guide de programmation bare-metal (1)

pour STM32 et autres microcontrôleurs

Sergey Lyubka (Irlande)

Vous souhaitez programmer des microcontrôleurs et interagir avec leurs niveaux matériels pour mieux comprendre leur fonctionnement ? Ce guide destiné aux développeurs vous aidera à démarrer en utilisant simplement le compilateur GCC et un manuel de référence. Les notions apprises ici vous aideront à mieux comprendre le fonctionnement des cadres tels que Cube, Keil et Arduino. Dans ce guide en deux parties, nous utiliserons le contrôleur STM32F429 de la carte Nucleo-F429ZI, mais, vous pouvez facilement appliquer les connaissances acquises à d'autres microcontrôleurs.

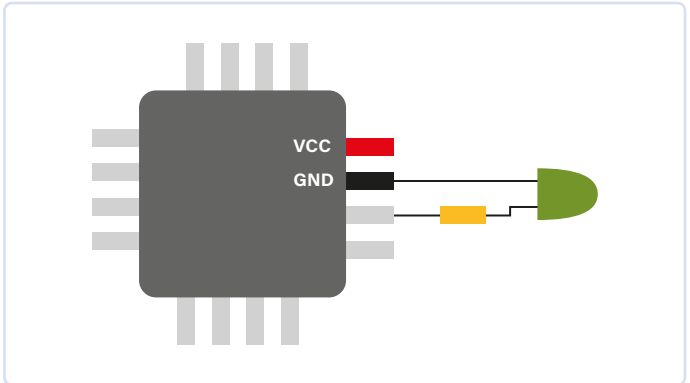

Figure 1. Le code du micrologiciel peut appliquer une tension de niveau haut ou bas sur une broche de signal, ce qui fait clignoter une LED.

Un microcontrôleur (µC, ou MCU) est un micro-ordinateur. Il est généralement doté d'un processeur, d'une mémoire vive, d'une mémoire flash pour stocker le code et d'un ensemble de broches. Certaines broches sont utilisées pour l'alimentation du microcontrôleur, généralement désignées par GND (masse) et VCC. D'autres broches sont utilisées pour communiquer avec le microcontrôleur en appliquant une tension de niveau haut ou bas à ces broches. L'un des exemples de communication les plus simples est de relier une LED à une broche : l'une des bornes de la LED est reliée à la masse (GND) et l'autre est reliée à une broche de signal avec une résistance de limitation de courant en série. Le micrologiciel peut appliquer une tension de haut ou bas niveau sur une broche de signal, ce qui fait clignoter la LED (**figure 1**).

Matériel et outils nécessaires

Tout au long de ce guide, nous utiliserons une carte de développement Nucleo-F429ZI (disponible chez Mouser et d'autres distributeurs). Pour suivre ce tutoriel, téléchargez le manuel de référence du MCU STM32F429 [1] puis le manuel d'utilisation de la carte de développement [2].

Pour démarrer, les outils suivants sont nécessaires :

ARM GCC, *https://launchpad.net/gcc-arm-embedded* - pour la compilation et l'édition des liens

GNU make, *https://gnu.org/software/make* - pour la construction automatisée

ST link, *https://github.com/stlink-org/stlink* - pour le flashage

Nous présentons ici les instructions d'installation pour Linux (Ubuntu).

Lancez un terminal, puis exécutez :

```
$ sudo apt -y update
$ sudo apt -y install gcc-arm-none-eabi make
stlink-tools
```

Pour configurer les outils sur un Mac ou un PC Windows, voir [3].

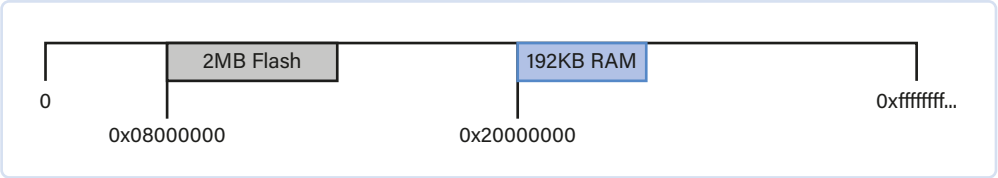

Figure 2. Emplacements des zones de la mémoire flash et de la mémoire vive du STM32F429.

Mémoire et registres

L'espace d'adressage d'un microcontrôleur à 32 bits, par exemple le STM32F429 de STMicroelectronics, est divisé en zones. Par exemple, une zone mémoire (à une adresse spécifique) est attribuée à la mémoire flash interne du microcontrôleur. Les instructions du micrologiciel sont lues et exécutées en lisant dans cette zone de mémoire. Une autre zone est la RAM, qui est également attribuée à une adresse spécifique. Nous pouvons lire et écrire n'importe quelle valeur dans la région RAM.

La section 2.3.1 du manuel de référence du STM32F429 [1] explique que la zone RAM commence à l'adresse 0x20000000 et a une taille de 192 KB. La section 2.4 nous indique que la mémoire flash commence à l'adresse 0x08000000. Notre microcontrôleur dispose de 2 Mo de mémoire flash, donc les zones flash et RAM sont réparties comme montré dans la **figure 2**.

Le manuel nous indique également qu'il existe de nombreuses autres zones mémoires. Leurs plages d'adresses sont indiquées dans la section 2.3 « Memory map ». Par exemple, il existe une zone « GPIOA » qui commence à 0x40020000 et qui a une taille de 1 Ko. Ces zones mémoires correspondent à différents *périphériques* dans le microcontrôleur – un circuit en silicium qui permet à certaines broches de se comporter d'une manière particulière. Une zone mémoire périphérique est un ensemble de registres de 32 bits. Chaque registre représente une plage de mémoire de 4 octets située à une certaine adresse, qui correspond à une certaine fonction du périphérique en question. En écrivant des valeurs dans un registre – en d'autres termes, en écrivant une valeur de 32 bits à une certaine adresse mémoire – nous pouvons contrôler le comportement d'un périphérique donné. En consultant ces registres, nous pouvons relire leurs contenus ou leurs configurations.

Il existe de nombreux périphériques différents. Parmi les plus simples, on trouve les ports GPIO (*General Purpose Input Output*), qui permettent à l'utilisateur de configurer les broches du microcontrôleur en mode « sortie » et de leur appliquer un niveau haut ou bas. Il est également possible de les configurer en mode « entrée » et de lire leurs tensions. Il existe un périphérique UART qui peut transmettre et recevoir des données série sur deux broches grâce à un protocole série. Il existe de nombreux autres périphériques. Un périphérique existe souvent sous plusieurs « versions », par exemple *GPIOA*, *GPIOB*, etc., qui contrôlent différents ensembles de broches du microcontrôleur. De même, on trouve *UART1*, *UART2*, etc., qui permettent l'implémentation de plusieurs canaux UART. Sur le STM32F429, il y a plusieurs périphériques GPIO et UART.

Par exemple, le périphérique *GPIOA* commence à l'adresse 0x40020000. Les registres GPIO sont décrits dans la section 8.4 [1]. Le manuel de référence indique que le registre `GPIOA_MODER` a un *offset* de 0, ce qui signifie que son adresse est `0x40020000 + 0`. Le format du registre est illustré dans la **figure 3**.

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

pin 15 … pin 8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

pin 7 … pin 3 … pin 2 … pin 1 … pin 0 (A0, B0, …)

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figure 3. La description des registres GPIO est disponible dans le manuel de référence. Un registre MODER contrôle 16 broches physiques. (Source : [1])

Le manuel indique que le registre *MODER* de 32 bits est une série de valeurs de 2 bits, 16 au total. Par conséquent, un registre *MODER* contrôle 16 broches matérielles, les bits 0…1 contrôlent la broche 0, les bits 2…3 contrôlent la broche 1, et ainsi de suite. La valeur de 2 bits permet de coder le mode de fonctionnement de la broche : 0 signifie entrée, 1 signifie sortie, 2 signifie « fonction alternative » (un comportement spécifique décrit ailleurs), et 3 signifie analogique. Le nom du périphérique étant *GPIOA*, les broches sont nommées « A0 », « A1 », etc. Pour le périphérique *GPIOB*, le nom des broches serait « B0 », « B1 »…
Si nous écrivons la valeur 0 dans le registre *MODER* de 32 bits, nous mettrons les 16 broches, de A0 à A15, en mode entrée :

```
* (volatile uint32_t *) (0x40020000 + 0) = 0;
  // Set A0...A15 to input mode
```

Nous aborderons la signification du qualificateur `volatile` ultérieurement. En définissant des bits spécifiques, nous pouvons configurer des broches spécifiques dans le mode souhaité. Par exemple, cet extrait configure la broche A3 en sortie :

```
* (volatile uint32_t *) (0x40020000 + 0) &= ~(3 << 6);
 // Clear bit range 6...7
* (volatile uint32_t *) (0x40020000 + 0) |= 1 << 6;
 // Set bit range 6...7 to 1
```

Expliquons ces opérations. Notre objectif est de mettre les bits 6…7, qui sont responsables de la broche 3 du périphérique *GPIOA*, à une valeur spécifique (1, dans notre cas). Cela se fait en deux étapes. Tout d'abord, nous devons effacer le contenu actuel des bits 6…7, car ils peuvent déjà contenir une certaine valeur. Ensuite, nous devons définir les bits appropriés pour obtenir la valeur que nous voulons. Nous devons donc commencer par mettre à 0 la plage de bits 6…7 (deux bits en position 6). Comment mettre à 0 un certain nombre de bits ? Les quatre étapes sont décrites dans le **tableau 1**.
Notez que la dernière opération fait passer N bits à la position X à 0 (combinés par ET logique avec 0), mais conserve les valeurs de tous les autres bits (combinés par ET avec 1). Il est important de conserver la valeur existante, car nous ne voulons pas modifier les paramètres dans d'autres plages de bits. En général, si nous voulons effacer N bits à la position X, nous pouvons écrire ce qui suit :

```
REGISTER &= ~((2^N - 1) << X);
```

Enfin, nous voulons attribuer une valeur spécifique à une plage de bits donnée. Nous décalons cette valeur de X positions vers la gauche, et nous la combinons par OU logique avec la valeur actuelle du registre (afin de conserver les valeurs des autres bits) :

```
REGISTER |= VALUE << X;
```

Tableau 1. Mise à zéro de certains bits.

Action	Expression	Bits (12 premiers bits sur 32)
Choisir un nombre avec N ensembles de bits contigus : 2^N-1, ici N = 2	`3`	`000000000011`
Décaler ce nombre de X positions vers la gauche	`(3<<6)`	`000011000000`
Inverser le nombre : transformer les zéros en uns et les uns en zéros	`~(3<<6)`	`111100111111`
ET logique avec la valeur existante	`VAL &= ~(3<<6)`	`xxxx00xxxxxx`

Programmation de périphériques aisée

Dans le paragraphe précédent, nous avons appris qu'il est possible d'écrire et lire un registre périphérique en accédant directement à certaines adresses de la mémoire. Examinons l'extrait qui permet de configurer la broche A3 en sortie :

```
* (volatile uint32_t *) (0x40020000 + 0) &= ~(3 << 6);
 // Clear bit range 6...7
* (volatile uint32_t *) (0x40020000 + 0) |= 1 << 6;
 // Set bit range 6...7 to 1
```

Sans commentaires détaillés, un tel code serait assez difficile à comprendre. Nous pouvons réécrire ce code et le rendre beaucoup plus lisible. L'idée est de représenter le périphérique sous la forme d'une structure contenant des champs de 32 bits. Les registres disponibles pour le périphérique GPIO sont décrits dans la section 8.4 du manuel de référence. Il s'agit de *MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, BSRR, LCKR, AFR*. Leurs décalages (offsets) sont 0, 4, 8, etc. Cela signifie que nous pouvons les représenter sous la forme d'une structure avec des champs de 32 bits, et créer un `#define` pour *GPIOA* :

```
struct gpio {
 volatile uint32_t MODER, OTYPER, OSPEEDR,
   PUPDR, IDR, ODR, BSRR, LCKR, AFR[2];
};
#define GPIOA ((struct gpio *) 0x40020000)
```

Ensuite, pour définir le mode de la broche GPIO, nous pouvons définir une fonction :

```
// Enum values are per reference manual: 0, 1, 2, 3
enum {GPIO_MODE_INPUT, GPIO_MODE_OUTPUT,
   GPIO_MODE_AF, GPIO_MODE_ANALOG};

static inline void gpio_set_mode
 (struct gpio *gpio, uint8_t pin, uint8_t mode) {
 gpio->MODER &= ~(3U << (pin * 2));
 // Clear existing setting
 gpio->MODER |= (mode & 3) << (pin * 2);
 // Set new mode
}
```

Nous pouvons réécrire le code pour A3 comme suit :

```
gpio_set_mode(GPIOA, 3 /* pin */, GPIO_MODE_OUTPUT);
 // Set A3 to output
```

Le microcontrôleur possède plusieurs périphériques GPIO (aussi appelés *banques*) : A, B, C, … K. D'après la section 2.3, ils sont espacés de 1 Ko : *GPIOA* est à l'adresse 0x40020000, *GPIOB* à 0x40020400, et ainsi de suite :

```
#define GPIO(bank) ((struct gpio *)
 (0x40020000 + 0x400 * (bank)))
```

Nous pouvons définir une numérotation qui inclut la banque et le numéro de la broche. Pour cela, nous utilisons une valeur `uint16_t` de 2 octets, où l'octet de poids fort désigne la banque GPIO, et l'octet de poids faible désigne le numéro de la broche :

```
#define PIN(bank, num) ((((bank) - 'A') << 8) | (num))
#define PINNO(pin) (pin & 255)
#define PINBANK(pin) (pin >> 8)
```

Ainsi, nous pouvons spécifier des broches pour n'importe quelle banque GPIO :

```
uint16_t pin1 = PIN('A', 3); // A3 - GPIOA pin 3
uint16_t pin2 = PIN('G', 11); // G11 - GPIOG pin 11
```

Réécrivons la fonction `gpio_set_mode()` pour qu'elle reçoive la configuration de la broche :

```
static inline void gpio_set_mode(uint16_t pin, uint8_t
mode) {
 struct gpio *gpio = GPIO(PINBANK(pin));
   // GPIO bank
 uint8_t n = PINNO(pin); // Pin number
 gpio->MODER &= ~(3U << (n * 2));
   // Clear existing setting
 gpio->MODER |= (mode & 3) << (n * 2);
   // Set new mode
}
```

Le code pour A3 est simple :

```
uint16_t pin = PIN('A', 3); // Pin A3
gpio_set_mode(pin, GPIO_MODE_OUTPUT); // Set to output
```

Notez que nous avons créé une API initiale utile pour le périphérique GPIO. D'autres périphériques, tels que UART (communication série) et autres, peuvent être configurés de la même manière. Il s'agit d'une bonne pratique de programmation qui rend le code clair et lisible par l'utilisateur.

Démarrage du microcontrôleur et table vectorielle

Lorsqu'un microcontrôleur ARM démarre, il lit une « table vectorielle » située au début de la mémoire flash. Une table vectorielle est un élément commun à tous les microcontrôleurs ARM : Il s'agit d'un tableau d'adresses 32 bits de gestionnaires d'interruptions. Les 16 premières entrées sont réservées par ARM et sont communes à tous les microcontrôleurs ARM. Le reste des gestionnaires d'interruptions est spécifique au microcontrôleur en question – il s'agit de gestionnaires d'interruptions pour les périphériques. Les microcontrôleurs plus simples avec peu de périphériques ont peu de gestionnaires d'interruption, et ceux qui sont plus complexes en ont plusieurs.

La table vectorielle du STM32F429 est documentée dans le tableau 62 du manuel de référence [1]. On y apprend qu'il existe 91 gestionnaires de périphériques, en plus des 16 standards.

Chaque entrée de la table vectorielle est une adresse d'une fonction que le microcontrôleur exécute lorsqu'une interruption matérielle (IRQ) est déclenchée. Les deux premières entrées, qui jouent un rôle clé dans le processus de démarrage du microcontrôleur, font exception. Ces deux premières valeurs sont un pointeur de pile initial et l'adresse de la fonction de démarrage (un point d'entrée du micrologiciel à exécuter).

Dans le code, nous devons donc veiller à ce que la deuxième valeur de 32 bits dans la mémoire flash contienne l'adresse de notre fonction de démarrage. Au démarrage, le microcontrôleur lira cette adresse dans la mémoire flash et passera à la fonction de démarrage.

Micrologiciel minimal

Créons un fichier, *main.c*, et spécifions notre fonction de démarrage, qui ne fait rien au départ (commence par une boucle infinie), et spécifions également une table vectorielle qui contient 16 entrées standard et 91 entrées STM32. Dans l'éditeur de votre choix, créez et ouvrez le fichier *main.c* et entrez ce qui suit :

```
// Startup code
__attribute__((naked, noreturn)) void _reset(void) {
  for (;;) (void) 0; // Infinite loop
}

extern void _estack(void); // Defined in link.ld

// 16 standard and 91 STM32-specific handlers
__attribute__((section(".vectors")))
      void (*tab[16 + 91])(void) = {_estack, _reset};
```

Pour la fonction `_reset()`, nous avons utilisé les attributs `naked` et `noreturn` spécifiques au compilateur GCC – ils signifient que le prologue et l'épilogue de la fonction standard ne doivent pas être créés par le compilateur et que la fonction ne retourne rien.

L'expression `void (*tab[16 + 91])(void)` signifie : définir un tableau de 16 + 91 pointeurs vers des fonctions qui ne renvoient rien `(void)`, et reçoivent deux arguments. Chacune de ces fonctions est un *IRQ handler* (*Interrupt ReQuest handler*). Un tableau de ces gestionnaires est appelé une table vectorielle.

La table vectorielle `tab` est placée dans une section séparée appelée `.vectors` – nous en aurons besoin plus tard pour indiquer à l'éditeur de liens de placer cette section au début du micrologiciel généré, consécutivement, au début de la mémoire flash. Les deux premières entrées sont la valeur du registre du pointeur de pile et le point

d'entrée du micrologiciel. Nous laissons le reste du tableau vectoriel rempli de zéros.

Compilation

Compilons notre code. Démarrez un terminal (ou l'invite de commande sous Windows) et exécutez :

```
$ arm-none-eabi-gcc -mcpu=cortex-m4 main.c -c
```

Cela fonctionne ! La compilation a produit un fichier, *main.o*, qui contient notre micrologiciel minimal qui ne fait rien. Le fichier *main.o* est au format binaire *ELF*, qui contient plusieurs sections (voir **listage 1**).
Notez que les adresses VMA/LMA des sections sont fixées à 0, ce qui signifie que le fichier *main.o* est incomplet, car il ne contient pas d'informations sur l'emplacement de ces sections dans l'espace d'adressage. Nous devons utiliser un éditeur de liens pour produire le fichier du micrologiciel complet, *firmware.elf*, à partir de *main.o*.
La section `.text` contient le code du micrologiciel, qui dans notre cas consiste simplement en une fonction `_reset()`, de deux octets – une instruction de saut à sa propre adresse. Il y a deux sections `.data` et `.bss` vides [8] (pour les variables non initialisées, mais déclarées, cette section est généralement remplie avec des 0).
Notre micrologiciel sera copié dans la mémoire flash à la position 0x8000000, mais notre section de données doit être dans la RAM – par conséquent, notre fonction `_reset()` doit copier le contenu de la section `.data` vers la RAM. Elle doit également écrire des zéros dans toute la section `.bss`. Dans notre cas, les sections `.data` et `.bss` sont vides, mais modifions tout de même la fonction `_reset()` pour les gérer correctement.
Pour ce faire, nous devons savoir où commence la pile et où commencent les sections data et bss. Nous pouvons le spécifier dans le script *linker*, qui est un fichier contenant les instructions de l'éditeur de liens concernant l'emplacement des différentes sections dans l'espace d'adressage et sur les symboles à créer.

Script *Linker*

Créez un fichier appelé `link.ld`, et copiez-collez le contenu de [4]. L'explication est donnée ci-dessous :

```
ENTRY(_reset);
```

Cette ligne indique à l'éditeur de liens la valeur de l'attribut « entry point » dans l'en-tête ELF généré – il s'agit donc d'une copie de ce que contient une table vectorielle. Il s'agit d'une aide pour débogueur (tel qu'Ozone, décrite dans la deuxième partie de ce guide) qui nous permet de placer un point d'arrêt au début du micrologiciel. Un débogueur ne connaît pas la table vectorielle, il se fie donc à l'en-tête ELF.

```
MEMORY {
flash(rx) : ORIGIN = 0x08000000, LENGTH = 2048k
sram(rwx) : ORIGIN = 0x20000000, LENGTH = 192k /*
remaining 64k in a separate address space */
}
```

Cela indique à l'éditeur de liens que nous avons deux zones mémoires dans l'espace d'adressage, ainsi que leurs adresses et leurs tailles.

```
_estack = ORIGIN(sram) + LENGTH(sram); /* stack points
to end of SRAM */
```

Cela indique à l'éditeur de liens de créer un symbole, `_estack`, contenant une valeur à la fin de la zone RAM. Ce sera la valeur initiale de notre pile !

```
.vectors : { KEEP(*(.vectors)) } > flash
.text : { *(.text*) } > flash
.rodata : { *(.rodata*) } > flash
```

Listage 1. Compilation du fichier main.o

```
$ arm-none-eabi-objdump -h main.o
 ...
 Idx Name          Size      VMA       LMA       File off  Algn
   0 .text         00000002  00000000  00000000  00000034  2**1
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
   1 .data         00000000  00000000  00000000  00000036  2**0
                   CONTENTS, ALLOC, LOAD, DATA
   2 .bss          00000000  00000000  00000000  00000036  2**0
                   ALLOC
   3 .vectors      000001ac  00000000  00000000  00000038  2**2
                   CONTENTS, ALLOC, LOAD, RELOC, DATA
 ...
```

Listage 2. Code de démarrage.

```
int main(void) {
   return 0; // Do nothing so far
}

// Startup code
__attribute__((naked, noreturn)) void _reset(void) {
   // memset .bss to zero, and copy .data section to RAM region
   extern long _sbss, _ebss, _sdata, _edata, _sidata;
   for (long *src = &_sbss; src < &_ebss; src++) *src = 0;
   for (long *src = &_sdata, *dst = &_sidata; src < &_edata;) *src++ = *dst++;

   main();             // Call main()
   for (;;) (void) 0;  // Infinite loop in the case if main() returns
 }
```

Ces lignes indiquent à l'éditeur de liens de placer la table vectorielle dans la mémoire flash en premier, suivie de la section `.text` (code du micrologiciel), puis des données en lecture seule `.rodata`. Vient ensuite la section `.data` :

```
.data : {
 _sdata = .; /* .data section start */
 *(.first_data)
 *(.data SORT(.data.*))
 _edata = .; /* .data section end */
} > sram AT > flash
_sidata = LOADADDR(.data);
```

Notez que nous demandons à l'éditeur de liens de créer les symboles `_sdata` et `_edata`. Nous les utiliserons pour copier la section de données en RAM dans la fonction `_reset()`.
Idem pour la section `.bss` :

```
.bss : {
 _sbss = .; /* .bss section start */
 *(.bss SORT(.bss.*) COMMON)
 _ebss = .; /* .bss section end */
} > sram
```

Code de démarrage

Nous pouvons maintenant mettre à jour la fonction `_reset()`. Nous copions la section `.data` dans la RAM, et initialisons la section `.bss` avec des zéros. Ensuite, nous appelons la fonction `main()` - et entrons dans une boucle infinie lorsque `main()` s'arrête (voir **listage 2**).
Le diagramme de la **figure 4** illustre comment `_reset()` initialise `.data` et `.bss`.
Le fichier *firmware.bin* n'est qu'une concaténation des trois sections : `.vectors` (table vectorielles IRQ), `.text` (code) et `.data` (données). Ces sections ont été créées conformément au script de l'éditeur de liens : `.vectors` se situe au début de la mémoire flash suivi par `.text` et `.data` se situe bien au-dessus. Les adresses dans `.text` se trouvent dans la mémoire flash, et les adresses dans `.data` se trouvent dans la mémoire vive. Si une fonction a une adresse, par exemple `0x8000100`, elle se trouve exactement à cette adresse dans la mémoire flash. Mais si le code accède à une variable de la section `.data` par son adresse, par exemple `0x20000200`, il n'y a rien à cette adresse, car, au démarrage, la section `.data` du fichier *firmware.bin* réside dans la mémoire flash ! C'est pourquoi le code de démarrage doit déplacer la section `.data` de la mémoire flash vers la RAM.
Nous sommes maintenant prêts à produire le fichier du micrologiciel complet, *firmware.elf* :

```
$ arm-none-eabi-gcc -T link.ld -nostdlib main.o -o
firmware.elf
```

Examinons les sections du fichier *firmware.elf* – voir **listage 3**.
La section `.vectors` se trouve au tout début de la mémoire flash, à l'adresse 0x8000000, suivie par la section `.text` à l'adresse 0x80001ac. Notre code ne crée pas de variables, il n'y a donc pas de section de données.

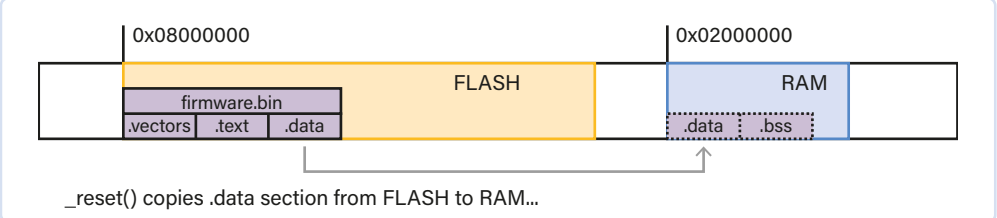

Figure 4. Le diagramme montre comment `_reset()` initialise `.data` et `.bss`.

Listage 3. Extrait du fichier firmware.elf

```
$ arm-none-eabi-objdump -h firmware.elf
 ...
 Idx Name          Size      VMA       LMA       File off  Algn
   0 .vectors      000001ac  08000000  08000000  00010000  2**2
                   CONTENTS, ALLOC, LOAD, DATA
   1 .text         00000058  080001ac  080001ac  000101ac  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 ...
```

Flasher le micrologiciel

Nous pouvons maintenant flasher le micrologiciel ! Tout d'abord, extrayez les sections du fichier *firmware.elf* en un seul bloc binaire contigu :

```
$ arm-none-eabi-objcopy -O binary firmware.elf firmware.bin
```

Utilisons ensuite l'utilitaire *st-link* pour flasher le *firmware.bin*. Branchez votre carte sur le port USB et exécutez :

```
$ st-flash --reset write firmware.bin 0x8000000
```

C'est fait ! Nous avons flashé un micrologiciel qui ne fait rien.

Makefile : automatisation de la construction

Au lieu de taper ces commandes de compilation, d'édition de liens et de flashage, nous pouvons utiliser l'outil de ligne de commande `make` pour automatiser l'ensemble du processus. L'utilitaire `make` utilise un fichier de configuration appelé *Makefile*, dans lequel il lit les instructions d'exécution. Cette automatisation est très utile, car elle permet également de documenter le processus de création du micrologiciel, les drapeaux de compilation utilisés, etc.
Il existe un excellent tutoriel sur *Makefile* [5]. Je le recommande à ceux qui veulent s'initier à `make`. Ci-dessous, j'énumère les concepts les plus essentiels nécessaires à la compréhension de notre *Makefile bare-metal* simple. Ceux qui sont déjà familiers avec `make` peuvent sauter cette section.
Le format du fichier *Makefile* est simple :

```
action1:
  command ... # Comments can go after hash symbol
  command .... # IMPORTANT: command must be preceded with
the TAB character
action2:
  command ... # Don't forget about TAB. Spaces won't work!
```

Nous pouvons utiliser `make` avec le nom de l'action (également appelé *target*) pour exécuter l'action correspondante :

```
$ make action1
```

Il est possible de définir des variables et de les utiliser dans les commandes. Les actions peuvent également correspondre aux noms des fichiers qui doivent être créés :

```
firmware.elf:
  COMPILATION COMMAND .....
```

De plus, toute action peut avoir une liste de dépendances. Par exemple, *firmware.elf* dépend de notre fichier source, *main.c*. Chaque fois que le fichier *main.c* est modifié, la commande `make build` recrée *firmware.elf* :

```
build: firmware.elf
firmware.elf: main.c
  COMPILATION COMMAND
```

Nous pouvons maintenant créer un Makefile pour notre micrologiciel. Nous définissons une action `build`/*target* :

```
CFLAGS ?= -W -Wall -Wextra -Werror -Wundef -Wshadow
-Wdouble-promotion \ -Wformat-truncation -fno-common
-Wconversion \ -g3 -Os -ffunction-sections -fdata-
sections -I. \ -mcpu=cortex-m4 -mthumb -mfloat-abi=hard
-mfpu=fpv4-sp-d16 $(EXTRA_CFLAGS)
LDFLAGS ?= -Tlink.ld -nostartfiles -nostdlib --specs nano.
specs -lc -lgcc -Wl,--gc-sections -Wl,-Map=$@.map
SOURCES = main.c
build: firmware.elf
firmware.elf: $(SOURCES)
  arm-none-eabi-gcc $(SOURCES) $(CFLAGS)
  $(LDFLAGS) -o $@
```

Nous y définissons les drapeaux de compilation. `?=` représente une valeur par défaut ; nous pouvons les remplacer à partir de la ligne de commande :

```
$ make build CFLAGS="-O2 ...."
```

Nous définissons les variables `CFLAGS`, `LDFLAGS`, et `SOURCES`. Puis nous indiquons à `make` que si une instruction *build* est reçue, un fichier *firmware.elf* doit être créé. Ce dernier dépend du fichier *main.c*, et pour le créer, `make` doit lancer le compilateur *arm-none-eabi-gcc* avec les drapeaux donnés. La variable spéciale `$@` se développe en un nom de cible – dans notre cas, *firmware.elf*.
Appelons `make` :

```
$ make build arm-none-eabi-gcc main.c -W -Wall -Wextra
-Werror -Wundef -Wshadow -Wdouble-promotion -Wformat-
truncation -fno-common -Wconversion -g3 -Os -ffunction-
sections -fdata-sections -I. -mcpu=cortex-m4 -mthumb
-mfloat-abi=hard -mfpu=fpv4-sp-d16 -Tlink.ld -nostartfiles
-nostdlib --specs nano.specs -lc -lgcc -Wl,--gc-sections
-Wl,-Map=firmware.elf.map -o firmware.elf
```

Listage 4. Extrait du fichier main.c du projet Blinky LED.

```
#include <inttypes.h>
#include <stdbool.h>
#define BIT(x) (1UL << (x))
#define PIN(bank, num) ((((bank) - 'A') << 8) | (num))
#define PINNO(pin) (pin & 255)
#define PINBANK(pin) (pin >> 8)

struct gpio {
   volatile uint32_t MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, BSRR, LCKR, AFR[2];
};
#define GPIO(bank) ((struct gpio *) (0x40020000 + 0x400 * (bank)))

// Enum values are per datasheet: 0, 1, 2, 3
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) {
   struct gpio *gpio = GPIO(PINBANK(pin));  // GPIO bank
   int n = PINNO(pin);                      // Pin number
   gpio->MODER &= ~(3U << (n * 2));         // Clear existing setting
   gpio->MODER |= (mode & 3) << (n * 2);    // Set new mode
}
```

Si nous l'exécutons à nouveau :

```
$ make build
make: Nothing to be done for 'build'.
```

L'utilitaire make examine les temps de modification de la dépendance *main.c* et *firmware.elf* – et ne fait rien si *firmware.elf* est à jour. Mais si nous modifions *main.c*, le prochain `make build` recompilera :

```
$ touch main.c # Simulate changes in main.c
$ make build
```

Il ne reste plus que la cible `flash` :

```
firmware.bin: firmware.elf
  $(DOCKER) $(CROSS)-objcopy -O binary $< $@ flash:
firmware.bin
  st-flash --reset write $(TARGET).bin 0x8000000
```

Voilà, c'est fait ! Maintenant, la dernière commande `make flash` crée un fichier *firmware.bin*, et le flashe sur la carte. Elle recompilera le micrologiciel si le fichier *main.c* change, parce que *firmware.bin* dépend de *firmware.elf*, et celui-ci, à son tour, dépend de *main.c*. Donc, le cycle de développement consisterait en ces deux actions en boucle :

```
# Develop code in main.c
$ make flash
```

Il est conseillé d'ajouter une cible `clean` pour supprimer les artefacts de `build` :

```
clean:
  rm -rf firmware.*
```

Le code source complet du projet se trouve dans le dossier minimal de *Step 0* [6].

LED clignotante

Maintenant que nous avons configuré l'infrastructure build / flash, il est temps que notre firmware apprenne à effectuer une tâche utile, par exemple faire clignoter une LED. La carte Nucleo-F429ZI possède trois LED intégrées. Dans la section 6.5 du manuel d'utilisation de la carte Nucleo [2], nous pouvons savoir à quelles broches sont connectées les LED intégrées :

- PB0 : LED verte
- PB7 : LED bleue
- PB14 : LED rouge

Modifions le fichier *main.c* et ajoutons nos définitions pour PIN, `gpio_set_mode()`. Dans la fonction `main()`, nous configurons la LED bleue en sortie, et nous lançons une boucle infinie. Tout d'abord, copions les définitions des broches et des GPIO dont nous avons parlé précédemment. Notez que nous ajoutons également une macro de commodité, `BIT(x)` (voir **listage 4**).
Lorsque certains microcontrôleurs sont mis sous tension, tous leurs périphériques sont alimentés et activés automatiquement. Les périphériques des microcontrôleurs STM32 restent désactivés par défaut afin d'économiser de l'énergie. L'activation d'un périphérique GPIO doit se faire via l'unité RCC (*Reset and Clock Control*). Dans la section 7.3.10 du manuel de référence, nous constatons que le registre *AHB1ENR* (*AHB1 peripheral clock enable register*) est responsable de l'activation ou de la désactivation des banques GPIO. Tout d'abord, nous ajoutons une définition pour toute l'unité RCC :

```
struct rcc {
  volatile uint32_t CR, PLLCFGR, CFGR, CIR, AHB1RSTR,
AHB2RSTR, AHB3RSTR, RESERVED0, APB1RSTR, APB2RSTR,
RESERVED1[2], AHB1ENR, AHB2ENR, AHB3ENR, RESERVED2,
APB1ENR, APB2ENR, RESERVED3[2], AHB1LPENR, AHB2LPENR,
AHB3LPENR, RESERVED4, APB1LPENR, APB2LPENR, RESERVED5[2],
BDCR, CSR, RESERVED6[2], SSCGR, PLLI2SCFGR;
};
#define RCC ((struct rcc *) 0x40023800)
```

Selon la documentation du registre *AHB1ENR*, les bits 0 à 8 inclus configurent l'horloge pour les banques GPIO GPIOA-GPIOE :

```
int main(void) {
  uint16_t led = PIN('B', 7); // Blue LED
  RCC->AHB1ENR |= BIT(PINBANK(led));
  // Enable GPIO clock for LED
  gpio_set_mode(led, GPIO_MODE_OUTPUT);
  // Set blue LED to output mode
  for (;;) asm volatile("nop"); // Infinite loop
  return 0;
}
```

Il ne reste plus qu'à découvrir comment activer ou désactiver une broche GPIO, puis à modifier la boucle principale pour activer une broche de LED, ajouter un délai, désactiver puis ajouter un délai. Selon la section 8.4.7 du manuel de référence, le registre *BSRR* est responsable de la mise de la tension au nivau haut ou bas. Les 16 bits de poids faible sont utilisés pour mettre le registre *ODR* (i.e. mettre la broche à l'état haut), et les 16 bits de poids fort sont utilisés pour réinitialiser le registre *ODR* (i.e. mettre la broche à l'état bas). Définissons une fonction API pour cet effet :

```
static inline void gpio_write(uint16_t pin, bool val) {
  struct gpio *gpio = GPIO(PINBANK(pin));
  gpio->BSRR |= (1U << PINNO(pin)) << (val ? 0 : 16);
}
```

Ensuite, nous devons implémenter une fonction de délai. Nous n'avons pas besoin d'un délai précis pour l'instant, alors définissons une fonction, `spin()`, qui exécute simplement une instruction NOP un certain nombre de fois :

```
static inline void spin(volatile uint32_t count) {
  while (count--) asm("nop");
}
```

Enfin, nous pouvons modifier notre boucle *main* pour faire clignoter une LED :

```
for (;;) {
  gpio_write(pin, true);
  spin(999999);
  gpio_write(pin, false);
  spin(999999);
}
```

Le code source complet du projet est disponible dans le dossier blinky de l'étape 1 [7]. Lancez `make flash` et admirez la LED bleue qui clignote !
Dans la deuxième partie de cet article, nous aborderons la sortie UART, le débogage, l'implémentation d'un serveur web, les tests automatiques, et bien plus encore. Restez à l'écoute !

220665-04

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.

Des questions, des commentaires ?

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

Produit

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

LIENS

[1] Manuel de référence RM0090 pour STM32F429 : https://bit.ly/3neE7S7
[2] Manuel d'utilisation de la carte Nucleo-144 (UM1974) : https://bit.ly/3oIBXKZ
[3] Cet article sur GitHub : https://github.com/cpq/bare-metal-programming-guide
[4] Contenu du fichier link.ld : https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal/link.ld
[5] Tutoriel pour Makefile : https://makefiletutorial.com/
[6] Programme de démonstration minimal (Step 0) : https://github.com/cpq/bare-metal-programming-guide/blob/main/step-0-minimal
[7] Programme de démonstration blinky (Step 1) : https://github.com/cpq/bare-metal-programming-guide/blob/main/step-1-blinky
[8] .bss [Wikipedia] : https://fr.wikipedia.org/wiki/Segment_BSS