

Informatique sur Systèmes Embarqués

Introduction au langage MicroPython
sur le microcontrôleur

ESP32

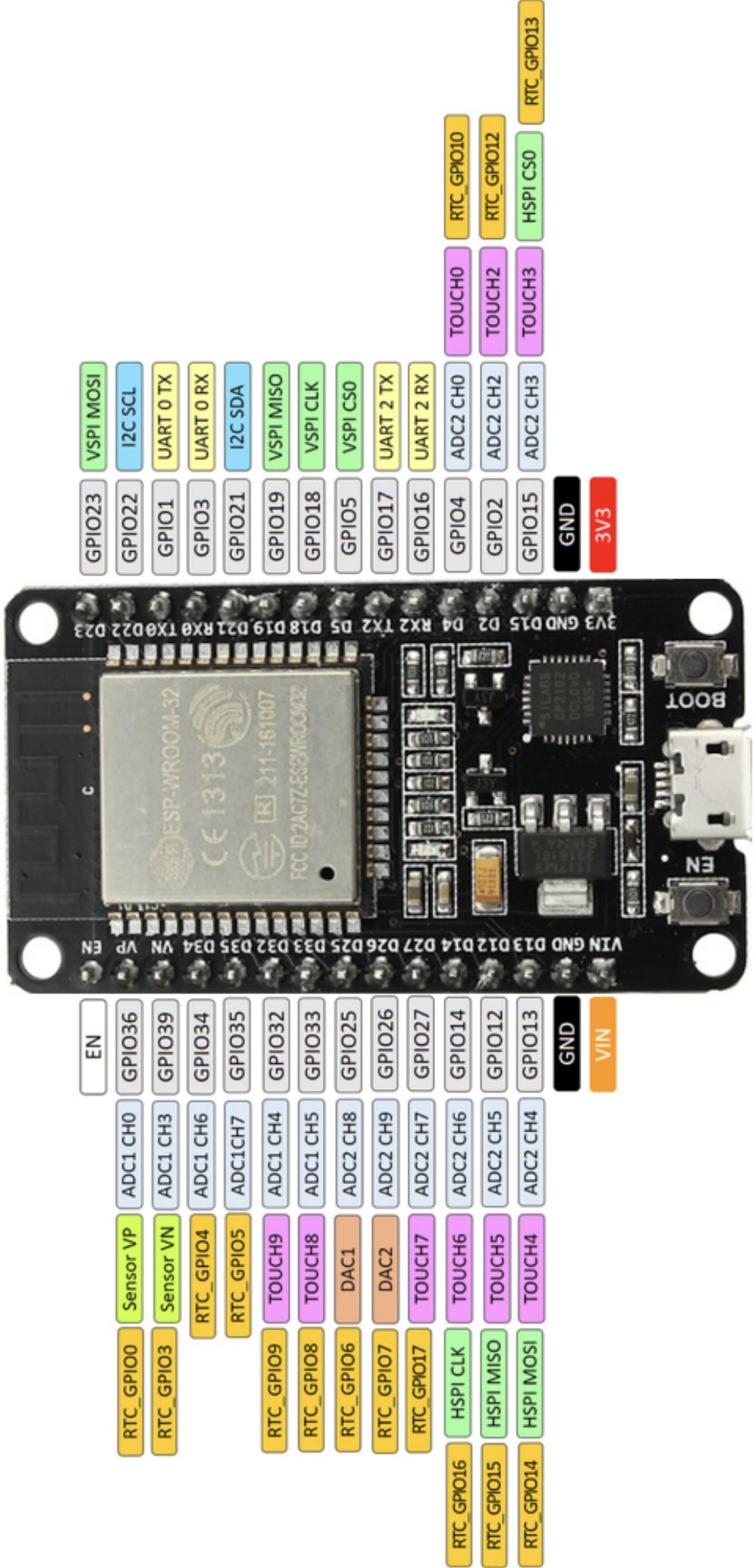


ENIM 2021

Vincent HERMITANT

ESP32 DEVKIT V1 - DOIT

version with 30 GPIOs



- Informatique sur systèmes embarqués -
Introduction au langage MicroPython
sur la base du micro-contrôleur ESP32

Vincent HERMITANT

21 janvier 2021

1 Introduction

1.1 Définition du terme « système embarqué »

On qualifie de « système embarqué » un système électronique et informatique autonome dédié à une tâche précise, souvent en temps réel, possédant une taille limitée et ayant une consommation énergétique restreinte. Ils désignent aussi bien le matériel que le logiciel utilisés.

Pour leur conception, il est nécessaire généralement de combiner des compétences en électronique, en informatique industrielle et en automatique. De tels systèmes sont nombreux dans des secteurs aussi variés que l'aéronautique, l'électroménager, le matériel médical, la téléphonie mobile, etc . . . Leurs cahiers des charges comportent plusieurs contraintes, dont :

- une puissance de calcul définie au plus juste afin de répondre aux besoins tout en respectant les contraintes temporelles et spatiales, l'objectif étant d'éviter les surcoûts et les éventuelles surconsommations d'énergie ;
- une sûreté de fonctionnement qui demande aux systèmes dits critiques de fournir des résultats exacts et pertinents ;
- une sécurité indispensable pour assurer la confidentialité des données utilisées, notamment pour les systèmes employés au service de la santé.

Les systèmes embarqués les plus basiques ne disposent, pour seule interface utilisateur, que de simples boutons ou LED. D'autres peuvent présenter un écran tactile ou un « joystick » qui permet de naviguer sur l'écran. D'autres enfin, sont connectés au réseau.

1.2 Résumé de ce que nous allons apprendre

L'ensemble «ESP32 et MicroPython» permet de piloter (c'est-à-dire commander) tout système mécanique, électronique, technologique, banc d'essai, par un langage de haut niveau. Ce langage est très puissant, largement utilisé dans le monde, et il est interprété sur une carte à microcontrôleur récente (sortie en 2016), puissante, équipée en Wifi et Bluetooth, et d'un coût de moins de 5 euros (en achat direct de Chine). Cette carte à micro-contrôleur est placée au cœur d'un écosystème de capteurs et actionneurs équipant l'installation, le prototype, ou le système.

Ce support de travaux pratiques a pour objectif de vous apprendre à programmer en MicroPython les systèmes embarqués, et en particulier l'ESP32. Nous apprendrons ce qu'est un microcontrôleur, comment le programmer, comment l'implémenter dans son système à piloter, et quels sont les composants à connecter autour. Nous commencerons par comprendre ce que signifie commander un système.

Nul besoin de connaître le langage Python, il sera fait une introduction sur les concepts de base, l'utilisation et création des fonctions, les structures de contrôle conditionnel et les boucles, la notion de classe et d'objet.

1.3 Déroulement des séances de Travaux Pratiques sur 6 séances

- Séance 1 : Support de TP (4h)
- Séance 2 : Support de TP (4h)
- Séance 3 : Support de TP (3h30) + discussion sur le projet (30 min)
- Séance 4 : Support de TP (3h30) + discussion sur le projet (30 min)
- Séance 5 : Support de TP (2h30) + interro de cours (30 min) + début du projet (1h)

- Séance 6 : Support de TP (2h) + interro de TP (45 min) + suite du projet (1h)
- En dehors des séances de TP : Suite du projet sur 4h
- En dehors des séances de TP : Soutenance du projet

1.4 Utilisation de ce support de TP

Les exercices de ce support sont très progressif, il est nécessaire de les faire dans l'ordre.

Tout au long de ce support de TP, de nombreux liens sont disponibles, il sont mis en évidence par une couleur bleue. Ils amènent vers des pages à lire pour plus d'informations, ou alors vers un dépôt Github qui permet de télécharger les codes cités en exemple.

Attention : Pour que les liens de ce support fonctionnent, il est nécessaire de télécharger le fichier PDF. Donc, *ne pas le lire en ligne*.

►Q.1: Télécharger le support PDF de cet ouvrage sur le lien vers mon dépôt Github :

<https://github.com/vincent-herm/ESP32-Les-bases>

1.5 Lien vers les principaux sites

Pas besoin de cliquer sur ces liens maintenant, c'est juste placé ici pour faciliter vos recherches au fil des questions.

Lien vers mon dépôt Github : <https://github.com/vincent-herm/ESP32-Les-bases>

Lien vers le site officiel de MicroPython

Téléchargement de la dernière version du Firmware du ESP32

Librairies de MicroPython : docs.MicroPython.org/en/latest/library/index.html

Le langage MicroPython : docs.MicroPython.org/en/latest/reference/index.html

Référence rapide pour ESP32 : docs.MicroPython.org/en/latest/esp32/quickref.html

Lien concernant les interruptions

Pas besoin de cliquer sur ces liens de la liste ci-dessous, c'est juste placé ici pour faciliter vos recherches au fil des questions.

Liste des codes mis en œuvre

1	test-multiple.py - Test si n est multiple de x	20
2	test-multiple-gestion-erreur.py - Test si n est multiple de x, et gestion erreur	21
3	liste-append.py - Création d'une liste par ajout d'éléments avec la méthode «append»	21
4	pgcd.py - Exemple du calcul du PGCD de deux nombres	22
5	pgcd2.py - Exemple du calcul du PGCD de deux nombres - version simplifiée	23
6	classe-1.py - Première approche des classes	24
7	classe-2.py - Deuxième approche des classes	24
8	classe-3.py - Troisième approche des classes	25
9	classe-4.py - Quatrième approche des classes	25
10	classe-5.py - Cinquième approche des classes	25
11	led.py - Fait clignoter une LED à 1 Hz ; solution 1	29
12	led-3-fois.py - Fait clignoter 3 fois une LED, puis s'arrête	30
13	led-temps.py - Fait clignoter 10 000 fois une LED, mesure le temps	30
14	led-bouton.py - Allume une LED si un bouton est pressé	31
15	led-stop.py - Fait clignoter une LED à 1 Hz avec STOP par bouton	32
16	tempo-eclairage.py - Temporisateur d'éclairage	32
17	tempo-eclairage-optimise.py - Temporisateur d'éclairage optimisé	33
18	pwm.py - PWM - 3 cycles de montée	35
19	calcul-correction.py - Calcul correction luminosité	36
20	pwm-correction.py - PWM avec correction luminosité	36
21	pwm-correction-input.py - PWM avec correction luminosité choisie	36
22	pwm-fonction-correction.py - PWM avec correction luminosité par fonction	37
23	neopixel.py - Test du ruban NeoPixel	37
24	neopixel-test.py - Test du ruban NeoPixel - Deuxième essai	38
25	neopixel-defilement.py - Défilement des LED	38
26	neopixel-x-allumes.py - L'utilisateur choisit d'allumer x LED	39

27	neopixel-random.py	- On allume x LED de manière aléatoire	39
28	tirage-aleatoire.py	- Tirage de 20 valeurs entières entre 0 et 8	40
29	neopixel-rainbow.py	- Arc-en-ciel sur NéoPixel	40
30	lecture-bp.py	- Lire l'état d'un poussoir, et allumer une LED	42
31	comptage-NFP.py	- Compter les appuis - NFP	43
32	comptage-bloquant.py	- Compter les appuis - Bloquant	43
33	comptage-ok.py	- Compter les appuis - NON Bloquant	44
34	comptage-thread.py	- Compter les appuis - NON Bloquant	45
35	touchpad.py	- Lire l'entrée capacitive, et allumer une LED	46
36	touchpad-comptage-bloquant.py	- Compter les appuis par TouchPad - Bloquant . . .	46
37	touchpad-comptage-ok.py	- Compter les appuis par TouchPad - NON Bloquant . . .	47
38	lecture-bp-temps-bloquant.py	- Mesure du temps d'appui - Bloquant	48
39	lecture-bp-temps-ok.py	- Mesure du temps d'appui - OK	49
40	methode-ticks.py	- Utilisation de la méthode ticks-ms	50
41	lecture-bp-temps-ticks-bloquant.py	- Mesure du temps d'appui par ticks - Bloquant .	50
42	lecture-bp-temps-ticks-ok.py	- Mesure du temps d'appui par ticks - NON bloquant .	51
43	mesure-temps-deux-fronts-1.py	- Mesure du temps entre deux fronts - solution 1 . .	51
44	mesure-temps-deux-fronts-2.py	- Mesure du temps entre deux fronts - solution 2 . .	52
45	mesure-temps-deux-fronts-3.py	- Mesure du temps entre deux fronts	53
46	pwm-allume-bp.py	- Allumage et extinction progressifs suite à un appui sur poussoir .	53
47	sequencEUR.py	- Séquenceur de 2 LED en décalage	54
48	boucle-rapide-simple.py	- Séquenceur de 2 LED en décalage	55
49	boucle-rapide-plus.py	- Structure améliorée par une boucle time très rapide (1 ms) .	55
50	interruption1.py	- Principe de l'interruption	57
51	interruption2.py	- Amélioration de la gestion de l'interruption	57
52	interruption3.py	- L'interruption allume la LED durant 3s	58

1.6 Commander un système

1.6.1 Point commun entre tous les systèmes/objets

Quel est le point commun entre une trottinette électrique, une chaudière d'appartement, une plaque à induction programmable, un radio réveil, et une voiture ?

Réponse : ils sont tous pilotés par un système électronique/informatique, c'est à dire qu'ils contiennent un système de commande, que nous pouvons appeler «contrôleur».

Tous les objets, systèmes, machines, de ce monde sont pilotés ; cela couvre les domaines et objets aussi divers que la machine à café, la serrure à carte de l'hôtel, un drone, une station météo en mer ou un ballon sonde, ...

Ce contrôleur effectue plusieurs opérations :

- Il reçoit des informations de type numérique, ou analogique. Pour cela, un ensemble de capteurs du système à commander est connecté sur ses broches d'entrée,
- Il fournit en réponse des informations destinés à piloter des actionneurs ou toute sorte de système en sortie (LED, relais, transistor, ...), l'information pouvant être sous format analogique, ou numérique.
- Il peut interagir avec un opérateur par le biais d'une IHM (Interface Homme / Machine), par exemple un clavier, écran, bouton, potentiomètres de réglage, ...
- Il peut être connecté via différents réseaux au monde extérieur.

Ainsi, le système de commande est un ensemble électronique/informatique qui est intégré dans une machine mécanique, électrique, et tout objet ou système qui a besoin d'être commandé, régulé, et qui a besoin d'interagir avec son environnement par le biais de capteurs.

1.6.2 Système de contrôle embarqué - systèmes embarqués

Les systèmes embarqués sont de plus en plus présents dans notre société où la technologie est omniprésente. Ces systèmes embarqués concernent souvent des applications souvent critiques et/ou temps-réel, et sont soumis à diverses contraintes non fonctionnelles comme l'occupation mémoire ou la consommation d'énergie.

Ils sont souvent «autonomes» et «intelligents», et doivent répondre également à d'autres problématiques comme la robustesse, la sécurité, la fiabilité et la sûreté de fonctionnement. Avec la généralisation des techniques d'échange de données, les systèmes embarqués sont maintenant «communicants».

1.7 Le langage MicroPython et les microcontrôleurs

1.7.1 Le microcontrôleur

Un microcontrôleur est un circuit intégré qui rassemble en un seul circuit (ou puce) les éléments essentiels d'un ordinateur : processeur, mémoires, unités périphériques et interfaces d'entrées-sorties (GPIO : Global Purpose Input Output). Il réunit toutes les fonctions requises pour lire des informations, traiter des données, et écrire en retour des informations vers le monde physique.

Ainsi, les microcontrôleurs sont fréquemment utilisés dans la gestion des moteurs automobiles, les télécommandes, les appareils de bureau, récepteurs GPS, l'électroménager, les jouets, la téléphonie mobile, etc...

Mais par rapport aux microprocesseurs polyvalents utilisés dans les ordinateurs personnels, les microcontrôleurs se caractérisent par un moins haut degré d'intégration, une plus faible consommation électrique, une vitesse de fonctionnement plus faible, (de quelques MHz jusqu'à environ un GHz), une mémoire assez limitée, et un coût réduit. Ils sont ainsi beaucoup moins puissants. A titre de comparaison, la fréquence d'horloge d'un ordinateur en 2020 est d'environ 3 GHz.

Les microcontrôleurs sont fréquemment utilisés dans les systèmes embarqués.

1.7.2 Le langage MicroPython

Le langage MicroPython est une réécriture de Python 3, destinée aux microcontrôleurs. Le tour de force des développeurs a été de réussir à rendre utilisable un langage aussi puissant que Python dans un environnement aussi limité que le microcontrôleur. A part quelques exceptions, les fonctions disponibles dans Python le sont aussi dans MicroPython.

Mais MicroPython n'est pas pourvu de la librairie complète de Python, mais seulement une portion bien choisie, ce qui lui permet d'être intégré et de fonctionner sous ces contraintes matérielles sévères, et un espace mémoire réduit.

Par contre, MicroPython contient en plus les librairies qui permettent l'accès matériel, de bas niveau, pour interagir facilement avec les entrées/sorties, et tout ce qui concerne la gestion de la wifi, et des connections réseau.

Ainsi, MicroPython offre au microcontrôleur des services assez proches d'un système d'exploitation, car il prend en charge le stockage et le transfert de fichiers dans la mémoire Flash, le support de la connexion Wifi-Ethernet, et autres services.

1.7.3 Le langage MicroPython associé

L'avantage d'associer MicroPython et l'ESP32 (par exemple) est de rendre l'électronique numérique aussi simple que possible. La démarche est rendue tellement simple qu'il suffit de se former 2 jours environ sur MicroPython et l'ESP32, pour être capable de développer tout seul des applications qui vous auraient semblé magiques un peu plus tôt ; seules quelques bases d'électronique et de programmation sont utiles.

Une autre plateforme a eu et a toujours eu beaucoup de passionnés, pour son aspect pratique et Open Source, c'est ARDUINO. Mais aujourd'hui, la puissance de la plateforme à base de MicroPython est telle que ARDUINO pourrait perdre de son aura. En effet, l'ESP32 est beaucoup plus puissant, interprète un code MicroPython beaucoup pratique que le C de l'ARDUINO à prendre en main. Les cas pour lesquels coder en langage C reste préférable sont très limités.

C'est pour cette raison que les travaux pratiques à l'ENIM sont réalisés aujourd'hui sur l'ESP32 alors qu'ils étaient en 2019 réalisés sur ARDUINO.

1.7.4 Exemple de code MicroPython

Exemple de code pour faire clignoter une LED, en la connectant sur une sortie numérique.

```
1 from machine import Pin
2 from time import sleep
3 led = Pin(2, Pin.OUT)
4 while True:
```

```

5     led.value(not led.value())
6     sleep(0.5)

```

Mais soulevons deux questions :

- Pourquoi utiliser le langage Python / MicroPython ?
- Pourquoi vouloir utiliser des appareils si petits et si peu puissants ?

1.7.5 Pourquoi utiliser le langage Python ?

Python est un langage de programmation informatique très puissant, facile d'accès, très répandu, et très expressif ; c'est à dire qu'en Python, on écrit du code qui reste très facile à lire, comme un langage naturel (anglais). De plus, Python tient sa vaste popularité à sa vaste communauté bien organisée et très active. En 2016, Python était le troisième langage le plus utilisé dans le monde, derrière le C, et le Java. et prend la place de leader en 2017. Depuis, [Python conforte sa place de leader en 2018 selon l'IEEE](#), la plus grande organisation mondiale de professionnels pour le développement de la technologie. En effet, Python qui a pris la tête du classement général des langages de programmation, dépassant Java et C, sur une bonne partie des critères suivants : popularité générale, langages en forte croissance, langages les plus demandés par les employeurs, les meilleurs langages pour le développement de sites et applications web, pour le développement d'applications mobiles, pour le développement d'applications d'entreprise, de bureau et scientifiques, et pour le développement de systèmes embarqués.

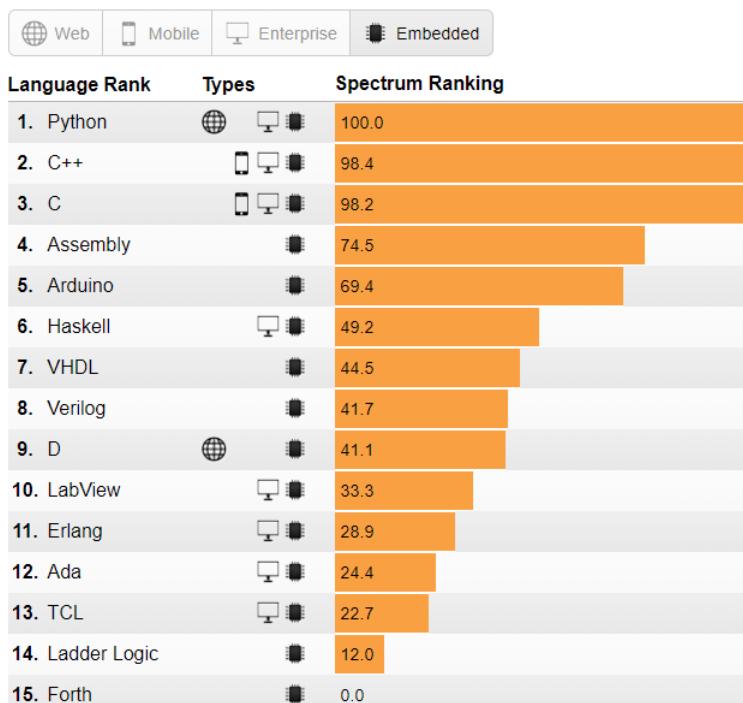


FIGURE 1 – Meilleurs langages pour le développement de systèmes embarqués (selon IEEE en 2018)

Toujours d'après ce même article, et toujours d'après [l'IEEE](#) , la popularité de Python « s'explique d'abord par le fait que Python est maintenant répertorié en tant que langage pour l'embarqué. L'écriture d'applications embarquées était la chasse gardée des langages compilés (comme le C, le C++, ...), parce que cela impliquait des machines avec une puissance de traitement et une mémoire limitées. Mais aujourd'hui, beaucoup de microcontrôleurs modernes ont assez de puissance pour héberger un interpréteur Python. Et un aspect intéressant de l'utilisation de Python dans ce domaine serait qu'il est très pratique dans certaines applications de communiquer avec du matériel via une invite interactive ou de recharger dynamiquement des scripts à la volée. »

Trois avantages cumulés Grace à l'efficacité de la reformulation de Python en MicroPython , nous cumulons trois avantages :

- La vaste communauté des programmeurs Python gagne l'opportunité de s'aventurer dans le développement de systèmes embarqués,
- Les développeurs de systèmes embarqués qui travaillent habituellement en langage C ont la possibilité de découvrir Python et la richesse de ses librairies,

- Les débutants en programmation ont rapidement accès à des fonctions très puissantes, qui permettent de développer rapidement et facilement des projets complets, mêlant contrôle, visuel, sonore, pilotage d'actionneurs, communication réseau, ...

1.7.6 Quelques cartes qui supportent le langage MicroPython

Cette liste est amenée à s'agrandir au fil des développements. Voir figure 2 page 8.

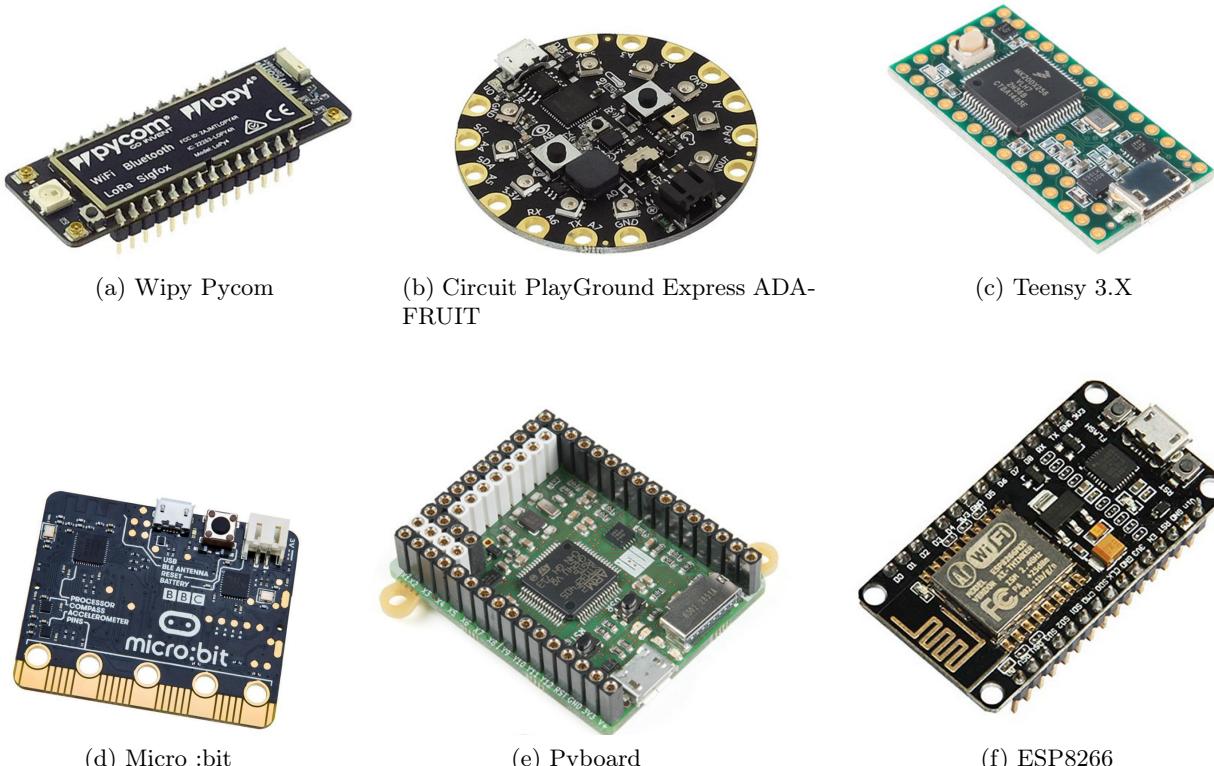


FIGURE 2 – cartes pour MicroPython

Liste des cartes qui supportent le langage MicroPython.

1.8 Pourquoi vouloir utiliser des appareils si petits et si peu puissants ?

1.8.1 La notion de puissance est très relative

Certes les micro-contrôleurs sont peu puissants, mais seulement par rapport à un ordinateur de bureau, alors que ce dernier est prévu pour du calcul intensif bien souvent, pour faire tourner des jeux ultra rapides, ou de lourds logiciels de CAO, simulation numérique, photo, ou vidéo, ...

Mais de quelle puissance de calcul a-t-on besoin pour piloter en vitesse par exemple un moteur ? Infiniment moins en réalité. Et pour allumer une lumière pendant 2 minutes suite à la présence d'une personne devant un capteur, ou pour faire défiler une barrière lumineuse ? Encore moins !

►Q.2: Est-il plus facile en terme de ressource processeur (la puissance de calcul) de piloter l'injection d'essence dans un moteur à combustion interne qui tourne à 6000 tours par minute (soit 100 tours par seconde), ou d'afficher une vidéo youtube ? Réponse : la puissance de calcul nécessaire est bien plus faible dans le cas de l'injection.

1.8.2 Une intelligence embarquée pour une air de magie

Et d'ailleurs, utiliser un micro-contrôleur permet de faire des choses extraordinaires, par le simple fait de pouvoir embarquer un système de contrôle dans n'importe quel objet de la vie courante.

Le système de commande a longtemps été une carte électronique, mais aujourd'hui cette carte ressemble à un ordinateur, car elle embarque un processeur, de la mémoire, du wifi, et aujourd'hui cette carte est souvent un micro-contrôleurs.

Ces systèmes de commande permettent des interactions avec le monde extérieur, ils permettent de réguler, automatiser, créer des interactions avec le monde extérieur, se connecter sur les différents réseaux, et en quelque sorte enchanter le monde.

Bien souvent, toutes ces opérations se font sans intervention humaine, les systèmes étant même parfois alimentés par des panneaux solaires pour une autonomie totale. En tout cas, pour le novice, cet aspect de la commande ressemble à de la magie.

1.9 L'ESP32, un micro-contrôleur très performant

L'ESP32 est un micro-contrôleur cadencé à 240 MHz et exécutant un code écrit en langage MicroPython hébergé dans sa mémoire Flash. Voir figure 3 page 9.

L'interpréteur (le logiciel) MicroPython est implanté dans cette même mémoire Flash ; il pèse environ 1,4 Mo, soit moins de la moitié de la capacité totale (4 Mo).

Ce microcontrôleur dispose d'interfaces WiFi et Bluetooth idéales pour les objets connectés. Des connecteurs latéraux mâles permettent d'enficher le module sur une plaque de montage rapide (connexions rapides) (ou «breadboard»). L'interface sans fil WiFi permet la création de point d'accès sans fil, l'hébergement d'un serveur, la connexion à internet et le partage des données par exemple.

Le module se programme directement à partir de l'IDE Thonny par exemple, et nécessite simplement un cordon microUSB. Voici ces caractéristiques :

- Alimentation : 5 Vcc via micro-USB
- Alimentation : 3,3 Vcc via broches Vin
- Microprocesseur : Tensilica LX6 Dual-Core
- Architecture : 32 bits
- Fréquence d'horloge : 240 MHz
- Mémoire SRAM : 512 ko (pour stocker les variables)
- Mémoire Flash : 4 Mo (pour stocker nos programmes)
- 10 E/S digitales compatibles PWM (Modulation de largeur d'impulsions)
- Interfaces : Entrées capacitatives, DAC (digital Analog Converter), ADC (Analog Digital Converter), I2C (Inter Integrated Circuit), SPI (Serial Peripheral Interface), UART (Universal Asynchronous Receiver/Transmitter),
- Interface WiFi 802.11 b/g/n 2,4 GHz (jusqu'à 150 Mbits/s)
- Antenne WiFi intégrée

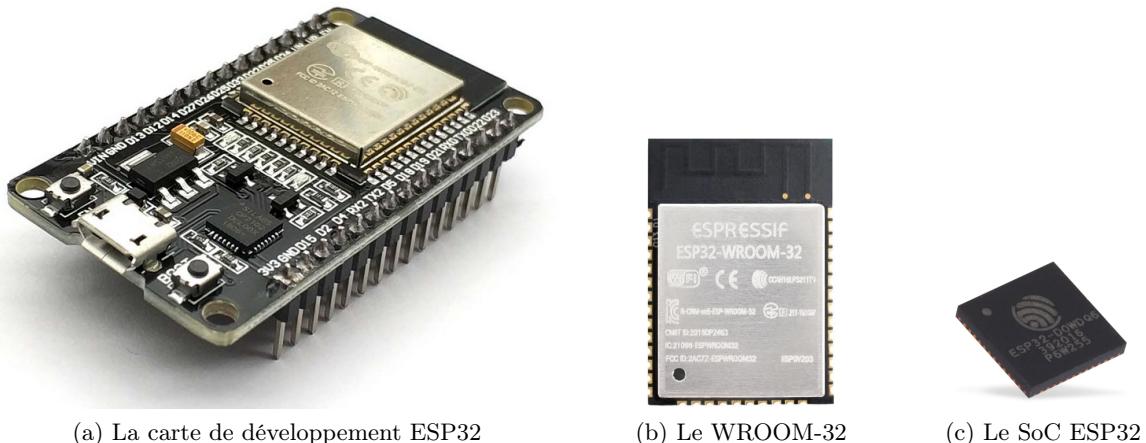


FIGURE 3 – ESP32

Pin OUT de la carte ESP32. Voir figure 4 page 10.

Sur ce diagramme des broches (Pin-OUT), toutes les broches peuvent être des sorties ou des entrées numériques (sauf 34,35,36,39 qui ne peuvent être que des entrées) ; mais certaines broches peuvent aussi avoir d'autres fonctions. Par exemple, la plupart des broches peuvent être des entrées analogiques. Elles sont notées ADC, et sont au nombre de 16

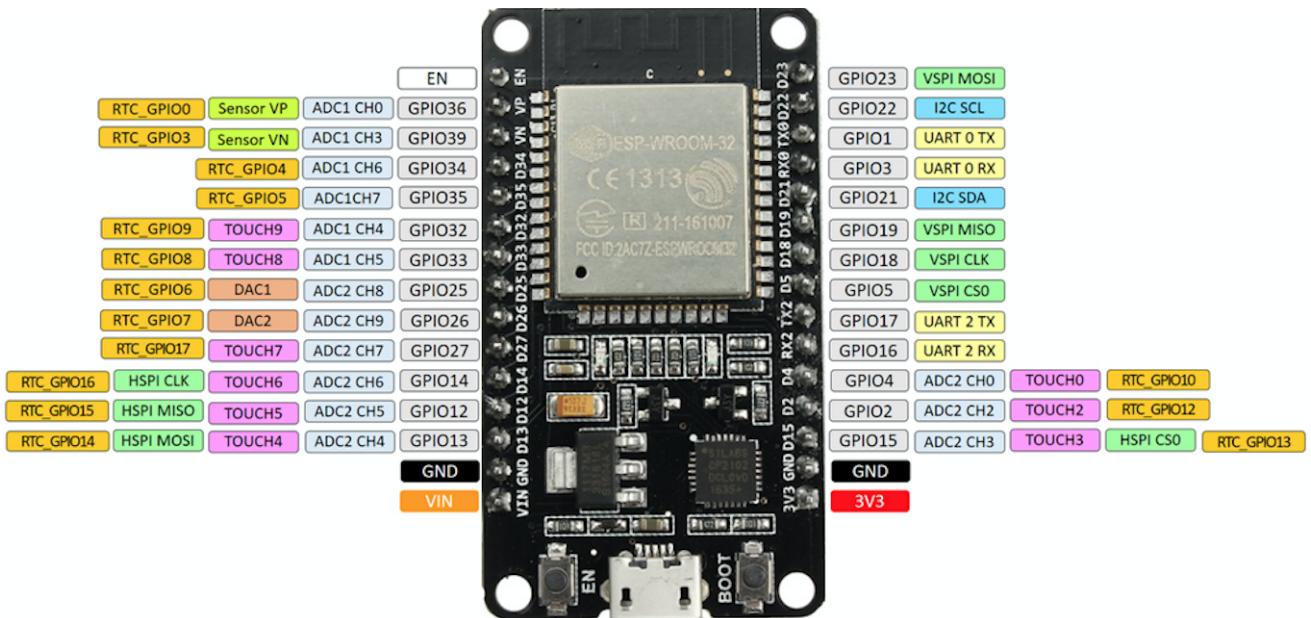


FIGURE 4 – PIN OUT de l'ESP32 DEVKIT V1 - 30 broches

2 Installations diverses

Ces installations ne seront à réaliser qu'une seule fois, lors de la première utilisation, et il sera nécessaire d'installer des drivers, utilitaires, etc ... Pour les séances de TP de l'ENIM, les installations sont déjà effectuées, vous pouvez aller à la section 2.1.4 page 11.

2.1 Installation de l'IDE Thonny

2.1.1 Définition de IDE ou «Integrated Development Environment»

On appelle IDE un environnement de développement intégré. C'est un ensemble d'outils informatique, conçu pour les programmeurs, destiné à assister et formaliser le travail de création de logiciels.

Un IDE comporte un «éditeur de texte» destiné à la programmation, ainsi que des fonctions qui permettent de démarrer le compilateur ou le «débogueur» qui permet d'exécuter ligne par ligne le programme en cours de construction.

Tous les outils de l'IDE sont prévus pour être utilisés ensemble. Cela permet d'augmenter la productivité des programmeurs en automatisant une partie des activités et en simplifiant les opérations.

Certains environnements sont dédiés à un langage de programmation en particulier ; c'est le cas de Thonny IDE, qui est à mon avis le mieux adapté pour une initiation à la programmation de l'ESP32.

2.1.2 L'IDE Thonny

Thonny est un IDE minimalisté qui permet d'apprendre le Python ; il a été mis au point par l'Université de Tartu en Estonie. Conçu pour les débutants, cet outil intègre son propre interpréteur Python 3.7 (mais vous pouvez aussi utiliser le vôtre), et offre des fonctionnalités plutôt pratiques quand on est dans un processus d'apprentissage. Il est intuitif, simple, en Open Source, et assez performant pour une prise en main. De plus, il est compatible avec Windows, MacOS, et Linux.

Nous utiliserons Thonny IDE, qui est normalement inclus sur notre distribution RaspBian du Raspberry. Pour MacOS et windows, il faudra l'installer. Lorsque nous installons Thonny IDE, il vient avec la dernière version de Python, soit la version 3.7 à ce jour.

L'avantage d'utiliser Thonny IDE est que non seulement il permet d'accéder à son propre interpréteur Python sur l'ordinateur, mais il peut aussi accéder à un interpréteur déporté dans une carte externe, par exemple celui qui sera téléversé dans notre contrôleur ESP32. Ainsi, nous pourrons programmer en MicroPython notre carte ESP32.

2.1.3 Installation de l'IDE Thonny

Attention : Cette installation n'est à réaliser qu'une seule fois. Pour les séances de TP de l'ENIM, l'installation est déjà effectuée, vous pouvez sauter à la section 2.1.4.

Sur Raspberry et Raspbian : Normalement, pour Linux RaspBian, Python 3 et Thonny IDE sont installés ; sinon voici la procédure d'installation :

- Ouvrir le shell de commande de RaspBian, et suivre la procédure :
- Pour installer Python, saisir : `sudo apt install python3 python3-pip python3-tk`
- Pour installer Thonny IDE, saisir : `sudo apt install python3-thonny`
- Puis lancer Thonny, saisir : `thonny`

Sur macOS et Windows : L'installation se fait de manière classique, comme tout logiciel :
[Lien vers le site officiel pour installer Thonny IDE.](#)

2.1.4 Test de l'IDE Thonny

Ouvrir l'IDE Thonny.

Thonny propose deux fenêtres, voir figure 6 :

- la console (ou le «shell»), en bas, pour saisir des instructions qui seront exécutées immédiatement, à chaque instruction saisie,
- la fenêtre d'édition de code, en haut, pour saisir un script (un ensemble d'instructions) qui sera exécuté en une fois lors de la demande d'exécution.

►Q.3: Afficher la fenêtre des variables en allant dans l'onglet «View», et activer l'onglet «Variables».

Nous allons maintenant demander à Thonny d'utiliser Python 3.7 de l'ordinateur pour exécuter les instructions suivantes. Pour cela, allons dans le menu : «Tools», «Options», «Interpréteur», et choisir : «Le même interpréteur qui exécute Thonny (par défaut)». Voir figure 5 page 11.

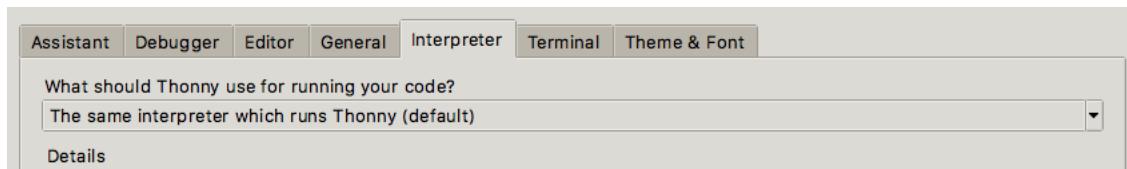


FIGURE 5 – Choix de l'interpréteur Python

►Q.4: Dans la console, saisir les trois lignes successivement, comme fait sur la figure 6, et remarquer l'apparition des variables, puis l'affichage du résultat.

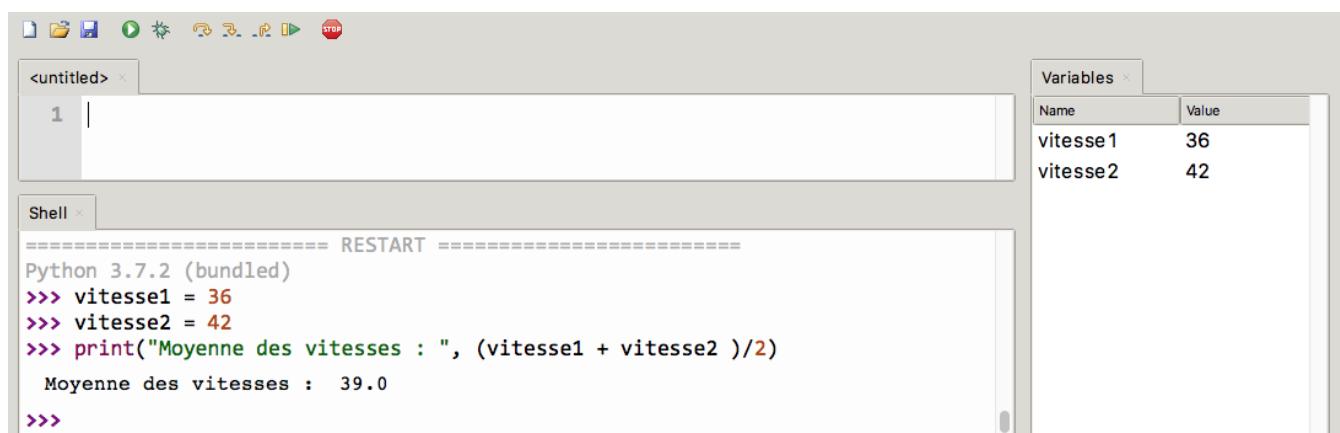


FIGURE 6 – L'IDE Thonny

►Q.5: Toujours dans la console (le shell), presser les flèches «flèche du haut» et «flèche du bas» pour vérifier que toutes les instructions saisies sont gardées en mémoire. S'en servir pour affecter maintenant la valeur 45 à ma variable `vitesse2`, puis afficher à nouveau la moyenne des vitesses.

2.1.5 Connection entre Thonny IDE et l'ESP32

Dans un second temps, ce sera la carte ESP32 qui exécutera ces instructions. Pour une première utilisation, notre carte ESP32 n'est pas encore «flashée» avec le firmware, c'est à dire avec le programme qui va interpréter le code MicroPython. Pour une première installation, il nous faut encore passer par les étapes suivantes :

- télécharger le firmware MicroPython, qui est l'interpréteur MicroPython implanté dans l'ESP32, et qui interprète une à une les instructions saisies dans la console, ou les lignes de codes du script,
- installer esptool.py à partir de Thonny,
- installer un driver si besoin pour dialoguer avec l'ESP32 (pour macOS et Windows),
- connecter la carte ESP32 à l'ordinateur,
- téléverser ce firmware dans la carte ESP32. On parle de «flashe» la carte. Il existe plusieurs outils pour faire cela. Les deux plus simples sont l'installation de l'utilitaire «esptool» pour Raspberry, et Thonny IDE pour macOS et Windows.
- indiquer à Thonny IDE que c'est le firmware qui doit exécuter les instructions.

Attention : Cette installation n'est à réaliser qu'une seule fois, lors de la première utilisation. Pour les élèves de l'ENIM, les cartes sont prêtes à l'emploi car le firmware est installé. Vous pouvez donc sauter à la section 2.2.4. Il sera nécessaire de faire toute cette procédure lorsque vous utiliserez une carte ESP neuve pour un projet personnel ou autre.

2.2 Préparer le téléchargement du firmware dans l'ESP32 (Flasher le ESP32)

2.2.1 Télécharger le firmware MicroPython

Téléchargement de la dernière version du FirmWare MicroPython pour ESP32. Voir figure 7 page 12. Par exemple, j'ai téléchargé la version GENERIC-SPIRAM, c'est à dire qui fonctionne avec une mémoire pSRAM externe de 4 MO, celle qui correspond à notre ESP32 :

`esp32spiram-idf3-20191105-v1.11-555-g80df377e9.bin`

Firmware for ESP32 boards

The following files are daily firmware for ESP32-based boards, with separate firmware for boards with and without external SPIRAM. Non-SPIRAM firmware will work on any board, whereas SPIRAM enabled firmware will only work on boards with 4MiB of external pSRAM.

Program your board using the esptool.py program, found [here](#). If you are putting MicroPython on your board for the first time then you should first erase the entire flash using:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash
```

From then on program the firmware starting at address 0x1000:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20190125-v1.10.bin
```

Firmware built with ESP-IDF v3.x, with support for LAN and PPP but no bluetooth:

- GENERIC : [esp32-idf3-20191105-v1.11-555-g80df377e9.bin](#)
- GENERIC : [esp32-idf3-20190529-v1.11.bin](#)
- GENERIC : [esp32-idf3-20190125-v1.10.bin](#)
- GENERIC : [esp32-idf3-20180511-v1.9.4.bin](#)
- GENERIC-SPIRAM : [esp32spiram-idf3-20191105-v1.11-555-g80df377e9.bin](#)
- GENERIC-SPIRAM : [esp32spiram-idf3-20190529-v1.11.bin](#)

FIGURE 7 – Site de téléchargement du firmware pour l'ESP32

2.2.2 Installation de l'utilitaire «esptool.py»

Cet utilitaire est fourni par «Espressif», la société qui produit l'ESP32.

Esptool est codé en Python, il est open-source, indépendant de la plateforme, et permet de communiquer avec l'ESP32, pour lui téléverser l'interpréteur MicroPython que nous venons de télécharger.

Sur Raspbian Il est nécessaire d'avoir Python 2.7 ou 3.4 minimum installé sur votre ordinateur, ce qui est normalement déjà fait lorsque nous avons installé Thonny IDE.

L'instruction pour installer esptool est :

```
$ pip install esptool
```

En cas de difficulté d'installation, essayer l'une ou l'autre des solutions :

```
$ python -m pip install esptool
```

```
$ pip2 install esptool
```

Sur MacOS ou windows Aller dans l'onglet de Thonny «Outils», puis «Gérer les Plugins», saisir dans «Rechercher un paquet sur Pypi» : esptool, et l'installer. Il est nécessaire de relancer Thonny IDE après cette installation.

2.2.3 Installation du driver CP2102

Ce driver permet de communiquer avec le circuit CP2102 de notre carte ESP32. Ce circuit adapte la liaison USB de l'ordinateur avec l'interface **UART** (Universal Asynchronous Receiver Transmitter, c'est un émetteur-récepteur asynchrone universel) de l'ESP32. Ce driver n'est pas installé par défaut sur MacOS ou windows, mais il l'est souvent sur les distributions Linux (Raspbian par exemple). Vous pouvez trouver ce driver avec Google en saisissant CP2102 driver download, ou ici :

[Téléchargement du driver CP2102.](#)

2.2.4 Connecter la carte au port UBS de l'ordinateur

Connecter à l'aide du câble micro-USB. Cette connexion alimente aussi la carte en 5V.

Aller dans l'onglet «Outils», «Options», puis «Interpréteur», et choisir «MicroPython (ESP32)». Ensuite, choisir le port sur lequel l'ESP32 est connecté à votre ordinateur. Voir figure 8.

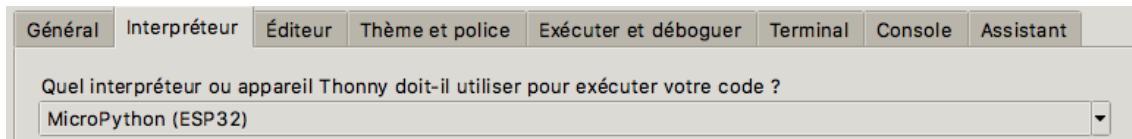


FIGURE 8 – Site de téléchargement du firmware pour l'ESP32

Si notre carte n'est pas encore chargée avec l'interpréteur MicroPython, ce qui est le cas pour une première utilisation, normalement Thonny répond :

Device is busy or does not respond. Your options :

- wait until it completes current work,
- use Ctrl+C to interrupt current work,
- use Stop/Restart to interrupt more and enter REPL.

... ce qui signifie que Thonny ne peut établir de communication avec la carte. Nous allons donc maintenant téléverser MicroPython dans la carte, ce que l'on appelle aussi «flasher» la carte. Ainsi, l'ordinateur et la carte pourront établir une communication ...

Pour les cartes utilisées en TP à l'ENIM, elles sont «flashées» avec un MicroPython récent. Pas besoin d'en installer un nouveau. Vous pouvez aller à la section 2.2.6.

2.2.5 Flasher notre carte ESP32

Procédure sur MacOS et windows : Dans la même fenêtre que précédemment, cliquer sous «Micrlogiciel» sur «Ouvrir la boîte de dialogue pour installer ou mettre à jour MicroPython sur votre appareil». Voir figure 9.

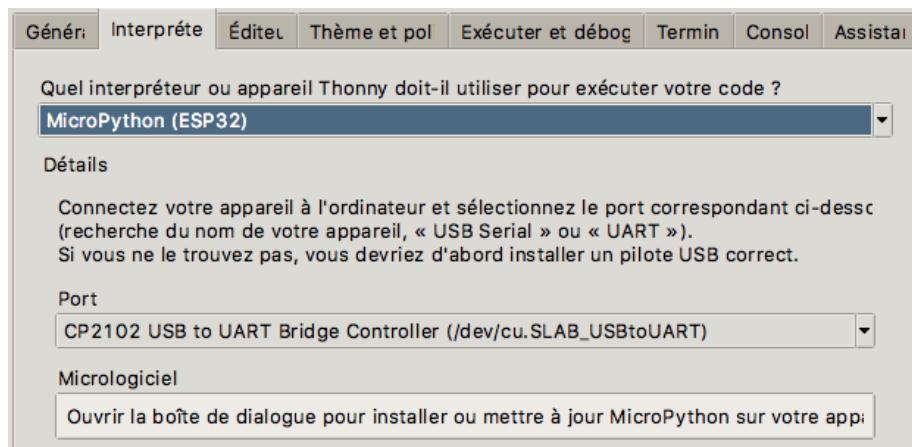


FIGURE 9 – Préparation au téléversement du firmware dans l'ESP32

Ensuite, choisir le port de communication, parcourir les fichiers pour choisir votre firmware qui a l'extension «.bin». Et laisser le temps (1 minute environ) à Thonny pour effacer la carte, puis téléverser le firmware. Voir figure 10.



FIGURE 10 – Téléversement du firmware dans l'ESP32

Procédure sur MacOs et windows : Il nous faut effacer la carte, puis téléverser le nouveau FirmWare. Saisir ces instructions dans la console (ou le Shell) :

- esptool.py --chip esp32 --port /dev/cu.SLAB_USBtoUART erase_flash
- esptool.py --chip esp32 --port /dev/cu.SLAB_USBtoUART --baud 460800
- write_flash -z 0x1000 esp32-20191011-v1.11-422-g98c2eabaf.bin

Le port sur lequel est connectée votre carte est à modifier en fonction de votre ordinateur *. Le fichier à téléverser est à modifier aussi ; il doit être placé dans le répertoire de base de votre ordinateur.

* : Normalement, il n'est pas nécessaire d'indiquer le port sur lequel votre carte est connectée, car «esptool.py» va énumérer tous les ports série connectés, et va essayer chacun d'eux pour trouver un composant «Espressif» connecté. Ainsi, on écrira simplement :

- esptool.py --chip esp32 erase_flash
- esptool.py --chip esp32 --baud 460800
- write_flash -z 0x1000 esp32-20191011-v1.11-422-g98c2eabaf.bin

2.2.6 Fin de l'installation

A partir de maintenant, Thonny est paramétré pour envoyer les instructions de l'utilisateur vers MicroPython, qui exécutera les instructions. Nous allons pouvoir découvrir le monde merveilleux de Python embarqué sur une carte à microcontrôleur.

Parfois il est utile de cliquer sur le bouton STOP de Thonny, pour qu'il se connecte avec la carte. Une fois la carte connectée, voici le message que vous obtenez en figure 11, avec la version du firmware embarqué :

```
MicroPython v1.11-549-gf2ecfe8b8 on 2019-11-01; ESP32 module with ESP32
Type "help()" for more information.
>>>
```

FIGURE 11 – Connexion valide avec la carte via MicroPython

Première partie

Les bases

3 Introduction à Thonny et Python

3.1 Prise en main de l'IDE Thonny et introduction à Python

3.1.1 Utiliser Thonny comme un interpréteur

Thonny possède un espace dans lequel il est possible de saisir directement les commandes. Cette zone est appelée «console». Saisies dans la console, les instructions sont interprétées directement. Comme nous avons dit que l'interpréteur est MicroPython dans l'ESP32, chaque instruction provoque un accès à l'ESP32. A partir de cette console, nous allons découvrir les bases de Python ...

3.1.2 Les opérateurs mathématiques

►Q.6: Saisir dans la console les commandes suivantes.

A chaque instruction saisie, il y a un transfert d'information. En effet, c'est MicroPython (implémenté sur l'ESP32) qui reçoit l'instruction à interpréter (transfert depuis l'ordinateur vers la carte), et qui renvoie le résultat de l'opération (un deuxième transfert, mais cette fois depuis la carte vers l'ordinateur).

```
>>> 2 + 3 * 4      # remarquez que la multiplication est prioritaire
14
>>> 15 / 3 - 1    # la division retourne un type "float"
4.0
>>> 13 // 4       # la division entière retourne un type "int"
3
>>> 13 % 2        # le modulo retourne le reste de la division entière
1
```

3.1.3 Les «labels» ou «étiquettes» - Attacher des noms aux nombres

Pour concevoir des programmes qui vont au delà de la simple opération arithmétique, nous devons affecter des noms aux nombres. Par exemple :

```
>>> a = 4          # affectation du label a au nombre 4
>>> a + 1
5
```

Nous avons donné le nom **a** au chiffre 4. Ainsi, **a** renvoie au nombre 4, ce qui est pratique car il suffit de changer le nombre vers lequel **a** pointe et Python se sert de cette nouvelle valeur dans la suite du code. Ce type de nom est appelé «label», ou «étiquette». On parle alors d'assigner (ou affecter) le label **a** au nombre 4.

Remarque concernant le terme «variable» : On entend souvent le terme «variable» qui décrit la même chose que notre «label». Toutefois, étant donné que le terme «variable» est aussi un terme mathématique utilisé au *x* dans l'équation par exemple $x - 1 = 1/x$, nous l'utiliserons uniquement dans le cas d'équations ou expressions mathématiques.

3.1.4 Types des données

Il existe 4 types de données :

►Q.7: Saisir dans la console les commandes suivantes ; pour chaque donnée, on afficher le type de l'objet pour comprendre.

Type de donnée	Description
int	Entier (positif ou négatif)
float	Flottant : nombre à virgule
str	String : chaîne de caractère
bool	Booléen : Vrai ou Faux

```

>>> x = 6          # assignation du label x au nombre 6
>>> type (x)      # retourne le type de la donnée
<class 'int'>    # retourne le résultat : classe entier
>>> y = x / 2     # retourne un float, même si x et 2 sont entiers
>>> y              # demande à afficher la valeur de y
3.0                # valeur affichée, et le point indique un type "float"
>>> type (y)
<class 'float'>
>>> message = " Bonjour à tous ! "   # identique à ' Bonjour à tous ! '
>>> type (message)
<class 'str'>
>>> e = True
>>> type (e)
<class 'bool'>

```

3.1.5 Changer les types de données

Nous pouvons changer les types de données, par exemple un type flottant en type entier, ou type chaîne de caractères. float -> int int -> float int -> str int -> bool (implicite)

```

>>> i = int(y)          # transformation en entier
>>> i
3
>>> type(i)
<class 'int'>
>>> f = float(2*i)     # transformation en flottant
>>> f
6.0
>>> texte = str(i)      # transformation en chaîne de caractère (string)
>>> texte
'3'
>>> texte * 5
'33333'               # la chaîne est dupliquée

```

3.1.6 La fonction print

Elle affiche dans la console le message entre parenthèses. Cela sera pratique pour le débogage, durant l'exécution d'un script, que nous verrons plus loin.

```

>>> print("J'aime le soleil !")
J'aime le soleil !

```

3.1.7 Les opérateurs de comparaison

Opérateurs	Description
==	Egal à
!=	Non égal à
>	Supérieur à
<	Inférieur
>=	Supérieur ou égal à
<=	Inférieur ou égal à

►Q.8: Saisir dans la console les commandes suivantes ; bien comprendre que l'opérateur == teste l'égalité, ce qui n'est pas équivalent à l'opération d'affectation qui est =.

```
>>> 2 == 3
False
>>> x == 6      # test si x = 6 ! on n'affecte pas le label x au nombre 6
True
>>> 3 > 2
True
```

3.2 Utilisation de la zone d'édition des scripts

Jusqu'à maintenant, nous avons saisi des instructions dans la console, et elles étaient exécutées instantanément. Maintenant, nous allons saisir nos instructions dans la zone d'édition des scripts, qui permet d'exécuter plusieurs ligne de code à la suite, et de pouvoir faire des structures de boucle et de condition.

3.2.1 Une boucle simple avec la structure While

La structure `While` permet d'exécuter en boucle des instructions, avec une condition d'arrêt. Le code suivant affiche les nombres entre 1 à 4, c'est à dire que 4 boucles sont exécutées.

```
1  nombre = 1
2  while nombre < 5:
3      print(nombre)
4      nombre = nombre + 1
```

On obtient cela :

```
1
2
3
4
```

Bien remarquer l'indentation (retrait de 2 ou 4 caractères «<espace», ou TAB) pour les 2 lignes contenues dans le `While` (les lignes 3 et 4). C'est ce qui permet à Python de comprendre que ce sont ces deux lignes qui seront exécutées en boucle, tant que le label `nombre` est inférieur à 5.

►Q.9: Exécuter ce code, en cliquant sur la flèche verte (le raccourci est F5). Remarquer que pour une première exécution, Thonny demande à enregistrer ce code. Nous l'enregistrons sur l'ordinateur (pas sur la carte ESP32), et dans votre espace personnel, sous le nom : «Test1».

3.2.2 Une boucle simple avec la structure for

La boucle `for` permet de répéter une boucle un certain nombre de fois, pour par exemple parcourir les éléments d'une liste. Par exemple, la boucle ci dessous est exécutée pour toutes les valeurs de `nombre` comprise entre 1 et 4. La fonction `range()` crée une sorte de liste d'entiers, du premier (inclus) au dernier (exclus!). On obtient alors la même chose que précédemment.

```
for nombre in range(1, 5):
    print(nombre)
```

Attention avant de saisir ce code! : afin de ne pas créer un nouveau fichier à chaque exemple, nous allons mettre en commentaire les 4 premières lignes avec le symbole `#`, sauter une ligne, et saisir les suivantes. Voir ci-dessous. Cela permet d'avoir accès très rapidement à toutes les lignes déjà saisies pour pouvoir récupérer des bouts de code. Nous ferons de même pour tous les exemples qui suivent.

```
1  # nombre = 1
2  # while nombre < 5:
3  #     print(nombre)
4  #     nombre = nombre + 1
5
6  for nombre in range(1, 5):
7      print(nombre)
```

►Q.10: Saisir ce code.

Le fait que le deuxième nombre (5) soit exclus dans l'instruction `range(1, 5)` est assez logique : si nous ne mettons qu'une valeur, par exemple 5, la fonction `range(5)` affiche 5 valeurs, et comme la première est toujours 0, elle finit bien à 4.

```
for nombre in range(5):
    print(nombre)
```

►Q.11: Tester cela.

3.3 Utilisation des listes

3.3.1 Creation d'une liste

Une liste est une structure iterative, definie comme la ligne 1 ci dessous par exemple. Elle est iterative car on peut parcourir ses elements un a un a l'aide d'une boucle `for`.

La structure «liste» est trs souvent utilises dans la programmation Python ; nous allons donc y passer quelques instants pour bien la comprendre.

Par defaut, nous avons un retour a la ligne apres chaque `print()` ; le paramtre `end=" "` dans la fonction `print()` permet de changer le caractre de fin de l'affichage, et de remplacer donc le retour a la ligne par un espace. Cela permet d'afficher tous les elements de la liste sur la meme ligne. Dans l'exemple suivant, on affiche *un par un* les elements de la liste qui contient tous les carres des nombres entre 1 et 10.

```
1 ma_liste = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
2 for element in ma_liste :
3     print(element, end=" ")
```

Pour obtenir cela :

```
1 4 9 16 25 36 49 64 81 100
```

►Q.12: Tester le bon fonctionnement.

Pour verifier que les elements sont affiches un par un, on peut mettre un delai (0.5 s par exemple) entre deux iterations pour voir defiler les 10 valeurs. Pour cela, nous avons ajoute la methode `sleep` (nous verrons plus tard ce que cela veut dire) du module `time`, qui permet d'attendre un temps indique en secondes.

```
1 from time import sleep
2 ma_liste = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
3 for element in ma_liste :
4     print(element, end=" ")
5     sleep(0.5)
```

Si nous voulions juste connaître la valeur de `ma-liste`, on aurait pu ecrire (dans la console par exemple, ou a la place des lignes 3 a 5) directement et plus simplement, en une seule operation d'affichage, l'instruction suivante :

►Q.13: Saisir dans la console l'instruction suivante :

```
1 >>> print(ma_liste)
2 [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Dans ce cas, `ma-liste` est affichee entre crochets (comme une liste). Attention, on ne realise pas la meme operation. Prec emment, on prenait un a un chaque element de la liste pour l'afficher, ou en faire un traitement.

►Q.14: Verifier que notre variable figure bien dans la fenetre des variables.

3.3.2 Creation d'une liste avec la methode «append»

La methode «append» s'applique a une liste, et permet d'ajouter un element a cette liste, a la fin, qui est ainsi modifiee. En utilisant cette methode pour creer et afficher la meme liste, nous pouvons iterer de 1 a 11, et ajouter a `ma_liste_calc` («calc» pour calculee) un element a chaque iteration. On commence par creer une liste vide en ligne 1.

L'instruction `**` correspond a l'operation mathematique puissance.

►Q.15: Tester le bon fonctionnement, c'est a dire afficher tous les elements de `ma_liste_calc`. Il faudra ajouter une ligne ou deux a ce code.

```
1 ma_liste_calc = []
2 for x in range(1, 11):
3     ma_liste_calc.append(x**2)
```

3.4 Comment tester si un nombre est pair ou multiple de x ?

Il est très fréquent en programmation d'avoir à tester si un nombre est multiple d'une valeur x . Nous allons nous en servir très souvent dans cet ouvrage.

Principe de la division et de son reste. Pour qu'un nombre soit pair, le principe est de le diviser par 2, et de tester si le reste de la division vaut 0 ; s'il vaut 0, le nombre est pair. Exemple :

— $13/2 = 6,5$; la valeur entière de la division vaut 6 et le reste vaut 1 car $13 = 6 * 2 + 1$. Donc le nombre 13 n'est pas pair.

— $12/2 = 6$; la valeur entière de la division vaut 6 et le reste vaut 0 car $12 = 6 * 2 + 0$. Donc le nombre 12 est pair.

Le principe est le même pour savoir s'il est multiple de 5 : on divise par 5 au lieu de 2. Exemple :

— $13/5 = 2,6$; la valeur entière de la division vaut 2 et le reste vaut 3 car $13 = 2 * 5 + 3$. Donc le nombre n'est pas multiple de 5.

— $15/5 = 3$; la valeur entière vaut 3 et le reste vaut 0 car $15 = 3 * 5 + 0$. Donc le nombre est multiple de 5.

Utilisation de l'instruction «modulo» %. Pour faire le calcul du reste de la division, il est courant d'utiliser une opération plus adaptée, qui est l'instruction «modulo», qui s'écrit %, et qui retourne directement le reste de la division, car le calcul de la partie entière ne nous avance pas pour tester si un nombre est divisible par un autre. Exemple, on teste si n est multiple de 5, pour les valeurs $n = 13$ et $n = 15$:

```
1 n = 13
2 print(n, "% 5 = ", n % 5)
3 print(n,"multiple de 5 ? : ", (n % 5) == 0)
4 n = 15
5 print(n, "% 5 = ", n % 5)
6 print(n,"multiple de 5 ? : ", (n % 5) == 0)
```

Ce qui nous donne :

```
13 % 5 = 3
13 multiple de 5 ? : False
15 % 5 = 0
15 multiple de 5 ? : True
```

►Q.16: Tester le bon fonctionnement.

►Q.17: Ajouter le code pour tester si 27 est multiple de 5.

3.5 Procédures et fonctions

3.5.1 Crédration de la procédure «multiple»

Une procédure est une série d'instructions :

- qui porte un nom,
- qui effectue un travail dont le résultat dépend de variables appelées arguments,
- et qui ne renvoie pas de résultat.

Pour définir une procédure, on utilise la commande `def`.

Pour simplifier le code, et ne pas réécrire les lignes 2 et 3 de l'exemple précédent, nous allons créer une procédure qui prend en argument n , et qui affiche les deux même lignes que précédemment.

```
1 def multiple(n):
2     print(n, "% 5 = ", n % 5)
3     print(n,"multiple de 5 ? : ", (n % 5) == 0)
4
5 multiple(7)
6 multiple(10)
```

►Q.18: Ajouter le code pour tester si 27 est multiple de 5.

Variante 1 : On aurait pu aussi, au lieu d'afficher les deux lignes avec `print`, retourner seulement l'information (True ou False) sur le nombre, à savoir s'il est multiple de 5 ou pas. Dans faire cela, nous pouvons créer une fonction. Voyons cela ...

3.5.2 Cr ation de la fonction «multiple»

Une fonction est une s rie d'instruction :

- qui porte un nom,
- qui effectue un travail dont le r sultat d pend de variables appel es arguments,
- et qui renvoie une valeur que l'on peut stocker, si on le souhaite, dans une variable.

Pour d finir une fonction, on utilise la commande `def`, et on renvoie le r sultat (une valeur) gr ce   la commande `return`. Par exemple :

```
def multiple(n):  
    return (n % 5) == 0 # Retourne VRAI si n est multiple de 5  
  
print("Le chiffre 10 est multiple de 5 ? : ", multiple(10))
```

Variante 2 : On aurait pu aussi, au lieu de tester par rapport   5, tester par rapport   n'importe quel nombre, x par exemple, comme cela :

```
def multiple(n, x):  
    return (n % x) == 0 # Retourne VRAI si n est multiple de x  
  
print("Le chiffre 10 est multiple de 4 ? : ", multiple(10, 4))
```

Variante 3 : On va maintenant demander   l'utilisateur d'indiquer deux nombres entiers, n et x . La fonction `input` retourne une cha ne de caract re, que l'on transforme en nombre entier. On indiquera le r sultat sous la forme :

- « Le nombre 12 n'est pas multiple de 5, car la division de 12 par 5 donne 2, et le reste vaut 2.», ou
- « Le nombre 12 est multiple de 4, car la division de 12 par 4 donne 3, et le reste est nul.».

Utilisation des liens web. Je rappelle que, tout au long de ce support de TP, de nombreux liens sont disponibles , ils sont mis en  vidence par une couleur bleue. Ils amnent vers des pages   lire pour de plus d'informations, et c'est ainsi que vous avez t l charg  le support PDF de cet ouvrage sur le lien vers mon d p t Github. Mais il est possible d'avoir et t l charger les fichiers texte de tous les codes que nous allons  tudier   partir de maintenant.

►Q.19: Cliquer sur le lien «test-multiple.py» du code ci-dessous. Vous arrivez alors sur mon d p t Github, et il ne reste plus qu'  cliquer sur RAW pour avoir la version brute du code (cela permet d'obtenir le texte «brut» sans mise forme), puis CTRL+A pour tout s lectionner tout le texte, puis CTRL+C pour le copier ; et enfin CTRL+V dans l' diteur de script Python pour le coller, puis l'ex cuter.

1 – test-multiple.py - Test si n est multiple de x

```
1 def multiple(n, x):  
2     return (n % x) == 0  
3  
4 nombre = int(input("Quel est le nombre n a tester ? : "))  
5 diviseur = int(input("Quel est le diviseur a tester ? : "))  
6  
7 if multiple(nombre, diviseur) :  
8     print("Le nombre {0} est multiple de {1}, car la division de {0} par {1}  
      donne {2}, et le reste est nul.".format(nombre, diviseur, int(nombre/  
      diviseur)))  
9 else :  
10    print("Le nombre {0} n'est pas multiple de {1} car la division de {0}  
      par {1} donne {2}, et le reste vaut {3}. ".format(nombre, diviseur, int(  
      nombre/diviseur), nombre%diviseur))
```

►Q.20: Tester ce programme avec les valeurs 12 et 5 par exemple, puis 12 et 4.

►Q.21: Remarquer l'indentation apr s la structure `if`.

►Q.22: Tester ce programme avec un nombre non entier, avec 12.2 et 4 par exemple. Constater que cela g n re une erreur, et comprendre que la conversion en entier ne peut se faire que si la cha ne de caract re contient uniquement un nombre entier.

```
ValueError: invalid syntax for integer with base 10
```

3.6 Gestion des erreurs

Pour prendre en compte le fait que l'utilisateur pourrait entrer une valeur non conforme, nous pouvons utiliser la structure `try ... except`. S'il n'y a pas d'erreur lors de l'exécution, la section `try` (essaie de faire ceci ...) se déroule normalement. En cas d'erreur, le programme saute à la section `except` (sinon faire cela ...).

2 – test-multiple-gestion-erreur.py - Test si n est multiple de x, et gestion erreur

```
1 def multiple(n, x):
2     return (n % x) == 0
3
4 try :
5     nombre = int(input("Quel est le nombre n à tester ? : "))
6     diviseur = int(input("Quel est le diviseur à tester ? : "))
7
8     if multiple(nombre, diviseur) :
9         print("Le nombre {} est multiple de {}.".format(nombre,diviseur))
10    else :
11        print("Le nombre {} n'est pas multiple de {}".format(nombre,diviseur))
12
13 except :
14     print("Veuillez saisir des valeurs entières !")
```

►Q.23: Tester ce code en saisissant la valeur 12.2 et 4, puis 12 et 4.

3.7 Revenons sur les listes

3.7.1 Crédation d'une liste, et utilisation de la structure condition : «if»

Nous voulons ajouter une condition : nous créons la liste des carrés de x , mais seulement pour les valeurs de x de 1 à 10 qui ne sont pas multiples de 5. Voici :

3 – liste-append.py - Crédation d'une liste par ajout d'éléments avec la méthode «append»

```
1 ma_liste = []
2 for x in range(1,11) :
3     if (x % 5) != 0 :          # si x n'est pas multiple de 5 :
4         ma_liste.append(x**2)  # ajout de x**2 à la liste en construction
5 for element in ma_liste :
6     print(element , end="-")
```

On obtient cela :

```
1-4-9-16-36-49-64-81-
```

►Q.24: Tester le bon fonctionnement.

Nous vérifions qu'il manque bien le 25 et le 100.

3.7.2 Crédation d'une liste «en compréhension»

La crédation d'une liste en compréhension permet de crée une liste en une seule instruction, même s'il y a une condition à respecter. Dans l'exemple ci-dessous, nous créons la liste des carrés de tous les éléments entre 1 et 10, sauf les éléments multiples de 5. Ainsi, au lieu d'écrire les lignes 1 à 4 de l'exemple précédent, nous écrivons :

```
ma_liste = [x*x for x in range(1, 11) if (x % 5) != 0]
```

►Q.25: Vérifier que cela fonctionne bien en affichant la liste.

►Q.26: Créer et afficher la liste des nombres qui sont multiples de 7 OU de 11, et ceci entre 1 et 50. Afin de mieux comprendre, votre liste contiendra entre autres les valeurs 21 (car $12 = 7 * 3$) et 22 (car $22 = 11 * 2$).

Dans l'exemple suivant, nous faisons l'intersection de deux listes, c'est à dire la liste des éléments communs :

```

1 liste1 = [1, 3, 5, 7, 9]
2 liste2 = [5, 6, 7, 8, 9]
3 liste_elements_comuns = [value for value in liste1 if value in liste2]
4 print(liste_elements_comuns)

```

►Q.27: Comprendre le programme.

3.7.3 La fonction «enumerate»

Cette fonction permet, à partir d'une liste ou toute autre structure sur laquelle il est possible d'itérer, de fournir une valeur qui s'incrémente à chaque élément de la liste, et de l'associer à cet l'élément. Nous pouvons alors, avec une écriture simplifiée, afficher les éléments de la liste et leur numéro grâce à une boucle `for` d'une syntaxe un peu particulière : `element` prend successivement pour valeur chaque élément de la liste, pendant que `compt` s'incrémente en partant de 0...

```

1 ma_liste = [x*x for x in range(1,11) if(x % 5) != 0]
2 for compt, element in enumerate(ma_liste) :
3     print (compt, element)

```

►Q.28: Vérifier le bon fonctionnement.

Réponse à la question de la section 3.7.2 :

```
ma_liste = [x for x in range(1, 50) if (x % 7) == 0 or (x % 11) == 0 ]
```

3.8 Exemple de synthèse : PGCD

Le programme suivant fait la synthèse de tout ce que nous venons de voir. Ce programme calcule tous les diviseurs de deux nombres, puis tous les diviseurs communs, puis le plus grand d'entre eux, appelé le PGCD (Plus Grand Commun Diviseur). Ainsi, il va fournir un résultat sous la forme (exemple avec 20 et 70) :

```

x = 20 ; y = 70
Liste des diviseurs de x : [1, 2, 4, 5, 10, 20]
Liste des diviseurs de y : [1, 2, 5, 7, 10, 14, 35, 70]
Liste des diviseurs communs : [1, 2, 5, 10]
PGCD : 10

```

Pour comprendre le code suivant, et en particulier pour savoir si un nombre x est divisible par un nombre e , il suffit de tester si x/e donne un nombre entier, ou encore, nous l'avons déjà vu, si $x \% e == 0$.

4 – pgcd.py - Exemple du calcul du PGCD de deux nombres

```

1 # PGCD : programme de calcul du PGCD de 2 entiers
2 # Auteur : Vincent HERMITANT - ENIM 2020
3 x = 20
4 y = 70
5 diviseurs_x=[]      # creation liste vide pour diviseurs de x
6 diviseurs_y=[]      # creation liste vide pour diviseurs de y
7
8 # creation d'une fonction pour trouver les elements communs a deux listes
9 def inter(liste1, liste2):
10    liste_elements_comuns = [value for value in liste1 if value in liste2]
11    return liste_elements_comuns
12
13 for a in range (1,max(x,y)+1):
14     if x % a == 0 :          # si x est divisible par a
15         diviseurs_x.append(a) # ajout a la liste des diviseurs de x
16     if y % a == 0 :          # si y est divisible par a
17         diviseurs_y.append(a) # ajout a la liste des diviseurs de y
18 diviseurs_comuns = inter(diviseurs_x, diviseurs_y)
19 plus_grand_commun_diviseur = max(diviseurs_comuns) # retourne le maxi
20
21 print("x = ", x, "; y = ", y)
22 print ("Liste des diviseurs de x :",diviseurs_x)
23 print("Liste des diviseurs de y :",diviseurs_y)

```

```

24 print("Liste des diviseurs communs : ", diviseurs_comuns)
25 print("PGCD : ", plus_grand_commun_diviseur)

```

►Q.29: Comprendre ce code.

Cet exemple (précédent) a eu pour but d'illustrer l'utilisation des listes. Cependant, le calcul du PGCD se réalise de manière plus directe, en 4 ou 5 lignes. Voir le code ci-dessous qui expose la manière de faire, ainsi que deux variantes. Ne pas essayer de comprendre ce code, il est juste mis là pour ceux qui sont plus avancés sur Python.

5 – pgcd2.py - Exemple du calcul du PGCD de deux nombres - version simplifiée

```

1 def pgcd(a,b):                      # Version 1 (de base)
2     while b != 0 :
3         r = a % b
4         print(a, b, r)
5         a,b = b,r
6     return a
7 x = 70 ; y = 20
8 print ("PGCD de", x, "et", y, ":", pgcd(70,20))
9
10 #def pgcd(a,b) :                  # Version 2 (simplifiee)
11 #    while b != 0 :
12 #        a , b = b , a % b
13 #    return a
14
15 #def pgcd(a,b) :                  # Version 3 (avec recursivite)
16 #    if b == 0 :
17 #        return a
18 #    else:
19 #        r = a % b
20 #        return pgcd(b,r)

```

4 Considérations sur la programmation orientée objet et les modules

4.1 Découverte de la programmation orientée objet

4.1.1 Introduction - Définitions

Python est un langage de programmation orienté objet (OOP : Objet-Oriented Programming). En python, tout est objet. Quand vous croyez utiliser un nombre, une variable, une fonction, ce sont des objets qui se cachent derrière. Par exemple, quand nous avons écrit `ma_liste.append()`, nous avons appliqué la méthode `append` sur l'objet `ma_liste` qui est une liste d'entiers.

Il y a deux concepts importants à comprendre à propos de l'OOP : les «classes» et les «objets».

Objet, méthodes, attributs : Un objet est une sorte de variable, qui peut contenir elle-même des fonctions et des variables.

- Les fonctions contenues dans les objets sont appelées des «méthodes»,
- Les variables contenues dans les objets sont appelées des «attributs».

Classe : Une «classe» est un «prototype» de l'objet, une sorte de modèle à partir duquel on va construire un objet. C'est à l'intérieur de la classe que nous allons définir les attributs (variables contenues dans l'objet) et les méthodes (les fonctions) qui caractérisent l'objet. Une classe est une collection d'attributs (données, variables) et méthodes à l'intérieur d'une entité unique.

Instanciation : A partir d'une classe, on peut créer un objet, qu'on appelle une «instance de classe». À travers l'objet créé, nous pouvons utiliser toutes les fonctionnalités de sa classe.

Ok, tout cela a l'air bien compliqué. Prenons un exemple très simple pour comprendre le concept.

4.1.2 Première approche - création directe de la classe

Nous voulons définir, par exemple, plusieurs personnes, dans un programme Python, en utilisant les mêmes attributs (variables). Les attributs seront, par exemple, son nom, son prénom, son âge, son lieu de résidence. Pour faire cela, nous pouvons utiliser l'entité «Personne», qui sera appelé une classe.

Créons alors cette classe appelée «Personne», qui aura les attributs : nom, prénom, âge, lieu de résidence. Je choisis de ne rien renseigner à ce stade de l'exemple.

6 – classe-1.py - Première approche des classes

```
1 class Personne :
2     nom = ""
3     prenom = ""
4     age = 0
5     lieu_residence = ""
6
7 personne1 = Personne()
8 personne2 = Personne()
9
10 personne1.nom = "DELUNE"
11 personne1.prenom = "Claire"
12 personne1.age = "30"
13 personne1.lieu_residence = "METZ"
14
15 personne2.nom = "CELERT"
16 personne2.prenom = "Jacques"
17 personne2.age = "32"
18 personne2.lieu_residence = "NANCY"
19
20 print(personne1.nom)
```

- Lignes 1 à 5 : Nous définissons une classe, qui contient 4 variables (rappel : variable = attribut), qui ne possède aucune valeur valeur pour le moment, à part des 0 et des chaînes de caractères vides par exemple.
- Lignes 7 et 8 : Nous définissons autant de personnes que nous voulons en utilisant la classe. Nous créons les objets `personne1` et `personne2`, qui sont deux instances de la classe `Personne()`,
- Lignes 10 à 18 : Pour chaque personne (c'est-à-dire chaque objet, ou encore pour chaque instance de la classe Personne), nous affectons le nom, prénom, âge, et lieu de résidence. Nous appelons cela «initialiser» chaque «attribut» de l'objet.
- Ligne 20 : Nous affichons l'attribut `nom` de la personne 1.

►Q.30: Tester ce code, puis afficher l'âge de la personne 2.

4.1.3 Deuxième approche - Utilisation d'un constructeur

Maintenant, nous allons utiliser un «constructeur», dont le rôle est de construire l'objet.

Le constructeur est une méthode spéciale de notre objet ; c'est la méthode qui se charge de créer ses attributs, et qui est toujours appelée lors de la création de l'objet (rappel : objet = instance de la classe). Il est défini sur les lignes 2 à 6.

7 – classe-2.py - Deuxième approche des classes

```
1 class Personne :
2     def __init__(self) :
3         self.nom = ""
4         self.prenom = ""
5         self.age = "0"
6         self.lieu_residence = ""
7
8 personne1 = Personne()
9 personne1.nom = "DELUNE"
10 personne1.prenom = "Claire"
11 personne1.age = "30"
12 personne1.lieu_residence = "METZ"
13
14 print(personne1.age)
```

Ici, le constructeur est vide, il est très courant pour les programmeurs de faire de cette manière lors de la création d'une classe. Vous ne verrez pas tout de suite l'intérêt, mais attendez de voir les deux approches suivantes pour poser la question.

►Q.31: Tester ce code.

Explication : Quand on tape `Personne()`, on appelle le constructeur de notre classe `Personne`, et celui-ci prend en paramètre une variable un peu mystérieuse : `self`. En fait, il s'agit tout simplement de notre objet en train de se créer. Il ne reste plus qu'à renseigner tous ses attributs un par un.

Mais ici encore ce n'est pas une manière de faire très élégante ni pratique.

4.1.4 Troisième approche - Définition des attributs à la création de l'objet

Nous allons alors définir directement les attributs lors de la création de la classe, ainsi l'objet est déjà défini dès sa création par tous ses attributs. Voici le code :

8 – classe-3.py - Troisième approche des classes

```
1 class Personne :
2     def __init__(self) :
3         self.nom = "DELUNE"
4         self.prenom = "Claire"
5         self.age = "30"
6         self.lieu_residence = "METZ"
7
8 personne1 = Personne()
9 print(personne1.age)
```

►Q.32: Tester ce code.

Tout cela fonctionne très bien. Pourtant, il y a un petit problème : avec cette manière de faire, tous les objets (les personnes) que nous allons créer possèdent par défaut les mêmes attributs, ce qui n'est encore une fois pas très élégant, même si nous pouvons les changer par la suite, comme déjà fait précédemment.

4.1.5 Quatrième approche - Les paramètres sont passés lors de la création de l'objet

Pour faire un constructeur un peu plus intelligent, nous allons faire en sorte qu'il puisse prendre en paramètre les attributs. Voyons cela :

9 – classe-4.py - Quatrième approche des classes

```
1 class Personne :
2     def __init__(self, nom, prenom, age, lieu_residence) :
3         self.nom = nom
4         self.prenom = prenom
5         self.age = age
6         self.lieu_residence = lieu_residence
7
8 personne1 = Personne("DELUNE", "Claire", 30, "METZ")
9 personne2 = Personne("CELERT", "Jacques", 32, "NANCY")
10 print(personne1.age)
11 print(personne2.lieu_residence)
```

►Q.33: Tester ce code.

Cette fois, tout à l'air bien cohérent. Pour créer un objet (une personne), nous lui passons lors de l'instanciation tous les paramètres.

4.1.6 Cinquième approche - Ajout à l'objet d'une méthode

Maintenant, nous allons créer la méthode (rappel : méthode = fonction) `description()`, qui va afficher toutes les informations sur une personne.

10 – classe-5.py - Cinquième approche des classes

```
1 class Personne :
2     def __init__(self, nom, prenom, age, lieu_residence) :
3         self.nom = nom
4         self.prenom = prenom
```

```

5     self.age = age
6     self.lieu_residence = lieu_residence
7 def description(self) :
8     print("{1} {0} a {2} ans et habite à {3}.".format(self.nom,
9         self.prenom, self.age, self.lieu_residence))
10
11 personne1 = Personne("DELUNE", "Claire", 30, "METZ")
12 personne1.description()

```

►Q.34: Tester ce code.

►Q.35: Ajouter un attribut à la classe, l'attribut «animal», et nous définirons que Claire a un chat.

►Q.36: Modifier la méthode `description` pour afficher à la suite de la phrase : « ... et possède un chat».

►Q.37: Créer un nouvel objet pour définir et afficher que Jacques possède une tortue.

Les classes, en résumé

Une classe est composée d'un constructeur qui initialise les attributs, et de l'ensemble des méthodes liées à cette classe. A partir de la classe, on crée les objets, appelés instances de classe.

4.2 Les modules et le module «machine»

4.2.1 Définition d'un module

Un module est en quelque sorte un bout de code enfermé dans un fichier, qu'on peut aussi appeler bibliothèque. Un module est composé de classes en général, mais aussi de fonctions et de variables qui ont un rapport entre elles. Nous pouvons utiliser nos propres modules, ou utiliser ceux fournis avec la distribution standard de Python.

Si l'on veut travailler avec les fonctionnalités prévues par le module, il suffit d'importer le module.

MicroPython est fourni avec seulement une petite partie des modules de Python car il doit pouvoir être implanté sur la carte ESP32, c'est-à-dire avec des contraintes matérielles particulières, qui n'existent pas sur un ordinateur classique, à savoir une taille mémoire très réduite en particulier. Par exemple, il n'a pas besoin des fonctions dédiées à l'affichage sur un moniteur car il n'en est pas pourvu.

En revanche, il a besoin de fonctions qui permettent d'accéder aux broches d'entrées/sorties (ou «pin», ou «pattes») du micro-contrôleur et au fonctionnalités matérielles. Le «module» `machine` est un ensemble de classes et fonctions dédiées à cela.

4.2.2 Exemple avec le module math - bibliothèque standard de Python

Ce premier exemple utilise un module de la bibliothèque standard de Python. Par exemple, si vous tapez dans la console l'instruction suivante, elle provoque une erreur. Testez cette instruction :

```
>>> sin(pi/2)
```

Mais si vous importez le module `math`, vous pouvez écrire :

```
>>> import math
>>> math.sin(math.pi/2)
```

Ou alors, si vous importez explicitement les deux fonctions `sin` et `pi`, on pourra écrire plus simplement :

```
>>> from math import sin, pi
>>> sin(pi/2)
```

4.2.3 Exemple avec le module esp ou esp32 - bibliothèque spécifique à l'ESP32

```
import esp
esp.flash_size() # retourne la taille de la mémoire Flash
```

4.2.4 Le module machine

L'instruction suivante permet de rendre le `machine` module accessible à l'interpréteur MicroPython :

```
>>> import machine
```

►Q.38: Depuis la console, saisir cette instruction.

Nous pouvons par exemple connaitre la fréquence de l'horloge du ESP32.

```
>>> machine.freq()      # retourne la fréquence de l'horloge de notre CPU
```

L'instruction ci-dessous rend accessible seulement la classe Pin du module Machine :

```
>>> from machine import Pin      # attention au P : MAJUSCULE
```

A partir de maintenant, nous pouvons créer les objets qui permettront d'accéder aux broches du micro-contrôleur et au fonctionnalités matérielles.

5 Généralités, et premiers tests sur les entrées et sorties numériques

5.1 Utilisation de la console pour lire ou écrire l'état d'une sortie numérique

Dans toute cette section, nos instructions seront saisies dans la console, une à une, pour une exécution et un retour d'information immédiat.

5.1.1 Déclarer un broche (une pin) comme étant une sortie numérique

A partir de maintenant, la classe Pin est importée, et nous allons créer les objets qui permettront de lire une entrée numérique, ou d'écrire un état logique sur une sortie numérique.

►Q.39: Depuis la console, écrire l'instruction suivante qui permet de créer l'objet led, et de déclarer que la broche (ou pin) 2 est en sortie.

```
>>> led = Pin(2, Pin.OUT)
```

5.1.2 Ecrire un état sur une sortie numérique

Nous pouvons maintenant faire passer au niveau Haut ou Bas la sortie, et connecter une LED sur cette sortie pour qu'elle s'allume à l'état Haut.

Une LED bleue est intégrée sur le circuit ESP32, sur la pin 2. La broche 2 est la seule à être munie d'une LED.

De plus, une LED jaune a été câblée sur la platine de connexions, sur la sortie 18, selon le schéma de la figure 12. Nous remarquons la mise en série d'une résistance de limitation de courant de $560\ \Omega$, qui permet de fixer le courant à la valeur d'environ $2\ mA$ dans la LED $((3,3V - 2,2V)/560\Omega = 2mA)$ lorsque la sortie est à l'état haut, soit $3,3V$. Rappel : $2,2V$ est environ la tension de seuil de la LED, c'est à dire la tension à ses bornes lorsque un courant la traverse.

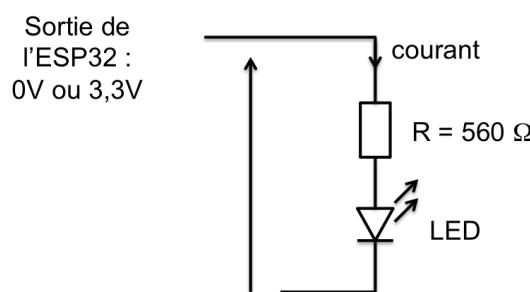


FIGURE 12 – Cablage de la LED, un niveau $3,3V$ provoque son allumage

►Q.40: Maintenant, mettre la sortie (2) à l'état haut, et voir la LED bleue s'allumer.

```
>>> led.value(1)
```

Puis on l'éteint :

```
>>> led.value(0)
```

Ce que nous avons fait, c'est utiliser la méthode value() de l'objet led, pour changer l'état de la pin.

►Q.41: Tester aussi les instructions, et vérifier que c'est juste une autre écriture pour ces fonctions :

```
>>> led.on()
>>> led.off()
```

On aurait pu simplifier ; au lieu de créer l'objet `led`, puis de mettre sa valeur à 1, une seule ligne permet de faire les deux opérations en même temps. Cependant, cela est moins pratique pour la suite, par exemple lorsqu'on doit faire changer l'état de la Pin plusieurs fois.

```
>>> Pin(2, Pin.OUT).value(1)
>>> Pin(2, Pin.OUT).off()
```

Nous aurions pu faire changer d'état n'importe quelle broche de l'ESP32, car elles sont toutes disponibles en sortie numérique.

►Q.42: Faire changer l'état de la LED jaune connectée sur la sortie 18.

5.1.3 Utilisation d'une LED au format «Grove»

Pour découvrir un connecteur «intéressant» et pratique, nous allons ajouter une LED rouge, une nouvelle LED qui sera au format Grove, que nous allons câbler sur la sortie 19.

►Q.43: Connecter la LED Grove sur la sortie 19, et vérifier qu'il est possible de l'allumer et l'éteindre. Les fils sont repérés comme ceci :

- Noir : Masse.
- Rouge : 3,3V.
- Blanc : fil non connecté.
- Jaune : tension de commande (0V ou 3,3V).

5.1.4 Lire un état sur une entrée numérique

Lire un état logique sur une entrée numérique, c'est lire une tension qui présente deux valeurs différentes, appelés niveaux. Cette tension doit varier suffisamment lors de l'appui par exemple sur un bouton poussoir, de manière à être détectable.

Dans l'idéal, cette tension devrait passer de 0V à 3,3V lors de l'appui (ou l'inverse), dans le cas d'une tension d'alimentation de 3,3V. En pratique, l'entrée reconnaît sans erreur le niveau logique dont la tension se situe dans deux plages de valeurs. Ainsi :

- si nous avons entre 2.5V et 3.6V, l'entrée reconnaît un état Haut,
- si nous avons entre -0.3V et 0.8V, l'entrée reconnaît un état Bas.

Ainsi, notre signal d'entrée doit être généré dans cette gamme de tension. Pour cela, nous utilisons par exemple un bouton poussoir, qui commande un inverseur qui ferme le contact pour une mise à 3.3V, la tension d'alimentation. Voir figure 13 page 28. Le fonctionnement est le suivant :

- lorsque le poussoir BP n'est pas pressé, le contact est ouvert, et la tension $u(t)$ est à 0V,
- lorsque le poussoir est pressé, le contact est fermé, et la tension $u(t)$ passe de 0 à 3,3V.

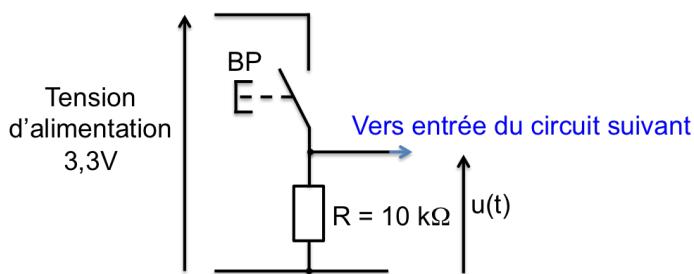


FIGURE 13 – Cablage du bouton poussoir, l'appui provoque le passage à 3,3V

Sur la platine de connexions, un bouton poussoir est connecté sur l'entrée 25 selon le schéma de la figure 13.

```
1 >>> bp = Pin(25, Pin.IN)
2 >>> bp.value()      # lit et retourne la valeur de l'objet bp
```

►Q.44: Vérifier que l'instruction en ligne 2 retourne 0 ou 1, en fonction du fait que vous appuyez ou pas sur le bouton poussoir lors de l'exécution. Pour cela, saisissez plusieurs fois l'instruction, ... ou lire la question suivante.

►Q.45: Afin d'éviter de saisir une instruction plusieurs fois, le raccourci clavier est, lorsque l'on est dans la console, les touches «flèche haut» et «flèche bas», qui permettent de remonter ou redescendre dans l'historique des instructions.

►Q.46: Connecter un bouton poussoir sur l'entrée 34. Pour cela, on utilise un connecteur «GROVE», voir figure 14 page 29, dont les fils sont repérés comme ceci :

- Noir : Masse.
- Rouge : 3,3V.
- Blanc : fil non connecté.
- Jaune : tension qui passe de 0 à 3,3V lors de l'appui.

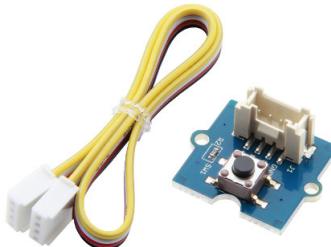


FIGURE 14 – Bouton poussoir Grove, avec son câble

```
1 >>> bpGrove = Pin(34, Pin.IN)
2 >>> bpGrove.value()      # lit et retourne la valeur de l'objet bp
```

►Q.47: Vérifier que l'instruction en ligne 2 retourne 0 ou 1, en fonction du fait que vous appuyez ou pas sur le bouton poussoir Grove.

5.2 Création d'un script

5.2.1 Faire clignoter une LED en continu - Boucle while

Nous allons faire clignoter la LED bleue connectée à la sortie 2.

11 – led.py - Fait clignoter une LED à 1 Hz ; solution 1

```
1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 while True:
7     led.value(1)          # on allume la LED
8     time.sleep(0.5)       # delai de 0.5 s
9     led.value(0)          # on eteint la LED
10    time.sleep(0.5)      # delai de 0.5 s
11
12 #while True:
13 #    if led.value() == 1 :           # si la LED est allumee...
14 #        led.value(0)               # on l'eteint
15 #    else :                         # sinon (elle est eteinte)...
16 #        led.value(1)               # on l'allume
17 #    time.sleep(0.5)                # delai de 0.5 s entre 2 boucles (1 Hz)
18
19 #while True:
20 #    led.value(not led.value()) ;  # inversion de l'état de la LED
21 #    time.sleep(0.5)              # delai de 0.5 s entre 2 boucles (1 Hz)
```

Les lignes 6 à 10 sont faciles à comprendre ; on allume ou on éteint la LED, et entre chaque changement on attend 0,5s, ce qui fait bien une période de 1s, soit une fréquence de 1 Hz.

Variante 1 : Les lignes 12 à 17 sont mises en commentaires. Elles réalisent à peu près la même chose que les lignes 6 à 10. En ligne 13, on teste si la LED est allumée à l'aide de l'instruction `led.value()` qui commande une lecture de l'état de la LED, ou plutôt de la pin qui pilote la LED. Si la pin est à l'état «haut», la lecture retourne la valeur «1». On teste si la lecture vaut 1, et si oui on éteint la LED avec

l'instruction `led.value(0)`, et sinon on l'allume avec l'instruction `led.value(1)`. Et ensuite, on attend 0,5 s avant de recommencer le cycle en revenant à la ligne 12.

►Q.48: Tester ce code. Attention, pour ce test, le plus simple est de permuter les mises en commentaires des lignes 12-17 et 6-10.

►Q.49: Sortir du programme ; pour cela, nous n'avons pas d'autre possibilité que de l'interrompre en cliquant sur le bouton «Stop» de Thonny.

►Q.50: Vérifier que nous pouvons simplifier en ligne 13 en écrivant simplement : `if led.value()` :

Variante 2 : Les lignes 19 à 21 sont mises en commentaires. Elles réalisent la même fonction que les lignes 12 à 17. Cette écriture signifie que l'on écrit, en ligne 20, à chaque itération, la valeur actuelle lue, mais *inversée* ; nous changeons donc bien d'état à chaque itération.

►Q.51: Vérifier cela en permutant les mises en commentaires des lignes 19-21 et 12-17.

►Q.52: Modifier le code pour faire clignoter la LED connectée sur la borne 18, puis 19.

5.2.2 Faire clignoter la LED un certain nombre de fois - L'instruction FOR

Décidons maintenant d'effectuer 6 changements d'état, soit 3 cycles complet allumage/extinction, chaque cycle durant une seconde (2 fois 0,5 s). Le programme dure 3 secondes. Le code est alors :

12 – led-3-fois.py - Fait clignoter 3 fois une LED, puis s'arrête

```
1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 for a in range (6) :
7     led.value(not led.value())
8     time.sleep(0.5)
```

La variable `a` prendra successivement toutes les valeurs entières définies par `range(6)`, soit les 6 valeurs de 0 à 5. Donc nous avons bien réalisé 3 cycles complets ON-OFF.

►Q.53: Vérifier le bon fonctionnement.

5.2.3 Mesure de temps

Nous allons, par exemple, mesurer le temps dont a besoin MicroPython pour allumer et éteindre la LED 10 000 fois, le plus rapidement possible, donc sans délai entre chaque état. Nous afficherons le temps total pour les 10 000 cycles, et le temps pour un seul cycle (en μ s). Nous ferons ainsi :

13 – led-temps.py - Fait clignoter 10 000 fois une LED, mesure le temps

```
1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5
6 start = time.ticks_ms()          # memorise la valeur du ticks au lancement
7
8 for iteration in range (10000) :
9     led.value(1) ; led.value(0)
10
11 delta = time.ticks_ms() - start # calcule l'écart : apres - avant
12
13 print("Temps total pour 10 000 cycles : ", delta, "ms")
14 print("Temps pour 1 cycle ON/OFF      :" , delta/10, "microsecondes")
15 print("Fréquence de basculement       : {0:2.1f} kHz".format(10000/delta))
```

►Q.54: Vérifier que le temps est assez court, en tout cas le clignotement est assez rapide pour ne pas être détectable à l'oeil. Nous voyons juste la LED s'allumer environ à la moitié de sa luminosité maxi, pendant moins d'une seconde, environ 200 ms.

En réalité, la LED est allumée durant la moitié de sa période, puis elle est éteinte durant l'autre moitié, et cela 10 000 fois durant ces 200 ms environ.

Nous remarquons en ligne 9 que plusieurs instructions peuvent se trouver sur la même ligne ; il faut juste les séparer par le signe « ; ». Cela peut être utile parfois, même si la lisibilité en est réduite. Cela est à éviter en général dans les programmes. Pourtant, dans cet ouvrage, j'utiliserais parfois cette manière de procéder, pour écourter la longueur du code, lorsque cela n'est pas gênant pour la lisibilité.

5.2.4 Considérations/rappels sur les bibliothèques - et sur le module time

Par défaut, avant l'importation, Python n'a pas accès à la fonction `sleep` car elle est incluse dans le module `time`. Cette fonction `sleep` permet de temporiser, d'attendre, un temps exprimé en secondes. Nous avons importé tout ce module `time`, donc il est nécessaire de définir quelle fonction nous appelons précisément, et cela se fait par `time.sleep()`.

En réalité, quand nous avons tapé `import time`, cela a créé un espace de noms dénommé `time`, contenant les variables et fonctions du module `time`, et ainsi cela rend accessible la fonction `sleep`.

En résumé sur les modules : Pour importer un module, même s'il fait partie de la distribution de MicroPython, il est nécessaire de le signifier avec l'instruction `import` de manière à le rendre accessible. Pour importer un module qui est déjà chargée dans la mémoire de l'ESP32, on opère de la même manière. Nous verrons cela plus loin lors de l'importation d'un module que nous créerons, et que nous chargerons en mémoire.

Nous pouvons aussi importer d'un seul coup toutes les fonctions du module `time`, et il aurait alors fallu écrire pour cela :

```
from time import *
...
sleep(0.5)
```

Cela nous permet d'éviter d'écrire `time.sleep`, mais directement `sleep`.

►Q.55: Tester cela.

Mais nous pouvons aussi, de manière plus explicite, importer seulement la fonction `sleep`. Voilà donc finalement la manière la plus élégante de faire, car nous précisons explicitement quelle fonction nous voulons utiliser, au lieu de toutes les importer :

```
from time import sleep
...
sleep(0.5)
```

►Q.56: Tester cela.

5.2.5 Revenons sur la lecture d'une entrée numérique

Nous n'avons utilisé jusqu'à maintenant que des sorties numériques. Nous avons ainsi pu allumer une LED ou commander n'importe quel système qui est capable de reconnaître (ou recevoir) un état logique et agir en conséquence...

Mais nous pouvons aussi lire l'état d'une entrée, ce qui permet de déterminer si un bouton a été pressé ou pas (le bouton pressé générera dans notre cas un niveau «Haut»). La lecture se fait, dans l'exemple qui suit, à la fréquence de 10 lectures par seconde, car nous attendons 0,1 s entre deux itérations (voir ligne 12).

L'exemple suivant allume une LED si le bouton est pressé. A chaque itération, nous lisons (ligne 7 à 11) l'état du bouton, et s'il est pressé, on allume, sinon on éteint.

Une autre manière de faire, sur les trois lignes commentées (lignes 14 à 16), est de lire l'état du bouton par `bouton.value()`, et d'affecter directement le résultat à la sortie 2 avec l'instruction `led.value(bouton.value())`, ce qui allume la LED s'il est «True».

►Q.57: Un bouton poussoir est déjà connecté sur l'entrée 25 . Vérifier le fonctionnement du programme, avec les deux manières de faire.

14 – led-bouton.py - Allume une LED si un bouton est pressé

```
1 from machine import Pin
2 from time import sleep
3
4 led = Pin(2, Pin.OUT)
5 bouton = Pin(25, Pin.IN)
```

```

6
7 while True :
8     if not bouton.value() :
9         led.value(1)
10    else :
11        led.value(0)
12    sleep(0.1)
13
14 #while True :
15 #    led.value(bouton.value())
16 #    sleep(0.1)

```

►Q.58: Vérifier que le temps de 0,1 s (ligne 12 ou 16) pour l'attente est correct. Vérifier qu'une valeur de 1 s n'offre pas un temps de réponse assez rapide entre l'appui et l'allumage de la LED.

5.2.6 Arrêt du programme par un bouton poussoir

Normalement, un programme, quand il est lancé, ne s'arrête que lorsque l'utilisateur le lui demande, à l'aide d'un bouton «Quitter» ou «Stop» par exemple.

Mais, pour faciliter et simplifier les choses, nous allons faire en sorte que le programme s'exécute durant un temps juste suffisant pour constater son bon fonctionnement, ou prévoir une sortie du programme par un appui sur un poussoir physique de notre montage par exemple.

Dans le code qui suit, la condition d'arrêt sera l'appui sur le bouton poussoir connecté à la borne 25.

15 – led-stop.py - Fait clignoter une LED à 1 Hz avec STOP par bouton

```

1 from machine import Pin
2 import time
3
4 led = Pin(2, Pin.OUT)
5 bp = Pin(25, Pin.IN)
6
7 while not bp.value():                      # continue tant que bp.value() == 0
8     led.value(not led.value())
9     time.sleep(0.5)

```

Attention : Dans ce code, la lecture du poussoir a lieu toutes les 0,5 seconde, il est donc possible que la lecture ne soit pas réalisée si l'appui ET le relâchement ont eu lieu tous les deux pendant l'attente entre deux lectures (ligne 9). Dans ce cas, appuyer un peu plus longtemps, pour être assuré que votre appui soit lu lors du passage par la ligne 7.

►Q.59: Tester en appuyant brièvement (fonctionnement aléatoire), puis longuement (fonctionnement correct) sur le poussoir.

5.3 Application : un temporisateur d'éclairage

Voyons maintenant un exemple qui résume tout ce que nous avons vu jusqu'à présent : un temporisateur d'éclairage. Ce temporisateur permet, lors de l'appui sur un bouton (pin 25), d'allumer pendant 2 secondes une lumière, simulée par la LED connectée sur la pin 2. A chaque appui sur le bouton, on affecte au compteur `compt` la valeur 20. A chaque itération, qui dure 0.1 s, le compteur est décrémenté. La lumière n'est allumée que si la valeur du compteur est positive, soit pendant $20 * 0,1s = 2s$.

16 – tempo-eclairage.py - Temporisateur d'éclairage

```

1 from machine import Pin
2 import time
3
4 led = Pin(21, Pin.OUT)
5 bp = Pin(39, Pin.IN)
6
7 compt = 0
8 while True:
9     if not bp.value():
10         compt = 20

```

```

11     print("Light ON")
12 print(compt)
13 if compt > 0:
14     led.value(1)
15 else:
16     led.value(0)
17 time.sleep(0.1)
18 compt = compt - 1
19 if compt == 0:
20     print("Light BBF")

```

►Q.60: Vérifier que l'appui sur le bouton, même si la lumière est déjà allumée, relance un cycle d'éclairage pour une durée de 2 s.

Pour comprendre en détail ce qu'il se passe, j'ai utilisé la fonction `print` pour suivre les valeurs du compteur, et l'état de la lumière. Nous constaterons que le compteur peut prendre des valeurs négatives, ce qui n'est pas gênant en première approche.

►Q.61: Vérifier que «Light ON» s'affiche quand on appuie sur le poussoir, et s'affiche à nouveau toutes les 0,1 s tant que le poussoir est maintenu pressé.

►Q.62: Remplacer la ligne 16 par le code `compt = compt - (compt > 0)`, afin d'éviter que le compteur ne prenne des valeurs négatives.

Voici maintenant quelques simplifications d'écriture, et une réduction du code à son essentiel :

17 – tempo-eclairage-optimise.py - Temporisateur d'éclairage optimisé

```

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 compt = 0
8 while True :
9     if bp.value() : compt = 20
10    led.value(compt > 0)   # si compt > 0, cela fait led.value(True)
11    sleep(0.1)
12    compt -= (compt > 0)  # raccourci pour : compt = compt - (compt > 0)

```

►Q.63: Arrêter le déroulement du programme par le bouton «Stop» de Thonny.

5.4 Flasher le code dans la carte

Notre temporisateur, pour le moment, doit être relié à l'ordinateur, ou une carte Raspberry qui le pilote, pour pouvoir fonctionner ; le code est seulement stocké dans l'IDE Thonny, et n'est jamais stocké sur la carte. La carte ESP32 reçoit seulement les instructions à réaliser, ou des morceaux de script, dans sa mémoire vive (RAM). L'ordinateur doit alors rester connecté à la carte, car il effectue le transfert des infos, et peut afficher, nous l'avons vu, les résultats avec la fonction `print`.

Cependant, nous souhaitons maintenant que le fonctionnement soit autonome, de manière à ce que l'ESP32 puisse être installé dans un endroit approprié pour sa fonction, c'est-à-dire dans une cage d'escalier par exemple, relié à un bouton du type interrupteur «Legrand», et qui commande un relais électromagnétique pour allumer ou éteindre un vraie ampoule à visser, du type 230V-10W à LED et culot E27, installé dans un luminaire fixé au plafond.

Nous allons donc envoyer, ou autrement dit «téléverser», le code dans la mémoire Flash (mémoire maintenue même hors tension) de la carte. Ce code sera téléversé comme étant le script principal, le `main.py`, c'est-à-dire comme celui qui doit être exécuté lorsque la carte est mise sous tension, et qu'elle n'est pas connectée à l'ordinateur et sous son contrôle.

Procédure : Sélectionner l'onglet «File», choisir «Save As», puis «MicroPython device», et enregistrer sous le nom `main.py`.

- Lorsque vous utilisez le nom `main.py`, alors à chaque démarrage de la carte, elle exécutera ce code là, en commençant s'il existe par le code contenu dans le `boot.py`, qui est utilisé pour initialiser le système.

- Si vous utilisez un autre nom, ce fichier sur la carte pourra seulement être utilisé par le fichier `main.py` en tant que ressource, pour y stocker des modules, des fonctions, des classes, des variables.

A partir de maintenant, le code est chargé en mémoire Flash, et il y restera même si la carte n'est plus sous tension. Il est alors possible de déconnecter l'ordinateur, et le code est exécuté si on alimente simplement la carte par une alimentation via l'USB ou une alimentation dédiée si la carte en est pourvue.

►Q.64: Connectez un chargeur de smartphone, pour constater que notre temporisateur fonctionne correctement.

►Q.65: Connecter le relais, et vérifier le bon fonctionnement.

6 Les sorties PWM

6.1 Retour sur le principe de la sortie numérique

Une sortie numérique génère au niveau Haut une tension d'environ 3,3V, et au niveau Bas une tension d'environ 0V. Il est possible d'obtenir d'autres niveaux de tension, c'est à dire toute valeur entre 0 et 3,3V, mais seulement avec des sorties analogiques, mais ce n'est pas l'objet de cette section. Voyons plutôt la sortie numérique PWM, qui permet de réaliser quelque chose de très similaire au comportement de la sortie analogique, et qui est plus souvent utilisée.

6.2 PWM pour utilisation avec une LED

Une sortie PWM est une sortie numérique, dont l'état de la sortie change à une haute fréquence (en général entre 100 et 40 000 Hz), de sorte que l'œil ne perçoivent qu'une valeur moyenne. Il est donc possible, avec seulement un état allumé et un état éteint, d'obtenir toutes les valeurs d'éclairement intermédiaire de 0 à 100%, pour des valeurs d'entrée de 0 à 1023, selon la figure suivante :

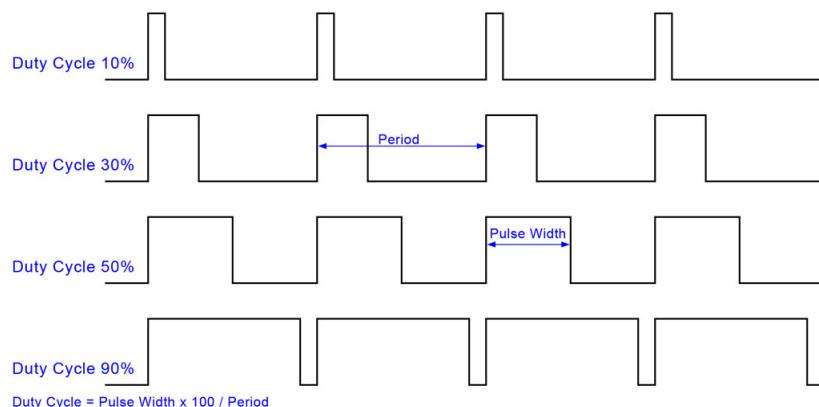


FIGURE 15 – Signal PWM généré par la fonction `AnalogWrite`, et rapport cyclique de 10% à 90%

Le rapport cyclique de 100% est obtenu avec la valeur maximale 1023.

6.2.1 Avec la console

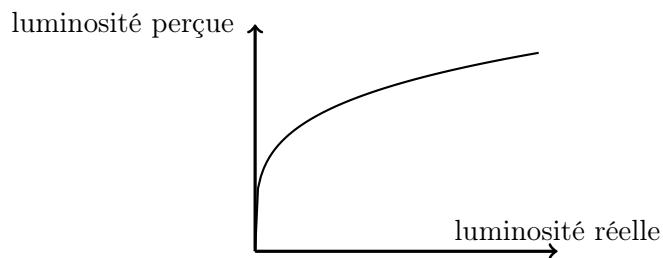
Les instructions suivantes sont saisies directement dans la console.

```
>>> from machine import Pin, PWM
>>> frequence = 500
>>> led = PWM(Pin(2), frequence) # initialise la fréquence du PWM
>>> led.duty (1023)
```

►Q.66: Faire varier la valeur du rapport cyclique, par exemple mettre 1023, 100, 10, 1.

►Q.67: Vérifier que la luminosité diminue.

►Q.68: Vérifier aussi que l'impression de lumière n'est pas proportionnelle au rapport cyclique. En réalité, notre œil réagit à peu près en fonction du logarithme ou de la racine carrée de la luminosité, c'est à dire qu'une lumière 10 fois plus forte (rapport cyclique 10 fois plus grand) sera perçue comme seulement 2 à 3 fois plus forte. Revérifier si besoin avec les valeurs 1023 et 100.



6.2.2 Création de trois cycles de montée

Pour vérifier la non linéarité, nous faisons croître le rapport cyclique de 0 à 1023, par pas de 10, toutes les 10 ms, donc en environ une seconde, et cela trois fois.

18 – pwm.py - PWM - 3 cycles de montée

```

1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq : 500 Hz
5
6 for boucle in range(3) :
7     for i in range(0, 1023, 10) :    # de 0 à 1023 par pas de 10
8         led.duty(i)
9         sleep_ms(10)
10 led.deinit()                      # libération des ressources du PWM

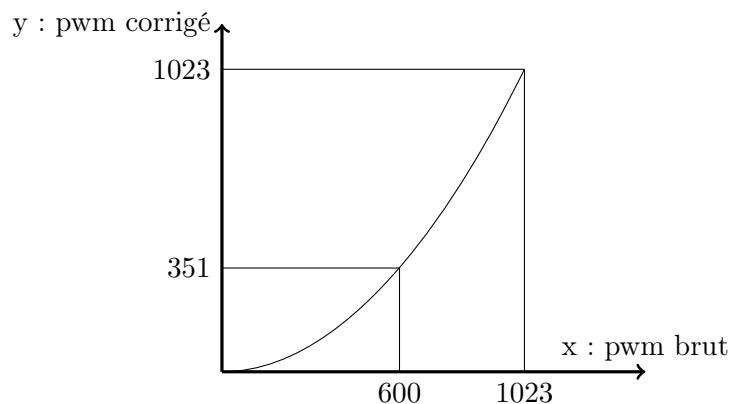
```

La montée de 0 à 1023 est linéaire, mais à cause de la non-linéarité de la perception de la luminosité, il nous semble que la montée en luminosité semble rapide ; en effet, la LED semble être rapidement allumée à la moitié du temps, puis semble rester allumée à pleine puissance, alors qu'en réalité le rapport cyclique continue d'augmenter.

►Q.69: Vérifier cela.

6.2.3 Prise en compte du caractère non linéaire de l'œil

Nous allons corriger cela en partie, en créant une progression non linéaire de la luminosité, et par exemple, pour simplifier, une progression parabolique. Le rapport cyclique brut sera noté x , et le rapport cyclique corrigé sera noté y . Ainsi, nous aurons un rapport cyclique corrigé $y = 0$ pour un rapport cyclique brut $x = 0$; de même nous aurons $y = 1023$ pour $x = 1023$, mais nous aurons une progression parabolique de la forme $y = x^2$ pour toute valeur de x entre 0 et 1023.



►Q.70: Comprendre que la manière de faire cela peut être la suivante :

- diviser x par 1023, pour avoir x variant linéairement de 0 à 1,
- prendre le carré du résultat pour avoir une parabole, variant aussi de 0 à 1,
- et multiplier par 1023 pour retrouver notre parabole variant de 0 à 1023.

Ainsi, nous aurons : $y = (x/1023)^2 * 1023$

►Q.71: Calculer la valeur de y pour $x = 600$.

19 – calcul-correction.py - Calcul correction luminosité

```
1 for valeur_brute in [0, 200, 400, 600, 800, 1023] :
2     valeur_corrigee = int((valeur_brute/1023)** 2 * 1023)
3     print('valeur brute : {0:4d} ; valeur corrigee : {1:4d} '.format
4           (valeur_brute , valeur_corrigee))
```

Dans ce code, nous prenons les 7 valeurs de la liste, de 0 à 1023, pour vérifier que nous obtenons après calcul une parabole passant par (0 ; 0) et (1023 ; 1023), avec son sommet en 0. Nous obtenons le résultat ci-dessous :

```
valeur brute :    0 ; valeur corrigee :    0
valeur brute :  200 ; valeur corrigee :   39
valeur brute :  400 ; valeur corrigee :  156
valeur brute :  600 ; valeur corrigee :  351
valeur brute :  800 ; valeur corrigee :  625
valeur brute : 1023 ; valeur corrigee : 1023
```

►Q.72: Comprendre aussi que nous avons arrondi les valeurs corrigées avec la fonction `int`, car la méthode `duty` attend des valeurs entières, sinon une erreur serait générée.

►Q.73: Remarquer que nous avons utilisé une nouvelle syntaxe pour l'affichage ; celle-ci nous permet d'aligner les chiffres à droite en indiquant 4 décimales. Le `{0:4d}` et le `{1:4d}` sont donc le premier (0) et le deuxième élément (1) de la liste de valeurs à afficher, avec 4 chiffres affichés, et indiqués à l'endroit où ils vont apparaître dans la chaîne de caractère. Les valeurs seront données par la «méthode» `format`.

Maintenant, nous allons utiliser cette correction pour le PWM.

20 – pwm-correction.py - PWM avec correction luminosité

```
1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq a 500 Hz
5
6 for boucle in range(3) :
7     for i in range(0, 1023, 10) :
8         led.duty(int((i/ 1023)** 2 * 1023))
9         sleep_ms(10)
10    led.deinit()
```

►Q.74: Vérifier que la montée en luminosité se fait de manière beaucoup plus linéaire.

Le coefficient 2 est donc en première approximation une valeur correcte. Mais peut être que nous pourrions l'ajuster. En effet, est-ce que le coefficient 3 permettrait une progression «ressentie» plus linéaire ? Une solution serait de tester le programme avec cette nouvelle valeur. Mais nous allons plutôt demander, au cours de l'exécution du code, à l'utilisateur quelle valeur de coefficient il souhaite tester :

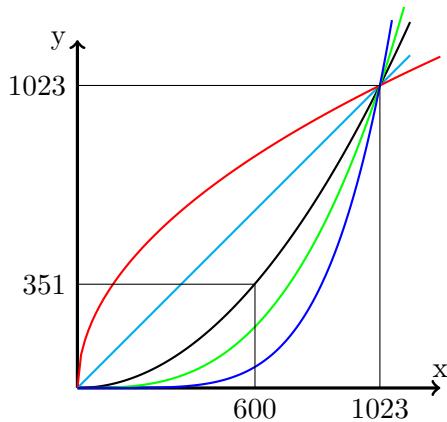
21 – pwm-correction-input.py - PWM avec correction luminosité choisie

```
1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # Initialise la freq a 500 Hz
5
6 coef = float(input('Coefficient : '))
7
8 for boucle in range(3) :
9     for i in range(0, 1023, 10) :
10        led.duty(int((i/1023)** coef * 1023))
11        sleep_ms(10)
12    led.deinit()
```

La fonction `input()` demande à l'utilisateur, par l'intermédiaire du Shell, quelle valeur il choisit pour son coefficient de correction. Cette valeur renournée est une chaîne de caractère, que l'on transforme en nombre flottant (à virgule) par la fonction `float`.

►Q.75: Vérifier les valeurs 2, 3, 10, puis 0.5, et toute valeur réelle que vous souhaitez, entre 0.1 et 10.

Sur le graphe ci dessous sont tracées les courbes pour un coefficient de 2 (en noir), 3 (en vert), 5 (en bleu), et aussi 1 (en cyan), et 0.5 (en rouge, on aura reconnu une fonction racine carrée).



Choisir la valeur de coefficient que vous préférez. Nous voyons en ligne 11 du code suivant que j'ai choisi le coefficient 3.5.

Maintenant, nous allons créer la fonction `correction()` pour simplifier l'écriture, et permettre une réutilisation aisée dans un autre code.

22 – pwm-fonction-correction.py - PWM avec correction luminosité par fonction

```

1 from time import sleep_ms
2 from machine import Pin, PWM
3
4 led = PWM(Pin(2, Pin.OUT), 500)      # initialise la freqence a 500 Hz
5
6 def correction(i, coef) :
7     return int ((i / 1023) ** coef * 1023)
8
9 for boucle in range(3) :
10    for i in range(0, 1023, 10) :
11        led.duty(correction(i, 3.5)) # utilisation fonction correction
12        sleep_ms(10)
13 led.deinit()

```

7 Utilisation du ruban de LED NeoPixel - Première approche

7.1 Mélange de couleurs

Incorporer de nombreuses LEDs dans un projet électronique peut compliquer sa réalisation, avec un code difficile à maintenir. Mais l'arrivée de LEDs disposant d'une puce pilote intégrée change radicalement la donne en allégeant le travail du microcontrôleur et le câblage. Elle permet aussi de se concentrer sur le code. Le WS2812, avec sa source de lumière intégrée, ou plus communément nommé «NeoPixel» par la société «Adafruit», est la dernière avancée dans la quête pour obtenir des LEDs simples à mettre en œuvre et bon marché.

Les LEDs rouge, vert et bleue sont intégrées côté à côté sur un petit support CMS (composants montés en surface), lui-même intégré sur la puce du pilote (le contrôleur), le tout contrôlé par un simple fil. Elles peuvent être utilisées individuellement, arrangées pour former une longue chaîne, ou assemblées pour réaliser des formes intéressantes. La couleur finale obtenue est l'addition des trois couleurs primaires, c'est-à-dire le rouge, le vert, et le bleu. Voir figure 16 page 38.

7.2 Premier essai

Le réglage de la luminosité (sur le principe du PWM vu précédemment) se fait ici de 0% à 100%, pour un réglage de 0 à 255 (et non plus 1023 comme précédemment). Ainsi, pour une valeur de 30, on aura 30/255 de la pleine luminosité (hors prise en compte de la non linéarité de l'œil ...)

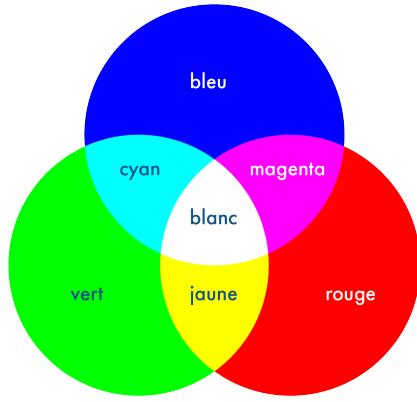
►Q.76: Tester le programme suivant pour comprendre :

23 – neopixel.py - Test du ruban NeoPixel

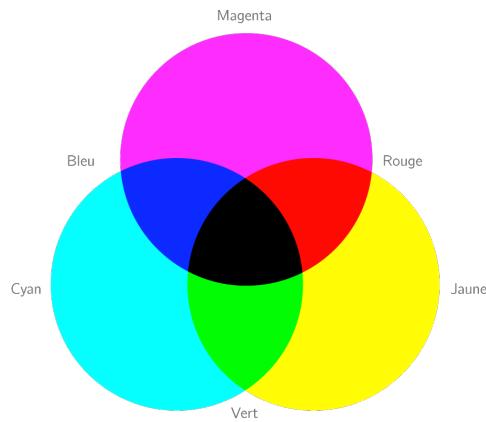
```

1 from machine import Pin, ADC
2 from neopixel import NeoPixel  # import de la classe NeoPixel du module

```



(a) Synthèse additive : rouge, vert, bleu



(b) Synthèse soustractive : magenta, cyan, jaune

FIGURE 16 – Les couleurs primaires

```

3
4 n = 8                                # nombre de pixels
5 p = 26                                 # pin de commande du neopixel
6 np = NeoPixel(Pin(p), n)                # creation de l'instance np
7
8 np[0] = (10, 0, 0)                      # rouge, 10/255 brightness, soit 4%
9 np[1] = (40, 0, 0)                      # rouge, 40/255 brightness, soit 16%
10 np[2] = (0, 40, 0)                     # vert, 40/255 brightness, soit 16%
11 np[3] = (0, 0, 40)                     # bleu, 40/255 brightness, soit 16%
12 np[4] = (40, 40, 0)                    # rouge + vert = jaune
13 np[5] = (0, 40, 40)                    # vert + bleu = cyan
14 np[6] = (40, 0, 40)                    # rouge + bleu = magenta (violet)
15 np[7] = (40, 40, 40)                  # rouge + vert + bleu = blanc
16 np.write()

```

Pour bien comprendre que chaque élément du NéoPixel est en fait composée de 3 LEDs, voici un code avec des luminosités plus faibles qui permettent de bien voir les différentes composantes.

7.3 Deuxième essai pour voir les 3 composantes RGB

24 – neopixel-test.py - Test du ruban NeoPixel - Deuxième essai

```

1 from machine import Pin, ADC
2 from neopixel import NeoPixel    # import de la classe NeoPixel du module
3
4 n = 8                                # nombre de pixels
5 p = 26                                 # pin de commande du neopixel
6 np = NeoPixel(Pin(p), n)                # creation de l'instance np
7
8 np[0] = (10, 0, 0)                      # rouge, 10/255 brightness, soit 4%
9 np[1] = (0, 10, 0)                      # vert, 10/255 brightness, soit 4%
10 np[2] = (0, 0, 10)                     # bleu, 10/255 brightness, soit 4%
11 np[3] = (10, 10, 10)                   # blanc = rouge + vert + bleu
12 np[4] = (50, 0, 0)                      # rouge, 50/255 brightness, soit 20%
13 np[5] = (0, 50, 0)                      # vert, 50/255 brightness, soit 20%
14 np[6] = (0, 0, 50)                      # bleu, 50/255 brightness, soit 20%
15 np[7] = (50, 50, 50)                   # blanc = rouge + vert + bleu
16 np.write()                            # Ecriture des valeurs sur le ruban

```

Maintenant, nous allumons toutes les LED, puis nous les éteignons, dans le même ordre.

25 – neopixel-defilement.py - Défilement des LED

```

1 from machine import Pin, ADC
2 from time import sleep

```

```

3 from neopixel import NeoPixel # import de la classe NeoPixel du module
4
5 n = 8 # nombre de pixels
6 p = 26 # pin de commande du neopixel
7 delai = .125
8 np = NeoPixel(Pin(p), n) # creation de l'instance np
9
10 for x in range(0, n):
11     np[x] = (0, 0, 255) # bleu, 100% brightness
12     np.write()
13     sleep(delai)
14
15 for x in range(0,n):
16     np[x] = (0, 0, 0) # eteint
17     np.write()
18     sleep(delai)

```

7.4 L'utilisateur choisit le nombre de LED allumées

Maintenant, nous allons allumer x pixels, x étant choisi par l'utilisateur. On allumera depuis la LED 0 jusqu'à la LED $x - 1$. Pour comprendre ce code, prenons par exemple la valeur de $x = 2$. Dans la boucle `for` la variable `led` prend les valeurs de 0 à 7 :

- lorsque `led = 0`, on a $0 < 2$, donc la led 0 est allumée,
- lorsque `led = 1`, on a $1 < 2$, donc la led 1 est allumée,
- lorsque `led = 2`, on n'a pas $2 < 2$, donc la led 1 n'est pas allumée,
- lorsque `led > 2`, les led sont éteintes.

Donc nous avons bien 2 LEDs allumées.

26 – neopixel-x-allumes.py - L'utilisateur choisit d'allumer x LED

```

1 from machine import Pin, ADC
2 from time import sleep
3 from neopixel import NeoPixel
4
5 n = 8 # nombre de pixels
6 p = 26 # pin de commande du neopixel
7 np = NeoPixel(Pin(p), n) # creation de l'instance np
8
9 x = int(input("Combien voulez vous de LED allumees ? : "))
10
11 for led in range(0, n):
12     np[led] = (0, 0, 20*(led<x)) # = 255 si (led < x) ; = 0 sinon
13 np.write()

```

►Q.77: Vérifier que l'on peut demander plus de 8 LED allumées, et que cela ne génère pas d'erreur.

►Q.78: Modifier le code pour allumer en vert les autres LED au lieu de les laisser éteintes.

7.5 Tirage aléatoire

Maintenant, nous allons effectuer un tirage aléatoire d'un nombre compris entre 0 et 8, et afficher le résultat sur le NeoPixel.

►Q.79: Vérifier le bon fonctionnement.

27 – neopixel-random.py - On allume x LED de manière aléatoire

```

1 from machine import Pin, ADC
2 from time import sleep
3 from neopixel import NeoPixel
4 from random import randint
5
6 n = 8 # nombre de pixels
7 p = 26 # pin de commande du neopixel
8 np = NeoPixel(Pin(p), n) # creation de l'instance np
9

```

```

10 x = randint(0,8)                      # tirage aleatoire entre 0 et 8
11 print(x)
12
13 for led in range(0, n):
14     np[led] = (0, 0, 50*(led<x))    # = 50 si (led < x) ; = 0 sinon
15 np.write()

```

Pour vérifier le bon fonctionnement de la fonction `randint`, nous allons effectuer un tirage de 20 valeurs, puis les classer, pour observer la répartition.

28 – tirage-aleatoire.py - Tirage de 20 valeurs entières entre 0 et 8

```

1 from random import randint
2
3 liste=[]
4
5 for i in range(20):          # pour 20 nombres...
6     tirage = randint(0,8)    # tirage entre 0 et 8
7     liste.append(tirage)    # ajout a la liste des tirages
8
9 print(liste)
10 liste.sort()                # classement de la liste dans l'ordre croissant
11 print(liste)

```

Et l'on obtient par exemple ceci :

```
[2, 0, 6, 6, 4, 4, 8, 8, 5, 2, 0, 5, 4, 7, 1, 1, 4, 3, 1, 7]
[0, 0, 1, 1, 1, 2, 2, 3, 4, 4, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8]
```

►Q.80: Vérifier le bon fonctionnement.

7.6 Arc-en-ciel

Et voici un joli effet «Arc-en-ciel» ...

29 – neopixel-rainbow.py - Arc-en-ciel sur NéoPixel

```

1 from machine import Pin, ADC
2 from time import sleep_ms
3 from neopixel import NeoPixel
4
5 n = 8                                # nombre de pixels
6 p = 26                                 # pin de commande du neopixel
7 np = NeoPixel(Pin(p), n)               # creation de l'instance np
8 bp = Pin(25, Pin.IN)
9
10 def wheel(pos):
11     if pos < 0 or pos > 255:
12         return (0, 0, 0)
13     if pos < 85:
14         return (255 - pos * 3, pos * 3, 0)
15     if pos < 170:
16         pos -= 85
17         return (0, 255 - pos * 3, pos * 3)
18     pos -= 170
19     return (pos * 3, 0, 255 - pos * 3)
20
21 def rainbow_cycle(wait):
22     for j in range(255):
23         for i in range(n):
24             rc_index = (i * 256 // n) + j
25             np[i] = wheel(rc_index & 255)
26         np.write()
27         sleep_ms(wait)
28
29 while not bp.value():
30     rainbow_cycle(1)

```

Mise en pratique : 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.

Deuxième partie

Structures de code, notions temporelles

8 Découverte de la carte ESP ENIM

Autour du microcontrôleur ESP32, j'ai développé cette carte (voir figure 17 page 41) 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.

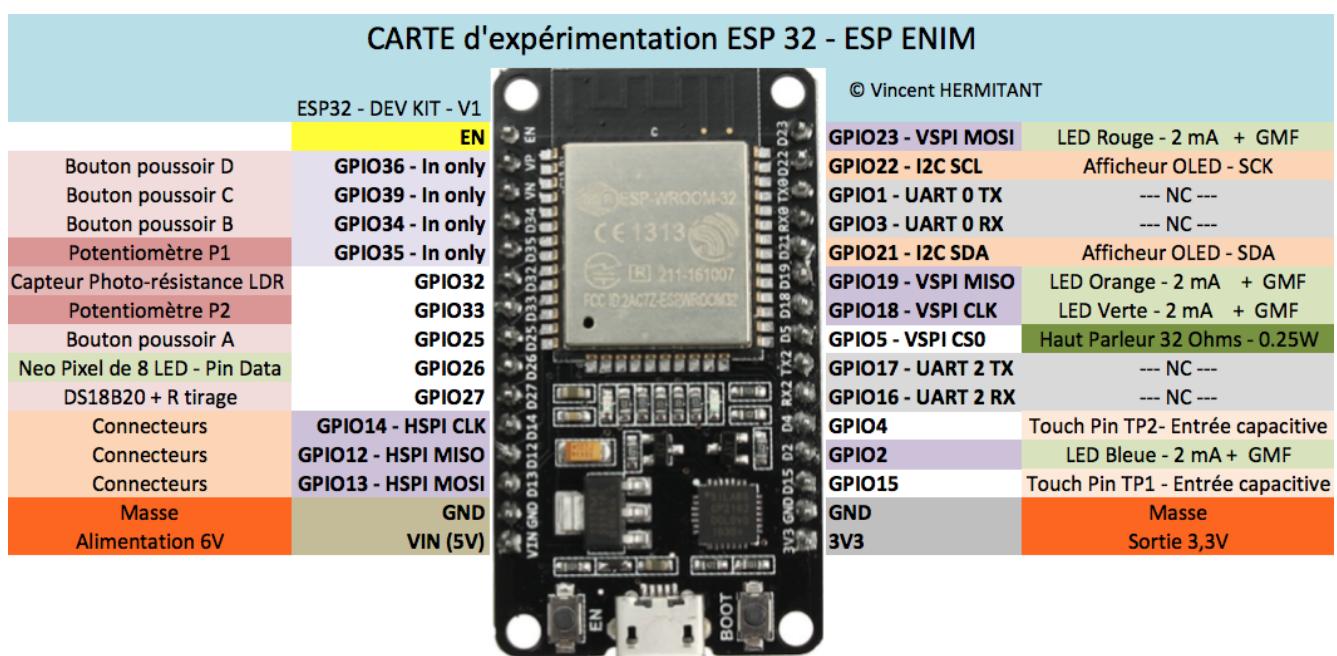


FIGURE 17 – Implantation des composants externes à l'ESP32 sur la carte ESP ENIM

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 suffira si besoin de le rajouter via les connecteurs Grove sur les sorties disponibles.

9 Lecture sur les entrées numériques et capacitatives

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.

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 écriture pour la même 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.

9.2 Fonction de comptage des appuis sur un poussoir

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.

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

9.2.1 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'état 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.

9.2.2 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(0.02)                # delai pour supprimer les rebonds du bp
```

►Q.87: Vérifier le fonctionnement.

►Q.88: Changer le fonctionnement pour compter (et inverser aussi la LED) lors du relâchement du poussoir. Attention, vous risquez de faire une solution qui compte 1 avant le premier relâchement.

►Q.89: 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 «bloquant» est rarement intéressant, sauf si notre programme n'a que cette opération de comptage à réaliser, ce qui est rare.

9.2.3 Solution de comptage NON bloquante, par détection de front

Pour nous défaire 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 :

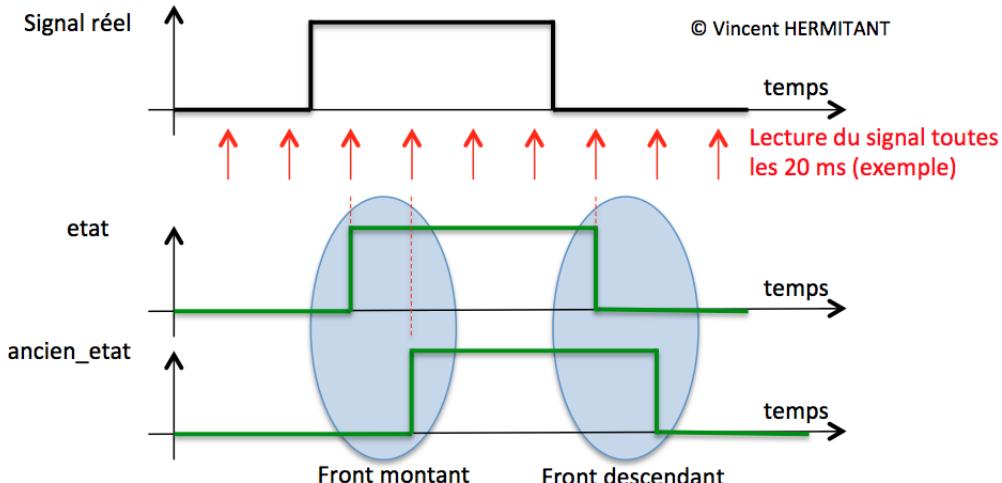


FIGURE 18 – Gestion des fronts pour éviter d'avoir une structure bloquante

- Toutes les 20 ms (c'est un exemple, un choix), une lecture de l'état du poussoir est effectuée. 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.

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

```

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.

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.90: En déplaçant juste l'instruction `not` en ligne 11, changer le fonctionnement pour compter (et inverser la LED) lors des fronts descendants.

►Q.91: Changer encore le fonctionnement pour compter lors des fronts montants ET des front descendants.

►Q.92: Incrémenter un deuxième compteur lors de l'appui sur un deuxième poussoir.

9.2.4 Solution de comptage NON bloquante, par utilisation des «Thread»

L'utilisation des «Thread» permet d'obtenir un fonctionnement en multitâche. Nous voyons dans l'exemple suivant que nous utilisons un compteur bloquant, et en parallèle nous faisons clignoter une LED, sans être empêché par la partie bloquante du compteur.

34 – comptage-thread.py - Compter les appuis - NON Bloquant

```

1 from machine import Pin
2 from time import sleep
3 import _thread
4
5 led_bleue = Pin(2, Pin.OUT)
6 bpA = Pin(25, Pin.IN)
7
8 led_orange = Pin(19, Pin.OUT)
9
10 def comptage():          # fonction de comptage (bloquant)
11     compt = 0
12     while True :
13         while not bpA.value():    # attente de l'appui
14             pass                 # ne fait rien
15         compt = compt + 1       # incrementation compteur
16         print("Compteur du bpA :",compt)
17         led_bleue.value(not led_bleue.value()) # inversion etat de la LED
18         sleep(.02)
19         while bpA.value():       # attente du relachement
20             pass
21         sleep(.02)
22
23 def clignotement():      # fonction de clignotement (bloquante)
24     while True:
25         led_orange.value(not led_orange.value()) # inversion etat de LED
26         sleep(.5)
27
28 _thread.start_new_thread(clignotement, ())
29 _thread.start_new_thread(comptage, ())

```

9.3 Les entrées capacitifs

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 capacitatives. 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.

9.3.1 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 :

35 – 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.93: Vérifier le bon fonctionnement. Noter les valeurs relevées lorsque l'on touche, et lorsque l'on ne touche pas la «Touch Pin».

►Q.94: 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.

9.3.2 Solution de comptage bloquante

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

36 – 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
```

On souhaiterait, en même temps que le comptage des appuis sur le TouchPad, lire l'état d'un bouton poussoir «bpA» pour compter les appuis et inverser l'état d'une LED à chaque appui, comme fait précédemment, et faire clignoter une LED à la fréquence de 1 Hz en même temps. Il est possible de réaliser cette opération avec les thread, simplement en récupérant l'exemple précédent «comptage-thread.py», et en ajoutant un thread.

►Q.95: Réaliser cela.

9.3.3 Solution de comptage NON bloquante, par détection de front

Comme avec les boutons pousoirs, 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.

37 – touchpad-comptage-ok.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.96: 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).

►Q.97: Connecter un fil conducteur sur la borne 4 de la carte, pour vérifier le même bon fonctionnement.

9.4 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.98: 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 :

9.4.1 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.

9.4.2 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 la vitesse ou la position de l'arbre moteur, 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 $250\ \mu s$, 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 1600 Hz et nous avons choisi de faire les mesures à la fréquence de 4000 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 μs /pulse	315,5 μs /mesure	250 μs /mesure

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

9.4.3 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 : $f.\text{éch.} > 3\ 200\ Hz$. Donc 4000 Hz convient bien ; mais 10 000 convient très bien aussi ; il est seulement inutile d'augmenter la charge du calcul.

9.4.4 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 ($1/0.02$). ainsi, nous pourrions presser et relâcher le poussoir 25 fois par seconde avant d'atteindre la limite de cette lecture. Nous serons sans doute toujours en dessous de cette valeur de 25 appuis par seconde.

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 procéder : par comptage de boucle puis par mesure de temps absolu (avec la méthode ticks) ; pour chacun des deux cas, nous verrons une solution bloquante (à l'aide des attentes dans des boucles While), et une 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 Solutions 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.

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

```
1 from machine import Pin
2 from time import sleep
```

```

3  bpB = Pin(34, Pin.IN)
4  compt_temps = 0
5
6
7  for t in range(5):          # on fait juste 5 cycles appui/relachement
8      while not bpB.value():  # on attend l'appui sur le bp
9          sleep(.02)         # on attend 20 ms, 50 boucles/s suffisant
10     compt_temps = 0
11     while bpB.value():    # on attend l'appui sur le bp
12         compt_temps += 1   # on incremente toutes les 20 ms
13         sleep(.02)        # delai entre deux boucles
14     print("Compteur : {0:2d}, soit : {1:2f} s".\
15           format(compt_temps, compt_temps/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.99: Tester le bon fonctionnement.

On souhaiterait, en même temps que la mesure de temps, lire l'état d'un bouton poussoir «bpA» pour compter les appuis et inverser l'état d'une LED à chaque appui, comme fait précédemment, et faire clignoter une LED à la fréquence de 1 Hz en même temps. Il est possible de réaliser cette opération avec les thread, simplement en récupérant l'exemple précédent «comptage-thread.py», et en ajoutant un thread.

►Q.100: Réaliser cela.

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 évènement, comme dans l'exemple précédent.

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

```

1  from machine import Pin
2  from time import sleep
3
4  bpB = Pin(34, Pin.IN)
5  compt_temps = 0
6  ancien_etat = bpB.value()                      # initialisation ancien_etat
7
8  for t in range(500):                            # on fait 500 cycles * 0.02 s = 10 s
9      etat = bpB.value()                          # lecture etat
10     if etat:                                    # tant qu'on appuie...
11         compt_temps += 1                         # ... on incremente
12     if ancien_etat and not etat:    # et si front descendant ! ...
13         print("Compteur : {0:2d}, soit : {1} s".\
14             format(compt_temps, compt_temps /50))
15     compt_temps = 0                           # on remet a 0
16     ancien_etat = etat                      # memorisation etat boucle avant
17     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.101: Tester le bon fonctionnement.

On souhaiterait, en même temps que la mesure de temps (non bloquante), faire clignoter une LED à la fréquence de 1 Hz en même temps. Il est possible de réaliser cette opération sans utiliser les thread.

►Q.102: Ajouter cette LED qui clignote.

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.

40 – 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.103: 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`.

41 – 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 bpB = Pin(34, Pin.IN)
6
7 for t in range(5):          # on fait juste 5 cycles appui/relachement
8     while not bpB.value():   # on attend l'appui sur le bp
9         pass                 # pas besoin de tempo
10    start = ticks_ms()      # memorisation de start lors du front montant
11    while bpB.value():       # on attend l'appui sur le bp
12        pass                 # pas besoin de tempo
13    stop = ticks_ms()       # stop sur front montant 2
14    delta = stop - start
15    print("Temps d'appui : {0} ms".format(delta))

```

On souhaiterait, en même temps que la mesure de temps bloquante, lire l'état d'un bouton poussoir «bpA» pour compter les appuis et inverser l'état d'une LED à chaque appui, comme fait précédemment, et faire clignoter une LED à la fréquence de 1 Hz en même temps. Il est possible de réaliser cette opération avec les thread, simplement en récupérant l'exemple précédent «comptage-thread.py», et en ajoutant un thread.

►Q.104: Réaliser cela.

10.2.3 Solution NON bloquante (détection de fronts)

42 – 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 : {} 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.105: Ajouter ce code.

10.3 Application - Mesure de temps entre deux évènements

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 le problème est juste de mesurer le temps de passage de l'objet entre les 2 capteurs, et qu'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.

43 – 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 cont = 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.106: Vérifier le bon fonctionnement.

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

►Q.107: 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 $1\ \mu s$, 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 $1\ \mu s$; 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.

44 – 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.108: Vérifier le bon fonctionnement.

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

►Q.109: 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.

►Q.110: Vérifier que si le bpA est déjà appuyé lorsqu'on lance l'exécution, le code ne fonctionne pas. Nous corrigeron cela ci-après.

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.

45 – 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'état BAS de A")
11 while bpA.value():      # attente a l'état haut
12     pass
13
14 print("Attente du front montant de bpA")
15 while not bpA.value():  # attente a l'état bas
16     pass
17 start = ticks_ms()      # start sur front montant 1
18 led.value(1)
19
20 print("Attente de l'état BAS de B")
21 while bpB.value():      # attente a l'état haut
22     pass
23
24 print("Attente du front montant de bpB")
25 while not bpB.value():  # attente a l'état 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.111: Vérifier le bon fonctionnement, même si le bpA (ou bpB) est pressé avant de lancer le code.

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.

46 – 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 fréquence à 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        while not bp.value(): pass # attente de l'appui
12        while bp.value(): pass # attente du relachement
13
14        for i in range(0, 1023, 10): # allumage de la LED
15            led.duty(correction(i, 3.5)) # utilisation fonction correction
16            sleep_ms(10)
17
18        while not bp.value():
19            pass # attente de l'appui
20        while bp.value():
21            pass # attente du relachement
22
23        for i in range(0, 1023, 10): # extinction de la LED
24            led.duty(correction(1023 - i, 3.5))
25            sleep_ms(10)

```

►Q.112: Vérifier le bon fonctionnement, et vérifier que c'est bien lors du relâchement que l'allumage et l'extinction ont lieu.

►Q.113: 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 terminé. 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 envoie 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.

47 – 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.114: Vérifier le bon fonctionnement.

►Q.115: Comprendre qu'il serait difficile, à partir de ce code, et sans les «Thread», 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.116: Comprendre qu'il serait difficile également, sans les «Thread», 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 n'ait pu le détecter.

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.

48 – 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.117: 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.118: 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 :`

49 – 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.119: Tester ce code.

►Q.120: Mesurer le temps réel que met la boucle pour faire les 4000 itérations.

Pour cela, on utilise la fonction `ticks_ms()` mais il diffère des deux correspondant 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 μ s environ ; en effet, les autres opérations de la boucle ont besoin d'un temps aussi, non négligeable, de l'ordre de quelques μ s, pour s'exécuter. Donc la boucle ne se fait pas en 1 000 μ s comme je souhaite, mais un peu plus, et peut être environ 1 050.

►Q.121: Modifier la valeur du délai (qui est de 1000 μ s), pour corriger ce décalage.

Où toutes les lignes sont commentées.

Mise en pratique : Serrure à code

►Q.122: Mémoriser dans une liste l'ordre dans lequel on appuie sur deux boutons poussoirs.

►Q.123: Lorsque une liste est reconnue (une liste que vous aurez déclaré comme étant le code attendu), 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.

12 Les «interruptions»

L'utilisation des «interruptions» est incontournable lors de la création de programmes.

12.1 Principe des interruptions

12.1.1 Qu'est ce que c'est ?

Une interruption est le fait de pouvoir détourner la boucle principale (du programme) de son déroulement normal. Ce détournement est provisoire, et a lieu lors de l'apparition d'un évènement *autorisé et prioritaire*, par exemple l'appui sur un bouton poussoir.

Lorsqu'un tel évènement est détecté, le micro-contrôleur exécute la fonction d'interruption prévue pour gérer cet évènement, puis retourne au programme principal. Le traitement de l'interruption doit être le plus court possible, afin de perturber le moins possible le déroulement normal du programme principal. Ainsi, lors du traitement de l'interruption, il n'y aura *aucune temporisation ou instruction longue ou bloquante*.

12.1.2 Utilisation des interruptions

L'utilisation des interruptions permet de laisser tourner le programme normalement, alors que ce dernier reste capable de répondre à des événements asynchrones (impromptus) en provenance du monde extérieur, ou intérieur (via un Timer). Ainsi, il n'est alors plus besoin de surveiller l'état d'une entrée en permanence par scrutation («polling»), pour savoir par exemple si quelqu'un a appuyé sur le poussoir.

Le mécanisme des interruptions est donc utilisé par exemple pour résoudre le problème de lecture des entrées, sans risque de louper un changement d'état, en associant le changement d'état sur une entrée à une interruption.

Les interruptions sont très utiles pour faire en sorte que les choses se fassent de manière «automatique», et peuvent permettre par exemple de détecter un front très facilement.

12.2 Exemples de code gérant les interruptions

12.2.1 Principe de l'interruption

Lors de l'interruption, la LED bleue change d'état, et nous affichons la pin qui a généré l'interruption. Ce paramètre «pin» est fourni à la fonction `gestion_interrupt1()` par la méthode `irq` appliquée à l'objet `bpA` en ligne 15.

La boucle principale est très simple, elle comporte 4 lignes (18 à 21). Pourtant, en tache de fond, la pin 25 est «surveillée», et dès qu'elle passe à l'état haut, une interruption est générée. Cette surveillance ne consomme aucune ressource du micro-contrôleur.

Le comptage, l'affichage, et l'attente de 0,25 s ne sont aucunement perturbés par l'interruption.

50 – interruption1.py - Principe de l'interruption

```
1 from machine import Pin
2 from time import sleep
3
4 # le parametre pin est fourni par la methode irq lors de l'appel
5 # ce parametre est la pin qui cause l'interruption
6 def gestion_interrupt(pin):          # fonction d'interruption
7     led_bleue.value(not led_bleue.value())    # bascule l'état de LED bleue
8     print('Interruption causee par :', pin)  # affiche la pin en cause
9
10 bpA = Pin(25, Pin.IN)
11 led_bleue = Pin(2, Pin.OUT)
12
13 # configure l'interruption sur le front montant de bpA,
14 # et renvoie vers la fonction gestion_interrupt pour son traitement
15 bpA.irq(trigger=Pin.IRQ_RISING, handler = gestion_interrupt)
16
17 compt = 0
18 while True:                      # boucle principale qui compte les 0.25 s
19     compt = compt + 1
20     print(compt)
21     sleep(0.25)
```

12.2.2 Amélioration de la gestion de l'interruption

Lors de l'interruption, notre programme principal est détourné de son déroulement normal, et va sans délai exécuter la fonction d'interruption.

La règle étant de raccourcir au maximum le traitement de cette interruption, nous allons reporter la fonction d'affichage `print` dans la boucle principale car nous avons déjà vu qu'elle nécessite un temps non négligeable, en comparaison avec d'autres instructions telles qu'un calcul numérique ou une affectation de variable.

Voici alors la manière de faire : nous mémorisons (ligne 9) le fait qu'il y a eu une interruption en utilisant la variable globale `inter` qui passe à l'état `True`, puis nous testons cette variable dans la boucle principale (lignes 25-27) pour gérer l'affichage, et remettre la variable `inter` à `False` (ligne 28).

Cet affichage attendra au maximum 0,25s pour être exécuté car il est dans la boucle principale cadencée à 0,25 s par cycle ; quant au basculement de la LED, il est réalisé immédiatement car il est placé dans la fonction de traitement de l'interruption.

51 – interruption2.py - Amélioration de la gestion de l'interruption

```
1 from machine import Pin
2 from time import sleep
3
4 # le parametre pin est fourni par la methode irq lors de l'appel
5 # ce parametre est la pin qui cause l'interruption
6 def gestion_interrupt(pin):          # fonction d'interruption
7     led_bleue.value(not led_bleue.value())    # bascule l'état de LED bleue
8     global inter                         # définition de la variable globale
9     inter = True                        # la variable globale est mise à True
10    global interrupt_pin                # définition de la variable globale
11    interrupt_pin = pin                # correspond à la pin d'interruption
12
13 bpA = Pin(25, Pin.IN)
14 led_bleue = Pin(2, Pin.OUT)
15
16 # configure l'interruption sur le front montant de bpA,
17 # et renvoie vers la fonction gestion_interrupt pour son traitement
18 bpA.irq(trigger=Pin.IRQ_RISING, handler = gestion_interrupt)
19
20 compt = 0
21 inter = False
```

```

22
23 while True:                      # boucle principale qui compte les 0.25 s
24     compt = compt + 1
25     print(compt)
26     if inter :
27         print('Interruption causee par :', interrupt_pin)
28         inter = False
29     sleep(.25)

```

12.2.3 L'interruption allume la LED durant 3 secondes

Maintenant, nous réutilisons une partie de ce que nous avions fait avec le temporisateur d'éclairage au chapitre 5.3 page 32, mais en utilisant cette fois-ci l'interruption. Nous voyons (ligne 30) que nous mettons la variable `temps` à 12, cette variable est décrémentée de 1 (ligne 26) à chaque itération de la boucle principale, et la LED est allumée si le temps est supérieur à 0 (ligne 27). Il n'y a plus de basculement de la LED dans la fonction de traitement de l'interruption.

52 – interruption3.py - L'interruption allume la LED durant 3s

```

1 from machine import Pin
2 from time import sleep
3
4 # le parametre pin est fourni par la methode irq lors de l'appel
5 # ce parametre est la pin qui cause l'interruption
6 def gestion_interrupt(pin):           # fonction d'interruption
7     global inter                      # definition de la variable globale
8     inter = True                      # la variable globale est mise a True
9     global interrupt_pin              # definition de la variable globale
10    interrupt_pin = pin              # correspond a la pin d'interruption
11
12 bpA = Pin(25, Pin.IN)
13 led_bleue = Pin(2, Pin.OUT)
14
15 # configure l'interruption sur le front montant de bpA,
16 # et renvoie vers la fonction gestion_interrupt pour son traitement
17 bpA.irq(trigger=Pin.IRQ_RISING, handler = gestion_interrupt)
18
19 compt = 0
20 inter = False
21 temps = 0
22
23 while True:                      # boucle principale qui compte les 0.25 s
24     compt = compt + 1
25     print(compt)
26     temps = temps - 1
27     led_bleue.value(temps > 0)    # allume LED si temps > 0
28     if inter :
29         print('Interruption causee par :', interrupt_pin)
30         temps = 12                 # temps = 12 pour duree 3 s d'allumage LED
31         inter = False
32     sleep(.25)

```