Tutoriels STM32F0

Périphériques standards

Arouna DARGA – enseignant chercheur Sorbonne Université
arouna.darga@sorbonne-universite.fr

Table des matières

1. General Purpose Input Output - GPIO

1.1 Introduction

Ce tutoriel est consacré à la mise en œuvre de quelques fonctions du **BSP**. Dans le monde des systèmes embarqués, **BSP** signifie "**Board Support Package**". En termes simples, il s'agit d'un ensemble de fonctions permettant la gestion des périphériques embarqués tels que les entrées/sorties, les afficheurs, les capteurs, les interfaces de communication...

Un ensemble de fonctions qui simplifie l'interface logicielle avec un périphérique donné est également appelé bibliothèque, ou pilote.

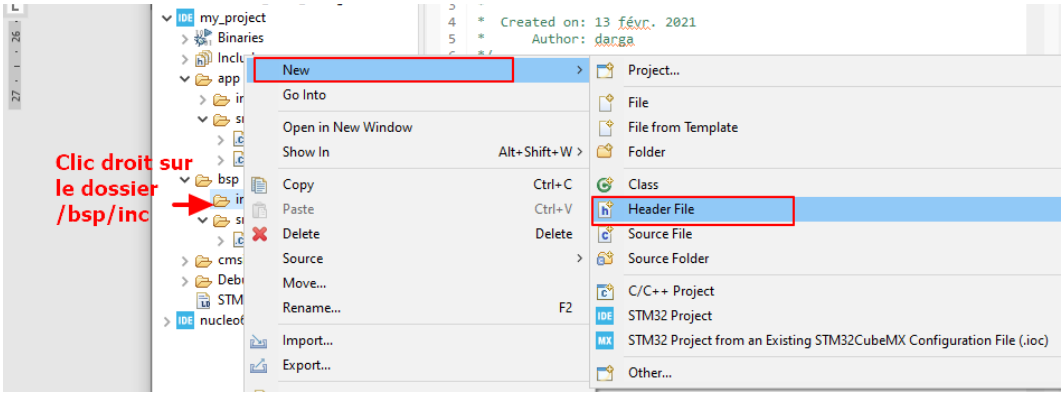
Les cartes Nucleo sont plutôt démunies en termes d'accessoires. En gros, vous disposez d'une LED, d'un bouton utilisateur et d'une interface série via le dongle ST-Link.

Les LED et les boutons sont directement interfacés avec les broches du MCU en mode sortie ou entrée. Ceux-ci sont configurés au moyen du périphérique **GPIO** (**General Purpose Input Output**). GPIO est le périphérique du MCU qui gère la configuration des broches et les entrées/sorties « Tout ou Rien » ou Numériques.

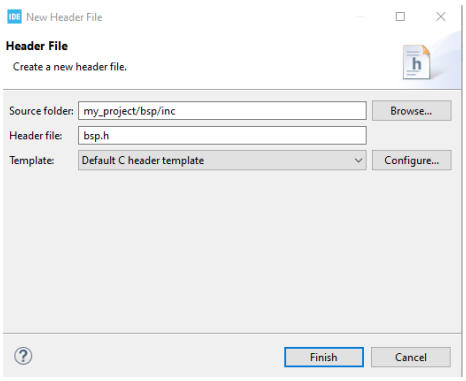
Lancez **STM32CubeIDE** et ouvrez le projet "***my_project*".** À cette étape, " ***my_project*** " est un projet de départ vide que vous avez créé dans le tutoriel **clonage d'un projet**, en dupliquant le projet modèle ***nucleo64_F072RB_template***. Le projet " ***my_project*** " sera utilisé par la suite dans les autres tutoriels qui surferont. Nous allons construire au fur et à mesures une bibliothèque ou librairie ou pilote.

Créez un nouveau fichier source **bsp.c** dans le dossier **bsp/src**, et un nouveau fichier d'en-tête **bsp.h** dans le dossier **bsp/inc** :

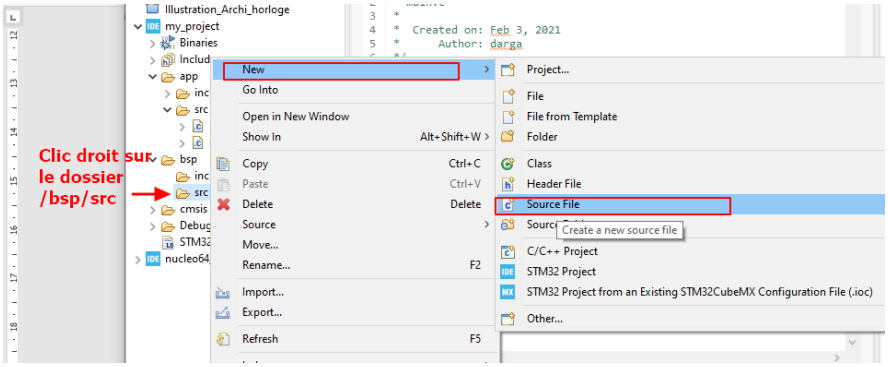
Ajout du fichier d'en-tête **bsp.h** : clic droit sur le dossier **bsp/inc** → **new** → **Header File**



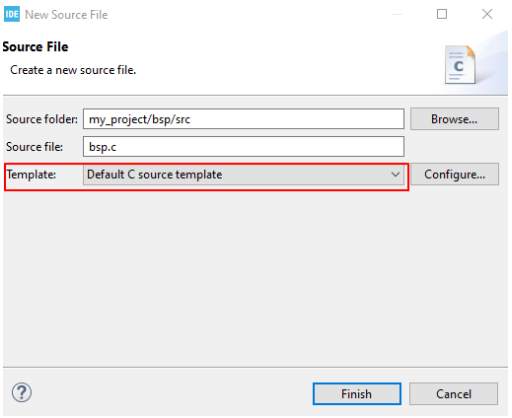
Lors de la création du fichier, garder les options (***default header template***) par défaut comme sur l'image ci-dessous



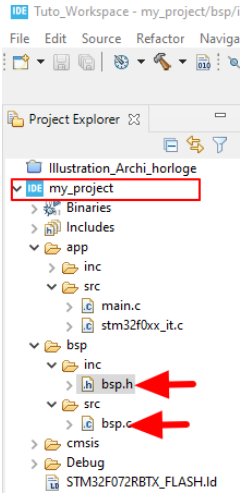
Ajout du fichier source **bsp.c** : clic droit sur le dossier **bsp/src** → **new** → **source File**



Lors de la création du fichier, garder les options (***default C source template***) par défaut comme sur l'image ci-dessous

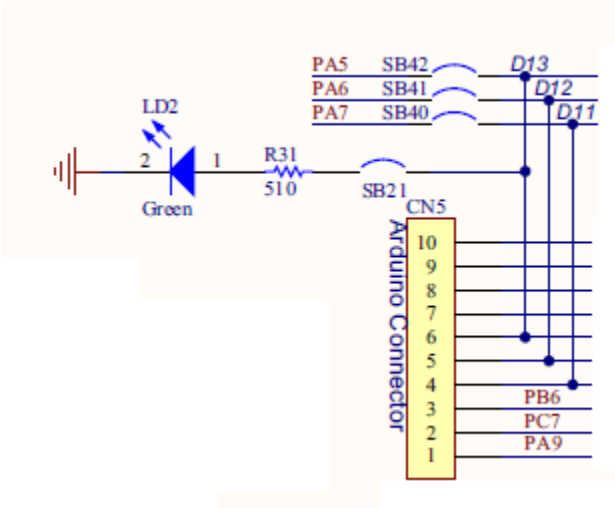


Votre explorateur de projets devrait ressembler à celui de l'image ci-dessous.



1.2 Driver de LED

Un coup d'œil aux schémas de la carte nous indique que la LED verte est connectée à la broche **PA5** du MCU. Étant donné que la cathode de la LED est mise à la terre, la LED est "ON" lorsque **PA5** est à sa tension logique élevée.



Prenons un moment pour réfléchir aux fonctions que nous aimerions avoir afin d'utiliser la LED :

- Une fonction qui permet d'allumer la LED
- Une fonction qui permet d'éteindre la LED
- Une fonction qui fait basculer (Toogle) l'état de la LED

Pour l'instant, nous ferons de fonctions simples : pas besoin d'argumenter (il n'y a qu'une seule LED), et ces fonctions peuvent ne rien renvoyer.

Notez qu'il serait possible d'utiliser une seule fonction qui règle l'état de la LED en utilisant un argument (ON/OFF/Toggle).

Ouvrez le fichier bsp**.h** et écrivez les prototypes de fonction suivants :

```
/*
 * bsp.h
 *
 *  Created on: 14 févr. 2021
 *      Author: darga
 */

#ifndef BSP_INC_BSP_H_
#define BSP_INC_BSP_H_

#include "stm32f0xx.h"

/*
 * fonctions du driver LED
 */

void  BSP_LED_Init      (void);
void  BSP_LED_On  (void);
void  BSP_LED_Off (void);
void  BSP_LED_Toggle    (void);



#endif /* BSP_INC_BSP_H_ */
```

Notez que lors de la création de nouveaux fichiers en-têtes, STM32CubeIDE protège automatiquement le code contre l'inclusion récursive :

```
#ifndef HEADERNAME_H_
#define HEADERNAME_H_
...
#endif
```

Il est fortement conseillé de le conserver...

L'inclusion récursive correspond à la situation ci-dessous, qui produirait une boucle d'inclusion infinie pendant le pré-processus de construction si elle n'était pas sous la protection des directives ci-dessus :

```
/*
 * header_A.h
 */

#include "header_B.h"

...
```

```
/*
 * header_B.h
 */

#include "header_A.h"

...
```

Ouvrez le fichier **bsp.c** et commencer la mise en œuvre des fonctions. Adopter une stratégie claire dans le nommage des fonctions et des variables. C'est TRÈS important lors de la phase de développement du code.

Dans ce qui suit, la fonction concerne le **BSP**, la **LED**, et son but est l'initialisation de la broche MCU. Nommons la fonction **BSP_LED_Init()** :

```
/*
 * bsp.c
 *
 *  Created on: 14 févr. 2021
 *      Author: darga
 */


#include "bsp.h"

/*
 * BSP_LED_Init()
 * Initialise la broche PA5 (LED) en sortie, mode Push-pull, vitesse de
rafraichissement High-Speed, pas de resistance de tirage
 * Etat initial mus a '0'
 */

void BSP_LED_Init()
{
	//activation de horloge du peripherique GPIOA
	// mettre le bit b17 du registre RCC_AHBENR a '1'
	// voir page 128 du manuel technique (User Manuel) du Microcontrôleur
STM32F072RB
	// le bit b17 de RCC_AHBENR est egalement defini =  RCC_AHBENR_GPIOAEN dans
le fichier stm32f0xx.h
	RCC->AHBENR |= (1<<17);   // ou  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

	// Configurer la broche PA5 en sortie
	// Ecrire "01" sur les bits b11b10 du registre (GPIOA_MODER)
	GPIOA->MODER &= ~(1<<11);
	GPIOA->MODER |= (1<<10);

	// Choisir option electronique Push-pull pour la sortie PA5
	// Ecrire "0" sur le bit b5 du registre (GPIOA_OTYPER)
	GPIOA->OTYPER &= ~(1<<5);

	// Choisir option electronique high speed pour la vitesse de
rafraichissement pour la sortie PA5
	// Ecrire "11" sur les bits b11b10 du registre (GPIOx_OSPEEDR)
	GPIOA->OSPEEDR |= (1<<11);
	GPIOA->OSPEEDR |= (1<<10);

	// Deactiver les resistances de tirages (Pull-up ou Pull-down) pour la
sortie PA5
	// Ecrire "00" sur les bits b11b10 du registre (GPIOx_PUPDR)
	GPIOA->PUPDR &= ~(1<<11);
	GPIOA->PUPDR &= ~(1<<10);

	// Mettre l'état initial de PA5 sur OFF
	GPIOA->ODR &= ~(1<<5);
}
```

Vous devez vous référer au manuel de référence pour une description complète du registre **RCC AHBENR**, et des registres **GPIO MODER, OTYPER, OSPEEDR, PUPDR, ODR**.

Viennent ensuite les trois fonctions de contrôle :

```
/*
 * BSP_LED_On()
 * Allume la LED lier a la broche PA5
 */

void BSP_LED_On()
{
	// Ecrire "1" sur le bit b5 du registre (GPIOA_ODR)
	GPIOA->ODR |= (1<<5);
}

/*
 * BSP_LED_Off()
 * Eteint la LED lier a la broche PA5
 */

void BSP_LED_Off()
{
	// Ecrire "0" sur le bit b5 du registre (GPIOA_ODR)
	GPIOA->ODR &= ~(1<<5);
}

/*
 * BSP_LED_Toggle()
 * Change etat de la LED lier a la broche PA5
 */

void BSP_LED_Toggle()
{
	GPIOA->ODR ^= (1<<5);
}
```

Testez les fonctions des LED dans le code principal. L'exemple ci-dessous doit être utilisé avec le mode pas à pas du débogueur, sinon le changement d'état est trop rapide pour être vu (à moins de faire une vérification) de PA5 avec un oscilloscope.

```
/*
 * main.c
 *
 *  Created on: Feb 3, 2021
 *      Author: darga
 */

#include "stm32f0xx.h"
#include "bsp.h"

static void SystemClock_Config(void);

int main()
{
	// Configure System Clock
	SystemClock_Config();

	// Initialize LED pin
	BSP_LED_Init();

	// Turn LED On
```

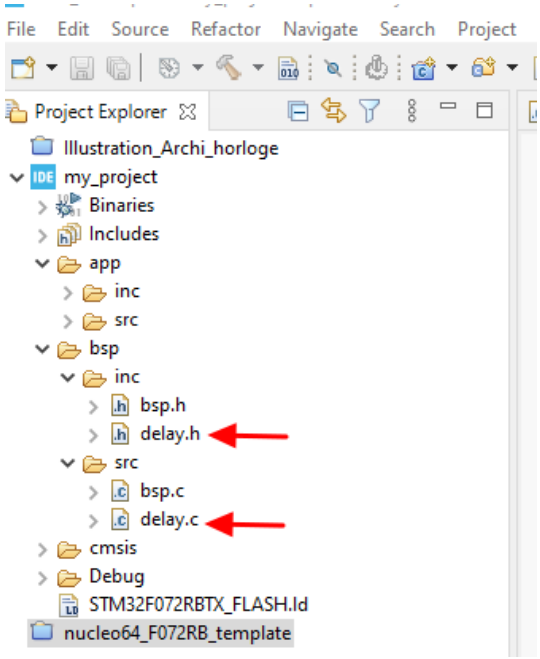
```
	BSP_LED_On();

	// Turn LED Off
	BSP_LED_Off();

	while(1)
	{
		// Toggle LED state
		BSP_LED_Toggle();
	}
}



/*
 *	Clock configuration for the Nucleo STM32F072RB board
 *	HSE input Bypass Mode		-> 8MHz
 *	SYSCLK, AHB, APB1			-> 48MHz
 *	PA8 as MCO with /16 prescaler		-> 3MHz
 *
 */

static void SystemClock_Config()
{
	……..
}
```

2. Délais d'attente software

Dans la première partie de ce tutoriel, nous mettrons en œuvre une fonction de délai naïve basée sur une boucle de comptage logicielle.

2.1 Implémentation du délai

Ouvrez le projet my**_projet** et ajoutez un nouveau fichier de code source **delay.c** et un nouveau fichier d'en-tête **delay.h** dans votre dossier bsp :



Supposons que nous voulons deux fonctions de temporisation : une pour la gamme des millisecondes, une pour la gamme des microsecondes. Ajoutez les deux prototypes de fonctions dans le fichier d'en-têtes **delay.h** :

```
/*
 * delay.h
 *
 *  Created on: 15 févr. 2021
 *      Author: darga
 */

#ifndef BSP_INC_DELAY_H_
#define BSP_INC_DELAY_H_

#include "stm32f0xx.h"

/*
 * Delais attentes ou temporisation Software
 */

void BSP_DELAY_ms (uint32_t delay);
void BSP_DELAY_us (uint32_t delay);


#endif /* BSP_INC_DELAY_H_ */
```

Ensuite il faut implémenter ces fonctions dans le fichier code source **delay.c** :

```
/*
 * delay.c
 *
 *  Created on: 15 févr. 2021
 *      Author: darga
 */

#include "delay.h"

/*
 *  Fonctions Basiques pour temporisation software
 */

void BSP_DELAY_ms(uint32_t delay)
{
      uint32_t    i;
      for(i=0; i<(delay*6000); i++) ;
   // adapter la valeur du coef selon la frequence du CPU
}

void BSP_DELAY_us(uint32_t delay)
{
      uint32_t    i;
      for(i=0; i<(delay*10); i++);
    // adapter la valeur du coef selon la frequence du CPU
}
```

A l'aide d'un oscilloscope (analog Discovery) réglez les valeurs des coefficients multiplicatifs pour obtenir les retards attendus.

Testez vos fonctions maintenant :

```
/*
 * main.c
 *
 *  Created on: Feb 3, 2021
 *      Author: darga
 */

#include "stm32f0xx.h"
#include "bsp.h"
#include "delay.h"

static void SystemClock_Config(void);


int main()
{
	// Configure System Clock
	SystemClock_Config();

	// Initialize LED pin
	BSP_LED_Init();

	// Main loop
	while(1)
	{
		BSP_LED_Toggle();
		BSP_DELAY_ms(500);
	}
}
```

2.2 Robustesse des délais

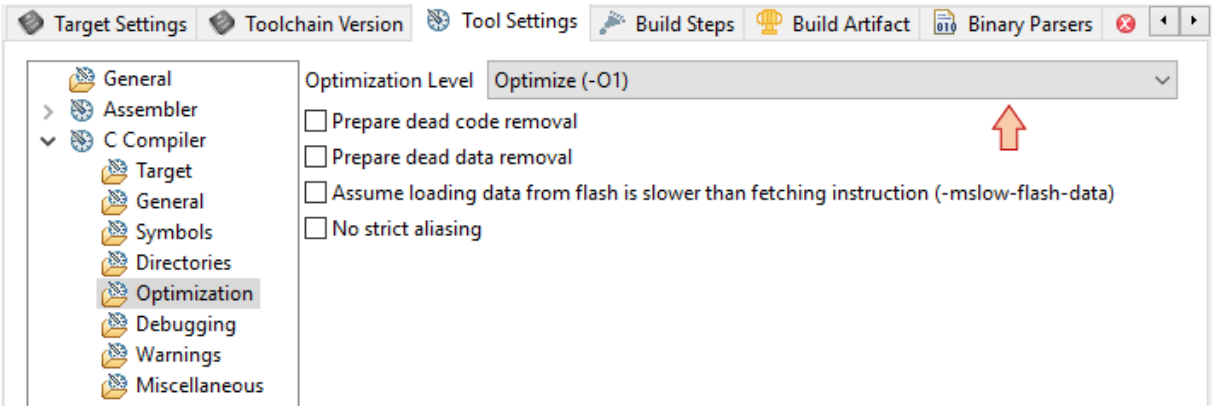
Examinons de plus près ce qui est réellement fait lorsqu'un retard est appelé. Démarrez une session de débogage et avancez dans le code pour atteindre le début de la fonction **BSP_DELAY_ms()** :

```
14 void BSP_DELAY_ms(uint32_t delay)
15 {
16     uint32_t    i;
17     for(i=0; i<(delay*2500); i++);      // Tuned for ms at 48MHz
18 }
19
```

Ouvrez la fenêtre (Vue) **Disassembly** et vous verrez que la boucle de comptage se traduit par 9 lignes de code assembler pour effectuer essentiellement les opérations requises. Nous pouvons voir qu'il y a un **opcode muls** dans la boucle, on peut donc supposer que la multiplication du **délai*2500** est répétée à chaque fois avant que la comparaison ne soit effectuée.

```
080005e2:   ldr     r3, [r7, #12]
080005e4:   adds    r3, #1
080005e6:   str     r3, [r7, #12]
080005e8:   ldr     r3, [r7, #4]
080005ea:   ldr     r2, [pc, #16]   ; (0x80005fc <delay_ms+40>)
080005ec:   muls    r2, r3
080005ee:   ldr     r3, [r7, #12]
080005f0:   cmp     r2, r3
080005f2:   bhi.n   0x80005e2 <delay_ms+14>
```

Terminez la session debug et ouvrez « project Properties ». Allez dans la catégorie **C/C++ Build → Settings** et ouvrir le tab **Tool Settings → Optimization** et fixer la valeur de **Optimization Level** à **-O1**.



Cliquez sur **Apply & Close** pour valider et fermer la fenêtre.

Nettoyer (CLEAN) le projet (c'est important, sinon le nouveau niveau d'optimisation n'est pas appliqué), puis le reconstruire (build, compilation). Exécutez le programme et regardez la LED. Que voyez-vous ? Elle clignote plus vite qu'avant...

Redémarrer une session de débogage. Faites un pas jusqu'à ce que vous atteigniez à nouveau la fonction **BSP_DELAY_ms()**. Regardez maintenant le désassemblage. La même boucle de comptage se traduit par 3 lignes de montage seulement. Le produit 500*2500=1250000 est maintenant stocké une fois dans r0 avant la boucle. Intelligent !

```
080004b6:   adds    r3, #1
080004b8:   cmp     r3, r0
080004ba:   bne.n   0x80004b6 <delay_ms+10>
```

De plus, la variable ***i*** n'existe plus en mémoire. Sa valeur est directement détenue par le registre r3 du CPU. Essayez d'ajouter &i dans la vue Expressions et vous obtiendrez ceci :

```
Failed to execute MI command:
-data-evaluate-expression &i
Error message from debugger back end:
Address requested for identifier "i" which is in register $r3
```

Répétez les étapes ci-dessus avec l'optimisation de niveau -O2. Si vous entrez dans le code d'assemblage, vous découvrirez que la boucle de comptage a été complètement supprimée par le compilateur... bizarre !

Pourquoi une boucle qui ne fait rien après tout...

Le fait est que le niveau d'optimisation change la façon dont votre code C se traduit en code assembleur, le rendant de plus en plus efficace à mesure que le niveau d'optimisation augmente. L'utilisation d'un niveau d'optimisation différent de -O0 nécessite une attention particulière car l'optimiseur supprime tout ce qui est considéré comme "inutile" (y compris les opérations et les variables).

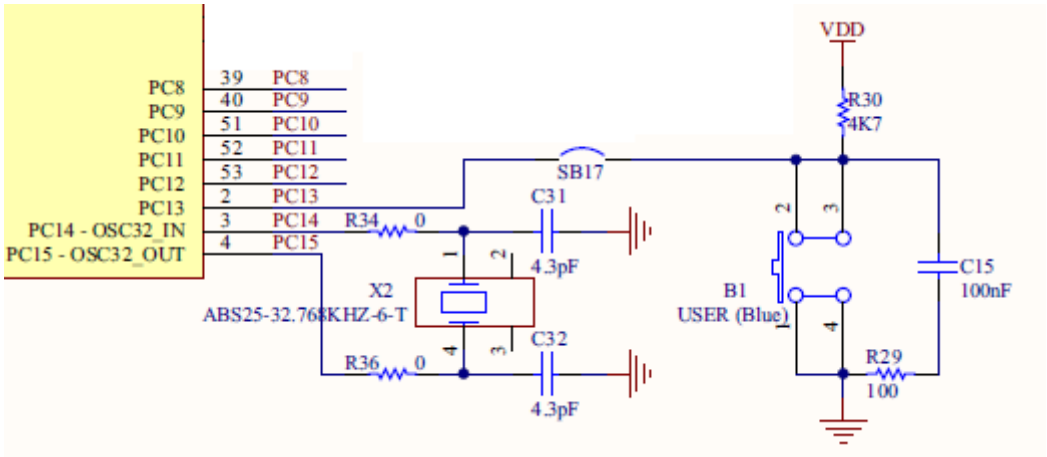
Un code qui est inutile pour l'optimiseur ne l'est pas toujours pour vous. Un message de débogage par exemple, ou des variables intermédiaires peuvent disparaître. Donc, faites attention aux optimisations, et c'est toujours une bonne idée de commencer à écrire du code sans optimisations. De cette façon, vous êtes sûr de trouver ce que vous attendez du débogueur.

L'objectif de l'expérience ci-dessus était de montrer clairement que la synchronisation basée sur le comptage de logiciels est très dépendante de paramètres externes tels que l'optimisation du code, ou la source/fréquence de l'horloge. Il est donc conseillé de mettre en œuvre des délais avec des approches plus robustes.

Pour l'instant, revenez au niveau **d'optimisation -O0**, nettoyez et reconstruisez le projet.

3. Driver de bouton poussoir

Encore une fois, nous commençons par les schémas de la carte. Le bouton utilisateur est un interrupteur qui relie la broche du PC13 à la terre lorsqu'il est enfoncé. Sinon, la broche PC13 est maintenue à un niveau logique élevé au moyen de la résistance de pull-up R30.



Afin d'interfacer le bouton-poussoir, nous allons écrire deux fonctions :

- Une fonction pour initialiser le PC13 comme broche d'entrée
- Une fonction qui renvoie l'état du bouton (0 pour relâché, 1 pour appuyé)

Vous pouvez ajouter ces prototypes de fonctions dans le fichier en-têtes **bsp.h**, en dessous des fonctions LED.

```
/*
 * fonctions du driver de bouton poussoir
 */

void    BSP_PB_Init     (void);
uint8_t BSP_PB_GetState (void);


#endif /* BSP_INC_BSP_H_ */

```

Modifier l'implémentation des fonctions dans **bsp.c** :

```
/*
 * BSP_PB_Init()
 * Initialise la broche PC13 (Bouton poussoir) en input sans les
resistance de tirages Pull-up/Pull-down
 */

void BSP_PB_Init()
{
	//activation de horloge du peripherique GPIOC
	// mettre le bit b19 du registre RCC_AHBENR a '1'
	RCC->AHBENR |= (1<<19);

	// Configurer la broche PC13 en Input
	// Ecrire "00" sur les bits b27b26 du registre (GPIOC_MODER)
	GPIOC->MODER &= ~(1<<27);
	GPIOC->MODER  &= ~(1<<26);
```

```
	// Deactiver les resistances de tirages (Pull-up ou Pull-down) pour
la sortie PA5
	// Ecrire "00" sur les bits  b27b26 du registre (GPIOC_PUPDR)
	GPIOC->PUPDR &= ~(1<<27);
	GPIOC->PUPDR  &= ~(1<<26);

}

/*
 * BSP_PB_GetState()
 * Renvoi etat du bouton (0=Non Appuyer, 1=Appuyer)
 */

uint8_t BSP_PB_GetState()
{
	uint8_t state;

	if ((GPIOC->IDR & (1<<13)) == (1<<13))
	{
		state = 0;
	}
	else
	{
		state = 1;
	}

	return state;
}
```

Ensuite, testez vos nouvelles fonctions dans le code **main.c** :

```
#include "stm32f0xx.h"
#include "bsp.h"


static void SystemClock_Config(void);

int main()
{
	// Configure System Clock
	SystemClock_Config();

	// Initialize LED pin
	BSP_LED_Init();
	// Initialize User-Button pin
	BSP_PB_Init();

	// Turn LED On
	BSP_LED_On();

	// Turn LED Off
	BSP_LED_Off();

	while(1)
	{
		// Turn LED On if User-Button is pushed down

		if (BSP_PB_GetState() == 1)
		{
			BSP_LED_On();
		}
		// Otherwise turn LED Off
		else
		{
			BSP_LED_Off();
		}
	}
}
```