

## Travaux Pratiques

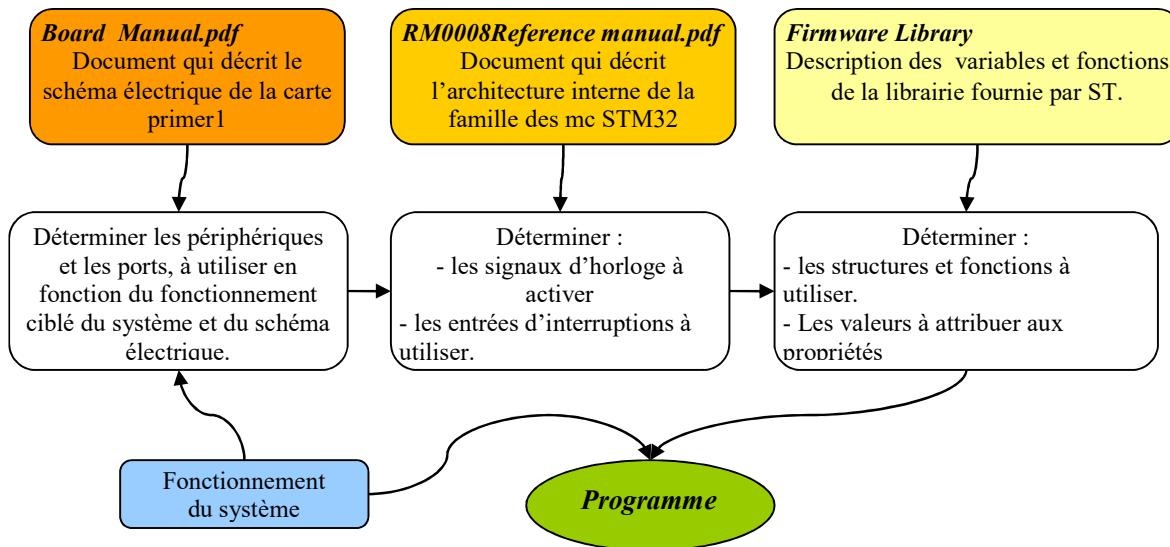
### I- INTRODUCTION

Contrairement au développement d'applications pour les PCs ou un programme peut être ‘généralement’ exécuté sur n’importe quelle machine à base d’un microprocesseur compatible x86, le développement d’une application embarqué est plus compliqué.

Rien ne garantit qu’une application fonctionnant sur une plateforme matérielle puisse fonctionner sur une autre. D'où la nécessité d'adapter le programme :

- A la structure interne du microcontrôleur utilisé.
- Au schéma électrique de la carte à programmer (connexions entre pins du microcontrôleur et composants externes tels que les leds les boutons poussoirs, afficheurs, etc.. .
- A la librairie de fonctions supportées par l'environnement de développement.

Ainsi, pour la programmation de la plateforme, il est indispensable d'utiliser principalement 3 documents conjointement :

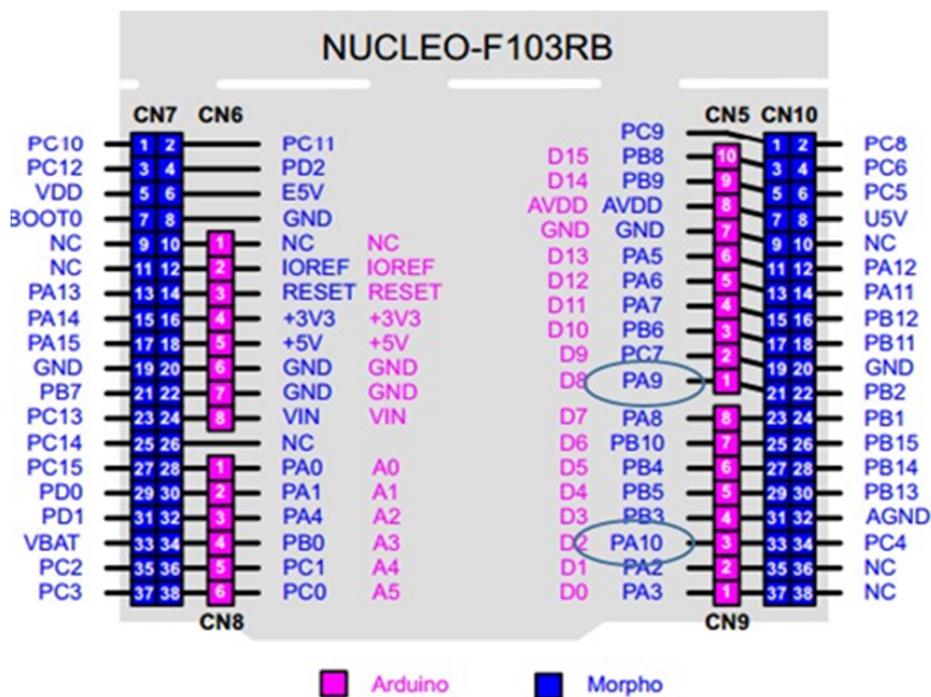
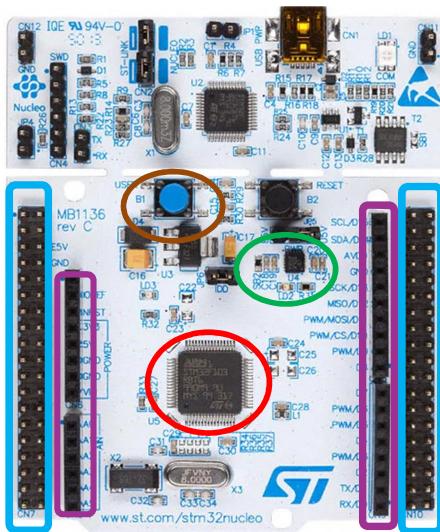


## II - Généralités

### II-1 : Les plateformes utilisées

Au cours des séances de TP, plusieurs cartes à base de microcontrôleurs STM32 seront utilisées :

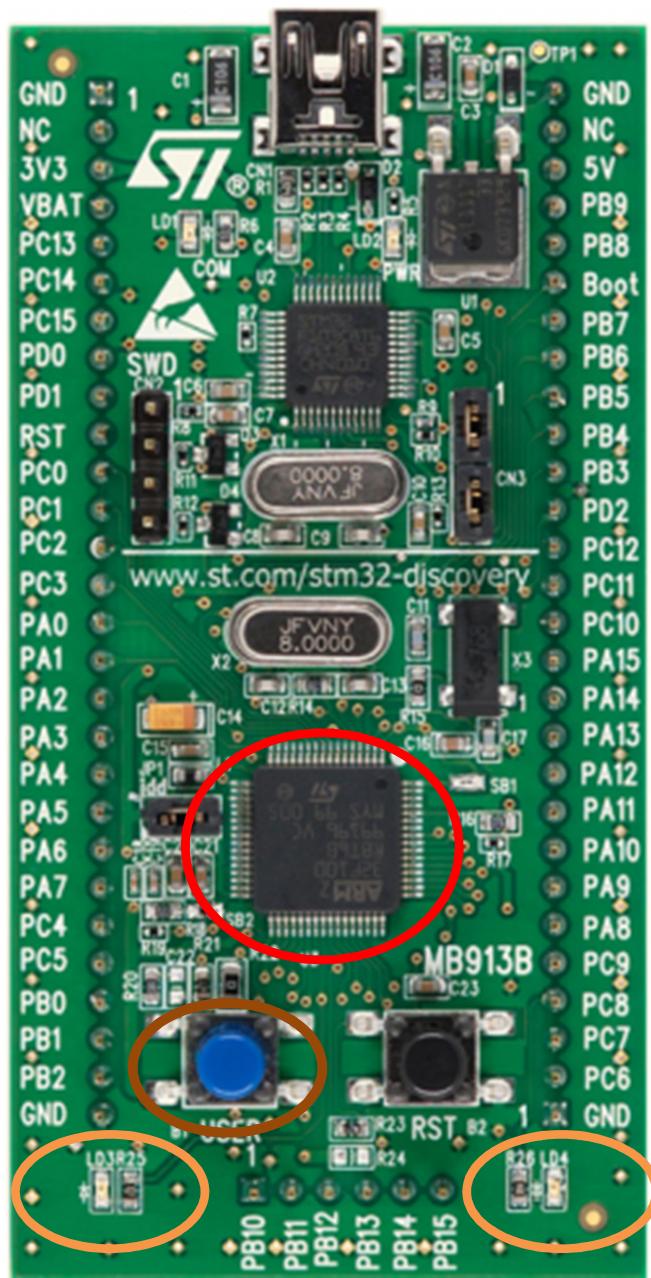
- 1- Les cartes NUCLEO F103 (à base de *microcontrôleurs STM32F103RB*)** : qui possèdent des connecteurs compatibles avec les cartes d'extension pour arduino. On trouve :
- Une **LED** reliée au pin **PA5**
  - Un **Bouton poussoir** relié au pin **PC13**
  - Un adaptateur de communication USB/série USART



## 2- Les cartes Discovery (à base de microcontrôleurs STM32F100RB)

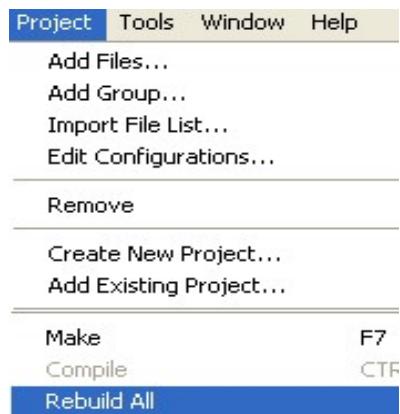
On trouve :

- 2 Leds reliées aux pins **PC8 et PC9**.
- Un **bouton poussoir** relié au pin **PA0**.

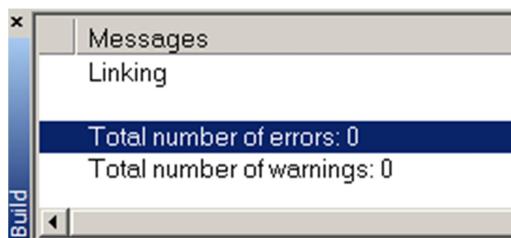


## ***II-2 : Utilisation de l'IDE keil***

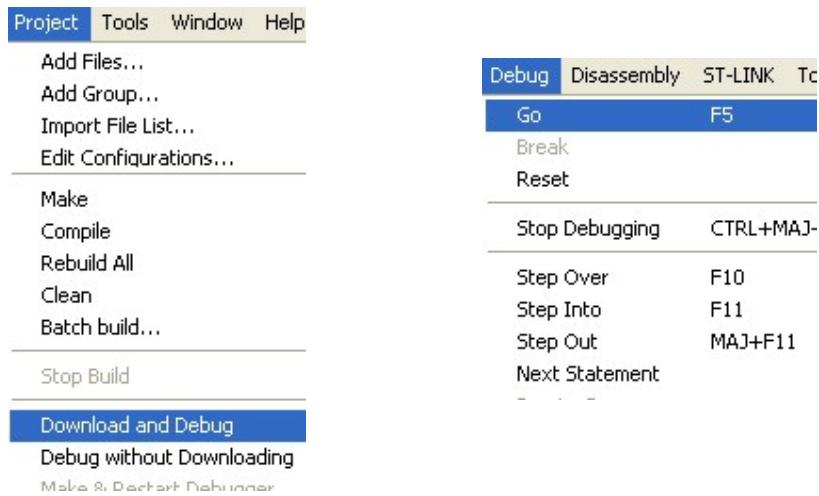
- 1 -** Commencer par ouvrir le projet contenant la manip **i**, i étant le numéro de la manip. Son chemin est : « **Projets / manipi / manipi.uvprojx** »
- 2-** Compléter le code permettant d'assurer le fonctionnement décrit dans le paragraphe relatif à la manip.
- 3-** Une fois le code terminé, le compiler avec l'outil IAR pour générer l'exécutable en utilisant la commande **Project>>Rebuild All** (voir figure ci-dessous).



Si tout se passe bien, on doit obtenir l'affichage suivant au niveau de la fenêtre « **Messages** » en bas de l'IDE de l'outil IAR. Sinon, des messages d'erreurs seront affichés.



4- Finalement il est possible de charger le programme sur la plateforme cible (**Project>>Download and Debug**) et l'exécuter (**Go**) :



**Rq :** Il est possible qu'un programme ne fonctionne pas comme prévu (même si la compilation est réussie). Dans ce cas, il est possible de trouver les erreurs en procédant au *débogage* du programme : il s'agit de l'exécuter pas à pas ou en insérant des *breakpoints*. A chaque arrêt du programme, il est possible de consulter les contenus des registres du microcontrôleur ainsi que les valeurs des variables utilisées et de vérifier s'ils correspondent au fonctionnement désiré.

### III - Les Manips

#### **1- Programmation par accès direct aux registres**

L'objectif est la familiarisation avec l'environnement de développement intégré Keil ainsi que le principe de fonctionnement et de programmation des interfaces de périphériques par accès direct aux registres.

**MANIP 1-1 :** On veut développer un programme ‘C’ qui permet de faire clignoter les deux Leds de la carte à base du microcontrôleur STM32 F1.

- Indiquer les pins à utiliser et trouver les adresses des registres des ports auxquels sont connectées les Leds ainsi que les positions des bits qui permettent de contrôler les pins. (consulter le ***RM0008Reference manual.pdf***).
- Ecrire le programme en ‘C’ en utilisant l'accès direct aux registres (Consulter le « ***chapitre 2 : entrées – sorties GPIO du STM32*** » du cours).
- compiler le programme et le charger dans la carte afin de l'exécuter et tester son bon fonctionnement .

**MANIP 1-2 :** On veut développer un programme en ‘C’ en utilisant l'accès direct aux registres et qui permet d'allumer les Leds quand on appuie sur le bouton poussoir et de les éteindre quand on le relâche.

- Préciser les pins à configurer (Leds, Bouton) et trouver les adresses du port auquel est connecté le bouton poussoir ainsi que les positions des bits qui permettent de contrôler les pins. (consulter le ***RM0008Reference manual.pdf***).
- Ecrire le programme en ‘C’ en utilisant l'accès direct aux registres (Consulter le « ***chapitre 2 : entrées – sorties GPIO du STM32*** » du cours).
- Compiler le programme et le charger dans la carte afin de l'exécuter et tester son bon fonctionnement.

## **Manip 1-3 : développement d'une couche d'abstraction matérielle**

### **MANIP 1-3 (Partie A):**

On veut développer un programme qui assure le même fonctionnement que manip\_1-2. Mais, cette fois, en faisant abstraction des adresses des registres au niveau de la configuration des GPIOs et utiliser des instructions d'accès ayant la forme **PERIPH → Registre** (exemple GPIOA→CRL) pour accéder au registre CRL du GPIOA. Pour cela :

- Au niveau du fichier « **stm32map.h** » ajouter le code (*structure et pointeurs*) permettant de décrire le **Memory Map** (Cartographie) relatif au GPIOs (A , B , C et D).
- Au niveau du fichier « **gpio.h** », ajouter les données relatives aux paramètres GPIO : (Les Pins, Les Modes). Ainsi que les prototypes des fonctions déclarées dans **gpio.c**
- Au niveau du fichier « **gpio.c** » ajouter le code des ***fonctions*** suivantes :

uint8\_t **GPIO\_TestPin** (GPIOx, GPIO\_Pin) : qui retourne l'état (0 ou 1) du Pin passé en paramètre

void **GPIO\_SetPin** (GPIOx, GPIO\_PIN) : qui met à 1 le(s) Pin(s) passé(s) en paramètre

void **GPIO\_ResetPin** (GPIOx, GPIO\_PIN) : qui met à 1 le(s) Pin(s) passé(s) en paramètre

- Ajouter le code permettant d'assurer le fonctionnement voulu au niveau du fichier « **main.c** ».Compiler le programme et le charger dans la carte afin de l'exécuter et tester son bon fonctionnement.

### **MANIP 1-3 (Partie B):**

On veut améliorer davantage l'abstraction en ajoutant une fonction qui permet d'assurer la configuration des modes des pins du GPIO qui décharge le programmeur des opérations de masquage, décalage, etc...

- 1- Commencer par ajouter au fichier **gpio.h** une structure **GPIO\_Struct** contenant les paramètres de configuration (pins, mode) :

```
Typedef struct {
    uint16_t GPIO_PIN ;
    uint8_t GPIO_Mode ;
} GPIO_Struct ;
```

**2-** Ainsi au niveau de **main.c**, la configuration se fera en initialisant une variable de type

GPIO\_Struct et en faisant appel à une fonction **GPIO\_Init** qui prend en paramètres :

- La variable de type GPIO\_Struct (les pins, ainsi que le mode)
- Le GPIO à configurer.

a) Ajouter la variable de type GPIO\_Struct

```
GPIO_Struct GPIO_InitStruct ;
```

b) Remplacer le code de configuration des pins (partie 3.1), par :

```
GPIO_InitStruct.GPIO_PIN = GPIO_PIN_0 ;  

GPIO_InitStruct.GPIO_Mode = GPIO_Mode_InputFloating ;  

GPIO_Init (GPIOA, &GPIO_InitStruct) ;
```

```
GPIO_InitStruct.GPIO_PIN = GPIO_PIN_8|GPIO_PIN_9 ;  

GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OutputPP2Mhz;  

GPIO_Init (GPIOC, &GPIO_InitStruct) ;
```

**3-** Ajouter au niveau de **gpio.c**, le code de la fonction permettant la configuration des pins d'un GPIO:

```
GPIO_Init (GPIO_TypeDef* GPIOx, GPIO_Struct GPIO_InitStruct)
```

//rq: l'accès aux membres des structures se fait de la sorte:

// **GPIO\_InitStruct → GPIO\_PIN** ou **GPIO\_InitStruct → GPIO\_Mode**.

//ne pas oublier d'ajouter (décommenter) le prototype de la fonction **GPIO\_Init** au fichier **gpio.h**.

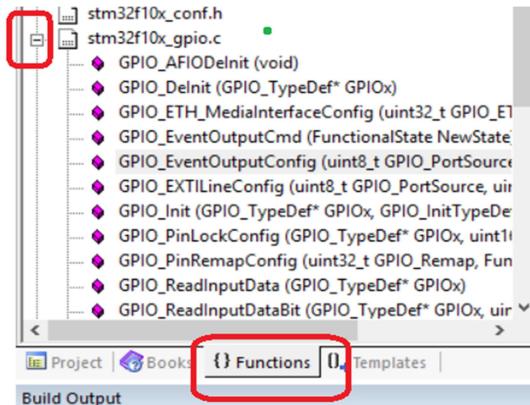
## 2- Programmation par utilisation de la librairie

L'objectif est de comprendre la programmation des interfaces de périphériques en utilisant les drivers fournis avec la librairie ST (ST FWL : FirmWare Library).

### MANIP 2-1 :

- On veut développer un programme qui permet d'allumer les Leds quand on appuie sur le bouton poussoir et de les éteindre quand on le relâche. La lecture de l'état du bouton étant assurée par le mécanisme de scrutation.
- Trouver les fonctions de la librairie qui permettent de mettre un pin à **1**, à **0**, ainsi que la fonction qui permettent de lire l'état d'un pin. (voir figure ci-dessous).

Dans la fenêtre « *project* », cliquer sur l'onglet « *Functions* », et ensuite sur le fichier relatif au périphérique : *stm32fxx\_ppp.c*, *ppp* étant le nom du périph  
(exemple : *stm32Fxxx\_gpio.c*)



- Compléter *le code source (main.c)*, compiler le programme et le tester sur la carte.

**MANIP 2-2 :** Ajouter une fonction **GPIO\_ToggleBits** à la librairie de ST (ST Firmware Library) et écrire un programme (en utilisant cette fonction) de telle sorte à ce qu'à chaque appui sur le bouton poussoir les leds changent d'état.

**Rq :** L'appel à la fonction devrait s'effectuer de la sorte `GPIO_ToggleBits (GPIOx, GPIO_Pin)`  
`GPIO_Pin` étant est un Pin ou une combinaison de pins (`GPIO_Pin_1 | GPIO_Pin_4 | .....`)

### **3- Mécanisme d'interruptions : EXTI, SYSTICK**

Il s'agit de comprendre le principe de programmation des périphériques de telle sorte à exploiter le mécanisme d'interruptions : on traite principalement les interruptions relatives aux entrées externes des GPIO (EXTI : EXTERNAL Interrupts) ainsi que celle relative au Timer système (SysTick).

#### **MANIP 3-1 (EXTI) :**

Il s'agit de reprendre le même fonctionnement de la **manip2-2** mais en utilisant le **mécanisme d'interruptions** au lieu du mécanisme de scrutation qui monopolise les ressources du microprocesseur : A chaque appui, on allume le(s) Led(s) pendant une certaine durée (0xFFFFFFF) et on les éteint.

- Compléter le code au niveau de **main.c** (configuration des EXTI)
- Compéter le code au niveau du handler dans **stm32fxxx\_it.c**.
- Tester le programme sur la carte.

#### **MANIP 3-2 (SysTick) :**

L'objectif est de faire clignoter le(s) led(s) toutes les secondes (1 seconde allumées, 1 seconde éteintes). La temporisation étant assurée par les interruptions issues du Timer Système (SysTick) avec une fréquence de 100 Hz soit une toutes les 10 msec.

- Compléter le code au niveau de **main.c** (configuration du Timer système)
- Compéter le code au niveau du handler dans **stm32fxxx\_it.c**.
- Tester le programme sur la carte.

#### **MANIP 3-3 (EXTI + SysTick) :**

Ecrire un programme qui, à l'initialisation, fait clignoter les leds toutes les 2 secondes.

Ensuite à chaque appui sur le bouton poussoir, la période de clignotement est divisé par 2 (0.5 sec, ensuite 0.25 sec, 0.125 sec , etc .....).

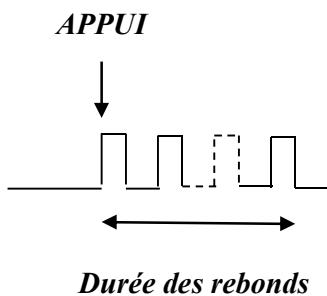
**Rq :** aussi bien la temporisation que la détection de l'état du bouton doit être gérée par utilisation du mécanisme d'interruptions.

La manip suivante est à préparer sous forme d'un **compte-rendu**

#### **MANIP 3-4 (Rebond) :**

Comme vous avez dû le remarquer pendant la *Manip3-3*, à chaque appui la période n'est pas divisée exactement par 2, mais par 4 ou 8 ou ... .

En fait, ceci est causé par le phénomène de rebond : Un appui sur le bouton poussoir se traduit par une suite de contacts entre les deux parties du boutons (+V et GND) et on a ainsi le signal suivant :



- 1- Confirmer cette hypothèse par un programme.
- 2- Ecrire un programme qui permet, pour chaque appui sur le bouton poussoir, de calculer le nombre de rebonds et de mesurer la durée totale des rebonds.
- 3- Donner un tableau qui donne les différentes valeurs trouvées pour 10 appuis.
- 4- Proposer une solution qui permet d'éviter les conséquences du phénomène de rebonds.

## **Manip OOP :**

### Développement d'une couche d'abstraction pour la programmation orientée objet des SOCs : Application aux GPIOs du STM32

Le projet **manipOOP\_1\_OOLayer** contient un exemple de programme orienté objet (c++) qui contient une classe **PinAsInput** permettant de configurer des pins du GPIO en **entrée** ainsi que la possibilité de lire leurs états (à partir du registre IDR).

Ainsi la configuration d'un pin (exemple **GPIOC pin 13** correspondant au bouton poussoir de la carte Nucléo) sera effectuée ainsi (en supposant une config par défaut Input Floating)

**PinAsInput Bouton (PC\_13);**

Et la lecture de l'état du pin pourra se faire en attribuant l'objet à une variable :

Uint16\_t **etat\_bouton** = **Bouton**;

Et on peut également effectuer des comparaisons entre l'objet et des valeurs :

**if (Bouton == ...)** ou bien **if (Bouton != ...)**

Ainsi, pour une application permettant de tester l'état du bouton poussoir de la carte Nucleo (PC13) et d'inverser d'une façon continue l'état de la LED (PA5) si le bouton poussoir est appuyé, sinon le garder inchangé (états de la Led), le code sera le suivant (*en utilisant l'approche objet uniquement pour l'entrée (bouton PC13). Pour les sorties (Led), on gardera l'approche accès registres*):

```

PinAsInput MyButton (PC_13);

int main ()
{
    //Enable GPIOA Clock (Led: PA5)
    RCC->APB2ENR |=0x04;

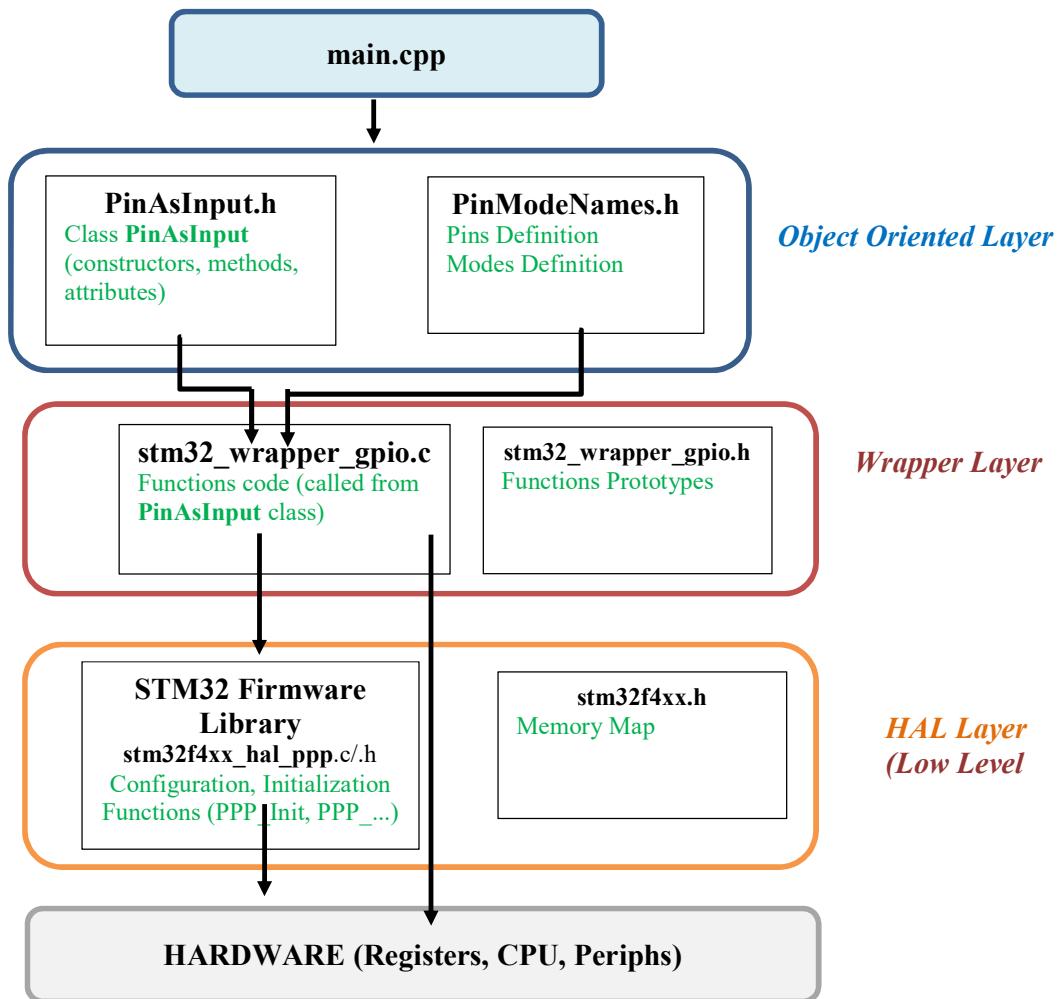
    //Configure PA5 as Output PP
    GPIOA -> CRL &= ~ (uint32_t) (0x0F<<20);
    GPIOA -> CRL |= (uint32_t) (0x02<<20);
    // Pour les autres registres, on gardera les valeurs par défaut 0

    //Led On
    GPIOA -> BSRR = 0x20;
    while (1) {

        if (MyButton == 0) { //button pressed
            GPIOA->ODR ^= 0x20; //Invert PA5 (Led) (^: XOR)
            for (i=0xFFFF; i>0; i--);
        }
    }
}

```

Dans cet exemple, la configuration des registres du GPIO en entrée (GPIOC), ainsi que l'activation des signaux d'horloge du RCC se fait d'une façon transparente au niveau des méthodes appelées par la classe ***PinAsInput***. (Les détails de l'implémentation de la classe traités en cours).



### Travail demandé :

Le projet ***manipOOP\_2\_OOLayer*** contient le même code que **manipOOP\_1\_OOLayer**, mais en plus les fichiers et fonctions (à compléter) qui permettent d'implémenter une classe ***PinAsOutput*** qui contiendra tout le traitement nécessaire pour la configuration et la manipulation des pins configurés en sortie.

En utilisant cette classe, le programme précédent deviendrait :

```

PinAsInput MyButton (PC_13);

PinAsOutput GreenLed (PA_5);

int main ()
{
    // PC9 ON & PC8 OFF
    GreenLed = 1;

    while (1) {

        if (MyButton == 0) { //button pressed
            //Toggle PA5
            GreenLed = !Greenled;
            for (i=0xFFFF; i>0; i--);
        }
    }
}

```

Tous les détails relatifs à la configuration, initialisation, lecture et écriture des pins seront ainsi encapsulés au niveau de la classe à développer.

### Partie 1 : Paramètres de configuration

On définit les paramètres relatifs aux modes des pins des GPIO (configurés en sortie) et aux vitesses comme données de types énumérés au niveau du fichier **Pin\_Mode\_Names.h** :

Mode	Vitesse
<pre> typedef enum{     PPull = 0,     ODraIn = 1,     AF_PPull = 2,     AF_ODraIn = 3,     OutDefault = PPull } PinOutMode; </pre>	<pre> typedef enum{     LowSpeed=0,     MediumSpeed=1,     HighSpeed=2,     VeryHighSpeed=3,     SpeedDefault=LowSpeed } PinSpeed; </pre>

Sachant que les paramètres mode et vitesse sont définis autrement au niveau du Firmware de ST (stm32f10x\_gpio.h) et se trouvent dans les variables suivantes.

	Mode	Vitesse
	GPIO_Mode_Out_OD	GPIO_Speed_10MHz
	GPIO_Mode_Out_PP	GPIO_Speed_2MHz
	GPIO_Mode_AF_OD	GPIO_Speed_50MHz
	GPIO_Mode_AF_PP	

### Question 1-1:

Donner le code des fonctions suivantes (devant se trouver au niveau du fichier `stm32_wrapper_gpio.c`) et dont le rôle est d'adapter les nouvelles représentations (valeurs) des modes et vitesses passées en paramètre à la fonction (Object Oriented layer) aux représentations (valeurs) utilisées au niveau du Firmware de ST et qui seront retournées par les fonctions :

```
int      get_gpio_out_mode (PinOutMode mode)
int      get_gpio_out_speed (PinSpeed speed)
```

**Remarque:** Le fichier `stm32f10x_gpio.h` est déjà inclus dans le projet.

Pour les vitesses on aura (**Low** ->2Mhz, **Medium** ->10Mhz et **High** -> 50Mhz). **VeryHigh** n'est pas utilisé.

### Partie 2 : Constructeurs de la classe

La déclaration d'un objet de la classe `PinAsOutput` peut se faire comme suit :

Déclaration	paramètres
<code>PinAsOutput Nom_Objet (Pin)</code>	Le <i>mode par défaut</i> (Out_PP) et la vitesse par défaut ( <b>Low Speed</b> ) seront utilisés pour la config du pin.
<code>PinAsOutput Nom_Objet (Pin, Mode)</code>	le mode passé en paramètre et la vitesse par défaut (Low Speed) seront utilisée.
<code>PinAsOutput Nom_Objet (Pin, Mode, Vitesse)</code>	le mode et la vitesse passés en paramètres seront utilisés.

Ainsi, la partie du code de la classe contenant les constructeurs sera :

```
namespace OOLayer {

    class PinAsOutput {
        PinName thepin;
        public:
            /** Constructors : Create a PinAsOutput connected to the specified pin */

            PinAsOutput (PinName pin) : thepin (pin) {
                gpio_init_out_def (pin);
            }

            PinAsOutput (PinName pin, PinMode mode) : thepin (pin) {
                gpio_init_out (pin, mode);
            }

            xxPinAsOutput (PinName pin, PinMode mode, PinSpeed speed) : thepin (pin) {
                gpio_init_out_ex (pin, mode, speed);
            }

            ....} // end class
    } // end SpaceName
```

- Ces Fonctions appelées par le constructeur devraient faire appel à la fonction `_gpio_init_out` qui permettra d'activer l'horloge du GPIO et de configurer le pin en sortie avec les paramètres spécifiés.

Ces fonctions ainsi que les prototypes sont déclarés au niveau de fichiers `stm32_wrapper_gpio.c.h`

**Question 2-1:** Compléter au niveau du fichier (`stm32_wrapper_gpio.c`) le code de la fonction:

```
static void _gpio_init_out (PinName pin, PinOutMode mode, PinSpeed speed)
```

### Partie 3 : Méthode de la classe

On veut ajouter deux méthodes à la classe

- La première fonction `read()` permettant de lire l'état du pin configuré en sortie (donc au niveau du registre ODR).

```
int read() {
    return gpio_read_out (thepin );
}
```

- La deuxième fonction `write (value)` permettant de d'écrire la valeur passé en paramètre au niveau du pin (registre BRR si value=0, sinon BSRR).

```
void write (int value) {
    gpio_write (thepin , value);
}
```

**Question 3-1:** Donner le code (au niveau de `stm32_wrapper_gpio.c`) des fonctions appelées par les méthodes de la classe :

```
void gpio_read_out (PinName pin)
int gpio_write (PinName pin, int value)
```

## Partie 4 : Surcharge de l'opérateur (=)

Normalement pour imposer une valeur à un pin, il faut faire appel à la méthode **write**. Ainsi, si on a un pin (exemple **PA5**) déclaré en sortie

```
PinAsOutput BlueLed (PA_5);
```

Alors pour imposer une valeur (1 ou 0) à ce pin, il faut utiliser la syntaxe :

```
BlueLed.write (valeur);
```

On veut faciliter l'écriture de telle sorte à pouvoir la remplacer par une syntaxe encore plus simple et plus lisible :

```
BlueLed = valeur;
```

Ceci passe par l'ajout d'une surcharge de l'opérateur **=**.

**Question 4-1:** Au niveau de la classe **PinAsOutput** (au niveau du fichier *PinAsOutput.h*) , ajouter le code assurant la surcharge de l'opérateur **=** quand on affecte à un objet une valeur :

```
void operator = (int value) {
    // .....
}
```

- On veut étendre la surcharge de telle sorte à pouvoir affecter à un objet (pin en sortie), la valeur d'un autre objet (autre pin configuré en sortie) de la façon suivante :

```
BlueLed = Greenled;
```

Ou par exemple

```
BlueLed = ! BlueLed;
```

**Question 4-2:** Au niveau de la classe **PinAsOutput** (au niveau du fichier *PinAsOutput.h*) , ajouter le code assurant la surcharge de l'opérateur **=** quand on affecte à un objet la valeur d'un autre objet.

## Manip4 : Accès Direct aux Mémoires (DMA)

**Objectif:** Il s'agit de comparer les performances des transferts de données entre deux zones mémoires : La Flash et la RAM dans le STM32F1 par utilisation du :

- Processeur GPP (instructions de lecture / écriture classiques).
- Direct Memory Access Controller.

Ce transfert va s'effectuer entre :

**Source:** `const uint32_t SRC_Const_Buffer[64]`: une chaîne de 64 données constantes de 32 bits logée dans la Flash (comme toutes les constantes)

**Destination:** `uint32_t DST_Buffer[64]`: une chaîne de 64 données de 32 bits logée dans la RAM

Les performances du transfert seront évaluées en termes de Nombre de Transferts de la chaîne de 64 données pendant **une seconde** (qui sera converti ensuite en **KOctets/sec**).

Dans les deux cas, au lancement du programme, le transfert doit commencer et à chaque transfert de la chaîne de la source vers la destination, on incrémentera une variable contenant le nombre de transferts de la chaîne.

Après exactement **1 seconde**, on arrête le transfert et on récupère la valeur représentant le nombre de transferts.

**Manip 4-1 (Transfert CPU):** Estimer les performances du transfert par utilisation du GPP (Processeur) exprimées par la variable `Nber_Transfer_CPU`.

**Manip 4-2 (Transfert DMA):** Estimer les performances du transfert par utilisation du DMAC exprimées par la variable `Nber_Transfer_DMA`.

- Comparer les résultats et expliquer.

**Manip 4-3 (Transfert DMA/CPU) :** Modifier le programme de la manip4-2 de telle sorte à lancer les deux transferts en même temps (Processeur et DMAC) et estimer les performances.

- Comparer les performances de transferts du Processeur (resp. DMAC) avec celles trouvées précédemment en partie 1 (resp partie2) et expliquer les résultats trouvés.

## Manip5 : Communications avec USART :

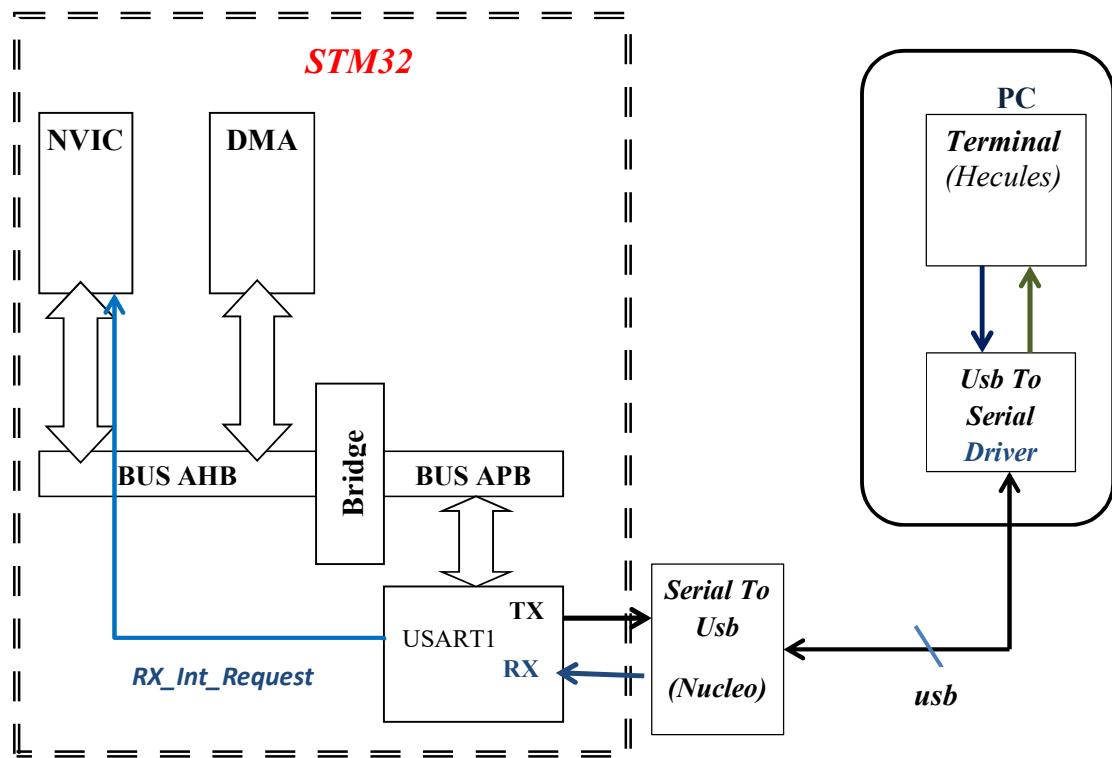
### Réception avec Interruption / Envoi Transfert DMA

Il s'agit de réaliser une communication série (UART) entre un microcontrôleur STM32 et un PC en utilisant le mécanisme d'interruption pour la réception (1<sup>ère</sup> partie) et le transfert DMA pour la transmission (2<sup>ème</sup> partie)

#### Manip 5 (1<sup>ère</sup> partie) :

L'interface série **USART1** (9600 bps, Pas de Parité, 2 bits de stop) est configurée pour déclencher en plus des *interruptions* à chaque *réception* d'un caractère (envoyé à partir du Terminal).

Ces caractères étant sauvegardés dans une variable **char Receive\_Buffer[10]**.



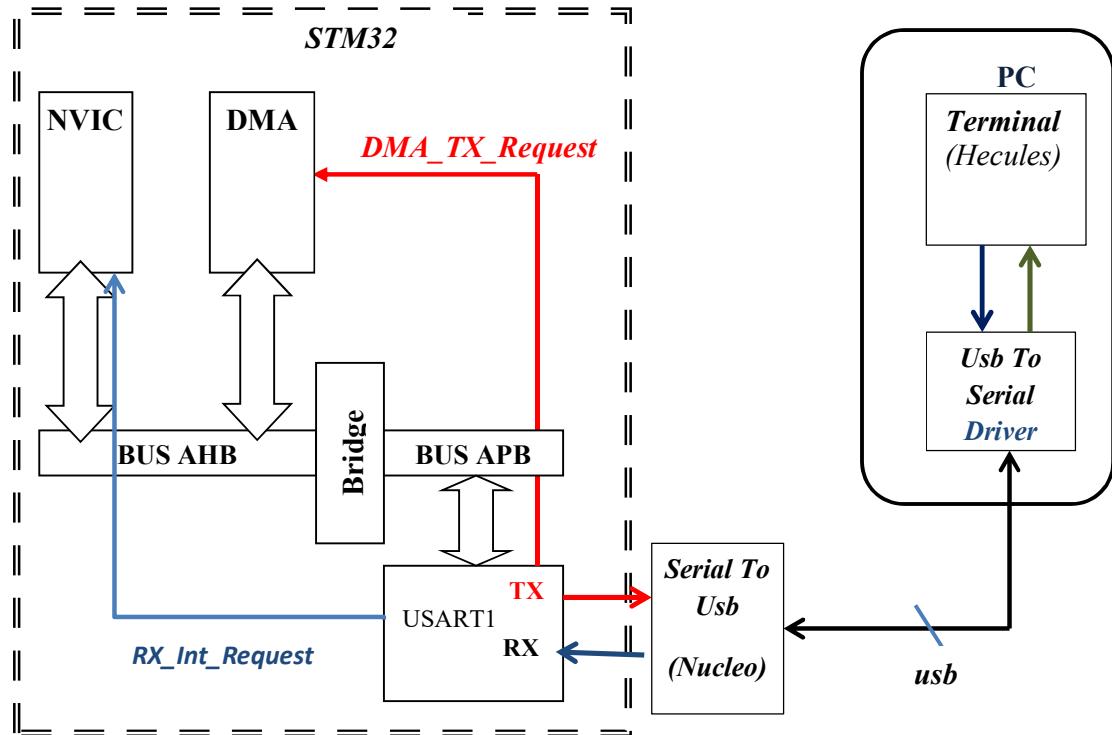
Une fois 10 caractères reçus, il faut comparer la chaîne de 10 caractères à une chaîne de caractères **password[] = "9876543210"**.

Si les 2 chaînes sont les mêmes, alors allumer la LED et attendre (delay (0xFFFF)), sinon la faire clignoter lentement.

### ***Manip 5 (2<sup>ème</sup> partie) :***

Dans cette partie, il faut ajouter le traitement suivant au cas où les deux chaines sont identiques :

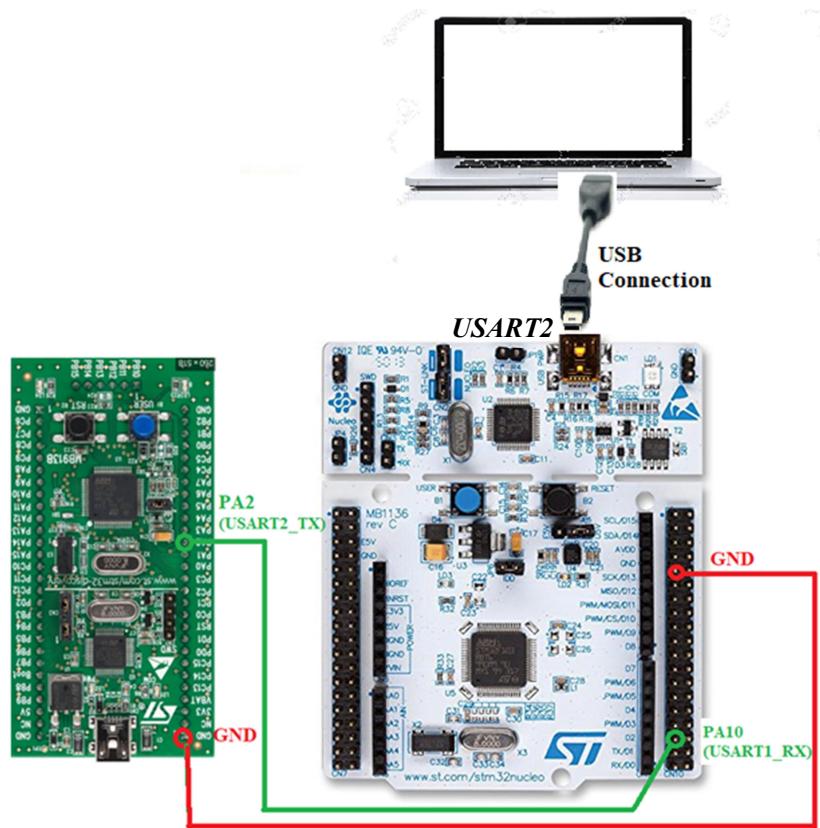
- envoyer la chaîne de caractères stockée dans la variable ***char Transmit\_Buffer*** en utilisant le transfert DMA (avec les interruptions « ***Transfer Complete*** » et « ***Transfer Error*** » du **canal DMA USART1\_TX** activée).
- 1) Préciser le canal DMA à utiliser.
  - 2) Compléter la fonction ***DMA\_Config()*** qui permet de configurer le transfert DMA (de la mémoire (string « ***Transmit\_Buffer*** » vers l'USART1).
  - 3) Compléter le code du Handler relatif aux interruptions DMA de telle sorte à éteindre la LED si la transmission s'est déroulée sans problèmes, sinon (en cas d'erreur) la faire clignoter rapidement.
  - 4) Compléter le code au niveau du ***main.c*** (déclencher le transfert DMA)



## Manip 6 : Communications Série Asynchrone (UART):

### Réception avec Transfert DMA / Envoi avec Interruptions

Il s'agit de réaliser une communication série Asynchrone :



- La carte Discovery (STM32 – USART2) envoie un caractère à chaque appui sur le bouton poussoir (PA0).
- La carte Nucleo (microcontrôleur **STM32 – USART1**) reçoit les données à partir de la carte Discovery (STM32) : Une liaison directe entre le pin PA2 (**USART2 - TX**) de la carte Discovery et le pin PA10 (**USART1 - RX**) de la carte Nucleo.
- La carte Nucleo (**STM32 – USART2**) envoie ensuite les données reçues à un PC à travers une connexion USB. En effet, La carte Nucleo est équipée d'un adaptateur Série/Usb qui permet ceci.)

### ***1) Test de l'application de la carte Discovery***

Le programme permettant d'envoyer les caractères à partir de la carte Discovery est déjà développé. Il s'agit de le tester avant de passer à l'étape suivante.

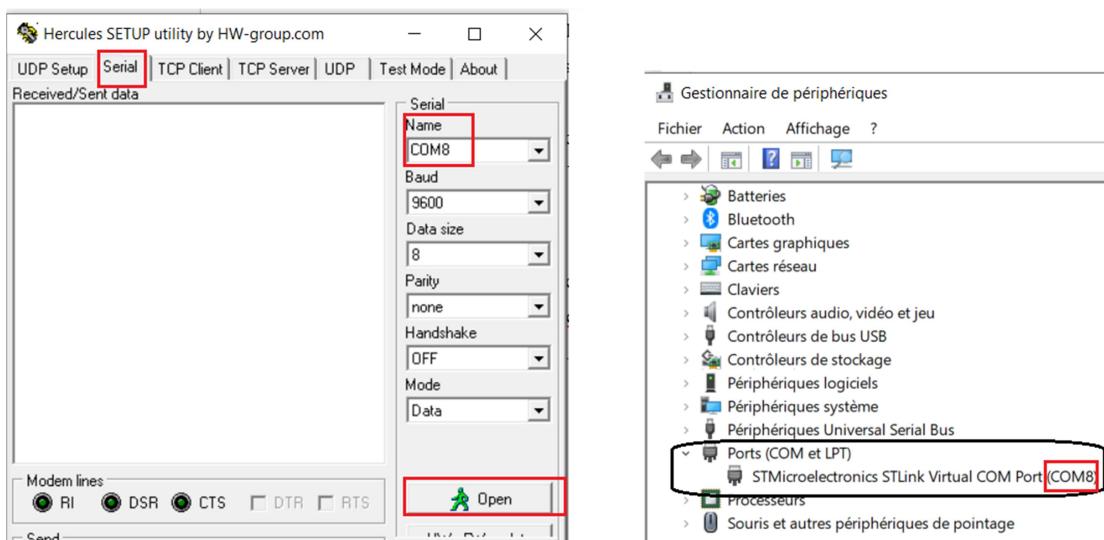
**A-** Ouvrir le projet « *Manip6\_Send\_Serial* » le compiler et programmer la carte Discovery F100RB : Cette application permet d'envoyer un caractère à chaque appui sur le bouton poussoir PA0.

**B-** Ouvrir le projet « *Manip6MBED\_Serial\_Gateway* », le compiler et programmer la carte Nucleo F103RB : Cette application développée avec l'environnement MBED (orienté objet, C++) permet d'écouter continuellement l'interface série USART1 relié à la carte Discovery, et à chaque fois qu'un caractère est reçu, il est envoyé au PC à travers l'interface série USART2 et l'état de la Led (PA5) est inversé.

(**RQ** : ne pas relier les 2 cartes en même temps au PC. Les programmer séparément).

**C-** Connecter les interfaces série des 2 cartes (Voir Figure ci-dessus), connecter les cartes à travers les interfaces USB au PC et lancer l'application Terminal ('Hercules') au niveau du PC.

**D-** Au niveau de l'application 'Hercules', ouvrir le port de communication série (COM x) attribué à la carte Nucleo. Le numéro du COM étant récupéré à partir du gestionnaire de périphériques de Windows (Voir Figure ci-dessous).

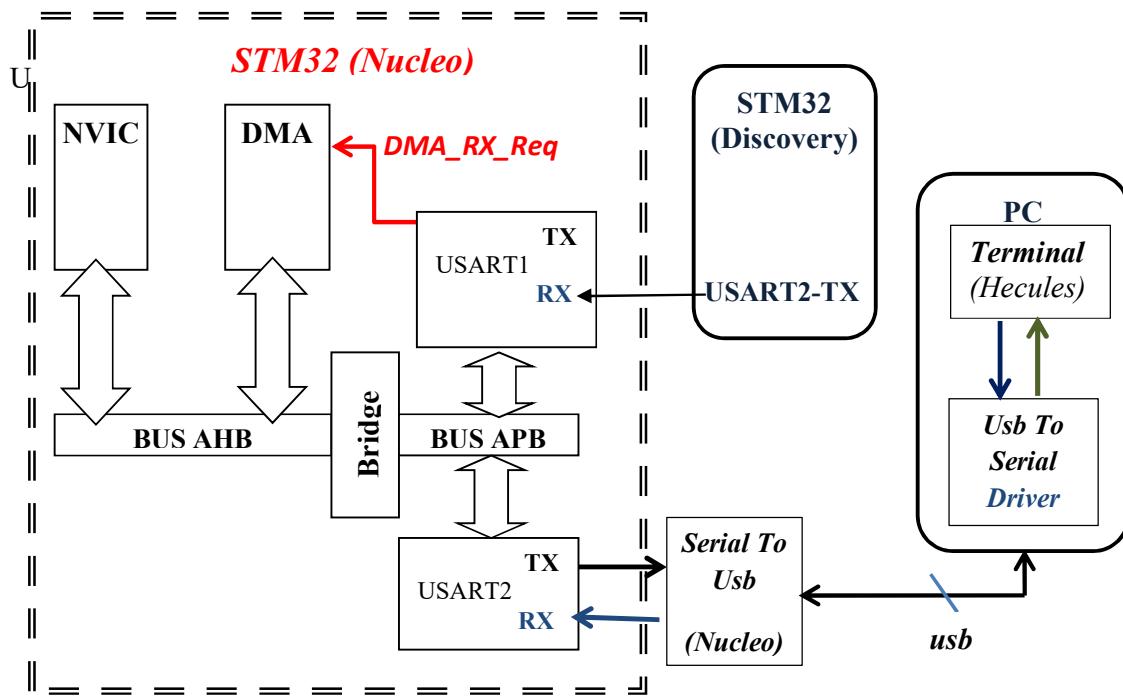


**E- Tester le fonctionnement du système :** A chaque appui sur le bouton poussoir de la carte Discovery, l'état de la Led devrait changer et un caractère doit être affiché au niveau de l'interface du terminal ‘Hercules’.

### Manip 6 (1<sup>ère</sup> partie) :

**A-** L'interface série **USART1** de la carte Nucleo doit être configuré avec les paramètres (9600 bps, Pas de Parité, 1 bit de stop) et pour déclencher des **Transferts DMA** sur le *canal* approprié afin de sauvegarder les caractères reçus dans la variable **char Receive\_Buffer[20]**.

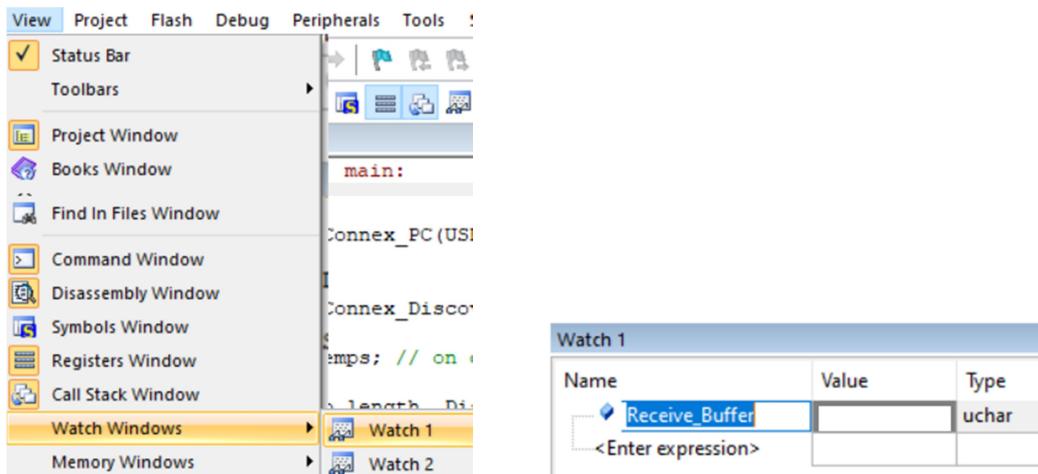
Compléter le code de la fonction **void Config\_USART1\_RX\_WITH\_DMA (void)** au niveau du fichier main.c afin d'assurer le fonctionnement décrit ci-dessus.



**B-** Une fois 20 caractères sont reçus, et le transfert est terminé, il faut comparer la chaîne reçue à la chaîne **char Compare\_Buffer [20]**. Si les deux sont les mêmes, alors allumer la Led PA5, sinon la faire clignoter 3 fois.

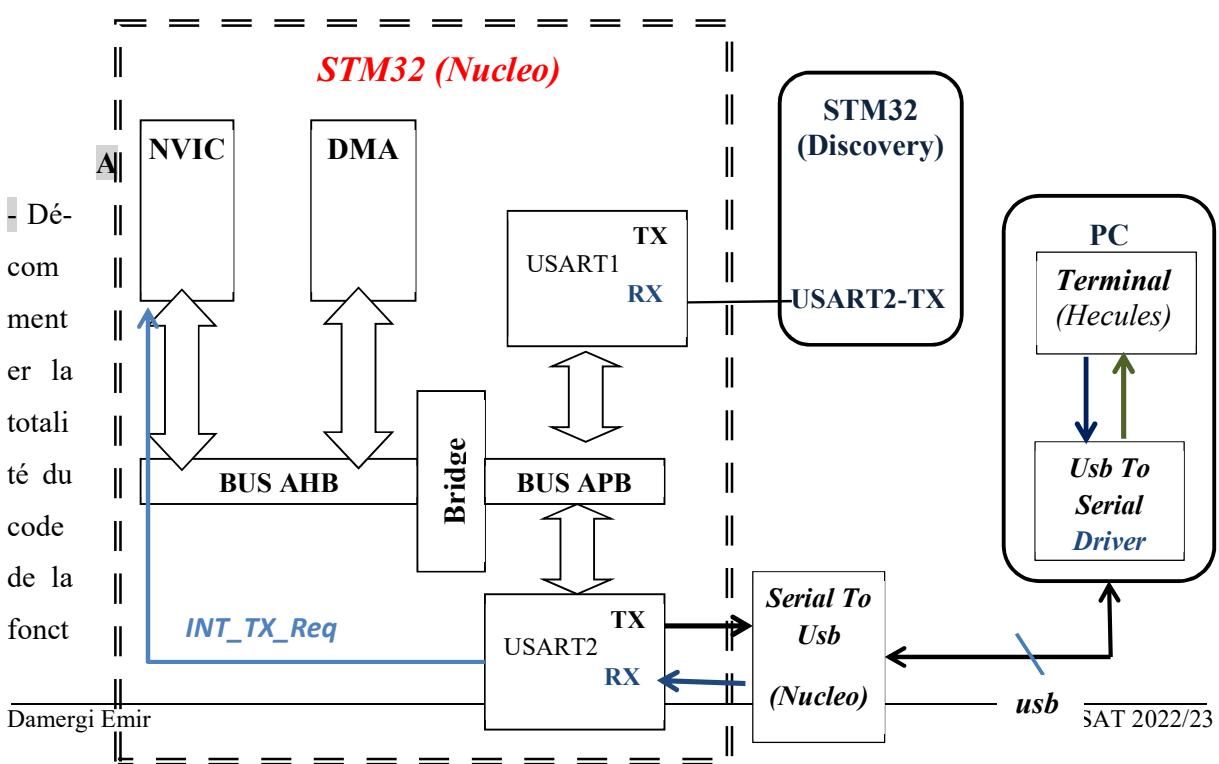
Compléter le code de la fonction relative aux interruptions du Transfert DMA au niveau du fichier **stm32f10x\_it.c**.

Pour tester si les caractères sont bien reçus, il est possible d'ajouter des Break Points et de suivre le contenu de la chaîne **Receive\_Buffer** en mode débogage en l'ajoutant au Watch Window (voir Figure suivante).

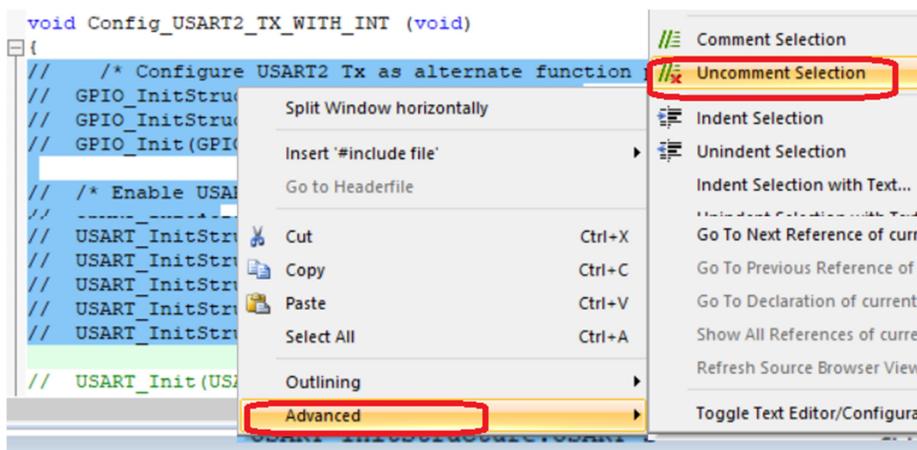


### Manip 6 (2<sup>ème</sup> partie) :

Dans cette partie, il faut ajouter le traitement suivant au cas où les deux chaînes sont identiques : envoyer la chaîne de caractères stockée dans la variable **char Transmit\_Buffer** vers le PC à travers l'interface USART2 et en utilisant le mécanisme d'interruptions.



ion **Config\_USART2\_TX\_WITH\_INT** au niveau du fichier main.c (*Voir Figure ci-dessous*)



Compléter le code afin de configurer l’interface série **USART2** de la carte Nucleo avec les paramètres (9600 bps, Pas de Parité, 1 bit de stop) ainsi que l’interruption relative à l’USART2 au niveau du NVIC.

**B-** Sachant qu’à la fin de la réception de la chaîne de 20 caractères (*receive\_Buffer*), et si elle est la même que celle stockée dans *Compare\_Buffer*, il faut envoyer la chaîne *Transmit\_Buffer* à travers l’interface USART2 et en utilisant le mécanisme d’interruptions : A quel niveau, il faut configurer et activer **USART2** afin qu’il envoie des requêtes d’interruptions pour la transmission.

Compléter le code assurant cette tâche.

**C-** Finalement, c’est au niveau de la fonction **USART2\_IRQHandler** que les instructions permettant d’envoyer la chaîne *Transmit\_Buffer* doivent être placées.

Dé-commenter la totalité du code de la fonction **USART2\_IRQHandler** au niveau du fichier *stm32f10x\_it.c* et compléter le code

Tester l’application en utilisant le terminal ‘**Hercules**’ au niveau du PC.