real-time prestaties of reacties met geringe latentie vereist; *std* kan onvoorspelbare vertragingen en overhead introduceren die de real-time prestaties kunnen beïnvloeden.
- Speciale eisen: *bare-metal* maakt meer maatwerk en fijnmazige controle over het gedrag van een applicatie mogelijk, wat nuttig kan zijn in gespecialiseerde of niet-standaard omgevingen.

Moet je overstappen van C naar Rust?

Als je een nieuw project of een taak begint waarbij geheugenveiligheid of concurrency vereist is, kan het het overwegen waard zijn om van C naar Rust over te stappen. Als je project echter al goed is opgezet en functioneel is in C, wegen de voordelen van een overstap naar Rust misschien niet op tegen de kosten van het herschrijven en opnieuw testen van je hele codebase. In dat geval kun je overwegen om de huidige C-codebase te behouden en te beginnen met het schrijven en toevoegen van nieuwe functies, modules en functionaliteit in Rust – het is relatief eenvoudig om C-functies aan te roepen vanuit Rust-code. Het is ook mogelijk om ESP-IDF componenten in Rust te schrijven [14]. De uiteindelijke beslissing om van C naar Rust over te stappen moet gebaseerd zijn op een zorgvuldige evaluatie van je specifieke behoeften en de afwegingen die daarbij komen kijken.

230569-03 (vertaling: Jelle Aarnoudse)

Over de auteur

Juraj Sadel is een embedded software ontwikkelaar met een passie voor Rust en toegewijd aan het verbeteren van embedded systemen. Hij is ook een actief lid van het Rust-team, draagt expertise bij aan de community en is enthousiast over de geavanceerde technologie van Espressif.

Vragen of opmerkingen?

Hebt u technische vragen of opmerkingen hebt naar aanleiding van dit artikel? Stuur een e-mail naar de auteur via juraj.sadel@espressif.com of naar de redactie van Elektor via redactie@elektor.com.

Listing 1. Blinky-voorbeeld met hosted environment en std.

```
// Import peripherals we will use in the example
 use esp_idf_hal::delay::FreeRtos;
 use esp_idf_hal::gpio::*;
 use esp_idf_hal::peripherals::Peripherals;

 // Start of our main function i.e entry point of our example
 fn main() -> anyhow::Result<()> {
     // Apply some required ESP-IDF patches
     esp_idf_sys::link_patches();

     // Initialize all required peripherals
     let peripherals = Peripherals::take().unwrap();

     // Create led object as GPIO4 output pin
     let mut led = PinDriver::output(peripherals.pins.gpio4)?;

     // Infinite loop where we are constantly turning ON and OFF the LED every 500ms
     loop {
         led.set_high()?;
         // we are sleeping here to make sure the watchdog isn't triggered
         FreeRtos::delay_ms(1000);

         led.set_low()?;
         FreeRtos::delay_ms(1000);
     }
 }
```

Listing 2. Bare-metal blinky-voorbeeld.

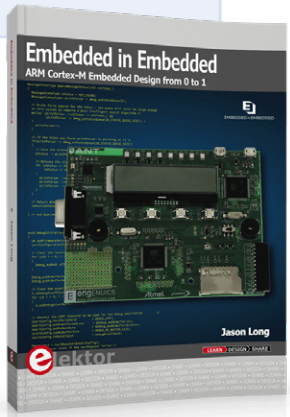
```
#![no_std]
 #![no_main]

 // Import peripherals we will use in the example
 use esp32c3_hal::{
     clock::ClockControl,
     gpio::IO,
     peripherals::Peripherals,
     prelude::*,
     timer::TimerGroup,
     Delay,
     Rtc,
 };
 use esp_backtrace as _;

 // Set a starting point for program execution
 // Because this is `no_std` program, we do not have a main function
 #[entry]
 fn main() -> ! {
     // Initialize all required peripherals
     let peripherals = Peripherals::take();
     let mut system = peripherals.SYSTEM.split();
     let clocks = ClockControl::boot_defaults(system.clock_control).freeze();

     // Disable the watchdog timers. For the ESP32-C3, this includes the Super WDT,
     // the RTC WDT, and the TIMG WDTs.
     let mut rtc = Rtc::new(peripherals.RTC_CNTL);
     let timer_group0 = TimerGroup::new(
         peripherals.TIMG0,
         &clocks,
         &mut system.peripheral_clock_control,
     );
     let mut wdt0 = timer_group0.wdt;
     let timer_group1 = TimerGroup::new(
         peripherals.TIMG1,
         &clocks,
         &mut system.peripheral_clock_control,
     );
     let mut wdt1 = timer_group1.wdt;

     rtc.swd.disable();
     rtc.rwdt.disable();
     wdt0.disable();
     wdt1.disable();

     // Set GPIO4 as an output, and set its state high initially.
     let io = IO::new(peripherals.GPIO, peripherals.IO_MUX);
     // Create led object as GPIO4 output pin
     let mut led = io.pins.gpio5.into_push_pull_output();

     // Turn on LED
     led.set_high().unwrap();

     // Initialize the Delay peripheral, and use it to toggle the LED state in a
     // loop.
     let mut delay = Delay::new(&clocks);
```

```
    // Infinite loop where we are constantly turning ON and OFF the LED every 500ms
    loop {
        led.toggle().unwrap();
        delay.delay_ms(500u32);
    }
}
```

Gerelateerde producten

- **J. Long, Embedded in Embedded (Elektor 2018)**
  Book: www.elektor.nl/18876
  E-book: www.elektor.nl/18877

- **A. He and L. He, Embedded Operating System (Elektor 2020)**
  Book: www.elektor.nl/19228
  E-book: www.elektor.nl/19214



WEBLINKS

[1] MIT Technology Review, "How Rust went from a side project to the world's most-loved programming language," 2023: https://technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language
[2] Rust Blog, "Announcing Rust 1.0," 2015: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html
[3] Rust Blog, "4 years of Rust," 2019: https://blog.rust-lang.org/2019/05/15/4-Years-Of-Rust.html
[4] Yalantis, "The state of the Rust market in 2023" : https://yalantis.com/blog/rust-market-overview
[5] Stack Overflow, "Stack Overflow Developer Survey," 2021: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted
[6] The Embedded Rust Book: https://docs.rust-embedded.org/book/intro/no-std.html#hosted-environments
[7] The Rust Standard Library (std): https://doc.rust-lang.org/std
[8] ESP-IDF: https://github.com/espressif/esp-idf
[9] Newlib-bibliotheek: https://sourceware.org/newlib
[10] Blinky-voorbeeld bovenop ESP-IDF: https://github.com/esp-rs/esp-idf-hal/blob/master/examples/blinky.rs
[11] ESP-IDF-HAL: https://github.com/esp-rs/esp-idf-hal/tree/master/examples
[12] Bare-metal blinky-voorbeeld: https://github.com/esp-rs/esp-hal/blob/main/esp32c3-hal/examples/blinky.rs
[13] ESP-HAL: https://github.com/esp-rs/esp-hal/tree/main
[14] ESP-IDF componenten in Rust: https://github.com/espressif/rust-esp32-example