Rust sur systèmes embarqués

ROUBIA Akram, MEMIL Dila

Résumé :

Le langage RUST gagne en popularité d'année en année, et ce que ce soit pour de simples petites applications personnelles, ou des projets à grande (voire très grande) échelle. Il se distingue des autres langages de programmation à travers son ingénieux système de référencement et d'emprunt qui permet une grande maniabilité sans négliger la sécurité ou les performances. Ses avantages font en sorte que RUST est de plus en plus utilisé dans la programmation embarquée. Pour illustrer cet usage, nous avons choisi d'utiliser ARIEL OS : un système d'exploitation embarqué en RUST, basé sur Embassy et tournant sur plusieurs familles de microcontrôleurs tels que nRF5x, RP2xxx, STM32 et ESP32. Cet outil intègre plusieurs fonctionnalités facilitant la programmation (multi-core scheduling, networking, RNG, embedded-test ...) et permet une grande portabilité du code à l'aide de son Hardware Abstraction Layer qui fait tourner le même code RUST sur les différentes familles de microcontrôleurs.

Mots-clés :

Programmation embarquée, RUST, ARIELOS, sécurité, modularité

Abstract :

The RUST language is gaining popularity year after year, whether for simple small personal applications or large (even very large) scale projects. It stands out from other programming languages through its ingenious referencing and borrowing system, which allows for great flexibility without compromising security or performance. Its advantages mean that RUST is increasingly used in embedded programming. To illustrate this use, we have chosen to use ARIEL OS: an embedded operating system in RUST, based on Embassy and running on several families of microcontrollers such as nRF5x, RP2xxx, STM32, and ESP32. This tool integrates several features that facilitate programming (multi-core scheduling, networking, RNG, embedded testing, etc.) and allows for high code portability using its Hardware Abstraction Layer, which easily runs the same RUST code on different microcontroller families.

Keywords:

Embedded programming, RUST, ARIELOS, security, modularity

Partie 1 : RUST, un langage qui se développe

Depuis sa création par Graydon Hoare (chez Mozilla), Rust est devenu l'un des langages de programmation les plus prometteurs et les plus dynamiques de ces dernières années. Ses premières versions remontent à 2015, mais ce qui distingue Rust, c'est son ambition initiale : combiner la sécurité, la concurrence, et la rapidité, objectifs que peu de langages atteignent simultanément.

Rust est aujourd'hui supporté par une communauté active et croissante. Son site officiel présente un écosystème riche : gestionnaire de paquets, compilation native, bonne documentation, outils modernes.
De plus, de très nombreux domaines d'application sont couverts : serveurs réseau, applications Web, services backend, systèmes embarqués, outils en ligne de commande, etc.

L'un des marqueurs forts de l'adoption de Rust est l'intérêt croissant des acteurs industriels : des entreprises variées (dans l'automobile, le ferroviaire, l'aéronautique, le cloud, ou encore le spatial) envisagent d'utiliser Rust pour ses garanties de sûreté et ses performances.
Des événements comme Rust Paris (organisé par Systematic Paris-Region) témoignent de cette dynamique : des conférences annuelles réunissent développeurs, industriels et décideurs pour échanger sur les usages concrets du langage dans l'industrie, le web, l'embarqué, la cybersécurité, etc.

Ce succès n'est pas qu'un effet de mode : Rust repose sur des fondations techniques solides.

- Performance : Rust compile en code natif, sans garbage-collector, ce qui lui permet d'exécuter des tâches exigeantes avec une grande efficacité, comparable aux langages classiques bas-niveau comme C ou C++.

- Sécurité mémoire & sûreté des threads :Grâce à un modèle d'ownership (propriété des données), de borrowing et de lifetimes, Rust élimine à la compilation de nombreuses catégories d'erreurs (fuites mémoire, accès concurrents dangereux, data races).

- Fiabilité et robustesse du code : L'approche stricte de Rust autour de la mémoire et des types rend les programmes plus fiables, particulièrement importants dans les applications critiques.

- Productivité & modernité : Rust fournit un gestionnaire de paquets, des outils modernes, une syntaxe expressive (traits, enums, pattern matching, macros), ce qui le rend plus agréable et productif que les environnements traditionnels très "bas niveau".

Ainsi, Rust n'est plus simplement un "langage système parmi d'autres" : il combine les atouts des langages bas-niveau avec la modernité et la sûreté attendues aujourd'hui. C'est

cette alchimie qui explique son adoption croissante, tant par des passionnés que par des acteurs industriels.

Partie 2 : L'embarqué, les usages et l'intérêt de RUST

Un des axes où Rust brille particulièrement est l'embarqué (systèmes contraints, microcontrôleurs, objets connectés, etc.). En combinant performance, faible empreinte, sécurité mémoire et richesse d'écosystème, Rust s'impose comme un excellent candidat pour le développement embarqué.

- Faible consommation & efficacité : Dans l'embarqué, les ressources sont limitées (mémoire, CPU, énergie). Rust, compilé en code natif, se prête naturellement à ces environnements contraints.

- Sécurité & fiabilité : Les systèmes embarqués souvent critiques (automobile, aéronautique, IoT, robotique) ne tolèrent pas les erreurs mémoire. Rust, par son modèle d'ownership, garantit la sûreté au moment de la compilation.

- Polyvalence d'usage : Que l'on vise des microcontrôleurs, des systèmes en temps réel, des serveurs réseau embarqués, ou des composants bas-niveau, Rust offre un pont entre contrôle bas-niveau et confort de haut-niveau.

L'industrie a commencé à regarder sérieusement Rust pour des projets embarqués et critiques. Dans des secteurs comme l'automobile, l'aéronautique, le ferroviaire ou le cloud, Rust est perçu comme un moyen de construire des systèmes sûrs, performants et durables.

Par exemple :

- Dans l'automobile, l'utilisation de Rust peut améliorer la sûreté logicielle des systèmes embarqués, tout en réduisant le risque de failles liées à la mémoire ou à la concurrence.

- Pour des systèmes critiques ou des applications IoT, où l'on combine contraintes de ressources, besoins de fiabilité, communication réseau, sécurité, Rust est un excellent compromis entre bas-niveau (contrôle du hardware) et abstractions modernes.

De plus, l'écosystème Rust propose désormais des ressources spécifiques pour l'embarqué (frameworks, crates, documentation …) ce qui facilite le développement, même pour des équipes habituées à des langages comme C/C++.

Tout n'est pas parfait : bien que Rust séduise, certains développeurs relèvent des obstacles. D'après un rapport récent, une part des utilisateurs trouve que Rust reste "complexe à

appréhender", notamment à cause de ses concepts mémoire (ownership, borrowing, lifetimes) et de sa relative jeunesse dans l'industrie.
De plus, malgré la montée en puissance, l'adoption dans certaines industries reste modérée, ce qui peut limiter la disponibilité de bibliothèques "business ready" ou d'exemples pour cas d'usage complexes.
Enfin, même si l'écosystème s'est beaucoup enrichi, la maîtrise de Rust pour l'embarqué requiert encore un bon niveau de compréhension des systèmes bas-niveau — ce qui peut représenter une barrière pour des équipes habituées à langages plus "haut-niveau".

Malgré ces défis, l'intérêt de Rust pour l'embarqué et les usages industriels est réel. L'argument "sécurité + performance + efficience énergétique" est particulièrement fort dans un contexte où l'on cherche à développer des systèmes fiables, durables, et efficaces sur le plan énergétique.
Dans un futur proche, on peut s'attendre à ce que Rust gagne encore du terrain dans des domaines sensibles : automobile, aéronautique, IoT, systèmes embarqués critiques, mais aussi cloud, services distribués, backend haute performance. L'équilibre entre performance et sécurité en fait un langage "polyvalent sécurisé", capable d'accompagner la montée en complexité des architectures modernes.

Partie 3 : Ariel OS

Ariel OS (https://github.com/ariel-os/ariel-os) est un système d'exploitation embarqué spécialement conçu pour les applications IoT sécurisées, à mémoire sécurisée et en réseau fonctionnant sur des microcontrôleurs à faible consommation. Entièrement écrit en Rust, Ariel OS tire parti des garanties de sécurité à la compilation offertes par ce langage pour réduire les vulnérabilités courantes observées dans les systèmes traditionnels basés sur le langage C. Il prend en charge toute une gamme d'architectures de microcontrôleurs 32 bits, notamment Cortex-M, RISC-V et Xtensa, garantissant ainsi une large compatibilité matérielle.
Ariel OS s'appuie sur les bases solides de l'écosystème Embedded Rust. Il intègre plusieurs bibliothèques et frameworks de haute qualité, tels que Embassy pour l'exécution asynchrone, esp-hal pour l'abstraction matérielle, defmt pour un débogage efficace, probe-rs pour l'interface des périphériques, sequential-storage pour les données persistantes et embedded-test pour les tests matériels clés en main. En plus de cela, Ariel OS fournit des fonctionnalités supplémentaires du système d'exploitation qui ne sont pas disponibles « prêtes à l'emploi » dans de nombreuses bibliothèques, notamment :

- Planificateur multicœur préemptif : gère les tâches sur plusieurs cœurs en temps réel.
- API périphériques portables : normalise l'accès aux périphériques sur différents matériels.
- Sécurité réseau renforcée : intègre des couches de sécurité supplémentaires pour les communications réseau.

- Système de méta-construction : utilise un système (appelé « laze ») qui abstrait les différences spécifiques aux cartes, simplifiant ainsi la configuration et réduisant le code standard.

Un des intérêts principaux d'Ariel OS et sur lequel nous nous sommes intéressés est la capacité de faire tourner le même code RUST sur plusieurs familles de microcontrôleurs sans pour autant tout reconfigurer. Cela se fait à travers le Hardware Abstraction Layer et la structure des applications ARIEL OS, qui sont composées de :

- README.md : Documente les fonctionnalités de l'application et fournit des instructions de configuration/d'utilisation pour le matériel pris en charge.

- Cargo.toml : Définit les dépendances Rust (HAL intégré, outils asynchrones) et les métadonnées pour les tests de capteurs multiplateformes.

- laze.yml : Configure les cibles de déploiement multi-appareils (micro:bit, RP2040) et les paramètres de la pile ISR pour les builds embarqués.

- main.rs : Implémente le code de l'application
- pins.rs : Cartographie les configurations de broches spécifiques au matériel pour l'intégration de capteurs portables sur les cartes de microcontrôleurs.

à travers le fichier pins.rs, nous pouvons configurer les broches spécifiques au matériel utilisé selon le protocole (I2C, UART …) et donc de ne pas avoir à le programmer dans le main.rs, et de le réutiliser pour toutes les applications utilisant le même protocole.

Références :

- RUST

https://rust-lang.org/fr/
Site officiel du langage de programmation RUST

- Prieur, Benoît. 5 bonnes raisons d'utiliser le langage Rust. ENI Blog, 22 août 2023. Editions ENI.

https://www.editions-eni.fr/blog/5-bonnes-raisons-dutiliser-le-langage-rust/
Description de quelques atouts de RUST (vitesse, sécurité, fiabilité…)

- Systematic Paris-Region. (8 mars 2024). Rust – Un langage sûr, performant, et éco responsable pour répondre aux enjeux des années à venir.

https://systematic-paris-region.org/rust-un-langage-sur-performant-et-ecoresponsable-pour-repondre-aux-enjeux-des-annees-a-venir/
Description de la dimension éco-responsable de RUST

- Ariel OS : An Embedded Rust Operating System for Networked Sensors & Multi-Core Microcontrollers, E. Frank, K. Schleiser, R. Fouquet, K. Zandberg, C. Amsüss & E. Baccelli, présenté à la conférence IEEE DCOSS-IoT 2025 (Berlin, juin 2025).

https://emmanuelbaccelli.com/Ariel-OS-DCOSS-2025.pdf
Article scientifique décrivant Ariel OS

https://ariel-os.github.io/ariel-os/dev/docs/book/
Book officiel de Ariel OS