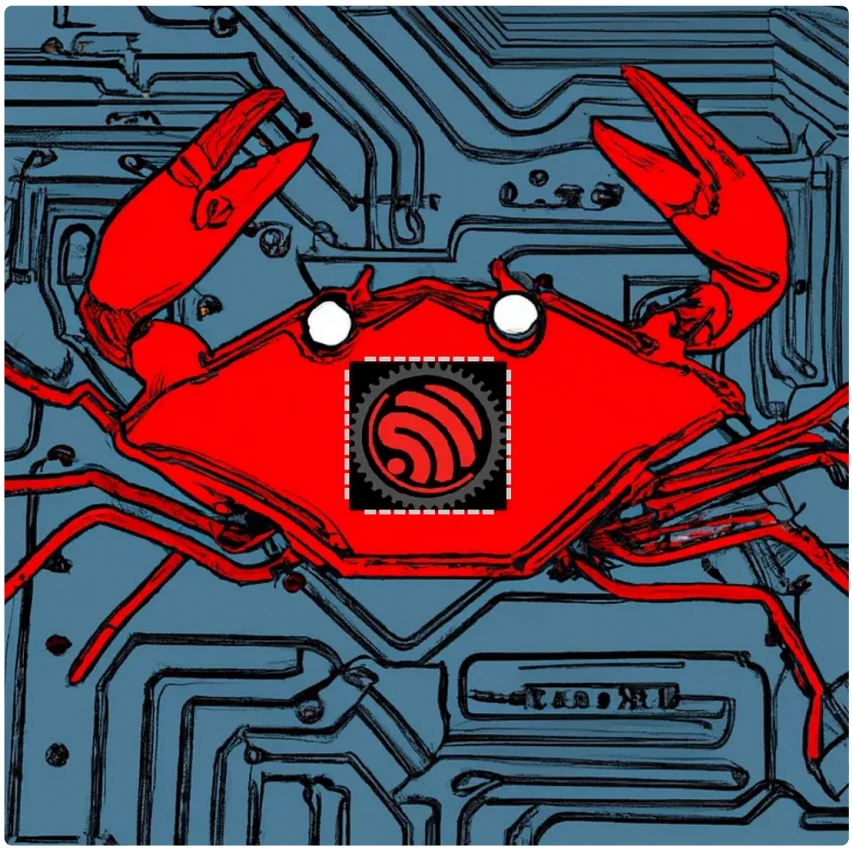
Rust + les systèmes embarqués

deux outils puissants pour le développement

Juraj Sadel (Espressif)

Principalement axé sur la sécurité de la mémoire et des processus, Rust est un langage populaire permettant la réalisation de logiciels fiables et sécurisés. Rust est-il pour autant une solution adéquate pour les applications embarquées ? Comme vous allez l'apprendre, Rust offre de nombreux avantages, en comparaison avec les langages de développement traditionnels pour les solutions embarquées, telles que C et C++, en particulier protection de la mémoire, support du parallélisme et performances.



Source : Image générée par DALL-E.

Au niveau mondial, Rust est devenu le nouveau langage le plus en vogue, de plus en plus de personnes s'y intéressent chaque année. Il est efficient sans le domaine de la sécurité de la mémoire et des processus concurrentiels, il se présente avec l'intention de produire des logiciels fiables et sécurisés. Le support de la concurrence des processus et du parallélisme est particulièrement remarquable dans le domaine du développement des logiciels embarqués où l'utilisation optimisée des ressources est critique.

Les débuts de Rust

L'idée initiale du langage de programmation Rust est née par accident. En 2006, à Vancouver, M. Graydon Hoare rentrait chez lui, mais en raison d'un bug logiciel, l'ascenseur était à nouveau en panne, M. Hoare vivait au 21ème étage, alors qu'il montait les escaliers, il réfléchissait. « Nous les informaticiens, nous ne sommes même pas capables d'assurer le contrôle d'un ascenseur sans qu'il ne dysfonctionne ! »

Cet incident amena M. Hoare à réfléchir à la conception d'un nouveau langage de programmation. Il souhaitait rendre possible l'écriture d'un programme compact, rapide, sans défauts mémoire. [1] Si vous êtes intéressé par davantage d'information technique détaillée sur l'histoire de Rust, merci de visiter [2] et [3].

Environ 18 ans après, Rust est devenu mondialement le langage le plus en vogue, et de plus en plus de personnes s'y intéressent chaque année. Au premier trimestre 2020, il y avait environ 600 000 développeurs Rust, et au premier trimestre 2022, ce nombre a franchi la barre des 2,2 millions. [4] Des sociétés importantes telles que Mozilla, Dropbox, Cloudflare, Discord, Facebook (Meta), Microsoft, et davantage utilisent Rust dans leurs logiciels. Dans les six dernières années, le langage Rust était le langage de programmation le plus « aimé » [5]

Développement des systèmes embarqués

Le développement des systèmes embarqués n'est pas aussi populaire que le développement web ou celui des logiciels personnels, voici quelques raisons pour lesquelles cela pourrait se révéler exact :

- Contraintes matérielles : les systèmes embarqués ont en général des ressources matérielles limitées, notamment au niveau de la mémoire et des performances. Cela rend plus difficile le développement du logiciel de ces systèmes.
- Marché niche plus réduit : le marché des systèmes embarqués est plus limité que celui des applications web ou personnelles, de ce fait, ils sont moins rémunérateurs pour les développeurs spécialisés en logiciels embarqués.
- Connaissances de bases spécifiques : la connaissance très spécialisée du matériel et des langages de programmation de bas-niveau est un besoin fondamental en développement de systèmes embarqués.
- Cycles de développement plus longs : le développement d'un logiciel pour système embarqué prend davantage de temps que celui d'une application personnelle ou web, en raison de la nécessité de tester et optimiser le code créé pour un matériel spécifique.
- Langages de programmation de bas-niveau : ces langages, comme par exemple l'assembleur ou le langage C, n'apportent pas une grande abstraction au programmeur et permettent l'accès direct aux ressources matérielles et à la mémoire, ce qui favorise les erreurs de programmation.

Seuls quelques exemples permettent de comprendre pourquoi et comment le développement de logiciels embarqués est unique est n'est pas aussi réputé et lucratif pour les jeunes programmeurs que ne l'est la programmation web. Si vous êtes habitués aux langages de programmation modernes les plus utilisés, comme Python, Javascript, ou C# avec lesquels vous n'avez pas à tenir compte de chaque cycle du processeur ou de chaque kilooctet de mémoire utilisé, débuter le développement de systèmes embarqués est un changement radical. Il peut être très décourageant d'aborder

Le support de la simultanéité et du parallélisme de Rust est particulièrement important pour le développement des systèmes embarqués où l'utilisation optimisée des ressources est importante.

le monde des systèmes embarqués, pas uniquement pour les débutants, mais aussi pour les développeurs expérimentés d'applications web, personnelles ou mobiles. C'est pourquoi il serait très intéressant, et nécessaire, de pouvoir disposer d'un langage de programmation moderne pour les systèmes embarqués.

Pourquoi Rust ?

Rust est un langage moderne relativement jeune qui se concentre sur la sécurité des accès mémoire et des processus, dont l'objectif est de produire des logiciels fiables et sécurisés. Le support de la simultanéité et du parallélisme est particulièrement important pour le développement de logiciels embarqués où l'utilisation optimisée des ressources est critique. La popularité grandissante de Rust et son écosystème en font une option intéressante pour les développeurs, particulièrement pour ceux qui recherchent un langage efficace et sécurisé. Ce sont les principales raisons pour lesquelles Rust devient un choix populaire, pas uniquement pour le développement des systèmes embarqués, mais particulièrement pour les projets qui ont pour priorité la sécurité, la protection et la fiabilité.

Avantages (en comparaison avec C/C++)

Passons en revue les avantages notables de Rust.

- Protection mémoire : Rust garantit une forte protection de la mémoire par son système de possession/prêt qui est très utile pour prévenir les bugs relatifs aux accès mémoire, tels que les pointeurs nuls, les déréférencements ou les dépassements de capacité des mémoires tampons. En d'autres termes, Rust garantit l'intégrité mémoire dès la phase de compilation grâce à son mécanisme possession/prêt. Ceci est particulièrement important dans les systèmes embarqués dans lesquels les ressources mémoire limitées rendent ces erreurs plus difficiles à localiser et corriger.
- Simultanéité : Rust apporte un excellent support pour les abstractions et la simultanéité sécurisée et le multi-tâche avec une syntaxe `async/await` standard et un type système puissant qui empêche les erreurs classiques telles que la concurrence des données. Cela facilite l'écriture de code concurrentiel sécurisé et performant, pas uniquement pour les systèmes embarqués.
- Performances : Rust est conçu pour obtenir des performances élevées et peut rivaliser avec C et C++ en ce domaine, tout en garantissant une sécurité mémoire élevée et le support de la simultanéité.
- Lisibilité : la syntaxe de Rust est conçue afin d'offrir une lisibilité accrue et de moins favoriser les erreurs que C ou C++, comme par exemple, le filtrage de vraisemblance, l'inférence de type, et la programmation fonctionnelle. Ceci facilite l'écriture et la maintenance du code, en particulier pour les projets complexes et les plus importants
- Essor de son écosystème : Rust possède un écosystème de bibliothèques grandissant (crates), d'outils, et de ressources pour (mais pas uniquement) le développement des systèmes embarqués. Rust le rend plus facile à aborder et pour trouver le support et les ressources nécessaires à un projet particulier.
- Gestionnaire de paquets et système de compilation : la distribution de Rust inclut un outil officiel appelé Cargo, qui est utilisé pour automatiser les processus de construction, test et publication, tout en permettant la création d'un nouveau projet et ses dépendances.

Désavantages (en comparaison avec C/C++)

Malgré tout, Rust n'est pas un langage parfait, il présente également quelques désavantages par rapport à d'autres langages (pas uniquement vis-à-vis de C ou C++).

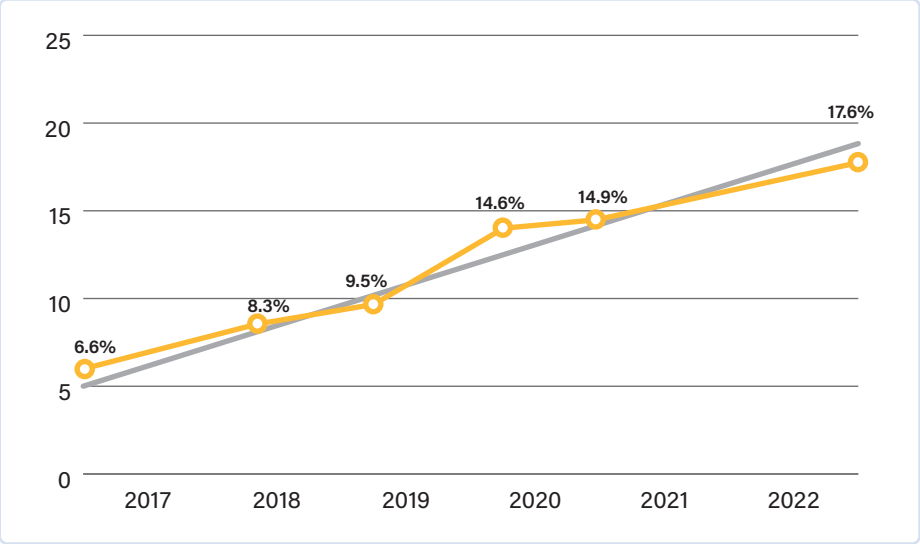


Figure 1. Accroissement du pourcentage de développeurs désirant développer en Rust. (Source : Yalantis [4])

- Temps d'apprentissage : Rust a une courbe d'apprentissage plus raide. Ses caractéristiques uniques, comme par exemple le mécanisme possession/prêt déjà mentionné, prennent davantage de temps pour leur compréhension et s'habituer à les utiliser, ce qui rend les débuts avec Rust plus difficiles.
- Temps de compilation : les particularités système et le vérificateur de prêt rendent le temps de compilation plus important par rapport à celui d'autres langages, en particulier dans le cas de projets d'envergure.
- Utilitaires : bien que l'écosystème de Rust croisse rapidement, il n'a pas encore atteint le même niveau de support que la plupart des autres langages de programmation. À titre d'exemple, C et C++ sont présents depuis plusieurs décennies et possèdent une base importante de code disponible. Cela peut rendre plus difficile la recherche et l'utilisation des outils appropriés à un projet spécifique.
- Manque de possibilités de contrôle de bas-niveau : les caractéristiques de protection de Rust imposent l'utilisation de C ou C++ pour le contrôle de bas-niveau. Cela rend plus délicat certaines opérations d'optimisation à bas-niveau ou l'interaction directe avec le matériel, mais cela reste possible.
- Taille de la communauté : Rust est un nouveau langage de programmation relativement jeune, comparativement avec les langages bien établis tels que C et C++. Cela implique que sa communauté de contributeurs et développeurs est plus réduite et qu'il dispose de moins de ressources, de bibliothèques de code et d'utilitaires.

Malgré tout, Rust présente de nombreux avantages par rapport aux langages de développement tels que C et C++, en particulier en matière de sécurité, simultanéité, performances, lisibilité du code et du fait de son écosystème en plein essor. De ce fait, Rust devient un choix de plus en plus populaire pour le développement des systèmes embarqués, notamment pour les projets qui imposent protection, sécurité et fiabilité. Les désavantages se Rust, par rapport à C et C++ sont surtout la conséquence de la jeunesse du langage Rust et de ses caractéristiques uniques. Malgré cela, de nombreux développeurs considèrent que les avantages de Rust le rendent compétitif pour certains projets.

Comment Rust fonctionne-t-il ?

Il existe plusieurs façons d'exécuter les logiciels basés sur Rust, selon l'environnement et les nécessités de l'application. Un code créé par Rust peut être utilisé selon deux modes : *environnement hôte* ou *bare-metal*. Voyons ce dont il s'agit.

Qu'est-ce que l'environnement hôte ?

L'*environnement hôte* de Rust s'approche de l'environnement habituel d'un PC [6], ce qui signifie qu'un système d'exploitation est présent. Avec un système d'exploitation, on peut créer la bibliothèque standard Rust (*std*). [7] Le terme *std* se réfère à la bibliothèque standard que l'on peut voir comme une collection de modules et types présents dans chaque installation Rust. La bibliothèque *std* offre un ensemble de fonctionnalités multiples permettant de créer des programmes Rust, incluant des structures de données, le support réseau, les mutex (accès exclusif à des ressources partagées) ainsi que les primitives de synchronisme, les entrées/sorties et bien davantage.

Selon l'approche de l'*environnement hôte*, on peut utiliser les fonctionnalités du cadre de développement C nommé ESP-IDF [8] car il offre l'environnement *newlib* [9], suffisamment puissant pour créer la bibliothèque standard *std* de Rust. En d'autres termes, grâce à l'*environnement hôte* de Rust (parfois nommé simplement *std*), on utilise ESP-IDF comme système d'exploitation et pouvons créer une application Rust superposée. De cette façon, on peut utiliser toutes les possibilités de la bibliothèque standard mentionnées précédemment, mais incorporer également des fonctionnalités C à partir des API ESP-IDF.

Dans le **listage 1**, vous pouvez voir comment se présente un exemple de clignotement d'une LED [10] développé à l'aide d'ESP-IDF (FreeRTOS). D'autres exemples se trouvent dans l'*esp-idf-hal* [11].

Quand est-il souhaitable d'utiliser l'environnement hôte ?

- Richesse des fonctionnalités : si votre système embarqué nécessite de nombreuses fonctionnalités, telles que le support de protocoles de communication, les entrées/sorties fichiers, ou des structures de données complexes, vous choisirez probablement l'approche de l'*environnement hôte* du fait de la présence de la bibliothèque *std* qui offre une large gamme de fonctions pouvant être utilisées pour la création relativement rapide et performante d'applications complexes.
- Portabilité : le contenu de la bibliothèque *std* apporte un ensemble d'API standardisées pouvant être utilisées avec un grand nombre de plateformes et architectures différentes, simplifiant l'écriture d'un code portable et réutilisable.
- Rapidité de développement : le contenu de *std* offre un ensemble de fonctionnalités riches pouvant être utilisées pour développer rapidement et efficacement des applications, sans devoir se préoccuper des détails de bas-niveau.

Qu'est-ce que le mode *Bare-Metal* ?

Bare-Metal (signifiant métal nu en anglais) signifie que l'on travaille sans système d'exploitation. Quand un programme

Listage 1. Exemple de clignotement d'une LED en mode environnement hôte et 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);
     }
 }
```

Rust est compilé avec l'attribut `no_std`, cela signifie que le programme n'aura pas accès à certaines possibilités. Cela ne veut pas nécessairement dire qu'en mode `no_std`, vous ne pouvez pas utiliser la communication réseau ou des structures complexes. Sans `std`, vous pouvez réaliser les mêmes fonctions qu'en utilisant std, mais cela est plus complexe et difficile. La programmation `no_std` se base sur un ensemble de caractéristiques du langage qui sont présentes dans tous les environnements Rust, par exemple les types de données, structures de contrôle ou management de la mémoire à bas-niveau. Cette approche est utile pour la programmation des systèmes embarqués où l'utilisation de la mémoire est souvent contrainte et le contrôle direct du matériel nécessaire.

Dans le **listage 2**, vous trouverez comment se présente un exemple de clignotement d'une LED [12] fonctionnant en mode *bare-metal* (sans système d'exploitation). D'autres exemples se trouvent dans l'*esp-hal* [13].

Quand est-il souhaitable d'utiliser le mode Bare-Metal ?

- Taille mémoire réduite : si votre système embarqué a des ressources limitées et possède une taille mémoire réduite, il sera préférable d'utiliser le mode *bare-metal* car les fonctionnalités de *std* augmentent significativement la taille mémoire de son module exécutable et allonge le temps de compilation.
- Contrôle direct des ressources matérielles : si votre système embarqué nécessite davantage de contrôle direct du matériel, comme par exemple des pilotes de bas-niveau ou l'accès à des dispositifs matériels spécialisés, vous devrez préférer l'utilisation du mode *bare-metal* car *std* ajoute de l'abstraction rendant difficile l'interaction directe avec le matériel.
- Contraintes du temps réel ou applications au timing critique : si votre système embarqué nécessite des performances en temps-réel ou des temps de réponse à faible latence, le mode *bare-metal* sera préférable, car *std* introduit des délais imprévisibles et une surcharge qui peut altérer les performances.
- Besoins spécifiques : le mode *bare-metal* permet davantage de personnalisation et le contrôle précis du comportement d'une application ce qui peut être utile dans le cas des environnements spécialisés ou non usuels.

Devez-vous passer de C à Rust ?

Si vous commencez un nouveau projet ou une tâche dans laquelle la sécurité mémoire ou la simultanéité est exigée, il pourrait être profitable de considérer le passage de C à Rust. Néanmoins, si votre projet est déjà bien entamé et fonctionnel en C, le bénéfice du passage à Rust ne justifierait probablement pas le coût induit par la réécriture et le test de l'ensemble de votre code. Dans ce cas, il vous sera possible d'envisager de conserver l'ensemble de votre code C et d'utiliser Rust pour ajouter de nouvelles possibilités, modules et fonctionnalités. Il est relativement facile d'appeler des fonctions C à partir du code Rust. Il est également possible de créer des composants ESP-IDF en utilisant Rust [14]. En définitive, le passage de C à Rust doit suivre une évaluation spécifique des besoins et des bénéfices attendus.

VF : Jean Boyer — 230569-04

Questions ou commentaires ?

Envoyez un courriel à l'auteur (juraj.sadel@espressif.com) ou contactez Elektor (redaction@elektor.fr).

Listage 2. Exemple de clignotement d'une LED en mode bare-metal

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

À propos de l'auteur

Juraj Sadel est un développeur de systèmes embarqués passionné par Rust, il se consacre à l'amélioration des systèmes embarqués. Il est également un membre actif de l'équipe Rust, faisant bénéficier la communauté de son expertise et de son enthousiasme pour la technologie de pointe d'Espressif.

Produits

- **J. Long, *Embedded in Embedded* (Elektor 2018)**
  version papier : www.elektor.fr/18876
  version numérique : www.elektor.fr/18877

- **A. He and L. He, *Embedded Operating System* (Elektor 2020)**
  version papier : www.elektor.fr/19228
  version numérique : www.elektor.fr/19214

LIENS

[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 library: https://sourceware.org/newlib
[10] Blinky example running on top of 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] Blinky example running on bare-metal: 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 components in Rust: https://github.com/espressif/rust-esp32-example