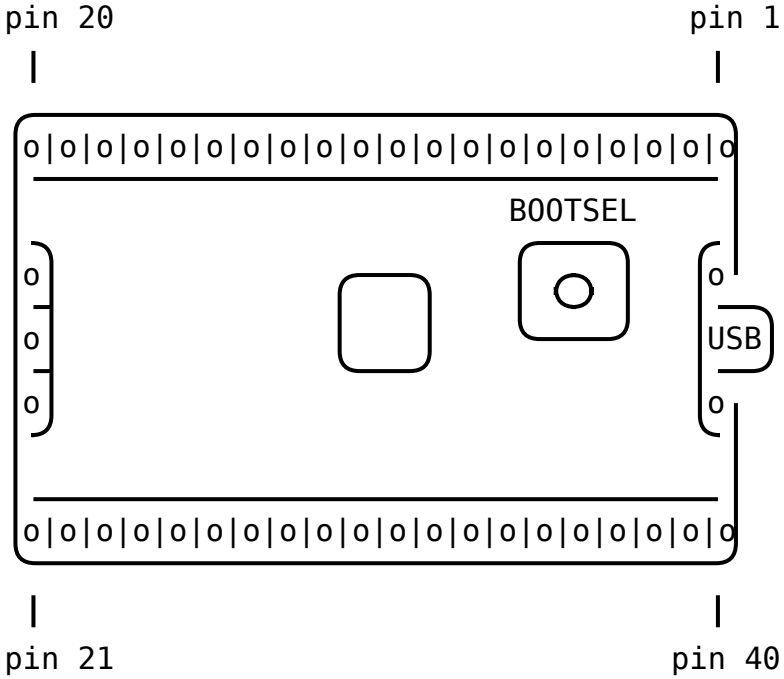
Installation de MicroPython sur Raspberry Pi PICO

Table of Contents

Raspberry Pi Pico dispose d'un mode BOOTSEL pour programmer le micrologiciel via le port USB. Maintenir le bouton BOOTSEL lors de la mise sous tension de la carte la mettra dans un mode spécial où elle apparaîtra comme un périphérique de stockage de masse USB.



Premier démarrage

Il faut s'assurer que Raspberry Pi Pico n'est branché sur aucune source d'alimentation : débrancher le câble micro USB s'il est branché, et déconnecter tous les autres fils susceptibles d'alimenter la carte, par ex. via la broche VSYS ou VBUS. Maintenant tenir le bouton BOOTSEL appuyer et brancher le câble micro USB (qui a l'autre extrémité branchée sur l'ordinateur).

Un lecteur appelé RPI-RP2 devrait apparaître. Faire glisser le fichier MicroPython **firmware.uf2** sur ce lecteur. Cette programme le firmware MicroPython sur la mémoire flash du Raspberry Pi Pico.

Cela devrait prendre quelques secondes pour programmer le fichier UF2 dans le flash. La carte redémarrera automatiquement une fois terminé, provoquant la disparition du lecteur RPI-RP2 et le démarrage en MicroPython.

Par défaut, MicroPython ne fait rien lors du premier démarrage, il attend que l'on tape d'autres

instructions.

Le pico ne dispose peut-être pas de bouton BOOTSEL. Si c'est le cas, il faut mettre à la terre la broche flash CS, en utilisant un cavalier, pour dire au RP2040 d'entrer dans le mode BOOTSEL au démarrage.

Compilation de MicroPython

Le binaire préconstruit qui peut être téléchargé à partir de la section **MicroPython** de la documentation devrait servir le plus cas d'utilisation, mais on peut créer son propre micrologiciel **MicroPython** à partir de la source si vous souhaitez personnaliser son bas niveau aspects.

Ces instructions pour obtenir et construire MicroPython supposent qu'on utilise Raspberry Pi OS sur un Raspberry Pi 4, ou une distribution Linux équivalente basée sur Debian fonctionnant sur une autre plate-forme.

Il faut commencer par créer un répertoire `/home/pi/picop`, pour conserver toutes les actions liées à pico, en utilisant ces instructions:

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

Ensuite, cloner le référentiel git micropython. Ces instructions récupéreront la dernière version du code source.

```
git clone -b master https://github.com/micropython/micropython.git
```

Une fois le téléchargement terminé, le code source de MicroPython doit se trouver dans un nouveau répertoire appelé `micropython`. Le référentiel MicroPython contient également des pointeurs (sous-modules) vers des versions spécifiques de bibliothèques dont il a besoin pour s'exécuter sur une carte particulière, comme le SDK dans le cas du RP2040. Il faut également les récupérer explicitement

```
cd micropython
git submodule update --init -- lib/pico-sdk lib/tinyusb
```

Les instructions suivantes supposent qu'on utilise un Raspberry Pi Pico. Certains détails peuvent différer si on construiset un firmware pour une autre carte basée sur RP2040. Le fournisseur de la carte doit détailler toutes les étapes supplémentaires nécessaires à la

construction firmware pour cette carte particulière. La version qu'on construit ici est assez générique, mais il pourrait y avoir quelques différences comme mettre le port série par défaut sur différentes broches, ou inclure des modules supplémentaires pour piloter cette carte Matériel.

Pour construire le port **MicroPython RP2040**, il faut installer des outils supplémentaires. Pour créer des projets, on a besoin de CMake, un outil multiplateforme utilisé pour créer le logiciel, et la chaîne d'outils intégrée GNU pour Arm, qui transforme le code source C de **MicroPython** dans en un programme binaire que les processeurs du RP2040 peuvent comprendre. **build-essential** est un ensemble d'outils dont on a besoin pour créer du code natif sur la machine hôte - cela est nécessaire pour certains outils internes de MicroPython et du SDK. On peut installer tous ces éléments via apt à partir de la ligne de commande. Tout ce qui est déjà installé sera ignoré par apt.

```
sudo apt update
sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-none-eabi build-
essential
```

Il faut d'abord créer un outil spécial pour les builds **MicroPython**, qui est livré avec le code source

```
make -C mpy-cross
```

On peut maintenant créer le port dont on a besoin pour RP2040, c'est-à-dire la version de MicroPython qui prend spécifiquement en charge le processeur.

```
cd ports/rp2
make
```

Si tout s'est bien passé, il y aura un nouveau répertoire appelé build `/ports/rp2/build` relatif au répertoire micropython, qui contient les nouveaux binaires du firmware. Les plus importants sont :

- **firmware.uf2** Un fichier binaire UF2 qui peut être glissé sur le lecteur RPI-RP2 qui apparaît une fois que votre Raspberry Pi Pico est en mode BOOTSEL. Le binaire du firmware que vous pouvez télécharger depuis la page de documentation est un fichier UF2, car ce sont les plus faciles à installer.
- **firmware.elf** Un autre type de fichier binaire, qui peut être chargé par un débogueur (tel que gdb avec openocd) sur Port de débogage SWD du RP2040. Ceci est utile pour déboguer soit un module C natif que vous avez ajouté à MicroPython, ou l'interpréteur central MicroPython lui-même. Le contenu binaire réel est le même que **firmware.uf2**

Comme expliqué ci-dessus, les commandes pour compiler Raspberry Pi Pico sont :

```
mkdir pico
cd pico
git clone -b master https://github.com/micropython/micropython.git
cd micropython
git submodule update --init -- lib/pico-sdk lib/tinyusb
```

```
sudo apt update
sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-none-eabi build-
essential
make -C mpy-cross
cd ports/rp2
make
```

Une fois la construction terminée, le firmware construit `firmware.uf2` sera disponible dans le répertoire `build-PICO`.

Construction de picotool

On aura également besoins de **picotool**, pour l'installer on il faut:

- définir **PICO_SDK_PATH** dans l'environnement ou le transmettre à cmake.
- installer également **libusb-1.0.**

```
mkdir pico
cd pico
git clone -b master https://github.com/raspberrypi/pico-sdk.git
git clone -b master https://github.com/raspberrypi/picotool.git
cd picotool
sudo apt-get install libusb-1.0-0-dev
mkdir build
cd build
export PICO_SDK_PATH=~/pico/pico-sdk
cmake ../
make
```

On peut rencontré des erreurs en essayant de construire **picotool** sur Linux avec **cmake**. Il semble qu'il manque une destination dans la commande d'installation. Pour résoudre ce bogue ajouter **RUNTIME DESTINATION** dans CMakeFiles.txt:

```
install(TARGETS picotool RUNTIME DESTINATION bin)
install(TARGETS picotool RUNTIME DESTINATION bin)
```

On peut maintenant utiliser **picotool** pour vérifier les modules inclus en tant que modules intégrés dans le firmware:

```
./picotool info /home/rafael/builds/pico/micropython/ports/rp2/build-
PICO/firmware.uf2
File /home/rafael/builds/pico/micropython/ports/rp2/build-PICO/firmware.uf2:

Program Information
 name:           MicroPython
```

```
 version:         v1.18-128-g2ea21abae
 features:        USB REPL
                  thread support
 frozen modules:  rp2, _boot, ds18x20, onewire, dht, uasyncio,
uasyncio/core,
                  uasyncio/event, uasyncio/funcs, uasyncio/lock,
                  uasyncio/stream, neopixel
```

Ajouts de modules

Cette configuration par défaut manque de certains modules importants. MicroPython permet d'ajouter d'autres modules en éditant le fichier manifeste :

Le contenu original du fichier vi `rp2/boards/manifest.py` est :

```
freeze("$(PORT_DIR)/modules")
freeze("$(MPY_DIR)/drivers/onewire")
freeze("$(MPY_DIR)/drivers/dht", "dht.py")
include("$(MPY_DIR)/extmod/uasyncio/manifest.py")
include("$(MPY_DIR)/drivers/neopixel/manifest.py"
```

Il faut le modifier et ajouter quelques lignes, donc `manifest.py` aura ce contenu :

```
freeze("$(PORT_DIR)/modules")
freeze("$(MPY_DIR)/drivers/onewire")
freeze("$(MPY_DIR)/drivers/dht", "dht.py")
include("$(MPY_DIR)/extmod/uasyncio/manifest.py")
include("$(MPY_DIR)/drivers/neopixel/manifest.py")

freeze("$(MPY_DIR)/tools", ("upip.py", "upip_utarfile.py"))
freeze("$(MPY_DIR)/drivers/display", "ssd1306.py")
include("$(MPY_DIR)/extmod/webrepl/manifest.py")

# Bibliothèques de micropython-lib, à inclure uniquement si le répertoire de
la bibliothèque existe
if os.path.isdir(convert_path("$(MPY_LIB_DIR)")):
    # file utilities
    freeze("$(MPY_LIB_DIR)/micropython/upysh", "upysh.py")

    # requests
    freeze("$(MPY_LIB_DIR)/python-ecosys/urequests", "urequests.py")
    freeze("$(MPY_LIB_DIR)/micropython/urllib.urequest",
"urllib/urequest.py")

    # umqtt
    freeze("$(MPY_LIB_DIR)/micropython/umqtt.simple", "umqtt/simple.py")
    freeze("$(MPY_LIB_DIR)/micropython/umqtt.robust", "umqtt/robust.py")
```

Certaines de ces bibliothèques se trouvent sur un autre référentiel github. Il faut donc clôner le dépôt micropython-lib pour utiliser **umqtt**, **urlib** et **requests**:

```
git clone https://github.com/micropython/micropython-lib
```

On peut maintenant reconstruire le firmware :

```
make -C mpy-cross
cd ports/rp2
make
```