Q&A avec Scott Mabin

# qui sont les développeurs de solutions embarquées Rust ?

## comment Espressif développe le langage Rust embarqué pour l'ESP32

**Intro et questions par Stuart Cording (Elektor)**

Il n'a qu'un peu plus de dix ans mais le langage de programmation Rust a déjà atteint la 17ème place dans l'index de la communauté de programmation TIOBE [1]. Il a gagné 194 places dans ce laps de temps pour se classer aux côtés de R, Ruby, Delphi, Scratch et MATLAB. De plus, Linus Torvalds, le père de Linux, a accepté des propositions visant à permettre l'utilisation de code écrit en Rust dans le noyau, ce que le C++ a tenté en vain de faire pendant des années. Le langage Rust a également de plus en plus d'adeptes parmi les développeurs de systèmes embarqués, ce sur quoi Espressif a décidé de s'appuyer.

Le langage C est le langage par défaut des systèmes embarqués depuis des décennies, laissant l'assembleur à ceux qui optimisent à la main ou développent des noyaux en temps réel. Développé dans les années 1970 par Dennis Ritchie alors qu'il travaillait aux Bell Labs, il est étroitement lié à la création du système d'exploitation Unix. Unix a été écrit en assembleur pour le PDP-7 [2] mais a dû être réécrit pour être porté sur le PDP-11 [3] (les PDP étaient des ordinateurs plus petits et polyvalents vendus comme alternatives aux ordinateurs centraux dans les années 1960). Le langage C a permis de développer un code, comme Unix, qui était indépendant du processeur et portable sur de nombreuses architectures différentes.

Les systèmes embarqués étaient initialement programmés en assembleur mais à mesure que le code devenait plus complexe et que les développeurs cherchaient à améliorer la réutilisation, ils ont, eux aussi, été attirés par le langage C. Parmi les autres caractéristiques qui leur ont facilité la vie, citons la prise en charge de la manipulation des bits de bas niveau, la définition fluide des variables, la facilité avec laquelle les algorithmes pouvaient être codés et la simplicité de la manipulation de la mémoire. Cependant, ce dernier point est à l'origine du blocage de la plupart des applications. Si les pointeurs permettent aux programmeurs d'accéder à (presque) n'importe quel emplacement de mémoire à n'importe quel moment, ce qui est très puissant et efficace dans un système embarqué, cela peut aussi être très dangereux. Les pointeurs peuvent pointer vers une mémoire longtemps après qu'elle a été désallouée ou la manipuler pour appeler une fonction qui entraîne une faille de sécurité. En fait, Microsoft attribue environ 70 % des problèmes logiciels en C/C++ à des bogues de corruption de la mémoire [4].

Rust s'attaque à ce problème en renforçant la sécurité de la mémoire. En outre, contrairement au code C/C++, de nombreux problèmes de programmation sont détectés à la compilation plutôt qu'à l'exécution. Enfin, grâce aux similitudes de syntaxe et de performance, les développeurs C et C++ se sentiront rapidement à l'aise, tant sur leur PC que sur les systèmes embarqués. Alors, dans quelle mesure les fabricants de microcontrôleurs prennent-ils au sérieux le langage Rust pour les systèmes embarqués ? Pour en savoir plus, Elektor s'est entretenu avec Scott Mabin, un jeune développeur qui, libéré du carcan des précédents langages, a décidé de se plonger encore plus profondément dans le monde de Rust. Utilisateur assidu des ESP32, il a beaucoup contribué au langage Rust embarqué, ce qui lui a valu d'être embauché par Espressif.

**Stuart Cording : les systèmes embarqués sont traditionnellement programmés en C. Mais vous avez décidé d'ignorer cela et de vous lancer directement dans Rust. Pourquoi ce choix ?**

**Scott Mabin :** j'ai utilisé Rust dans mon projet de fin d'études à l'université. J'ai construit une montre intelligente – pas intelligente selon les normes actuelles, mais elle faisait plus que donner l'heure. Une partie de ce projet visait à déterminer si Rust pouvait remplacer

le langage C pour le développement de micrologiciels et quels étaient les compromis possibles. C'est à ce moment-là que j'en suis tombé amoureux. Je me suis demandé pourquoi tout le monde n'utilisait pas Rust. Il offrait tellement de possibilités qu'une fois compilé, on avait la tranquillité d'esprit de savoir que toute une catégorie de situations de compétition et de mémoire avaient été éliminées. Bien sûr, je comprends que certains projets ont un héritage et que l'on ne veuille pas apprendre et déployer un nouveau langage sans raison valable. Néanmoins, il me semblait insensé qu'il n'y ait pas plus de gens qui se tournent vers Rust.

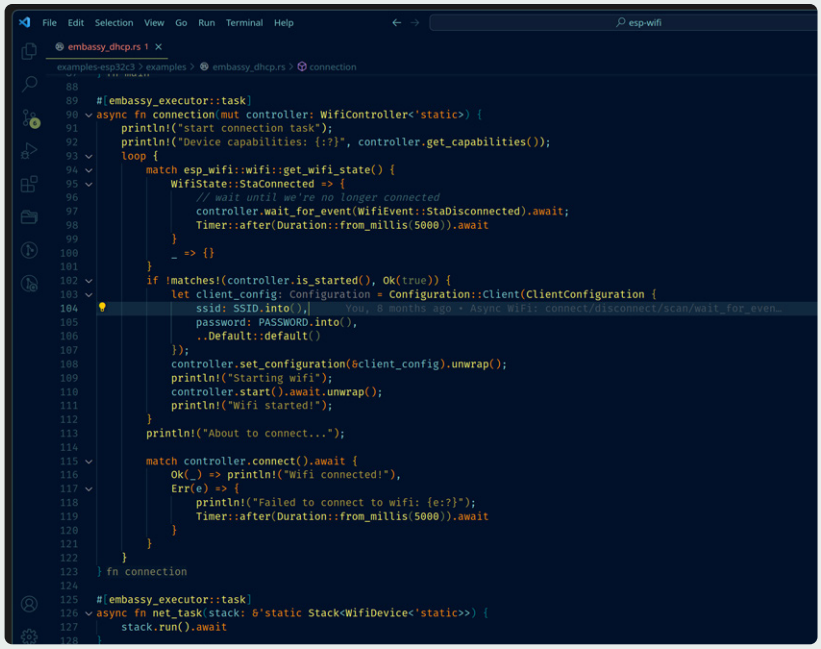
**Stuart Cording : Rust est déjà supporté par la communauté sur STM32 et certains appareils Nordic. Pourquoi avoir choisi Espressif à l'époque ?**
**Scott Mabin :** à la maison, j'avais tous ces ESP32 qui traînaient et je me suis dit : « je peux sûrement aussi programmer ces choses en Rust ? ». Et bien, cela m'a rapidement conduit dans une voie inconnue assez opaque. Le plus gros problème était que le noyau, Xtensa, qui alimente les appareils tels que l'ESP8266, n'était supporté que par le compilateur GCC et non pas par LLVM. Je me suis tourné vers *mrustc* [5], un outil qui convertit Rust en C et peut ensuite être compilé mais ce processus de compilation croisée ne m'a pas convaincu. Heureusement, Espressif a publié un embranchement (fork) pour la chaîne de compilation LLVM afin de supporter Xtensa et, bien qu'initialement difficile à utiliser, cela signifiait que vous pouviez compiler du Rust natif pour les appareils ESP32. Grâce à cela, j'ai écrit mon premier code de LED clignotante en Rust ; enfin, je dis « en Rust » mais il se contentait d'écrire dans des registres et n'était pas très Rust dans sa structure. Cependant, c'était un premier succès que j'ai documenté dans mon premier article de blog [6] relatant mon expérience avec Rust sur ESP32.

**Stuart Cording : vous avez donc commencé à bricoler avec Rust sur ESP32. Comment s'est déroulée la relation avec Espressif ?**
**Scott Mabin :** j'ai créé le groupe *esp-rs* sur GitHub pendant mon temps libre pour rassembler des projets Rust pour ESP32. Avec d'autres membres de la communauté, nous avons mis en place un support pour les périphériques, tels que SPI et autres. Malheureusement, le Wifi nous échappait encore. Puis, après un an de travail communautaire et d'échange sur nos progrès, Espressif a proposé de m'embaucher pour assurer le support de Rust pour leur écosystème. L'équipe d'Espressif s'est toujours montrée très serviable à notre égard, répondant aux demandes de support sur leurs forums ou Reddit. Mais lorsqu'ils m'ont contacté, il est apparu clairement qu'ils s'intéressaient à Rust depuis un certain temps.

**Stuart Cording : y a-t-il des domaines d'application spécifiques qui suscitent cet intérêt pour Rust ou s'agit-il d'une sorte de boule de cristal vaporeuse de la part d'Espressif ? Et quel est leur niveau d'implication ?**
**Scott Mabin :** l'intérêt des clients est indéniable. Les clients utilisent principalement Rust dans des projets IdO ou des applications connectées à l'internet. La plupart des projets sont assez simples, comme les enregistreurs de données. Ensuite, il y a des applications plus complexes telles que la capture de mouvement VR pour le corps entier et le capteur de ciel nocturne en matériel libre. Espressif s'efforce également de préparer l'avenir, en prévision du moment où Rust jouera un rôle plus important dans la vie des développeurs de systèmes embarqués. Environ 10 ou 11 d'entre nous contribuent à l'équipe « Rust Enablement » d'Espressif, dont environ la moitié travaillent à temps plein. De plus, « Rust embedded » dispose d'une communauté impressionnante qui contribue également.

**Stuart Cording : vous avez fait du développement en C embarqué et vous utilisez maintenant Rust quotidiennement sur des microcontrôleurs. Qu'est-ce que les programmeurs C traditionnels doivent prendre en compte lorsqu'ils passent à Rust ?**
**Scott Mabin :** si vous venez du développement en C pur et que vous n'avez aucune expérience du C++, c'est assez difficile. Le changement le plus radical est que Rust nécessite beaucoup d'écriture. Ensuite, il y a l'aspect ressources. Les petits microcontrôleurs dotés de quelques kilo-octets de mémoires SRAM et flash auront du mal à s'adapter en raison des contraintes de mémoire. Dans de tels cas, vous finirez par écrire du Rust de type C pour réduire la taille de la mémoire, perdant ainsi certains des avantages de ce nouveau langage. Vous bénéficierez toujours des avantages de Rust en matière de sécurité de la mémoire, donc ce langage reste donc meilleur que le C. Si on les compare dos à dos, les applications en C et en Rust ont des performances à peu près équivalentes. Dans certains cas, Rust est meilleur, mais la taille du programme est généralement plus importante.

**Stuart Cording : alors que les fournisseurs de microcontrôleurs offrent un excellent support pour les pilotes de périphériques, le contrôle de la version de tout ce code pour le développeur en tant qu'intégrateur est souvent une belle pagaille. Les bibliothèques (crates) Rust seront-ils une raison de changer de langage de programmation ?**
**Scott Mabin :** oui, je pense que les bibliothèques sont un énorme avantage – un gros bonus qui va au-delà du langage lui-même. Les bibliothèques offrent un écosystème d'empaquetage ou, au minimum, une



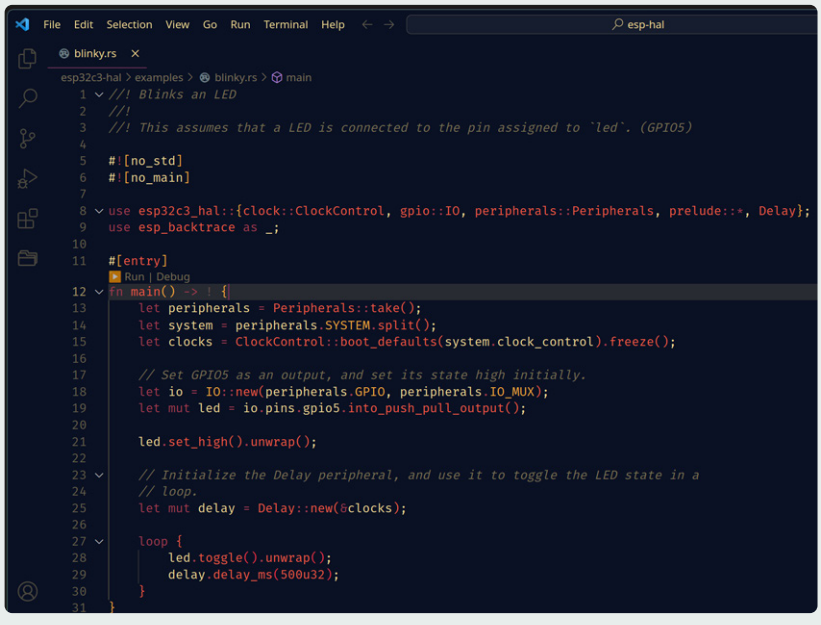

▲

*Figure 1. Exemple d'application Wifi écrite en Rust pour l'ESP32.*

méthode pour intégrer des dépendances sans les définir manuellement dans votre système de construction. La bibliothèque *embedded_hal* [7] offre une interface fonctionnelles pour les périphériques communs, tels que I²C. Si vous prenez maintenant un pilote de capteur de température I²C, il fonctionne simplement par-dessus. Si vous changez de microcontrôleur ou de capteur de température, il fonctionnera toujours, ce qui est très sympa. Mais cela ne fait qu'effleurer la surface. Les bibliothèques telles que *heapless* [8] permettent la création de structures de données allouées statiquement telles que les files d'attente, les vecteurs, les tables de hachage, les ensembles de hachage et les chaînes de caractères. Dans tous les projets C sur lesquels j'ai travaillé, quelqu'un s'assied et écrit une file d'attente, passant de nombreuses heures à perfectionner une structure de données de programmation standard. Ce temps peut désormais être consacré au développement d'applications.

*Figure 2. Scott Mabin recommande VS Code pour le développement de Rust. Ici, on peut voir le code classique de la LED clignotante.*

▼




**Stuart Cording : tout comme le C, Rust propose une bibliothèque standard. Les développeurs d'applications embarquées détestent la bibliothèque standard, alors peuvent-ils aussi se passer de celle de Rust ?**

**Scott Mabin :** Historiquement, je pense que nous avons eu du mal à décrire les cas d'utilisation pour lesquels un programmeur embarqué professionnel devrait utiliser une bibliothèque standard [9]. Ils jettent un coup d'œil et se disent « c'est beaucoup trop de travail » et je suis tout à fait d'accord avec eux. Cependant, la bibliothèque standard Rust est vraiment utile à plusieurs égards. Tout d'abord, si vous connaissez Rust mais pas de solution embarquée, ça vous semble familier. Vous avez toutes les tâches, le réseau et d'autres choses auxquelles vous êtes habitués. Deuxièmement, si vous programmez des ESP32 en utilisant l'ESP-IDF [10] (Espressif IoT Development Framework), vous pouvez créer des projets de bibliothèque standard Rust et commencer à travailler. Si vous la comparez à la bibliothèque standard Python, elle est en fait assez légère et, bien qu'il y ait du travail, elle n'est pas aussi mauvaise qu'elle en a l'air. Elle pose quelques principes supplémentaires par rapport à la bibliothèque de base (dont vous aurez besoin au minimum), telles que la disponibilité des tâches et du réseau.

De nombreuses personnes développent des systèmes Rust embarqués sans la bibliothèque standard (*no_std* [11]) et, si elles savent ce qu'elles font, optent pour l'approche *no_std* si cela a du sens pour le projet. Cependant, vous devrez choisir les éléments que vous voulez et les fusionner ensemble. Ainsi, par exemple, la bibliothèque Wifi contient un exemple de conversation avec un serveur HTTP ou de mise en place d'un point d'accès et d'exécution d'un serveur (**figure 1**). Vous pouvez faire dans *no_std* tout ce que vous pouvez faire avec *std* – c'est juste plus de travail. Avec *std*, c'est un environnement « tout inclus ».

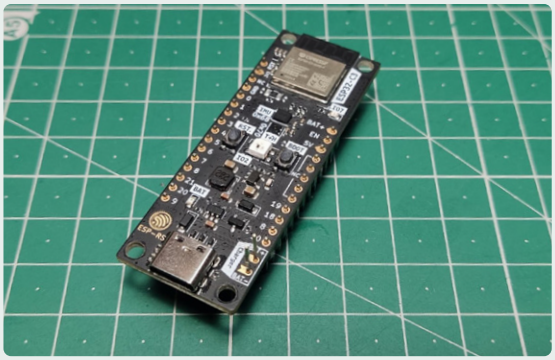
**Stuart Cording : une partie du problème réside-t-elle dans le fait qu'il est temps d'accepter que nous avons besoin de microcontrôleurs dotés de grandes mémoires ?**

**Scott Mabin :** Je pense que, quelle que soit la taille des microcontrôleurs, il y aura toujours quelqu'un qui voudra fabriquer un million d'appareils et qui aura besoin de la solution la plus petite et la moins chère possible. Il y aura toujours un cas d'utilisation pour l'assembleur et le C – et peut-être même le Rust – dans un contexte spécifique pour cette petite application compacte. Mais je pense qu'il y a une tendance générale vers des microcontrôleurs plus gros et que l'expérience du développeur devient prioritaire sur les coûts du matériel. Ce n'est qu'une observation que j'ai faite au cours de ma courte période dans l'industrie, mais c'est ce qui semble se dessiner.

**Stuart Cording : Rust n'en est qu'à ses débuts. Comment Espressif va-t-il assurer la formation des développeurs ?**
**Scott Mabin :** nous nous sommes associés à Ferrous Systems [12], une société de conseil en Rust, pour proposer un cours de formation que nous avons rendu open source. Il existe des supports de formation pour l'approche de la bibliothèque standard et nous avons presque terminé d'ajouter l'approche de la bibliothèque non standard.

**Stuart Cording : qu'en est-il des environnements de développement et des outils de débogage ?**
**Scott Mabin :** j'admire l'esthétique d'environnements tels que vim, Neovim ou Emacs, mais comme je n'ai pas l'expérience de ces outils, ils me gênent. J'utilise donc VS Code (**figure 2**) qui fait l'affaire pour moi. Avec les ESP plus récents (C3 et plus), nous avons un module appelé USB Serial JTAG. Il s'agit d'un périphérique intégré extrêmement petit qui offre un port série et un dispositif JTAG (**figure 3**). Pour l'utiliser, il suffit de le connecter au port USB de votre PC et il est détecté automatiquement par OpenOCD ou *probe-rs*, une boîte à outils de débogage Rust dont l'objectif est similaire à celui d'OpenOCD.

**Stuart Cording : la prise en charge d'un nouveau langage sur un processeur est une tâche énorme. Où en êtes-vous et que reste-t-il à faire ?**
**Scott Mabin :** nous avons pratiquement tout mis en place pour la bibliothèque standard mais nous n'avons pas tout couvert pour ESP-IDF. ESP-IDF est vaste, existe depuis des années et est écrit en C. Nous utilisons donc *bindgen* pour créer des liens avec Rust. Fondamentalement, nous devons examiner les interfaces qui ne sont pas encore implémentées et créer une API Rust pour elles. Nous devons décider des priorités au cas par cas. Pour *no_std*, nous parlons de programmation en Rust pur, donc nous devons écrire du code pour tout le matériel existant. Nous avons le Wifi, le Bluetooth et le



◀

*Figure 3. L'ESP32-C3 intègre un module de débogage USB avec une interface série, ce qui évite d'avoir recours à des sondes externes.*

support Thread sur toutes les puces. Sur une échelle de 0 à 100, je dirais que nous avons fait entre 65 et 70 % du chemin. Malheureusement, il s'agit d'un objectif un peu mouvant car nous devons continuellement prendre en charge de nouveaux appareils au fur et à mesure de leur sortie. Nous avons discuté de la possibilité de créer certains éléments de l'ESP-IDF en Rust lorsque cela s'avérait judicieux. Par exemple, Rust est très bien adapté au passage de flux d'octets. Ainsi, si un nouveau composant est nécessaire, nous le créerons peut-être en Rust et fournirons une API C. Nous devrons voir si les avantages l'emportent sur les efforts.

**Stuart Cording : enfin, où allez-vous, en tant que développeur de matériel embarqué, pour obtenir plus d'informations sur Rust ?**
**Scott Mabin :** Je traîne surtout sur le canal ESP Rust matrix [13] et sur divers autres canaux Rust embedded. À part cela, Google, parfois Reddit ; lire du bon code Rust peut m'aider à comprendre et à apprendre des choses que je ne connaissais pas ⏮

*VF : Chris Elsass* — 230616-04

### Questions ou commentaires ?
Contactez Elektor (redaction@elektor.fr).

### À propos de Scott
Scott Mabin est ingénieur en logiciels embarqués. Après avoir exploré les capacités de Rust embarqué à l'université, il a décidé de l'expérimenter sur l'ESP32 pendant son temps libre. Ces efforts ont conduit Espressif à lui demander de rejoindre leur équipe pour se concentrer sur l'amélioration du support de Rust sur leurs microcontrôleurs sans fil et leurs solutions AIoT.

### LIENS
[1] Index de la communauté de programmation TIOBE : https://tinyurl.com/tioberust
[2] PDP-7 [Wikipédia] : https://tinyurl.com/pdp7wikipedia
[3] PDP-11 [Wikipédia] : https://tinyurl.com/pdp11wikipedia
[4] Équipe du CSEM, « une approche proactive d'un codage plus sécurisé », Microsoft, juillet 2019 : https://tinyurl.com/msrcsecurecode
[5] Projet mrustc : https://tinyurl.com/mrustcproject
[6] S. Mabin, « Rust sur l'ESP32 », septembre 2019 : https://tinyurl.com/rustesp32
[7] Documentation de la bibliothèque embedded_hal : https://tinyurl.com/embeddedhalcrate
[8] heapless Crate Documentation : https://tinyurl.com/heaplesscrate
[9] The Rust on ESP Book, "Using the Standard Library (std)": https://tinyurl.com/rustbookstd
[10] Guide de Programmation ESP-IDF « Get started » : https://tinyurl.com/espidfgetstarted
[11] The Rust on ESP Book, "Using the Core Library (no_std)" : https://tinyurl.com/rustbooknostd
[12] Formation « Embedded Rust on Espressif » : https://tinyurl.com/rustonespressif
[13] Rust user group on Matrix : https://tinyurl.com/rustmatrixug