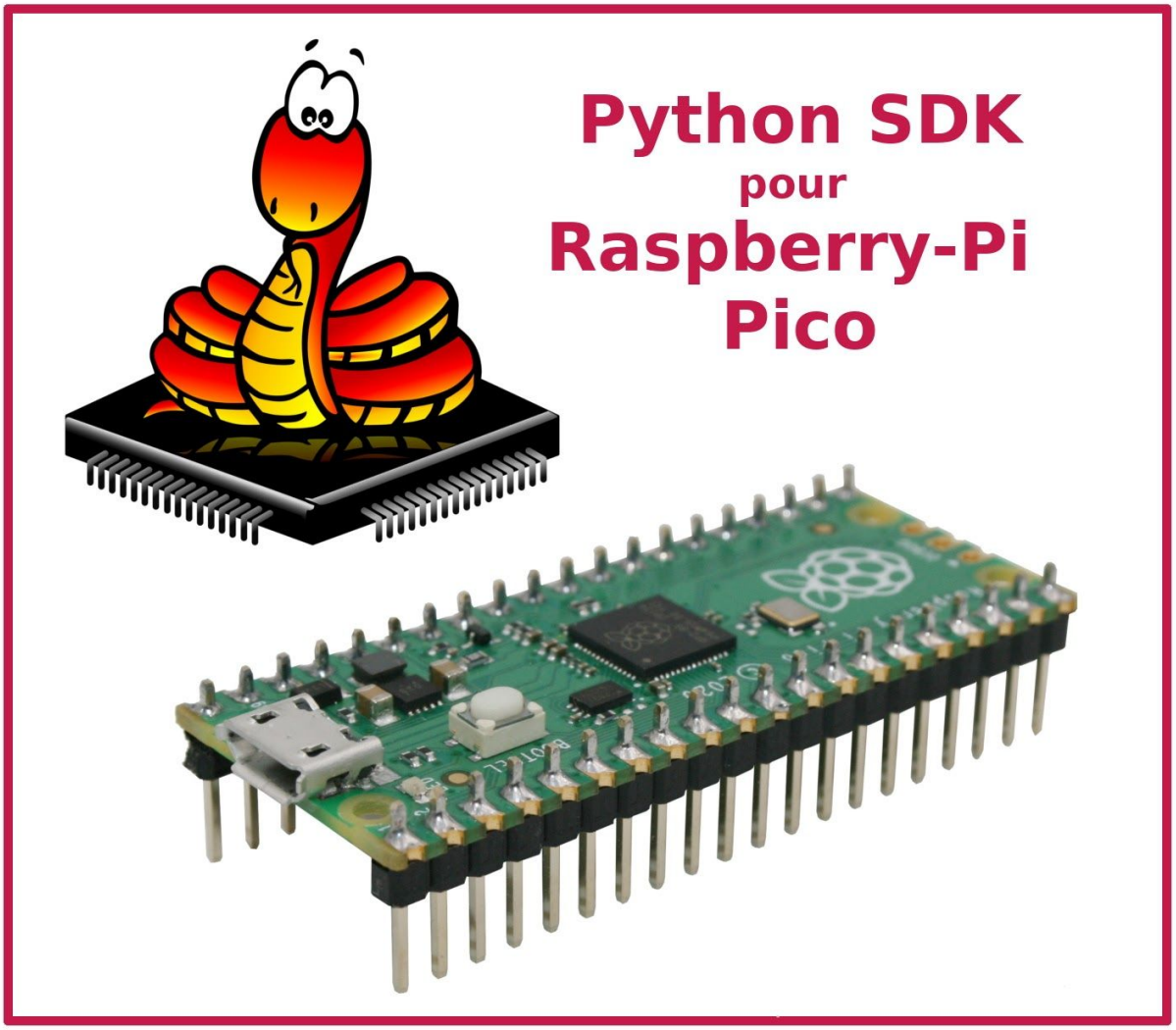


SDK Python pour le Raspberry-Pi Pico

Dernière version numérique disponible sur:
http://df.mchobby.be/RASPBERRY-PICO/Python_SDK_FR.pdf

Colophon

Note
cette section du document original est préservée en l'état sans traduction.

© 2020 Raspberry Pi (Trading) Ltd.
The documentation of the RP2040 microcontroller is licensed under a Creative Commons Attribution-NoDerivatives 4.0
International (CC BY-ND).
build-date: 2020-12-23
build-version: githash: ab06b03-clean (pico-sdk: 1a4974f-clean pico-examples: db698ab-clean)

Legal Disclaimer Notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI (TRADING) LTD ("RPTL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPTL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
RPTL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.
The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPTL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.
RPTL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPTL or other third party intellectual property right.
HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPTL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

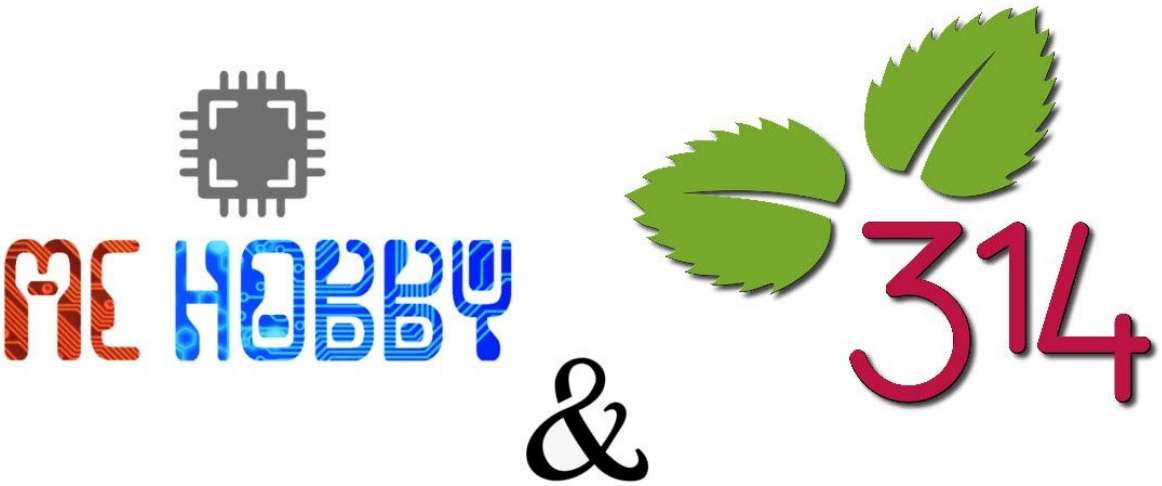
Raspberry Pi products are provided subject to RPTL's Standard Terms. RPTL's provision of the RESOURCES does not expand or otherwise modify RPTL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Traduction, autorisation, crédits et licence

La présente traduction est issue du document "**Python SDK**" édité par la fondation RaspberryPi (voir Colophon ci-dessus, version anglaise disponible sur pico.raspberrypi.org).

Le présent document est une **version traduite et augmentée** par **MC Hobby et Framboise314 (François Mocq)** avec l'autorisation de la fondation Rasperry-Pi.

La version française est publiée sous licence identique à l'original Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).



Historique

19 jan. 2021 : publication initiale
21 jan. 2021 : ajout de contenu, page de garde, crédit
22 jan. 2021 : ajout Motor Shield / Motor FeatherWing (annexes)
24 jan. 2021 : ajout afficheur TFT 2.4" FeatherWing (ILI934x, annexes)
25 jan. 2021 : ajout Hat Sense (annexes)
26 jan. 2021 : ajout Hat PiFace (annexes)

Chapitre 1. L'environnement MicroPython

TO DO: Talk about C vs Micro Python here? Need to mention how to use the stuff in the BootRom from Python here.

MicroPython implémente la syntaxe complète de Python 3.4 (y compris les exceptions , `with` , `yield from` , etc., et en plus les mots-clés `async/await` de Python 3.5). Les types de données de base suivants sont fournis : `str` (y compris le support Unicode de base), `bytes`, `bytearray`, `tuple`, `list`, `dict`, `set`, `frozenset`, `array.array`, `collections.namedtuple` , classes et instances. Les modules intégrés comprennent `sys`, `time`, `struct` , etc. Certains ports prennent en charge le module `_thread` (multithreading). Notez que seul un sous-ensemble des fonctionnalités de Python 3 est implémenté pour les types de données et les modules.

MicroPython peut exécuter des scripts sous forme de source textuelle ou de bytecode précompilé, dans les deux cas, soit à partir d'un système de fichiers sur l'appareil, soit "gelé" (*frozen*) dans le firmware MicroPython.

Obtenir le firmware MicroPython sur le Pico

Avant de téléverser le firmware MicroPython sur la carte Pico, il faut soit le compiler sur son ordinateur à partir du code source de MicroPython, soit télécharger le firmware MicroPython préalablement compilé.

Binaire pré-complié
Un binaire pré-compilé du micrologiciel/firmware MicroPython est disponible sur les page de démarrage Pico.

Si vous avez téléchargé un firmware MicroPython déjà compilé alors vous pouvez directement passer au point 1.3 .

1.1. Obtenir MicroPython pour le microcontrôleur RP2040

IMPORTANT
Les instructions suivantes supposent que vous utilisez une carte Raspberry Pi Pico et certains détails peuvent différer si vous utilisez une carte différente, basée sur le RP2040. Elles supposent également que vous utilisez un système d'exploitation Raspberry Pi OS fonctionnant sur un Raspberry Pi 4, ou une distribution Linux équivalente basée sur Debian et fonctionnant sur une autre plate-forme.

Commencez par créer un répertoire pico afin de regrouper toutes les commandes liées à pico. Ces instructions permettent de créer un répertoire pico à l'adresse `/home/pi/pico` .

```
$ cd ~/
$ mkdir pico
$ cd pico
```

Ensuite, clonez le dépôt git de `micropython` .

```
$ git clone -b pico git@github.com:raspberrypi/micropython.git
```

Pour construire le MicroPython de Raspberry Pi Pico, vous devrez également installer quelques outils supplémentaires. Pour créer des projets, vous aurez besoin de CMake, un outil multi-plateforme utilisé pour créer le logiciel, et de la chaîne d'outils embarqués GNU pour Arm. Vous pouvez installer ces deux outils via `apt` à partir de la ligne de commande. Tout ce que vous avez déjà installé sera ignoré par `apt` .

```
$ sudo apt update
$ sudo apt install cmake gcc-arm-none-eabi
```

1.2. Compiler MicroPython pour le RP2040

Pour compiler MicroPython, déplacez vous dans dans le répertoire `micropython` et saisissez les commande suivantes,

```
$ cd micropython
$ git submodule update --init --recursive
$ make -C mpy-cross
$ cd ports/rp2
$ make
```

Entre autres, nous avons maintenant construit :

- `firmware.elf` , qui est utilisé par le débogueur
- `firmware.uf2` , qui peut être glissé/déplacé sur le périphérique de stockage de masse USB RP2040

vous pouvez les trouver dans le répertoire `ports/rp2/build/`.

1.3. Installation de MicroPython sur le Raspberry Pi Pico

La méthode la plus simple pour charger un logiciel sur une carte à base de RP2040 consiste à la monter comme un dispositif de stockage de masse USB. Cela vous permet de faire glisser le `firmware.uf2` sur la carte pour programmer la mémoire flash. Connectez le Raspberry Pi Pico à votre Raspberry Pi en utilisant un câble micro-USB, tout en vous assurant de maintenir le bouton BOOTSEL enfoncé. Ceci force le PICO à passer en mode « stockage de masse USB ».

NOTE
Si vous ne suivez pas ces instructions sur un Raspberry Pi Pico, vous n'avez peut-être pas de bouton `BOOTSEL`. Si c'est le cas, vous devez vérifier s'il existe un autre moyen de mettre à la masse la broche CS de la mémoire flash, un cavalier par exemple, pour indiquer au RP2040 d'entrer en mode de démarrage USB lors du boot. Si ce n’est pas possible, vous pouvez charger le code en utilisant l'interface Serial Wire Debug (voir le livre **Getting Started with Raspberry Pi Pico** pour plus de détails).

Chapitre 2. Connexion à MicroPython REPL

MicroPython pour Raspberry Pi Pico et d'autres cartes basées sur RP2040 supporte le Serial-over-USB (liaison série sur le port USB).

2.1. Connexion à partir d'un Raspberry Pi via USB

Une fois que MicroPython a été installé, vous pouvez installer minicom :

```
$ sudo apt install minicom
```

et ensuite ouvrir le port série :

```
$ minicom -b 115200 -o -D /dev/ttyACM0
```

Lorsque vous alimentez le Raspberry Pi Pico, vous devriez voir à l'écran,

```
MicroPython v1.12-725-g315e7f50c-dirty on 2020-08-21; Raspberry Pi
PICO with cortex-m0plus
Type "help()" for more information.
>>>
```

lorsque le Raspberry Pi Pico est alimenté pour la première fois.

CONSEIL
Dans minicom, l'appui sur `CTRL-A` suivi de `U` ajoutera des retours chariot à la sortie série, de sorte que chaque impression se terminera par un saut de ligne. Pour quitter minicom, utilisez `CTRL-A` suivi de `X`.

NOTE
Il arrive rarement que vous ne puissiez pas vous connecter au Raspberry Pi Pico, vous devrez peut-être redémarrer votre Raspberry Pi 4 dans ce cas.

2.2. Connexion à partir du GPIO Raspberry Pi 4

MicroPython pour le RP2040 fournit également le REPL sur l'`UART0` , du Raspberry Pi Pico, donc vous pouvez aussi vous connecter au REPL par ce moyen. La première chose à faire est d'activer le port série UART sur le Raspberry Pi 4. Pour ce faire, lancez `raspi-config`

```
$ sudo raspi-config
```

Et allez dans `Options d'interface -> Port série` puis sélectionnez "Non" lorsqu'on vous demande "Voulez vous que le prompt shell soit accessible via l'interface série" (*"Would you like a login shell to be accessible over serial?"*) et sélectionnez "Oui" lorsqu'on vous demande "Voulez vous activer le port série matériel?" (*"Would you like the serial port hardware to be enabled?"*).

Vous devriez voir quelque-chose comme ci-dessous:

- L’invite de commande sur port série est désactivé
- L’interface série est activée

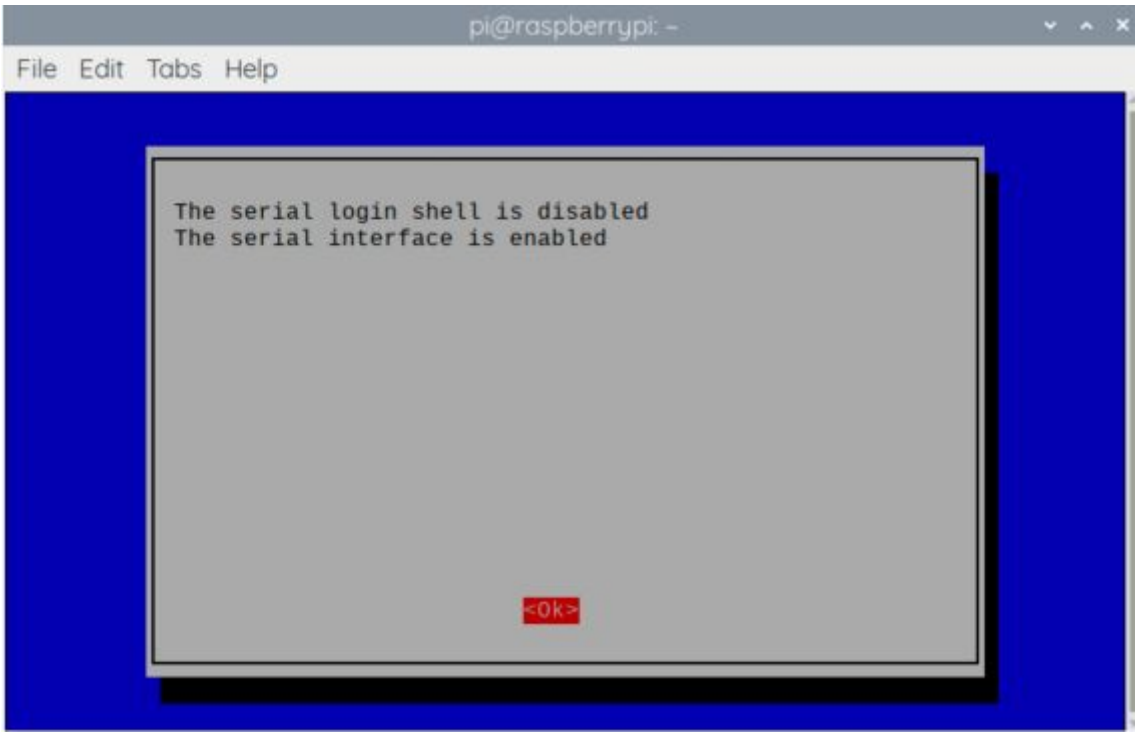


Figure 1. Activer le port série UART en utilisant raspi-config sur un Raspberry-Pi.

En quittant `raspi-config`, vous devez choisir "Oui" et redémarrer votre Raspberry Pi pour activer le port série.
Vous devez ensuite connecter le Raspberry Pi et le Raspberry Pi Pico comme ci-dessous :

Raspberry Pi	Raspberry Pi Pico
GND	GND
GPIO15 (UART_RX0)	GPIO0 (UART0_TX)
GPIO14 (UART_TX0)	GPOI1 (UART0_RX)

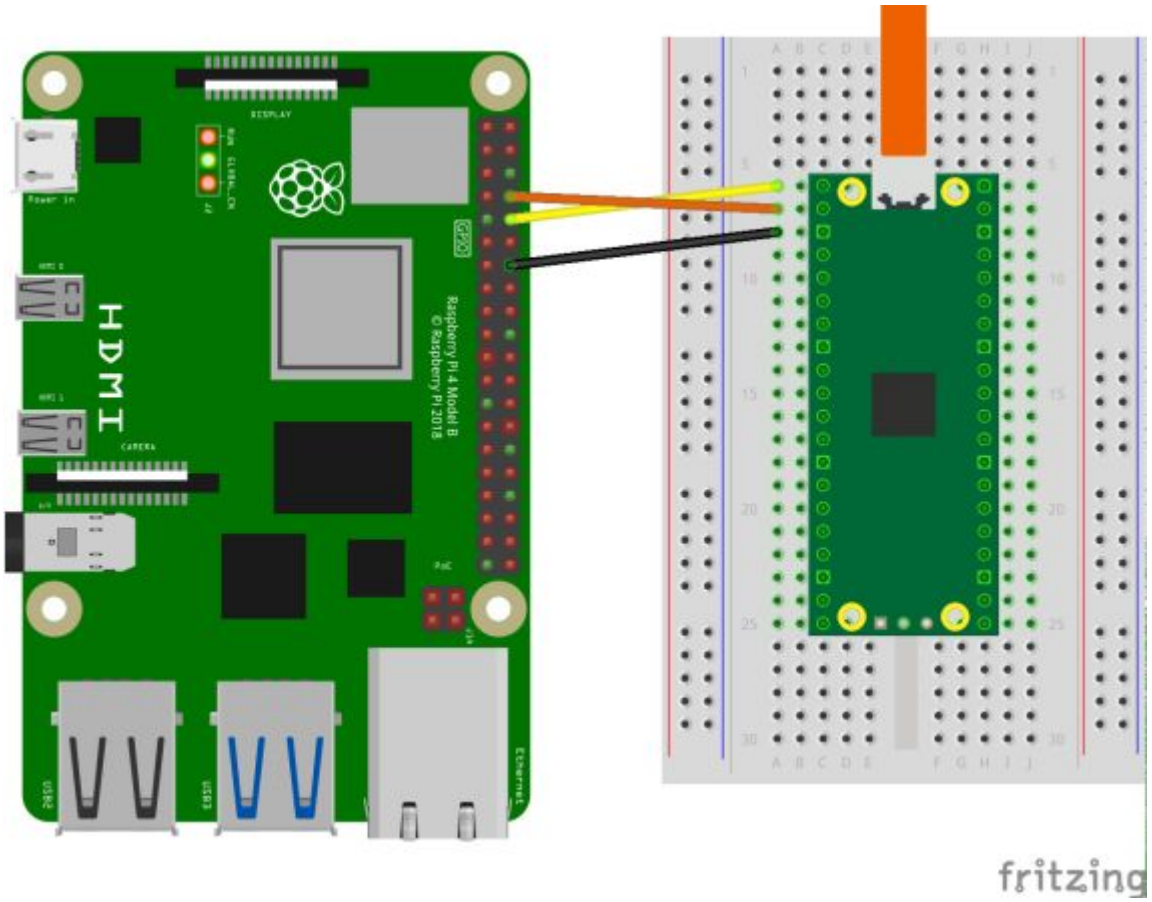


Figure 2. Pi 4 et Raspberry Pi Pico sont connectés par l'UART0 .

puis connectez-vous à la carte en utilisant `minicom` connecté à `/dev/serial0`,

```
$ minicom -b 115200 -o -D /dev/serial0
```

2.3 Connexion à partir d'un Mac via USB

Si vous utilisez une version récente de macOS comme Catalina, les pilotes devraient déjà être chargés. Sinon, consultez le site web du fabricant pour les pilotes de puces FTDI. Ensuite, vous devez utiliser un programme Terminal, par exemple Serial ou similaire pour vous connecter au port série.

NOTE
Serial comprend également le driver si nécessaire

Le port série-sur-USB s'affiche sous la forme `/dev/tty.usbmodem0000000000001`. Si vous utilisez Serial, il sera affiché comme "Board in FS mode - CDC" dans la fenêtre de sélection du port lorsque vous ouvrez l'application.
Connectez-vous à ce port et appuyez sur Retour et vous devriez voir l'invite REPL.

2.4. Dire "Hello World" à partir de REPL

Une fois connecté, vous pouvez vérifier que tout fonctionne en tapant un simple "Hello World" dans le REPL,

```
>>> print('hello pico!')
hello pico!
>>>>
```

Chapitre 3. MicroPython sur RP2040

Actuellement, les fonctionnalités suivantes sont incluses:

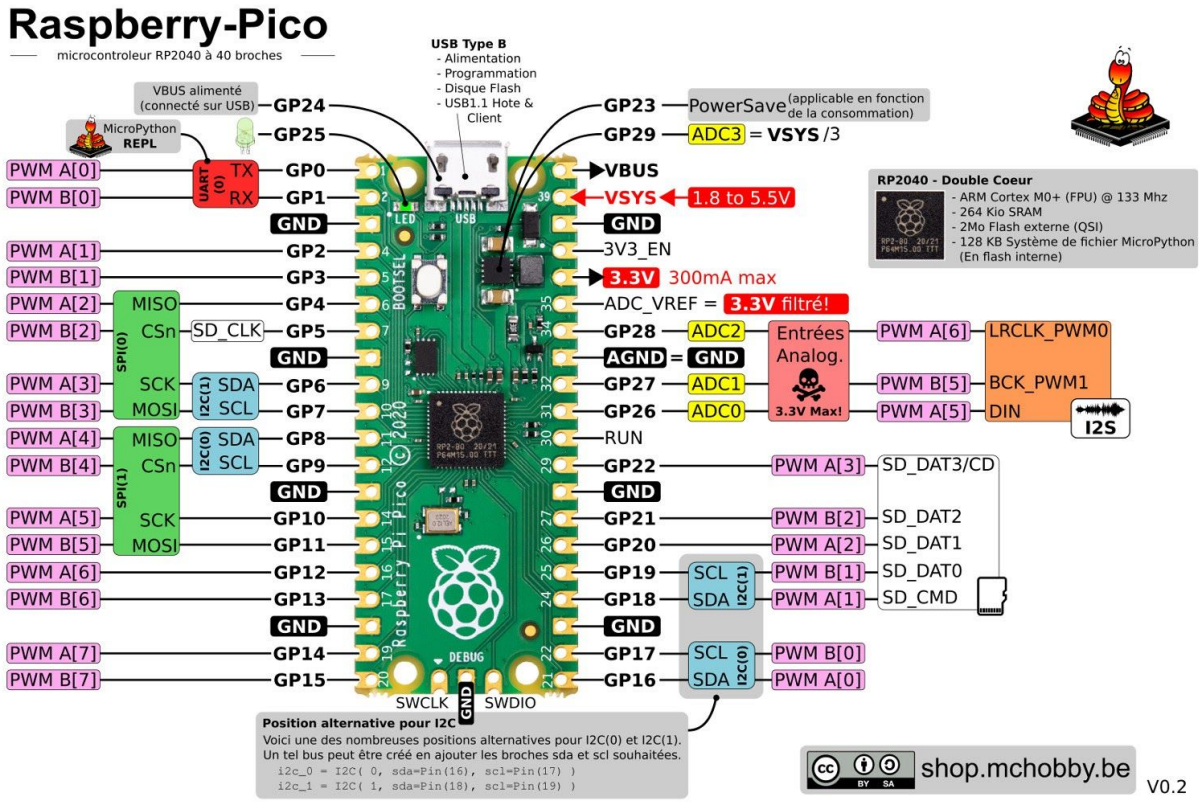
- REPL via USB et UART (sur GP0/GP1).
- Système de fichier de 128Kio dans la mémoire Flash interne utilisant `littlefs2`.
- Le module `utime` avec les fonctions sleep et ticks .
- Le module `ubinascii` .
- Le module `machine` avec quelques fonctions de bases.
    - La classe `machine.Pin` .
    - La classe `machine.Timer` .
    - La classe `machine.ADC` .
    - Les classes `machine.I2C` et `machine.SoftI2C` .
    - Les classes `machine.SPI` et `machine.SoftSPI` .
    - La classe `machine.WDT` (WatchDog).
    - La classe `machine.PWM` .
    - La classe `machine.UART` .
- Le module spécifique `rp2` .
    - Support de `PIO`.
- Le support multi-coeurs est assuré par le module standard `_thread` .

La documentation concernant MicroPython est disponible sur https://docs.micropython.org.

TO DO: parler des fonctionnalités spécifiques de la puce

3.1.Brochage et MicroPython

Utiliser MicroPython sur le Pico c'est aussi envisager exploiter les entrées-sorties. Un schéma du brochage s'impose.

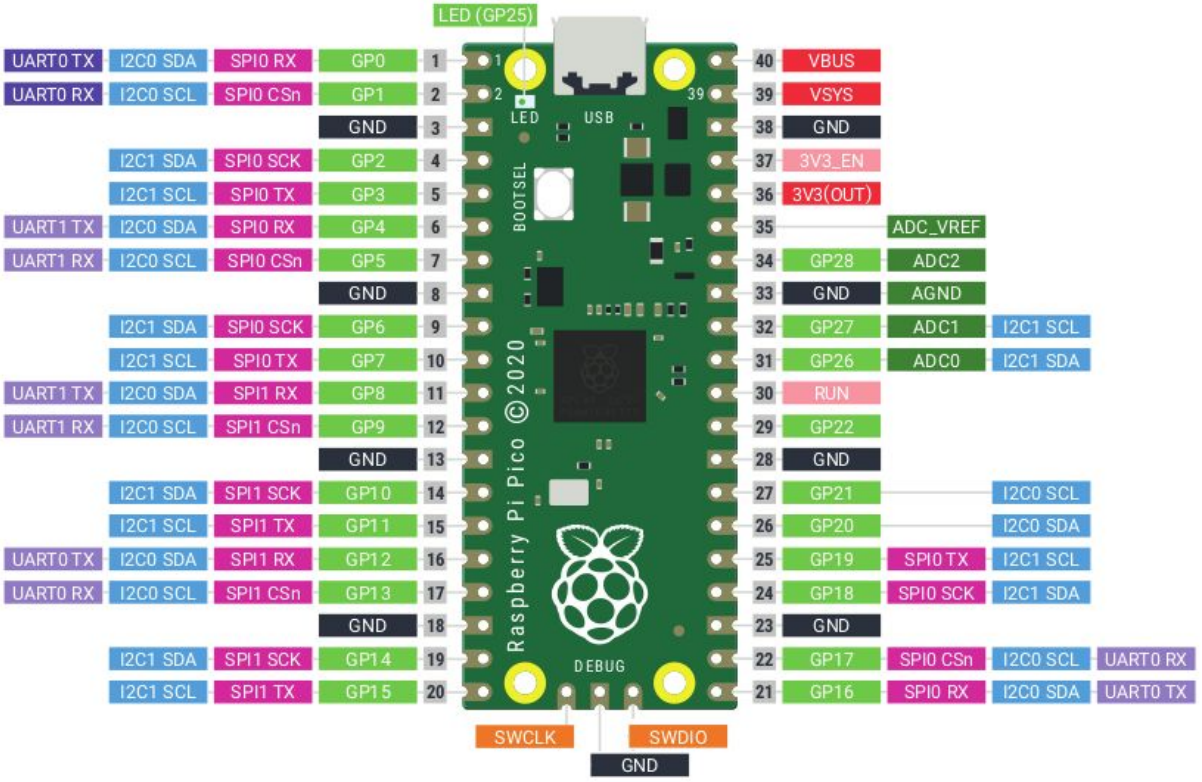


Pour connaître toutes les positions alternatives des bus, le lecteur pourra se référer au brochage officiel disponible dans la documentation.

Quelques broches disposent de caractéristiques spécifique:

- **VBUS :** tension d'entrée sur le port micro-USB. Tension nominale de 5V lorsque l'USB est branché sinon 0 Volts (USB débranché ou non alimenté).
- **VSYS :** tension d'alimentation principale du système (1.8V à 5.5V). Elle est utilisée par le régulateur de tension pour générer les 3.3V utilisé par le RP2040.
- **3V3_EN :** cette broche est branchée à VSYS via une résistance pull-up. De cette façon, le régulateur 3.3V reste actif. Placer cette broche à la masse désactive le régulateur de tension (et le microcontrôleur).
- **3V3 :** sortie du régulateur de tension 3.3V utilisé pour alimenter toute la carte. Le courant max disponible sur cette broche dépend de l'alimentation sur VSYS et de la charge sur le processeur. Ne pas dépasser un courant de 300mA sur cette broche.
- **ADC_VREF :** référence de tension 3.3V pour le convertisseur Analogique-Vers-Numérique. La tension présente sur cette broche est plus stable. Elle peut aussi être utilisée comme référence de tension.
- **AGND :** masse de référence pour le circuit de conversion analogique. Il y a un plan de masse sous les pistes des entrées analogiques et connecté directement sur cette broche.
- **RUN :** broche *Enabled* du microcontroleur RP2040. Cette broche est maintenue à 3.3V via une résistance pull-up de 50 KOhms (interne au RP2040). Placer cette broche à la masse pour réinitialiser le microcontrôleur.

Voici le brochage officiel du Raspberry-Pi Pico.



3.1 Faire clignoter une LED avec MicroPython

La LED présente sur la carte Raspberry Pi Pico est connectée sur la broche GPIO 25. Il est possible de l'allumer et l'éteindre depuis REPL.
Saisissez les instructions suivantes l'invite de commande REPL est visible,

```
>>> from machine import Pin
>>> led = Pin(25, Pin.OUT)
```

puis vous pouvez allumer la LED avec,

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

puis l'éteindre avec,

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

En option, vous pouvez également utiliser un timer (élément d'horodatage) pour faire clignoter la LED sur la carte.

```
1 from machine import Pin, Timer
2
3 led = Pin(25, Pin.OUT)
4 tim = Timer()
```

```
5 def tick(timer):
6 global led
7 led.toggle()
8
9 tim.init(freq=2.5, mode=Timer.PERIODIC, callback=tick)
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/blink/blink.py Lines 1 - 9

3.2 Sortie Numérique

Brancher une LED externe

3.3 Entrée Numérique

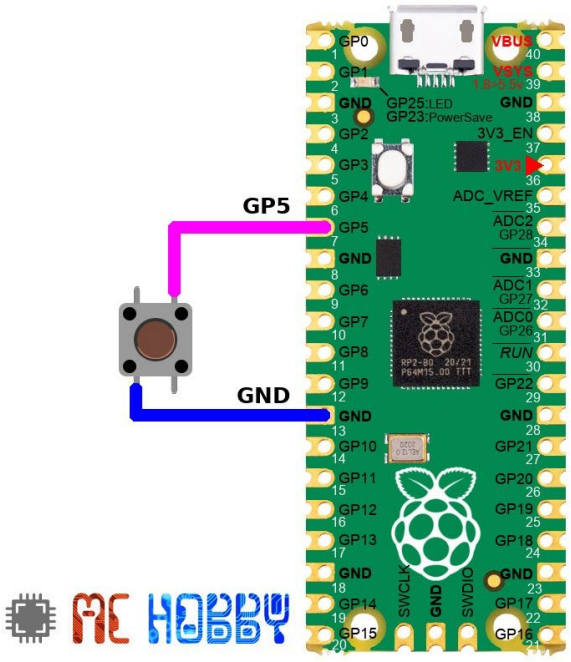
Les entrées numérique du RP2040 disposent de nombres fonctionnalités:

- Ajustement du courant de sortie de 2mA a 12mA,
- Vitesse de modification du signal de sortie (rapide ou lent)
- Activation d'hystérésis sur l'entrée (Mode Trigger de Schmitt)
- **Résistance de rappel pull-up / pull-down sur les entrées**
- Désactivation du buffer d'entrée (diminution de la consommation)

La plupart de ces fonctionnalités sont accessibles en programmant le RP2040 en C++.

MicroPython prend en charge l'activation des résistance pull-up / pull-down (et la force du courant de sortie).

La résistance pull-up permet de ramener la tension d'une broche d'entrée à 3.3V lorsque l'état de la broche n'est pas forcé à un niveau logique par l'électronique de commande.



Lorsque le bouton est pressé, le potentiel de la broche GP5 est forcé à la masse (signal bas). Lorsque le bouton est relâché, le potentiel de GP5 est ramené à 3.3V par la résistance pull-up interne.

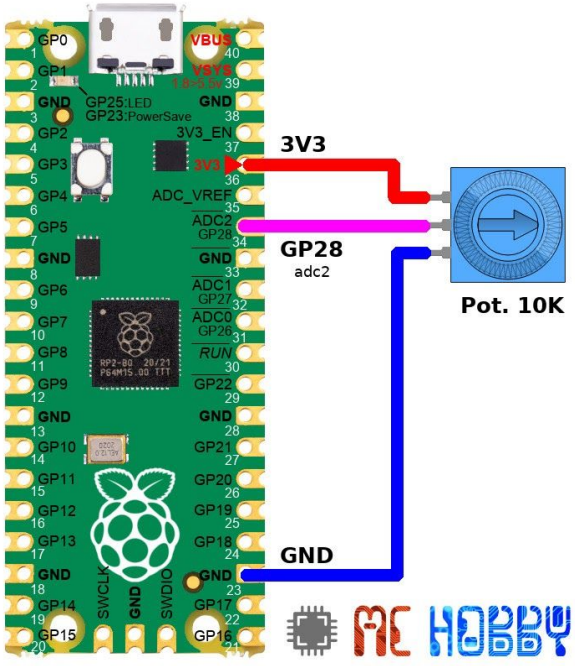
```
1 from machine import Pin
2 import time
3 btn = Pin(5, Pin.IN, Pin.PULL_UP )
4
5 while True:
6   if btn.value()==False:
7     print( "Bouton enfoncé" )
8   else:
9     print( "." )
10  time.sleep( 0.200 ) # 200 ms
```

3.4 Entrée Analogique

Le pico dispose de 4 entrées analogiques dont une est réservée à la lecture de la tension de la broche VSYS (alimentation système).
L'objet `ADC` effectue une lecture par l'intermédiaire de la méthode `read_u16()`. Celle-ci retourne une valeur entre 0 et 65535 (encodée sur 16bits) pour une gamme de tension par défaut du régulateur de tension (0 à 3.3 Volts).

3.4.1 Exemple du potentiomètre

En utilisant un potentiomètre 10 KOhms linéaire, il est possible de générer une tension arbitraire entre 0V et 3.3 Volts.



La lecture de l'entrée analogique ADC2 (GP28) peut se faire à l'aide du code d'exemple suivant:

```
1 from machine import ADC
2 import time
3
4 adc = ADC(2)
5 while True:
6   value = adc.read_u16()
7   volt = value * 3.3 / 65535
8   print( "Value: %s " % value )
9   print( "ADC voltage: %s volts " % volt )
10   time.sleep(0.100) # 100 ms
```

3.4.2 Tension sur VSYS

La broche VSYS permet d'alimenter la carte avec une source de tension de 1.8V à 5.5V. Le convertisseur d'alimentation est capable de travailler sur toute la gamme de tension pour produire une tension de sortie 3.3V stable. Idéal donc pour les projets sur piles.

Il est donc pertinent de pouvoir lire cette tension directement depuis le Pico. Cela permettra d'avertir l'utilisateur qu'il est temps de changer la pile.
Pour rappel ADC3 = VSYS / 3

L'exemple ci-dessous indique comment lire cette entrée analogique

```
1 from machine import ADC
2
3 adc = ADC(3)
4 vadc = adc.read_u16() * 3.3 / 65535
5 vsys = vadc * 3
6 print( "VSYS: %s volts " % vsys )
7 print( "ADC voltage: %s volts " % vadc )
```

3.4.3 Température Interne

Le microcontroleur RP2040 dispose d'un capteur de température branché sur l'entrée analogique. Il permet de relever la température autour du microcontrôleur et, bien entendu, l'activité du RP2040 influence quand même cette température.

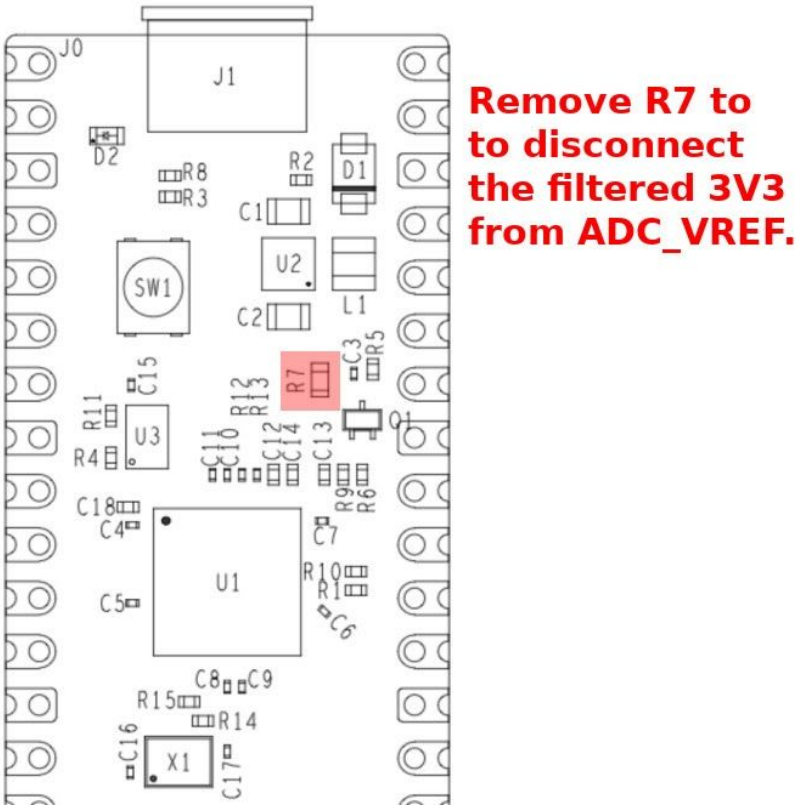
```
1 from machine import ADC
2 import time
3
4 adc = ADC(ADC.CORE_TEMP)
5 vadc = adc.read_u16() * 3.3 / 65535
6 temp  = 27 - (vadc - 0.706) / 0.001721
7 print( "Temp: %s degrees" % temp )
```

3.4.4 La référence analogique

La tension de référence analogique **ADC_VREF** est fixée matériellement à 3.3V (issus de la tension d'alimentation passée dans un filtre).

ATTENTION
En l'état il n'est pas possible d'appliquer une tension de référence différente pour le convertisseur ADC (donc sur ADC_VREF).

En enlevant la résistance R7 (package 0603) sur le Pico, il est possible d'isoler la broche ADC_VREF pour y appliquer une tension de référence différente (entre 2V et 3.3V). A noter que l'ADC du RP2040 n'a été qualifié pour une tension de 3.0 et 3.3V.



3.5. Interruptions

Vous pouvez fixer une IRQ (fonction d'interruption) comme ceci:

```
1 from machine import Pin
2
3 p2 = Pin(2, Pin.IN, Pin.PULL_UP)
4 p2.irq( lambda pin: print("IRQ with flags:",
5   pin.irq().flags() ), Pin.IRQ_FALLING)
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/irq/irq.py Lines 1 - 5

qui devrait afficher quelque-chose lorsque la broche GPIO 2 (GP2) rencontre un flan descendant (*falling edge*).

3.6. Support multi-coeur

Exemple d'utilisation:

```
1  import time, _thread, machine
2
3  def task(n, delay):
4    led = machine.Pin(25, machine.Pin.OUT)
5    for i in range(n):
6      led.high()
7      time.sleep(delay)
8      led.low()
9      time.sleep(delay)
10   print('c est fait')
11
12 _thread.start_new_thread(task, (10, 0.5))
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/multicore/multicore.py Lines 1 - 12

Seul un thread peut être démarré/exécuté à n'importe quel moment, parce qu'il n'y a pas de RTOS mais seulement un second core. Le GIL (*Global Interpreter Lock* de Python) n'est pas activé, par conséquent, il est possible d'exécuter du code Python concurrent sur les deux coeurs (core0 et core1). Il faudra donc mettre en place des mécanismes de blocages (locks) pour les données partagées.

3.7. I2C

3.7.1 Un premier exemple

L'exemple ci-dessus indique comment créer un bus I2C. Le code reprend la création de deux bus avec, respectivement, les numéros de bus et broches associées.
Le paramètre `freq` permet de préciser la vitesse de communication sur le bus I2C. Celle-ci est réduite à 100.000 Hertz (100 KHz) là où la valeur par défaut est de 400 KHz.

```
1 from machine import Pin, I2C
2
3 i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=100000)
4 i2c.scan()
5 i2c.writeto(76, b'123')
6 i2c.readfrom(76, 4)
7
8 i2c = I2C(1, scl=Pin(7), sda=Pin(6), freq=100000)
9 i2c.scan()
10 i2c.writeto_mem(76, 6, b'456')
11 i2c.readfrom_mem(76, 6, 4)
```

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/i2c.py
Lines 1 - 11

Enfin, ce second exemple construit un objet `I2C` sans spécifier broches et fréquence. Les paramètres par défaut sont donc d'application.

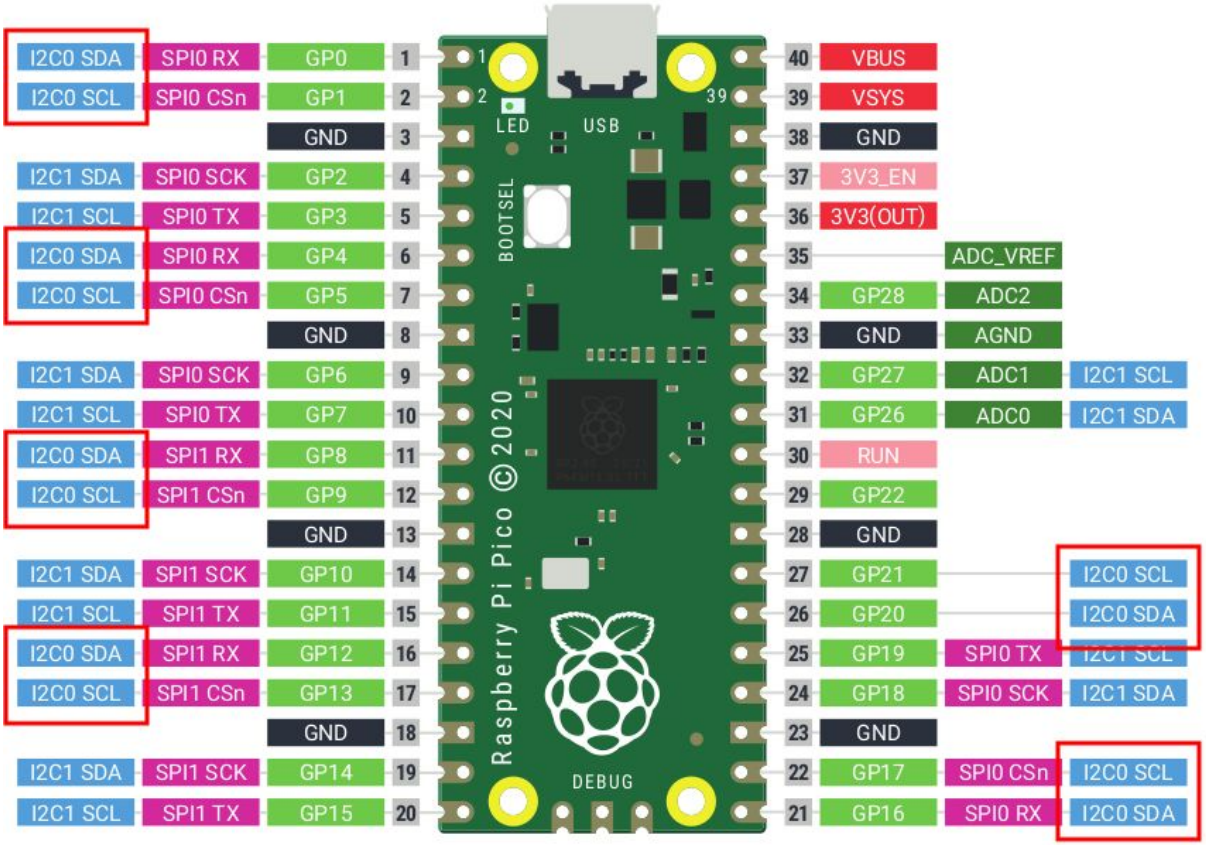
```
1 from machine import I2C
2
3 i2c = I2C(0) # donc SCL=Pin(9), SDA=Pin(8), freq=400000
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/i2c_without_freq.py Lines 1 - 3

ATTENTION
Il peut exister des bogues lors d'opérations de lectures/écritures sur des périphériques ne répondant pas à l'adresse mentionnée. Le matériel semble bloquer dans certains cas.

3.7.2 Multitude de bus

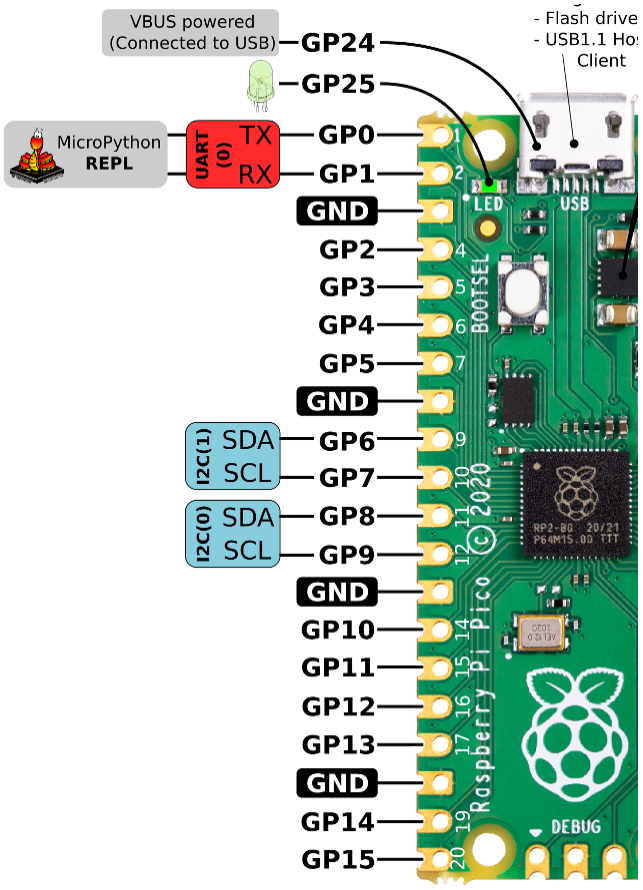
Selon le standard MicroPython, préciser les broches `sda` et `scl` active le mode BitBanging (I2C logiciel) plutôt que le support matériel du bus I2C. Un bus matériel étant toujours plus rapide qu'un bus logiciel.
Il faut cependant reconnaître que la flexibilité du RP2040 permet de placer un même bus I2C, SPI ou UART en de très nombreuses positions. Il faut avouer qu'il y a de quoi déstabiliser plus d'un Maker.
Ainsi, le bus `I2C(0)` sur de nombreuses positions comme l'indique l'image ci-dessous.



Le numéro de bus peut donc être accompagné de l'identification des broches utilisées.

Note
Si vous êtes nouveau venu dans le monde MicroPython, il est préférable d'utiliser la configuration par défaut des bus (donc sans préciser les numéro de broches) car cela raccourcit l'écriture du code et évite aussi les confusions (en employant le bus 0 un peu partout en fonction de l'humeur du jour).

Avec un peu de travail il est possible d'identifier le bus I2C(0) et bus I2C(1) par défaut.



C'est bien entendu ceux qu'il est recommandé d'utiliser en priorité.

3.8. SPI

Exemple d'utilisation:

```
1 from machine import SPI
2
3 spi = SPI(0)
4 spi = SPI(0, 100_000)
5 spi = SPI(0, 100_000, polarity=1, phase=1)
6
7 spi.write('test')
8 spi.read(5)
9
10 buf = bytearray(3)
11 spi.write_readinto('out', buf)
```

Exemples Pico MicroPython:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/sqi/sqi.py Lines 1 - 11

NOTE
La broche "*chip select*" du périphérique SPI doit être contrôlée séparément en utilisant une classe `machine.Pin`.

3.9. PWM

3.9.1 PWM et LED interne

Exemple utilisant une sortie PWM pour faire pulser une LED:

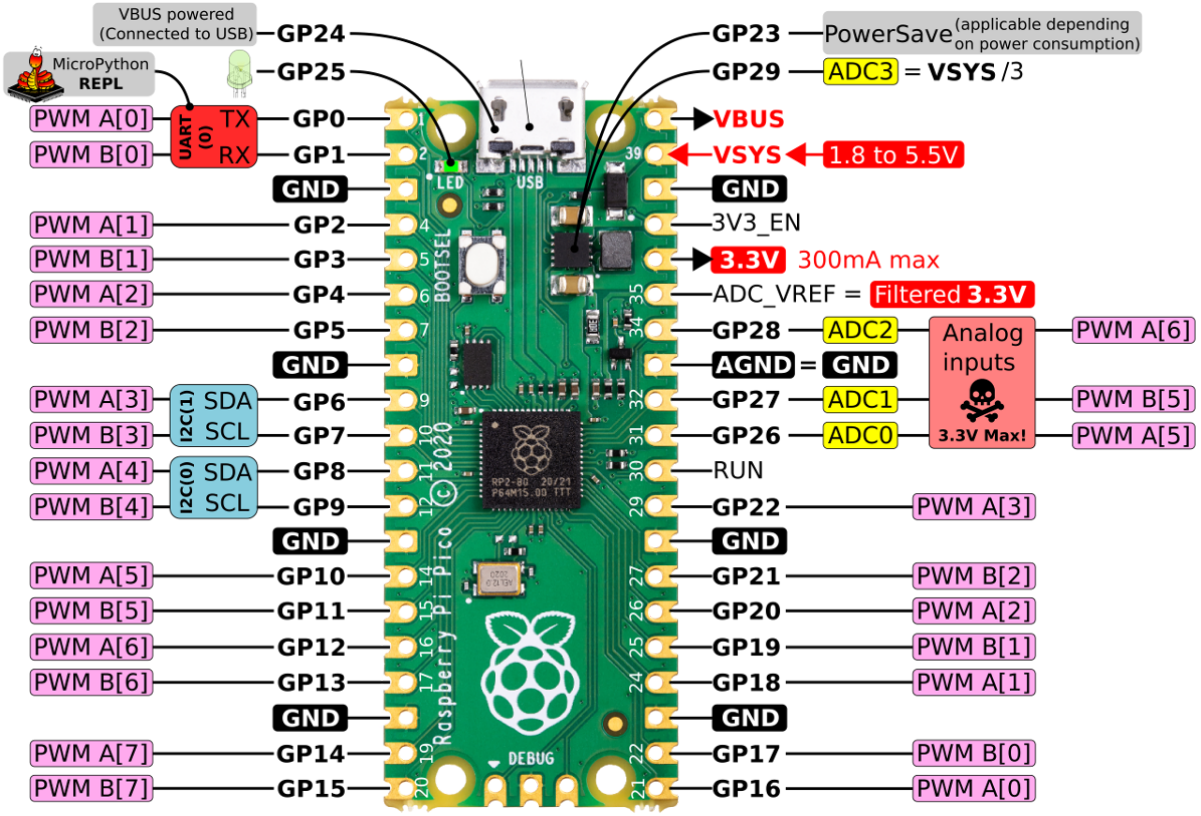
```
1 # Exemple utilisant PWM pour faire pulser une LED.
2
3 import time
4 from machine import Pin, PWM
5
6
7 # Construire l'objet PWM, avec LED sur broche GPIO 25.
8 pwm = PWM(Pin(25))
9
10 # Fixer la fréquence PWM.
11 pwm.freq(1000)
12
13 # Pulser la LED plusieurs fois.
14 duty = 0
15 direction = 1
16 for _ in range(8 * 256):
17   duty += direction
18   if duty > 255:
19     duty = 255
20     direction = -1
21   elif duty < 0:
22     duty = 0
23     direction = 1
24   pwm.duty_u16(duty * duty)
25   time.sleep(0.001)
```

Examples Pico MicroPython:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pwm/pwm_fade.py Lines 1 - 25

3.9.2 Organisation du module PWM

Tous les GPIOs du Pico sont capables de produire un signal PWM mais pas sur tous les GPIOs en même temps!

Le contrôleur PWM du Pico dispose de 8 générateurs (appelés *PWM Slice* en anglais) numérotés de 0 à 7. Chaque générateur dispose de deux canaux de sortie A et B. Ces 16 sorties PWM sont réparties sur les 26 GPIOs, il y a donc de la redondance.



Le graphique révèle, par exemple, que les GPIOs GP0 et GP16 utilisent le même générateur et canal `PWM_A[0]`. Il n'est donc pas question d'utiliser du PWM en même temps sur ces deux GPIOs.

Autre subtilité commune à tous les générateurs PWM, les canaux A et B étant liés au même générateur, ils en partagent donc la fréquence. Par exemple, GP0 est sur le `PWM_A[0]` et `PWM_B[0]` sur GP1. En conséquence, la modification de la fréquence PWM sur GP1 sera également appliquée à GP0. Cette caractéristique peut parfois conduire à des bogues (signaux incohérents) dont l'origine pourrait facilement passer pour une bogue MicroPython.

3.9.2 LED RGB et PWM :TODO:

xxxx

3.9.3 Servo moteur :TODO:

xxxx

3.10. PIO - Programmable IO

Statut du support PIO
Le statut des développements du support PIO peut être consulté sur ce ticket Github. Ce support préliminaire de PIO peut être instable.

Pour le moment vous pouvez définir des blocs PIO (Programmable IO) et les utiliser dans le périphérique PIO. Vous trouverez plus de documentation sur le PIO au chapitre 3 de la fiche technique RP2040 et au chapitre 4 du livre Pico C/C++ SDK.

Le micro-python Pico Raspberry Pi a été doté d'un nouveau décorateur `@rp2.asm_pio`, ainsi que d'une classe `rp2.PIO`. La définition d'un programme PIO, et la configuration de la machine d'état, en 2 parties logiques :

- La définition du programme, y compris le nombre de broches utilisées et si ce sont des broches d'entrée/sortie, se trouve dans la définition `@rp2.asm_pio`. C'est proche de ce que l'outil `pioasm` du SDK Pico générerait à partir d'un fichier `.pio` (mais ici, tout est défini en Python).
- L'instanciation du programme, qui définit la fréquence de la machine d'état et les broches à lier. Celles-ci sont définies lors de la configuration d'un SM pour l'exécution d'un programme particulier.

L'objectif était de permettre à un programme d'être défini une fois et ensuite facilement instancié plusieurs fois (si nécessaire) avec différents GPIO. Un autre objectif était de faciliter les choses basiques sans s'encombrer d'une configuration PIO/SM trop lourde.

Exemple d'utilisation, pour faire clignoter la LED embarquée connectée à la GPIO 25,

```
1 import time
2 from rp2 import PIO, asm_pio
3 from machine import Pin
4
5 # Définit le programme de clignotement. Il utilise
  # un GPIO défini comme sortie dans l'instruction.
6 # Utitiser des délais pour rendre le clignotement visible.
7 @asm_pio(set_init=rp2.PIO.OUT_LOW)
8 def blink():
9   wrap_target()
10   set(pins, 1) [31]
11   nop() [31]
12   nop() [31]
13   nop() [31]
14   nop() [31]
15   set(pins, 0) [31]
16   nop() [31]
17   nop() [31]
18   nop() [31]
19   nop() [31]
```

```
20   wrap()
21
22 # Instantie une machine à état pour le programme
   # de clignotement, à 1000Hz, sur le Pin(25)
   # (LED de la carte rp2)
23 sm = rp2.StateMachine(0, blink, freq=1000,
                           set_base=Pin(25))
24 # Exécuter la machine d'état pendant 3 secondes.
25 # La LED devrait clignoter.
26 sm.active(1)
27 time.sleep(3)
28 sm.active(0)
```

Examples Pico MicroPython: :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_blink.py Lignes 1 - 28

ou via un `exec` explicite.

```
 1 # Exemple utilisant PIO pour allumer la LED
 2 # via un exec explicite.
 3 # Montre comment :
 4 # - utiliser set_init and set_base
 5 # - utiliser StateMachine.exec
 6
 7 import time
 8 from machine import Pin
 9 import rp2
10
11 # Définir un programme vide qui utilise une seule broche
12 @rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
13 def prog():
14 pass
15
16
17 # Construit la machine d'état, relie la Pin(25)
18 sm = rp2.StateMachine(0, prog, set_base=Pin(25))
19
20 # Active la Pin avec l'instruction exec
21 sm.exec("set(pins, 1)")
22
23 # Pause pendant 500ms.
24 time.sleep(0.5)
25
26 # Désactive la Pin avec l'instruction exec
27 sm.exec("set(pins, 0)")
```

Pico MicroPython Exemples :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_exec.py Lignes 1 - 27

Quelques points à noter,

- Toute la configuration du programme (par exemple, autopull) se fait dans le décorateur `@asm_pio`. Seules la fréquence et les broches de base sont définies dans le constructeur StateMachine.
- [n] est utilisé pour le délai, `.set(n)` est utilisé pour `sideset`
- L'assembleur détectera automatiquement si sideset est utilisé partout ou seulement sur quelques instructions, et positionne automatiquement le bit `SIDE_EN`

L'idée est que pour les 4 jeux de broches (in, out, set, sideset, sauf jmp) qui peuvent être connectés à une machine d'état, il y a les éléments suivants qui doivent être configurés pour chaque jeu :

1. GPIO de base
2. nombre de GPIO consécutifs
3. le sens initial du GPIO (en entrée ou en sortie)
4. valeur initiale du GPIO (0 ou 1)

Dans la conception de l'API Python pour les PIO, ces 4 éléments sont divisés en "déclaration" (éléments 2-4) et "instanciation" (élément 1). En d'autres termes, un programme est écrit avec les points 2-4 fixés pour ce programme (par exemple, un pilote WS2812 aurait 1 broche de sortie) et le point 1 est libre de changer sans changer le programme (par exemple, à quelle broche le WS2812 est connecté).

Ainsi, dans le décorateur `@asm_pio`, vous déclarez les points 2 à 4, et dans le constructeur `StateMachine`, vous indiquez la broche de base à utiliser (point 1). Cela facilite la définition d'un programme unique et son instanciation à plusieurs reprises sur différentes broches (vous ne pouvez pas vraiment changer les éléments 2 à 4 pour une instanciation différente du même programme, cela n'a pas vraiment de sens).

Et le même mot-clé arg (dans le cas de `sideset_pins`) est utilisé à la fois pour la déclaration et l'instanciation, pour montrer qu'ils sont liés.

Pour déclarer plusieurs broches dans le décorateur (le compte, c'est-à-dire le point 2 ci-dessus), on utilise un tuple/liste de valeurs. Et chaque élément du tuple/liste spécifie les éléments 3 et 4. Par exemple :

```
1 @asm_pio(set_pins=(PIO.OUT_LOW, PIO.OUT_HIGH, PIO.IN_LOW),
sideset_pins=PIO.OUT_LOW)
2 def foo():
3 ....
4
5 sm = StateMachine(0, foo, freq=10000, set_pins=Pin(15),
                    sideset_pins=Pin(22))
```

Dans cet exemple :

- il y a 3 broches connectées à la SM, et leur état initial (défini lors de la création de la StateMachine) est : sortie basse, sortie haute, entrée basse (utilisée pour le drain ouvert)

- il y a une sideset pin, état initial bas
- les 3 broches commencent à la `Pin(15)`
- la sideset pin 1 commence à la `Pin(22)`

La raison d'avoir les constantes `OUT_LOW`, `OUT_HIGH`, `IN_LOW` et `IN_HIGH` est que la valeur de la broche et la direction sont automatiquement définies avant le démarrage du programme PIO (au lieu de gaspiller des instruction pour faire `set(pindirs, 1)` etc au démarrage).

3.10.1 IRQ

Les IRQ du PIO sont gérées, par exemple

```
 1 import time
 2 import rp2
 3
 4 @rp2.asm_pio()
 5 def irq_test():
 6   wrap_target()
 7   nop() [31]
 8   nop() [31]
 9   nop() [31]
10   nop() [31]
11   irq(0)
12   nop() [31]
13   nop() [31]
14   nop() [31]
15   nop() [31]
16   irq(1)
17   wrap()
18
19
20 rp2.PIO(0).irq(lambda pio: print(pio.irq().flags()))
21
22 sm = rp2.StateMachine(0, irq_test, freq=1000)
23 sm.active(1)
24 time.sleep(1)
25 sm.active(0)
```

Exemples de MicroPython Pico :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_irq.py Lignes 1 - 25

Un exemple de programme qui clignote à 1 Hz et positionne une IRQ à 1 Hz pour imprimer l'horodatage actuel en millisecondes,

```
1 # Exemple utilisant un PIO pour faire clignoter
```

```
2 # une LED et déclencher une IRQ à 1Hz
3 import time
4 from machine import Pin
5 import rp2
6
7
8  @rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
9  def blink_1hz():
10   # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
11   irq(rel(0))
12   set(pins, 1)
13   set(x, 31) [5]
14   label("delay_high")
15   nop() [29]
16   jmp(x_dec, "delay_high")
17
18   # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
19   set(pins, 0)
20   set(x, 31) [6]
21   label("delay_low")
22   nop() [29]
23   jmp(x_dec, "delay_low")
24
25
26 # Crée une machine d'état avec le programme blink_1hz,
   # sortant sur la broche GPIO 25.
27 sm = rp2.StateMachine(0, blink_1hz, freq=2000,
                        set_base=Pin(25))
28
29 # Configure le gestionnaire d'interruption pour qu'il
   #  affiche l'heure en millisecondes
30 sm.irq(lambda p: print(time.ticks_ms()))
31
32 # Démarre la machine d'état.
33 sm.active(1)
```

Pico MicroPython Examples :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_1hz.py Lignes 1 - 33

ou attendre un changement de valeur sur une broche et positionner une IRQ.

```
1 # Exemple utilisant un PIO pour attendre le changement
2 # d'état d'une pin et déclencher une IRQ.
3 # Montre comment :
4 # - wrapper un PIO
5 # - instruction wait pour mettre un PIO
  #   en attente sur une pin d'entrée
6 # - Instruction irq sur PIO, mode bloqué avec numéro
  #   d'IRQ relatif relative IRQ number
```

```
7 # - Configurer la broche in_base pour une machine d'état
8 # - configurer un gestionnaire d'IRQ pour une
  #   machine d'état
9 # - Instancier 2 machines avec le même programme
10 #   mais différentes broches
11 import time
12 from machine import Pin
13 import rp2
14
15
16 @rp2.asm_pio()
17 def wait_pin_low():
18   wrap_target()
19
20   wait(0, pin, 0)
21   irq(block, rel(0))
22   wait(1, pin, 0)
23
24   wrap()
25
26
27 def handler(sm):
28   # Afficher (wrapping) un timestamp, et
     #l'objet Statemachine.
29   print(time.ticks_ms(), sm)
30
31
32 # Instancier StateMachine(0) avec le programme
   # wait_pin_low sur la Pin(16).
33 pin16 = Pin(16, Pin.IN, Pin.PULL_UP)
34 sm0 = rp2.StateMachine(0, wait_pin_low, in_base=pin16)
35 sm0.irq(handler)
36 # Instancier StateMachine(1) avec le
37 # programme wait_pin_low sur la Pin(17).
38 pin17 = Pin(17, Pin.IN, Pin.PULL_UP)
39 sm1 = rp2.StateMachine(1, wait_pin_low, in_base=pin17)
40 sm1.irq(handler)
41
42 # Démarrer les machines d'état.
43 sm0.active(1)
44 sm1.active(1)
45
46 # Maintenant si les Pin(16) ou Pin(17) sont mises à 0,
   # un message est affiché sur REPL.
```

Exemples de MicroPython Pico :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_pinchange.py Lignes 1 - 46

3.10.2. WS2812 LED (NeoPixel)

Une LED WS2812 (NeoPixel) peut être pilotée par le programme suivant,

```
 1 # Exemple utilisant PIO pour commander un
 2 #                   jeu de LEDs WS2812.
 3 import array, time
 4 from machine import Pin
 5 import rp2
 6
 7 # Configure le nombre de LEDs WS2812.
 8 NUM_LEDS = 8
 9
10
11 @rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW,
               out_shiftdir=rp2.PIO.SHIFT_LEFT,
               autopull=True,   pull_thresh=24)
12 def ws2812():
13   T1 = 2
14   T2 = 5
15   T3 = 3
16   wrap_target()
17   label("bitloop")
18   out(x, 1) .side(0) [T3 - 1]
19   jmp(not_x, "do_zero") .side(1) [T1 - 1]
20   jmp("bitloop") .side(1) [T2 - 1]
21   label("do_zero")
22   nop() .side(0) [T2 - 1]
23   wrap()
24
25
26 # Crée la machine d'état avec le programme ws2812
   # sortant sur la Pin(22).
27 sm = rp2.StateMachine(0, ws2812, freq=8_000_000,
                         sideset_base=Pin(22))
28 # Démarrer la machine d'état, elle attendra les
29 # data sur son FIFO
30 sm.active(1)
31
32 # Affiche un motif sur les LEDs via un tableau de
   # valeurs de LEDs RGB
33 ar = array.array("I", [0 for _ in range(NUM_LEDS)])
34
35 # Rotation des couleurs
36 for i in range(4 * NUM_LEDS):
37   for j in range(NUM_LEDS):
38     r = j * 100 // (NUM_LEDS - 1)
```

```
39      b = 100 - j * 100 // (NUM_LEDS - 1)
40      if j != i % NUM_LEDS:
41        r >>= 3
42        b >>= 3
43      ar[j] = r << 16 | b
44    sm.put(ar, 8)
45    time.sleep_ms(50)
46
47  # Diminution de luminosité.
48  for i in range(24):
49    for j in range(NUM_LEDS):
50      ar[j] >>= 1
51    sm.put(ar, 8)
52    time.sleep_ms(50)
```

Exemples de MicroPython Pico :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_ws2812.py Lignes 1 - 52

3.10.3. UART TX

Un exemple d'UART TX,

```
1  # Exemple utilisant PIO pour créer une interface TX UART
2
3  from machine import Pin
4  from rp2 import PIO, StateMachine, asm_pio
5
6  UART_BAUD = 115200
7  PIN_BASE = 10
8  NUM_UARTS = 8
9
10
11 @asm_pio(sideset_init=PIO.OUT_HIGH, out_init=PIO.OUT_HIGH,
                out_shiftdir=PIO.SHIFT_RIGHT)
12 def uart_tx():
13   # Bloque avec TX désactivé jusqu'à ce que des données
     # soient disponibles
14   pull()
15   # Initialise le compteur de bits, pour 8 cycles
16   set(x, 7) .side(0) [7]
17   # Shift 8 data bits, 8 cycles par bit
18   label("bitloop")
19   out(pins, 1) [6]
20   jmp(x_dec, "bitloop")
21   # Positionne le stop bit pour un total de 8 cycles
     # (inclus 1 pour pull())
22   nop() .side(1) [6]
23
```

```
24
25 # Maintenant nous avons 8 TX UART sur les pins 10 to 17.
   # Utilise la même vitesse de transmission (baudrate)
   # pour toutes les sorties.
26 uarts = [ ]
27 for i in range(NUM_UARTS):
28   sm = StateMachine( i, uart_tx,
29       freq=8*UART_BAUD, sideset_base=Pin(PIN_BASE + i),
         out_base=Pin(PIN_BASE + i)
30       )
31   sm.active(1)
32   uarts.append(sm)
33
34 # Nous pouvons envoyer des caractères sur chaque UART
   # en les poussant dans la pile FIFO TX
35 def pio_uart_print(sm, s):
36   for c in s:
37     sm.put(ord(c))
38
39
40 # Envoyer un message différent sur chaque UART
41 for i, u in enumerate(uarts):
42   pio_uart_print(u, "Hello from UART {}!\n".format(i))
```

Pico MicroPython Exemples :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_uart_tx.py Lignes 1 - 42

NOTE
Vous devez spécifier un état initial de la broche OUT dans votre programme afin de pouvoir passer le mappage OUT à votre instanciation SM, même si dans ce programme il est redondant car les mappages se chevauchent.

3.10.4. SPI

Un exemple de SPI.

```
1 from machine import Pin
2
3 @rp2.asm_pio(out_shiftdir=0, autopull=True, pull_thresh=8,
           autopush=True, push_thresh=8,
           sideset_init=(rp2.PIO.OUT_LOW, rp2.PIO.OUT_HIGH),
           out_init=rp2.PIO.OUT_LOW)
4 def spi_cpha0():
5   # Note X must be preinitialised by setup code before
    # first byte, we reload after sending   each byte
6   # Would normally do this via exec() but in this case
```

```
    # it's in the instruction memory and is   only run once
7   set(x, 6)
8   # Actual program body follows
9   wrap_target()
10   pull(ifempty) .side(0x2) [1]
11   label("bitloop")
12   out(pins, 1) .side(0x0) [1]
13   in_(pins, 1) .side(0x1)
14   jmp(x_dec, "bitloop") .side(0x1)
15
16   out(pins, 1) .side(0x0)
17   set(x, 6) .side(0x0) # Note this could be replaced with
                          # mov x, y for programmable frame
                          # size
18   in_(pins, 1) .side(0x1)
19   jmp(not_osre, "bitloop").side(0x1)# Fallthru if TXF
20                                     # empties
21   nop() .side(0x0) [1] # CSn back porch
22   wrap()
23
24
25 class PIOSPI:
26
27   def __init__(self, sm_id, pin_mosi, pin_miso, pin_sck,
                  cpha=False, cpol=False, freq=1000000):
28     assert(not(cpol or cpha))
29     self._sm = rp2.StateMachine(sm_id, spi_cpha0,
                  freq=4*freq, sideset_base=Pin(pin_sck),
                  out_base=Pin(pin_mosi),
                  in_base=Pin(pin_sck))
30     self._sm.active(1)
31
32   # Note this code will die spectacularly cause we're not
     #draining the RX FIFO
33   def write_blocking(wdata):
34     for b in wdata:
35       self._sm.put(b << 24)
36
37   def read_blocking(n):
38     data = []
39     for i in range(n):
40       data.append(self._sm.get() & 0xff)
41     return data
42
43   def write_read_blocking(wdata):
44     rdata = []
45     for b in wdata:
46       self._sm.put(b << 24)
```

```
47        rdata.append(self._sm.get() & 0xff)
48      return rdata
```

Pico MicroPython Exemples :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_spi.py Lignes 1 - 48

NOTE
Ce programme SPI prend en charge les tailles de trame programmables (en maintenant la valeur de rechargement du compteur X dans le registre Y) mais actuellement, il ne peut pas être utilisé, car le seuil d'autopull est associé au programme, au lieu de l'instanciation SM.

3.10.5. PWM

Un exemple de PWM,

```
1 # Example of using PIO for PWM, and fading the brightness of
an LED
2
3 from machine import Pin
4 from rp2 import PIO, StateMachine, asm_pio
5 from time import sleep
6
7
8 @asm_pio(sideset_init=PIO.OUT_LOW)
9 def pwm_prog():
10    pull(noblock) .side(0)
11    mov(x, osr) # Maintient la donnée la plus récente en
                  #    cache dans X, pour le recyclage par
                  #    noblock
12    mov(y, isr) # ISR doit etre préchargé avec count_max PWM
13    label("pwmloop")
14    jmp(x_not_y, "skip")
15    nop() .side(1)
16    label("skip")
17    jmp(y_dec, "pwmloop")
18
19
20 class PIOPWM:
21    def __init__(self, sm_id, pin, max_count, count_freq):
22      self._sm = StateMachine(sm_id, pwm_prog,
                 freq=2 * count_freq,
                 sideset_base=Pin(pin))
23      # Charger max_count dans ISR avec exec()
24      self._sm.put(max_count)
```

```
25      self._sm.exec("pull()")
26      self._sm.exec("mov(isr, osr)")
27      self._sm.active(1)
28      self._max_count = max_count
29
30    def set(self, value):
31      # Valeur min = -1 (vraiment éteint), la valeur 0
        # laisse encore un très étroite pulsation
32      value = max(value, -1)
33      value = min(value, self._max_count)
34      self._sm.put(value)
35
36
37  # GPIO 25 = LED sur la carte Pico
38  pwm = PIOPWM(0, 25, max_count=(1 << 16) - 1,
                 count_freq=10_000_000)
39
40  while True:
41    for i in range(256):
42      pwm.set(i ** 2)
43      sleep(0.01)
```

Pico MicroPython Exemples :
https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_pwm.py Lignes 1 - 43

3.10.6. Utiliser `pioasm`

En plus d'écrire du code PIO en ligne dans votre script MicroPython, vous pouvez utiliser l'outil `pioasm` du SDK C/C++ pour générer un fichier Python.

```
$ pioasm -o python input (output)
```

Pour plus d'informations sur `pioasm`, voir le livre Pico C/C++ SDK qui parle du SDK C/C++.

Chapitre 4. Utilisation d'un Environnement de développement Intégré (IDE)

Des paquets sont disponibles pour Linux, MS Windows et macOS. Après l'installation, l'utilisation de l'environnement de développement Thonny est la même sur les trois plateformes. La dernière version (version 3.3.0) de Thonny peut être téléchargée sur Github à l'adresse https://github.com/thonny/thonny/releases/tag/v3.3.0.

NOTE
Après la date de lancement, Thonny devrait être téléchargé à partir de thonny.org

Alternativement, si vous travaillez sur un Raspberry Pi, vous devez installer Thonny en utilisant `apt` en ligne de commande,

```
$ sudo apt install thonny
```

Cela ajoutera une icône Thonny au menu du Raspberry Pi dans : Allez dans Menu Raspberry Pi -> Programmation -> IDE Thonny Python pour ouvrir l'environnement de développement.

NOTE
Lorsque vous ouvrez Thonny pour la première fois, sélectionnez "Mode Standard". Pour certaines versions, ce choix sera fait via une fenêtre popup lors de la première ouverture de Thonny. Cependant, pour la version Raspberry Pi, vous devez cliquer sur le texte en haut à droite de la fenêtre pour passer en "Mode Standard".

Téléchargez les fichiers nécessaires à Pico sur Github, https://github.com/raspberrypi/thonny-pico/releases/download/v0.1-post/thonny_rpi_pico-0.1-py3-none-any.whl. Ce fichier .whl peut être installé dans une installation Thonny existante (version 3.3.0b6 ou supérieure)

Démarrez Thonny et allez dans "Outils -> Gérer les plug-ins" et cliquez sur le lien "Installer à partir d'un fichier local" dans le panneau de droite, puis sélectionnez le fichier pour Pico (voir figure 3). Cliquez sur le bouton "Fermer" pour terminer. Ensuite, vous devez **quitter et redémarrer Thonny**.

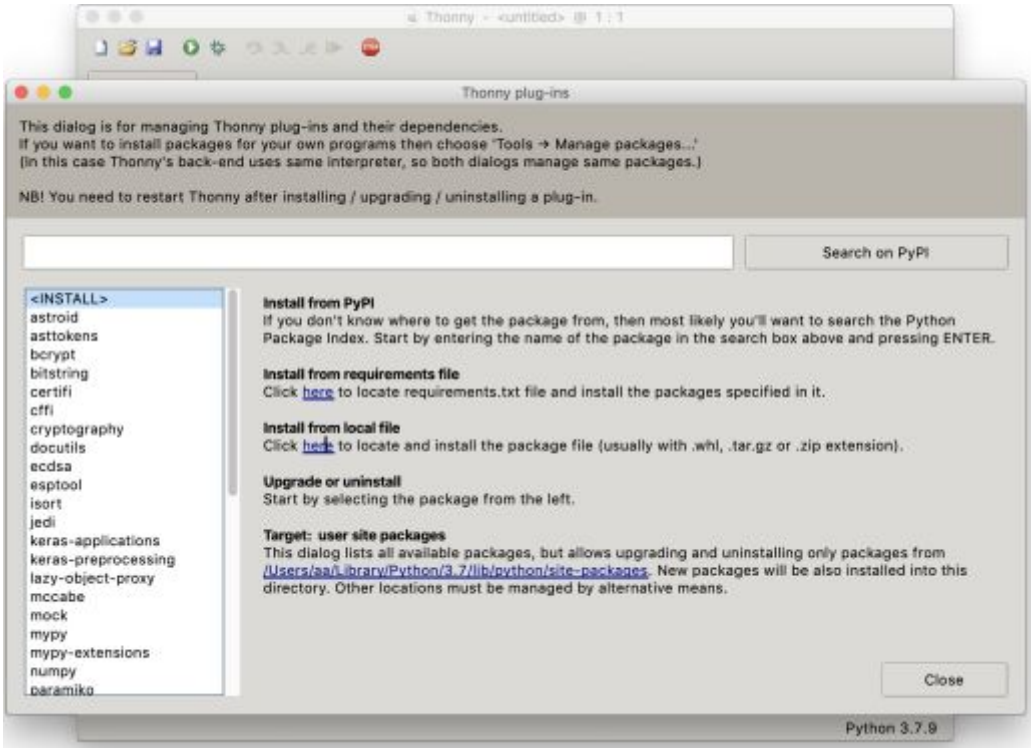


Figure 3. Installing the Raspberry Pi Pico Wheel file.

TO DO: Rewrite this section ahead of launch

4.1.1. Connexion au Raspberry Pi Pico depuis Thonny

Connectez votre ordinateur et le Raspberry Pi Pico, voir le chapitre 2. Ensuite, ouvrez le menu Exécuter et sélectionnez Exécuter -> Sélectionner l'interpréteur, en choisissant "MicroPython (Raspberry Pi Pico)" dans le menu déroulant, voir la figure 4.

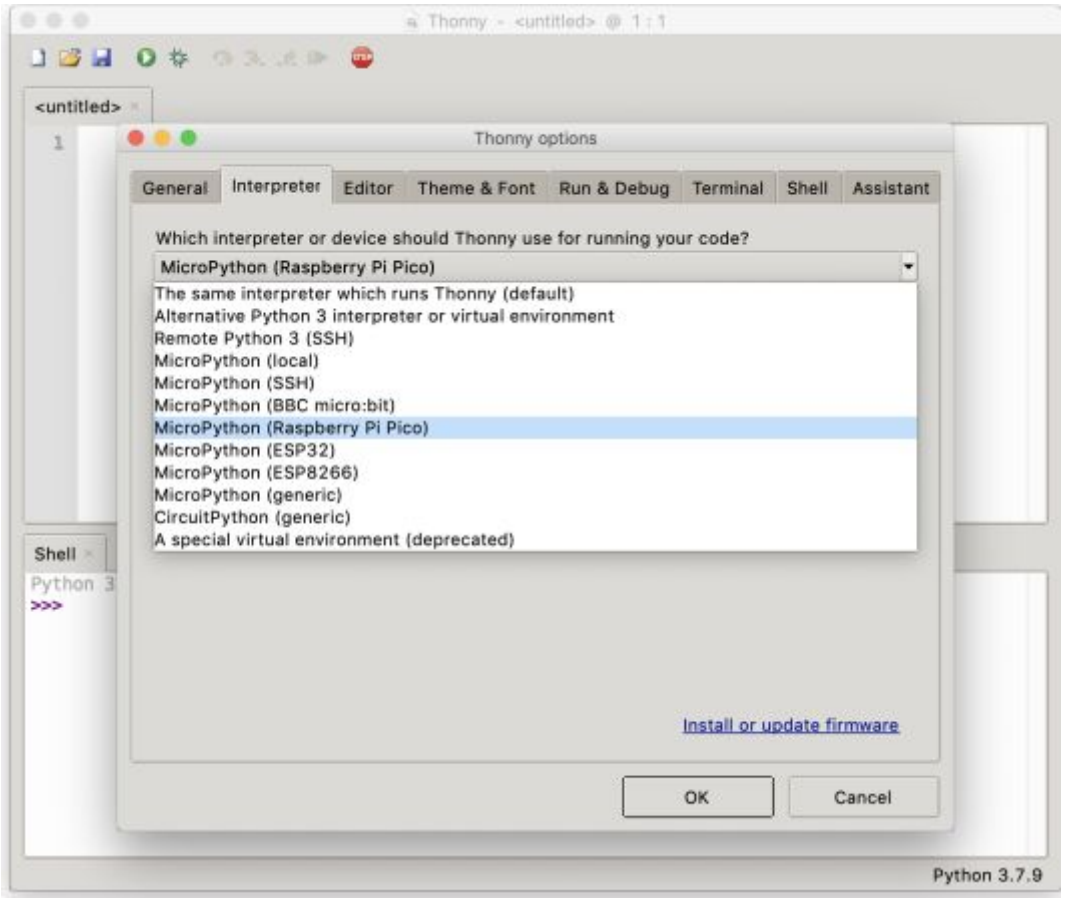


Figure 4. Selecting the correct MicroPython interpreter inside the Thonny environment.

Appuyez sur "OK". Si votre Raspberry Pi Pico est branché et qu'il fonctionne avec MicroPython, Thonny devrait se connecter automatiquement au REPL.

Si ce n'est pas le cas, allez dans le menu Outils -> Options, et sélectionnez votre port série dans le menu déroulant de l'onglet "Interpréteur". Sur le Raspberry Pi, le port série sera "Board in FS Mode - Board CDC (/dev/ttyACM0)", ce qui devrait vous connecter automatiquement au REPL du Raspberry Pi Pico. Ensuite, allez dans le menu "View" et sélectionnez l'option "Variables" pour ouvrir le panneau des variables.

NOTE
Dans le rare cas où vous ne pouvez pas vous connecter au Raspberry Pi Pico, vous devrez peut-être redémarrer votre Raspberry Pi.

Vous pouvez maintenant accéder au REPL depuis le panneau Shell,

```
>>> print('Hello Pico!')
```

```
Hello Pico!
>>>
```

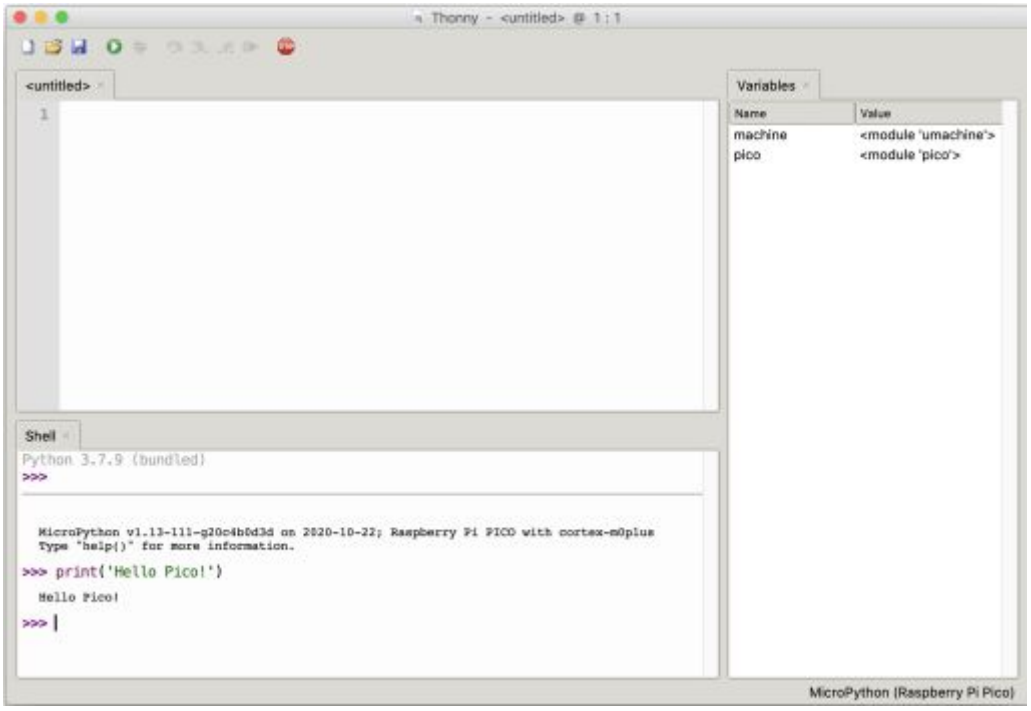


Figure 5. Saying "Hello Pico!" from the MicroPython REPL inside the Thonny environment.

4.1.2. Clignotement de la LED avec Thonny

Vous pouvez utiliser un timer pour faire clignoter la LED embarquée.

```
1 from machine import Pin, Timer
2
3 led = Pin(25, Pin.OUT)
4 tim = Timer()
5 def tick(timer):
6 global led
7 led.toggle()
8
9 tim.init(freq=2.5, mode=Timer.PERIODIC, callback=tick)
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/blink/blink.py Lines 1 - 9

Saisissez le code dans le panneau principal, puis cliquez sur le bouton vert de lancement. Thonny vous présentera un popup, cliquez sur "MicroPython device" et entrez "test.py" pour enregistrer le code dans le Raspberry Pi Pico voir figure 6.

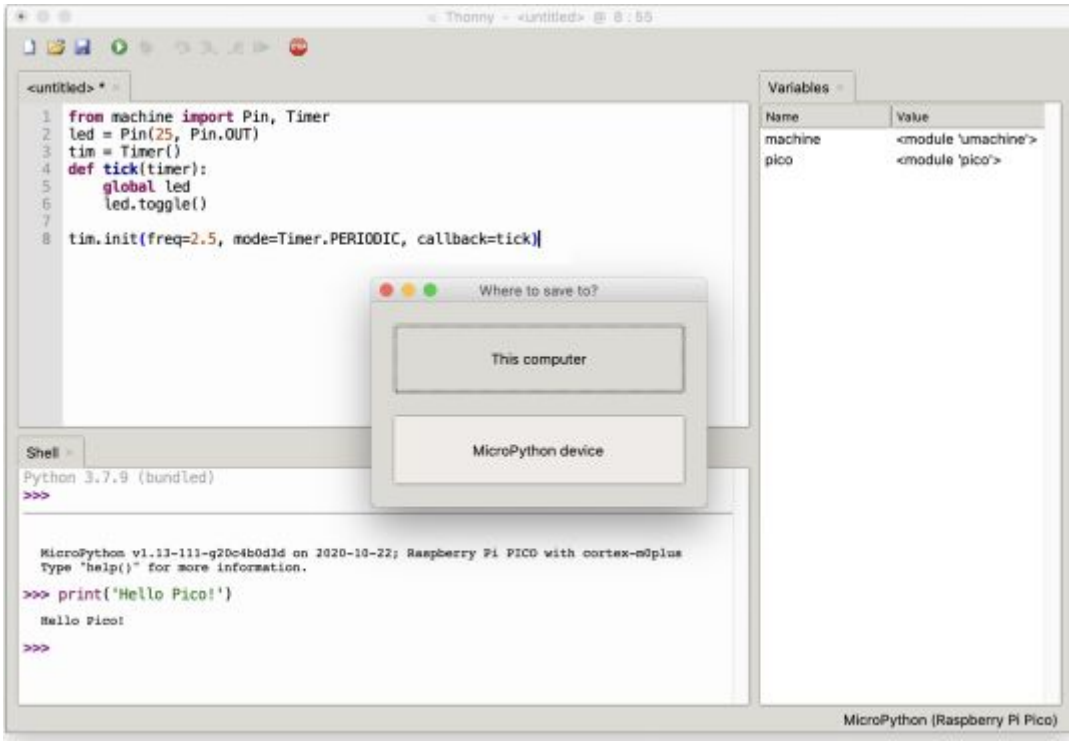


Figure 6. Saving code to the Raspberry Pi Pico inside the Thonny environment.

NOTE
Si vous "enregistrez un fichier sur l'appareil" et lui donnez le nom spécial main.py, alors MicroPython commencera à exécuter ce script dès que Raspberry Pi Pico sera alimenté.

Le programme devrait être téléchargé sur le Raspberry Pi Pico en utilisant le REPL, et commencer à s'exécuter automatiquement. Vous devriez voir la LED embarquée, connectée à la broche 25 du GPIO, se mettre à clignoter, et les variables changer dans la fenêtre de variables de Thonny, voir la figure 7.

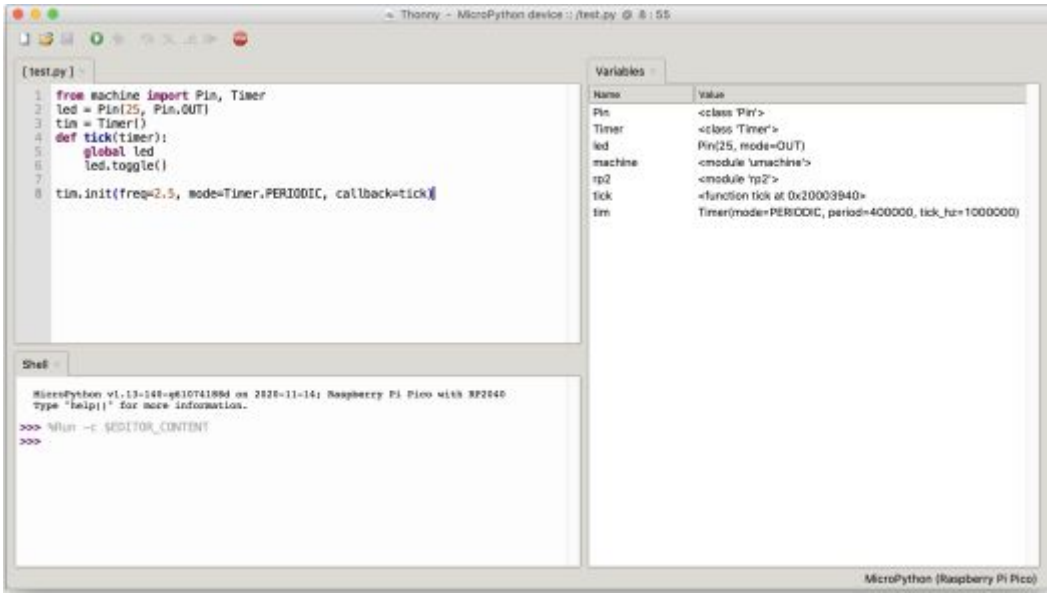


Figure 7. Blinking an LED using a timer from the Thonny environment.

4.2. Utilisation de rshell

Le Remote Shell pour MicroPython (rshell) est un simple shell qui fonctionne sur l'hôte et utilise le REPL de MicroPython pour envoyer du code python au Raspberry Pi Pico afin d'obtenir des informations sur le système de fichiers, et pour copier des fichiers vers et depuis le propre système de fichiers de MicroPython.

Comme RShell exploite la communication série, il faut veiller à clôturer votre éventuelle session terminal/minicom/screen/putty avant de démarrer RShell.

Avec RShell est l'utilitaire recommandé par la fondation pour communiquer avec votre carte MicroPython. François Mocq de Framboise314 à eu l'occasion de testé Ampy, raison pour laquelle il est documenté ci dessous.

Vous pouvez installer RShell en utilisant ,

```
$ sudo apt install python3-pip
$ sudo pip3 install rshell
```

NOTE
Il est recommandé de réduire la taille de la mémoire tampon utilisée par RShell pour les échanges avec la carte Pico. La taille recommandée est de 32 octets (au lieu de 512).
`rshell -p /dev/serial0 --buffer-size 32`

4.2.1 depuis l'UART du Raspberry Pi

En supposant que l'UART est connecté à votre Raspberry Pi, voir la section 2.1, vous pouvez alors vous connecter à Raspberry Pi Pico en utilisant,

```
$ rshell -p /dev/serial0 --buffer-size 32
Connecting to /dev/serial0 (buffer-size 32)...
Trying to connect to REPL
connected
Testing if sys.stdin.buffer exists ... N
Retrieving root directories ...
Setting time ... Aug 21, 2020 15:35:18
Evaluating board_name ... pyboard
Retrieving time epoch ... Jan 01, 2000
Welcome to rshell. Use Control-D (or the exit command) to exit
rshell.
/home/pi>
```

4.2.2 Via USB

La connexion USB du Pico expose également un bus série (Série-via-USB). Une connexion série peut également être établie par le biais de cette connexion.
Si vous branchez le pico sur le port USB d'un RaspberryPi (ou d'une machine Linux), la connexion série apparaît sous le nom `/dev/ttyUSB0` (ou équivalent).

Dans ce cas, la communication est établie à l'aide de la commande:

```
$ rshell -p /dev/ttyUSB0 --buffer-size 32
Connecting to /dev/serial0 (buffer-size 32)...
Trying to connect to REPL
```

4.2.3 Documentation RShell

La documentation complète de `rshell` se trouve sur le dépôt Github du projet : project's Github repository.
Une traduction française de cette documentation est disponible sur le Wiki de MCHobby.

4.3. Utilisation de Ampy

Ampy est un outil aux fonctionnalités équivalentes à RShell produit par Adafruit Industries. Plutôt que de proposer un "shell" pour manipuler la carte MicroPython, Ampy propose un utilitaire en ligne de commande. Les différents paramètres de la ligne de commande AMPY permettent d'envoyer un script/fichier, afficher le contenu d'un fichier, exécuter un script et récupérer le résultat de l'exécution.

Comme RShell exploite la communication série, il faut veiller à clôturer votre éventuelle session terminal/minicom/screen/putty avant de démarrer RShell.

Si l'outil Ampy est moins complet que RShell, il en reste néanmoins très efficace et permet parfois de se sortir d'un mauvais pas.

Vous pouvez installer Ampy en utilisant ,

```
$ sudo apt install python3-pip
$ sudo pip3 install adafruit-ampy
```

4.3.1 Aide mémoire

voici un petit aide mémoire des commandes Amoy utilisable avec votre Pico.

Exécuter le fichier test.py sur la carte et récupérer le résultat.

```
ampy --port /port/serie run test.py
```

Copier le fichier test.py sur la carte

```
ampy --port /port/série put test.py
```

Copier le fichier test.py dans un sous répertoire de la carte EN le renommant bar.py

```
ampy --port /serial/port put test.py /foo/bar.py
```

Récupérer le fichier boot.py depuis la carte (et en faire une copie locale)

```
ampy --port /serial/port get boot.py
```

Récupérer le fichier boot.py depuis la carte (et lui donner un autre nom sur le système de fichiers local)

```
ampy --port /serial/port get boot.py board_boot.py
```

Lister les fichiers présents sur la carte

```
ampy --port /serial/port ls
```

Effacer le fichier test.py présent sur la carte

```
ampy --port /serial/port rm test.py
```

4.3.2 Documentation

Une documentation Française de Ampy est disponible sur le Wiki de MCHobby
https://wiki.mchobby.be/index.php?title=FEATHER-CHARGER-FICHIER-MICROPYTHON-AMPY

Annexe A: Notes d’application

Utilisation d'un affichage graphique OLED basé sur le SSD1306

Afficher une image et du texte sur un écran graphique OLED à base de SSD1306 piloté par I2C.

Raccordement

Voir la figure 8 pour le câblage et 8a pour les câblage des écrans OLED.

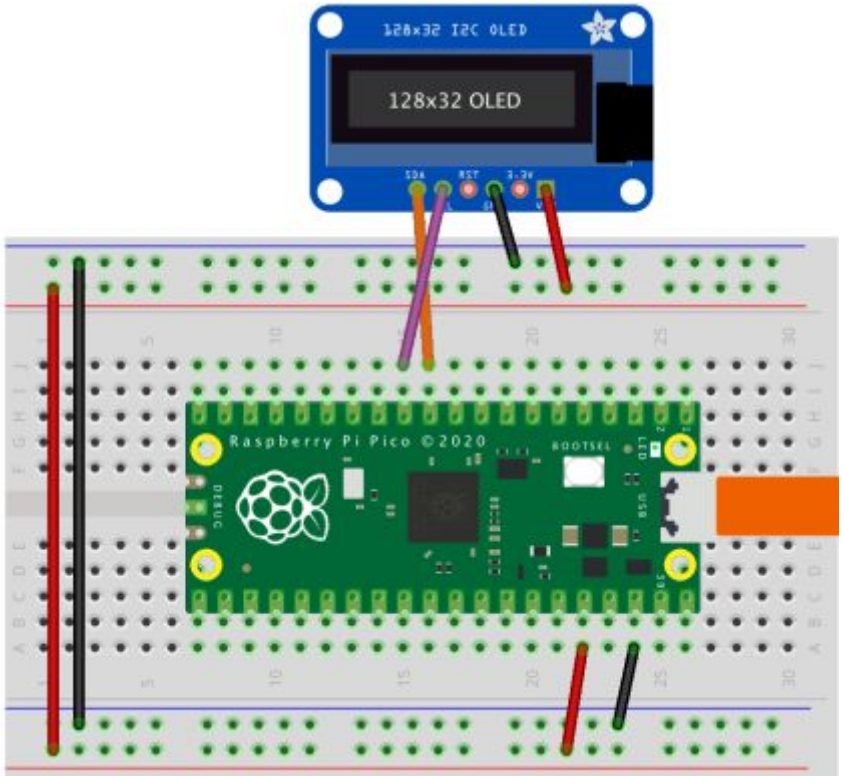


Figure 8. Brancher un OLED sur le Pico via I2C (OLED Adafruit)

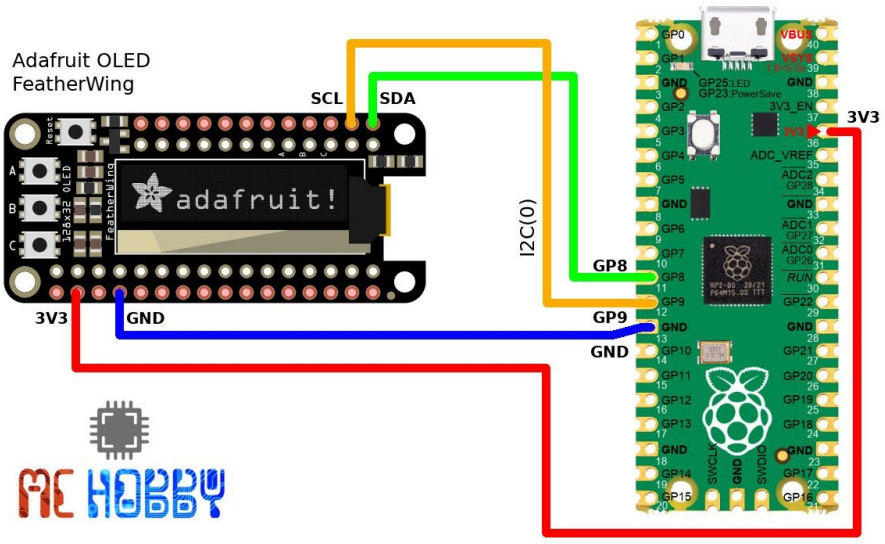
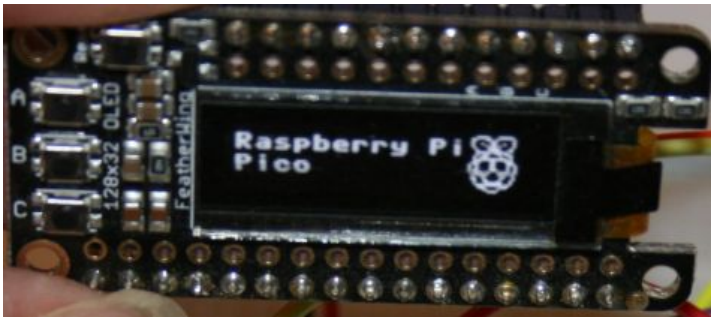


Figure 8a. Brancher un OLED FeatherWing

Liste des Fichiers

Une liste de fichiers avec la description de leur fonction :

- `ssd1306.py` : pilote micropython pour écran OLED SSD1306. Ce dernier est disponible sur le GitHub de MicroPython. https://github.com/micropython/micropython/blob/master/drivers/display/ssd1306.py
- `i2c_1306oled_using_defaults.py` : exemple utilisant le bus par défaut
- `i2c_1306oled_with_freq.py` : fixe explicitement le fréquence



i2c_1306oled_using_defaults.py

Exemple de code.

```
1 # Affiche des Images et du Texte sur un écran OLED ssd1306 OLED piloté en
I2C
2 from machine import Pin, I2C
3 from ssd1306 import SSD1306_I2C
4 import framebuf
5
6 WIDTH = 128 # Largeur OLED
7 HEIGHT = 32 # Hauteur OLED
```

```
8
9 i2c = I2C(0) # Init I2C(0) avec scl=GP9, sda=GP8, freq=400000
10 print("I2C Address: "+hex(i2c.scan()[0]).upper()) # affiche adresse
11 print("I2C Configuration: "+str(i2c)) # Affiche config I2C
12
13
14 oled = SSD1306_I2C(WIDTH, HEIGHT, i2c) # Init afficheur oled
15
16 # Raspberry Pi logo sous forme d'un tableau d'octets de 32x32
17 buffer
=bytearray(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00|?\x00\x01\x
86
@\x80\x01\x01\x80\x80\x01\x11\x88\x80\x01\x05\xa0\x80\x00\x83\xc1\x00\x00C\xe
3\x00\x00
~\xfc\x00\x00L'\x00\x00\x9c\x11\x00\x00\xbf\xfd\x00\x00\xe1\x87\x00\x01\xc1\x
83\x80\x02A
\x82@\x02A\x82@\x02\xc1\xc2@\x02\xf6>\xc0\x01\xfc
=\x80\x01\x18\x18\x80\x01\x88\x10\x80\x00\x8c!\x00\x00\x87\xf1\x00\x00\x7f\xf
6\x00\x00
8\x1c\x00\x00\x0c
\x00\x00\x03\xc0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")
18
19 # Charge le logo raspberry pi dans le framebuffer (l'image mesure 32x32)
20 fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)
21
22 # Efface l'ancien contenu de l'écran au cas où il y ait des choses dessus
23 oled.fill(0)
24
25 # Envoie l'image du framebuffer vers l'écran
26 oled.blit(fb, 96, 0)
27
28 # Ajouter du texte
29 oled.text("Raspberry Pi",5,5)
30 oled.text("Pico",5,15)
31
32 # Finalement faire afficher l'image et le texte par l'écran OLED.
33 oled.show()
```

Un exemplaire de ce code est disponible ici:
https://github.com/mchobby/pyboard-driver/blob/master/Pico/examples/i2c_1306oled.py

i2c_1306oled_with_freq.py

Exemple de code, fixe explicitement une fréquence

Voici le début de l'exemple précédent où la fréquence de bus est réduite à 200 KHz.

```
1 # Afficher image et texte sur afficheur OLED ssd1306 (en I2C)
2 from machine import Pin, I2C
3 from ssd1306 import SSD1306_I2C
4 import framebuf
```

```
5
6 WIDTH = 128 # largeur afficheur oled
7 HEIGHT = 32 # hauteur afficheur oled
8 # Initialiser l'I2C avec broche GP8 & GP9 (I2C0)
  # Afficher adresse périphérique
9 i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=200000)
10 print("I2C Adresse: "+ hex(i2c.scan()[0]).upper())
11 print("I2C Configuration: "+str(i2c)) # Affiche config I2C
12
13 # Initialiser l'afficheur OLED
14 oled = SSD1306_I2C(WIDTH, HEIGHT, i2c)
```

Pico MicroPython Examples:
https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1306oled/i2c_1306oled_with_freq.py
Lines 1 - 33

Liste de matériel

Désignation	Quantité	Détails
Breadboard	1	modèle générique
Raspberry-Pi Pico	1	http://raspberrypi.org/
Écran OLED Monochrome 128x32 I2C	1	https://www.adafruit.com/product/931
Ecran OLED FeatherWing 128x32 I2C	(alternative)	Adafruit

Utilisation de PIO pour piloter un ensemble de NeoPixel Ring (LED WS2812)

Combinaison de la démo PIO WS2812 avec l'exemple de code NeoPixel "essentiel" d'Adafruit pour montrer des remplissages de couleurs, des poursuites et bien sûr un arc-en-ciel sur un anneau de 16 LED.

Raccordement

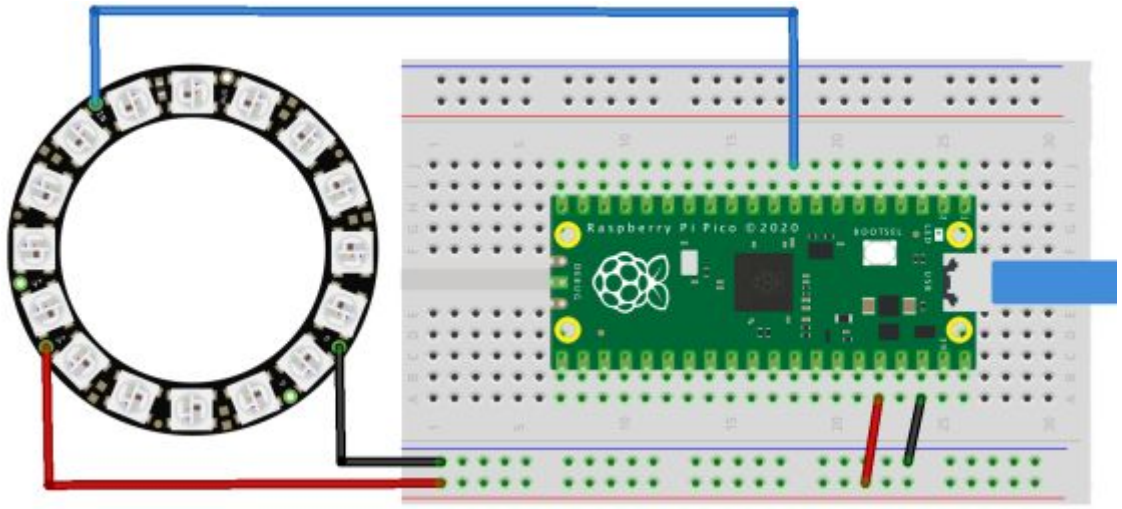


Figure 9. Wiring the 16-LED NeoPixel Ring to Pico (GP6)

Liste des Fichiers

Une liste de fichiers avec la description de leur fonction ;

- `neopixel_ring.py` : utilisation de NeoPixels avec Pico

```
1 # Exemple utilisant PIO pour piloter des LEDs WS2812 .
2
3 import array, time
4 from machine import Pin
5 import rp2
6
7 # Configurer le nombre de LEDs WS2812.
8 NUM_LEDS = 16
9 PIN_NUM = 6
10 brightness = 0.2
11
12 @rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW,
       out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True,
       pull_thresh=24)
13 def ws2812():
14   T1 = 2
15   T2 = 5
16   T3 = 3
17   wrap_target()
18   label("bitloop")
19   out(x, 1) .side(0) [T3 - 1]
20   jmp(not_x, "do_zero") .side(1) [T1 - 1]
21   jmp("bitloop") .side(1) [T2 - 1]
22   label("do_zero") 23 nop() .side(0) [T2 - 1]
24   wrap()
```

```
25
26 # Créer une machine à état avec le programme ws2812, sortie
27 # sur la broche GPIO 6.
28 sm = rp2.StateMachine(0, ws2812, freq=8_000_000,
           sideset_base=Pin(PIN_NUM))
29 # Démarrer la machine à état fini qui attend les données
30 # sur sa pile FIFO.
31 sm.active(1)
32
33 # Afficher le motif LEDs via un tableau de valeurs LED RGB.
34 ar = array.array("I", [0 for _ in range(NUM_LEDS)])
35
36 ##########################################################
37 def pixels_show():
38   dimmer_ar = array.array("I", [0 for _ in range(NUM_LEDS)])
39   for i,c in enumerate(ar):
40     r = int(((c >> 8) & 0xFF) * brightness)
41     g = int(((c >> 16) & 0xFF) * brightness)
42     b = int((c & 0xFF) * brightness)
43     dimmer_ar[i] = (g<<16) + (r<<8) + b
44   sm.put(dimmer_ar, 8)
45   time.sleep_ms(10)
46
47 def pixels_set(i, color):
48   ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]
49
50 def pixels_fill(color):
51   for i in range(len(ar)):
52     pixels_set(i, color)
53
54 def color_chase(color, wait):
55   for i in range(NUM_LEDS):
56     pixels_set(i, color)
57     time.sleep(wait)
58     pixels_show()
59   time.sleep(0.2)
60
61 def wheel(pos):
62   # Entre une valeur de 0 à 255 et obtenir une couleur.
63   # Transition de couleur de r - v - b - retour a r.
64   if pos < 0 or pos > 255:
65     return (0, 0, 0)
66   if pos < 85:
67     return (255 - pos * 3, pos * 3, 0)
68   if pos < 170:
69     pos -= 85
70     return (0, 255 - pos * 3, pos * 3)
71   pos -= 170
```

```
72    return (pos * 3, 0, 255 - pos * 3)
73
74
75  def rainbow_cycle(wait):
76    for j in range(255):
77      for i in range(NUM_LEDS):
78        rc_index = (i * 256 // NUM_LEDS) + j
79        pixels_set(i, wheel(rc_index & 255))
80      pixels_show()
81      time.sleep(wait)
82
83  BLACK = (0, 0, 0)
84  RED = (255, 0, 0)
85  YELLOW = (255, 150, 0)
86  GREEN = (0, 255, 0)
87  CYAN = (0, 255, 255)
88  BLUE = (0, 0, 255)
89  PURPLE = (180, 0, 255)
90  WHITE = (255, 255, 255)
91  COLORS = (BLACK, RED, YELLOW, GREEN, CYAN, BLUE, PURPLE, WHITE)
92
93  print("Remplir")
94  for color in COLORS:
95  pixels_fill(color)
96  pixels_show()
97  time.sleep(0.2)
98
99  print("chasse couleur")
100  for color in COLORS:
101    color_chase(color, 0.01)
102
103  print("rainbow")
104  rainbow_cycle(0)
```

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/neopixel_ring/neopixel_ring.py Lines 1 - 104

Liste de matériel

Désignation	Quantité	Détails
Breadboard	1	modèle générique
Raspberry-Pi Pico	1	http://raspberrypi.org/
NeoPixel Ring	1	https://www.adafruit.com/product/1463

Utilisation d'un shield moteur Adafruit

Adafruit propose un MotorShield (pour Arduino, Adafruit 2348) et un Motor FeatherWin (pour Feather, Adafruit 2927) tous deux utilisables avec le Raspberry-Pi Pico.

Ces deux shields utilisent une interface I2C pour être contrôlés depuis le Pico.
Attention cependant à configurer le Motor Shield pour une tension logique 3.3V (voir graphique).

Pour plus d'informations, voir le [GitHub esp8266-upy/adfmotors]

Raccordement

Voici les schémas de raccordement.

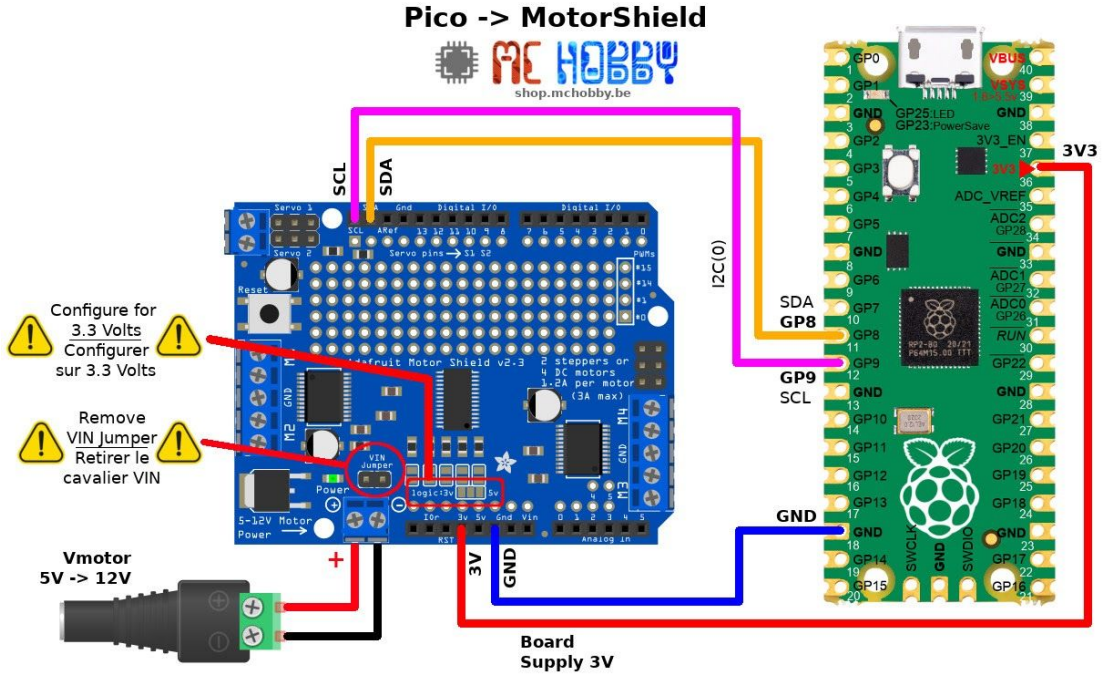


Figure 9: Brancher le MotorShield sur le Pico

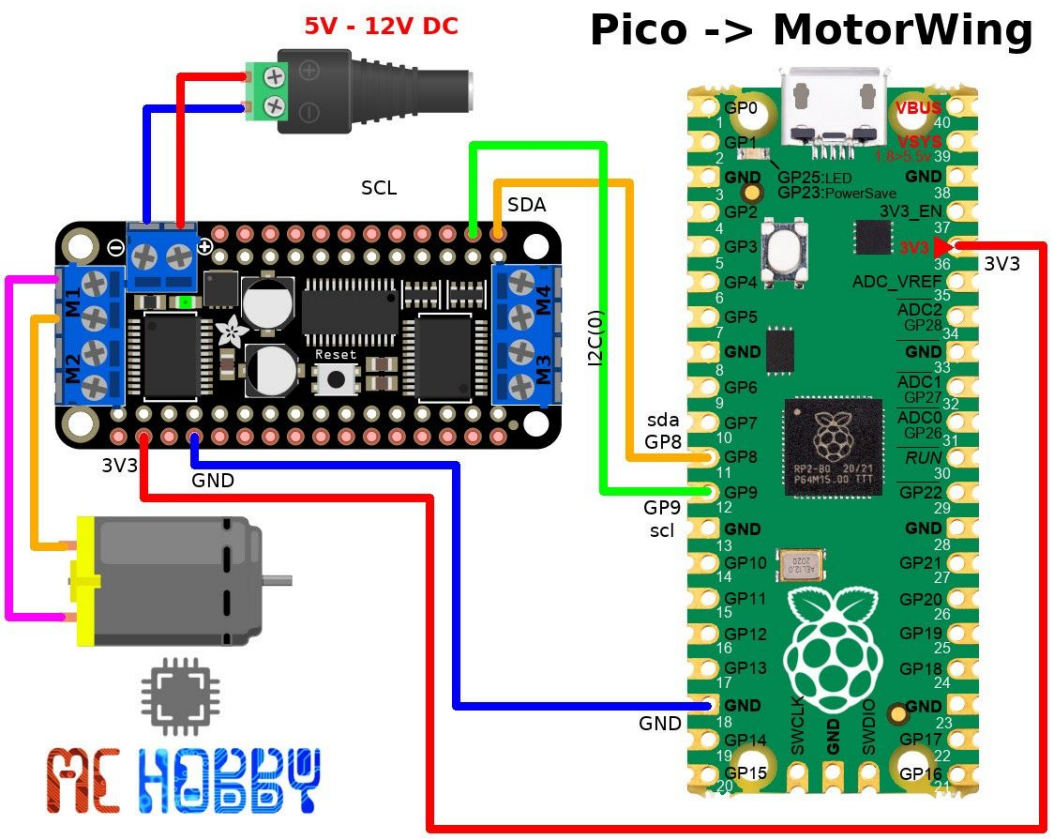


Figure 9a: Brancher le Motor FeatherWing sur le Pico

Liste des fichiers

Les fichiers nécessaires sont disponibles dans le répertoire GitHub esp8266-upy/adfmotors/lib/
Url complète: https://github.com/mchobby/esp8266-upy/tree/master/adfmotors/lib

- `pca9685.py`
- `motorbase.py`
- `motorshield.py` : pour le **MotorShield** d'Adafruit
- `motorwing.py` : pour le **Motor FeatherWing** d'Adafruit

L'interface de programmation de la bibliothèque est très proche de la version Arduino. Il est donc relativement simple de passer de C à MicroPython puisque les méthodes sont identiques.

A noter que pour utiliser:

- un **Motor shield**, il faudra instancier une classe `MotorShield` du module `motorshield.py` .
- un **Motor FeatherWing**, il faudra instancier une classe `MotorWing` du module `motorwing.py` .

Sinon, le reste des codes d'exemples et les fonctionnalités sont identiques entre MotorShield et Motor FeatherWing (à l'exception des broches PWM supplémentaires disponibles uniquement sur le MotorShield (mais pas sur le Motor FeatherWing).

Voici un exemple de contrôle du moteur continu branché sur le port M1 d'un Motor FeatherWing.

```
from machine import I2C
from motorwing import MotorWing
from motorbase import FORWARD, BACKWARD, BRAKE, RELEASE
from time import sleep

# Raspberry-Pi Pico - SDA=GP8, SCL=GP9
i2c = I2C(0)

# Test the various motors on the MotorShield
sh = MotorWing( i2c )
motor = sh.get_motor(1) # Moteur M1
try:
     motor.speed( 128 ) # Vitesse initiale
     motor.run( FORWARD )
     # Attendre arret du script par l utilisateur
     # en pressant Ctrl+C
     while True:
          sleep( 1 )
except KeyboardInterrupt:
     motor.run( RELEASE )

print( "That's all folks")
```

Exemple issu de
https://github.com/mchobby/esp8266-upy/blob/master/adfmotors/examples/motorwing/test_dcmotor_m1.py

Autres exemples

De nombreux autres exemples sont disponibles dans le dépôt.
https://github.com/mchobby/esp8266-upy/tree/master/adfmotors/examples

Liste de matériel

Désignation	Quantité	Détails
Breadboard	1	modèle générique
Raspberry-Pi Pico	1	http://raspberrypi.org/
Motor Shield	1	https://www.adafruit.com/product/1438

Motor FeatherWing	1	https://www.adafruit.com/product/2927
Moteur continu	1	https://www.adafruit.com/product/711
Connecteur d’alimentation	1	https://www.adafruit.com/product/368

Utilisation d'un TFT 2.4” FeatherWing (ILI934x)

Adafruit propose un afficheur TFT couleurs 16 bit en format FeatherWing (pour Feather, Adafruit 3315). Cet afficheur dispose d'une résolution de 320x240 et s’appuie sur contrôleur ILI9341 qui peut être utilisé avec un Raspberry-Pi Pico.

Cet afficheur utilise une interface SPI pour être piloté depuis le Pico.

Pour plus d’informations, voir le [GitHub esp8266-upy/ili934x]

Raccordement

Voici les schémas de raccordement.

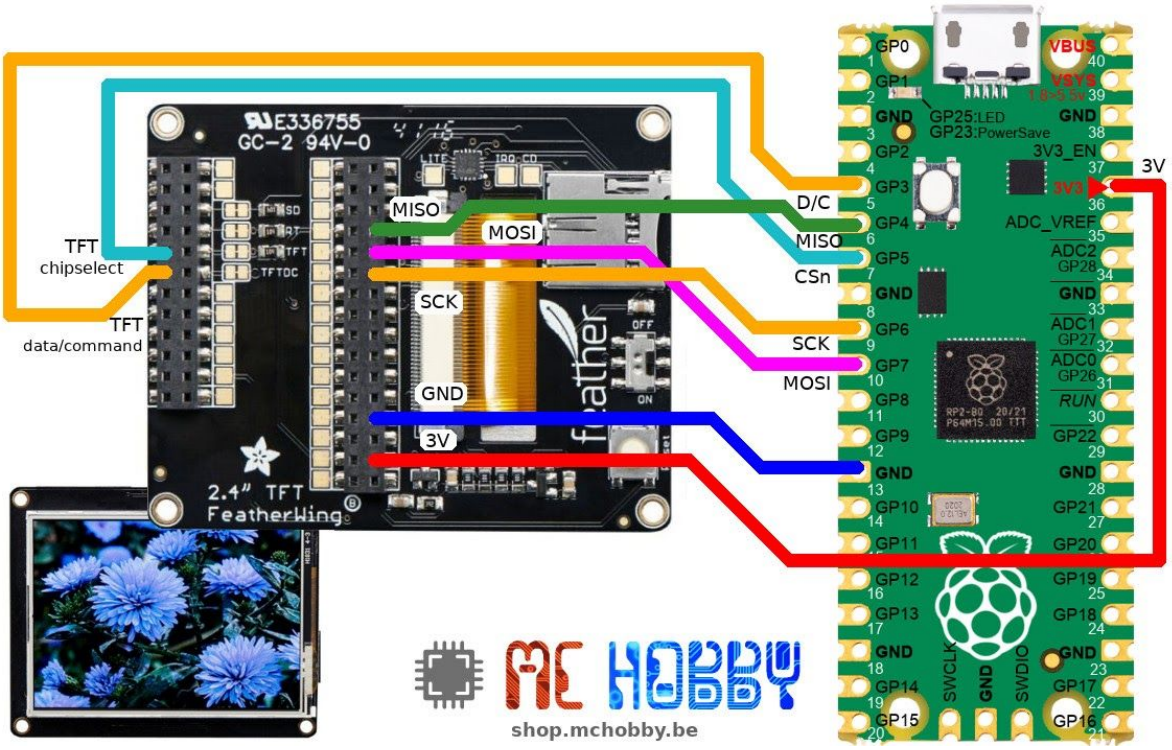


Figure 10: raccorder l’afficheur sur le Pico

Liste des fichiers

Les fichiers nécessaires sont disponibles dans le répertoire [GitHub esp8266-upy/ili934x/lib/]
Url complète: [https://github.com/mchobby/esp8266-upy/tree/master/ili934x/lib]

- `ili934x.py` : pilote pour **ILI934x** avec API FrameBuffer mais écrivant directement dans la mémoire du contrôleur (ne nécessite donc pas de mémoire sur le Pico)
- `fdrawer.py` : class permettant de dessiner des Font (ex: veram_m15.bin) sur l’afficheur. Voir l’archive `dependencies.zip` .
- `veram_m15.bin` : fichier de Font Vera en taille 15. D’autres fichiers sont disponibles dans le projet FreeType-Generator. [https://github.com/mchobby/freetype-generator]

L’interface de programmation de la bibliothèque est très proche de la version Arduino. Il est donc relativement simple de passer de C à MicroPython puisque les méthodes sont identiques.

Voici un exemple affichant 100 rectangles au hasard sur l’afficheur avec des couleurs de bordure aléatoire.

```
from machine import SPI,Pin
from ili934x import *
import urandom

# Raspberry-Pi Pico
spi = SPI( 0 )
cs_pin = Pin(5) # GP5
dc_pin = Pin(3) # GP3
rst_pin = None

lcd = ILI9341( spi, cs=cs_pin, dc=dc_pin, rst=rst_pin,
               w=320, h=240, r=0 )
lcd.erase()

colors = [NAVY, DARKGREEN, DARKCYAN, MAROON, PURPLE, OLIVE,
          LIGHTGREY,  DARKGREY, BLUE, GREEN, CYAN, RED, MAGENTA,
          YELLOW, WHITE, ORANGE, GREENYELLOW ]
color_count = len( colors )

for i in range(100):
     lcd.rect( urandom.randint(0,lcd.width-1), # Coord X
               urandom.randint(0,lcd.height-1),# Coord Y
               urandom.randint(1,50),          # largeur
               urandom.randint(1,50),          # Hauteur
               colors[urandom.randint(0,color_count-1)] # Couleur
             )
```

Ce qui produit le résultat suivant sur l’afficheur

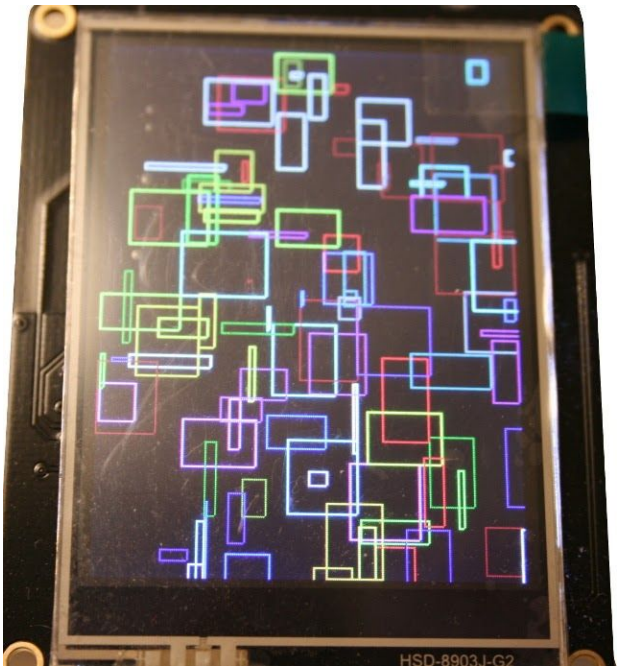


Figure 10a: résultat du script

Autres exemples

De nombreux autres exemples sont disponibles dans le dépôt.
https://github.com/mchobby/esp8266-upy/tree/master/ili934x/examples

Liste de matériel

Désignation	Quantité	Détails
Breadboard	1	modèle générique
Raspberry-Pi Pico	1	http://raspberrypi.org/
TFT 2.4” FeatherWing	1	https://www.adafruit.com/product/3315

Utilisation du Hat Sense

Le Hat Sense (pour Raspberry-Pi) peut être utilisé avec le Raspberry-Pi Pico sous MicroPython

Ce shield utilise l'interface I2C pour être contrôlé depuis le Pico. Si le bus I2C est en logique 3.3V, la carte est également alimentée en 5V via le VBUS. Il faut donc que le Pico soit branché sur un ordinateur (ou source d'alimentation USB) pour pouvoir fournir les 5 volts nécessaires au Sense-Hat.

Pour plus d'informations, voir le Github esp8266-upy/hat-sense .

Raccordement

Voici le schéma de raccordement.

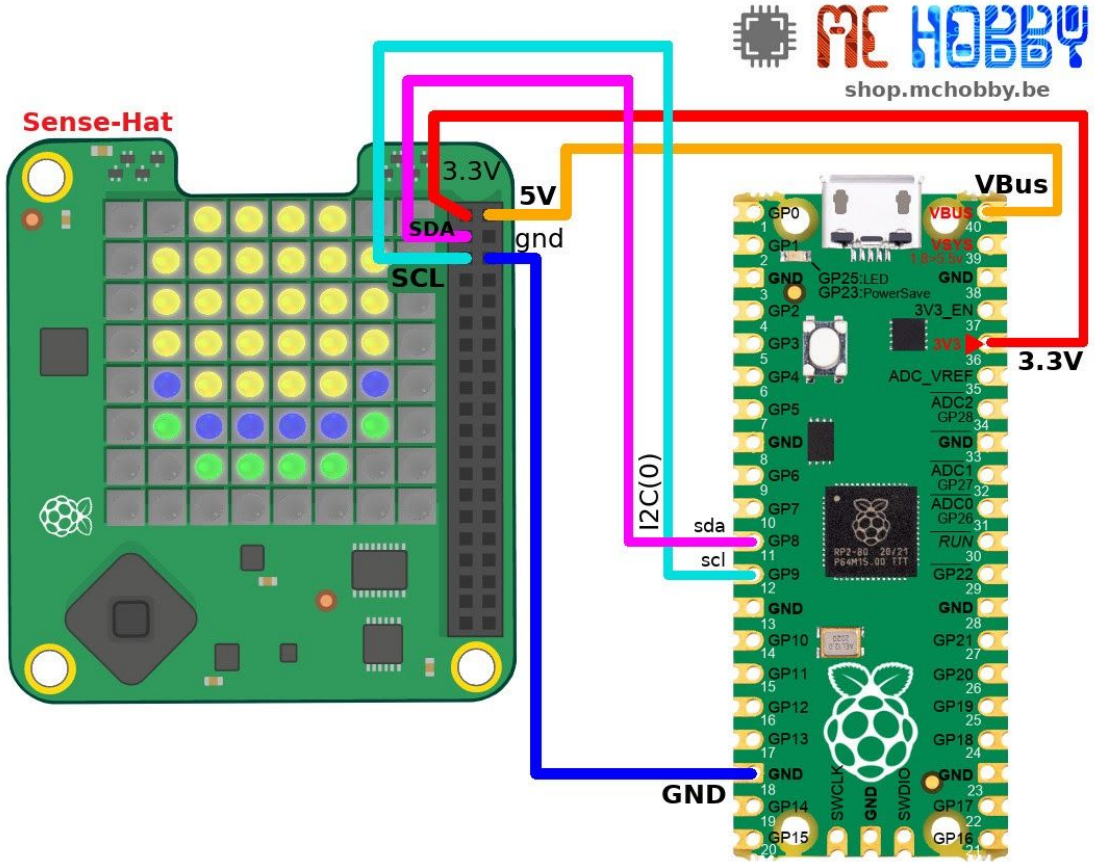


Figure 11: raccordement du sense-hat sur le Pico

Liste des fichiers

Les fichiers nécessaires sont disponibles dans le répertoire Github esp8266-upy/hat-sense
Url complète: https://github.com/mchobby/esp8266-upy/tree/master/hat-sense

- `sensehat.py` : pilote MicroPython pour le Hat
- `icons.py` : collection d'icônes encodés en format binaire.

L'exemple suivant affiche la collection d'icônes sur l'afficheur en générant aléatoirement une couleur d'affichage pour chaque icône.

```
from machine import I2C
from sensehat import SenseHat
from icons import *
import time
from random import randint

# Raspberry-Pi Pico, Sda=GP8, Scl=GP9
i2c = I2C( 0 )
hat = SenseHat( i2c )

# iteration à travers les icones
while True:
   for icon in all_icons:
      hat.clear()
      hat.icon( icon,
                color=hat.color( randint(0,255),
                                 randint(0,255),
                                 randint(0,255) ) )
      hat.update()
      time.sleep(1)
```

Autres exemples

De nombreux autres exemples testant les autres fonctionnalités du Hat sont disponibles dans le dépôt
https://github.com/mchobby/esp8266-upy/tree/master/hat-sense

Liste de matériel

Désignation	Quantité	Détails
Breadboard	1	modèle générique
Raspberry-Pi Pico	1	http://raspberrypi.org/
Sense Hat	1	

Utilisation d'un Hat PiFace

Le PiFace Digital (et PiFace Digital 2) ont connu leurs heures de gloire au lancement du Raspberry-Pi. Il s'agit d'une carte d'interface très répandue qui peut aussi être utilisée avec un Raspberry-Pi Pico.

Pour plus d'informations, voir le [Github esp8266-upy/hat-piface](#) .

Raccordement

Le PiFace utilise une interface SPI, voici le schéma de raccordement du Pico sur le PiFace.

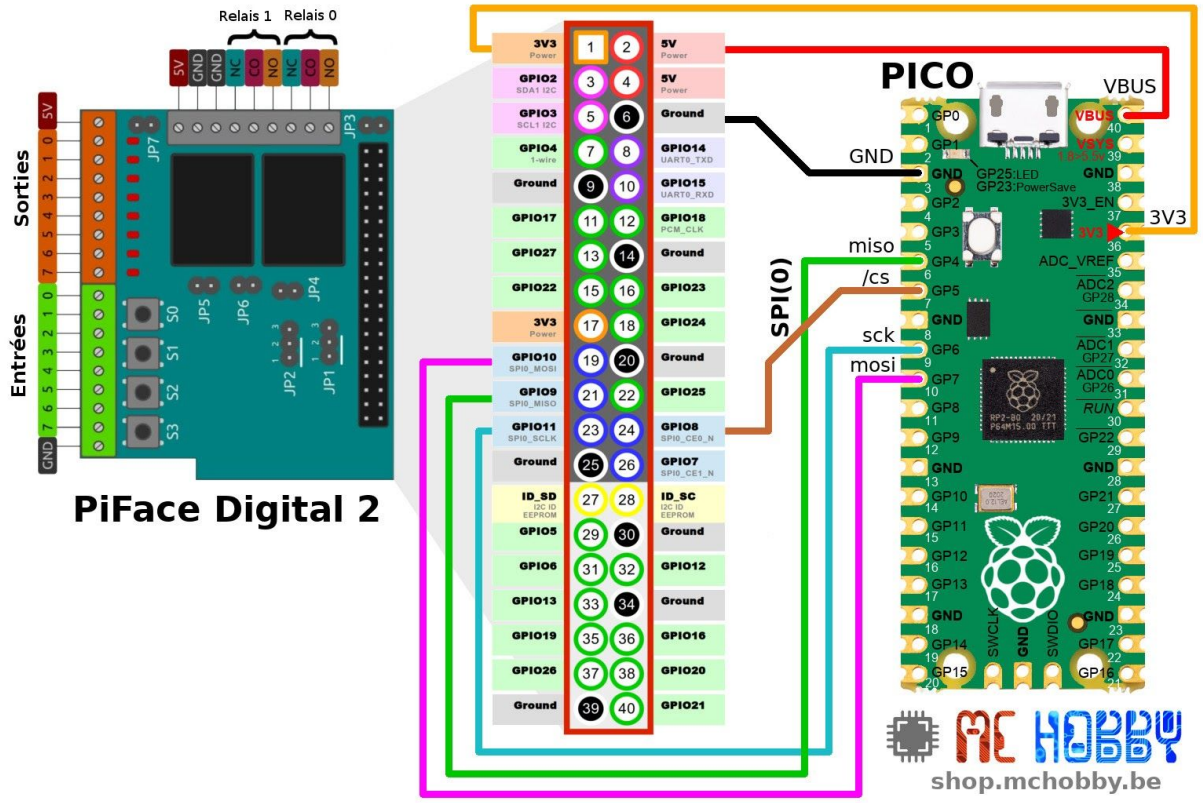


Figure 12: brancher le Pico sur un PiFace

Liste des fichiers

Les fichiers nécessaires sont disponibles dans le répertoire [GitHub esp8266-upy/hat-piface/lib/](#)
Url complète: [https://github.com/mchobby/esp8266-upy/tree/master/hat-piface/lib](#)

- `piface.py` : pilote permettant de lire l'état des entrées et modifier l'état des sorties.
- `mcp23Sxx.py` : le MCP23S17 est le composant central de la carte PiFace. Ce fichier, disponible dans l'archive **dependencies.zip** est utilisé par la bibliothèque piface.

L'exemple suivant active les sorties du piface une à une.

```
from machine import SPI, Pin
```

```
from piface import PiFace
import time

# Raspberry-pico
spi = SPI( 0, phase=0, polarity=0 )
cs = Pin( 5, Pin.OUT, value=True )

piface = PiFace( spi, cs, device_id=0x00 )

# changer l'état de toutes les sorties (chasse)
try:
     print( "Press CTRL+C to halt script" )
     while True:
          for i in range( 8 ): # 0..7
               piface.outputs[i] = True
               time.sleep_ms( 300 )
               piface.outputs[i] = False
except:
     piface.reset() # Reset de toutes les sorties
```

L'exemple ci-dessous lit l'état de toutes les entrées et affiche un message dans la session REPL si un bouton est pressé (ou entrée activée).

```
from machine import SPI, Pin
from piface import PiFace
import time

# Raspberry-pico
spi = SPI( 0, phase=0, polarity=0 )
cs = Pin( 5, Pin.OUT, value=True )

piface = PiFace( spi, cs, device_id=0x00 )

# lecture de toutes les entrées (une a la fois)
try:
     print( "Press CTRL+C to halt script" )
     while True:
          for i in range( 0, 8 ): # 0..7
               if piface.inputs[ i ]:
                    print( "Input %s is pressed" % i )
          time.sleep_ms( 300 )
except:
     print( "That s all folks" )
```

Plus d'exemples sur le GitHub Github esp8266-upy/hat-piface .

Si vous avez besoin de savoir comment **réaliser les branchements d'appareils** sur la carte PiFace, vous pouvez vous référer aux différentes pages de la documentation disponible sur ce wiki:
https://wiki.mchobby.be/index.php?title=PiFace2-Manuel