Nieuwe embedded-ontwikkelingen

Rust en het up-to-date houden van IoT-implementaties

Embedded Development

Stuart Cording (Elektor)

Terwijl de embedded systemen die op de markt verschijnen het doen lijken alsof de technologie snel vooruitgaat, is de industrie zelf in vergelijking daarmee traag. Daarom was het een schok toen Raspberry Pi, de bekende fabrikant van single-board computers, de RP2040-microcontroller (MCU) uitbracht met zijn dubbele Cortex-M0+ cores en zonder onboard flash. Dual-core in deze klasse van apparaten is ongehoord. Maar dat is dan ook het spannendste wat er is. Vooruitgang in de wereld van embedded systemem is normaliter afgemeten, doelgericht en weloverwogen. Maar er zijn verschillende zaken gaande die de ontwikkeling van embedded systemen in het komende decennium kunnen veranderen, zoals we hieronder zullen zien.

De ontwikkeling van embedded software is zonder C bijna ondenkbaar. Toen assembler te omslachtig werd om volledige toepassingen te ontwikkelen, werd het door C verdrongen, behalve in speciale gevallen waar geoptimaliseerde, handgecodeerde assembler de enige optie was. De taal, ontwikkeld door Dennis Ritchie [1] bij Bell Labs, biedt voldoende flexibiliteit om complexe toepassingen te ontwikkelen, maar geeft ook gemakkelijk toegang tot registers. Dat is van cruciaal belang voor het schrijven van compacte microcontrollercode die registertoegang in interrupt-routines afhandelt. Het is ook eenvoudig om taken zoals bitmanipulatie van registers uit te voeren. En in tegenstelling tot assemblercode is de resulterende code gemakkelijker te lezen. C blijft een geliefde programmeertaal, met (in enquêtes en marktanalyses) een plaats in de top drie van programmeertalen – zie **figuur 1** [2][3].

C is oud

C, dat in 1972 werd ontwikkeld, is nu echter 50 jaar oud. Het heeft een reeks beperkingen die welbekend zijn en waarvan vele te maken hebben met het gebruik van pointers. Terwijl pointers het voor embedded ontwikkelaars gemakkelijk maken om registers te benaderen, kunnen ze ook resulteren in ongewenste *out-of-bound* geheugentoegang. Vergeleken met modernere programmeertalen voeren C-compilers bovendien relatief weinig controles van de code uit. Als gevolg daarvan worden ongebruikte variabelen gewoon genegeerd, iets wat op een fout in de code kan wijzen.

Om ervoor te zorgen dat onveilige C-code niet in embedded systemen terecht komt, gebruiken ontwikkelaars coderingsstandaarden zoals MISRA C [4]. Deze standaard kwam tot stand toen C in de auto-industrie steeds belangrijker werd als programmeertaal voor embedded

systemen. C++ lost een aantal van de pointer-problemen van C op met referenties [5] die niet kunnen worden veranderd om naar een ander object te verwijzen, niet NULL kunnen zijn en die bij het aanmaken geïnitialiseerd moeten worden. Desondanks heeft AUTOSAR, een samenwerkingsverband van ontwikkelaars van systemen voor de automobielindustrie, in een document van enkele honderden bladzijden richtlijnen opgesteld voor het gebruik van C++ voor veiligheidsgerelateerde toepassingen [6]. Hoewel bekwaamheid in deze gevestigde talen dus essentieel is voor ontwikkelaars van embedded systemen, moge duidelijk zijn dat elke taal genoeg tekortkomingen heeft om richtlijnen nodig te maken waarmee veel voorkomende programmeerfouten worden vermeden.

Rust verschijnt op het toneel

Rust is ontstaan als een potentiële mededinger die zichzelf aanprijst als zeer geschikt voor het ontwikkelen van veilige systemen. Het begon als een privéproject van Graydon Hoare in 2006, en werd omstreeks 2010 een door zijn werkgever, Mozilla Research, gesponsord project. In 2021 werd de Rust Foundation [7] opgericht nadat herstructurering van het bedrijf gevolgen had voor het Rust-ontwikkelteam.
Wat Rust anders maakt is dat bij het compileren veel problemen worden ontdekt en gemarkeerd, vaak problemen die C/C++ compilers zouden negeren. Er is ook een *ownership*-systeem voor variabelendeclaraties geïmplementeerd. Dit maakt gebruik van een *borrow-checker* om het misbruik van variabelen al tijdens het compileren te voorkomen. Bovendien moeten lees- en schrijftoegang tot variabelen die per referentie worden doorgegeven, expliciet worden gedeclareerd. De syntax van Rust lijkt veel op die van C en C++, met gebruik van accolades rond functies en de bekende *control keywords*.

Vanwege de nadruk op veilige code zijn veel inspanningen gedaan om Rust ingang te doen vinden in de *bare metal community* van embedded software-ontwikkelaars. Vanwege de aard van embedded programmering kan de statische controle door de compiler echter problemen veroorzaken bij het implementeren van sommige soorten code. Om dit te omzeilen kunnen sommige delen van de code als *unsafe* worden gemarkeerd om bijvoorbeeld *pointer dereferencing* toe te staan. De gedachte hierachter is dat, door code secties expliciet als *unsafe* te definiëren, de code het omzeilen van de regels van Rust verduidelijkt.

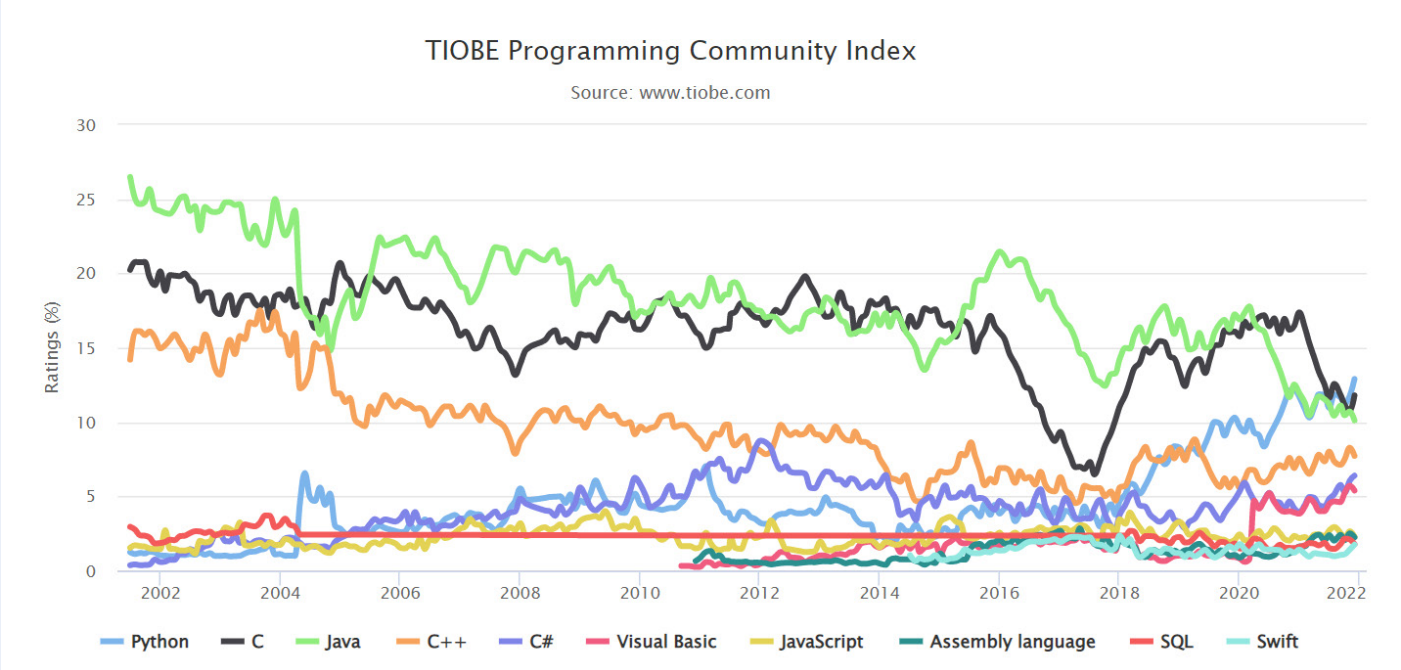
Rust in de praktijk

De beste manier om Rust te leren kennen is met de Raspberry Pi. De installatie is eenvoudig, conform de richtlijnen op de website rustup.rs [8]. Vanaf de opdrachtregel voert u het volgende commando in en volgt u de instructies:

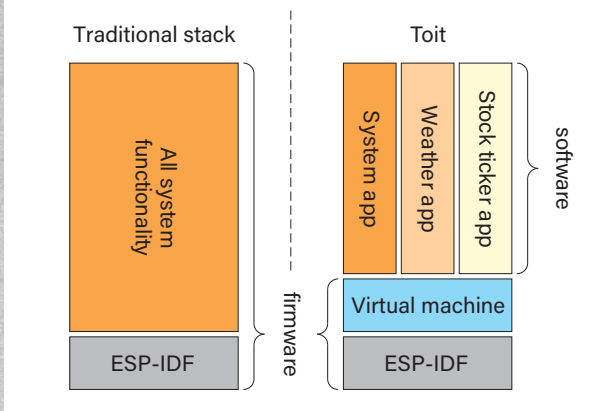
```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

In tegenstelling tot C/C++. dat binaire bestanden genereert, staat de uitvoer van RUST bekend als een *crate*. De package manager Cargo wordt gebruikt om het commandoregel-compilatieproces te vereenvoudigen. Cargo slaat de gegevens op die nodig zijn om de crate te genereren en stelt de ontwikkelaar in staat om de packages te definiëren die nodig zijn om de crate te bouwen. Door Cargo als volgt vanaf de opdrachtregel aan te roepen, wordt een nieuw Rust-project gegenereerd:

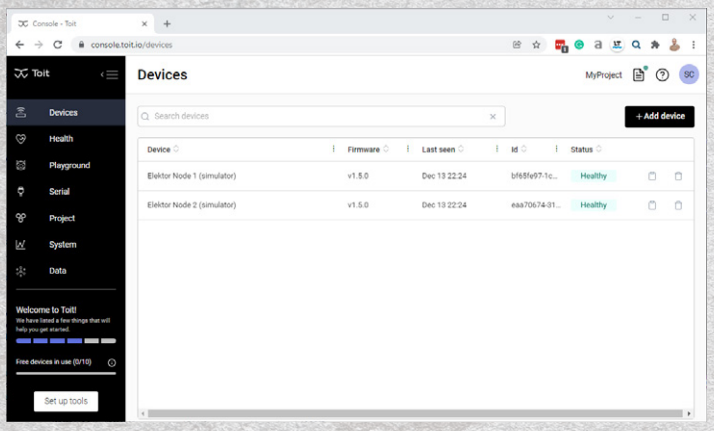
```
cargo new rust_test_project
```



Figuur 1. C is en blijft populair als programmeertaal en staat in enquêtes en marktanalyses regelmatig bij de top drie (bron: www.tiobe.com).



Figuur 2. Toit voert IoT-code uit als app op een virtuele machine die op een ESP32 draait (bron: Toit).



Figuur 3. De Toit Console in een browser, hier met twee gesimuleerde ESP32-nodes.

Als u de map van het nieuwe project opent, ziet u een bestand met de naam `Cargo.toml`. Dit bestand moet worden aangepast zoals vereist. Voor een eenvoudige Raspberry Pi-applicatie die een LED laat knipperen die is aangesloten op een GPIO, moet een geschikte crate-dependency voor de toegang tot de GPIO-pinnen worden gedefinieerd. Crates worden gedeeld op crates.io [9], een platform gewijd aan de crates van de Rust-community. Een geschikte crate is *rppal*, die kan worden gevonden via de zoekfunctie. Het platform laat zien of de crate op dit moment gebouwd kan worden, geeft het versienummer, en biedt documentatie en voorbeeldcode. De naam van de crate en het versienummer worden vervolgens toegevoegd als *dependencies* in `Cargo.toml` (**listing 1**).

Listing 1. De rppal-dependency toevoegen aan Cargo.toml in een Rust project.

```
[package]
name = "rust_test_project"
version = "0.1.0"
edition = "2021"
[dependencies]
rppal = "0.13.1"
```

In de `src`-directory treft de ontwikkelaar een eenvoudig *Hello World*-voorbeeldproject aan in het Rust-broncodebestand `main.rs`. Wanneer deze code wordt vervangen door **listing 2** zal een LED die is aangesloten op GPIO 23 (pin 16 van de Raspberry Pi-header) tien keer knipperen. De compilatie wordt uitgevoerd vanaf de opdrachtregel met `cargo build`, en de crate wordt uitgevoerd met `cargo run`. Iets wat het gebruik van Rust op sommige microcontrollers belemmert is de noodzaak om een LLVM-toolchain te gebruiken. Als deze beschikbaar is, kunt u beginnen, met behulp van een van de verschillende online-tutorials. Er is een voorbeeld voor de BBC micro:bit [10] dat laat zien dat, zodra u de Rust-syntaxis begrepen hebt (**listing 3**, [11]), dit sterk lijkt op het schrijven van code in C/C++. Er is ook een ander voorbeeld, voor de STM32 [12].

Listing 2. Een knipperende LED in Rust.

Een knipper-LED in Rust is vergelijkbaar met C-code. Dit voorbeeld is gebaseerd op code die wordt meegeleverd met de rppal-crate

```
use std::error::Error;
use std::thread;
use std::time::Duration;
use rppal::gpio::Gpio;
use rppal::system::DeviceInfo;
// Gpio uses BCM pin numbering. BCM GPIO 23 is tied to physical pin 16.
const GPIO_LED: u8 = 23;
fn main() -> Result<(), Box<dyn Error>> {
    let mut n = 1;
    println!("Blinking an LED on a {}.", DeviceInfo::new()?.model());
    let mut pin = Gpio::new()?.get(GPIO_LED)?.into_output();
    while n < 11 {
        // Blink the LED by setting the pin's logic level high for 500 ms.
        pin.set_high();
        thread::sleep(Duration::from_millis(500));
        pin.set_low();
        thread::sleep(Duration::from_millis(500));
        n += 1;
    }
    Ok(())
}
```

Listing 3. Vereenvoudigde code-snippet die de toestand van een ingangspin op de BBC micro:bit controleert met behulp van Rust.

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

extern crate panic_abort;
extern crate cortex_m_rt as rt;
extern crate microbit;

use rt::entry;
use microbit::hal::prelude::*;

#[entry]
fn main() -> ! {
     if let Some(p) = microbit::Peripherals::take() {
        // Split GPIO
        let mut gpio = p.GPIO.split();

        // Configure button GPIO as input
        let button_a = gpio.pin17.into_floating_input();

        // loop variablew
        let mut state_a_low = false;

        loop {
             // Get button state
             let button_a_low = button_a.is_low();

            if button_a_low && !state_a_low {
                 // Output message
             }

            if !button_a_low && state_a_low {
                 // Output message
             }

            // Store button states
             state_a_low = button_a_low;
         }
     }
     panic!("End");
 }
```

Is Rust de toekomst?

Dus, wat houdt het gebruik van Rust in embedded systemen tegen? Wel, als u op zoek bent naar een robuuste programmeertaal die geschikt is voor gebruik in veiligheidskritische systemen, dan kunt u al terecht bij Ada [13]. Ondanks het feit dat het al 40 jaar oud is, is het er nog niet in geslaagd om C te verdringen, ondanks het feit dat het specifiek is ontwikkeld voor gebruik in real-time embedded systemen. Een ander aspect is het jarenlange succes van C en de aanwezigheid van talloze ontwikkelaars, tools en bibliotheken met code.
De beschikbaarheid van de Crate package manager zou echter kunnen helpen om de aanvaarding van Rust te versnellen. Het bijhouden van perifere bibliotheken van halfgeleiderleveranciers die in C/C++ zijn geschreven kan een ondoorzichtige uitdaging zijn, dus de expliciete definitie van de versie van de crates die in `Cargo.toml` worden gebruikt, kan als een belangrijke verbetering worden beschouwd. Het kan ook het leven vereenvoudigen van MCU-fabrikanten die proberen hun enorme productportfolio's te ondersteunen. Ada heeft echter al gereageerd met de release in 2020 van hun *Alire* package manager [14]. En net als Rust biedt Ada al ondersteuning voor de BBC micro:bit, mocht u de twee talen willen vergelijken op dezelfde MCU [15].

IoT-apparaten up-to-date houden

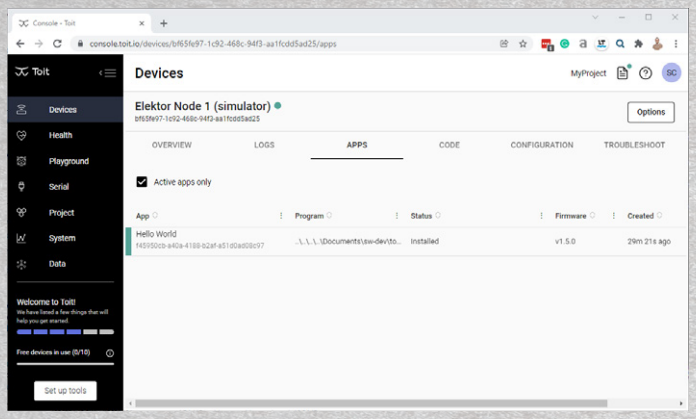
Nu steeds meer embedded apparaten op netwerken zijn aangesloten, is de grootste uitdaging om ze up-to-date te houden met de meest recente firmware-versie. Traditioneel moeten firmware-updates draadloos worden gedownload waarbij de binaire code in flash wordt geprogrammeerd met behulp van een on-chip bootloader die in een beschermd gebied van het apparaat is opgeslagen. Het risico bestaat echter dat deze nieuwe codeversie niet correct wordt geïmplementeerd, waardoor het product onbruikbaar wordt. Een andere mogelijkheid is dat de code wel werkt, maar dat een softwarefout in de nieuwe code verhindert dat toekomstige updates worden geïmplementeerd, waardoor apparaten in het veld kwetsbaar blijven. Hoewel het grootste deel van de toepassing correct functioneert, wordt het toestel onbruikbaar, misschien door een fout in een enkele regel code.
Virtuele machines zijn al jaren een standaardmethode voor het implementeren van meerdere apps of besturingssystemen op servers. Met een virtuele machine kan een besturingssysteem worden uitgevoerd terwijl het lijkt alsof het op specifieke hardware draait. Mocht het besturingssysteem catastrofaal crashen, dan heeft dat alleen gevolgen voor de betreffende virtuele machine, niet voor de overige systemen die op de server draaien. Desktopgebruikers zijn bekend met virtualisatiesoftware zoals VirtualBox, VMware en Parallels, waarmee ze software of alternatieve besturingssystemen kunnen uitproberen zonder hun primaire machine aan risico's bloot te stellen.

Updating IoT node-firmware: aan de slag met Toit

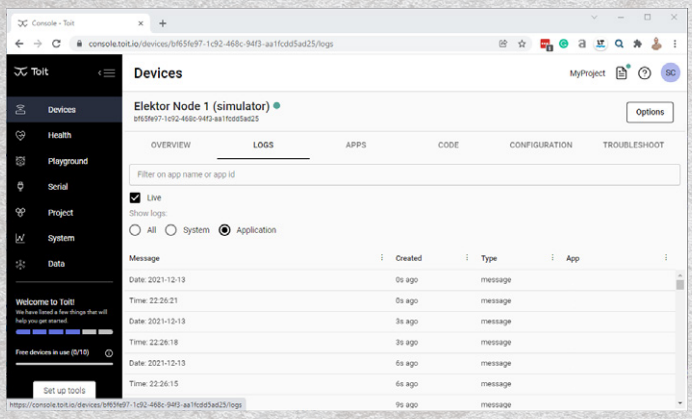
Het team van Toit, een Deens bedrijf dat is opgericht door voormalige Google-technici, vroeg zich af waarom geen virtualisatie werd ingezet op microcontrollers voor IoT-toepassingen. Deze hebben immers regelmatige updates nodig om bugs te verhelpen en kunnen zelfs wijzigingen nodig hebben om verbeteringen te ondersteunen die zijn doorgevoerd in de clouddiensten waarmee ze communiceren. En uiteraard wil niemand IoT-nodes in het veld handmatig flashen als de apparaten door een mislukte update vastlopen.
Voor de eerste experimenten kozen ze voor de ESP32 van Espressif, om precies te zijn de ESP32-WROOM-32E met een dual-core 32-bit Xtensa LX6 microprocessor, 520 KB SRAM en 4 MB flash. Het platform beschikt al over het ESP-IDF (Espressif IoT Development Framework), dus het is klaar voor gebruik als IoT-node. Toit heeft vervolgens een virtuele machine gebouwd (**figuur 2**) waarmee apps veilig kunnen worden uitgevoerd. Apps worden geschreven in Toit, een high-level objectgeoriënteerde en veilige taal.

Toit testen met gesimuleerde ESP32-nodes

Het beheren en implementeren van apps op een verzameling IoT-nodes gebeurt met behulp van de Toit Console in combinatie met Visual Studio Code van Microsoft. Voor wie het platform slechts wil uitproberen, kan met de Toit Console gesimuleerde ESP32-devices gebruiken. Na het installeren van de Toit-uitbreiding voor Visual Studio Code kunnen apps lokaal worden geschreven en gesimuleerd of worden geïnstalleerd op apparaten die op de Console zijn aangesloten.
Toit-apps worden door een YAML-bestand begeleid. Dit markup-be-



Figuur 4. De succesvol geïnstalleerde Hello World-app op een gesimuleerde node.



Figuur 5. De LOGS-uitvoer toont de tijd en datum die elke drie seconden door de Toit-app wordt geleverd, zoals gedefinieerd in het YAML-bestand.

stand wordt gebruikt om de naam van de Toit-app en het broncode-bestand te declareren en triggers te definiëren. Deze kunnen de app uitvoeren na het opstarten of met ingestelde tijdsintervallen.

Voor de installatie van apps of updates hoeft het ESP32-device niet te worden gedeactiveerd of uitgeschakeld. In plaats daarvan werkt de virtuele machine de app ter plekke bij en activeert deze volgens de in het YAML-bestand opgegeven instellingen. Als de app op de een of andere manier crasht, bijvoorbeeld door een stack overflow, blijven de overige apps zonder problemen draaien [16]. Zulke fouten kunnen worden onderkend via de Console door het protocol te bestuderen.

Een eenvoudige *Hello World*-app die de tijd en datum weergeeft, is te zien in **listing 4**; het bijbehorende YAML-bestand in **listing 5**. **Figuur 3** toont twee gesimuleerde nodes in de Console, terwijl in **figuur 4** de succesvol geïnstalleerde *Hello World*-app te zien is. Tenslotte toont **figuur 5** de uitvoer van datum en tijd om de drie seconden, zoals gedefinieerd met de `on_interval: “3s”` trigger in het YAML-bestand. De projectcode is gemaakt in Visual Studio Code en met behulp van de Toit-extensie geïnstalleerd op de betreffende node, gekoppeld aan het Console-account van de gebruiker (**figuur 6**).

Op het eerste gezicht lijkt de Toit-benadering een beetje restrictief. De virtualisatie-omgeving staat alleen controle toe van de GPIO’s, de seriële interface (UART), SPI of I²C. Bij veel IoT-nodes die sensorgegevens verzamelen en deze naar de cloud doorsturen, lijkt het echter voldoende flexibel te zijn. Toit heeft ook een eigen package manager (register) [17], die drivers levert voor een reeks sensoren, invoerapparatuur, LCD’s en andere zaken. De omgeving ondersteunt ook de low power-modus van de ESP32, waardoor de controller jarenlang op twee AA-batterijen moet kunnen werken [18]. Als een app wordt geactualiseerd terwijl de IoT-node in deep sleep-modus verkeert, doet de Console dat pas zodra de node een volgende keer actief wordt.

Rust en Toit – de toekomst van embedded?

Twee dingen vallen op bij zowel Rust als Toit. Beide zijn eenvoudig op te starten, en beide gebruiken package managers om de low-level drivers af te handelen. Hierdoor kunnen ontwikkelaars zich concentreren op hun werk – het bouwen van applicaties. Nu die steeds complexer worden, is dit hergebruik van code essentieel om de ontwikkeltijd te verkorten en producten sneller op de markt te brengen.

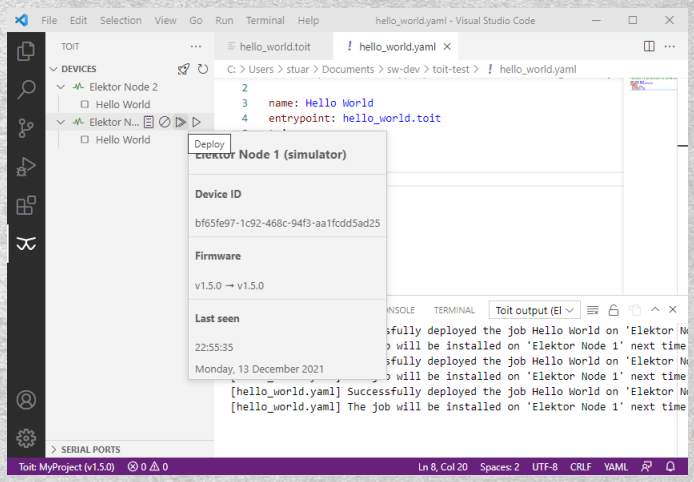
Rust is er echter nog lang niet. Met C en C++ verankerd in de wereld van embedded ontwikkeling, is het moeilijk om een zwakke plek in het geheel te vinden waar het een dermate groot voordeel biedt dat ontwikkelaars overstappen. En met Ada als gevestigde taal voor veiligheids-

Listing 4. Een eenvoudige Toit-app die geformatteerde tijd- en datum-strings uitvoert naar de log in Console.

```
main:
  time := Time.now.local
  print "Time: $(%02d time.h):$(%02d time.m):$(%02d time.s)"
  print "Date: $(%04d time.year)-$(%02d time.month)-$(%02d time.day)"
```

Listing 5. Het bijbehorende YAML-bestand dat de Toit Console vertelt hoe de app moet worden geïnstalleerd.

```
name: Hello World
entrypoint: hello_world.toit
triggers:
  on_boot: true
  on_install: true
  on_interval: "3s"
```



Figuur 6. Met behulp van de Toit-extensie voor Visual Studio Code kunnen wijzigingen in apps onmiddellijk naar IoT-nodes worden overgedragen zonder bang te hoeven zijn dat apparaten in het veld vastlopen.

kritische systemen, vallen de voordelen van Rust eigenlijk in het niet. Toit lost daarentegen een groot probleem op: het up-to-date houden van IoT-nodes in het veld en het uitrollen van nieuwe functies naar de gebruikers, en dat alles zonder dat die apparaten vastlopen. Sommige ontwikkelaars zullen zich zorgen maken over de vereiste minimum flash van 4 MB en over het feit dat momenteel alleen de ESP32 wordt ondersteund. Mocht de markt echter vragen om andere MCU's, dan lijkt er geen technische reden te zijn waarom andere platformen in de toekomst niet ondersteund kunnen worden.

210652-03

Vragen of opmerkingen?

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

Een bijdrage van

Tekst en illustraties: **Stuart Cording**
Redactie: **Jens Nickel, CJ Abate**
Vertaling: **Hans Adams**
Layout: **Harmen Heida**

GERELATEERDE PRODUCTEN

- Boek: D. Ibrahim, BBC micro:bit (Elektor 2016, SKU 17972) www.elektor.nl/17972
- Joy-IT BBC micro:bit Go Set (SKU 18930) www.elektor.nl/18930

WEBLINKS

[1] Dennis Ritchie, Computer History Museum: https://bit.ly/3oOXG1A
[2] P. Jansen, "TIOBE Index for december 2021," TIOBE Software BV, December 2021: https://bit.ly/3EXx8Ri
[3] S. Cass, "Top Programming Languages 2021," IEEE Spectrum, 2021: https://bit.ly/3oPJq8z
[4] MISRA-website: https://bit.ly/3s1Mv7D
[5] "C++ References," Tutorials Point: https://bit.ly/31Y6k53
[6] "Guidelines for the use of the C++14 language in critical and safety-related systems," AUTOSAR, oktober 2018: https://bit.ly/3dO3zWl
[7] Rust Foundation-website: https://bit.ly/3DNgO45
[8] rustup-website: https://bit.ly/3EUTiDR
[9] Rust Crate Registry-website: https://bit.ly/3dLKmor
[10] droogmic, "MicroRust," april 2020: https://bit.ly/3EUWFdM
[11] droogmic, "MicroRust: Buttons," april 2020: https://bit.ly/3DWes30
[12] bors en NitinSaxenait, "Discovery," december 2021: http://bit.ly/3GERDT7
[13] Get Ada Now-website: https://bit.ly/3GHyikw
[14] F. Chouteau, "First beta release of Alire, the package manager for Ada/SPARK," AdaCore, oktober 2020: https://bit.ly/3EUXOSC
[15] F. Chouteau, "Microbit_examples," december 2021: https://bit.ly/3mnH7bx
[16] "Watch us turn an ESP32 into a full computer!," Toit, maart 2021: https://bit.ly/3IM0WT8
[17] Toit Package Registry-website: https://bit.ly/3yokpo7
[18] Toit Docs: Prerequisites-website: https://bit.ly/3oNSECq