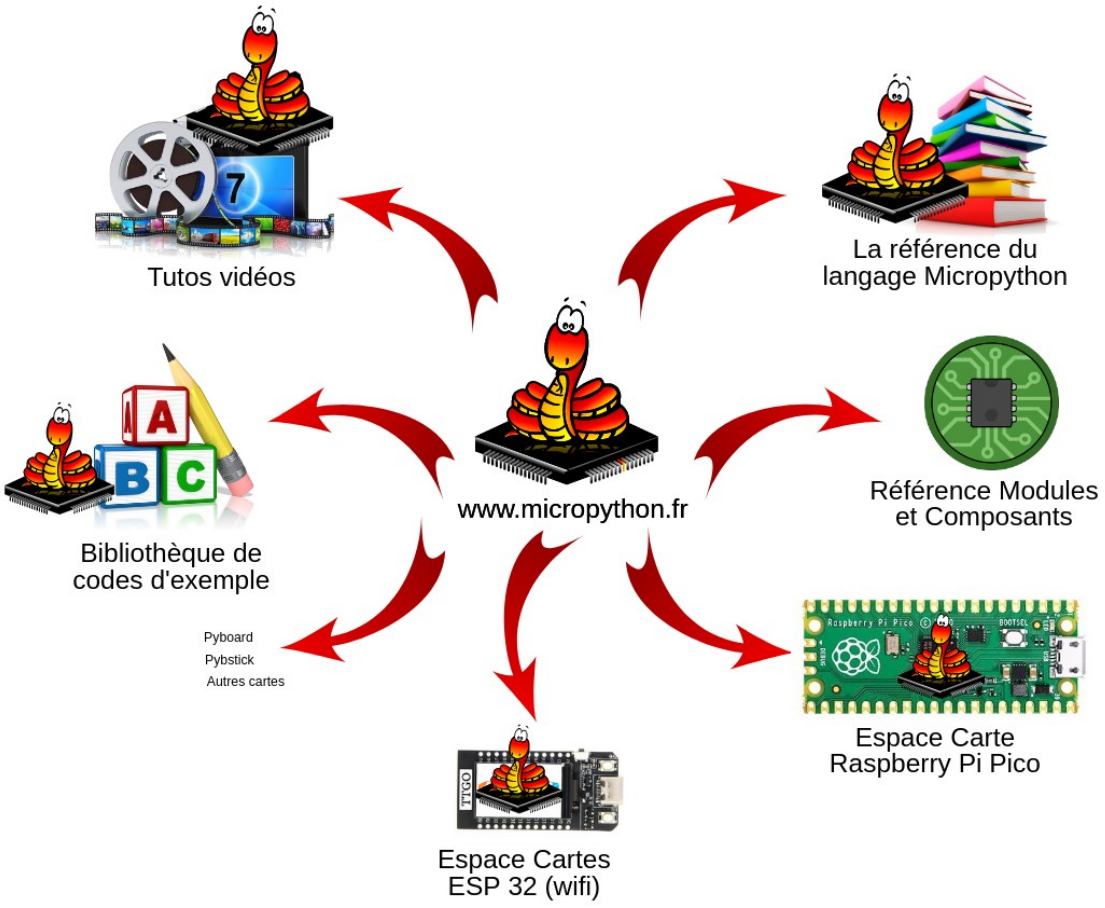
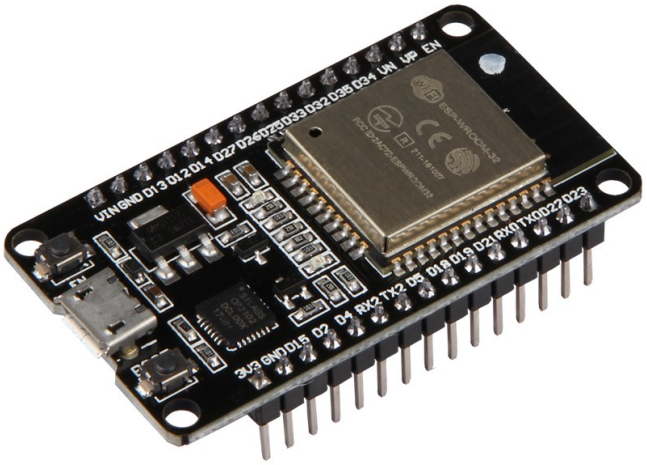
Micropython.fr



Micropython.fr | Port ESP 32



Bienvenue !

Bienvenue sur l'espace du site www.micropython.frdédiés aux cartes ESP 32 programmées en MicroPython.

Les cartes ESP 32 sont particulièrement adaptées à l'utilisation de MicroPython :

- le micro-contrôleur des cartes ESP est propulsé à 240 Mhz
- le port dédié à l'ESP 32 est bien développé et intègre en natif des classes très intéressantes telles que le support 1-wire ou Neopixel
- la RAM de ces cartes est parmi les plus élevées (110K de Free RAM sur ESP32 WROOM et 4Mo sur ESP32 WROVER)
- et surtout les cartes ESP 32 disposent en natif du wifi
- plusieurs cartes existent même avec écran TFT intégré
- les prix sont très intéressants
- c'est la catégorie de carte la plus développée et très utilisée par la communauté MicroPython.

Bref, quasiment incontournable, surtout si on veut faire des objets connectés.

Next

News autour des ESP

- Une nouvelle carte à ESP 32 + e-Ink display par LilyGo : https://www.cnx-software.com/2021/06/23/lilygo-mini-e-paper-core-combines-esp32-with-1-02-inch-epaper-display-in-3d-printed-enclosure/

- Une série de carte avec écran eInk : Inkplate 6Plus model, TTGO 5, le M5 Stack en version e-Ink , la watchy et un dernier en 4.2", le paperd.ink 4.2. La tendance aux cartes intégrées µc+écran est bel et bien là. Le tout programmable en micropython (ESP 32).

- La carte Arduino Nano RP2040 Connect, la carte idéale ??!! J'ai lu çà dans une newsletter promotionnelle il y a peu... mais c'est un peu du grand n'importe quoi : la carte est vendue 29€... et c'est en fait un RP2040 couplée à un ESP32 pour le wifi ! Et il n'y a pas d'écran TFT ou OLED sur la carte. La carte idéale ? C'est plutôt le TTGO Display : un ESP 32, un écran TFT miniature 240x235, GPIO, au format d'une carte nano. Et le prix est aussi idéal à 10€ ! Ici par exemple : https://opencircuit.fr/Product/TTGO-T-Display-V1.1-ESP32-avec-%C3%A9cran-TFT-1.14 Conseil d'ami : pour 30€, achetez vous plutôt 3 TTGO display qu'un Nano RP2040 connect... voir même un core M5Stack pour quelques euros de plus : https://shop.mchobby.be/fr/m5stack-esp/1896-m5stack-kit-esp32-basic-iot-dev-3232100018969-m5stack.html avec boitier, BP en façade, lecteur de carte SD, etc.

- une nouvelle carte avec écran elink 1024x758 propulsé par un ESP 32 : https://www.cnx-software.com/2021/06/02/inkplate-6plus-esp32-epaper-display-high-resolution-touchscreen-frontlight/ La tendance des cartes avec écran + ESP 32 est donc clairement là, entre les TTGO OLED, les M5Stack, et donc ce type d'écran.

- M5Stacks devient une solution de logique de contrôle à part entière, et voici un premier exemple avec le bras robotisé Cobot : https://www.cnx-software.com/2021/05/30/mycobot-robotic-arm-is-offered-with-raspberry-pi-4-or-m5stack-esp32-modules/ C'est beau, c'est propre !

- Une version originale de carte ESP32 : https://www.cnx-software.com/2021/05/28/t-nixie-tube-esp32nixie-tube-lookalike/

- Arduino sort (enfin....) sa carte à base de RP2040, avec wifi + Bluetooth (basé sur un ESP 32 en fait intégré à la carte). Donc le coup de génie d'Arduino c'est de coupler un RP2040 (125 Mhz) à un ESP 32 (240Mhz) ! Et de vendre le tout à 25€ ! Je rappelle qu'un Pi Pico, c'est seulement 4.5€, qu'un ESP32 c'est moins de 10€... qu'en ESP avec camera c'est 12€... En fait pour 25€, on peut sans problème avoir un ESP 32 avec chargeur Lipo intégré (made in UE) + ecran OLED 128x64 + pack LiOn 1400mAh... Je dis çà, je dis rien... Pour ceux que çà intéresserait malgré tout, c'est là : https://www.cnx-software.com/2021/05/17/arduino-nano-rp2040-connect-wifi-bluetooth-board/

- M5Stack intégrant 1 ESP 32 et donc wifi+bluetooth, 1 afficheur TFT 320x240, bus i2C, lecteur carte SD, un HP, connectique lipo, le tout dans un boitier très propre avec 3 BP en façade, connectique en latéral, c'est seulement 36€... et çà tourne sous Micro Python. On pourrait avoir un peu mieux en prix en pièces détachées, mais si on considère le boitier et l'équipement intégré, on est loin de quelque chose d'excessif vu le temps gagné pour la mise en oeuvre : https://www.gotronic.fr/art-kit-de-developpement-m5-basic-k001-27881.htm ou https://shop.mchobby.be/fr/m5stack-esp/1896-m5stack-kit-esp32-basic-iot-dev-3232100018969-m5stack.html

- Un projet intéressant de montre openhardware à base d'ESP 32 : https://www.cnx-software.com/2021/04/07/lilygo-open-smartwatch-open-source-hardware-esp32-watch-by-pauls-3d-things/

- une carte que je trouve vraiment très cool : la carte ESP 32 au format UNO, la Wemos D1 32. 4Mo de Flash, 521K de RAM, l'implémentation de toutes les fonctionnalités de la carte Arduino UNO avec un brochage identique (à part A0 et A1 qui ne sont pas disponibles) mais GPIO en 3.3V par contre (ce qui ne pose pas de problème majeur en pratique, c'est comme un Raspberry Pi), et donc wifi en natif... et bien sûr Micropython tourne dessus sans problème ! Bref, la carte Arduino UNO sous Micropython ! Et tout çà pour seulement 10€ ! Petite page dédié ici : Mise en route de la carte ESP 32 UNO D1 32

- A un moment donné, çà devient du grand n'importe quoi : voilà maintenant une carte pour Pico qui permet d'ajouter... un ESP 32 pour avoir le support du wifi !! Euh du con... pourquoi t'utilises pas directement l'ESP 32 dans ce cas ?! Vu ici et l'article de blog est dans le même ton... : https://www.cnx-software.com/2021/05/06/pico-wireless-is-an-esp32-add-on-board-for-raspberry-pi-pico/ Ceci étant, çà révèle ce qui manque sur le Pi Pico : la connectivité wifi. L'ESP 32 apparaît donc le bon choix dès lors qu'on a besoin

de connectivité, et notamment les cartes au format ESP 32 UNO pour ceux qui aiment le format Arduino (brochage homogène, etc.)

- Une série de tutos dédiés à MicroPython sur ESP32 (tous les tutos "non wifi" sont en fait compatibles avec n'importe quelle carte) : https://www.dfrobot.com/blog-1569.html
- Une autre carte à ESP 32 - à priori pas trop intéressante car chère et "hardware spécifique", mais bon, je signale : https://www.cnx-software.com/2021/05/03/atmegazero-esp32-s2-board-supports-oled-displays
- Les ESP32 RiscV débarquent : https://www.cnx-software.com/2021/04/24/ai-thinker-esp32-c3-modules-compatible-esp8266-esp32/
- Une version RISC-V de l'ESP 32 : https://www.cnx-software.com/2021/03/26/esp32-c3-devkitm-1-risc-v-wifi-ble-board-to-launch-for-8-modules-for-1-8/
- Un module ESP 32 avec écran paper Ink (j'ai tendance à préférer le modulaire indépendant, mais çà existe en intégré) : https://fr.aliexpress.com/item/1005002006058892.html
- Une montre conectée à ESP 32, la Lily Go 2020 avec accélérometre, écran tactile, etc. : http://www.lilygo.cn/prod_view.aspx?TypeId=50053&Id=1290
- Ce n'est pas du micropython mais c'est quand même bon à savoir : portage du firmware grbl sur ESP 32 existe : https://github.com/bdring/Grbl_Esp32

Previous Next

Introduction aux cartes ESP

Le port ESP (8266 et 32) est un des ports les plus développés pour Micropython (plus grand nombre de librairies, etc.)

La première chose à dire concernant les ESP 32 et 8266, c'est que c'est un "vrai bordel" : il y a plein de versions, de copies, etc... et il y a un peu (beaucoup ?) de quoi s'y perdre avec ces modèles. Ceci étant, ce sont des cartes qui ont un très bon support avec MicroPython, j'ai envie de dire que n'importe quel ESP 32 fera le job sur un projet. Même les modèles les plus "light" ou les plus "anciens" ont 2MB de Flash.

Les versions

Petite vue d'ensemble de cartes ESP :

Note

On appelle :

- **module** : le composant lui-même sous forme d'un micro-pcb : https://www.espressif.com/en/products/modules
- **kit-dev** : des cartes avec connecteurs pour exposer les broches du processeur. Plusieurs versions existent aux noms différents : https://www.espressif.com/en/products/devkits

Lequel choisir ?

Le point critique le plus important avec une carte Micropython est la quantité de RAM disponible. Premièrement, entre le 8266 et le ESP32, y'a pas photo : ESP32 (110Ko free RAM contre 40Ko Free RAM sur ESP 8266) !Ensuite, parmi les ESP32, la carte à privilégier est la carte ESP 32 WROVER qui dispose de plusieurs Mo de SPIRAM. Donc, si on n'a pas raison particulière de prendre un WROOM,

un WROVER sera le mieux. La seule différence entre les 2 modèles sera quelques broches GPIO de moins, en raison de broches utilisées pour la communication avec la SPIRAM.

ESP 8266

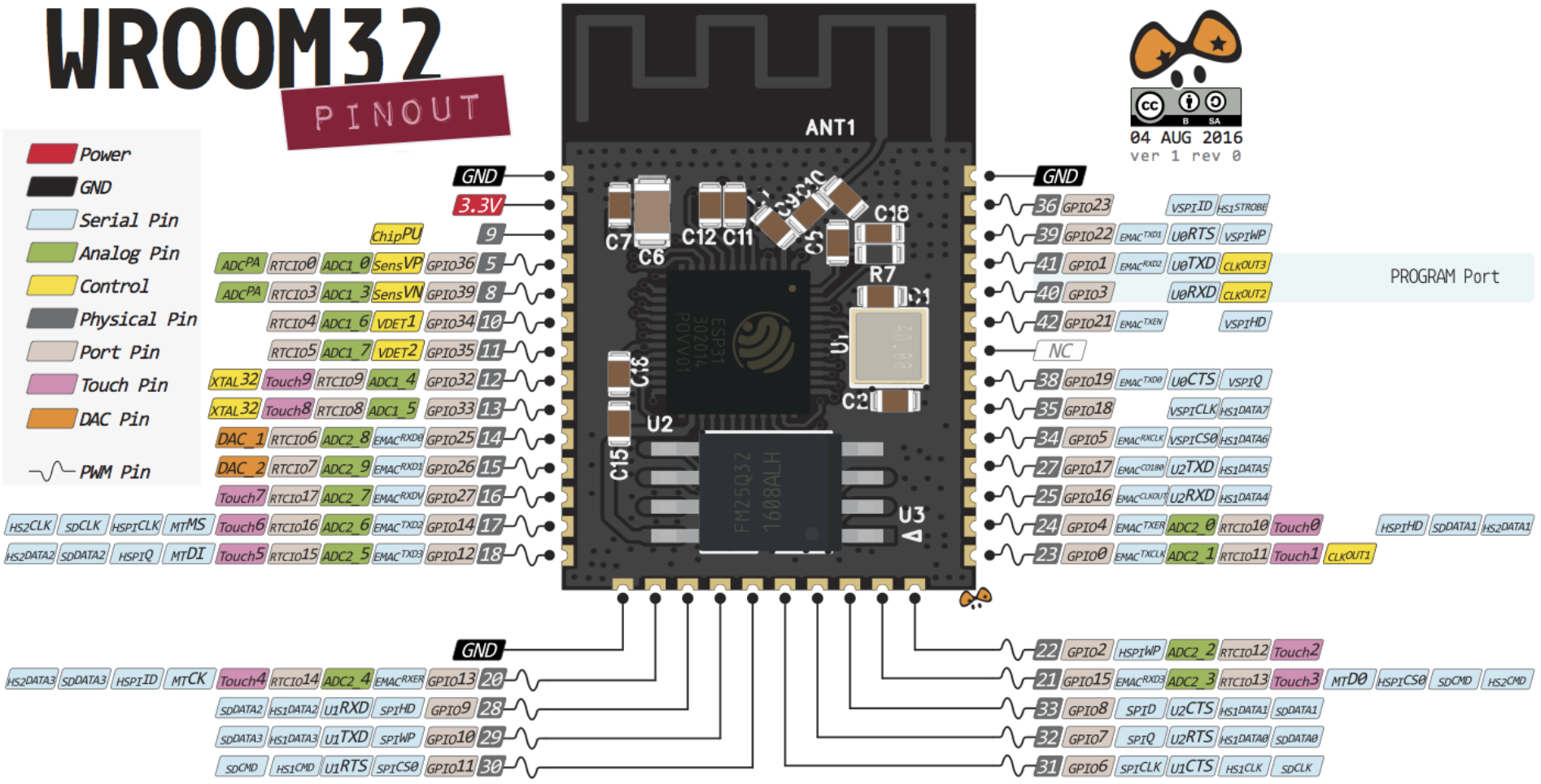
Modèle le plus ancien mais qui est très peu cher et très répandu. Ne dispose que d'une voie analogique et d'une RAM limitée ce qui est vite un problème avec MicroPython, notamment si on veut faire des micro-serveurs embarqués.

ESP 32

La gamme est plutôt large... et décrite ici : https://en.wikipedia.org/wiki/ESP32

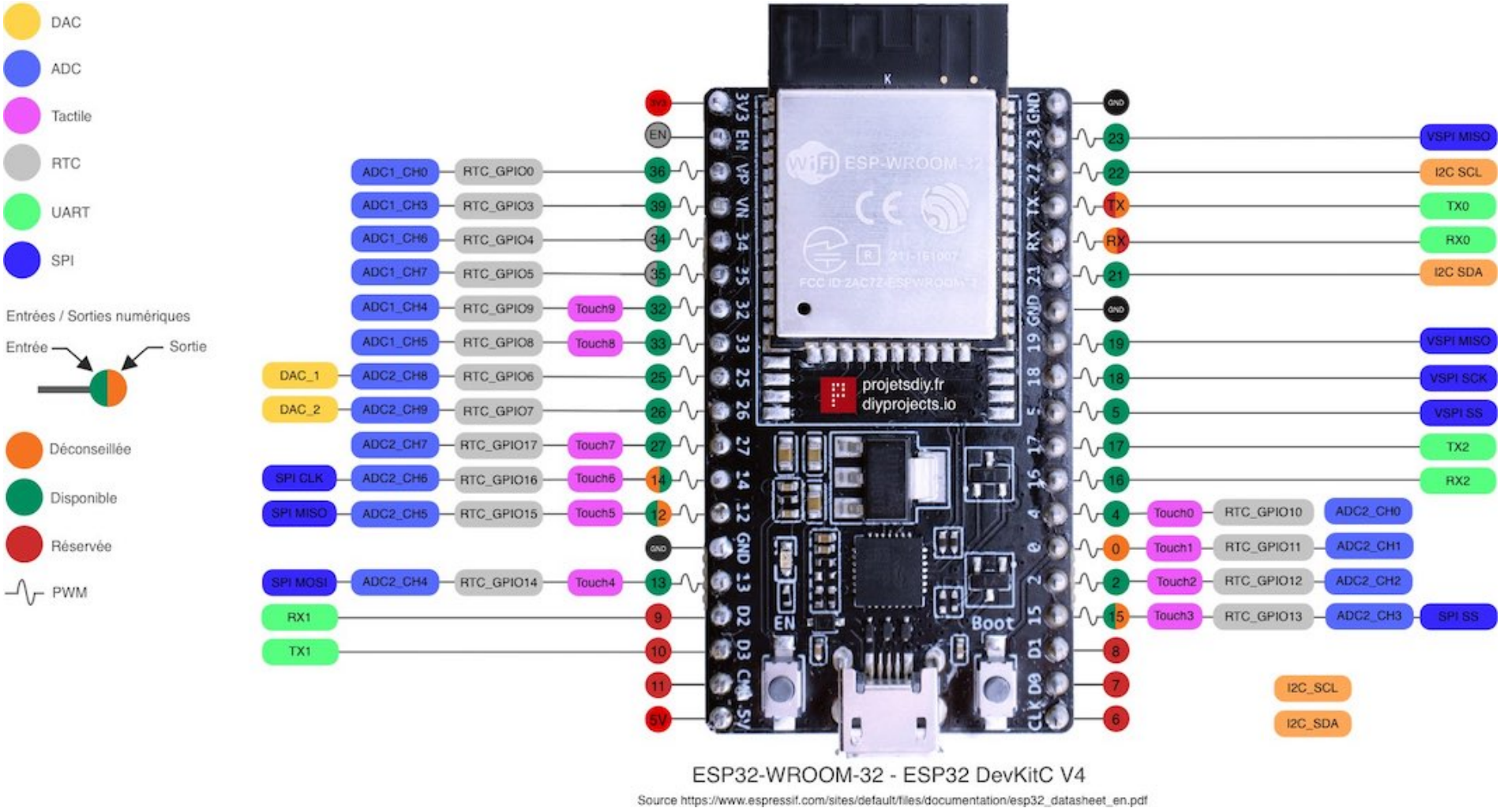
Voir également : https://www.espressif.com/en/products/socs/esp32/resources

Voici le brochage de l'ESP 32 WROOM "as is" au format module :



https://randomnerdtutorials.com/esp32-pinout-reference-gpios/

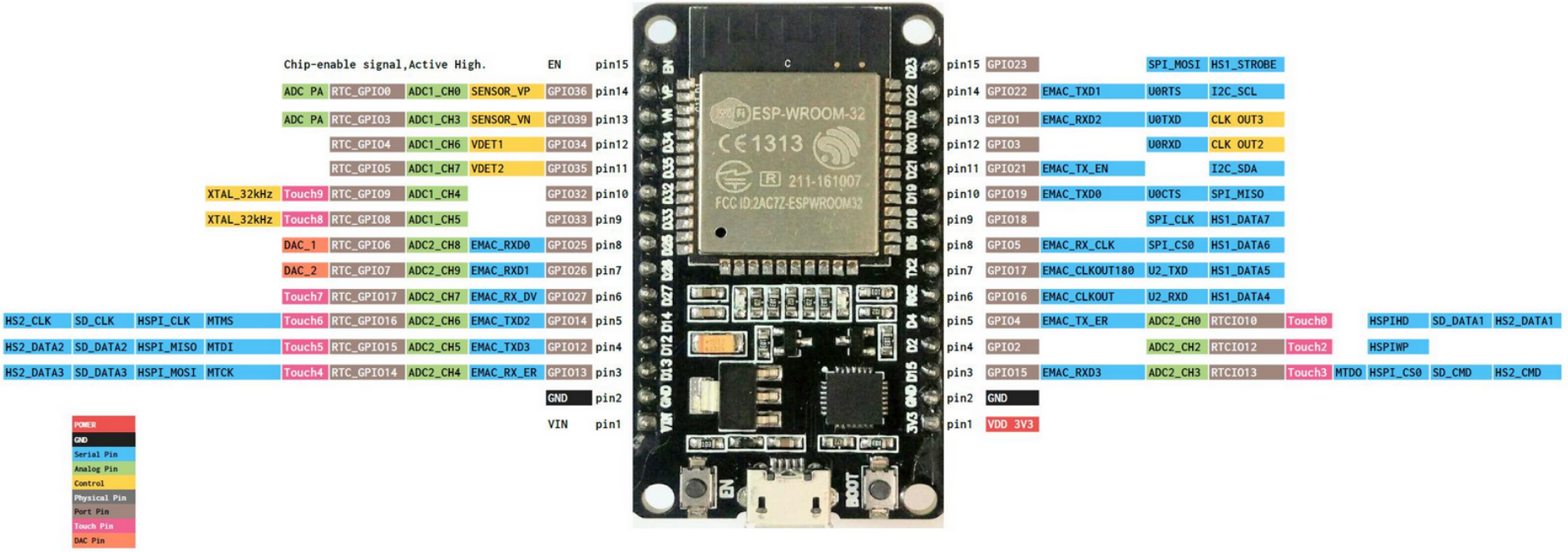
Voici le brochage d'un ESP 32 VROOM sur une carte de développement type (kit-dev) :



ESP32-WROOM-32 - ESP32 DevKitC V4

Source https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Ou bien :



Info

Il y a quelques variations selon le modèle, mais globalement on retrouve le même brochage. Par exemple ici, entre les 2 versions, il y a des différences sur les broches réservées qui ne sont pas forcément exposées.

Dispose de :

- une 15aine de GPIO
- d'autant de broches ADC en 12 bits (18 voies)
- 2 x DAC
- I2C, SPI

Page très bien faite ici : https://projetsdiy.fr/esp32-gpio-broches-fonctions-io-pwm-rtc-i2c-spi-adc-dac/

Comparaison ESP 8266 / ESP 32

Pour la faire simple, ESP 8266 c'est wifi mais pas de bluetoouth, une fréquence processeur moindre (84Mhz), une seule broche ADC en 10 bits. Concrètement, intéressant sur projets simples. On a I2C, SPI, etc.

ESP 32 : c'est wifi, bluetooth, frequence processeur à 240Mhz, plein de broches ADC (8) en 12 bits, DAC, capteur température et effet hall onboard, etc... bref, la totale. Un peu plus cher, mais bien plus polyvalent. On a SPI, I2C, etc en double.

✎ En pratique : ESP8266 sur les projets où on n'a pas de gros besoins de mesure analogique, peu de besoins évolutifs. ESP32 pour tous les autres cas. Et si la différence de prix n'est pas significative, prendre ESP 32

Voici un comparatif entre les 2 modèles :

	ESP8266	ESP32
MCU	Xtensa Single-core 32-bit L106	Xtensa Dual-Core 32-bit LX6 with 600 DMIPS
802.11 b/g/n Wi-Fi	HT20	HT40
Bluetooth	**No**	Bluetooth 4.2 and BLE
Typical Frequency	80 MHz	160 MHz
SRAM	**No**	**Yes**
Flash	**No**	**Yes**
GPIO	17	36
Hardware /Software PWM	None / 8 channels	None / 16 channels
SPI/I2C/I2S/UART	2/1/2/2	4/2/2/2
ADC	10-bit	12-bit
CAN	**No**	**Yes**
Ethernet MAC Interface	**No**	**Yes**
Touch Sensor	**No**	**Yes**
Temperature Sensor	**No**	**Yes**
Hall effect sensor	**No**	**Yes**
Working Temperature	-40°C to 125°C	-40°C to 125°C

source : https://community.wia.io/d/53-esp8266-vs-esp32-what-s-the-difference

Autres liens utiles :

- https://www.electronicshub.org/esp32-vs-esp8266/
- https://circuits4you.com/2019/03/02/esp32-vs-esp8266/

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

> **Note**
>
> Pour ce qui concerne le WROOVER avec de la SPiRAM, prendre la version avec SPIRAM.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

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

- on flashe micropython :

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20210416-unstable-v1.14-170-ga9bbf7
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Liens utiles

- Voir ici : https://projetsdiy.fr/reinstaller-firmware-micropython-esp8266-esp32/
- Flasher MicroPython sur une carte NodeMCU : https://icircuit.net/nodemcu-getting-started-micropython/2406 | https://icircuit.net/esp32-micropython-getting-started/1999

Discussion technique

- ~~Ici une version de Micropython pour ESP 32 qui est censée améliorer les possibilités : https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki~~

Connexion en Wifi

Une des caractéristiques intéressante des modules ESP, c'est de disposer de la connexion Wifi.

On a ici besoin d'un réseau wifi, que l'on peut utiliser sans sécurisation à des fins de tests, la sécurisation pouvant être faîte plus tard (il n'y a aucun danger particulier si de plus le réseau n'est pas connecté à internet). Dans l'exemple ici, on a un réseau wifi local `dlink` public.

On commence par importer le module `network` :

```
import network
```

Ensuite, on fait :

```
>>> import network
>>> station=network.WLAN(network.STA_IF)
>>> station.active(True)
True
>>> station.connect("dlink","")
>>> station.isconnected()
True
>>> station.ifconfig()
('192.168.0.101', '255.255.255.0', '192.168.0.1', '0.0.0.0')
```

Bah bah... tout ok en moins de 2 !

Lien : https://projetsdiy.fr/tutoriel-micropython-gerer-connexion-wifi/

Test de la réponse

Ensuite, il faut créer un socket qui va permettre de gérer la connexion (à faire une fois) :

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)
```

Ensuite on gère la réception d'une connexion et la réponse à envoyer (code à boucler) :

```
s.accept()
#(<socket>, ('192.168.0.100', 51836))
conn,addr=s.accept()
print(conn)
#<socket>
print(addr)
#('192.168.0.100', 51844)
print(conn.recv(1024))
#b'GET / HTTP/1.1\r\nHost: 192.168.0.101\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686; rv:78.0) Gecko/20100101 Firefox/78.
conn.settimeout(None)
response="Hello !"
conn.send('HTTP/1.1 200 OK\n')
conn.send('Content-Type: text/html\n')
conn.send('Connection: close\n\n')
conn.sendall(response)
conn.close()
```

Bref, on peut gérer assez simplement un petit serveur local et récupérer de la donnée sans fil. Vu que l'on est en wifi, un simple routeur de récup' fera l'affaire et on pourra avoir une portée non négligeable sans trop de problèmes.

Fonctions de base

GPIO

Il existe une LED onboard sur la GPIO 2 :

```
>>> from machine import Pin
>>> led=Pin(2,Pin.OUT)
>>> led.on()
>>> led.off()
```

Sur une brohce externe, çà donne :

```
from machine import Timer, Pin

led=Pin(15, Pin.OUT)

led.on()
led.offf()
```

PWM

```
from machine import PWM

pwm=PWM(Pin(15))
pwm.freq(1000)
pwm.duty(1000) # 0-1023
pwm.duty(100)
```

ADC

https://docs.micropython.org/en/latest/esp32/quickref.html#adc-analog-to-digital-conversion

La mesure analogique est disponible sur les broches 32 à 39 avec MicroPython (noté que l'ESP fournit bien plus de broches ADC que çà...), mais 8 broches ADC, c'est déjà pas mal. Les broches sont en 12 bits (par défaut) configurables. La mesure est renvoyée sur une échelle 0-4095 (=12 bits).

Point particulier : les broches disposent d'un atténuateur / amplificateur de niveau de tension. Par défaut, à 0dB, la mesure se fait sur 0-1V (et pas 3.3V !!). Pour avoir l'échelle en 3.3V, il faut utiliser l'amplification 11dB.

Le constructeur ADC se base sur une broche Pin(xx) où xx est le numéro de la broche GPxx utilisée.

```
>>> from machine import ADC, Pin
>>> sensor=ADC(Pin(32))
>>> sensor.read() # echelle 0-1v par defaut
0
>>> sensor.read()
3919
>>> sensor.atten(ADC.ATTN_11DB) # pour echelle 0-3.3V
>>> sensor.read()
442
>>> sensor.read()
4095
```

Note

L'ESP 32 disposent de 2 modules ESP 32, ADC1 et ADC2. Si le wifi est utilisé, alors ADC2 n'est pas utilisable. C'est pourquoi Micropython n'implémente pas ADC2 par défaut. On a donc seulement 4 broches analogiques. Si on a besoin de plus, il est facile d'ajouter un multiplexeur analogique ou bien un module analogique i2C. La limitation est la même en C / Arduino car liée à l'ESP 32 lui-même.

Voir : https://github.com/micropython/micropython/issues/6219

Timer

Timer 0 à 3 disponibles : le constructeur nécessite de fournir un index de timer.

```
from machine import Timer, Pin

led=Pin(15, Pin.OUT)

timer=Timer(0)
```

```
def loop(timer):
    if led.value():led.off()
    else:led.on()

timer.init(freq=10, callback=loop)
```

SPI

ESP 8266

Attention, la configuration est un peu spécifique : on ne précise pas les broches et l'indice doit être 1 (0 correspondant à la Flash)

https://docs.micropython.org/en/latest/esp8266/quickref.html?highlight=spi#hardware-spi-bus

```
>>> sp=SPI(1) # MISO=GPIO12 MOSI=GPIO13 et SCK=GPIO14 - SPI 0 = flash !
```

ESP 32

https://docs.micropython.org/en/latest/esp32/quickref.html#hardware-spi-bus

RTC

```
from machine import RTC
rtc.datetime()
rtc=RTC()
rtc.datetime()
(2000, 1, 1, 5, 1, 11, 49, 170430)
```

RMT

https://docs.micropython.org/en/latest/esp32/quickref.html#rmt

Fonctions spécifiques ESP 32

voir ici : https://docs.micropython.org/en/latest/library/esp32.html

Communication WebREPL

Une chose très intéressante avec les ESP 32 est la possibilité de communiquer en mode interpréteur via le réseau wifi sur lequel il se trouve.

Pour cela, il suffit d'activer le serveur REPL sur l'ESP 32 (intégré par défaut dans la version Micropython des ESP)

```
>>> import network
>>> wifi=network.WLAN(network.STA_IF)
>>> wifi.isconnected()
False
>>> wifi.active(True)
True
>>> wifi.connect("dlink","")
>>> wifi.isconnected()
True
>>> wifi.ifconfig()
('192.168.0.101', '255.255.255.0', '192.168.0.1', '0.0.0.0')
>>> import webrepl
>>> webrepl.start()
WebREPL daemon started on ws://192.168.0.101:8266
Started webrepl in normal mode
>>>
```

On peut mettre çà dans une fonction dans le boot.py aussi.

+/- Configuration

```
>>> import webrepl_setup
WebREPL daemon auto-start status: enabled

Would you like to (E)nable or (D)isable it running on boot?
(Empty line to quit)
> E
Would you like to change WebREPL password? (y/n) n
No further action required
>>> import webrepl
>>> webrepl.start()
Started webrepl in normal mode
```

Lien utile :

*https://www.techcoil.com/blog/how-to-setup-micropython-webrepl-on-your-esp32-development-board/

Le client webrepl

C'est une appli web qui utilise websocket :

Installer localement.

Lancer serveur avec n'importe quel serveur de fichier statique de son choix :

```
php -S 127.0.0.1:8000
```

Se connecter sur :

```
http://127.0.0.1:8000/webrepl.html
```

Le terminal est inactif dans mon cas bien que la connexion semble se faire correctement.

Projets à base d'ESP 32

- un serveur simple : https://randomnerdtutorials.com/micropython-esp32-esp8266-bme680-web-server/
- contrôler servo : https://icircuit.net/micropython-controlling-servo-esp32-nodemcu/2385

Sources intéressantes

- https://projetsdiy.fr/microcontroleurs-mcu/esp32-iot/page/2/ Les projets MicroPython et Arduino sont mélangés mais il y a quelques bonnes pages MicroPython.
- https://www.techcoil.com/blog/how-to-setup-micropython-on-your-esp32-development-board-to-run-python-applications/
- Une présentation très complète avec de bonnes explications est disponible ici (bilan des ESP en 2020) : https://projetsdiy.fr/quelle-carte-esp32-choisir-developper-projets-diy-objets-connectes/

Voir également les fonctions spécifiques à l'ESP 32 fournies par Micropython :
https://docs.micropython.org/en/latest/library/esp32.html

Previous Next

Critères de choix d'une carte ESP 32

Les cartes ESP 32 sont nombreuses... et il y a de quoi s'y perdre. Voici quelques critères de choix :

Le prix

C'est un critère important pour chacun d'entre nous : on veut bien payer ce qu'il faut... mais il ne faut pas non plus nous prendre pour des "billes"... ! Oui, le tout chinois a sa limite, et un approvisionnement auprès d'un fournisseur direct France ou Europe est préférable à n'importe quel fournisseur "market place".

Mais pour autant on ne va pas payer non plus 2 ou 3 fois le prix !

Pour fixer les idées, en 2021 et auprès d'un fournisseur France voire Europe :

- une carte "façon nano" (ou Dev-kit) ESP 32 de base s'achète aux alentours de 8 à 10 € TTC
- une carte ESP 32 avec afficheur OLED intégré (i2c voire SPI) sera disponible de 10 à 15€
- un module complet à base d'ESP 32 avec écran TFT, carte SD, etc sera vendu entre 35 et 40 euros TTC
- une montre à ESP 32 sera également vendue à ce prix là, entre 35 et 40€ voire 25€ pour les modèles OLED.

L'analyse du prix est à faire en tenant en compte l'équipement de la carte. Ainsi, si sur une carte à 8€, je n'ai aucun équipement particulier, il faut prendre en compte les éléments intégrés pour une carte un peu plus chère. Et si on en a besoin, çà vaut largement la peine de faire le saut.

Lipo or not Lipo

Un détail important est la présence d'un chargeur lipo intégré. Si le chargeur lipo est intégré, il est facile de connecter un lipo et de charger le lipo par simple connexion via un connecteur USB

Le type de port USB : micro-USB ou USB-C

Point non pas essentiel mais à prendre en compte : le port USB de la carte est-il de type micro-USB ou de type USB-C (ce qui devient de plus en plus fréquent).

Si le port est en USB-C, il faudra un câble USB-C, et/ou un adaptateur USB-C vers micro-USB (çà coûte que 1€ environ). Ceci permettra de pouvoir utiliser un bloc d'alimentation classique micro-USB avec la carte ESP notamment.

> **Note**
>
> Certaines cartes se démarquent de ce point de vue, notamment la Pybstick qui a un connecteur A pour engagement direct dans le port USB comme une clé USB en quelque sorte. Mais ce n'est pas de la carte ESP 32

Le modèle de l'ESP 32 : WROOM ou WROVER

Pour dire les choses simplement, on a 2 types d'ESP 32 :

- les ESP 32 WROOM sans SPIRAM intégrée
- les ESP 32 WROVER qui ont de la mémoire SPIRAM intégrée

Les ESP 32 WROOM sont les plus répandus et suffiront pour l'essentiel des projets, à savoir sans capture d'image, etc. Les WROVER seront à utiliser si on a besoin de faire de la capture d'image, etc. Mais c'est un usage probablement "limite" pour du microcontrôleur et dans ce cas la question d'utiliser un nano-pc se posera.

La mémoire FLASH et la RAM

Pour toute carte ESP 32, un point important est la quantité de mémoire FLASH et RAM disponible.

La configuration minimale usuelle, c'est 2Mo de FLASH et 512Ko de RAM. Mais si on a plus, c'est mieux, cependant non essentiel, surtout si le prix explose

L'équipement associé

Elément clé pour une carte ESP 32, c'est de bien analysé l'équipement associé déjà intégré dans la carte ou le module , notamment :

L'écran graphique

Des cartes avec écran OLED ou TFT sont désormais disponible à un coût très bas. Pour 10€ on a la carte avec l'écran OLED !

A savoir : les écrans à communication SPI (en général couleur) ont une réactivité plus grande que les écrans i2C... mais d'un autre côté, les écrans i2C sont monochromes et de ce point de vue nécessitent moins de vitesse pour une réactiivité identique.

On peut gérer avec un bon résultat un écran TFT couleur jusqu'à 320x240 en SPI avec un ESP 32.

Module carte SD

La présence d'un module carte SD est un plus indéniable, pour stocker des images, faire du datalogging. Si c'est présent, c'est très bien. Sinon, çà sera utilisable facilement sous forme d'un module "breakout" séparé, à communication SPI.

Module RTC (horloge temps réel)

La présence d'un module RTC avec sa pile intégré est un plus, notamment dans le cas de la montre (sic !). Mais on pourra suppléer cette absence par la mise à l'heure de la carte par le réseau d'une part ou bien par l'utilisation d'un module "breakout" i2c

La présence de bouton poussoir

Si on dispose de 1 ou plusieurs bouton poussoir usager, c'est un plus indéniable. 3 est l'idéal (+/-/ok)

La présence d'un bouton de reset

De la même façon, la présence d'un bouton de Reset est quasi-indispensable.

LEDs "onboard"

Intéressant pour des signalisations de "vie"... Soit que la LED soit sur la ligne du wifi, soit sur la ligne USB...

Fréquence du processeur

En général, c'est 240Mhz, mais on peut avoir des cartes à seulement 84Mhz. Vérifier.

La connectique physique de la carte

Regarder si la carte a déjà un bornier soudé ou pas sur les GPIO. Les cartes sans le bornier soudé peuvent être intéressante pour permettre de souder directement dessus un capteur ou autre.

Les caractéristiques de l'alimentation de la carte

Vérifier la présence du 3.3V et du 5V, les cartes étant des cartes 3.3V sur les GPIO. Le +5V peut être utile pour alimenter certains dispositifs (LCD i2C par exemple)

GPIO disponibles sur bornier

Analyser les GPIO de l'ESP 32 qui sont disponibles sur bornier. En pratique, il faut notamment disposer sur bornier de :

- l'i2C (broches 21 et 22)

- SPI
- UART
- ADC
- DAC
- voire IIS mais c'est moins essentiel.

Sur des cartes / modules qui ont un équipement intégré conséquent, les GPIO disponibles sur bornier seront généralement moindre que les "dev-kits" bruts, mais ce n'est vraiment pas un problème car en pratique, si on a déjà un bon équipement de la carte, il sera moins nécessaire d'utiliser des équipements complémentaires.

Trous de fixations ?

Tout bête, mais encore faut-il y penser : est-ce que la carte dispose de trous de fixation ? C'est toute même assez vite nécessaire.

Les cartes ESP 32 : notre choix

L'ESP 32 Dev-kit avec Lipo : la carte Olimex

Sur un projet où on a besoin d'une carte brute, la carte Olimex avec Lipo est le bon choix. Prendre en version WROOM :

https://shop.mchobby.be/fr/esp8266-esp32-wifi-iot/1626-module-esp32-wroom-esp32-coreboard-avec-chargeur-d-accu-lipo-3232100016262-olimex.html

L'ESP 32 au format carte UNO

Si on aime a envie de retrouver le format Arduino UNO en migrant vers une plateforme ESP 32, la version carte UNO avec ESP 32 est une bonne option. Cela permet potentiellement de réutiliser du matériel Arduino que l'on a, de travailler facilement avec une plaque d'essai (breadboard). L'intérêt ici est de bénéficier du régulateur présent sur la carte :

https://www.otronic.nl/a-60659537/esp32/wemos-d1-r32-esp32-4mb-development-board-wifi-bluetooth-dual-core-arduino-uno-r3-formaat/

L'ESP 32 avec écran TFT couleur

Si on a besoin sur un petit projet d'un affichage, la carte idéale est le TTGO. Pour un prix inférieur à 10€ par l'un des fabricants leader de cartes à ESP 32, on dispose sur la carte notamment de 2 BP et reset, d'un écran TFT couleur de 240x135 pixels (SPI) à driver ST7789, d'un chargeur Lipo, nombreuses GPIO sur bornier à souder.

https://opencircuit.fr/Produit/TTGO-T-Display-V1.1-ESP32-avec-%C3%A9cran-TFT-1.14

Assez comparable, les modules stick de M5Stack mais avec moins de GPIO exposée, mais dans un petit boîtier.

L'ESP 32 sur équipé : M5Stack

Que ce soit pour apprendre sur une unique plateforme tous les concepts de la programmation d'un microcontrôleur ou bien pour des projets nécessitant affichages, stockage de data, communication wifi, etc. la plateforme idéale est le core M5Stack.

Le module core est par ailleurs extensible par des stacks de la même gamme, sorte de shields, ou bien par des breakout classiques. C'est une plateforme très bien pensée et facile à mettre en oeuvre.

https://shop.mchobby.be/fr/m5stack-esp/1896-m5stack-kit-esp32-basic-iot-dev-3232100018969-m5stack.html

Et même... la montre connectée ESP 32 !

Si on souhaite du compact, on peut même disposer d'une vraie montre dans un boîtier très propre et qui contient un ESP 32 équipé d'un écran, d'un module RTC, d'un accéléromètre, vibreur, lipo, etc... Pas de GPIO exposée.

Un côté gadget très clair et assumé mais très intéressante malgré tout car dispose du wifi, permettant d'envisager des alarmes sur message MQTT par exemple. Et tout simplement, quoi de plus sympa que de se programmer sa montre connectée ?

https://opencircuit.fr/Produit/TTGO-Programmable-T-Watch-2020-ESP32-Noir

Au final

On peut vraiment prendre la carte que l'on veut, tout çà se programme en micropython et on pourra passer d'une plateforme à l'autre très facilement en fonction de ses besoins. On peut ainsi imaginer une grappe de TTGO ou de "dev-kit" ESP 32 qui envoient leurs mesures sur un M5Stack central qui rassemble de résultat.

C'est très polyvalent, rapide à mettre en oeuvre, et plaisant à utiliser sans y passer "4 plombes"... !

Suivez le guide !

Previous Next

Installation de Micropython sur ESP

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

> **Note**
>
> Pour ce qui concerne le WROOVER avec de la SPiRAM, prendre la version avec SPIRAM.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

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

- on flashe micropython :

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20210416-unstable-v1.14-170-ga9bbf7
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Liens utiles

- Voir ici : https://projetsdiy.fr/reinstaller-firmware-micropython-esp8266-esp32/
- Flasher MicroPython sur une carte NodeMCU : https://icircuit.net/nodemcu-getting-started-micropython/2406 | https://icircuit.net/esp32-micropython-getting-started/1999

Discussion technique

- ~~Ici une version de Micropython pour ESP 32 qui est censée améliorer les possibilités : https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki~~

Bases utilisation GPIO avec les ESP

GPIO

Il existe une LED onboard sur la GPIO 2 :

```
>>> from machine import Pin
>>> led=Pin(2,Pin.OUT)
>>> led.on()
>>> led.off()
```

Sur une brohce externe, çà donne :

```
from machine import Timer, Pin

led=Pin(15, Pin.OUT)

led.on()
led.offf()
```

PWM

```
from machine import PWM

pwm=PWM(Pin(15))
pwm.freq(1000)
pwm.duty(1000) # 0-1023
pwm.duty(100)
```

ADC

https://docs.micropython.org/en/latest/esp32/quickref.html#adc-analog-to-digital-conversion

La mesure analogique est disponible sur les broches 32 à 39 avec MicroPython (noté que l'ESP fournit bien plus de broches ADC que çà...), mais 8 broches ADC, c'est déjà pas mal. Les broches sont en 12 bits (par défaut) configurables. La mesure est renvoyée sur une échelle 0-4095 (=12 bits).

Point particulier : les broches disposent d'un atténuateur / amplificateur de niveau de tension. Par défaut, à 0dB, la mesure se fait sur 0-1V (et pas 3.3V !!). Pour avoir l'échelle en 3.3V, il faut utiliser l'amplification 11dB.

Le constructeur ADC se base sur une broche Pin(xx) où xx est le numéro de la broche GPxx utilisée.

```
>>> from machine import ADC, Pin
>>> sensor=ADC(Pin(32))
>>> sensor.read() # echelle 0-1v par defaut
0
>>> sensor.read()
3919
>>> sensor.atten(ADC.ATTN_11DB) # pour echelle 0-3.3V
>>> sensor.read()
442
>>> sensor.read()
4095
```

Note

L'ESP 32 disposent de 2 modules ESP 32, ADC1 et ADC2. Si le wifi est utilisé, alors ADC2 n'est pas utilisable. C'est pourquoi Micropython n'implémente pas ADC2 par défaut. On a donc seulement 4 broches analogiques. Si on a besoin de plus, il est facile

d'ajouter un multiplexeur analogique ou bien un module analogique i2C. La limitation est la même en C / Arduino car liée à l'ESP 32 lui-même.

Voir : https://github.com/micropython/micropython/issues/6219

Timer

Timer 0 à 3 disponibles : le constructeur nécessite de fournir un index de timer.

```
from machine import Timer, Pin

led=Pin(15, Pin.OUT)

timer=Timer(0)

def loop(timer):
    if led.value():led.off()
    else:led.on()

timer.init(freq=10, callback=loop)
```

SPI

ESP 8266

Attention, la configuration est un peu spécifique : on ne précise pas les broches et l'indice doit être 1 (0 correspondant à la Flash)

https://docs.micropython.org/en/latest/esp8266/quickref.html?highlight=spi#hardware-spi-bus

```
>>> sp=SPI(1) # MISO=GPIO12 MOSI=GPIO13 et SCK=GPIO14 - SPI 0 = flash !
```

ESP 32

https://docs.micropython.org/en/latest/esp32/quickref.html#hardware-spi-bus

RTC

```
from machine import RTC
rtc.datetime()
rtc=RTC()
rtc.datetime()
(2000, 1, 1, 5, 1, 11, 49, 170430)
```

RMT

https://docs.micropython.org/en/latest/esp32/quickref.html#rmt

Fonctions spécifiques ESP 32

voir ici : https://docs.micropython.org/en/latest/library/esp32.html

Communication WebREPL

Une chose très intéressante avec les ESP 32 est la possibilité de communiquer en mode interpréteur via le réseau wifi sur lequel il se trouve.

Pour cela, il suffit d'activer le serveur REPL sur l'ESP 32 (intégré par défaut dans la version Micropython des ESP)

```
>>> import network
>>> wifi=network.WLAN(network.STA_IF)
```

```
>>> wifi.isconnected()
False
>>> wifi.active(True)
True
>>> wifi.connect("dlink","")
>>> wifi.isconnected()
True
>>> wifi.ifconfig()
('192.168.0.101', '255.255.255.0', '192.168.0.1', '0.0.0.0')
>>> import webrepl
>>> webrepl.start()
WebREPL daemon started on ws://192.168.0.101:8266
Started webrepl in normal mode
>>>
```

On peut mettre çà dans une fonction dans le boot.py aussi.

+/- Configuration

```
>>> import webrepl_setup
WebREPL daemon auto-start status: enabled

Would you like to (E)nable or (D)isable it running on boot?
(Empty line to quit)
> E
Would you like to change WebREPL password? (y/n) n
No further action required
>>> import webrepl
>>> webrepl.start()
Started webrepl in normal mode
```

Lien utile :

*https://www.techcoil.com/blog/how-to-setup-micropython-webrepl-on-your-esp32-development-board/

Le client webrepl

C'est une appli web qui utilise websocket :

Installer localement.

Lancer serveur avec n'importe quel serveur de fichier statique de son choix :

```
php -S 127.0.0.1:8000
```

Se connecter sur :

```
http://127.0.0.1:8000/webrepl.html
```

Le terminal est inactif dans mon cas bien que la connexion semble se faire correctement.

Previous Next

Les bases du réseau avec les ESP

Ce qui est très (très) intéressant avec les ESP 8266 et 32, c'est de disposer du wifi en natif, ce qui est idéal pour créer des objets connectés.

Connexion en Wifi

Une des caractéristiques intéressante des modules ESP, c'est de disposer de la connexion Wifi.

On a ici besoin d'un réseau wifi, que l'on peut utiliser sans sécurisation à des fins de tests, la sécurisation pouvant être faîte plus tard (il n'y a aucun danger particulier si de plus le réseau n'est pas connecté à internet). Dans l'exemple ici, on a un réseau wifi local `dlink` public.

On commence par importer le module `network` :

```
import network
```

Ensuite, on fait :

```
>>> import network
>>> station=network.WLAN(network.STA_IF)
>>> station.active(True)
True
>>> station.connect("dlink","")
>>> station.isconnected()
True
```

```
>>> station.ifconfig()
('192.168.0.101', '255.255.255.0', '192.168.0.1', '0.0.0.0')
```

Bah bah... tout ok en moins de 2 !

Lien : https://projetsdiy.fr/tutoriel-micropython-gerer-connexion-wifi/

Test de la réponse

Ensuite, il faut créer un socket qui va permettre de gérer la connexion (à faire une fois) :

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)
```

Ensuite on gère la réception d'une connexion et la réponse à envoyer (code à boucler) :

```
s.accept()
#(<socket>, ('192.168.0.100', 51836))
conn,addr=s.accept()
print(conn)
#<socket>
print(addr)
#('192.168.0.100', 51844)
print(conn.recv(1024))
#b'GET / HTTP/1.1\r\nHost: 192.168.0.101\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686; rv:78.0) Gecko/20100101 Firefox/78.0
conn.settimeout(None)
response="Hello !"
conn.send('HTTP/1.1 200 OK\n')
conn.send('Content-Type: text/html\n')
```

```
conn.send('Connection: close\n\n')
conn.sendall(response)
conn.close()
```

Bref, on peut gérer assez simplement un petit serveur local et récupérer de la donnée sans fil. Vu que l'on est en wifi, un simple routeur de récup' fera l'affaire et on pourra avoir une portée non négligeable sans trop de problèmes.

Previous Next

Mise en route TTGO Display (ESP32)

Nous présentons ici la procédure pour flasher la version « clé en main » (conseillé) de MicroPython.

Télécharger le binaire (ou le compiler, voir section suivante)

Télécharger ici la version voulue (Generic car pas de psRAM) : https://github.com/russhughes/st7789_mpy/tree/master/firmware/T-DISPLAY

Drivers CP2104

La carte TTGO vont utiliser pour communiquer un driver CP2104 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher le binaire MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

on efface la mémoire flash de la carte :

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

on flashe micropython :

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

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

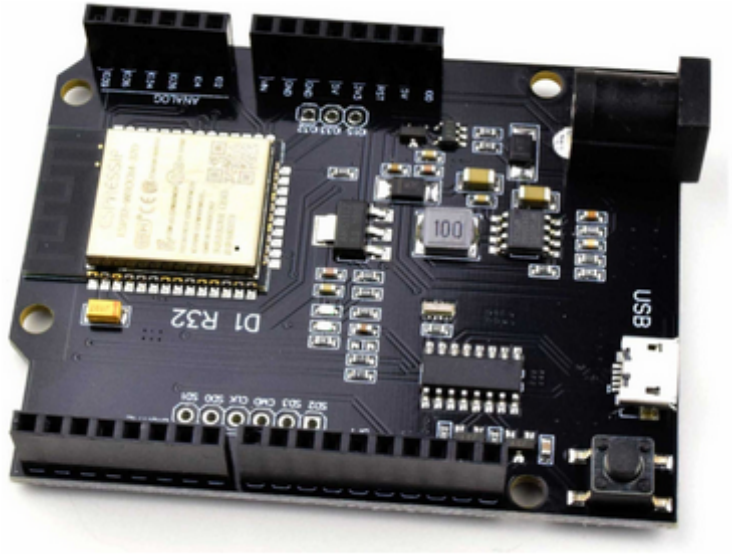
Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Ce qui donne par exemple :

```
MicroPython v1.14-150-g42035e5ed-dirty on 2021-04-11; TTGO T-Display with ESP32
Type "help()" for more information. [backend=GenericMicroPython]
>>>
```

Previous Next

Mise en route de la carte ESP 32 UNO D1 32



La carte

Il s'agit d'une carte à base d'ESP 32 au format Arduino UNO et qui reproduit l'ensemble de ce que l'on trouve sur la carte Arduino UNO en terme de brochage (à la différence d'une carte ESP 8266 qui n'a pas toutes les broches analogique notamment, etc.)

✎ Cette carte permet de faire tourner Micropython et permet donc de s'initier à Micropython tout en bénéficiant du format familier d'une carte Arduino ! **Et sans besoin de shield complémentaire, on dispose du wifi en natif !!** Une très bonne option pour passer à MicroPython !

⚠ Tous les shields Arduino ne seront pas forcément utilisables avec cette carte qui a des GPIO en 3.3V : pour les broches en sortie, pas de problème, mais pour les broches en entrée et les mesures analogiques, le 5V est interdit ! Aucun problème par contre (une nouvelle fois) avec les modules "prêts à câbler". **Soyez vigilants donc et ne connectez pas un shield Arduino les "yeux fermés" sur cette carte !**

Infos techniques

- Carte propulsée par un ESP32 à 240Mhz
- RAM : 512 Ko
- Flash : 4MB à priori
- 18 GPIO 3.3V sur bornier dont 12 broches analogiques (!) en 12 bits - 3.3V **ATTENTION : avec Micropython, on accède en natif que à A3-A5 soit 4 broches analogiques**
- jusqu'à 16 PWM
- I2C sur bornier SDA (GP21) / SCL (GP22)
- SPI

Note

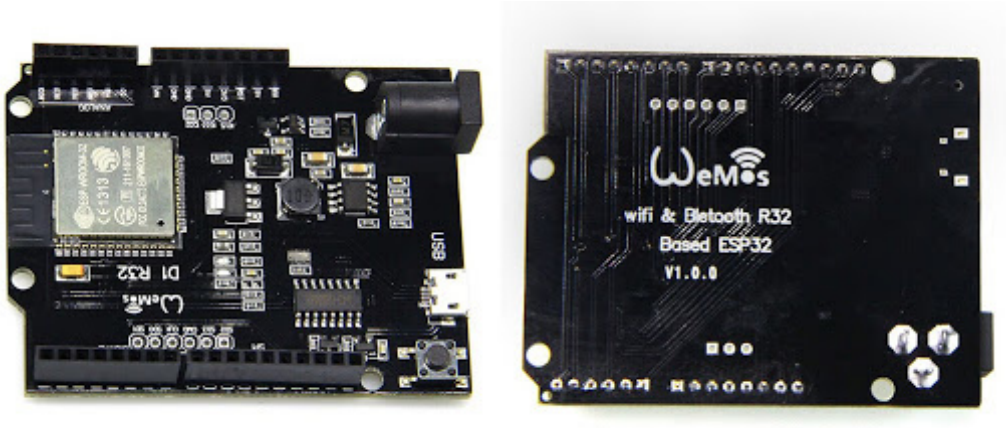
L'ESP 32 disposent de 2 modules ESP 32, ADC1 et ADC2. Si le wifi est utilisé, alors ADC2 n'est pas utilisable. C'est pourquoi Micropython n'implémente pas ADC2. On a donc seulement 4 broches analogiques. Si on a besoin de plus, il est facile d'ajouter un multiplexeur analogique ou bien un module analogique i2C. La limitation est la même en C / Arduino car liée à l'ESP 32 lui-même.

https://github.com/micropython/micropython/issues/6219

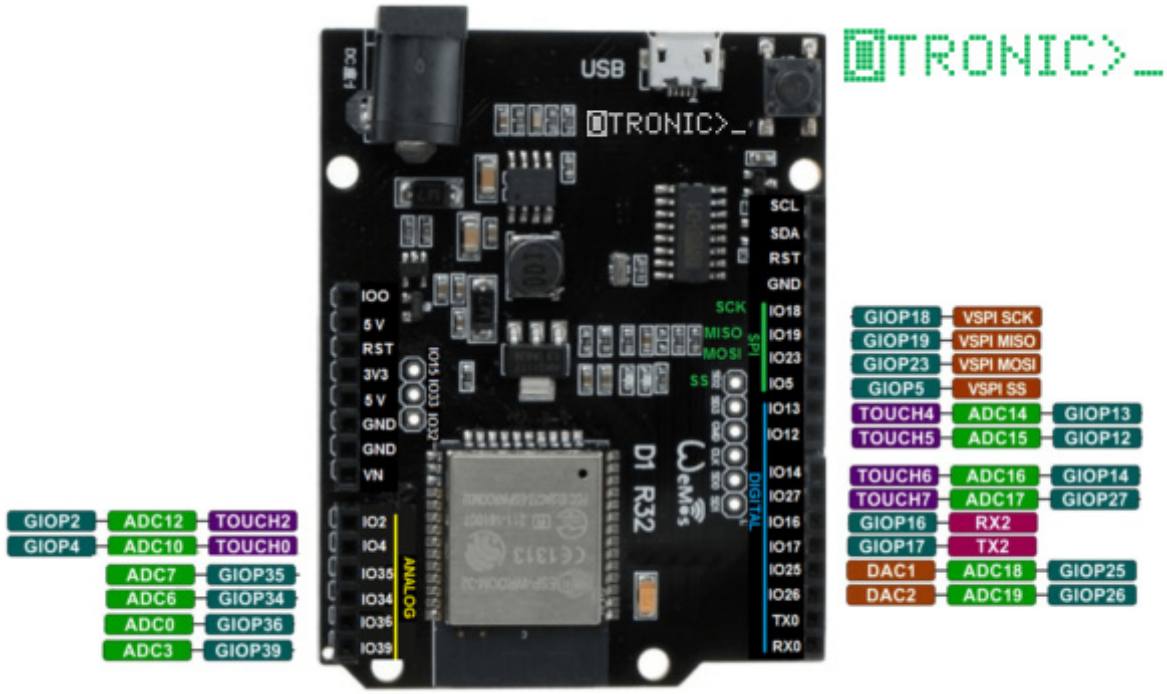
Et par rapport à une carte Arduino UNO, on a en plus :

- GPIO pour capteur sensitif

- un port UART sur GPIO (en plus de Tx/Rx sur D0/D1 utilisé par la communication USB)
- quelques GPIO supplémentaires disponibles sur connecteur à souder sur la carte
- Wifi
- Bluetooth



Brochage



Alim :

- 6-12V sur Vin
- 5V : 5V In/out
- 3.3V

> A la différence d'un ESP 32 classique, ici on dispose d'entrée de jeu d'un régulateur sur la carte permettant une alimentation externe par du 6-12V ou bien par USB, exactement comme une carte Arduino UNO.

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

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

- on flashe micropython :

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20210416-unstable-v1.14-170-ga9bbf7
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Test de micropython

```
>>> import machine
>>> machine.freq()
160000000 # 160 Mhz
```

Sur la broche D2 (GP26) on peut faire (connecter une LED) :

```
>>> from machine import Pin
>>> ledR=Pin(26, Pin.OUT)
>>> ledR.on()
>>> ledR.off()
```

Evidemment, ce qui est le plus intéressant et pratique, c'est de pouvoir garder la numérotation des broches conforme à celle d'une carte Arduino, et pour cela, il suffit de faire :

```
D=[-1,-1,26,25,17,16,27,14,12,13,5,23,19,18,2, 4, 35, 34,36,39] # correspondance ESP 32 et UNO
A=[12,10,7,8,0,3,19,18,17,16,15,14] # broches analogiques
```

Attention, la correspondance peut varier selon le modèle de carte. A adapter à votre situation

On peut dès lors faire :

```
>>> from machine import Pin
>>> led=Pin(D[2], Pin.OUT)
>>> led.on()
>>> led.off()
```

Une autre possibilité est de faire :

```
D0, D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15, D16, D17,D18,D19=[-1,-1,26,25,17,16,27,14,12,13,5,23,19,18,2, 4, 3
A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11=[12,10,7,8,0,3,19,18,17,16,15,14] # broches analogiques
```

Du coup, on pourra faire :

```
>>> from machine import Pin
>>> led=Pin(D2, Pin.OUT)
>>> led.on()
>>> led.off()
```

Pour rendre les choses définitives, on mettre cette déclaration dans le boot.py (fichier exécuté automatiquement au lancement) :

```
>>> f=open("boot.py","a")
>>> f.write("""# remap pin au format Arduino
D0, D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15, D16, D17,D18,D19=[-1,-1,26,25,17,16,27,14,12,13,5,23,19,18,2, 4, 3
A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11=[12,10,7,8,0,3,19,18,17,16,15,14] # broches analogiques
""")
162
>>> f.close()
```

Débrancher/rebrancher la carte pour bonne prise en compte.

A présent, on doit pouvoir directement pouvoir faire :

```
>>> from machine import Pin
>>> led=Pin(D2, Pin.OUT)
>>> led.on()
>>> led.off()
```

Où acheter ?

- On la trouvera ici notamment (commande Europe, livraison France pour ~2€) : https://www.otronic.nl/a-60659537/esp32/wemos-d1-r32-esp32-4mb-development-board-wifi-bluetooth-dual-core-arduino-uno-r3-formaat/

Liens utiles

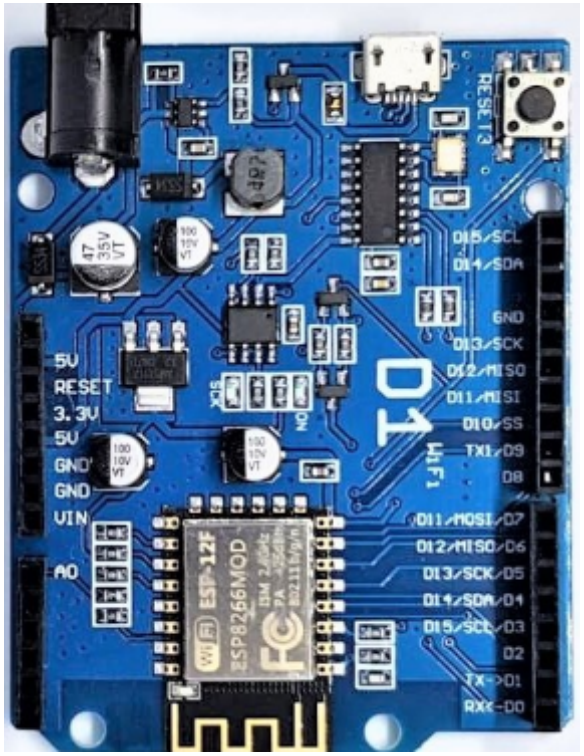
- https://www.otronic.nl/a-60659537/esp32/wemos-d1-r32-esp32-4mb-development-board-wifi-bluetooth-dual-core-arduino-uno-r3-formaat/
- https://www.instructables.com/The-Greatest-Arduino-UNO-in-the-World/
- https://www.cnx-software.com/2017/09/04/espduino-32-wemos-d1-r32-esp32-boards-support-some-arduino-uno-shields/

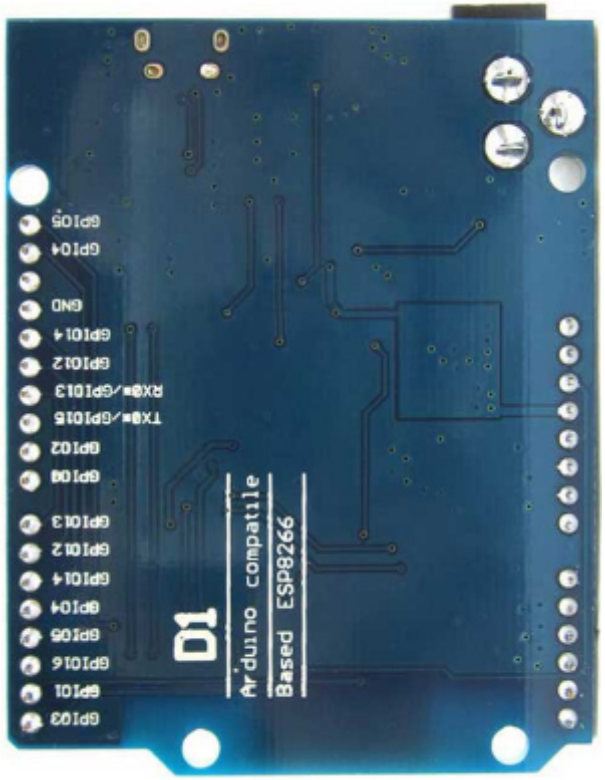
Previous Next

Mise en route de la carte ESP 8266 UNO

La carte

Il s'agit d'une carte au format de la carte Arduino UNO mais avec un coeur ESP 8266.





Infos techniques

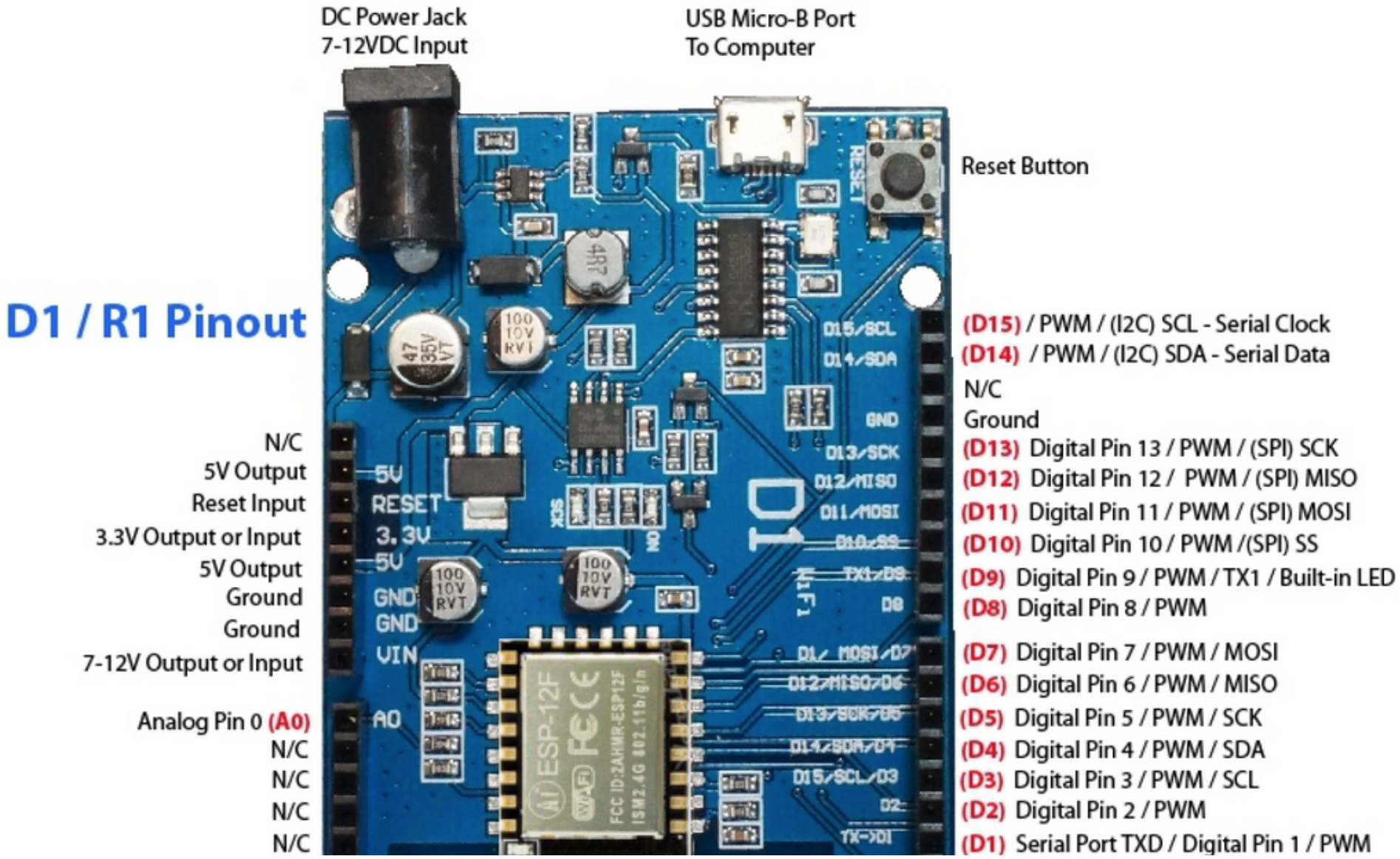
- Carte propulsée par un ESP8266EX à
- Flash : 4MB

Brochage

Quelques différences avec la UNO "vraie" :

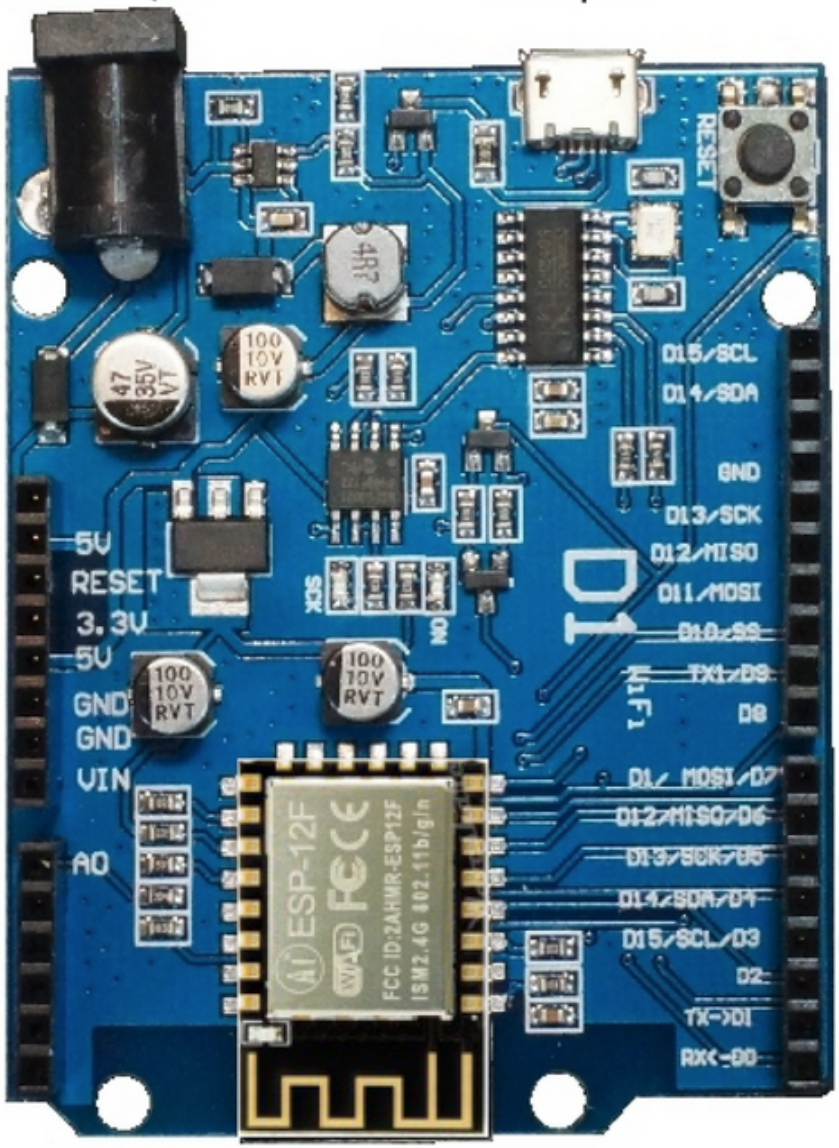
- les broches GPIO sont en 3.3V (comme la nano) !

- 1 seule voie analogique, A0
- les broches A1 à A5 sont non-connectées mais i2C disponible sur D3/D4 ou en séparé sur D14/D15.
- les broches D0 et D1 sont utilisés par la communication série USB comme sur une UNO et ne sont pas disponibles.



Le point important par contre : la dénomination des broches n'est pas celle de l'ESP 8266 et la correspondance est la suivante :

D1 / R1 Pinout



Red numbers in paranthesis are the name to use when referencing that pin.

Attention, la correspondance peut varier selon le modèle de carte. A adapter à votre situation

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp8266/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

On peut prendre à priori ici une version 2MB ou plus car on a 4MB à priori

La procédure est ici : https://docs.micropython.org/en/latest/esp8266/tutorial/intro.html#deploying-the-firmware

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

```
esptool.py --chip esp8266 --port /dev/ttyUSB0 erase_flash
```

- on flashe micropython :

```
esptool.py --chip esp8266 --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0 esp8266-20210418-v1.15.bin
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Test de micropython

On se connecte avec Thonny. On dispose de la LED onboard sur la broche D13 (GP14) et on peut donc faire :

```
>>> from machine import Pin
>>> led=Pin(14, Pin.OUT)
>>> led.on()
>>> led.off()
>>> led.on()
```

Sur la broche D3 (GP16) on peut faire :

```
>>> ledR=Pin(16, Pin.OUT)
>>> ledR.off()
>>> ledR.on()
>>> ledR=Pin(03, Pin.OUT)
>>> ledR.on()
```

Evidemment, ce qui est le plus intéressant et pratique, c'est de pouvoir garder la numérotation des broches conforme à celle d'une carte Arduino, et pour cela, il suffit de faire :

```
D=[3,1,16,5,4,14,12,13,00,02,15,13,12,14,4, 5] # correspondance ESP 8266 et UNO
```

Attention, la correspondance peut varier selon le modèle de carte. A adapter à votre situation

On peut dès lors faire :

```
>>> from machine import Pin
>>> led=Pin(D[13])
>>> led=Pin(D[13], Pin.OUT)
>>> led.on()
>>> led.off()
```

Une autre possibilité est de faire :

```
D0, D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15=[3,1,16,5,4,14,12,13,00,02,15,13,12,14,4, 5] # correspondance ESP 8
```

Pour rendre les choses définitives, on pourra faire :

```
>>> f=open("pins.py","w")
>>> f.write("""# remap pin au format Arduino
D0, D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15=[3,1,16,5,4,14,12,13,00,02,15,13,12,14,4, 5] # correspondance ESP 8
""")
163
>>> f.close
<bound_method>
>>> f.close()
>>> import pins
>>> dir()
['bdev', 'f', 'machine', '__name__', 'uos', 'c', 'w', 'gc', 'pins']
>>> from pins import *
>>> dir()
['D1', 'w', 'D0', 'D3', '__name__', 'D2', 'D5', 'D6', 'D10', 'D11', 'D12', 'D9', 'D8', 'D13', 'gc', 'f', 'pins', 'bdev', '
```

Ou même faire cette déclaration dans le boot.py

```
>>> f=open("boot.py","a")
f.write("""# remap pin au format Arduino
D0, D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15=[3,1,16,5,4,14,12,13,00,02,15,13,12,14,4, 5] # correspondance ESP 8
162
>>> f.close()
```

Débrancher/rebrancher la carte pour bonne prise en compte.

Liens utiles

- https://www.vs-elec.fr/fr/carte-sans-fil/4-d1-wifi-esp-12f-esp8266-compatible-arduino-3665662004420.html
- https://www.cnx-software.com/2015/10/21/9-esp8266-d1-board-features-arduino-uno-headers/

Previous Next

Mise en route de la carte Olimex ESP 32 Dev Kit Lipo

La carte

Il s'agit d'une carte au format dev kit C usuel expressif. **L'intérêt de cette carte, c'est qu'elle dispose d'un connecteur / chargeur lipo intégré.**

Infos techniques

- Carte propulsée par un ESP32
- Flash : 2MB à priori. Pas très clair les différences intermodèles. Pas de PSRAM sur mon modèle.
- Lipo : à priori utilisable avec un Lipo 3.7V. Sur le site, conseille **Lipo 3.7V** en 1400 mAh.

Brochage

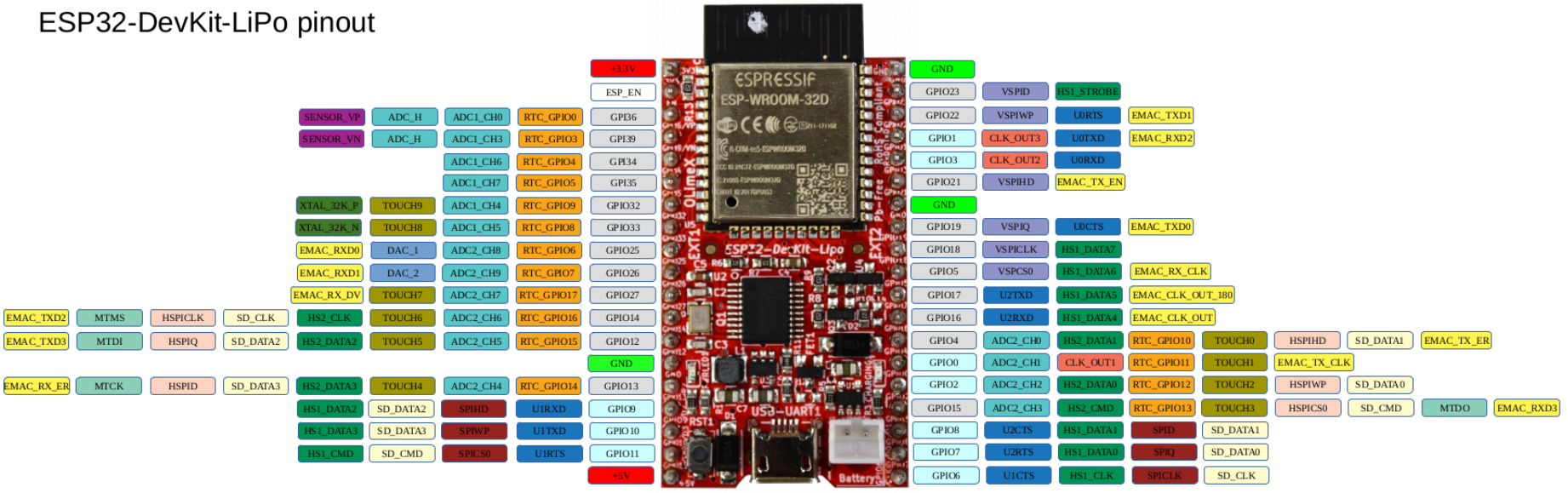
Le Wroom ne dispose pas de SRAM alors que le Wrover si. Toutes les broches idem sauf 2 en moins sur le Wrover (la 16 et la 17).

Wroom : 4Mo de Flash + 512K de RAM Wrover : 4 Mo de Flash + 512K RAM + 8Mo SPIRAM (+++)

> Info
>
> La différence de prix entre les 2 n'est que de 1 ou 2 €... mais clairement, disposer de SPIRAM offre une possibilité de développements beaucoup plus grande, la limite principale d'un développement sur ESP étant la RAM, notamment lors du chargement de fichiers, etc.

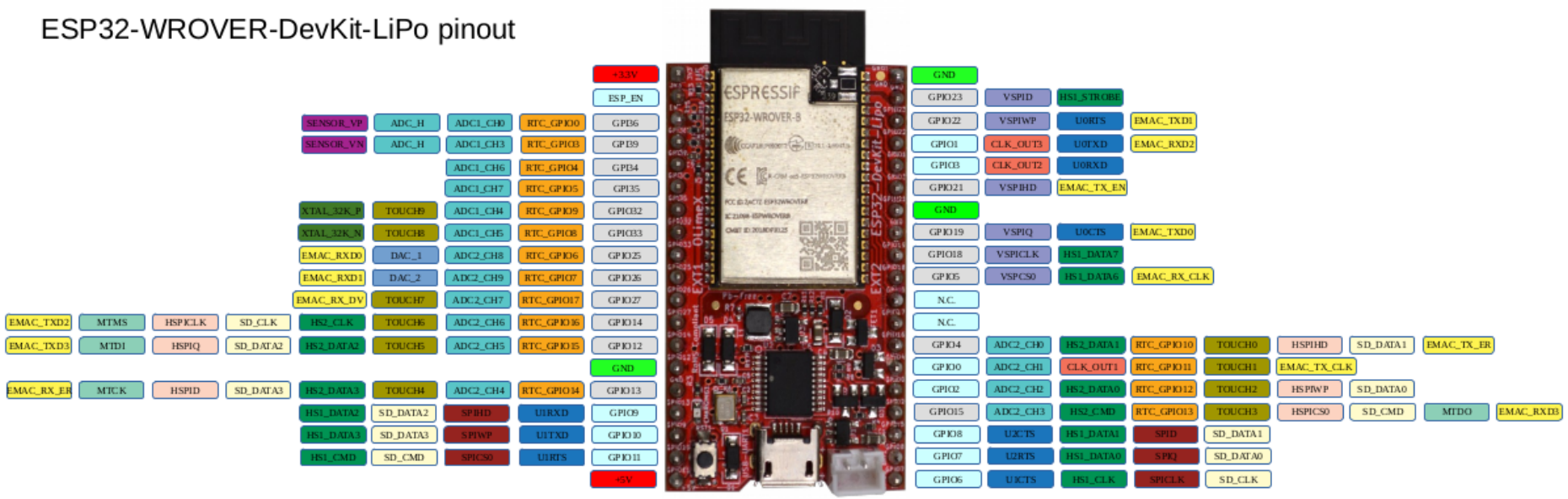
Wroom

ESP32-DevKit-LiPo pinout



GPIOs which are not available:	GPIO6, GPIO7 ,GPIO8, GPIO9, GPIO10, GPIO11 internal SPI Flash
	ESP-EN – RESET
	GPI39 if PWR_SENS_E1 jumper is closed used to monitor if power supply is present
	GPI34 if BAT_SENS_E1 jumper is closed used to monitor the battery voltage level
	GPIO0, GPIO1, GPIO2, GPIO3 connected to CH340T USB serial converter for programming and communication

Wrover



GPIOs which are not available:	GPIO6, GPIO7 ,GPIO8, GPIO9, GPIO10, GPIO11 internal SPI Flash
	ESP-EN – RESET
	GPI39 if PWR_SENS_E1 jumper is closed used to monitor if power supply is present
	GPI34 if BAT_SENS_E1 jumper is closed used to monitor the battery voltage level
	GPIO0, GPIO1, GPIO2, GPIO3 connected to CH340T USB serial converter for programming and communication
	N.C. - not connected, you can't use these pins

Pour les 2 :

Bon à savoir : d'après le datasheet de la carte :

- VBAT sur GP35
- VIN sur GP39
- pas de LED onboard ?
- RX/TX sur GPIO01 et GPIO03

Alim :

- 5V : 5V In/out
- 3.3V : Out only à 600mA max (pas mal).

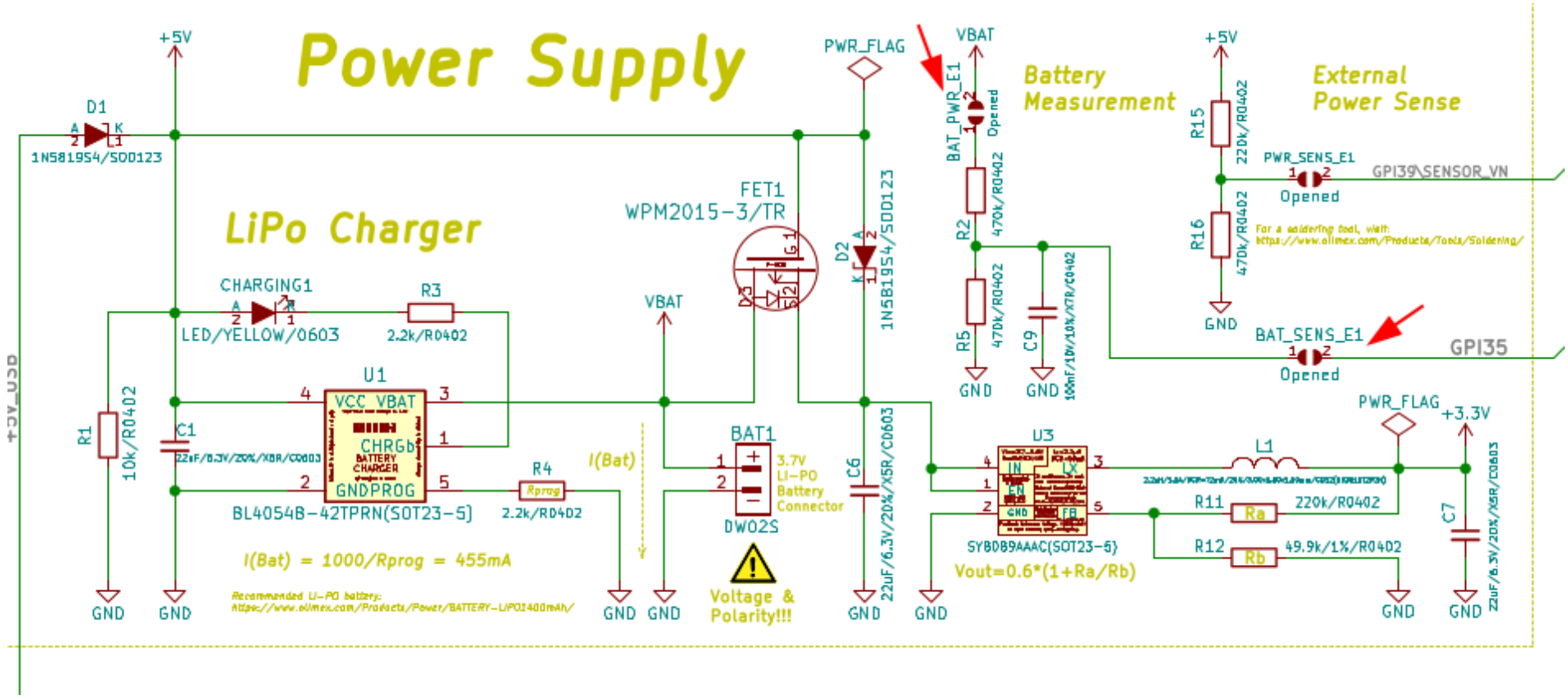
ESP32-DevKit-Lipo_Rev_B.pdf

Note sur VBatt Lipo sur GP35

On dispose de VBatt Lipo sur GP 35 mais il faut pour cela faire 2 soudures :

- une en face de GP 35
- une à côté du lipo

On a alors accès à la tension de VBat/2 (pont diviseur par 2 R de 470K) sur GP 35. Cela donne donc indirectement un état de la charge du Lipo.



Voir ici pour les tensions du Lipo en fonction de la charge. Ici, le Lipo est un 1S = 3.7V. https://blog.ampow.com/lipo-voltage-chart/

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

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

- on flashe micropython :

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20210416-unstable-v1.14-170-ga9bbf7
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Test de micropython

```
>>> import machine
>>> machine.freq()
160000000 # 160 Mhz
>>> from machine import Pin
```

```
>>> led=Pin(23, Pin.OUT)
>>> led.on()
>>> led.off()
```

Liens utiles

- https://www.olimex.com/Products/IoT/ESP32/ESP32-DevKit-LiPo/open-source-hardware

Previous Next

Carte ESP 32 M5Stacks

La gamme M5 Stacks fournit un système de "modules" ou plutôt de "blocs" stackable pour des projets.

Plusieurs gammes existent mais pour faire simple, on a 3 "séries" différentes : Core, atom et pico.

La gamme Core offre un module de base qui intègre un écran TFT utilisant le driver

Previous Next

M5 Stack : Intro

La gamme M5 stacks est intéressante et un peu à part.

Le point de départ est un module "Core" qui comprend :

- un ESP 32 et donc avec Wifi, BT, etc.
- un écran TFT 320x240 (à communication SPI - déjà câblé)
- BP x 3
- lecteur SD
- GPIO exposée en latéral sur bornier F ou M, en symétrique
- un Lipo intégré dans le "bottom" de 110mAh / 3.7V (et donc à priori le chargeur est sur la carte)
- un HP avec ampli audio...
- livré avec câble USB-C, quelques jumper...
- les fonctions de l'ESP 32 donc ADC, DAC, I2C, UART, SPI, et même IIS...

NOtes techniques :

L'ESP 32 a 2 SPI : * le SPI sur bornier est le même que la SD Card * l'autre SPI est utilisé par l'écran

Techniquement :

- l'écran TFT est un 320x240 couleur utilisant le driver ILI9341

L'écran TFT est un 320x240, en couleur, à comm' SPI ce qui est nickel pour faire des petits oscillos, etc... avec une bonne réactivité.

Bien vu :

- reprise en symétrique mâle et femelle des GPIO et des Bus
- connecteur pour Lipo additionnel (contient un lipo de base de 150 mAh)
- le bouton de reset / ON-OFF latéral
- i2C sur Grove
- des aimants pour fixation sur plaque métallique
- boitier en plastique (polycarbonate = recyclable)

Bémols :

- connexion USB-C mais cordon livré avec la carte
- Connecteur Grove (mais non obligatoire...)
- ADC limitées (2 seulement)

Le tout dans un boîtier très propre.

Voir : https://shop.m5stack.com/collections/m5-core/products/basic-core-iot-development-kit

Ce qui est intéressant c'est :

- le côté "clé en main"
- signalétique évidente sur les boîtiers qui indiquent tout ce dont on dispose dedans
- le côté "stackable" facile, et notamment un module proto à 4€
- le prix est correct : c'est un peu plus cher que les éléments pris isolément, mais si on considère le boitier, le côté fini et clé en main, les équipements intégrés, clairement, c'est loin d'être déconnant...

Il y a une volonté opensource et il n'y a rien en soi qui empêche de faire des modules soi-même. Le projet est annoncé au départ "opensource".

De toutes les solutions existantes, c'est la "moins con" pour ne pas dire la plus intelligente de tout l'existant, une solution qui n'enferme pas outre mesure tout en simplifiant la vie. On reste sur un ESP 32 qui est le "must" pour le wifi,

Entièrement documenté et notamment le câblage interne : https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/docs/schematic/Core/M5-Core-Schematic(20171206).pdf

Principe

On peut enlever le fond ce qui expose l'intérieur : une carte avec l'ESP 32 et l'ecran TFT et les BP, le ON/OFF, etc.

Les GPIO sont organisés en bornier I/O (GP, ADC, DAC, IIS, UART) et borniers Bus (SPI, I2C) mais toutes les broches sont utilisables avec les broches de l'ESP 32 correspondantes.

Previous Next

Mise en route d'un core M5 Stack

Présentation

Il s'agit d'un module basé sur un ESP 32

Infos techniques

Avant tout un ESP 32 :

- Carte propulsée par un ESP32 à 240Mhz avec wifi et Bluetooth
- RAM : 512 Ko
- Flash : 4MB à priori

- 18 GPIO 3.3V sur bornier dont 12 broches analogiques (!) en 12 bits - 3.3V **ATTENTION : avec Micropython, on accède en natif que à A3-A5 soit 4 broches analogiques**
- jusqu'à 16 PWM
- I2C sur bornier SDA (GP21) / SCL (GP22)
- SPI

Le module "Core" intègre de plus :

- un écran TFT 320x240 (SPI)
- lecteur carte SD (SPI)
- 3 BP en façade
- Un BP on/off/reset
- un lipo 110 mAh / 3.7V

> **Note**
>
> L'ESP 32 disposent de 2 modules ESP 32, ADC1 et ADC2. Si le wifi est utilisé, alors ADC2 n'est pas utilisable. C'est pourquoi Micropython n'implémente pas ADC2. On a donc seulement 4 broches analogiques. Si on a besoin de plus, il est facile d'ajouter un multiplexeur analogique ou bien un module analogique i2C. La limitation est la même en C / Arduino car liée à l'ESP 32 lui-même.
>
> https://github.com/micropython/micropython/issues/6219

Brochage

Alim In :

- 5V sur USB-C

Alim Out :

- 5V out
- 3.3V out

Drivers CH3040

Les cartes ESP 32 vont utiliser pour communiquer un driver CH340 qui va fournir un port série au format `ttyUSB0` sur Gnu/Linux.

Ces drivers sont directement présents sous Gnu/Linux. Mais doivent à priori être installés sous WIndows et Os X.

Flasher MicroPython

Pour cela on utilise un outil dédié écrit en python, `esptool.py` et qui s'installe logiquement avec l'installateur python `pip` :

```
sudo apt-get install python3-pip
sudo pip3 install esptool
```

Télécharger la version adaptée de MicroPython : https://micropython.org/download/esp32/ La version générique fonctionne que la carte dispose ou non de SRAM externe.

Noter que l'on dispose aussi d'une version dédiée de MicroPython pour M5Stack, mais elle n'a pas forcément toutes les fonctionnalités utiles, etc.

Ensuite, on peut flasher facilement le firmware `micropython` dans l'ESP 32. Ouvrir un terminal dans le répertoire où se trouve le binaire puis successivement :

- on efface la mémoire flash de la carte :

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

- on flashe micropython :

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000 esp32-20210416-unstable-v1.14-170-ga9bbf7
```

Une fois fait, on peut tester que tout est OK en se connectant avec un terminal série sur `/dev/ttyUSB0`

Avec Thonny par exemple, aller dans Tools > Options > Interpreter > sélectionner la bonne carte

Test de micropython

```
>>> import machine
>>> machine.freq()
160000000 # 160 Mhz
```

Sur la broche GP5 on peut faire (connecter une LED) :

```
>>> from machine import Pin
>>> ledR=Pin(5, Pin.OUT)
>>> ledR.on()
>>> ledR.off()
```

Note

Le nombre de GPIO disponibles sur bornier est réduit ou plus exactement, on a aussi les broches dites de "Bus" aussi

Aperçu de la carte

	GND	1	2	ADC1	GPIO35
	GND	3	4	ADC2	GPIO36
	GND	5	6	RESET	EN
GPIO23	MOSI	7	8	DAC0/AUDIO_L	GPIO25
GPIO19	MISO	9	10	DAC1/AUDIO_R	GPIO26
GPIO18	SCK	11	12	3.3V	
GPIO3	IO0/RXD1	13	14	IO1/TXD1	GPIO1
GPIO16	IO2/RXD2	15	16	IO3/TXD2	GPIO17
GPIO21	IO4/SDA	17	18	IO5/SCL	GPIO22
GPIO2	IO6	19	20	IO7	GPIO5
GPIO12	IO8/IIS_SCLK	21	22	IO9/IIS_WS	GPIO13
GPIO15	IO10/IIS_OUT	23	24	IO11/IIS_MCLK/BOOT	GPIO0
	HPWR	25	26	ADC0/IIS_IN	GPIO34
	HPWR	27	28	5V	
	HPWR	29	30	BATTERY	

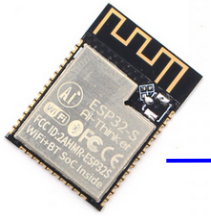
20 GPIO internes sur connecteur 15x2
I2C / SPI / UART x2 / IIS
ADC x 2 / DAC x 2
3V3 et 5V

Ampli Audio 1W NS4148

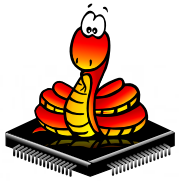
Audio	ESP32
Input	GP25 / DAC 0

Ecran TFT couleur 2" 320x240 pixels
Driver ILI9342C - Comm' SPI

TFT	ESP32
Backlight	GP32
#RST	GP33
R/S	GP27
MOSI	GP23
SCK	GP18
CS	GP14



ESP 32 240MHz
520Ko RAM
4Mo Flash
Wifi + Bluetooth



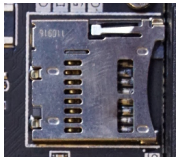
BP ON/OFF - Reset

Vin 5V/500mA sur USB-c

BP x 3 avec rappel au +3.3V câblé

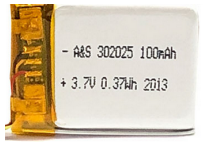
BP	ESP32
BP A	GP37
BP B	GP38
BP C	GP39

Lecteur Carte Micro-SD



SD	ESP32
SPI_SDCS	GP04
SPI_SDDI	GP23
SPI_CLK	GP18
SPI_SDDO	GP19

15 GPIO sur Dupont F + M dont :
I2C / SPI / UART
ADC x 2 / DAC x 2
3V3 et 5V



Chargeur Lipo / Gestion alim IP5306
I2C (0x75)
Lipo 3.7V / 110 mAh intégré
+ connecteur Lipo

Où acheter ?

-

Liens utiles

- Projets avec M5Stack
- Un projet ici avec boitier openscad : https://tinkerman.cat/post/m5stack-node-things-network
- https://github.com/xoseperez/m5stack-rfm95/blob/master/enclosure/m5stack-rfm95-bottom.stl
- https://github.com/xoseperez/m5stack-rfm95/tree/master/enclosure
- https://github.com/xoseperez/m5stack-rfm95/blob/master/enclosure/m5stack-rfm95.scad
- Pour écran du M5Stack : cf Micropython compilé avec Driver C intégré. : https://github.com/russhughes/ili9342c_mpy/tree/main/examples/M5STACK

Note : à priori le ILI9341 devrait fonctionner avec.

Previous Next

Picoweb

Picoweb est un microframework : picoweb, façon bottlepy ou flask.. :

Liens utiles

- https://icircuit.net/make-esp32-http-webserver-using-micropython/2152
- < https://github.com/pfalcon/picoweb>
- https://www.dfrobot.com/blog-934.html
- https://www.dfrobot.com/blog-742.html

Ce framework permet de créer des routes à la façon de Bottlepy par exemple.
https://github.com/pfalcon/picoweb

- Ici aussi : https://www.cnx-software.com/2017/10/16/esp32-micropython-tutorials/

Pré-requis

Etre connecté au réseau wifi pour avoir accès au web :

```
>>> import network
>>> wifi=network.WLAN(network.STA_IF)
>>> wifi.active(True)
True
>>> wifi.connect("ssid","password")
>>> wifi.isconnected()
True
```

Ceci peut aussi être automatisé dans un `boot.py`

Installation

Note : l'installation va nécessiter que l'ESP ait accès au Web au moins pour l'installation. picoweb permet aussi l'utilisation de template ce qui à priori est intéressant.

On fait l'installation à l'aide de pip depuis le micropython !

```
>>> import upip as pip
>>> pip.install("picoweb")
Installing to: /lib/
Warning: micropython.org SSL certificate is not validated
Installing picoweb 1.8.2 from https://files.pythonhosted.org/packages/c2/22/a1eb0cf52b72e818fe47acadaf8ade200d7c0c7c6fc5ac
Installing pycopy-uasyncio 3.7 from https://files.pythonhosted.org/packages/e5/58/80b8b403c52ea88d44844570dbe487d7a4b3045a
Installing pycopy-pkg_resources 0.2.1 from https://files.pythonhosted.org/packages/05/4a/5481a3225d43195361695645d78f44395
Installing pycopy-uasyncio.core 2.3.2 from https://files.pythonhosted.org/packages/ca/b2/c5bba0bde7022b6d927a6144c026d7bf3
>>>
```

Enorme !

Perso, je trouve çà "énorme" de pouvoir aussi facilement installer un truc sur l'ESP à partir de micropython !

Application de test

Ici, une simple webapp de test où on explore les possibilités, notamment d'utiliser du fichier statique... et çà fonctionne !

```
import network
import picoweb
```

```
wifi = network.WLAN(network.STA_IF)
if not wifi.isconnected():
  print('connecting to network ... ')
  wifi.active(True)
  wifi.connect("DWR-921-6E38","12345678910")
  while not wifi.isconnected():
    pass

print('network config:', wifi.ifconfig())

ipadd=wifi.ifconfig()

app = picoweb.WebApp(__name__)

@app.route("/")
def index(req, resp):
    print(req.method) # info sur la methode requete GET ou POST
    yield from picoweb.start_response(resp)
    yield from resp.awrite("Hello !")
    yield from resp.awrite("Or my <a href='file'>file</a>.")
    # lien renvoie vers une route dediee
    yield from resp.awrite("Or my <a href='boot.py'>file py</a>.")
    yield from resp.awrite("Or my <a href='smoothie.js'>file js</a>.")
    yield from resp.awrite("Or my <a href='html'>page html</a>.")


@app.route("/file")
def static_file(req, resp):
        yield from app.sendfile(resp, "boot.py")

@app.route("/boot.py")
def static_file(req, resp):
        yield from app.sendfile(resp, "boot.py")
```

```
@app.route("/smoothie.js")
def static_file(req, resp):
        yield from app.sendfile(resp, "smoothie.js")

@app.route("/html")
def page_html(req, resp):
        page=b"""
                <!DOCTYPE HTML>

        <!-- Debut de la page HTML  -->
        <html>

                <head> <!-- Debut entete -->

                        <meta charset="utf-8" /> <!-- Encodage de la page  -->
                        <title>JavaScript: Test librairie Smoothie</title> <!-- Titre de la page -->

                        <!-- Inclusion librairies / codes Javascript externes -->
                        <script src="smoothie.js" type="text/javascript" ></script>

                        <!-- Debut du code Javascript  -->
                        <script language="javascript" type="text/javascript">


                                // variables globales
                                //var elem; // variable globale élément du DOM
                        var randomSerie = new TimeSeries(); // crée un objet série de date - fournit par lib smoothie
                                var smoothieGraph = new SmoothieChart({millisPerPixel:50}); // crée un objet graphique smo

                                window.onload = function () { // au chargement de la page

                                        // acces aux éléments du DOM
                                        //elem=document.getElementById("myElem");
```

```
                                        // définition de la gestion des évènements du DOM
                                        //elem.setAttribute('onchange', 'fonction()'); // appel la fonction voulue

                                        // code a exécuter au chargement de la page

                                        // initialisation du graphique smoothie
                                        smoothieGraph.addTimeSeries(randomSerie, { strokeStyle: 'rgba(0, 255, 0, 1
                                        smoothieGraph.streamTo(document.getElementById("smoothieCanvas"), 1000); /

                                setInterval(function() { refreshSmoothieGraph()}, 500); // fixe délai actualisatio

                                        // le résultat final est le fruit du mixage entre vitesse de défilement et

                        } // fin onload

                        // fonctions de gestion des évènement du DOM
                        function fonction(){


                        } // fin fonction

                        // autres fonctions

                        // fonction d'actualisation de la série de valeur à intervalle régulier
                        function refreshSmoothieGraph(){
                                randomSerie.append(new Date().getTime(), Math.random() * 4096); // ajoute donnée a
                        } // fin refresh smootthie

                </script> <!-- Fin du code Javascript -->

        </head> <!-- Fin entete -->

        <body > <!-- Debut Corps de page HTML -->
```

```
                    <!-- == placer ici le code HTML de la page == -->
                    <canvas id="smoothieCanvas" width="600" height="300"></canvas> <!-- canvas pour graphique smoothie


                </body> <!-- Fin de corps de page HTML  -->

            </html> <!-- Fin de la page HTML  -->

            """
            yield from picoweb.start_response(resp)
            yield from resp.awrite(page)


app.run(debug=-1, port=8000, host =ipadd[0]) # debug -1 pour eviter probleme ulogging
```

Doc minimale

run(host="127.0.0.1", port=8081, debug=False, lazy_init=False, log=None)

- mettre debug=-1 pour ne pas avoir message erreur avec ulogging
- port par défaut est 8081... fixé ce que l'on veut.

Pour l'analyse de la requete, voir ici :

- https://www.dfrobot.com/blog-747.html

Autour de senfile

COntent Type : https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/Content-Type

Mime Type : https://developer.mozilla.org/fr/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Note : gérer le chargement des librairies :

https://aaronsmith.online/easily-load-an-external-script-using-javascript/ https://webexplorar.com/jquery-check-page-is-fully-loaded-or-not/

Compléments potentiellement utiles

Un tuto pas mal ici : https://itywik.org/2018/10/30/eight-micropython-python-experiments-for-the-esp32/

Il est possible d'utiliser des versions compressées de librairies qui seront décompressés par le navigateur : https://www.alsacreations.com/article/lire/914-compression-pages-html-css-gzip-deflate.html

Ici par exemple, dans le cas de jquery, l'utilisation de minifié + gzipped fait passer une lib de 255Ko à 29Ko... à tester du coup ! https://mathiasbynens.be/demo/jquery-size

Dans le cas précis de Brython, il est aussi possible de mémoire de "compiler" en JS non ?

Test de Brython sur l'ESP

Création d'un répertoire static pour y placer :

- brython.js : attention le fichier est gros (700Ko et la copie prend un peu de temps) Une autre option serait de la mettre sur un serveur commun sur le réseau local, un Pi.
- les libs éventuelles

Trop lourd pour l'ESP

Par contre, bien que fonctionnelle, clairement, cette façon de faire est trop lourde pour l'ESP : çà met plusieurs 10aine de secondes à transférer le fichier brython.js... et on n'a pas la lib standard.

Donc, à minima, préférer un accès web : Et si on veut isoler le réseau, mettre un Pi avec fichier de serveur statique, voire même le poste depuis lequel on accède.

A noter que l'adresse à utiliser dans le index.html sera celle sur laquelle on accède au brython.js avec le navigateur. Le plus simple est de la tester en manuel pour éviter les gags...

Création d'un répertoire app pour y placer :

- index.html
- scriptBrython.py

Serveur picoweb qui a routes :

- /app/index.html
- /app/scriptBrython.py
- /static/brython.js

Test Ajax

cf essai Brython

Ou bien avec Jquery.

Cf dans ce cas les alternatives légères à jquery d'une part. https://www.technotification.com/2019/06/5-lightweight-jquery-alternatives-2019.html

Autour jQuery :

https://raygun.com/blog/jquery-is-undefined/ https://stackoverflow.com/questions/1956719/jquery-get-is-not-a-function https://www.pierre-giraud.com/jquery-apprendre-cours/creation-requete-ajax/

Test websocket

Libs websockets en Micropython :

- https://github.com/danni/uwebsockets

Au passage découverte d'un autre projet de webserver : https://github.com/jczic/MicroWebSrv2

Un projet intéressant ici utilisant MicroWebserver et websocket : https://www.rototron.info/raspberry-pi-esp32-micropython-websockets-tutorial/ https://www.rototron.info/raspberry-pi-esp32-micropython-websockets-tutorial/

Previous Next

Test de serveur simple

L'intérêt majeur de l'ESP est de pouvoir créer des petits serveurs simples. Donc ici, on explore la possibilité de faire çà.

Une très bonne source : https://microcontrollerslab.com/esp32-esp8266-micropython-web-server/

Ce qu'il faut savoir :

Le module ESP a 2 modes possibles :

- soit comme un poste sur le réseau wifi
- soit comme un point wifi à lui-tout seul.

Débrouillage dans l'interpréteur

```
>>> ## initialisation connexion
>>> import socket
>>> import network
>>> wifi=network.WLAN(network.STA_IF)
>>> wifi.active()
False
>>> wifi.connect("dlink","")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Wifi Unknown Error 0x0005
>>> wifi.active(True)
True
>>> wifi.connect("dlink","")
```

```
>>> wifi.isconnected()
True
>>> ## le serveur
>>> from machine import Timer
>>> timer=Timer(0)
>>> page="""Hello world !""")
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax
>>> page="""Hello world !"""
>>> s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.bind(('',80))# port
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
>>> s.bind('',80)# port
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
>>> s.bind(("",80))
>>> s.listen(5) # max connection
Traceback (most recent call last):
  File "<stdin>", line 2
SyntaxError: invalid syntax
>>> s.listen(5)
>>> def loop(timer):
    conn,addr=s.accept()
    conn.settimeout(3.0)
    request=conn.recv(1024)
    conn.settimeout(None)
    request=str(request)
    print(request)
    response=page
    conn.send('HTTP/1.1 200 OK\n')
```

```
    conn.send('Content-Type: text/html\n')
    conn.send('Connection: close\n\n')
    conn.sendall(response)
    conn.close()

>>> while True:
    loop()
```

Ensuite se connecter sur l'ip de l'ESP 32, sur le port sépcifié :

```
192.168.4.100:80
```

On obtient la page et dans le terminal on a la réponse qui apparaît à chaque requete.

```
b'GET / HTTP/1.1\r\nHost: 192.168.4.100\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686; rv:78.0) Gecko/20100101 Firefox/78.0\
```

Rendre le choses plus définitives

Pour rendre les choses plus définitives, on crée 2 fichier :

- un `boot.py` qui va contenir la partie activation de la connexion
- un `main.py` qui va gérer la partie serveur à proprement parler

Questions

Est-il possible ou facile de servir des fichiers statiques et donc librairie *.js avec ESP 32 ?

Un microframework : picoweb, façon bottlepy ou flask.. :

- https://icircuit.net/make-esp32-http-webserver-using-micropython/2152
- < https://github.com/pfalcon/picoweb>
- https://www.dfrobot.com/blog-934.html
- https://www.dfrobot.com/blog-742.html

Ce framework permet de créer des routes à la façon de Bottlepy par exemple.
https://github.com/pfalcon/picoweb

https://www.cnx-software.com/2017/10/16/esp32-micropython-tutorials/

Note : l'installation va nécessiter que l'ESP ait accès au Web au moins pour l'installation. picoweb permet aussi l'utilisation de template ce qui à priori est intéressant.

Si on veut servir des librairies js, etc... on peut :

- les intégrer directement dans le code de la page que l'on envoie si elles ne sont pas trop grosses, ce qui est le cas de la majorité des libs js, notamment a format "minifié"
- les ouvrir à partir fichiers locaux sur la Flash de l'ESP (on a 2 Mo de dispo, ce qui laisse quand même de la marge) selon route prédéfinie avec picoweb
- les ouvrir de façon similaire depuis une carte SD si besoin de plus... os
- utiliser un serveur local, un pi par exemple, qui va fournir les lib pour tous les esp présents sur le réseau. C'est une bonne alternatives
- on peut aussi imaginer que le poste depuis lequel on accède aux ESP serve les libs utiles
- et bien sûr, que le réseau ait accès au web, auquel cas, les libs sont dispos

Ce qui est très tentant ici c'est d'utiliser Brython pour le client et faire des pages très propres...

Tester aussi ajax, ce que devrait à priori permettre facilement picoweb via les routes.

Et aussi

Un autre projet de serveur Micropython qui supporte Ajax et websocket : https://github.com/jczic/MicroWebSrv2

Gérer les requetes avec les sockets

Le microframework, c'est bien, mais çà peut poser des problèmes, ne pas être ultra rapide, et par conséquent, çà peut être bien d'utiliser les sockets et gérer directement les requêtes reçues, etc.

A la mano

Probablement encore le plus efficace :

Voici une requête GET type :

```
GET / HTTP/1.1
Host: 192.168.0.52:8080
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: fr,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-GPC: 1
```

Là, ce qu'on veut c'est extraire ce qu'il y a derrière le GET, le / et le texte qui suit.

Une façon simple consiste à faire :

Avec un module ?

Pour se simplifier la vie, on a un module disponible, 'urllib.parse'

Installation (le wifi étant actif sur l'ESP)

```
import upip as pip
pip.install("urllib.parse")
```

https://pypi.org/project/micropython-urllib.parse/

Ne fonctionne pas pour moi

Problèmes de mémoire

https://stackoverflow.com/questions/62550514/how-to-resolve-memoryerror-memory-allocation-failed-with-micropython

```
import micropython
micropython.alloc_emergency_exception_buf(100)
```

Voir : https://docs.micropython.org/en/latest/library/micropython.html?highlight=module%20micropython#micropython.alloc_emergency_exception_buf A mettre au début du code.

Lib alternative à jQuery pour Ajax call ?

j'utilise / ai besoin de jQuery essentiellement pour simplifier l'appel Ajax... mais si on peut faire autrement, alors, j'en n'ai plus besoin et on peut faire des petits serveurs ESP entièrement "embarqué" et donc en hotspot ce qui peut être particulièrement pratique. Il suffit de se connecter sur l'ESP pour contrôler, avoir les datat, etc..

Très bonne page ici : https://dzone.com/articles/top-javascript-libraries-for-making-ajax-calls

Fetch a l'air de ne même pas nécessiter de lib...

- https://www.w3schools.com/js/js_api_fetch.asp
- https://www.javascripttutorial.net/javascript-fetch-api/
- Tuto ici qui refait l'historique et lie vers explication des promises : bien fait : https://www.freecodecamp.org/news/javascript-fetch-api-tutorial-with-js-fetch-post-and-header-examples/

Librairie js intéressantes dans un contexte de microwebserveur embarqué ?

Alpine.js : une lib de 6.4 (minified + gzipped) qui permet de coder "inside the html" et qui permet la capture d'évènement, l'interaction avec les éléments eux-mêmes, etc.

- https://www.smashingmagazine.com/2020/03/introduction-alpinejs-javascript-framework/
- intro qui est pas mal du tout : https://devdojo.com/tnylea/alpinejs-for-beginners
- ajax : https://dberri.com/lets-build-an-ajax-form-with-alpine-js/ et https://codewithhugo.com/alpinejs-x-data-fetching/

Previous Next

Websocket avec micropython ?

Techniquement, on a une implémentation indépendante côté serveur et côté Javascript. Et finalement la notion de serveur et de client disparaît avec le Micropython

Côté serveur (micropython)

Une page qui semble être un bon point de départ est :

https://www.rototron.info/raspberry-pi-esp32-micropython-websockets-tutorial/

Une lib qui ne semble pas être très développée cependant :

https://github.com/danni/uwebsockets

Pour mémoire, en Python

Note : çà semble presque plus facile de faire un serveur websocket en Micropython qu'en Python... du moins avec bottlepy.

Le lien précédent permet de créer un client-websocket en Python :

https://www.rototron.info/raspberry-pi-esp32-micropython-websockets-tutorial/

Côté Javascript :

Les websocket sont natives dans le navigateur.

https://www.tutorialspoint.com/websockets/websockets_javascript_application.htm

A noter que Brython permet d'utiliser simplement les websockets, en tout cas permet de faire assez facilement une page côté client.

Previous Next

Accessoires

- Boîte 3D avec OLED : https://www.thingiverse.com/thing:4331477

Previous Next

Projets à base d'ESP

Projets à base d'ESP 32

- un serveur simple : https://randomnerdtutorials.com/micropython-esp32-esp8266-bme680-web-server/
- contrôler servo : https://icircuit.net/micropython-controlling-servo-esp32-nodemcu/2385

Sources intéressantes

- https://projetsdiy.fr/microcontroleurs-mcu/esp32-iot/page/2/ Les projets MicroPython et Arduino sont mélangés mais il y a quelques bonnes pages MicroPython.
- https://www.techcoil.com/blog/how-to-setup-micropython-on-your-esp32-development-board-to-run-python-applications/
- Une présentation très complète avec de bonnes explications est disponible ici (bilan des ESP en 2020) : https://projetsdiy.fr/quelle-carte-esp32-choisir-developper-projets-diy-objets-connectes/

Voir également les fonctions spécifiques à l'ESP 32 fournies par Micropython :
https://docs.micropython.org/en/latest/library/esp32.html

Previous Next

Envoi de message Element / Matrix depuis un ESP

A priori, on peut passer par une simple requête web, si compte existant sur lequel on veut poster.

Il y a tout ce qu'il faut ici du coup : https://www.matrix.org/docs/guides/client-server-api et il y a même un exemple complet ici que je vais regarder de près : https://jsfiddle.net/gh/get/jquery/1.8.3/matrix-org/matrix.org/tree/master/jekyll/_posts/howtos/jsfiddles/example_app

En JS, aux variables près :

```
var sendMessage = function(roomId, body) {
    var msgId = $.now();

    var url = "http://localhost:8008/_matrix/client/api/v1/rooms/$roomid/send/m.room.message?access_token=$token";
    url = url.replace("$token", accountInfo.access_token);
    url = url.replace("$roomid", encodeURIComponent(roomId));

    var data = {
        msgtype: "m.text",
        body: body
    };

    $.ajax({
        url: url,
        type: "POST",
        contentType: "application/json; charset=utf-8",
        data: JSON.stringify(data),
        dataType: "json",
        success: function(data) {
```

```
            $("#body").val("");
        },
        error: function(err) {
            alert(JSON.stringify($.parseJSON(err.responseText)));
        }
    });
};
```

Installation d'un serveur local Element ?

Tiens ici la doc d'install du serveur Matrix : https://github.com/matrix-org/synapse/blob/master/INSTALL.md#debianubuntu Reste à voir si çà passe sur un Pi. C'est du Python 3 à priori.

Qestion

Mais pourquoi installer (et administrer) un serveur en local ? Pourquoi ne pas taper directement le salon sensor56 par exemple ?

Oui, probablement qu'on peut aller taper directement le serveur sur lequel on est inscrit. Là c'est pour la démo. Mais en fait, faut essayer. L'important est que le princpe n'est pas plus compliqué que d'envoyer du HTML. Par contre, le serveur local vient peut être simplifier l'authentification et il sert de passerelle de toute façon vers le serveur où tu es inscrit car tous les serveurs Matrix se voient.

Ici explication : https://simplabs.com/blog/2020/11/02/bringing-matrix-to-elixir/

Lu sur le fil Te Domum :

"Sur le salon TeDomum, tu utilises "ton" serveur (en l'occurence matrix.org) pour intéragir avec le salon. Quand le serveur matrix.org reçoit un nouveau message de ta part sur ce salon, il enregistre le message dans sa copie du salon puis envoie le message à tous les serveurs qui sont dans ce salon. Chaque serveur reçoit la copie du message et l'enregistrent à leur tour. Puis les messages sont distribués aux clients qui attendent qu'on leur transmette des informations.

En gros: Toi -> matrix.org -> tous les serveurs participants -> les utilisateurs participants Comme j'ai dit, chaque serveur qui participe a la copie de l'historique (je simplifie) et de l'état du salon "

Previous Next

Test MQTT + Domoticz avec ESP 32

Installation du Pi

cf procédure usuelle

Note : il faut que le Pi soit sur un réseau connecté à internet au moins pour la mise en place.

Installation MQTT sur le Pi

Installation du brocker Mosquitto

Installation sur le Pi se fait en 3-4 lignes de commande : https://randomnerdtutorials.com/how-to-install-mosquitto-broker-on-raspberry-pi/

< Autre page très complète ici : https://projetsdiy.fr/mosquitto-broker-mqtt-raspberry-pi/

```
sudo apt-get update
sudo apt-get install  mosquitto
sudi apt-get install mosquitto-clients
```

Activation avec :

```
sudo systemctl enable mosquitto.service
sudo service mosquitto start
```

Une fois installé, on peut tester que tout ok en faisant :

```
sudo service mosquitto status
```

Au lancement suivant, il sera actif au démarrage.

Test du broker

https://randomnerdtutorials.com/testing-mosquitto-broker-and-client-on-raspbbery-pi/

On dispose de 2 commande pour s'abonner (subscribe) ou publier (publish). Logiquement, la commande "subscribe" va écouter et la commande "publish" est exécutée en attendant la suivante.

On va ici faire un test de bon fonctionnement du broker sur le poste lui-même qui ressemble à un test de serveur sur le poste lui-même.

Abonnement

On va commencer par lancer un abonnement de test que l'on va nommer :

```
mosquitto_sub -d -t testTopic
```

On a ce moment là une activité dans le terminal qui reste ouverte, en écoute, un peu comme un serveur qu'on lance.

Publication

On peut dès lors publier dans un second terminal en faisant :

```
mosquitto_pub -d -t testTopic -m "Hello world!"
```

On remarque du coup qu'à chaque envoi d'un "Hello World!", il est reçu dans le terminal précédent "subscribe".

Test avec ESP 32 ?

OK, on va essayer de faire çà dans l'interpréteur... à priori, à partir du moment où on a toujours notre MQTT qui écoute le "testTopic", on devrait pouvoir écrire dessus.

On commence par se connecter au réseau local (qui devra ici avoir accès à internet) :

```
>>> import network
>>> wifi=network.WLAN(network.STA_IF)
>>> wifi.active(True)
True
>>> wifi.connect("ssid","password")
>>> wifi.isconnected()
True
```

On a besoin de la librairie umqttsimple (à faire seulement la première fois)

```
>>> import upip as pip
>>> pip.install('micropython-umqtt.robust')
>>> pip.install('micropython-umqtt.simple')
```

Warning

Il y a 2 paquets à installer, robust dépendant de simple. Voir : https://boneskull.com/micropython-on-esp32-part-2/

Une fois fait, on peut utiliser mqtt sur l'ESP :

```
from umqtt.robust import MQTTClient
```

Ensuite, on crée le client (l'ESP est un client, et tout ce qui n'est pas le broker est un client).

```
>>> client = MQTTClient("123456789","192.168.0.52") # id,ipdubroker
>>> client.connect() # renvoie 0 ou erreur
0
>>> client.publish('testTopic', "72")
>>> client.publish('testTopic', "Hello  !")
```

Si tout est OK, vous devez voir dans la fenêtre de "subscribe" créée précédemment arriver les messages envoyés...

On peut aussi s'abonner à un "topic" et le topic ici est "testTopic".

```
>>> client.publish('testTopic', "72")
>>> client.publish('testTopic', "Hello  !")
>>> def reception(topic,msg):
    print(topic,":",msg)

>>> client.set_callback(reception)
>>> client.connect()
0
>>> client.publish('testTopic', "Hello  !")
>>> client.subscribe('testTopic')
>>> client.connect()
0
>>> client.publish('testTopic', "Hello  !")
```

Ensuite, on peut envoyer message sur "testTopic" depuis terminal du Pi.

Pour moi, je n'arrive pas voir le message qui censé être reçu par ESP envoyé depuis terminal "publish" du Pi. quelque chose ne doit pas être OK, mais c'est déjà très bien de pouvoir avoir la data en provenance de l'ESP.

En fait, il faut que le nom du topic soit au format bytes suivi d'un '/#' selon :

```
>>> topic='testTopic'.encode()+b"/#"
>>> topic
b'testTopic/#'
>>> client.subscribe(topic)
b'testTopic' : b'Hello world!'
b'testTopic' : b'Hello  !'
b'testTopic' : b'Hello world!'
>>> client.subscribe(topic)
>>> client.subscribe(topic)
b'testTopic' : b'Hello world!'
b'testTopic' : b'Hello world!'
b'testTopic' : b'Hello world!'
b'testTopic' : b'Hello world!'
```

Par contre, on a les messages à l'appel de client.subscribe() mais on n'a pas de retour de la fonction de callback, du moins dans l'interpréteur. A ce stade, je suis obligé d'appeler subscribe() pour lire le flux entrant. A voir si on peut faire mieux... Mais c'est déjà pas mal du tout.

https://www.programcreek.com/python/example/115319/umqtt.simple.MQTTClient

Installation de Domoticz et test

cf déjà fait... ?

```
curl -sSL install.domoticz.com | sudo bash
sudo apt-get upgrade
```

voir : https://projetsdiy.fr/installer-domoticz-raspbian-raspberry-pi3/

Ensuite, on peut se connecter sur ip.du.poste:8080 en http et 443 en https

On obtient assez facilement l'interface de domoticz.

Ensuite, si on veut utiliser avec MQTT : https://www.domoticz.com/wiki/MQTT

Il y a une procédure liée à l'interface à prendre en main :

- ajouter un hardware qui va être un Virtual MQTT
- ensuite créer via setup > hardware un Dummy (via liste déroulante)
- dans la liste qui apparaît, le nouvel item a un bouton créer nouveu device > créer un "température"
- ensuite créer un message de la forme : `mosquitto_pub -h localhost -m '{ "idx" : 1, "nvalue" : 0, "svalue" : "25.0" }' -t 'domoticz/in'` si depuis un terminal de test

ou bien si on envoie depuis l'interpréteur :

```
client.publish('domoticz/in', '{ "idx" : 1, "nvalue" : 0, "svalue" : "43.0" }')
```

C'est la forme minimale de message. Le "fléchage" de la donnée reçue se fait via l'index

Point important : il faut que l'idx soit le numéro du devive obtenu à partir de MQTT.

Noter qu'il est possible de configurer assez finement ce que l'on reçoit par MQTT ou ce que l'on ré-envoie. MQTT to Domiticz ou l'inverse.

Un truc important : domoticz reçoit bien mais ne "mémorise" que toutes les 5 minutes. Et le graphique ne se met pas à jour à l'arrivée d'une nouvelle data. Donc Domoticz, c'est pour du datalogging longue durée.

Ce qui est cohérent, c'est de faire un serveur du capteur lui-même qui permet le visuel live du capteur. Et d'utiliser MQTT vers Domoticz pour avoir la data longue-durée.

Des alternatives à Domoticz : https://alternativeto.net/software/domoticz/

notamment :

- openHAB
- Jeedom

Test avec Node JS

Node JS, notamment avec le JS Chart est une bonne alternative pour de la visualisation "multi-canal" dans une même interface.

D'autant que c'est assez léger au final lorsque çà tourne sur un Pi.

De plus, les messages MQTT peuvent être traités simultanément par MQTT et NodeJS, le broker faisant le lien.

Previous