guide de programmation Bare-Metal (3)

en-têtes CMSIS, tests automatiques et serveur web

Sergey Lyubka (Irlande)

Dans les deux premières parties de ce guide, nous avons appris à accéder aux broches d'un microcontrôleur, à l'horloge système et à l'UART, et à réaliser nos premiers exemples de micrologiciels avec des scripts de l'éditeur de liens et des fichiers Makefile. Dans cette dernière partie de la série, nous allons nous simplifier encore la tâche grâce à des en-têtes et des bibliothèques prédéfinies. Nous programmerons un serveur web et apprenons à automatiser les constructions et les tests de ces micrologiciels plus complexes.

Note de la rédaction : ce guide est un document vivant sur GitHub [1].

En-têtes CMSIS du fabricant

Dans les articles précédents [2][3], nous avons développé le micrologiciel uniquement avec les fiches techniques, l'éditeur et le compilateur GCC. Nous avons créé les définitions des structures pour les périphériques manuellement, en utilisant les fiches techniques.
Maintenant que vous avez une idée générale du fonctionnement, il est temps de présenter les en-têtes CMSIS. Il s'agit de fichiers d'en-tête contenant toutes les définitions, créés et fournis par le fabricant du microcontrôleur. Ils contiennent des définitions pour les blocs internes et les périphériques de ce microcontrôleur, et sont donc assez volumineux.
CMSIS est l'acronyme de *Common Microcontroller Software Interface Standard* (norme d'interface logicielle commune pour les microcontrôleurs). Il s'agit d'une base commune permettant aux fabricants de microcontrôleurs de spécifier les API des périphériques. Étant donné que CMSIS est une norme ARM et que les en-têtes CMSIS sont fournis par le fabricant, ils constituent une source d'autorité. Il est donc préférable de les utiliser plutôt que d'écrire les définitions soi-même.
Il existe deux séries d'en-têtes CMSIS :

- En-têtes CMSIS du cœur ARM. Ils décrivent le cœur ARM et sont publiés par ARM sur GitHub [4]
- En-têtes CMSIS des fabricants de microcontrôleurs. Ils décrivent les périphériques du microcontrôleur et sont publiés par le fabricant du microcontrôleur. Dans notre cas, ST les publie à l'adresse [5]

Nous pouvons extraire ces en-têtes avec un simple extrait de Makefile :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
cmsis_core:
  git clone --depth 1 -b 5.9.0
   https://github.com/ARM-software/CMSIS_5 $@
cmsis_f4:
  git clone --depth 1 -b v2.6.8
   https://github.com/STMicroelectronics/
   cmsis_device_f4 $@
```

Le package CMSIS de ST fournit également des fichiers de démarrage pour tous leurs microcontrôleurs. Nous pouvons les utiliser au lieu d'écrire nous-mêmes le fichier *startup.c*. Le fichier *startup* fourni par ST appelle la fonction `SystemInit()`, nous la définissons donc dans le fichier *main.c*.
Maintenant, remplaçons nos fonctions API dans *hal.h* par des définitions CMSIS, et laissons le reste du micrologiciel intact. Dans *hal.h*, supprimez toutes les API et définitions de périphériques, et ne laissez que les *includes* C standard, l'inclusion CMSIS du fabricant, les définitions de PIN, BIT, FREQ, et la fonction d'aide `timer_expired()`.
Si nous essayons de reconstruire le micrologiciel – `make clean build`, GCC échouera, signalant l'absence de `systick_init()`, `GPIO_MODE_OUTPUT`, `uart_init()`, et UART3. Ajoutons ces éléments, en utilisant les fichiers CMSIS de STM32.
Commençons par `systick_init()`. Les en-têtes CMSIS d'ARM fournissent une fonction `SysTick_Config()` qui effectue la même tâche, nous allons donc l'utiliser.
Vient ensuite la fonction `gpio_set_mode()`. L'en-tête *stm32f429xx.h* contient une structure `GPIO_TypeDef`, identique à `struct gpio`. Utilisons-la :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/hal.h*)

```
#define GPIO(bank) ((GPIO_TypeDef *)
  (GPIOA_BASE + 0x400U * (bank)))
enum { GPIO_MODE_INPUT, GPIO_MODE_OUTPUT,
   GPIO_MODE_AF, GPIO_MODE_ANALOG };

static inline void gpio_set_mode
      (uint16_t pin, uint8_t mode) {
  GPIO_TypeDef *gpio =
   GPIO(PINBANK(pin));  // GPIO bank
```

Les fonctions `gpio_set_af()` et `gpio_write()` sont également simples – il suffit de remplacer `struct gpio` par `GPIO_TypeDef`. Vient ensuite l'UART. Il y a un `USART_TypeDef`, et des définitions pour USART1, USART2, USART3. Utilisons-les :

```
#define UART1 USART1
#define UART2 USART2
#define UART3 USART3
```

Dans `uart_init()` et le reste des fonctions UART, remplacez `struct uart` par `USART_TypeDef`. Le reste est inchangé !
Et voilà, c'est fait. Reconstruisez et reflashez le micrologiciel. La LED clignote, l'UART affiche la sortie. Félicitations, nous avons adapté le code du micrologiciel pour utiliser les fichiers d'en-tête CMSIS du fabricant. Maintenant, réorganisons un peu le dépôt en déplaçant tous les fichiers standards dans le répertoire *include* et en mettant à jour *Makefile* pour que GCC en soit averti :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
-I. -Iinclude -Icmsis_core/CMSIS/Core/Include -Icmsis_
f4/Include \
```

Incluons également l'en-tête CMSIS en tant que dépendance pour le fichier binaire :

```
firmware.elf: cmsis_core cmsis_f4 mcu.h
      link.ld Makefile $(SOURCES)
```

Nous nous retrouvons avec un exemple de projet réutilisable. Vous pouvez trouver le code source complet du projet dans le répertoire *step-5-cmsis* du projet [6].

Réglage des horloges

Après le démarrage, le processeur Nucleo-F429ZI fonctionne à 16 MHz. La fréquence maximale est de 180 MHz. Notez que la fréquence de l'horloge système n'est pas le seul facteur à prendre en compte. Les périphériques sont connectés à différents bus, APB1 et APB2, qui sont cadencés différemment. Leurs vitesses d'horloge sont configurées par les valeurs du prédiviseur de fréquence (*prescaler*), définies dans le contrôleur RCC (RCC gère les horloges du système et des périphériques). La source d'horloge principale du CPU peut également être différente – nous pouvons utiliser soit un oscillateur à cristal externe (HSE), soit un oscillateur interne (HSI). Dans notre exemple, nous utiliserons l'HSI.
Lorsque le CPU exécute des instructions à partir de la mémoire flash, la vitesse de lecture (environ 25 MHz) devient un goulot d'étranglement si l'horloge du CPU est plus rapide. Plusieurs astuces peuvent aider. La préfixation des instructions en est une. Nous pouvons également donner un indice au contrôleur de la mémoire flash quant à la vitesse de l'horloge système : cette valeur est appelée `FLASH_LATENCY`. Pour une horloge système de 180 MHz, la valeur `FLASH_LATENCY` est de 5. Les bits 8 et 9 du contrôleur de la flash activent les caches d'instructions et de données :

```
FLASH->ACR |= FLASH_LATENCY | BIT(8) |
    BIT(9); // Flash latency, caches
```

La source d'horloge (HSI ou HSE) utilise un matériel appelé PLL, qui multiplie la fréquence de la source par une certaine valeur. Ensuite, un ensemble de diviseurs de fréquence est utilisé pour régler l'horloge système et les horloges APB1 et APB2. Afin d'obtenir une valeur maximale de l'horloge système (180 MHz), plusieurs valeurs de diviseurs PLL et de prescaler APB sont possibles. La section 6.3.3 du manuel de référence du contrôleur STM32F4xx [7] nous indique les valeurs maximales pour l'horloge APB1 : ≤ 45 MHz, et l'horloge APB2 : ≤ 90 MHz. Cela réduit la liste des combinaisons possibles. Ici, nous avons choisi les valeurs manuellement. Notez que des outils comme CubeMX peuvent automatiser le processus et le rendre simple et visuel.

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h*)

```
 // 6.3.3: APB1 clock <= 45MHz;
 //       APB2 clock <= 90MHz
 // 3.5.1, Table 11: configure flash
// latency (WS) in accordance to clock freq
 // 33.4: The AHB clock must be at least
 // 25 MHz when Ethernet is used
 enum { APB1_PRE = 5 /* AHB clock / 4 */,
   APB2_PRE = 4 /* AHB clock / 2 */ };
 enum { PLL_HSI = 16, PLL_M = 8,
   PLL_N = 180, PLL_P = 2 };
      // Run at 180 Mhz
 #define FLASH_LATENCY 5
 #define SYS_FREQUENCY ((PLL_HSI * PLL_N /
       PLL_M / PLL_P) * 1000000)
 #define APB2_FREQUENCY
   (SYS_FREQUENCY / (BIT(APB2_PRE - 3)))
 #define APB1_FREQUENCY
   (SYS_FREQUENCY / (BIT(APB1_PRE - 3)))
```

Nous sommes maintenant prêts à utiliser un algorithme simple pour régler l'horloge des bus et des périphériques. Il peut ressembler à ceci :

- Optionnellement, activer le FPU
- Définir la latence de la flash
- Choisir une source d'horloge, une PLL et des prescalers APB1 et APB2
- Configurer le RCC en fixant les valeurs respectives
- Déplacer l'initialisation de l'horloge dans un fichier séparé *sysinit.c* ; la fonction `SystemInit()` qui est automatiquement appelée par le code de démarrage

Voir **listage 1** !

Listage 1. Initialisation de l'horloge.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/sysinit.c]

```
uint32_t SystemCoreClock = SYS_FREQUENCY;

 void SystemInit(void) {  // Called automatically by startup code
   SCB->CPACR |= ((3UL << 10 * 2) | (3UL << 11 * 2));  // Enable FPU
   FLASH->ACR |= FLASH_LATENCY | BIT(8) | BIT(9);      // Flash latency, prefetch
   RCC->PLLCFGR &= ~((BIT(17) - 1));                   // Clear PLL multipliers
   RCC->PLLCFGR |= (((PLL_P - 2) / 2) & 3) << 16;      // Set PLL_P
   RCC->PLLCFGR |= PLL_M | (PLL_N << 6);               // Set PLL_M and PLL_N
   RCC->CR |= BIT(24);                                 // Enable PLL
   while ((RCC->CR & BIT(25)) == 0) spin(1);           // Wait until done
   RCC->CFGR = (APB1_PRE << 10) | (APB2_PRE << 13);    // Set prescalers
   RCC->CFGR |= 2;                                     // Set clock source to PLL
   while ((RCC->CFGR & 12) == 0) spin(1);              // Wait until done

   RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;    // Enable SYSCFG
   SysTick_Config(SystemCoreClock / 1000);  // Sys tick every 1ms
 }
```

Nous devons également modifier *hal.h* – en particulier le code d'initialisation de l'UART. Les différents contrôleurs UART fonctionnent sur des bus différents : UART1 fonctionne sur un APB2 rapide, et le reste des UART fonctionne sur un APB1 plus lent. À une vitesse par défaut de 16 MHz, rien ne change. Mais, lorsqu'ils fonctionnent à des vitesses plus élevées, APB1 et APB2 peuvent avoir des horloges différentes, nous devons donc adapter le calcul du débit en bauds pour l'UART. Voir **listage 2**.
Reconstruisez et re-flashez, et notre carte fonctionnera à sa vitesse maximale, 180 MHz ! Le code source complet du projet peut être trouvé dans le répertoire *step-6-clock* du projet [8].

Serveur web avec tableau de bord

Le Nucleo-F429ZI est équipé d'Ethernet. Le matériel Ethernet nécessite deux composants : un PHY (qui transmet/reçoit les signaux électriques vers et au média, tel que le cuivre, le câble optique, etc.) et un MAC (qui pilote le contrôleur PHY). Sur la carte Nucleo, le contrôleur MAC est intégré et le PHY est externe (plus précisément, il s'agit du LAN8720a de Microchip).
MAC et PHY peuvent utiliser plusieurs interfaces. Nous utiliserons RMII. Pour cela, nous devons configurer un certain nombre de broches pour qu'elles utilisent leur fonction alternative (AF). Pour implémenter un serveur web, nous avons besoin de trois composants logiciels :

- un pilote réseau, qui envoie/reçoit des trames vers/depuis le contrôleur MAC via Ethernet
- une pile de réseau, qui analyse les trames et interprète TCP/IP
- une bibliothèque réseau qui interprète le protocole HTTP

Nous utiliserons la bibliothèque réseau Mongoose [9] qui implémente tout cela dans un seul fichier. Il s'agit d'une bibliothèque à double licence (*GPLv2/commercial*) conçue pour rendre le développement de réseaux intégrés rapide et facile.
Copiez donc *mongoose.c* [10] et *mongoose.h* [11] dans votre projet. Nous disposons désormais d'un pilote, d'une pile réseau et d'une bibliothèque. Mongoose fournit également un grand nombre d'exemples, dont un exemple de tableau de bord [12]. Il met en œuvre de nombreuses fonctions, telles que la connexion au tableau de bord, l'échange de données en temps réel via WebSocket, un système de fichiers intégré, la communication MQTT, etc. Utilisons donc cet exemple. Copiez deux fichiers supplémentaires :

- *net.c* [13] — implémente les fonctions du tableau de bord
- *packed_fs.c* [14] — contient des fichiers GUI HTML/CSS/JS

Nous devons indiquer à Mongoose les fonctionnalités à activer. Cela peut se faire via les drapeaux de compilation (*flags*), en définissant des constantes de préprocesseur. Alternativement, les mêmes constantes peuvent être définies dans le fichier *mongoose_custom.h*. Adoptons la seconde solution. Créez un fichier *mongoose_custom.h* contenant le code suivant :

```
#pragma once
#define MG_ARCH MG_ARCH_NEWLIB
#define MG_ENABLE_MIP 1
#define MG_ENABLE_PACKED_FS 1
#define MG_IO_SIZE 512
#define MG_ENABLE_CUSTOM_MILLIS 1
```

Il est temps d'ajouter du code réseau à *main.c*. Nous incluons `#include «mongoose.c»`, initialisons les broches RMII Ethernet et activons Ethernet dans le RCC. Voir **listage 3**.
Le pilote de Mongoose utilise l'interruption Ethernet, nous devons donc mettre à jour le fichier *startup.c* et ajouter *ETH_IRQHandler* à la table vectorielle. Réorganisons la définition de la table vectorielle dans *startup.c* pour ce qu'aucune modification ne soit nécessaire pour ajouter une fonction de gestion des interruptions. L'idée est d'utiliser un concept de "symbole faible".
Une fonction peut être marquée comme *faible* et fonctionner normalement. La différence consiste à définir une fonction portant le même nom ailleurs dans le code source. Normalement, deux

Listage 2. Initialisation de l'UART.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h]

```
static inline bool uart_init(USART_TypeDef *uart, unsigned long baud) {

  // https://www.st.com/resource/en/datasheet/stm32f429zi.pdf
  uint8_t af = 7;           // Alternate function
  uint16_t rx = 0, tx = 0;  // pins
  uint32_t freq = 0;        // Bus frequency. UART1 is on APB2, rest on APB1

  if (uart == USART1) {
    freq = APB2_FREQUENCY, RCC->APB2ENR |= BIT(4);
    tx = PIN('A', 9), rx = PIN('A', 10);
  } else if (uart == USART2) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(17);
    tx = PIN('A', 2), rx = PIN('A', 3);
  } else if (uart == USART3) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(18);
    tx = PIN('D', 8), rx = PIN('D', 9);
  } else {
    return false;
  }
```

Listage 3. Initialisation de l'Ethernet, activation des broches MAC GPIO.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/nucleo-f429zi/main.c]

```
uint16_t pins[] = ;
  for (size_t i = 0; i < sizeof(pins) / sizeof(pins[0]); i++) {
    gpio_init(pins[i], GPIO_MODE_AF, GPIO_OTYPE_PUSH_PULL, GPIO_SPEED_INSANE,
              GPIO_PULL_NONE, 11);
  }
  nvic_enable_irq(61);                          // Setup Ethernet IRQ handler
  RCC->APB2ENR |= BIT(14);                      // Enable SYSCFG
  SYSCFG->PMC |= BIT(23);                       // Use RMII. Goes first!
  RCC->AHB1ENR |= BIT(25) | BIT(26) | BIT(27);  // Enable Ethernet clocks
  RCC->AHB1RSTR |= BIT(25);                     // ETHMAC force reset
  RCC->AHB1RSTR &= ~BIT(25);                    // ETHMAC release reset
```

fonctions portant le même nom font échouer la compilation. Cependant, si une fonction est marquée comme faible, la compilation réussit et l'éditeur de liens sélectionne une fonction non faible. Cela permet de fournir une fonction "par défaut" dans un modèle de code, avec la possibilité de l'ignorer en créant simplement une fonction portant le même nom ailleurs dans le code.

Voici comment cela fonctionne dans notre exemple. Nous souhaitons remplir une table vectorielle avec des gestionnaires par défaut, mais donner à l'utilisateur la possibilité de remplacer n'importe quel gestionnaire. Pour cela, nous créons une fonction `DefaultIRQHandler()` et la marquons comme faible. Ensuite, pour chaque gestionnaire d'IRQ, nous déclarons un nom de gestionnaire et en faisons un alias de la fonction `DefaultIRQHandler()` :

```
void __attribute__((weak)) DefaultIRQHandler(void) {
for (;;) (void) 0;
}
#define WEAK_ALIAS
  __attribute__((weak, alias("DefaultIRQHandler")))
WEAK_ALIAS void NMI_Handler(void);
WEAK_ALIAS void HardFault_Handler(void);
WEAK_ALIAS void MemManage_Handler(void);
...
__attribute__((section(".vectors")))
  void (*tab[16 + 91])(void) =
    { 0, _reset, NMI_Handler,
     HardFault_Handler, MemManage_Handler,
    ...
```

Listage 4. Initialisation de la bibliothèque Mongoose.

```
struct mg_mgr mgr;        // Initialise Mongoose event manager
mg_mgr_init(&mgr);        // and attach it to the MIP interface
mg_log_set(MG_LL_DEBUG);  // Set log level
struct mip_driver_stm32 driver_data = {.mdc_cr = 4};  // See driver_stm32.h
struct mip_if mif = {
     .mac {2, 0, 1, 2, 3, 5}
     .use_dhcp = true,
     .driver = &mip_driver_stm32,
     .driver_data = &driver_data,
};
mip_init(&mgr, &mif);
extern void device_dashboard_fn(struct mg_connection *, int, void *, void *);
mg_http_listen(&mgr, "http://0.0.0.0", device_dashboard_fn, &mgr);
MG_INFO(("Init done, starting main loop"));
```

Listage 5. Makefile avec les références des bibliothèques.

```
847 3 mongoose.c:6784:arp_cache_add     ARP cache: added 0xc0a80001 @
90:5c:44:55:19:8b
 84e 2 mongoose.c:6817:onstatechange     READY, IP: 192.168.0.24
 854 2 mongoose.c:6818:onstatechange            GW: 192.168.0.1
 859 2 mongoose.c:6819:onstatechange            Lease: 86363 sec
 LED: 1, tick: 2262
 LED: 0, tick: 2512
```

Maintenant, nous pouvons définir le gestionnaire d'IRQ de notre choix dans notre code, et il remplacera le gestionnaire par défaut. C'est ce qui se passe dans notre exemple : il existe une fonction `ETH_IRQHandler()` définie par le pilote STM32 de Mongoose, qui remplace le gestionnaire par défaut.

L'étape suivante consiste à initialiser la bibliothèque Mongoose : créez un gestionnaire d'événements, configurez le pilote réseau et démarrez une connexion HTTP en écoute. Voir **listage 4**.

Il ne reste plus qu'à ajouter un appel à `mg_mgr_poll()` dans la boucle *main*.

Maintenant, ajoutez les fichiers *mongoose.c*, *net.c*, et *packed_fs.c* au Makefile. Reconstruisez et re-flashez la carte. Connectez une console série à la sortie de débogueur et observez la carte obtenir une adresse IP par DHCP. Voir **listage 5**.

Ouvrez un navigateur à cette adresse IP, et vous obtenez un tableau de bord fonctionnel, avec un graphique en temps réel via WebSocket, MQTT, l'authentification, et d'autres choses encore ! Voir la description complète pour plus de détails.

Le code source complet du projet se trouve dans le répertoire *step-7-webserver*. [15].

Constructions automatisées du micrologiciel (Software CI)

Avoir un test d'intégration continue (CI) est une bonne pratique dans tout projet de programmation. À chaque modification apportée au répertoire, l'intégration continue reconstruit et teste automatiquement tous les composants.

GitHub facilite cette tâche. Nous pouvons créer un fichier de configuration *CI .github/workflows/test.yml*. Dans ce fichier, nous pouvons installer le GCC d'ARM et exécuter `make` dans chaque répertoire d'exemple pour construire les fichiers respectifs du micrologiciel. En bref, cela indique à GitHub d'exécuter chaque répertoire lors d'un `push` :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

```
name: build
on: [push, pull_request]
```

Ceci installe le compilateur GCC d'ARM :

```
- run: sudo apt -y install
  gcc-arm-none-eabi make stlink-tools
```

Cela permet de construire le micrologiciel dans chaque répertoire :

```
- run: make -C step-0-minimal
- run: make -C step-1-blinky
- run: make -C step-2-systick
- run: make -C step-3-uart
- run: make -C step-4-printf
- run: make -C step-5-cmsis
- run: make -C step-6-clock
- run: make -C step-7-webserver/nucleo-f429zi
- run: make -C step-7-webserver/pico-w
```

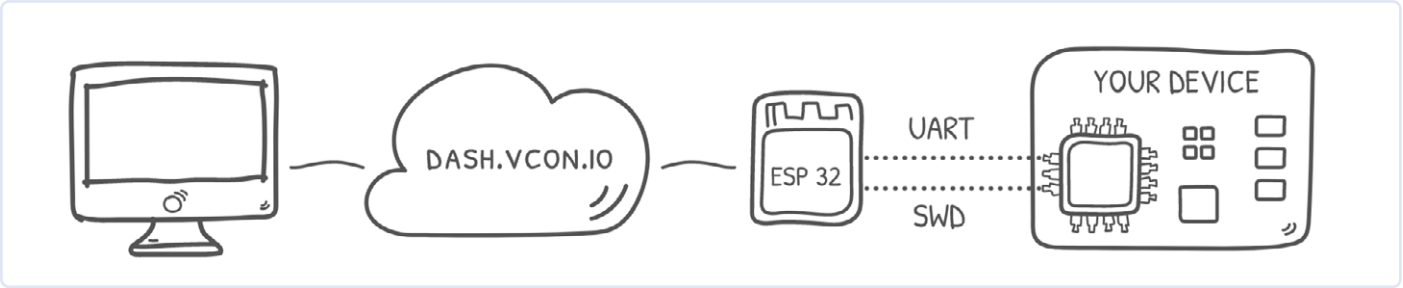


Figure 1. Configuration de l'ESP32 en tant que programmateur contrôlé à distance et inscription sur vcon.io.

C'est tout ! C'est très simple et très puissant. Maintenant, si nous apportons une modification au répertoire qui interrompt une construction, GitHub nous en informera. En cas de succès, GitHub ne fera rien. Voici un exemple d'exécution réussie [16].

Tests automatisés des micrologiciels (Hardware CI)

Serait-il possible de tester les fichiers binaires du micrologiciel construit sur un matériel réel, pour assurer que le micrologiciel construit est correct et fonctionnel ?
La construction d'un tel système ad hoc n'est pas simple. Par exemple, on peut installer une poste de travail de test dédié, y connecter un dispositif de test (par exemple une carte Nucleo-F429ZI), écrire un logiciel pour télécharger un micrologiciel à distance, et effectuer des tests avec un débogueur intégré. Cette méthode est possible, mais délicate, et demande beaucoup d'efforts et d'attention.
L'autre solution consiste à utiliser l'un des systèmes commerciaux de test de matériel, les *Embedded Eoard Farms* (EBF), bien que ces solutions commerciales soient assez coûteuses. Mais il existe un moyen plus simple.

Solution : ESP32 + vcon.io

Utilisons le service *http://vcon.io* service, qui permet la mise à jour à distance du micrologiciel et la surveillance de l'UART :

- Utilisez un ESP32 ou ESP32-C3 (une carte de développement abordable).
- Flasher un micrologiciel pré-construit, transformant ainsi l'ESP32 en un programmateur contrôlé à distance.
- Connectez l'ESP32 à l'appareil cible : broches SWD pour le clignotement, broches UART pour la lecture des données de sortie.
- Configurez l'ESP32 pour qu'il s'enregistre sur le tableau de bord de gestion *vcon.io* [17].

Lorsque cela est fait, votre appareil cible aura une API RESTful authentifiée et sécurisée pour reflasher et recueillir les données de sortie de l'appareil. Elle peut être appelée depuis n'importe où, par exemple depuis le logiciel CI (voir **figure 1**).

Note : le service *vcon.io* est géré par Cesanta - la société pour laquelle je travaille. Il s'agit d'un service payant avec un quota de gratuité : si vous n'avez que quelques appareils à gérer, le service est entièrement gratuit.

Configuration et câblage de l'ESP32

Utilisez n'importe quel ESP32 ou ESP32-C3 - une carte de développement, un module, ou votre appareil personnalisé. Je recommande la carte ESP32-C3 XIAO pour son prix abordable et sa petite taille. Nous allons supposer que l'appareil cible est une carte Raspberry Pi W5500-EVB-Pico [18] avec une interface Ethernet intégrée. Si votre appareil est différent, modifiez le câblage en fonction du brochage.

- Suivez les instructions [19] pour flasher votre ESP32.
- Configurez le réseau [20] pour enregistrer l'ESP32 sur le tableau de bord
- Suivez la section *Câblage* [21] pour connecter l'ESP32 à votre appareil.

La **figure 2** montre à quoi ressemble la configuration sur une plaque d'essai. La **figure 3** montre à quoi ressemble le tableau de bord d'un appareil configuré.
Vous pouvez maintenant reflasher votre appareil avec une seule commande :

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/
ID/ota --data-binary @firmware.bin
```

où `API_KEY` est la clé d'authentification sur *dash.vcon.io*, `ID` est le numéro de l'appareil enregistré, et *firmware.bin* est le nom du nouveau micrologiciel. Vous pouvez obtenir la clé API_KEY en cliquant sur le lien *api key* dans un tableau de bord. L'ID de l'appareil est indiqué dans le tableau.
Nous pouvons également capturer la sortie d'un appareil avec une seule commande :

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/
ID/tx?t=5
```

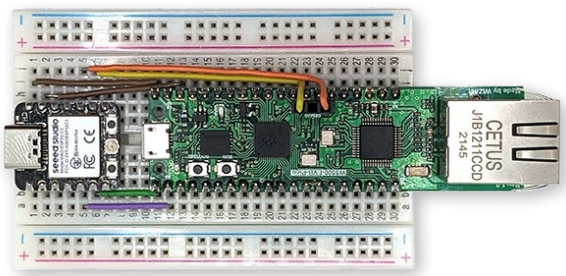


Figure 2. Configuration de l'appareil sur une plaque d'essai.

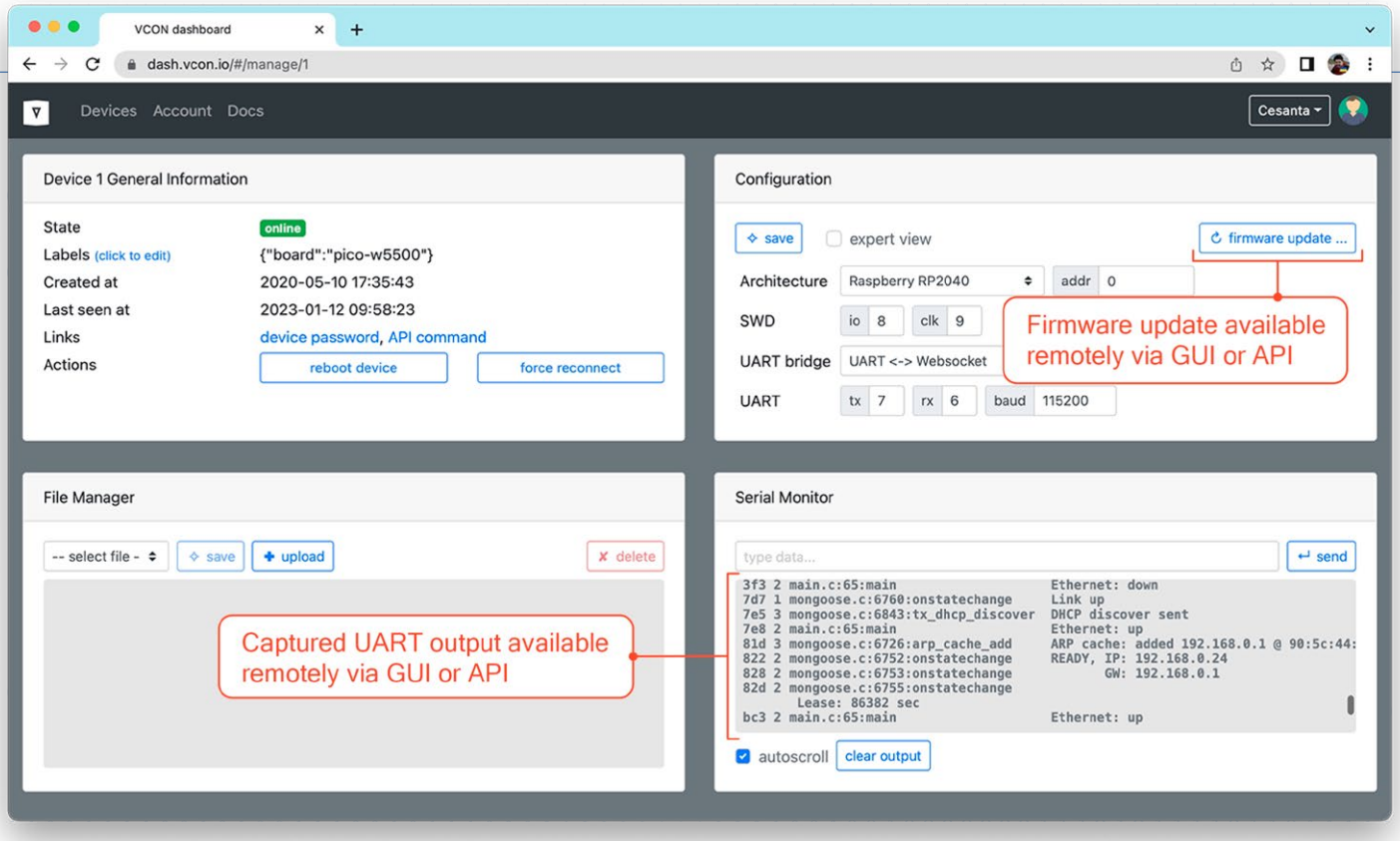


Figure 3. Tableau de bord de l'appareil configuré.

Ici, `t=5` signifie : attendre 5 secondes pendant la capture du signal de sortie de l'UART.
Maintenant, nous pouvons utiliser ces deux commandes dans n'importe quelle plateforme CI logicielle pour tester un nouveau micrologiciel sur un appareil réel, et tester la sortie UART de l'appareil par rapport à certains mots-clés prévus.

Intégration avec les actions de GitHub

Le logiciel CI construit une image du micrologiciel, et maintenant nous pouvons même la tester sur un vrai matériel ! Nous devrions ajouter quelques commandes supplémentaires qui utilisent l'utilitaire `curl` pour envoyer le micrologiciel construit à la carte de test, et ensuite capturer sa sortie de débogage.
La commande `curl` nécessite une clé API secrète, que nous ne voulons pas exposer au public. En général, la bonne façon de procéder est la suivante :

- Allez dans les paramètres de votre projet sur GitHub (vous pouvez cloner le dépôt avec les exemples du serveur web), puis allez sur / *Secrets* / *Actions*
- Cliquez sur le bouton *New repository secret*
- Nommez-le `VCON_API_KEY`, collez la valeur dans une case *Secret*, cliquez sur *Add secret*

L'un des projets d'exemple [22] construit un micrologiciel pour la carte RP2040-W5500, alors flashons-le avec une commande `curl` et une clé API sauvegardée. La meilleure façon est d'ajouter une cible de Makefile pour les tests, et de laisser *GitHub Actions* (logiciel *CI*) l'appeler :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

```
- run: make -C step-7-webserver/pico-w5500 test VCON_API_KEY=${{secrets.VCON_API_KEY}}
```

Notez que nous donnons une variable d'environnement `VCON_API_KEY` à `make`. Notez également que nous invoquons la cible `test` de Makefile, qui devrait construire et tester le micrologiciel. Dans le **listage 6**, vous pouvez voir la cible `test` de Makefile.

Explication :

- ligne 34 : la cible `test` dépend de la cible du téléversement, c'est pourquoi le téléversement est exécuté en premier. (voir ligne 38)
- ligne 35 : lit l'UART pendant 5 secondes et l'enregistre dans */tmp/output.txt*
- ligne 36 : cherche la chaîne de caractères `Ethernet : up` dans la sortie, et échoue si elle n'est pas trouvée.
- ligne 38 : la cible de `upload` dépend de la version, c'est pourquoi `build` s'exécute avant de tester le micrologiciel.
- ligne 39 : nous flashons le micrologiciel à distance. Le drapeau `--fail` de l'utilitaire `curl` le fait échouer si la réponse du serveur n'est pas réussie (not HTTP 200 OK).

Dans le **listage 7**, vous pouvez trouver l'exemple de sortie de la commande `make test` décrite ci-dessus.
C'est fait ! Maintenant, nos tests automatiques assurent que le micrologiciel peut être construit, qu'il est bootable, et qu'il initialise la pile réseau correctement. On peut facilement développer cette méthode : il suffit d'ajouter des actions plus complexes dans les fichiers binaires de votre micrologiciel, d'imprimer le résultat via l'UART, et de vérifier la sortie attendue au cours du test.

Bons tests !

220665-C-04

Listage 6. Makefile pour le serveur web sur le Pico.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/pico-w5500/Makefile]

```
32 # Requires env variable VCON_API_KEY set
33 DEVICE_URL ?= https://dash.vcon.io/api/v3/devices/1
34 test: update
35   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/tx?t=5 | tee /tmp/output.txt
36   grep 'Ethernet: up' /tmp/output.txt

38 update: build
39   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/ota --data-binary @firmware.bin
```

Listage 7. Sortie de la commande Make Test.

```
$ make test
 curl --fail ...
 {"success":true,"written":59904}
 curl --fail ...
 3f3 2 main.c:65:main                    Ethernet: down
 7d7 1 mongoose.c:6760:onstatechange     Link up
 7e5 3 mongoose.c:6843:tx_dhcp_discover  DHCP discover sent
 7e8 2 main.c:65:main                    Ethernet: up
 81d 3 mongoose.c:6726:arp_cache_add     ARP cache: added 192.168.0.1 @
90:5c:44:55:19:8b
 822 2 mongoose.c:6752:onstatechange     READY, IP: 192.168.0.24
 827 2 mongoose.c:6753:onstatechange            GW: 192.168.0.1
 82d 2 mongoose.c:6755:onstatechange            Lease: 86336 sec
 bc3 2 main.c:65:main                    Ethernet: up
 fab 2 main.c:65:main                    Ethernet: up
```

Questions ou commentaires ?

Envoyez un courriel à l'auteur (sergey.lyubka@cesanta.com) ou contactez Elektor (redaction@elektor.fr).

À propos de l'auteur

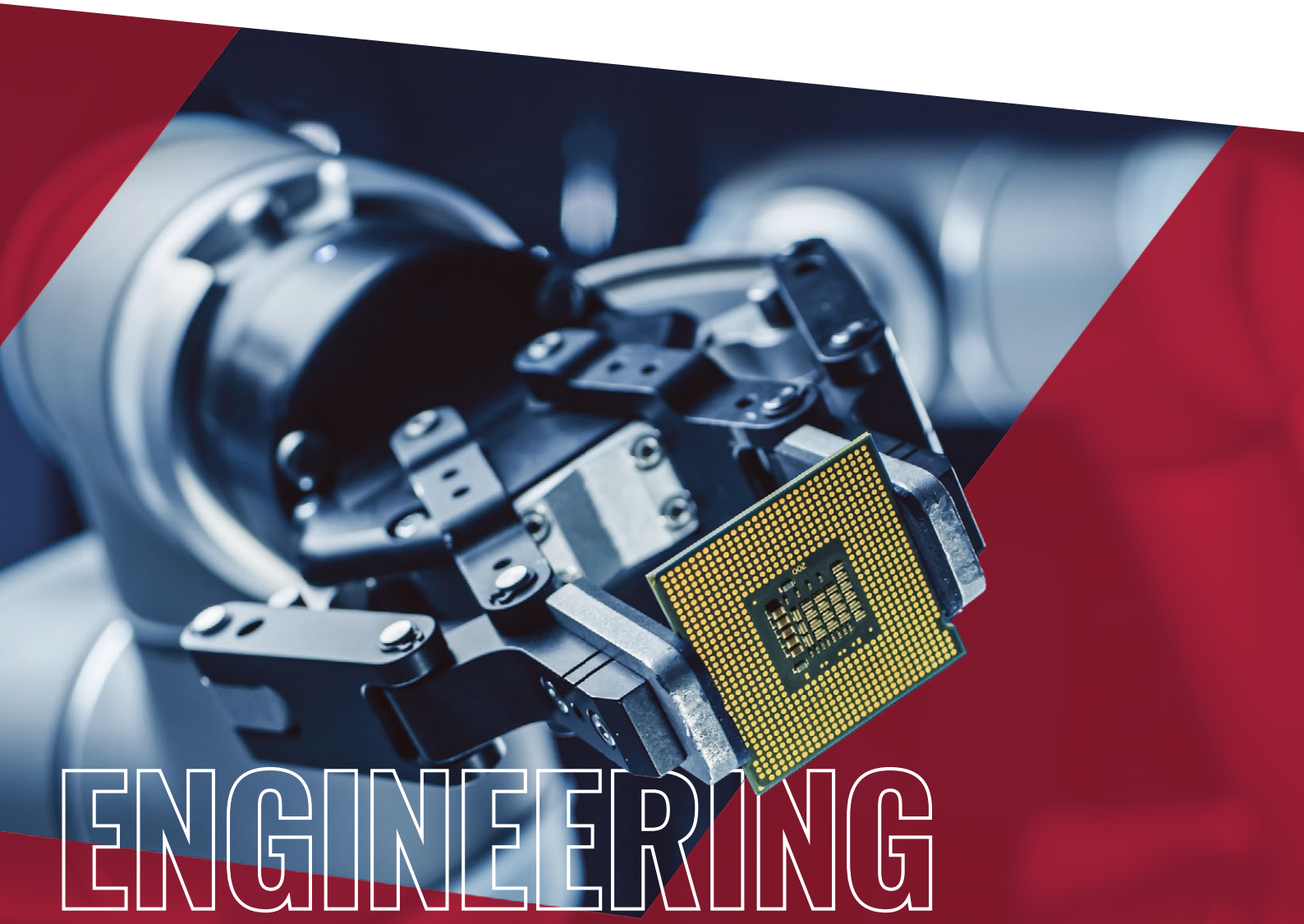
Sergey Lyubka est ingénieur et entrepreneur. Il est titulaire d'un Master en physique de l'université d'État de Kiev, en Ukraine. Sergey est directeur et cofondateur de Cesanta, une entreprise technologique basée à Dublin, en Irlande (Embedded Web Server for electronic devices : https://mongoose.ws). Il est passionné par la programmation embarquée « bare-metal » de réseaux

Produits

- **Dogan Ibrahim,** *Nucleo Boards Programming with the STM32CubeIDE* **(Elektor 2020)**
  www.elektor.fr/19530
- **Dogan Ibrahim,** *Programming with STM32 Nucleo Boards* **(Elektor 2015)**
  www.elektor.com/18585

LIENS

[1] Ce guide sur GitHub : https://github.com/cpq/bare-metal-programming-guide
[2] Sergey Lyubka, « Guide de programmation bare-metal (1) » Elektor 7-8/2023 : https://elektormagazine.fr/220665-04
[3] Sergey Lyubka, « guide de programmation bare-metal (2) » Elektor 9-10/2023 : https://elektormagazine.fr/220665-B-02
[4] CMSIS version 5 : https://github.com/ARM-software/CMSIS_5
[5] En-têtes STM32 CMSIS pour la famille F4 : https://github.com/STMicroelectronics/cmsis_device_f4
[6] Step 5 CMSIS folder : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-5-cmsis
[7] Manuel de référence RM0090 pour les contrôleurs STM32F4xx [PDF] : https://tinyurl.com/stm32f4man
[8] Step 6 Clock Folder : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-6-clock
[9] Bibliothèque du réseau Mongoose : https://github.com/cesanta/mongoose
[10] mongoose.c : https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.c
[11] mongoose.h : https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.h
[12] Exemples de Mongoose : https://github.com/cesanta/mongoose/tree/master/examples/device-dashboard
[13] Exemple de serveur web, net.c : https://raw.githubusercontent.com/cesanta/mongoose/master/examples/device-dashboard/net.c
[14] Exemple de serveur web, packed_fs.c : https://tinyurl.com/packedfsc
[15] Step 7 Webserver directory : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-7-webserver
[16] Example: Successful run of Continuous Integration : https://tinyurl.com/bmgoodrun
[17] vcon.io: https://dash.vcon.io/
[18] Carte Raspberry Pi W5500-EVB-Pico : https://docs.wiznet.io/Product/iEthernet/W5500/w5500-evb-pico
[19] Flashage de l'ESP32 : https://vcon.io/docs/#module-flashing
[20] Configuration du réseau : https://vcon.io/docs/#module-registration
[21] Câblage : https://vcon.io/docs/#module-to-device-wiring
[22] Exemple de serveur web pour la carte Pico : https://tinyurl.com/picowebeg



ENGINEERING A BETTER WORLD

IMAGINE THAT. WE CAN.

Let us take you on a journey of discovery. Whether you design, build or maintain, we have ingenious technologies and solutions to help engineer a better world for everyone.