Was gibt‘s Neues in der Embedded-Entwicklung?

Rust und die Aktualisierung von IoT-Implementierungen

Embedded Development

Von **Stuart Cording** (Elektor)

Obwohl die Technologie für eingebettete Systeme scheinbar schnell voranschreitet, ist die Branche selbst im Vergleich dazu eher langsam. Deshalb war es fast ein kleiner Schock, als ein bekannter Hersteller von Einplatinencomputern, die Raspberry-Pi-Foundation, den RP2040-Mikrocontroller mit zwei Cortex-M0+-Kernen und ohne Onboard-Flash herausbrachte. Ein Dual-Core-Prozessor in dieser Geräteklasse ist ein absolutes Novum, aber der RP2400 ist auch schon das Spannendste auf dem Markt, dessen Fortschritt ansonsten messbar, sinnvoll und überlegt ist. Es deuten sich aber Fortschritte an, die die Entwicklung eingebetteter Systeme im kommenden Jahrzehnt verändern könnten.

Die Entwicklung eingebetteter Software ohne C ist kaum vorstellbar. Als Assembler für die Entwicklung vollständiger Anwendungen zu umständlich wurde, verdrängte C diese Sprache (außer wenn ein hochoptimierter, von Hand geschriebener Assembler die einzige Option war). Die von Dennis Ritchie [1] in den Bell Labs entwickelte Sprache ist ausreichend flexibel für komplexe Anwendungen und bietet gleichzeitig einen einfachen Zugriff auf Register. Dies ist entscheidend für einen kompakten Mikrocontroller-Code, der Registerzugriffe in Interrupt-Routinen verarbeitet. Auch Aufgaben wie die Bitmanipulation von Registern lassen sich damit leicht umsetzen. Und im Gegensatz zu Assembler-Code ist C-Code leichter zu lesen. Außerdem rangiert C in Umfragen und Marktanalysen stets unter den drei am häufigsten verwendeten Programmiersprachen (**Bild 1**) [2][3].

C ist alt

Allerdings ist C, das im Jahr 1972 entwickelt wurde, inzwischen 50 Jahre alt. Die Programmiersprache hat eine Reihe von bekannten Einschränkungen, von denen viele mit der Verwendung von Pointern zusammenhängen. Pointer erleichtern Entwicklern von eingebetteten Systemen zwar den Zugriff auf Register, sie können aber auch zu unerwünschten Speicherzugriffen außerhalb des zulässigen Bereichs führen. Außerdem führen C-Compiler im Vergleich zu moderneren Programmiersprachen vergleichsweise wenige Codeprüfungen durch. So werden ungenutzte Variablen einfach ignoriert, was ein Zeichen für einen Fehler im Code sein kann.

Um unsicheren C-Code in eingebetteten Systeme zu vermeiden, verwenden Entwickler Codierungsstandards wie MISRA C [4]. Dieser Standard entstand, als C in der Automobilindustrie als Programmiersprache für eingebettete Systeme an Bedeutung gewann. C++ löste

einige der Probleme mit Pointern durch *references* [5], die nicht geändert werden können, um auf ein anderes Objekt zu verweisen, nicht NULL sein können und bei der Erstellung initialisiert werden müssen. Trotzdem hat AUTOSAR, eine Partnerschaft von Entwicklern von Automobilsystemen, in einem mehrere hundert Seiten umfassenden Dokument Richtlinien für die Verwendung von C++ für sicherheitsrelevante Anwendungen herausgegeben [6]. Obwohl also die Beherrschung dieser etablierten Sprachen für Entwickler von eingebetteten Systemen unabdingbar ist, weist jede dieser Sprachen so viel Schwächen auf, dass Richtlinien erforderlich sind, um häufige Programmierfehler zu vermeiden.

Einführung von Rust

Rust hat sich als potenzieller Herausforderer herauskristallisiert und wirbt mit der Eignung zur Entwicklung sicherer Systeme. Rust begann 2006 als privates Projekt von Graydon Hoare und wurde 2010 zu einem von seinem Arbeitgeber Mozilla Research geförderten Projekt. Als 2021 Umstrukturierungen des Unternehmens Auswirkungen auf das Rust-Entwicklungsteam hatten, wurde die Rust Foundation [7] gegründet.
Das Besondere an Rust ist, dass viele Probleme schon bei der Kompilierung erkannt und markiert werden, oft auch solche, die C/C++-Compiler ignorieren würden. Es wurde auch ein *Ownership*-System für Variablendeklarationen implementiert, das einen *Borrow-Checker* verwendet, um eine missbräuchliche Anwendung von Variablen bereits während der Kompilierung auszuschließen. Außerdem muss der Lese- und Schreibzugriff auf Variablen, die per Referenz übergeben werden, explizit deklariert werden. Die Syntax von Rust ähnelt weitgehend C und C++, wobei geschweifte Klammern um Funktionen und die bekannten Steuer-Keywords verwendet werden.
Da der Schwerpunkt auf sicherem Code liegt, wurde Rust besonders für die Bare-Metal-Community zur Entwicklung eingebetteter Software interessant. Aufgrund der Natur der eingebetteten Programmierung kann die vom Compiler eingesetzte statische Prüfung jedoch bei der Implementierung einiger Codetypen zu Problemen führen. Einige Codeabschnitte können als *unsafe* markiert werden, um zum Beispiel die De-Referenzierung von Pointern zu ermöglichen. Durch die explizite Definition von Codeabschnitten als *unsafe* kann die Umgehung der Rust-Regeln verdeutlicht werden.

Rust auf den Prüfstand gestellt

Um ein Gefühl für Rust zu bekommen, installiert man Rust am besten auf einem Raspberry Pi. Die Installation ist einfach, wenn man den Hinweisen auf der Website *rustup.rs* [8] folgt. Geben Sie auf der Kommandozeile einfach den folgenden Befehl ein und folgen Sie den Anweisungen:

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

Im Gegensatz zu C/C++, das Binärdateien erzeugt, wird die Ausgabe von Rust als *Crate* bezeichnet. Der Paketmanager Cargo sorgt für eine einfache Befehlszeilenkompilierung. Er enthält die Daten, die für die Erzeugung des *Crate* benötigt werden und ermöglicht es dem Entwickler, die für die Erstellung des *Crate* erforderlichen Pakete zu definieren. Durch den Aufruf von Cargo auf der Kommandozeile wird ein neues Rust-Projekt wie folgt erzeugt:

```
cargo new rust_test_project
```

Wenn Sie das Verzeichnis des neuen Projekts öffnen, sehen Sie eine Datei namens *Cargo.toml*. Diese Datei muss unter Umständen geändert werden. Für eine einfache Raspberry-Pi-Anwendung, die eine an einen

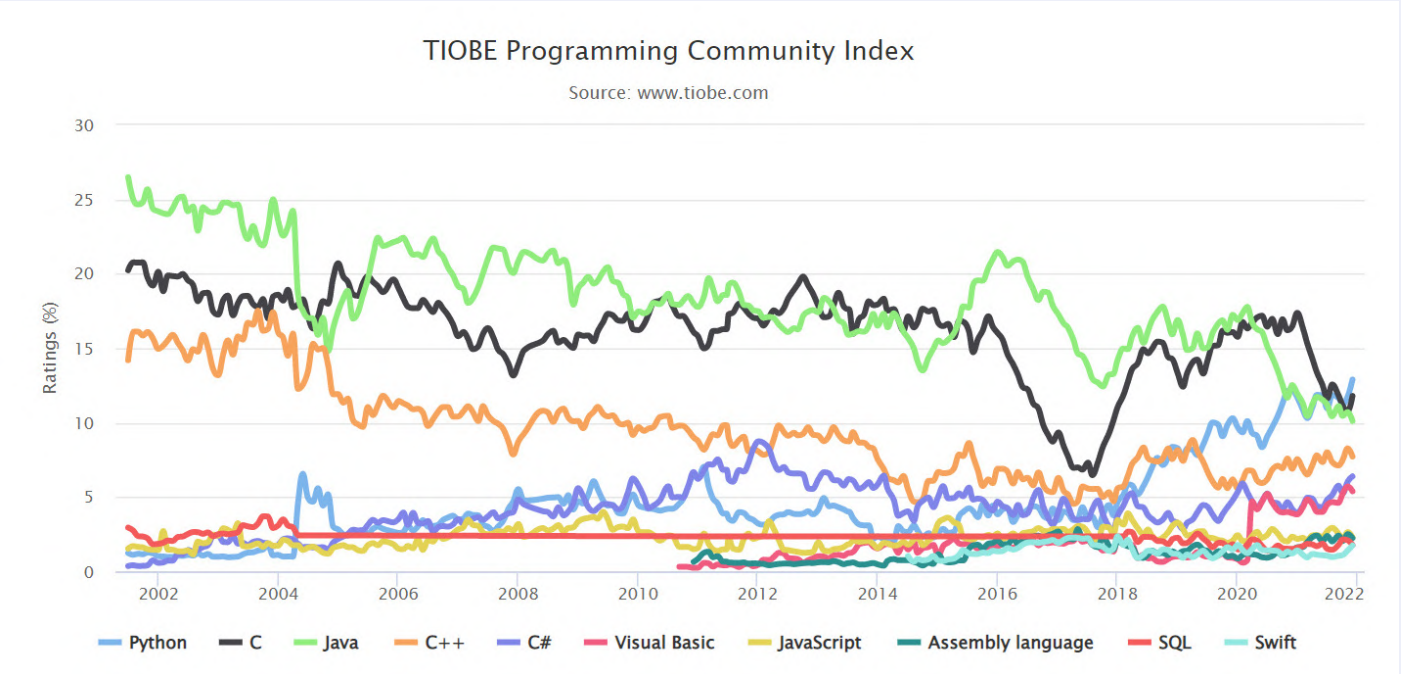

Bild 1. C ist als Programmiersprache nach wie vor sehr beliebt und belegt in Umfragen und Marktanalysen regelmäßig einen der ersten drei Plätze. (Quelle: www.tiobe.com)

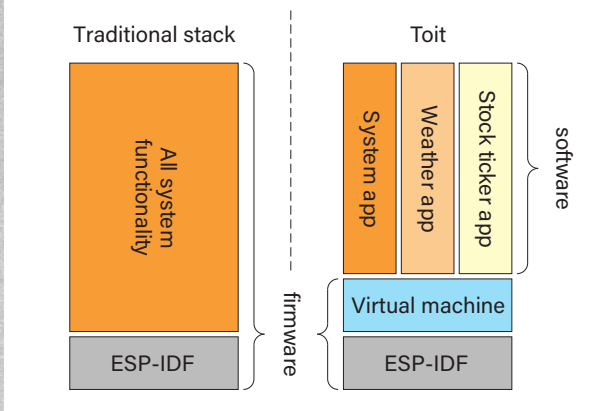

Bild 2. Toit führt IoT-Code als Apps auf einer virtuellen Maschine aus, die auf einem ESP32 läuft. (Quelle: Toit)

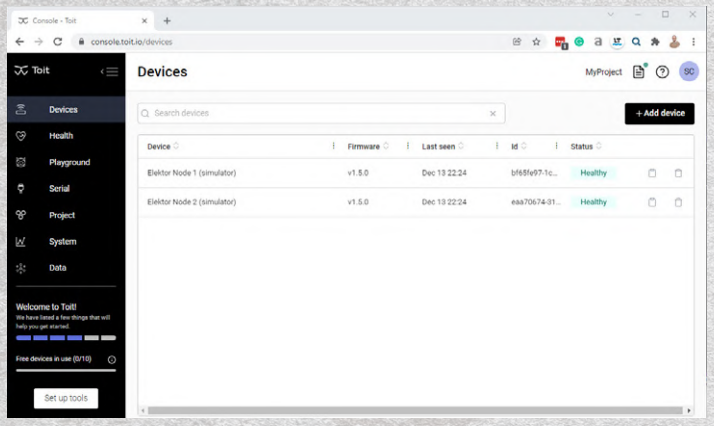

Bild 3. Die Toit-Konsole in einem Browser, hier mit zwei simulierten ESP32-Knoten.

GPIO-Pin angeschlossene LED blinken lässt, muss eine geeignete Crate-Abhängigkeit für den Zugriff auf die GPIO-Pins definiert werden. Crates werden auf der Crate-Plattform der Rust-Community crates.io [9] zur Verfügung gestellt. Ein geeigneter Crate ist *rppal*, der über die Suchfunktion gefunden werden kann. Die Plattform gibt die Versionsnummer an und stellt eine Dokumentation und Beispielcode zur Verfügung. Der Name des Crate und die Versionsnummer werden dann als *dependencies* in *Cargo.toml* hinzugefügt (**Listing 1**).

Listing 1. Hinzufügen der rppal-Abhängigkeit zu Cargo.toml in einem Rust-Projekt.

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

Im *src*-Verzeichnis findet der Entwickler ein einfaches *Hello-World*-Beispielprojekt in der Rust-Quellcodedatei *main.rs*. Wenn man diesen Code durch **Listing 2** ersetzt, kann man eine LED an GPIO 23 (Pin 16 des Raspberry-Pi-Headers) zehnmal blinken lassen. Die Kompilierung erfolgt über die Kommandozeile mit `cargo build`, das Crate wird mit `cargo run` ausgeführt.

Um Rust auf Mikrocontrollern verwenden zu können, ist eine LLVM-Toolchain erforderlich. Wenn diese vorhanden ist, können Sie mit Hilfe eines der verschiedenen Online-Tutorials loslegen. Ein Beispiel für den BBC micro:bit [10] zeigt, dass das Schreiben von Code, hat man erst einmal die Rust-Syntax im Griff, dem von C/C++ sehr ähnlich ist (**Listing 3** [11]). Ein weiteres Beispiel für den STM32 [12] ist ebenfalls enthalten.

Gehört Rust die Zukunft?

Was steht also der Einführung von Rust für Embedded-Anwendungen im Wege? Wenn Sie nach einer robusten Programmiersprache für den

Listing 2. Blinken einer LED in Rust.

Das Blinken einer LED in Rust ähnelt dem in C geschriebenen Code.
Dieses Beispiel basiert auf dem Code, der im rppal-Crate enthalten ist.

```
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. Vereinfachter Codeschnipsel, der den Zustand eines Eingangspins des BBC micro:bit mit Rust überprüft.

```
#![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");
 }
```

Einsatz in sicherheitskritischen Systemen suchen, ist doch Ada [13] eigentlich eine gute Wahl. Doch obwohl Ada bereits 40 Jahre alt ist, hat sie C nicht verdrängen können, obwohl sie speziell für den Einsatz in eingebetteten Echtzeitsystemen entwickelt wurde. Ein weiterer Aspekt ist der jahrelange Erfolg von C, mit unzähligen Entwicklern, Werkzeugen und Code-Bibliotheken.

Der Crate-Paketmanager könnte jedoch dazu beitragen, die Verbreitung von Rust zu beschleunigen. Den Überblick über die in C/C++ geschriebenen Peripherie-Bibliotheken der Halbleiterhersteller zu behalten, kann schwierig und undurchsichtig sein, so dass die explizite Definition der Version der in *Cargo.toml* verwendeten Crates als bedeutende Verbesserung angesehen werden könnte. Es könnte auch das Leben der MCU-Hersteller erleichtern, die ihre riesigen Produktportfolios unterstützen wollen. Ada hat im Jahr 2020 bereits mit der Veröffentlichung des Paketmanagers *Alire* reagiert [14] und unterstützt genau wie Rust den BBC micro:bit. Damit können Sie beide Sprachen auf der gleichen MCU vergleichen [15].

IoT-Geräte auf dem neuesten Stand halten

Da immer mehr eingebettete Geräte mit Netzwerken verbunden sind, besteht die größte Herausforderung darin, sie mit der neuesten Version der Firmware auf dem neuesten Stand zu halten. Traditionell müssen Firmware-Updates über Funk heruntergeladen werden, wobei die Binärdaten über einen On-Chip-Bootloader im geschützten Bereich des Geräts in den Flash-Speicher programmiert werden. Dabei besteht das Risiko, dass die neue Codeversion nicht korrekt installiert wird, so dass das Produkt nicht mehr funktioniert und unbrauchbar wird. Alternativ kann der Code zwar funktionieren, aber ein Softwarefehler im neuen Code könnte verhindern, dass künftige Updates eingespielt werden, so dass die Geräte in der Praxis anfällig bleiben. Auf diese Weise wird das Gerät vielleicht nur aufgrund eines Fehlers in einer einzigen Codezeile unbrauchbar, obwohl der Großteil der Anwendung korrekt funktioniert.

Virtuelle Maschinen sind seit Jahren eine Standardmethode, um Anwendungen oder Betriebssysteme auf Servern bereitzustellen. Die virtuelle Maschine gaukelt einem Betriebssystem vor, es würde auf dedizierter Hardware ausgeführt. Sollte das Betriebssystem schwerwiegend versagen, ist nur die betroffene virtuelle Maschine davon betroffen, nicht aber die übrigen auf dem Server laufenden Systeme. Desktop-Benutzer kennen Virtualisierungssoftware wie VirtualBox, VMware und Parallels, mit der sie Software oder alternative Betriebssysteme testen können, ohne ihren primären Rechner einem Risiko auszusetzen.

Aktualisierung der IoT-Knoten-Firmware: Der Weg zu Toit

Das Team von Toit, einem dänischen Unternehmen, das von ehemaligen Google-Ingenieuren gegründet wurde, fragte sich, warum die Virtualisierung nicht auch auf Mikrocontrollern eingesetzt wird, auf denen IoT-Anwendungen laufen. Schließlich müssen diese regelmäßig aktualisiert werden, um Fehler zu beheben und möglicherweise um Verbesserungen in der Cloud zu unterstützen.

Für die ersten Schritte haben sie sich für den ESP32 von Espressif entschieden, genauer gesagt, den ESP32-WROOM-32E mit einem 32-bit-Dual-Core Xtensa LX6 Mikroprozessor, 520 KB SRAM und 4 MB Flash. Für diese Plattform gibt es bereits das ESP-IDF (Espressif IoT Development Framework), so dass sie für den Einsatz als IoT-Knoten gut vorbereitet ist. Toit hat dann eine virtuelle Maschine (**Bild 2**) entwickelt, auf der Anwendungen sicher ausgeführt werden können. Die Anwendungen sind in Toit geschrieben, einer objektorientierten und sicheren Hochsprache.

Testen von Toit mit simulierten ESP32-Knoten

Die Verwaltung und Bereitstellung von Anwendungen auf einer Ansammlung von IoT-Knoten erfolgt über die Toit-Konsole in Verbindung mit Visual Studio Code von Microsoft. Für diejenigen, die nur daran interessiert sind, die Plattform zu testen, ermöglicht die Toit Console die Verwendung von simulierten ESP32-Geräten. Nach der Installation der Toit-Erweiterung für Visual Studio Code können Anwendungen geschrieben und lokal simuliert oder an Geräte verteilt werden, die mit der Konsole verbunden sind.

Toit-Apps werden von einer YAML-Datei begleitet. In dieser Markup-Language-Datei werden die Namen der Toit-App und der Quellcodedatei deklariert und Trigger definiert, die die App nach dem

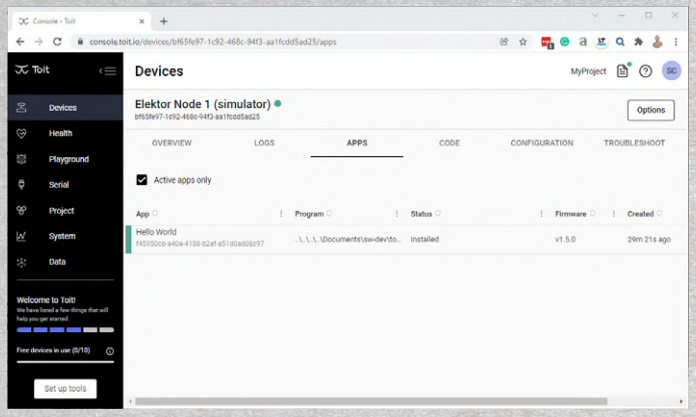

Bild 4. Die Hello-World-App wurde erfolgreich auf einem simulierten Knoten installiert

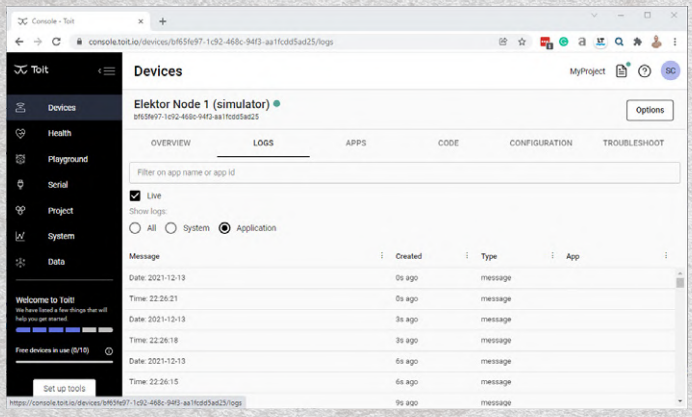

Bild 5. Die LOGS-Ausgabe zeigt die Uhrzeit und das Datum, die von der Toit-App alle drei Sekunden ausgegeben werden, wie in der YAML-Datei definiert.

Booten oder nach bestimmten Zeitintervallen ausführen.
Die Bereitstellung von Apps oder Updates erfordert nicht, dass das ESP32-Zielgerät deaktiviert oder die Stromversorgung unterbrochen werden muss. Stattdessen aktualisiert die virtuelle Maschine die App an Ort und Stelle und triggert sie gemäß den in der YAML-Datei angegebenen Einstellungen. Wenn die App in irgendeiner Weise versagt, zum Beispiel durch einen Speicherüberlauf, werden die übrigen Apps ohne Probleme weiterhin ausgeführt [16]. Solche Fehler können über die Konsole diagnostiziert werden, indem man sich das Protokoll ansieht.
Listing 4 zeigt eine einfache *Hello-World*-App, die Uhrzeit und Datum ausgibt, **Listing 5** die zugehörige YAML-Datei. In **Bild 3** sind zwei simulierte Knoten in der Konsole zu sehen, während **Bild 4** die erfolgreich installierte *Hello-World*-App zeigt. **Bild 5** zeigt die Ausgabe von Datum und Uhrzeit in Drei-Sekunden-Intervallen, wie sie mit dem Trigger `on_interval: "3s"` in der YAML-Datei festgelegt wurden.
Der Projektcode wurde in Visual Studio Code erstellt und mit der Toit-Erweiterung auf dem ausgewählten Knoten bereitgestellt, der mit dem Konsolen-Konto des Users verbunden ist (**Bild 6**).
Auf den ersten Blick mag der Toit-Ansatz ein wenig restriktiv erscheinen. Die Virtualisierungsumgebung erlaubt nur die Steuerung der GPIOs, der seriellen Schnittstelle (UART), SPI oder I²C. Da jedoch viele IoT-Knoten Sensordaten sammeln und diese an die Cloud melden, dürfte dies für die meisten Anwendungen ausreichend Flexibilität bieten. Toit betreibt auch einen eigenen Paketmanager (Registry) [17], der Treiber für eine Reihe von Sensoren, Eingabegeräten, LCDs und anderen Dienstprogrammen zur Verfügung stellt. Die Umgebung unterstützt auch den Stromsparmodus des ESP32, der es angeblich ermöglicht, das Gerät jahrelang mit zwei AA-Batterien zu betreiben [18]. Wenn eine App-Aktualisierung erfolgt, während sich der IoT-Knoten im Tiefschlafmodus befindet, stellt die Konsole sie dann bereit, wenn der Knoten das nächste Mal aufwacht.

Rust und Toit - die Zukunft von Embedded?

Zwei Dinge stechen bei Rust und Toit hervor. Beide sind einfach in Betrieb zu nehmen und beide verwenden Paketmanager, um die Low-Level-Treiber zu verwalten. Dadurch können sich die Entwickler auf ihre eigentliche Aufgabe konzentrieren, nämlich das Erstellen von Anwendungen. Da die Anwendungen immer komplexer werden, ist eine solche Wiederverwendung von Code unerlässlich, um die Entwicklungszeit zu verkürzen und Produkte schneller auf den Markt zu bringen.
Rust steht vor einer gewaltigen Aufgabe: Da C und C++ in der Welt der Embedded-Entwicklung so fest verankert sind, dürfte es schwierig sein, eine Lücke zu finden, in der Rust einen so gewichtigen Vorteil bietet, dass er Entwickler anlockt. Und da sich Ada bereits als die

Listing 4. Eine einfache Toit-Anwendung, die formatierte Zeit- und Datumsstrings in der Log-Konsole ausgibt.

```
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. Die zugehörige YAML-Datei, die der Toit-Konsole mitteilt, wie die Anwendung bereitgestellt werden soll.

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

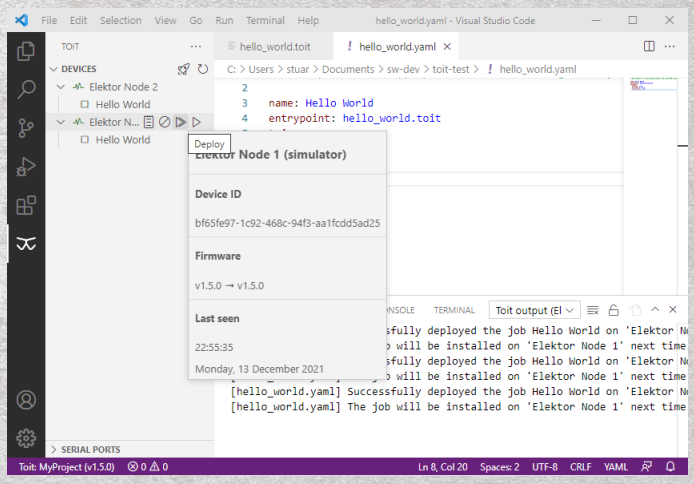

Bild 6. Mit der Toit-Erweiterung in Visual Studio Code können Änderungen an Anwendungen auf IoT-Knoten bereitgestellt werden, ohne dass die Geräte vor Ort aus dem Netz entnommen werden müssen.

Sprache für sicherheitskritische Systeme etabliert hat, verschwinden die Vorteile von Rust.

Toit hingegen löst ein großes Problem: Es hält entfernte IoT-Knoten auf dem neuesten Stand, stellt neue Funktionen zur Verfügung und das alles, ohne die Geräte aus dem System zu entfernen. Einige Entwickler dürften über nur 4 MB Flash und die Tatsache, dass derzeit nur der ESP32 unterstützt wird, die Stirn runzeln, sollte jedoch eine Nachfrage nach anderen MCUs entstehen, dürfte es keinen technischen Grund zu geben, warum andere Plattformen in Zukunft nicht unterstützt werden sollten.

210652-02

Haben Sie Fragen oder Kommentare?

Haben Sie technische Fragen oder Kommentare zu diesem Artikel? Dann wenden Sie sich bitte an den Autor unter stuart.cording@elektor.com oder an die Elektor-Redaktion unter redaktion@elektor.de.

Ein Beitrag von

Text und Illustrationen: **Stuart Cording**
Redaktion: **Jens Nickel, CJ Abate**
Übersetzung: **Rolf Gerstendorf**
Layout: **Harmen Heida**

WEBLINKS

[1] Dennis Ritchie, Computer History Museum: https://bit.ly/3oOXG1A
[2] P. Jansen, „TIOBE Index for Dezember 2021“, TIOBE Software BV, Dezember 2021: https://bit.ly/3EXx8Ri
[3] S. Cass, „Top Programming Languages 2021“, IEEE Spectrum, 2021: https://bit.ly/3oPJq8z
[4] Webseite MISRA: 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] Webseite Rust Foundation: https://bit.ly/3DNgO45
[8] Webseite rustup: https://bit.ly/3EUTiDR
[9] Webseite Rust Crate Registry: 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 and NitinSaxenait, „Discovery“, Dezember 2021: http://bit.ly/3GERDT7
[13] Webseite Get Ada Now: 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“, Dezember 2021: https://bit.ly/3mnH7bx
[16] „Watch us turn an ESP32 into a full computer!“, Toit, März 2021: https://bit.ly/3IM0WT8
[17] Webseite Toit Package Registry: https://bit.ly/3yokpo7
[18] Webseite Toit-Docs, Voraussetzungen: https://bit.ly/3oNSECq