

# Step Débute 1

## LED clignote à 2Hz, temporisation par boucle

### Documents utiles :

Cortex-M3 Programming Manual (PM0056.pdf)

### How to ?

Compiler/assembler/Edition de liens  
Chronométrer une durée d'exécution  
Observer une variable dans la watch windows  
Observer une variable dans la mémoire  
Observer variable / pin dans le logic analyser  
Observer un signal sur cible réelle

## 1. Prise en main KEIL

Observer variable ou pin dans le Logic analyser

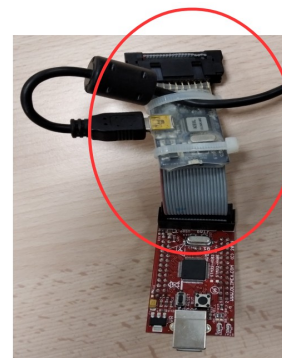
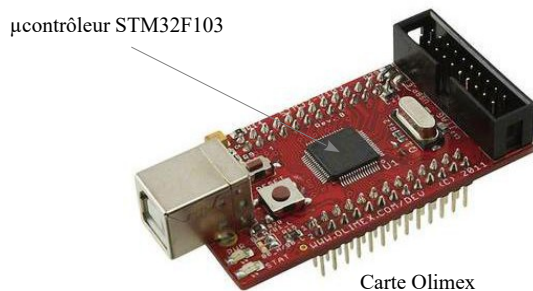
### Le point périph ...

Le GPIO (port d'entrée / sortie )

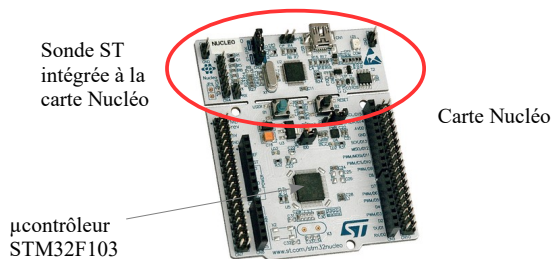
1. Déployer l'archive *PjtKEIL\_StepDeb\_1.zip*
2. Ouvrir le projet KEIL (.uvprojx), sélectionner la cible sur laquelle le code va tourner. Pour cela, explorer le menu déroulant *Select Target*. Trois cibles ont été pré-configurées :

- **Simu** : le code est simulé par KEIL (donc par le PC). On peut donc tester le code SANS le processeur physique.
- **CibleSondeKEIL** : le code exécutable (.elf) est chargé dans une carte contenant un *STM32F103* via une sonde de **débuggage de type ULINK2/ME**. Ce sont les cartes *Olimex* correspondant au projet LASER :

Sonde ULINK2/ME KEIL connectée la carte Olimex



- **CibleSondeST** : le code exécutable (.elf) est chargé dans une carte contenant un *STM32F103* via une sonde de **débuggage de type ST**. Les sondes peuvent être intégrées à la carte en question (carte *Nucléo*) ou externalisées. Dès lors on peut aussi les utiliser pour venir commander une carte *Olimex*.



Sonde ST externalisée connectable à une carte Olimex via le connecteur HE10 noir



Il appartiendra à chacun de s'interroger sur le matériel à disposition pour faire un choix pertinent de la sonde à utiliser lorsque l'on exécutera le code sur le micro-contrôleur physiquement. Les deux types de sondes *ST* et *Ulink2/ME* seront utilisées en TP.

## 2. Analyse du code

### 2.1. Principal.c

Il contient la fonction `main()` qui est composée de deux parties :

- **l'initialisation des périphériques** qui s'exécute **une seule fois**. Son rôle est de configurer des parties matérielles du micro-contrôleur. Les fonctions appelées sont contenues dans *DriverJeuLaser.lib*, et sont accessibles via l'API *DriverJeuLaser.h* pour une utilisation en langage C, ou via l'API *DriverJeuLaser.inc* pour une utilisation en assembleur.

La première fonction, `CLOCK_Configure()` est à lancer OBLIGATOIREMENT en tout premier lieu. C'est elle qui organise tout le système d'horloge du *STM32F103*. En particulier, elle cadence le CPU interne à 72MHz à partir d'un quartz externe de 8MHz (via une PLL).

La seconde fonction `GPIO_Configure(GPIOB, 1, OUTPUT, OUTPUT_PPULL)` a pour rôle de configurer la pin 1 du port B en sortie. Une fois la fonction réalisée, l'utilisateur peut positionner la pin à '1' (3,3V) ou '0' (0V). Pour plus d'information voir [Point périphérique de micro-contrôleur](#)

- **La boucle principale.**

#### Remarque TRES IMPORTANTE

Il est possible d'utiliser un OS dans le domaine de l'embarqué (par exemple *FreeRTOS* ou même *Linux*), mais ici nous programmerons en **"Bare-metal"** c'est à dire SANS OS (pratique très courante en embarqué). Le *main* ne rend donc JAMAIS la main à un OS. La fonction *main* ne doit donc JAMAIS se finir par un `return`. **Elle doit toujours comporter une boucle sans fin**. Dans le cas contraire, le logiciel plante gravement en allant chercher des instructions dans une zone mémoire vierge...

## 2.2. Delay.s

1. Recherchez toutes les directives dans ce programme. Vous devez être capable de décrire précisément à quoi elles servent. Dans le cas contraire, discutez en avec votre enseignant ou votre entourage. Vous devez tout comprendre (sauf peut être *Thumb* et *Preserve8* qui sont un peu particulières).

**NB** : une directive d'assemblage = tout ce qui n'est pas une instruction exécutée par le CPU...

2. Analysez maintenant le code, apportez des commentaires pertinents.

Exemple de ce qu'il ne faut pas faire :

```
ldr r1,[r0] ; mets dans r1 le contenu mémoire d'adresse r0 (format 32 bits).
```

Ce commentaire est correct, mais n'amène rien de plus que ce que dit l'instruction *asm*. Un lecteur familier avec l'*asm ARM* sait déjà tout ça.

Le commentaire, pour qu'il soit pertinent doit être contextualisé au programme. Ici on préférera largement :

```
ldr r1,[r0] ; r1 = VarTime
```

3. Expliquer clairement le rôle de l'instruction *bx lr* en fin de routine *asm*.

## 3. Travail sur cible simulation

1. Choisir la cible simulation (cf §1.)
2. Compiler et procéder à l'édition de lien. Il existe 3 boutons (situés à gauche de la barre d'outils) :

### **How to ?** Assembler / compiler / faire l'édition de lien


- *Translate* : crée le *.obj* à partir du fichier source actif (*.asm* ou *.c*). Si le source est un *.asm*, l'action *translate* correspond à l'**assemblage** du fichier (*.asm* → *.obj*). Si le source est un *.c*, l'action *translate* correspond à la **compilation** du fichier (*.c* → *.obj*).

- *Build* : réalise l'**édition de lien** à partir des *.obj* déjà présents dans le répertoire */OBJ* du répertoire de travail (là où est le projet *KEIL .uvproj*). S'il manque des objets, KEIL va les obtenir en compilant et assemblant toutes les sources dont l'objet est absent.

- *Rebuild* : même si les *.obj* sont tous présents, KEIL les efface et les reconstruit tous à partir des sources du projet.

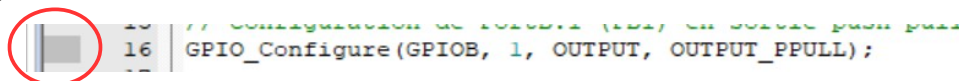
**NB** : selon la taille du projet et les modifications que vous apportez ici ou là, il convient de bien choisir laquelle des ses 3 options est la plus pertinente.


Logiquement cette opération se fait sans erreur sur ce projet.

3. Basculer en mode **débuggage** (cible simulation ici), cliquer sur le bouton 
4. Placer judicieusement des points d'arrêt pour mesurer la période exacte du signal carré généré sur la pin 1 du GPIOB, qu'on note souvent PB1.


### **How to ?** Chronométrer un temps d'exécution

Les endroits possibles pour placer un point d'arrêt sont repérés par un carré gris foncé dans la marge à gauche :



Lancer code (run) : 

L'exécution s'arrête au point d'arrêt.

→ Clic droit sur le champ *t1* en bas à droite de la fenêtre KEIL : 

→ *Reset Stop Watch (t1)* → le temps est mis à 0.

→ Lancer à nouveau le programme jusqu'au prochain point d'arrêt → il suffit de lire le temps écoulé.

**Remarque** : vous pouvez aussi exprimer le temps d'exécution en nombre de cycles machine : voir le champ *states* dans la fenêtre *registers* à gauche.

#### 5. Observer la variable *VarTime* au cours de l'exécution

##### **How to ?** Observer une variable dans une watch Window

→ survoler la variable avec la souris puis clic droit → *Add 'NomVariable' to... watch 1*

→ Il est possible que dans la *watch 1* il soit écrit « *cannot evaluate* ». Cela signifie que la variable n'est pas globale. Il faut donc la rendre globale provisoirement pour le test. C'est trivial en c. Pour une variable en assembleur, il faut l'exporter avec la directive *export : export VarTime*.

→ on peut alors choisir la base de représentation (décimal, hexadécimal...),

→ pour observer une évolution en cours d'exécution, il faut aller dans le menu :

*View* → cocher le champ *Periodic Window Update*

#### 6. Observer la variable *VarTime* dans la mémoire.

##### **How to ?** Observer une variable dans la mémoire

Si la variable est statique, elle se retrouve affectée à une adresse fixe par l'éditeur de lien. Pour la trouver :

→ ouvrir le fichier *.map* qui se trouve dans le répertoire */OBJ*

→ chercher la variable (Ctrl F) puis copier l'adresse d'affectation de la variable

→ Ouvrir la fenêtre mémoire : *View* → *Memory window* → *Memory 1* (par exemple)

→ coller l'adresse dans le champ correspondant de *Memory 1*.

→ notez qu'ici, non seulement on peut fixer la base, mais aussi le format d'affichage (byte/half word/word). Ne pas hésiter à changer par curiosité...

##### **Autre option sans le fichier .map**

→ repérer le moment où le programme assembleur charge le registre contenant l'adresse de la variable (ici *ldr r0, =VarTime*),

→ copier l'adresse dans le registre concerné (ici R0) et coller dans la fenêtre mémoire. (**NB** : vous pouvez aussi taper directement R0 dans le champs d'adresse de la mémoire, mais du coup cette valeur bougera au gré des changements de R0...).

7. Observer le signal électrique sortant de PB1 en simulation

### **How to ? Observer une variable ou une pin dans le logic analyser**

On peut non seulement observer l'évolution d'une pin mais aussi une variable au cours du temps dans le *logic analyser* à condition qu'elle soit **globale** (donc exportée en asm).

→ ouvrir le logic analyser : *View* → *analysis Windows* → *Logic Analyser*

#### **Observer PB1 :**

→ depuis le logic analyser, *Setup...* puis cliquer sur l'icône *new...*



→ saisir *portb.1*

→ *Display type* = 'bit'

→ *Close*

On peut alors lancer le programme, on observe le signal évoluer. On peut utiliser les curseurs pour faire des mesures.

#### **Observer une variable :**

→ procéder de même, mais saisir le nom de la variable souhaitée,

→ *Display type* = 'Analog', préciser alors le min et le max attendus par cette variable.

On pourra essayer avec la variable *VarTime*. On mettra 1 000 000 en décimal pour le max.

## 4. Travail sur cible µcontrôleur STM32F103

Selon le matériel que vous aurez à disposition, vous travaillerez avec l'une ou l'autre des sondes (cf §1). A vous de faire le bon choix de sonde sous KEIL...

8. connecter la carte µcontrôleur au PC via la sonde et son câble USB.
9. Lancer la session de débogage après avoir choisi la bonne cible. Vous êtes maintenant prêt à interagir avec le « vrai » processeur exactement (ou presque) comme en simulation...
10. Reprendre les activités précédentes (celles qui peuvent être réalisées en réel), c'est à dire :
  - Observer la variable *VarTime* des deux manières déjà vues,

**NB :** La mesure de temps et la visualisation avec l'analyseur logique ne sont plus possibles avec les outils vus en simulation.
11. Observer le signal de sortie PB.1 à l'oscilloscope, vérifier la durée à l'état haut et à l'état bas.

### **How to ? Observer l'évolution du port de sortie sur la cible réelle ?**

→ repérer la pin souhaitée (ici PB1). Si vous utilisez la carte *Nucléo*, voir le document *UM1724 User Manual STM32 Nucleo boards* page 27 ou 54.

→ observer le signal à l'oscilloscope en oubliant pas de relier le GND à la référence de l'oscilloscope.

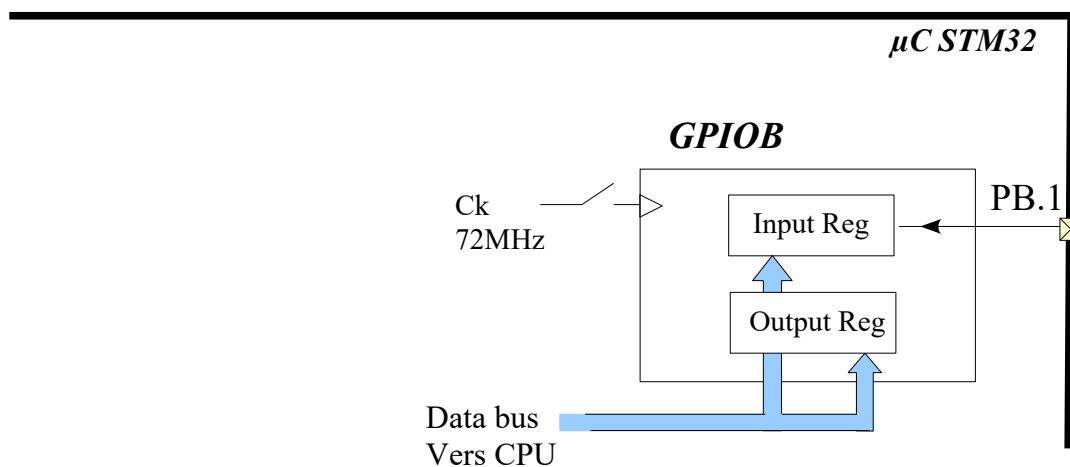
## 5. Point périphérique de micro-contrôleur

### 5.1. Configuration

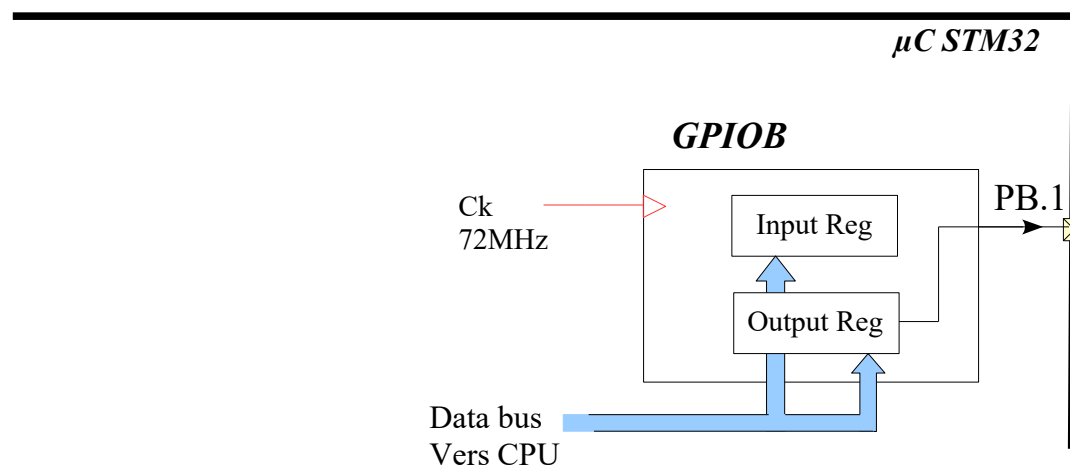
Fonction de configuration : *char* GPIO\_Configure(*GPIO\_TypeDef* \* Port, *int* Broche, *int* Sens, *int* Techno)

Exemple : GPIO\_Configure(GPIOB, 1, OUTPUT, OUTPUT\_PPULL)

**Avant l'exécution (reset  $\mu$ C) :** La pin 1 de GPIOB est en configurée en entrée par mesure de sécurité, Le GPIO n'est pas clocké (son entrée d'horloge est inactive).



**Après l'exécution :** le GPIO est clocké, la pin 1 de GPIOB est en sortie



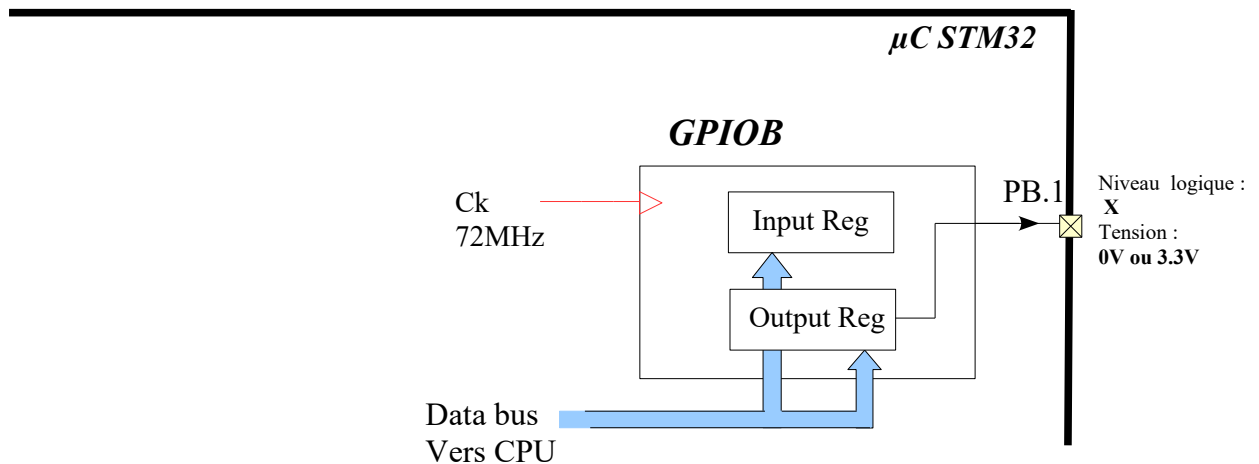
## 5.2. Utilisation

Fonction d'utilisation :

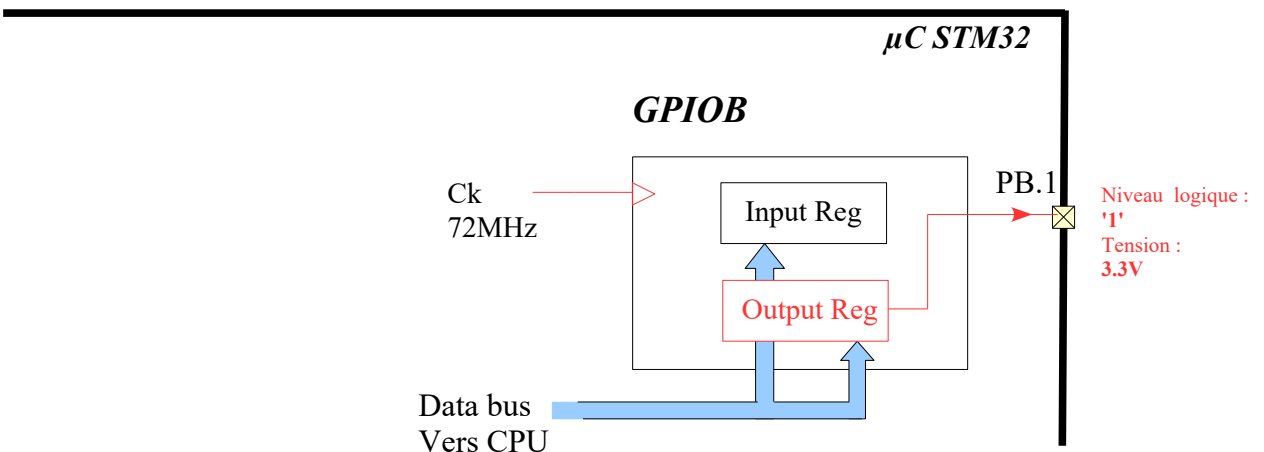
```
void GPIOA_Set(char Broche);      void GPIOA_Clear( char Broche);
void GPIOB_Set(char Broche);      void GPIOB_Clear( char Broche);
void GPIOC_Set( char Broche);      void GPIOC_Clear( char Broche);
```

Exemple : `GPIOB_Set(1);`

**Avant l'exécution :** La pin PB1 a un niveau logique inconnu (X), soit 0V, soit 3.3V.



**Après l'exécution :** La pin PB1 a un niveau logique '1', soit 3.3V.



[Retour](#)