quoi de neuf dans le développement de l'embarqué ?

Rust et mises à jour des déploiements IoT

Embedded Development

Stuart Cording (Elektor)

Alors que les annonces de commercialisation de systèmes embarqués se succèdent, donnant l'impression que la technologie avance rapidement, le secteur lui-même paraît lent. C'est pourquoi ce fut un choc quand la fondation Raspberry Pi, célèbre créateur d'ordinateurs monocartes, sortit le microcontrôleur (MCU) RP2040 doté de deux cœurs Cortex-M0+, sans flash intégrée : un double cœur dans cette catégorie, c'était du jamais vu. Mais l'excitation est vite retombée. Tout compte fait, les progrès des systèmes embarqués sont mesurés, sensés et réfléchis. Mais, nous le verrons, les évolutions en cours pourraient changer le développement de l'embarqué dans la décennie à venir.

Le développement de logiciels embarqués sans C est difficile à imaginer. Le langage C a supplanté l'assembleur devenu trop lourd pour développer des applications entières, sauf si la seule option est l'assembleur hautement optimisé et codé à la main. Le langage, développé par Dennis Ritchie [1] des Bell Labs, offre assez de souplesse pour développer des applications complexes, tout en permettant un accès aisé aux registres. C'est essentiel pour écrire un code de MCU compact qui gère cet accès pour les routines d'interruption. L'écriture de tâches telles que la manipulation des bits des registres est aussi aisée. Et, contrairement à l'assembleur, le code résultant est facile à lire. Le C occupe aussi une place de choix en se classant parmi les trois premiers langages de programmation dans les enquêtes et les analyses de marché (**fig. 1**) [2] [3].

Si c'est du C, c'est daté !

Le langage C remonte à 1972, soit 50 ans. Ses limitations bien connues sont, pour beaucoup, liées à l'utilisation de pointeurs. Si ces pointeurs facilitent l'accès aux registres aux développeurs de systèmes embarqués, ils peuvent aussi causer de dangereux accès mémoire hors limites. En outre, loin des langages plus modernes, les compilateurs C effectuent peu de vérifications de code. Ils ignorent les variables inutilisées, elles sont pourtant un indice d'erreur de codage.

Une norme de codage telle que MISRA C [4] permet de s'assurer que seul un code C sécurisé est intégré dans un système embarqué. Cette norme est née de l'importance croissante du C pour programmer les systèmes embarqués dans l'industrie automobile. Le C++ résout certains problèmes des pointeurs du C par des *références* [5] non modifiables pour référencer un autre objet, elles ne peuvent pas être *NULL* et doivent être initialisées à la création. Malgré cela, AUTOSAR, un partenariat de développeurs de systèmes automobiles, a élaboré un guide de directives [6] de plusieurs centaines de pages sur l'utilisation du C++ dans les applications liées à la sécurité. Ainsi, bien que la maîtrise de ces langages établis soit essentielle pour les développeurs, il apparaît que chaque langage a assez de défauts pour qu'il faille écrire des directives pour éviter les erreurs courantes.

Présentation de Rust

Se proclamant *très adapté* au développement de systèmes sûrs, Rust apparaît comme un concurrent potentiel. Au début (2006), Rust est un projet privé de Graydon Hoare, puis vers 2010, son employeur, Mozilla Research, le sponsorise. La Rust Foundation [7] est créée en 2021, après une restructuration de l'entreprise qui toucha l'équipe de développement de Rust.

Rust est différent car, au moment de la compilation, de nombreux problèmes sont détectés et signalés, là où souvent les compilateurs C/C++ les ignoreraient. Pour les déclarations de variables, un système de *propriété* existe. Pour éviter l'utilisation abusive de variables, il utilise un *vérificateur d'emprunt* qui s'applique à la compilation. Il faut aussi explicitement déclarer l'accès en lecture/écriture des variables passées par référence. La syntaxe de Rust ressemble beaucoup à celle du C/C++, avec l'usage familier de crochets autour des fonctions et mots-clés de contrôle.

La priorité à la sécurité du code justifie les efforts déployés pour fournir Rust aux développeurs de logiciels embarqués sur matériel dédié. Cependant, la nature de la programmation embarquée fait que la vérification statique du compilateur peut créer des problèmes avec certains types de code. Pour contourner ces problèmes, le code peut, par endroits être marqué comme *non-sûr* et, par ex., permettre de « déréférencer » des pointeurs. Définir explicitement des sections de code comme non-sûres clarifie le contournement des règles de Rust.

Faites un tour avec Rust

L'une des meilleures façons d'aborder Rust est d'expérimenter avec un Raspberry Pi. En suivant les instructions du site rustup.rs [8], l'installation est simple. Depuis la ligne de commande, il suffit de saisir la chaîne ci-après et de suivre les instructions :

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

Contrairement au C/C++ qui sort des binaires, RUST produit un *crate* (littéralement une caisse, la déclinaison Rust du paquet). Le gestionnaire de paquets Cargo simplifie le processus de compilation en ligne de commande. Il stocke les données nécessaires pour produire le crate et permet au développeur de définir les logiciels requis pour le construire. En invoquant Cargo depuis la ligne de commande, un nouveau projet Rust est produit comme suit :

```
cargo new rust_test_project
```

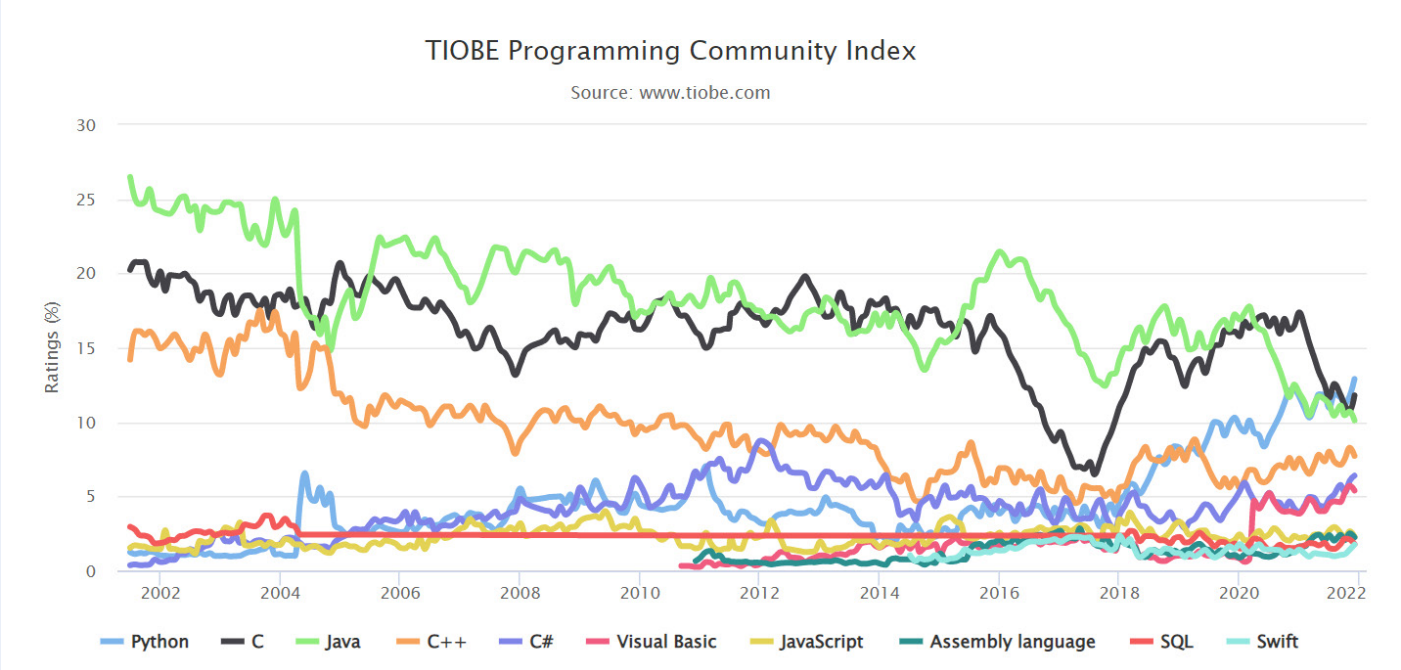

Figure 1. Le langage C reste un langage de programmation apprécié, se plaçant régulièrement dans le trio de tête des enquêtes et analyses de marché. (Source : www.tiobe.com)

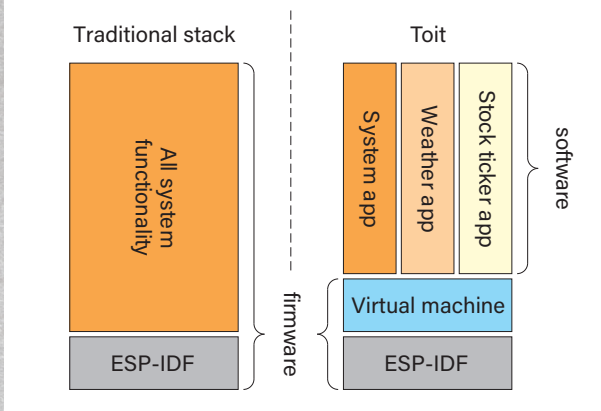

Figure 2. Toit exécute le code IoT sous forme d'applis au-dessus d'une machine virtuelle tournant sur un ESP32. (Source : Toit)

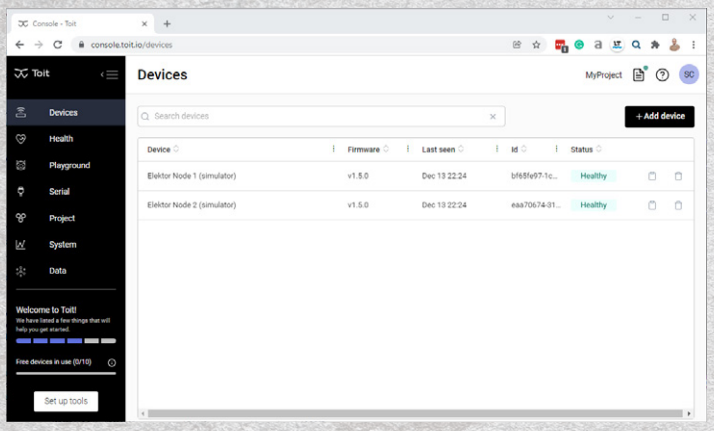

Figure 3. La console Toit affiche dans un navigateur deux nœuds ESP32 simulés.

Le dossier du nouveau projet contient un fichier nommé `Cargo.toml`. Il faut modifier ce fichier selon les besoins. Dans une simple application Raspberry Pi faisant clignoter une LED reliée à une broche GPIO, il faut définir une « dépendance de crate » idoine pour accéder aux lignes GPIO. Des crates sont partagés sur crates.io [9], la page créée à cet effet par la communauté Rust. La fonction de recherche permet de trouver un crate approprié : *rppal*. La plateforme indique si le crate peut être construit ou non, fournit le n° de version, une documentation et des exemples de code. Le nom et le n° de version du crate sont ensuite ajoutés comme *dépendances* dans `Cargo.toml` (**list. 1**).

Dans le dossier `src`, le développeur trouvera le fichier de code source Rust `main.rs`, un exemple de projet simple, *Hello World*. Remplacer ce code par le **list. 2** permet de faire clignoter dix fois une LED connectée à la borche GPIO 23 (br. 16 du connecteur Raspberry Pi). La compilation s'effectue en ligne de commande à l'aide de `cargo build`, et le crate s'exécute avec `cargo run`.

Un obstacle à l'utilisation de Rust sur certains µcontrôleurs est la nécessité d'utiliser une chaîne d'outils LLVM (*Low Level Virtual Machine*). Si celle-ci est disponible, l'utilisation d'un tutoriel en ligne peut commencer. Un exemple pour la carte BBC micro:bit [10] est fourni. Dès la syntaxe de Rust assimilée, il se montre très similaire à l'écriture de code en C/C++ (**list. 3** [11]). Un autre exemple pour STM32 [12] est fourni.

Listage 1. Ajout de la dépendance rppal à Cargo.toml dans un projet Rust.

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

Listage 2. Faire clignoter une LED en Rust.

Le code Rust pour faire clignoter une LED est similaire au code C. Cet exemple se base sur le code inclus dans le crate *rppal*.

```
use std::error::Error;
 use std::thread;
 use std::time::Duration;
use rppal::gpio::Gpio;
 use rppal::system::DeviceInfo;
// La GPIO utilise les n° de br. du BCM. BCM GPIO 23 est lié à la br. physique 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 {         pin.set_
        // Faire clignoter la LED en mettant la br. à 1 durant 500 ms.
high();         thread::sleep(Duration::
from_millis(500));
pin.set_low();         thread::sleep(Duration::
from_millis(500));         n += 1;
     }
     Ok(())
 }
```

Listage 3. Extrait de code Rust simplifié qui vérifie l'état d'une broche d'entrée sur la carte BBC micro:bit.

```
#![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() {
         gpio         let mut
p.GPIO.split();// Split GPIO
 =

        // Configure le bouton GPIO en entrée
         let button_a = gpio.pin17.into_floating_input();

        // variable de boucle
         let mut state_a_low = false;

        loop {
             let // Obtenir l'état du bouton
button_a_low = button_a.is_low();

            if button_a_low && !state_a_low {
             }// Message de sortie

 {            if !button_a_low && state_a_low
                 // Message de sortie
             }

            // Mémoriser l'état du bouton
;             state_a_low = button_a_low
  }
     }
     panic!("End");
 }
```

Rust, est-ce l'avenir ?

Si oui, qu'est-ce qui freine l'adoption de Rust pour l'embarqué ? D'une part, il existe déjà Ada [13], un langage de programmation robuste adapté aux systèmes de sécurité critique. Malgré ses 40 ans, il n'a pas réussi à supplanter le C, bien qu'il fût développé spécifiquement pour les systèmes embarqués en temps réel. D'autre part, le C est partout : développeurs innombrables, outils et bibliothèques de code disponibles.

Toutefois, le gestionnaire de paquets Crate pourrait contribuer à accélérer l'adoption de Rust. Garder la trace des bibliothèques périphériques des fournisseurs écrites en C/C++ peut être difficile et opaque, aussi la définition explicite de la version des crates utilisée dans `Cargo.toml` pourrait susciter une large adhésion. Cela peut aussi simplifier la vie des fabricants de MCU qui peinent à prendre en charge leurs énormes portefeuilles de produits. Cependant, Ada a déjà réagi en sortant en 2020 son gestionnaire de paquets *Alire* [14]. Et, à l'instar de Rust, Ada inclut déjà le support de la carte BBC micro:bit. Pour comparer les deux langages avec un MCU, voir ce lien [15].

Maintenir les dispositifs IoT à jour

Avec un nombre toujours croissant de dispositifs embarqués connectés aux réseaux, le plus grand défi est de maintenir leur micrologiciel à jour. Ces m. à j. se font classiquement par téléchargement sans fil, en déposant le code binaire en mémoire flash à l'aide d'un chargeur d'amorçage embarqué et résidant en zone protégée. Toutefois, une mise à jour du µcode risque de ne pas se déployer correctement, et de rendre le produit inutilisable. Il peut aussi arriver qu'un bogue du nouveau code empêche les m. à j. ultérieures de se déployer, laissant les dispositifs touchés vulnérables. Ainsi, bien que l'application fonctionne correctement, l'appareil devient inutilisable, parfois à cause d'une erreur dans une seule ligne de code.

Depuis des années, les machines virtuelles (VM) déploient de façon standard plusieurs applications ou systèmes d'exploitation (OS) sur des serveurs. La VM exécute un OS, en faisant croire qu'il se trouve sur un matériel spécifique. En cas de défaillance catastrophique de cet OS, seule la VM concernée est touchée, et non les autres systèmes fonctionnant sur le serveur. Les utilisateurs de postes de travail connaissent les logiciels de virtualisation tels que VirtualBox, VMware et Parallels, leur permettant de tester des logiciels ou d'autres OS sans mettre en péril leur machine principale.

Mise à jour du µlogiciel des nœuds IoT : Toit sur place

L'équipe de Toit, entreprise danoise créée par d'anciens ingénieurs de Google, s'est demandé pourquoi la virtualisation était absente des MCU exécutant des applications IoT. En effet, elles nécessitent des m. à j. régulières pour corriger les bugs et prendre en charge les évolutions des services cloud avec lesquels elles communiquent. Il va de soi qu'aucune firme ne flashera manuellement les nœuds IoT sur place si des appareils sont bloqués après une m. à j.

Pour commencer, ils ont opté pour l'ESP32 d'Espressif. Ils recommandent l'ESP32-WROOM-32E avec microprocesseur Xtensa LX6 à 32 bits à double cœur, 520 Ko de SRAM et 4 Mo de flash. La plateforme a déjà son ESP-IDF (*Espressif IoT Development Framework*). Elle est donc prête à l'emploi comme nœud IoT. Toit a ensuite conçu une VM (**fig. 2**) qui permet aux applications de s'exécuter par-dessus en toute sécurité. Les applis sont écrites en Toit, un langage de haut niveau, orienté objet et sûr.

Test de Toit avec des nœuds ESP32 simulés

La gestion et le déploiement d'applications sur un ensemble de nœuds IoT s'effectuent via la console Toit couplée à Visual Studio Code de Microsoft. Pour qui ne souhaite que tester la plateforme, la console Toit permet d'utiliser des dispositifs ESP32 simulés. Après installation de l'extension Toit pour Visual Studio Code, les applications peuvent être écrites et simulées localement ou déployées sur les appareils connectés à la console.

Un fichier YAML accompagne les applis Toit. Ce fichier de langage de balisage sert à déclarer le nom de l'appli Toit, le fichier de code source et à définir les déclencheurs. Ces derniers peuvent lancer l'appli après le démarrage, l'installation et à intervalles de temps définis.

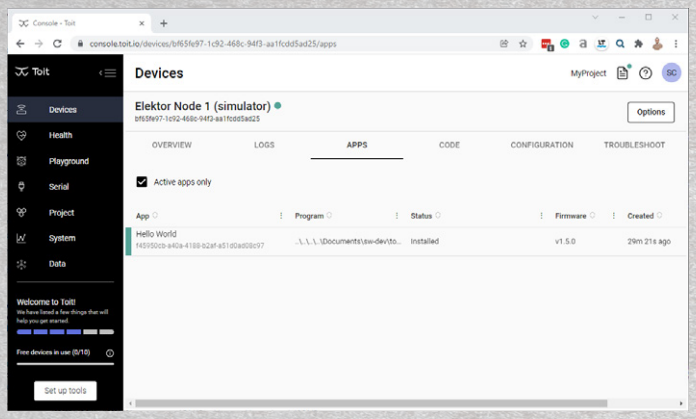

Figure 4. L'application Hello World installée avec succès sur un nœud simulé.

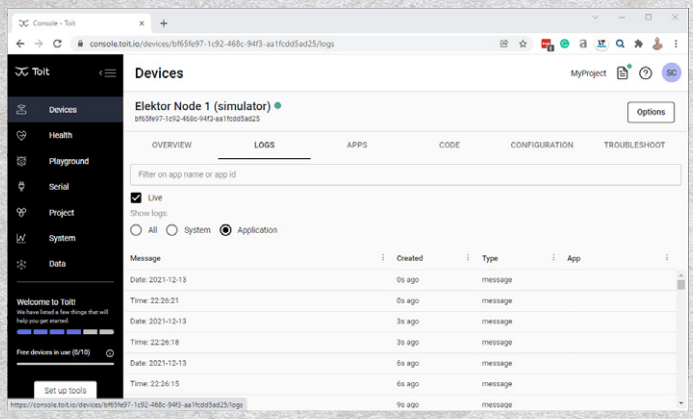

Figure 5. La sortie LOGS montre l'heure et la date émises par l'application Toit toutes les trois secondes, comme défini dans le fichier YAML.

Le déploiement d'applis ou de m. à j. ne nécessite ni la désactivation ni la mise hors tension du dispositif ESP32 cible. À la place, la machine virtuelle met à jour l'appli in situ et la lance selon les paramètres fournis par le fichier YAML. Si l'appli échoue, par ex. à cause d'un débordement de pile, les autres applis continuent de s'exécuter [16]. L'examen du journal de la Console permet de diagnostiquer de telles erreurs.

Le **listage 4** présente une appli simple *Hello World* affichant l'heure et la date, accompagnée de son fichier YAML (**list. 5**). La **figure 3** montre deux nœuds simulés dans la Console, tandis que la **figure 4** montre l'appli *Hello World* installée avec succès. Enfin, la **figure 5** montre la sortie de la date et de l'heure à intervalles de trois secondes, comme défini à l'aide du déclencheur `on_interval: "3s"` dans le fichier YAML. Visual Studio Code crée le code du projet et l'extension Toit le déploie sur le nœud choisi attaché au compte Console de l'utilisateur (**fig. 6**).

À première vue, l'approche Toit peut sembler un peu restrictive. L'environnement de virtualisation ne permet que le contrôle des GPIO, de l'UART, du SPI ou de l'I²C. Cela dit, vu que les nœuds IoT se contentent en général de collecter des données de capteurs pour les envoyer dans le cloud, cela fournit assez de flexibilité pour les applis. Toit exploite également son propre gestionnaire de paquets (registre) [17], fournissant des pilotes pour divers capteurs, périphériques d'entrée, écrans LCD et d'autres utilitaires. L'environnement prend aussi en charge le mode basse consommation de l'ESP32, et selon eux, deux piles AA font fonctionner le dispositif pendant des années [18]. Si une m. à j. d'appli arrive lorsque le nœud IoT est en mode sommeil profond, la Console la déploie au prochain réveil du nœud.

Rust et Toit – l'avenir de l'embarqué ?

Rust et Toit font tous deux ressortir deux choses : ils sont simples de mise en œuvre et gèrent les pilotes de bas niveau via des gestionnaires de logiciels. Ainsi les développeurs se concentrent sur leur travail : créer des applis. Les applis devenant de plus en plus complexes, cette réutilisation du code est vitale pour réduire le temps de développement et de commercialisation des produits.

Rust a une énorme montagne à gravir. Le C/C++ est si bien ancré dans le développement embarqué, qu'il est difficile de trouver son talon d'Achille et d'offrir un avantage assez important pour attirer les développeurs. De plus, déjà établi comme langage pour les systèmes critiques de sécurité, Ada fait écran aux avantages de Rust.

Toit, en revanche, résout un casse-tête majeur : maintenir les nœuds IoT distants à jour, déployer de nouvelles fonctions auprès de la base d'uti-

Listage 4. Application simple écrite en Toit qui envoie des chaînes date et heure formatées au journal de la 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)"
```

Listage 5. Fichier auxiliaire YAML indiquant à la Console Toit comment déployer l'appli.

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

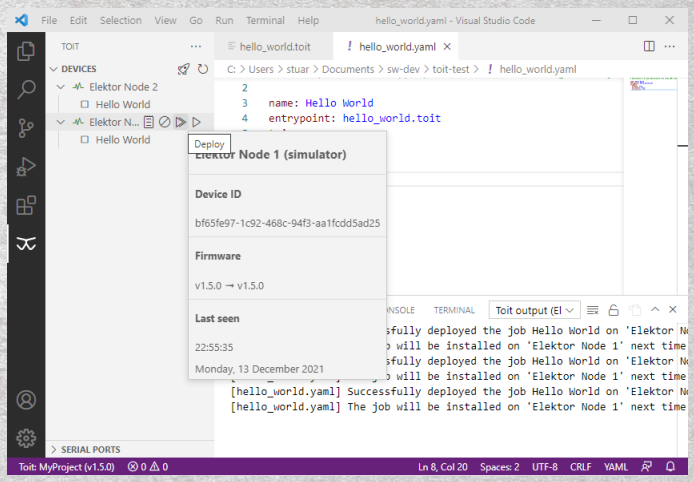

Figure 6. Grâce à l'extension Toit de Visual Studio Code, les m. à j. d'applis peuvent être déployées sans délai à distance sur les nœuds IoT sans crainte de les mettre en panne.

lisateurs, le tout sans aller sur place. Certains développeurs s'inquiéteront du minimum exigé de 4 Mo de flash et de la seule prise en charge actuelle d'ESP32. Toutefois, si une demande du marché pour d'autres MCU apparaît, il ne semble pas y avoir de raison technique qui empêcherait la prise en charge d'autres plateformes.

210652-04

Des questions, des commentaires ?

Envoyez un courriel à l'auteur (stuart.cording@elektor.com) ou contactez Elektor (redaction@elektor.fr).

Contributeurs

Texte et illustrations : **Stuart Cording**
Rédaction : **Jens Nickel, C. J. Abate**
Mise en page : **Harmen Heida**
Traduction : **Yves Georges**

PRODUITS

- Livre en anglais « BBC micro:bit » de D. Ibrahim, Elektor, 2016
  www.elektor.fr/17872
- BBC micro:bit Go Set de Joy-IT
  www.elektor.fr/18930

LIENS (TOUS EN ANGLAIS)

[1] Dennis Ritchie, Musée de l'histoire de l'informatique : https://bit.ly/3oOXG1A
[2] P. Jansen, « TIOBE Index for December 2021 », TIOBE Software BV, 12/2021 : https://bit.ly/3EXx8Ri.
[3] S. Cass, « Top Programming Languages 2021 », IEEE Spectrum, 2021 : https://bit.ly/3oPJq8z.
[4] Site de la 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, 10/2018 : https://bit.ly/3dO3zWl.
[7] Site de la Fondation Rust : https://bit.ly/3DNgO45
[8] Site rustup : https://bit.ly/3EUTiDR
[9] Site Rust, registre Crate : https://bit.ly/3dLKmor
[10] droogmic, « MicroRust », 04/2020 : https://bit.ly/3EUWFdM.
[11] droogmic, « MicroRust: Buttons », 04/2020 : https://bit.ly/3DWes30
[12] bors et NitinSaxenait, « Discovery », 12/2021 : http://bit.ly/3GERDT7.
[13] Site Get Ada Now : https://bit.ly/3GHyikw
[14] F. Chouteau, « First beta release of Alire, the package manager for Ada/SPARK », AdaCore, 10/2020 : https://bit.ly/3EUXOSC.
[15] F. Chouteau, « Microbit_examples », 12/2021 : https://bit.ly/3mnH7bx.
[16] « Watch us turn an ESP32 into a full computer! », Toit, 03/2021 : https://bit.ly/3IM0WT8.
[17] Page du registre des logiciels Toit : https://bit.ly/3yokpo7
[18] Docs Toit, prérequis, site : https://bit.ly/3oNSECq