

Structures de code, notions temporelles

Mise en pratique de la séance 1 : On clignote

►Q.81: Faire clignoter (dans une boucle `while` infinie) une LED connectée sur la Pin 2, à la fréquence de 10 Hz. Le temps de chaque état (Haut puis Bas) sera, vous l'aurez facilement calculé, de 50 ms.

►Q.82: Arrêter le programme au bout de 2 s, en mesurant le temps écoulé à l'aide de la méthode `ticks_ms()`. Le test d'arrêt pourra se faire sur la boucle `while`, du genre : « boucler tant que le temps écoulé est inférieur à 2000 ms ».

►Q.83: Ajouter le code qui permet à l'utilisateur, avant de lancer le clignotement, de choisir la fréquence de clignotement à l'aide de la fonction `input`.

►Q.84: Ajouter une boucle While pour que le programme redemande automatiquement et indéfiniment à l'utilisateur, après chaque durée de clignotement de 2 s, la nouvelle fréquence qu'il désire pour le prochain cycle.

8 Découverte de la carte ESP ENIM

Autour du microcontrôleur ESP32, j'ai développé cette carte (voir figure 17 page 42) pour rendre accessibles tous les composants dont nous aurons l'usage au cours de ces travaux pratiques, et aussi pour la plupart des usages que nous pourrons en faire pour contrôler différents systèmes pour des projets ENIM ou personnels.

CARTE d'expérimentation ESP 32 - ESP ENIM

ESP32 - DEV KIT - V1

© Vincent HERMITANT

ESP32 - DEV KIT - V1		EN
Bouton poussoir D	GPIO36 - In only	GPIO23 - VSPI MOSI
Bouton poussoir C	GPIO39 - In only	GPIO22 - I2C SCL
Bouton poussoir B	GPIO34 - In only	GPIO1 - UART 0 TX
Potentiomètre P1	GPIO35 - In only	GPIO3 - UART 0 RX
Capteur Photo-résistance LDR	GPIO32	GPIO21 - I2C SDA
Potentiomètre P2	GPIO33	GPIO19 - VSPI MISO
Bouton poussoir A	GPIO25	GPIO18 - VSPI CLK
Neo Pixel de 8 LED - Pin Data	GPIO26	GPIO5 - VSPI CS0
DS18B20 + R tirage	GPIO27	GPIO17 - UART 2 TX
Connecteurs	GPIO14 - HSPI CLK	GPIO16 - UART 2 RX
Connecteurs	GPIO12 - HSPI MISO	GPIO4
Connecteurs	GPIO13 - HSPI MOSI	GPIO2
Masse	GND	GPIO15
Alimentation 6V	VIN (5V)	GND
		3V3




FIGURE 17 – Implantation des composants externes à l'ESP32 sur la carte ESP ENIM

Sur la plupart des entrées/sorties se trouvent des connecteurs Grove. Se trouvent ainsi :

1. Des entrées :

- 4 boutons poussoirs, BPA, BPB, BPC, BPD, sur les entrées 25, 34, 35, et 36, avec contact additionnel sur borne à vis,
- 2 potentiomètres P1 et P2 sur les entrées analogiques 33 et 35, avec connecteur Grove additionnel connectés via le sélecteur P1 et P2,
- 2 surfaces Touch Pin en cuivre, et contact additionnel sur borne à vis, sur les entrées capacitatives 15 et 4,
- 1 DS18B20, capteur de température, sur l'entrée 27, avec connecteur additionnel à vis,
- 1 LDR, capteur de lumière, sur l'entrée 32, avec connecteur additionnel à vis, connecté via le sélecteur LDR,

2. Des sorties :

- 4 LED, de couleurs différentes, bleu, vert, jaune, rouge, avec connecteur Grove additionnel, sur les sorties 2, 18, 19, 23,
- 1 Neopixel sur la sortie 26 - avec un connecteur additionnel pour brancher un autre NéoPixel,
- 1 afficheur OLED sur les bornes 21 (SDA) et 22 (SCL) - pas de connecteur additionnel,
- 1 haut-parleur sur la sortie 5 - pas de connecteur additionnel,

3. Des entrées ou sorties : 3 connecteurs libres de tout composant, sur les bornes 12, 13, et 14, qui pourront être choisie en entrée ou en sortie.

Cette carte permettra de concevoir, ou commander aisément de nombreux systèmes, car elle comporte assez de poussoir, sorties, LED, potentiomètres pour répondre à la majorité des besoins. Je n'ai pas ajouté de relais de puissance pour ne pas la surcharger ; il sera nécessaire de le rajouter via les connecteurs Grove sur les sorties disponibles.

9 Comptage sur les entrées numériques et capacitives

Nous avons déjà vu précédemment qu'une entrée numérique permet de lire un état logique. La plupart des broches de l'ESP32 peuvent être configurées pour être des entrées ou des sorties, mais certaines broches ne peuvent être programmées que comme étant des entrées ; c'est le cas des broches 34, 35, 36, 39.

Nous nous servirons des entrées pour compter le nombre d'appuis, ou d'évènements détectés.

9.1 Entrée numérique

9.1.1 Lecture simple sans comptage - Rappel

Nous avons vu au chapitre 5.2.5 page 31 le code suivant qui permet de lire l'état du poussoir et d'allumer une LED (connectée sur la sortie 25) s'il est pressé, et l'éteindre sinon. La sortie du poussoir est connectée sur l'entrée numérique, en borne 2.

- Le poussoir provoque, lorsqu'il est pressé, une tension de 3.3V, ce qui sera reconnue par l'entrée comme un niveau haut ;
- il provoque, lorsqu'il n'est pas pressé, une tension de 0V, qui est reconnue par l'entrée comme un niveau bas.

30 – lecture-bp.py - Lire l'état d'un poussoir, et allumer une LED

```
1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 for t in range(500):      # pour que le code s'arrete au bout de 10 s
8     led.value(bp.value())
9     sleep(.02)
10
11 #for t in range(500):      # autre ecriture pour la meme chose
12 #     if bp.value() :
13 #         led.value(1)
14 #     else :
15 #         led.value(0)
16 #     sleep(.02)
```

La deuxième partie du code (lignes 11 à 16), commentée par les #, réalise la même chose que les lignes 7 à 9, mais codée différemment.

►Q.85: Oter la mise en commentaire de cette partie, et commenter les lignes 7 à 9, pour vérifier que c'est bien la même chose.

Maintenant, nous allons compter le nombre de fois que nous appuyons sur le bouton poussoir BP, et en même temps inverser l'état de la LED à chaque appui sur ce bouton poussoir ; on parle de basculement de la LED.

9.1.2 Solution de comptage qui ne fonctionne pas - NFP

Voici un code qui ne fonctionne pas. Je l'ai placé ici juste pour comprendre le piège dans lequel on pourrait tomber dans nos futurs projets. Nous ne tenterons pas de le faire fonctionner.

31 – `comptage-NFP.py` - Compter les appuis - NFP

```
1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6 compt = 0
7
8 for t in range(500):           # le programme s'arrete au bout de 10 s
9     if bp.value():
10        compt = compt + 1      # inversion de l'etat de la LED
11        print("compteur", compt) # affiche le compt si le bp est presse
12        led.value(not led.value()) # basculement de la LED
13    sleep(.02)
```

►Q.86: Vérifier que ce code Ne Fonctionne Pas (d'où l'extension -NFP dans le nom de fichier pour le rappeler). En effet, toutes les 0.02 s, si le poussoir est pressé, le basculement de la LED est réalisé, et le compteur est incrémenté. Pourtant, je ne veux incrémenter *que lors de l'appui*, et non pas lorsque je reste appuyé, soit toutes les 0,02 s.

Nous allons donc voir deux solutions qui fonctionnent :

- la première attend l'appui puis le relâchement ; elle est bloquante,
- l'autre va détecter le moment où j'appuie, c'est-à-dire le front montant ; elle n'est pas bloquante.

9.1.3 Solution de comptage bloquante

Dans l'exemple ci-dessous, le programme est toujours en train d'attendre ; il attend soit l'appui, soit le relâchement. On appelle cela un programme bloquant, car il attend quelque chose (l'appui sur un poussoir dans notre cas) pour avancer dans le code.

Ce programme fonctionne, mais bloque d'éventuelles parties de code que l'on voudrait pouvoir exécuter en plus des fonctions de détection d'appui pour le comptage. En effet, il fonctionne par scrutation, donc il passe son temps à attendre les appuis, puis les relâchements.

Remarque : Le temps de cycle de la boucle principale (la boucle `for`) est indéterminé (non connu à l'avance) car il dépend des moments où j'appuie et relâche le poussoir. Ainsi, le programme pourrait ne jamais se terminer selon les cas. Il se termine uniquement lorsque j'ai appuyé et relâché 10 fois le poussoir.

32 – `comptage-bloquant.py` - Compter les appuis - Bloquant

```
1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6 compt = 0
7
8 for cycle in range(10):       # on fait juste 10 cycles appui/relachement
9     while not bp.value():     # on attend l'appui sur le bp
10        pass                  # on ne fait rien, mais il faut le dire
11    compt = compt + 1
12    print("Compteur :", compt)
13    led.value(not led.value()) # basculement de la LED
14    sleep(0.02)               # delai pour supprimer les rebonds du bp
15    while bp.value():         # on attend l'appui sur le bp
16        pass                  # on ne fait rien, mais il faut le dire
17    sleep(.02)                # delai pour supprimer les rebonds du bp
```

►Q.87: Vérifier le fonctionnement.

►Q.88: Imaginer combien il serait difficile, avec cette solution, d'incrémenter un deuxième compteur pour compter les appuis sur un deuxième poussoir.

Ce type de fonctionnement est rarement intéressant, sauf si notre programme n'a que cette opération de comptage à réaliser. En général, ce n'est pas le cas.

9.1.4 Solution de comptage NON bloquante, par détection de front

Pour nous débarrasser de ce problème, nous allons utiliser une structure non bloquante, qui sera réalisée en utilisant les notions de «front» (détection de front montant ou de front descendant). Ainsi, nous devons tester si le poussoir vient d'être pressé, c'est-à-dire s'il est pressé dans cette itération de la boucle mais non pressé dans l'itération précédente. Le diagramme des temps figure 18 montre comment détecter un front sans structure bloquante. Voici le principe :

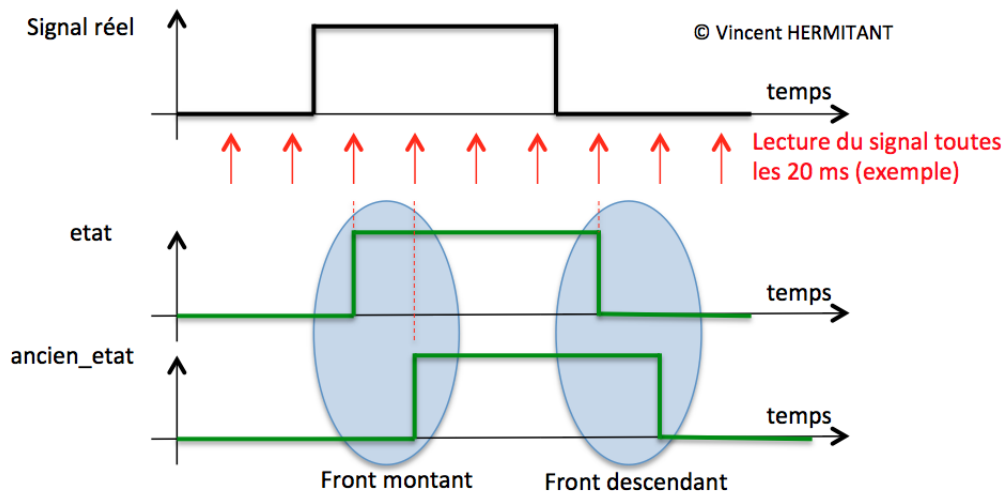


FIGURE 18 – Gestion des fronts pour éviter d'avoir une structure bloquante

- Toutes les 20 ms (c'est un exemple, un choix), une lecture est effectuée de l'état du poussoir. Cette lecture est comparée avec l'état précédent, qui doit alors être mémorisé dans une variable. Les deux variables sont appelées : **etat** et **ancien_etat**.
- La variable **etat** est assignée avec la valeur lue au début de la boucle principale. L'entrée n'est lue qu'une fois dans la boucle principale.
- Cette variable est copiée dans **ancien_etat** à la fin de la boucle principale.

Pour la première itération de la boucle, on a besoin d'affecter à la variable **ancien_etat** la valeur initiale, celle qu'on lit en ligne 7. Elle est ainsi existante lors de la première exécution de la boucle principale, qui en a besoin.

Durant l'exécution de la boucle principale, on dispose donc des deux variables, qui correspondent à la lecture du poussoir à deux moments successifs. C'est en comparant ces deux valeurs qu'on peut facilement détecter un :

- front montant si **etat** vaut 1 et **ancien_etat** vaut 0,
- front descendant si **etat** vaut 0 et **ancien_etat** vaut 1,
- front quelconque si **etat** et **ancien_etat** sont différents,
- pas de front si **etat** et **ancien_etat** sont égaux.

Il n'y a maintenant plus de structure (bloquante) dans la boucle principale ; cette boucle principale pourrait donc s'exécuter à une fréquence très élevée, vu que les quelques lignes de programme qu'elle comporte peuvent s'exécuter en quelques μ s.

On limitera donc la fréquence de la boucle principale en mettant une attente de 20 ms par exemple. *Remarque* : Le temps de cycle de la boucle principale (la boucle **for**) est *déterminé*. Il est de 20 ms, et c'est la valeur que j'ai choisie arbitrairement. Ce temps ne change pas, que j'appuie ou pas sur le poussoir. Comme j'ai programmé 500 boucles, le programme dure forcément 10 s. Ainsi, il n'est pas utile d'arrêter le programme avec le bouton STOP de Thonny IDE.

33 – `comptage-ok.py` - Compter les appuis - NON Bloquant

```
1 from machine import Pin
```

```

2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6 compt = 0
7 ancien_etat = bp.value()
8
9 for cycle in range(500):          # on fait 500 cycles, donc durant 10 s
10     etat = bp.value()             # = 1 si on appuie
11     if not ancien_etat and etat : # si front montant
12         compt = compt + 1         # on incremente
13         print("Compteur :", compt)
14         led.value(not led.value()) # basculement de la LED
15     sleep(.02)                    # temps de cycle
16     ancien_etat = etat            # memorise l'etat de la boucle avant

```

Le test de la ligne 11, qui détecte si nous sommes en présence d'un front montant, aurait pu s'écrire aussi, si c'est plus simple à comprendre pour vous :

```

if ancien_etat == 0 and etat == 1 :    # test si front montant

```

►Q.89: En déplaçant juste l'instruction `not` en ligne 11, changer le fonctionnement pour compter lors des fronts descendants.

►Q.90: Changer encore le fonctionnement pour compter lors des fronts montants ET des front descendants.

9.2 Les entrées capacitives

En plus des entrées numériques qui lisent une tension qui prend 2 niveaux logiques, l'ESP32 est muni de 10 pins que l'on peut paramétrer comme étant des entrées capacitives. Elles permettent une utilisation en tant que touches tactiles ; la valeur lue dépend de la capacité mesurée sur sa borne, et varie par exemple lorsqu'on la touche. La valeur lue est (sur la carte dont nous disposons) supérieure à 150 lorsqu'on ne la touche pas, et inférieure à 60 lorsqu'on la touche.

9.2.1 Lecture simple d'une entrée capacitive, sans comptage

Sur la carte ESP ENIM, deux de ces entrées, choisies sur les pins 4 et 15, sont utilisées ici :

34 – touchpad.py - Lire l'entrée capacitive, et allumer une LED

```

1 from machine import TouchPad, Pin
2 from time import sleep
3
4 tp1 = TouchPad(Pin(15))          # Touch Pin 1 sur pin 15
5 tp2 = TouchPad(Pin(4))           # Touch Pin 2 sur pin 4
6
7 for boucle in range(400) :       # duree de la boucle : 4 s
8     etat1 = tp1.read()            # lecture du TouchPad 1
9     etat2 = tp2.read()
10    print("Touche 1 : {0:3d}  Touche 2 : {1:3d}".format(etat1, etat2))
11    sleep(.01)

```

►Q.91: Vérifier le bon fonctionnement. Noter les valeurs relevées lorsque l'on touche, et lorsque l'on ne touche pas la «Touch Pin».

►Q.92: Ajouter quelques lignes de code pour allumer la LED bleue si la Touch Pin TP1 est touchée et l'éteindre si elle n'est pas touchée, et la même chose avec la LED verte et la Touch Pin TP2.

9.2.2 Solution de comptage bloquante

Avec ce que nous venons de voir, nous pouvons adapter le comptage au Touch Pad. Voici donc la version bloquante :

35 – touchpad-comptage-bloquant.py - Compter les appuis par TouchPad - Bloquant

```

1 from machine import TouchPad, Pin

```

```

2 from time import sleep
3
4 tp1 = TouchPad(Pin(15))
5 led_bleue = Pin(2, Pin.OUT)
6 compt = 0
7 seuil = 100
8 ancien_etat = tp1.read()
9
10 for boucle in range (10) :           # on fait juste 10 cycles, et on arrete
11     while not tp1.read() < seuil :    # on attend l'appui sur le touch pad
12         pass                          # on ne fait rien, mais il faut le dire
13     compt = compt + 1
14     print("Compteur :",compt)
15     led_bleue.value(not led_bleue.value()) # basculement de led_bleue
16     while tp1.read() < seuil :        # on attend l'appui sur le touch pin
17         pass                          # on ne fait rien, mais il faut le dire

```

►Q.93: Imaginer combien il serait difficile d'incrémenter un deuxième compteur lors des appuis sur un deuxième touch pad.

9.2.3 Solution de comptage NON bloquante, par détection de front

Comme avec les boutons poussoirs, pour détecter le front sur l'entrée capacitive, on mémorise l'état précédent. La seule différence, c'est que l'état est le résultat du test de la valeur lue, à savoir si elle est supérieure ou inférieure à 100.

36 – touchpad-comptage-front.py - Compter les appuis par TouchPad - NON Bloquant

```

1 from machine import TouchPad, Pin
2 from time import sleep
3
4 tp1 = TouchPad(Pin(15))
5 led_bleue = Pin(2, Pin.OUT)
6 compt = 0
7 seuil = 100
8 ancien_etat = tp1.read()
9
10 for boucle in range (500) :           # duree de la boucle : 10 s
11     etat = tp1.read() < seuil          # le test renvoie 1 si on touche
12     if etat and not ancien_etat :      # si front montant
13         compt = compt + 1              # on incremente
14         print("Compteur :",compt)
15         led_bleue.value(not led_bleue.value()) # bascule de led_bleue
16     ancien_etat = etat                 # memorisation ancien etat
17     sleep(.02)                         # temps de cycle

```

►Q.94: Ajouter quelques lignes de code pour décrémenter le même compteur, sur chaque front montant sur une deuxième entrée capacitive TP2 (celle sur la pin 4).

9.3 La scrutation et ses contraintes - Théorème de Shannon

J'ai choisi un temps de 0.02 s entre chaque itération, et cela permet dans notre cas un bon fonctionnement. Mais que se passerait-il si je change le temps de boucle, par exemple si je l'augmente ?

►Q.95: A partir du fichier `comptage-ok.py`, faire l'essai avec un temps de cycle de 0.25 s (et mettre dans la boucle for : `range(40)` pour garder une durée de programme de 10 s. Vérifier le dysfonctionnement, qui apparaît lorsque l'on appuie de manière assez brève sur le poussoir. Comprendre d'où vient le dysfonctionnement.

Etudions ce phénomène ... Une entrée doit être lue à une fréquence suffisante pour ne pas manquer d'évènement. On lira son état (ou sa valeur) d'une manière compatible avec la nature de l'entrée. On parle de «lecture des entrées par scrutation» («*polling*» en anglais). Prenons deux exemples :

9.3.1 Exemple pour comprendre : le tourniquet

Un tourniquet est placé à l'entrée d'un commerce. Chaque fois qu'un client passe, une barrière lumineuse est coupée, et un niveau logique passe à 1. Une lecture de ce niveau logique dix fois par seconde (donc toutes les 100 ms) permet un comptage correct du nombre de client. En effet, il paraît peu probable que deux clients coupent le faisceau lumineux dans le même dixième de seconde.

Ainsi, dans la boucle de mesure, nous choisirons un délai de 100 ms.

9.3.2 Exemple pour quantifier : le moteur sur banc d'essai

Un moteur, pouvant tourner à un maximum de 8 000 tours par minute, entraîne un disque composé de 12 bandes noires (signal à 0) séparées par 12 bandes blanches (signal à 1) lues par un capteur optique. Le signal change donc de valeur 24 fois par tour du disque, soit 192 000 fois par minute, soit 3 200 fois par seconde, soit toutes les 312.5 μ s.

Exprimé autrement, la fréquence de notre signal est de 96 000 périodes/mn (soit 192 000 / 2 car une période ou impulsion c'est 1 état haut puis 1 état bas), soit 96 000/60 = 1 600 Hz.

Pour connaître sa vitesse ou sa position, il est nécessaire de lire le signal au moins toutes les 312.5 μ s pour ne pas louper de changement d'état, et compter ces changements d'état. On peut choisir par exemple de lire le signal toutes les 250 μ s, soit une fréquence d'acquisition de $1 \div 250 \mu s = 4$ kHz. On appelle cette fréquence : *fréquence d'échantillonnage*.

Pour résumer : la fréquence du signal est de 1 600 Hz et nous avons choisi de faire les mesures à la fréquence de 4 000 Hz, qui est au moins deux fois supérieure à la fréquence du signal.

Temps	Rotation moteur	Signal disque	Echantillonnage limite	Echantillonnage choisi
Par minute	8 000 tr/min	96 000 pulse/min		
Par seconde	133,3 tr/s	1 600 Hz	3 200 éch/s	4 000 éch/s
Seconde par ...	7,5 ms/tr	625 μ s/pulse	315,5 μ s/mesure	250 μ s/mesure

TABLE 1 – Fréquences concernant le moteur, son disque, son échantillonnage

9.3.3 Théorème de Shannon

D'après le théorème de Shannon, pour ne pas manquer de période du signal, la fréquence d'échantillonnage doit être strictement supérieure à 2 fois la fréquence du signal à mesurer, soit dans le cas de notre moteur : f.éch. > 3 200 Hz. Donc 4 000 Hz convient bien ; mais 10 000 convient très bien aussi, même si c'est inutile de surcharger le traitement du signal.

9.3.4 Explication du choix du temps de boucle de 20 ms

Pour comprendre pourquoi le temps de 0.02 s est une valeur qui convient bien à notre besoin de détecter le fait de presser le poussoir, comprenons d'abord que nous faisons ainsi une lecture à la fréquence de 50 valeurs par secondes (1/0.02). Par conséquent, nous pouvons presser et relâcher le poussoir 25 fois par seconde avant d'atteindre la limite de lecture. Nous serons sans doute toujours en dessous de cette limite.

10 Mesure de temps

Nous allons mesurer le temps durant lequel nous appuyons sur un bouton poussoir, c'est-à-dire le temps pendant lequel l'entrée est à l'état Haut. Nous allons étudier quatre manières de faire, par comptage de boucle et par mesure de temps absolu (avec la méthode ticks) ; puis avec une solution bloquante (à l'aide des attentes dans des boucles While), et l'autre non bloquante (par détection de front), ce qui fait quatre combinaisons :

- une solution par comptage de boucle, bloquante,
- une solution par comptage de boucle, NON bloquante,
- une solution par mesure de temps absolu, bloquante,
- et une solution par mesure de temps absolu, NON bloquante.

Cela permettra de bien comprendre la différence, pour ne pas tomber dans le piège, et pour ne pas concevoir un programme qui bloque, et qui empêcherait d'autres parties de code de se dérouler.

10.1 Solution par comptage de boucles

Dans les deux exemples qui suivent, nous comptons le nombre d'itérations d'une boucle qui ont lieu pendant l'appui sur le poussoir. J'ai choisi un temps de boucle de 0.02 s. La résolution de notre mesure sera donc de 0.02 s.

10.1.1 Solution bloquante

Cette solution est bloquante car, par exemple en ligne 9, nous constatons que nous restons dans la boucle While tant que l'on n'appuie pas sur le poussoir.

37 – `lecture-bp-temps-bloquant.py` - Mesure du temps d'appui - Bloquant

```
1 from machine import Pin
2 from time import sleep
3
4 bp = Pin(25, Pin.IN)
5 compt = 0
6
7 for t in range(5):          # on fait juste 5 cycles appui/relachement
8     while not bp.value():    # on attend l'appui sur le bp
9         sleep(.02)           # on attend 20 ms, 50 boucles/s suffisant
10    compt = 0
11    while bp.value():         # on attend l'appui sur le bp
12        compt = compt + 1     # on incremente toutes les 20 ms
13        sleep(.02)           # delai entre deux boucles
14    print("Compteur : {0:2d}, soit : {1} s".format(compt, compt / 50))
```

Explications :

- Lignes 9 et 10 : on attend l'appui sur le poussoir en scrutant l'entrée, et on boucle dans le vide tant que son état est à zéro, avec une attente de 20 ms entre chaque lecture. Cette attente n'est pas obligatoire, mais ce n'est pas la peine d'aller plus vite dans la boucle.
- Ligne 11 : lorsque le poussoir est pressé, l'état de l'entrée devient `True`, et le compteur est mis à 0,
- Lignes 12, 13, 14 : Après cet appui, tant que le poussoir reste pressé, le compteur est incrémenté, toutes les 20 ms. Cette attente ne doit pas être enlevée, et le temps d'appui sur le poussoir sera obtenu en multipliant le nombre de boucle effectuées par ce temps de boucle.
- Ligne 15 : au relâchement, la valeur du compteur est affichée, et elle représente le nombre de cinquantièmes de secondes, que l'on traduit en secondes en multipliant par 0.02, ou en divisant par 50.

►Q.96: Tester le bon fonctionnement.

On souhaiterait, en même temps que la mesure de temps, lire l'état d'un bouton poussoir «bpB», qui allume une LED lorsqu'il est pressé, et l'éteint dans le cas contraire, comme fait précédemment. Il est impossible de réaliser cette opération en utilisant les lignes de code déjà rencontrées ...

```
1 led = Pin(2, Pin.OUT)
2 bpB = Pin(34, Pin.IN)
3 .....
4 led.value(bp2.value())
```

►Q.97: Comprendre cette impossibilité.

10.1.2 Solution NON bloquante (détection de fronts)

Contrairement à l'exemple précédent, le temps de boucle est ici défini à l'avance, il vaut 20 ms, et le programme s'arrête automatiquement au bout de 10 s car la boucle `for` va s'exécuter 500 fois. Nous utilisons à nouveau la détection des fronts pour ne pas être obligé d'attendre un événement, comme dans l'exemple précédent.

38 – `lecture-bp-temps-ok.py` - Mesure du temps d'appui - OK

```
1 from machine import Pin
```



```

2 from time import sleep
3
4 bp = Pin(25, Pin.IN)
5 compt = 0
6 ancien_etat = bp.value()           # initialisation ancien_etat
7
8 for t in range(500):               # on fait 500 cycles * 0.02 s = 10 s
9     etat = bp.value()              # lecture etat
10    if etat :                       # tant qu'on appuie...
11        compt = compt + 1           # ... on incremente
12    if ancien_etat and not etat :   # et si front descendant ! ...
13        print("Compteur : {0:2d}, soit : {1} s".format(compt, compt / 50))
14        compt = 0                  # on affiche et remet a 0
15    ancien_etat = etat              # memorisation etat boucle avant
16    sleep(0.02)                     # delai de boucle

```

Explications :

- Nous sommes dans une boucle principale rapide, qui incrémente de 1 chaque 20 ms, et cela tant que le poussoir est pressé,
- Lors du relâchement, c'est-à-dire lors du front descendant de `etat`, nous affichons la valeur du compteur, et cette valeur représente le nombre de cinquantièmes de secondes écoulés pendant le dernier appui; on la convertit en secondes en divisant par 50, on l'affiche, et on remet à 0 le compteur.
- On mémorise l'état de la boucle précédente, pour pouvoir détecter le front descendant, qui correspond à `ancien_etat = 1` et `etat = 0`

►Q.98: Tester le bon fonctionnement.

On souhaite, comme précédemment, en même temps que ce code s'exécute, ajouter une opération de lecture de l'état d'un bouton poussoir `bpB`, qui allume une LED lorsqu'il est pressé, et l'éteint dans le cas contraire. Cette fois-ci, le programme est NON bloquant, donc l'opération est possible (et facile).

►Q.99: Ajouter les 3 lignes suivantes au bon endroit pour réaliser cette lecture/écriture.

```

1 led = Pin(2, Pin.OUT)
2 bpB = Pin(34, Pin.IN)
3 .....
4 led.value(bpB.value())

```

10.2 Solution par mesure de temps absolu avec la méthode `ticks-ms`

10.2.1 Introduction à la méthode `ticks-ms`

Cette méthode `ticks_ms` est incluse dans le module `time`. Elle retourne la valeur en millisecondes du temps écoulé depuis l'instant de la mise sous tension de notre carte ESP32, ou du dernier appui sur le bouton STOP de Thonny IDE, c'est à dire la dernière connexion de la carte à l'ordinateur.

Voyons comment fonctionne cette méthode sur l'exemple suivant; lors de l'appui sur le poussoir, on affiche la valeur de `ticks_ms`, et on boucle toutes les 100 ms.

39 – `methode-ticks.py` - Utilisation de la méthode `ticks-ms`

```

1 from machine import Pin
2 from time import sleep, sleep_ms, sleep_us, ticks_ms, ticks_us
3
4 bp = Pin(25, Pin.IN)
5
6 for t in range(100):               # duree = 100 * 0.1 = 10 s
7     if bp.value() :
8         print (ticks_ms())
9     sleep(0.1)

```

►Q.100: Tester ce code.

Dans les deux solutions suivantes, pour mesurer un temps d'appui, nous mémoriserons le temps courant dans une variable, et lorsque l'appui et le relâchement ont eu lieu, nous calculerons la différence entre la nouvelle mesure du temps courant et la première que nous avons mémorisée. Voici les deux solutions, bloquantes et NON bloquantes.

10.2.2 Solution bloquante

Dans cette solution, la résolution n'est pas fixée à 0.02 s. Le temps de boucle est très rapide car il n'y a pas de fonction `sleep` dans les boucles d'attente, juste l'instruction `pass` qui ne prend pas de temps pour son exécution. La précision de la mesure est juste dépendante de celle de la fonction `ticks_ms`.

40 – `lecture-bp-temps-ticks-bloquant.py` - Mesure du temps d'appui par ticks - Bloquant

```
1 from machine import Pin
2 from time import sleep, sleep_ms, sleep_us, ticks_ms, ticks_us
3 cont = True
4
5 led = Pin(2, Pin.OUT)
6 bp = Pin(25, Pin.IN)
7
8 for t in range(5):          # on fait juste 5 cycles appui/relachement
9     while not bp.value():    # on attend l'appui sur le bp
10        pass                 # pas besoin de tempo
11    start = ticks_ms()        # memorisation de start lors du front montant
12    while bp.value():        # on attend l'appui sur le bp
13        pass                 # pas besoin de tempo
14    stop = ticks_ms()         # stop sur front montant 2
15    delta = stop - start
16    print("Temps d'appui : {0} ms".format(delta))
```

On souhaiterait, comme précédemment, en même temps que la mesure de temps, lire l'état d'un bouton poussoir «bpB», qui allume une LED lorsqu'il est pressé, et l'éteint dans le cas contraire, comme fait précédemment. Il est impossible de réaliser cette opération à cause encore du caractère bloquant de ce code.

►Q.101: Comprendre qu'il est impossible de réaliser cette opération.

10.2.3 Solution NON bloquante (détection de fronts)

41 – `lecture-bp-temps-ticks-ok.py` - Mesure du temps d'appui par ticks - NON bloquant

```
1 from machine import Pin
2 from time import sleep, sleep_ms, sleep_us, ticks_ms, ticks_us
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 ancien_etat = bp.value()    # initialisation ancien_etat
8 for t in range(10000):      # duree = 10000 * 0.001 = 10 s
9     etat = bp.value()       # lecture etat
10    if etat and not ancien_etat : # condition vrai : front montant
11        start = ticks_ms()     # memorisation de start
12    if not etat and ancien_etat : # condition vrai : front descendant
13        stop = ticks_ms()      # stop sur front montant 2
14        delta = stop - start    # calcul du temps d'appui
15        print("Temps d'appui : {0} ms".format(delta))
16    ancien_etat = etat         # memorisation ancien-etat
17    sleep_ms(1)
```

Etant donné que la structure est non bloquante, nous pouvons ajouter du code pour allumer la LED bleue lors de l'appui sur le poussoir bpB.

►Q.102: Ajouter ce code.

10.3 Application - Mesure de temps entre deux événements

Considérons deux capteurs qui vont détecter le passage d'un objet, et passer chacun à leur tour à l'état Haut au moment de la présence de l'objet devant la cellule. Cela pourrait permettre de mesurer la vitesse moyenne entre les deux capteurs, si l'on connaît précisément la distance entre les deux capteurs. Simulons chaque capteur par des poussoirs bpA et bpB. Nous allons voir deux solutions très proches :

10.3.1 Solution 1 : utilisation d'un compteur de boucles

C'est une solution qui ressemble à ce que nous avons déjà pratiqué dans le cas précédent de la mesure de temps d'appui. Attention ! Ceci est une solution bloquante, mais nous supposons que la problème est juste de mesurer le temps de passage de l'objet entre les 2 capteurs, et que aucune opération n'a besoin d'être effectuée en même temps. Donc voici :

- Nous attendons le front montant de bpA.
- Ensuite nous attendons le front montant de bpB, et pendant cette attente nous incrémentons chaque milliseconde la variable `compt` tant que bpB reste à l'état Bas, c'est-à-dire tant que nous n'appuyons pas dessus.
- Nous arrêtons de compter au moment de l'appui sur bpB (donc du front montant) ; la valeur de `compt` représente le temps en millisecondes.

42 – `mesure-temps-deux-fronts-1.py` - Mesure du temps entre deux fronts - solution 1

```
1 ##### mesure de temps entre 2 fronts #####
2 from machine import Pin
3 from time import sleep, sleep_ms, sleep_us
4 compt = True
5
6 led = Pin(2, Pin.OUT)
7 bpA = Pin(25, Pin.IN)
8 bpB = Pin(34, Pin.IN)
9 compt = 0
10
11 print("Attente du front montant de bpA")
12 while not bpA.value(): # attente a l'etat bas du front montant
13     pass
14 led.value(1)
15
16 print("Attente du front montant de bpB")
17 while not bpB.value(): # attente a l'etat bas du front montant
18     compt = compt + 1
19     sleep_ms(1)
20 led.value(0)
21
22 print("Temps entre deux fronts : {0} ms \n".format(compt))
```

►Q.103: Vérifier le bon fonctionnement.

Imaginons que le temps entre les deux événements soit très court, de l'ordre de la milliseconde, par exemple compris entre 100 μs et 10 ms. Nous avons donc besoin d'une précision supérieure à la milliseconde.

►Q.104: Modifier alors le programme pour attendre, mesurer et indiquer des temps en microsecondes. Vérifier que cela ne fonctionne pas très bien. L'explication est juste ci-après...

Attention au temps de boucle Pour faire cette question, vous avez sans doute changé la fonction `sleep_ms(1)` par `sleep_us(1)`. Dans les faits, cette manière de faire ne fonctionne pas correctement ; en effet, même si la fonction `sleep_us(1)` attend assez précisément 1 μs , les autres instructions dans la boucle (test de l'appui sur le poussoir, et incrémentation de la variable `compt`) demandent pour leur exécution quelques μs également, donc le temps de boucle réel sera de plusieurs μs et non 1 μs ; donc nous faisons moins de boucle que prévu ; le temps mesuré est donc faux, il est plus petit que la réalité.

10.3.2 Solution 2 : utilisation de la fonction `ticks_ms()`

Dans cette solution, nous attendons le front montant de bpA pour déclencher un top de départ, et enregistrer la valeur du temps dans la variable `start`. Nous attendons ensuite le front montant de bpB pour déclencher un top de fin, et enregistrer la valeur du temps dans la variable `stop`. La variable `delta` représente le temps entre les deux fronts.

43 – `mesure-temps-deux-fronts-2.py` - Mesure du temps entre deux fronts - solution 2

```
1 ##### mesure de temps entre 2 fronts #####
2 from machine import Pin
```

```

3 from time import sleep, sleep_ms, sleep_us, ticks_ms, ticks_us
4 cont = True
5
6 led = Pin(2, Pin.OUT)
7 bpA = Pin(25, Pin.IN)
8 bpB = Pin(34, Pin.IN)
9
10 print("Attente du front montant de bpA")
11 while not bpA.value(): # attente a l'etat bas
12     pass
13 start = ticks_ms()      # start sur front montant 1
14 led.value(1)
15
16 print("Attente du front montant de bpB")
17 while not bpB.value(): # attente a l'etat bas
18     pass
19 stop = ticks_ms()       # stop sur front montant 2
20 led.value(0)
21
22 delta = stop - start
23 print("Temps entre deux fronts : {0} ms \n".format(delta))

```

►Q.105: Vérifier le bon fonctionnement.

Imaginons encore une fois que le temps entre les deux évènements soit très court, par exemple de l'ordre la milliseconde, et donc que nous ayons besoin d'une précision supérieure à la milliseconde.

►Q.106: Modifier le programme pour mesurer et indiquer le temps en microsecondes.

Ainsi, avec cette méthode, le problème de temps de boucle rencontré précédemment n'a plus d'importance.

10.3.3 Solution avec ticks_ms(), et ajout d'une précaution

Par précaution, nous pouvons tester aussi si l'objet n'est pas déjà devant la cellule alors qu'il ne devrait pas y être, et attendre qu'il en sorte. Cela pourrait arriver dans certaines configurations, ou si l'on exécute plusieurs cycles qui se chevauchent. On attend alors l'état Bas de A avant son état Haut.

44 – `mesure-temps-deux-fronts-3.py` - Mesure du temps entre deux fronts

```

1 ##### mesure de temps entre 2 fronts #####
2 from machine import Pin
3 from time import sleep, sleep_ms, sleep_us, ticks_ms, ticks_us
4 cont = True
5
6 led = Pin(2, Pin.OUT)
7 bpA = Pin(25, Pin.IN)
8 bpB = Pin(34, Pin.IN)
9
10 print("Attente de l'etat BAS de A")
11 while bpA.value():      # attente a l'etat haut
12     pass
13
14 print("Attente du front montant de bpA")
15 while not bpA.value(): # attente a l'etat bas
16     pass
17 start = ticks_ms()      # start sur front montant 1
18 led.value(1)
19
20 print("Attente de l'etat BAS de B")
21 while bpB.value():      # attente a l'etat haut
22     pass
23
24 print("Attente du front montant de bpB")
25 while not bpB.value(): # attente a l'etat bas
26     pass
27 stop = ticks_ms()       # stop sur front montant 2

```

```

28 led.value(0)
29
30 delta = stop - start
31 print("Temps entre deux fronts : {0} ms \n".format(delta))

```

►Q.107: Vérifier le bon fonctionnement.

10.4 Application : Allumage progressif et extinction progressive

Dans l'exemple suivant, l'appui (suivi du relâchement) provoque l'allumage progressif de la LED bleue. Puis l'appui (suivi du relâchement) provoque l'extinction progressive de cette même LED. Ce programme est bloquant, car rien d'autre ne peut être effectué durant l'attente de l'appui et du relâchement.

45 – **pwm-allume-bp.py** - Allumage et extinction progressifs suite à un appui sur poussoir

```

1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # initialise la frequence a 500 Hz
5 bp = Pin(25, Pin.IN)
6
7 def correction(i, coef) :
8     return int ((i / 1023) ** coef * 1023)
9
10 while True :
11
12     while not bp.value() : pass      # attente de l'appui
13     while bp.value() : pass          # attente du relachement
14
15     for i in range(0, 1023, 10) :    # allumage de la LED
16         led.duty(correction(i, 3.5)) # utilisation fonction correction
17         sleep_ms(10)
18
19     while not bp.value() : pass      # attente de l'appui
20     while bp.value() : pass          # attente du relachement
21
22     for i in range(0, 1023, 10) :    # extinction de la LED
23         led.duty(correction(1023-i, 3.5))
24         sleep_ms(10)

```

►Q.108: Vérifier le bon fonctionnement, et vérifier que c'est bien au relâchement que l'allumage et l'extinction ont lieu.

►Q.109: Inverser ce fonctionnement, c'est-à-dire que c'est maintenant lors de l'appui qu'a lieu le cycle d'allumage et d'extinction.

11 Comportement multitâches

Le programme que nous venons de voir a un déroulement purement séquentiel, c'est-à-dire que l'action suivante ne démarre que lorsque l'action précédente est terminée. Le déroulement du programme est donc linéaire, c'est-à-dire qu'il n'y a jamais plusieurs opérations qui se déroulent en parallèle. Mais en général, dans les programmes plus élaborés, différentes actions se déroulent en même temps, et à des fréquences différentes. Imaginons un système qui doit par exemple faire toutes ces opérations en même temps :

- effectuer la lecture d'un poussoir à la fréquence de 50 lectures par seconde qui permettra de déclencher une action,
- effectuer une régulation de température toutes les 2 secondes (c'est-à-dire lecture de la température, calcul du signal de commande, et envoi de la nouvelle commande de chauffage),
- effectuer une régulation de vitesse d'un moteur toutes les 5 ms,
- recevoir de l'utilisateur la nouvelle valeur de consigne de température.

Faire une seule action à la fois est souvent très facile. La difficulté est de faire tourner toutes les opérations en même temps. Par exemple, il ne faut pas qu'une partie du code soit bloquante. Voyons maintenant plusieurs structures de code.

11.1 Structure de base - le séquenceur bloquant

Cette structure permet d'exécuter dans un certain ordre différentes tâches. Par exemple, le code suivant permet de faire clignoter à une fréquence de 1 Hz (période de 1 s, soit 500 ms à l'état haut et 500 ms à l'état bas), et avec un décalage d'une demi-période entre les deux LED, soit un déphasage d'un quart de période. La LED verte est en retard de 0,25 s sur la bleue.

46 – `sequenceur.py` - Séquenceur de 2 LED en décalage

```
1 from machine import Pin
2 from time import sleep
3
4 led_bleue = Pin(2, Pin.OUT)
5 led_verte = Pin(18, Pin.OUT)
6
7 while True:
8     led_bleue.on() ; sleep(0.25)
9     led_verte.on() ; sleep(0.25)
10    led_bleue.off() ; sleep(0.25)
11    led_verte.off() ; sleep(0.25)
```

►Q.110: Vérifier le bon fonctionnement.

►Q.111: Comprendre qu'il serait difficile, à partir de code, de faire clignoter une troisième LED à une fréquence différente, par exemple 5 Hz, de manière indépendante des deux premières, qui continuent leur clignotement régulier. En effet, durant les moments d'attente, le séquenceur ne fait rien d'autre et il est bloqué dans l'attente jusqu'à son terme. Il faudrait, pour pouvoir faire autre chose avant le terme, découper ce temps d'attente pour intercaler la troisième LED, et la faire s'allumer puis s'éteindre.

►Q.112: Comprendre qu'il serait difficile également de scruter l'état d'un bouton poussoir, pour détecter si l'on appuie dessus. En effet, durant les 0.25 s d'attente, le poussoir aura pu être pressé et relâché, sans que le code ne l'ait détecté.

11.2 Structure améliorée par une boucle rapide - Séquenceur non bloquant

Voici un code qui réalise la même chose que précédemment, à la différence que le programme n'est pas bloquant. En effet, il sera possible de faire quelque chose en parallèle, c'est à dire en même temps que le clignotement des deux LED.

47 – `boucle-rapide-simple.py` - Séquenceur de 2 LED en décalage

```
1 from machine import Pin
2 from time import sleep, sleep_ms
3
4 led_bleue = Pin(2, Pin.OUT)
5 led_verte = Pin(18, Pin.OUT)
6 compt = 0
7
8 while compt < 400 :                # on boucle durant 4 s (400 * 10 ms)
9     compt +=1                      # on incremente chaque 10 ms
10    if compt % 50 == 0 :            # compt % 50 est vrai chaque 0.5 s
11        led_bleue.value(not led_bleue.value()) # on bascule la LED bleue.
12    if (compt + 25) % 50 == 0 :     # vrai chaque 0.5 s, mais 0.25 s plus tot
13        led_verte.value(not led_verte.value()) # on bascule la LED verte
14    sleep_ms(10)                    # temps de cycle 10 ms
```

►Q.113: Ajouter une LED jaune, qui clignotera à la fréquence de 5 Hz, soit une période de 0.2 seconde, soit 0.1 s pour l'état bas, et 0.1 s pour l'état haut.

►Q.114: Ajouter un bouton poussoir qui permet, lors de son appui (front montant), de faire basculer l'état de la LED rouge (à chaque appui on change l'état). On utilise les outils de la structure non bloquante vue précédemment, c'est-à-dire la mémorisation des états courant et précédents.

Augmentation de la fréquence de la boucle. Peut être aurons nous besoin de faire clignoter une nouvelle LED à la fréquence de 100 Hz (ou toute autre opération d'ailleurs). Cela pose une difficulté car même au maximum de fréquence, notre LED ne peut basculer que chaque 10 ms, ce qui fait une période de 20 ms, soit une fréquence de 50 Hz.

Si nous voulons aller plus vite, il faudra changer le temps de boucle, et passer par exemple à 1 ms, ce qui permettra de réaliser une opération chaque milliseconde. Voir le code suivant, qui fait clignoter la LED à 500 Hz, car elle bascule toutes les 1 ms avec l'instruction `if compt % 1 == 0` :

48 – `boucle-rapide-plus.py` - Structure améliorée par une boucle time très rapide (1 ms)

```
1 from machine import Pin
2 from time import sleep, sleep_ms, sleep_us
3
4 led_bleue = Pin(2, Pin.OUT)
5 led_verte = Pin(23, Pin.OUT)
6 compt = 0
7
8 while compt < 4000 :           # on arrete la While apres 4 secondes
9     compt +=1                  # on incremente chaque 1000 us
10    if compt % 1 == 0 :         # compt % 1 est vrai a chaque boucle
11        led_bleue.value(not led_bleue.value()) # on bascule la LED bleue
12    sleep_us(1000)              # temps de cycle 1 ms
```

►Q.115: Tester ce code.

►Q.116: Mesurer le temps réel que met la boucle pour faire les 4000 itérations.

Pour cela, on utilise la fonction `ticks_us()` avant et après, puis la différence des deux correspond au temps d'exécution.

Ce temps que nous venons de mesurer devrait être de 4 s exactement ; pourtant, nous constatons qu'il est plus grand, soit 4 200 000 μs environ ; en effet, les autres opérations de la boucle ont besoin d'un temps aussi, non négligeable, de l'ordre de quelques μs , pour s'exécuter. Donc la boucle ne se fait pas en 1 000 μs comme je souhaite, mais un peu plus, et peut être environ 1 050.

►Q.117: Modifier la valeur du délai (qui est de 1000 μs), pour corriger ce décalage.

On trouvera peut être cette valeur dans le fichier `machine.py`.

EXO3 : serrure à code

►Q.118: Mémoriser dans une liste l'ordre dans lequel on appuie sur deux boutons poussoirs.

►Q.119: Lorsque une liste est reconnue (une enregistrée), mettre à l'état haut 1 s la sortie 2 ; sinon, toutes les 3 s, on recommence, c'est-à-dire que l'on remet la liste à zéro.