# Compte-Rendu de TP2-TP3

Youssef Chemrakhi et Théodore Condette

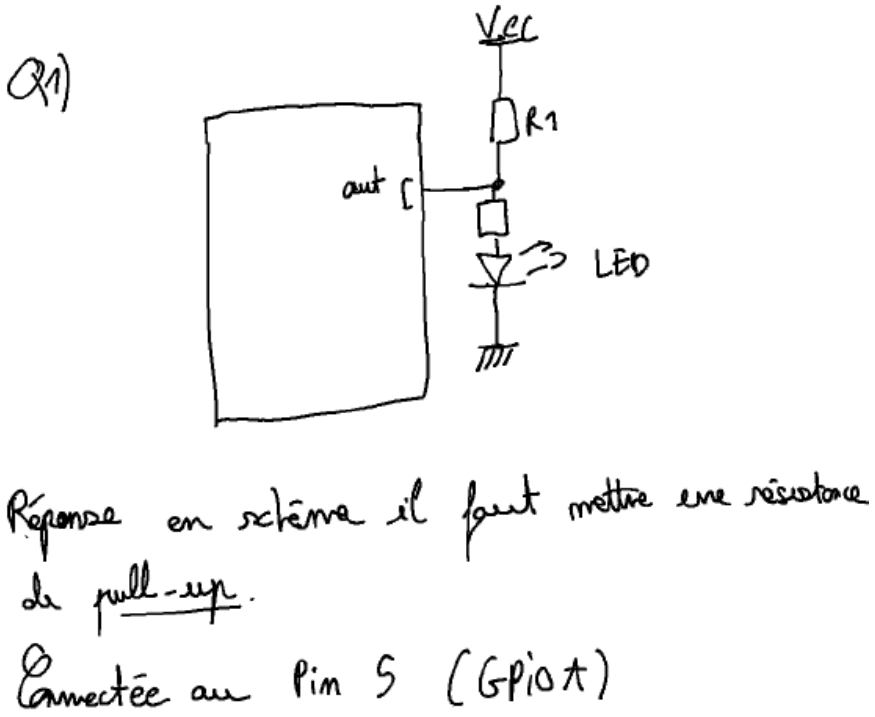



FIGURE 1 – Préparation

Pour allumer une LED avec un niveau logique 1, la LED est connectée en série avec une résistance entre la broche GPIO du microcontrôleur et la masse (GND). Lorsque la broche est à 1, un courant circule et la LED s'allume. Lorsqu'elle est à 0, la LED est éteinte.

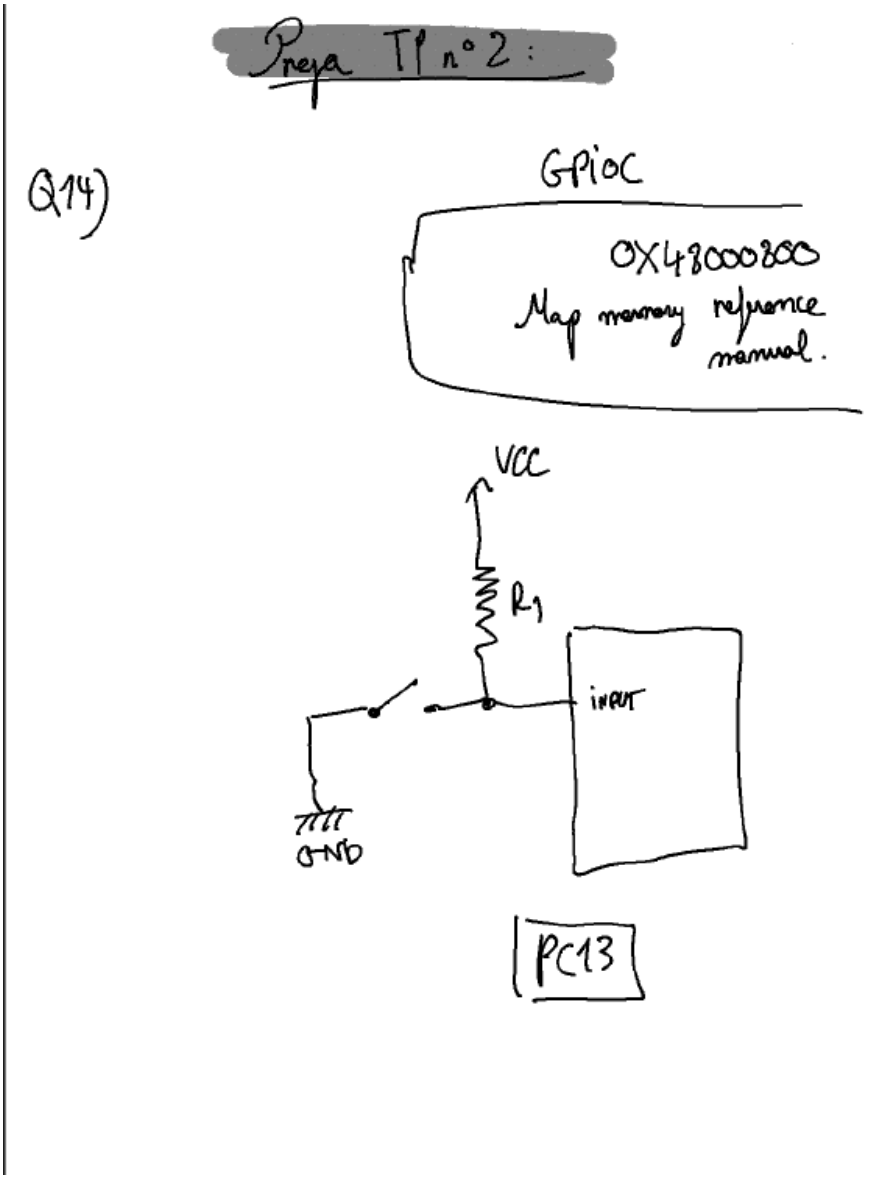



FIGURE 2 – Préparation

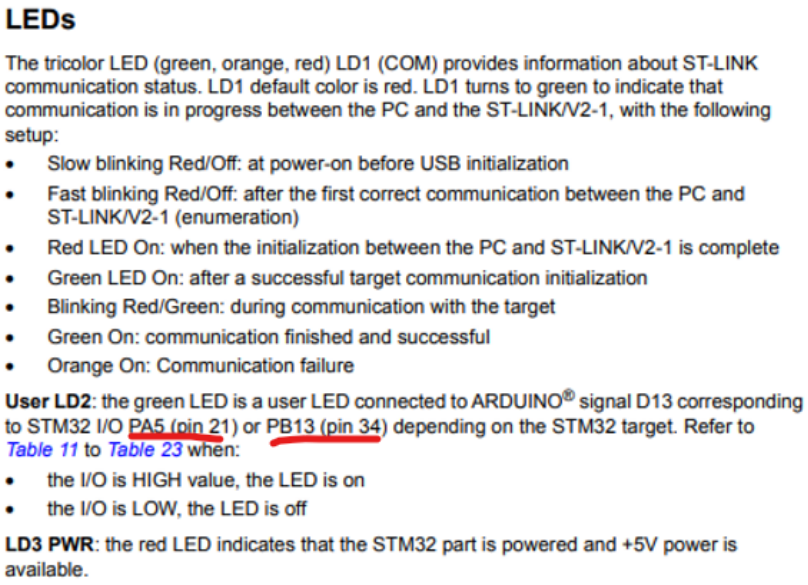
## Q3 : Création des fichiers LED

Les fichiers suivants sont créés :
— `LED.c` dans `Core/Src`
— `LED.s` dans `Core/Startup`
— `LED.h` dans `Core/Inc`

## 0.1 Activation de la LED verte

Tout d'abord, l'objectif est d'activer la LED verte. Avant de se lancer dans l'écriture du code, il est nécessaire de connaître plusieurs informations préalables. La première consiste à identifier le *pin* sur lequel est connectée la LED.

Comme indiqué sur la figure ci-dessous, la LED verte peut être raccordée à deux *pins* différents selon le type de carte Nucleo utilisé : soit `PA5`, soit `PB13`.



**LEDs**

The tricolor LED (green, orange, red) LD1 (COM) provides information about ST-LINK communication status. LD1 default color is red. LD1 turns to green to indicate that communication is in progress between the PC and the ST-LINK/V2-1, with the following setup:

- Slow blinking Red/Off: at power-on before USB initialization
- Fast blinking Red/Off: after the first correct communication between the PC and ST-LINK/V2-1 (enumeration)
- Red LED On: when the initialization between the PC and ST-LINK/V2-1 is complete
- Green LED On: after a successful target communication initialization
- Blinking Red/Green: during communication with the target
- Green On: communication finished and successful
- Orange On: Communication failure

**User LD2**: the green LED is a user LED connected to ARDUINO® signal D13 corresponding to STM32 I/O PA5 (pin 21) or PB13 (pin 34) depending on the STM32 target. Refer to *Table 11* to *Table 23* when:

- the I/O is HIGH value, the LED is on
- the I/O is LOW, the LED is off

**LD3 PWR**: the red LED indicates that the STM32 part is powered and +5V power is available.

FIGURE 3 – Documentation de la LED verte.

Afin de distinguer ces deux cas, on s'appuie sur les schémas 2 et 3, qui représentent respectivement le schéma de la carte Nucleo et la carte réelle.

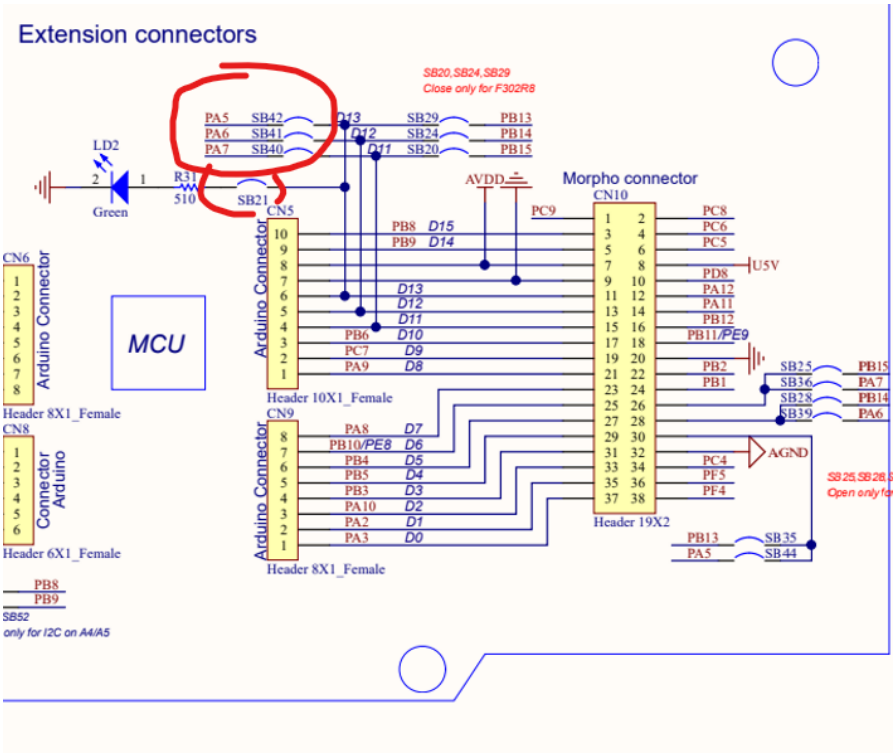



FIGURE 4 – Documentation de la nucléo (Schéma2).

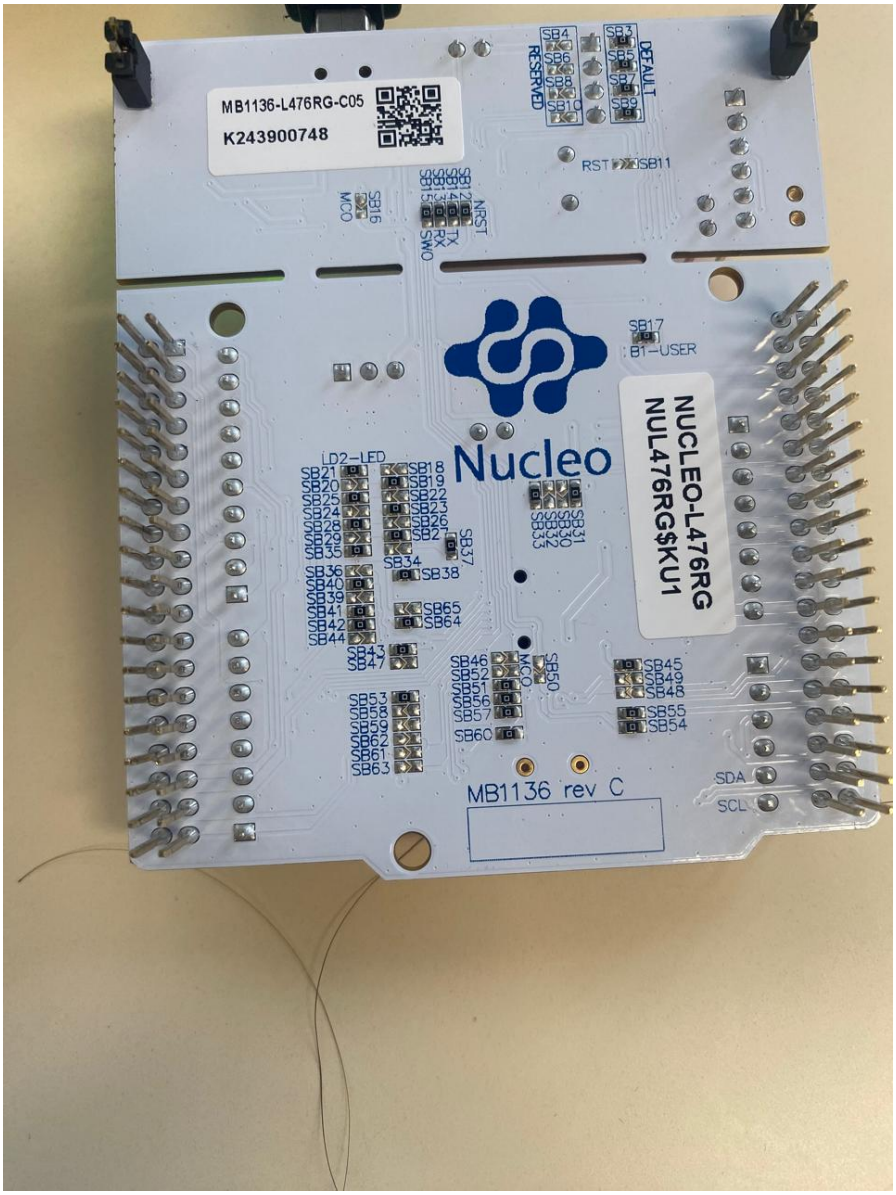



FIGURE 5 – Photo réelle de la nucléo (Schéma3).

Dans notre cas, les straps 21 et 42 sont raccordés. Le *pin* à utiliser est donc `GPIOA` broche 5 (`PA5`). Une fois ce point identifié, l'objectif est de piloter ce périphérique, à savoir la LED verte.

## 0.2 Autorisation et configuration d'un périphérique

Avant de pouvoir contrôler un périphérique, il est indispensable d'autoriser son horloge. Cette autorisation se fait via le registre `RCC` (*Reset and Clock Control*). Sans cette étape, le périphérique ne peut pas fonctionner.

Ensuite, pour manipuler le périphérique concerné, il est nécessaire d'accéder à ses registres propres. Pour cela, on se réfère à la *memory map* du microcontrôleur. Par exemple, le port `GPIOA` est associé au registre `RCC_AHBR`. Cette information est disponible dans la documentation constructeur, comme illustré dans les figures suivantes.



Table 2. STM32L49x/L4Ax devices memory map and peripheral register boundary addresses(1)

| Bus | Boundary address | Size (bytes) | Peripheral | Peripheral register map |
|---|---|---|---|---|
| AHB4 | 0xA000 1000 - 0xA000 13FF | 1 KB | QUADSPI | Section 17.6.14: QUADSPI register map |
| AHB3 | 0xA000 0400 - 0xA000 0FFF | 3 KB | Reserved | - |
| | 0xA000 0000 - 0xA000 03FF | 1 KB | FMC | Section 16.7.8: FMC register map |
| - | 0x5006 0C00 - 0x5FFF FFFF | ~260 MB | Reserved | - |
| AHB2 | 0x5006 0800 - 0x5006 0BFF | 1 KB | RNG | Section 27.7.4: RNG register map |
| | 0x5006 0400 - 0x5006 07FF | 1 KB | HASH | Section 29.7.8: HASH register map |
| | 0x5006 0000 - 0x5006 03FF | 1 KB | AES(2) | Section 28.7.18: AES register map |
| | 0x5005 0400 - 0x5005 FFFF | 63 KB | Reserved | - |
| | 0x5005 0000 - 0x5005 03FF | 1 KB | DCMI | Section 20.5.12: DCMI register map |
| | 0x5004 0400 - 0x5004 FFFF | 63 KB | Reserved | - |
| | 0x5004 0000 - 0x5004 03FF | 1 KB | ADC | Section 18.8: ADC register map |
| | 0x5000 0000 - 0x5003 FFFF | 256 KB | OTG_FS | Section 47.15.57: OTG_FS register map |
| | 0x4800 2400 - 0x4FFF FFFF | ~127 MB | Reserved | - |
| | 0x4800 2000 - 0x4800 23FF | 1 KB | GPIOI | Section 8.5.13: GPIO register map |
| | 0x4800 1C00 - 0x4800 1FFF | 1 KB | GPIOH | Section 8.5.13: GPIO register map |
| | 0x4800 1800 - 0x4800 1BFF | 1 KB | GPIOG | Section 8.5.13: GPIO register map |
| | 0x4800 1400 - 0x4800 17FF | 1 KB | GPIOF | Section 8.5.13: GPIO register map |
| | 0x4800 1000 - 0x4800 13FF | 1 KB | GPIOE | Section 8.5.13: GPIO register map |
| | 0x4800 0C00 - 0x4800 0FFF | 1 KB | GPIOD | Section 8.5.13: GPIO register map |
| | 0x4800 0800 - 0x4800 0BFF | 1 KB | GPIOC | Section 8.5.13: GPIO register map |
| | 0x4800 0400 - 0x4800 07FF | 1 KB | GPIOB | Section 8.5.13: GPIO register map |
| | 0x4800 0000 - 0x4800 03FF | 1 KB | GPIOA | Section 8.5.13: GPIO register map |
| - | 0x4002 BC00 - 0x47FF FFFF | ~127 MB | Reserved | - |

FIGURE 6 – Documentation de la LED verte.

## 0.3 Memory map et registres du GPIO

Enfin, l'étude de la *memory map* permet de connaître l'allocation mémoire de chaque périphérique. Chaque périphérique occupe une zone mémoire de `1 KB`. Après avoir repéré la zone mémoire correspondant au périphérique souhaité, il est nécessaire de consulter la documentation afin de comprendre le fonctionnement du *pin* concerné.

Dans le cas du `GPIOA`, ces informations sont détaillées dans le *Reference Manual*, notamment à la page 303. Il est alors possible d'identifier l'adresse de base du périphérique ainsi que les différents offsets des registres nécessaires à sa configuration.

## Q4 : Activation du GPIO

Le registre permettant d'activer un port GPIO est `RCC_AHB2ENR`. Pour la LED verte, le port utilisé est `GPIOA` et le bit à activer est `GPIOAEN`. L'adresse du registre `RCC_AHB2ENR` est `0x4002104C`.

## Q5 : Fonction LED_Enable

La fonction `LED_Enable` active l'horloge du port GPIOA en mettant à 1 le bit `GPIOAEN` du registre `RCC_AHB2ENR`. Cette étape est indispensable pour pouvoir accéder aux registres du GPIOA. Le bon fonctionnement est vérifié en mode debug à l'aide de la vue mémoire ou de la vue SFR.

```
1 LDR R0,=0x4002104C
2 MOV R1,#0x01
3 STR R1,[R0]
```

Listing 1 – Fonction LED_Enable

L'inconvénient de cette manipulation est qu'elle entraîne l'écrasement des autres périphériques.

Voici une solution plus ingénieuse.

```
1 LDR R0,=0x4002104C
2 MOV R1,#0x01
3 LDR R2,[R0]
4 ORR R1,R1,R2
5 STR R1,[R0]
```

Listing 2 – Fonction LED_Enable V.2

## Q6 : Configuration GPIO (préparation)

La broche de la LED est configurée en sortie *General Purpose Output Push-Pull* sans résistance de pull-up ou pull-down afin de garantir un fonctionnement simple, stable et peu consommateur.

Configuration de PA5 :

— MODER = 01 (sortie)
— OTYPER = 0 (push-pull)
— OSPEEDR = 00 (low speed)
— PUPDR = 00 (no pull-up / pull-down)

Registres utilisés :

— `GPIOA_MODER` : 0x48000000
— `GPIOA_OTYPER` : 0x48000004
— `GPIOA_OSPEEDR` : 0x48000008
— `GPIOA_PUPDR` : 0x4800000C
— `GPIOA_IDR` : 0x48000010
— `GPIOA_ODR` : 0x48000014

Après reset, les GPIO sont configurés en entrée, ce qui impose de modifier le registre MODER.

## Q7 : Fonction LED_Configure

La fonction `LED_Configure` modifie principalement le registre `GPIOA_MODER` afin de configurer la broche PA5 en sortie. Les autres registres ne sont pas modifiés car leurs valeurs par défaut sont adaptées à l'utilisation de la LED.

```
1 .equ GPIOA, 0x48000000
2 .equ GPIOA_MODER, GPIOA+0x00
3     LDR R0,= GPIOA_MODER
4     LDR R1,[R0]
5     LDR R2,=0xFFFFF3FF
6     AND R1,R2
7     LDR R3,=0x400
8     ORR R1,R3
9     STR R1,[R0]
```

Listing 3 – Fonction LED_Configure

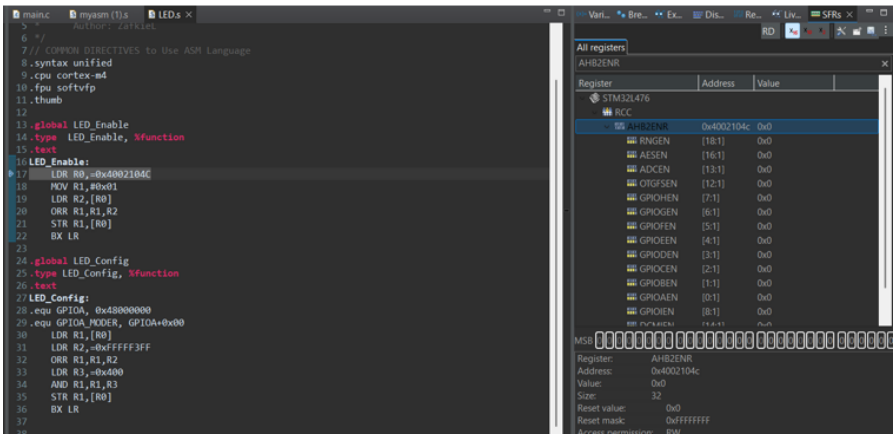



FIGURE 7 – LED_Configure et LED_Enable avant éxecution.

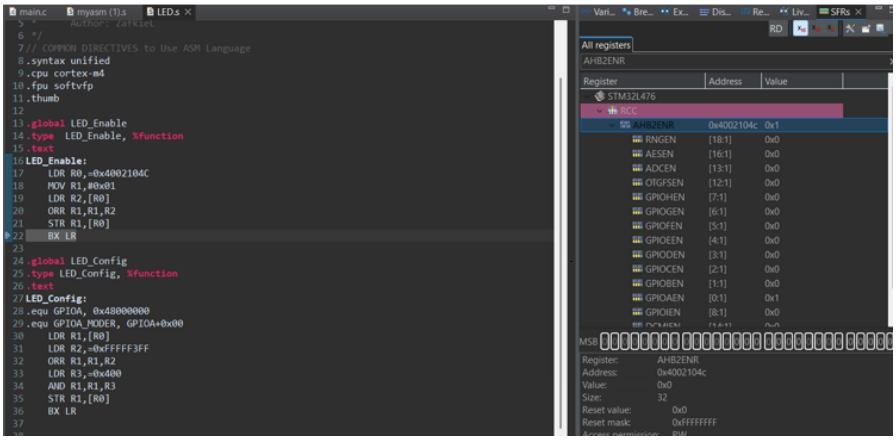



FIGURE 8 – LED_Configure et LED_Enable après éxecution.

## Q8 : Fonction LED_DriveGreen

```
39 .global LED_DriveGreen
40 .type LED_DriveGreen, %function
41 .text
42 LED_DriveGreen:
43 .equ GPIOA, 0x48000000
44 .equ GPIOA_ODR, GPIOA+0x14
45     PUSH {R4-R7, LR}
46     LDR R4,=GPIOA_ODR
47     MOV R5,#1
48     AND R0,R0,R5
49     LDR R6,[R4]
50     CMP R0,#1
51     BEQ SET_ON
52
53     LDR R7,=0x0DF
54     AND R6,R6,R7
55     STR R6,[R4]
56     B RET_FUNCT
57 SET_ON:
58     LDR R7,=0x20
59     ORR R6,R6,R7
60 RET_FUNCT:
61     STR R6,[R4]
62     POP {R4-R7, PC}
```

FIGURE 9 – LED_DriveGreen.

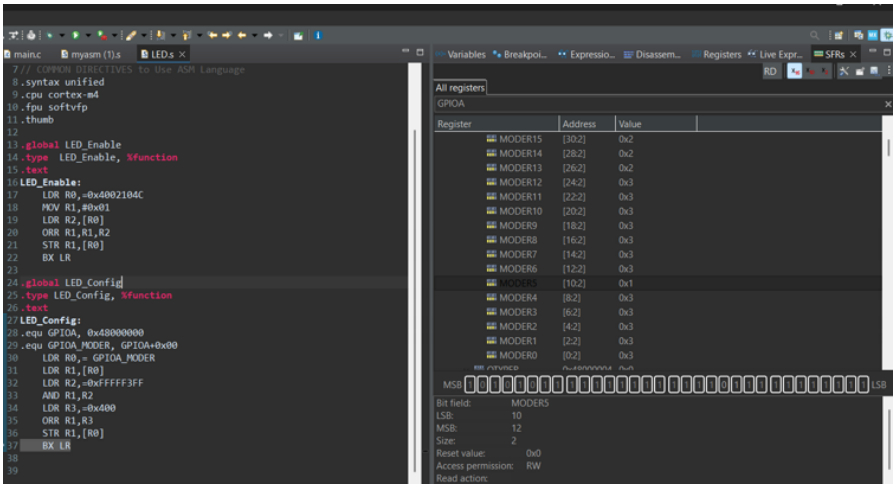



FIGURE 10 – Visualisation du registre MODER de GPIOA dans STM32CubeIDE.

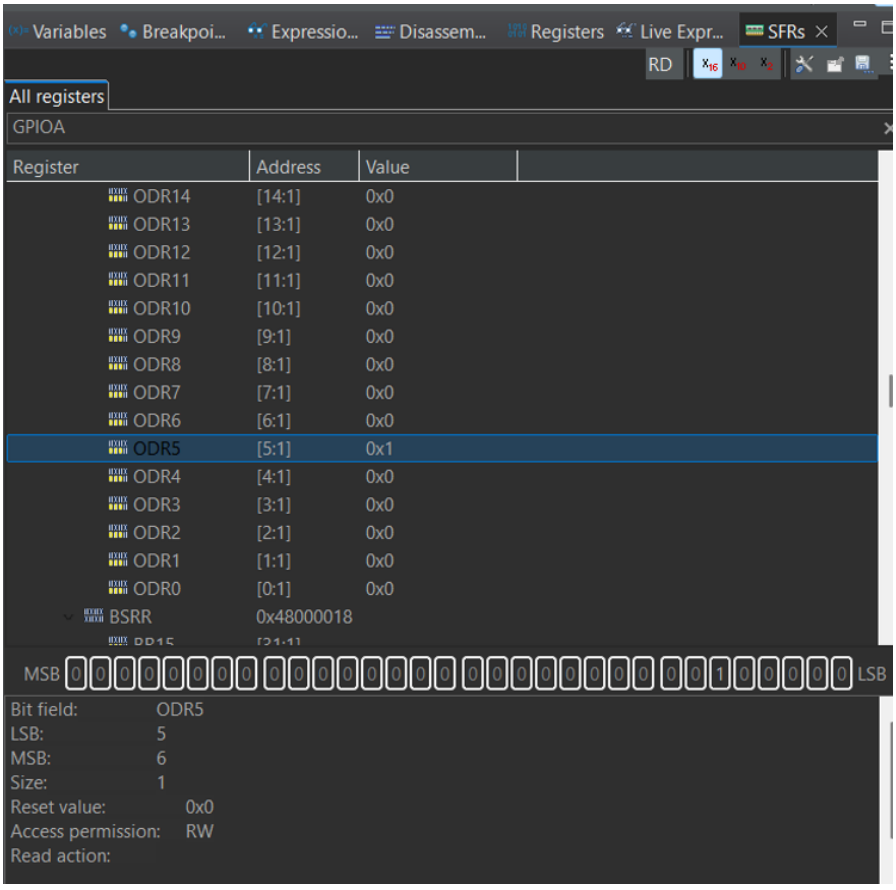



FIGURE 11 – Visualisation du registre ODR de GPIOA dans STM32CubeIDE.

GPIOA est activé par le registre AHB2ENR il est configuré par le GPIOA_MODER ce qui nous permet par la suite de la piloter à travers la modification du GPIOA_ODR (ici elle est allumée).

# Q9 — Green LED Switching State

On crée un fichier `utils.c` contenant deux fonctions : `setup()` et `loop()`, ainsi qu'un fichier `utils.h` avec leurs prototypes. Une variable globale entière est utilisée pour stocker l'état de la LED verte (0 ou 1).

```c
#include "utils.h"
int GreenLED_state = 0;

void setup(void) {
    GreenLED_state = 1;
}
void loop(void) {
    LED_DriveGreen(GreenLED_state);
    GreenLED_state = 1 - GreenLED_state;
    UTILS_WaitN10ms(100);
}


void UTILS_WaitN10ms(int N){
    int n,i,s=0;
    for(n=1;n<=N;n++){
        for(i=0;i<2500;i++){
            s=s+i;
        }
    }
}
```

FIGURE 12 – Ensemble des fonctions.

## Variable globale

```
int GreenLED_state;
```

Cette variable est initialisée dans la fonction `setup()` :

```
GreenLED_state = 0;
```

Puis utilisée dans la fonction `loop()` pour faire clignoter la LED :

```
LED_DriveGreen(GreenLED_state);
GreenLED_state = 1 - GreenLED_state;
```

## Rôle du breakpoint

Un breakpoint permet d'arrêter l'exécution du programme à une ligne précise afin d'observer le comportement du système : valeur des variables, déroulement des instructions, contenu des registres.

Sans breakpoint, le programme s'exécute en continu et la LED clignote normalement, mais on ne peut pas analyser l'exécution pas à pas.

## Configuration Mémoire du Système

— Les variables globales sont situées au début de la RAM.
— La RAM commence à l'adresse $0x20000000$.
— Dans notre système, les entiers (`int`) sont codés sur 32 bits (4 octets).
— Il n'y a aucune autre variable globale à part `GreenLED_State`.

Dans notre cas voici une photo du memory map pour notre projet :

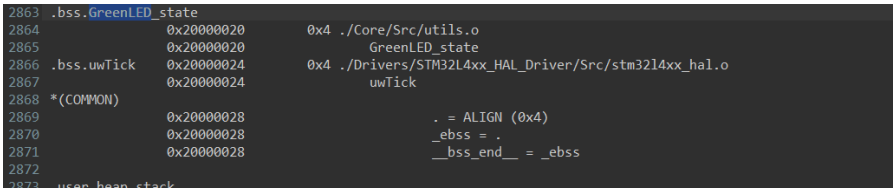



FIGURE 13 – Emplacement de GreenLED_state dans la mémorie.

En assembleur on peut traduire le loop() par ceci :

| Ligne | Instruction | Commentaire |
|---|---|---|
| 1 | `LDR R0, =GreenLED_state` | ; R0 prend l'adresse de la variable d'état |
| 2 | `LDR R1, [R0]` | ; R1 prend la valeur stockée (0 ou 1) |
| 3 | `BL LED_DriveGreen` | ; Appel de la fonction de contrôle hardware |
| 4 | `MOV R2, #1` | ; Charge la constante 1 dans R2 |
| 5 | `SUB R1, R2, R1` | ; R1 = 1 - R1 |
| 6 | `STR R1, [R0]` | ; Sauvegarde le nouvel état en mémoire |

## Commentaire de loop()

Les deux points à observer dans cette traduction en code ASM sont :
— Dans la première ligne l'assembleur associe GreenLED_state à son adresse en mémoire ce qui permet ensuite à R1 de prendre la valeur de GreenLED_state avec l'instruction LDR.
— Les deux lignes après l'appel de LED_DriveGreen servent à exécuter :

— LED_DriveGreen(GreenLED_state) ;
— GreenLED_state = 1 - GreenLED_state ;

Et en effet la LED clignote bien !

# Q10 + Q11 Clignotement de la LED verte – Analyse du code

On utilise une variable globale `GreenLED_state` pour mémoriser l'état de la LED verte. La fonction `setup()` initialise cet état à 1 (LED allumée au départ).

## Fonction `loop()`

La fonction `loop()` réalise le clignotement :
— elle applique l'état courant à la LED avec `LED_DriveGreen(GreenLED_state)` ;
— elle inverse ensuite l'état avec `GreenLED_state = 1 - GreenLED_state` ;
— elle attend un certain temps avec `UTILS_WaitN10ms(100)`.

Comme la fonction d'attente est de 10 ms par unité, l'appel avec 100 donne :

$$100 \times 10\,ms = 1000\,ms$$

La LED change donc d'état toutes les 1 s, soit une période complète ON+OFF de 2 s.

## Fonction d'attente

La fonction d'attente est basée sur deux boucles imbriquées :

```
void UTILS_WaitN10ms(int N){
    int n,i,s=0;
    for(n=1;n<=N;n++){
        for(i=0;i<2500;i++){
            s=s+i;
        }
    }
}
```

Le principe est un délai actif (*busy wait*) :
— aucune minuterie matérielle n'est utilisée ;
— le processeur exécute simplement des additions dans une boucle ;
— la durée dépend directement de la fréquence CPU.

### Précision du délai

Le délai est approximatif (erreur typiquement de l'ordre de 10–20%) car :
— le nombre exact de cycles par itération dépend de la compilation ;
— les branchements de boucle ajoutent des cycles ;
— toute modification de la fréquence CPU change la durée réelle.

Si l'horloge passe de 2 MHz à 80 MHz, le délai devient environ 40 fois plus court, et la LED clignote beaucoup plus vite si on ne retouche pas les constantes.

## Q13 – Création des fichiers pour le bouton

On crée deux fichiers : `button.c` dans `Src/` et `button.h` dans `Inc/`. Le fichier d'en-tête contient les prototypes :

```
#ifndef BUTTON_H
#define BUTTON_H

void BUTTON_Enable(void);
int BUTTON_GetBlueLevel(void);

#endif
```

## Q15 – Fonction BUTTON_Enable avec pointeurs

Implémentation avec pointeur direct sur le registre RCC :

```
void BUTTON_Enable(void){
    int* RCC_AHB2ENR = (int*)0x4002104C;
    *RCC_AHB2ENR = (*RCC_AHB2ENR) | 0x4;    // active l'
       horloge GPIOC (bit 2)
}
```

Effet : le registre `RCC_AHB2ENR` est modifié pour activer le port GPIOC, indispensable avant toute configuration ou lecture du bouton.

## Q16 – Fonction BUTTON_Config (structures stm32l4xx)

La fonction `BUTTON_Config()` configure le pin du bouton (PC13) en entrée avec résistance de tirage interne (pull-up) en utilisant les structures définies dans `stm32l4xx.h` comme `GPIOx->MODER` et `GPIOx->PUPDR`.

— Bits 27 :26 de MODER = 00 ⇒ mode entrée
— Bits 27 :26 de PUPDR = 01 ⇒ pull-up activé

```
void BUTTON_Config(void){
  GPIOC->MODER &= 0xF3FFFFFF;    // bits 27:26 = 00 -> input
  GPIOC->PUPDR &= 0xF3FFFFFF;    // remise a 00
  GPIOC->PUPDR |= 0x04000000;    // bit 26 = 1 -> pull-up
}
```

Test : en observant les registres `MODER` et `PUPDR` dans le debugger, on vérifie que les bits du pin 13 sont correctement positionnés.

# Q17 – Fonction BUTTON_GetBlueLevel (HAL)

La fonction lit l'état logique du bouton via la fonction HAL `HAL_GPIO_ReadPin`. Le bouton étant en pull-up, l'entrée vaut 0 quand il est appuyé.

```
int BUTTON_GetBLueLevel(void){
  if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET)
     {
    return 1;    // bouton appuye
  }
  else {
    return 0;    // bouton relache
  }
}
```

## Test simple

Un test minimal consiste a appeler la fonction dans la boucle principale et a placer un point d'arret :

```
int level = BUTTON_GetBLueLevel();
```

En appuyant / relachant le bouton, la valeur de `level` doit passer de 0 a 1 dans le debugger, ce qui valide le bon fonctionnement.

## Synthèse

Ce TP a permis de comparer plusieurs niveaux d'accès aux périphériques STM32 :
— Assembleur : contrôle total mais complexité élevée

— C avec pointeurs : plus simple mais risqué

L'analyse du code assembleur montre le respect des conventions ARM : passage des paramètres par registres, sauvegarde/restauration des registres et retour par le PC.