ENSEA

Beyond Engineering

MICROPROCESSEURS

Compte Rendu TP2&3

BARE METAL

François-Gabriel Sourbé

1 Goals and means

Le but de ce TP est d'arriver à faire fonctionner la LED verte présente sur la carte Nucleo utilisée, de la même manière que dans les projets du premier semestre, mais en utilisant cette fois ci le langage assembleur plutôt que le C.

2 Green LED

2.1 LED and STM32

Q1 : LED (Preparation)

Pour allumer une LED avec un `1` logique, on branche le pin de sortie sur l'anode (côté positif +) tandis que la cathode (côté négatif –) de la LED doit être relié à la masse. Ainsi, lorsque le microcontrôleur met le pin au niveau bas `0`, il n'y a pas de différence de tension donc la LED est éteinte, et lorsque qu'on passe au niveau haut, il y a bien une différence de tension avec le potentiel le plus élevé du côté de l'anode, donc la LED s'allume. On pense également à rajouter une résistance en série avec la LED pour limiter l'intensité du courant et donc ne pas endommager cette dernière.

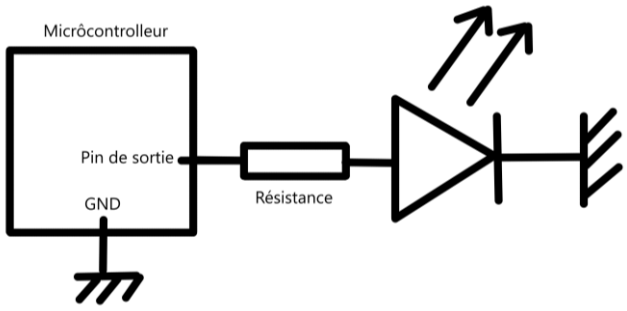


Fig. 1 – Dessin du schéma pour allumer une LED avec un `1` depuis un pin de sortie

D'après la documentation fournie de la carte, la LED verte utilisateur est connectée au port `GPIOA` sur la broche `5` (abrégé en pin `PA5`).

2.2 Your STM32CubeIDE project

Q2 : Opening previous project (or create a new one)

On crée un nouveau projet dans STM32 à partir du fichier `.ioc` fournit et en copiant le fichier `myasm.s` dans le dossier `Startup` du projet avec un glisser-déposer

2.5 New files for LED

Q3 : Creation of files for your green LED and other LEDs

On crée les différents fichiers dans les dossiers demandés en effectuant un clic-droit sur un dossier puis en sélectionnant « New > File » et en donnant le nom demandé et avec la bonne extension suivant le dossier sélectionné.

2.6 Initialization of the green LED

Enable GPIO

Q4 : Enabling GPIO port (Preparation)

Les ports GPIO sont connectés à un bus : « Advanced High-performance Bus 2 » (AHB2). Pour activer un périphérique, il faut activer son horloge en modifiant le bit qui lui est associé dans le registre « Reset and Clock Control » (RCC) correspondant. Dans notre cas, le registre est `RCC_AHB2ENR` (les lettres `ENR` à la fin du nom signifient « ENable Register »).

D'après la Memory Map de notre STM32 que l'on retrouve dans la Data Sheet fournie, le bloc RCC commence à l'adresse `0x40021000`, et d'après le Reference Manual également fourni (section 6.4.17) il faut ajouter un offset `0x4C` pour trouver le registre AHB2ENR qui nous intéresse. L'adresse de `RCC_AHB2ENR` est donc `0x4002104C`.

Finalement, la LED étant située sur le port GPIOA comme vu à la question 1, il faut activer ce dernier en mettant à `1` le bit `GPIOAEN`, qui est le premier bit de l'octet situé à l'adresse trouvée précédemment.

Q5 : Enabling GPIO port (in Bare Metal Programming: Assembly language)

On cherche à écrire une fonction pour activer l'horloge du port auquel est connecté la LED verte.
Comme vu à la question précédente, il faut pouvoir mettre à `1` un bit précis d'un octet sans modifier les autres bits. Pour cela, on utilise la méthode « Read-Modify-Write » (Lire-Modifier-Écrire) :

1. On stocke dans un registre la valeur de l'élément qu'on veut modifier.
2. On réalise ensuite un `OU` bit à bit entre cette sauvegarde et un octet dont tous les bits sont nuls sauf celui à la position qui nous intéresse (dans notre cas il suffit donc d'écrire `#1` puisqu'on touche au premier bit)
3. On écrit ce résultat à l'emplacement de départ

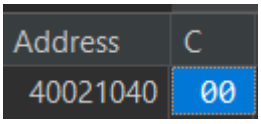
```
// COMMON DIRECTIVES to Use ASM Language
  .syntax unified
  .cpu cortex-m4
  .fpu softvfp
  .thumb

//////////////////////////////
  .global LED_Enable  // Rend la fonction visible pour le linker
  .text               // Section de code
LED_Enable:
    PUSH {R4,LR}       // Sauvegarde du contexte pour assurer la transparence
    LDR R4, =0x4002104C // Charge l'adresse du registre RCC_AHB2ENR dans R1
    LDR R0, [R4]        // Charge la valeur à l'adresse contenue dans R1 vers R0
    ORR R0, R0, #1      // R0 = R0 OR 1. Met le premier bit à 1 sans toucher aux autres
    STR R0, [R4]        // Stocke le contenu de R0 à l'adresse contenue dans R1
    POP {R4,PC}        // Restauration et retour
//////////////////////////////
```

Fig. 2 – Entête du fichier `LED.s` et code de la fonction `LED_Enable`

```
#ifndef LED_H
#define LED_H

void LED_Enable(void);

#endif
```

Fig. 3 – Contenu du fichier `LED.h` incluant le prototype de la fonction `LED_Enable`

On vient ensuite modifier la ligne 32 du fichier `myasm.s` pour que le Branch se fasse sur la fonction que l'on vient d'écrire. On réalise enfin une exécution pas à pas en surveillant la mémoire ou le registre via la vue SFR pour vérifier que le code est correct.



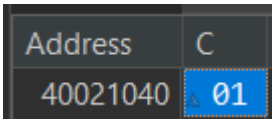


Fig. 4 et 5 – Contenu de l'octet stocké à l'adresse `0x4002104C` avant et après l'exécution de la fonction `LED_Enable`

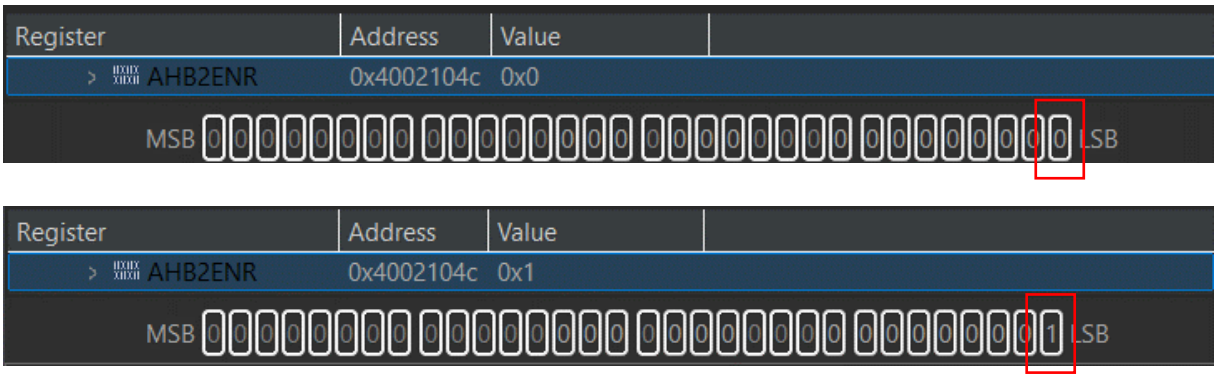


Fig. 6 et 7 – Contenu du registre `RCC_AHB2ENR` avant et après l'exécution de la fonction `LED_Enable`

Seul le premier bit a été modifié pour passer à `1` : le port GPIOA a bien été activé sans modifier l'état des autres ports

*NB : Après chaque test la valeur finale du registre est toujours la même : **1**. On pourrait donc penser qu'il suffit d'assigner directement 1 au registre pour avoir le même résultat. Mais en réalité, on réalise à chaque fois le test après un Reset, donc avec toujours la même valeur de départ : 0. Or, dans une application réelle du code, on ne sait pas forcément quand sera appelée la fonction LED_Enable et donc on ne peut pas connaitre avec certitude la valeur de départ du registre, raison pour laquelle on s'assure de pouvoir conserver la valeur des bits qui n'ont pas besoin d'être modifier pour ne pas créer de problèmes avec le reste du programme.*

Configuration of GPIO pin for green LED

Q6 : Configuration for the Green LED (Preparation)

On configure la LED en Output Push-Pull :

- « Output » (littéralement « Sortie ») car on veut envoyer un signal pour contrôler la LED et non pas recevoir une information comme en « Input ».

- « Push-Pull » car c'est dans cette configuration que les transistors du microcontrôleur peuvent fournir ou absorber du courant sur la broche, ce qui correspond à ce qui est demandé dans notre application (allumer/éteindre la LED en mettant le signal au niveau haut/bas comme vu à la question 1). Il n'y a donc pas besoin d'amener du courant par une autre source à l'aide de résistances de « Pull-Up » / « Pull-Down » (PU/PD).

On cherche donc les 7 bits à modifier pour mettre le pin de la LED dans la configuration donnée par l'énoncé. On s'aide pour cela de la section 8.5 du Reference Manual.

- Pour le mode, le registre est `GPIOA_MODER`, et les bits à modifier sont les bits 10 et 11 appelés `MODE5`. Il faut le 11^{e} à `0` et le 10^{e} à `1` pour le mode « Output ».

- Pour le type de sortie, on modifie le 5^{e} bit nommé `OT5` du registre `GPIOA_OTYPER`. On le met à `0` pour passer en « Push-Pull ».

- Pour la vitesse de sortie, il s'agit de `OSPEED5`, les bits 10 et 11 de `GPIOA_OSPEEDR`. On met `00` pour mettre la vitesse la plus faible « Low Speed ».

- Enfin pour les résistances de PU/PD, puisqu'il n'y en a aucune on met à `00` les bits 10 et 11 nommés `PUPD5` dans `GPIOA_PUPDR`

On cherche maintenant les adresses de différents registres.
D'après la Memory Map, pour le bus AHB2, les registres concernant le périphérique GPIOA commencent à partir de l'adresse `0x48000000`. Il reste ensuite à trouver dans la section 8.5 du Reference Manual les offsets pour chaque type de registre cherché.

Nom de registre	Offset
GPIOx_MODER	0x00
GPIOx_OTYPR	0x04
GPIOx_OSPEEDR	0x08
GPIOx_PUPDR	0x0C
GPIOx_IDR	0x10
GPIOx_ODR	0x14

Fig. 8 – Offset associés à différents registres (`x` désigne la lettre d'un port de `A` à `I`)

Pour obtenir les adresses exactes des registres qui nous intéressent, il suffit donc d'additionner l'adresse de base pour les registres du GPIOA et l'offset pour le type de registre cherché.

Ensuite, pour accéder dans un registre donné aux paramètres du pin n : si le paramètre n'a que deux états possibles, il n'y a besoin que d'un seul bit par pin donc on regarde le bit à la position n dans le registre (donc le 5^{e} bit dans notre cas) ; et pour les paramètres plus complexes qui nécessite deux bits, on regarde aux position $2n$ et $2n + 1$ (soit 10 et 11 ici). Cela correspond à ce qui a été trouvé plus tôt.

Après chaque Reset, les registres sont réinitialisés à une certaine valeur connue que l'on peut trouver dans le Reference Manual.

Nom de registre	Valeur de Reset du registre	Valeur de Reset des bits relatifs au pin 5
GPIOA_MODER	0xABFFFFFF	11
GPIOA_OTYPR	0x00000000	0
GPIOA_OSPEEDR	0x0C000000	00
GPIOA_PUPDR	0x64000000	00

Fig. 9 – Valeurs de différents registres après un Reset

On voit donc qu'après un Reset les bits relatifs au pin de la LED n'ont pas besoin d'être modifiés car ils ont déjà la valeur adéquate vue plus tôt, sauf pour le mode de sortie : le pin est initialisé en Analogique (donc une entrée) plutôt qu'en sortie, c'est donc pour cela qu'on doit modifier le registre `MODER` après un Reset.

Q7 : Configuration of the Green LED

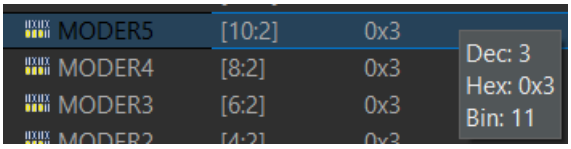
Comme pour la question 5, on doit modifier un bit précis d'un registre sans toucher aux autres.
Pour le registre `MODER`, comme vu à la question précédente il faut passer le bit 10 à `1` et le 11 à `0`. On a déjà vu comment passer un bit précis à `1`, et pour `0` il faut utiliser le même principe mais avec l'opération inverse : on fait un `ET` bit à bit avec des `1` partout sauf aux positions où on veut passer à `0`.

```
//////////////////////////////

  .global LED_Config
  .text

LED_Config:
  PUSH {R4,R5,LR}
  BL LED_Enable        // On s'assure que GPIOA est activé
                       // pour pouvoir faire les modifications
  LDR R1, =0x48000000
  LDR R0, [R1]
  ORR R0, R0, #1024    // Met le bit 10 à 1 sans toucher aux autres
                       // 1024 = 2^10 -> tous les bits à 0 sauf le 10
  LDR R2, =0xFFFFF7FF // On ne pourra pas faire AND avec un élément de
                       // 32 bits donc on le charge dans un registre
                       // 0x7FF = 0111 1111 1111 -> seul le bit 11 est à 0
  AND R0, R0, R2
  STR R0, [R1]
  POP {R4,R5,PC}

//////////////////////////////
```

Fig. 10 – Code de la fonction `LED_Config`

En théorie on devrait également inclure les modifications avec la même technique pour les autres registres (`OTYPE`, `PUPD`, `OSPEED`) si on ne pouvait pas être sur qu'ils ont à la bonne valeur. Toutefois, dans notre cas on sait que ces registres ne seront pas modifiés par notre programme et vont donc garder leur valeur initiale après un Reset (qui est déjà celle qui nous convient comme vu lors de la préparation). Nous n'avons donc pas besoin de les modifier pour ce projet.

On peut maintenant vérifier que le code est correct en surveillant la valeur du registre.



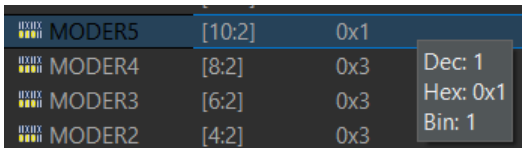


Fig. 11 et 12 – Valeur de `MODE5` avant et après l'exécution de `LED_Config`

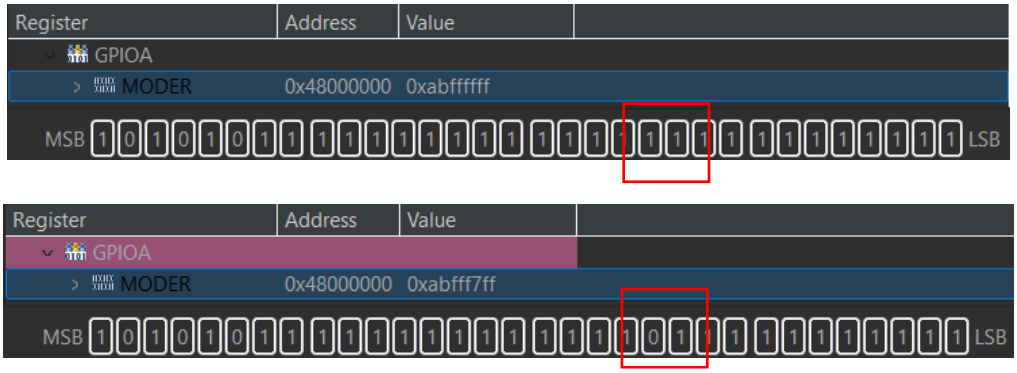


Fig. 11 et 12 – Valeur de `GPIOA_MODER` avant et après l'exécution de `LED_Config`

Ainsi, si on modifie directement la valeur de `ODR5` (le bit 5 de `GPIOA_ODR`) pour le mettre à 1, on peut voir la LED s'allumer.

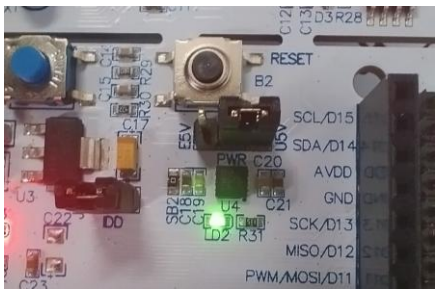


Fig. 13 – LED verte de la carte allumée après avoir exécuté `LED_Enable` et mit `ODR5` dans `GPIOA_ODR` à `1`

Q8 : Display the Green LED

On traduit directement en assembleur le pseudo code donné (on utilise les méthodes vues précédemment pour écrire `1` ou `0` au bon endroit)

```
////////////////////////////////

  .global LED_DriveGreen
  .text

LED_DriveGreen:
  PUSH {R4,R5,LR}
  AND R0, #1
  CMP R0, #1
  BEQ LED_On
LED_Off:        // L'étiquette n'a pas d'autre utilité que la clartée de lecture
  LDR R4, =0x48000014
  LDR R0, [R4]
  LDR R5, =0xFFFFFFEF
  AND R0, R0, R5
  STR R0, [R4]
  POP {R4,R5,PC}

////////////////////////////////
```

Fig. 14 – Code de la fonction `LED_DriveGreen`

```
////////////////////////////////

  .global LED_On
  .text

LED_On:
  LDR R4, =0x48000014
  LDR R0, [R4]
  ORR R0, R0, #32
  STR R0, [R4]
  POP {R4,R5,PC}

////////////////////////////////
```

Fig. 15 – Code de la fonction auxiliaires `LED_On` utilisée par `LED_DriveGreen` pour allumer la LED si besoin

On teste tous les cas de figures possible (c'est à dire `LED_DriveGreen(1)` et `LED_DriveGreen(0)` quand `ODR5` est à déjà `0` puis quand il est à `1`) à l'aide du débugger pour s'assurer que tout fonctionne correctement.

Q9 : Green LED switching state

On crée `utils.c` et son fichier d'entête dans les dossiers adéquats.

```c
#include <stm32l4xx_hal.h>
#include "utils.h"
#include "LED.h"

int GreenLED_state;

void setup(void){
    LED_Config();
    GreenLED_state=1;
}

void loop(void){
    LED_DriveGreen(GreenLED_state);
    GreenLED_state=1-GreenLED_state;
}
```

```c
#ifndef INC_UTILS_H_
#define INC_UTILS_H_

void setup();
void loop();

#endif /* INC_UTILS_H_ */
```

Fig. 16 et 17 – Contenu des fichiers `utils.c` et `utils.h`

On vient ensuite modifier la ligne 32 du fichier `myasm.s` pour que le branch se fasse sur `setup`, puis la ligne 34 pour que le `branch` dans la boucle infinie se fasse sur `loop`.

Pour éviter que la LED soit constamment en train de changer d'état, on met un breakpoint dans loop pour mettre le programme en pause à chaque passage dans la boucle

La variable globale `GreenLED_state` est stockée dans la RAM, plus précisément dans la section `.bss` de la SRAM1, dont l'adresse de départ est `0x20000000`. Comme on n'initialise pas d'autre variable globale, elle devrait être située dans les premières adresse après le début de la section. Et effectivement dans notre cas, à chaque fois qu'on exécute pas à pas le programme et que l'on surveille cette zone de la mémoire, on peut voir que l'instruction `GreenLED_state=1-GreenLED_state` dans la fonction `loop` fait alterner entre `0` et `1` l'élément stocké à l'adresse `0x20000020`.

Fig. 18 et 19 – Contenu de la variable stockée à l'adresse `0x20000020`, alternant après chaque exécution de la fonction `loop`

Si on veut définir le plus grand tableau d'entier possible comme variable globale, il faut tout d'abord s'assurer que l'entièreté du tableau soit dans le même bloc mémoire (SRAM1 ou SRAM2) puisque ceux-ci sont situés à des emplacement différents dans les adresses mémoire alors qu'un tableau est un ensemble d'élément dont les adresses doivent se succéder.
Si on choisit le plus grand bloc qui est la SRAM1 avec 96 Ko disponibles au total (soit $96 \times 1024 = 98304$ octets, alors sachant qu'un entier est codé sur 4 octets on en déduit qu'un tableau ne peut pas avoir une taille supérieure à $N_{max} = \frac{98304}{4} = 24576$.
Toutefois, il faut noter qu'on tableau de cette taille occuperai l'entièreté de la SRAM1, mais il doit également rester de la place en mémoire pour la Pile. En pratique la taille maximale qu'il est possible d'accorder à un tableau en tant que variable globale est donc légèrement inférieure au N_{max} calculé précédemment (diminuer la taille de 256 permet de garder 1 Ko de disponible)

Enfin, voici la traduction en assembleur du code de la fonction `loop` en suivant les règles du compilateur :

```
loop:
  PUSH {R4,LR} // Sauvegarde du contexte

  // LED_DriveGreen(GreenLED_state)
  LDR R4, =GreenLED_state // Si la variable globale est définie
                          // dans le même fichier que la fonction
                          // alors on connait son adresse
                          // et on peut la mettre dans un registre
  LDR R0, [R4]            // On met la valeur de GreenLED_state dans R0
                          // car c'est l'argument de la fonciton suivante
  BL  LED_DriveGreen

  // GreenLED_state=1-GreenLED_state
  LDR R0, [R4]    // On charge GreenLED_state dans un registre
  MOV R1, #1      // On prépare la constante 1
  SUB R0, R1, R0  // R0=R1-R0
  STR R0, [R4]    // On écrit le résultat dans GreenLED_state

  POP {R4,PC}  // On restaure le contexte
```

Fig. 20 – Code expliqué de la fonction `loop` traduite en assembleur

Q10 : Green LED blinking

On ajoute dans `utils.c` une variable globale `int wait_time` ainsi que la fonction `UTILS_WaitN10ms` fournie dans l'énoncé et on pense à ajouter son appel à la fin de la fonction `loop`.

On peut s'intéresser au fonctionnement de la fonction d'attente : La fonction consiste à faire tourner une boucle contenant une seule opération pendant 2500 itérations, elle-même dans une boucle à N itérations (N étant l'argument de la fonction).
La boucle à 2500 itérations prend 10 ms à être exécutée et la fonction complète dure donc environ $N \times 10\ ms$.
Si on veut que la LED soit allumée pendant 500 ms puis éteinte pendant 500 ms à nouveau, il faut alors renseigner comme argument $N = 50$. Cela fonctionne lorsqu'on exécute le code sur la carte.

2.7 Initializations from ioc file

Q11 : modification of the startup file

Après avoir créé le dossier demandé à la racine du projet, on y glisse le fichier `myasm.s` qui n'est donc plus dans le dossier `Startup`.
L'initialisation du code n'est donc plus la même que jusqu'à présent : l'énoncé nous indique que l'horloge du processeur est maintenant réglée à 80 MHz.
Ainsi, lorsqu'on appelle les fonctions `setup` et `loop` aux endroits demandés et qu'on exécute le code, on voit la LED clignoter extrêmement rapidement : c'est parce que le processeur a une cadence 40 fois plus élevée, la fonction d'attente vue à la question précédente ne s'exécute plus en 500 ms mais bien plus vite

Q12 : modification of the waitN10ms

Puisque l'on vient de voir que le processeur va 40 fois plus vite, il suffit que la fonction d'attente prenne 40 fois plus de temps à être exécutée pour retrouver le comportement que l'on avait précédemment : on modifie la boucle à 2500 itérations pour qu'elle en fasse maintenant 100000.

3 Beautiful Blue Button

Q13 : Creation of files for the buttons

On crée le fichier `button.c` en faisant « Clic Droit > New > Source File » sur le dossier `Core/Src`, et `button.h` dans le dossier `Core/Inc` en sélectionnant cette fois ci « Header File ».

Q14: Initializations and use of the Blue Button (preparation)

D'après le « User Manual », le bouton bleu destiné à l'utilisateur est sur le pin `PC13` (pin 13 du port GPIOC).
Le bouton est connecté entre la broche du microcontrolleur et la masse, il s'agit d'une configuration « Input with Pull-Up resistor » (la résistance est déjà soudée sur la carte).
Les registres impliqués pour définir cette configuration sont :

- `RCC_AHB2ENR` situé à l'adresse `0x4002104C` comme vu plus tôt pour activer l'horloge du port C
- GPIOC_MODER avec pour adresses respectives `0x48000800`.

Pour ce qui est de la résistance de Pull-Up, il n'est pas nécessaire de modifier le registre PUPDR, car la résistance de Pull-Up est déjà configurée correctement en interne sur le contrôleur, la configuration par défaut est suffiante.

Q15: Write BUTTON_Enable (in C with home made pointers)

On code en C une fonction permettant d'activer l'horloge du GPIOC :

```c
#include "button.h"

void BUTTON_Enable(){
    // On met dans un pointeur l'adresse du registre qu'on souhaite modifier
    int* RCC_AHB2ENR = (int*)0x4002104C; // typage forcé pour éviter un warning
    // On met à 1 le 3e bit du contenu pointé grâce à un OU bit à bit
    *RCC_AHB2ENR |= (1<<2);
}
```

Fig. 21 – Contenu du fichier `button.c`

On n'oublie pas d'écrire le prototype de la fonction dans `button.h`, on peut ainsi inclure ce dernier dans `utils.c` pour ajouter dans la fonction `setup` un appel à la fonction que l'on vient de coder.

On place ensuite un breakpoint avant l'appel de BUTTON_Enable pour pouvoir surveiller l'évolution de `RCC_AHB2ENR` grâce au debugger.

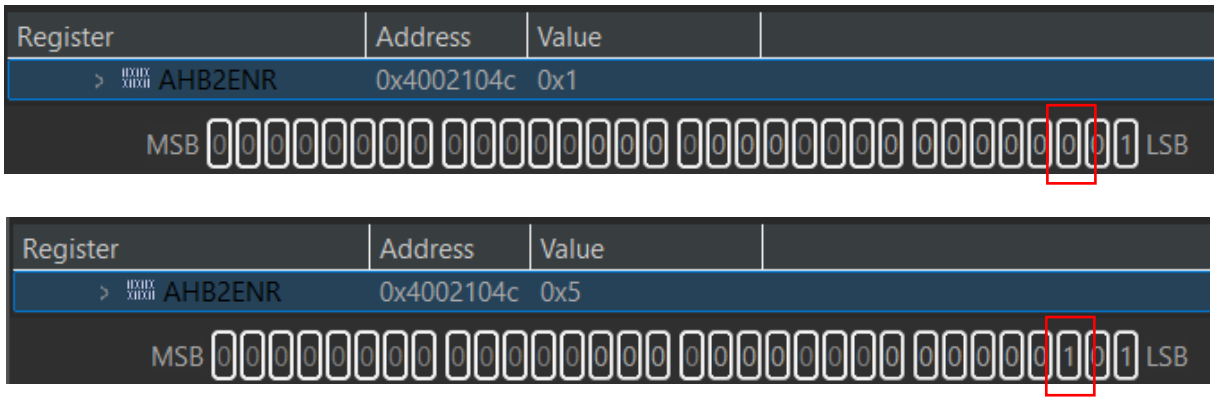


Fig. 22 et 23 – Contenu du registre `RCC_AHB2ENR` avant et après l'exécution de la fonction `BUTTON_Enable`

Après chaque Reset, les horloges de tous les ports sont désactivées (afin de n'activer que les ports utilisés et donc d'économiser de l'énergie). Il est donc nécessaire d'appeler la fonction à chaque démarrage.

Q16: Write BUTTON_Config (data structures from stm32l4xx.h)

On souhaite modifier le registre GPIOC pour activer le bouton en utilisant directement les structures définies dans `stm32l4xx.h`. En effet la bibliothèque contient des pointeurs vers les différents registres et on peut ensuite en modifier chaque champ. Dans notre cas il faut donc modifier `GPIOC->MODER`.

Comme vu précédemment, pour modifier les éléments relatifs au pin 13, il faut modifier les bits 26 et 27 pour les paramètres codés sur deux bits (ce qui est le cas ici).

Ainsi, pour passer en input, il faut écrire 00 au niveau de ces deux bits dans MODER.
On réutilise donc la méthode précédente : sur les bits où on veut écrire 0, on fait un ET bit à bit avec un 0 sur le bit qu'on veut modifier (et 1 sur les autres), et à l'inverse pour écrire 1 on fait un OU avec un 1 sur les bits concernés et 0 ailleurs.

```
#include <stm32l4xx.h>

void BUTTON_Config(){
    GPIOC->MODER &= ~(3<<26);
    // On écrit 3 (soit 11 en binaire) et on le décale de 26 bits avec <<
    // Puis on inverse avec ~
    // Donc les bits 26 et 27 sont à 0 et les autres à 1
}
```

Fig. 24 – Code de la fonction BUTTON_Config

On ajoute l'appel de la fonction dans `setup` et on vérifie qu'elle est correcte avec le débugger.

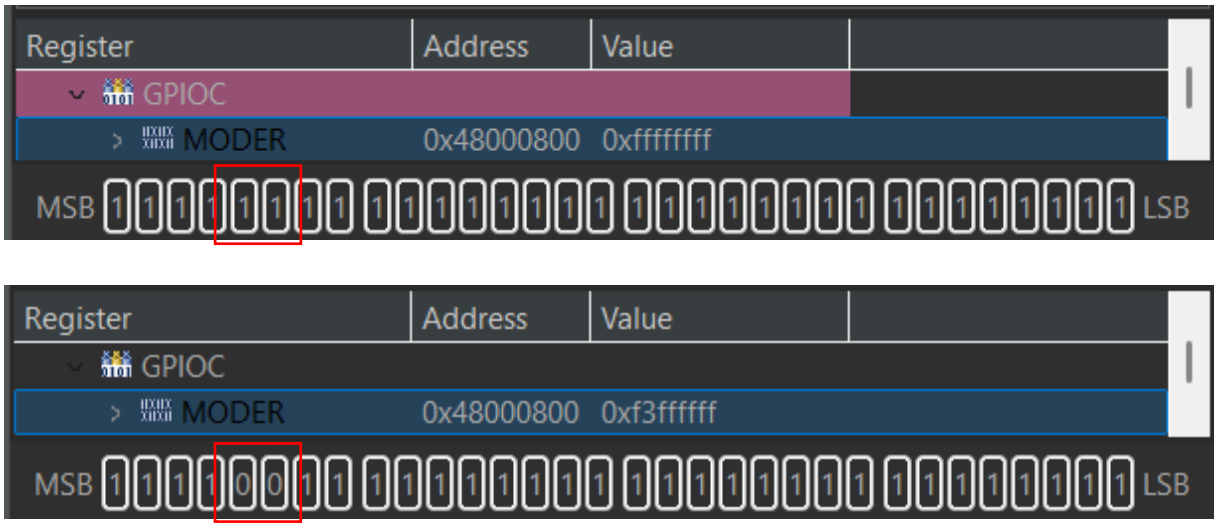


Fig. 25 et 26 – Valeur de `GPIOC_MODER` avant et après l'exécution de `BUTTON_Config`

Q17: Write BUTTON_GetBlueLevel (HAL functions) and test!

On utilise la fonction `HAL_GPIO_ReadPin` pour savoir si le bouton est pressé ou non (puisqu'il y a une résistance de Pull-Up, la fonction renvoie `0` si on l'appelle quand le bouton est pressé et `1` sinon)

```c
#include <stm32l4xx_hal.h>

int BUTTON_GetBlueLevel(){
    return HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13);
}
```

Fig. 27 – Code de la fonction `BUTTON_GetBlueLevel`

On crée ensuite un test rapide pour vérifier le fonctionnement du bouton qui augmente la vitesse du clignotement de la LED lorsque le bouton est appuyé

```
void button_test(){
    if(BUTTON_GetBlueLevel()==GPIO_PIN_RESET){
        wait_time=50;
    }
    else{
        wait_time=100;
    }
}
```

Fig. 28 – Fonction de test du bouton ajoutée dans `utils.c`

Ainsi, lorsqu'on ajoute l'appel à ce test dans la boucle infinie de la fonction `main`, on constate que la LED clignote deux fois plus vite lorsque le bouton est pressé et reprend la vitesse normale lorsqu'on le relâche.

Q18: Control the speed of the LED blinking

On modifie la fonction `loop` pour que, lorsque le bouton soit appuyé, si la LED clignote 1 fois par seconde, elle clignote 5 fois plus vite, et vice versa.

```
void loop(void){
    if(BUTTON_GetBlueLevel()==GPIO_PIN_RESET){
        if(wait_time=50){
            wait_time=10;
        }
        else{
            wait_time=50;
        }
    }
    LED_DriveGreen(GreenLED_state);
    GreenLED_state=1-GreenLED_state;
    UTILS_WaitN10ms(wait_time);
}
```

Fig. 29 – Ajout dans la fonction `loop` du contrôle de la vitesse de clignotement via le bouton bleu

Lorsqu'on exécute le code et que l'on essaie de vérifier son fonctionnement, on constate qu'il fonctionne, mais pas parfaitement : par moment l'appui n'est pas détecté. En effet, comme on utilise une boucle for pour attendre entre deux exécutions de la fonction loop, pendant tout ce temps l'exécution du code est coincée dans la boucle et ne peux pas exécuter la partie qui vérifie l'état du bouton et modifie la vitesse.

Pour régler ce problème, on pourrait utiliser la méthode vue au semestre précédent : mettre en place des interruptions, qui permettraient de détecter un évènement comme l'appui sur le bouton et d'agir immédiatement.

Q19: Summary of your work

Dans ce TP nous avons pu utiliser différents moyens d'intéragir avec un appareil STM32.

Tout d'abord, l'Assembleur. On écrit directement des valeurs binaires dans les adresses mémoire via des instructions ASM. On a ainsi un contrôle absolu sur le matériel et le temps d'exécution, mais reste très complexe à écrire et à maintenir (difficilement lisible).

Nous avons ensuite utilisé du code C avec pointeurs vers des adresses physiques fixes.
Cela est lisible que l'ASM et permet d'utiliser les opérations logiques du C, mais reste fastidieux. Si l'adresse change (nouvelle puce), tout le code est à refaire.

Enfin, nous avions écrit du code C avec les structures de registres définies par `stm32l4xx.h`. Le code est lisible (noms explicites), l'autocomplétion aide le développeur, et les adresses sont gérées par les fichiers d'en-tête, mais nécessite toujours une connaissance approfondie du Reference Manual (quels bits modifier pour quel mode).

Nous avons également pu écrire du code assembleur en ayant conscience des règles de compilation pour qu'il fonctionne correctement avec le reste du code en C : les registres `R0` à `R3` sont utilisés pour les arguments de fonctions, et les registres `R4` à `R11` sont les variables locales. Si ces dernières sont modifiées dans une fonction, on doit les sauvegarder dans la Pile au début pour pouvoir les restaurer à la fin. Et de la même façon, s'il y a des appels de fonctions imbriqués, il faut aussi penser à sauvegarder `LR` pour pouvoir revenir à la fonction appelante.