guide de programmation bare-metal (2)

timer précis, UART et débogage

Sergey Lyubka (Irelande)

Dans la première partie du guide, nous avons appris à accéder aux registres du microcontrôleur pour contrôler les broches. De plus, nous avons créé un micrologiciel simple et notre première démonstration de LED clignotante avec un script d'éditeur de liens et d'un fichier *Makefile*. Dans cet épisode de la série, nous abordons la synchronisation précise via les horloges système, l'UART et le débogage.

Note de la rédaction : ce guide est un document vivant sur GitHub [1] qui évolue. Nous avons donc décidé de poursuivre cette série par un épisode supplémentaire, qui sera publié dans le prochain numéro du magazine Elektor (11-12/2023).

Blinky avec interruption de Systick

Pour notre premier exemple de clignotement de LED «Blinky», nous avons utilisé une fonction de délai appelée `spin()` qui exécute simplement des instructions NOP un certain nombre de fois (voir la première partie de la série [2]).

Afin d'assurer une temporisation beaucoup plus précise, nous devons activer l'interruption SysTick d'ARM. SysTick est un compteur matériel de 24 bits, et fait partie du noyau ARM, il est donc documenté par le manuel *Arm® v7-M Architecture Reference Manual* [3]. Le manuel indique que SysTick possède quatre registres :

- CTRL — utilisé pour activer/désactiver SysTick
- LOAD — valeur initiale du compteur
- VAL — la valeur actuelle du compteur, décrémentée à chaque cycle d'horloge
- CALIB — registre de calibration

Chaque fois que VAL passe à zéro, une interruption SysTick est générée. L'index de l'interruption SysTick dans la table vectorielle est 15, nous devons donc le définir. Au démarrage, notre carte Nucleo-F429ZI de STMicroelectronics tourne à 16 MHz. Nous pouvons configurer le compteur SysTick pour qu'il déclenche une interruption toutes les millisecondes.

Tout d'abord, définissons un périphérique SysTick. Il existe quatre registres et, d'après le manuel de référence d'Arm, l'adresse du SysTick est 0xe000e010. Donc :

```
struct systick {
  volatile uint32_t CTRL, LOAD, VAL, CALIB;
};
#define SYSTICK ((struct systick *) 0xe000e010)
```

Ensuite, ajoutez une fonction API qui le configure. Nous devons activer SysTick dans le registre `SYSTICK->CTRL` et le cadencer via `RCC->APB2ENR`, comme décrit dans la section 7.3.14 du manuel [4] :

```
#define BIT(x) (1UL << (x))
static inline void systick_init(uint32_t ticks) {
  // SysTick timer is 24 bits
  if ((ticks - 1) > 0xffffff) return;
  SYSTICK->LOAD = ticks - 1;
  SYSTICK->VAL = 0;
  // Enable systick
  SYSTICK->CTRL = BIT(0) | BIT(1) | BIT(2);
  RCC->APB2ENR |= BIT(14); // Enable SYSCFG
}
```

La carte Nucleo-F429ZI tourne à 16 MHz, c'est-à-dire que si nous appelons `systick_init(16000000 / 1000)`, une interruption SysTick sera générée toutes les millisecondes. Nous devrions définir une fonction de gestion d'interruption – en voici une qui incrémente simplement un compteur de millisecondes de 32 bits :

```
// "volatile" is important!!
static volatile uint32_t s_ticks;
void SysTick_Handler(void) {
  s_ticks++;
}
```

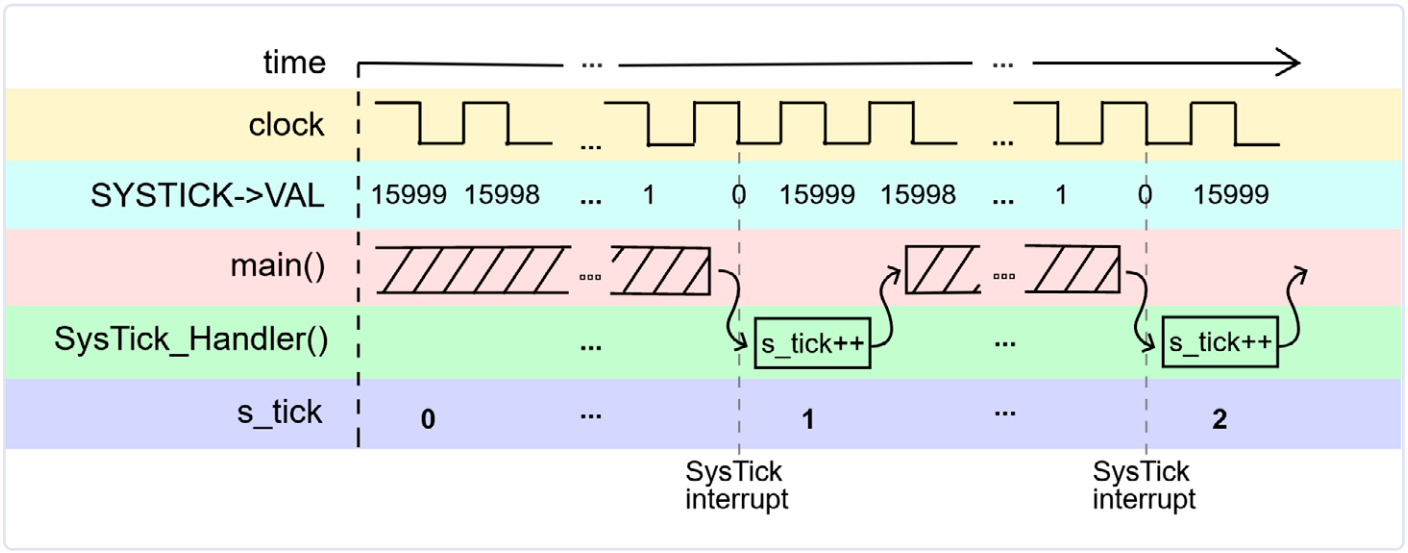

Figure 1. Représentation temporelle de l'exécution interrompue du micrologiciel avec la fonction SysTick_Handler().

Avec une horloge de 16 MHz, nous initialisons le compteur `SysTick` pour qu'il déclenche une interruption tous les 16 000 cycles : La valeur initiale de `SYSTICK->VAL` est de 15 999, puis elle se décrémente à chaque cycle jusqu'à atteindre 0, lorsqu'une interruption est déclenchée. L'exécution du code du micrologiciel est interrompue, et la fonction `SysTick_Handler()` est appelée pour incrémenter la variable `s_tick`. La **figure 1** illustre ceci sur une échelle de temps. Le qualificateur `volatile` est nécessaire ici parce que `s_ticks` est modifié par le gestionnaire d'interruption. `volatile` empêche le compilateur d'optimiser/cacher la valeur de `s_ticks` dans un registre du CPU ; le code généré accède toujours à la mémoire. C'est pourquoi le qualificateur `volatile` est également inclus dans les structures de définition des périphériques. Pour bien comprendre cela, démontrons-le avec une fonction simple : la fonction `delay()` d'Arduino. Utilisons la variable `s_ticks` :

```
// This function waits "ms" milliseconds
void delay(unsigned ms) {
  // Time in the future when we need to stop
  uint32_t until = s_ticks + ms; /
  while (s_ticks < until) (void) 0; // Loop until then
}
```

Compilons maintenant ce code avec et sans le qualificateur `volatile` de `s_ticks` et comparons-le avec le code assembleur compilé :

```
// NO VOLATILE: uint32_t s_ticks;
ldr r3, [pc, #8] // cache s_ticks
ldr r3, [r3, #0] // in r3
adds r0, r3, r0 // r0 = r3 + ms
cmp r3, r0 // ALWAYS FALSE
bcc.n 200000d2
bx lr

// VOLATILE: volatile uint32_t s_ticks;
ldr r2, [pc, #12]
ldr r3, [r2, #0] // r3 = s_ticks
adds r3, r3, r0 // r3 = r3 + ms
ldr r1, [r2, #0] // RELOAD: r1 = s_ticks
cmp r1, r3 // compare
bcc.n 200000d2
bx lr
```

Sans le qualificateur `volatile`, la fonction `delay()` bouclera indéfiniment et ne renverra rien. C'est parce qu'elle met en cache (optimise) la valeur de `s_ticks` dans un registre et ne la met jamais à jour. Le compilateur fait cela parce qu'il ne sait pas que `s_ticks` sera mis à jour ailleurs par le gestionnaire d'interruption ! En revanche, le code compilé avec `volatile`, charge la valeur de `s_ticks` à chaque itération. La règle de base est donc la suivante : **les valeurs en mémoire qui sont mises à jour par les gestionnaires d'interruption ou par le matériel doivent être déclarées comme** `volatile`**.**

Nous devons maintenant ajouter le gestionnaire d'interruption `SysTick_Handler()` à la table vectorielle :

```
__attribute__((section(".vectors")))
    void (*tab[16 + 91])(void) = {
    _estack, _reset, 0, 0, 0, 0, 0, 0, 0, 0,
                     0, 0, 0, 0,  SysTick_Handler
};
```

Nous avons maintenant une horloge précise à la milliseconde ! Créons une fonction d'aide pour les horloges périodiques arbitraires :

```
// t: expiration time, prd: period,
//    now: current time. Return true if expired
bool timer_expired(uint32_t *t, uint32_t prd,
                   uint32_t now) {
  if (now + prd < *t) *t = 0;
    // Time wrapped? Reset timer
  if (*t == 0) *t = now + prd;
    // First poll? Set expiration
  if (*t > now) return false;
    // Not expired yet,return
  *t = (now - *t) > prd ? now + prd : *t + prd;
    // Next expiration time
  return true; // Expired, return true
}
```

Nous pouvons maintenant mettre à jour la boucle principale et utiliser un timer précis pour le clignotement de la LED. Par exemple, utilisons un intervalle de clignotement de 250 millisecondes :

```
// Declare timer and 500ms period
uint32_t timer, period = 500;
for (;;) {
  if (timer_expired(&timer, period, s_ticks)) {
    static bool on;       // This block is executed
    gpio_write(led, on); // Every "period" milliseconds
    on = !on; // Toggle LED state
  }
  // Here we could perform other activities!
}
```

Notez qu'en utilisant SysTick avec une fonction d'aide `timer_expired()`, la boucle principale (également appelée «superloop») devient non bloquante. Cela signifie qu'à l'intérieur de cette boucle, nous pouvons effectuer de nombreuses actions – par exemple, avoir différents timers avec différentes périodes, et ils seront tous déclenchés à temps.
Le code source complet du projet est disponible dans le dossier *step-2-systick* [5].

Ajouter une sortie de débogage UART

Ajoutons maintenant une analyse lisible au micrologiciel. L'un des périphériques du microcontrôleur est une interface UART série. La topographie de la mémoire présentée dans la section 2.3 du manuel montre qu'il existe plusieurs contrôleurs UART/USART - c'est-à-dire que des circuits à l'intérieur du microcontrôleur peuvent échanger des données par l'intermédiaire de certaines broches lorsqu'ils sont correctement configurés. Une configuration UART minimale utilise deux broches : RX (réception) et TX (transmission).
La section 6.9 du manuel de la carte Nucleo [6] montre que l'un des contrôleurs, USART3, utilise les broches PD8 (TX) et PD9 (RX) et est connecté au débogueur ST-LINK intégré. Cela signifie que si nous configurons USART3 et transmettons des données via la broche PD9, nous pouvons les voir sur notre PC via une connexion USB ST-LINK.
Nous allons donc créer une API pratique pour l'UART, comme nous l'avons fait pour les GPIO. La section 30.6 [4] présente un résumé des registres UART. La structure UART correspondante `struct` est :

```
struct uart {
  volatile uint32_t SR, DR, BRR, CR1, CR2, CR3, GTPR;
};
#define UART1 ((struct uart *) 0x40011000)
#define UART2 ((struct uart *) 0x40004400)
#define UART3 ((struct uart *) 0x40004800)
```

Pour configurer un UART, il faut :

- Activer l'horloge UART en définissant le bit approprié dans `RCC->APB2ENR`.
- Définir le mode de fonctionnement alternatif (*alternate function*) des broches RX et TX. Il peut y avoir plusieurs fonctions alternatives (AF) pour une broche donnée, selon le périphérique utilisé. La liste des fonctions alternatives se trouve dansle tableau 12 de la fiche technique du STM32F429ZI. [7].
- Régler la vitesse de transmission (fréquence de réception/transmission de l'horloge) via le registre BRR.
- Activer la réception et la transmission du périphérique via le registre CR1.

Nous savons déjà comment configurer une broche GPIO dans un mode spécifique. Si une broche est en mode AF, nous devons également spécifier le «numéro de fonction», c'est-à-dire le périphérique exact qui prend le contrôle. Cela est possible via le registre AFR (*Alternate Function Register*) du périphérique GPIO. En lisant la description du registre `AFR` dans le manuel de référence, nous voyons que le numéro *AF* occupe quatre bits, donc la configuration complète pour 16 broches occupe deux registres.
Afin de masquer complètement le code spécifique au registre de l'API GPIO, déplaçons l'initialisation de l'horloge GPIO dans la fonction `gpio_set_mode()` :

```
static inline void
  gpio_set_mode(uint16_t pin, uint8_t mode) {
    struct gpio *gpio = GPIO(PINBANK(pin)); // GPIO bank
    int n = PINNO(pin); // Pin number
// Enable GPIO clock
    RCC->AHB1ENR |= BIT(PINBANK(pin));
    ...
```

Maintenant, créeons une fonction API d'initialisation de l'UART - voir **listage 1**.
Enfin, nous avons besoin de fonctions pour lire et écrire sur l'UART. La section 30.6.1 du manuel de référence [4] précise que le registre d'état, SR, indique si les données sont prêtes :

```
static inline int uart_read_ready(struct uart *uart) {
  // If RXNE bit is set, data is ready
  return uart->SR & BIT(5);
}
```

Il est possible d'extraire l'octet de données lui-même du registre de données DR :

```
static inline uint8_t uart_read_byte(struct uart *uart) {
  return (uint8_t) (uart->DR & 255);
}
```

Nous pouvons également transmettre un seul octet via le registre de données. Après avoir défini un octet à écrire, il faut attendre la fin de la transmission, qui sera indiquée par le Bit 7 du registre *Status* :

Listage 1. Fonction API d'initialisation de l'UART.

```
#define FREQ 16000000  // CPU frequency, 16 Mhz
 static inline void uart_init(struct uart *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

   if (uart == UART1) RCC->APB2ENR |= BIT(4);
   if (uart == UART2) RCC->APB1ENR |= BIT(17);
   if (uart == UART3) RCC->APB1ENR |= BIT(18);
   if (uart == UART1) tx = PIN('A', 9), rx = PIN('A', 10);
   if (uart == UART2) tx = PIN('A', 2), rx = PIN('A', 3);
   if (uart == UART3) tx = PIN('D', 8), rx = PIN('D', 9);

  gpio_set_mode(tx, GPIO_MODE_AF);
   gpio_set_af(tx, af);
   gpio_set_mode(rx, GPIO_MODE_AF);
   gpio_set_af(rx, af);
   uart->CR1 = 0;                          // Disable this UART
   uart->BRR = FREQ / baud;                // FREQ is a UART bus frequency
   uart->CR1 |= BIT(13) | BIT(2) | BIT(3);  // Set UE, RE, TE
 }
```

```
static inline void uart_write_byte(struct uart *uart,
                                   uint8_t byte) {
  uart->DR = byte;
  while ((uart->SR & BIT(7)) == 0) spin(1);
}
```

Et pour l'écriture dans un tampon :

```
static inline void
  uart_write_buf(struct uart *uart,
                 char *buf, size_t len) {
    while (len-- > 0)
      uart_write_byte(uart, *(uint8_t *) buf++);
}
```

Initialisons l'UART dans la fonction `main()` :

```
…
uart_init(UART3, 115200); // Initialize UART
```

Nous pouvons maintenant afficher le message *hi\r\n* à chaque fois que la LED clignote !

```
if (timer_expired(&timer, period, s_ticks)) {
  …
  uart_write_buf(UART3, "hi\r\n", 4); // Write message
}
```

Reconstruisez, re-flashez et connectez un programme de terminal au port ST-LINK. Sur mon Mac, j'utilise *cu*, que vous pouvez également utiliser sous Linux. Sous Windows, l'utilitaire PuTTY [8] fonctionne bien. Lancez un terminal et observez les messages :

```
$ cu -l /dev/cu.YOUR_SERIAL_PORT -s 115200
hi
hi
```

Le code source complet du projet est disponible dans le dossier *step-3-uart* [9].

Rediriger printf() vers UART

Dans cette section, nous appelons la fonction `printf()` au lieu de `uart_write_buf()`, ce qui nous permet d'obtenir une sortie formatée - d'avoir plus de flexibilité d'affichage d'informations de diagnostic, en implémentant ce que l'on appelle "printf()-style debugging".

La suite d'outils GNU d'ARM que nous utilisons est livrée non seulement avec un compilateur GCC et d'autres outils, mais aussi avec une bibliothèque C appelée *newlib* [10]. La bibliothèque *newlib* a été développée par RedHat pour les systèmes embarqués.

Si le micrologiciel appelle une fonction de la bibliothèque C standard, par exemple `strcmp()`, un code de *newlib* sera ajouté par l'éditeur de liens GCC.

Certaines des fonctions C standard que la bibliothèque *newlib* implémente d'une manière spéciale, en particulier les opérations d'entrée/sortie (IO) de fichiers : ces fonctions appellent finalement un ensemble de fonctions d'E/S de bas niveau appelées *syscalls*.

Par exemple

- `fopen()` appelle éventuellement `_open()`
- `fread()` appelle éventuellement une fonction `_read()` de bas niveau
- `fwrite()`, `fprintf()`, `printf()` appellent éventuellement une fonction `_write()` de bas niveau
- `malloc()` appelle éventuellement `_sbrk()`, etc.

Listage 2. La fonction main() devient assez compacte

```
#include "hal.h"

static volatile uint32_t s_ticks;
 void SysTick_Handler(void) {
   s_ticks++;
 }

int main(void) {
   uint16_t led = PIN('B', 7);            // Blue LED
   systick_init(16000000 / 1000);         // Tick every 1 ms
   gpio_set_mode(led, GPIO_MODE_OUTPUT);  // Set blue LED to output mode
   uart_init(UART3, 115200);              // Initialise UART
   uint32_t timer = 0, period = 500;      // Declare timer and 500ms period
   for (;;) {
     if (timer_expired(&timer, period, s_ticks)) {
       static bool on;                      // This block is executed
       gpio_write(led, on);                 // Every 'period' milliseconds
       on = !on;                            // Toggle LED state
       uart_write_buf(UART3, "hi\r\n", 4);  // Write message
     }
     // Here we could perform other activities!
   }
   return 0;
 }
```

Ainsi, en modifiant l'appel système (syscall) de `_write()`, nous pouvons rediriger `printf()` vers ce que nous voulons. Ce mécanisme est appelé « *IO retargeting* ».
Note : L'EDI Cube de STM32 utilise aussi le compilateur GCC d'ARM avec newlib, c'est pourquoi les projets Cube incluent typiquement un fichier *syscalls.c*. D'autres suites d'outils, comme CCS de TI et CC de Keil, peuvent utiliser une bibliothèque C différente avec un mécanisme de *retargeting* légèrement différent. Nous utilisons newlib, donc modifions l'appel syscall de `_write()` pour imprimer sur UART3.
Avant cette étape, organisons le code source comme suit :

- Déplacer toutes les définitions d'API dans le fichier *mcu.h*.
- Déplacer le code de démarrage dans *startup.c*.
- Créer un fichier vide *syscalls.c* pour les syscalls de *newlib*.
- Modifier Makefile pour ajouter *syscalls.c* et *startup.c* à la construction.

Après avoir déplacé toutes les définitions d'API dans *mcu.h*, le fichier *main.c* devient assez compact. Notez qu'il ne contient aucune mention des registres de bas niveau, seulement des fonctions API de haut niveau faciles à comprendre – voir **listage 2**.
Maintenant, nous allons recibler `printf()` vers UART3. Dans le fichier vide *syscalls.c*, copiez/collez le code suivant :

```
#include "mcu.h"
int _write(int fd, char *ptr, int len) {
  (void) fd, (void) ptr, (void) len;
  if (fd == 1) uart_write_buf(UART3, ptr, (size_t) len);
  return -1;
}
```

Ici, si le descripteur de fichier sur lequel nous écrivons `fd`, est 1 (ce qui est un descripteur de sortie standard), alors écrivez le tampon sur UART3. Dans le cas contraire, il faut l'ignorer. C'est la base du *retargeting* !
Reconstruire ce micrologiciel résulte en un tas d'erreurs de linker, comme le montre le **listage 3**.
Comme nous avons utilisé une fonction stdio de *newlib*, nous devons fournir à la bibliothèque le reste des appels systèmes. Ajoutons un simple *stub* qui ne fait rien (**listage 4**).
La reconstruction ne donne aucune erreur. Dernière étape : remplacer l'appel de `uart_write_buf()` dans la fonction `main()` par un appel de `printf()` qui affiche quelque chose d'utile, par exemple l'état d'une LED et la valeur actuelle de systick :

```
// Write message
printf("LED: %d, tick: %lu\r\n", on, s_ticks);
```

La sortie série ressemble à ceci :

```
LED: 1, tick: 250
LED: 0, tick: 500
LED: 1, tick: 750
LED: 0, tick: 1000
```

Félicitations ! Nous avons appris comment fonctionne le *retargeting* d'entrées-sorties et nous pouvons maintenant déboguer le micrologiciel avec la fonction `printf()-debug`. Vous pouvez trouver le code source complet du projet dans le dossier *step-4-printf* [11].

Listage 3. Un tas d'erreurs d'édition de liens

```
../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a(lib_a-sbrkr.o): in function `_sbrk_r':
 sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
 closer.c:(.text._close_r+0xc): undefined reference to `_close'
 lseekr.c:(.text._lseek_r+0x10): undefined reference to `_lseek'
 readr.c:(.text._read_r+0x10): undefined reference to `_read'
 fstatr.c:(.text._fstat_r+0xe): undefined reference to `_fstat'
 isattyr.c:(.text._isatty_r+0xc): undefined reference to `_isatty'
```

Listage 4. Ajout de simples stubs

```
int _fstat(int fd, struct stat *st) {
   (void) fd, (void) st;
   return -1;
 }

void *_sbrk(int incr) {
   (void) incr;
   return NULL;
 }

int _close(int fd) {
   (void) fd;
   return -1;
 }

int _isatty(int fd) {
   (void) fd;
   return 1;
 }

int _read(int fd, char *ptr, int len) {
   (void) fd, (void) ptr, (void) len;
   return -1;
 }

int _lseek(int fd, int ptr, int dir) {
   (void) fd, (void) ptr, (void) dir;
   return 0;
 }
```

Déboguer avec Ozone de Segger

Que se passe-t-il si notre micrologiciel est bloqué quelque part et que la fonction de débogage `printf()` ne fonctionne pas ? Et si même le code de démarrage ne fonctionne pas ? Nous aurions besoin d'un débogueur. Il y a beaucoup d'options, mais je recommanderais d'utiliser le débogueur Ozone de Segger. Pourquoi ? Parce qu'il est autonome ; il n'a pas besoin de configuration d'EDI. Nous pouvons envoyer notre fichier *firmware.elf* directement à Ozone, et il récupérera nos fichiers sources.

Téléchargez donc Ozone depuis le site de Segger [12]. Avant de pouvoir l'utiliser avec la carte Nucleo, nous devons convertir le micrologiciel *ST-LINK* du débogueur intégré en un micrologiciel *jlink* qu'Ozone supporte. Suivez les instructions sur le site de Segger [13] :

- Lancez Ozone. Choisissez notre appareil dans l'assistant (**figure 2**).
- Sélectionnez un débogueur que nous allons utiliser - il s'agit d'un ST-LINK (**figure 3**).
- choisissez le fichier *firmware.elf* (**figure 4**).
- Laissez les valeurs par défaut sur la page suivante, cliquez sur *Finish*, et le débogueur sera chargé (notez que le code source *mcu.h* est récupéré), voir **figure 5**.
- Cliquez sur le bouton vert pour télécharger, exécutez le micrologiciel, et c'est fini. (**figure 6**).

Nous pouvons maintenant parcourir le code en une seule étape, définir des points d'arrêt et effectuer les opérations de débogage habituelles. Une chose à noter est la vue *Peripherals* d'Ozone (**figure 7**). En l'utilisant, nous pouvons directement examiner ou définir l'état des périphériques. Par exemple, allumons une LED verte intégrée (PB0) :

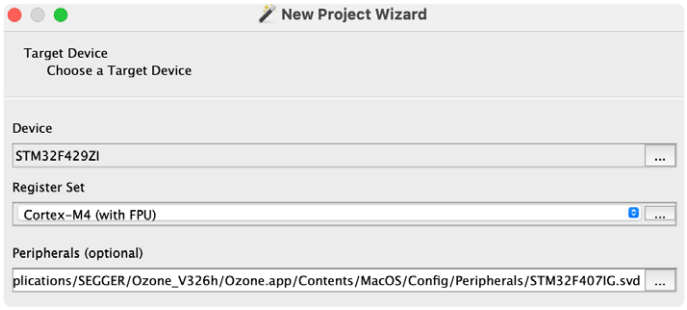

Figure 2. Sélectionnez l'appareil dans l'assistant.

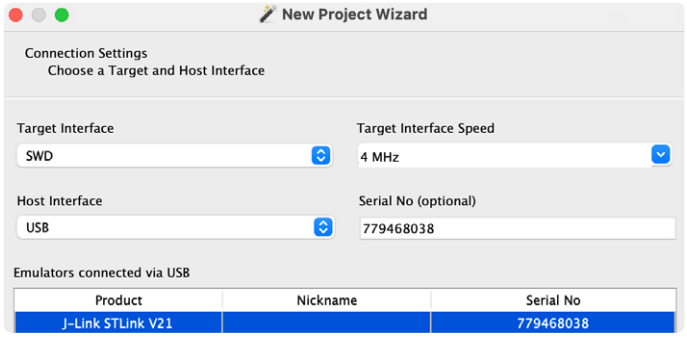

Figure 3. Sélectionnez STLink comme débogueur.

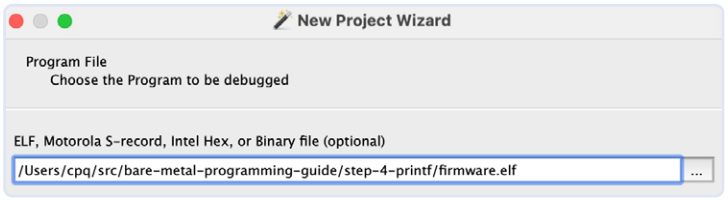

Figure 4. Le programme à déboguer est dans le fichier firmware.elf.

1. Faisons d'abord le pointage de la GPIOB. Allez sur *Peripherals RCC AHB1ENR*, et activez le bit *GPIOBEN* — mettez le à 1 (**figure 8**).
2. Allez sur *Peripherals GPIO GPIOB MODER*, mettez *MODERo* à 1 (sortie) (**figure 9**).
3. Allez sur *Peripherals GPIO GPIOB ODR*, mettez *ODRo* à 1 (on) (**figure 10**).

Maintenant, une LED verte devrait s'allumer ! Débogage réussi ! Dans la troisième partie de cette série, nous implémenterons un serveur web. En outre, nous montrerons comment un programme peut être testé automatiquement, et bien d'autres choses encore. Restez à l'écoute !

220665-B-04

À propos de l'auteur

Sergey Lyubka est ingénieur et entrepreneur. Il est titulaire d'un MSc 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 de réseaux embarqués bare-metal.

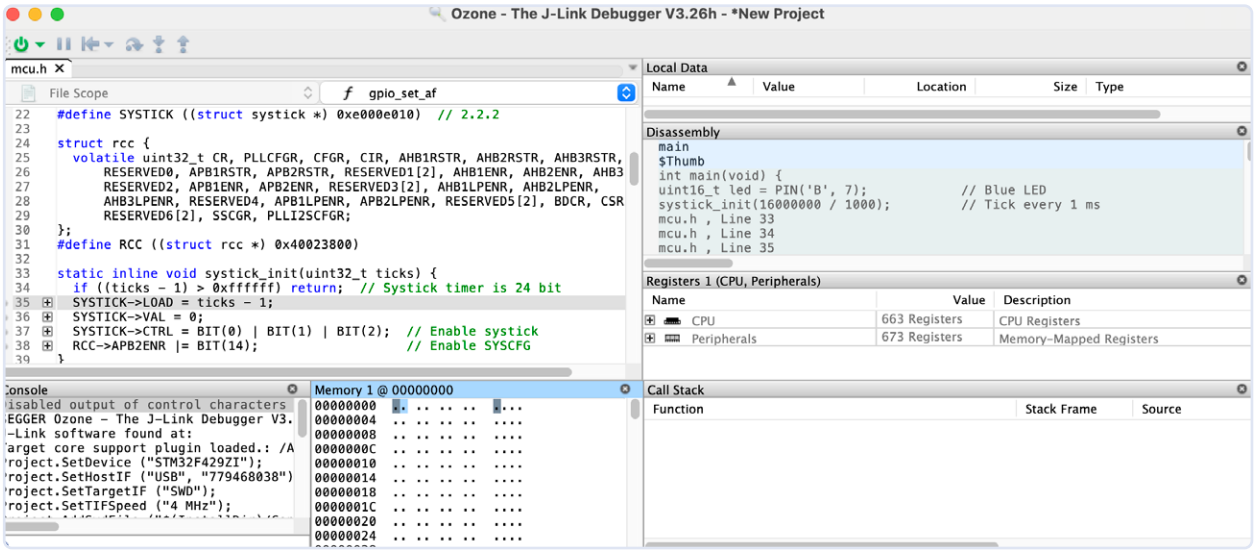

Figure 5. Le débogueur est chargé, et le fichier mcu.h apparaîtra bientôt.

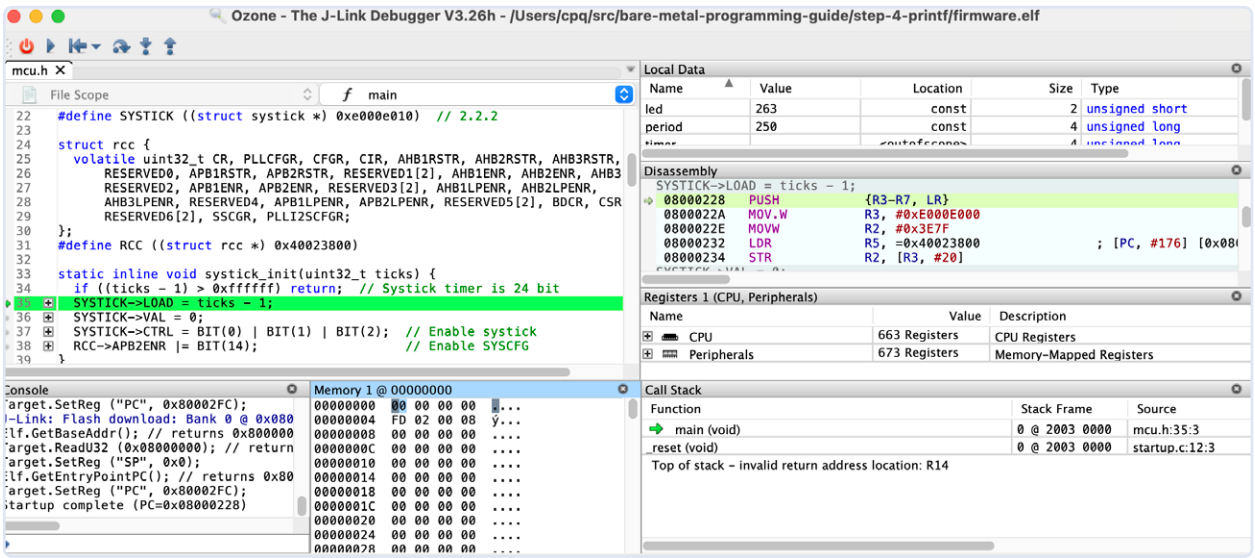

Figure 6. Après avoir exécuté le micrologiciel, il s'arrête à la ligne SYSTICK->LOAD = ticks - 1;.

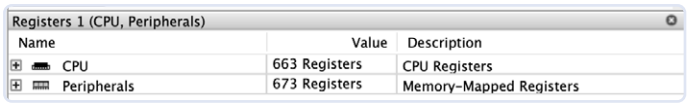

Figure 7. La vue Peripherals d'Ozone facilite l'examen et la configuration des périphériques.

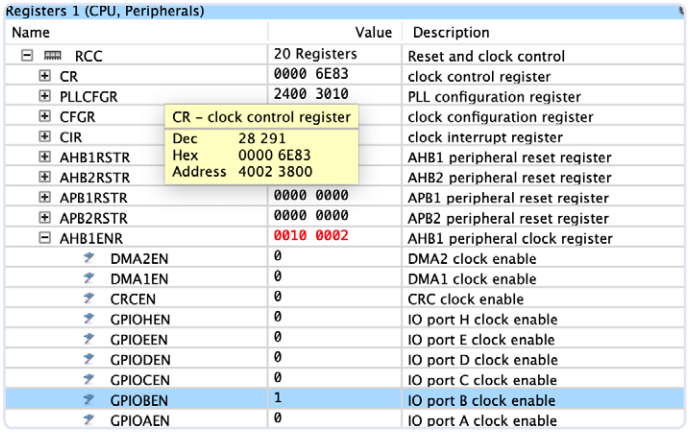

Figure 8. Activation de l'horloge sur le port B en mettant la valeur de GPIOBEN à 1.

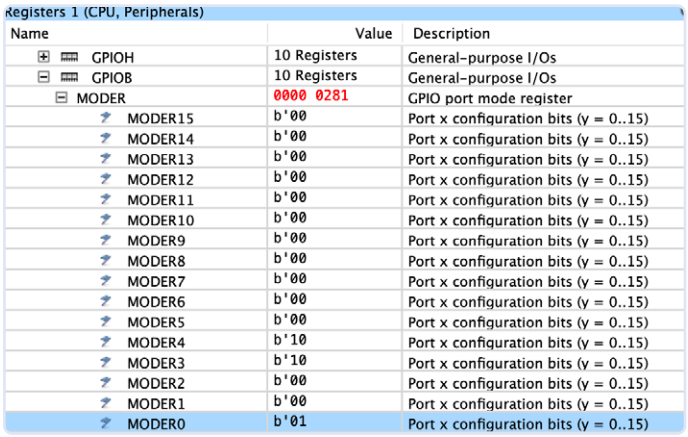

Figure 9. Mettez MODER0 à 1 (et sélectionnez donc la sortie) dans les périphériques GPIO.

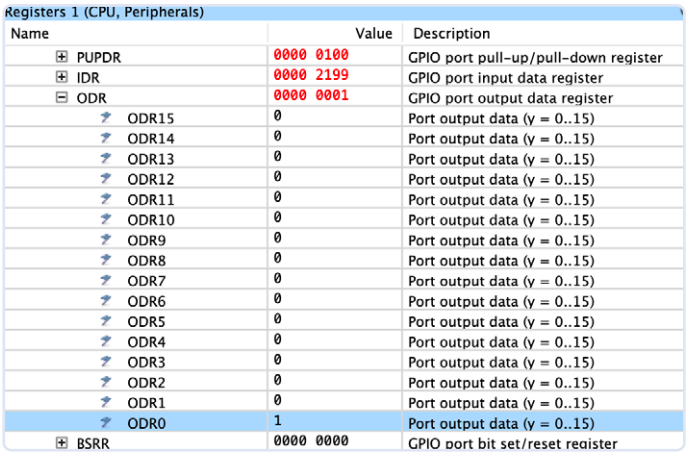

Figure 10. Activez ODR0 en sélectionnant la valeur 1 dans ODR (GPIO).

Questions ou commentaires ?

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

Produits

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

LIENS

[1] Dépôt GitHub pour cet article : https://github.com/cpq/bare-metal-programming-guide
[2] Sergey Lyubka, " Guide de programmation Bare-Metal (2) " Elektor 9-10/2023 : https://elektormagazine.fr/220665-04
[3] Manuel de référence de l'architecture Arm v7-M : https://developer.arm.com/documentation/ddi0403/ee
[4] Manuel de référence RM0090 pour STM32F429 : https://bit.ly/3neE7S7
[5] Step 2 SysTick folder: https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-2-systick
[6] Manuel d'utilisation de la carte Nucleo-144 (UM1974) : https://bit.ly/3oIBXKZ
[7] Fiche technique de la STM32F429ZI : https://st.com/resource/en/datasheet/stm32f429zi.pdf
[8] PuTTY : https://putty.org
[9] Dossier de l'étape 3 UART : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-3-uart
[10] Bibliothèque C newlib : https://sourceware.org/newlib
[11] Dossier de l'étape 4 printf : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-4-printf
[12] Ozone — Débogueur J-Link et analyseur de performances :
https://segger.com/products/development-tools/ozone-j-link-debugger
[13] Conversion du ST-LINK intégré en J-Link :
https://segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board