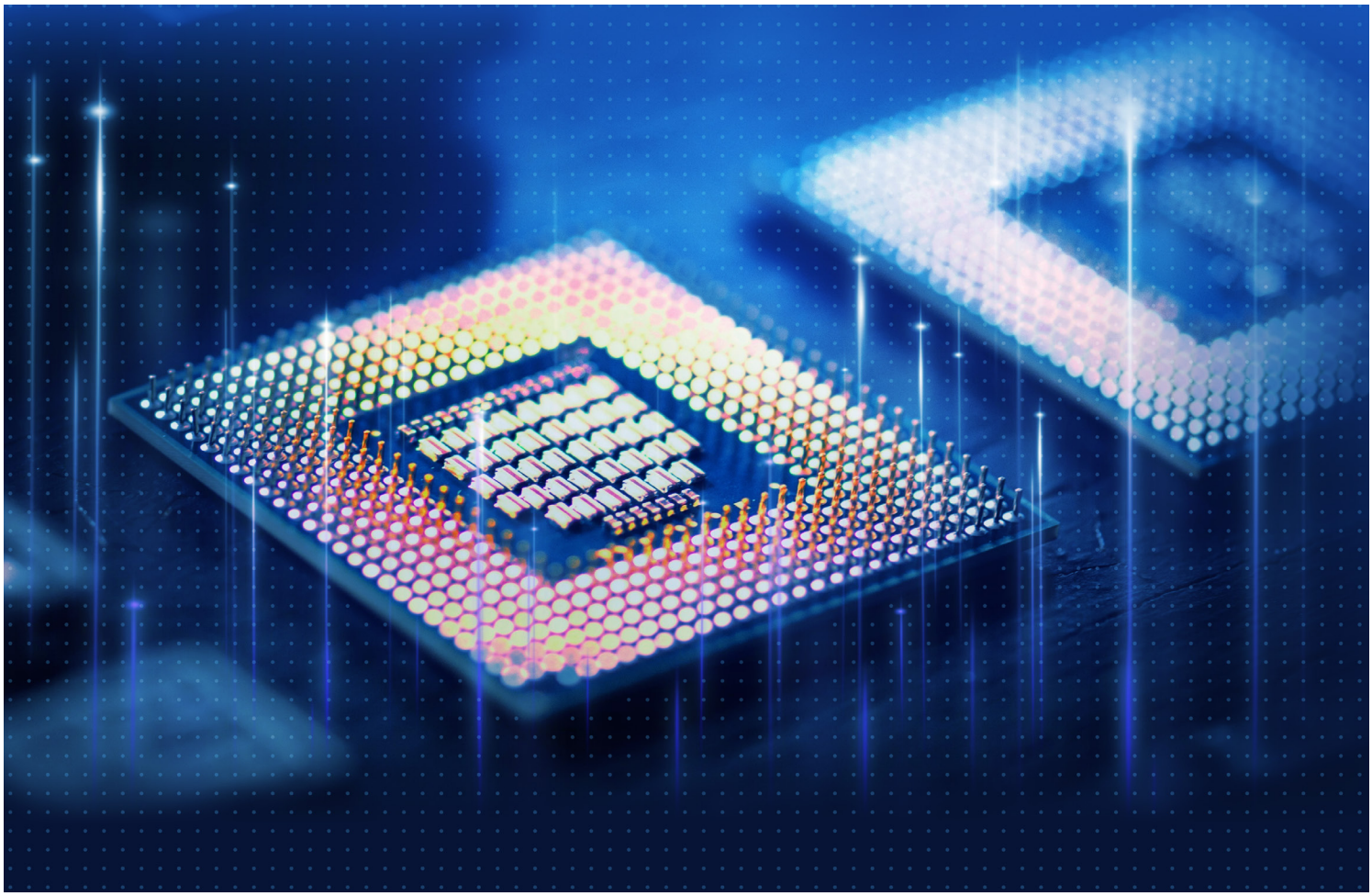
Guide des langages de programmation pour systèmes embarqués

Quels sont les avantages de chaque langage de programmation embarqué ? D'Ada et assembleur à C et Rust

Par Stuart Cording



(Source photo : Adobe Stock – Rawpixel.com)

MOUSER ELECTRONICS

Sommaire

Introduction

Lorsqu'ils entrent dans le monde des systèmes embarqués, les développeurs sont confrontés à une multitude de langages de programmation différents. Chaque langage possède ses avantages et ses inconvénients, mais dans de nombreux cas, le choix du langage ne dépendra pas de vous. Si vous travaillez pour un employeur, vous devrez probablement adopter le langage qu'il utilise pour développer le code de ses applications de microcontrôleurs. En effet, la grande expérience du langage, l'investissement dans les outils utilisés avec la famille de microcontrôleurs préférée et les lignes infinies de code réutilisable que personne ne souhaite modifier le pousseront à conserver son langage de choix.

Bien sûr, votre établi privé vous offre une plus grande marge de manœuvre. Si tel est le cas, ce guide vous présentera certains des langages alternatifs dont vous avez peut-être entendu parler, mais que vous n'avez pas encore eu l'occasion d'essayer.

Nous nous concentrerons sur la version microcontrôleur de « Hello World ! » (l'application LED clignotante) afin de conserver une approche simple et propice aux comparaisons. Au fil de ce guide, vous découvrirez les éléments clés de chaque langage ainsi que les informations nécessaires à l'outillage pour générer du code à destination du microcontrôleur de votre choix.

« Hello World » pour les systèmes embarqués

L'exemple de code « Hello World ! » remonte à des ouvrages comme « The C Programming Language » de Kernighan et Ritchie *(Figure 1)*. Une fois compilé et exécuté, le code affiche le message « Hello World ! » à l'écran.

Figure 1 : *le projet « Hello, World » original utilisé dans « The C Programming Language » de Kernighan et Ritchie (Source : capture d'écran créée par l'auteur)*

Étant donné que les développeurs de logiciels embarqués bare metal, qui conçoivent des logiciels sans utiliser de système d'exploitation, n'ont pas d'écran ou de clavier raccordé à leurs microcontrôleurs, le clignotement d'une LED connectée à un port d'entrée/sortie à usage général (GPIO) est devenu une tradition *(Figure 2)* : elle utilise le strict minimum de code et de matériel pour vérifier que le code peut être écrit, compilé et transféré dans un microcontrôleur. Cette tradition est reprise ici pour nous permettre d'explorer une gamme de langages de programmation établis et plus récents.

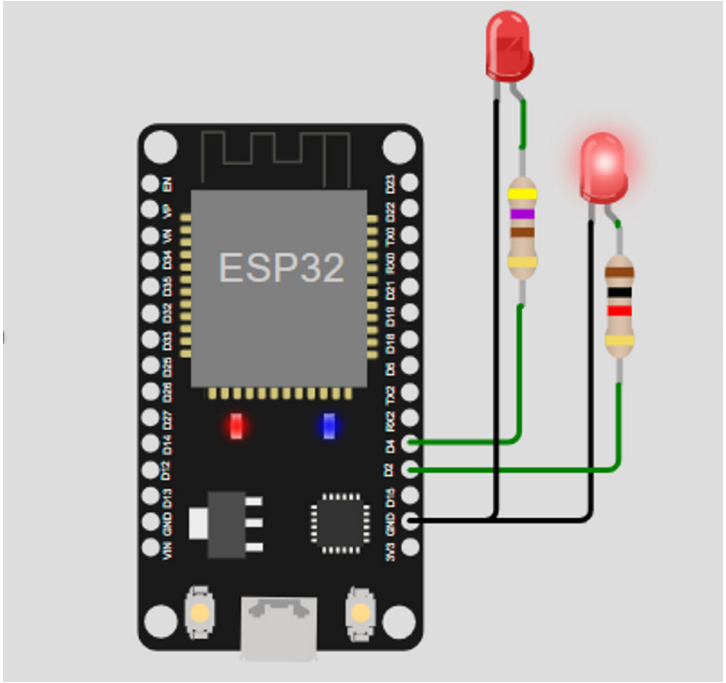


Figure 2 : *application LED clignotante utilisant un microcontrôleur. Les LED sont connectées entre la broche GPIO sélectionnée et la masse via une résistance (Source : www.Wokwi.com)*

Assembleur

Pour notre exemple de LED clignotante programmée en assembleur, nous utiliserons le microcontrôleur PIC16F18855 8 bits[1] de Microchip Technology, qui peut être trouvé sur la carte de développement MPLAB Xpress[2] (DM164141). Nous lui associerons le périphérique PORTA GPIO, et plus précisément la broche RA0, la première broche de ce port *(Figure 3)*.

Figure 3 : *la carte de développement Xpress de Microchip comporte un microcontrôleur 8 bits simple : le PIC16F18855 (Source : https://www.mouser.fr/)*

Le principe de fonctionnement est simple, quels que soient le langage de programmation et le microcontrôleur. La broche souhaitée (RA0) est configurée comme sortie à l'aide d'un registre de configuration (bit 0 de TRISA). Une fois configurée, la broche peut être activée ou désactivée en écrivant un « 1 » ou un « 0 » dans le bit associé d'un registre de contrôle (bit 0 de LATA).

Ces microcontrôleurs PIC16 comportent également un registre nommé PORTA, mais il doit uniquement être utilisé pour la lecture des entrées.

Bien que l'approche paraisse de prime abord assez simple, le cheminement pour arriver à ce stade s'avère un peu complexe en assembleur.

Le code se compose d'une section « principale » et d'une section « boucle ». La section principale configure les broches de PORTA en sorties, puis s'assure qu'elles sont toutes à l'état bas en effaçant les registres TRISA et LATA. L'instruction assembleur `clrf` est utilisée à cet effet : effacer le registre de fichier. Cependant, les broches de PORTA partagent des fonctionnalités numériques et analogiques, et les broches sont analogiques par défaut. Ainsi, le registre ANSELA doit être effacé en utilisant la même instruction.

En raison de la structure complexe de la mémoire de données du PIC16, le registre associé à ANSELA ne se trouve pas dans la même banque que les registres LATA et TRISA (Figure 4). Pour garantir l'écriture dans le bon registre, la pseudo-instruction `BANKSEL` indique aux outils d'assemblage d'insérer le code requis pour sélectionner la banque correcte. Par mesure de sécurité, une instruction `BANKSEL` doit être insérée avant d'accéder à tout registre. Cependant, cette exigence ajoute des instructions supplémentaires qui consomment de la mémoire et prennent du temps à s'exécuter,

TABLE 3-3: PIC16(L)F18855 MEMORY MAP BANK 0-7

	BANK 0		BANK 1		BANK 2		BANK 3		BANK 4	
000h	Core Registers (Table 3-2)	080h	Core Registers (Table 3-2)	100h	Core Registers (Table 3-2)	180h	Core Registers (Table 3-2)	200h	Core Registers (Table 3-2)	
00Bh		08Bh		10Bh		18Bh		20Bh		
00Ch	PORTA	08Ch	ADRESL	10Ch	ADCNT	18Ch	SSP1BUF	20Ch	TMR1L	
00Dh	PORTB	08Dh	ADRESH	10Dh	ADRPT	18Dh	SSP1ADD	20Dh	TMR1H	
00Eh	PORTC	08Eh	ADPREVL	10Eh	ADLTHL	18Eh	SSP1MSK	20Eh	T1CON	
00Fh	—	08Fh	ADPREVH	10Fh	ADLTHH	18Fh	SSP1STAT	20Fh	T1GCON	
010h	PORTE	090h	ADACCL	110h	ADUTHL	190h	SSP1CON1	210h	T1GATE	
011h	TRISA	091h	ADACCH	111h	ADUTHH	191h	SSP1CON2	211h	T1CLK	
012h	TRISB	092h	—	112h	ADSTPTL	192h	SSP1CON3	212h	TMR3L	
013h	TRISC	093h	ADCON0	113h	ADSTPTH	193h	—	213h	TMR3H	
014h	—	094h	ADCON1	114h	ADFLTRL	194h	—	214h	T3CON	
015h	—	095h	ADCON2	115h	ADFLTRH	195h	—	215h	T3GCON	
016h	LATA	096h	ADCON3	116h	ADERRL	196h	SSP2BUF	216h	T3GATE	
017h	LATB	097h	ADSTAT	117h	ADERRH	197h	SSP2ADD	217h	T3CLK	
018h	LATC	098h	ADCLK	118h	—	198h	SSP2MSK	218h	TMR5L	
019h	—	099h	ADACT	119h	RC1REG	199h	SSP2STAT	219h	TMR5H	
01Ah	—	09Ah	ADREF	11Ah	TX1REG	19Ah	SSP2CON1	21Ah	T5CON	
01Bh	—	09Bh	ADCAP	11Bh	SP1BRGL	19Bh	SSP2CON2	21Bh	T5GCON	
01Ch	TMR0L	09Ch	ADPRE	11Ch	SP1BRGH	19Ch	SSP2CON3	21Ch	T5GATE	
01Dh	TMR0H	09Dh	ADACQ	11Dh	RC1STA	19Dh	—	21Dh	T5CLK	
01Eh	T0CON0	09Eh	ADPCH	11Eh	TX1STA	19Eh	—	21Eh	CCPTMRS0	
01Fh	T0CON1	09Fh	—	11Fh	BAUD1CON	19Fh	—	21Fh	CCPTMRS1	
020h	General Purpose Register 96 Bytes	0A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h	General Purpose Register 80 Bytes	220h	General Purpose Register 80 Bytes	
07Fh		0EFh		16Fh		1EFh		26Fh		
		0F0h	Common RAM (Accesses 70h – 7Fh)	170h	Common RAM (Accesses 70h – 7Fh)	1F0h	Common RAM (Accesses 70h – 7Fh)	270h	Common RAM (Accesses 70h – 7Fh)	
		0FFh		17Fh		1FFh		27Fh		

Legend: = Unimplemented data memory locations, read as '0'.

Figure 4 : *la mémoire de données n'est pas adressable linéairement sur un PIC16, ce qui nécessite de sélectionner la bonne banque avant d'accéder au registre souhaité (Source : https://www.mouser.fr/)*

ce qui constitue notre premier défi en utilisant l'assembleur. En tant que programmeur, vous devrez comprendre l'architecture de la mémoire du microcontrôleur : vous pourrez ainsi identifier la possibilité d'éviter ces instructions supplémentaires afin d'économiser de la mémoire ou d'accélérer l'exécution de votre code.

La section en boucle est l'endroit où la LED est contrôlée *(Figure 5)*. La valeur 0x01 est écrite dans le registre de travail du PIC16, WREG (`movlw`), qui est écrit dans le registre LATA pour allumer la LED (`movwf`). L'écriture de 0x00 dans WREG puis son déplacement dans LATA désactivent à nouveau la broche RA0.

Une section non-code précède le code. L'allocation de mémoire PSECT définit où les lignes suivantes doivent être allouées en mémoire. Il peut s'agir d'un vecteur de réinitialisation, de la SRAM, de l'EEPROM ou de la mémoire de programme. Les définitions CONFIG définissent les mots de configuration du PIC16 qui configurent la fonctionnalité post-mise sous tension, comme le choix de l'oscillateur, et dans ce cas désactivent l'horloge de surveillance. La directive PROCESSOR est utilisée pour spécifier le microcontrôleur exact pour lequel le code est écrit. Enfin, les définitions de tous les registres du microcontrôleur sont nécessaires. Elles sont fournies par la ligne : `#include <xc.inc>`.

Au fur et à mesure que le code est assemblé, les outils utilisent finalement un fichier d'inclusion nommé pic16f18855.inc. L'outil d'assemblage en une seule étape pic-as (inclus avec la chaîne d'outils XC8[3]) gère les adresses de registre et attribue des variables et du code de programme à la mémoire disponible.

Il convient également de noter que nous n'avons inclus aucun code pour retarder l'exécution, de sorte que la LED clignotera exceptionnellement rapidement ! Nous pourrions implémenter une boucle active en assembleur ou utiliser un minuteur matériel, mais de tels ajouts dépassent le cadre de cet exemple.

Résumé : programmation en assembleur

La programmation en assembleur est idéale pour le microcontrôleur, mais pas nécessairement pour le programmeur. À ce niveau, nous parlons plus ou moins le langage du dispositif que nous programmons, en expliquant instruction par instruction ce qu'il faut faire. Une telle approche peut s'avérer très utile pour écrire du code déterministe avec des exigences de minutage strictes sur des microcontrôleurs plus simples.

Cependant, nous devons également comprendre intimement les complexités de l'architecture du dispositif, comme la mémoire de données en banque du PIC16, car nous devons nous assurer que la bonne banque est sélectionnée avant de modifier les valeurs des registres.

La portabilité constitue un autre problème. Ce code n'a pas été écrit pour un PIC16, mais pour un PIC16F18855. Le même code pourrait être utilisé sur certains dispositifs similaires, mais nous devrons découvrir et prendre en compte les différences potentielles. Il est possible d'écrire du code assembleur réutilisable dans de nombreux projets, en particulier si le code ne repose pas sur des registres périphériques. Par exemple, vous pouvez créer un algorithme de contrôle PID réutilisable. Bien entendu, si vous passez à un dispositif ARM ou RISC-V, vous devrez réécrire ce code.

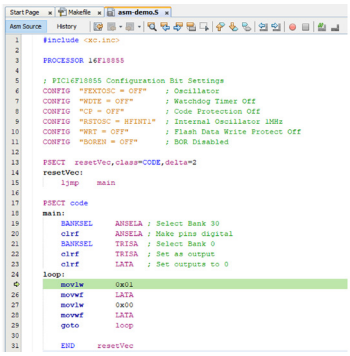


Figure 5 : *code assembleur PIC16 pour implémenter une application LED clignotante (Source : capture d'écran créée par l'auteur)*

Alors, quel rôle joue l'assembleur aujourd'hui ? La plupart des organisations souhaitent un code réutilisable qui n'est pas lié à un ou à un nombre limité de microcontrôleurs. De moins en moins de développeurs créent des applications uniquement en assembleur, à moins qu'elles ne soient très simples. Plus rapide, la programmation en C est également réutilisable, et le code résultant n'est pas très différent de celui que vous écrivez vous-même. L'assembleur est principalement utilisé pour créer du code optimisé dans des langages de haut niveau, comme C ou C++, afin d'accélérer un algorithme ou de réduire ses besoins en mémoire. Il est donc utile de comprendre les instructions sous-jacentes utilisées par les microcontrôleurs, mais de nos jours, l'écriture d'une application entière en assembleur est rare.

C

Le langage de programmation C est un élément essentiel du développement de systèmes embarqués. Langage de niveau supérieur à l'assembleur, C accroît la liberté du programmeur, qui peut alors se concentrer sur l'écriture de son application sans se soucier du matériel sous-jacent sur lequel le code s'exécutera. Cependant, il peut toujours accéder facilement et directement au matériel, ce qui lui permet d'écrire du code de manière efficace et compacte. Si nécessaire, les outils de compilation prennent en charge l'assembleur en ligne, le code assembleur écrit dans le code source C.

Notre exemple suivant s'en tient au même microcontrôleur PIC16F18855, mais utilise l'EDI basé sur le cloud Microchip Xpress[4]. Nous avons ainsi un accès facile à un exemple de code et à un compilateur sans avoir à installer les outils localement.

Le code est devenu plus simple, en partie parce que nous écrivons en C et en partie parce que l'EDI gère une partie de la configuration du microcontrôleur et du processus de construction pour nous. Comme c'est la convention en C, le point de départ du code de l'utilisateur se trouve dans `main`(). Comme précédemment, nous avons configuré les registres nécessaires pour faire de la broche PA0 une sortie numérique *(Figure 6)*. Ensuite, dans une boucle `while`(1) sans fin, la broche RA0 est activée et désactivée. Au lieu de se préoccuper de la manière d'écrire des valeurs dans les registres à l'aide du code assembleur, nous attribuons simplement la valeur souhaitée au registre, par exemple `LATA = 0x01`. Le compilateur C sélectionne les instructions appropriées.

Par rapport à l'assembleur, nous pouvons voir qu'il n'y a pas de pseudo-commandes BANKSEL. C'est parce que ce langage de haut niveau gère également pour nous la complexité du matériel sous-jacent. Si un changement de banque est nécessaire pour cette architecture, nous en insérerons un. Si nous compilons alors ce code pour un autre microcontrôleur où les registres sont attribués à différentes banques, il fonctionnera toujours ! Nous disposons désormais d'un code portable, un code qui peut être écrit une fois et utilisé sur une variété de microcontrôleurs.

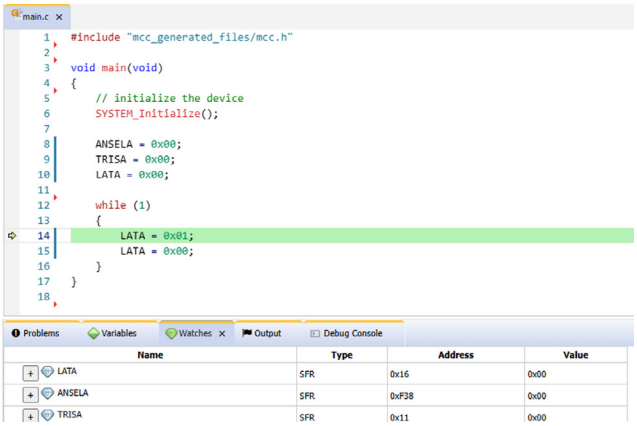


Figure 6 : *l'application LED clignotante pour le PIC16, cette fois écrite en C (Source : capture d'écran créée par l'auteur)*

Nous voyons également une fonction appelée `SYSTEM_Initialiser`(). Il s'agit de l'une des nombreuses fonctions de bibliothèque fournies par Microchip qui gèrent la configuration interne de base (oscillateur, minuteries, broches) commune à la plupart des applications. Étant donné que des éléments comme l'initialisation sont communs à tous les projets, nous pouvons écrire des bibliothèques réutilisables comme celle-ci et les partager avec des collègues et d'autres développeurs.

Les mots de configuration ne sont pas définis dans le code source, bien qu'ils puissent l'être. Au lieu de cela, les paramètres souhaités sont enregistrés dans le fichier projet de l'EDI pour ce projet. Afin de rendre ces paramètres visibles pour les autres, il est plus logique qu'ils soient inclus et documentés dans le code source. Après tout, le fichier de projet d'un EDI peut ne pas être lisible à l'avenir s'il stocke des données dans un format propriétaire. Cela facilite également le suivi des modifications lors du stockage du code dans un référentiel comme git.

Le code est construit à l'aide de la chaîne d'outils XC8 de Microchip, avec le processus de construction enregistré dans l'EDI et le linker reliant le code aux registres et à la carte mémoire du processeur choisi.

Résumé : programmation en C

C trouve un juste milieu entre le développement de code portable et lisible et l'offre directe de méthodes de contrôle du matériel du microcontrôleur. Certains programmeurs le qualifient de langage assembleur portable. Il n'est pas étonnant qu'il constitue le langage de programmation privilégié des développeurs de systèmes embarqués depuis près de quatre décennies.

Actuellement, il n'existe aucune raison pratique d'abandonner le C. Largement compris, il fournit tout le nécessaire aux développeurs. C'est toujours le langage utilisé pour programmer le noyau Linux, les bases de données et même les logiciels de montage vidéo, donc le langage et le développement de la chaîne d'outils se poursuivront dans un avenir prévisible.

Parmi les inconvénients, il est facile d'écrire du code C qui compile, mais qui échoue à fonctionner ou à fonctionner comme prévu, et la liberté de manipuler la mémoire partout peut entraîner des problèmes. En conséquence, des langages « mémoire sécurisée » comme Rust ont le potentiel de repousser C de sa position de premier langage de programmation embarqué, mais nous y reviendrons plus tard.

C++

Le C++ était initialement considéré comme un langage de programmation pour les systèmes informatiques complexes, mais grâce à des plateformes comme Arduino, la carrière d'une génération de développeurs a commencé avec une expérience dans ce langage. C'est pourquoi, pour nous guider, nous avons sélectionné un Arduino DUE. Cette carte alimentée par ARM Cortex-M3 dispose de nombreux périphériques intégrés, de broches GPIO, de 512 Ko de mémoire flash et jusqu'à 100 Ko de SRAM *(Figure 7)*.

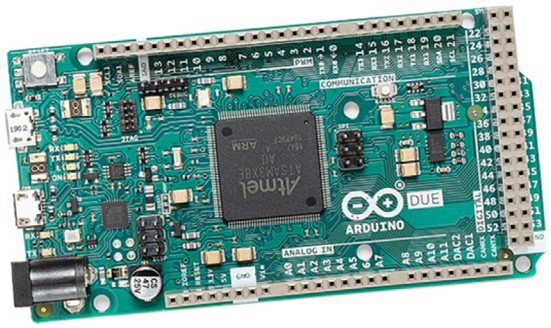


Figure 7 : *l'Arduino DUE fait partie des nombreuses cartes de développement qui ont initié une génération de programmeurs au C++ embarqué (Source : https://www.mouser.fr/)*

Comme son nom l'indique, C++ s'appuie sur C en ajoutant des fonctionnalités supplémentaires. C'est un langage procédural, ce qui signifie que les données et les procédures sont conservées séparément. C++ utilise le paradigme de programmation orientée objet (POO), où les données et les fonctions qui les manipulent (méthodes) sont combinées dans des objets (classes).

Par exemple, si vous développez du code pour implémenter des feux de circulation en C, vous créerez probablement une série de fonctions (par exemple, TrafficLightxxx) qui allument et éteignent les LED du feu de circulation à l'aide d'une machine à états. Si vous décidez d'ajouter un deuxième ensemble de feux de circulation, vous devez copier les fonctions (par exemple, TrafficLightTwoxxx), les coller dans votre fichier C et modifier toutes les lignes de code qui manipulent les LED du deuxième ensemble de feux de circulation. Que se passe-t-il lorsque vous devez ajouter un troisième ensemble ? C'est là que la POO intervient.

En C++, vous créeriez un objet feu de circulation (appelé classe) avec sa propre machine à états, ses variables et ses méthodes *(Figure 8)*. Au départ, cet objet définit uniquement le fonctionnement du feu de circulation et fournit des variables pour enregistrer son état actuel : il ne consomme pas encore de ressources mémoire. Ce n'est que lorsqu'une classe de feu de circulation est instanciée que la mémoire dont elle a besoin est allouée. À ce stade, vous pouvez également savoir quelles LED (c'est-à-dire quelles broches GPIO) sont attribuées à cet objet feu de circulation. Ensuite, lorsque vous avez besoin d'un deuxième, d'un troisième ou d'un quatrième feu, vous suivez le même processus d'instanciation. Chaque objet feu de circulation sait quelles LED lui appartiennent et grâce à la machine d'état interne, dans quel état se trouve chaque feu de circulation.

Nous pouvons également étendre notre feu de circulation pour ajouter les fonctionnalités d'un passage piéton en utilisant l'héritage *(Figure 9)*. Une nouvelle classe passage piéton est définie à partir de la classe feu de circulation et ajoute les feux rouges « Ne traversez pas » et verts « Traversez », ainsi que le bouton-poussoir pour demander la traversée. Dans les deux cas, tout ce qui est associé à la manipulation du matériel du microcontrôleur, comme les GPIO ou les temps d'attente, peut être abstrait, ce qui rend le code portable et réutilisable.

Il convient de noter qu'une approche orientée objet du développement de code peut être implémentée en C, mais C++ facilite la tâche, car la POO est native du langage.

Pour revenir à la tâche à accomplir, examinons la variante C++ d'une LED clignotante sur l'Arduino DUE. Comme on peut le constater, elle ne diffère pas beaucoup de la variante C. Au lieu d'assigner des valeurs à un registre, une certaine abstraction est mise en œuvre en utilisant la fonction `digitalWrite`(). Ce code fait clignoter la LED déjà implémentée sur le circuit imprimé connecté à la broche numérique 13, une broche définie par l'environnement comme `LED _ BUILTIN`. La configuration initiale de cette broche est gérée par la fonction appelée `pinMode`().

Si cet exemple ressemble beaucoup au C, c'est parce qu'il s'agit bien de C *(Figure 10)*. Les fonctions de manipulation des broches numériques dans l'environnement Arduino sont écrites en C et se trouvent dans le fichier « wiring_digital.c » sur votre PC.

Pour vous faire une idée précise du C++ sur Arduino, vous devez examiner la fonctionnalité `Serial` qui envoie et reçoit des données

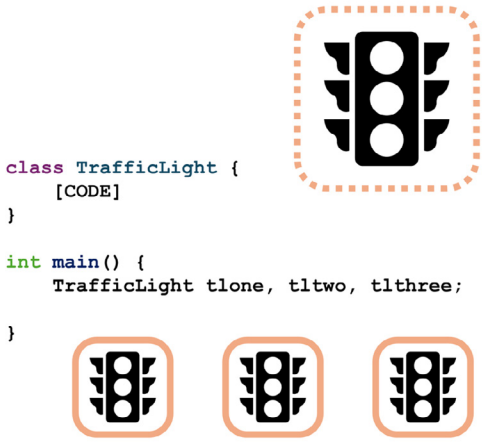


Figure 8 : *en utilisant la POO, un objet feu de circulation peut être défini comme une classe, puis plusieurs feux de circulation, indépendants les uns des autres, mais utilisant le même code, peuvent être instanciés (Source : réalisé dans PowerPoint)*

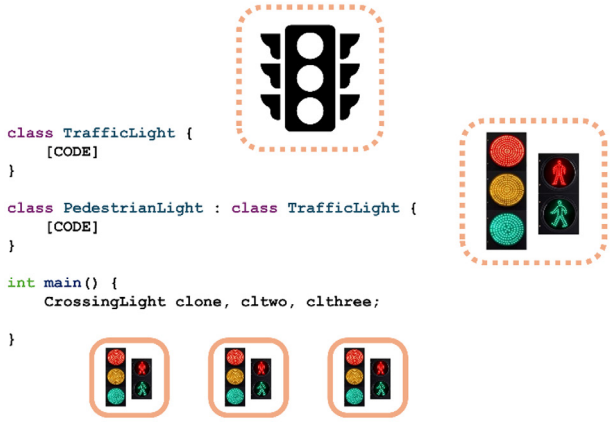


Figure 9 : *grâce à la POO, la classe TrafficLight peut être étendue pour prendre en charge la fonctionnalité de passage piéton (Source : réalisé dans PowerPoint)*

Blink.ino

```
1  void setup() {
2    // Initialize digital pin LED_BUILTIN as an output.
3    pinMode(LED_BUILTIN, OUTPUT);
4  }
5
6  // The loop function runs over and over again forever
7  void loop() {
8    digitalWrite(LED_BUILTIN, HIGH);  // Turn the LED on
9    delay(500);                       // Wait for a 500 ms
10   digitalWrite(LED_BUILTIN, LOW);   // turn the LED off
11   delay(500);                       // wait for a 500 ms
12 }
13
```

Figure 10 : *le code de l'application LED clignotante utilisant l'environnement Arduino n'utilise en réalité pas de C++ (Source : capture d'écran créée par l'auteur)*

via l'interface USB du DUE ou les interfaces UART matérielles *(Figure 11)*. L'implémentation peut être trouvée dans le fichier « HardwareSerial.cpp » et « HardwareSerial.h » sur votre PC. L'interface série est implé
mentée comme une classe C++ qui hérite de certaines fonctionnalités d'une autre classe, `Stream`, qui hérite d'une troisième classe, `Print`. Print est l'endroit où sont implémentées les fonctionnalités `print`() et println(). Nous pouvons ainsi implémenter facilement un exemple classique « Hello World ! » qui s'imprime sur un terminal. Cependant, nous pouvons également hériter `Stream` et `Print` et écrire une classe qui utilise les interfaces SPI ou I²C à la place.

Le processus de construction est similaire à la construction de code C, mais est masqué par l'environnement de développement Arduino (à moins que vous n'ayez activé la construction détaillée), masquant une grande partie de la complexité d'initialisation et des paramètres des bits de configuration.

```
1  void setup() {
2    Serial.begin(9600);
3  }
4  void loop() {
5    Serial.println("Hello World!");
6
7    while (true) {
8      continue;
9    }
10 }
11
```

Figure 11 : *la classe Serial qui permet d'afficher du texte via le moniteur série montre le C++ en action sur un Arduino (Source : capture d'écran créée par l'auteur)*

Résumé : programmation en C++

Les développeurs expérimentés souligneront que les runtimes C++ sont plus volumineux qu'un runtime C, ce qui signifie qu'il y a un peu plus de surcharge avec C++. Cependant, de nombreux microcontrôleurs actuels disposent de quantités exceptionnellement généreuses de mémoire flash et de SRAM. La surcharge devient donc moins préoccupante. Certaines bibliothèques d'interface utilisateur graphique (GUI) sont écrites en C++, car l'approche POO facilite le suivi de nombreux éléments visuels et de leurs différentes propriétés (couleur, taille, texte, police, état enfoncé). Ceci est utile lors de la création de bibliothèques pour des interfaces graphiques telles que celles fournies par The Qt Company[7].

Essentiellement, C++ permet aux programmeurs de conserver ensemble les données et les méthodes qui les manipulent, autorisant ainsi la création d'objets réutilisables. En conséquence, le développement peut être plus complexe et les binaires plus volumineux qu'avec C. Ces inconvénients demeurent un faible prix à payer par rapport aux avantages. Il est également possible de rester proche du matériel, tout comme avec C.

Ada

Le langage Ada, qui tire son nom d'Ada Lovelace, a été développé pour remplacer les nombreux langages différents utilisés par le ministère de la Défense des États-Unis à la fin des années 1970. Conçu pour les systèmes intégrés en temps réel, il est utilisé dans des applications critiques pour la sécurité et la mission telles que l'aérospatiale, les chemins de fer et la banque, pour n'en citer que quelques-unes. Il reste en développement continu, les ajouts les plus récents au langage étant publiés dans la norme ISO 8652:2023.

Les avantages incluent un typage fort, qui permet de déterminer plus clairement au moment de la compilation les valeurs qui peuvent être attribuées à une variable, ce qui contribue à rendre le code écrit en Ada intrinsèquement plus sûr. Il existe

Figure 12 : *le Raspberry Pi est une bonne plateforme pour écrire et compiler du code en Ada afin de créer une application LED clignotante (Source : https://www.mouser.fr/)*

également des vérifications au moment de la compilation et des vérifications au moment de l'exécution. Celles-ci entraînent une augmentation du code et du temps d'exécution, mais les avantages sont considérés comme supérieurs aux coûts. Il s'intègre également bien avec d'autres langages tels que C et C++, ce qui signifie que le langage le plus approprié peut être utilisé pour coder différentes parties d'une application. Bien que le langage soit assez verbeux, ce qui rend l'écriture plus fastidieuse, les défenseurs affirment que le code n'en est que plus lisible pour ceux qui le maintiennent après le déploiement.

Parmi les inconvénients, le langage Ada est souvent considéré comme difficile à apprendre.

L'exemple de LED clignotante ici fonctionne sur le Raspberry Pi[9] *(Figure 12)*. Il diffère considérablement de nos exemples précédents, bénéficiant du système d'exploitation et de ses nombreuses bibliothèques.

L'exemple ici utilise l'interface C dans Ada pour appeler l'utilitaire `gpioset`[10] en tant que commande shell via `libc` *(Figure 13)*. Le code source est stocké dans un fichier adb (corps Ada). La clause de contexte précise que le package `Interface.C` sera utilisé. L'équivalent d'une fonction `main`() en C est la clause de procédure portant le nom `Ada _ Blink _ Demo`. Ce nom doit correspondre au nom du fichier de code source. La boucle infinie appelle la procédure `GPIO _ Set` pour activer et désactiver la broche 17. L'instruction de temporisation fait également partie du langage Ada.

Pour une véritable implémentation bare metal, des exemples sont disponibles pour les microcontrôleurs STM32. Ils sont écrits à l'aide d'Alire, un référentiel de bibliothèques Ada prenant en charge divers microcontrôleurs ARM et RISC-V, capteurs et algorithmes. Alire simplifie la mise en œuvre de fonctionnalités communes tout en gardant une trace de la version d'une bibliothèque utilisée.

```
1   with Interfaces.C;
2
3   procedure Ada_Blink_Demo is
4
5      procedure GPIO_Set (Pin : Natural; On : Boolean) is
6         package C renames Interfaces.C;
7         use type C.int;
8
9         function System (command : C.char_array) return C.int
10          with Import, Convention => C;
11        --  Import libc `system` function to execute a shell command
12
13        Command : aliased constant C.char_array :=
14          C.To_C ("gpioset 0" & Pin'Img & "=" & (if On then "1" else "0"));
15        --  Create the command string with a format: gpioset 0 <pin>=<1|0>
16     begin
17        if System (Command) /= 0 then
18           raise Program_Error with "gpioset failed";
19        end if;
20     end GPIO_Set;
21  begin
22     loop
23        GPIO_Set (17, True);
24        delay 0.5;
25        GPIO_Set (17, False);
26        delay 0.5;
27     end loop;
28  end Ada_Blink_Demo;
```

Figure 13 : *dans cet exemple pour le Raspberry Pi, Ada utilise l'interface C pour appeler gpiopiset afin de contrôler notre LED (Source : capture d'écran créée par l'auteur)*

Résumé : programmation en Ada

Bien qu'Ada puisse être assimilé de prime abord à un langage de programmation marginal, il est utilisé dans de nombreuses applications critiques pour la sécurité et les missions, telles que l'exploration spatiale et les aéronefs. De nombreux problèmes de codage potentiels peuvent être découverts au moment de la compilation, un avantage par rapport à l'écriture en C, où les bogues sont généralement détectés après que le code a été flashé sur le microcontrôleur. Le référentiel de code Alire aide également les développeurs en partageant du code et des algorithmes couramment utilisés. Malheureusement, la courbe d'apprentissage est considérée comme assez abrupte et Ada ne s'est pas imposé auprès des développeurs de systèmes intégrés en dehors des applications critiques pour la sécurité et la mission.

MicroPython

Il existe de nombreux langages de programmation dans le monde, mais peu d'entre eux finissent par être utilisés pour programmer des microcontrôleurs. La petite implémentation de Python MicroPython constitue une exception et a été adaptée à des appareils tels que le STM32, l'ESP32, le SAMD, le Raspberry Pi Pico et le nRF52. La carte MicroPython pyboard d'AdaFruit *(Figure 14)* représente un bon point de départ. Exécutant la version 3.4 de Python, elle utilise un STM32F405RG avec un ARM Cortex-M4 fonctionnant jusqu'à 168 MHz.

Python est un langage très convivial pour les débutants qui prend en charge la programmation orientée objet et procédurale. Parce qu'il est largement utilisé et multiplateforme, il permet à une personne expérimentée en Python de créer plus facilement du code pour un microcontrôleur. Cependant, il s'agit également

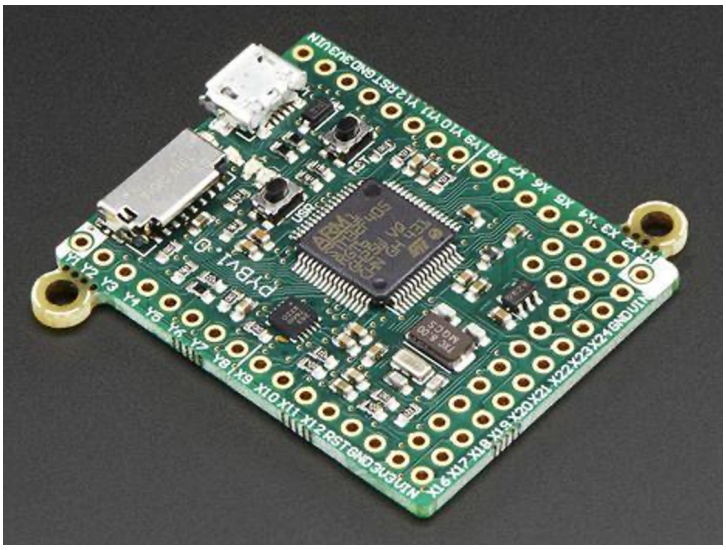


Figure 14 : *dans cet exemple pour le Raspberry Pi, Ada utilise l'interface C pour appeler gpiopiset afin de contrôler notre LED (Source : https://www.mouser.fr/)*

d'un langage interprété, exécuté ligne par ligne. Comparé aux langages compilés, il laisse peu de place aux développeurs pour optimiser le code des applications en temps réel.

MicroPython et les cartes prises en charge sont préparés pour que des périphériques et des capacités spécifiques fonctionnent immédiatement. Cela inclut l'interface série pour accéder à la carte via USB, quatre LED connectées à des GPIO et des PWM, des broches CAN et CNA, une horloge en temps réel (RTC) et une interface pour carte SD. Une partie de la mémoire flash du microcontrôleur est également définie pour être utilisée comme clé USB qui apparaît lorsque la carte est connectée à un PC.

Tout comme les premiers ordinateurs personnels des années 1980 qui prenaient en charge BASIC, en connectant la pyboard à une interface USB et en démarrant un programme de terminal comme PuTTy, vous êtes accueilli par une interface de ligne de commande qui donne accès à une invite REPL (Read Evaluate Print Loop). Vous pouvez aussi écrire un script Python et l'enregistrer sur la clé USB. Après avoir réinitialisé la carte, le script sera exécuté.

Pour contrôler les LED intégrées, il suffit d'importer le support LED spécifique à la carte à partir du module `pyb` *(Figure 15)*. Ensuite, en utilisant les méthodes `on`() et `off`(), nous pouvons faire clignoter notre LED, même si ce n'est qu'une seule fois !

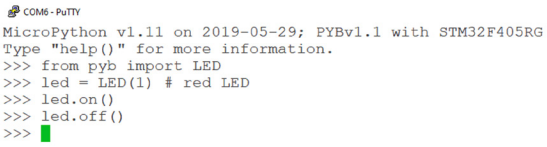


Figure 15 : *REPL permet au matériel du microcontrôleur d'être contrôlé depuis une ligne de commande dans MicroPython (Source : capture d'écran créée par l'auteur)*

Si nous voulons que notre véritable LED clignote, nous pouvons enregistrer un script appelé `main.py` sur la clé USB. Cette fois, nous ajoutons le module `time` et la méthode LED `toggle`(), ce qui nous permet d'allumer et d'éteindre la LED une fois par seconde *(Figure 16)*.

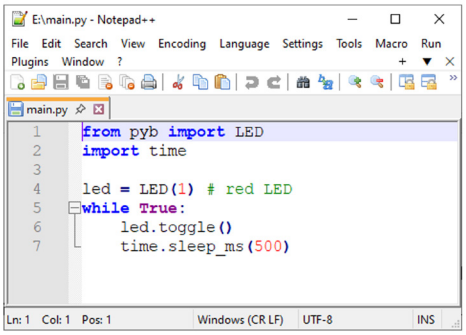


Figure 16 : *un script MicroPython qui implémente le code LED clignotante (Source : capture d'écran créée par l'auteur)*

L'accès direct aux registres d'un microcontrôleur est possible, mais la lisibilité du code est perdue, et cela nécessite une compréhension avancée des registres et une attention particulière à ne rien perturber qui soit déjà contrôlé par l'environnement MicroPython[15].

Résumé

MicroPython offre une autre couche d'abstraction du matériel du microcontrôleur, avec un contrôle implémenté dans divers modules. Cependant, si un périphérique que vous souhaitez utiliser n'est pas implémenté, vous devrez reconstruire l'ensemble de l'environnement avec le module que vous avez écrit pour le prendre en charge.

Si vous êtes habitué à la programmation en Python et que vous souhaitez mettre en place un système embarqué, c'est un excellent langage. De plus, il est tout à fait adapté à l'enseignement de la programmation, car les entrées sont évaluées immédiatement, ce qui évite les difficultés liées à la compilation de code C/C++ qui, une fois flashé sur le microcontrôleur, ne fonctionne pas réellement.

Il existe cependant des limites. MicroPython n'est pas une implémentation complète de Python[16], et vous n'aurez pas à importer de nouveaux modules sur un microcontrôleur à partir d'Internet. De plus, les microcontrôleurs ont une quantité de mémoire limitée, de sorte qu'un code complexe et des classes héritées et étendues peuvent ne pas être possibles.

Rust

Rust est le langage le plus récent parmi ceux abordés ici, avec la première version stable mise à disposition en 2015. Dix ans plus tard, il a atteint le top 20 de l'indice TIOBE[18] qui suit la popularité du langage de programmation. Son accent mis sur les performances et la sécurité, en particulier la sécurité mémoire, le rend particulièrement attractif pour les développeurs de systèmes intégrés. En conséquence, la prise en charge des microcontrôleurs tels que les appareils STM32, nRF52 et RISC-V a commencé en 2017. Pour garder les choses relativement simples, cet exemple active une LED connectée à la broche 17 d'un Raspberry Pi 3.

Contrairement au C, Rust est un langage fortement typé. Cela signifie que, contrairement au C, Rust n'autorise pas la conversion implicite de type, comme l'affectation d'une variable de type `char` à une variable définie comme un `int`. À la place, une variable définie comme `i8` (équivalent du char en Rust) doit être déclarée temporairement comme `i32` (équivalent du int en Rust) dans l'affectation *(Figure 17)*. Cette approche garantit que les erreurs qui peuvent survenir en C ne peuvent pas se produire en Rust. De plus, le compilateur

```c
// Example in C

char a = 10;
int b = a + 5;
```

```rust
// Example in Rust

let a: i8 = 10;
let b: i32 = a as i32 + 5;
```

Figure 17 : *étant donné que Rust est fortement typé, la promotion d'entiers doit être effectuée explicitement, contrairement à la promotion implicite qui peut être utilisée en C (Source : capture d'écran créée par l'auteur)*

effectue de nombreuses garanties de sécurité qui évitent les comportements indéfinis qui peuvent être difficiles à déboguer.

Cependant, tout comme C, vous pouvez simplement coder en Rust et générer du code bare metal sans inclure de bibliothèques standard. L'environnement de construction est également étonnamment verbeux, rendant les messages d'erreur exploitables et fournissant souvent des références à la documentation. Une interface de fonction externe (FFI, Foreign Function Interface) est également proposée, ce qui permet d'écrire du code Rust capable de réutiliser des bibliothèques C existantes.

L'avantage le plus significatif de Rust par rapport à C/C++ est peut-être l'outillage. Outre le compilateur (rustc), le système de compilation et gestionnaire de paquets cargo de Rust fait entrer l'expérience de développement dans le XXIe siècle par rapport à C.

Après avoir installé Rust[19], pour créer un nouveau projet avec un environnement de compilation, il suffit d'exécuter :

```
cargo new blinking_led
```

Le système crée un nouveau dossier appelé `blinking_led`, avec une arborescence de dossiers et un fichier de code source initial appelé `src/main.rs`. Il existe également un fichier appelé `Cargo.toml`, un fichier manifeste Rust *(Figure 18)*. Celui-ci sert, entre autres, à conserver les métadonnées du projet, les dépendances et la configuration de compilation. C'est un avantage considérable par rapport à la programmation en C, où la version du compilateur et celles des bibliothèques fournies par les fabricants de microcontrôleurs doivent être suivies manuellement.

```
stuart@raspberrypi3b: ~/sw-dev/mouser/rust-blinky/led-blinking
  GNU nano 7.2                                Cargo.toml
[package]
name = "led-blinking"
version = "0.1.0"
edition = "2024"

[dependencies]
rppal = "0.14.1"
```

Figure 18 : *le fichier manifeste Rust « Cargo.toml » définit les dépendances du projet et définit la configuration de construction (Source : capture d'écran créée par l'auteur)*

Pour notre projet de LED clignotante, nous utiliserons rppal[20] (Raspberry Pi Peripheral Access Library) pour accéder aux broches GPIO. Il s'agit de l'une des nombreuses bibliothèques différentes qui, dans la terminologie Rust, sont appelées « crates ». La broche est en réalité manipulée dans un fichier appelé `bcm.rs`[21].

Le code rappelle celui du C/C++ (Figure 19). Il commence par importer les bibliothèques standard (gestion des erreurs et temporisations via les threads), suivies de GPIO et système à partir de la crate rppal. Le point d'entrée du code est également `main`(), comme en C. `Box<dyn Error>` renvoie une erreur issue de tout point du code qui échoue à s'exécuter lorsqu'un ? est rencontré, comme dans `DeviceInfo::new()?.model()`, une méthode qui tente d'afficher le nom du modèle de Raspberry Pi à l'écran. Une instance de la broche GPIO souhaitée est ensuite créée et configurée comme sortie, puis dans la `boucle`, la broche est mise à l'état haut et bas avec un délai de 500 ms.

Pour compiler le projet, il suffit d'exécuter :

```
cargo build
```

Avant la compilation du projet, cette commande télécharge la version spécifiée des crates listées dans le manifeste. L'exécutable se trouve ensuite dans `target/debug/blinking_led`,, qui peut être débogué à l'aide du débogueur GNU gdb, comme en C/C++.

```
main.rs
1  use std::error::Error;
2  use std::thread;
3  use std::time::Duration;
4
5  use rppal::gpio::Gpio;
6  use rppal::system::DeviceInfo;
7
8  // GPIO uses BCM pin numbering. BCM GPIO 17 used.
9  const GPIO_LED: u8 = 17;
10
11 fn main() -> Result<(), Box<dyn Error>> {
12     println!("Blinking an LED on a {}.", DeviceInfo::new()?.model());
13
14     let gpio = Gpio::new().unwrap();
15     let mut pin = gpio.get(GPIO_LED).unwrap().into_output();
16
17     loop {
18         // Blink the LED.
19         pin.set_high();
20         thread::sleep(Duration::from_millis(500));
21
22         pin.set_low();
23         thread::sleep(Duration::from_millis(500));
24     }
25 }
```

Figure 19 : *le code LED clignotante pour Raspberry Pi, écrit en Rust (Source : capture d'écran créée par l'auteur)*

Résumé

Rust est plus qu'un langage de programmation embarqué compétent. C'est un langage pour l'ère moderne. Langage à bas niveau à l'instar de C, il apporte toutefois un gestionnaire de paquets pour importer des bibliothèques comme Alire d'Ada et Python. Cela aide le développeur à garder une trace des versions du compilateur et des crates qui fonctionnaient lors de la création de son application. Par rapport au C, le compilateur est également utilement verbeux plutôt que cryptique et offre une sécurité de mémoire renforcée au moment de la compilation.

Les seuls motifs actuels de plaintes sont que le langage peut être difficile à apprendre, l'écosystème est limité et il peut être difficile de trouver une expertise en raison du faible nombre de développeurs utilisant Rust pour des applications intégrées. Bien que ce langage en soit encore à ses débuts comparé au C, des fournisseurs comme Espressif se sont déjà engagés à le prendre en charge avec leurs dispositifs. Le noyau Linux a également approuvé Rust aux côtés de C pour son développement. Des outils comme Ferrocene[22], un compilateur Rust certifié pour les systèmes critiques pour la sécurité et la mission, sont étudiés par les développeurs automobiles. Leur généralisation n'est donc probablement qu'une question de temps.

Des langages divers pour des besoins divers

Les programmeurs de systèmes embarqués ont grandi avec une mémoire limitée, ce qui les a poussés à adopter des approches créatives pour mettre en œuvre des applications en temps réel strict. En comparant le projet de LED clignotante dans les langages les plus utilisés pour programmer des microcontrôleurs, nous avons pu voir les défis et les limites, et nous faire une idée de l'outillage disponible.

L'assembleur nous permet de contrôler chaque aspect du code d'une application avec une précision cyclique si nécessaire. Toutefois, comme les développeurs d'UNIX l'ont découvert, le portage de code écrit en assembleur est un travail difficile, c'est pourquoi il a été réécrit en C[23].

Depuis lors, C, C++ et Ada ont tous trouvé les espaces d'application qui leur conviennent le mieux, reflétant les avantages et les limites de chaque langage. MicroPython est un excellent petit langage pour ceux qui ont de l'expérience en Python, permettant aux développeurs de passer facilement du développement sur PC au développement sur microcontrôleur. Cependant, le fait qu'il s'agisse d'un langage interprété signifie que l'environnement entier doit être reconstruit pour ajouter des modules supplémentaires.

Rust est la surprise du lot, suscitant rapidement l'intérêt de la communauté des systèmes embarqués. Comme Ada, il allie sécurité et prise en charge du code hérité écrit en C. Cependant, son outillage moderne constitue potentiellement son aspect le plus intéressant: il permet de créer des projets au sein desquels les dépendances et la version du compilateur utilisées sont clairement identifiées.

La décennie à venir s'annonce passionnante. La connaissance de l'assembleur continuera d'être utile lors de l'optimisation d'algorithmes spécifiques. Cependant, pour ceux qui étudient actuellement ou qui débutent leur carrière dans les systèmes embarqués, il est conseillé de développer une compréhension de Rust en parallèle du C/C++.

Références

[1]https://eu.mouser.com/new/microchip/microchip-pic16f188xx-microcontrollers/

[2]https://eu.mouser.com/ProductDetail/Microchip-Technology/DM164141?qs=5aG0NVq1C4zIZJRIkiTc0g%3D%3D

[3]https://www.microchip.com/en-us/tools-resources/develop/mplab-xc-compilers/xc8

[4]https://www.microchip.com/en-us/tools-resources/develop/mplab-xpress

[5]https://www.microchip.com/en-us/tools-resources/develop/mplab-xc-compilers/xc8

[6]https://eu.mouser.com/new/arduino/arduino-due/

[7]https://www.qt.io/

[8]https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html

[9]https://eu.mouser.com/new/raspberry-pi/raspberry-pi-5-sbc/

[10]https://manpages.debian.org/experimental/gpiod/gpioset.1.en.html

[11]https://github.com/GNAT-Academic-Program/stm32_blinky_demo

[12]https://alire.ada.dev/

[13]https://eu.mouser.com/ProductDetail/Adafruit/2390?qs=GURawfaeGuDCdUqnWpea7Q%3D%3D

[14]https://www.putty.org/

[15]https://docs.micropython.org/en/latest/esp32/tutorial/peripheral_access.html

[16]https://docs.micropython.org/en/latest/genrst/index.html

[17]https://docs.micropython.org/en/latest/reference/constrained.html

[18]https://www.tiobe.com/tiobe-index/

[19]https://www.geeksforgeeks.org/how-to-install-rust-on-raspberry-pi/

[20]https://crates.io/crates/rppal

[21]https://github.com/golemparts/rppal/blob/master/src/gpio/gpiomem/bcm.rs

[22]https://ferrocene.dev/en/

[23]https://en.wikipedia.org/wiki/History_of_Unix