BBC micro:bit MicroPython Documentation

Version 0.5.0

Multiple authors

sept. 04, 2018

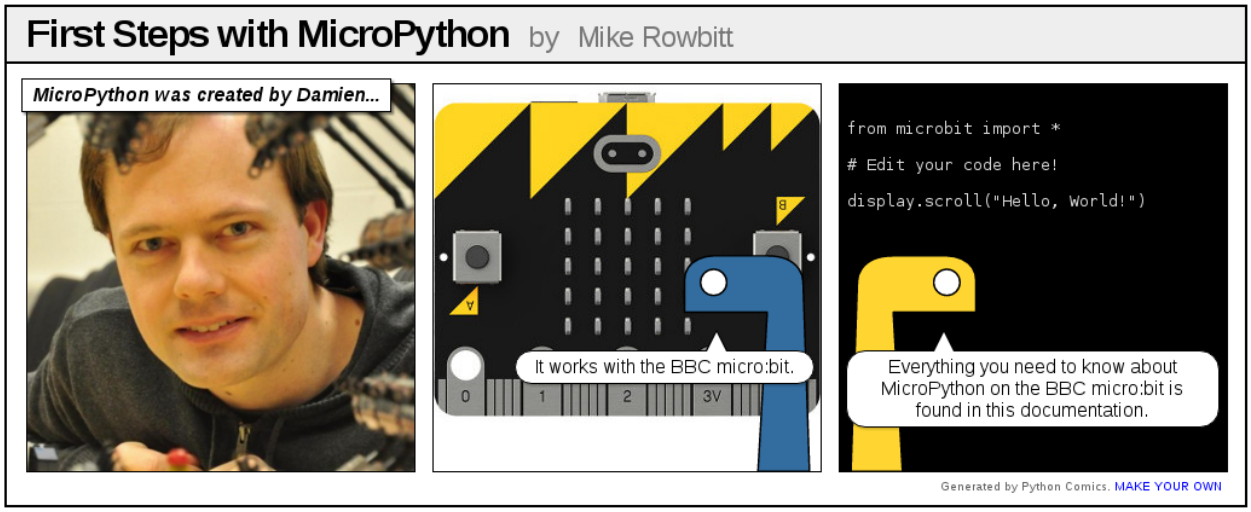
Tutorials

Bienvenu !

Le BBC micro :bit est un petit dispositif informatique pour les enfants. L'un des langages qu'il comprend est le langage de programmation poopulaire Python. La version utilisée sur le BBC micro :bit est appelée MicroPython.

Cette documentation comprend des leçons pour les enseignants et une documentation de l'API pour les développeurs (regarde l'index sur la gauche). Nous espérons que tu aimeras développer en MicroPython pour le BBC micro :bit.

Si tu es un programmeur débutant, un enseignant ou si tu ne sais pas par quoi commencer, vas voir les tutoriels.



Note : Ce projet est en cours de développement. Merci d'aider les autre développeurs en ajoutant des astuces, des guides et des questions/réponses à ce document. On te remercie d'avance !

Les projets lié au MicroPython sur le BBC micro :bit comprennent :

- Mu - un éditeur de code simple pour les enfants, les enseignants et les programmeurs débutants. C'est probablement la façon la plus simple de programmer en MicroPython sur le BBC micro :bit.
- uFlash - un outil en ligne de commande pour flasher des script Python directement sur le BBC micro :bit.

CHAPITRE 1

Introduction

Nous suggérons de télécharger et d'utiliser l'éditeur mu pour travailler avec ces tutoriels. Les instructions pour télécharger et installer Mu se trouvent sur son site. Tu seras peut-être amené à installer un pilote de périphérique selon ton système d'exploitation (les instructions sont sur le site web).

Mu fonctionne sous Windows, OSX et Linux.

Une fois que Mu est installé, connecte ton micro :bit à ton ordinateur avec un câble USB.

Ecris ton script dans la fenêtre d'édition et click sur le bouton « Flash » pour le transférer dans le micro :bit. Si ça ne marche pas, vérifie que ton micro :bit apparait comme un stockage USB dans ton explorateur de fichiers.

1.1 Hello, World !

La manière traditionnelle de commencer la programmation dans un nouveau langage est de demander à votre ordinateur de dire « Hello, World ! ».

C'est facile avec MicroPython :

```
from microbit import *
display.scroll("Hello, World!")
```

Chaque ligne fait quelque chose de spécial. La première ligne :

```
from microbit import *
```

... demande à MicroPython de récupérer tout ce dont il a besoin pour fonctionner avec le micro :bit BBC. Tout cela se trouve dans un module appelé `microbit``(un module est une bibliothèque de code préexistant). Quand tu ``import` quelque chose, tu dis à MicroPython que tu veux l'utiliser, et `*` est la façon qu'a Python de dire « tout » . Donc, en français, « je veux pouvoir tout

utiliser depuis la bibliothèque de code microbit ». `from microbit import *`

La deuxième ligne :

```
display.scroll("Hello, World!")
```

... indique à MicroPython d'utiliser l'affichage pour faire défiler la chaîne de caractères « Hello, World ! ». La partie `display` de cette ligne est un *objet* du module `microbit` qui représente l'affichage physique du périphérique (on dit « objet » au lieu de « chose », « quoi » ou « doodah »). Nous pouvons dire à l'affichage de faire les choses avec un point `.` suivi de ce qui ressemble à une commande (en fait, c'est quelque chose que nous appelons une *méthode*). Dans ce cas, nous utilisons la méthode `scroll`. Puisque `scroll` doit savoir quels caractères faire défiler sur l'affichage physique, nous les spécifions entre guillemets (`"`) entre les parenthèses (`(` et `)`). Ce sont les *arguments*. Ainsi,‘‘display.scroll(« Hello, World ! »)‘‘ signifie, en français, « Je veux que tu utilises l'écran pour faire défiler le texte “Hello, World !” ». Si une méthode n'a pas besoin d'arguments on utilisant des parenthèses vides comme ceci : `()`.

Copie le code « Hello, World ! » dans ton éditeur et flash-le sur le périphérique. Peux-tu trouver comment changer le message ? Peux-tu le faire dire bonjour ? Par exemple, je pourrais dire « Bonjour, Nicolas ! ». Voici un indice, tu dois changer l'argument de la méthode de défilement.

Avertissement : Cela peut ne pas fonctionner. :-)

C'est là que les choses amusantes commencent et que MicroPython essaie d'être utile. S'il rencontre une erreur, il fera défiler un message utile sur l'écran du micro-bit. Si c'est le cas, il t'indiquera le numéro de ligne où l'erreur peut être trouvée.

Python s'attend à ce que tu tapes EXACTEMENT la bonne chose. Ainsi, par exemple, `Microbit`, `microbit` et `microBit` sont toutes des choses différentes pour Python. Si MicroPython se plaint à propos d'un `NameError` c'est probablement parce que tu as tapé quelque chose de manière incorrecte. C'est comme la différence entre faire référence à « Nicholas » et « Nicolas ». Ce sont deux personnes différentes mais leurs noms sont très similaires.

Si MicroPython se plaint de `SyntaxError` tu as simplement tapé du code d'une manière que MicroPython ne peut pas comprendre. Vérifie que qu'il ne manque pas de caractères spéciaux comme `"` ou `:`. C'est comme mettre un point au milieu d'une phrase. Il est difficile de comprendre exactement ce que tu veux dire.

Ton microbit peut cesser de répondre : tu ne peux plus lui envoyer un nouveau code ou entrer des commandes dans le REPL. Si cela se produit, essaye de le rallumer. En d'autres termes, débranche le câble USB (et le câble de la batterie s'il est connecté), puis rebranche le câble. Tu devras peut-être également quitter et redémarrer ton éditeur de code.

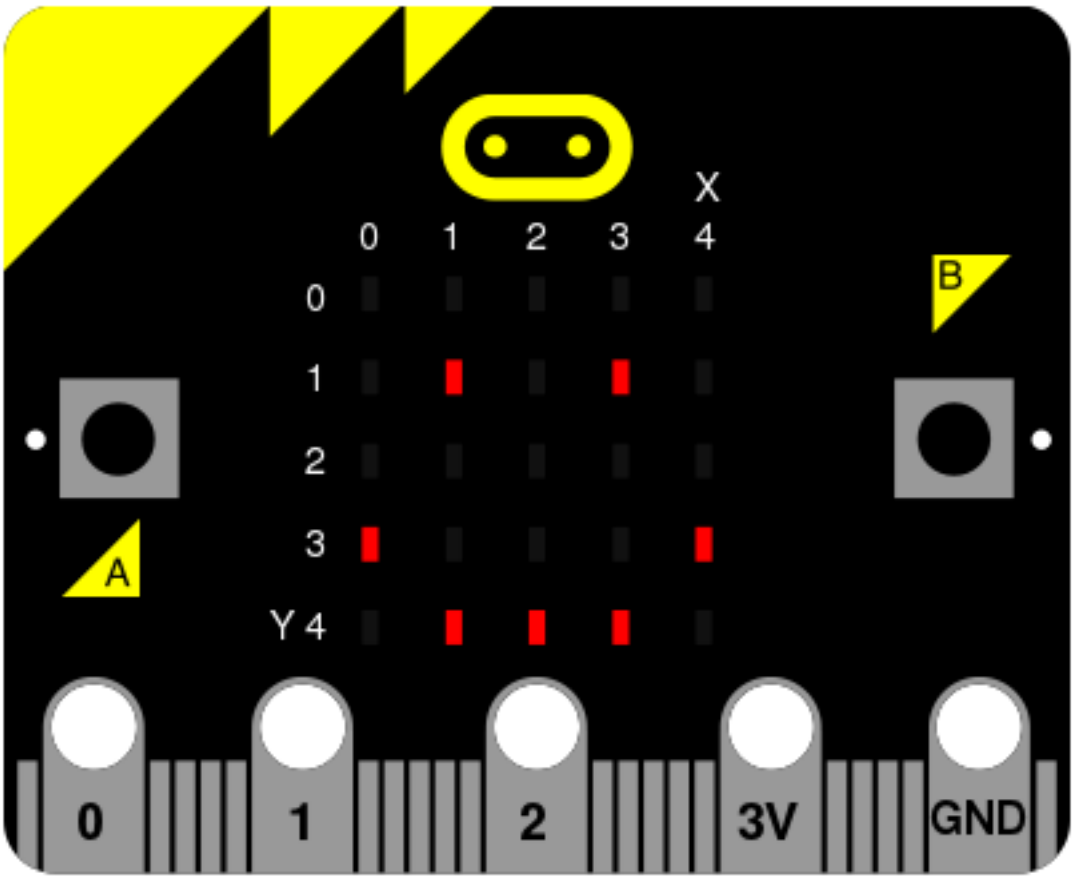
1.2 Images

MicroPython est à peu près aussi bon en art que tu pourrais l'être si la seule je que tu avais était une grille de LED de 5x5. MicroPython te donne pas mal de contrôle sur l'affichage de façon à ce que tu puisses créer toute sorte d'effets intéressants.

MicroPython est fourni avec beaucoup d'images intégrées à montrer sur l'affichage. Par exemple, pour que l'appareil ait l'air heureux, tape :

```
from microbit import *
display.show(Image.HAPPY)
```

J'imagine que tu peux te rappeler ce fait la première ligne. La seconde utilise l'objet `display` pour `show` (montrer) une image intégrée. L'image *heureux* que nous voulons afficher fait partie del'objet `Image` et est appelée `HAPPY`. On écrit `show` pour l'utiliser en la mettant entre parenthèses (`(` et `)`).



Voici la liste des images intégrées :

- `Image.HEART`
- `Image.HEART_SMALL`
- `Image.HAPPY`
- `Image.SMILE`
- `Image.SAD`
- `Image.CONFUSED`
- `Image.ANGRY`
- `Image.ASLEEP`
- `Image.SURPRISED`
- `Image.SILLY`
- `Image.FABULOUS`
- `Image.MEH`
- `Image.YES`
- `Image.NO`
- `Image.CLOCK12`, `Image.CLOCK11`, `Image.CLOCK10`, `Image.CLOCK9`, `Image.CLOCK8`, `Image.CLOCK7`, `Image.CLOCK6`, `Image.CLOCK5`, `Image.CLOCK4`, `Image.CLOCK3`, `Image.CLOCK2`, `Image.CLOCK1`
- `Image.ARROW_N`, `Image.ARROW_NE`, `Image.ARROW_E`, `Image.ARROW_SE`, `Image.ARROW_S`, `Image.ARROW_SW`, `Image.ARROW_W`, `Image.ARROW_NW`
- `Image.TRIANGLE`
- `Image.TRIANGLE_LEFT`
- `Image.CHESSBOARD`
- `Image.DIAMOND`
- `Image.DIAMOND_SMALL`

- `Image.SQUARE`
- `Image.SQUARE_SMALL`
- `Image.RABBIT`
- `Image.COW`
- `Image.MUSIC_CROTCHET`
- `Image.MUSIC_QUAVER`
- `Image.MUSIC_QUAVERS`
- `Image.PITCHFORK`
- `Image.XMAS`
- `Image.PACMAN`
- `Image.TARGET`
- `Image.TSHIRT`
- `Image.ROLLERSKATE`
- `Image.DUCK`
- `Image.HOUSE`
- `Image.TORTOISE`
- `Image.BUTTERFLY`
- `Image.STICKFIGURE`
- `Image.GHOST`
- `Image.SWORD`
- `Image.GIRAFFE`
- `Image.SKULL`
- `Image.UMBRELLA`
- `Image.SNAKE`

Il y en a pas mal ! Pourquoi ne pas modifier le code qui donne à micro :bit l'air heureux pour voir à quoi ressemble les autres images intégrées ? (Il te suffit de remplacer `Image.HAPPY` par l'une des images intégrées de la liste ci-dessus.)

1.2.1 Images personnelles

Bien sûr, tu veux que ta propre image s'affiche sur le micro :bit, non ?

C'est facile.

Chaque pixel LED sur l'affichage physique peut prendre une parmi dix valeurs. Si un pixel prend la valeur `0` (zéro) c'est qu'il est éteint. Litéralement, il a une luminosité de zéro.En revanche, s'il prend la valeur `9` il est à la luminosité maximale. Les valeurs de `1` à `8` représentent des niveaux de luminosité entre éteint (`0`) et « à fond » (`9`).

Muni de ces informations, il est possible de créer une nouvelle image comme ça :

```
from microbit import *

boat = Image("05050:"
             "05050:"
             "05050:"
             "99999:"
             "09990")

display.show(boat)
```

(Une fois lancé, l'appareil devrait afficher un bateau à voile dont le mât est moins brillant que la coque).

As-tu compris comment dessiner une image ? As-tu remarqué que chaque ligne de l'affichage physique est représentée par une line de nombres se terminant par `:` et entourée de `"` guillemets doubles ? Chaque nombre indique une luminosité. Il y a cinq lignes de cinq nombres donc il est possible de spécifier la luminosité individuelle de chacune des cinq LED sur chacune des cinq lignes sur l'affichage physique. C'est comme ça qu'on créé une image.

Simple !

En fait, tu n'as pas besoin d'écrire tout ça sur plusieur lignes. Si tu te sent capable de garder la trace de chaque ligne, tu peux le ré-écrire comme ça :

```
boat = Image("05050:05050:05050:99999:09990")
```

1.2.2 Animation

Les images statiques sont amusantes, mais c'est encore plus amusant de les faire bouger. C'est aussi incroyablement facile à faire avec MicroPython ~ il suffit d'utiliser une liste d'images.

Voici une liste de courses :

```
Oeufs
Bacon
Tomates
```

Et voici comment la représenter en Python :

```
courses = ["Oeufs", "Bacon", "Tomates" ]
```

J'ai simplement créé une liste nommée `courses` et elle contient trois éléments. Python sait que c'est une liste car elle est contenue dans des crochets (`[` et `]`). Les éléments de la liste sont séparés par des virgules (`,`) et dans cet exemple les éléments sont trois chaînes de caractères : `"Oeufs"`, `"Bacon"` et `"Tomates"`. Nous savons que ce sont des chaînes de caractères parce qu'elles sont contenues des guillemets `"`.

Tu peux stocker n'importe quoi dans une liste en Python. Voici une liste de nombres :

```
premiers = [2, 3, 5, 7, 11, 13, 17, 19]
```

Note : Les nombres n'ont pas besoin d'être entre guillemets puisqu'ils représentent une valeur (plutôt qu'une chaînes de caractères). C'est la différence entre `2` (la valeur numérique 2) et `"2"` (le caractères, le chiffre, qui représente le nombre 2). Ne t'inquiète pas si ce n'est pas très clair pour l'instant. Tu t'y habitueras bientôt.

Il est même possible de stocker des choses de catégories différentes dans une même liste :

```
list_variee = ["salut!", 1.234, Image.HAPPY]
```

As-tu remarqué le dernier élément ? C'était une image !

On peut dire à MicroPython d'animer une liste d'images. Par chance npus avons deux listes d'images déjà prêtes. Elles s'appellent `Image.ALL_CLOCKS` et `Image.ALL_ARROWS` :

```
from microbit import *

display.show(Image.ALL_CLOCKS, loop=True, delay=100)
```

Comme avec une seule image, on utilise `display.show` pour la montrer sur l'affichage du matériel. Mais ici on indique à MicroPython d'utiliser `Image.ALL_CLOCKS` et il comprend qu'il doit montrer chaque image de la liste, l'une après l'autre. On indique aussi à MicroPython de parcourir la liste d'images en boucle (pour que l'animation se répète indéfiniment) en écrivant `loop=True`. De plus, nous lui indiquons que nous voulons un temps de 10 millisecondes entre chaque image avec l'argument `delay=100`.

Peux-tu trouver comment animer la liste `Image.ALL_ARROWS` ? Comment éviterais-tu de la parcourir en boucle indéfiniment ? (Indice : le contraire de `True` est `False` bien que la valeur par défaut de `loop` soit `False`) ? Peux-tu changer la vitesse de l'animation ?

Enfin, voici comment créer ta propre animation. Dans mon exemple, je vais faire couler mon bateau en bas de l'affichage :

```
from microbit import *

bateau1 = Image("05050:"
                "05050:"
                "05050:"
                "99999:"
                "09990")

bateau2 = Image("00000:"
                "05050:"
                "05050:"
                "05050:"
                "99999")

bateau3 = Image("00000:"
                "00000:"
                "05050:"
                "05050:"
                "05050")

bateau4 = Image("00000:"
                "00000:"
                "00000:"
                "05050:"
                "05050")

bateau5 = Image("00000:"
                "00000:"
                "00000:"
                "00000:"
                "05050")

bateau6 = Image("00000:"
                "00000:"
                "00000:"
                "00000:"
                "00000")

tous_les_bateaux = [bateau1,bateau2,bateau3,bateau4,bateau5,bateau6]
display.show(tous_les_bateaux, delay=200)
```

Voici comment le code marche :

- Je créé six images de `bateau` de la même façon que ce que j'ai décris au-dessus.
- Ensuite, je les mets dans une liste que j'appelle `tous_les_bateaux`
- Enfin, je demande `display.show` pour animer la liste avec un délai de 200 millisecondes
- Puisque je n'ai pas déclaré `loop=True`, le bateau ne coulera qu'une fois

(rendant ainsi mon animation scientifiquement correcte). :-)

Que voudrais-tu animer ? Peux-tu animer un effet spécial ? Comment ferais-tu un fondu d'image en sortie et en ouverture ?

1.3 Boutons

Jusqu'à maintenant nous avons créé du code qui fait faire quelque chose à l'appareil. C'est ce qu'on appelle une *sortie* ou *output*. Cependant, nous avons aussi besoin que l'appareil réagisse à quelque chose.C'est ce qu'on appelle des *entrées* ou *inputs*.

C'est facile à retenir : une sortie est ce que l'appareil fait ressortir vers le monde extérieur tandis qu'une entrée c'est qui provient du monde extérieur et entre dans l'appareil.

Les moyens les plus évidents d'entrées sur le micro :bit sont ses deux boutons, nommés `A` et `B`. D'une certaine façon, nous avons besoin que MicroPython réagisse à l'appui sur ces boutons.

C'est extrêmement simple :

```
from microbit import *

sleep(10000)
display.scroll(str(button_a.get_presses()))
```

Tout ce que fait ce script c'est de dormir pendant dix mille millisecondes, (c'est-à-dire 10 secondes) et ensuite de faire défiler le nombre d'appuis sur le bouton `A`. C'est tout !

Bien que ce soit un script plutôt inutile, il montre quelques nouvelles idées intéressantes :

1. La *fonction* `sleep` fait dormir le micro :bit pendant un certain nombre de millisecondes. Si tu veux une pause dans ton programme, c'est ce qu'il faut faire. Une *fonction* est comme une *méthode* mais elle n'est pas attachée par un point à un *objet*.
2. Il y un objet nommé `button_a` et il permet d'obtenir le nombre de fois où il a été pressé avec la *méthode* `get_presses`.

Puisque `get_presses` renvoie une valeur numérique et que `display.scroll` ne peut afficher que des caractères, nous devons convertir la valeur numérique en une chaîne de caractères. Nous le faisons avec la fonction `str` (qui un raccourci pour *string* qui signifie chaîne en anglais). Cette fonction converti son argument en une chaîne de caractères.

La troisième ligne est un peu comme un oignon. Si les parenthèses sont les couches de l'oignon alors tu remarqueras que `display.scroll` contient `str` qui lui-même contient `button_a.get_presses`. Python essaye d'interpréter d'abord ce qui se trouve le plus à l'intérieur, puis remonte les couches vers l'extérieur. On appelle ça en anglais *nesting* (imbrication)- qui est l'équivalent en programmation d'une poupée russe.

Supposons que tu as appuyé 10 fois sur le bouton. Voilà comment Python interprète ce qu'il se passe sur la troisème ligne :

Python voit la ligne complète et obtient la valeur de `get_presses` :

```
display.scroll(str(button_a.get_presses()))
```

Maintenant que Python sait combien d'appui sur le bouton il y eu, il converti la valeur numérique en une chaîne de caracères

```
display.scroll(str(10))
```

Enfin, Python sait ce qu'il doit faire défiler sur l'affichage :

```
display.scroll("10")
```

Même si ça à l'air d'être beaucoup de travail, MicroPython fait ça de façon extrêmement rapide.

1.3.1 Boucles événementielles

Tu auras souvent besoin que ton programme soit dans l'attente que quelque chose se produise. Pour ce faire, tu devras le faire « boucler » sur un morceau de code qui défini comment réagir à certains événements attendus comme l'appui sur un bouton.

Pour faire des boucles en Python on utilise le mot-clé `while` qui signifie *tant que*. Il vérifie si quelque chose est `True` c'est-à-dire *Vrai*. Tant que c'est le cas, il exécute un *bloc de code* appelé *corps* de la boucle. Lorsque ce n'est plus le cas, il sort de la boucle (en ignorant son corps) et le reste du programme peut continuer.

En Python, il est facile de définir un bloc de code. Disons que j'ai une liste de chose à faire écrite sur un bout de papier. Elle ressemble surement à ça

```
Courses
Réparer la gouttière
Tondre la pelouse
```

Si je veux rendre ma liste un peu plus précise, je peux écrire quelque chose comme ça

```
Courses:
    Oeufs
    Bacon
    Tomates
Réparer la gouttière:
    Emprunter l'échelle du voisin
    Trouver un marteau et des clous
    Rendre l'échelle
Tondre la pelouse:
    Vérifier qu'il n'y a pas de grenouille près de l'étang
    Vérifier le niveau d'essence dans la tondeuse
```

Il est évident que les tâches principales sont divisées en sous-tâches qui sont *indentées* en-dessous de la tâche principale à laquelle elles sont reliées. Ainsi `Oeufs`, `Bacon` et `Tomates` sont clairement reliés à `Courses`. En indentant les lignes, on permet facilement de voir en un coup d'oeil comment les différentes tâches sont reliées entres elles.

Là encore on parle de *nesting* (imbrication). On utilise l'imbrication pour définir des blocs de code comme ça

```
from microbit import *

```

(suite sur la page suivante)

(suite de la page précédente)

```
while running_time() < 10000:
    display.show(Image.ASLEEP)

display.show(Image.SURPRISED)
```

La fonction `running_time` renvoie le nombre de millisecondes depuis que l'appareil a démarré.

La ligne `while running_time() < 10000:` vérifie si le temps écoulé est inférieur à 10000 millisecondes (c'est-à-dire 10 secondes). Tant que c'est le cas, il affichera `Image.ASLEEP`. Remarque la façon dont le code est indenté en dessous de l'instruction `while` comme dans notre *liste de choses à faire*.

Evidemment, si le temps écoulé est supérieur ou égal à 1000 millisecondes alors l'affichage montrera `Image.SURPRISED`. Pourquoi ? Parce que la condition du `while` sera fausse (`running_time` ne sera plus `<10000`). Dans ce cas la boucle est terminée et le programme continue après le corps de la boucle `while`. Cela fait comme si l'appareil était endormi pendant 10 secondes avant de se réveiller avec un air surpris. Essaie-le !

1.3.2 Gérer un événement

Si nous voulons que MicroPython réagisse aux événement « *appui sur un bouton* » nous devrions le mettre dans une boucle infinie et vérifier si le bouton `is_pressed`.

Une boucle infinie est facile :

```
while True:
    # Faire des trucs
```

(Rappelle-toi, `while` vérifie si quelque chose est `True` pour déterminer si il doit exécuter son corps. Puisque `True` est évidemment `True` tout le temps, on obtient une boucle infinie !)

Faisons un cyber-animal très simple. Il est tout le temps triste sauf quand tu appuies sur le bouton `A`. Si tu appuies sur le bouton `B`, il meurt. (Je me rends compte que ce n'est pas un jeu très amusant, donc peut-être que tu peux trouver comment l'améliorer) :

```
from microbit import *

while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif button_b.is_pressed():
        break
    else:
        display.show(Image.SAD)

display.clear()
```

As-tu vu comment on vérifie quel bouton est pressé ? On utilise `if` (qui veut dire *si*), `elif` (qui veut dire *autre si*) et `else` (qui veut dire *sinon*). Ce sont des *instructions conditionnelles* et elles marchent comme ça :

```
if quelque chose est vrai (``True``):
    # fais un truc
elif autre chose est vrai (``True``):
    # fais un autre truc
else:
    # fais encore autre chose.
```

C'est très proche de l'anglais !

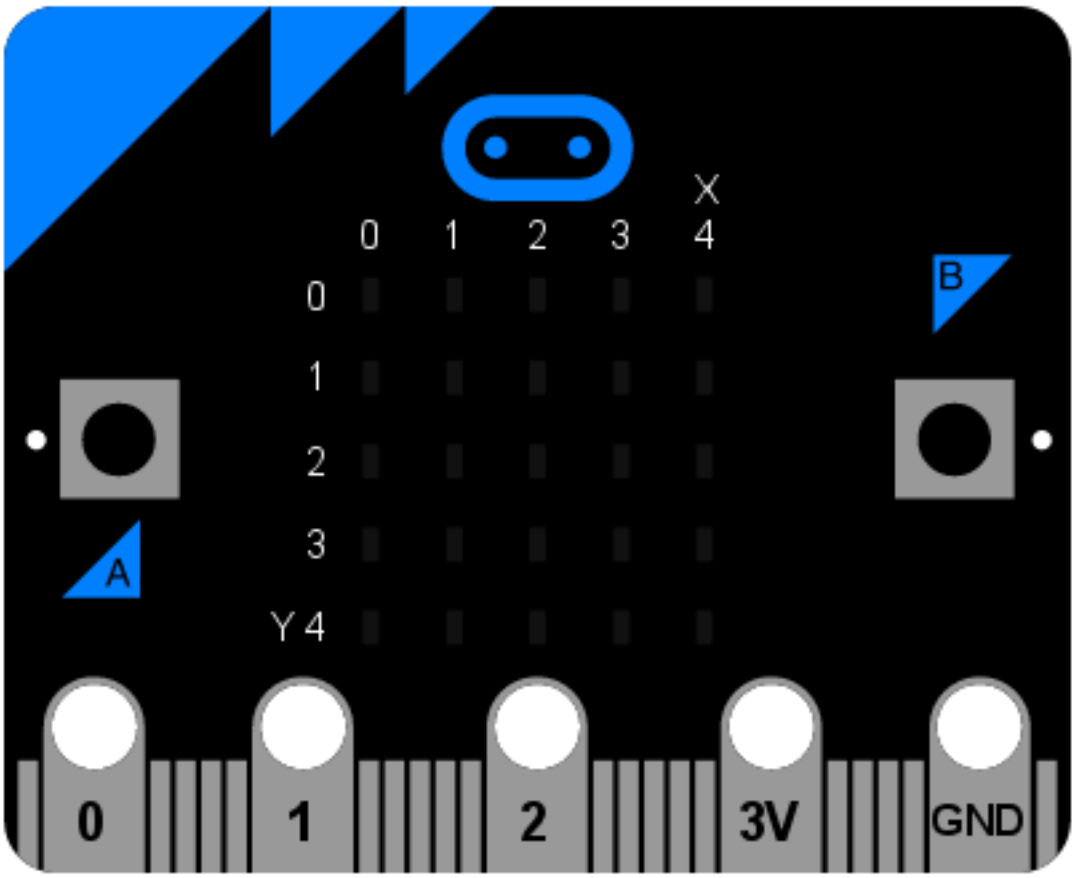
La méthode `is_pressed` ne renvoie que deux résultats possibles : `True` ou `False`. Si tu appuies sur le bouton, elle renvoie `True`, sinon elle renvoie `False`.Finalement, exprimé en français, le code ci-dessus dit : « Pour toujours, si le bouton A est pressé montre un visage joyeux, sinon, si le bouton B est pressé sort de la boucle, sinon montre un visage triste. » On peut sortir de la boucle infinie avec l'instruction `break`.

A la toute fin, lorsque notre cyber-animal est mort, on efface l'affichage (avec la méthode `clear`).

Peux-tu trouver des façons de rendre ce jeu moins tragique ? Comment pourrais-tu vérifier que les deux boutons sont pressés ? (Indice : Pyhon possède des opérateurs logiques : `and` -> *et* ; `or` -> *ou* ; `not` -> *contraire de*)

1.4 Entrée/Sortie

Il y a des bandes de métal sur le côté bas du BBC micro :bit qui lui font des sortes de dents. Ce sont les pin d'entrée/sortie (ou pin E/S pour faire court)



Certains de ces pins sont plus gros que d'autres donc il est possible d'y accrocher des pinces crocodiles. Ce sont numérotés 0, 1, 2, 3V et GND (les ordinateurs comptent toujours à partir de zéro). Si tu branches une carte dédiée à ton micro :bit, il est également possible de relier des câbles aux autres pins (plus petits).

Chaque pin sur le BBC micro :bit est représenté par un *objet* appelé `pinN` où `N` est le numéro du pin. Donc, par exemple, pour faire quelque chose avec le pin numéroté 0 (zéro), on utilise l'objet appelé `pin0`.

Facile !

Ces objets ont des *méthodes* variées qui leur sont associées en fonction de ce que ce pin particulier est capable de faire.

1.4.1 Python chatouilleux

L'exemple le plus simple d'entrée par les pins est de déterminer si ils sont touchés. Donc, tu peux chatouiller ton appareil et le faire rire comme ça :

```
from microbit import *

while True:
    if pin0.is_touched():
        display.show(Image.HAPPY)
    else:
        display.show(Image.SAD)
```

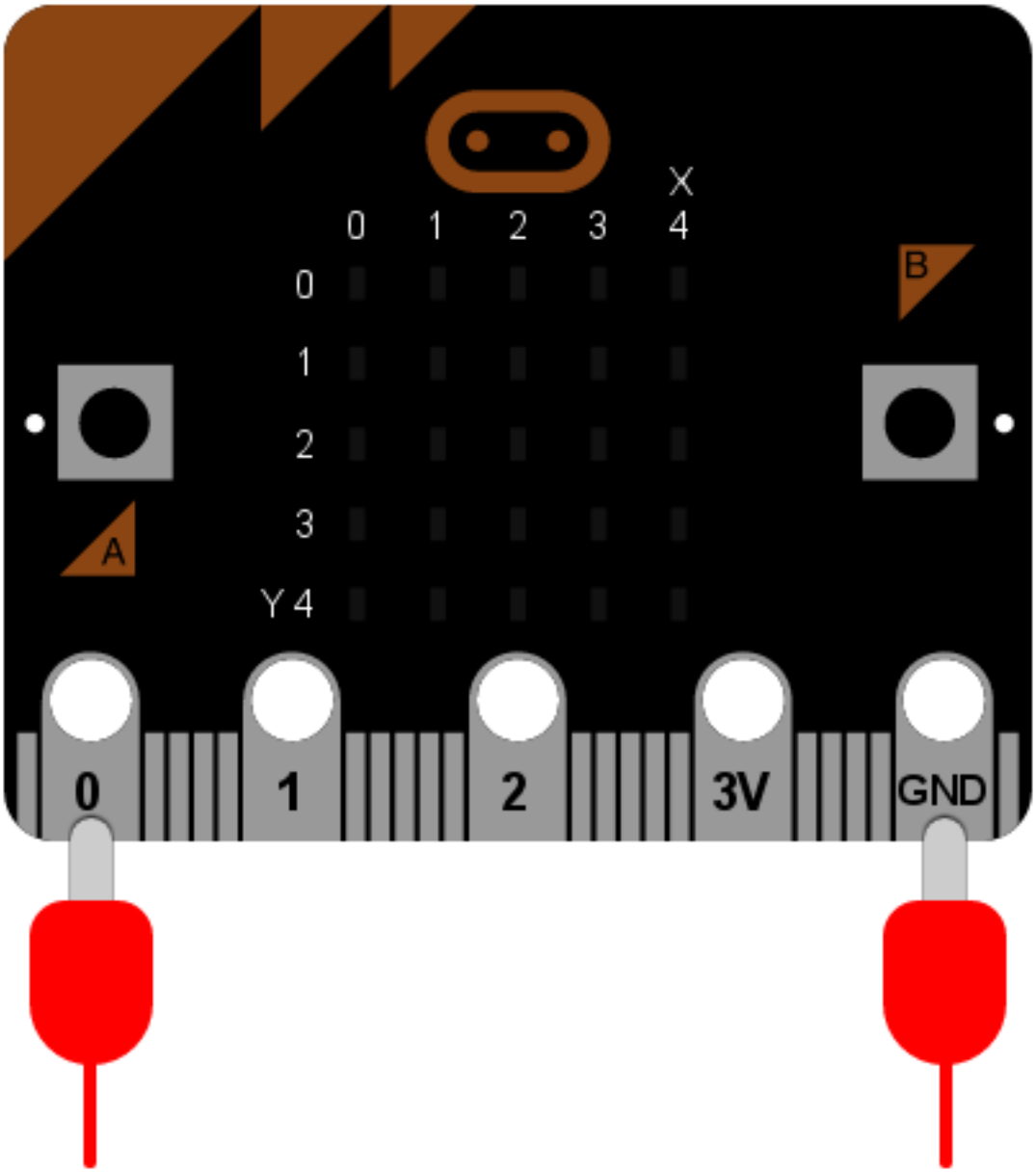
Avec une main, tiens ton appareil par le pin GND. Puis, avec ton autre main, touche (ou chatouille) le pin 0 (zéro). Tu devrais voir l'image affichée passer de grognon à content.

C'est une forme simpliste de mesure de l'entrée. On commence vraiment à s'amuser lorsque l'on connecte des circuits et d'autres appareils grâce aux pins.

1.4.2 Bip Bip

La chose la plus simple que nous puissions raccorder à l'appareil est un buzzer. Nous allons l'utiliser comme une sortie.

Ces petits appareils émettent un bip aigü lorsqu'ils sont connectés à un circuit. Pour en relier un à ton micro :bit, tu dois raccorder une pince crocodile aux pins 0 et GND (voir ci-dessous)



Le câble partant du pin 0 doit être raccordé au connecteur positif du buzzer et celui partant du GND au connecteur négatif.

Le programme suivant fera émettre un son au buzzer :

```
from microbit import *

pin0.write_digital(1)
```

C'est marrant pendant à peu près 5 secondes et ensuite tu auras envie d'arrêter cet horrible couinement. Améliorons notre exemple en le faisant bipper par intervalles :

```
from microbit import *

while True:
```

(suite sur la page suivante)

(suite de la page précédente)

```
    pin0.write_digital(1)
    sleep(20)
    pin0.write_digital(0)
    sleep(480)
```

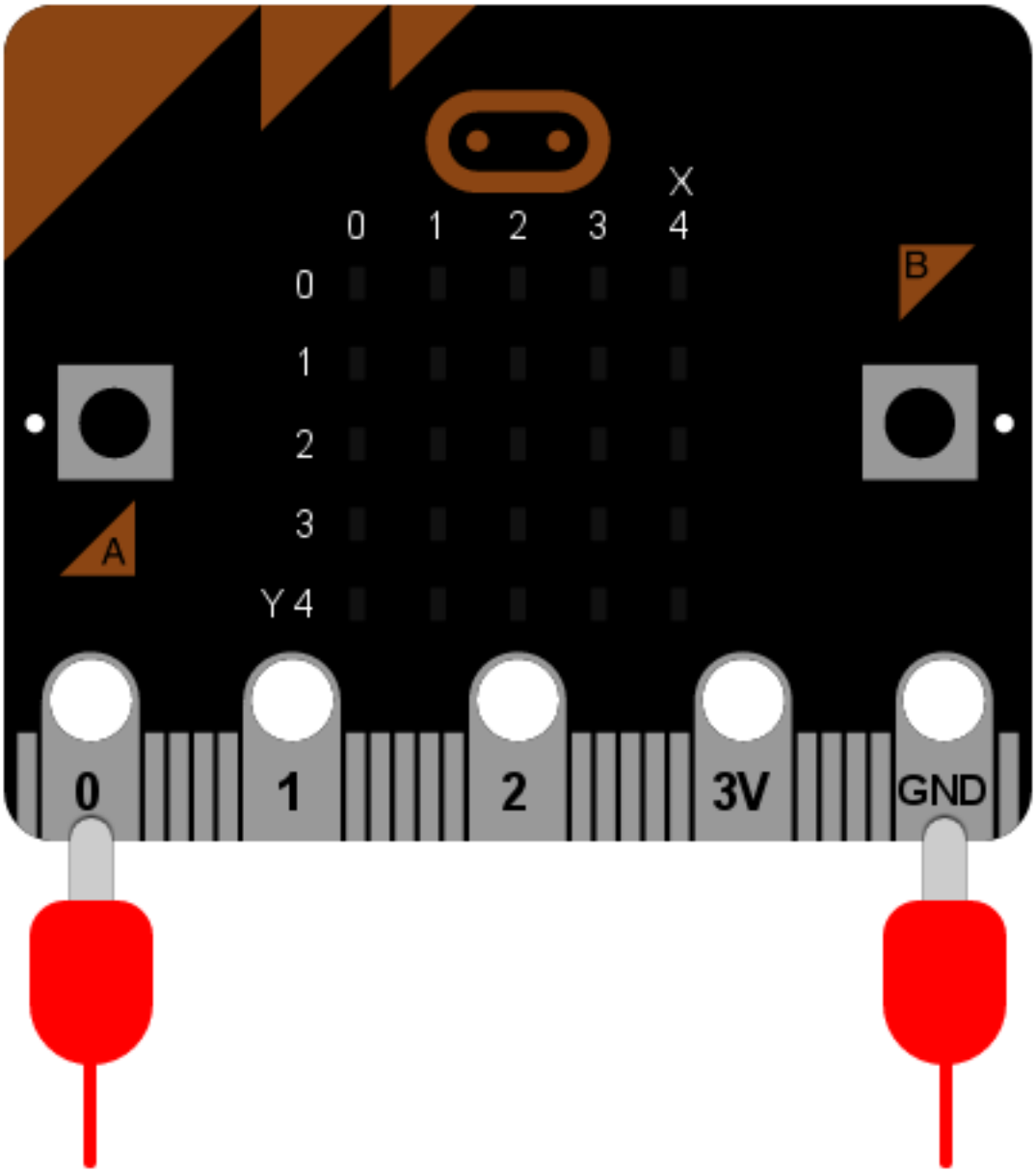
Peux-tu comprendre comment ce script fonctionne ? Rappele-toi que `1` est « on » et `0` est « off » dans le monde digital.

L'appareil est mis dans une boucle infinie et met immédiatement le pin 0 sur « on ». Ce qui fait que le buzzer émet un son. Pendant qu'il fait du bruit, l'appareil dort pendant vingt millisecondes puis passe le pin 0 sur « off ». Cela donne l'effet d'un bip court (20ms). Enfin, l'appareil dort pendant 480 millisecondes avant de recommencer la boucle indéfiniment. Ce qui signifie que tu obtiens deux « bip » par seconde (un toutes les 500 millisecondes)

On a fait un métronome !

1.5 Musique

MicroPython sur le BBC micro :bit est fourni avec un module de musique et de son puissant. Il est très facile de générer des bips avec l'appareil *si tu le raccordes à un haut-parleur*. Utilise des pinces crocodile pour relier les pins 0 et GND aux entrées positives et négatives du haut-parleur.



Note : N'essaye pas de le faire avec un simple buzzer Piezo - ce genre de buzzer est seulement capable de jouer un seul ton.

Jouons de la musique ! :

```
import music

music.play(music.NYAN)
```

Remarque que nous importons le module `music`. Il contient les méthodes utilisées pour produire et contrôler le son.

MicroPython a tout un tas de mélodies pré-programmées. En voici la liste complète :

- `music.DADADADUM`
- `music.ENTERTAINER`

— music.PRELUDE
— music.ODE
— music.NYAN
— music.RINGTONE
— music.FUNK
— music.BLUES
— music.BIRTHDAY
— music.WEDDING
— music.FUNERAL
— music.PUNCHLINE
— music.PYTHON
— music.BADDY
— music.CHASE
— music.BA_DING
— music.WAWAWAWAA
— music.JUMP_UP
— music.JUMP_DOWN
— music.POWER_UP
— music.POWER_DOWN

Prend le code d'exemple et change la mélodie. Quelle est ta préférée ? Comment pourrais-tu utiliser ces airs comme des signaux ou des indices ?

1.5.1 Wolfgang Amadeus Microbit

Créer ta propre mélodie est facile !

Chaque note a un nom (comme `C#` ou `F`), une octave (pour dire à MicroPython à quelle hauteur il faut jouer la note) et une durée (combien de temps elle dure). Les octaves sont indiquées par un nombre ~ 0 est la plus basse et 8 est à peu près aussi haut que ce dont tu auras besoin à moins que tu ne fasses de la musique pour chiens. La durée s'exprime aussi à l'aide de nombres. Plus la valeur de la durée est grande plus la note sera jouée longtemps. Ces valeurs sont reliées entres elles - par exemple, une durée de `4` durera deux fois plus longtemps qu'une durée de `2` (et ainsi de suite). Si tu utilises la note nommée `R` alors MicroPython jouera une pause (c'est-à-dire un silence) pendant la durée spécifiée.

Chaque note est représentée par une chaîne de caractères comme ça :

```
NOTE[octave][:durée]
```

Par exemple, `"A1:4"` représente la note nommée `A` dans l'octave numéro 1 à jouer sur une durée de `4`.

Fais une liste de note pour créer une mélodie (c'est équivalent à créer une animation avec une liste d'images). Par exemple, voici comment faire jouer à MicroPython le début de « Frère Jacques » :

```
import music

tune = ["C4:4", "D4:4", "E4:4", "C4:4", "C4:4", "D4:4", "E4:4", "C4:4",
        "E4:4", "F4:4", "G4:8", "E4:4", "F4:4", "G4:8"]
music.play(tune)
```

Note : MicroPython t'aide à simplifier de telles mélodies. Il se rappellera de l'octave et de la durée jusqu'à ce que tu les changes. Grâce à cela, l'exemple ci-dessus peut-être ré-écrit de cette façon :

```
import music
```

(suite sur la page suivante)

(suite de la page précédente)

```
tune = ["C4:4", "D", "E", "C", "C", "D", "E", "C", "E", "F", "G:8",
        "E:4", "F", "G:8"]
music.play(tune)
```

Remarque que l'octave et la durée ne changent que lorsqu'elles le doivent. Ça fait beaucoup moins à taper et c'est plus simple à lire.

1.5.2 Effets sonores

MicroPython te permets de faire des sons qui ne sont pas des notes de musique. Par exemple, voici comment créer un effet de sirène de police :

```
import music

while True:
    for freq in range(880, 1760, 16):
        music.pitch(freq, 6)
    for freq in range(1760, 880, -16):
        music.pitch(freq, 6)
```

Remarque comment la méthode `music.pitch` est utilisée dans cet exemple. Elle attend une fréquence. Par exemple, une fréquence de `440` est la même chose qu'une note `A` (qui correspond au LA) utilisée pour accorder un orchestre symphonique.

Dans l'exemple ci-dessus la fonction `range` est utilisée pour générer un assortiment de valeurs numériques. Ces nombres sont utilisés pour définir la hauteur du ton. Les trois arguments de la fonction `range` sont la valeur de départ, la valeur de fin et la taille du pas. Ainsi, la première utilisation de `range` consiste à dire, en français, « créé un assortiment de nombres compris entre 880 et 1760 de 16 en 16 ». Sa deuxième utilisation dit « créé un assortiment de nombres compris entre 1760 et 880 de -16 en -16 ». C'est comme ça que l'on obtient une liste de fréquences qui montent puis qui descendent comme une sirène.

Puisque la sirène doit durer éternellement, elle est contenue dans une boucle `while` infinie.

Surtout, nous avons introduit une nouvelle sorte de boucle à l'intérieur de la boucle `while` : la boucle `for`. En français ça revient à dire « pour chaque élément dans une certaine collection, fais des trucs avec ». Plus précisément, dans l'exemple ci-dessus ça dit « pour chaque fréquences dans l'assortiment de fréquences, joue la hauteur de cette fréquence pendant 6 millisecondes ». Remarque que les choses à faire dans cette boucle sont indentées (comme on l'a vu précédemment) de façon à ce que Python sache quel code exécuter avec chaque élément.

1.6 Hasard

Des fois tu as envie des laisser le hasard choisir, ou d'introduire un peu de hasard : tu veux que l'appareil agisse aléatoirement.

MicroPython est fourni avec un module `random` qui facilite l'introduction de hasard et d'un peu de chaos dans ton code. Par exemple, voilà comment montrer un nom au hasard sur l'affichage :

```
from microbit import *
import random

noms = ["Mary", "Yolanda", "Damien", "Alia", "Kushal", "Mei Xiu", "Zoltan" ]

display.scroll(random.choice(noms))
```

La liste `noms` contient sept noms définis par des chaînes de caractères. La ligne finale est *imbriquée* (L'effet *oignon* présenté plus tôt) : la méthode `random.choice` prend la liste `noms` comme argument et renvoie un élément choisi au hasard. Cet élément (le nom choisi au hasard) est l'argument de la méthode `dispaly.scroll`.

Peux-tu modifier la liste pour y inclure ton propre ensemble de noms ?

1.6.1 Nombres aléatoires

Les nombres aléatoires sont très utiles. Ils sont communs dans les jeux. Pour quelle autre raison avons-nous des dés ?

MicroPython est fourni avec plusieurs méthodes utiles pour les nombres aléatoires. Voici comment faire un simple dé :

```
from microbit import *
import random

display.show(str(random.randint(1, 6)))
```

A chaque fois que l'appareil est réinitialisé, il affiche un nombre entre 1 et 6. Tu commences à avoir l'habitude de l'imbrication, donc il est important que tu remarques `random.randint` qui renvoie un nombre entier entre ses deux arguments inclus (un nombre entier est appelé *integer* en anglais d'où le nom de la méthodes). Et que puisque `display.show` nécessite un argument sous forme de caractère on utilise la fonction `str```pour convertir la valeur numérique en un caractère (on converti par exemple ``6` en `"6"`)

Si tu sais que tu voudras toujours un nombre entre `0` et `N` alors tu dois utiliser la méthode `random.range`. Si tu lui fournis un seul argument elle te renverra un entier aléatoire **strictement** inférieur à cet argument. (ce qui est différent du comportement de la méthode‘‘random.randint‘‘)

Des fois tu as besoin de nombres décimaux. Ils sont appelés nombres *à virgule flottante* en informatique souvent abrégé en `float` et il est possible d'en générer de façon aléatoire avec la méthode `random.random`. Elle ne renvoie que des valeurs comprises entre `0.0` et `1.0` inclus. Si tu as besoin de nombres décimaux aléatoires plus grands, ajoute les résultats de `random.randrange` et de `random.random` comme ça :

```
from microbit import *
import random

reponse = random.randrange(100) + random.random()
display.scroll(str(reponse))
```

1.6.2 Les Graines du Chaos

Les générateurs de nombres aléatoire d'un ordinateur ne sont pas vraiment aléatoires. Ils fournissent seulement des résultats semblant aléatoires étant donnée une valeur initiale appelée valeur *graine* (ou *seed* en anglais). Cette graine est souvent générée à partir de valeurs plus ou moins aléatoires comme l'heure du moment ou des lectures de capteurs comme des thermomètres intégrés aux puces électroniques.

Parfois tu auras besoin d'un comportement à peu près aléatoire mais répétable : c'est-à-dire une source de hasard reproductible. C'est comme de dire que tu veux les mêmes cinq valeurs aléatoire à chaque fois que tu lances cinq fois de suite un dé.

C'est facile à accomplir en fixant toi-même la valeur *graine*. A partir d'une même valeur *graine* le générateur de nombres aléatoires produira toujours la même suite de nombres « aléatoires ». La graine est fixée avec la méthode `random.seed` et n'importe quel nombre entier. Cette version du programme de dé produit toujours les mêmes résultats :

```
from microbit import *
import random

random.seed(1337)
while True:
    if button_a.was_pressed():
        display.show(str(random.randint(1, 6)))
```

Comprends-tu pourquoi ce programme nécessite l'appuie sur le bouton A plutôt que de réinitialier le micro :bit comme dans le premier exemple ?

1.7 Mouvement

Ton BBC micro :bit est munit d'un accéléromètre. Il mesure le mouvement selon trois axes :

— X - l'inclinaison de gauche à droite.
— Y - l'inclinaison d'avant en arrière.
— Z - le mouvement haut et bas.

Il y a une méthode pour chaque axe qui renvoie un nombre positif ou négatif qui indique une mesure en milli-g. Lorsque la lecture est de 0, tu es « aligné » selon cet axe.

Par exemple, voici un « niveau à bulle » très simple qui utilise `get_x` pour mesurer l'aligmenet de l'appareil selon l'axe X :

```
from microbit import *

while True:
    lecture = accelerometer.get_x()
    if lecture > 20:
        display.show("D")
    elif lecture < -20:
        display.show("G")
    else:
        display.show("-")
```

Si tu tiens l'appareil horizontalement il devrait afficher - ; en revanche, si tu l'inclines vers la gauche ou vers la droite il devrait afficher `G` ou `D` respectivement.

Nous voulons que l'appareil réagisse aux changement en permanence, donc nous utilisons une boucle `while` infinie. La première chose que l'on fait *dans le corps de cette boucle* est une mesure selon l'axe X que l'on nomme `lecture`. L'accéléromètre étant *tellement* sensible j'ai mis une marge de +/-20 pour le niveau. L'instruction `else` signifie que si `lecture` n'est pas entre -20 et 20 alors on considère qu'on n'est pas de niveau. Pour chacune des conditions on utilise l'affichage pour montrer le caractère approprié.

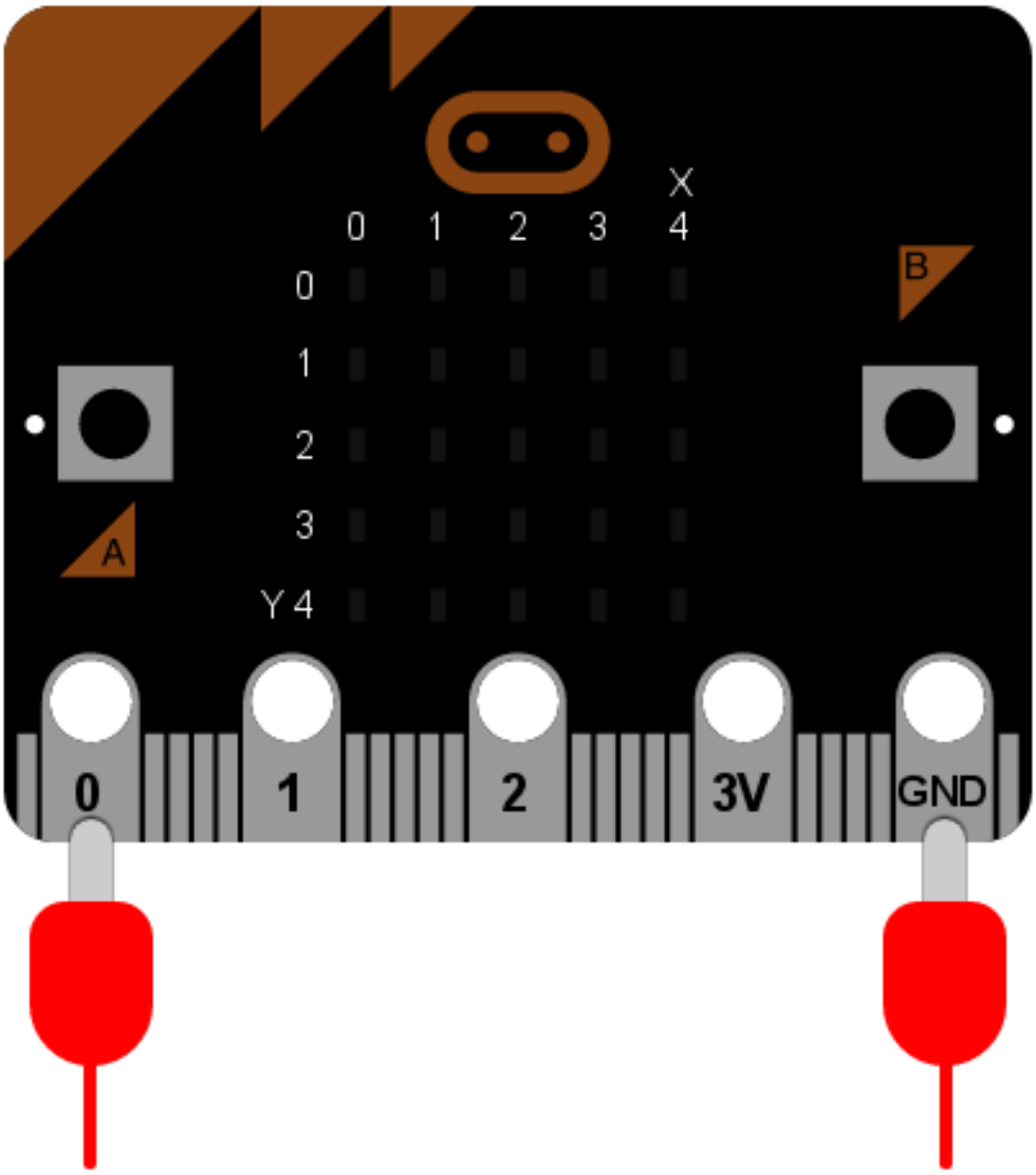
Il y a aussi une méthode `get_y` pour l'axe Y et une méthode `get_z` pour l'axe Z.

Si tu t'es déjà demandé comment un téléphone portable sait dans quel sens afficher les images sur son écran, c'est parce qu'il utilise un accéléromètre de la même façon que le programme ci-dessus. Les manettes de jeux contiennent aussi des accéléromètres pour t'aider à tourner et à te déplacer dans les jeux.

1.7.1 Chaos musical

L'un des aspects les plus merveilleux du MicroPython sur le BBC micro :bit est la façon dont il te permet facilement de relier les différentes possiblités de l'appareil entres elles. Par exemple, transformons-le en un instrument de musique (en quelque sorte)

Connecte un haut-parleur comme tu l'as fait dans le tutoriel sur la musique. Utilise des pinces crocodile pour relier les pin 0 et GND aux connecteurs positif et négatif du haut-parleur - le sens n'a pas d'importance.



Que se passe-t-il si nous utilisons les données de l'accéléromètre pour les jouer comme des hauteurs de note ?

```
from microbit import * import music

while True : music.pitch(accelerometer.get_y(), 10)
```

La ligne clé est à la fin et est remarquablement simple. Nous *imbriquons* la lecture de l'inclinaison sur l'axe Y en tant que fréquence dans la méthode `music.pitch`. Nous ne la jouons que 10 millisecondes car nous voulons que le ton change rapidement à mesure que l'appareil est incliné. Puisque l'appareil est dans une boucle infinie il réagit constamment aux changements de la mesure sur l'axe Y.

C'est tout !

Incline l'appareil en avant et en arrière. Si la lecture de l'inclinaison sur l'axe Y est positive, cela changera le ton joué par le micro :bit.

Imagine un orchestre symphonique complet de ces appareil. Peux-tu jouer une mélodie ? Comment pourrais-tu améliorer le programme pour que le micro :bit joue de façon plus musicale ?

1.8 Gestes

L'effet collatéral le plus intéressant d'un accélèromètre est la détection des gestes.

MicroPython est capable de reconnaître les gestes suivants :

- haut-> `up`
- bas -> `down`
- gauche -> `left`
- droite -> `right`
- face en haut -> `face up`
- face en bas -> `face down`
- chute libre -> `free fall`
- accélération correspondant à 3, 6 ou 8 fois celle de la chute libre -> `3g`, `6g` ou `8g`
- secousse -> `shake`

Les gestes sont toujours représentés par des chaînes de caractères.

Pour obtenir le geste effectué, on utilise la méthode `accelerometer.current_gesture`. Son résultats est l'un des noms de geste listés ci-dessus. Par exemple, ce programme rendra votre appareil heureux seulement lorsque sa face est tournée vers le haut

```
from microbit import *

while True:
    geste = accelerometer.current_gesture()
    if geste == "face up":
        display.show(Image.HAPPY)
    else:
        display.show(Image.ANGRY)
```

Encore une fois, puisque nous voulons que l'appareil réagisse à des circomstances changeantes, nous utilisons une boucle `while`. A l'intérieur du corps de la boucle, le geste est lu et stocké dans `geste`. L'instruction conditionnelle `if` verifie si `geste` est égal à `face up` (Python utilise == pour tester une égalité car un simple signe égal = est utilisé pour l'affectation - tout comme lorsque nous affectons le geste lu à l'objet `geste`). Si le geste est égal à `face up` alors on utilise l'affichage pour montrer un visage heureux. Sinon, l'appareil a l'air mécontent.

1.8.1 Magic-8

Une balle Magic-8 est un jouet inventé dans les année 1950. L'idée est de lui poser une question à laquelle on peut répondre oui ou non, de la secouer et d'attendre qu'elle nous révèle la vérité. C'est plutôt facile à programmer :

```
from microbit import *
import random
reponses = [
    "C'est certain",
    "C'est décidément ainsi",
    "Sans aucun doute",
    "Oui définitivement",
    "Vous pouvez compter dessus",
    "Comme je le vois, oui",
    "Probablement",
    "ça semble bien",
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "Oui",
    "Les signes pointent vers Oui",
    "Réponse brumeuse, essaye encore",
    "Demander à nouveau plus tard",
    "Mieux vaut ne pas te le dire maintenant",
    "Je ne peux pas prédire maintenant",
    "Concentre-toi et demande à nouveau",
    "Ne compte pas dessus",
    "Ma réponse est non",
    "Mes sources disent non",
    "ça ne semble pas si bon",
    "Très douteux",
]
while True:
    display.show("8")
    if accelerometer.was_gesture("shake"):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(reponses))
```

La plus grande partie du programme est une liste nommée `reponses`. Le jeu se trouve dans la boucle `while` à la fin.

L'état par défaut du jeu est l'affichage du caractère `"8"`. Le programme doit détecter si le micro :bit a été secoué. La méthode `was_gesture` utilise son argument (dans ce cas `shake` puique l'on veut détecter une secousse) pour retourner un `True` ou un `False`. Si l'appareil a été secoué, l'instruction `if` exécutera le bloc de code dans lequel l'écran est effacé pendant une seconde (de façon à ce que l'appareil semble réfléchir à ta question) et affiche une réponse choisie au hasard.

Pourquoi ne pas lui demander si c'est le meilleurs programme jamais écrit ? Que pourrais-tu faire pour « tricher » et faire en sorte que la réponse soit toujours positive ou négative ? (Indice : utilise les boutons.)

1.9 Direction

Il y a une boussole sur le BBC micro :bit. Si tu veux faire une station météo tu peux l'utiliser pour déterminer la direction du vent.

1.9.1 Boussole

Il peut aussi t'indiquer la direction du Nord comme ça :

```
from microbit import *

compass.calibrate()

while True:
    aiguille = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[aiguille])
```

Note :

Tu dois calibrer la boussole avant de l'utiliser. Ne pas le faire entraine des résultats inutilisables. La méthode `calibration` lance un petit jeu pour aider l'appareil à déterminer sa position par rapport au champs magnétique de la Terre.

Pour calibrer la boussole, incline le micro :bit dans tous les sens jusqu'à ce qu'un cercle de pixel soit dessiné sur les bords de l'affichage

---

Le programme prend la direction fournie par `compass.heading` et, en utilisant de simple maths de façon astucieuse, division entière `//` and modulo `%`, détermine le nombre que l'aiguille de la montre doit pointer pour indiquer approximaivement le nord.

1.10 Storage

Sometimes you need to store useful information. Such information is stored as data : representation of information (in a digital form when stored on computers). If you store data on a computer it should persist, even if you switch the device off and on again.

Happily MicroPython on the micro :bit allows you to do this with a very simple file system. Because of memory constraints **there is approximately 30k of storage available** on the file system.

What is a file system ?

It's a means of storing and organising data in a persistent manner - any data stored in a file system should survive restarts of the device. As the name suggests, data stored on a file system is organised into files.

A computer file is a named digital resource that's stored on a file system. Such resources contain useful information as data. This is exactly how a paper file works. It's a sort of named container that contains useful information. Usually, both paper and digital files are named to indicate what they contain. On computers it is common to end a file with a

`.something` suffix. Usually, the « something » indicates what type of data is used to represent the information. For example, `.txt` indicates a text file, `.jpg` a JPEG image and `.mp3` sound data encoded as MP3.

Some file systems (such as the one found on your laptop or PC) allow you to organise your files into directories : named containers that group related files and sub-directories together. However, *the file system provided by MicroPython is a flat file system*. A flat file system does not have directories - all your files are just stored in the same place.

The Python programming language contains easy to use and powerful ways in which to work with a computer's file system. MicroPython on the micro :bit implements a useful subset of these features to make it easy to read and write files on the device, while also providing consistency with other versions of Python.

Avertissement : Flashing your micro :bit will DESTROY ALL YOUR DATA since it re-writes all the flash memory used by the device and the file system is stored in the flash memory.

However, if you switch off your device the data will remain intact until you either delete it or re-flash the device.

1.10.1 Open Sesame

Reading and writing a file on the file system is achieved by the `open` function. Once a file is opened you can do stuff with it until you close it (analogous with the way we use paper files). It is essential you close a file so MicroPython knows you've finished with it.

The best way to make sure of this is to use the `with` statement like this :

```
with open('story.txt') as my_file:
    content = my_file.read()
print(content)
```

The `with` statement uses the `open` function to open a file and assign it to an object. In the example above, the `open` function opens the file called `story.txt` (obviously a text file containing a story of some sort). The object that's used to represent the file in the Python code is called `my_file`. Subsequently, in the code block indented underneath the `with` statement, the `my_file` object is used to `read()` the content of the file and assign it to the `content` object.

Here's the important point, *the next line containing the* `print` *statement is not indented*. The code block associated with the `with` statement is only the single line that reads the file. Once the code block associated with the `with` statement is closed then Python (and MicroPython) will automatically close the file for you. This is called context handling and the `open` function creates objects that are context handlers for files.

Put simply, the scope of your interaction with a file is defined by the code block associated with the `with` statement that opens the file.

Confused ?

Don't be. I'm simply saying your code should look like this :

```
with open('some_file') as some_object:
    # Do stuff with some_object in this block of code
    # associated with the with statement.

# When the block is finished then MicroPython
# automatically closes the file for you.
```

Just like a paper file, a digital file is opened for two reasons : to read its content (as demonstrated above) or to write something to the file. The default mode is to read the file. If you want to write to a file you need to tell the `open` function in the following way :

```
with open('hello.txt', 'w') as my_file:
    my_file.write("Hello, World!")
```

Notice the `'w'` argument is used to set the `my_file` object into write mode. You could also pass an `'r'` argument to set the file object to read mode, but since this is the default, it's often left off.

Writing data to the file is done with the (you guessed it) `write` method that takes the string you want to write to the file as an argument. In the example above, I write the text « Hello, World ! » to a file called « hello.txt ».

Simple !

Note : When you open a file and write (perhaps several times while the file is in an open state) you will be writing OVER the content of the file if it already exists.

If you want to append data to a file you should first read it, store the content somewhere, close it, append your data to the content and then open it to write again with the revised content.

While this is the case in MicroPython, « normal » Python can open files to write in « append » mode. That we can't do this on the micro :bit is a result of the simple implementation of the file system.

1.10.2 OS SOS

As well as reading and writing files, Python can manipulate them. You certainly need to know what files are on the file system and sometimes you need to delete them too.

On a regular computer, it is the role of the operating system (like Windows, OSX or Linux) to manage this on Python's behalf. Such functionality is made available in Python via a module called `os`. Since MicroPython **is** the operating system we've decided to keep the appropriate functions in the `os` module for consistency so you'll know where to find them when you use « regular » Python on a device like a laptop or Raspberry Pi.

Essentially, you can do three operations related to the file system : list the files, remove a file and ask for the size of a file.

To list the files on your file system use the `listdir` function. It returns a list of strings indicating the file names of the files on the file system :

```
import os
my_files = os.listdir()
```

To delete a file use the `remove` function. It takes a string representing the file name of the file you want to delete as an argument, like this :

```
import os
os.remove('filename.txt')
```

Finally, sometimes it's useful to know how big a file is before reading from it. To achieve this use the `size` function. Like the `remove` function, it takes a string representing the file name of the file whose size you want to know. It returns an integer (whole number) telling you the number of bytes the file takes up :

```
import os
file_size = os.size('a_big_file.txt')
```

It's all very well having a file system, but what if we want to put or get files on or off the device ?

Just use the `microfs` utility !

1.10.3 File Transfer

If you have Python installed on the computer you use to program your BBC micro :bit then you can use a special utility called `microfs` (shortened to `ufs` when using it in the command line). Full instructions for installing and using all the features of microfs can be found in its documentation.

Nevertheless it's possible to do most of the things you need with just four simple commands :

```
$ ufs ls
story.txt
```

The `ls` sub-command lists the files on the file system (it's named after the common Unix command, `ls`, that serves the same function).

```
$ ufs get story.txt
```

The `get` sub-command gets a file from the connected micro :bit and saves it into your current location on your computer (it's named after the `get` command that's part of the common file transfer protocol [FTP] that serves the same function).

```
$ ufs rm story.txt
```

The `rm` sub-command removes the named file from the file system on the connected micro :bit (it's named after the common Unix command, `rm`, that serves the same function).

```
$ ufs put story2.txt
```

Finally, the `put` sub-command puts a file from your computer onto the connected device (it's named after the `put` command that's part of FTP that serves the same function).

1.10.4 Mainly main.py

The file system also has an interesting property : if you just flashed the MicroPython runtime onto the device then when it starts it's simply waiting for something to do. However, if you copy a special file called `main.py` onto the file system, upon restarting the device, MicroPython will run the contents of the `main.py` file.

Furthermore, if you copy other Python files onto the file system then you can `import` them as you would any other Python module. For example, if you had a `hello.py` file that contained the following simple code :

```
def say_hello(name="World"):
    return "Hello, {}!".format(name)
```

. . . you could import and use the `say_hello` function like this :

```
from microbit import display
from hello import say_hello

display.scroll(say_hello())
```

Of course, it results in the text « Hello, World ! » scrolling across the display. The important point is that such an example is split between two Python modules and the `import` statement is used to share code.

Note : If you have flashed a script onto the device in addition to the MicroPython runtime, then MicroPython will ignore `main.py` and run your embedded script instead.

To flash just the MicroPython runtime, simply make sure the script you may have written in your editor has zero characters in it. Once flashed you'll be able to copy over a `main.py` file.

1.11 Speech

Avertissement : WARNING ! THIS IS ALPHA CODE.

We reserve the right to change this API as development continues.

The quality of the speech is not great, merely « good enough ». Given the constraints of the device you may encounter memory errors and / or unexpected extra sounds during playback. It's early days and we're improving the code for the speech synthesiser all the time. Bug reports and pull requests are most welcome.

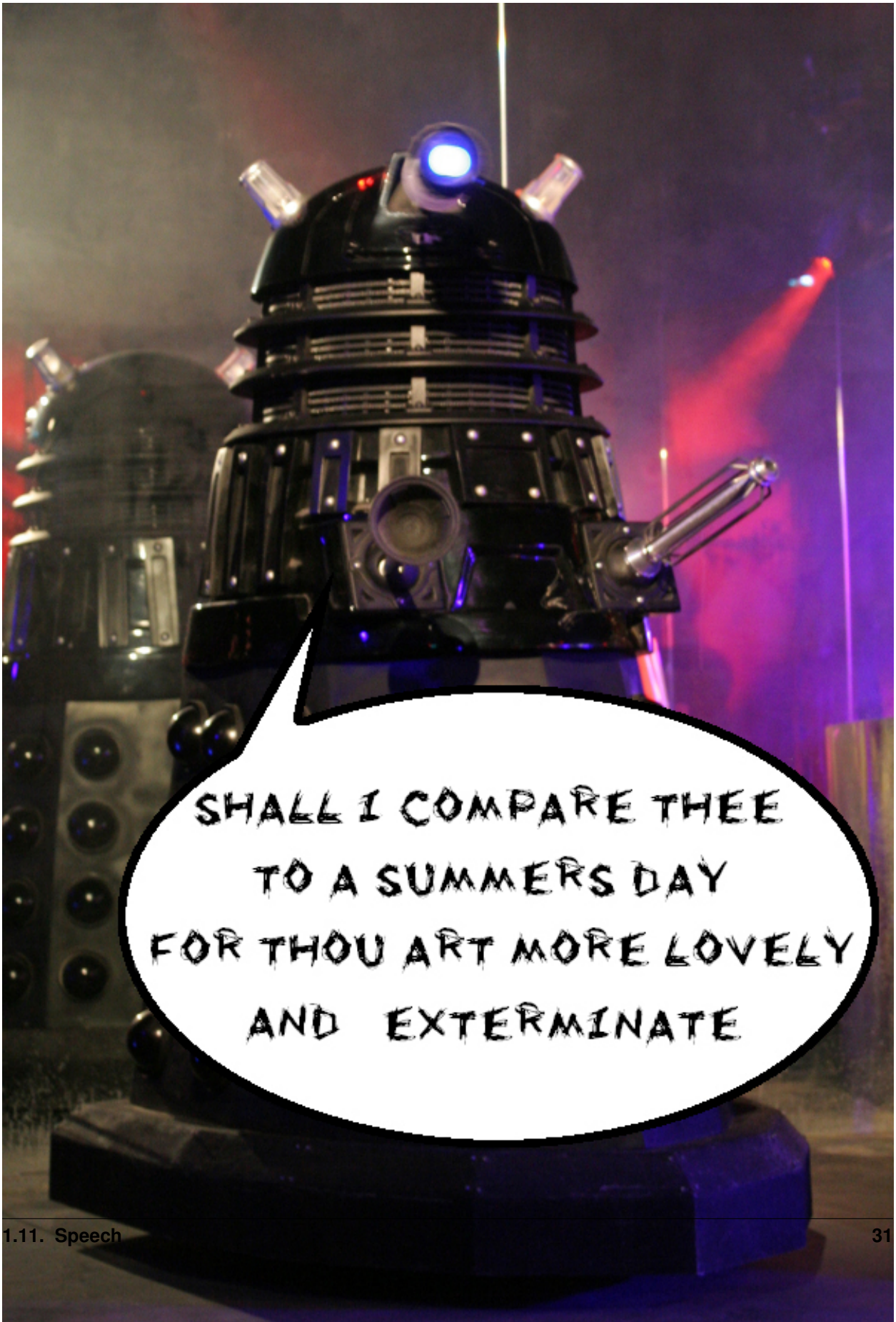
Computers and robots that talk feel more « human ».

So often we learn about what a computer is up to through a graphical user interface (GUI). In the case of a BBC micro :bit the GUI is a 5x5 LED matrix, which leaves a lot to be desired.

Getting the micro :bit talk to you is one way to express information in a fun, efficient and useful way. To this end, we have integrated a simple speech synthesiser based upon a reverse-engineered version of a synthesiser from the early 1980s. It sounds very cute, in an « all humans must die » sort of a way.

With this in mind, we're going to use the speech synthesiser to create…

1.11.1 DALEK Poetry



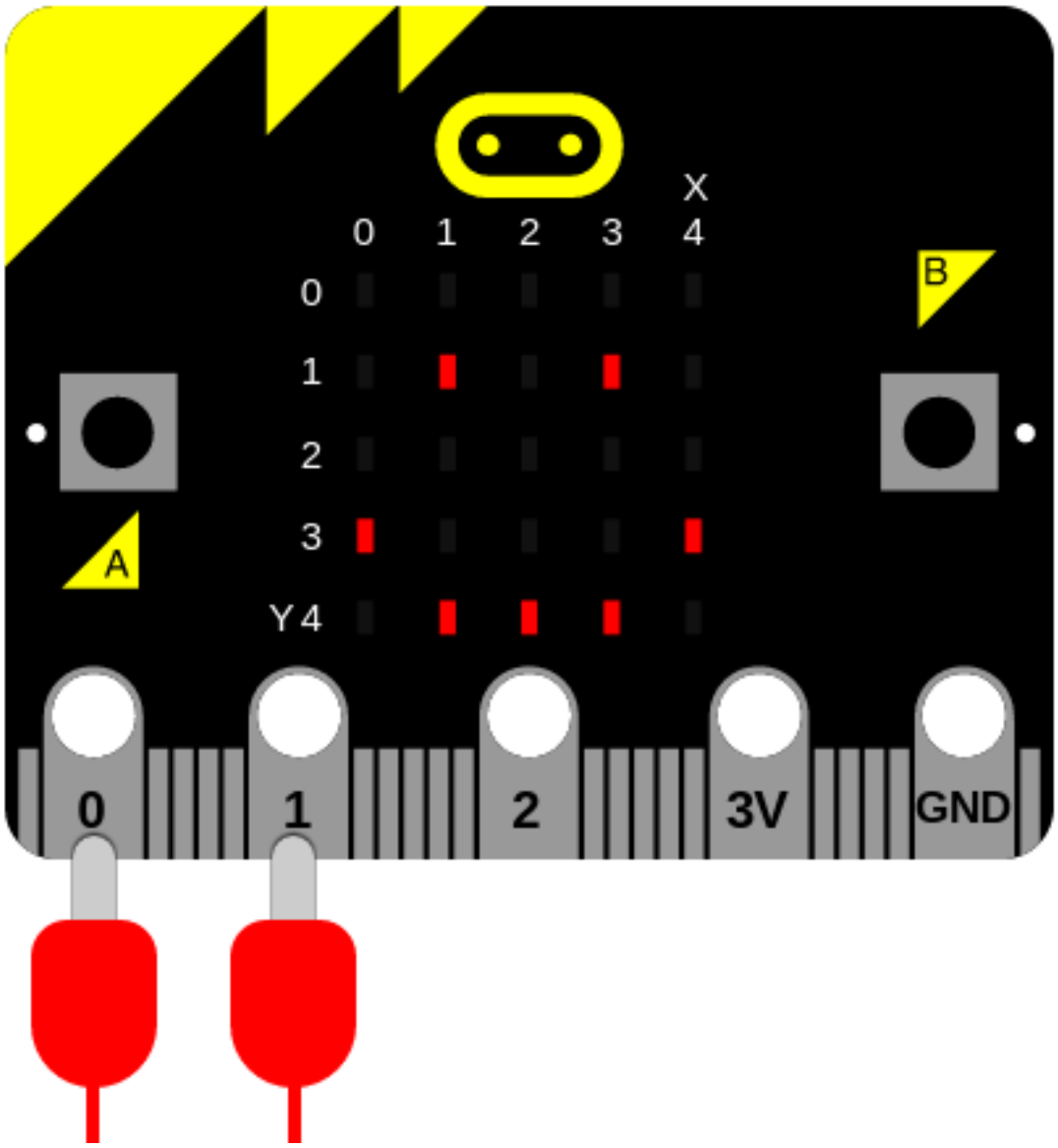
It's a little known fact that DALEKs enjoy poetry ~ especially limericks. They go wild for anapestic meter with a strict AABBA form. Who'd have thought ?

(Actually, as we'll learn below, it's The Doctor's fault DALEKs like limericks, much to the annoyance of Davros.)

In any case, we're going to create a DALEK poetry recital on demand.

1.11.2 Say Something

Before the device can talk you need to plug in a speaker like this :



The simplest way to get the device to speak is to import the `speech` module and use the `say` function like this :

```
import speech

speech.say("Hello, World")
```

While this is cute it's certainly not DALEK enough for our taste, so we need to change some of the parameters that the speech synthesiser uses to produce the voice. Our speech synthesiser is quite powerful in this respect because we can change four parameters :

- `pitch` - how high or low the voice sounds (0 = high, 255 = Barry White)
- `speed` - how quickly the device talks (0 = impossible, 255 = bedtime story)
- `mouth` - how tight-lipped or overtly enunciating the voice sounds (0 = ventriloquist's dummy, 255 = Foghorn Leghorn)
- `throat` - how relaxed or tense is the tone of voice (0 = falling apart, 255 = totally chilled)

Collectively, these parameters control the quality of sound - a.k.a. the timbre. To be honest, the best way to get the tone of voice you want is to experiment, use your judgement and adjust.

To adjust the settings you pass them in as arguments to the `say` function. More details can be found in the `speech` module's API documentation.

After some experimentation we've worked out this sounds quite DALEK-esque :

```
speech.say("I am a DALEK - EXTERMINATE", speed=120, pitch=100, throat=100, mouth=200)
```

1.11.3 Poetry on Demand

Being Cyborgs DALEKs use their robot capabilities to compose poetry and it turns out that the algorithm they use is written in Python like this :

```
# DALEK poetry generator, by The Doctor
import speech
import random
from microbit import sleep

# Randomly select fragments to interpolate into the template.
location = random.choice(["brent", "trent", "kent", "tashkent"])
action = random.choice(["wrapped up", "covered", "sang to", "played games with"])
obj = random.choice(["head", "hand", "dog", "foot"])
prop = random.choice(["in a tent", "with cement", "with some scent",
                      "that was bent"])
result = random.choice(["it ran off", "it glowed", "it blew up",
                        "it turned blue"])
attitude = random.choice(["in the park", "like a shark", "for a lark",
                          "with a bark"])
conclusion = random.choice(["where it went", "its intent", "why it went",
                            "what it meant"])

# A template of the poem. The {} are replaced by the named fragments.
poem = [
    "there was a young man from {}".format(location),
    "who {} his {} {}".format(action, obj, prop),
    "one night after dark",
    "{} {}".format(result, attitude),
    "and he never worked out {}".format(conclusion),
    "EXTERMINATE",
]

# Loop over each line in the poem and use the speech module to recite it.
for line in poem:
    speech.say(line, speed=120, pitch=100, throat=100, mouth=200)
    sleep(500)
```

As the comments demonstrate, it's a very simple in design :

— Named fragments (`location`, `prop`, `attitude` etc) are randomly generated from pre-defined lists of possible values. Note the use of `random.choice` to select a single item from a list.
— A template of a poem is defined as a list of stanzas with « holes » in them (denoted by `{}`) into which the named fragments will be put using the `format` method.
— Finally, Python loops over each item in the list of filled-in poetry stanzas and uses `speech.say` with the settings for the DALEK voice to recite the poem. A pause of 500 milliseconds is inserted between each line because even DALEKs need to take a breath.

Interestingly the original poetry related routines were written by Davros in FORTRAN (an appropriate language for DALEKS since you type it ALL IN CAPITAL LETTERS). However, The Doctor went back in time to precisely the point between Davros's unit tests passing and the deployment pipeline kicking in. At this instant he was able to insert a MicroPython interpreter into the DALEK operating system and the code you see above into the DALEK memory banks as a sort of long hidden Time-Lord Easter Egg or Rickroll.

1.11.4 Phonemes

You'll notice that sometimes, the `say` function doesn't accurately translate from English words into the correct sound. To have fine grained control of the output, use phonemes : the building-block sounds of language.

The advantage of using phonemes is that you don't have to know how to spell ! Rather, you only have to know how to say the word in order to spell it phonetically.

A full list of the phonemes the speech synthesiser understands can be found in the API documentation for speech. Alternatively, save yourself a lot of time by passing in English words to the `translate` function. It'll return a first approximation of the phonemes it would use to generate the audio. This result can be hand-edited to improve the accuracy, inflection and emphasis (so it sounds more natural).

The `pronounce` function is used for phoneme output like this :

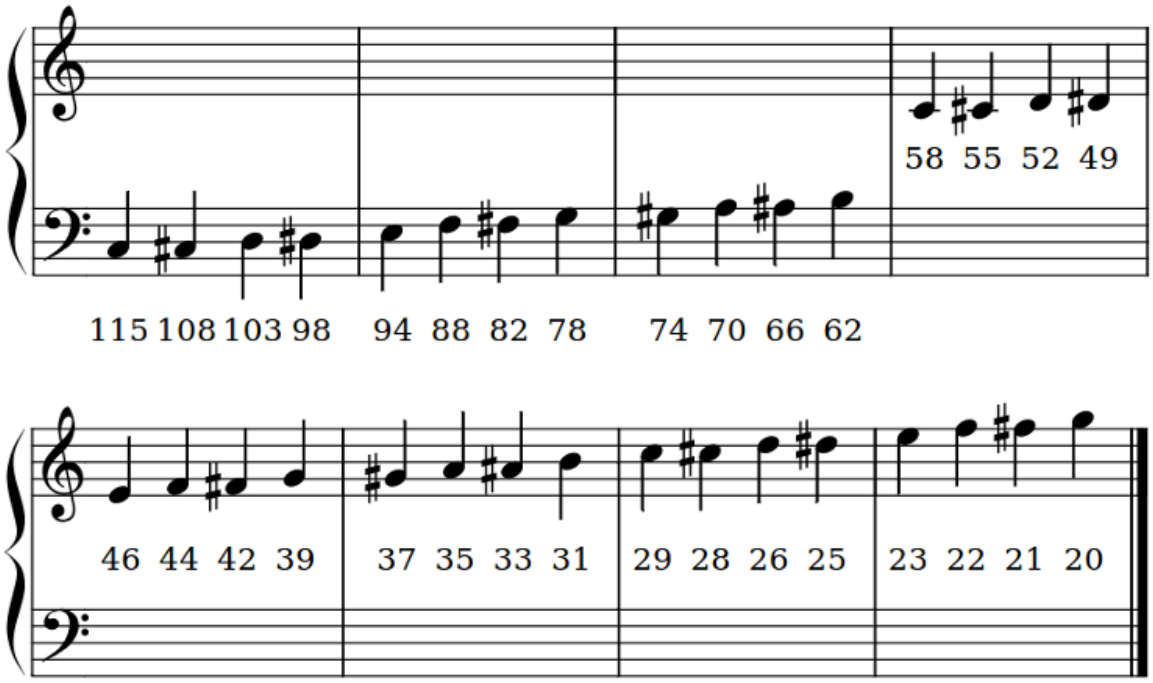
```
speech.pronounce("/HEH5EH4EH3EH2EH2EH3EH4EH5EHLP.")
```

How could you improve on The Doctor's code to make it use phonemes ?

1.11.5 Sing A Song of Micro :bit

By changing the `pitch` setting and calling the `sing` function it's possible to make the device sing (although it's not going to win Eurovision any time soon).

The mapping from pitch numbers to musical notes is shown below :



The `sing` function must take phonemes and pitch as input like this :

```
speech.sing("#115DOWWWW")
```

Notice how the pitch to be sung is prepended to the phoneme with a hash (#). The pitch will remain the same for subsequent phonemes until a new pitch is annotated.

The following example demonstrates how all three generative functions (`say`, `pronounce` and `sing`) can be used to produce speech like output :

```
import speech
from microbit import sleep

# The say method attempts to convert English into phonemes.
speech.say("I can sing!")
sleep(1000)
speech.say("Listen to me!")
sleep(1000)

# Clearing the throat requires the use of phonemes. Changing
# the pitch and speed also helps create the right effect.
speech.pronounce("AEAE/HAEMM", pitch=200, speed=100)  # Ahem
sleep(1000)

# Singing requires a phoneme with an annotated pitch for each syllable.
solfa = [
    "#115DOWWWWWW",   # Doh
    "#103REYYYYYY",   # Re
    "#94MIYYYYYY",    # Mi
    "#88FAOAOAOAOR",  # Fa
    "#78SOHWWWWW",    # Soh
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "#70LAOAOAOAOR",   # La
    "#62TIYYYYYY",     # Ti
    "#58DOWWWWWW",     # Doh
]

# Sing the scale ascending in pitch.
song = ''.join(solfa)
speech.sing(song, speed=100)
# Reverse the list of syllables.
solfa.reverse()
song = ''.join(solfa)
# Sing the scale descending in pitch.
speech.sing(song, speed=100)
```

1.12 Réseau

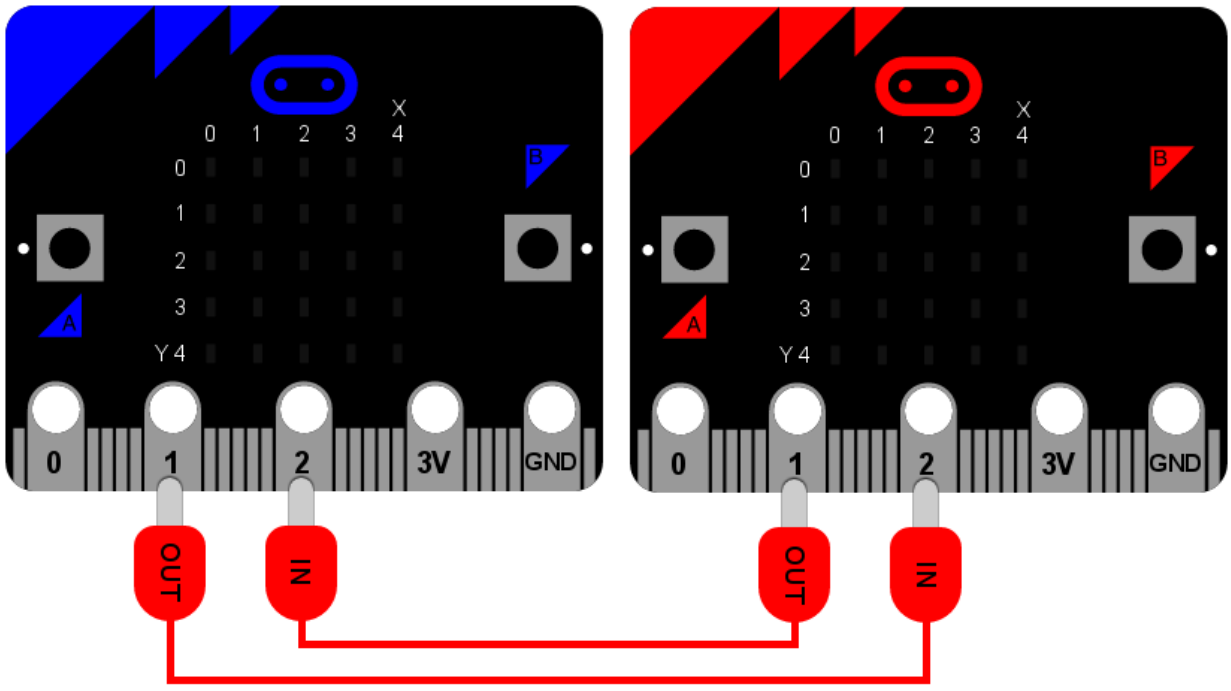
Il est possible de connecter des appareils pour qu'ils s'envoient et reçoivent des messages l'un de l'autre.

Travailler en réseau est difficile et c'est ce que montre le programme décrit plus bas. Cependant, le bon côté de ce projet est qu'il contient tous les aspects classiques de la programmation en réseau que tu as besoin de connaître. Il est aussi remarquablement simple et amusant.

Mais d'abord, décrivons le cadre…

1.12.1 Connexion

Imagine un réseau comme une série de couches. Tout en bas il y a les aspects les plus fondamentaux de la communication : il doit y avoir des sortes de chemins pour qu'un signal se rende d'un appareil à un autre. Parfois ceci est obtenu par des connexions radio, mais dans cet exemple nous allons simplement utiliser deux câbles.



C'est sur cette base solide que nous allons construire toutes les autres couches de notre *pile réseau*.

Comme le schéma le montre, les micro :bits bleu et rouge sont connectés par des pinces crocodile. Ils utilisent tous les deux le pin 1 pour la sortie et le pin 2 pour l'entrée. La sortie de l'un des deux appareils est connectée à l'entrée de l'autre. C'est un peu comme savoir dans quel sens tenir un combiné de téléphone - un côté a un microphone (l'entrée) et l'autre un haut-parleur (la sortie). L'enregistrement de ta voix par ton microphone est retransmise par le haut-parleur de ton interlocuteur. Si tu tiens le téléphone à l'envers, tu obtiendras un résultat bizarre !

C'est exactement la même chose dans cet exemple : tu dois connecter les câbles correctement !

1.12.2 Signal

La couche suivante de notre *pile réseau* est le signal. Souvent elle dépendra des caractéristiques de la connexion. Dans notre exemple il s'agit simplement de signaux digitaux on et off envoyés au travers des câbles par les pins d'E/S.

Si tu te rappelles bien, il est possible d'utiliser les pins d'E/S comme ça :

```
pin1.write_digital(1)  # bascule le signal sur on
pin1.write_digital(0)  # bascule le signal sur off
input = pin2.read_digital()  # lit la valeur du signal (soit 1 soit 0)
```

L'étape suivante implique la description de la façon de gérer un signal. Pour cela nous avons besoin d'un…

1.12.3 Protocole

Si tu rencontres la Reine un jour, il y a des attentes sur le comportement que tu devras adopter. Par exemple, lorsqu'elle arrive tu dois faire une révérence, si elle te tend la main, serre-la poliement, adresse-toi à elle en disant « votre majesté » et après « madame » et ainsi de suite. Cet ensemble de règles s'appelle le protocole royal. Un protocole explique comment se comporter dans une situation particulière (comme celle de ta rencontre avec la Reine). Un protocole est pré-défini pour s'assurer que tout le monde comprend ce qu'il se passe avant qu'une situation donnée ne survienne.

C'est pour cette raison que nous définissons et utilisons des protocoles de communication pour les messages au travers d'un réseau informatique. Les ordinateurs doivent s'entendre avant de savoir comment envoyer et recevoir des messages. Le protocole le plus connu est peut-être le protocole de transfert hypertexte (HTTP) utilisé par le World Wide Web.

Un autre protocole célèbre pour l'envoi de messages (qui précède les ordinateurs) est le code Morse. Il définit comment envoyer des messages basés sur des caractères via des signaux on / off de longues ou de courtes durées. Souvent, ces

signaux sont joués comme des bips. Les longues durées sont appelées des tirets (‘‘ -) alors que les durées courtes sont des points (. ‘‘). En combinant des tirets et des points, Morse définit un moyen d'envoyer des caractères.Par exemple, voici comment est défini l'alphabet Morse standard :

.- A .— J . . . S .—- 1 —-. 9 -. . . B -.- K - T ..— 2 —— 0 -.-. C .-.. L ..- U . . . – 3 -.. D – M . . . - V . . . .- 4 .
E -. N .– W . . . .. 5 ..-. F — O -..- X -. . . . 6 –. G .–. P -.– Y –. . . 7 . . . . H –.- Q –.. Z —.. 8 .. I .-. R

Compte tenu du tableau ci-dessus, pour envoyer le caractère « H » le signal est allumé quatre fois pendant une courte durée, indiquant quatre points (‘‘ . . . .) . Pour la lettre "L" le signal est également allumé quatre fois, mais le deuxième signal a une durée plus longue (.- .. ‘‘).

Evidemment, le timing du signal est important : il faut pouvoir distinguer un point d'un tiret. C'est un autre point d'un protocole, se mettre d'accord sur de telles choses afin que toutes les mises en œuvre du protocole fonctionnent avec tout le monde. Dans cet exemple nous allons juste dire que :

— Un signal d'une durée inférieure à 250 millisecondes est un point.
— Un signal d'une durée de 250 millisecondes à moins de 500 millisecondes est un tiret.
— Toute autre durée de signal est ignorée.
— Une pause / trou supérieure à 500 millisecondes dans le signal indique la fin d'un caractère.

De cette manière, l'envoi d'une lettre « H » est défini comme quatre signaux « on » pas plus long que 250 millisecondes chacun, suivi d'une pause supérieure à 500 millisecondes (indiquant la fin du caractère).

1.12.4 Message

Nous sommes enfin à un stade où nous pouvons construire un message - un message qui signifie quelque chose pour nous les humains. C'est la couche supérieure de notre *pile réseau*.

En utilisant le protocole défini ci-dessus, je peux envoyer la séquence de signaux suivante au travers du câble physique à l'autre micro :bit

```
...././.-../.-../---/.--/---/.-./.-../-..
```

Peux-tu trouver ce qu'il dit ?

1.12.5 Application

C'est très bien d'avoir une pile réseau, mais vous avez également besoin d'un moyen d'interagir avec elle - une forme d'application pour envoyer et recevoir des messages. Bien que HTTP soit intéressant * la plupart des gens * ne le savent pas et laissent leur navigateur Web le gèrer - la *pile réseau * sous-jacente du World Wide Web est cachée (comme il se doit).

Alors, quelle sorte d'application devrions-nous écrire pour le BBC micro :bit ? Comment devrait-elle fonctionner, du point de vue de l'utilisateur ?

De toute évidence, pour envoyer un message, vous devez pouvoir saisir des points et des tirets (nous pouvons utiliser le bouton A pour cela). Si nous voulons voir le message que nous avons envoyé ou reçu, nous devrions être en mesure de déclencher le défilement de l'affichage (nous pouvons utilisez le bouton B pour cela). Enfin, ceci étant du code Morse, si un haut-parleur est raccordé, nous devrions être en mesure de jouer les bips comme une forme de rétroaction sonore pendant que l'utilisateur entre son message.

1.12.6 Le Résultat Final

Voici le programme, dans toute sa splendeur et annoté avec plein de commentaires pour que tu puisses voir ce qui se passe

```
from microbit import *
import music

# Une table de correspondance du code Morse.
MORSE_CODE_LOOKUP = {
    ".-": "A",
    "-...": "B",
    "-.-.": "C",
    "-..": "D",
    ".": "E",
    "..-.": "F",
    "--.": "G",
    "....": "H",
    "..": "I",
    ".---": "J",
    "-.-": "K",
    ".-..": "L",
    "--": "M",
    "-.": "N",
    "---": "O",
    ".--.": "P",
    "--.-": "Q",
    ".-.": "R",
    "...": "S",
    "-": "T",
    "..-": "U",
    "...-": "V",
    ".--": "W",
    "-..-": "X",
    "-.--": "Y",
    "--..": "Z",
    ".----": "1",
    "..---": "2",
    "...--": "3",
    "....-": "4",
    ".....": "5",
    "-....": "6",
    "--...": "7",
    "---..": "8",
    "----.": "9",
    "-----": "0"
}


def decode(buffer):
    # Essaie d'obtenir une correspondance entre le contenu du buffer et le
    # code Morse. Si il n'y en a pas, renvoie juste un point.
    return CODE_MORSE_CORRESPONDANCE.get(buffer, '.')


# Image correspondant à un POINT.
POINT = Image("00000:"
              "00000:"
              "00900:"
              "00000:"
              "00000:")
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Image correspondant à un TIRET.
TIRET = Image("00000:"
              "00000:"
              "09990:"
              "00000:"
              "00000:")


# Pour créer un POINT tu dois maintenir le bouton pendant moins de 250ms
SEUIL_POINT = 250
# Pour créer un TIRET tu dois maintenir le bouton pendant moins de 500ms
SEUIL_TIRET = 500


# Contient le signal Morse entrant.
buffer = ''
# Contient le Morse traduit en caractères.
message = ''
# Le temps depuis lequel l'appareil a attendu le prochain appui sur une touche.
debut_attente = running_time()


# Met l'appareil dans une boucle pour attendre et réagir aux appuis sur un
# bouton
while True:
    # Détermine le temps que l'appareil a attendu l'appui
    attente = running_time() - debut_attente
    # Réinitialise l'horodatage de temp_presse
    temps_touche_presse = None
    # Si le bouton A est mainteu appuyé alors...
    while button_a.is_pressed():
        # Emet un bip - c'est du code Morse tu sais ;-)
        music.pitch(880, 10)
        # Met le pin1 (sortie) sur "on"
        pin1.write_digital(1)
        # ...et si il n'y a pas encore de temps_touche_presse alors on le met␣
↪maintenant!
        if not temps_touche_presse:
            temps_touche_presse = running_time()
    # Alternativment, si le pin2 (input) reçoit un signal, on fait comme  si
    # c'était un appui sur le bouton A
    while pin2.read_digital():
        if not temps_touche_presse:
            temps_touche_presse = running_time()
    # On récupère l'heure actuelle et on l'appelle temps_touche_haute
    temps_touche_haute = running_time()
    # Met le pin1 (sortie) sur "off"
    pin1.write_digital(0)
    # Si il y a un temps_touche_presse (créé lorsque le bouton A a été pressé
    # la  première fois).
    if temps_touche_presse:
        # ...détermine depuis combien de temps il est appuyé
        duree = temps_touche_haute - temps_touche_presse
        # Si la durée est inférieur à la durée maximale d'un DOT...
        if duree < SEUIL_POINT:
            # ... alors ajoute un point au buffer contenant le code Morse entrant
```

(suite sur la page suivante)

(suite de la page précédente)

```
            # et montre un point sur l'affichage..
            buffer += '.'
            display.show(POINT)
        # Sinon, si la durée  est inférieur à la durée maximale d'un TIRET...
        # (mais plus longue que celle d'un POINT ~qui a été géré avant)
        elif duree < SEUIL_TIRET:
            # ... alors ajoute un tiret au buffer contenant le code Morse entrant
            # et affiche-le.
            buffer += '-'
            display.show(TIRET)
        # Sinon, toutes les autres durée d'appui sont ignorées (ce n'est pass
        # nécessaire mais on le met pour la compréhension)
        else:
            pass
        # L'appui sur le bouton a été géré, on peut réinitialiser le temps
        # d'attente pour le prochain appui.
        debut_attente = running_time()

    # Sinon, il n'y a pas eu d'appui pendant ce cycle de la boucle, donc il
    # faut vérifier qu'il n'y a pas eu de pause indiquant la fin du
    # code Morse du caractère entrant. La pause doit être plus longue que
    # la durée du code d'un TIRET.
    elif len(buffer) > 0 and attente > SEUIL_TIRET:
        # Le buffer n'est pas vide et on a atteitn la fin du code donc...
        # il faut décoder le buffer entrant.
        character = decode(buffer)
        # Puis réinitialiser le buffer
        buffer = ''
        # Afficher le caractère décodé
        display.show(character)
        # et ajouter le caractère au message.
        message += character
    # Enfin, si le bouton B a été appuyé pendant tout ce temps là...
    if button_b.was_pressed():
        # ... affiche le message,
        display.scroll(message)
        # et réinitialise-le  (prêt pour un nuoveau message).
        message = ''
```

Comment l'améliorerais-tu ? Peux-tu changer la définition d'un point et d'un tiret pour que utilisateurs rapides de code Morse puissent l'utiliser ? Que se passe-t-il si les deux appareils envoient en même temps ? Que pourriez-vous faire pour gérer cette situation ?

1.13 Radio

L'interaction à distance ressemble à de la magie.

La magie peut être utile si tu es un elfe, un sorcier ou une licorne, mais ces choses n'existent que dans les histoires.

Cependant, il y a quelque chose de beaucoup mieux que la magie : la physique !

L'interaction sans fil n'est que de la physiwue : les ondes radio (une sorte de radiation électromagnétique, un peu comme la lumière visible) ont certaines propriétés (comme l'amplitude, la pulsation ou la période) modulées par un émetteur de façon à ce que cette information puisse être encodée et ainsi diffusée. Lorsque des ondes radio rencontrent un conducteur électrique (c'est-à-dire une antenne), elles provoquent l'apparition d'un courant alternatif duquel l'information contenue dans les ondes peut être extraite et retraduite dans sa forme originale.

1.13.1 Couches après couches

Si tu te rappelles bien, les réseaux sont fabriqués en couches.

Le besoin le plus fondamental d'un réseau sont des sortes de chemins pour qu'un signal se rende d'un appareil à un autre. Dans notre tutoriel sur les réseaux nous utilisions des câbles connectés au pins d'E/S. Grâce au module radio on peut se passer des câbles et utiliser la physique résumée plus haut comme l'invisible connexion entre les appareils.

La couche suivante dans la pile réseau est également différente par rapport à notre exemple dans le tutoriel sur les réseaux. Dans l'exemple câblé on utilisait le *on* et le *off* digital (1 ou 0) pour envoyer et recevoir un signal. Avec le module radio inclu sur le micro :bit la plus petite partie utilisable d'un signal est l'octet.

1.13.2 Octets

Un octet est une unité d'information qui est (habituellement) constituée de huit bits. Un bit est la plus petite unité d'information possible puisqu'il ne peut être que dans deux états : éteint ou allumé (*on* ou *off* ~ 1 ou 0)

Les octets fonctionnent comme des sortes d'abaques : chaque position dans l'octet est comme une colonne dans un abaque - elle représente un nombre qui lui est associé. Dans un abaque il y a habituellement une colonne pour les milliers, une pour les centaines, une pour les dizaines et une pour les unités. Dans un octet la position la plus à gauche représente 128, puis viennent 64, 32, 16, 8, 4, 2 et 1. Au fur et à mesure que les bits (les signaux on/off) sont envoyés dans l'air, ils sont recombinés en octets par le récepteur.

As-tu repéré le schéma ? (Indice : base 2.)

En ajoutant les nombres associés aux positions mises sur *on* dans un octet on peut représenter tous les nombres entre 0 et 255. L'image ci-dessous montre comment ça marche avec cinq bits pour compter de 0 à 31 :

Si on se met d'accord sur ce que représente chacun des 256 nombres (encodé par un octet) ~ comme des caractère par exemple ! ~ alors on peut commencer à envoyer du texte, un caractère par octet.

Et en fait, certains y ont déjà pensé ! Utiliser des octets pour encoder et décoder de l'information est très habituel. Cela correspond à peu près à la couche de protocole du code Morse dans notre exemple de réseau câblé.

Une super série d'explications claires à destination des enfants (et des enseignants) sur « tout est octet » peut être trouvée sur le site web de CS unplugged

1.13.3 Addressage

Le problème avec les ondes radio c'est qu'on ne peut pas transmettre directement à une personne. Toute personne avec l'antenne appropriée peut recevoir les messages que tu envoies. Il est donc important d'être capable de déterminer qui devrait recevoir les diffusions.

La façon dont est conçu le module radio dans le micro :bit règle le problème de façon assez simple :
— Il est possible de régler la radio sur des canaux différents (numérotés 0-100).
Ça marche exactement comme les Talkie-Walkie d'enfants : tout le monde se met sur le même canal et tout le monde entend ce que quiconque émet sur ce canal. Tout comme avec les Talkie-Walkie, il peut y avoir de petites interférences entre canaux adjacents. * Le module radio te permet de préciser deux informations : une adresse et un groupe. L'adresse est comme une adresse postale tandis qu'un groupe est comme un destinataire spécifique à cette adresse. La chose importante est que la radio va filtrer les messages qu'elle reçoit qui ne correspondent pas à *ton* adresse et à *ton* groupe. Il faut donc préparer l'adresse et le groupe que ton application va utiliser.

Bien sûr, le micro :bit reçoit aussi les messages diffusés pour les autres combinaisons d'adresse et de groupe. Mais ce qui est important c'est que tu n'as pas besoin de t'occuper de les filtrer. Néanmoins, si quelqu'un était suffisament intelligent, il pourrait lire * tout le trafic du réseau sans fils* quelque soit la cible adresse/groupe à laquelle il est destiné. Dans ce cas il est indispensable d'utiliser des moyens de communication cryptés pour que seul le destinataire souhaité

soit capable de lire le message diffusé. La cryptographie est un sujet fascinant mais, malheureusement, au-delà de la portée de ce tutoriel.

1.13.4 Lucioles

Ceci est une luciole :

C'est une sorte d'insecte qui utilise la bioluminescence pour signaler (sans fil) sa présence à ses amis. Voilà ce que ça donne lorsqu'ils communiquent :

La BBC a une video plutôt sympa de lucioles disponible en ligne.

Nous allons utiliser le module radio pour créer quelque chose se rapprochant d'un essaim de lucioles émettant des signaux les unes vers les autres.

Tout d'abord on `import radio` pour rendre les fonctions disponibles pour ton programme en Python. Ensuite on fait appel à la fonction `radio.on()` pour allumer la radio. Puisque la radio consomme de l'énergie et de la mémoire on a fait en sorte que *tu* décides quand elle est allumée ou pas (il y a, évidemment, une fonction `radio.off()`).

A ce moment là, le module radio est configuré avec des paramètres par défaut qui lui permettent d'être compatible avec d'autres plateforme qui pourrait vouloir interagir avec le BBC micro :bit. Il est possible de contrôler plusieurs des caractéristiques évoquées plus haut (comme les canaux et l'adressage) ainsi que la quantité d'énergie utilisée pour la diffusion des messages et la quantité de RAM que les message entrant dans la queue occupent. La documentation de l'API contient toutes les informations dont tu as besoin pour configurer ta radio selon tes besoins.

En supposant que nous soyons contents des réglages par défaut, le moyen le plus simple d'envoyer un message est comme ceci :

```
radio.send("un message")
```

L'exemple utilise la fonction `send` pour simplement diffuser la chaîne de caractères « un message ». Recevoir un message est encore plus simple :

```
nouveau_message = radio.receive()
```

Au fur et à mesure que les messages arrivent, il sont mis dans une file de messages. La fonction `receive` renvoie le plus ancien message de la queue sous la forme d'une chaîne de caractères, faisant ainsi de la place pour les nouveaux messages entrant. Si la queue de messages est pleine, alors les nouveaux messages entrant sont ignorés.

C'est vraiment tout ce que ça fait ! (Bien que le module radio soit également suffisament puissant pour te permettre d'envoyer n'importe quel type de données, pas seulement des chaînes de caractères. Regarde la documentation de l'API pour savoir comment ça marche.)

Armé de ces connaissances, il est facile de faire des lucioles micro :bit comme ça :

```
import radio
import random
from microbit import display, Image, button_a, sleep

# Création de la liste "flash" contenant les images de l'animation
# Comprends-tu comment ça fonctionne ?
flash = [Image().invert()*(i/9) for i in range(9, -1, -1)]

# La radio ne marchera pas sauf si on l'allume !
radio.on()
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Boucle événementielle.
while True:
    # Le bouton A envoie un message "flash"
    if button_a.was_pressed():
        radio.send('flash')  # a-ha
    # On lit tous les messages entrant
    incoming = radio.receive()
    if incoming == 'flash':
        # Si il y a un message "flash" entrant
        # on affiche l'animation du flash de luciole après une petite
        # pause de durée aléatoire.
        sleep(random.randint(50, 350))
        display.show(flash, delay=100, wait=False)
        # On re-diffuse aléatoirement le message flash après une petite
        # pause
        if random.randint(0, 9) == 0:
            sleep(500)
            radio.send('flash')  # a-ha
```

Les choses importantes se passent dans la boucle événementielle. Tout d'abord, on vérifie si le bouton A a été pressé et, si c'est le cas, on utilise la radio pour envoyer le message « flash ». Puis on lit tous les messages de la queue avec `radio.receive()`. Si il y a un message, on dort pendant un court instant aléatoire (pour rendre l'affichage plus intéressant) et on utilise `display.show()` pour animer le flash de la luciole. Enfin, pour rendre le tout un peu excitant, on choisit un nombre au hasard de sorte qu'il y ait une chance sur dix de rediffuser le message « flash » à tous le monde (c'est ce qui rend possible la propagation de l'affichage des lucioles sur plusieurs appareils). Si la re-diffusion est choisie alors on attend une demie seconde (pour être sûrs que l'affichage initial du message flash soit terminé) avant de renvoyer le signal « flash ». Puisque ce code est inséré dans une instruction `while True`, il se répète en boucle indéfiniment.

Le résultat final (en utilisant un groupe de micro :bit) devrait ressembler à ça :

```
.. image:: mb-firefly.gif
```

1.14 Et après ?

Ces tutoriels sont seulement les premiers pas dans l'utilisation de MicroPython avec le BBC micro :bit. Une analogie musicale : tu connais l'utilisation basique d'un instrument très simple et tu peux facilement jouer « Une souris verte »

C'est une réussite sur laquelle il faut t'appuyer pour aller plus loin.

Un voyage excitant t'attend pour devenir un codeur virtuose.

Tu rencontreras de la frustration, des échecs et de la folie. Dans ces moments, rappelle-toi que tu n'es pas seul. Python détient une arme secrète : la plus incroyable communauté de programmeurs de la planète. Rapproche-toi de cette communauté et tu te feras des amis, tu trouveras des mentors, vous vous aiderez les uns les autres et partagerez des ressources.

Les exemples dans ces tutoriels sont simples à expliquer mais peuvent ne pas être les mises en oeuvres les plus simples ou les plus efficaces. Nous avons laissé de côté un tas de *trucs super amusants* pour nous concentrer sur ton apprentissage des bases. Si tu veux *vraiment* savoir comment faire décoller MicroPython sur le BBC micro :bit alors lis la documentation de référence de l'API. Elle contient les informations sur *toutes* les possibilités disponibles.

Explore, expérimente et n'aies pas peur d'essayer des trucs ~ ce sont les caractéristiques d'un codeur virtuose. Pour t'encourager nous avons caché un certains nombre « d'Easter eggs » dans MicroPython et dans les éditeurs de code

(dans TouchDevelop et dans Mu). Ce sont des récompenses amusantes pour ceux qui fouillent et regardent plus loin que le bout de leur nez.

Ce genre de qualité en Python est important : c'est l'un des langages de programmation professionnels les plus populaires.

Etonne-nous avec ton code ! Fabrique des choses qui nous enchantent ! La plupart d'entre nous, nous amusons !

Bon bidouillage !

Python est l'un des langages de programmation les plus populaires du monde. Tous les jours, sans le savoir, tu utilises probablement un logiciel écrit en Python. Toutes sortes de sociétés et d'organisations utilisent Python dans une grande variété d'applications. Google, NASA, Bank of America, Disney, CERN, YouTube, Mozilla, The Guardian - la liste continue et couvre tous les secteurs de l'économie, des sciences et des arts.

Par exemple, te rappeles-tu de l'annonce de la découverte des ondes gravitationelles ? Les instruments utilisés pour faire les mesures étaient controllés avec Python.

Dit simplement, si tu enseignes ou apprends le Python, tu développes un talent de grande valeur qui s'applique à tous les domaines de l'activté humaine.

L'un de ces domaines est l'appareil micro :bit de la BBC. Il embarque une version de Python appelée MicroPython qui est destinée à tourner sur de petits ordinateurs comme le BBC micro :bit. C'est une implémentation complète de Python 3 donc lorsque tu passeras sur d'autres ordinateurs (comme programmer en Python sur une carte Raspberry Pi) tu utiliseras exactement le même langage.

MicroPython n'inclut pas toutes les bibliothèques standards qui sont présentes dans la version « habituelle » de Python. Néanmoins, nous avons créé un module spécial `microbit` dans MicroPython qui te permet de contrôler l'appareil.

Python et MicroPython sont des logiciels libres. Cela ne signifie pas seulement qu'il n'y a rien à payer pour utiliser Python, mais aussi que tu es libre de contribuer à la communauté Python. Cela peut être fait sous la forme de code, de documentation, de rapports de bugs, de gestion de groupe ou d'écriture de tutoriels (comme celui-ci). En fait, toutes les ressources Python pour le BBC micro :bit ont été créées par une équipe internationale de bénévoles travaillant sur leur temps libre.

Ces leçons présentent MicroPython et le BBC micro :bit en étapes faciles à suivre. N'hésite pas à les adopter et à les adapter pour des leçons en classe, ou peut-être simplement à les suivre par toi-même chez toi.

Tu y arriveras mieux si tu explores, expérimentes et joues. Tu ne peux pas casser un BBC micro :bit avec un code incorrect. Lance-toi !

Un mot d'avertissement : *tu vas souvent rater*, et ce n'est pas grave. **L'échec est le moyen qu'ont les bon développeurs de logiciel d'apprendre**. Ceux parmi nous qui développent des logiciels s'amusent beaucoup à rechercher les bugs et à éviter la répétition des erreurs.

En cas de doute, rappele-toi la maxime Zen de MicroPython :

```
Code,
Bidouille,
Sobriété,
Reste simple,
Petit, c'est beau,

Sois courageux! Casse les trucs! Apprend et amuse-toi!
Exprime-toi avec MicroPython.

Bon bidouillage! :-)
```

Bonne chance !

CHAPITRE 2

micro :bit Micropython API

Avertissement : As we work towards a 1.0 release, this API is subject to frequent changes. This page reflects the current micro :bit API in a developer-friendly (but not necessarily kid-friendly) way. The tutorials associated with this documentation are a good place to start for non-developers looking for information.

2.1 The microbit module

Everything directly related to interacting with the hardware lives in the *microbit* module. For ease of use it's recommended you start all scripts with :

```
from microbit import *
```

The following documentation assumes you have done this.

There are a few functions available directly :

```
# sleep for the given number of milliseconds.
sleep(ms)
# returns the number of milliseconds since the micro:bit was last switched on.
running_time()
# makes the micro:bit enter panic mode (this usually happens when the DAL runs
# out of memory, and causes a sad face to be drawn on the display). The error
# code can be any arbitrary integer value.
panic(error_code)
# resets the micro:bit.
reset()
```

The rest of the functionality is provided by objects and classes in the microbit module, as described below.

Note that the API exposes integers only (ie no floats are needed, but they may be accepted). We thus use milliseconds for the standard time unit.

2.1.1 Buttons

There are 2 buttons :

```
button_a
button_b
```

These are both objects and have the following methods :

```
# returns True or False to indicate if the button is pressed at the time of
# the method call.
button.is_pressed()
# returns True or False to indicate if the button was pressed since the device
# started or the last time this method was called.
button.was_pressed()
# returns the running total of button presses, and resets this counter to zero
button.get_presses()
```

2.1.2 The LED display

The LED display is exposed via the *display* object :

```
# gets the brightness of the pixel (x,y). Brightness can be from 0 (the pixel
# is off) to 9 (the pixel is at maximum brightness).
display.get_pixel(x, y)
# sets the brightness of the pixel (x,y) to val (between 0 [off] and 9 [max
# brightness], inclusive).
display.set_pixel(x, y, val)
# clears the display.
display.clear()
# shows the image.
display.show(image, delay=0, wait=True, loop=False, clear=False)
# shows each image or letter in the iterable, with delay ms. in between each.
display.show(iterable, delay=400, wait=True, loop=False, clear=False)
# scrolls a string across the display (more exciting than display.show for
# written messages).
display.scroll(string, delay=400)
```

2.1.3 Pins

Provide digital and analog input and output functionality, for the pins in the connector. Some pins are connected internally to the I/O that drives the LED matrix and the buttons.

Each pin is provided as an object directly in the `microbit` module. This keeps the API relatively flat, making it very easy to use :

- pin0
- pin1
- ...
- pin15
- pin16
- *Warning : P17-P18 (inclusive) are unavailable.*
- pin19
- pin20

Each of these pins are instances of the `MicroBitPin` class, which offers the following API :

```
# value can be 0, 1, False, True
pin.write_digital(value)
# returns either 1 or 0
pin.read_digital()
# value is between 0 and 1023
pin.write_analog(value)
# returns an integer between 0 and 1023
pin.read_analog()
# sets the period of the PWM output of the pin in milliseconds
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period(int)
# sets the period of the PWM output of the pin in microseconds
# (see https://en.wikipedia.org/wiki/Pulse-width_modulation)
pin.set_analog_period_microseconds(int)
# returns boolean
pin.is_touched()
```

2.1.4 Images

Note : You don't always need to create one of these yourself - you can access the image shown on the display directly with *display.image*. *display.image* is just an instance of *Image*, so you can use all of the same methods.

Images API :

```
# creates an empty 5x5 image
image = Image()
# create an image from a string - each character in the string represents an
# LED - 0 (or space) is off and 9 is maximum brightness. The colon ":"
# indicates the end of a line.
image = Image('90009:09090:00900:09090:90009:')
# create an empty image of given size
image = Image(width, height)
# initialises an Image with the specified width and height. The buffer
# should be an array of length width * height
image = Image(width, height, buffer)

# methods
# returns the image's width (most often 5)
image.width()
# returns the image's height (most often 5)
image.height()
# sets the pixel at the specified position (between 0 and 9). May fail for
# constant images.
image.set_pixel(x, y, value)
# gets the pixel at the specified position (between 0 and 9)
image.get_pixel(x, y)
# returns a new image created by shifting the picture left 'n' times.
image.shift_left(n)
# returns a new image created by shifting the picture right 'n' times.
image.shift_right(n)
# returns a new image created by shifting the picture up 'n' times.
image.shift_up(n)
# returns a new image created by shifting the picture down 'n' times.
image.shift_down(n)
```

(suite sur la page suivante)

(suite de la page précédente)

```
# get a compact string representation of the image
repr(image)
# get a more readable string representation of the image
str(image)

#operators
# returns a new image created by superimposing the two images
image + image
# returns a new image created by multiplying the brightness of each pixel by n
image * n

# built-in images.
Image.HEART
Image.HEART_SMALL
Image.HAPPY
Image.SMILE
Image.SAD
Image.CONFUSED
Image.ANGRY
Image.ASLEEP
Image.SURPRISED
Image.SILLY
Image.FABULOUS
Image.MEH
Image.YES
Image.NO
Image.CLOCK12 # clock at 12 o' clock
Image.CLOCK11
... # many clocks (Image.CLOCKn)
Image.CLOCK1 # clock at 1 o'clock
Image.ARROW_N
... # arrows pointing N, NE, E, SE, S, SW, W, NW (microbit.Image.ARROW_direction)
Image.ARROW_NW
Image.TRIANGLE
Image.TRIANGLE_LEFT
Image.CHESSBOARD
Image.DIAMOND
Image.DIAMOND_SMALL
Image.SQUARE
Image.SQUARE_SMALL
Image.RABBIT
Image.COW
Image.MUSIC_CROTCHET
Image.MUSIC_QUAVER
Image.MUSIC_QUAVERS
Image.PITCHFORK
Image.XMAS
Image.PACMAN
Image.TARGET
Image.TSHIRT
Image.ROLLERSKATE
Image.DUCK
Image.HOUSE
Image.TORTOISE
Image.BUTTERFLY
Image.STICKFIGURE
Image.GHOST
```

(suite sur la page suivante)

(suite de la page précédente)

```
Image.SWORD
Image.GIRAFFE
Image.SKULL
Image.UMBRELLA
Image.SNAKE
# built-in lists - useful for animations, e.g. display.show(Image.ALL_CLOCKS)
Image.ALL_CLOCKS
Image.ALL_ARROWS
```

2.1.5 The accelerometer

The accelerometer is accessed via the `accelerometer` object :

```
# read the X axis of the device. Measured in milli-g.
accelerometer.get_x()
# read the Y axis of the device. Measured in milli-g.
accelerometer.get_y()
# read the Z axis of the device. Measured in milli-g.
accelerometer.get_z()
# get tuple of all three X, Y and Z readings (listed in that order).
accelerometer.get_values()
# return the name of the current gesture.
accelerometer.current_gesture()
# return True or False to indicate if the named gesture is currently active.
accelerometer.is_gesture(name)
# return True or False to indicate if the named gesture was active since the
# last call.
accelerometer.was_gesture(name)
# return a tuple of the gesture history. The most recent is listed last.
accelerometer.get_gestures()
```

The recognised gestures are : `up`, `down`, `left`, `right`, `face up`, `face down`, `freefall`, `3g`, `6g`, `8g`, `shake`.

2.1.6 The compass

The compass is accessed via the *compass* object :

```
# calibrate the compass (this is needed to get accurate readings).
compass.calibrate()
# return a numeric indication of degrees offset from "north".
compass.heading()
# return an numeric indication of the strength of magnetic field around
# the micro:bit.
compass.get_field_strength()
# returns True or False to indicate if the compass is calibrated.
compass.is_calibrated()
# resets the compass to a pre-calibration state.
compass.clear_calibration()
```

2.1.7 I2C bus

There is an I2C bus on the micro :bit that is exposed via the *i2c* object. It has the following methods :

```
# read n bytes from device with addr; repeat=True means a stop bit won't
# be sent.
i2c.read(addr, n, repeat=False)
# write buf to device with addr; repeat=True means a stop bit won't be sent.
i2c.write(addr, buf, repeat=False)
```

2.1.8 UART

Use `uart` to communicate with a serial device connected to the device's I/O pins :

```
# set up communication (use pins 0 [TX] and 1 [RX]) with a baud rate of 9600.
uart.init()
# return True or False to indicate if there are incoming characters waiting to
# be read.
uart.any()
# return (read) n incoming characters.
uart.read(n)
# return (read) as much incoming data as possible.
uart.readall()
# return (read) all the characters to a newline character is reached.
uart.readline()
# read bytes into the referenced buffer.
uart.readinto(buffer)
# write bytes from the buffer to the connected device.
uart.write(buffer)
```

CHAPITRE 3

Microbit Module

The `microbit` module gives you access to all the hardware that is built-in into your board.

3.1 Functions

`microbit.`**`panic`**`(`*n*`)`
Enter a panic mode. Requires restart. Pass in an arbitrary integer <= 255 to indicate a status :

```
microbit.panic(255)
```

`microbit.`**`reset`**`()`
Restart the board.

`microbit.`**`sleep`**`(`*n*`)`
Wait for `n` milliseconds. One second is 1000 milliseconds, so :

```
microbit.sleep(1000)
```

will pause the execution for one second. `n` can be an integer or a floating point number.

`microbit.`**`running_time`**`()`
Return the number of milliseconds since the board was switched on or restarted.

`microbit.`**`temperature`**`()`
Return the temperature of the micro :bit in degrees Celcius.

3.2 Attributes

3.2.1 Buttons

There are two buttons on the board, called `button_a` and `button_b`.

Attributes

button_a
A `Button` instance (see below) representing the left button.

button_b
Represents the right button.

Classes

class Button
Represents a button.

Note : This class is not actually available to the user, it is only used by the two button instances, which are provided already initialized.

---

is_pressed()
Returns `True` if the specified button `button` is currently being held down, and `False` otherwise.

was_pressed()
Returns `True` or `False` to indicate if the button was pressed (went from up to down) since the device started or the last time this method was called. Calling this method will clear the press state so that the button must be pressed again before this method will return `True` again.

get_presses()
Returns the running total of button presses, and resets this total to zero before returning.

Example

```
import microbit

while True:
    if microbit.button_a.is_pressed() and microbit.button_b.is_pressed():
        microbit.display.scroll("AB")
        break
    elif microbit.button_a.is_pressed():
        microbit.display.scroll("A")
    elif microbit.button_b.is_pressed():
        microbit.display.scroll("B")
    microbit.sleep(100)
```

3.2.2 Input/Output Pins

The pins are your board's way to communicate with external devices connected to it. There are 19 pins for your disposal, numbered 0-16 and 19-20. Pins 17 and 18 are not available.

For example, the script below will change the display on the micro :bit depending upon the digital reading on pin 0 :

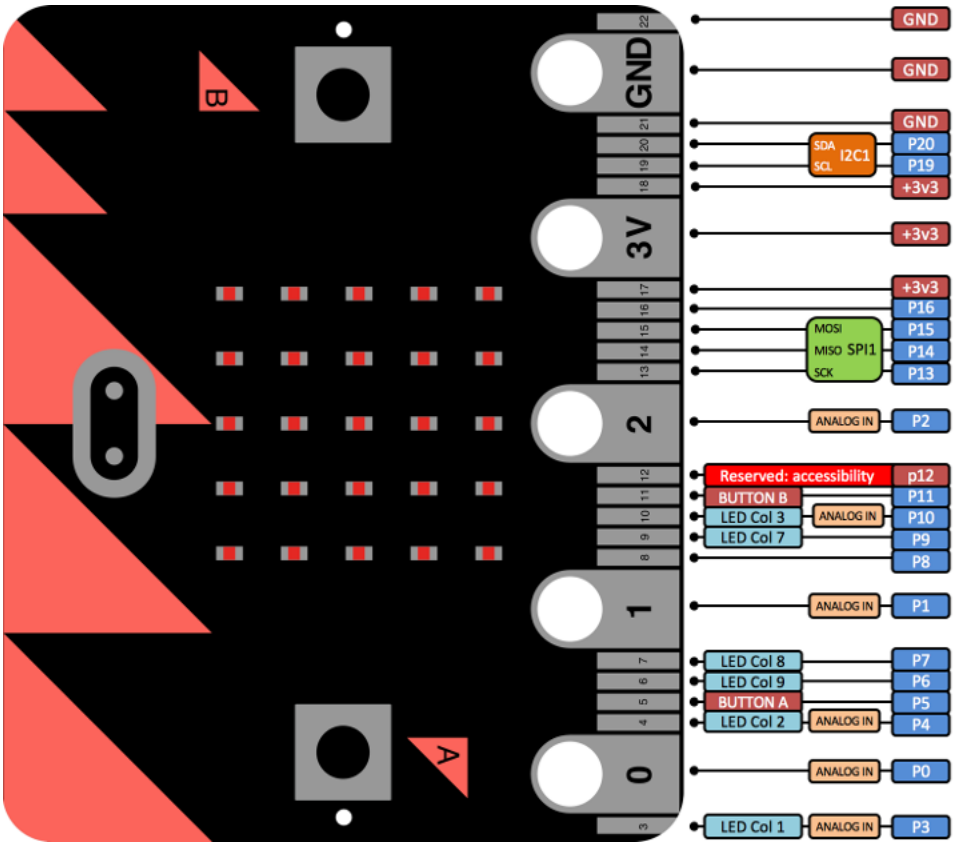
```
from microbit import *


while True:
    if pin0.read_digital():
        display.show(Image.HAPPY)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    else:
        display.show(Image.SAD)
```

Pin Functions



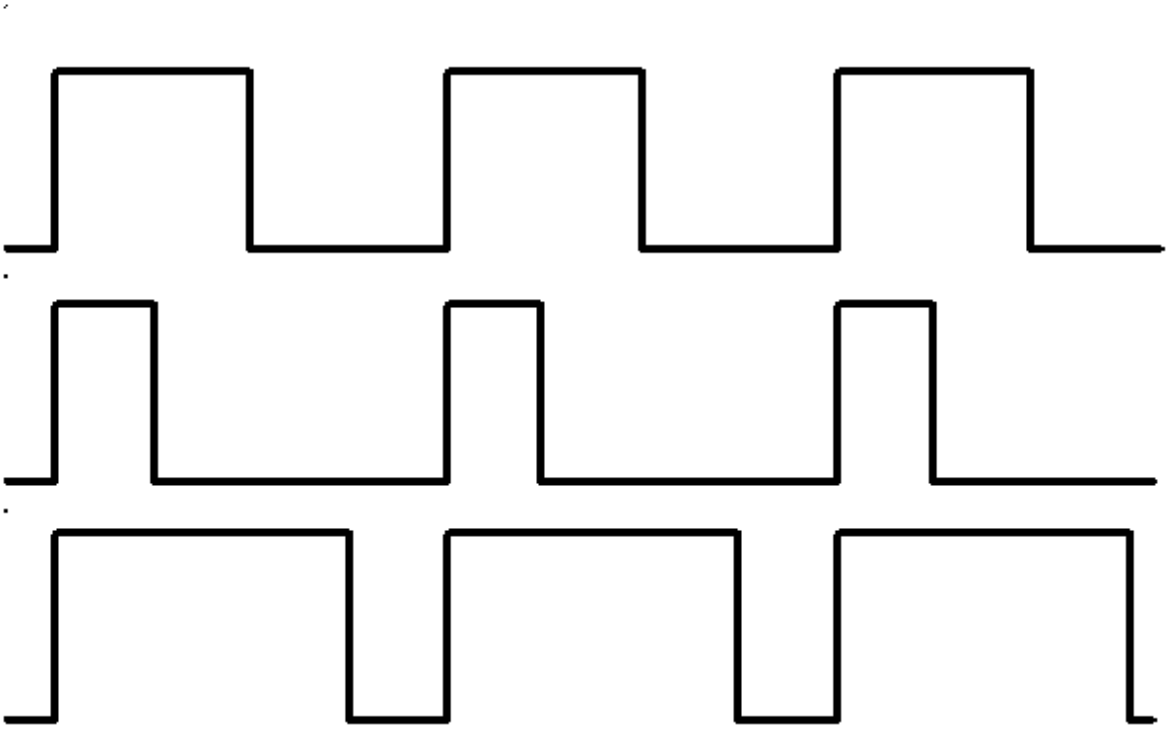
Those pins are available as attributes on the `microbit` module :`microbit.pin0` - `microbit.pin20`.

Pin	Type	Function
0	Touch	Pad 0
1	Touch	Pad 1
2	Touch	Pad 2
3	Analog	Column 1
4	Analog	Column 2
5	Digital	Button A
6	Digital	Row 2
7	Digital	Row 1
8	Digital	
9	Digital	Row 3
10	Analog	Column 3
11	Digital	Button B
12	Digital	
13	Digital	SPI MOSI
14	Digital	SPI MISO
15	Digital	SPI SCK
16	Digital	
19	Digital	I2C SCL
20	Digital	I2C SDA

The above table summarizes the pins available, their types (see below) and what they are internally connected to.

Pulse-Width Modulation

The pins of your board cannot output analog signal the way an audio amplifier can do it – by modulating the voltage on the pin. Those pins can only either enable the full 3.3V output, or pull it down to 0V. However, it is still possible to control the brightness of LEDs or speed of an electric motor, by switching that voltage on and off very fast, and controlling how long it is on and how long it is off. This technique is called Pulse-Width Modulation (PWM), and that's what the `write_analog` method below does.



Above you can see the diagrams of three different PWM signals. All of them have the same period (and thus frequency), but they have different duty cycles.

The first one would be generated by `write_analog(511)`, as it has exactly 50% duty – the power is on half of the time, and off half of the time. The result of that is that the total energy of this signal is the same, as if it was 1.65V instead of 3.3V.

The second signal has 25% duty cycle, and could be generated with `write_analog(255)`. It has similar effect as if 0.825V was being output on that pin.

The third signal has 75% duty cycle, and can be generated with `write_analog(767)`. It has three times as much energy, as the second signal, and is equivalent to outputting 2.475V on th pin.

Note that this works well with devices such as motors, which have huge inertia by themselves, or LEDs, which blink too fast for the human eye to see the difference, but will not work so good with generating sound waves. This board can only generate square wave sounds on itself, which sound pretty much like the very old computer games – mostly because those games also only could do that.

Classes

There are three kinds of pins, differing in what is available for them. They are represented by the classes listed below. Note that they form a hierarchy, so that each class has all the functionality of the previous class, and adds its own to that.

Note : Those classes are not actually available for the user, you can't create new instances of them. You can only use the instances already provided, representing the physical pins on your board.

---

class microbit.**MicroBitDigitalPin**

read_digital()
Return 1 if the pin is high, and 0 if it's low.

write_digital(*value*)
Set the pin to high if `value` is 1, or to low, if it is 0.

class microbit.**MicroBitAnalogDigitalPin**

read_analog()
Read the voltage applied to the pin, and return it as an integer between 0 (meaning 0V) and 1023 (meaning 3.3V).

write_analog(*value*)
Output a PWM signal on the pin, with the duty cycle proportional to the provided `value`. The `value` may be either an integer or a floating point number between 0 (0% duty cycle) and 1023 (100% duty).

set_analog_period(*period*)
Set the period of the PWM signal being output to `period` in milliseconds. The minimum valid value is 1ms.

set_analog_period_microseconds(*period*)
Set the period of the PWM signal being output to `period` in microseconds. The minimum valid value is 256µs.

class microbit.**MicroBitTouchPin**

is_touched()
Return `True` if the pin is being touched with a finger, otherwise return `False`.
This test is done by measuring how much resistance there is between the pin and ground. A low resistance gives a reading of `True`. To get a reliable reading using a finger you may need to touch the ground pin with another part of your body, for example your other hand.

The pull mode for a pin is automatically configured when the pin changes to an input mode. Input modes are when you call `read_analog` / `read_digital` / `is_touched`. The default pull mode for these is, respectively, `NO_PULL`, `PULL_DOWN`, `PULL_UP`. Calling `set_pull` will configure the pin to be in `read_digital` mode with the given pull mode.

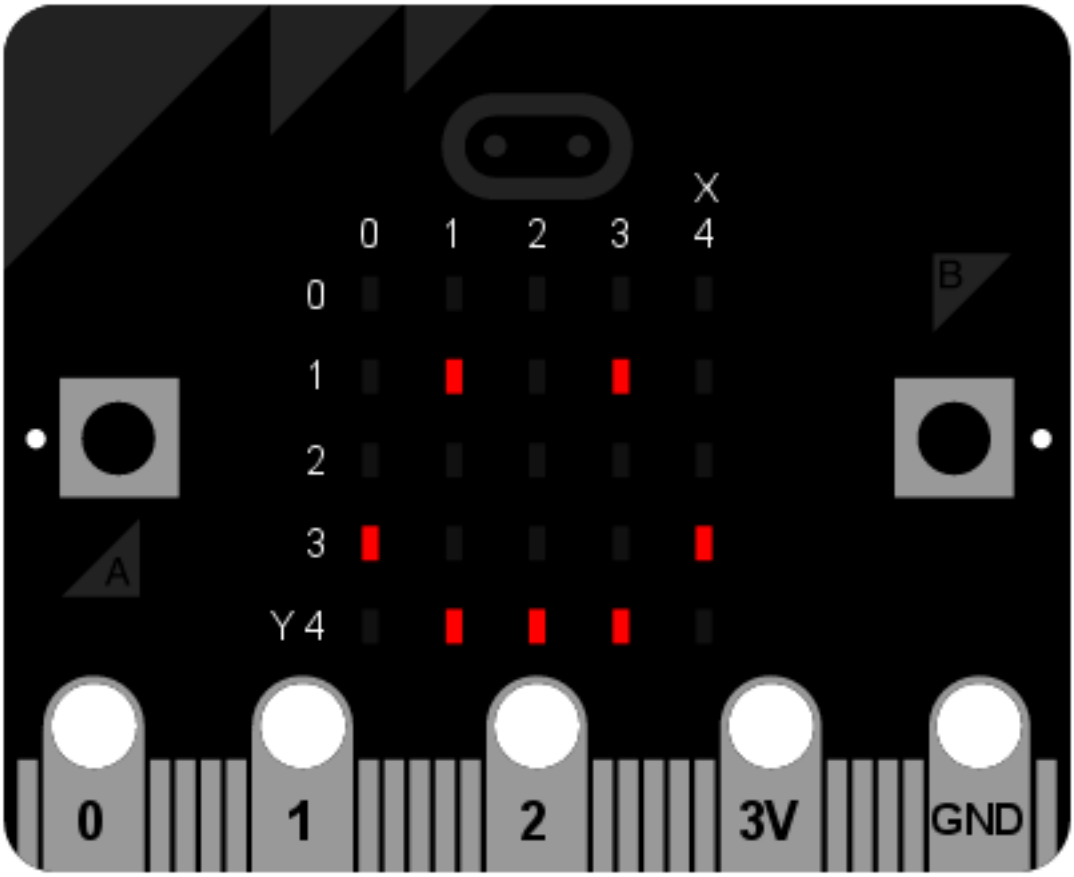
Note : Also note, the micro :bit has external weak (10M) pull-ups fitted on pins 0, 1 and 2 only, in order for the touch sensing to work. See the edge connector data sheet here : http ://tech.microbit.org/hardware/edgeconnector_ds/

3.3 Classes

3.3.1 Image

The `Image` class is used to create images that can be displayed easily on the device's LED matrix. Given an image object it's possible to display it via the `display` API :

```
display.show(Image.HAPPY)
```



Classes

class microbit.**Image**(*string*)
class microbit.**Image**(*width=None*, *height=None*, *buffer=None*)

If `string` is used, it has to consist of digits 0-9 arranged into lines, describing the image, for example :

```
image = Image("90009:"
              "09090:"
              "00900:"
              "09090:"
              "90009")
```

will create a 5×5 image of an X. The end of a line is indicated by a colon. It's also possible to use a newline (n) to indicate the end of a line like this :

```
image = Image("90009\n"
              "09090\n"
              "00900\n"
              "09090\n"
              "90009")
```

The other form creates an empty image with `width` columns and `height` rows. Optionally `buffer` can be an array of `width``×``height` integers in range 0-9 to initialize the image.

width()

Return the number of columns in the image.

height()
Return the numbers of rows in the image.

set_pixel(*x*, *y*, *value*)
Set the brightness of the pixel at column `x` and row `y` to the `value`, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

get_pixel(*x*, *y*)
Return the brightness of pixel at column `x` and row `y` as an integer between 0 and 9.

shift_left(*n*)
Return a new image created by shifting the picture left by `n` columns.

shift_right(*n*)
Same as `image.shift_left(-n)`.

shift_up(*n*)
Return a new image created by shifting the picture up by `n` rows.

shift_down(*n*)
Same as `image.shift_up(-n)`.

crop(*x*, *y*, *w*, *h*)
Return a new image by cropping the picture to a width of `w` and a height of `h`, starting with the pixel at column `x` and row `y`.

copy()
Return an exact copy of the image.

invert()
Return a new image by inverting the brightness of the pixels in the source image.

fill(*value*)
Set the brightness of all the pixels in the image to the `value`, which has to be between 0 (dark) and 9 (bright).

This method will raise an exception when called on any of the built-in read-only images, like `Image.HEART`.

blit(*src*, *x*, *y*, *w*, *h*, *xdest=0*, *ydest=0*)
Copy the rectangle defined by `x`, `y`, `w`, `h` from the image `src` into this image at `xdest`, `ydest`. Areas in the source rectangle, but outside the source image are treated as having a value of 0.

`shift_left()`, `shift_right()`, `shift_up()`, `shift_down()` and `crop()` can are all implemented by using `blit()`. For example, img.crop(x, y, w, h) can be implemented as :

```
def crop(self, x, y, w, h):
    res = Image(w, h)
    res.blit(self, x, y, w, h)
    return res
```

Attributes

The `Image` class also has the following built-in instances of itself included as its attributes (the attribute names indicate what the image represents) :

- `Image.HEART`
- `Image.HEART_SMALL`
- `Image.HAPPY`
- `Image.SMILE`
- `Image.SAD`
- `Image.CONFUSED`
- `Image.ANGRY`

- Image.ASLEEP
- Image.SURPRISED
- Image.SILLY
- Image.FABULOUS
- Image.MEH
- Image.YES
- Image.NO
- Image.CLOCK12, Image.CLOCK11, Image.CLOCK10, Image.CLOCK9, Image.CLOCK8, Image.CLOCK7, Image.CLOCK6, Image.CLOCK5, Image.CLOCK4, Image.CLOCK3, Image.CLOCK2, Image.CLOCK1
- Image.ARROW_N, Image.ARROW_NE, Image.ARROW_E, Image.ARROW_SE, Image.ARROW_S, Image.ARROW_SW, Image.ARROW_W, Image.ARROW_NW
- Image.TRIANGLE
- Image.TRIANGLE_LEFT
- Image.CHESSBOARD
- Image.DIAMOND
- Image.DIAMOND_SMALL
- Image.SQUARE
- Image.SQUARE_SMALL
- Image.RABBIT
- Image.COW
- Image.MUSIC_CROTCHET
- Image.MUSIC_QUAVER
- Image.MUSIC_QUAVERS
- Image.PITCHFORK
- Image.XMAS
- Image.PACMAN
- Image.TARGET
- Image.TSHIRT
- Image.ROLLERSKATE
- Image.DUCK
- Image.HOUSE
- Image.TORTOISE
- Image.BUTTERFLY
- Image.STICKFIGURE
- Image.GHOST
- Image.SWORD
- Image.GIRAFFE
- Image.SKULL
- Image.UMBRELLA
- Image.SNAKE

Finally, related collections of images have been grouped together :

```
* ``Image.ALL_CLOCKS``
* ``Image.ALL_ARROWS``
```

Operations

```
repr(image)
```

Get a compact string representation of the image.

```
str(image)
```

Get a readable string representation of the image.

```
image1 + image2
```

Create a new image by adding the brightness values from the two images for each pixel.

```
image * n
```

Create a new image by multiplying the brightness of each pixel by `n`.

3.4 Modules

3.4.1 Display

This module controls the 5×5 LED display on the front of your board. It can be used to display images, animations and even text.

Functions

`microbit.display.`**`get_pixel`**(*x*, *y*)
Return the brightness of the LED at column `x` and row `y` as an integer between 0 (off) and 9 (bright).

`microbit.display.`**`set_pixel`**(*x*, *y*, *value*)
Set the brightness of the LED at column `x` and row `y` to `value`, which has to be an integer between 0 and 9.

`microbit.display.`**`clear`**()
Set the brightness of all LEDs to 0 (off).

`microbit.display.`**`show`**(*image*)
Display the `image`.

`microbit.display.`**`show`**(*value*, *delay=400*, *, *wait=True*, *loop=False*, *clear=False*)
If `value` is a string, float or integer, display letters/digits in sequence. Otherwise, if `value` is an iterable sequence of images, display these images in sequence. Each letter, digit or image is shown with `delay` milliseconds between them.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `clear` is `True`, the display will be cleared after the iterable has finished.

Note that the `wait`, `loop` and `clear` arguments must be specified using their keyword.

Note : If using a generator as the `iterable`, then take care not to allocate any memory in the generator as allocating memory in an interrupt is prohibited and will raise a `MemoryError`.

`microbit.display.`**`scroll`**(*value*, *delay=150*, *, *wait=True*, *loop=False*, *monospace=False*)
Scrolls `value` horizontally on the display. If `value` is an integer or float it is first converted to a string using `str()`. The `delay` parameter controls how fast the text is scrolling.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `monospace` is `True`, the characters will all take up 5 pixel-columns in width, otherwise there will be exactly 1 blank pixel-column between each character as they scroll.

Note that the `wait`, `loop` and `monospace` arguments must be specified using their keyword.

`microbit.display.`**`on`**`()`
Use on() to turn on the display.

`microbit.display.`**`off`**`()`
Use off() to turn off the display (thus allowing you to re-use the GPIO pins associated with the display for other purposes).

`microbit.display.`**`is_on`**`()`
Returns `True` if the display is on, otherwise returns `False`.

`microbit.display.`**`read_light_level`**`()`
Use the display's LEDs in reverse-bias mode to sense the amount of light falling on the display. Returns an integer between 0 and 255 representing the light level, with larger meaning more light.

Example

To continuously scroll a string across the display, and do it in the background, you can use :

```
import microbit

microbit.display.scroll('Hello!', wait=False, loop=True)
```

3.4.2 UART

The `uart` module lets you talk to a device connected to your board using a serial interface.

Functions

`microbit.uart.`**`init`**(*baudrate=9600*, *bits=8*, *parity=None*, *stop=1*, ***, *tx=None*, *rx=None*)
Initialize serial communication with the specified parameters on the specified `tx` and `rx` pins. Note that for correct communication, the parameters have to be the same on both communicating devices.

> **Avertissement :** Initializing the UART on external pins will cause the Python console on USB to become unaccessible, as it uses the same hardware. To bring the console back you must reinitialize the UART without passing anything for `tx` or `rx` (or passing `None` to these arguments). This means that calling `uart.init(115200)` is enough to restore the Python console.

The `baudrate` defines the speed of communication. Common baud rates include :
— 9600
— 14400
— 19200
— 28800
— 38400
— 57600
— 115200

The `bits` defines the size of bytes being transmitted, and the board only supports 8. The `parity` parameter defines how parity is checked, and it can be `None`, `microbit.uart.ODD` or `microbit.uart.EVEN`. The `stop` parameter tells the number of stop bits, and has to be 1 for this board.

If `tx` and `rx` are not specified then the internal USB-UART TX/RX pins are used which connect to the USB serial convertor on the micro :bit, thus connecting the UART to your PC. You can specify any other pins you want by passing the desired pin objects to the `tx` and `rx` parameters.

Note : When connecting the device, make sure you « cross » the wires – the TX pin on your board needs to be connected with the RX pin on the device, and the RX pin – with the TX pin on the device. Also make sure the ground pins of both devices are connected.

`uart.`**`any`**`()`
Return `True` if any characters waiting, else `False`.

`uart.`**`read`**`(`[*nbytes*]`)`
Read characters. If `nbytes` is specified then read at most that many bytes.

`uart.`**`readall`**`()`
Read as much data as possible.

Return value : a bytes object or `None` on timeout.

`uart.`**`readinto`**`(`*buf*[, *nbytes*]`)`
Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value : number of bytes read and stored into `buf` or `None` on timeout.

`uart.`**`readline`**`()`
Read a line, ending in a newline character.

Return value : the line read or `None` on timeout. The newline character is included in the returned bytes.

`uart.`**`write`**`(`*buf*`)`
Write the buffer of bytes to the bus.

Return value : number of bytes written or `None` on timeout.

3.4.3 SPI

The `spi` module lets you talk to a device connected to your board using a serial peripheral interface (SPI) bus. SPI uses a so-called master-slave architecture with a single master. You will need to specify the connections for three signals :

— SCLK : Serial Clock (output from master).
— MOSI : Master Output, Slave Input (output from master).
— MISO : Master Input, Slave Output (output from slave).

Functions

`microbit.spi.`**`init`**`(`*baudrate=1000000*, *bits=8*, *mode=0*, *sclk=pin13*, *mosi=pin15*, *miso=pin14*`)`
Initialize SPI communication with the specified parameters on the specified `pins`. Note that for correct communication, the parameters have to be the same on both communicating devices.

The `baudrate` defines the speed of communication.

The `bits` defines the size of bytes being transmitted. Currently only `bits=8` is supported. However, this may change in the future.

The `mode` determines the combination of clock polarity and phase according to the following convention, with polarity as the high order bit and phase as the low order bit :

SPI Mode	Polarity (CPOL)	Phase (CPHA)
0	0	0
1	0	1
2	1	0
3	1	1

Polarity (aka CPOL) 0 means that the clock is at logic value 0 when idle and goes high (logic value 1) when active ; polarity 1 means the clock is at logic value 1 when idle and goes low (logic value 0) when active. Phase (aka CPHA) 0 means that data is sampled on the leading edge of the clock, and 1 means on the trailing edge (viz. https://en.wikipedia.org/wiki/Signal_edge).

The `sclk`, `mosi` and `miso` arguments specify the pins to use for each type of signal.

`spi.`**`read`**(*nbytes*)
Read at most `nbytes`. Returns what was read.

`spi.`**`write`**(*buffer*)
Write the `buffer` of bytes to the bus.

`spi.`**`write_readinto`**(*out*, *in*)
Write the `out` buffer to the bus and read any response into the `in` buffer. The length of the buffers should be the same. The buffers can be the same object.

3.4.4 I²C

The `i2c` module lets you communicate with devices connected to your board using the I²C bus protocol. There can be multiple slave devices connected at the same time, and each one has its own unique address, that is either fixed for the device or configured on it. Your board acts as the I²C master.

We use 7-bit addressing for devices because of the reasons stated here.

This may be different to other micro :bit related solutions.

How exactly you should communicate with the devices, that is, what bytes to send and how to interpret the responses, depends on the device in question and should be described separately in that device's documentation.

Functions

`microbit.i2c.`**`init`**(*freq=100000*, *sda=pin20*, *scl=pin19*)
Re-initialize peripheral with the specified clock frequency `freq` on the specified `sda` and `scl` pins.

> **Avertissement :** Changing the I²C pins from defaults will make the accelerometer and compass stop working, as they are connected internally to those pins.

`microbit.i2c.`**`scan`**()
Scan the bus for devices. Returns a list of 7-bit addresses corresponding to those devices that responded to the scan.

`microbit.i2c.`**`read`**(*addr*, *n*, *repeat=False*)
Read `n` bytes from the device with 7-bit address `addr`. If `repeat` is `True`, no stop bit will be sent.

`microbit.i2c.`**`write`**(*addr*, *buf*, *repeat=False*)
Write bytes from `buf` to the device with 7-bit address `addr`. If `repeat` is `True`, no stop bit will be sent.

Connecting

You should connect the device's `SCL` pin to micro :bit pin 19, and the device's `SDA` pin to micro :bit pin 20. You also must connect the device's ground to the micro :bit ground (pin `GND`). You may need to power the device using an external power supply or the micro :bit.

There are internal pull-up resistors on the I^2C lines of the board, but with particularly long wires or large number of devices you may need to add additional pull-up resistors, to ensure noise-free communication.

3.4.5 Accelerometer

This object gives you access to the on-board accelerometer. The accelerometer also provides convenience functions for detecting gestures. The recognised gestures are : `up`, `down`, `left`, `right`, `face up`, `face down`, `freefall`, `3g`, `6g`, `8g`, `shake`.

Functions

`microbit.accelerometer.`**`get_x`**`()`
Get the acceleration measurement in the `x` axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.`**`get_y`**`()`
Get the acceleration measurement in the `y` axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.`**`get_z`**`()`
Get the acceleration measurement in the `z` axis, as a positive or negative integer, depending on the direction.

`microbit.accelerometer.`**`get_values`**`()`
Get the acceleration measurements in all axes at once, as a three-element tuple of integers ordered as X, Y, Z.

`microbit.accelerometer.`**`current_gesture`**`()`
Return the name of the current gesture.

Note : MicroPython understands the following gesture names : `"up"`, `"down"`, `"left"`, `"right"`, `"face up"`, `"face down"`, `"freefall"`, `"3g"`, `"6g"`, `"8g"`, `"shake"`. Gestures are always represented as strings.

`microbit.accelerometer.`**`is_gesture`**`(`*name*`)`
Return True or False to indicate if the named gesture is currently active.

`microbit.accelerometer.`**`was_gesture`**`(`*name*`)`
Return True or False to indicate if the named gesture was active since the last call.

`microbit.accelerometer.`**`get_gestures`**`()`
Return a tuple of the gesture history. The most recent is listed last. Also clears the gesture history before returning.

Examples

A fortune telling magic 8-ball. Ask a question then shake the device for an answer.

```
# Magic 8 ball by Nicholas Tollervey. February 2016.
#
# Ask a question then shake.
#
```

(suite sur la page suivante)

(suite de la page précédente)

```
# This program has been placed into the public domain.
from microbit import *
import random

answers = [
    "It is certain",
    "It is decidedly so",
    "Without a doubt",
    "Yes, definitely",
    "You may rely on it",
    "As I see it, yes",
    "Most likely",
    "Outlook good",
    "Yes",
    "Signs point to yes",
    "Reply hazy try again",
    "Ask again later",
    "Better not tell you now",
    "Cannot predict now",
    "Concentrate and ask again",
    "Don't count on it",
    "My reply is no",
    "My sources say no",
    "Outlook not so good",
    "Very doubtful",
]

while True:
    display.show('8')
    if accelerometer.was_gesture('shake'):
        display.clear()
        sleep(1000)
        display.scroll(random.choice(answers))
    sleep(10)
```

Simple Slalom. Move the device to avoid the obstacles.

```
# Simple Slalom by Larry Hastings, September 2015
#
# This program has been placed into the public domain.

import microbit as m
import random

p = m.display.show

min_x = -1024
max_x = 1024
range_x = max_x - min_x

wall_min_speed = 400
player_min_speed = 200

wall_max_speed = 100
player_max_speed = 50

speed_max = 12
```

(suite sur la page suivante)

(suite de la page précédente)

```
while True:

    i = m.Image('00000:'*5)
    s = i.set_pixel

    player_x = 2

    wall_y = -1
    hole = 0

    score = 0
    handled_this_wall = False

    wall_speed = wall_min_speed
    player_speed = player_min_speed

    wall_next = 0
    player_next = 0

    while True:
        t = m.running_time()
        player_update = t >= player_next
        wall_update = t >= wall_next
        if not (player_update or wall_update):
            next_event = min(wall_next, player_next)
            delta = next_event - t
            m.sleep(delta)
            continue

        if wall_update:
            # calculate new speeds
            speed = min(score, speed_max)
            wall_speed = wall_min_speed + int((wall_max_speed - wall_min_speed) *␣
↪speed / speed_max)
            player_speed = player_min_speed + int((player_max_speed - player_min_
↪speed) * speed / speed_max)

            wall_next = t + wall_speed
            if wall_y < 5:
                # erase old wall
                use_wall_y = max(wall_y, 0)
                for wall_x in range(5):
                    if wall_x != hole:
                        s(wall_x, use_wall_y, 0)

        wall_reached_player = (wall_y == 4)
        if player_update:
            player_next = t + player_speed
            # find new x coord
            x = m.accelerometer.get_x()
            x = min(max(min_x, x), max_x)
            # print("x accel", x)
            s(player_x, 4, 0) # turn off old pixel
            x = ((x - min_x) / range_x) * 5
            x = min(max(0, x), 4)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            x = int(x + 0.5)
            # print("have", position, "want", x)

            if not handled_this_wall:
                if player_x < x:
                    player_x += 1
                elif player_x > x:
                    player_x -= 1
            # print("new", position)
            # print()

        if wall_update:
            # update wall position
            wall_y += 1
            if wall_y == 7:
                wall_y = -1
                hole = random.randrange(5)
                handled_this_wall = False

            if wall_y < 5:
                # draw new wall
                use_wall_y = max(wall_y, 0)
                for wall_x in range(5):
                    if wall_x != hole:
                        s(wall_x, use_wall_y, 6)

        if wall_reached_player and not handled_this_wall:
            handled_this_wall = True
            if (player_x != hole):
                # collision! game over!
                break
            score += 1

        if player_update:
            s(player_x, 4, 9) # turn on new pixel

        p(i)

    p(i.SAD)
    m.sleep(1000)
    m.display.scroll("Score:" + str(score))

    while True:
        if (m.button_a.is_pressed() and m.button_a.is_pressed()):
            break
        m.sleep(100)
```

3.4.6 Compass

This module lets you access the built-in electronic compass. Before using, the compass should be calibrated, otherwise the readings may be wrong.

Avertissement : Calibrating the compass will cause your program to pause until calibration is complete. Calibration consists of a little game to draw a circle on the LED display by rotating the device.

Functions

microbit.compass.**calibrate**()
: Starts the calibration process. An instructive message will be scrolled to the user after which they will need to rotate the device in order to draw a circle on the LED display.

microbit.compass.**is_calibrated**()
: Returns `True` if the compass has been successfully calibrated, and returns `False` otherwise.

microbit.compass.**clear_calibration**()
: Undoes the calibration, making the compass uncalibrated again.

microbit.compass.**get_x**()
: Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

microbit.compass.**get_y**()
: Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

microbit.compass.**get_z**()
: Gives the reading of the magnetic force on the `x` axis, as a positive or negative integer, depending on the direction of the force.

microbit.compass.**heading**()
: Gives the compass heading, calculated from the above readings, as an integer in the range from 0 to 360, representing the angle in degrees, clockwise, with north as 0.

microbit.compass.**get_field_strength**()
: Returns an integer indication of the magnitude of the magnetic field around the device.

Example

```
"""
    compass.py
    ~~~~~~~~~~

    Creates a compass.

    The user will need to calibrate the compass first. The compass uses the
    built-in clock images to display the position of the needle.

"""
from microbit import *


# Start calibrating
compass.calibrate()

# Try to keep the needle pointed in (roughly) the correct direction
while True:
    sleep(100)
    needle = ((15 - compass.heading()) // 30) % 12
    display.show(Image.ALL_CLOCKS[needle])
```

CHAPITRE 4

Bluetooth

While the BBC micro :bit has hardware capable of allowing the device to work as a Bluetooth Low Energy (BLE) device, it only has 16k of RAM. The BLE stack alone takes up 12k RAM which means there's not enough room to run MicroPython.

Future versions of the device may come with 32k RAM which would be sufficient. However, until such time it's highly unlikely MicroPython will support BLE.

Note : MicroPython uses the radio hardware with the `radio` module. This allows users to create simple yet effective wireless networks of micro :bit devices.

Furthermore, the protocol used in the `radio` module is a lot simpler than BLE, making it far easier to use in an educational context.

CHAPITRE 5

Local Persistent File System

It is useful to store data in a persistent manner so that it remains intact between restarts of the device. On traditional computers this is often achieved by a file system consisting of named files that hold raw data, and named directories that contain files. Python supports the various operations needed to work with such file systems.

However, since the micro :bit is a limited device in terms of both hardware and storage capacity MicroPython provides a small subset of the functions needed to persist data on the device. Because of memory constraints **there is approximately 30k of storage available** on the file system.

> **Avertissement :** Re-flashing the device will DESTROY YOUR DATA.
>
> Since the file system is stored in the micro :bit's flash memory and flashing the device rewrites all the available flash memory then all your data will be lost if you flash your device.
>
> However, if you switch your device off the data will remain intact until you either delete it (see below) or re-flash the device.

MicroPython on the micro :bit provides a flat file system ; i.e. there is no notion of a directory hierarchy, the file system is just a list of named files. Reading and writing a file is achieved via the standard Python `open` function and the resulting file-like object (representing the file) of types `TextIO` or `BytesIO`. Operations for working with files on the file system (for example, listing or deleting files) are contained within the *os* module.

If a file ends in the `.py` file extension then it can be imported. For example, a file named `hello.py` can be imported like this : `import hello`.

An example session in the MicroPython REPL may look something like this :

```
>>> with open('hello.py', 'w') as hello:
...     hello.write("print('Hello')")
...
>>> import hello
Hello
>>> with open('hello.py') as hello:
...   print(hello.read())
```

(suite sur la page suivante)

(suite de la page précédente)

```
...
print('Hello')
>>> import os
>>> os.listdir()
['hello.py']
>>> os.remove('hello.py')
>>> os.listdir()
[]
```

open (*filename*, *mode='r'*)

Returns a file object representing the file named in the argument `filename`. The mode defaults to `'r'` which means open for reading in text mode. The other common mode is `'w'` for writing (overwriting the content of the file if it already exists). Two other modes are available to be used in conjunction with the ones describes above : `'t'` means text mode (for reading and writing strings) and `'b'` means binary mode (for reading and writing bytes). If these are not specified then `'t'` (text mode) is assumed. When in text mode the file object will be an instance of `TextIO`. When in binary mode the file object will be an instance of `BytesIO`. For example, use `'rb'` to read binary data from a file.

class TextIO

class BytesIO

Instances of these classes represent files in the micro :bit's flat file system. The TextIO class is used to represent text files. The BytesIO class is used to represent binary files. They work in exactly the same except that TextIO works with strings and BytesIO works with bytes.

You do not directly instantiate these classes. Rather, an appropriately configured instance of the class is returned by the `open` function described above.

close()

Flush and close the file. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise an exception.

name()

Returns the name of the file the object represents. This will be the same as the `filename` argument passed into the call to the `open` function that instantiated the object.

read(*size*)

Read and return at most `size` characters as a single string or `size` bytes from the file. As a convenience, if `size` is unspecified or -1, all the data contained in the file is returned. Fewer than `size` characters or bytes may be returned if there are less than `size` characters or bytes remaining to be read from the file.

If 0 characters or bytes are returned, and `size` was not 0, this indicates end of file.

A `MemoryError` exception will occur if `size` is larger than the available RAM.

readinto(*buf*, *n=-1*)

Read characters or bytes into the buffer `buf`. If `n` is supplied, read `n` number of bytes or characters into the buffer `buf`.

readline(*size*)

Read and return one line from the file. If `size` is specified, at most `size` characters will be read.

The line terminator is always `'\n'` for strings or `b'\n'` for bytes.

writable()

Return `True` if the file supports writing. If `False`, `write()` will raise `OSError`.

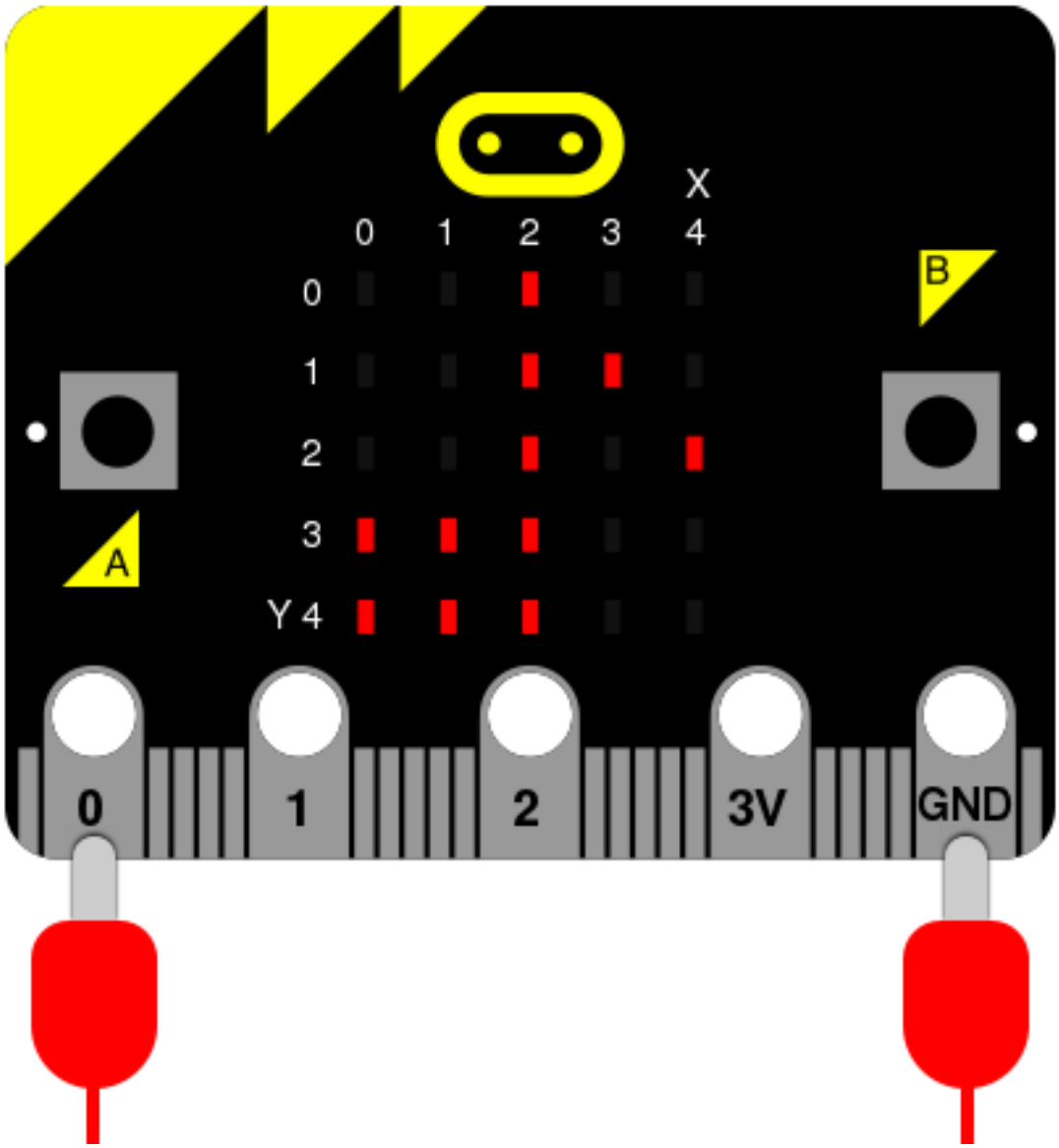
write(*buf*)

Write the string or bytes `buf` to the file and return the number of characters or bytes written.

CHAPITRE 6

Music

This is the `music` module. You can use it to play simple tunes, provided that you connect a speaker to your board. By default the `music` module expects the speaker to be connected via pin 0 :



This arrangement can be overridden (as discussed below).

To access this module you need to :

```
import music
```

We assume you have done this for the examples below.

6.1 Musical Notation

An individual note is specified thus :

```
NOTE[octave][:duration]
```

For example, `A1:4` refers to the note « A » in octave 1 that lasts for four ticks (a tick is an arbitrary length of time defined by a tempo setting function - see below). If the note name `R` is used then it is treated as a rest (silence).

Accidentals (flats and sharps) are denoted by the `b` (flat - a lower case b) and `#` (sharp - a hash symbol). For example, `Ab` is A-flat and `C#` is C-sharp.

Note names are case-insensitive.

The `octave` and `duration` parameters are states that carry over to subsequent notes until re-specified. The default states are `octave = 4` (containing middle C) and `duration = 4` (a crotchet, given the default tempo settings - see below).

For example, if 4 ticks is a crotchet, the following list is crotchet, quaver, quaver, crotchet based arpeggio :

```
['c1:4', 'e:2', 'g', 'c2:4']
```

The opening of Beethoven's 5th Symphony would be encoded thus :

```
['r4:2', 'g', 'g', 'g', 'eb:8', 'r:2', 'f', 'f', 'f', 'd:8']
```

The definition and scope of an octave conforms to the table listed on this page about scientific pitch notation. For example, middle « C » is `'c4'` and concert « A » (440) is `'a4'`. Octaves start on the note « C ».

6.2 Functions

`music.`**`set_tempo`**(*ticks=4*, *bpm=120*)

Sets the approximate tempo for playback.

A number of ticks (expressed as an integer) constitute a beat. Each beat is to be played at a certain frequency per minute (expressed as the more familiar BPM - beats per minute - also as an integer).

Suggested default values allow the following useful behaviour :
— `music.set_tempo()` - reset the tempo to default of ticks = 4, bpm = 120
— `music.set_tempo(ticks=8)` - change the « definition » of a beat
— `music.set_tempo(bpm=180)` - just change the tempo

To work out the length of a tick in milliseconds is very simple arithmetic : `60000/bpm/ticks_per_beat` . For the default values that's `60000/120/4 = 125 milliseconds` or `1 beat = 500 milliseconds`.

`music.`**`get_tempo`**()

Gets the current tempo as a tuple of integers : `(ticks, bpm)`.

`music.`**`play`**(*music*, *pin=microbit.pin0*, *wait=True*, *loop=False*)

Plays `music` containing the musical DSL defined above.

If `music` is a string it is expected to be a single note such as, `'c1:4'`.

If `music` is specified as a list of notes (as defined in the section on the musical DSL, above) then they are played one after the other to perform a melody.

In both cases, the `duration` and `octave` values are reset to their defaults before the music (whatever it may be) is played.

An optional argument to specify the output pin can be used to override the default of `microbit.pin0`.

If `wait` is set to `True`, this function is blocking.

If `loop` is set to `True`, the tune repeats until `stop` is called (see below) or the blocking call is interrupted.

`music.`**`pitch`**(*frequency*, *duration=-1*, *pin=microbit.pin0*, *wait=True*)

Plays a pitch at the integer frequency given for the specified number of milliseconds. For example, if the frequency is set to 440 and the length to 1000 then we hear a standard concert A for one second.

If `wait` is set to `True`, this function is blocking.

If `duration` is negative the pitch is played continuously until either the blocking call is interrupted or, in the case of a background call, a new frequency is set or `stop` is called (see below).

`music.`**`stop`**(*pin=microbit.pin0*)
Stops all music playback on a given pin.

`music.`**`reset`**()
Resets the state of the following attributes in the following way :
— `ticks = 4`
— `bpm = 120`
— `duration = 4`
— `octave = 4`

6.2.1 Built in Melodies

For the purposes of education and entertainment, the module contains several example tunes that are expressed as Python lists. They can be used like this :

```
>>> import music
>>> music.play(music.NYAN)
```

All the tunes are either out of copyright, composed by Nicholas H.Tollervey and released to the public domain or have an unknown composer and are covered by a fair (educational) use provision.

They are :
— `DADADADUM` - the opening to Beethoven's 5th Symphony in C minor.
— `ENTERTAINER` - the opening fragment of Scott Joplin's Ragtime classic « The Entertainer ».
— `PRELUDE` - the opening of the first Prelude in C Major of J.S.Bach's 48 Preludes and Fugues.
— `ODE` - the « Ode to Joy » theme from Beethoven's 9th Symphony in D minor.
— `NYAN` - the Nyan Cat theme (http://www.nyan.cat/). The composer is unknown. This is fair use for educational porpoises (as they say in New York).
— `RINGTONE` - something that sounds like a mobile phone ringtone. To be used to indicate an incoming message.
— `FUNK` - a funky bass line for secret agents and criminal masterminds.
— `BLUES` - a boogie-woogie 12-bar blues walking bass.
— `BIRTHDAY` - « Happy Birthday to You… » for copyright status see : http://www.bbc.co.uk/news/world-us-canada-34332853
— `WEDDING` - the bridal chorus from Wagner's opera « Lohengrin ».
— `FUNERAL` - the « funeral march » otherwise known as Frédéric Chopin's Piano Sonata No. 2 in B minor, Op. 35.
— `PUNCHLINE` - a fun fragment that signifies a joke has been made.
— `PYTHON` - John Philip Sousa's march « Liberty Bell » aka, the theme for « Monty Python's Flying Circus » (after which the Python programming language is named).
— `BADDY` - silent movie era entrance of a baddy.
— `CHASE` - silent movie era chase scene.
— `BA_DING` - a short signal to indicate something has happened.
— `WAWAWAWAA` - a very sad trombone.
— `JUMP_UP` - for use in a game, indicating upward movement.
— `JUMP_DOWN` - for use in a game, indicating downward movement.
— `POWER_UP` - a fanfare to indicate an achievement unlocked.
— `POWER_DOWN` - a sad fanfare to indicate an achievement lost.

6.2.2 Example

```
"""
    music.py
    ~~~~~~~~

    Plays a simple tune using the Micropython music module.
    This example requires a speaker/buzzer/headphones connected to P0 and GND.
"""
from microbit import *
import music

# play Prelude in C.
notes = [
    'c4:1', 'e', 'g', 'c5', 'e5', 'g4', 'c5', 'e5', 'c4', 'e', 'g', 'c5', 'e5', 'g4',
↪'c5', 'e5',
    'c4', 'd', 'a', 'd5', 'f5', 'a4', 'd5', 'f5', 'c4', 'd', 'a', 'd5', 'f5', 'a4',
↪'d5', 'f5',
    'b3', 'd4', 'g', 'd5', 'f5', 'g4', 'd5', 'f5', 'b3', 'd4', 'g', 'd5', 'f5', 'g4',
↪'d5', 'f5',
    'c4', 'e', 'g', 'c5', 'e5', 'g4', 'c5', 'e5', 'c4', 'e', 'g', 'c5', 'e5', 'g4',
↪'c5', 'e5',
    'c4', 'e', 'a', 'e5', 'a5', 'a4', 'e5', 'a5', 'c4', 'e', 'a', 'e5', 'a5', 'a4',
↪'e5', 'a5',
    'c4', 'd', 'f#', 'a', 'd5', 'f#4', 'a', 'd5', 'c4', 'd', 'f#', 'a', 'd5', 'f#4',
↪'a', 'd5',
    'b3', 'd4', 'g', 'd5', 'g5', 'g4', 'd5', 'g5', 'b3', 'd4', 'g', 'd5', 'g5', 'g4',
↪'d5', 'g5',
    'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g', 'c5', 'b3', 'c4', 'e', 'g', 'c5', 'e4', 'g
↪', 'c5',
    'a3', 'c4', 'e', 'g', 'c5', 'e4', 'g', 'c5', 'a3', 'c4', 'e', 'g', 'c5', 'e4', 'g
↪', 'c5',
    'd3', 'a', 'd4', 'f#', 'c5', 'd4', 'f#', 'c5', 'd3', 'a', 'd4', 'f#', 'c5', 'd4',
↪'f#', 'c5',
    'g3', 'b', 'd4', 'g', 'b', 'd', 'g', 'b', 'g3', 'b3', 'd4', 'g', 'b', 'd', 'g', 'b
↪'
]

music.play(notes)
```

CHAPITRE 7

NeoPixel

The `neopixel` module lets you use Neopixel (WS2812) individually addressable RGB LED strips with the Microbit. Note to use the `neopixel` module, you need to import it separately with :

```
import neopixel
```

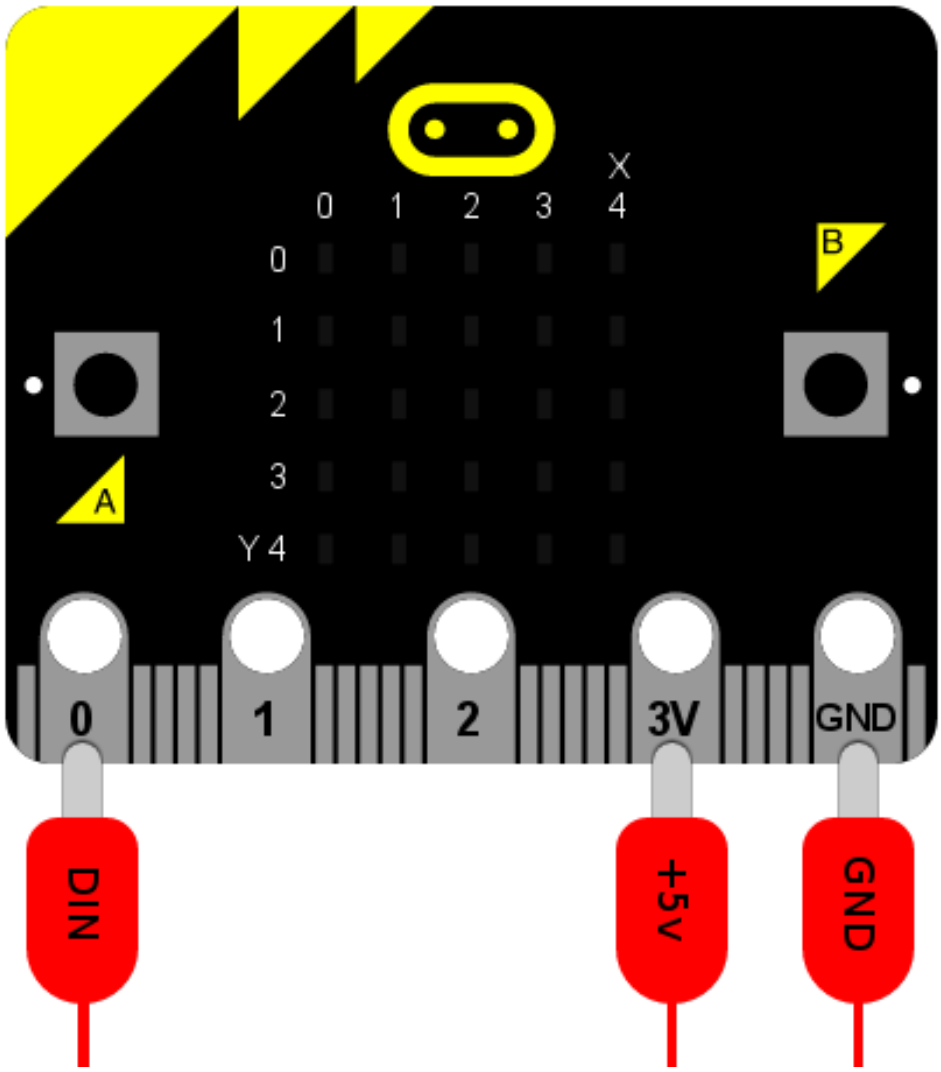
Note : From our tests, the Microbit Neopixel module can drive up to around 256 Neopixels. Anything above that and you may experience weird bugs and issues.

NeoPixels are fun strips of multi-coloured programmable LEDs. This module contains everything to plug them into a micro :bit and create funky displays, art and games such as the demo shown below.

To connect a strip of neopixels you'll need to attach the micro :bit as shown below (assuming you want to drive the pixels from pin 0 - you can connect neopixels to pins 1 and 2 too). The label on the crocodile clip tells you where to attach the other end on the neopixel strip.

Avertissement : Do not use the 3v connector on the Microbit to power any more than 8 Neopixels at a time.

If you wish to use more than 8 Neopixels, you must use a separate 3v-5v power supply for the Neopixel power pin.



7.1 Classes

class neopixel.**NeoPixel**(*pin*, *n*)

Initialise a new strip of `n` number of neopixel LEDs controlled via pin `pin`. Each pixel is addressed by a position (starting from 0). Neopixels are given RGB (red, green, blue) values between 0-255 as a tuple. For example, `(255,255,255)` is white.

clear()

Clear all the pixels.

show()

Show the pixels. Must be called for any updates to become visible.

7.2 Operations

Writing the colour doesn't update the display (use `show()` for that).

```
np[0] = (255, 0, 128)  # first element
np[-1] = (0, 255, 0)  # last element
np.show()  # only now will the updated value be shown
```

To read the colour of a specific pixel just reference it.

```
print(np[0])
```

7.3 Using Neopixels

Interact with Neopixels as if they were a list of tuples. Each tuple represents the RGB (red, green and blue) mix of colours for a specific pixel. The RGB values can range between 0 to 255.

For example, initialise a strip of 8 neopixels on a strip connected to pin0 like this :

```
import neopixel
np = neopixel.NeoPixel(pin0, 8)
```

Set pixels by indexing them (like with a Python list). For instance, to set the first pixel to full brightness red, you would use :

```
np[0] = (255, 0, 0)
```

Or the final pixel to purple :

```
np[-1] = (255, 0, 255)
```

Get the current colour value of a pixel by indexing it. For example, to print the first pixel's RGB value use :

```
print(np[0])
```

Finally, to push the new colour data to your Neopixel strip, use the .show() function :

```
np.show()
```

If nothing is happening, it's probably because you've forgotten this final step.. !

Note : If you're not seeing anything change on your Neopixel strip, make sure you have `show()` at least somewhere otherwise your updates won't be shown.

7.4 Example

```
"""
    neopixel_random.py

    Repeatedly displays random colours onto the LED strip.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    This example requires a strip of 8 Neopixels (WS2812) connected to pin0.

"""
from microbit import *
import neopixel
from random import randint

# Setup the Neopixel strip on pin0 with a length of 8 pixels
np = neopixel.NeoPixel(pin0, 8)

while True:
    #Iterate over each LED in the strip

    for pixel_id in range(0, len(np)):
        red = randint(0, 60)
        green = randint(0, 60)
        blue = randint(0, 60)

        # Assign the current LED a random red, green and blue value between 0 and 60
        np[pixel_id] = (red, green, blue)

        # Display the current pixel data on the Neopixel strip
        np.show()
        sleep(100)
```

CHAPITRE 8

The os Module

MicroPython contains an os module based upon the os module in the Python standard library. It's used for accessing what would traditionally be termed as operating system dependent functionality. Since there is no operating system in MicroPython the module provides functions relating to the management of the simple on-device persistent file system and information about the current system.

To access this module you need to :

```
import os
```

We assume you have done this for the examples below.

8.1 Functions

os.**listdir**()
: Returns a list of the names of all the files contained within the local persistent on-device file system.

os.**remove**(*filename*)
: Removes (deletes) the file named in the argument filename. If the file does not exist an OSError exception will occur.

os.**size**(*filename*)
: Returns the size, in bytes, of the file named in the argument filename. If the file does not exist an OSError exception will occur.

os.**uname**()
: Returns information identifying the current operating system. The return value is an object with five attributes :
  — sysname - operating system name
  — nodename - name of machine on network (implementation-defined)
  — release - operating system release
  — version - operating system version
  — machine - hardware identifier

Note : There is no underlying operating system in MicroPython. As a result the information returned by the `uname` function is mostly useful for versioning details.

CHAPITRE 9

Radio

The `radio` module allows devices to work together via simple wireless networks.

The radio module is conceptually very simple :

- Broadcast messages are of a certain configurable length (up to 251 bytes).
- Messages received are read from a queue of configurable size (the larger the queue the more RAM is used). If the queue is full, new messages are ignored. Reading a message removes it from the queue.
- Messages are broadcast and received on a preselected channel (numbered 0-83).
- Broadcasts are at a certain level of power - more power means more range.
- Messages are filtered by address (like a house number) and group (like a named recipient at the specified address).
- The rate of throughput can be one of three pre-determined settings.
- Send and receive bytes to work with arbitrary data.
- Use *receive_full* to obtain full details about an incoming message : the data, receiving signal strength, and a microsecond timestamp when the message arrived.
- As a convenience for children, it's easy to send and receive messages as strings.
- The default configuration is both sensible and compatible with other platforms that target the BBC micro :bit.

To access this module you need to :

```
import radio
```

We assume you have done this for the examples below.

9.1 Constants

`radio.`**`RATE_250KBIT`**
: Constant used to indicate a throughput of 256 Kbit a second.

`radio.`**`RATE_1MBIT`**
: Constant used to indicate a throughput of 1 MBit a second.

`radio.`**`RATE_2MBIT`**
: Constant used to indicate a throughput of 2 MBit a second.

9.2 Functions

`radio.`**`on`**`()`
Turns the radio on. This needs to be explicitly called since the radio draws power and takes up memory that you may otherwise need.

`radio.`**`off`**`()`
Turns off the radio, thus saving power and memory.

`radio.`**`config`**`(**`*`kwargs`*`)`
Configures various keyword based settings relating to the radio. The available settings and their sensible default values are listed below.

The `length` (default=32) defines the maximum length, in bytes, of a message sent via the radio. It can be up to 251 bytes long (254 - 3 bytes for S0, LENGTH and S1 preamble).

The `queue` (default=3) specifies the number of messages that can be stored on the incoming message queue. If there are no spaces left on the queue for incoming messages, then the incoming message is dropped.

The `channel` (default=7) can be an integer value from 0 to 83 (inclusive) that defines an arbitrary « channel » to which the radio is tuned. Messages will be sent via this channel and only messages received via this channel will be put onto the incoming message queue. Each step is 1MHz wide, based at 2400MHz.

The `power` (default=6) is an integer value from 0 to 7 (inclusive) to indicate the strength of signal used when broadcasting a message. The higher the value the stronger the signal, but the more power is consumed by the device. The numbering translates to positions in the following list of dBm (decibel milliwatt) values : -30, -20, -16, -12, -8, -4, 0, 4.

The `address` (default=0x75626974) is an arbitrary name, expressed as a 32-bit address, that's used to filter incoming packets at the hardware level, keeping only those that match the address you set. The default used by other micro :bit related platforms is the default setting used here.

The `group` (default=0) is an 8-bit value (0-255) used with the `address` when filtering messages. Conceptually, « address » is like a house/office address and « group » is like the person at that address to which you want to send your message.

The `data_rate` (default=radio.RATE_1MBIT) indicates the speed at which data throughput takes place. Can be one of the following contants defined in the `radio` module : `RATE_250KBIT`, `RATE_1MBIT` or `RATE_2MBIT`.

If `config` is not called then the defaults described above are assumed.

`radio.`**`reset`**`()`
Reset the settings to their default values (as listed in the documentation for the `config` function above).

Note : None of the following send or receive methods will work until the radio is turned on.

`radio.`**`send_bytes`**`(`*`message`*`)`
Sends a message containing bytes.

`radio.`**`receive_bytes`**`()`
Receive the next incoming message on the message queue. Returns `None` if there are no pending messages. Messages are returned as bytes.

`radio.`**`receive_bytes_into`**`(`*`buffer`*`)`
Receive the next incoming message on the message queue. Copies the message into `buffer`, trimming the end of the message if necessary. Returns `None` if there are no pending messages, otherwise it returns the length of the message (which might be more than the length of the buffer).

`radio.`**`send`**`(`*`message`*`)`
Sends a message string. This is the equivalent of `send_bytes(bytes(message, 'utf8'))` but with `b'\x01\x00\x01'` prepended to the front (to make it compatible with other platforms that target the micro :bit).

`radio.`**`receive`**`()`

Works in exactly the same way as `receive_bytes` but returns whatever was sent.

Currently, it's equivalent to `str(receive_bytes(), 'utf8')` but with a check that the the first three bytes are `b'\x01\x00\x01'` (to make it compatible with other platforms that may target the micro :bit). It strips the prepended bytes before converting to a string.

A `ValueError` exception is raised if conversion to string fails.

`radio.`**`receive_full`**`()`

Returns a tuple containing three values representing the next incoming message on the message queue. If there are no pending messages then `None` is returned.

The three values in the tuple represent :
— the next incoming message on the message queue as bytes.
— the RSSI (signal strength) : a value between 0 (strongest) and -255 (weakest) as measured in dBm.
— a microsecond timestamp : the value returned by `time.ticks_us()` when the message was received.
For example :

```
details = radio.receive_full()
if details:
    msg, rssi, timestamp = details
```

This function is useful for providing information needed for triangulation and/or triliteration with other micro :bit devices.

9.2.1 Examples

```
# A micro:bit Firefly.
# By Nicholas H.Tollervey. Released to the public domain.
import radio
import random
from microbit import display, Image, button_a, sleep

# Create the "flash" animation frames. Can you work out how it's done?
flash = [Image().invert()*(i/9) for i in range(9, -1, -1)]

# The radio won't work unless it's switched on.
radio.on()

# Event loop.
while True:
    # Button A sends a "flash" message.
    if button_a.was_pressed():
        radio.send('flash')  # a-ha
    # Read any incoming messages.
    incoming = radio.receive()
    if incoming == 'flash':
        # If there's an incoming "flash" message display
        # the firefly flash animation after a random short
        # pause.
        sleep(random.randint(50, 350))
        display.show(flash, delay=100, wait=False)
        # Randomly re-broadcast the flash message after a
        # slight delay.
        if random.randint(0, 9) == 0:
            sleep(500)
            radio.send('flash')  # a-ha
```

CHAPITRE 10

Random Number Generation

This module is based upon the `random` module in the Python standard library. It contains functions for generating random behaviour.

To access this module you need to :

```
import random
```

We assume you have done this for the examples below.

10.1 Functions

`random.`**`getrandbits`**(*n*)
Returns an integer with `n` random bits.

Avertissement : Because the underlying generator function returns at most 30 bits, `n` may only be a value between 1-30 (inclusive).

`random.`**`seed`**(*n*)
Initialize the random number generator with a known integer `n`. This will give you reproducibly deterministic randomness from a given starting state (`n`).

`random.`**`randint`**(*a*, *b*)
Return a random integer `N` such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

`random.`**`randrange`**(*stop*)
Return a randomly selected integer between zero and up to (but not including) `stop`.

`random.`**`randrange`**(*start*, *stop*)
Return a randomly selected integer from `range(start, stop)`.

`random.`**`randrange`**(*start*, *stop*, *step*)
Return a randomly selected element from `range(start, stop, step)`.

random.**choice**(*seq*)
Return a random element from the non-empty sequence `seq`. If `seq` is empty, raises `IndexError`.

random.**random**()
Return the next random floating point number in the range [0.0, 1.0)

random.**uniform**(*a*, *b*)
Return a random floating point number `N` such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

CHAPITRE 11

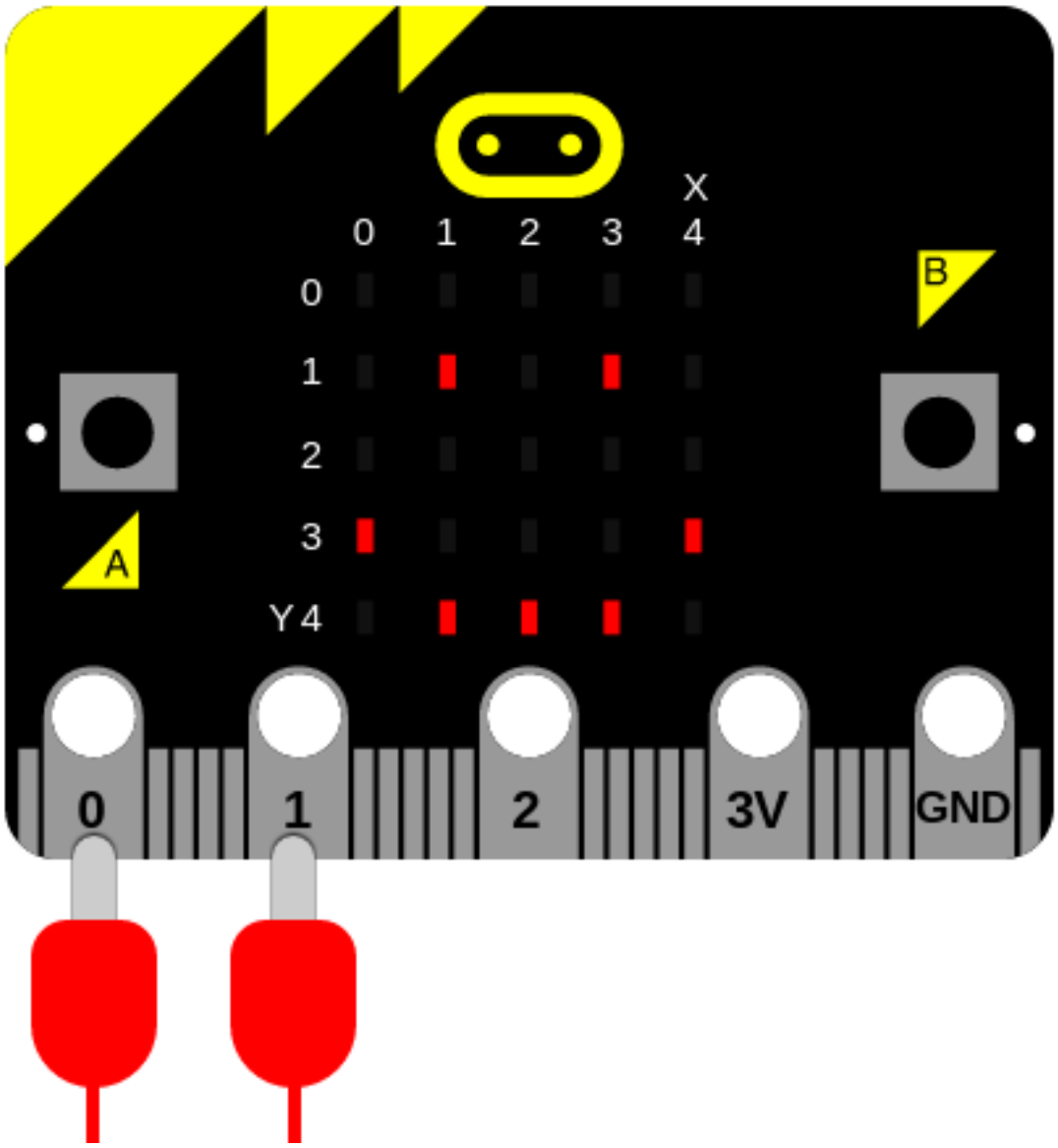
Speech

Avertissement : WARNING ! THIS IS ALPHA CODE.

We reserve the right to change this API as development continues.

The quality of the speech is not great, merely « good enough ». Given the constraints of the device you may encounter memory errors and / or unexpected extra sounds during playback. It's early days and we're improving the code for the speech synthesiser all the time. Bug reports and pull requests are most welcome.

This module makes microbit talk, sing and make other speech like sounds provided that you connect a speaker to your board as shown below :



Note : This work is based upon the amazing reverse engineering efforts of Sebastian Macke based upon an old text-to-speech (TTS) program called SAM (Software Automated Mouth) originally released in 1982 for the Commodore 64. The result is a small C library that we have adopted and adapted for the micro :bit. You can find out more from his homepage. Much of the information in this document was gleaned from the original user's manual which can be found here.

The speech synthesiser can produce around 2.5 seconds worth of sound from up to 255 characters of textual input.

To access this module you need to :

```
import speech
```

We assume you have done this for the examples below.

11.1 Functions

`speech.translate`(*words*)
Given English words in the string `words`, return a string containing a best guess at the appropriate phonemes to pronounce. The output is generated from this text to phoneme translation table.

This function should be used to generate a first approximation of phonemes that can be further hand-edited to improve accuracy, inflection and emphasis.

`speech.pronounce`(*phonemes*, *, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Pronounce the phonemes in the string `phonemes`. See below for details of how to use phonemes to finely control the output of the speech synthesiser. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice.

`speech.say`(*words*, *, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Say the English words in the string `words`. The result is semi-accurate for English. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice. This is a short-hand equivalent of : `speech.pronounce(speech.translate(words))`

`speech.sing`(*phonemes*, *, *pitch=64*, *speed=72*, *mouth=128*, *throat=128*)
Sing the phonemes contained in the string `phonemes`. Changing the pitch and duration of the note is described below. Override the optional pitch, speed, mouth and throat settings to change the timbre (quality) of the voice.

11.2 Punctuation

Punctuation is used to alter the delivery of speech. The synthesiser understands four punctuation marks : hyphen, comma, full-stop and question mark.

The hyphen (`-`) marks clause boundaries by inserting a short pause in the speech.

The comma (`,`) marks phrase boundaries and inserts a pause of approximately double that of the hyphen.

The full-stop (`.`) and question mark (`?`) end sentences.

The full-stop inserts a pause and causes the pitch to fall.

The question mark also inserts a pause but causes the pitch to rise. This works well with yes/no questions such as, « are we home yet ? » rather than more complex questions such as « why are we going home ? ». In the latter case, use a full-stop.

11.3 Timbre

The timbre of a sound is the quality of the sound. It's the difference between the voice of a DALEK and the voice of a human (for example). To control the timbre change the numeric settings of the `pitch`, `speed`, `mouth` and `throat` arguments.

The pitch (how high or low the voice sounds) and speed (how quickly the speech is delivered) settings are rather obvious and generally fall into the following categories :

Pitch :
- 0-20 impractical
- 20-30 very high
- 30-40 high
- 40-50 high normal
- 50-70 normal
- 70-80 low normal

— 80-90 low
— 90-255 very low
(The default is 64)

Speed :
— 0-20 impractical
— 20-40 very fast
— 40-60 fast
— 60-70 fast conversational
— 70-75 normal conversational
— 75-90 narrative
— 90-100 slow
— 100-225 very slow
(The default is 72)

The mouth and throat values are a little harder to explain and the following descriptions are based upon our aural impressions of speech produced as the value of each setting is changed.

For mouth, the lower the number the more it sounds like the speaker is talking without moving their lips. In contrast, higher numbers (up to 255) make it sound like the speech is enunciated with exagerated mouth movement.

For throat, the lower the number the more relaxed the speaker sounds. In contrast, the higher the number, the more tense the tone of voice becomes.

The important thing is to experiment and adjust the settings until you get the effect you desire.

To get you started here are some examples :

```
speech.say("I am a little robot",  speed=92, pitch=60, throat=190, mouth=190)
speech.say("I am an elf", speed=72, pitch=64, throat=110, mouth=160)
speech.say("I am a news presenter", speed=82, pitch=72, throat=110, mouth=105)
speech.say("I am an old lady", speed=82, pitch=32, throat=145, mouth=145)
speech.say("I am E.T.", speed=100, pitch=64, throat=150, mouth=200)
speech.say("I am a DALEK - EXTERMINATE", speed=120, pitch=100, throat=100, mouth=200)
```

11.4 Phonemes

The `say` function makes it easy to produce speech - but often it's not accurate. To make sure the speech synthesiser pronounces things *exactly* how you'd like, you need to use phonemes : the smallest perceptually distinct units of sound that can be used to distinguish different words. Essentially, they are the building-block sounds of speech.

The `pronounce` function takes a string containing a simplified and readable version of the International Phonetic Alphabet and optional annotations to indicate inflection and emphasis.

The advantage of using phonemes is that you don't have to know how to spell ! Rather, you only have to know how to say the word in order to spell it phonetically.

The table below lists the phonemes understood by the synthesiser.

Note : The table contains the phoneme as characters, and an example word. The example words have the sound of the phoneme (in parenthesis), but not necessarily the same letters.

Often overlooked : the symbol for the « H » sound is `/H`. A glottal stop is a forced stoppage of sound.

```
SIMPLE VOWELS                          VOICED CONSONANTS
IY           f(ee)t                    R         (r)ed
IH           p(i)n                     L         a(ll)ow
EH           b(e)g                     W         a(w)ay
AE           S(a)m                     W         (wh)ale
AA           p(o)t                     Y         (y)ou
AH           b(u)dget                  M         Sa(m)
AO           t(al)k                    N         ma(n)
OH           c(o)ne                    NX        so(ng)
UH           b(oo)k                    B         (b)ad
UX           l(oo)t                    D         (d)og
ER           b(ir)d                    G         a(g)ain
AX           gall(o)n                  J         (j)u(dg)e
IX           dig(i)t                   Z         (z)oo
                                       ZH        plea(s)ure
DIPHTHONGS                             V         se(v)en
EY           m(a)de                    DH        (th)en
AY           h(igh)
OY           b(oy)
AW           h(ow)                     UNVOICED CONSONANTS
OW           sl(ow)                    S          (S)am
UW           cr(ew)                    SH         fi(sh)
                                       F          (f)ish
                                       TH         (th)in
SPECIAL PHONEMES                       P          (p)oke
UL           sett(le) (=AXL)           T          (t)alk
UM           astron(om)y (=AXM)        K          (c)ake
UN           functi(on) (=AXN)         CH         spee(ch)
Q            kitt-en (glottal stop)    /H         a(h)ead
```

The following non-standard symbols are also available to the user :

```
YX           diphthong ending (weaker version of Y)
WX           diphthong ending (weaker version of W)
RX           R after a vowel (smooth version of R)
LX           L after a vowel (smooth version of L)
/X           H before a non-front vowel or consonant - as in (wh)o
DX           T as in pi(t)y (weaker version of T)
```

Here are some seldom used phoneme combinations (and suggested alternatives) :

```
PHONEME        YOU PROBABLY WANT:     UNLESS IT SPLITS SYLLABLES LIKE:
COMBINATION
GS             GZ e.g. ba(gs)         bu(gs)pray
BS             BZ e.g. slo(bz)        o(bsc)ene
DS             DZ e.g. su(ds)         Hu(ds)son
PZ             PS e.g. sla(ps)        -----
TZ             TS e.g. cur(ts)y       -----
KZ             KS e.g. fi(x)          -----
NG             NXG e.g. singing       i(ng)rate
NK             NXK e.g. bank          Su(nk)ist
```

If you use anything other than the phonemes described above, a `ValueError` exception will be raised. Pass in the phonemes as a string like this :

```
speech.pronounce("/HEHLOW")  # "Hello"
```

The phonemes are classified into two broad groups : vowels and consonants.

Vowels are further subdivided into simple vowels and diphthongs. Simple vowels don't change their sound as you say them whereas diphthongs start with one sound and end with another. For example, when you say the word « oil » the « oi » vowel starts with an « oh » sound but changes to an « ee » sound.

Consonants are also subdivided into two groups : voiced and unvoiced. Voiced consonants require the speaker to use their vocal chords to produce the sound. For example, consonants like « L », « N » and « Z » are voiced. Unvoiced consonants are produced by rushing air, such as « P », « T » and « SH ».

Once you get used to it, the phoneme system is easy. To begin with some spellings may seem tricky (for example, « adventure » has a « CH » in it) but the rule is to write what you say, not what you spell. Experimentation is the best way to resolve problematic words.

It's also important that speech sounds natural and understandable. To help with improving the quality of spoken output it's often good to use the built-in stress system to add inflection or emphasis.

There are eight stress markers indicated by the numbers `1` - `8`. Simply insert the required number after the vowel to be stressed. For example, the lack of expression of « /HEHLOW » is much improved (and friendlier) when spelled out « /HEH3LOW ».

It's also possible to change the meaning of words through the way they are stressed. Consider the phrase « Why should I walk to the store ? ». It could be pronounced in several different ways :

```
# You need a reason to do it.
speech.pronounce("WAY2 SHUH7D AY WAO5K TUX DHAH STOH5R.")
# You are reluctant to go.
speech.pronounce("WAY7 SHUH2D AY WAO7K TUX DHAH STOH5R.")
# You want someone else to do it.
speech.pronounce("WAY5 SHUH7D AY2 WAO7K TUX DHAH STOHR.")
# You'd rather drive.
speech.pronounce("WAY5 SHUHD AY7 WAO2K TUX7 DHAH STOHR.")
# You want to walk somewhere else.
speech.pronounce("WAY5 SHUHD AY WAO5K TUX DHAH STOH2OH7R.")
```

Put simply, different stresses in the speech create a more expressive tone of voice.

They work by raising or lowering pitch and elongating the associated vowel sound depending on the number you give :

1. very emotional stress
2. very emphatic stress
3. rather strong stress
4. ordinary stress
5. tight stress
6. neutral (no pitch change) stress
7. pitch-dropping stress
8. extreme pitch-dropping stress

The smaller the number, the more extreme the emphasis will be. However, such stress markers will help pronounce difficult words correctly. For example, if a syllable is not enunciated sufficiently, put in a neutral stress marker.

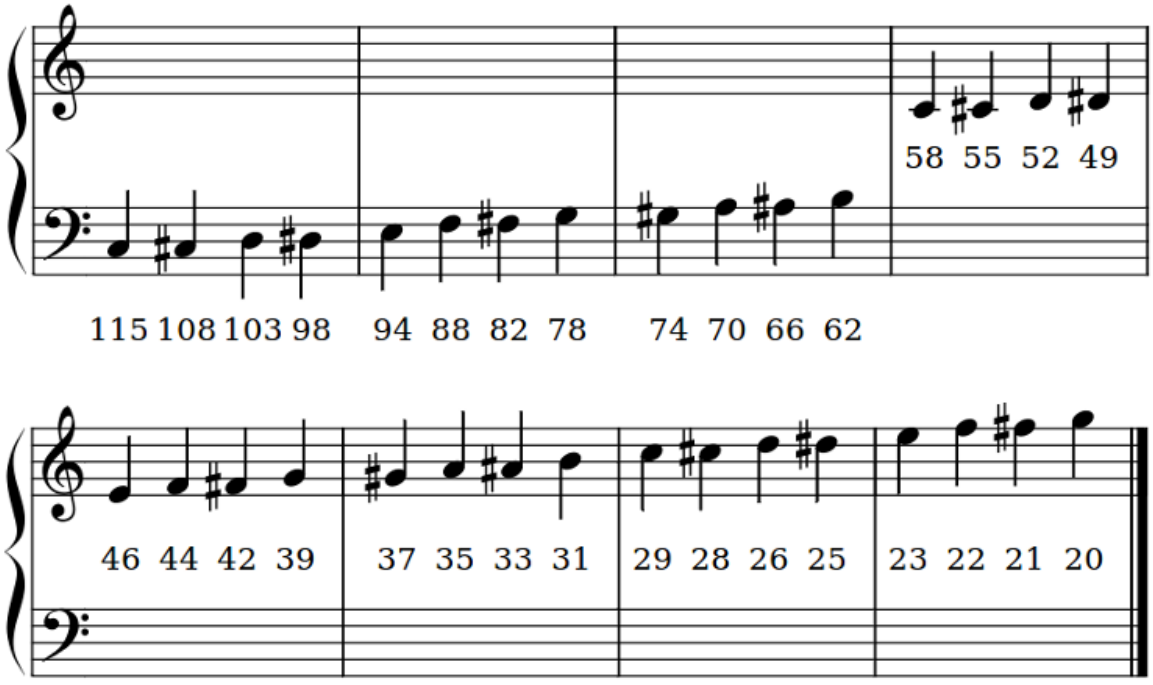
It's also possible to elongate words with stress markers :

```
speech.pronounce("/HEH5EH4EH3EH2EH2EH3EH4EH5EHLP.")
```

11.5 Singing

It's possible to make MicroPython sing phonemes.

This is done by annotating a pitch related number onto a phoneme. The lower the number, the higher the pitch. Numbers roughly translate into musical notes as shown in the diagram below :



Annotations work by pre-pending a hash (#) sign and the pitch number in front of the phoneme. The pitch will remain the same until a new annotation is given. For example, make MicroPython sing a scale like this :

```
solfa = [
    "#115DOWWWWWW",    # Doh
    "#103REYYYYYY",    # Re
    "#94MIYYYYYY",     # Mi
    "#88FAOAOAOAOR",   # Fa
    "#78SOHWWWWW",     # Soh
    "#70LAOAOAOAOR",   # La
    "#62TIYYYYYY",     # Ti
    "#58DOWWWWWW",     # Doh
]
song = ''.join(solfa)
speech.sing(song, speed=100)
```

In order to sing a note for a certain duration extend the note by repeating vowel or voiced consonant phonemes (as demonstrated in the example above). Beware diphthongs - to extend them you need to break them into their component parts. For example, « OY » can be extended with « OHOHIYIYIY ».

Experimentation, listening carefully and adjusting is the only sure way to work out how many times to repeat a phoneme so the note lasts for the desired duration.

11.6 How Does it Work ?

The original manual explains it well :

> First, instead of recording the actual speech waveform, we only store the frequency spectrums. By doing this, we save memory and pick up other advantages. Second, we [...] store some data about timing. These are numbers pertaining to the duration of each phoneme under different circumstances, and also some data on transition times so we can know how to blend a phoneme into its neighbors. Third, we devise a system of rules to deal with all this data and, much to our amazement, our computer is babbling in no time.
>
> —S.A.M. owner's manual.

The output is piped through the functions provided by the `audio` module and, hey presto, we have a talking micro :bit.

11.7 Example

```
import speech
from microbit import sleep

# The say method attempts to convert English into phonemes.
speech.say("I can sing!")
sleep(1000)
speech.say("Listen to me!")
sleep(1000)

# Clearing the throat requires the use of phonemes. Changing
# the pitch and speed also helps create the right effect.
speech.pronounce("AEAE/HAEMM", pitch=200, speed=100)  # Ahem
sleep(1000)

# Singing requires a phoneme with an annotated pitch for each syllable.
solfa = [
    "#115DOWWWWWW",   # Doh
    "#103REYYYYYY",   # Re
    "#94MIYYYYYY",    # Mi
    "#88FAOAOAOAOR",  # Fa
    "#78SOHWWWWW",    # Soh
    "#70LAOAOAOAOR",  # La
    "#62TIYYYYYY",    # Ti
    "#58DOWWWWWW",    # Doh
]

# Sing the scale ascending in pitch.
song = ''.join(solfa)
speech.sing(song, speed=100)
# Reverse the list of syllables.
solfa.reverse()
song = ''.join(solfa)
# Sing the scale descending in pitch.
speech.sing(song, speed=100)
```

CHAPITRE 12

Installation

This section will help you set up the tools and programs needed for developing programs and firmware to flash to the BBC micro :bit using MicroPython.

12.1 Dependencies

12.2 Development Environment

You will need :

— git
— yotta

Depending on your operating system, the installation instructions vary. Use the installation scenario that best suits your system.

Yotta will require an ARM mbed account. It will walk you through signing up if you are not registered.

12.3 Installation Scenarios

12.3.1 Windows

When installing Yotta, make sure you have these components ticked to install.

— python

— gcc
— cMake
— ninja
— Yotta
— git-scm
— mbed serial driver

12.3.2 OS X

12.3.3 Linux

These steps will cover the basic flavors of Linux and working with the micro :bit and MicroPython. See also the specific sections for Raspberry Pi, Debian/Ubuntu, and Red Hat Fedora/Centos.

Debian and Ubuntu

```
sudo add-apt-repository -y ppa:team-gcc-arm-embedded
sudo add-apt-repository -y ppa:pmiller-opensource/ppa
sudo apt-get update
sudo apt-get install cmake ninja-build gcc-arm-none-eabi srecord libssl-dev
pip3 install yotta
```

Red Hat Fedora/CentOS

Raspberry Pi

12.4 Next steps

Congratulations. You have installed your development environment and are ready to begin *flashing firmware* to the micro :bit.

CHAPITRE 13

Flashing Firmware

13.1 Building firmware

Use yotta to build.

Use target bbc-microbit-classic-gcc-nosd :

```
yt target bbc-microbit-classic-gcc-nosd
```

Run yotta update to fetch remote assets :

```
yt up
```

Start the build with either yotta :

```
yt build
```

. . . or use the Makefile :

```
make all
```

The result is a microbit-micropython hex file (i.e. `microbit-micropython.hex`) found in the build/bbc-microbit-classic-gcc-nosd/source from the root of the repository.

The Makefile does some extra preprocessing of the source, which is needed only if you add new interned strings to `qstrdefsport.h`. The Makefile also puts the resulting firmware at build/firmware.hex, and includes some convenience targets.

13.2 Preparing firmware and a Python program

tools/makecombined

hexlify

13.3 Flashing to the micro :bit

Installation Scenarios

CHAPITRE 14

Accessing the REPL

Accessing the REPL on the micro :bit requires :

- Using a serial communication program
- Determining the communication port identifier for the micro :bit
- Establishing communication with the correct settings for your computer

If you are a Windows user you'll need to install the correct drivers. The instructions for which are found here :

https://developer.mbed.org/handbook/Windows-serial-configuration

14.1 Serial communication

To access the REPL, you need to select a program to use for serial communication. Some common options are *picocom* and *screen*. You will need to install program and understand the basics of connecting to a device.

14.2 Determining port

The micro :bit will have a port identifier (tty, usb) that can be used by the computer for communicating. Before connecting to the micro :bit, we must determine the port identifier.

14.3 Establishing communication with the micro :bit

Depending on your operating system, environment, and serial communication program, the settings and commands will vary a bit. Here are some common settings for different systems (please suggest additions that might help others)

Settings

- *Windows*
- *OS X*
- *Linux*
- *Debian and Ubuntu*

— *Red Hat Fedora/CentOS*
— *Raspberry Pi*

CHAPITRE 15

Developer FAQ

Note : This project is under active development. Please help other developers by adding tips, how-tos, and Q&A to this document. Thanks !

Where do I get a copy of the DAL ? A : Ask Nicholas Tollervey for details.

CHAPITRE 16

Contributing

Hey ! Many thanks for wanting to improve MicroPython on the micro :bit.

Contributions are welcome without prejudice from *anyone* irrespective of age, gender, religion, race or sexuality. Good quality code and engagement with respect, humour and intelligence wins every time.

- If you're from a background which isn't well-represented in most geeky groups, get involved - *we want to help you make a difference*.
- If you're from a background which *is* well-represented in most geeky groups, get involved - *we want your help making a difference*.
- If you're worried about not being technical enough, get involved - *your fresh perspective will be invaluable*.
- If you think you're an imposter, get involved.
- If your day job isn't code, get involved.
- This isn't a group of experts, just people. Get involved !
- This is a new community, so, get involved.

We expect contributors to follow the Python Software Foundation's Code of Conduct : https://www.python.org/psf/codeofconduct/

Feedback may be given for contributions and, where necessary, changes will be politely requested and discussed with the originating author. Respectful yet robust argument is most welcome.

16.1 Checklist

- Your code should be commented in *plain English* (British spelling).
- If your contribution is for a major block of work and you've not done so already, add yourself to the AUTHORS file following the convention found therein.
- If in doubt, ask a question. The only stupid question is the one that's never asked.
- Have fun !

- genindex
- modindex
- search

Index des modules Python

m

n

o

r

s

Index